Skriptum gemäß schulinternem Lehrplan

Grundkurs Informatik – Q1

Stand: 21. Juni 2016

Sprachvariante: Python

Inhaltsverzeichnis

[1] Die Idee für die Variante, von der OOM zum ER zu kommen, ist in (Löffler 2010) dargestellt.

Vorwort

Mit diesem Skriptum legen wir ein Dokument für unsere Schülerinnen und Schüler vor, das es ermöglichen soll, den Unterricht vor- und nachzubereiten.

Die verwendeten Arbeitsmaterialien wurden in den vergangenen Jahren von Informatikreferendarinnen und Informatikreferendaren, sowie den Absolventinnen und Absolventen der Fachseminare Informatik an den Zentren für schulpraktische Lehrerausbildung (ZfsL) Hamm, Arnsberg und Solingen entwickelt und im Unterricht erprobt.

Teile des Skriptums wurden so geändert, dass nunmehr die Unterstützung von Typen in Python realisiert ist.

! Elemente dieses Skriptums wurden dem Skriptum der Veranstaltung »Einführung in die Informatik – Teil 1« des Wintersemesters 2012/2013 von Prof. Dr. Till Tantau entnommen.

Die Materialien stehen unter einer freien Lizenz (🅭🅯🅭🅭 – Erläuterung siehe unten) und sind zum größten Teil über die Webseite `http://ddi.uni-wuppertal.de/material/` öffentlich zugänglich (vgl. (Pieper und Müller 2014)).

Vorbereitung

Was brauchen Sie?

In unserem Unterricht werden Sie viele schriftliche Notizen erstellen – Sie erhalten Informationsblätter und Arbeitsblätter. Für den Fließtext, den Sie schreiben werden, benötigen Sie regelmäßig Kugelschreiber oder Füller. Ihre Aufzeichnungen müssen Sie auf kariertem DIN-A4-Papier vornehmen, da häufig auch kleine Skizzen anzufertigen sind. Dazu benötigen Sie zwingend einen angespitzten Bleistift (HB) und zwei farbige Stifte (grün und rot). Damit Skizzen vernünftig aussehen, benötigen Sie ein Lineal und ein Geodreieck. Sammeln Sie Ihre Unterlagen in einem Hefter, den Sie im Unterricht immer dabei haben und den Sie jederzeit abgeben können. Wir sammeln Ihren Hefter gelegentlich ein und ziehen Ihre Aufzeichnungen zur Bewertung Ihrer sonstigen Mitarbeit heran.

! Versehen Sie jedes Arbeitsblatt, jedes Informationsblatt und jede Kompetenzüberprüfung, die wir Ihnen geben, unverzüglich mit Ihrem Namen und heften Sie diese Materialien in Ihren Hefter, der ebenfalls mit Ihrem Namen gekennzeichnet werden muss.

Gegebenenfalls erhalten Sie durch uns einen geschützten Zugang zu Informatiksystemen. Account und Passwort für solche Zugänge dürfen keinesfalls an andere (auch nicht im Kurs) weitergegeben werden.

	read	**w**rite	e**x**ecute
user	yes	yes	yes
group	no	no	no
other	no	no	no

Vorhaben 1

Verwaltung von Daten in linearen Datenstrukturen[1]

1.1 Welche Kompetenzen sollen Sie in diesem Vorhaben erwerben?

Die Schülerinnen und Schüler

- stellen lineare und nichtlineare Strukturen grafisch dar und erläutern ihren Aufbau (IF1, D)
- erläutern Operationen dynamischer (linearer oder nicht-linearer) Datenstrukturen (IF2, A),
- implementieren Operationen dynamischer (linearer oder nicht-linearer) Datenstrukturen (IF2 LK, I),
- modellieren Klassen mit ihren Attributen, Methoden und ihren Assoziationsbeziehungen unter Angabe von Multiplizitäten (M),
- ordnen Klassen, Attributen und Methoden ihre Sichtbarkeitsbereiche zu (M),
- stellen die Kommunikation zwischen Objekten grafisch dar (D),
- stellen Klassen und ihre Beziehungen in Diagrammen grafisch dar (D).

[1] Einige Elemente dieses Vorhabens wurden der Veranstaltung »Einführung in die Informatik – Teil 1« aus dem Wintersemester 2012/2013 von Prof. Dr. Till Tantau zum Thema »Arrays« entnommen (dort: Kapitel 13).

1.2 Lineare Datenstrukturen

1.2.1 Aufzählungen, Reihungen, Felder, Tabellen, Ketten

Verglichen mit einem einzelnen Zeichen (einem `character`) ist eine Zeichenkette (String) wesentlich aufregender. Mit einzelnen Zeichen kann man nicht sonderlich viel anfangen: Man kann sie ausgeben, sie einlesen und – wenn man möchte – auch vergleichen, dann hört es aber auch schon auf. Wie viel mehr ist mit Zeichenketten (Strings) möglich: Man kann **in ihnen** suchen, sie umdrehen, zerhacken, zusammenfügen, durcheinanderwirbeln, verschlüsseln, trimmen, vergleichen, einlesen, ausgeben, verrücken und noch vieles mehr. Zeichen werden eigentlich überhaupt erst interessant, wenn man sie zu ganzen Ketten zusammenstellt.

Wie steht es nun um Zahlen? Mit einer einzelnen Zahl kann man schon wesentlich mehr anstellen als mit einem einzelnen Zeichen. Bekanntermaßen kümmert sich ein ganzer Teilzweig der Mathematik, die Zahlentheorie, liebevoll um die vielfältigen Eigenschaften von einzelnen Zahlen. Wie viel aufregender müssen die Dinge dann erst werden, wenn wir Zahlen zu Ketten zusammenfügen?

Arrays[2] kennen Sie schon, auch wenn Ihnen das noch niemand verraten hat: listenförmige Zusammenstellungen gleichartiger Daten (Objekte), wie in Telefonbüchern, SMS-Listen, Fächeranordnungen in Ihrem Stundenplan und vieles andere mehr können als Arrays modelliert und implementiert werden.

Arrays kann man nicht nur aus Zahlenobjekten bilden. Allgemein kann man für Objekte *jeder* Klasse einen Array bilden. Dieser enthält dann ein erstes Element, ein zweites Element, ein drittes Element und so weiter bis zu einem letzten, n-ten Element, wobei n die *Länge* des Arrays bezeichnet (also 1 bis n).

Um ein Element in einem Array zu bezeichnen, benutzt man in der Informatik typischerweise die folgende Schreibweise (Notation): Man stellt den Index in eckige Klammern hinter den Bezeichner für das Array, also `x[2]`. Dies ist kein Tippfehler: Das *dritte* Element des Arrays erhält man über den Index 2, denn die Zählung beginnt bei 0. Es ist also `x[0]` das *erste* Element des Arrays und `x[n-1]` das *letzte* Element, wenn `x` die Länge `n` hat.

Auf den ersten Blick scheint es etwas merkwürdig zu sein, dass die Array-Indizierung in Python[3] bei 0 statt bei 1 beginnt. Dieser Eindruck täuscht: In Wirklichkeit ist dies

[2] In der deutschen Sprache wird die englische Bezeichnung Arrays (Plural – der Singular ist *Array*) mit: Aufzählungen, Reihungen, Felder, Tabellen, Ketten übersetzt. Als Geschlecht für den englischen Begriff wird mal das Maskulinum, mal das Neutrum verwendet, also: *der Array* oder *das Array*.

[3] Diese Indizierung gilt für die meisten Programmiersprachen – nicht nur für Python.

nicht nur eine kleine Merkwürdigkeit, sondern eine noch viel diabolischere Gemeinheit als der Umstand, dass in vielen Programmiersprachen[4] Zuweisungen durch ein einfaches Gleichheitszeichen geschrieben werden. Sollten Sie aus Versehen schreiben `if a=b`, obwohl Sie `if a==b` meinen, so wird der Übersetzer oder der Interpreter Ihnen dies mitteilen (*Syntaxfehler*) . Sollten Sie aus Versehen versuchen, mittels `x[10]` statt mit `x[9]` auf das zehnte Element eines Arrays zugreifen, so merken Sie das erst, wenn Ihr Programm schon längst läuft und schlimmstenfalls schon beim Kunden im Einsatz ist – also viel zu spät, um noch etwas zu ändern.

[4] Leider konnten die Varianten der Algol-Sprachfamilie nicht durchgesetzt werden, die für die Zuweisung die Konstruktion := vorgesehen haben. Noch schöner wären ← oder der umgekehrte Pfeil, wie von Konrad Zuse in der Programmiersprache Plankalkül vorgesehen.

Ideen zur Umsetzung

Die Probleme einer Bank.

Eine Bank möchte mit einem Programm die Kundennamen von Konten verwalten:

Kontonummer	Kundenname
1	
2	
3	
4	
...	...
500	

Wie sollte dies in Python abgebildet werden?

Erste mögliche Implementierung.

Aufgabe 1.1

```
kontoeigner1= "Zeisel"
kontoeigner2= "Charly"
kontoeigner3= "Wacker"
kontoeigner4= "Bär"
# ...
kontoeigner500= "Ackermann"
```

Geben Sie Gründe an, diese Implementierung *nicht* zu wählen. L

Zweite mögliche Implementierung.

```
kunden= [""] * 501

kunden[1]= "Zeisel"
kunden[2]= "Charly"
# ...
kunden[500]= "Ackermann"
```

Nun ist folgendes möglich:

```
1 gesucht= "Wacker"
2 for derKunde in kunden:
3   if gesucht == derKunde:
4     print (gesucht, "ist␣Kunde␣der␣Bank.")
```

Was sind Arrays?

- *Arrays* (Felder) entsprechen indizierten Variablen in der Mathematik (x_i).
- Sie halten eine *feste* Anzahl von Objekten, die alle aus *derselben Klasse* instanziiert sein müssen.
- Arrays liegen als »Block« irgendwo im Speicher, alle Werte hintereinander weg.

Beispiel
Die Bank würde einen Array `kunden` mit/aus `str`[5]-Werten benutzen. Man beachte: `kunden` ist *ein* Objekt, das gewissenmaßen eine »interne Struktur« hat.

Beispiel
Eine Moleküldatenbank könnte einen Array benutzen, in der jeder Eintrag ein Molekül darstellt/repräsentiert.

1.2.2 Array-Typen

Was sind Array-Typen?

- Hat man eine Klasse `BspKlasse` gegeben, so ist `list`[6] der Typ eines *Arrays von `BspKlasse`-Objekten*.
- Nun kann man ein Objekt dieses Typs festlegen:

  Beispiel: `kunden= [""] * 501` legt ein Array-Objekt mit 501 Einzelobjekten, die Zeichenketten enthalten können, fest. In diesem Beispiel werden alle Einträge mit "" vorbelegt.[7]
- Der Array-*Typ* legt die Größe des Arrays *nicht* fest. Ein Array-Objekt kann Arrays beliebiger Größe aufnehmen.
- Jeder *konkrete* Array hat aber eine *feste* Größe.

[5] Innerhalb von Python werden die eingebauten Standarddatentypen klein geschrieben, obwohl sie Klassen sind, also bei uns mit einem Großbuchstaben beginnen müssten.
[6] Bei `list` handelt es sich wieder um eine Klasse – nicht um ein Objekt.
[7] In Programmiersprachen mit expliziten Deklarationen werden normalerweise zu Beginn keine konkreten Werte in ein Array gelegt. Dies kann zu heimtückischen Fehlern führen, da dann nicht festliegt, welche Werte an einzelnen Positionen in dem Array stehen, man sie aber benutzen kann.

Analogie: Der Typ `String`[8] (Zeichenkette) legt auch die Länge der Zeichenkette nicht fest, aber jede konkrete Zeichenkette hat eine feste Länge.

Erzeugung von Arrays

Lebenszyklus eines Arrays

- Man kann einen Array auf zwei Arten *erzeugen*. Der erzeugte Array hat dann eine *feste, unveränderliche Größe*.
- Dann können Arrays *benutzt* werden.
- Werden sie nicht mehr gebraucht, werden sie *automatisch gelöscht*.

> **!** Will man die Größe eines Arrays *ändern*, so muss man einen neuen Array der gewünschten Größe erzeugen und dann die Elemente aus dem alten Array in den neuen Array kopieren.

Wie erzeugt man neue Arrays?

Erste Methode

Bei der Deklaration eines Array-Objekts darf man mittels einer speziellen Notation direkt einen Array angeben:

```
1 kunden= ["Zeisel", "Charly", "Wacker", "Bär"]
2 # Jetzt enthält das Objekt kunden
3 # einen Array der Länge 4
```

Zweite Methode

Man erzeugt einen leeren Array einer bestimmten Größe mittels einer anderen speziellen Notation:

```
1 kunden= [""] * 4
2 # Jetzt enthält das Objekt kunden
3 # einen Array der Länge 4
```

[8] Innerhalb von Python3 – in der Konsole – können Sie mittels `type( ... )` die Art eines Objekts herausfinden – für Zeichenketten liefert `type("Informatik")` als Ergebnis `<class 'str'>`.

Zugriff auf Arrays

Wie greift man auf Arrays zu?

- Hat man ein Objekt `reihung` des Typs `list`[9], so kann man mittels `reihung[5]` auf das sechste(!) Element zugreifen:
  Die Zählung fängt nämlich wie bei Zeichenketten (Strings) bei 0 an.
- Zugriff außerhalb der Größe des Arrays führt zum Absturz:

```
1  werte= [0] * 1000
2
3  # Beliebter Anfängerfehler:
4  index= 1
5  while index <= 1000:
6    werte[index]= 4
7    index= index + 1
8
9    # Tausendmal berührt,
10   # Tausendmal ist nichts passiert,
11   # Tausend und eine Nacht,
12   # Da hat es
13   # »IndexError: list assignment index out of range«
14   # gemacht ... (in Anlehnung an Lage 1991)
```

Schreibtischtest – Wertetabelle

Um herauszufinden, warum der Absturz erfolgt, empfiehlt sich die Erstellung einer Wertetabelle. Damit diese hier gut dargestellt werden kann, sehen wir davon ab, 1000 Einträge zu nehmen, sondern beschränken uns auf zehn `werte= [0] * 10`.

Der Schreibtischtest wird folgendermaßen durchgeführt: Sie legen einen Finger auf den Quelltext, führen aus, was dort steht und ändern ggf. einen Wert in der Tabelle. Sollte das Objekt auf der linken Seite einer Zuweisung bisher noch nicht in der linken Spalte der Tabelle stehen, so fügen Sie den Objektbezeichner dort an. Wird ein Wert geändert, so streicht man den alten Wert dünn durch (damit er erkennbar bleibt) und schreibt den neuen Wert dahinter. In dem folgenden Beispiel wird für das Array eine geänderte Darstellung gewählt, weil die Tabelle sonst sehr lang würde – die Einträge werden in der Tabelle unter den jeweiligen Index geschrieben.

Zwei Dinge fallen auf:

[9] Die Standarddatentypen in Python werden immer klein geschrieben, obwohl sie Klassen sind, also bei uns mit einem Großbuchstaben beginnen müssten. Im Anhnag C.5 – Python3 – eingebaute, vordefinierte Bezeichner – finden Sie eine Übersicht über **alle** Bezeichner, die in Python3 vorbelegt sind.

werte	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	
	0	~~0~~ 4	~~0~~ 4	~~0~~ 4	~~0~~ 4	~~0~~ 4	~~0~~ 4	~~0~~ 4	~~0~~ 4	~~0~~ 4	
index		~~1~~	~~2~~	~~3~~	~~4~~	~~5~~	~~6~~	~~7~~	~~8~~	~~9~~	*10*

Tabelle 1.1: Wertetabelle mit Ergebnis des Schreibtischtests

1. `werte[0]` wird nicht geändert
2. `index` erhält den Wert 10 und dann soll `werte[10]` geändert werden. Das Element `[10]` des Array gibt es aber nicht, so dass Python eine Fehlermeldung erzeugt (siehe Zeile 13 im Quellcode auf Seite 14).

Eine wichtige (interne) Methode.

- Für ein Array-Objekt `arrObj` liefert *`arrObj.__len__()`* die Größe des Arrays. Diese Methode liefert die Länge des Arrays, das ist die Anzahl der Objekte in diesem Array-Objekt.

Algorithmen auf Arrays

Umdrehen eines Arrays – Variante 1

```
1 vektor= [0] * 1000
2 # ... vektor wird gefüllt
3 index= 0
4 while index<vektor.__len__()//2: # // ganzzahlige Division
5   temp          = vektor[index]
6   vektor[index]= vektor[vektor.__len__()-index-1]
7   vektor[vektor.__len__()-index-1]= temp
8   index= index + 1
```

Umdrehen eines Arrays – Variante 2

```
1 vektor= [0] * 1000
2 # ... vektor wird gefüllt
3 # Drehe vektor um - es geht auch einfacher
4 index= 0
5 while index<vektor.__len__()//2: # // ganzzahlige Division
6   vektor[index], vektor[vektor.__len__()-index-1]=
       vektor[vektor.__len__()-index-1], vektor[index]
7   index= index + 1
```

Aufgabe 1.2

Geben Sie ein Programm mit einer Zählschleife an, das zwei Arrays verkettet.

Es sollen in `z` zuerst die Werte aus `a1` kommen, gefolgt von den Werten aus `a2`.

```
# Zwei Array-Objekte
a1= ["h", "a", "l"] # oder etwas anders
a2= ["l", "o"]      # oder etwas anders

# Der Array, in den die Verkettung hinein soll:
z= ["␣"] * (a1.__len__() + a2.__len__())
```

Schreiben Sie bitte Ihre Lösung in das folgende Textfeld:

L

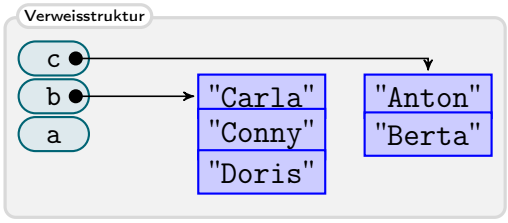
1.2.3 Speicherung von Arrays

Verweistypen

Wohin mit dem Array?

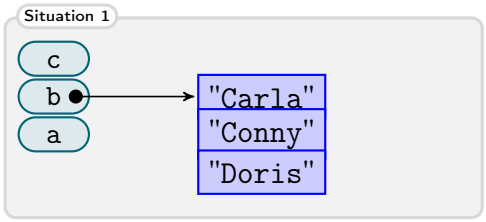
- Bei einem Array kann der Übersetzer (oder der Interpreter) offenbar nicht den Speicherbedarf vorher bestimmen.
- Deshalb reserviert der Übersetzer (oder der Interpreter) lediglich den Platz für einen *Verweis*.

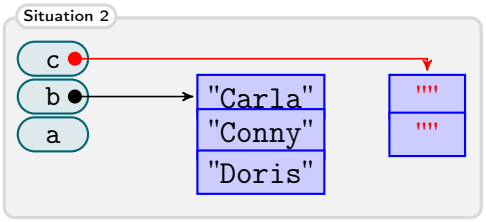
```
a= str()
b= ["Carla","Conny","Doris"]
c= ["Anton","Berta"]
```



Erzeugung eines neuen Arrays mittels `=[""]*2`.

```
1 a= str()
2 b= ["Carla","Conny","Doris"]
3 c= []                 # Situation 1
4 c= [""] * 2           # Situation 2
5 print(a,b,c)
```

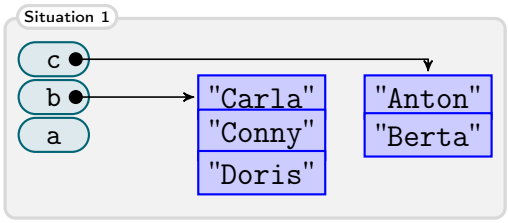


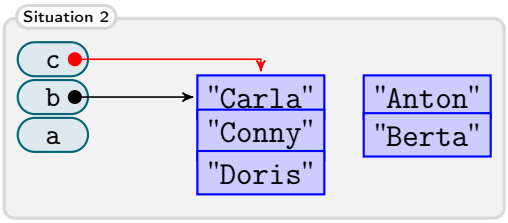


Zuweisung und Vergleich von Verweistypen

Zuweisung von Arrays.

```
1 a= str()
2 b= ["Carla","Conny","Doris"]
3 c= ["Anton","Berta"]          # Situation 1
4 c= b                          # Situation 2
5 print(a,b,c)
```





Es gibt unerwartete Effekte bei Zuweisungen von Arrays.

- Wenn man mittels `b= c` einem Array-Objekt `b` ein anderes Array-Objekt `c` zuweist, so *verweisen `b` und `c` auf dasselbe Array-Objekt*.
- Ändert man dann `b`, so ändert man auch gleichzeitig `c`,
  was man meistens nicht will.
- Vergleicht man zwei Array-Objekte mittels ==, so wird lediglich überprüft, ob die Array-Objekte auf dasselbe Array-Objekt verweisen und *nicht, ob die Arrays die selben Element haben*;
  und auch dies will man meistens nicht.

Moral

1. Zuweisung von Arrays sind mit Vorsicht zu genießen.
2. Vergleiche von Arrays sind mit Vorsicht zu genießen.

Zusammenfassung – Lineare Datenstrukturen

Lineare Datenstrukturen – Zusammenfassung

1. *Arrays* fassen Tabellen von Werten zu einem Objekt zusammen.
2. Die Zählung beginnt bei *0*.
3. Arrays haben vielfältige *Anwendungen*.
4. Man sollte Arrays nicht mittels = *zuweisen* und nicht mittels == *vergleichen*.

Aufgabe 1.3

1. Arrays statt Strings

   Schreiben Sie ein Programm, das die komplementäre Sequenz zu einer Basenfolge berechnet.

   Das jeweilige Komplement ergibt sich nach den folgenden Ersetzungen:

   - `"a"` → `"t"`
   - `"t"` → `"s"`
   - `"c"` → `"g"`
   - `"g"` → `"c"`

   Die Sequenzen sind als Array aus einzelen Zeichenobjekten gespeichert:

```
sequenz    = ["t","c","c","t","a","t"]
komplement= [""] * sequenz.__len__()
```

Um ein Array auszugeben, können Sie mit `print(...)` arbeiten.

2. Arrays verarbeiten

   Schreiben Sie ein Programm, das zu einem Array von Zahlen das Minimum, das Maximum und den Mittelwert ermittelt. Testen Sie Ihr Programm mit folgendem Array:

```
numbers= [3, 0, 610, 4181, 1, 89, 377, 13, 34, 2584, 1,
   1597, 144, 233, 21, 55, 987, 5, 8, 17711, 6765, 28657,
   2, 10946]
```

3. Syntaxfehler finden

   Betrachten Sie folgenden Python-Code:

```
a= [1,2,3]
b= {0} * 2*a.__len__()
for i in range(0, a.__len__()):
  b[2*i]:= a[i]
  b[2*i+1]= a[i]
```

   a) In zwei Zeilen des Codes befinden sich Fehler. Wie lauten diese Zeilen richtig?

   b) Wie lautet der Inhalt des Arrays `b` nach Ausführung des von Ihnen **berichtigten** Codes?

1.3 Nichtlineare Datenstrukturen – Listen: Konzepte und Erstellung[10]

Listen sind eine so genannte *fortgeschrittene Datenstruktur*. »Fortgeschritten« heißen sie, weil sie im Gegensatz zu Arrays eine, wie Sie noch sehen werden, recht komplexe innere Struktur aufweisen, mit Verwaltungsklassen, Zellklassen und wilden Verzeigerungen im Speicher. Der Name »Liste« ist eigentlich eher schlecht gewählt (aber, wie so vieles historisch gewachsenes, nicht mehr zu ändern). Unter einer Liste stellt man sich landläufig eine zeilenweise untereinandergeschriebene Aufstellung

[10] Teile dieses Abschnitts wurden der Veranstaltung »Einführung in die Informatik – Teil 1« aus dem Wintersemester 2012/2013 von Prof. Dr. Till Tantau zu den Themen »Listen – Konzepte und Erstellung« und »Listen – Iteration und Modifikation« entnommen (dort: Kapitel 25 und 26) und an einigen Stellen geändert.

von Punkten vor. Jedoch meint man in der Informatik mit »Liste« ein eher chaotische »Verkettung« der Listenpunkte, wo bei jedem Punkt am Ende steht, wo der nächste Punkt zu finden ist.

Einen normalen Array kann man sich ganz gut als eine sehr lange Straße vorstellen, an deren Rand nummerierte Häuser stehen. In jedem Haus »wohnt« ein Array-Element, in Haus 0 beispielsweise ein »A«, in Haus 1 ein »C« und in Haus 2 noch ein »C«. Natürlich fängt die Nummerierung von Häusern in Wirklichkeit bei 1 an, aber die Informatik ist eben nicht die Wirklichkeit. Was passiert nun, wenn man nach dem, sagen wir, zweiundvierzigsten Haus ein weiteres einfügen will? Dies ist bei Arrays nicht möglich. Vielmehr muss man sich eine neue Straße suchen mit einem Haus mehr und alle Elemente bis zum zweiundvierzigsten Haus in das Haus mit derselben Nummer in der neuen Straße umziehen lassen, dann alle Elemente in Häusern aus der alten Straße mit höheren Nummern in die Häuser mit der eins höheren Nummer in der neuen Straße. Ein reichlich aufwendiger Vorgang, bei dem Kolonnen von Umzugswagen benötigt werden.

Eine Liste im Informatiksinne kann man sich eher als Zeltplatz vorstellen. Überall stehen wild verstreut Zelte herum, in denen Elemente hausen. Die genaue Position eines Zeltes auf dem Platz ist völlig unerheblich. Wenn man aber eine Reihenfolge auf den Zelten braucht, so speichert man bei jedem Zelt neben dem Element auch *den Ort, wo sich das nächste Zelt in der Reihenfolge befindet*. Diese Information könnte man zum Beispiel außen auf das Zelt malen. Will man nun die Zelte gemäß diese Reihenfolge besuchen, so geht man zum ersten Zelt, dessen Ort am Eingang des Zeltplatzes auf einer besonderen Tafel steht. Von dort aus besucht man das Zelt, dessen Ort auf dem ersten Zelt steht. Von dort aus das Zelt, dessen Ort auf dem zweiten Zelt steht; und so weiter. Auf dem letzten Zelt steht dann »Ende«. Kommt ein neuer Zelter, so kann er einfach irgendwo sein Zelt aufschlagen. Auf sein Zelt schreibt man dann den Ort des alten ersten Zeltes (der ja auf der Tafel am Eingang steht) und schreibt dafür auf die Tafel Eingang den Ort des neuen Zeltes. Ähnlich einfach kann man auch Zelte in der Mitte oder am Ende einer Liste einfügen und Zelte können auch recht leicht den Zeltplatz verlassen.

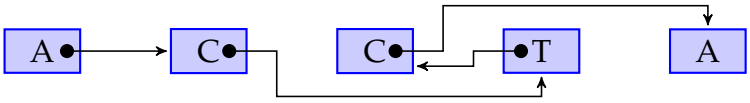
Die Organisation von Daten als Zeltplatz ist sehr vorteilhaft, wenn ständig Leute (Elemente) kommen und gehen. Jedoch dauert bei dieser Organisation das Finden von Zelt Nummer 123 in der Reihenfolge recht lange, man muss 123 Mal kreuz und quer über den Zeltplatz laufen. Bei einer Straße ist es hingegen sehr leicht und schnell möglich, Haus 123 zu finden.

Moral von der Geschicht: Ob man sein Daten als Zeltplatz (=Liste) oder als Straße (=Array) organisiert, hängt hauptsächlich davon ab, wie oft »Daten kommen und gehen«.

1.3.1 Idee hinter Listen

Die Datenstruktur der Liste

- Die *Liste* ist eine Datenstruktur, die nach dem Vorbild eines kafkaesquen Amtes aufgebaut ist.
- Jeder Sachbearbeiter weiß über irgendwas Bescheid, für alles andere wird man zum nächsten Sachbearbeiter geschickt.
- Der nächste Sachbearbeiter sitzt in der Regel nicht nebenan, sondern irgendwo anders.



Vor- und Nachteile von Listen gegenüber Arrays und Strings.

Vorteile

Folgende Operationen gehen *sehr leicht* und *sehr schnell*:

+ Einfügen neuer Elemente.
+ Verketten von Listen zu neuen Listen.
+ Löschen vorhandener Elemente.
+ Ausschneiden von Teilen aus einer Liste.

Nachteile

- Um das i-te Element zu finden, muss man »i Sachbearbeiter nacheinander aufsuchen«.
- Deshalb ist binäre Suche *nicht möglich* und man muss immer *lineare Suche* durchführen.

1.3.2 Die Datenstruktur »einfach verkettete Liste«

Idee

Listen bestehen aus Zellen.

- Eine Liste besteht aus vielen *Zell-Objekten* (»Informatikzellen«, nicht biologische Zellen).
- Eine Zelle speichert
  1. Einen Wert, wie zum Beispiel eine Base.
  2. Einen Verweis auf die nächste Zelle in der Liste.
- In der letzten Zelle ist der Verweis auf die nächste Zelle `None`.
- Das *Listen-Objekt* speichert lediglich einen Verweis auf die erste Zelle.

Umsetzung in Python-Code

Die Sequenz- und die Zell-Klasse

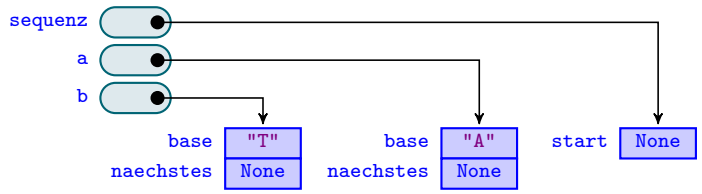
```
1 class Zelle:
2   # Die Basen ("A", "C", "G", "T"): base
3   # Nächstes Listenelement: naechstes
4   def __init__(self):
5     self.base= ""
6     self.naechstes= None
7 class DNASequenz:
8   def __init__(self):
9     self.start= None    # Objekt der Klasse Zelle
```

Erzeugung einer Liste

Schritt 1

Der folgende Code erzeugt erst ein Listenobjekt und zwei Zellobjekte. Die Zellobjekte haben schon Daten, sind aber noch nicht verkettet.

```
1 if __name__ == "__main__":
2   sequenz= DNASequenz()
3   a= Zelle()
4   b= Zelle()
5   a.base= "A"
6   b.base= "T"
```

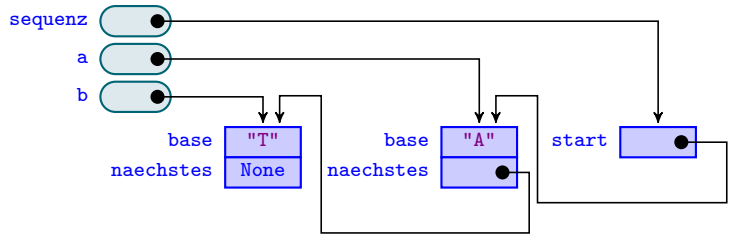


Erzeugung einer Liste

Schritt 2

Nun werden die Objekte verkettet.

```
1  # ...
2  sequenz.start= a
3  a.naechstes  = b
```



Zur Übung

Aufgaben 1.4

a) Geben Sie den Code der Klassen `Schuelerliste` und `Schuelerzelle` zur Verwaltung einer Liste von Schülern an.
b) Geben Sie weiter den Code zur Erzeugung einer Schülerliste mit drei Schülern an.

1.3.3 Operationen auf Listen

Einfügen am Anfang

Einfügen eines neuen Elements am Anfang

Problemstellung

Am Anfang der DNA-Sequenz soll eine neue Base angefügt werden.

Algorithmus

1. Erzeuge eine *neue Zelle* für die neue Base.
2. Der *Nachfolger* dieser neuen Zelle ist der *Start der Liste*.
3. Setze den *Start der Liste* auf die *neue Zelle*.

Beachte:

- Dieser Algorithmus ist eine *Fähigkeit der Listen-Klasse*: Man kann ein Objekt der Listen-Klasse durch eine Nachricht »bitten«, ein Element hinzuzufügen.
- Deshalb fügen wir diese Fähigkeit als Methode `fuegeBaseVornEin` der Klasse `DNASequenz` hinzu.

Code der Methode.

```
1  class DNASequenz:
2    def __init__(self):
3      self.start= None    # Objekt der Klasse Zelle
4
5
6    def fuegeBaseVornEin(self, b: 'str'):
7      neueZelleVorn= Zelle()
8
9      neueZelleVorn.base= b
10     neueZelleVorn.naechstes= self.start
11
12     self.start= neueZelleVorn
```

Zur Übung

Aufgabe 1.5

Visualisieren Sie wie zuvor graphisch alle Objekte und Attribute, die folgender Code erzeugt:

```
1  sequenz= DNASequenz()
2
3  sequenz.fuegeBaseVornEin("A")
4  sequenz.fuegeBaseVornEin("C")
5  sequenz.fuegeBaseVornEin("T")
```

1.3.4 Löschen am Anfang

Löschen des Elements am Anfang

Problemstellung

Das Element am Anfang der Liste soll gelöscht werden.

Algorithmus

Ersetze `start` durch den Nachfolger von `start`.

```
1 class DNASequenz:
2   # ...
3   def loescheErstes(self):
4     self.start= self.start.naechstes
```

Zur Übung

Aufgabe 1.6

Visualisieren Sie wie zuvor graphisch alle Objekte und Variablen, die folgender Code erzeugt:

```
1   sequenz= DNASequenz()
2
3   sequenz.fuegeBaseVornEin("A")
4   sequenz.fuegeBaseVornEin("C")
5   sequenz.fuegeBaseVornEin("T")
6   sequenz.loescheErstes()
7   sequenz.loescheErstes()
```

Zusammenfassung zu Listen – Erstellung und Verzeigerung

Listen – Erstellung und Verzeigerung – Zusammenfassung

1. Listen sind eine *Datenstruktur*, ähnlich dem Array.
2. Man kann in Listen Elemente *einfügen* und *löschen*.
3. Dies geht schnell, da *lediglich Verweise lokal verändert* werden.

1.3.5 Listen – Iteration und Modifikation. Von verbogenen Zeigern

Einführung – »Besuch« der Elemente, Ändere die Liste

So. Nun ist die Liste da. Und was jetzt?

Im letzten Abschnitt ging es darum, Listen aufzubauen – ein speichertechnisch chaotischer, aber doch erfolgreicher Vorgang. Um nun etwas mit der Liste anzufangen, muss man auch »die Liste entlanglaufen« können. Beispielsweise könnte man dann bei jedem besuchten Listenelement (= Zelt) den Inhalt ausgeben (= den Insassen des Zeltes zu einer Unterschrift bewegen auf einer Unterschriftenliste beispielsweise gegen den kommerziellen Walfang, Softwarepatente, globale Erwärmung oder die Revision der Abschaffung der Abschaffung lokalitätsspezifischer Rauchverbote). Dieses Entlanglaufen ist recht einfach zu implementieren.

Etwas vertrackter wird die Sache, wenn man irgendwo inmitten der Liste Elemente einfügen oder löschen möchte. Letztendlich muss man nur die »richtigen Zeiger umsetzen«, jedoch liefert dies etwas mystischen Code wie

```
1 loesche_naechste_zelle.naechstes=
    loesche_naechste_zelle.naechstes.naechstes
```

Alles klar? Am Ende dieses Abschnitts hoffentlich schon.

1.3.6 Iteration

Idee

Wie besucht man alle Elemente einer Liste?

Problemstellung

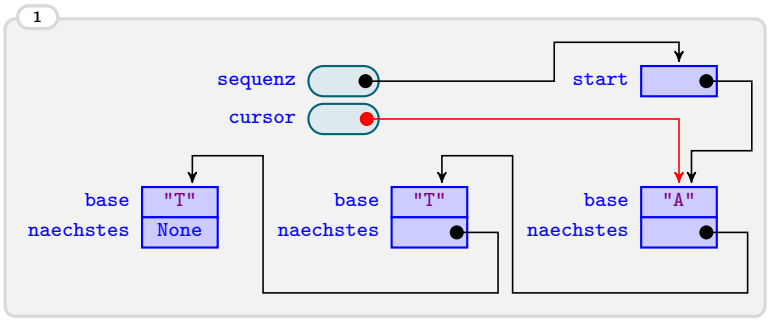
Wir wollen alle Elemente (= Zellen) einer Liste »besuchen« und dort »irgend was tun«.

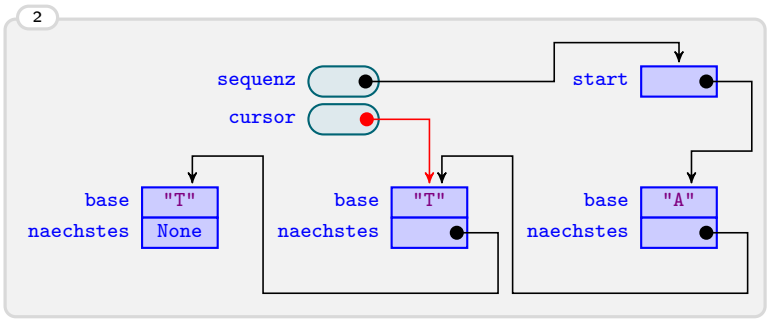
Iterativer Algorithmus

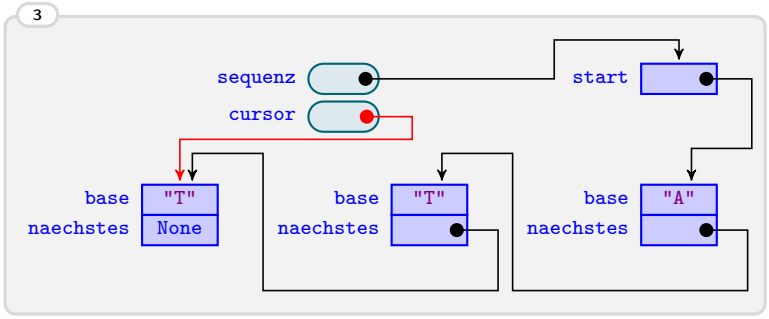
1. Setze `cursor` auf die Startzelle.
2. Tue das Gewünschte für diese Startzelle.
3. Solange `cursor` einen Nachfolger hat tue:
   a) Setze `cursor` auf den Nachfolger von `cursor`.
   b) Tue das Gewünschte für die aktuelle Stelle.

Wer zeigt wann wohin?

```
1  cursor= sequenz.start            # Startsituation
2  tue_was(cursor)
3  while cursor.naechstes != None:
4    cursor= cursor.naechstes       # Vorwärts!
5    tue_was(cursor)
```







Eine effizientere Version des Codes.

Der Quellcode (siehe Seite 27) hat zwei Nachteile:

1. Der Aufruf von `tue_was` steht zweimal im Code. Dies ist unschön, wenn man stattdessen etwas Komplexeres machen möchte.

2. Der Code funktioniert nicht, wenn die Liste leer ist, also sofort `cursor == None` gilt.

Diese Probleme lassen sich wie folgt umgehen:

```
cursor= sequenz.start
while cursor != None:
  tue_was(cursor)
  cursor= cursor.naechstes
```

Eine rekursive Variante (für Profis).

Man kann das Problem auch kurz und elegant rekursiv lösen:

Rekursiver Algorithmus

Wenn es überhaupt Zellen gibt:

1. Tue was für die erste Zelle.
2. Wende den Algorithmus auf den Rest an.

```
def rekursiverAlgorithmus(self, c: 'Zelle'):
  if c != None:
    tue_etwas(c)
    rekursiverAlgorithmus(c.naechstes)

# Aufruf:
rekursiverAlgorithmus(sequenz.start)
```

Anwendung: Längenbestimmung

Problemstellung: Die Länge einer Liste bestimmen.

- Die Länge einer Liste »sieht man ihr nicht an«.
- Man *muss* alle Elemente einmal besuchen und dabei einen Zähler für jedes besuchte Element hochzählen.
- Die Aktion »tue was« ist hier gerade das Hochzählen dieses Zählers.

```
# Algorithmus zum Zählen der Elemente einer Liste
zaehler= 0
cursor= sequenz.start
while cursor != None:
  zaehler= zaehler + 1
  cursor= cursor.naechstes
```

Die Längenmethode.

- Die Längenberechnung sollte durch eine Methode der Listenklasse implementiert werden.
- Dann kann man ein Listenobjekt erstellen und es später mittels einer Nachricht fragen, was seine Länge ist.

```
class DNASequenz:
  # ...
  def length(self):
    zaehler= 0
    cursor= self.start
    while cursor != None:
      zaehler= zaehler + 1
      cursor= cursor.naechstes
    return zaehler
```

Zur Übung

Aufgabe 1.7

Geben Sie den Code einer Methode `zaehle_As` an, die die Anzahl an »A«'s in der Liste zurückgibt. L

Anwendung: Ausgabe aller Elemente

Problemstellung: Ausgabe aller Elemente.

- Um alle Elemente einer Liste auszugeben, muss man sie einfach alle besuchen.
- Die Aktion »tue was« ist dann gerade die Ausgabe.

```
class DNASequenz:
  # ...
  def printDNA(self):
    cursor= self.start
    while cursor != None:
      print(cursor.base, end="")  # Python3 - ohne Zeilenumbruch
      cursor= cursor.naechstes
    print()
```

Zur Übung

Aufgabe 1.8

Geben Sie den Code einer Methode an, die die Basen in der Liste als einen String zurückgibt. Die Idee ist, jede Base nacheinander an das Ende eines Strings anzuhängen:

```
return_me= return_me + cursor.base
```

1.3.7 Anwendung: Map

Problemstellung: Verändern aller Elemente.

- Wir wollen nun alle Elemente einer Liste verändern, beispielsweise durch ihre Komplemente ersetzen.
- Die Aktion »tue was« ist dann gerade diese Modifikation.
- Ein solches Verändern aller Element wird in der funktionalen Programmierung *Map* genannt.

```
class DNASequenz:
  # ...
  def komplement(self):
    cursor= self.start
    while cursor != None:
      if    cursor.base == "A": cursor.base = "T"
      elif cursor.base == "T": cursor.base = "A"
      elif cursor.base == "C": cursor.base = "G"
      elif cursor.base == "G": cursor.base = "C"
      cursor= cursor.naechstes
```

Anwendung: Suche

Problemstellung: Suche nach einem Element.

- Wir wollen nun ein Element finden mit einer bestimmten Eigenschaft; beispielsweise das erste »A«.
- Die Aktion »tue was« ist dann der Test, ob der `cursor` eine Zelle mit der gesuchten Eigenschaft erreicht hat.

```
class DNASequenz:
  # ...
  def suche_erstes_a(self):
    cursor= self.start
    while cursor != None:
      if cursor.base == "A":
        return cursor          # Gefunden! Danke und tschüss.
      cursor= cursor.naechstes
    return None                # Nicht gefunden. Grrr.
```

1.3.8 Modifikation

Einfügen von Elementen

Wie fügt man ein Element in die Mitte einer Liste ein?

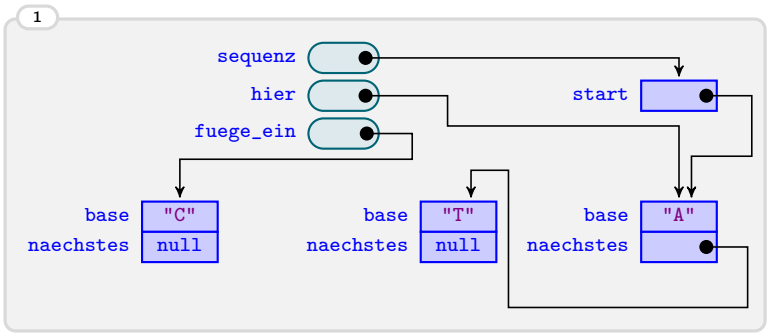
- Wir wollen ein neues Element nicht am Anfang einer Liste, sondern irgendwo zwischendrin einfügen.
- Dazu muss man *eigentlich nur lokal die Verkettung ändern*.
- Genauer braucht man zunächst einen Verweis auf ein Element `a`, *nach dem man* das neue Element `b` einfügen möchte.
- Dann ändert man zwei Verweise:
  - Der Nachfolger von `a` ist nun `b`.
  - Der Nachfolger von `b` ist nun der alte Nachfolger von `a`.

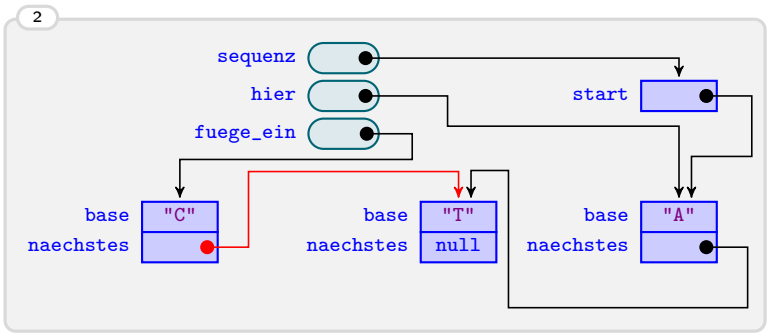
Der Code einer Einfügemethode.

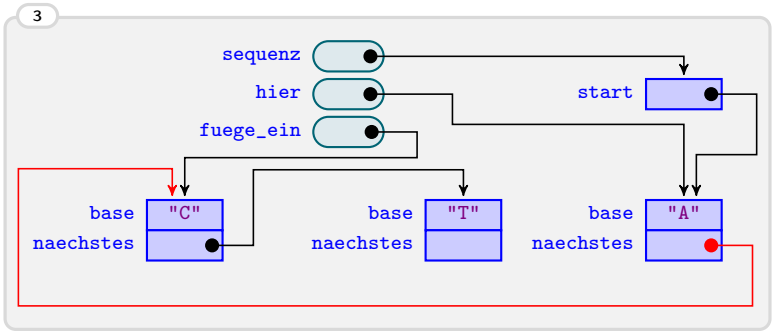
```
class DNASequenz:
  # ...
  def fuege_ein_nach(self, hier: 'Zelle', b: 'str'):
    fuege_ein           = Zelle()
    fuege_ein.base      = b
    fuege_ein.naechstes= hier.naechstes
    hier.naechstes      = fuege_ein
```

Wer zeigt wann wohin?

```
1                                                  # 1. Ausgangssituation
2  fuege_ein.naechstes= hier.naechstes             # 2. Erste Veränderung
3  hier.naechstes     = fuege_ein                  # 3. Fertig
```







Löschen von Elementen

Wie löscht man ein Element aus der Mitte einer Liste?

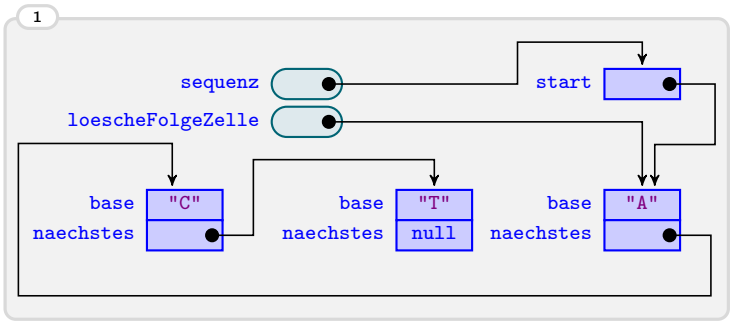
- Wir wollen ein Element irgendwo in der Mitte einer Liste löschen.

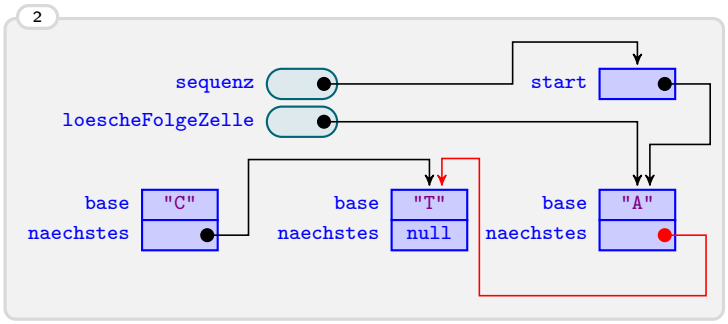
- Dazu muss lediglich den *Nachfolger* des Vorgängers ändern.

```
class DNASequenz:
  # ...
  def loesche_nach(self, loescheFolgeZelle: 'Zelle'):
    # Löscht die Zelle, die auf loescheFolgeZelle folgt:
    loescheFolgeZelle.naechstes=
       loescheFolgeZelle.naechstes.naechstes
```

Wer zeigt wann wohin?

```
# 1. Vorher
loescheFolgeZelle.naechstes=
   loescheFolgeZelle.naechstes.naechstes
# 2. Nachher
```





Verketten von Listen

Wie verkettet man zwei Listen?

- Wir wollen aus zwei Listen eine Liste machen.

- Dazu muss der *Nachfolger* des letzten Elements der einen Liste der Anfang der zweiten Liste werden.
- Dazu muss man allerdings erst »das Ende finden« – dies macht man mit einer Iteration.

Zur Übung

Aufgabe 1.9

Geben Sie den Code einer Methode zur Verkettung zweier Listen an: H

```
1 class DNASequenz:
2   # ...
3   def konkateniere_mit(self, mir):
```

Tragen Sie Ihren Python-Quellcode in das folgende Textfeld ein:

L

Zusammenfassung – Arbeiten mit Listen: Besuchen, Einfügen und Löschen

Listen – Besuchen Einfügen und Löschen – Zusammenfassung

1. Eine *Iteration* besucht alle Elemente einer Liste einmal.
2. Man kann in in Listen Elemente an beliebigen Stellen *einfügen* und *löschen*.
3. Dies geht schnell, da *lediglich Verweise lokal verändert* werden.

Vorhaben 2

Lineare Datenstrukturen – Suchen und Sortieren[1]

2.1 Welche Kompetenzen sollen Sie in diesem Vorhaben erwerben?

Die Schülerinnen und Schüler

- analysieren und erläutern Algorithmen und Programme (IF2, A)
- modifizieren Algorithmen und Programme (IF2, I)
- stellen iterative und rekursive Algorithmen umgangssprachlich und grafisch dar (IF2, D)
- entwickeln iterative und rekursive Algorithmen unter Nutzung der Strategien »Modularisierung« und »Teilen und Herrschen« (IF2, M)
- implementieren iterative und rekursive Algorithmen auch unter Verwendung von dynamischen Datenstrukturen (IF2, I)
- testen Programme systematisch anhand von Beispielen (IF2, I)
- erläutern Operationen dynamischer (linearer oder nicht-linearer) Datenstrukturen (IF2, A)
- implementieren und erläutern iterative und rekursive Such- und Sortierverfahren (IF2, I)
- beurteilen die Effizienz von Algorithmen unter Berücksichtigung des Speicherbedarfs und der Zahl der Operationen (IF2, A)
- nutzen die Syntax und Semantik einer Programmiersprache bei der Implementierung und zur Analyse von Programmen (IF3, I)
- beurteilen die syntaktische Korrektheit und die Funktionalität von Programmen (IF3, A)

[1] Teile dieses Abschnitts wurden der Veranstaltung »Einführung in die Informatik – Teil 1« aus dem Wintersemester 2012/2013 von Prof. Dr. Till Tantau zu den Themen **Suchalgorithmen und Sortieralgorithmen** entnommen (dort: Kapitel 16 und 17) und an einigen Stellen geändert.

- interpretieren Fehlermeldungen und korrigieren den Quellcode (IF3, I)
- wenden eine didaktisch orientierte Entwicklungsumgebung zur Demonstration, zum Entwurf, zur Implementierung und zum Test von Informatiksystemen an (IF4, I)

2.2 Varianten des Suchproblems

Es gibt viele Varianten des Suchproblems.

In der folgenden Darstellung wird häufig von *Wert* gesprochen. Exakt formuliert, müsste dort immer *Wert des Objekts* geschrieben und gesagt werden.

Suchen eines Wertes

Eingaben:

- Ein Array von Werten und
- *ein Wert*, der gesucht wird.

Ausgabe:

- *Eine* Position, an der der Wert im Array vorkommt.

Suchen aller Werte

Eingaben:

- Ein Array von Werten und
- *ein Wert*, der gesucht wird.

Ausgabe:

- *Alle* Positionen, an der der Wert im Array vorkommt.

Suchen eines Wertes mit einer Eigenschaft

Eingaben:

- Ein Array von Werten und
- *eine Eigenschaft*, die Werte haben können oder auch nicht.

Ausgabe:

- *Eine* Position eines Wertes, der die Eigenschaft hat.

Suchen aller Werte mit einer Eigenschaft
Eingaben:

- Ein Array von Werten und
- *eine Eigenschaft*, die Werte haben können oder auch nicht.

Ausgabe:

- *Alle* Positionen von Werten, die die Eigenschaft haben.

2.2.1 Lineare Suche

Idee

Die lineare Suche ist das einfachste Suchverfahren.

- Bei der *linearen Suche* werden einfach alle Werte (Objekte) des Arrays überprüft.
- Eine oder alle Positionen, an denen Werte mit der gewünschten Eigenschaft sind, werden zurückgegeben.

Implementation

Beispiel einer linearen Suche.

Finden einer Telefonnummer, die auf 6 endet.

```
1  telefonNummern= ["7974311",
2                   "2147856",
3                   "2161555",
4                   "5553466"]
5
6  # Suche linear nach einer Telefonnummer, die auf 6 endet.
7  nummer= ""
8  index= 0
9  while index < telefonNummern.__len__():
10   if telefonNummern[index][-1] == "6":
11     nummer= telefonNummern[index]
12   index= index + 1
13
14 # nummer ist jetzt "5553466"
```

Beispiel einer linearen Suche.

Finden der Position einer Telefonnummer in einem Array.

```
telefonNummern= ["7974311",
                 "2147856",
                 "2161555",
                 "5553466"]
# Suche linear nach "2161555":

position_wert= -1 # noch nicht gefunden
index= 0
while index < telefonNummern.__len__():
  if telefonNummern[index] == "2161555":
    position_wert= index
  index= index + 1

# position_wert ist jetzt 2.
```

Eine allgemeine lineare Suche.

```
class SuchAlgorithmen:

  def lineareSuche(self, zeichenketten: 'list', wert: 'str'):
    # Findet erstes Vorkommen von wert in zeichenketten.
    # Kommt es nicht vor, wird -1 zurückgegeben

    index= 0
    while index < zeichenketten.__len__():
      if zeichenketten[index] == wert:
        return index
      index= index + 1
    return -1
```

Laufzeit

Aufgabe 2.1

Wie viele Vergleiche führt die Methode `lineareSuche` bei einem Array der Länge n

1. mindestens,
2. höchstens und
3. im Durchschnitt aus?

2.2.2 Binäre Suche

Idee

Die Grundidee der binären Suche.

Beobachtung

- Suchen wir einen Namen im Telefonbuch, so suchen wir diesen natürlich nicht linear.
- Vielmehr fangen wir grob in der Mitte an und gehen dann sprungweise nach vorne oder nach hinten.

Binäre Suche

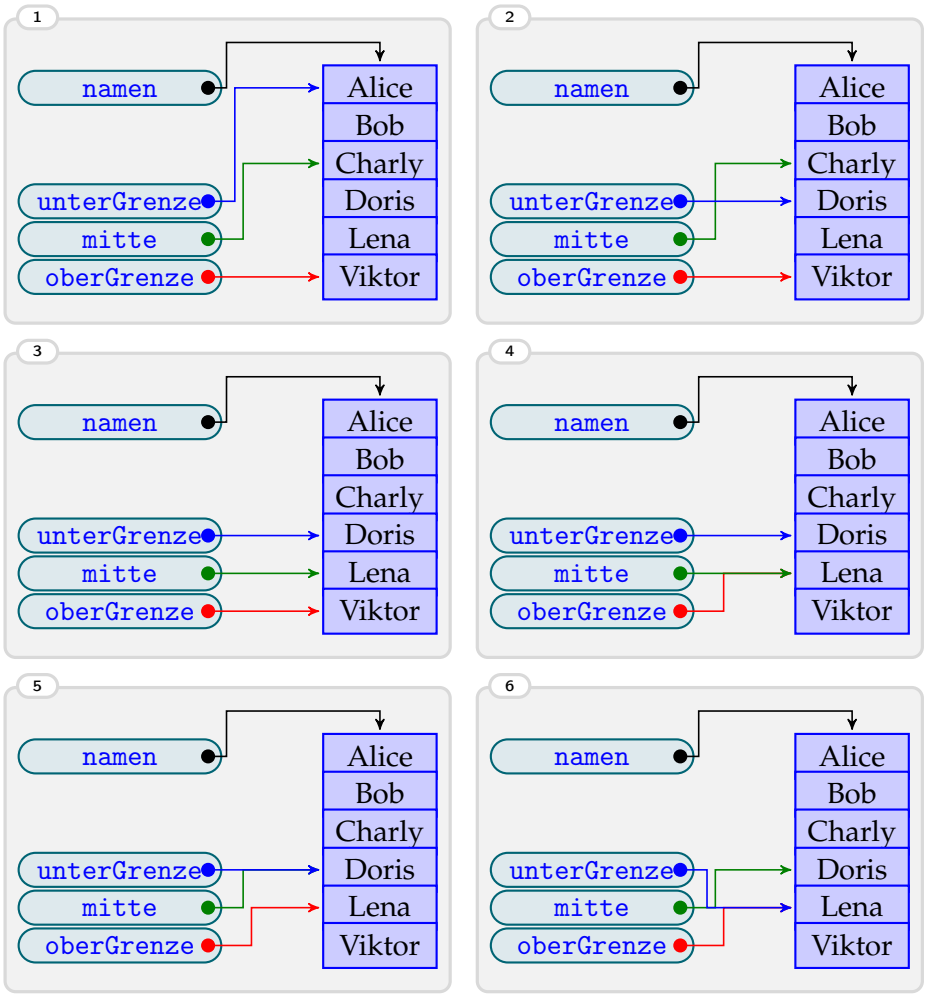
- Binäre Suche arbeitet auf *sortierten* Arrays.
- Man *halbiert* zu Anfang den Suchraum in der Mitte.
- Dann behandelt man nur noch eine der beiden Seiten.

Implementation

Beispiel einer binären Suche.

```
1  namen= ["Alice", "Bob", "Charly", "Doris", "Lena", "Viktor"]
2
3  # Binäre Suche nach "Lena" mit Python3
4  unterGrenze= 0
5  oberGrenze= namen.__len__() - 1
6
7  while unterGrenze != oberGrenze:
8    mitte= (unterGrenze + oberGrenze) // 2 # ganzzahlige Division
9    if namen[mitte] < "Lena":
10     unterGrenze= mitte + 1
11   else:
12     oberGrenze= mitte
13
14 # unterGrenze muss nun 4  sein
```

Beispiel einer binären Suche – Suche nach "Lena"



Eine allgemeine binäre Suche.

```
1  class SuchAlgorithmen:
2    # ...
3    def binaereSuche(self, zeichenketten: 'list', wert: 'str'):
4      # Finde Vorkommen von wert in zeichenketten (sortiert)
5      unterGrenze= 0
6      oberGrenze= zeichenketten.__len__() - 1
7
8      while unterGrenze != oberGrenze:
9        # Berechne Mitte des Intervalls
10       mitte= (unterGrenze + oberGrenze) // 2 # ganzzahlige Div.
11
12       if zeichenketten[mitte] < wert:
```

```
13          # Im oberen Intervall, erhöhe untere Schranke
14          unterGrenze= mitte + 1
15        else: # Unteres Intervall, senke obere Schranke
16          oberGrenze= mitte
17      return unterGrenze
```

Laufzeit

Aufgabe 2.2

Wie viele Vergleiche führt die Methode `binaereSuche` bei einem Array der Länge n

1. mindestens,
2. höchstens und
3. im Durchschnitt aus?

Vorherige Sortierung

Sortierung vor dem Suchen ist notwendig, wenn öfters gesucht werden soll.

- Binäre Suche *funktioniert nur* auf sortierten Daten.
- Will man unbedingt binäre Suche verwenden, so muss man die Daten *vor dem Suchen Sortieren*.
- Das Sortieren von Daten dauert aber (wesentlich) länger als *eine* lineare Suche.
- Sind die Daten aber einmal sortiert, gehen *nachfolgende* binäre Suchen schnell.

Folgerung
Vor dem Suche zu Sortieren lohnt sich nur, wenn in den Daten *mehrmals* gesucht werden soll.

(Genauer: Vorheriges Sortieren lohnt sich ab etwa $\log_2 n$ Suchen.)

Zusammenfassung – Lineare Suche und Binäre Suche

Lineare Suche und Binäre Suche – Zusammenfassung

1. *Lineare Suche* durchläuft einfach alle Elemente.
2. *Binäre Suche* verkleinert den Suchraum in jedem Schritt auf die Hälfte.
3. Binäre Suche funktioniert nur auf *sortierten* Daten.
4. Binäre Suche ist *schnell* ($\log_2 n$ Vergleiche im Schnitt).
5. Lineare Suche ist *langsam* ($\frac{n}{2}$ Vergleiche im Schnitt).

2.3 Sortieralgorithmen – Skat, Telefonbuch, Atome

Wie wir schon im vorherigen Abschnitt gesehen haben, ist der zweite Hauptsatz der Thermodynamik für Informatiksysteme ein echtes Problem: Ständig wird alles unordentlich. Man kann sich, wie im letzten Abschnitt geschehen, damit behelfen, ständig in den Daten im Speicher herumzusuchen. Wir haben aber auch gesehen, dass eine solche Suche besonders schnell geht, wenn die Daten sortiert – also wohlgeordnet – sind. Damit beißt sich die Katze in den Schwanz: Weil die Daten nicht sortiert sind, müssen wir suchen; aber um schnell zu suchen, müssen die Daten sortiert sein.

Um diesen mehr oder weniger gordischen Knoten zu durchschlagen, brauchen wir *Sortieralgorithmen*. Diese bekommen einen Array von Zahlen oder Dingen als Eingabe und ändern die Reihenfolge der Elemente des Arrays derart, dass hinterher alles schön sortiert ist. Danach fällt uns auch das Suchen viel leichter.

Das Sortieren von Zahlen ist schwieriger als das Suchen nach Zahlen. Dies ist zum einen ein Unglück, denn man muss komplexere Algorithmen lernen, diese sind schwieriger zu programmieren und sie sind langsamer als Suchalgorithmen. Aus Sicht Theoretischer Informatiker ist es hingegen ein Glück, denn so kann man viel mehr forschen, veröffentlichen und Drittmittel einwerben. Tatsächlich können Sie auch heute noch Artikel über Sortierverfahren auf renommierten Konferenzen vorstellen und in renommierten Zeitschriften veröffentlichen.

Sortieren lohnt sich nicht immer. Stellen Sie sich Ihren unaufgeräumten Schreibtisch vor. Sie suchen ein bestimmtes Blatt, von dem Sie wissen, dass es »da irgendwo sein muss«. Sie könnten nun hingehen und zunächst den ganzen Schreibtisch komplett aufräumen, alles gegebenenfalls abstauben und dann abheften, so dass sie am Ende das gewünschte Blatt mit einem Griff finden werden. In der Regel werden Sie dies aber nicht tun, sondern einfach kurz den Schreibtisch nach dem gesuchten Blatt

durchwühlen. Das Aufräumen lohnt sich nur, wenn Sie in den nächsten Tagen ständig unterschiedliche Dinge suchen werden. Ganz ähnlich die Sachlage beim Sortieren von Daten in Informatiksystemen: Muss man nur einmalig etwas in den Daten suchen, so lohnt es sich nicht, diese erst zu sortieren. Werden Sie aber immer und immer wieder in den Daten suchen, so empfiehlt sich die vorherige Sortierung.

2.3.1 Die Problemstellung

Motivation

Wo taucht Sortieren überall auf?

Problemvarianten

Was meint man mit »Sortieren« eigentlich?

Das einfachste Sortierproblem

Eingabe Array von Zahlen

Ausgabe Array mit denselben Zahlen, aber in der Reihenfolge so verändert, dass jede Zahl höchstens so groß wie ihr Nachfolger ist.

Eine »Veränderung in der Reihenfolge« nennt man auch *Permutation*.

Definition des allgemeinen Sortierproblems.

Das allgemeine Sortierproblem

Eingabe Array von Objekten, die sich vergleichen lassen

Ausgabe Eine Permutation der Objekte, so dass jedes Objekt in der neuen Reihenfolge kleiner oder gleich dem nachfolgenden ist.

Wünschenswerte Eigenschaften von Sortieralgorithmen.

Folgende Eigenschaften sind bei Sortieralgorithmen besonders wünschenswert:

1. Ein Verfahren ist *stabil*, falls sich die Reihenfolge von gleichen Elementen nicht ändert.

   Beispiel: Eine Adressliste wird nach Namen sortiert. Kommt vor der Sortierung Peter Müller aus Berlin vor Peter Müller aus Aachen, so soll dies nach der Sortierung immer noch der Fall sein.

2. Ein Verfahren ist *in-place*, falls es lediglich eine kleine Menge extra Speicher benötigt, falls es also keine Kopie des Arrays benötigt.

3. Das Verfahren sollte mit *möglichst wenigen* Vergleichen, Vertauschungen und Verschiebungen auskommen.

2.3.2 Sortieralgorithmen

Bubble-Sort

Der Bubble-Sort-Algorithmus

Idee

- Wir sind fertig, wenn für je zwei aufeinanderfolgende Objekte gilt, dass das erste kleiner oder gleich dem zweiten ist.
- Also suchen wir nach Paaren, bei denen dies nicht der Fall ist, und tauschen sie aus.

Erste Version von Bubble-Sort

```
1  def stupidSort(array):
2    i= 0
3    while i < array.__len__()-1:
4      if array[i] > array[i+1]:
5        # Korrigiere die Reihenfolge:
6        swap(array, i, i+1)
7
8        # Neustart
9        i= 0
10     else:
11       i= i+1
12
```

```
13 def swap(array, i, j):
14   # Vertausche array[i] und array[j]
15   temp= array[i]
16   array[i]= array[j]
17   array[j]= temp
18
19
20 if __name__ == "__main__":
21   zusort= [4,3,2,5,1]
22   print ("Vorher:␣",zusort)
23   stupidSort(zusort)
24   print ("Nachher:",zusort)
```

Verbesserungen der ersten Version

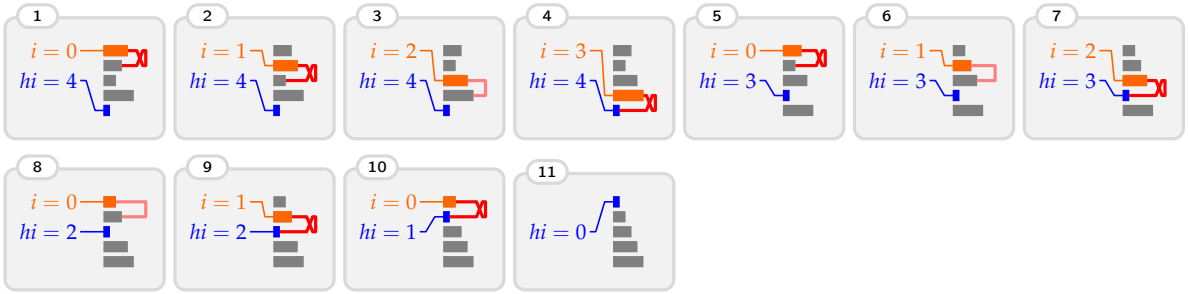
- Es macht keinen Sinn, nach jeder Vertauschung wieder am Anfang zu beginnen.
- Stattdessen macht man einfach mit dem nächsten Element weiter.
- Dann ist am Ende eines Durchgangs das größte Element am Ende.
- Jede folgende Runde kann dann eins früher enden.

Zweite Version von Bubble-Sort

```
1 def bubbleSort(array):
2   for hi in range(array.__len__()-1,0,-1):
3     for i in range(0,hi):
4       if array[i] > array[i+1]:
5         array[i+1],array[i]= array[i],array[i+1] # Vertauschen
6
7 if __name__ == "__main__":
8   zusort= [4,3,2,5,1]
9   print ("Vorher:␣",zusort)
10  bubbleSort(zusort)
11  print ("Nachher:",zusort)
```

Ablauf von Bubble-Sort an einem Beispiel

Der zu sortierende Array sei `array= [4,3,2,5,1]`.



Aufgabe 2.3

Bei einem Array der Länge n, wie viele

- Vergleiche macht Bubble-Sort *mindestens* (grob)?
- Vergleiche macht Bubble-Sort *höchstens* (grob)?
- Vertauschungen macht Bubble-Sort *mindestens* (grob)?
- Vertauschungen macht Bubble-Sort *höchstens* (grob)?

Vor- und Nachteile von Bubble-Sort

Vorteile

+ Einfach zu programmieren.
+ Einfach zu verstehen.
+ Kann leicht modifiziert werden, so dass er bei sortierten Daten sehr schnell ist.
+ In-place und stabil.

Nachteile

- Bei zufälligen Daten langsam, da viele Vergleiche.
- Bei fast sortierten Daten trotzdem langsam.

Selection-Sort

Der Selection-Sort-Algorithmus

Idee

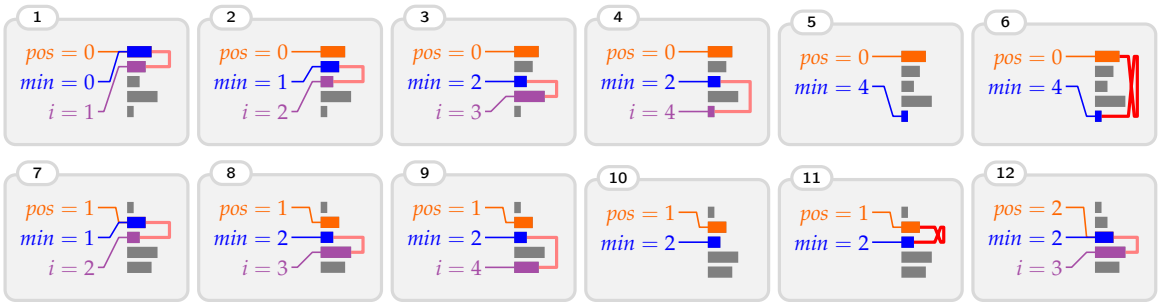
- Wir wollen möglichst wenig vertauschen.
- Deshalb suchen wir zunächst das kleinste Element im Array und tauschen es an die erste Stelle.
- Im Rest suchen wir dann wieder das kleinste und tauschen es an die zweite Stelle, und so weiter.

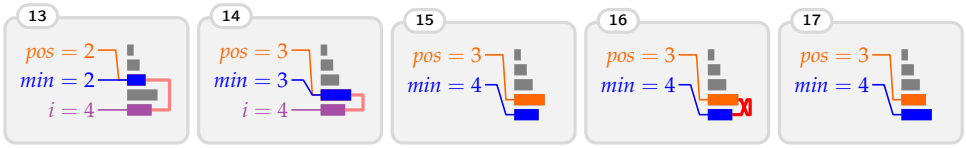
Selection-Sort

```
1  def selectionSort(array):
2    for pos in range(0, array.__len__()-1):
3      # Finde erstes Minimum nach pos
4      min= pos
5
6      for i in range(pos+1,array.__len__()):
7        if array[min] > array[i]:
8          min= i
9
10     array[pos],array[min]=  array[min],array[pos]
11
12 if __name__ == "__main__":
13   zusort= [4,3,2,5,1]
14   print ("Vorher:␣",zusort)
15   selectionSort(zusort)
16   print ("Nachher:",zusort)
```

Ablauf von Selection-Sort an einem Beispiel

Der zu sortierende Array sei `array= [4,3,2,5,1]`.





Vor- und Nachteile von Selection-Sort

Vorteile

+ Einfach zu verstehen.
+ Minimale Anzahl an Vertauschungen.
+ In-place.

Nachteile

- Etwas aufwändiger zu programmieren.
- Immer viele Vergleiche, selbst bei sortierten Daten.
- Nicht stabil.

Insertion-Sort

Der Insertion-Sort-Algorithmus

Idee

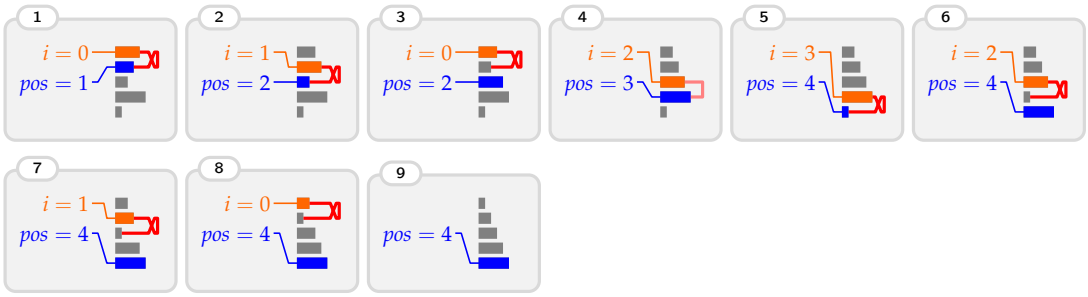
- Wir halten den ersten Teil des Arrays immer sortiert.
- Um den sortierten Teil des Arrays um ein Element zu erweitern, tauschen wir dies so lange nach links, bis es am Ziel angekommen ist.

Insertion-Sort

```
1  def insertionSort(array):
2    for pos in range(1, array.__len__()-1):
3      i= pos-1
4      while i >= 0 and array[i] > array[i+1]:
5        array[i],array[i+1]= array[i+1],array[i]
6        i= i - 1
7
8  if __name__ == "__main__":
9    zusort= [4,3,2,5,1]
10   print ("Vorher:␣",zusort)
11   insertionSort(zusort)
12   print ("Nachher:",zusort)
```

Ablauf von Insertion-Sort an einem Beispiel

Der zu sortierende Array sei `array= [4,3,2,5,1]`.



Vor- und Nachteile von Insertion-Sort

Vorteile

+ Einfach zu verstehen.
+ Sehr schnell bei sortierten und fast sortierten Daten.
+ In-place und stabil.

Nachteile

- Bei zufälligen Daten viele Vergleiche und Vertauschungen.

Aufgabe 2.4

Bei einem Array der Länge n, wie viele

- Vergleiche macht Insertion-Sort *mindestens* (grob)?
- Vergleiche macht Insertion-Sort *höchstens* (grob)?
- Vertauschungen macht Insertion-Sort *mindestens* (grob)?
- Vertauschungen macht Insertion-Sort *höchstens* (grob)?

Wie viele Vergleiche werden mindestens benötigt?

- Bubble-, Insertion- und Selection-Sort benötigten *grob* $n^2/2$ Vergleiche im schlimmsten Fall.
- Dies ist nicht optimal, Merge-Sort benötigt lediglich *grob* $n \log_2 n$ Vergleiche.
- Man kann sicherlich nicht mit weniger als mit $n - 1$ Vergleichen auskommen.
- Wie viele Vergleiche benötigt also ein *optimaler* Algorithmus?

Sortieren – Zusammenfassung

Sortieren – Zusammenfassung

1. Sortieren ist ein abstraktes Problem, das in sehr vielen Anwendungen auftaucht.
2. Verschiedene *Sortiertalgorithmen* haben Vor- und Nachteile.
3. Für kleine Datenmengen liefert *Insertion-Sort* gute Ergebnisse.
4. Die vorgestellten Sortieralgorithmen brauchen alle bis zu n^2 Vergleiche, optimal wären $n \log_2 n$ Vergleiche.

Vorhaben 3

Bäume[1]

3.1 Welche Kompetenzen sollen Sie in diesem Vorhaben erwerben?

Die Schülerinnen und Schüler

- stellen lineare und nichtlineare Strukturen grafisch dar und erläutern ihren Aufbau (IF1, D),
- ordnen Attributen, Parametern und Rückgaben von Methoden einfache Datentypen, Objekttypen sowie lineare und nichtlineare Datensammlungen zu (IF1, M),
- entwickeln iterative und rekursive Algorithmen unter Nutzung der Strategien »Modularisierung« und »Teile und Herrsche« (IF2, M),
- implementieren iterative und rekursive Algorithmen auch unter Vewendung von dynamischen Datenstrukturen (IF2, I),
- erläutern Operationen dynamischer (linearer oder nicht-linearer) Datenstrukturen (IF2, A),
- implementieren und erläutern iterative und rekursive Such- und Sortierverfahren (IF2, I),
- beurteilen die Effizienz von Algorithmen unter Berücksichtigung des Speicherbedarfs und der Zahl der Operationen (IF2, A),

Bevor Sie weiterlesen, nehmen Sie sich kurz Zeit, einen Blick aus dem Fenster zu wagen. Sollten Sie sich im Freien befinden, schauen Sie sich einfach etwas um; befinden Sie sich in einem Raum ohne Fenster, so nehmen Sie sich ein Beispiel an Ihren Vorfahren, die ihr Höhlendasein auch vor einigen tausend Jahren erfolgreich hinter sich gelassen haben und mit dieser Entscheidung im Großen und Ganzen immer noch zufrieden sind. Dort erblicken Sie mit hoher Wahrscheinlichkeit einen Baum. Wenn

[1] Teile dieses Abschnitts wurden der Veranstaltung »Einführung in die Informatik – Teil 1« aus dem Wintersemester 2012/2013 von Prof. Dr. Till Tantau zu dem Thema **Bäume** entnommen (dort: Kapitel 27) und an einigen Stellen geändert.

nicht, so sollten Sie vielleicht über einen Umzug nachdenken. Mit Ihrem geschulten Blick werden Sie unschwer die wichtigsten Teile eines Baumes entdecken: Da gibt es zum einen den Stamm, die Verzweigungen und Äste, die Blätter und sicherlich auch eine Wurzel, auch wenn Sie sie nicht sehen können, da sie *unter* der Erde ist.

Die Bestandteile eines Baumes finden sich auch in der Datenstruktur des Baumes wieder. Zunächst gibt es eine *Wurzel*, von der alles ausgeht. Ohne Wurzel kein Baum. Von der Wurzel gehen dann Äste aus, von denen wiederum Äste ausgehen und so weiter, bis die letzten Äste in Blättern enden. Die Informatikbäume haben keinen Stamm, man hat ihn wegrationalisiert, weshalb Informatikbäume eigentlich etwas bescheidener »Büsche« heißen sollten, aber diese kleine biologische Ungenauigkeit sei verziehen.

Was Biologen den Informatikern hingegen nicht verzeihen, ist der Umstand, dass Informatiker die Wurzel eines Baumes grundsätzlich *oben*, die Blätter hingegen *unten* anordnen. Aus diesem Grund sprechen Informatiker auch gerne statt von der Höhe eines Baumes von dessen *Tiefe*. Bei diesem biologischen Bildungsstand der Informatikerzunft stimmt es durchaus bedenklich, dass die großen Bio-Datenbanken von Informatikerinnen programmiert und am Leben gehalten werden. Leuten, die nach einem Ahornblatt als erstes in einer Tropfsteinhöhle suchen würden, vertrauen wir Genomdaten an! Dies sind übrigens dieselben Leute, die die Software von Flugzeugautopiloten programmieren und vorher in ihrem Studium gelernt haben, dass ein *Clean Crash* tolerierbar ist. Guten Flug.

3.2 Motivation

3.2.1 Modellierung

»Bäume« sind hierarchische Strukturen.

Bäume in der Informatik

Der Begriff *Baum* steht in der Informatik allgemein für eine *hierarchische Struktur*.

Beispiele: Dinge, die als Bäume modelliert werden können

- Dateibäume
- Menüs
- Verwaltungsstrukturen in Behörden und Firmen
- Genetische Stammbäume

Erste Motivation von Bäumen.

Baumartige Strukturen kommen so häufig vor, dass es sich lohnt,

1. ihre Eigenschaften allgemein zu studieren und
2. ihre Implementation zu beherrschen.

3.2.2 Bessere Datenstrukturen

Zweite Motivation: Ungelöste Probleme

Betrachten Sie eine der folgenden Situationen:

- Eine Telekom-Firma möchte die Telefonnummern der deutschen Bevölkerung verwalten.
- Eine Biotech-Firma möchte ihre Produktdatenbank verwalten.
- Eine Interessensgruppe möchte ihre Mitgliederliste verwalten.

Probleme

- Sortierte Arrays sind als Datenstrukturen wenig geeignet, da ständig neue Einträge hinzukommen und alte gelöscht werden.

  Einfügen und Löschen dauern bei Arrays zu lange.

- Sortierte Listen sind als Datenstrukturen wenig geeignet, da ständig Einträge gesucht werden müssen.

  Suchen dauert bei Listen zu lange.

Zweite Motivation von Bäumen.

In speziellen Bäume, genannt *Suchbäume*, kann man

- in Zeit $\mathcal{O}(\log n)$ Elemente einfügen,
- in Zeit $\mathcal{O}(\log n)$ Daten löschen,
- in Zeit $\mathcal{O}(\log n)$ Daten suchen.

Suchbäume sind also ein guter Ausgleich der Eigenschaften von Listen und Arrays.

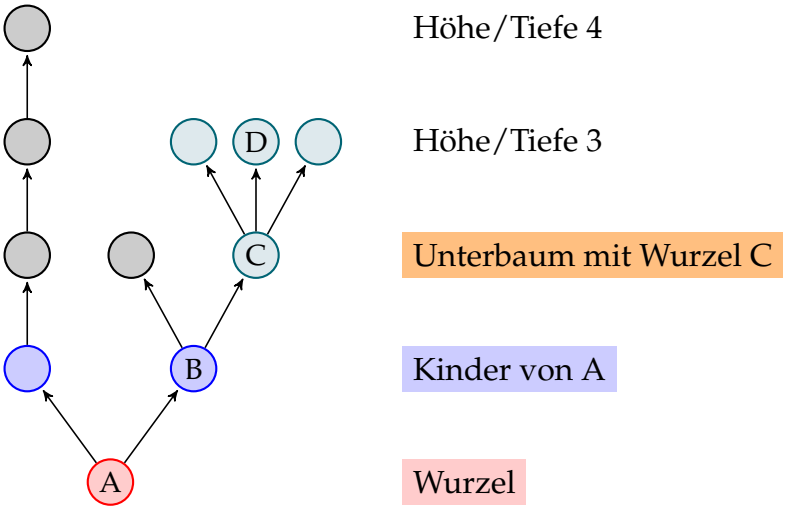
3.3 Begriffe

3.3.1 Der Begriff des Baumes

Woraus bestehen Bäume in der Informatik?

- Informatik-Bäume beginnen an einer *Wurzel*.
- Die Wurzel hat eine Reihe von *Kindern*.
- Jedes Kind kann *wieder Kinder* haben.
- Die Wurzel und alle ihre Kinder und Kindeskinder heißen *Knoten*.
- Knoten ohne Kinder heißen *Blätter*.

Beispiel eines Baumes.



Wichtige Begriffe zu Bäumen

Knoten Die Elemente des Baumes

Elternknoten Knoten, von denen Pfeile zu Kindern ausgehen.

Kinder Knoten, die durch Pfeile mit einem Elternknoten verbunden sind

Enkelkinder Kinder von Kindern

Nachfahren Kinder, Enkelkinder, Urenkelkinder, usw.

Vorfahren Elternknoten, Großelternknoten, usw.

Wurzel Ursprung des Baumes; einziger Knoten ohne Elternknoten

Blatt Knoten ohne Kinder

innerer Knoten Knoten mit Kindern

äußerer Knoten Andere Bezeichnung für Blätter

Tiefe Bei Knoten, die Entfernung zur Wurzel; bei Bäumen, die maximale Tiefe

Höhe Andere Bezeichnung für Tiefe (!)

Unterbaum Baum, der entsteht, wenn man nur einen Knoten und seine Nachfahren betrachtet

Teilbaum Baum, der entsteht, wenn man einige Knoten weglässt

Aufgabe 3.1

Schlagen Sie Definitionen der folgenden Begriffe vor:

- Geschwisterknoten,
- Schicht,
- Level,
- Einzelkind,
- Onkel,
- Tante.

3.3.2 Arten von Bäumen

Varianten von Bäumen

Wo stehen die Daten?

1. In allen Knoten (Normalfall).
2. Nur in den Blättern.
3. Zusätzlich oder ausschließlich an den Kanten.

Wie viele Kinder gibt es?

1. Beliebig viele Kinder an allen Knoten.
2. Genau zwei oder null Kinder (Binärbaum[2]).

[2] Zur Erarbeitung des Binärbaums steht das »Leitprogramm« **Einführung in verzweigte Datenstrukturen** zur Verfügung.

3. Höchstens zwei Kinder (auch Binärbaum genannt, grrr).
4. Genau zwei, drei oder null Kinder (2-3-Baum).

3.4 Bäume in Python

3.4.1 Beteiligte Klassen

Wie implementiert man Bäume in Python?

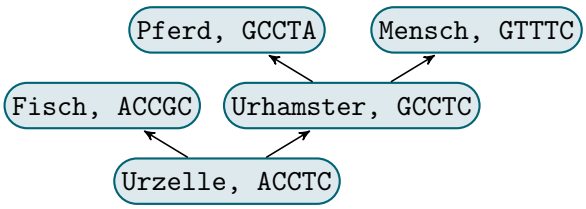
Wie bei Listen benutzt man mehrere Klassen, um Bäume zu implementieren:

1. Eine *Knotenklasse*. Diese entspricht den »Zellen« bei Listen.
2. Eine *eigentliche Baumklasse*, die einen Verweis auf die Wurzel enthält.
3. Optional eine Klasse für die *Werte*, die an den Knoten stehen.

   Beispiel: Bei einem phylogenetischen Baum ist dies die Klasse `Spezies`

Implementation phylogenetischer Bäume

Die `Spezies`-Klasse.




```
1 class Spezies:
2   def __init__(self, name: 'str', genombe: 'str'):
3     self.speziesName= name
4     self.genom      = genombe
```

Implementation phylogenetischer Bäume

Die Knoten-Klasse

Implementation mit beliebig vielen Kindern:

```
class Node:
  def __init__(self):
    # Die Spezies, die der Knoten speichert
    self.spezies   = None  # Spezies(..., ...)
    # Eine ganze Brut von Kindern...
    self.children= [] # Liste aus Elementen der Klasse Node()
```

Implementation mit höchstens zwei Kindern:

```
class Node:
  def __init__(self, spez: 'Spezies'):
    self.spezies   = spez  # Spezies(..., ...)
    self.leftChild = None  # Node
    self.rightChild= None  # Node
```

Bemerkungen:

- Zur Vereinfachung betrachten wie im Folgenden nur binäre Bäume.
- Man kann zusätzliche ein `parent` Attribut einführen. Dies entspricht dem `prev`-Attribut bei doppelt verketteten Listen.

Implementation phylogenetischer Bäume

Die eigentliche Baumklasse

```
class PhylogeneticTree:
  def __init__(self):
    self.root= None     # Wurzel - Klasse Node

  # Methoden
  def isEmpty(self):
    return self.root == None
```

3.4.2 Konstruktion

Konstruktion neuer Bäume

- Ein neu erzeugter Baum ist erstmal leer.
- Dann kann man nach und nach Elemente einfügen.

- Alternativ kann man aber auch neue Bäume erzeugen, indem man zwei bestehende zu einem neuen zusammenfasst.

```
class PhylogeneticTree:
  # Konstruktor, der einen Baum aufbaut
  # mit bereits konstruierten linken und rechten
  # Teilbäumen. Die Wurzel wird neu erzeugt.
  def __init__(self,
      spezies: 'Spezies',
      left: 'PhylogeneticTree', right: 'PhylogeneticTree'):
    self.root= Node(spezies)
    if left != None:
      self.root.leftChild= left.root
    if right != None:
      self.root.rightChild= right.root
```

Aufgabe 3.2

Visualisieren Sie die Objekte und Verweise, die durch folgenden Code erzeugt werden:

```
if __name__ == "__main__":
  start = Spezies("first␣cell", "CCCT")
  human = Spezies("human",      "ACGT")
  monkey= Spezies("monkey",     "ACCT")
  fish  = Spezies("Hommingberger␣Gepardenforelle", "CTCT")

  myTree= PhylogeneticTree(start,                \
           PhylogeneticTree(monkey, None,        \
           PhylogeneticTree(human, None, None)   \
           ),                                    \
           PhylogeneticTree(fish, None, None)    \
          )
```

3.4.3 Einfügen und Löschen

Einfügen und Löschen in Bäumen

- Das Einfügen und Löschen einzelner Knoten in der Mitte eines Baumes ist *fummelig*.
- Einfacher ist es, ganze Teilbäume zu löschen oder einzufügen.

```
class PhylogeneticTree:
  def addTreeAsLeftChild(self,
      where: 'PhylogeneticTree',
      what: 'PhylogeneticTree'):
    where.leftChild= what.root
```

3.4.4 Traversierung

Traversierung von Bäumen

- *Traversieren* bedeutet, dass man alle Knoten eines Baumes abläuft.
- An jedem Knoten kann man nun etwas »tun«, beispielsweise, die Spezies ausdrucken oder verändern.
- Die *Reihenfolge*, in der die Knoten traversiert werden, ist bei Bäumen nicht ganz klar.
- Es gibt drei wichtige Reihenfolgen
  1. In-Order:

     Erst den linken Teilbaum, dann der Knoten, dann der rechte Teilbaum.
  2. Pre-Order:

     Erst der Knoten, dann der linke Teilbaum, dann der rechte Teilbaum.
  3. Post-Order:

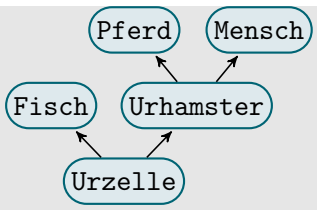
     Erst der linke Teilbaum, dann der rechte Teilbaum, dann der Knoten.

Beispiel einer Traversierung

```
class PhylogeneticTree:
  def printAllSpeziesPreOrder(self, node: 'Node'):
    if node != None:
      print(node.spezies.speziesName)
      self.printAllSpeziesPreOrder(node.leftChild)
      self.printAllSpeziesPreOrder(node.rightChild)
```

Aufgabe 3.3

Was geben die Methoden bei Eingabe `Urzelle` aus?




```
1  def printAllSpeziesPostOrder(self, node: 'Node'):
2    if node != None:
3      self.printAllSpeziesPostOrder(node.leftChild)
4      self.printAllSpeziesPostOrder(node.rightChild)
5      print(node.spezies.speziesName)
6
7  def printAllSpeziesInOrder(self, node: 'Node'):
8    if node != None:
9      self.printAllSpeziesInOrder(node.leftChild)
10     print(node.spezies.speziesName)
11     self.printAllSpeziesInOrder(node.rightChild)
```

Bäume – Zusammenfassung

1. Unter *Bäumen* versteht man in der Informatik alle Arten hierarchischer Strukturen.
2. Bäume bilden eine *Datenstruktur*, die ähnlich wie Listen aufgebaut ist.
3. *Einfügen und Löschen* ist bei Bäumen etwas komplizierter.
4. Man kann Bäume in unterschiedlichen Reihenfolgen *traversieren*.

Vorhaben 4

Graphen[1]

4.1 Welche Kompetenzen sollen Sie in diesem Vorhaben erwerben?

Die Schülerinnen und Schüler

- stellen lineare und nichtlineare Strukturen grafisch dar und erläutern ihren Aufbau (IF1, D),
- stellen iterative und rekursive Algorithmen umgangssprachlich und grafisch dar (IF2, D),
- entwickeln iterative und rekursive Algorithmen unter Nutzung der Strategien »Modularisierung«, »Teile und Herrsche« ***und »Backtracking«*** (IF2, M),
- implementieren iterative und rekursive Algorithmen auch unter Verwendung von dynamischen Datenstrukturen (IF2, I),
- erläutern Operationen dynamischer (linearer und nicht-linearer) Datenstrukturen (IF2, A),
- ***implementieren Operationen dynamischer (linearer oder nicht-linearer) Datenstrukturen (IF2, I),***
- beurteilen die Effizienz von Algorithmen unter Berücksichtigung des Speicherbedarfs und der Zahl der Operationen (IF2, A).

Bei Graphen geht es darum, Sachverhalte *darzustellen*, weshalb sich auch das Wort Graphik davon ableitet. Dabei sind, mathematisch gesehen, Graphen ein *sehr* allgemeines Konzept, fast so allgemein wie Mengen oder Relationen. Deshalb kann man auch alles und jedes als Graph darstellen, was auch gerne getan wird. Graphen modellieren »Dinge und ihre Beziehungen«.

[1] Teile dieses Abschnitts wurden der Veranstaltung »Einführung in die Informatik – Teil 1« aus dem Wintersemester 2012/2013 von Prof. Dr. Till Tantau zu dem Thema **Graphen** entnommen (dort: Kapitel 30) und an einigen Stellen geändert.

Anders als beispielsweise bei Mengen oder Relationen gibt es nicht »die« Definition von Graphen. Zwar ist die grundlegende Idee, Dinge und ihre Beziehungen zu modellieren, immer gleich, die konkreten mathematischen Ausprägungen können aber sehr variieren. Graphen können ge-allesmögliche sein, so zum Beispiel gerichtet, gewichtet, gelabelt, gefärbt und das auch alles in Kombination. Jede dieser Grapharten hat aber ihre Daseinsberechtigung, wie wir sehen werden. Um die mathematische Definition von Graphen in einem Informatiksystem abzubilden, werden in diesem Kapitel zwei Verfahren kurz skizziert: Adjazenzlisten und Adjazenzmatrizen.

Die Wirklichkeit lediglich als mehr oder weniger schöne Graphen zu modellieren, ist für sich genommen noch nicht sonderlich aufregend. Auf ein aufgeregtes »Das WWW kann man als ein Graph modellieren!« werden wohl die meisten Menschen zunächst etwas kühl »wie nett« erwidern. Spannend wird es erst, wenn man nun versucht, *Graphprobleme* zu lösen. Dies sind Probleme, bei denen die Eingaben ein Graph ist. Ein typisches Graphproblem ist das *Erreichbarkeitsproblem*, bei dem man wissen möchte, ob man von einem gegebenen Knoten einen anderen gegebenen Knoten erreichen kann. In diesem Vorhaben lernen wir einige solcher Probleme kennen.

4.2 Modellierung mit Graphen

4.2.1 Modellierung mittels Graphen

Graphen: Knoten und Kanten

Definition 1 (Graph)**.** Ein *Graph* besteht aus *Knoten*, die durch *Kanten* verbunden sind.

Die *Idee* ist, dass

- die Knoten *Dinge* modellieren und
- die Kanten *Beziehungen* modellieren.

Beispiel: Verkehrsnetze

Wir können *Verkehrsnetze* mittels Graphen wie folgt modellieren:

- Die *Knoten* sind Orte oder Straßenkreuzungen.
- Die *Kanten* sind Straßen.

Wenn wir ein Verkehrsnetz als Graph modellieren, welche Daten sollten wir dann speichern betreffend

- die Knoten (die Orte) und
- die Kanten (die Straßen)?

Beispiele aus der Molekularbiologie

Szenario: Moleküle
Wir können *Moleküle* mittels Graphen wie folgt modellieren:

- Die *Knoten* sind die Atome.
- Die *Kanten* sind die Bindungen.

Szenario: Genregulation
Wir können *Genregulation* mittels Graphen wie folgt modellieren:

- Die *Knoten* sind Gene.
- Die *Kanten* sind die Abhängigkeiten zwischen Genen; Kanten geben also an, wie Gene einander beeinflussen.

4.2.2 Arten und Formalisierungen

Es gibt vieles, was man bei Graphen festlegen kann.

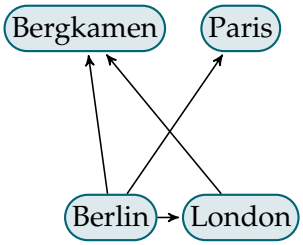
- Welche Art von Knoten gibt es?
- Sind Beziehungen gerichtet oder ungerichtet?
- Sind die Knoten beschriftet?
- Sind die Kanten beschriftet?
- ...

Der mathematische Begriff des Graphen.

Definition 2 (Graph)**.** Ein *Graph* besteht aus einer Knotenmenge V und einer Kantenmenge $E \subseteq V \times V$.

Solche Graphen nennen wir auch *gerichtete Graphen*.

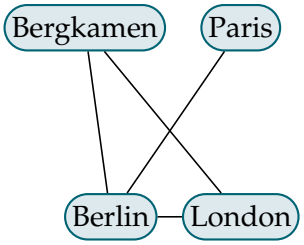
Beispiel: Gerichteter Graph
$V = \{\text{Berlin}, \text{London}, \text{Paris}, \text{Bergkamen}\}$.
$E = \{(\text{Berlin}, \text{London}), (\text{Berlin}, \text{Paris}), (\text{Berlin}, \text{Bergkamen}), (\text{London}, \text{Bergkamen})\}$



Spezielle Graphen: Ungerichtete Graphen.

- Bei *ungerichteten* Graphen haben die Kanten keine Richtung.
- Dies kann man mathematisch unterschiedlich modellieren. Typisch ist, dass man annimmt, dass E symmetrisch ist. (Das heißt, falls $(u, v) \in E$, so auch $(v, u) \in E$.)

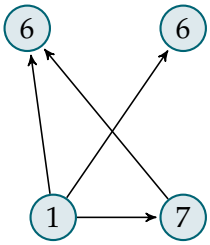
Beispiel: Ungerichteter Graph



Spezielle Graphen: Knotengewichtete Graphen.

- Bei *knotengewichteten* Graphen wird jedem Knoten noch eine Zahl zugeordnet, genannt *Gewicht* des Knotens.

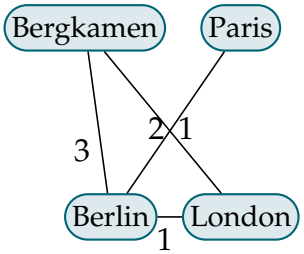
Beispiel: Knotengewichteter Graph



Spezielle Graphen: Kantengewichtete Graphen.

- Bei *kantengewichteten* Graphen wird jeder Kante noch eine Zahl zugeordnet, genannt *Gewicht* der Kante.

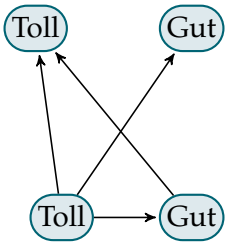
Beispiel: Kantengewichteter Graph



Spezielle Graphen: Gelabelte Graphen.

- Manchmal möchte man den Knoten und/oder den Kanten noch ein *Label* (einen kleinen Notizzettel) ankleben.

Beispiel: Knotengelabelter Graph



4.2.3 Graphprobleme

Was sind Graphprobleme?

Definition 3. *Graphprobleme* sind alle Problemstellungen, bei denen die Eingaben (kodierte) Graphen sind.

Insbesondere gibt es Graphprobleme als

1. Entscheidungsprobleme, wobei dann die Frage ist, ob ein gegebener Graph eine bestimmte Eigenschaft hat wie »Gibt es einen Weg vom ersten zum letzten Knoten?«; sowie als
2. Optimierungsprobleme, wobei dann die Frage ist, wie eine optimale Lösung aussieht (»Wie sieht der kürzeste Weg vom ersten zum letzten Knoten aus«).

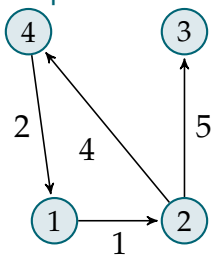
4.3 Graphen in Python

4.3.1 Adjazenzmatrizen

Was ist eine Adjazenzmatrix?

- Ein Graph habe n Knoten.
- Bilde nun eine Tabelle, die n Zeilen und n Spalten hat.
- Trage in die i-te Zeile und die j-te Spalte das Gewicht der Kante vom i-ten Knoten zum j-ten Knoten ein. Trage dort eine 0 ein, wenn es keine Kante gibt.

Graph



Adjazenzmatrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 5 & 4 \\ 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{pmatrix}$$

Graphen als Adjazenzmatrizen in Python

```
1  class GraphAlsAdjazenzMatrix:
2    def __init__ (self, zahlDerKnoten):
3      self.knotenZahl= zahlDerKnoten
4      self.gewichte= [[0]*zahlDerKnoten for i in
           range(zahlDerKnoten)]
5
6    def fuegeKnotenZu(self, u, v, knotenGewicht):
7      self.gewichte[u-1][v-1]= knotenGewicht
8
9    def entferneKnoten(self, u, v):
10     self.gewichte[u-1][v-1]= 0
11
12   def verbunden(self, u, v):
13     return self.gewichte[u-1][v-1] > 0 # Boole'scher Ausdruck
14
15   def gibGewicht(self, u, v):
16     return self.gewichte[u-1][v-1]
```

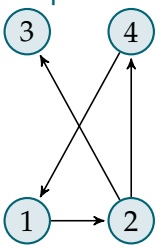
4.3.2 Adjazenzlisten

Was ist eine Adjazenzliste?

- Ein Graph habe n Knoten.

- Für jeden Knoten wird eine Liste gespeichert, in der alle Knoten gespeichert sind, zu denen er eine Kante hat.

Graph



Adjazenzlisten
Liste von Knoten 1: 2
Liste von Knoten 2: 3, 4
Liste von Knoten 3: leer
Liste von Knoten 4: 1

4.4 Graphproblem: Der Handlungsreisende

4.4.1 Problemstellung

Problem des Handlungsreisenden.

- Ein Handlungsreisender möchte in einer Rundreise eine Reihe von Städten besuchen.
- Dabei soll er jede Stadt genau einmal besuchen.
- Die Benzinkosten für eine Fahrt zwischen zwei Städten hängt linear von deren Entfernung ab.
- Ziel ist es, eine möglichst billige Rundreise zu finden.

Formalisierungen des Handlungsreisenden-Problems

Euklidisches Handlungsreisenden-Problem

Eingaben Ein Menge von Punkten in der Ebene.

Ausgabe Eine Rundreise (Folge der Punkte, die jeden Punkt genau einmal enthält), so dass die Summe der Länge der Strecken entlang der Rundreise minimal ist.

Allgemeines Handlungsreisenden-Problem

Eingaben Ein kantengewichteter Graph.

Lösungen Rundreise (Folge von miteinander verbundenen Knoten, die jeden Knoten genau einmal enthält), so dass die Summe der Gewichte der Kanten entlang der Rundreise minimal ist.

Zusammenfassung – Graphen

Graphen – Zusammenfassung

1. Graphen bestehen aus *Knoten* und *Kanten*.
2. Graphen *modellieren* viele unterschiedliche Dinge und es gibt sie in vielen Varianten.
3. Graphen kann man als *Adjazenzlisten* oder als *Adjazenzmatrizen* speichern.
4. Ein *Graphproblem* ist ein Problem, bei dem die Eingaben Graphen sind.

Vorhaben 5

Vielfalt beim Einsatz von Klassen durch Abstrakte Klassen, Polymorphie und MVC[1]

5.1 Welche Kompetenzen sollen Sie in diesem Vorhaben erwerben?

Die Schülerinnen und Schüler

- analysieren und erläutern objektorientierte Modellierungen (IF1, A),
- modellieren abstrakte und nicht abstrakte Klassen unter der Verwendung von Vererbung durch Spezialisierung und Generalisierung (IF1, M),
- verwenden bei der Modellierung geeigneter Problemstellungen Möglichkeiten der Polymorphie (IF1, M),
- implementieren Klassen in einer Programmiersprache auch unter Nutzung dokumentierter Klassenbibliotheken (IF1, I),
- analysieren und erläutern Algorithmen und Programme (IF2, A),
- modifizieren Algorithmen und Programme (IF2, I),
- modellieren Klassen mit ihren Attributen, Methoden und ihren Assoziationsbeziehungen unter Angabe von Multiplizitäten (M),
- stellen die Kommunikation zwischen Objekten grafisch dar (D),
- nutzen das verfügbare Informatiksystem zur strukturierten Verwaltung von Dateien unter Berücksichtigung der Rechteverwaltung (K),
- dokumentieren Klassen (D).

[1] Teile dieses Abschnitts wurden dem vorbereiteten Skriptum der Veranstaltung »Einführung in die Informatik – Teil 1« aus dem Wintersemester 2012/2013 von Prof. Dr. Till Tantau zu dem Thema **Abstrakte Datenstruktur** entnommen und an einigen Stellen geändert.

Beim Modellieren und beim Programmieren stellen wir häufig fest, dass Elemente, die wir schon einmal entwickelt haben, in ähnlicher Form wieder auftauchen. Wiederholt auftauchende Problemsituationen und damit verbunden – wiederholt in ähnlicher Form auftretende Problemlösungen – können in den Wissenschaften, in Handwerken und im Alltag identifiziert werden. Wird eine solche (Problem-)Situation identifiziert, kann man auf die Lösungsideen, die schon entwickelt wurden, zurückgreifen.

Um das Konzept zu verstehen, werden wir in diesem Vorhaben zwei Zugänge der Informatik kennenlernen, die uns Hinweise geben, wie eine Form der Abstraktion durch die Trennung der Schnittstelle von der Implementierung geleistet und realisiert werden kann und die komplexere, bereits entwickelte Problemlösungen in Form von Musterlösungen verfügbar gemacht werden können.

Die Vorüberlegungen führen dazu, dass bezüglich der abstrakten Datenstrukturen folgende Detailkompetenzen ausgewiesen werden können:

- Das Konzept der abstrakten Datenstruktur kennen.
- Wichtige abstrakte Datenstrukturen kennen.
- Vor- und Nachteile von Listen und Arrays benennen.

5.2 Konzept des abstrakten Datentyps (ADT)

5.2.1 Interface versus Implementation

Die Trennung von Interface und Implementation.

- In der Informatik versucht man, folgende Fragen zu *trennen*:
  1. *Was* sollen die Objekte einer Klasse können?
  2. *Wie* sollen sie dies erreichen?
- Gründe für die *Trennung* sind:
  - Die erste Frage muss man klären, bevor andere Leute die Klasse benutzen können. Die zweite kann man später klären.
  - Hat man viele Klassen, so hängen die Antworten auf die erste Frage zusammen. Die Antworten auf die zweite Frage können separat gefunden werden.
  - Bei gleicher Antwort auf die erste Frage kann es unterschiedliche Antworten auf die zweite geben, die verschiedene Vor- und Nachteile haben.

- Die Antwort auf die erste Frage nennt man *das Interface* (die Schnittstelle), die Antwort auf die zweite Frage *die Implementation*.

Array oder Liste? Who cares?

Was soll eine Klasse zum Shotgun-Sequencing können?

Es ist (vereinfacht) eine Datenstruktur gesucht, die möglichst effizient folgendes leistet:

- In den Objekten sollen DNA-Sequenzen gespeichert werden.
- Ein Objekt soll die in ihm gespeicherte Sequenz ausgeben können.
- An die Sequenz eines Objektes soll man die Sequenz eines anderen Objektes anhängen können.

Wie soll die Klasse dies erreichen?

Antwort 1 Wir benutzen einen Array von `char`-Werten, also Zeichen, die die Basen speichern.

Antwort 2 Wir benutzen eine einfach verkettete Liste.

Antwort 3 Wir benutzen eine doppelt verkettete Liste.

Antwort 4 Wir benutzen ...

5.2.2 Idee des abstrakten Datentyps

- Bei einem *abstrakten Datentyp* schreibt man nun auf, welche Operationen möglich sein sollen und was diese leisten sollen.
- Man gibt also ein *Interface* an.

  Beispiel: Wir benötigen:

  - Ausgabe, also eine Print-Methode: `print ()`
  - Verkettungsmethode: `concat (dnaSequence: DNASequenceADT)`
- Es kann nun verschiedene *Implementationen* (Klassen) geben, die alle diese Methoden implementieren.
- Das Interface ist *abstrakt*, da es noch keine Implementation vorgibt. Deshalb spricht man von einem *abstrakten* Datentyp.

5.2.3 Syntax von Interfaces

Python hat eine spezielle Syntax für Interfaces.

- Wir werden Interfaces lediglich zur *Beschreibung* von abstrakten Datentypen nutzen.
- Man kann noch mehr machen mit Interfaces, dies werden wir aber hier nicht thematisieren.

```
from abc import ABCMeta, abstractmethod
# Interfaces werden ähnlich wie Klassen aufgeschrieben
class DNASequenceADT(metaclass=ABCMeta):
  @abstractmethod
  def print(self): pass

  @abstractmethod
  def concat(self, sequenz: "DNASequenceADT"): pass

  # Man listet die gewünschten Methoden auf.
  # Man darf keine Attribute angeben und die Methoden
  # dürfen nicht implementiert werden.
```

5.2.4 Beispiele von abstrakten Datentypen

DNA-Sequenzen

Der abstrakte Datentyp DNA-Sequenz.

Wir betrachten im Folgenden verschiedene *Implementationen* des folgenden abstrakten Datentyps:

```
class DNASequenceADT(metaclass=ABCMeta):
  @abstractmethod
  def print(self): pass

  @abstractmethod
  def concat(self, sequenz: "DNASequenceADT"): pass
```

Implementation 1: Arrays zur Speicherung der Sequenzen.

- Wir speichern die Sequenz der Basen *in einem Array*: Jedes Element des Arrays enthält einfach eine Base.
- Die *erste Klasse*, die das Interface `DNASequenceADT` *implementiert*, hat ein *Attribut*, das dieses Array speichert.
- Erhält ein so implementiertes DNA-Sequenz-Objekt die Nachricht `print`, so iteriert die Implementation dieser Methode über die Elemente des Arrays.

Code der Implementation mit Arrays.

```
class DNASequence_1:
  __metaclass__= DNASequenceADT

  def __init__(self, base: "DNASequenceADT"):
    # Erzeuge eine DNA-Sequenz mit nur einer Base
    self.bases= base

  # Methoden
  def __len__(self):
    return self.bases.__len__()

  def __iter__(self):
    return self.bases.__iter__()

  def print(self):
    for elem in self.bases:
      print (elem, end= "")
    print ()
```

```
  def concat (self, seq: "DNASequenceADT"):
    # Die Basen aus seq sollen an das Ende von bases
    # angefügt werden.

    # Neuen Array erzeugen:
    new_array= [""] * (self.bases.__len__() + seq.__len__())

    # Kopiere eigenen Basen an den Anfang:
    i= 0
    for elem in self.bases:
      new_array[i]= self.bases[i]
      i= i+1

    # Kopiere die anderen Basen dahinter:
```

```
15      for elem in seq:
16        new_array[(i-1) + self.bases.__len__()]= elem
17        i= i+1
18
19      # Der neue Basenarray ist nun der aktuelle
20      self.bases= new_array
```

Folgender Code erzeugt mehrere Objekte (intern auch Arrays!).

Visualisieren Sie alle Objekte und alle Verweise.

```
1  print ("Interface␣wird␣mit␣Array␣realisiert:")
2  a= DNASequence_1('A')
3  b= DNASequence_1('C')
4
5  a.print()
6  b.print()
7
8  a.concat(b)
9  a.print()
```

Implementation 2: Liste zur Speicherung der Sequenzen.

- Man kann *eine einfach verkettete Liste* zur Speicherung der Sequenzen nutzen: Jede Zelle der Liste enthält einfach eine Base.
- Die *zweite Klasse*, die das Interface `DNASequenceADT` implementiert, hat diesmal ein Attribut, das *die erste Zelle der Liste speichert*.
- Erhält ein so implementiertes DNA-Sequenz-Objekt die Nachricht `print`, so iteriert die Implementation dieser Methode von der erste Zelle aus über alle Zellen.

Code der ersten Implementation mit Listen.

```
1  class DNASequence_2:
2    __metaclass__= DNASequenceADT
3
4    # Konstruktor
5    def __init__(self, bas: "DNASequenceADT"):
6      # Erzeuge eine DNA-Sequenz mit nur einer Base
7      self.start      = Cell()
8      self.start.base= bas
9      self.start.next= None
10
```

```
  # Methoden
  def print(self):
    cursor= self.start
    while cursor != None:
      print(cursor.base,end= "")
      cursor= cursor.next
    print()
```

```
  def concat (self, seq: "DNASequenceADT"):
    # Die Basen in seq sollen an das Ende von bases
    # angefügt werden.

    # Finde das Ende:
    cursor= self.start
    while cursor.next != None:
      cursor= cursor.next

    # Hänge seq an:
    cursor.next= seq.start
```

Folgender Code erzeugt mehrere Objekte (intern mehrere Zellen). Visualisieren Sie alle Objekte und alle Verweise.

```
print ("Interface␣wird␣mit␣Liste␣realisiert:")
a= DNASequence_2('A')
b= DNASequence_2('C')
a.print()
b.print()

a.concat(b)
a.print()
```

Implementation 3: Nochmal eine Liste zur Speicherung der Sequenzen.

- Ein *Nachteil* der zweiten Lösung ist, dass man beim *Verketten jedesmal das Ende suchen muss*.
- *Besser* ist es, *sich das Ende einfach zu merken*.

Code der zweiten Implementation mit Listen.

```
class DNASequence_3:
  __metaclass__= DNASequenceADT

  # Konstruktor
  def __init__(self, bas: "DNASequenceADT"):
    # Erzeuge eine DNA-Sequenz mit nur einer Base
    self.start     = Cell()
    self.start.base= bas
    self.start.next= None
    self.end= self.start
```

```
  # Methoden
  def print(self):
    cursor= self.start
    while cursor != None:
      print(cursor.base,end= "")
      cursor= cursor.next
    print()

  def concat (self, seq: "DNASequenceADT"):
    # Die Basen in seq sollen an das Ende von bases
    # angefügt werden.
    self.end.next= seq.start
    self.end     = seq.end
```

Visualisieren Sie die Objekte und Verweise:

```
  print ("Interface␣wird␣mit␣Liste␣(inkl.␣Endemarkierung)␣
     realisiert:")
  a= DNASequence_3('A')
  b= DNASequence_3('C')
  a.print()
  b.print()

  a.concat(b)
  a.print()
```

Implementation 4: Doppelt verkettete Liste.

- Die einfach verkettete Liste hat den *Nachteil*, dass man sie nur schwierig *rückwärts durchlaufen* kann.
- Eine Lösung hierzu ist die *doppelt verkettete Liste*.
- Jede Zelle speichert auch einen Verweis auf die Vorgängerzelle.

Code der dritten Implementation mit Listen.

```
class Cell:
  def __init__(self):
    self.base= ""
    self.next= None
    self.prev= None
```

```
class DNASequence_4:
  __metaclass__= DNASequenceADT

  # Konstruktor
  def __init__(self, bas: "DNASequenceADT"):
    # Erzeuge eine DNA-Sequenz mit nur einer Base
    self.start     = Cell()
    self.start.base= bas
    self.start.next= None
    self.start.prev= None
    self.end       = self.start
```

```
  # Methoden
  def print(self):
    cursor= self.start
    while cursor != None:
      print(cursor.base,end= "")
      cursor= cursor.next
    print()

  def concat (self, seq: "DNASequenceADT"):
    # Die Basen in seq sollen an das Ende von bases
    # angefügt werden.
    self.end.next = seq.start
    seq.start.prev= self.end
    self.end      = seq.end
```

Vergleich der Implementationen

Zeitverbrauch bei n Basen in der ersten Liste und m Base in der zweiten Liste.

Implementation	Konstruktor	`print`	`concat`
Arrays	$O(1)$	$O(n)$	$O(n+m)$
Liste	$O(1)$	$O(n)$	$O(n)$
Liste einfach, mit Ende	$O(1)$	$O(n)$	$O(1)$
Liste doppelt, mit Ende	$O(1)$	$O(n)$	$O(1)$

Zusammenfassung – Abstrakte Datenstruktur

Abstrakte Datenstruktur – Zusammenfassung

1. Eine *abstrakte Datenstruktur* beschreibt eine Menge von Methoden eines Datentyps und deren gewünschte Eigenschaften.
2. Eine abstrakte Datenstruktur kann man *unterschiedlich implementieren*.
3. Verschiedene Implementationen haben *verschiedene Vor- und Nachteile*.

5.3 MVC[2]

5.3.1 Muster

Sammlungen von erfolgreichen und von Expertinnen geprüften Musterlösungen können angelegt werden und führen dazu, dass bei neu auftretenden Problemsituationen Teile als *im Prinzip schon gelöst* gelten können, diese Elemente sind die Muster (engl.: pattern).

Es gibt einige Muster, die Sie bereits kennen: Standardverfahren zum Suchen und Sortieren; Möglichkeiten, einen binären Baum zu traversieren, ...

[2] Die Idee für die Variante, von der OOM zum ER zu kommen, ist in (Löffler 2010) dargestellt.

5.3.2 OOM mit MVC

Die Abkürzung MVC bedeutet: Model, View, Control. Mit *Model* wird das Fachmodell bezeichnet, das Modell, das Sie als Ergebnis der objektorientierten Modellierung entwickeln/mehrfach entwickelt haben. Dieses Modell wird auf einem Informatiksystem zum Ablauf gebracht. Im Folgenden wird also das System betrachtet, das auf einem Informatiksystem läuft.

Mit *View* wird dann die Anzeige oder die Darstellung von Ausgaben der modellierten Lösung (Model) im Ablauf bezeichnet, wohingegen mit *Control* die Verknüpfung des Fachmodells mit der Ein- und der Ausgabe gemeint ist.

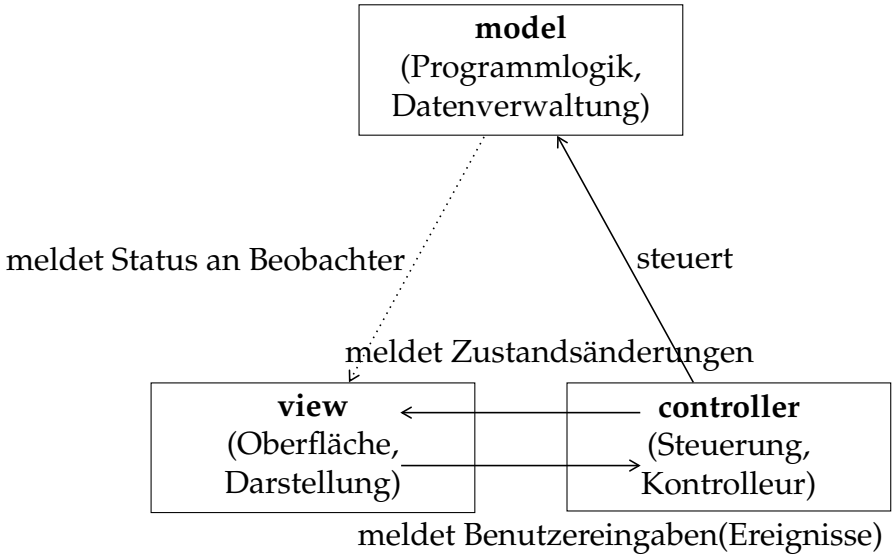


Abbildung aus (Löffler 2010, S. 14 – Abb. 6)

In der Informatik wurde eine Sammlung mit der Bezeichnung *Entwurfsmuster* von (Gamma u. a. 1996) veröffentlicht.

Um eine vorgenommene Modellierung für das MVC-Muster anzupassen, wird mit dem erstellten Klassendiagramm gearbeitet. Dies bedeutet im Umkehrschluss, dass Sie zunächst Ihre objektorientierte Modellierung vornehmen, um dann auf der Ebene des Klassendiagramms nachzusehen, wofür welche Klasse verantwortlich ist.

An dieser Stelle kommt das Muster ins Spiel: Sie müssen entscheiden, welche Klasse für welche Aufgabe zuständig ist und die Aufteilung gemäß MVC vornehmen. Häufig ist dies nicht sehr schwer – führt aber ab und zu dazu, dass man sich die Kommunikation zwischen den Objekten der Klassen genauer ansehen muss, um diese notwendige Trennung/Aufspaltung der Klassenverantwortlichkeiten sinnvoll durchführen zu können.

Eine Problemstellung mit Erweiterung und einigen Anwendungsfällen, an denen die Lösungsideen geprüft werden können, finden sich im Anhang A.4.

wird noch ergänzt

Vorhaben 6

Wissensbasierte Modellierung mit Datenbanken[1]

6.1 Welche Kompetenzen sollen Sie in diesem Vorhaben erwerben?

Die Schülerinnen und Schüler

- ermitteln für anwendungsbezogene Problemstellungen Entitäten, zugehörige Attribute, Relationen und Kardinalitäten (IF1, M),
- stellen Entitäten mit ihren Attributen und die Beziehungen zwischen Entitäten mit Kardinalitäten in einem Entity-Relationship-Diagramm grafisch dar (IF1, D),
- modifizieren eine Datenbankmodellierung (IF1, M),
- modellieren zu einem Entity-Relationship-Diagramm ein relationales Datenbankschema (IF1, M),
- bestimmen Primär- und Sekundär- und Fremdschlüssel (IF1, M),
- analysieren und erläutern eine Datenbankmodellierung (IF1, A),
- stellen grafisch den Ablauf einer Anfrage an ein Datenbanksystem dar (Client-Server-Modell) (IF4, M),
- erläutern die Eigenschaften normalisierter Datenbankschemata (IF1, A),
- überprüfen Datenbankschemata auf vorgegebene Normalisierungseigenschaften (IF1, D),
- überführen Datenbankschemata in die erste bis dritte Normalform (IF1, M),
- ermitteln Ergebnisse von Datenbankabfragen über mehrere verknüpfte Tabellen (D),

[1] Teile dieses Abschnitts wurden der Veranstaltung »Einführung in die Informatik – Teil 1« aus dem Wintersemester 2012/2013 von Prof. Dr. Till Tantau zu den Themen **Datenbanken** und SQL entnommen (dort: Kapitel 35–39) und an einigen Stellen geändert.

- analysieren und erläutern die Syntax und Semantik einer Datenbankabfrage (A),
- verwenden die Syntax und Semantik einer Datenbankabfragesprache, um Informationen aus einen Datenbanksystem zu extrahieren (I),
- erläutern Eigenschaften, Funktionsweisen und Aufbau von Datenbanksystemen unter dem Aspekt der sicheren Nutzung (A).

Was tun Computer eigentlich den ganzen Tag? Die Bezeichnung »Computer« ebenso wie das deutsche »Rechner« legt nahe, dass sie ständig spannende mathematische Probleme lösen, die die Menschheit der Weltformel etwas näher bringt. Die Wirklichkeit sieht leider etwas prosaischer aus: In Wirklichkeit ähneln Computer eher gewissenhaften Verwaltungsbeamten. Hauptsächlich sind sie damit beschäftigt, in irgendwelche Tabellen Daten einzufügen, sie zu löschen oder in diesen Tabellen zu suchen. (Daher erscheint es mir wenig wahrscheinlich, dass Computer, sollten sie jemals intelligent werden, sofort wie Skynet in *Terminator* die Weltherrschaft übernehmen werden. Realistischer erscheint, dass sie wie Marvin in *Per Anhalter durch die Galaxis* gelangweilt und depressiv ihre Einfüge-, Löschen- und Suchoperationen durchführen werden.)

Die Informatik-Teildisziplin der *Datenbanken* beschäftigt sich mit der Frage, wie man Programme so gestalten kann, so dass sie diese Einfügen-, Löschen- und Suchoperationen möglichst effizient hinbekommen. Man kann sagen, dass diese Teildisziplin mehr als erfolgreich war: In Form der *relationalen Datenbanken*, die durch die *Structured Query Language* angesprochen werden, gibt es Systeme, die nur noch wenige Wünsche offen lassen. Solche Datenbanken können riesige Mengen an Daten hocheffizient verwalten (»riesig« heißt »viele, viele Terabyte«). Sie sind sowohl frei wie kommerziell verfügbar. Sie werden durch eine einheitliche, standardisierte und recht einfache Sprache angesprochen. Schließlich (das freut den Theoretiker besonders) steckt hinter ihnen eine ebenso elegante wie einfache Theorie, die auch noch wirklich nützlich ist.

In diesem Teil über Datenbanken soll es zunächst darum gehen, wie Datenbanken aufgebaut sind und wie die angesprochene Theorie dahinter aussieht. Wir werden nur so genannte relationale Datenbanken behandeln (aus Gründen, die noch genauer erläutert werden). Dann schauen wir uns die Sprache SQL genauer an, mit der man mit Datenbanksystemen kommuniziert.

6.2 Datenbanksysteme

Das Problem, eine große Menge an Daten zu speichern und in ihnen zu suchen, ist uns schon häufiger über den Weg gelaufen. Wir haben auch schon mehr oder weni-

ger raffinierte Datenstrukturen kennengelernt, um Daten so zu speichern, dass man effizient darauf zugreifen kann: Man kann die Daten in einem sortierten Array speichern, in einer verketteten Liste oder in einem Suchbaum.

Trotzdem beschleicht Sie wahrscheinlich auch das Gefühl, dass die Programme, die die Daten einer Datenbank verwalten, keine einfachen Python-Programme mit ein paar Suchbäumen sein werden. Auf eine solche Datenbank greifen jede Sekunde viele unterschiedliche Benutzer gleichzeitig zu, der Service der Datenbank muss auch verfügbar bleiben, wenn eine der Platten der Server kaputtgeht, und noch viele weitere Anforderungen werden an Datenbanken gestellt, die wir mit unseren bisherigen Programmierkenntnissen nur schwer werden lösen können.

Glücklicherweise muss man das Rad nicht immer wieder neu erfinden. Vielmehr gibt es fix und fertig implementierte *Datenbanksysteme*, welche die oben angesprochenen Probleme (und noch einige Probleme, an die Sie oder ich noch gar nicht gedacht haben) in sehr effizienter Weise lösen. Der interne Aufbau von solchen Datenbanksystemen ist eine Wissenschaft für sich – uns wird nur interessieren, wie man sie benutzt. (Es sei aber verraten, dass intern sehr fortgeschrittene Suchbäume genutzt werden.)

Ein Datenbanksystem ist also ein Programm, das es in der Regel mehreren Personen gleichzeitig erlaubt, auf Daten zuzugreifen. Dabei werden die klassischen Grundoperationen unterstützt: Einfügen, Löschen und Suchen.

Wenn nun aber Datenbanksysteme keine Python-Programm sind, so stellt sich das *Modellierungsproblem* neu. Sie erinnern sich: In Python haben wir mittels Objekten und Klassen »die Wirklichkeit« modelliert und waren auch recht zufrieden damit. Leider hat sich bei Datenbanksystemen die Objektorientierung noch nicht so stark durchgesetzt wie bei Programmiersprachen – die meisten Datenbanksysteme sind leider nicht objektorientiert. Deshalb wird zur Modellierung der Wirklichkeit für nichtobjektorientierte Datenbanksysteme ein Vorläufer der Klassendiagramme verwendet: Die *Entity-Relationship-Diagramme*.

Zunächst soll es darum gehen, was man von einem Datenbanksystem prinzipiell erwarten darf. Danach werden Entity-Relationship-Modelle vorgestellt. Die genaue Syntax zur realen Kommunikation mit einem Datenbanksystem werden wir in dem Abschnitt über SQL kennen lernen.

6.2.1 Was sind Datenbanken?

Datenbanken und Datenbanksysteme.

Was sind Datenbanken?

Eine *Datenbank* ist ein strukturierte Sammlung von Daten.

- Man kann Daten in eine Datenbank *einfügen*.

- Man kann Daten aus einer Datenbank *löschen*.
- Man kann nach Daten in einer Datenbank *suchen*.

Was sind Datenbanksysteme?

Ein *Datenbanksystem* ist ein Programm, das eine oder mehrere Datenbanken verwaltet.

Was bieten Datenbanksysteme?

Ein Datenbanksystem kann:

1. Daten aus Datenbanken physikalisch effizient speichern.
2. Zugang zu Daten in Datenbanken herstellen.
   - Man kann Daten in Datenbanken *einfügen*.
   - Man kann Daten in Datenbanken *löschen*.
   - Man kann Daten in Datenbanken *suchen*.
3. Benutzer verwalten.
   - Mehrere Benutzer können *gleichzeitig* auf die Datenbanken zugreifen.
   - Benutzer können verschiedene *Rechte* haben (wie »darf nur Suchen«).

Sollte man eine Datenbank verwenden?

Was für Datenbanken spricht

+ Grundoperationen sind *viel schneller und besser* implementiert als man sie »selbst programmieren könnte«.
+ Daten sind immer *automatisch* auf externen Speichermedien[2] gesichert.
+ *Mehrere Benutzer* können *gleichzeitig* zugreifen.
+ *Verschiedene Programme* können *gleichzeitig* zugreifen.
+ Datenbanksysteme können *sehr billig* sein.

[2] Externe Speicher sind z. B. Festplatten oder SSDs.

Was gegen Datenbanken spricht

- Die Grundoperationen *Einfügen*, *Löschen* und *Suchen* kann man auch »selbst programmieren«.
- Man muss *neue Sprachen* lernen (zum Beispiel SQL).
- Datenbanksysteme können *sehr teuer* sein.

6.2.2 Aufbau von Datenbanken

Schichten, durch die eine Anfrage an eine Datenbank durchläuft.

1. *Anwendungsprogramme*

   Sie stellen Anfragen an die Datenbank mit Hilfe einer speziellen *Anfragesprache*.

2. *Externe Schemata*

   Die Anfragen einer Anwendung beziehen sich auf eines von mehreren externen Schemata. Sie »gaukeln einen bestimmten Aufbau der Daten vor«. So kann ein Schema Teile der Datenbank ausblenden, auf die ein Programm keinen Zugriff haben soll.

3. *Konzeptionelles Schema*

   Die Anfragen in Bezug auf ein externes Schema werden in Anfragen in Bezug auf das konzeptionelle Schema umgewandelt. Das konzeptionelle Schema beschreibt, wie die Daten logisch strukturiert sind. Dieses Schema ist das zentrale Schema, das man beim Aufbau einer Datenbank zu Anfang festlegen muss.

4. *Internes Schema*

   Anfragen werden weiter verwandelt in Anfragen in Bezug auf ein internes Schema. Es wird vom Datenbanksystem automatisch erstellt und ist eine optimierte, hocheffiziente Verwaltungsstruktur.

5. *Externe Speicher*

   Die Anfragen in Bezug auf des interne Schema werden in Zugriffe auf die *externen Speichermedien* umgewandelt.

Welche Schichten gehören zu einer Datenbank?

- Nur die drei mittleren Schichten gehören zu einer Datenbank und werden vom Datenbanksystem verwaltet.
- Die *externen Schemata* werden bei der Erstellung der Datenbank angegeben, können aber oft auch noch später geändert werden.
- Das *konzeptionelle Schema* wird bei der Erstellung der Datenbank einmal angegeben und dann in der Regel nicht mehr geändert.
- Das *interne Schema* wird automatisch erzeugt und man hat darauf keinen Zugriff.

Merke

Die Anfragesprachen für Datenbanken erlauben zwei unterschiedliche Dinge:

1. Erstellung und Veränderung der Schemata.
2. Modifikation der Daten, wenn das Schema festgelegt ist.

Datenbanken und Mehrbenutzerbetrieb.

- Wenn mehrere Anwender gleichzeitig Daten in der Datenbank *suchen*, ist das im Allgemeinen kein Problem.
- Wenn sie aber gleichzeitig Daten *ändern* wollen (schlimmstenfalls sogar die gleichen Daten), so können *vielfältige Konflikte* entstehen.
- *Ein Datenbanksystem kümmert sich um all diese Probleme und löst sie automatisch auf.*

6.2.3 Arten von Datenbanken

Arten von Datenbanken.

Datenbanksysteme unterscheiden sich unter anderem in folgenden Punkten.

- Menge der verwaltbaren Daten (von einigen Megabytes bis zu tausenden Terabytes).
- Anzahl der verwaltbaren Benutzer (von einem einzigen bis zu Millionen).
- Art der verwaltbaren Daten (Tabellen, Graphiken, Objekte, Filme).
- Art der verwaltbaren Schemata (relational, hierarchisch, objekt-orientiert).
- Geschwindigkeit und Sicherheit.
- Hersteller und Preis.

Arten von Schemata.

Ein fundamentaler Unterschied zwischen Datenbanken ist, wie ihre Schemata aufgebaut sein können:

relational In der Datenbank lassen sich (nur) Relationen (hocheffizient) speichern. Eine Relation setzt verschiedene Dinge in Beziehung, wie zum Beispiel Schafe mit den für sie sequenzierten Fragmenten.

hierarchisch In der Datenbank lassen sich (nur) hierarchische Strukturen speichern.

OO In der Datenbank lassen sich (nur) Objekte im Sinne der objektorientierten Programmierung speichern. Das Schema ist durch die Klassenstruktur gegeben.

Wir betrachten nur relationale Datenbanken.

Wir werden im Folgenden *nur relationale Datenbanken* betrachten. Dies hat verschiedene Gründe:

- Diese Datenbanksysteme sind *ausgereift*.
- Es gibt *frei verfügbare, gute* Implementationen von relationalen Datenbanken.
- Sie lassen sich *extrem gut optimieren* und sind daher oft *sehr effizient*.
- Es gibt eine *einheitliche, einfache Anfragesprache* für sie, nämlich *SQL*.

Hier ein paar Nachteile von relationalen Datenbanken:

- Das relationale Modell passt schlecht zum objektorientierten Modell, das Anwendungsprogramme benutzen.
- In manchen Situationen sind hierarchische Datenbanken wesentlich schneller.

6.3 Datenmodelle

Was sind Datenmodelle?

- Die verschiedenen Arten von Schemata spiegeln verschiedene Arten von *Datenmodellen* wider.
- Ein Datenmodell beschreibt *die konzeptionelle Struktur* der Daten.

Wir betrachten nur E/R-Modelle.

Es gibt zwei wichtige Arten von Datenmodellen:

- Entity-Relationship-Modelle.
  - Sie passen gut zu relationalen Datenbanken, weshalb wir diese betrachten werden.
  - Wie man vom E/R-Modell zu den Relationen in einer relationalen Datenbank kommt, wird im nächsten Abschnitt erklärt.
- UML-Modelle (unified modelling language)
  - UML-Modelle passen gut zu objektorientierten Programmen, weshalb sie in der Softwaretechnik viel eingesetzt werden.

6.4 Das E/R-Modell

6.4.1 Einführung

Was ist ein E/R-Modell?

- Das *Entity-Relationship-Modell* ist ein *Datenmodell*.
- Es dient dazu, *Entitäten* und deren *Beziehungen* zu beschreiben.
- Ein E/R-Modell legt fest, welche *Arten* von Entitäten es geben kann und welche *Arten* von Beziehungen.
- Es legt noch nicht fest, welche Entitäten und Beziehungen zwischen konkreten Entitäten es gibt.

Die Bestandteile eines E/R-Modells.

Ein E/R-Modell besteht aus drei Arten von Dingen:

1. *Entitätstypen*

   Dies sind Arten von Objekten oder Dingen, über die Daten in der Datenbank gespeichert werden sollen.

   Beispiel: Schafe, Gene

2. *Attribute*

   Dies sind Eigenschaften von Entitäten, die in der Datenbank gespeichert werden sollen.

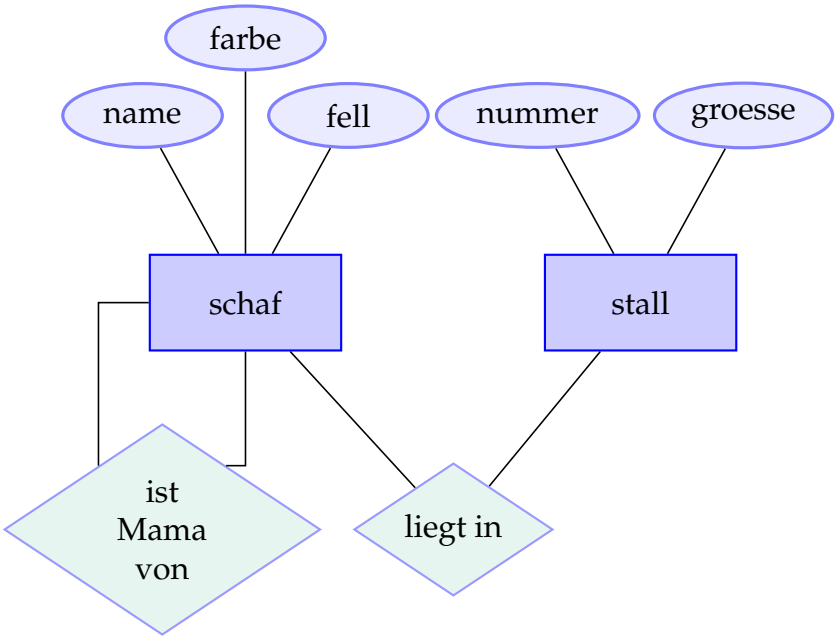
   Beispiel: Farbe des Schafs, Basenposition des Gens

3. *Relationships*

   Dies sind Beziehungen, die zwischen Entitäten bestehen.

   Beispiel: Schafe haben Gene, Gene wirken zusammen mit anderen Genen

Beispiel: Ein Entity-Relationship-Diagramm.



6.4.2 Entitäten

Näheres zu Entitäten.

- Eine *Entität* ist ein Objekt, über das wir Daten speichern.

  Beispiel: Das Schaf Dolly

- Eine *Entitätsmenge* ist eine Menge von Entitäten. Man spricht aber (salopp und fälschlicherweise) auch oft von *entities*, obwohl man Entitätsmengen meint.

  Beispiel: Schafe, schwarze Schafe

6.4.3 Attribute

Näheres zu Attributen.

- Eine *Attribut* ist eine Eigenschaft von Entitäten.

  Beispiel: Die Farbe von Schafen

- Die Menge aller Entitäten, die bestimmte Attribute haben, bilden den *Entitätstyp* dieser Attribute.

  Beispiel: Schafe haben eine Farbe, eine Fellart, einen Namen.

- In einem E/R-Modell zeichnet man Entitätstypen als *Rechtecke* auf.
- In einem E/R-Modell zeichnet man Attribute als *Ovale* auf mit einer Kante zum Entitätstyp.

Vereinbarungen zur Darstellung finden sich im Anhang C.7.

Entitätstabellen enthalten Entitäten mit ihren Attributen.

- Für jeden Entitätstyp gibt es in der Datenbank später eine *Tabelle*.
- Diese enthält für jede Entität eine *Zeile*.
- Die *Spalten* sind die Attribute der Entität.
- Ein Spalte, anhand derer man die Entität eindeutig identifizieren kann, heißt *Schlüsselattribut* oder einfach nur *Schlüssel*.

Beispiel

name	farbe	fell
Dolly	weiß	lockig
Max	schwarz	wuschelig
Peter	schwarz	wuschelig
Flauschi	beige	glatt

6.4.4 Relationships

Näheres zu Relationships.

- Eine *Relationship* ist eine Beziehung zwischen zwei oder mehr Entitäten.

  Beispiel: Dolly *hat* das Asthma-Gen

- Der *Relationshiptyp* beschreibt, dass Beziehungen zwischen den Entitäten bestimmter Typen bestehen können.

  Beispiel: Schafe *haben* Gene

- In einem E/R-Modell zeichnet man Relationshiptypen als *Rauten* auf – mit Kanten zu den Entitätstypen.

Relationship-Tabellen enthalten Tupel von Entitätsschlüsseln.

- Für jeden Relationshiptyp gibt es in der Datenbank später wieder eine *Tabelle*.
- Diese enthält für jede Relationship eine *Zeile*.
- Die *Spalten* sind die Schlüssel der beteiligten Entitätstypen.

Beispiel

Die Tabelle des Relationshiptyps »haben«.

schafs-name	gen-name
Dolly	Asthma-Gen
Dolly	Intelligenz-Gen
Max	Asthma-Gen
Peter	Intelligenz-Gen

Relationships können auch Attribute haben.

- Auch Relationships können *Attribute* haben.
- Diese werden wie bei Entitäten im Diagramm als Ovale dargestellt.

Beispiel

Die Tabelle des Relationshiptyps »leiden an«.

schafs-name	krankheit	seit
Dolly	Scrapie	1. Juli 2015
Dolly	MKS	2. Juli 2015
Max	MKS	2. Juli 2015
Peter	Scrapie	5. Juli 2015

Zusammenfassung – Einführung Datenbanken

Einführung Datenbanken – Zusammenfassung

1. Datenbanken ermöglichen *effizientes* und *sicheres* Einfügen, Löschen und Suchen von Daten.
2. Datenbanken besitzen ein *konzeptionelles Schema*, das vorschreibt, wie die zu speichernden Daten organisiert sind.
3. E/R-Modelle bestehen aus *Entitäten*, *Attributen* und *Relationships*.

6.5 SQL

Im ersten Teil ging es darum, was eine Datenbank *prinzipiell* ist und was *prinzipiell* in sie hinein soll. Mit Prinzipienreiterei kommt man aber auf Dauer nicht weit, irgendwann muss auch mal Butter bei die Fische – konkret: wir brauchen ist eine *Sprache* zur Kommunikation mit Datenbanksystemen. Dabei handelt es sich weder um eine Seitenbeschreibungssprache wie HTML oder LaTeX (schließlich gibt es hier keine Seiten, die beschrieben werden wollen) noch um eine Programmiersprache wie Python, denn wir wollen ja keine Schleifen durchlaufen, sondern Tabellen verwalten. Die Sprache SQL, die zur Kommunikation mit Datenbanksystemen dient, ist deshalb eine ganz eigene Sprache, die eigentlich keine Gemeinsamkeiten mit anderen Sprachen hat. Sie werden also eine neue Sprache lernen müssen.

So weit die schlechte Nachricht. Die gute Nachricht ist, dass man auch »nur« SQL lernen muss; im Gegensatz zu den normalen Programmiersprachen, die es wie Sand am Meer gibt, gibt es in der Welt der relationalen Datenbanken eigentlich nur SQL. Kann man diese Sprache, so kann man mit jedem Datenbankserver reden, egal ob es sich um einen Miniserver für ein einzelnes Programm handelt oder um das Data-Warehouse eines Großunternehmens. (Der Teufel steckt aber natürlich auch bei SQL im Detail.)

Zunächst einmal geht es darum, Daten in eine Datenbank *hinein* zu bekommen und zu verändern. Dafür sind, wenig überraschend, Befehle mit Namen wie »create«, »insert into« oder »delete« zuständig. Um Daten *hinaus* zu bekommen benutzt man den »select«-Befehl, der aber etwas komplexer ist und dem zwei Abschnitte gewidmet werden.

Mit den oben genannten Befehlen hat man alles beisammen, um Datenbanksysteme zu nutzen: Möchte man eine Datenbank anlegen, so überlegt man sich zunächst mit einem E/R-Diagramm, wie die Daten zu modellieren sind.

Dann legt man mit diversen Create-Befehlen die notwendigen Tabellen an. Später werden diese Tabellen mit Insert-Befehlen gefüllt und mit Delete-Befehlen aktualisiert. Select-Befehle verwendet man schließlich, um im laufenden Betrieb Daten aus der Datenbank zu extrahieren und damit Information zu generieren.

6.5.1 SQL zur Kommunikation

Was ist SQL?

- SQL steht für *structured query language*, deren Syntax »angeblich menschenlesbar« ist.
- Sie erfüllt drei Hauptfunktionen:
  1. Sie stellt Befehle zur Verfügung, um Relationen und Datenbanken (Ansammlungen von Relationen) zu erstellen und zu verwalten.
  2. Sie stellt Befehle zur Verfügung, um Einträge in Tabellen *einzufügen* und *zu löschen*.
  3. Sie erlaubt es, *Anfragen* zu formulieren wie »Wer ist der Vater von Dolly?«
- Die Sprache ist *deklarativ*. Das bedeutet, dass man bei Anfragen angibt, was man gerne hätte, aber nicht, wie man das berechnen sollte.

Prinzipielle Kommunikation mit einer Datenbank.

- Datenbanken werden von einem *Datenbanksystem* verwaltet.
- Das Programm, das die Verwaltung durchführt, heißt auch manchmal *Datenbankserver*.
- Um mit dem Datenbankserver zu reden, wird eine *Datenbank-Shell* benutzt. (Zur Erinnerung: Eine Shell ist ein Programm, mit dem man dialogbasiert mit einem anderen Programm spricht.)
- In der Datenbank-Shell gibt man SQL-Befehle ein, die an den Datenbankserver übermittelt werden. Die Antworten des Servers zeigt die Shell dann an.[3]

[3] Wir arbeiten mit dem Datenbankmanagementsystem (DBMS) SQLite und der Python-Schnittstelle sqlite3, damit auf Ihren mobilen Systemen die Datenbanken erstellt, gewartet und genutzt werden können.

6.5.2 SQLite und Python

Erste nützliche Befehle.

Auswahl der Datenbank.

Da wir mit Hilfe von Python auf Datenbanken zugreifen möchten, die sich auf dem eigenen mobilen System befinden, benutzen wir eine Schnittstelle, die – auch mobil – zur Verfügung steht, um mit Datenbanken zu arbeiten.[4]

Auswahl der Datenbankdatei durch Angabe des Pfades. Dazu wird der komplette Pfad bis zu der Datei, die die Datenbank enthält, angegeben:

```
sqlite_datei= "/sdcard/db/lh_db.sqlite"
```
[5]

Für Python3 steht eine Schnittstelle (genauer: ein Modul) bereit, mit dem aus Python eine Verbindung zu der Datenbank, die sich in der Datei befindet, hergestellt wird:

```
1 import sqlite3
2 verbindungDB= sqlite3.connect(sqlite_datei)
3 pos_zeig= verbindungDB.cursor()
```

Zwei weitere Methoden der Datenbankverbindung aus dem Python-Modul sqlite3 werden benötigt, um die Arbeit(en) mit der Datei ordentlich zu beenden:

- Wurden einige Aktionen mit der Datenbankd durchgeführt, die *Änderungen* in der Datenbank zur Folge haben, muss mit `verbindungDB.commit()` bestätigt werden, dass die Änderungen auch tatsächlich in die Datenbankdatei aufgenommen werden.
- Durch `verbindungDB.close()` muss die Verbindung zur Datenbank am Ende *immer* geschlossen werden.

Erste nützliche Befehle.

Beschreibung der Tabellen.

Durch

[4] Als Quelle für die folgenden Hinweise wird (**Raschka_SQLITE-Python2014**) herangezogen.

[5] Wählen Sie für Ihre Datenbank als Präfix Ihre Initialien – bei mir ist dies "lh" – bitte legen Sie das Verzeichnis zunächst an. Das Anlegen eines Verzeichnisses kann (auch) innerhalb von Python erledigt werden. Mittels

```
1 from os import mkdir
2 mkdir("/sdcard/db")
```

wird das Verzeichnis angelegt.

```
tabelle_name1= 'lh_tabelle_1'
tabelle_name2= 'lh_tabelle_2'
```

werden Python-Objekte der Klasse Zeichenkette angelegt und mit den rechts stehenden Werten belegt. Semantisch sind dies unsere Tabellenbezeichner.

```
neues_feld= 'lh_1st_spalte'
feld_typ= 'INTEGER'
```

legt das Attribut fest, das wir verwenden wollen. Neben dem Bezeichner für das Attribut muss in SQL immer *vor der Benutzung* Art/Sorte/Typ für die Attributwerte festgelegt werden.

Nun wird die Aktion: »Anlegen einer Tabelle« auf der Datenbank ausgeführt:

```
pos_zeig.execute('CREATE␣TABLE␣{tn}␣({nf}␣{ft})'\
        .format(tn=tabelle_name1, nf=neues_feld, ft=feld_typ))
```

Eine weitere Tabelle wird angelegt, die sich in derselben Datenbank befinden soll.

```
pos_zeig.execute('CREATE␣TABLE␣{tn}␣({nf}␣{ft}␣PRIMARY␣KEY)'\
        .format(tn=tabelle_name2, nf=neues_feld, ft=feld_typ))
```

Durch `PRIMARY KEY` wird das *Schlüsselattribut* zur eindeutigen Kennzeichnung eines Datensatzes angegeben. Auf der Seite 97 sehen Sie eine weitere Möglichkeit, um Tabellen zu erstellen.

Erste nützliche Befehle.

Anzeigen einer Tabelle.

Ist man an einer bestimmten Tabelle interessiert, so werden Daten mittels `SELECT` aus der Tabelle ausgewählt.

Auswahl der Anzahl der Zeilen der Tabelle *tabelle* mit

```
    pos_zeig.execute('SELECT␣COUNT(*)␣FROM␣{}'\
        .format(tabelle_name))
```

Um den Inhalt der ganzen Tabelle auszuwählen, arbeitet man mit der Auswahlanweisung

```
    pos_zeig.execute('SELECT␣*␣FROM␣{}'\
        .format(tabelle_name))
```

Um die ausgewählten Zeilen auszugeben, wird mit die Methode *fetchall* verwendet:

```
1         print (pos_zeig.fetchall())
```

`SELECT * FROM table LIMIT 10` wählt die ersten 10 Zeilen der Tabelle aus.

6.6 Erstellen einer Datenbank

6.6.1 Vom E/R-Modell zur Datenbank

Wiederholung: Entitätstabellen enthalten Entitäten mit ihren Attributen.

- Für jeden Entitätstyp gibt es in der Datenbank eine *Tabelle*.
- Diese enthält für jede Entität eine *Zeile*.
- Die *Spalten* sind die Attribute der Entität.
- Ein Spalte, anhand derer man die Entität eindeutig identifizieren kann, heißt *Schlüsselattribut* oder einfach nur *Schlüssel*.

Beispiel

name	farbe	fell
Dolly	weiß	lockig
Max	schwarz	wuschelig
Peter	schwarz	wuschelig
Flauschi	beige	glatt

Wiederholung: Relationshiptabellen enthalten Tupel von Entitätsschlüsseln.

- Für jeden Relationshiptyp gibt es in der Datenbank wieder eine *Tabelle*.
- Diese enthält für jede Relationship eine *Zeile*.
- Die *Spalten* sind die Schlüssel der beteiligten Entitätstypen.

Beispiel

Die Tabelle des Relationshiptyps »haben«.

schafs-name	gen-name
Dolly	Asthma-Gen
Dolly	Intelligenz-Gen
Max	Asthma-Gen
Peter	Intelligenz-Gen

6.6.2 Anlegen von Tabellen

Wie legt man Tabellen an?

In der aktuellen Datenbank kann man Tabellen auch wie folgt anlegen:

```
1  import sqlite3
2
3  sqlite_datei= '/sdcard/db/lh_db.sqlite'
4
5  verbindungDB= sqlite3.connect(sqlite_datei)
6  pos_zeig= verbindungDB.cursor()
7  sql_kommandos= [
8  "CREATE TABLE schaf (name CHAR(20) PRIMARY KEY, farbe CHAR(20),
       fell CHAR(20))",
9  "CREATE TABLE stall (nummer INTEGER PRIMARY KEY, groesse REAL)",
10 "CREATE TABLE liegt_in (name CHAR(20), nummer INTEGER)"]
11
12 for kommando in sql_kommandos:
13   pos_zeig.execute(kommando)
```

Mit dem Python-Quellcode werden die drei Tabellen für die beiden Entitäten `schaf`, `stall` und für die Relationship `liegt_in` aus dem Beispiel auf Seite 89 angelegt.

Syntax des CREATE-Befehls.

- Dem `CREATE TABLE`-Befehl folgt der Name der Tabelle.
- Danach kommen in Klammern die Attribute der Tabelle.
- Die wichtigsten erlaubten Typen sind:
  - `INTEGER` ist eine aus max 64-Bit bestehende ganze Zahl
  - `REAL` ist eine 64-Bit Gleitkommzahl
  - `CHAR(n)` ist eine Zeichenkette der Länge maximal $n \leq 255$
  - `TEXT` ist ein String der Länge maximal 65535
  - `BLOB` (binary large object) ist ein (rotes blubberndes) Ding der Größe maximal 65536 Byte.

Wie fügt man etwas in eine Tabelle ein?

Man kann Werte in eine Tabelle wie folgt einfügen:

```
sql_kommandos= [
"""INSERT INTO schaf
  VALUES ("Dolly","weiss",  "lockig"),
         ("Flauschi",  "schwarz","wuschelig"),
         ("Peter",  "schwarz","wuschelig")""",
"""INSERT INTO stall
  VALUES (1,50.0),
         (5,25.0)""",
"""INSERT INTO liegt_in
  VALUES ("Dolly", 1),
         ("Flauschi", 5)"""
]

for kommando in sql_kommandos:
  pos_zeig.execute(kommando)

verbindungDB.commit()
verbindungDB.close()
```

Daten der angelegten Tabellen

```
Gesamtzeilenzahl: 3

Spalteninfo:
ID, Name, Typ, NichtNull, Voreinstellungswert, Primärschlüssel
(0, 'name', 'CHAR(20)', 0, None, 1)
(1, 'farbe', 'CHAR(20)', 0, None, 0)
(2, 'fell', 'CHAR(20)', 0, None, 0)

Einträge pro Spalte:
fell: 3
farbe: 3
name: 3
[('Dolly', 'weiss', 'lockig'), \
 ('Flauschi', 'schwarz', 'wuschelig'), \
 ('Peter', 'schwarz', 'wuschelig')]

Gesamtzeilenzahl: 2

Spalteninfo:
ID, Name, Typ, NichtNull, Voreinstellungswert, Primärschlüssel
```

```
(0, 'nummer', 'INTEGER', 0, None, 1)
(1, 'groesse', 'FLOAT', 0, None, 0)

Einträge pro Spalte:
groesse: 2
nummer: 2
[(1, 50.0), (5, 25.0)]

Gesamtzeilenzahl: 2

Spalteninfo:
ID, Name, Typ, NichtNull, Voreinstellungswert, Primärschlüssel
(0, 'name', 'CHAR(20)', 0, None, 0)
(1, 'nummer', 'INTEGER', 0, None, 0)

Einträge pro Spalte:
name: 2
nummer: 2
[('Dolly', 1), ('Flauschi', 5)]
```

6.6.3 Löschen

Wie löscht man Dinge aus einer Datenbank?

- Um die ganze Datenbank zu löschen, schreibt man

  ```
  DROP DATABASE molecular_sheep
  ```

- Um eine Tabelle zu löschen, schreibt man

  ```
  DROP TABLE liegt_in
  ```

- Um einen Eintrag aus einer Tabelle zu löschen, schreibt man

  ```
  DELETE FROM schaf WHERE name = "Dolly" AND farbe = "weiss"
  ```

  Wir werden später noch sehen, dass dies hier nur ein Spezialfall ist. Man kann allgemein Anfragen benutzen zum Löschen.

Zusammenfassung

1. Eine *Datenbank-Schnittstelle* wie *`sqlite3`* dient dazu, eine Kommunikation mit Datenbanken zu realisieren.
2. Die Sprache *SQL* dient dazu, Datenbanken zu verwalten und Anfragen zu formulieren.

3. Datenbanken und Tabellen kann man mit den Befehlen `CREATE`, `INSERT`, `DROP` und `DELETE` verwalten.

6.7 Relationenalgebren

Wer ist Dollys Vater?

6.7.1 Ziele des Abschnitts

- Konzept der Relation-Algebra kennen
- Anfragen mittels Operationen formulieren können
- Joins verstehen und anwenden können

6.7.2 Vorüberlegungen

Wie sollte man Daten in einer Datenbank organisieren? Als eine riesige unsortierte Ansammlung von Daten (also eine Art Ursuppe)? Als komplexes, hierarchisches Gebilde? Als verzweigter Graph mit Schleifen? Als Menge von Objekten mit Verweisen, so wie in Python? Und welche Organisationsform ist eigentlich für das Suchen besonders geeignet?

Der Vater der *relationalen Datenbanken*, Edgar Codd, hat folgende scheinbar schlichte Antwort auf diese Fragen gegeben: Eine Datenbank besteht für ihn aus einer Menge von Tabellen mit einer flexiblen Anzahl an Zeilen und einer für jede Tabelle festen Anzahl an Spalten. Punkt. Nicht mehr und nicht weniger. Da »Tabellen« mathematisch nichts anderes sind als »Relationen«, spricht man von relationalen Datenbanken.

Die einzige Aufgabe eines relationalen Datenbanksystems ist es nun, solche Tabellen/Relationen so effizient wie irgend möglich zu speichern. Da Tabellen eher schlichte Geschöpfe sind (verglichen beispielsweise mit Bäumen), ist dies auch ausgesprochen gut möglich. Tatsächlich sind beispielsweise objektorientierte Datenbanken, bei denen statt Tabellen wie in Python Objekte mit Verweisattributen direkt gespeichert werden, noch nicht so effizient wie relationale Datenbanken.

Da es in relationalen Datenbanken wirklich *nur* Tabellen gibt, muss man alles auf Tabellen zurückführen, was nicht immer ganz logisch ist. Sucht man etwas in einer solchen Datenbank, wie zum Beispiel Dollys Vater, so ist die Idee, aus bestehenden Tabellen neue Tabellen auf geschickte Weise zu erzeugen (zur Erinnerung: Tabellen manipulieren können relationale Datenbanksysteme sehr gut). Am Ende hat man

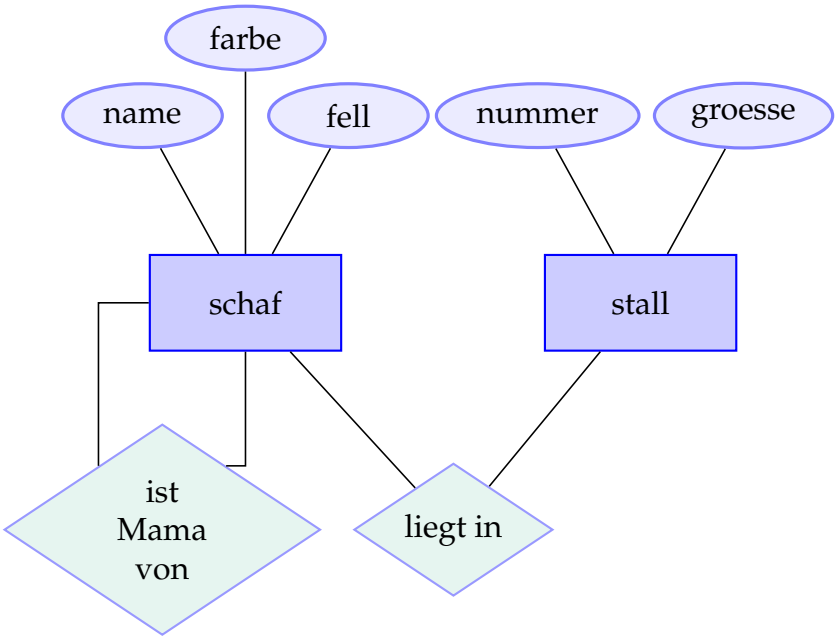
dann eine Tabelle mit nur einer Zeile, die die gewünschte Information enthält. Um relationale Datenbanken zu benutzen, muss man »in Tabellen denken lernen«.

In dem folgenden Text werden wir uns die Theorie hinter diesen Tabellen etwas genauer anschauen. Insbesondere soll es darum gehen, was Tabellen mathematisch sind und wie man aus bestehenden Tabellen neue Tabellen machen kann. Im nächsten Abschnitt werden wir diese Theorie dann direkt auf eine reale Datenbank anwenden.

6.7.3 Einführung

Alles ist eine Tabelle

Wiederholung: Ein Entity-Relationship-Diagramm.



Alles ist eine Tabelle

- Das E/R-Diagramm beschreibt die *Struktur* der Daten.
- Hinter jedem Entitätstyp (»schaf« und »stall«) steht eine Tabelle.
  - Die Spalten der Tabelle sind die Attribute dieses Typs.
  - Die Zeilen der Tabelle sind die Entitäten dieses Typs.
  - Genau eine der Spalten bildet den primären Schlüssel.

- Hinter jedem Relationshiptyp (»ist Mama von«, »liegt in«) steht ebenfalls eine Tabelle.
  - Die Spalten der Tabelle sind die Primärattribute der Entitäten, die in Beziehung gesetzt werden.
  - Die Zeilen der Tabelle repräsentieren Beziehungen zwischen den in der Zeile angegebenen Entitäten.

Relationen-Algebren

Die Relationen-Algebra.

Idee

- Die Tabellen für Entitäten und für Relationships werden zwar unterschiedlich interpretiert, *mathematisch* und *speichertechnisch* gibt es aber *keinen Unterschied*.
- Man konzentriert sich deshalb darauf, *Tabellen besonders effizient* zu verwalten.
- Statt *Tabellen* spricht man mathematisch von *Relationen*. In einer *relationalen Datenbank* werden (wenig überraschend) genau solche *Relationen* verwaltet.

Merke: Die Datenbank »weiß nichts« von dem E/R-Modell; aus ihrer Sicht wird eine Menge gleichberechtigter Tabellen verwaltet.

Die Relationen-Algebra.

Was sind Relationen?

Definition 4. Seien A_1 bis A_n Mengen. Eine *Relation* auf diesen Mengen ist eine Teilmenge von $A_1 \times \cdots \times A_n$.[6]

Bemerkungen:

- Das *Kreuzprodukt* $A_1 \times \cdots \times A_n$ enthält gerade alle Tupel, deren erste Komponente aus A_1 stammt, deren zweite Komponente aus A_2 und so weiter.
- Eine *Teilmenge* dieses Kreuzprodukts ist also eine Menge von Tupeln, wobei jedes jeweils ein Element aus A_1 in Beziehung setzt mit einem Element aus A_2 und gleichzeitig einem aus A_3 und so weiter.
- Ist beispielsweise A_1 die Menge aller Namen, A_2 die Menge aller Farben und A_3 die Menge aller Fellarten, so ist eine Relation auf A_1 bis A_3 gerade eine Tabelle für den Entitätstyp Schaf.

[6] $\times$ wird »Kreuz« gesprochen.

- Es gibt zwei Unterschiede zwischen Tabellen und Relationen: Eine *Relation kann jedes Tupel nur einmal enthalten und die Reihenfolge der Tupel ist egal*.

Die Relationen-Algebra.

Operationen auf Relationen.

- In einer Datenbank werden viele Relationen gespeichert, eine pro Entitätstyp und Relationshiptyp.
- Solche Relationen lassen sich nun *verknüpfen*, um neue Relationen zu bilden. Beispielsweise kann man aus der Relation aller Schafe durch Filterung die Relation aller schwarzen Schafe gewinnen.
- Die mathematische Klasse aller Relationen zusammen mit Operationen wie »Filterung« heißt mathematisch etwas hochtrabend *Relationen-Algebra*.

6.7.4 Operationen auf Relationen

Vereinigung und Schnitt

Die Operationen »Vereinigung« und »Schnitt«.

Definition 5. Seien $A, B \subseteq A_1 \times \cdots \times A_n$ Relationen auf denselben Mengen. Dann heißt $A \cup B$ die *Vereinigung* von A und B und $A \cap B$ der *Schnitt* von A und B.

Beispiel

- Sei A die Relation, die nur schwarze Schafe enthält (also alle Tupel in der Schafstabelle, bei denen die Farbkomponente »schwarz« lautet).
- Sei B die Relation, die nur lockige Schafe enthält.
- Dann ist $A \cap B$ die Relation, die nur lockige, schwarze Schafe enthält.
- Dann ist $A \cup B$ die Relation, die alle Schafe enthält, die lockig oder schwarz sind.

Selektion

Die Operation »Selektion«.

Definition 6. Seien $A \subseteq A_1 \times \cdots \times A_n$ eine Relation und P ein Prädikat auf Tupeln (also ein Test, der für jedes Tupel entweder wahr oder falsch ist). Dann ist die Selektion $\sigma_P(A) \subseteq A_1 \times \cdots \times A_n$ die Relation, die nur diejenigen Tupel aus A enthält, für die P gilt.[7]

Beispiel

- $\sigma_{\text{farbe}=\text{schwarz}}(\text{schaf})$ ist die Relation aller schwarzen Schafe.
- $\sigma_{\text{Größe}>50}(\text{stall})$ ist die Relation aller Ställe, die größer als 50 sind.
- Die folgenden Relationen sind gleich:
  1. $\sigma_{\text{farbe}=\text{schwarz}}(\text{schaf}) \cap \sigma_{\text{fell}=\text{lockig}}(\text{schaf})$
  2. $\sigma_{\text{farbe}=\text{schwarz} \wedge \text{fell}=\text{lockig}}(\text{schaf})$

Zur Übung

Sei A die Relation aller schwarzen Schafe, B die Relation aller lockigen Schafe und C die Relation aller kranken Schafe.

Versuchen Sie, mit Hilfe von Vereinigung, Schnitt und/oder Selektion folgende Mengen zu beschreiben:

1. Die Menge alle schwarzen Schafe, die lockig und krank sind.
2. Die Menge aller gesunden lockigen Schafe.

Projektion

Die Operationen »Projektion«.

Definition 7. Seien $A \subseteq A_1 \times \cdots \times A_n$ eine Relation und seien A_{i_1} bis A_{i_j} einige dieser Mengen. Dann entsteht die *Projektion* $\pi_{A_{i_1},\ldots,A_{i_j}}(A) \subseteq A_{i_1} \times \cdots \times A_{i_j}$, indem man »nur die Spalten« A_{i_1} bis A_{i_j} betrachtet.[8]

[7] σ ist ein griechischer Kleinbuchstabe und wird »sigma« gesprochen.
[8] π ist ein griechischer Kleinbuchstabe und wird »pi« gesprochen.

Beispiel

- schaf =

name	farbe	fell
Dolly	weiß	lockig
Flauschi	schwarz	wuschelig
Peter	schwarz	wuschelig
Flauschi	beige	glatt

- $\pi_{\text{farbe,fell}}(\text{schaf}) =$

farbe	fell
weiß	lockig
schwarz	wuschelig
beige	glatt

Kreuzprodukt

Die Operation »Kreuzprodukt«.

Definition 8. Seien $A \subseteq A_1 \times \cdots \times A_n$ und $B \subseteq B_1 \times \cdots \times B_m$ Relationen. Dann ist das *Kreuzprodukt* $A \times B \subseteq A_1 \times \cdots \times A_n \times B_1 \times \cdots \times B_m$ die Relation, die alle Kombinationen von Elementen aus A und Elementen aus B enthält.

- Beim Kreuzprodukt paart man alle Elemente der ersten Menge mit allen Elementen der zweiten Menge.
- Die Größe des Kreuzproduktes ist gerade das Produkt der Größen der Relationen A und B.

Beispiel für das Kreuzprodukt.

Gegeben seien folgende Relationen:

name	farbe	fell
Dolly	weiß	lockig
Flauschi	schwarz	wuschelig
Peter	schwarz	wuschelig

schafs-name	stall-nummer
Dolly	1
Peter	5

Ihr Kreuzprodukt lautet:

name	farbe	fell	schafs-name	stall-nummer
Dolly	weiß	lockig	Dolly	1
Dolly	weiß	lockig	Peter	5
Flauschi	schwarz	wuschelig	Dolly	1
Flauschi	schwarz	wuschelig	Peter	5
Peter	schwarz	wuschelig	Dolly	1
Peter	schwarz	wuschelig	Peter	5

Join

Eine typische Anfrage.

- Nehmen wir an, wir wollen wissen, in *welchen Ställen schwarze Schafe schlafen*.

  Problem: Diese Information liefert uns *weder* die Schafrelation *noch* die *liegt-in*-Relation.

- Wir müssen diese beiden Tabellen deshalb *vereinigen*.

  Problem: Das Kreuzprodukt setzt immer alles mit allem in Relation.

- Wir filtern deshalb nach solchen Paaren im Kreuzprodukt, bei denen der Schafname im ersten Teil des Tupels zum Schafname im zweiten Teil des Tupels passt.

  Problem: Dies liefert große Tupel, wir wollen aber nur die Stall-Nummer wissen.

- Dann projizieren wir auf die Stall-Nummer.

$$\pi_{\text{stall-nummer}}\big(\sigma_{\text{name}=\text{schafs-name}}\big(\sigma_{\text{farbe}=\text{schwarz}}(\text{schaf}) \times \text{liegt in}\big)\big).$$

Die Anfrage Schritt für Schritt.

Die Ausgangstabellen.

schaf =

name	farbe	fell
Dolly	weiß	lockig
Flauschi	schwarz	wuschelig
Peter	schwarz	wuschelig

liegt in =

schafs-name	stall-nummer
Dolly	1
Peter	5

Die Anfrage Schritt für Schritt.

Die gefilterte Schaftabelle.

$\sigma_{\text{farbe=schwarz}}(\text{schaf}) =$

name	farbe	fell
Flauschi	schwarz	wuschelig
Peter	schwarz	wuschelig

liegt in $=$

schafs-name	stall-nummer
Dolly	1
Peter	5

Die Anfrage Schritt für Schritt.

Das Kreuzprodukt.

$\sigma_{\text{farbe=schwarz}}(\text{schaf}) \times \text{liegt in} =$

name	farbe	fell	schafs-name	stall-nummer
Flauschi	schwarz	wuschelig	Dolly	1
Flauschi	schwarz	wuschelig	Peter	5
Peter	schwarz	wuschelig	Dolly	1
Peter	schwarz	wuschelig	Peter	5

Die Anfrage Schritt für Schritt.

Die Selektion.

$\sigma_{\text{name=schafs-name}}(\sigma_{\text{farbe=schwarz}}(\text{schaf}) \times \text{liegt in}) =$

name	farbe	fell	schafs-name	stall-nummer
Peter	schwarz	wuschelig	Peter	5

Die Anfrage Schritt für Schritt.

Die Projektion.

$\pi_{\text{stall}}\left(\sigma_{\text{name=schafs-name}}(\sigma_{\text{farbe=schwarz}}(\text{schaf}) \times \text{liegt in})\right) =$

stall-nummer
5

Zur Übung

Beschreiben Sie mit Hilfe der bisherigen Operationen folgende Relation: Sie soll alle Paare enthalten von einem Schafsnamen zusammen mit der Größe des Stalls, in dem das Schaf liegt.

(Wem das zu einfach ist: Alle Paare von Schafsnamen zusammen mit der Größe des Stalls, in dem die Mutter des Schafs liegt.)

Die Operation »Join«.

- Es kommt sehr oft vor, dass man zwei Relationen derart »zusammenfügen« möchte, dass nur diejenigen Tupel des Kreuzproduktes betrachtet werden sollen, bei denen zwei bestimmte Attribute gleich sind.
- Man nennt dies einen *Join* der Relationen *anhand* zweier Attribute.

Definition 9. Sei $A \subseteq A_1 \times \cdots \times A_n$ und $B \subseteq B_1 \times \cdots \times B_m$.

Sei $X = A_i = B_j$ für geeignete i und j.

Dann ist der *Join* $\theta_X(A, B)$[9] die Menge

$\sigma_{\text{alte } i\text{-Komponente}=\text{alte } j\text{-Komponente}}(A \times B)$.

Beispiel

Die Anfrage von vorhin lässt sich nun kürzer schreiben:

$$\pi_{\text{stall}}(\theta_{\text{name}}(\sigma_{\text{farbe}=\text{schwarz}}(\text{schaf}), \text{liegt in})).$$

Zusammenfassung des Abschnitts

1. Relationale Datenbanken speichern *Relationen*.
2. Die *Relationenalgebra* ist die mathematische Klasse aller Relationen zusammen mit Operationen wie *Kreuzprodukt*, *Selektion* und *Projektion*.

[9] θ ist ein griechischer Kleinbuchstabe und wird »theta« gesprochen. Es gibt diesen Buchstaben auch in einer anderen Schreibweise: ϑ

6.8 SQL – Einfache Anfragen

Die strukturierte Anfragesprache

6.8.1 Ziele des Abschnitts

- SQL-Anfragen an eine Datenbank formulieren können
- Einfache SQL-JOINs benutzen können
- SQL-Funktionen benutzen können

6.8.2 Vorüberlegungen

In den vorherigen beiden Abschnitten haben wir schon ein Bild davon erhalten, wie man mit relationalen Datenbanken Daten modellieren kann und mit – mathematisch ganz vornehm – algebraischen Operatoren Dinge wiederfindet. Aber wie bringe ich nun der Maschine bei, dass ich gerne den JOIN $\theta_{A,B}(A)$ hätte? Ein Theta sucht man auf den meisten Tastaturen vergeblich. Kurz: Wie sage ich es meinem Computer?

In dieser ersten Vorlesung zu SQL soll es darum gehen, wie man Anfragen an eine Datenbank syntaktisch korrekt formuliert. Dazu wird der SELECT-Befehl verwendet, der ein wahres Wunderkind ist. Wir haben ihn schon in der eher schlichten Version `SELECT * FROM foo;` kennengelernt, – wahre Meister können aber mit einem mehrseitigen SELECT-Befehl Daten aus einer Datenbank herauskitzeln, von denen Normalsterbliche gar nicht vermuten würden, dass sie überhaupt in der Datenbank drin sind. Beginnen werden wir mit einfachen Anfragen. Am Ende stehen dann schon recht komplexe SELECT-Befehle, mit denen Sie dann auch wenigstens ein bisschen mitkitzeln können.

Wenn Sie ein wenig mit dem SELECT-Befehl geübt haben, werden Sie bald feststellen, dass es nicht sonderlich viel Spaß macht, diesen Befehl immer wieder in einer Shell einzugeben. Als Mensch möchte man sich nämlich eigentlich gar nicht auf die »Niederungen« der Datenbank-Ebene begeben – das sollen doch bitte Informatiksysteme machen. (Schließlich wollen Sie im Internet auch nicht ständig kryptische IP-Adressen eingeben – auch das sollen die Maschinen selber verwalten.) Aus diesem Grund wird im nächsten Kapitel gezeigt, wie man statt mit einer Shell von Python aus mit einer Datenbank kommuniziert. Das ändert aber nichts daran, dass man auch von Python aus mittels SELECT-Befehlen mit der Datenbank »redet«. Es lohnt sich also so oder so, diesen Befehl gut zu beherrschen.

6.8.3 Einfache Anfragen

Grundaufbau

Formulierung einer Anfrage in SQL.

- Anfragen haben generell folgende Form:

```
SELECT A_1,...,A_n
FROM R_1,...,R_m
WHERE P
```

  Hierbei sind
  - A_1 bis A_n Attribute,
  - R_1 bis R_m Relationen und
  - P ein Prädikat.

- Der Effekt dieser Anfrage ist es, die folgende Relation auszugeben:

$$\pi_{A_1,\ldots,A_n}(\sigma_P(R_1 \times \cdots \times R_m))$$

Beispiele einfacher Anfragen.

- Welche schwarzen Schafe gibt es?

```
mysql> SELECT name
         FROM schaf
         WHERE farbe = "schwarz";
+----------+
| name     |
+----------+
| Flauschi |
| Peter    |
+----------+
```

- Welche schwarzen Schafe gibt es und welches Fell haben sie?

```
mysql> SELECT name, fell
         FROM schaf
         WHERE farbe = "schwarz";
+----------+-----------+
| name     | fell      |
+----------+-----------+
```

```
| Flauschi | wuschelig |
| Peter    | wuschelig |
+----------+-----------+
```

Beispiele einfacher Anfragen.

- Welche Ställe haben eine Größe von mindestens 50?

```
mysql> SELECT nummer
         FROM stall
         WHERE groesse >= 50;
+--------+
| nummer |
+--------+
|      1 |
+--------+
```

- Welche Stall-Entitäten haben eine Größe von mindestens 50?

```
mysql> SELECT *
         FROM stall
         WHERE groesse >= 50;
+--------+---------+
| nummer | groesse |
+--------+---------+
|      1 |      50 |
+--------+---------+
```

Verbesserte Ausgaben

Beschränkung der Zeilenanzahl.

- Möchte man wissen, was beispielsweise alles in der Tabelle `stall` steht, so geht dies sehr leicht mit folgender Anfrage:

```
mysql> SELECT * FROM stall;
+--------+---------+
| nummer | groesse |
+--------+---------+
|      1 |      50 |
|      5 |      25 |
+--------+---------+
```

- Bei großen Tabellen dauert dies aber viel zu lange.

- Der Trick ist nun, hinter einen `SELECT`-Befehl eine *Limitierung* zu schreiben:

```
mysql> SELECT * FROM stall LIMIT 5;
```

  Dies sorgt dafür, dass nur die ersten 5 Zeilen ausgegeben werden.

Beschränkung der Strings.

- Man kann statt einem gesamten String wie einem Namen oder einer DNA-Sequenz auch nur einen Teil ausgeben lassen.
- Dazu schreibt man statt des Attributes einfach `substring(`n`,`s`,`l`)`, wobei n der Name des Attributs ist, s das Anfangszeichen im String (mit 1 beginnend) und l die Länge des gewünschten Teilstrings ist.

  In `sqlite` wird statt `substring` `substr` verwendet.

```
mysql> SELECT substring(name,1,3), farbe, fell FROM stall;
+----------------------+---------+-----------+
| substring(name,1,3) | farbe   | fell      |
+----------------------+---------+-----------+
| Dol                  | weiss   | lockig    |
| Max                  | schwarz | wuschelig |
| Pet                  | schwarz | wuschelig |
+----------------------+---------+-----------+
mysql> SELECT substring(name,2,3), farbe, fell FROM stall;
+----------------------+---------+-----------+
| substring(name,2,3) | farbe   | fell      |
+----------------------+---------+-----------+
| oll                  | weiss   | lockig    |
| ax                   | schwarz | wuschelig |
| ete                  | schwarz | wuschelig |
+----------------------+---------+-----------+
```

Sortierung der Ausgabe.

- Oft möchte man die Ausgabe *sortieren*. (Normalerweise werden die Daten in einer nicht vorhersagbaren Reihenfolge ausgegeben.)
- Dazu kann man nach einem `SELECT`-Befehl ein *Sortierungsattribut* angeben:

```
mysql> SELECT * FROM stall ORDER BY groesse;
+--------+---------+
| nummer | groesse |
+--------+---------+
|      5 |      25 |
```

```
|       1 |       50 |
+---------+----------+
```

- *Nach* dem Attribut kann man noch `DESC` angeben, um die Reihenfolge umzudrehen.

```
mysql> SELECT * FROM stall ORDER BY nummer desc;
+---------+----------+
| nummer  | groesse  |
+---------+----------+
|       5 |       25 |
|       1 |       50 |
+---------+----------+
```

Zur Übung

Geben Sie den SQL-Befehl an, mit dessen Hilfe Sie die fünf größten Ställe angezeigt bekommen.

JOINs

Zur Erinnerung

- Will man Daten aus zwei Relationen *zusammenfügen*, so benötigt man ein *Kreuzprodukt*, gefolgt von einer *Filterung*.
- Bei der Filterung streicht man alle Zeilen, in der zwei bestimmte gleichbenannte Attribute nicht gleich sind.
- Eine solche Verbindung nennt man auch einen *JOIN* anhand dieses Attributs.

Beispiele einer Kreuzproduktes.

Das Kreuzprodukt von Schafen und der liegt-in-Relation.

```
mysql> SELECT * FROM schaf, liegt_in;
+----------+---------+-----------+----------+--------+
| name     | farbe   | fell      | name     | nummer |
+----------+---------+-----------+----------+--------+
| Dolly    | weiss   | lockig    | Dolly    |      1 |
| Dolly    | weiss   | lockig    | Flauschi |      5 |
| Flauschi | schwarz | wuschelig | Dolly    |      1 |
```

```
| Flauschi | schwarz | wuschelig | Flauschi |       5 |
| Peter    | schwarz | wuschelig | Dolly    |       1 |
| Peter    | schwarz | wuschelig | Flauschi |       5 |
+----------+---------+-----------+----------+---------+
```

Die Ställe der schwarzen Schafe.

Die Anfrage $\pi_{\text{Stall-Nummer}}(\sigma_{\text{Name=Schaf-Name}}(\sigma_{\text{Farbe=schwarz}}(\text{Schaf}) \times \text{Liegt-in}))$ in SQL:

```
mysql> SELECT nummer
         FROM schaf, liegt_in
         WHERE
           farbe       = "schwarz" AND
           schaf.name  = liegt_in.name;
+--------+
| nummer |
+--------+
|      5 |
+--------+
```

Eine komplexe Anfrage.

Welche Größe haben die Ställe der Schafe?

```
mysql> SELECT schaf.name, groesse
         FROM schaf, liegt_in, stall
         WHERE
           schaf.name    = liegt_in.name AND
           stall.nummer  = liegt_in.nummer;
+----------+---------+
| name     | groesse |
+----------+---------+
| Dolly    |      50 |
| Flauschi |      25 |
+----------+---------+
```

Zur Übung

Geben Sie den SQL-Befehl an, mit dessen Hilfe Sie alle Schafsnamen zusammen mit der Größe des Stalls bekommen, in dem das Schaf schläft.

6.8.4 Anfragen für Fortgeschrittene

Prädikate

Wie lautet die Prädikatsyntax?

- Prädikate folgen in SQL dem Schlüsselwort `WHERE`.
- Sie sind (ähnlich wie bei Python) boolesche Ausdrücke.
- Die Variablen in einem Prädikat sind gerade die Attribute einer Zeile. (Haben zwei Relationen `R_1` und `R_2` das gleiche Attribut `A`, so muss man `R_1.A` oder `R_2.A` schreiben.)
- Die Hauptunterschiede zu Python-Ausdrücken:
  1. Gleichheit wird mit = statt mit == geprüft.
  2. Strings lassen sich mittels <, <= usw. lexikographisch vergleichen.
  3. Statt `!`, `||` und `&&` kann man (und tut man) auch `NOT`, `OR` und `AND` schreiben.
  4. Mit `LIKE` kann man komplexe Textvergleiche anstellen (dazu gleich mehr).

Beispiele für Prädikate.

```
mysql> SELECT ... WHERE
   farbe = "schwarz" AND
   !(nummer < 5);
...
mysql> SELECT ... WHERE
  liegt_in.nummer = stall.nummer AND
  !(farbe LIKE "wei%");
...
```

Die drei Arten von Stringvergleichen in SQL.

In SQL gibt es drei Methoden, Strings zu vergleichen:

1. Ein *direkter lexikographischer Vergleich* mittels = oder >=.
2. Eine Art *Ähnlichkeitsvergleich* mittels `a LIKE b`. Dieser Vergleich ist wahr, falls:
   a) Die Strings `a` und `b` gleich sind, wobei aber

b) für jedes Vorkommen des Unterstrichs in `b` an der entsprechenden Stelle in `a` ein beliebiges Zeichen stehen darf und

c) für jedes Vorkommen eines Prozentzeichens in `b` an der entsprechenden Stelle in `a` eine beliebige (auch leere) Zeichenfolge stehen darf.

Ist beispielsweise `b` der String `H_llo`, so könnte `a` sowohl `Hallo` als auch `Hello` sein, nicht aber `Haallo`. Ist `b` der String `H%llo`, so könnte `a` nun `Haallo` sein, nicht aber `Halo`.

Funktionen

Neu: Zeilenweise Anwendung statt Projektionen.

Definition 10. Seien $A \subseteq A_1 \times \cdots \times A_n$ eine Relation und seien $f_i \colon A_1 \times \cdots \times A_n \to B_i$ Funktionen. Dann ist die *zeilenweise Anwendung* dieser Funktionen wie folgt definiert:[10]

$$\begin{aligned}\zeta_{f_1,\ldots,f_m}(A) &= \{(f_1(a),\ldots,f_m(a)) \mid a \in A\} \\ &\subseteq B_1 \times \cdots \times B_m.\end{aligned}$$

Beispiel

$\zeta_{\text{erstes Zeichen von(Name),Länge(Farbe)+Länge(Fell)}}(\text{Schaf}) =$

B_1	B_2
D	10
F	16
P	16

Die zeilenweise Anwendung in SQL.

- Bei einem SELECT-Befehl kann man statt der Attribute auch beliebige Funktionen angeben.
- Tatsächlich sind die normalen Projektionen auch nur Spezialfälle der zeilenweise Anwendung einer (sehr einfachen) Funktion.
- Welche Funktionen möglich sind, muss man in der Dokumentation nachschlagen. Einige Beispiele: `abs` für den Absolutbetrag ein Zahl, `sin` für den Sinus einer Zahl, `length` für die Länge eines Strings, `substring` für einen Teilstring, `concat` für die Verkettung mehrerer Strings.

```
1 mysql> SELECT
2   substring(name,1,1), length(farbe)+length(fell)
3   FROM schaf;
```

[10] ζ ist ein griechischer Kleinbuchstabe und wird »zeta« gesprochen.

Zusammenfassung des Abschnitts

1. Die grundlegende Syntax einer Anfrage lautet

   ```
   SELECT A_1,...,A_n FROM R_1,...,R_m WHERE P.
   ```

2. Man kann statt Projektionen auch beliebige Funktionen zeilenweise anwenden.
3. Man kann Aggregate benutzen, um Spalten zu aggregieren.

Wird noch ergänzt

Anhang A

Arbeits- und Informationsblätter, Lernzielkontrollen

A.1 Vorhaben Q1-1

A.1.1 Eingangsbefragung – Selbsteinschätzung

Name: **Datum:**

Aussage	trifft gar nicht zu → trifft voll zu
Erhalte ich eine informatikbezogene Problemsituation, kann ich dazu ein informatisches Modell entwickeln.	☐ ☐ ☐ ☐
Ich kann einen konkreten Ablauf als Sequenzdiagramm darstellen.	☐ ☐ ☐ ☐
Objektorientierte Modellierung führt vom Problem zu Objekten zu Klassen und zum Programm.	☐ ☐ ☐ ☐
Der Unterschied zwischen Anfrage und Auftrag ist mir klar.	☐ ☐ ☐ ☐
Die Umsetzung eines Struktogramms in eine Methode und umgekehrt stellt für mich kein Problem dar.	☐ ☐ ☐ ☐
Ich kann das Konzept der Rekursion an mehreren Beispielen erklären.	☐ ☐ ☐ ☐
Ich kann das Konzept der von-Neumann-Maschine erklären.	☐ ☐ ☐ ☐

Die Turing-Maschine ist das Modell für die Arbeit einer Mathematikerin.	☐ ☐ ☐ ☐
Ich kann einen Automaten implementieren.	☐ ☐ ☐ ☐
Ich kann Wörter einer Sprache mit Hilfe der Syntax überprüfen.	☐ ☐ ☐ ☐
Bei der Datenbankmodellierung werden Daten von den Operationen getrennt.	☐ ☐ ☐ ☐
Der Informatikunterricht hat mir bisher viel Spaß gemacht.	☐ ☐ ☐ ☐
Ich habe bisher im Informatikunterricht viel gelernt.	☐ ☐ ☐ ☐
Ab und zu überforderte mich der Informatikunterricht.	☐ ☐ ☐ ☐
Ich habe mich im Informatikunterricht manchmal unterfordert gefühlt.	☐ ☐ ☐ ☐

A.1.2 Eingangskompetenzen – Beispielmodellierung

Name: **Datum:**

WM-Finale

Lesen Sie die folgende Situationsbeschreibung[1] und bearbeiten Sie anschließend die darunter stehenden Aufgaben.

> Der Spieler Arno – er spielt für Deutschland im zentralen Mittelfeld und hat die Rückennummer 10 – ist im Ballbesitz und passt den Spielball von seiner jetzigen Position – etwa 30m vor dem gegnerischen Tor (das Tor vor der Südkurve) – zu seinem Mitspieler Bert, der als Stürmer die Rückennummer 11 trägt und sich etwa beim Elfmeterpunkt befindet. Bert nimmt den Spielball an, doch Christiano, der als Stürmer mit der Rückennummer 9 eigentlich für Italien das gegnerische Tor vor der Nordkurve treffen sollte, grätscht Bert um, so dass der nun die Gelegenheit hat, einen Strafstoß auszuführen.

[1] Zum Hintergrund: Wir befinden uns im Finale der Fußball-Weltmeisterschaft 2010 in Südafrika. Die Szene ereignet sich in der 89. Minute beim Stand von 4:4 zwischen Deutschland und Italien

Bert legt den Spielball elf Meter vor das Tor und schießt ihn auf das Tor. Fragend blickt Bert auf das intelligente Tor, das nun allen anzeigt, dass ein Tor gefallen ist. Insbesondere Arno und Bert fangen daraufhin an zu jubeln.

Aufgabe

1. Identifizieren Sie mit Hilfe des Verfahrens von Abbott alle in der obigen Situationsbeschreibung vorkommenden Objekte mitsamt ihrer Attribute und Dienste.
2. Erstellen Sie ein Objektdiagramm für die Situation vor dem Pass von Arno zu Bert. Finden Sie dazu eine Möglichkeit, darzustellen, wo der Ball sich befindet. Manche Zusammenhänge müssen Sie ggf. aus dem Kontext herleiten.
3. Erstellen Sie auch ein Objektdiagramm für den Augenblick am Ende der Situationsbeschreibung.
4. Erstellen Sie ein Sequenzdiagramm, das den Ablauf der Situation visualisiert. Möglicherweise müssen Sie dazu Ihr Objektdiagramm aufgrund fehlender Dienste oder Attribute ergänzen bzw. ändern.
5. Erstellen Sie ein Klassendiagramm zur Modellierung der obigen Situationsbeschreibung inklusive der Beziehungen zwischen den Klassen. Aktualisieren Sie ggf. Ihr Objektdiagramm bzw. das Sequenzdiagramm.
6. Überführen Sie das Sequenzdiagramm in ein Programm (ohne die Klassen zu programmieren). Damit soll die oben dargestellte Situation durchgespielt werden und nach jedem Schritt ausgegeben werden, was gerade passiert ist. Aktualisieren Sie dabei ggf. Ihre vorher erstellten Diagramme.

A.1.3 Lineare Datenstruktur – Array – Lernzielkontrolle 1

1. Aufgabe (7 Punkte) Was ist ein Array?

a) Beschreiben Sie stichwortartig mit eigenen Worten, was ein Array ist.

b) Geben Sie ein Beispiel für ein Array an.

c) Können die folgenden Daten zu einem Array zusammengefasst werden?

- `"Waldemar", 12.34, True, "Werner", 567, 8.654`

Hinweis: Die Kommata trennen die einzelnen Elemente, gehören also nicht zu den Datenelementen.

Begründen Sie kurz Ihre Entscheidung.

2. Aufgabe (3 Punkte) Array – Festlegung und Zugriff

a) Es soll ein Array für einzelne Zeichen erstellt werden, in dem zwanzig Zeichen Platz finden. Geben Sie eine Möglichkeit an, diese Arbeit (mit Python) zu erledigen. Das Array soll mit `meineZeichenkette` benannt werden.

b) Geben Sie an, wie auf das **siebte** Element von `meineZeichenkette` zugegriffen werden kann.

3. Aufgabe (10 Punkte) Array – Änderung und Schreibtischtest

a) Die folgende Festlegung ist in einem Python-Programm vorgegeben:

```
zahlenliste=[73, 8, 19, 108, 25]
```

Setzen Sie zunächst für das erste Element dieses Arrays den Tag Ihrer Geburt, für das dritte Element die Nummer des Monats Ihrer Geburt und für das fünfte Element das Jahr Ihrer Geburt.

Schreiben Sie diese drei Zeilen als Python-Quellcode auf.

b) Prüfen Sie den folgenden Programmabschnitt, in dem Sie einen Schreibtischtest für das von Ihnen in der Teilaufgabe 3a **geänderte** Array durchführen und die Wertetabelle aufschreiben.

```
1 ...
2 was=zahlenliste[0]
3 index=1
4 while index < zahlenliste.__len__():
5   was=was+zahlenliste[index]
6   index=index+1
```

A.1.4 Lineare Datenstruktur – Array – Lernzielkontrolle 2

Lernzielkontrolle 2 – Musterlösung

1. Aufgabe (10 Punkte) Array – Änderung und Schreibtischtest

a) Die folgende Festlegung ist in einem Python-Programm vorgegeben:

```
1 zeichenliste=["R", "K", "M", "Z", "B"]
```

Setzen Sie zunächst für das erste Element dieses Arrays den Anfangsbuchstaben des Vornamens Ihrer Mutter, für das dritte Element den Anfangsbuchstaben des Vornamens Ihres Vaters und für das fünfte Element den Anfangsbuchstaben Ihres Nachnamens.

Schreiben Sie dazu nötigen Änderungen im Python-Quellcode auf (drei Zeilen).

Lösung **3 Punkte**

```
zeichenliste[0]="J"
zeichenliste[2]="E"
zeichenliste[4]="H"
```

b) Prüfen Sie den folgenden Programmabschnitt, in dem Sie einen **Schreibtischtest** für das von Ihnen in der Teilaufgabe 1a **geänderte** Array durchführen und die vollständige Wertetabelle aufschreiben.

```
zn=zeichenliste[0]
index=1
while index < zeichenliste.__len__():
  if zeichenliste[index] < zn:
    zn = zeichenliste[index]
  index=index+1
print(zn)
```

Lösung **7 Punkte**

Bezeichner	Wert(e)				
zeichenliste	[0] ~~R~~ J	[1] K	[2] ~~M~~ E	[3] Z	[4] ~~B~~ H
zn	~~J~~ Z				
index	~~1~~ ~~2~~ ~~3~~ ~~4~~ 5				

2. Aufgabe (15 Punkte) Häufigkeitsbestimmung

a) Beschreiben Sie stichwortartig mit eigenen Worten, wie Sie bei einem Array aus Zahlen herausfinden, wie oft eine bestimmte Zahl auftritt (Vorgehen).

Lösung **5 Punkte**

Zu Beginn muss man sich merken, dass `gesucht` bisher nicht aufgetreten ist: `gesuchtAnzahl` wird auf Null gesetzt.

In der Liste wird nach und nach jedes Element mit `gesucht` verglichen – stimmen die beiden überein, so wird `gesuchtAnzahl` um eins erhöht.

b) Geben Sie ein **Programmstück** an, das die Häufigkeit einer Zahl bei einem beliebigen Array aus Zahlen ermittelt.

Lösung **6 Punkte**

```
gesuchtAnzahl= 0

index=0
while index < zahlen.__len__():
  if zahlen[index] == gesucht:
    gesuchtAnzahl= gesuchtAnzahl+1
  index=index+1
```

c) Führen Sie einen **Schreibtischtest** für Ihren in der Teilaufgabe 2b angegebenen Algorithmus mit den als Zahlen interpretierten Ziffern Ihres Geburtsdatums als Elemente des Arrays durch.

Beispiel: Geburtsdatum ist 23.5.1998 → `zahlen= [2, 3, 5, 1, 9, 9, 8]`

Ist nun die gesuchte Zahl durch `gesucht=9` gegeben, müsste als Ergebnis `2` ermittelt werden.

Lösung **4 Punkte**

Bezeichner	Wert(e)						
zahlen	[0]	[1]	[2]	[3]	[4]	[6]	[7]
	2	3	5	1	9	9	8
gesucht	9						
gesuchtAnzahl	0	1	2				
index	0	1	2	3	4	5	6	7

A.2 Vorhaben Q1-2

A.2.1 Suchen und Sortieren

Übung zu Quellcode

1. Aufgabe Beschreiben Sie Zeile für Zeile, was hier geschieht.

Die Klassen `Spielkarte` und `KartenListe` sind folgendermaßen festgelegt:

```
class Spielkarte:
  def __init__(self):
    self.aufschrift= ""
    self.naechste= None
class KartenListe:
  def __init__(self):
    self.start= None    # Objekt der Klasse Spielkarte
```

```
if __name__ == "__main__":
  folge= KartenListe()
  a= Spielkarte()
  b= Spielkarte()
  a.aufschrift= "Karo␣As"
  b.aufschrift= "Kreuz␣10"
```

2. Aufgabe Nun wird der folgende Quellcode ausgeführt:

```
  # ...
  folge.start= a
  a.naechste= b
```

Stellen Sie das Ergebnis nach der Durchführung durch Angabe des Objektdiagramms grafisch dar.

A.3 Vorhaben Q1-3 und Q1-4

A.3.1 Bäume und Graphen

Lernzielkontrolle 5 mit Lösungshinweisen

1. Aufgabe (4 Punkte) Begriffe und Visualisierung

a) Erläutern Sie wesentliche Begriffe, die in der Informatik im Zusammenhang mit der Darstellung (Zeichnung) von Bäume und Graphen verwendet werden.

b) Zuordnung: Verdeutlichen Sie an einem Beispiel die in 1a erläuterten Begriffe. Dazu können Sie auch den unten (in Aufgabe 3a) dargestellten Baum wählen und dort die Begriffe eintragen, die Sie erläutern (dann bitte mit einem Verweis).

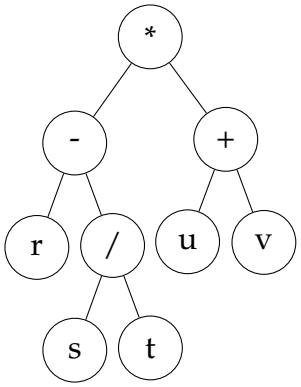
2. Aufgabe (3 Punkte) Schreibweisen mathematischer Ausdrücke

a) Um mathematische Ausdrücke darzustellen, haben Sie in der Mathematik eine Schreibweise kennengelernt – diese Schreibweise hat in der Informatik eine besondere Bezeichnung. Geben Sie diese Bezeichnung an und erläutern Sie sie kurz.

b) Zwei weitere Bezeichnungen haben Sie im Zusammenhang mit Bäumen kennengelernt. Geben Sie diese Bezeichnungen ebenfalls an (mit Erläuterung)

3. Aufgabe (8 Punkte) Umwandlung Darstellungsformen

a) Stellen Sie den folgenden Baum in den drei in der Aufgabe 2 angegebenen Formen dar:



b) Stellen Sie den folgenden, in der üblichen Schreibweise der Mathematik vorliegenden Term in anderen Formen dar:

$$a + b * (c - d) - e$$

i. als Baum

ii. in den anderen beiden von Ihnen in der Aufgabe 2b) erläuterten Darstellungen.

A.4 Vorhaben Q1-5 und Q1-6 Von der OOM über das MVC-Muster zum E/R-Modell

A.4.1 Situationsbeschreibung – Modellierungsaufgabe

Viele Anwendungen im Internet benötigen die Anbindung an eine Datenbank, weil große Datenmengen verwaltet werden müssen, auf die zahlreiche Benutzer zugreifen. Am Beispiel der imaginären Website »Music4You«, von der man Musikstücke gegen Bezahlung herunterladen kann, werden wir betrachten, welche Schritte bei der Entwicklung einer Datenbank durchlaufen werden müssen.

Bevor man eine Datenbank implementieren kann, muss man zunächst die Analyse- und dann die Modellierungsphase durchführen. Dabei können Sie auf Verfahren zurückgreifen, die Sie bereits kennen. In der Analysephase wird das Verfahren nach Abbott verwendet, um die Problembeschreibung zu untersuchen. Als Ergebnis erhält man die Objekte und die zugehörigen Attribute und Methoden.

Problembeschreibung:

Am 27.10.2009 besucht Alexander Beier die Website »Music4You« und meldet sich mit seiner E-Mail-Adresse »alexb@bergkamen.de« und seinem Passwort »VM!ditfm« an. Um 16.34 Uhr lädt er »Stadt« von Cassandra Steen und Adel Tawil für 0,99 Euro herunter. Während er darauf wartet, dass der Download abgeschlossen ist, liest er sich die Kurzinformation zu Adel Tawil durch. Alexander erfährt Folgendes: »Adel Tawil heißt mit vollem Namen Adel Salah Mahmoud Eid El-Tawil und wurde am 15. August 1978 in Berlin geboren. Adels Karriere begann Ende der 1990er-Jahre, als er Mitglied der Boygroup The Boyz war. Zusammen mit Annette Humpe bildet er seit 2004 das Duo Ich+Ich.« Auch für Cassandra Steen gibt es eine Kurzinformation, aber da der erste Download beendet ist, sucht Alexander weiter. Er wählt um 16.40 Uhr »Gives You Hell« von The All-American Rejects für 1,29 Euro aus.

Elif Ceylan ruft an diesem Tag die Website das erste Mal auf. Bevor sie das Angebot nutzen kann, muss sie zunächt ihre E-Mail-Adresse »celif@bergkamen.de«, ein Passwort, ihre Kontonummer (1234567), ihre Bankleitzahl (41051845) und den Namen ihrer Bank (Sparkasse Bergkamen-Bönen) angeben. Ebenso wie Alexander mag sie das Lied »Stadt« und lädt es um 18.17 Uhr herunter. Als Geschenk für ihren Freund lädt sie um 18.20 Uhr »21 Guns« von Green Day herunter. Dieses Lied erschien im Jahr 2009, hat eine Dauer von 4:34 min und als Genre ist Alternative angegeben. Um 18.23

Uhr lädt sie das ebenfalls im Jahr 2009 veröffentlichte Stück »Heavy Cross« von Gossip herunter, das eine Länge von 3:14 min hat und zum Genre Rock gehört. Beide Musikstücke kosten 1,19 Euro.

Erweiterung

Am Montag, dem 2.11.2009, erzeugt die Buchhaltung der Website »Music4You« um 12.00 Uhr die Rechnung der 44. Kalenderwoche für Alexander Beier.

Die Rechnung nimmt die Downloads von »Stadt« und »Gives You Hell« auf, die Alexander am 27.10.2009 um 16.34 Uhr bzw. um 16.40 Uhr heruntergeladen hat. Weitere Lieder hat Alexander in der 44. KW nicht gekauft.

A.4.2 Prüfung und Ergänzung der Modellierung durch Anwendungsfälle

Anwendungsfall »Anmeldevorgang«

Alexander Beier möchte, dass ihm die Website »Music4You« die Berechtigung erteilt, Musikstücke herunterzuladen. Damit »Music4You« prüfen kann, ob Alexander Beier berechtigt ist, die Website zu nutzen, erfragt »Music4You« Alexanders E-Mail-Adresse »alexb@bergkamen.de« und Passwort »VM!ditfm«. Dazu wird das Anmeldeformular verwendet. Im Anschluss erteilt sie Alexander die Berechtigung die Website zu nutzen.

Abbildung A.1: Anmeldeformular

Abbildung A.2: Erfolgreiche Anmeldung

Download eines Musikstück

Elif Ceylan wählt »Heavy Cross« aus und gibt damit der Website »Music4You« den Auftrag, den Download zu starten. »Music4You« veranlasst, dass der dritte Download von Elif protokolliert, dass am 27.10.2009 um 18.23 Uhr »Heavy Cross« heruntergeladen wird. Die Website erfragt von »Heavy Cross« den Pfad zur zugehörigen mp3-Datei und erhält als Antwort »C:\Download\Gossip-Heavy_Cross.mp3«. Diese Information gibt sie an Elif weiter.

Suche nach einem Musikstück

Alexander Beier sucht nach einem Musikstück mit dem Titel »Gives You Hell«. Dazu benutzt er das Suchformular der Website »Music4You«.

Zunächst erfragt »Music4You« den Titel von »Stadt« und vergleicht ihn mit dem gesuchten Titel. Dann erfragt die Website den Titel von »Gives You Hell« und vergleicht ihn mit dem gesuchten Titel. Da sie eine Übereinstimmung feststellt, zeigt sie »Gives You Hell« an. Dazu ermittelt »Music4You« den Titel, die Interpreten und den Preis von »Gives You Hell«.

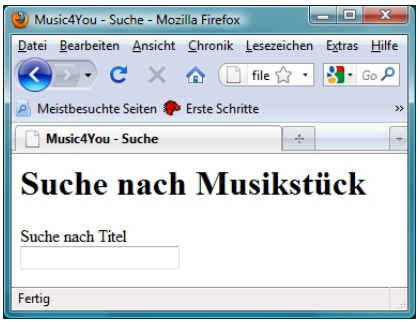
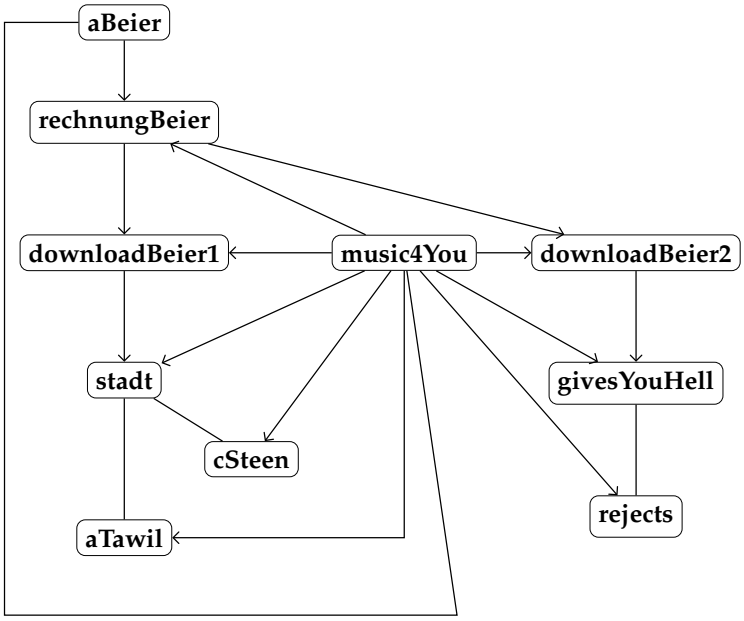


Abbildung A.3: Suchformular

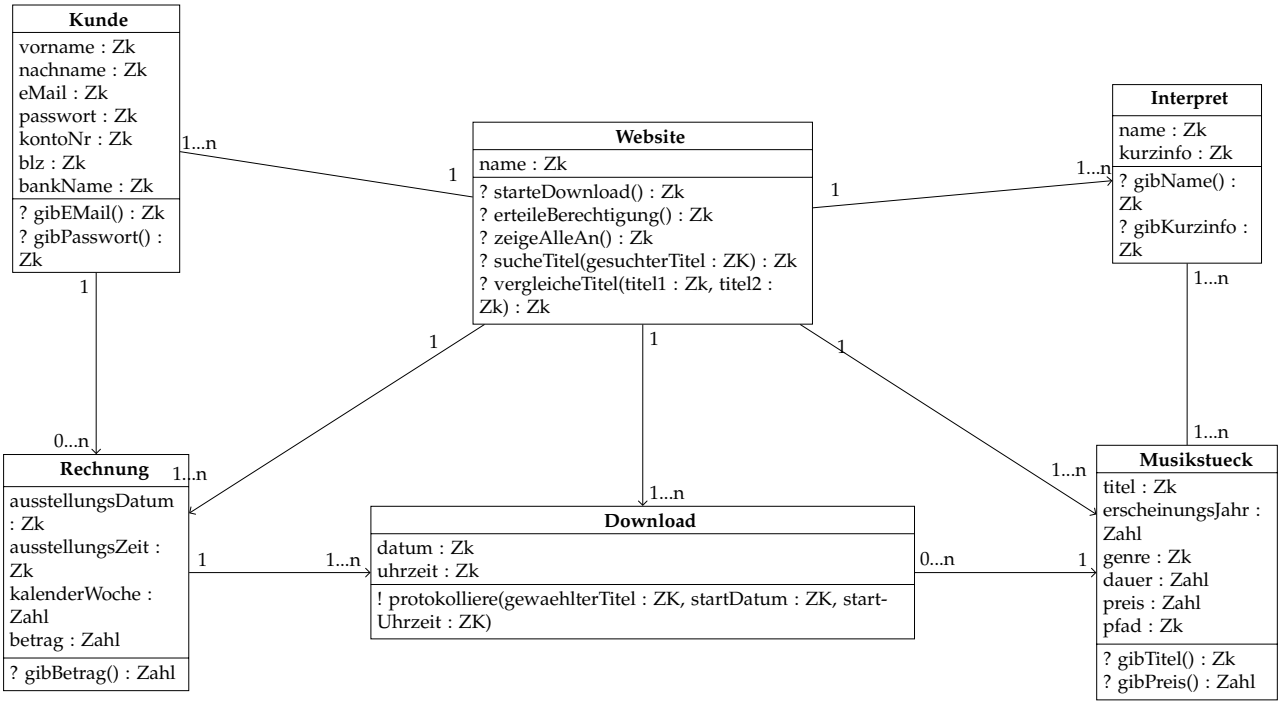
Abbildung A.4: Erfolgreiche Suche

A.4.3 Mögliche Ergebnisse

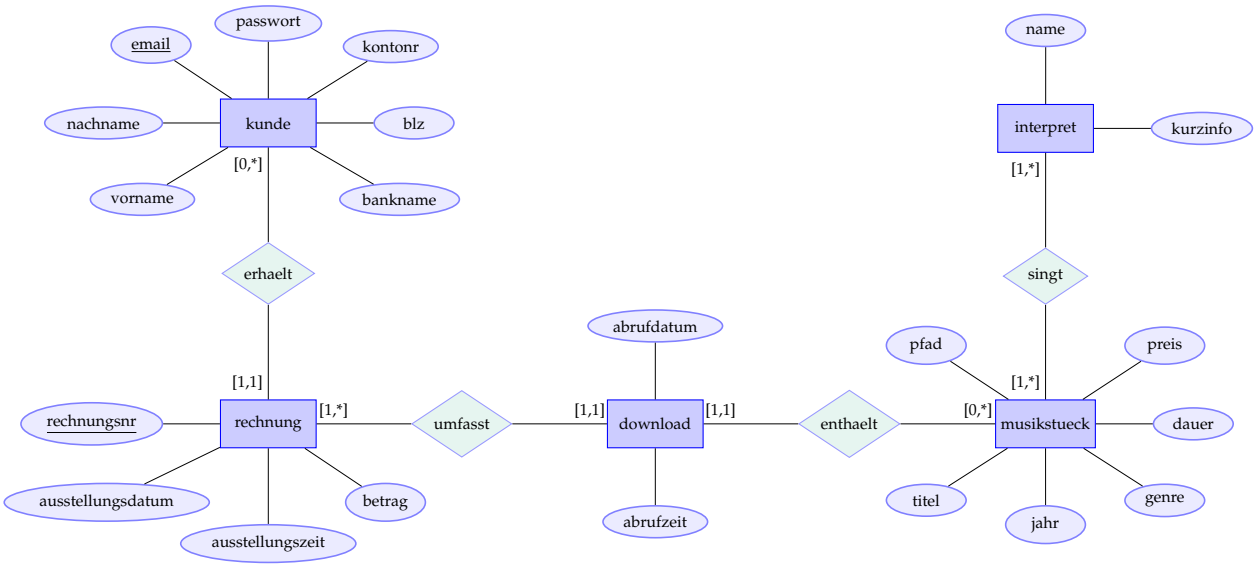
Beispiel für ein Objektdiagramm – Grobskizze



Beispiel für ein Klassendiagramm



Beispiel für ein E/R-Diagramm



A.4.4 Normalisierung

Bitte studieren Sie, um die folgenden Aufgaben bearbeiten zu können, das Leitprogramm Datenbanken.

Normalisierung

Eine Datenbank habe das folgende Relationenschema:

```
fahrer (fid, name, vorname)
lkw (kennzeichen, zuladung, ↑fahrer)
adressen (adid, anschrift, plz, ort)
bearbeitet (↑lkw, ↑auftrag)
auftraege (aid, ↑auftraggeber, eingangsdatum, zieldatum, groesse, route,
   ↑start, ↑ziel, entfernung)
auftraggeber (aaid, name, anschrift)
```

Aufgabe

1. Zählen Sie die Definitionen der ersten drei Normalformen auf.
2. Nehmen Sie Stellung dazu, ob die gegebene Datenbank sich jeweils in der ersten bis dritten Normalform befindet. Gehen Sie davon aus, dass die Datenbank immer so umgewandelt wurde, dass sie der vorherigen Normalform entspricht.
3. Führen Sie die nötigen Änderungen durch, um die Datenbank in die dritte Normalform zu bringen.

Tabelle A.2: Inhalt der Tabelle fahrer

fid	name	vorname
1	Shuster	Sven
2	Eisenhower	Nicole
3	Jaeger	Jessika
4	Grunewald	Sarah
5	Nagel	Kathrin
6	Beckenbauer	Thorsten
7	Daecher	Manuela
8	Hoch	Katrin
9	Abendroth	Wolfgang

Tabelle A.3: Inhalt der Tabelle lkw

kennzeichen	zuladung	fahrer
AK-I23	24	3
AK-I342	4	2
AK-I42	14	1
AK-T434	14	5
AK-T5133	4	8
AK-T838	12	0

Tabelle A.4: Inhalt der Tabelle adressen

adid	anschrift	plz	ort
1	Gubener Str. 71	57589	Niederirsen
2	Augsburger Strasse 21	57520	Harbach
3	Schönwalder Allee 30	57577	Hamm
4	Michaelkirchstr. 96	57537	Hövels
5	Lützowplatz 71	57539	Roth
6	Hauptstraße 2	57581	Katzwinkel
7	Karl-Liebknecht-Strasse 21	57518	Betzdorf
8	An Der Urania 45	57584	Scheuerfeld

Tabelle A.5: Inhalt der Tabelle bearbeitet

lkw	auftrag
AK-I23	1
AK-I23	3
AK-I342	1
AK-I42	2
AK-T434	2
AK-T5133	4

Tabelle A.6: Inhalt der Tabelle auftraege

aid	auftraggeber	eingangsdatum	zieldatum	groesse	route	start	ziel	entfernung
1	2	2014-03-10	2014-03-17	81	2	4	5	10
2	3	2014-03-12	2014-03-18	120	1	8	4	9
3	2	2014-03-15	2014-03-23	16	2	4	5	10
4	1	2014-03-15	2014-03-18	44	3	5	3	2

Tabelle A.7: Inhalt der Tabelle auftraggeber

aaid	name	anschrift
1	Lebensmittel Freytag	Schönwalder Allee 30, 57577 Hamm
2	Friedmans	An Der Urania 45, 57584 Scheuerfeld
3	Akede	Gubener Str. 71, 57589 Niederirsen
4	Ewer	Hauptstraße 2, 57581 Katzwinkel

A.4.5 Die MySQL-Shell

Zugang zu einer molekularbiologischen Datenbank

A.4.6 Fallbeispiel: Ensembl-Datenbank.

Fallbeispiel: Ensembl-Datenbank.

- Die *Ensembl-Datenbank* ist eine große Datenbank, die vielfältige molekularbiologische Daten speichert.
- Der Ensembl-Server ist ein Informatiksystem irgendwo auf diesem Globus, auf dem das *MySQL-Server-Programm* läuft und auf dessen externen Speichermedien die Daten lagern.
- Auf einem beliebigen Informatiksystem kann man nun die Datenbankshell `mysql` starten und die Adresse des Ensembl-Servers angeben.
- Daraufhin hat man (nur lesenden) Zugriff auf alle Tabellen der Ensembl-Datenbanken.
- Genauer verwaltet der Ensembl-Server viele Datenbanken und jede enthält wiederum viele Tabellen. Als Nutzer muss man am Anfang eine dieser Datenbanken auswählen, man arbeitet dann nur noch mit den Tabellen dieser einen Datenbank.

A.4.7 Die MySQL-Shell

Start der MySQL-Shell.

- Das Programm `mysql` dient dazu, eine Verbindung zu einem Datenbankserver aufzubauen.
- Es nimmt viele optionale Parameter, die man sich mittels `mysql --help` alle anzeigen lassen kann.
- Nach dem *Prompt* kann man nun SQL-Befehle eingeben.
  - Nützlich sind dort die Hoch- und Runterpfeiltasten, um die letzten eingegebenen Befehle nochmal zu betrachten.
  - Befehle sollte man mit einem Semikolon (wie in Python) abschließen.

Die wichtigsten Optionen sind:

`--host=` Dieser Option sollte der Name des Servers folgen (bei Ensembl beispielsweise `ensembldb.ensembl.org`).

`--user=` Dieser Option sollte der Name des Benutzers folgen, auf dessen Account man sich einloggen möchte (bei Ensembl beispielsweise `anonymous`).

Beispiele für den Verbindungsaufbau mit einer Datenbank.

Verbindung mit Ensembl aufbauen

```
1 humbert@abercorn:~$ mysql \
2     --host=ensembldb.ensembl.org --user=anonymous
3 Welcome to the MySQL monitor.  Commands end with ; or \g.
4 mysql>
```

Die Backslashes an den Zeilenenden bedeuten, dass es dort eigentlich *keinen* Zeilenumbruch geben darf (der Text hat nicht in eine Zeile gepasst).

Anhang B

Hinweise

1.9 50 Berücksichtigen Sie die Elemente aus dem letzten Abschnitt, um die Aufgabe zu erledigen. Zurück

Anhang C

Rezepte

C.1 Zahl$_x$ $\rightarrow$ Dezimalzahl$_{10}$

Umrechnung von Zahldarstellungen in das Dezimalsystem

Ziel

Sichere und schnelle Umrechnung von Zahlen – benötigt werden nur die Rechenoperationen Addition und Multiplikation.

Rezept

Sollen Dualzahlen, Oktalzahlen, Hexadezimalzahlen oder . . . *in das Dezimalsystem* umgerechnet werden, so kann dies mit Hilfe des HORNER-Schemas schnell und sicher erledigt werden, da nur die Rechenoperationen Addieren und Multiplizieren benutzt werden.

Regeln zur Umrechnung in eine Dezimalzahl

Die einzelnen Ziffern der umzuwandelnden Zahl werden mit etwas Abstand aufgeschrieben. Man beginnt dann ganz links mit der höchsten Stelle. Von oben nach unten wird dann ziffernweise addiert, von unten nach schräg rechts eine Ebene höher wird mit der **Basis** des jeweiligen Zahlensystems (also 2 bei Dualzahlen, 8 bei Oktalzahlen und 16 bei Hexadezimalzahlen) multiplziert (vgl. erstes Beispiel).

Beispiele – Aufgabe

Bitte rechnen Sie zunächst alle Beispiele selbst durch und prüfen Sie, ob Ihre Ergebnisse mit den hier dargestellten übereinstimmen.

- $11101010_2 \rightarrow$

	1	1	1	0	1	0	1	0
2		2	6	14	28			
	1	3	7	14	$\cdot 2$			

Tragen Sie die fehlenden Zahlen ein und ermitteln Sie das Gesamtergebnis. Das Ergebnis ist der Dezimalwert von 11101010_2.

- $7502_8 \rightarrow$ 8

7	5	0	2
	56	488	3904
7	61	488	3906

- $3A5F_{16} \rightarrow$ 16

3	10	5	15
	48	928	14928
3	58	933	14943

- $11001_2 \rightarrow$ 2

1	1	0	0	1
	2	6	12	24
1	3	6	12	25

C.2 Dezimalzahl$_{10}$ $\rightarrow$ Zahl$_x$

Divisionsverfahren – eine Dezimalzahl in ein anderes Zahlensystem umwandeln

Ziel

Sichere Umrechnung von Dezimalzahlen – benötigt werden die Rechenoperationen Subtraktion und Division (ganzzahlig).

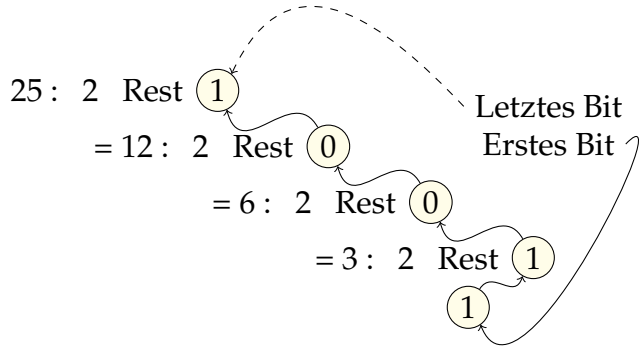
Rezept

Regeln zur Umrechnung einer Dezimalzahl

Als *Basis* wird im Folgenden die Basis des Zahlensystems bezeichnet, in das die Dezimalzahl umgewandelt werden soll: Die Dezimalzahl wird – wie in der Grundschule – fortlaufend durch die **Basis** geteilt (dividiert). Der Rest der Divisionsoperation wird rechts aufgeschrieben, das Ergebnis wird eine Zeile tiefer notiert. Mit dem Ergebnis wird ebenso verfahren. Das Gesamtergebnis ergibt sich durch das Aufschreiben der Reste – beginnend mit dem zuletzt errechneten Rest.

Beispiel – Aufgabe

Die Dezimalzahl 25_{10} soll in eine Dualzahl (Basis 2) umgewandelt werden:



Prüfen Sie das Ergebnis, indem Sie 11001_2 mit dem HORNER-Schema in eine Dezimalzahl umrechnen.

C.3 Wir »spielen« einen Prozessor

Schaltnetz zur Realisierung der Addition zweier Binärzahlen

Ziel

Vorlage mit dem Layout für die verschiedenen Gatter und ihre Verbindungen.

Rezept

Schaltungslayout – Schulhof oder Turnhalle

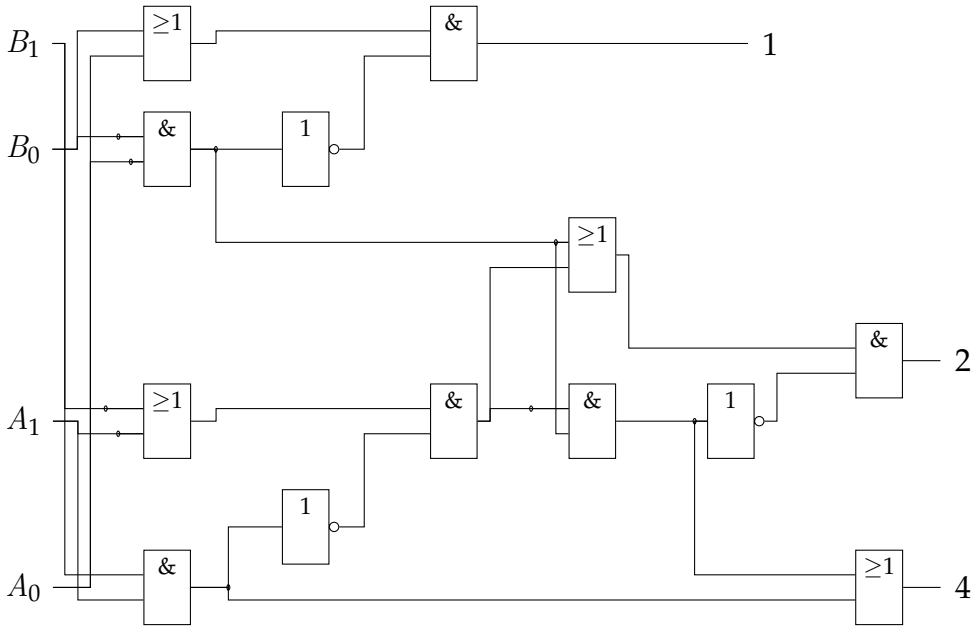
Die Grundidee besteht darin, dass Schülerinnen und Schüler schrittweise die Addition von zwei Binärzahlen (zweistellig) mit Hilfe eines Schaltwerks durchführen.
Dazu benötigen wir pro Eingabestelle jeweils zwei Schüler oder Schülerinnen – also in diesem Beispiel acht Schüler für die Eingabewerte. Die Schaltelemente müssen so besetzt werden, dass die anschließenden Verzweigungen realisiert werden können → 23 Schülerinnen und Schüler könnten damit »beschäftigt« werden, damit alle Eingänge »beschickt« werden.
Beim ersten »Takt« gehen diese Schüler zu den Eingängen der `Oder`- beziehungsweise `Und`-Schaltelemente.
Dort wird nach der folgenden Regel gearbeitet:

- 0 führt zum einfachen Hinstellen bei dem Schaltelement,

- 1 führt dazu, dass eine Hand auf die Schulter des Schaltelements gelegt wird.

- Ist das Schaltelement ein `Oder`, so reicht eine Hand auf der Schulter, damit das Element eine 1 weitergibt – keine Hand führt zur 0.
- Ist das Schaltelement ein `Und`, so sind zwei Hände auf der Schulter nötig, damit das Element eine 1 weitergibt – andernfalls wird 0 weitergegeben.

Das **Schaltelement** geht beim nächsten Takt eine Stufe weiter.
Hier taucht ein weiteres Schaltelement auf – das `Nicht`. Es kehrt um, was eingegeben wird: also wird aus 1 eine 0 und umgekehrt.



Die Idee wurde mir durch eine Anfrage via Twitter von @ManchesterBudo bekannt, die zu einem Videobeitrag bei der BBC (von Dave Cliff) führte, in dem das »Spiel« durchgeführt wird: `http://www.bbc.co.uk/programmes/p01m5xfs` veröffentlicht am 25. November 2013 – geprüft: 5. September 2016.

C.4 Schlüsselworte in Python3

In Python3 festgelegte Schlüsselworte anzeigen

Ziel
Reservierte Worte in Python3.

Rezept

In der Python3-Konsole können die Schlüsselworte abgefragt werden:

```
>>> from keyword import kwlist
>>> print(kwlist)
```

Die lexikographisch geordnete Liste aller 33 Schlüsselworte aus Python3. Nicht alle werden für den Informatikunterricht benötigt.

- False
- None
- True
- and
- as
- assert
- break
- class
- continue
- def
- del
- elif
- else
- except
- finally
- for
- from
- global
- if
- import
- in
- is
- lambda
- nonlocal
- not
- or
- pass
- raise
- return
- try
- while
- with
- yield

Schlüsselworte werden auch reservierte Bezeichner oder festgelegte Bezeichner genannt. Diese können Kategorien zugeordnet werden, die im Folgenden für einige der Bezeichner angegeben sind.

Werte `True`, `False`, `None`

Logische Operatoren `or`, `and`, `not`, `in`

Elemente für Kontrollstrukturen `if`, `try`, `while`

Strukturausweis – OO: Klasse und Methode `class`, `def`

Schnittstellennutzung `from`, `import`

C.5 Python3 – eingebaute, vordefinierte Bezeichner

In Python3 eingebaute, vordefinierte Bezeichner anzeigen

Ziel

Eingebaute, vordefinierte Bezeichner in Python3.

Rezept

In der Python3-Konsole können die eingebauten, vordefinierten Bezeichner herausgefunden werden:

```
>>> dir(__builtins__)
```

Die Liste der vordefinierten Bezeichner in Python3 enthält 148 Elemente. Hier werden – in lexikographischer Reihenfolge – die angegeben, die häufiger benötigt werden.

- abs
- all
- any
- ascii
- bin
- bool
- bytearray
- bytes
- callable
- chr
- class-method
- compile
- complex
- dict
- dir
- divmod
- enumerate
- eval
- exec
- exit
- filter
- float
- format
- frozenset
- getattr
- globals
- hasattr
- hash
- help
- hex
- id
- input
- int
- isinstance
- issubclass
- iter
- len
- license
- list
- locals
- map
- max
- min
- next
- object
- oct
- open
- ord
- pow
- print
- property
- quit
- range
- repr
- reversed
- round
- set
- setattr
- slice
- sorted
- static-method
- str
- sum
- super
- tuple
- type
- vars
- zip

Die vordefinierten Bezeichner sind oft mit Aktionen verbunden und können so als Funktionen benutzt werden. Im Folgenden werden einige Kategorien mit Beispielen angegeben. Weder die Kategorien, noch die Beispiele sind vollständig.

Mathematische Funktionen `abs`, `divmod`, `len`, `max`, `sum`

Datentypen (auch zur Umwandlung) `bin`, `bool`, `chr`, `dict`, `float`

Introspektion `callable`, `dir`, `isinstance`

C.6 Python3 – Erzeugen von »Zufallszahlen«

Python3 und »zufällige« Zahlen

Ziel

»Zufallszahlen« in Python3 erstellen.

Rezept

Geben Sie in der Python3-Konsole die hinter >>> stehenden Zeichen ein:

```
>>> from random import randint
>>> randint(1,6)
```

Damit haben Sie den Zufallszahlengenerator von Python zur Simulation eines Würfels genutzt und einmal »gewürfelt«. Die auf diese Weise erzeugten »Zufallszahlen« haben eine Periode von $2^{19937} - 1$. Damit sind diese Zahlen **nicht wirklich** zufällig. Dennoch sollte dies auch für einen etwas längeren Spieleabend ausreichend viele »zufällige« Zahlen produzieren.

C.7 ER-Diagrammvereinbarungen

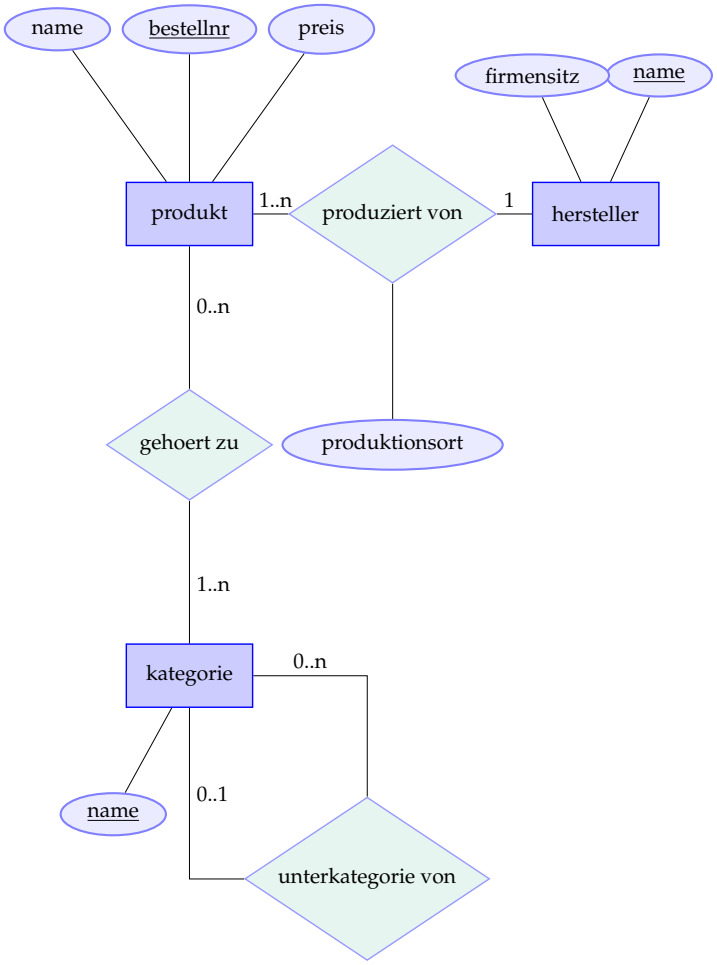
ER-Diagramme dienen der Darstellung der Ergebnisse der ER-Modellierung

Ziel

ER-Diagramme lesen und erstellen.

Rezept

Zur graphischen Darstellung der Beziehungen zwischen Tabellen in einer relationalen Datenbank werden ER-Diagramme herangezogen. Entitäten werden dabei in Rechtecke geschrieben und mit den Attributen in Ovalen verbunden. Schlüsselattribute werden dabei durch unterstreichen deutlich gemacht. Die Beziehungen werden mit Rauten zwischen den Entitäten gezeichnet an deren Verbindungslinien die Kardinalitäten angegeben werden. Diese Beziehungen können selber auch zusätzliche Attribute haben.



C.8 SQL-Datenbankabfragesprache

SQL-Festlegungen zur Notation

Ziel

Korrekte Abfragen in SQL erstellen.

Rezept

Bei der Datenbankabfragesprache SQL sind alle Schlüsselworte in Großbuchstaben zu schreiben. Außerdem ist das abschließenden Semikolon mit anzugeben. Tabellennamen und Spaltennamen sollen klein geschrieben werden und keine Sonderzeichen enthalten. Daher müssen sie innerhalb von Befehlen auch nicht in Backticks eingefasst werden. Die Wahl zwischen einfachen oder doppelten Anführungszeichen bei dem Einfassen von Zeichenketten ist frei zu treffen. Dieses sollte, wie im folgenden Beispiel, innerhalb eines Befehls einheitlich sein.

```
SELECT name, preis FROM produkt WHERE hersteller =
   "Augusto" AND ursprungsland = "Deutschland";
```

Literatur

Gamma, Erich u. a. (1996). *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. 1. Aufl. Design Patterns, 1995, Deutsche Übersetzung von Dirk Riehle. Bonn: Addison-Wesley. ISBN: 3-89319-950-0.

Lage, Klaus (1991). 1.000 *und* 1 *Nacht – Zoom*. Songtext. URL: `http://is.gd/WR8P4C` (besucht am 22. 03. 2016).

Löffler, Susanne (2010). »Von der objektorientierten Modellierung zur Datenbank – ein Konzept und seine Umsetzung mit Mobiltelefonen in der gymnasialen Oberstufe«. Hausarbeit gemäß OVP. Hamm: Studienseminar für Lehrämter an Schulen – Seminar für das Lehramt für Gymnasien/Gesamtschulen. URL: `https://is.gd/RTQrgm` (besucht am 28. 05. 2016).

Pieper, Johannes und Dorothee Müller, Hrsg. (2014). *Material für den Informatikunterricht*. Arnsberg, Dortmund, Hamm, Solingen, Wuppertal. URL: `http://uni-w.de/1t` (besucht am 29. 04. 2016).