

Skriptum gemäß schulinternem Lehrplan

Grundkurs Informatik – Q1

Stand: 07. März 2016

Sprachvariante: Java



Inhaltsverzeichnis

Skriptum Q1	1
Inhaltsverzeichnis	4
Vorwort	5
1 Datenstrukturen	7
1.1 Kompetenzen	7
1.2 Lineare Datenstrukturen	8
1.2.1 Aufzählungen, Reihungen, Felder, Tabellen, Ketten	8
1.2.2 Array-Typen	11
1.2.3 Speicherung von Arrays	15
Zusammenfassung	17
1.3 Nichtlineare Datenstrukturen	18
1.3.1 Idee hinter Listen	19
1.3.2 Die Datenstruktur	20
1.3.3 Operationen auf Listen	22
1.3.4 Löschen am Anfang	23
Zusammenfassung	24
1.3.5 Listen – Iteration und Modifikation. Von verbogenen Zeigern	24
1.3.6 Iteration	25
1.3.7 Anwendung: Map	29
1.3.8 Modifikation	30
Zusammenfassung	33
2 Suchen und Sortieren	34
2.1 Kompetenzen	34
2.2 Varianten des Suchproblems	35
2.2.1 Lineare Suche	36
2.2.2 Binäre Suche	38
Zusammenfassung	41
2.3 Sortieralgorithmen – Skat, Telefonbuch, Atome	41
2.3.1 Die Problemstellung	42
2.3.2 Sortieralgorithmen	43
Zusammenfassung	49

3	Bäume	50
3.1	Kompetenzen	50
3.2	Motivation	51
3.2.1	Modellierung	51
3.2.2	Bessere Datenstrukturen	52
3.3	Begriffe	53
3.3.1	Der Begriff des Baumes	53
3.3.2	Arten von Bäumen	54
3.4	Bäume in Python	55
3.4.1	Beteiligte Klassen	55
3.4.2	Konstruktion	56
3.4.3	Einfügen und Löschen	57
3.4.4	Traversierung	58
	Zusammenfassung	59
4	Graphen	60
4.1	Kompetenzen	60
4.2	Modellierung mit Graphen	61
4.2.1	Modellierung mittels Graphen	61
4.2.2	Arten und Formalisierungen	62
4.2.3	Graphprobleme	64
4.3	Graphen in Python	64
4.3.1	Adjazenzmatrizen	64
4.3.2	Adjazenzlisten	65
4.4	Graphproblem: Der Handlungsreisende	66
4.4.1	Problemstellung	66
	Zusammenfassung	67
5	Abstrakte Klassen, Polymorphie, MVC	68
5.1	Kompetenzen	68
5.2	Konzept ADT	69
5.2.1	Interface versus Implementation	69
5.2.2	Idee des abstrakten Datentyps	70
5.2.3	Syntax von Interfaces	71
5.2.4	Beispiele	71
	Zusammenfassung	77
5.3	MVC	77
6	Datenbanken	78
6.1	Kompetenzen	78
6.2	Datenbanksysteme	79
6.2.1	Was sind Datenbanken?	80
6.2.2	Aufbau von Datenbanken	82
6.2.3	Arten von Datenbanken	83



6.3	Datenmodelle	84
6.4	Das E/R-Modell	85
6.4.1	Einführung	85
6.4.2	Entitäten	86
6.4.3	Attribute	87
6.4.4	Relationships	87
	Zusammenfassung	88
6.5	SQL	89
6.5.1	SQL zur Kommunikation	89
A	Arbeits- und Informationsblätter, Lernzielkontrollen	91
A.1	Vorhaben Q1-1	91
A.1.1	Eingangsbefragung – Selbsteinschätzung	91
A.1.2	Eingangskompetenzen – Beispielmodellierung	92
B	Hinweise	94
C	Rezepte	95
C.1	$Zahl_x \rightarrow \text{Dezimalzahl}_{10}$	95
C.2	$\text{Dezimalzahl}_{10} \rightarrow Zahl_x$	96
C.3	Wir »spielen« einen Prozessor	97
C.4	Schlüsselworte in Python3	98
C.5	Python3 – eingebaute, vordefinierte Bezeichner	99
C.6	Python3 – Erzeugen von »Zufallszahlen«	100
	Literatur	101



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 Lehrerbildung (ZfsL) Hamm, Arnsberg und Solingen entwickelt und im Unterricht erprobt.

Teile des Skriptums wurden so geändert, dass nunmehr die Unterstützung von Typen 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)).

Das vorliegende Dokument steht unter der »Creative Commons Lizenz«  – BY-NC-ND. Dies bedeutet: bei weiterer Verwendung des Textes sind die Namen der Autoren zu nennen, die Weiternutzung darf ausschließlich nicht kommerziell erfolgen, das Dokument darf nicht bearbeitet werden. Details zu den Creative Commons (CC) Lizenzen finden sich unter: <http://creativecommons.org/licenses/?lang=de>

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	write	execute
user	yes	yes	yes
group	no	no	no
other	no	no	no

Vorhaben 1

Verwaltung von Daten in linearen Datenstrukturen¹

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).

¹Einige 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?

Arrays² 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 Java³ bei 0 statt bei 1 beginnt. Dieser Eindruck täuscht: In Wirklichkeit ist dies nicht nur eine kleine Merkwürdigkeit, sondern eine noch viel diabolischere Gemeinheit als

²In 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*.

³Diese Indizierung gilt für die meisten Programmiersprachen – nicht nur für Java.



der Umstand, dass in vielen Programmiersprachen⁴ 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 in den meisten Fällen mitteilen (*Inkompatible Typen*). 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.

⁴Leider konnten die Varianten der Algol-Sprachfamilie nicht durchgesetzt werden, die für die Zuweisung die Konstruktion `:=` vorgesehen haben. Noch schöner wären \leftarrow 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 Java abgebildet werden?

Erste mögliche Implementierung.



Aufgabe 1.1

```

1 String kontoeigner1 = "Zeisel"
2 String kontoeigner2 = "Charly"
3 String kontoeigner3 = "Wacker"
4 String kontoeigner4 = "Bär"
5 // ...
6 String kontoeigner500 = "Ackermann"

```

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

Zweite mögliche Implementierung.

```

1 String[] kunden = new String[501];
2
3 kunden[1] = "Zeisel";
4 kunden[2] = "Charly";
5 // ...
6 kunden[500] = "Ackermann";

```

Nun ist folgendes möglich:



```
1 String gesucht = "Wacker";
2 for (String derKunde : kunden) {
3     if (gesucht.equals(derKunde)) {
4         System.out.println(gesucht + " ist Kunde der Bank.");
5     }
6 }
```

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 `String`-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 `BspKlasse[]` der Typ eines Arrays von `BspKlasse`-Objekten.
- 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.

Analogie: Der Typ `String` (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 = new String[4]
2 // Jetzt enthält das Objekt kunden
3 // einen Array der Länge 4
```

Zugriff auf Arrays

Wie greift man auf Arrays zu?

- Hat man ein Array-Objekt *reihung*, 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 int[] werte = new int[1000];
2
3 // Beliebter Anfängerfehler:
4 int 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 // »Exception java.lang.ArrayIndexOutOfBoundsException: 1000«
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 `int[] werte= new int[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.

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

Tabelle 1.1: Wertetabelle mit Ergebnis des Schreibtischtests

Zwei Dinge fallen auf:

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 Java eine Fehlermeldung erzeugt (siehe Zeile 13 im Quellcode auf Seite 13).



Eine wichtige (interne) Methode.

- Für ein Array-Objekt `arrObj` liefert `arrObj.length` die Größe des Arrays. Dieses Attribut beinhaltet die Länge des Arrays, das ist die Anzahl der Objekte in diesem Array-Objekt.

Algorithmen auf Arrays

Umdrehen eines Arrays

```

1 int[] vektor = new int[1000];
2 // ... vektor wird gefüllt
3 int index= 0;
4 while (index < vektor.length / 2) {
5     int temp = vektor[index];
6     vektor[index] = vektor[vektor.length-index-1];
7     vektor[vektor.length-index-1] = temp;
8     index = index + 1;
9 }
10 System.out.println("Vektor:" + Arrays.toString(vektor));

```

Hilfsmittel für Array lassen sich wie hier mit `import java.util.Arrays`; vor der Klassendefinition einbinden.

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`.

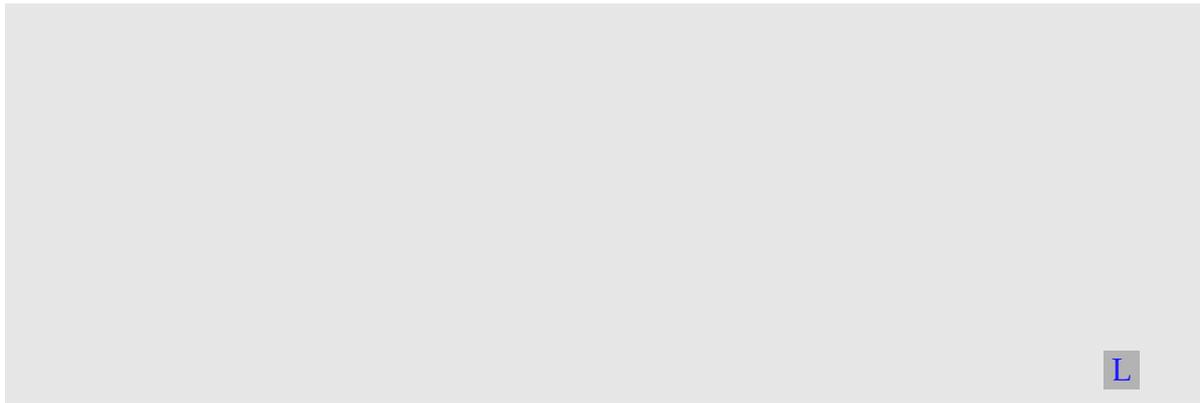
```

1 // Zwei Array-Objekte
2 String[] a1= {"h", "a", "l"}; // oder etwas anders
3 String[] a2= {"l", "o"};      // oder etwas anders
4
5 // Der Array, in den die Verkettung hinein soll:
6 String[] z= new String[a1.length + a2.length];

```

Schreiben Sie bitte Ihre Lösung in das folgende Textfeld:





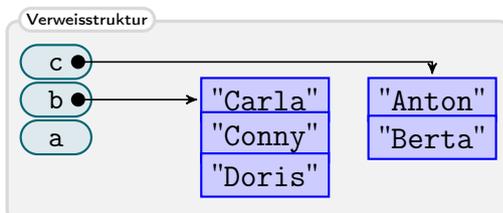
1.2.3 Speicherung von Arrays

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*.

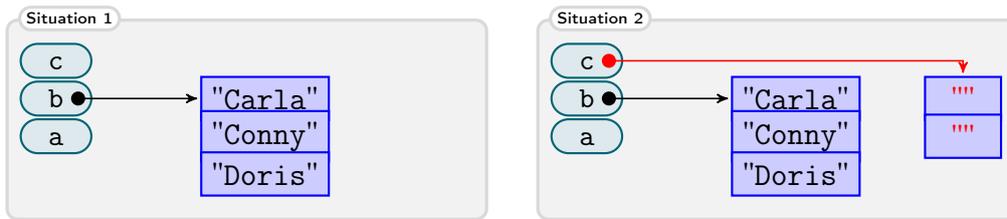
```
1 a= str()
2 b= ["Carla", "Conny", "Doris"]
3 c= ["Anton", "Berta"]
```



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)
```



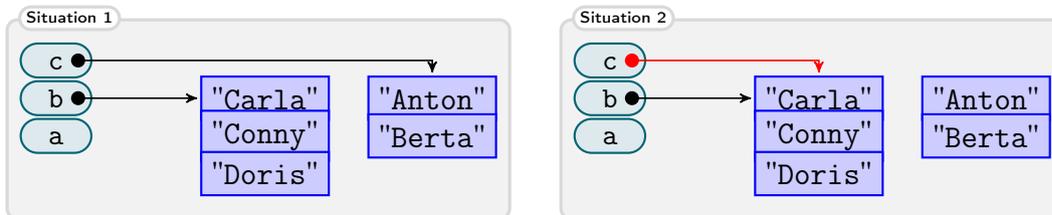


Zuweisung und Vergleich von Verweistypen

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" → "a"
- "c" → "g"
- "g" → "c"

Die Sequenzen sind als Array aus einzelnen Zeichenobjekten gespeichert:

```
1 sequenz = ["t", "c", "c", "t", "a", "t"]
2 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:

```
1 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:



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

- In zwei Zeilen des Codes befinden sich Fehler. Wie lauten diese Zeilen richtig?
- Wie lautet der Inhalt des Arrays **b** nach Ausführung des von Ihnen **berichtigten** Codes?

1.3 Nichtlineare Datenstrukturen – Listen: Konzepte und Erstellung⁵

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 untereinanderbeschriebene Aufstellung 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.

⁵Teile 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.



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.

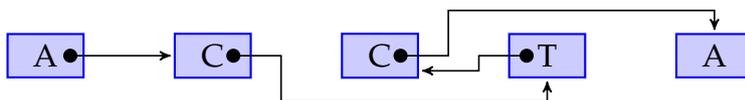
Die Organisation von Daten als Zeltplatz ist sehr vorteilhaft, wenn ständig Leute (Elemente) kommen und gehen. Jedoch dauert bei dieser Organisation das Finden von Zelt Nummer 123 in der Reihenfolge recht lange, man muss 123 Mal kreuz und quer über den Zeltplatz laufen. Bei einer Straße ist es hingegen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

Moral von der Geschichte: 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.
- + Verketteten 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

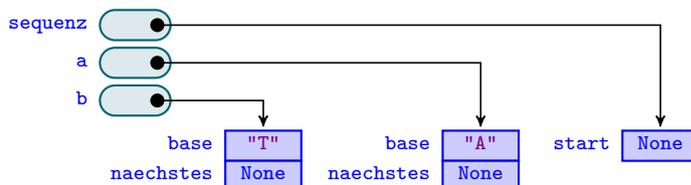
Schritt 1

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

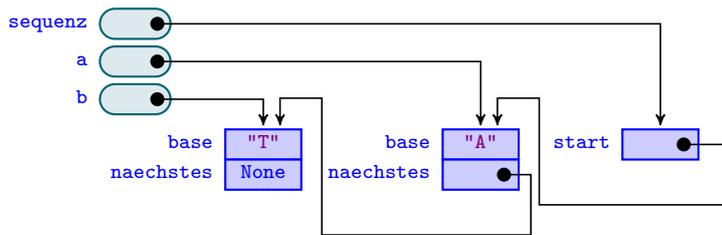
Nun werden die Objekte verkettet.

```

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

```





Zur Übung

Aufgaben 1.4

- Geben Sie den Code der Klassen `Schuelerliste` und `Schuelerzelle` zur Verwaltung einer Liste von Schülern an.
- 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

- Erzeuge eine *neue Zelle* für die neue Base.
- Der *Nachfolger* dieser neuen Zelle ist der *Start der Liste*.
- 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«.

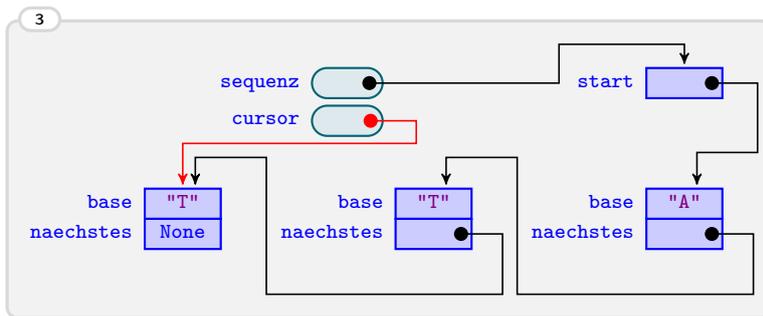
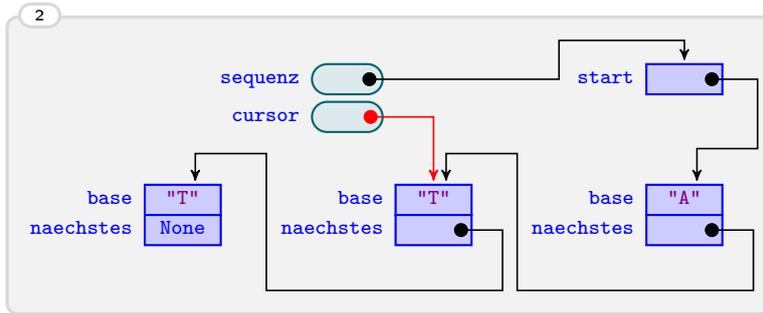
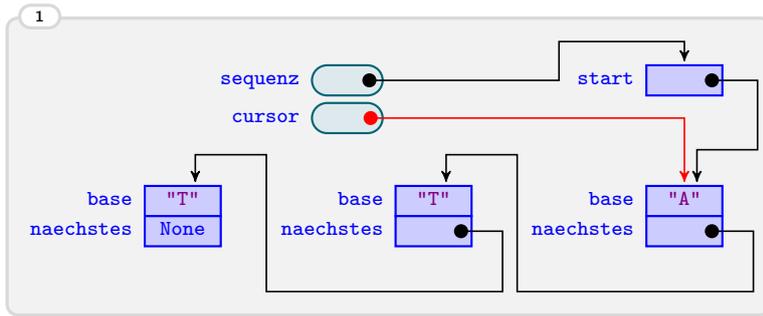
Iterativer Algorithmus

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 25) 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:



```
1 cursor= sequenz.start
2 while cursor != None:
3     tue_was(cursor)
4     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.

```
1 def rekursiverAlgorithmus(self, c: 'Zelle'):
2     if c != None:
3         tue_etwas(c)
4         rekursiverAlgorithmus(c.naechstes)
5
6 # Aufruf:
7 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.

```
1 # Algorithmus zum Zählen der Elemente einer Liste
2 zaehler= 0
3 cursor= sequenz.start
4 while cursor != None:
5     zaehler= zaehler + 1
6     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.

```
1 class DNasequenz:
2     # ...
3     def length(self):
4         zaehler = 0
5         cursor = self.start
6         while cursor != None:
7             zaehler = zaehler + 1
8             cursor = cursor.naechstes
9         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.

```
1 class DNasequenz:
2     # ...
3     def printDNA(self):
4         cursor = self.start
5         while cursor != None:
6             print(cursor.base, end="") # Python3 - ohne Zeilenumbruch
7             cursor = cursor.naechstes
8         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:

```
1 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.

```
1 class DNasequenz:
2     # ...
3     def komplement(self):
4         cursor= self.start
5         while cursor != None:
6             if cursor.base == "A": cursor.base = "T"
7             elif cursor.base == "T": cursor.base = "A"
8             elif cursor.base == "C": cursor.base = "G"
9             elif cursor.base == "G": cursor.base = "C"
10            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.



```
1 class DNasequenz:
2     # ...
3     def suche_erstes_a(self):
4         cursor= self.start
5         while cursor != None:
6             if cursor.base == "A":
7                 return cursor          # Gefunden! Danke und tschüss.
8             cursor= cursor.naechstes
9     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.

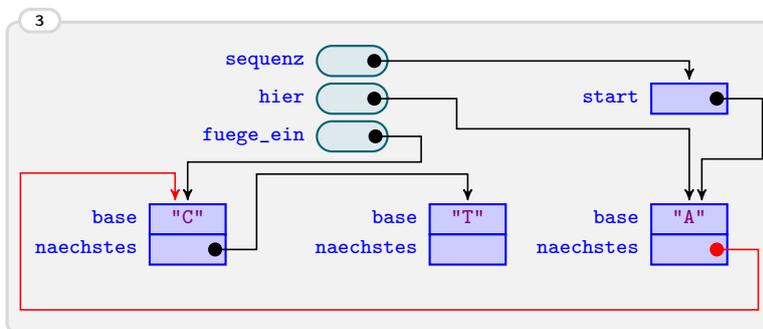
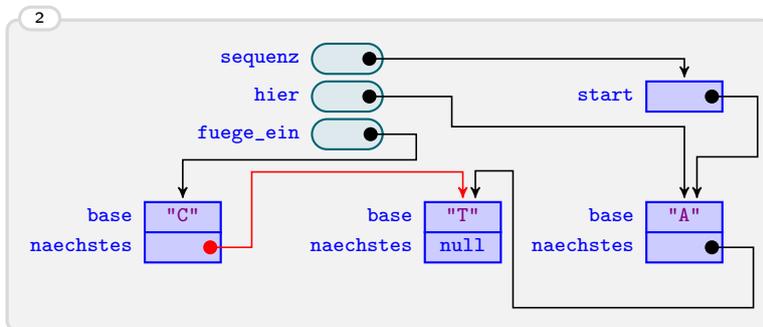
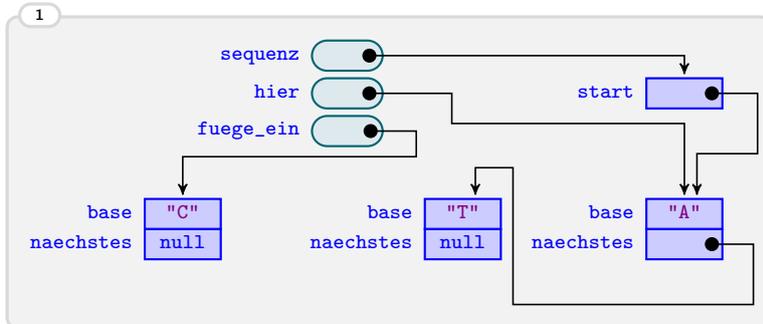
```
1 class DNasequenz:
2     # ...
3     def fuege_ein_nach(self, hier: 'Zelle', b: 'str'):
4         fuege_ein          = Zelle()
5         fuege_ein.base     = b
6         fuege_ein.naechstes= hier.naechstes
7         hier.naechstes     = fuege_ein
```



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.

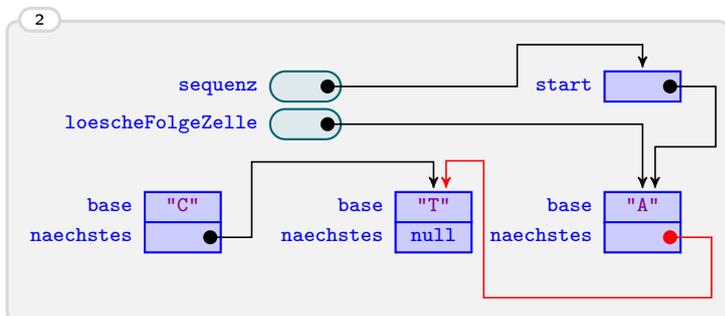
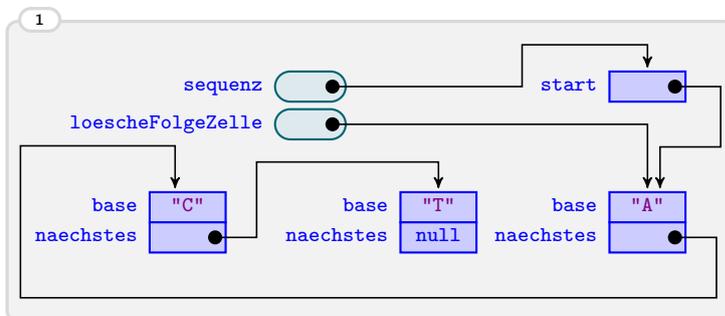
```

1 class DNasequenz:
2     # ...
3     def loesche_nach(self, loescheFolgeZelle: 'Zelle'):
4         # Löscht die Zelle, die auf loescheFolgeZelle folgt:
5         loescheFolgeZelle.naechstes =
            loescheFolgeZelle.naechstes.naechstes
    
```

Wer zeigt wann wohin?

```

1 # 1. Vorher
2 loescheFolgeZelle.naechstes =
    loescheFolgeZelle.naechstes.naechstes
3 # 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 Sortieren¹

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)

¹Teile 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.

```

1 telefonNummern= ["7974311",
2                 "2147856",
3                 "2161555",
4                 "5553466"]
5 # Suche linear nach "2161555":
6
7 position_wert= -1 # noch nicht gefunden
8 index= 0
9 while index < telefonNummern.__len__():
10     if telefonNummern[index] == "2161555":
11         position_wert= index
12         index= index + 1
13
14 # position_wert ist jetzt 2.

```

Eine allgemeine lineare Suche.

```

1 class SuchAlgorithmen:
2
3     def lineareSuche(self, zeichenketten: 'list', wert: 'str'):
4         # Findet erstes Vorkommen von wert in zeichenketten.
5         # Kommt es nicht vor, wird -1 zurückgegeben
6
7         index= 0
8         while index < zeichenketten.__len__():
9             if zeichenketten[index] == wert:
10                return index
11            index= index + 1
12        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.

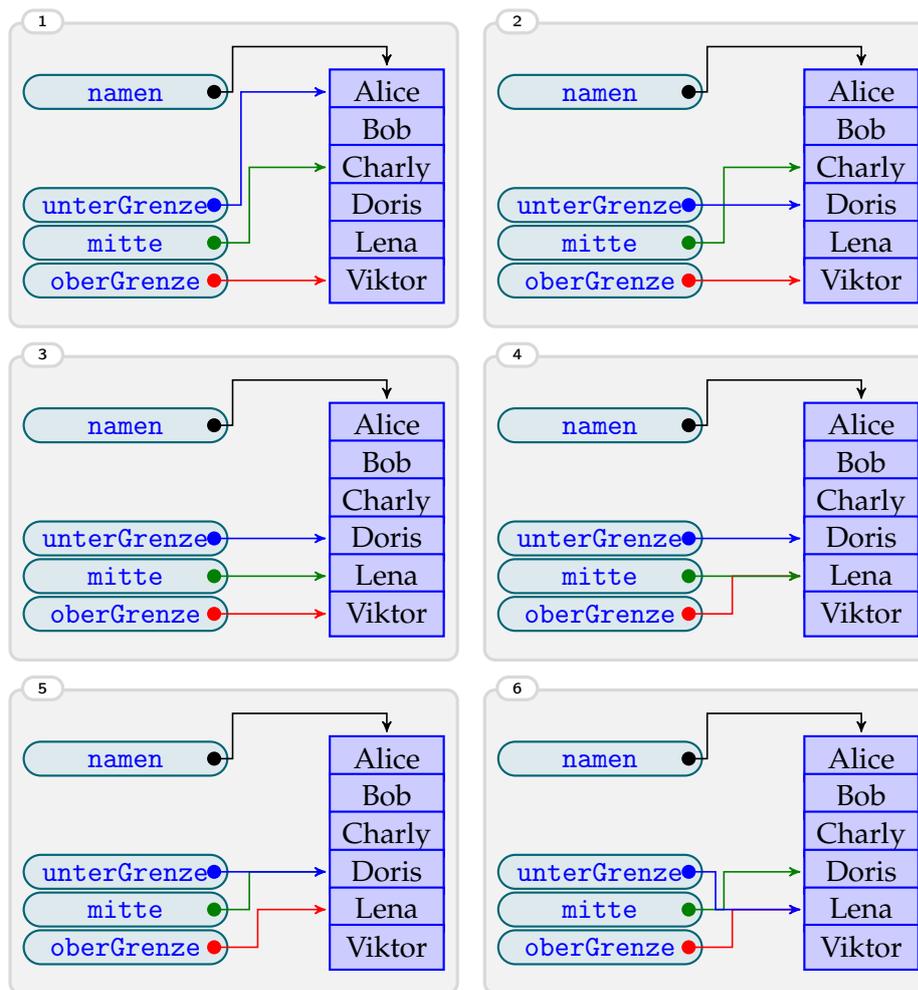
Implementation

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.

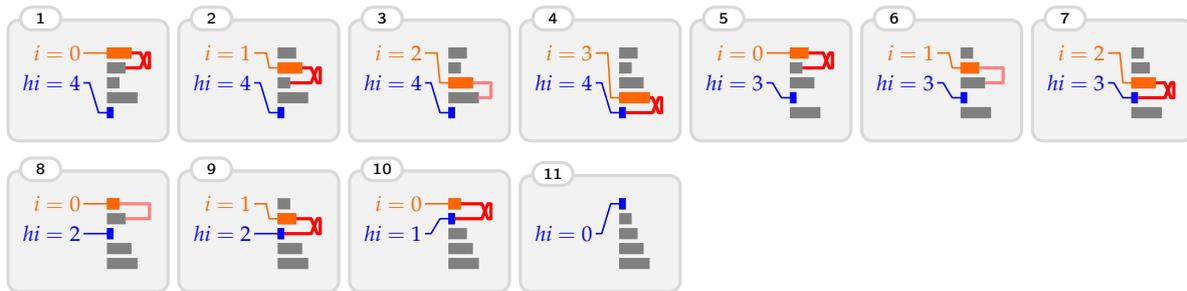
Zweite Version von Bubble-Sort

```
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

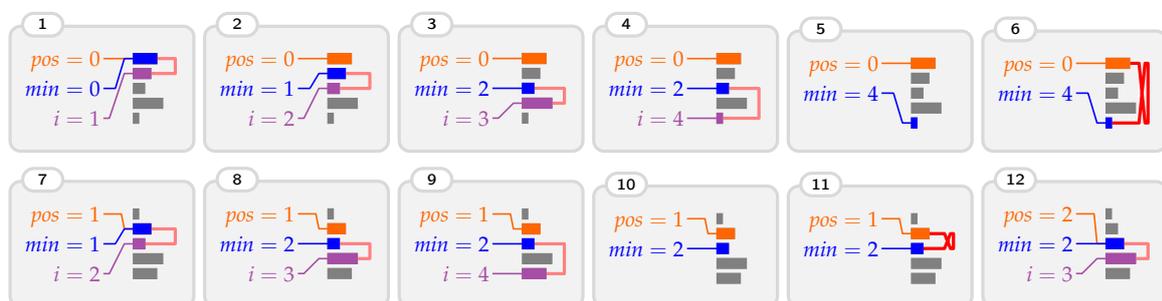
```

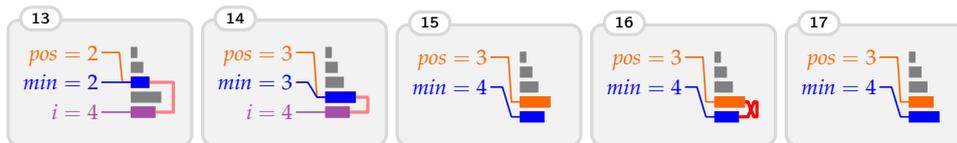
1 def selectionSort(array):
2     for pos in range(0, array.__len__() - 1):
3         # Finde erstes Minimum nach pos
4         min= pos
5
6         for i in range(pos+1, array.__len__()):
7             if array[min] > array[i]:
8                 min= i
9
10        array[pos], array[min]= array[min], 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

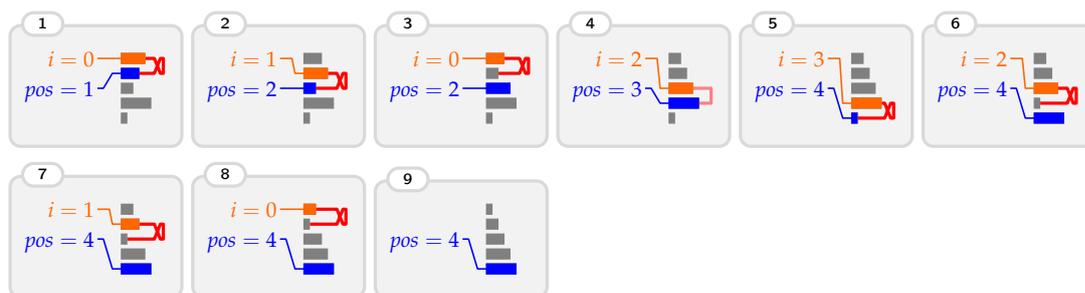
```

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 benötigen *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 *Sortieralgorithmen* 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

Bäume¹

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 Verwendung 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

¹Teile 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.



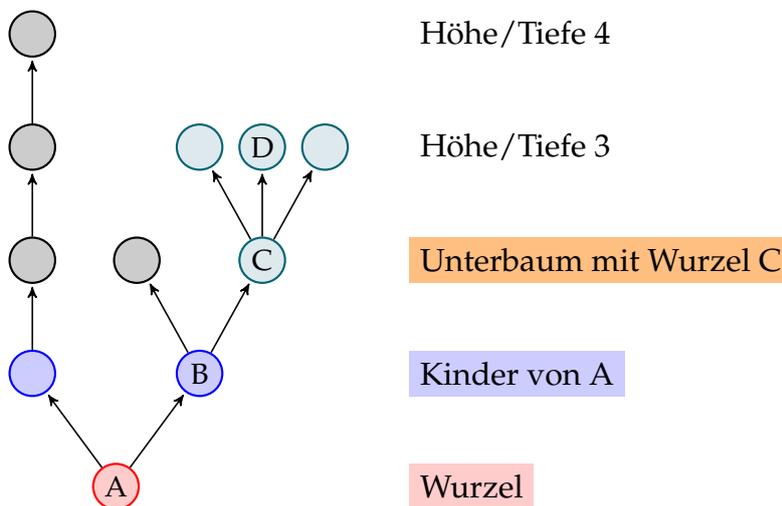
3.3 Begriffe

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 Kindeskindern 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 (Binärbaum²).

²Zur 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?

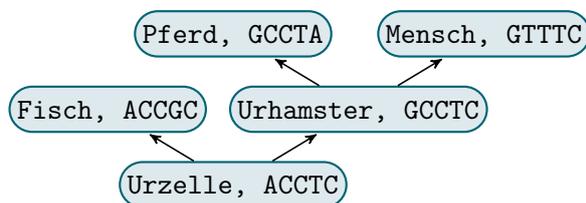
Wie bei Listen benutzt man mehrere Klassen, um Bäume zu implementieren:

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:

```

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

```

Implementation mit höchstens zwei Kindern:

```

1 class Node:
2     def __init__(self, spez: 'Spezies'):
3         self.spezies = spez # Spezies(..., ...)
4         self.leftChild = None # Node
5         self.rightChild = None # Node

```

Bemerkungen:

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

Implementation phylogenetischer Bäume

Die eigentliche Baumklasse

```

1 class PhylogeneticTree:
2     def __init__(self):
3         self.root = None # Wurzel - Klasse Node
4
5     # Methoden
6     def isEmpty(self):
7         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.

```

1 class PhylogeneticTree:
2     # Konstruktor, der einen Baum aufbaut
3     # mit bereits konstruierten linken und rechten
4     # Teilbäumen. Die Wurzel wird neu erzeugt.
5     def __init__(self,
6                 spezies: 'Spezies',
7                 left: 'PhylogeneticTree', right: 'PhylogeneticTree'):
8         self.root= Node(spezies)
9         if left != None:
10            self.root.leftChild= left.root
11        if right != None:
12            self.root.rightChild= right.root

```

Aufgabe 3.2

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

```

1 if __name__ == "__main__":
2     start = Spezies("first_cell", "CCCT")
3     human = Spezies("human", "ACGT")
4     monkey= Spezies("monkey", "ACCT")
5     fish = Spezies("Hommingberger_Gepardenforelle", "CTCT")
6
7     myTree= PhylogeneticTree(start, \
8                             PhylogeneticTree(monkey, None, \
9                             PhylogeneticTree(human, None, None) \
10                            ), \
11                             PhylogeneticTree(fish, None, None) \
12                            )

```

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.



```
1 class PhylogeneticTree:
2     def addTreeAsLeftChild(self,
3         where: 'PhylogeneticTree',
4         what: 'PhylogeneticTree'):
5         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.

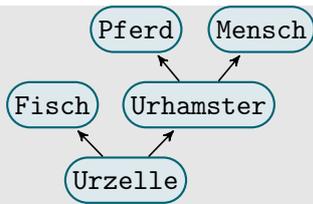
Beispiel einer Traversierung

```
1 class PhylogeneticTree:
2     def printAllSpeziesPreOrder(self, node: 'Node'):
3         if node != None:
4             print(node.spezies.speziesName)
5             self.printAllSpeziesPreOrder(node.leftChild)
6             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

Graphen¹

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«.

¹Teile 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.

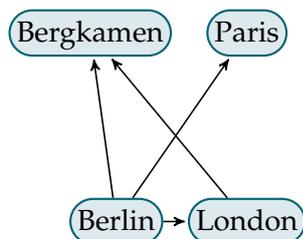
Definition 2 (Graph). Ein *Graph* besteht aus einer Knotenmenge V und einer Kantenmenge $E \subseteq V \times V$.

Solche Graphen nennen wir auch *gerichtete Graphen*.

Beispiel: Gerichteter Graph

$V = \{\text{Berlin, London, Paris, Bergkamen}\}$.

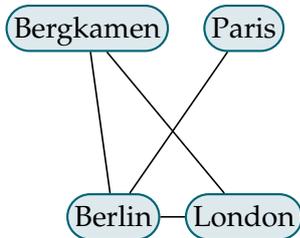
$E = \{(\text{Berlin, London}), (\text{Berlin, Paris}), (\text{Berlin, Bergkamen}), (\text{London, Bergkamen})\}$



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 $(v, u) \in E$.)

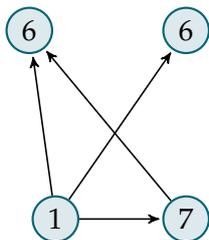
Beispiel: Ungerichteter Graph



Spezielle Graphen: Knotengewichtete Graphen.

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

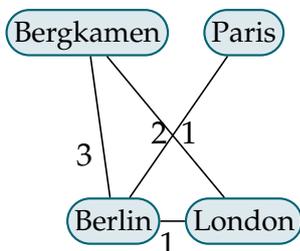
Beispiel: Knotengewichteter Graph



Spezielle Graphen: Kantengewichtete Graphen.

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

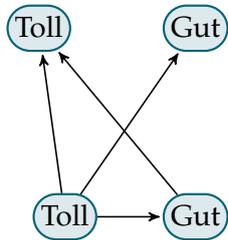
Beispiel: Kantengewichteter Graph



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«).

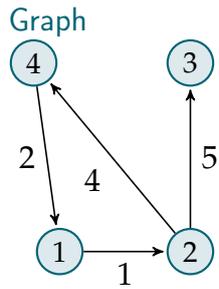
4.3 Graphen in Python

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.





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
5                             range(zahlDerKnoten)]
6
7     def fuegeKnotenZu(self, u, v, knotenGewicht):
8         self.gewichte[u-1][v-1] = knotenGewicht
9
10    def entferneKnoten(self, u, v):
11        self.gewichte[u-1][v-1] = 0
12
13    def verbunden(self, u, v):
14        return self.gewichte[u-1][v-1] > 0 # Boole'scher Ausdruck
15
16    def gibGewicht(self, u, v):
17        return self.gewichte[u-1][v-1]

```

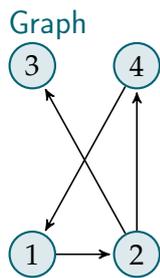
4.3.2 Adjazenzlisten

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.



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 Eine 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 MVC¹

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).

¹Teile 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.

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

Einführung kommt noch.

5.2 Konzept des abstrakten Datentyps

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 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, 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

Die 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:

- Printmethode: `print ()`
- Verkettungsmethode: `concat (dnaSequence)`
- 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

Java 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.

```
1 # Interfaces werden ähnlich wie Klassen aufgeschrieben
2 from abc import ABCMeta, abstractmethod
3 class DNASequenzADT(metaclass=ABCMeta):
4     @abstractmethod
5     def print(self): pass
6
7     @abstractmethod
8     def concat(self, sequenz): pass
9
10 # Man listet die gewünschten Methoden auf.
11 # Man darf keine Attribute angeben und die Methoden
12 # 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:

```
1 class DNASequenzADT(metaclass=ABCMeta):
2     @abstractmethod
3     def print(self): pass
4
5     @abstractmethod
6     def concat(self, sequenz): 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.

```

1 class DNASequence_1:
2     __metaclass__ = DNASequenceADT
3
4     def __init__(self, base: DNASequenceADT):
5         # Erzeuge eine DNA-Sequenz mit nur einer Base
6         self.bases = base
7
8     # Methoden
9     def __len__(self):
10        return self.bases.__len__()
11
12    def __iter__(self):
13        return self.bases.__iter__()
14
15    def print(self):
16        for elem in self.bases:
17            print (elem, end=" ")
18        print ()

```

```

1     def concat (self, seq: DNASequenceADT):
2         # Die Basen aus seq sollen an das Ende von bases
3         # angefügt werden.
4
5         # Neuen Array erzeugen:
6         new_array= [""] * (self.bases.__len__() + seq.__len__())
7
8         # Kopiere eigenen Basen an den Anfang:
9         i= 0
10        for elem in self.bases:
11            new_array[i]= self.bases[i]
12            i= i+1
13
14        # Kopiere andere 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= DNASequenc_1('A')
3     b= DNASequenc_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 `DNASequencADT` 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 ersten Zelle aus über alle Zellen.

Code der ersten Implementation mit Listen.

```

1 class DNASequenc_2:
2     __metaclass__= DNASequencADT
3
4     # Konstruktor
5     def __init__(self, bas):
6         # Erzeuge eine DNA-Sequenz mit nur einer Base
7         self.start = Cell()
8         self.start.base= bas
9         self.start.next= None
10

```



```

11 # Methoden
12 def print(self):
13     cursor= self.start
14     while cursor != None:
15         print(cursor.base,end="")
16         cursor= cursor.next
17     print()

1 def concat (self, seq: DNASequencADT):
2     # Die Basen in seq sollen an das Ende von bases
3     # angefügt werden.
4
5     # Finde das Ende:
6     cursor= self.start
7     while cursor.next != None:
8         cursor= cursor.next
9
10    # Hänge seq an:
11    cursor.next= seq.start

```

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

```

1 print ("Interface wird mit Liste realisiert:")
2 a= DNASequenc_2('A')
3 b= DNASequenc_2('C')
4 a.print()
5 b.print()
6
7 a.concat(b)
8 a.print()

```

Implementation 3: Nochmal eine Liste zur Speicherung der Sequenzen.

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

Code der zweiten Implementation mit Listen.



```

1 class DNASequence_3:
2     __metaclass__ = DNASequenceADT
3
4     # Konstruktor
5     def __init__(self, bas):
6         # Erzeuge eine DNA-Sequenz mit nur einer Base
7         self.start = Cell()
8         self.start.base = bas
9         self.start.next = None
10        self.end = self.start

```

```

1     # Methoden
2     def print(self):
3         cursor = self.start
4         while cursor != None:
5             print(cursor.base, end=" ")
6             cursor = cursor.next
7         print()
8
9     def concat(self, seq: DNASequenceADT):
10        # Die Basen in seq sollen an das Ende von bases
11        # angefügt werden.
12        self.end.next = seq.start
13        self.end = seq.end

```

Visualisieren Sie die Objekte und Verweise:

```

1 print ("Interface wird mit Liste (inkl. Endemarkierung)
2     realisiert:")
3 a = DNASequence_3('A')
4 b = DNASequence_3('C')
5 a.print()
6 b.print()
7
8 a.concat(b)
9 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.

```

1 class Cell:
2     def __init__(self):
3         self.base= ""
4         self.next= None
5         self.prev= None

1 class DNASequene_4:
2     __metaclass__ = DNASequeneADT
3
4     # Konstruktor
5     def __init__(self, bas):
6         # Erzeuge eine DNA-Sequenz mit nur einer Base
7         self.start = Cell()
8         self.start.base= bas
9         self.start.next= None
10        self.start.prev= None
11        self.end = self.start

1     # Methoden
2     def print(self):
3         cursor= self.start
4         while cursor != None:
5             print(cursor.base, end=" ")
6             cursor= cursor.next
7         print()
8
9     def concat (self, seq: DNASequeneADT):
10        # Die Basen in seq sollen an das Ende von bases
11        # angefügt werden.
12        self.end.next = seq.start
13        seq.start.prev= self.end
14        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

Sammlungen von erfolgreichen und von Expertinnen geprüften Musterlösungen können angelegt werden und führe dazu, dass bei neu auftretenden Problemsituationen Teile – bereits durch Muster – als *im Prinzip schon gelöst* gelten können. Die Voraussetzung ist, dass man diese Muster (engl.: pattern) auch kennt.

In der Informatik wurde eine Sammlung mit der Bezeichnung *Entwurfsmuster* von (Gamma u. a. 1996) veröffentlicht. Allerdings gibt es auch Muster, die Sie bereits kennen: Standardverfahren zum Suchen und Sortieren; Möglichkeiten, einen binären Baum zu traversieren;

wird noch ergänzt



Vorhaben 6

Wissensbasierte Modellierung mit Datenbanken¹

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),

¹Teile 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 einem 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-Programme 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 werden 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 eine 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 der Festplatte gesichert.
- + *Mehrere Benutzer* können *gleichzeitig* zugreifen.
- + *Verschiedene Programme* können *gleichzeitig* zugreifen.
- + Datenbanksysteme können *sehr billig* sein.

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 *Festplatten* umgewandelt.

Welche Schichten gehören zu einer Datenbank?

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

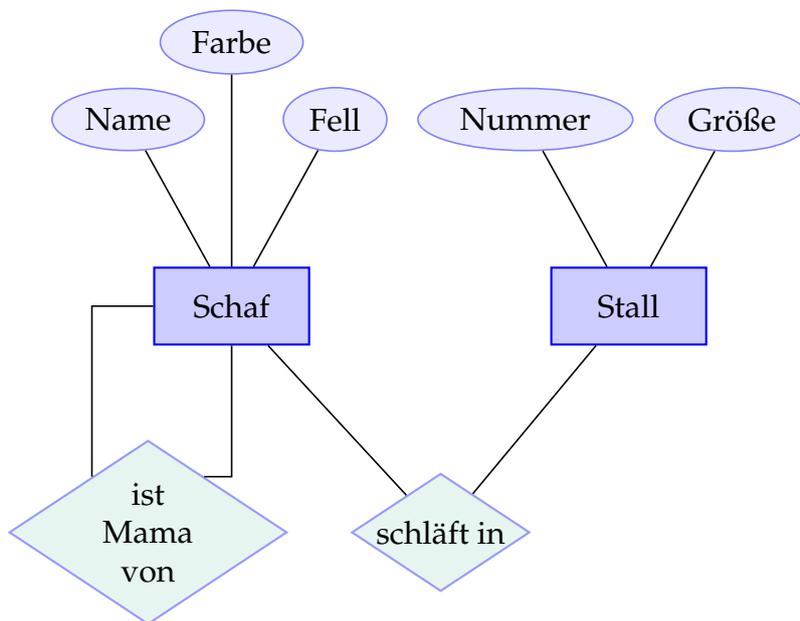
Beispiel: Farbe des Schafs, Basenposition des Gens

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

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 malt 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 L^AT_EX (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 gleich zwei Abschnitte gewidmet sind.

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- und Delete-Befehlen gefüllt und auf dem neuesten Stand gehalten. Select-Befehle verwendet man schließlich, um im laufenden Betrieb Information aus der Datenbank zu gewinnen.

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 *einzu*fügen und zu *lös*chen.
 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.

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.	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
Ich kann einen konkreten Ablauf als Sequenzdiagramm darstellen.	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
Objektorientierte Modellierung führt vom Problem zu Objekten zu Klassen und zum Programm.	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
Der Unterschied zwischen Anfrage und Auftrag ist mir klar.	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
Die Umsetzung eines Struktogramms in eine Methode und umgekehrt stellt für mich kein Problem dar.	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
Ich kann das Konzept der Rekursion an mehreren Beispielen erklären.	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
Ich kann das Konzept der von-Neumann-Maschine erklären.	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>

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

A.1.2 Eingangskompetenzen – Beispielmodellierung

Name:

Datum:

WM-Finale

Lesen Sie die folgende Situationsbeschreibung¹ 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.

¹Zum 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.



Anhang B

Hinweise

1.949 Berücksichtigen Sie die Elemente aus dem letzten Abschnitt, um die Aufgabe zu erledigen. [Zurück](#)

Anhang C

Rezepte

C.1 Zahl_x → Dezimalzahl₁₀

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) multipliziert (vgl. erstes Beispiel).

Beispiele – Aufgabe

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

$$\bullet \ 11101010_2 \rightarrow 2 \begin{array}{r} 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \\ 2 \ 6 \ 14 \ 28 \\ \hline 1 \ 3 \ 7 \ 14 \end{array}$$

Tragen Sie die fehlenden Zahlen ein und ermitteln Sie das Gesamtergebnis. Das Ergebnis ist der Dezimalwert von 11101010₂.

$$\bullet \quad 7502_8 \rightarrow 8 \quad \begin{array}{r|rrrr} 7 & 5 & 0 & 2 \\ & 56 & 488 & 3904 \\ \hline & 7 & 61 & 488 & \boxed{3906} \end{array}$$

$$\bullet \quad 3A5F_{16} \rightarrow 16 \quad \begin{array}{r|rrrr} 3 & 10 & 5 & 15 \\ & 48 & 928 & 14928 \\ \hline & 3 & 58 & 933 & \boxed{14943} \end{array}$$

$$\bullet \quad 11001_2 \rightarrow 2 \quad \begin{array}{r|rrrrr} 1 & 1 & 0 & 0 & 1 \\ & 2 & 6 & 12 & 24 \\ \hline & 1 & 3 & 6 & 12 & \boxed{25} \end{array}$$

C.2 Dezimalzahl₁₀ → Zahl_x

Divisionsverfahren – eine Dezimalzahl in ein anderes Zahlensystem umwandeln

Ziel

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



Rezept

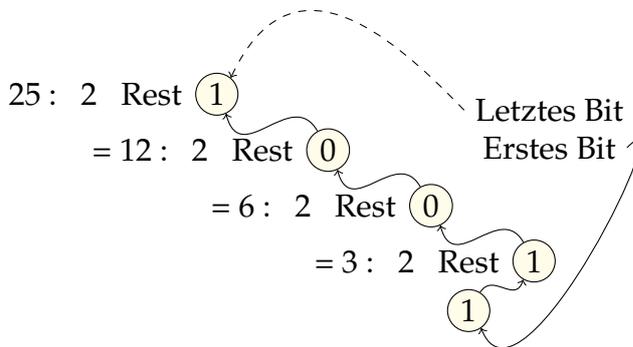
Regeln zur Umrechnung einer Dezimalzahl

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:

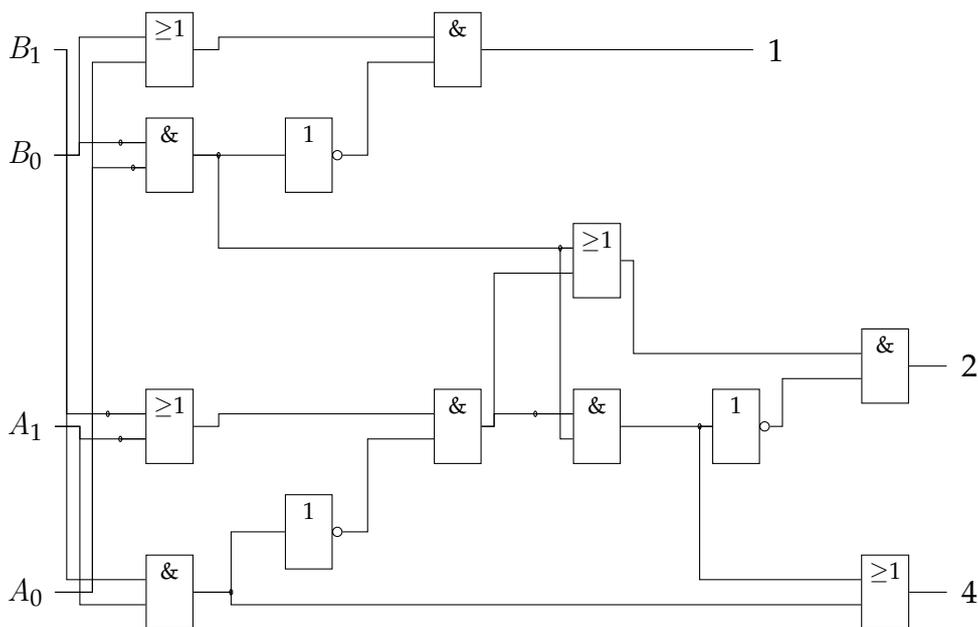
- 0 führt zum einfachen Hinstellen bei dem Schaltelement,



- 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/p01m5xf8> veröffentlicht am 25. November 2013 – geprüft: 13. Dezember 2014.

C.4 Schlüsselworte in Python3

In Python3 festgelegten Schlüsselworte anzeigen

Ziel

Reservierte Worte in Python3.



Rezept

In der Python3-Konsole können die Schlüsselwörter abgefragt werden:

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

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

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

Schlüsselwörter 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:

```
1 >>> 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
- 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:

```
1 >>> from random import randint
2 >>> 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.



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).
- Pieper, Johannes und Dorothee Müller, Hrsg. (2014). *Material für den Informatikunterricht*. Arnsberg, Dortmund, Hamm, Solingen, Wuppertal. URL: <http://ddi.uni-wuppertal.de/material/materialsammlung/> (besucht am 23.03.2016).