Vorlesung

8

Aggregierte Datentypen

Inhalt

Vorlesung

7

Lernziele: Aggregierte Datentypen sind sehr wichtige Elemente des Programmierens. Gerade Python bietet hier mächtige Konstrukte an. Sie sollen diese Konstrukte und ihre Eigenarten verstehen lernen.

***Beachten Sie:** Programmieren erfordert Disziplin, Ausdauer, abstraktes Denkvermögen, Kreativität und hohe Lernbereitschaft! Auch wenn es sehr viele Details sind – durch Verstehen und Ausprobieren lernt man sie.*

1 Aggregierte Datentypen in Python

Python stellt sehr mächtige **Kollektionen** von Datentypen zur Verfügung. Die folgende Abbildung gibt einen Überblick: Dabei sind `set`, `list` und `dictionary` veränderlich (mutable) alle anderen `frozenset`, `string`, `ustring` und `tupel` nichtveränderlich (unmutable). Die String-Datentypen wurden schon in Vorlesung 1 vorgestellt.

Grundsätzlich gilt in Python, dass Kollektionen beliebig ineinander verschachtelt werden können.

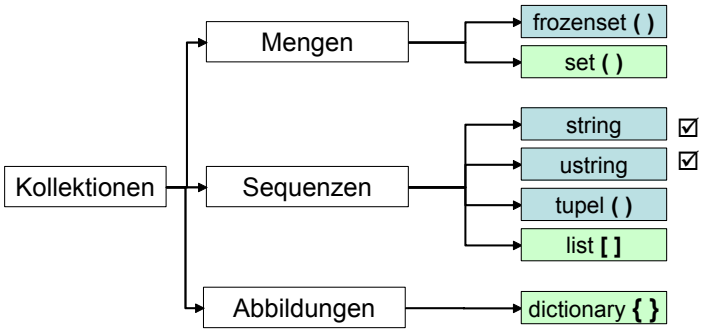


Abbildung 1 Aggregierte Datentypen in Python

1.1 Sequenzdatentypen: Liste – Tupel – String

Möchte man mehrere Objekte zusammenfassen, so verwendet man eine Liste, ein Tupel oder einen String. **Listen** sind **veränderliche** (*mutable*) Felder von Objektreferenzen auf die numerisch aus der Menge $\mathbf{N}_0$ (also von 0 .. n-1 bei n Objekten) indiziert (also per Zähler) zugegriffen werden kann – **Tupel** ist die **unveränderliche** (*unmutable*) Variante davon. **Strings** und ustrings sind Tupel aus Zeichen. Die Spezialisierung erlaubt es, diverse zusätzliche Operationen (Methoden), z.B. zur Formatierung anzubieten. Die Grundoperationen sind aber die Selben!

Tupel, Listen und Strings sind also Zusammenstellungen von Objekten, die eine Reihenfolge bewahren (im Gegensatz zu Mengen, siehe unten). Man nennt sie in Python **Sequenzdatentypen**.

1.1.1 Erzeugung von Literalen und einfache Zugriffe

Listen werden in **eckigen** Klammern notiert wie z.B. [x , y, z]; die Werte darin sind durch Komma getrennt.

[ ]	eine leere Liste
liste = [1, 2, 3]	eine Liste mit drei Integer-Elementen
liste[1]	holt das Element mit Index=2
liste = ['spam', [1, 2,3], 3.141]	eine verschachtelte Liste: liste [1][0] holt das Element 1
liste = list('spam')	erzeugt eine Liste durch Aufruf des Typkonstruktors

Tupel enthält ebenfalls durch Komma getrennte Werte, aber in **runden** Klammern: (x , y, z). Die umschließenden Klammern können manchmal weggelassen werden, z.B. in Schleifenköpfen oder einfachen Zuweisungen. Die Erzeugung ist sehr ähnlich zu Listen, nur mit runden Klammern:

( )	ein leeres Tupel
tuple = (1, 2, 3)	ein Tupel mit drei Integer-Elementen
tuple(1)	holt das Element 2
Tuple = ('spam', (1, 2,3), 3.141)	eine verschachteltes Tupel: tupel [1][0] holt das Element 1
tuple = tuple('spam')	erzeugt ein Tupel durch Aufruf des Typkonstruktors

Sonderschreibweisen:

tuple = 1, 2, 3	das gleiche Tupel wie in anderer Schreibweise, ist aber in Funktionsaufrufen nicht erlaubt, weil nicht zwischen Parametern und Tupel-Elementen unterschieden werden kann.
(1,)	ein Tupel mit einem Element und kein geklammerter Ausdruck. Dies wird durch das Komma deutlich gemacht.

Die Anweisung `t = 12345, 54321, 'hello!'` ist ein Beispiel für das so genannte Tupel-Einpacken (engl. *tuple packing*): die Werte 12345, 54321 und 'hello!' werden in ein Tupel eingepackt. Die umgekehrte Operation ist auch möglich, z.B.:

```
>>> x, y, z = t
```

Dies nennt man, sinnigerweise, Tupel-Auspacken (engl. *tuple unpacking*). Das Auspacken von Tupeln verlangt, dass die Liste der Variablen auf der linken Seite die gleiche Anzahl von Elementen hat wie das Tupel selbst. Man beachte, dass Mehrfachzuweisungen lediglich eine Kombination von Ein- und Auspacken von Tupeln ist!

Gelegentlich ist die entsprechende Operation auf Listen nützlich: Listen-Auspacken. Sie wird unterstützt, indem die Variablenliste in eckige Klammern gesetzt wird:

```
>>> a = ['spam', 'eggs', 100, 1234]

>>> [a1, a2, a3, a4] = a
```

Strings werden in Anführungszeichen notiert: 'xyz' oder "xyz" oder " " "xyz" " ", siehe Vorlesungskapitel 1.

1.1.2 Sequenz-Operationen

Strikt einheitlich können in Python die folgenden Operationen auf Strings, Listen und Tupel ausgeführt werden (S und T seien in folgender Tabelle Strings oder Listen oder Tupel)

Operationen auf Sequenz-datentypen	Beschreibung	Klassenmethoden
`X in S` `X not in S`	Test auf Enthaltensein	`__contains__`
`S + T`	Verkettung	`__add__`
`S*i` `i*S`	Wiederholung N ist ein Integer	`__mul__`
`S[i]`	Indizierung mit Zahl	`__getitem__`
`S[i:j]`	Teilbereich (Slicing)	`__getslice__` `__getitem__`
`len(S)`	Länge	`__le__`
`min(S)`	minimales Element	
`max(S)`	maximales Element	
`iter(S)`	Iterator-Objekt	`__iter__`
`for X in S`	Iteration	`__getitem__` `__iter__`

Als veränderlicher Datentyp sind auf **Listen** auch folgende Operationen erlaubt:

Operationen auf Listen	Beschreibung	Klassenmethoden
`S[i] = x`	Zuweisung an das i-te Element	`__setitem__`
`S[i,j,k] = T`	Teilbereichszuweisung, S wird von i bis j durch T ersetzt, mit einer optionalen Schrittweite k	`__setslice__` `__setitem__`
`del S[i]`	Indizierten Eintrag löschen	`__delitem__`
`del S[i,j,k]`	Teilbereichslöschung, sonst wie oben	`__delslice__` `__delitem__`
`del S`	löscht S gänzlich, eine spätere Referenzierung von S ist ein Fehler	

Methoden auf Listen	Beschreibung	Klassenmethoden
`S.insert(i,x)`	Fügt ein Element x an einer bestimmten Stelle i ein. Das erste Argument ist der Index, vor dem eingefügt werden soll.	
`S.append(T)`	Hängt die Liste T an S an: äquivalent zu S.insert(len(S), T)	
`S.index(x)`	Gib den Index des ersten Elements in der Liste zurück, dessen Wert gleich x ist. Falls kein solcher existiert, so ist dies ein Fehler.	
`S.remove(x)`	Entferne das erste Element der Liste, dessen Wert x ist. Falls kein solches existiert, so ist dies ein Fehler.	
`S.sort()`	Sortiere die Elemente der Liste, in der Liste selbst.	
`S.reverse()`	Invertiere die Reihenfolge der Listenelemente, in dieser selbst.	
`S.count(x)`	Gib die Anzahl des Auftretens des Elements x in der Liste zurück.	
`S.pop([i])`	Entfernt Element an Position i undgibt diesen zurück (default (ohne Angabe von i = letzte Element	

Darüber hinaus unterstützt Python noch **Vergleichsoperatoren** für Sequenzen: Sequenz-Objekte dürfen mit anderen Objekten vom gleichen Sequenz-Typ verglichen werden. Der Vergleich basiert auf lexikographischer Ordnung: zuerst werden die ersten beiden Elemente verglichen, und wenn sie sich unterscheiden, bestimmt dies bereits das Resultat. Wenn sie gleich sind, werden die nächsten beiden Elemente verglichen, und so weiter, bis eine der beiden Sequenzen erschöpft ist. Wenn zwei zu vergleichende Elemente selbst Sequenzen vom gleichen Typ sind, wird der lexikographische Vergleich rekursiv fortgesetzt. Falls alle Elemente einer Sequenz gleich sind, werden die Sequenzen als gleich betrachtet. Falls eine Sequenz eine Anfangssequenz der anderen ist, ist die gekürzte Sequenz die kleinere. Die lexikographische Ordnung für Strings verwendet die ASCII-Ordnung für einzelne Zeichen. Einige Beispiele für Vergleiche von Sequenzen desselben Typs:

```
(1, 2, 3) < (1, 2, 4)
[1, 2, 3] < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4) < (1, 2, 4)
(1, 2) < (1, 2, -1)
(1, 2, 3) == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)
```

Man beachte, dass es erlaubt ist, Objekte verschiedenen Typs zu vergleichen. Das Resultat ist deterministisch, aber beliebig: die Typen sind nach ihrem Namen geordnet. Daher ist eine Liste immer kleiner als ein String, ein String immer kleiner als ein Tupel, etc. Ge-

mischte numerische Typen werden nach ihrem numerischen Wert verglichen, d.h. 0 ist gleich 0.0, etc. [1]

1.2 Set – Frozenset

Mengenobjekte werden mit dem eingebauten Funktionen `set` (änderbar) oder `frozenset` (unveränderlich) erzeugt

```
>>> x = set('abc')
>>> y = set('bcde')
>>> x
{'a', 'c', 'b'}
>>> 'b' in x
True
>>> x & y    # Schnittmenge
{'c', 'b'}
>>>
```

Operationen und Methoden auf Mengenobjekten sind:

is ∈	x in M	(True/False)	
		M.issubset(N)	N istTeilmenge von N (True/False)
		M.isssuperset(N)	N ist Obermenge von N (True/False)
∪	M|N	M.union(N)	Vereinigung von M und N
	M|=N	M.update(N)	
∩	M&N	M.intersection(N)	Schnittmenge von M und N
	M&=N	M.intersection_update(N)	
\	M-N	M.difference(N)	Differenz von M und N
	M-=N	M.difference_update(N)	
Δ		M.symmetric_difference(N)	Symmetrische Differenz von M und N
		M.symmetric_difference_update(N)	
		M.add(x)	Füge ein Element x zur Menge M hinzu. (Hat keine Wirkung, wenn x schon Element von M ist)
		M.clear()	Erzeugt die Leere Menge M
		M.pop()	Entfernt ein Element von der Menge M
		M.remove(x)	Entferne Element x von der Menge M (x muss ein Element von M sein, sonst Key error)

1.3 Dictionaries

Dictionaries (zu Deutsch: *Wörterbücher*) sind in Python ein eingebauter Datentyp. Im Gegensatz zu Sequenzen, die mit einem Zahlen-Intervall indiziert werden, werden Dictionaries über Schlüssel indiziert, die irgendeinen unveränderlichen Typ haben können. Strings und Zahlen

[1] Man sollte sich **nicht** auf diese Regeln verlassen, da sie sich in einer zukünftigen Version der Sprache ändern könnten.

können immer solche Schlüssel sein. Tupel können als Schlüssel verwendet werden, wenn sie nur Strings, Zahlen oder Tupel enthalten. Listen können nicht als Schlüssel verwendet werden. Am besten stellt man sich Dictionaries als eine ungeordnete Menge von Schlüssel:Wert-Paaren vor, unter der Randbedingung, dass die Schlüssel eindeutig sein müssen (innerhalb eines Dictionaries). Ein Paar geschweiften Klammern erzeugt ein leeres Dictionary {}. Die initialen Schlüssel:Wert-Paare in dem Dictionary werden mit einer durch Kommata getrennte Liste von Schlüssel:Wert-Paaren innerhalb der Klammern angegeben. In diesem Format werden Dictionaries auch ausgegeben.

Dictionaries implementieren sog. partielle Funktionen: „Eine partielle Funktion ist eine rechtseindeutige Relation R. (Hier ist R ist eine Menge von n-Tupeln). Objekte, die in der Relation R zueinander stehen, bilden ein n-Tupel, das Element von R ist."

In Python benutzt man Zweier-Tupel der Form (Schlüssel, Wert), geschrieben zum Beispiel als {Schlüssel:Wert}. „Rechtseindeutig" bedeutet, dass es für jeden Schlüssel nur einen eindeutigen Wert gibt. Da der Wert wieder ein Tupel sein kann, aber auch eine Liste, etc. sind beliebigen partielle Funktionen implementierbar.

Die Hauptoperationen auf einem Dictionary sind das Speichern eines Wertes unter einem Schlüssel und das Abrufen dieses Wertes bei Angabe des Schlüssels. Es ist auch möglich, ein Schlüssel:Wert-Paar mit `del` zu löschen. Wird beim Speichern ein Schlüssel-Wert-Paar verwendet, bei dem der Schlüssel bereits existiert, so wird der alte damit assoziierte Wert überschrieben. Es ist ein Fehler, einen Wert mit einem nicht existierenden Schlüssel abzurufen.

Die `keys()`-Methode eines Dictionary-Objektes gibt eine Liste aller in dem Dictionary verwendeten Schlüssel zurück in zufälliger Reihenfolge. Brauchen Sie sie sortiert, dann wenden Sie einfach die `sort()`-Methode auf die Liste der Schlüssel an. Um zu testen, ob ein einzelner Schlüssel in einem Dictionary vorkommt, verwende man die has `key()`-Methode eines Dictionaries.

Operationen auf Dictionaries sind:

```
D[k]                Indizierung mit Schlüssel k
D[k]=X              Wertzuweisung über Schlüssel k
del D[k]            Lösche Eintrag über Schlüssel k
len (D)             Länge (Anzahl der Einträge)
k in D D.has_key(k) Test auf Enthaltensein des Schlüssels k
k not in D
```

```
D.clear()            Löscht alle Einträge in D
D.copy()             Ergibt eine flache Kopie von D
D.get(k[,d])         liefert D[k], wenn k in D, sonst d (wenn angegeben)
oder None .
D.items()            Liste von D's (key, value)-Paaren, als 2-Tuple
dictD.iteritems()    -> an iterator over the (key, value) items of D
D.iterkeys()         -> an iterator over the keys of D
D.itervalues()       -> an iterator over the values of D
D.keys()             -> list of D's keys
D.pop(k[,d]) -> v, remove specified key and return the corresponding value
D.popitem()          -> (k, v), remove and return some (key, value) pair
                     as a 2-tuple; but raise KeyError if D is empty
D.setdefault(k[,d])  -> D.get(k,d), also set D[k]=d if k not in D
D.update(E, **F)     -> None.  Update D from E and F: for k in E: D[k] =
                     E[k](if E has keys else: for (k, v) in E: D[k] = v)
                     then: for k in F: D[k] = F[k]
D.values()           -> list of D's values
```

1.4 Iteration und weitere Programmierhinweise

Wir haben die Python `for`-Schleife und ihre Möglichkeiten zur Iteration über beliebige Sequenzdatentypen schon kennen gelernt.

Es gibt diverse weitere „Werkzeuge“ aus der funktionalen Programmierung, die in Kombination mit Listen sehr nützlich sind:

- `filter()`
- `map()`
- `reduce()`

`filter``(function, sequence)` gibt eine Sequenz (wenn möglich, desselben Typs) zurück, die aus den Elementen der Sequenz besteht, für die `function`(item) wahr ist, z.B. um Primzahlen zu berechnen:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

`map``(function, sequence)` ruft `function`(`item`) für jedes Listenelement auf und gibt eine Liste der zurückgegebenen Werte zurück. Zum Beispiel könnte man Kubikzahlen wie folgt berechnen:

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Man darf mehr als eine Sequenz übergeben; die Funktion muss dann genauso viele Argumente haben wie es Sequenzen gibt, und sie wird mit dem entsprechenden Element der jeweiligen Sequenz (oder `None`, falls eine Sequenz kürzer als die andere ist) aufgerufen. Falls `None` als Funktion übergeben wird, wird es mit einer Funktion ersetzt, die ihr(e) Argument(e) zurückgibt. Kombiniert man diese zwei Spezialfälle, so sehen wir, dass `` `map(None, list1 , list2 )' `` eine bequeme Art ist, ein Paar von Listen in eine Liste von Paaren umzuwandeln, z.B.:

```
>>> seq = range(8)
>>> def square(x): return x*x
```

```
...
>>> map(None, seq, map(square, seq))
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7,
49)]
```

reduce(func, sequence) gibt einen einzigen Wert zurück, der sich ergibt, wenn man die zweistellige Funktion func auf die ersten zwei Elemente der Sequenz, dann auf das Resultat und das nächste Element, etc. anwendet. Um etwa die Summe der Zahlen von 1 bis 10 zu berechnen:

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
```

Falls die Sequenz nur ein Element hat, wird dieses eine zurück gegeben. Falls sie leer ist, wird eine Ausnahme ausgelöst. Ein drittes Argument kann übergeben werden, um einen Startwert anzugeben. In diesem Fall wird der Startwert bei einer leeren Sequenz zurück gegeben. Die Funktion wird dann auf den Startwert und das erste Element erstmalig angewendet, dann auf das Resultat und das nächste Element, u.s.w. Beispiel:

```
>>> def sum(seq):
... def add(x,y): return x+y
... return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

1.5 Vertiefung: Referenzen und Kopien

Alle Daten eines Python-Programmes basieren auf dem Konzept eines Objektes. Objekte umfassen grundlegende Datentypen wie Zahlen, Zeichenketten, Listen und Dictionaries. Es ist auch möglich, benutzerdefinierte Datentypen in Form von Klassen oder Erweiterungstypen zu erstellen.

Jedes Datum im Speicher ist ein *Objekt*. Jedes Objekt hat eine Identität, einen Typ und einen Wert. Der *Typ* eines Objektes (der seinerseits eine spezielle Art von Objekt ist) beschreibt die interne Repräsentation des Objektes wie auch die Methoden und Operationen, die es unterstützt. Wird ein Objekt eines bestimmten Typs erzeugt, so wird dieses Objekt manchmal als eine *Instanz* dieses Typs bezeichnet (obwohl eine Instanz eines Typs nicht mit einer Instanz einer benutzerdefinierten Klasse verwechselt werden sollte). **Nachdem ein Objekt erzeugt wurde, können dessen Identität und Typ nicht mehr verändert werden.** Falls dessen Wert jedoch verändert werden kann, so sagt man, das Objekt ist *veränderlich*. Falls der Wert nicht verändert werden kann, so spricht man entsprechend von einem *unveränderlichen* Objekt.

Ein Objekt, das Verweise auf andere Objekte enthält, bezeichnet man als **Container** oder Sammlung. Zusätzlich zum Wert, den sie repräsentieren, definieren viele Objekte eine Anzahl von Datenattributen und Methoden. Ein Attribut ist eine mit dem Objekt assoziierte Eigenschaft oder ein Wert. Eine *Methode* ist eine Funktion, die eine gewisse Operation auf einem Objekt ausführt, sobald sie angestoßen wird. Attribute und Methoden werden mit dem Punkt-Operator (.) beim Objekt angesprochen, für das sie definiert sind.

Für alle Objekte gilt, dass alle Referenzen auf sie gezählt werden. Der Referenzzähler eines Objektes wird immer dann erhöht, wenn das Objekt einem Namen zugewiesen oder in einen Container gelegt wird, wie z.B. in Listen, Tupeln oder Dictionaries:

```
a = 3.4           # Erzeugt ein Objekt '3.4'.
b = a             # Erhöht Referenzzähler von '3.4'.
c = []
c.append(b)       # Erhöht Referenzzähler von '3.4'.
```

In diesem Beispiel wird ein einzelnes Objekt mit dem Wert 3.4 erzeugt. `a` ist lediglich ein Name, der das gerade erzeugte Objekt referenziert. Wenn `a` an `b` zugewiesen wird, dann wird `b` zu einem neuen Namen für dasselbe Objekt und der Referenzzähler des Objektes wird um eins erhöht. Genauso erhöht sich der Referenzzähler dann, wenn `b` an eine Liste angefügt wird. Im gesamten Beispiel gibt es genau ein Objekt mit dem Wert 3.4. Alle anderen Operationen erzeugen lediglich neue Namen für dasselbe Objekt.

Der Referenzzähler eines Objektes verringert sich bei Benutzung der Anweisung `del` oder dann, wenn eine lokale Referenz den Gültigkeitsbereich verlässt (oder neu zugewiesen wird). Beispiel:

```
del a             # Verringere Referenzzähler von '3.4'.
b = 7.8           # Verringere Referenzzähler von '3.4'.
c[0] = 2.0        # Verringere Referenzzähler von '3.4'.
```

Sobald der Referenzzähler eines Objektes auf Null sinkt, wird es speicherbereinigt. In einigen Fällen jedoch kann es zu zirkulären Abhängigkeiten zwischen einer Menge von Objekten kommen, die nicht mehr in Gebrauch sind. Beispiel:

```
a = {}
b = {}
a['b'] = b        # a enthält Referenz auf b.
b['a'] = a        # b enthält Referenz auf a.
del a
del b
```

In diesem Beispiel verringern die `del`-Anweisungen die Referenzzähler von `a` und `b` und vernichten damit die Namen, die auf die entsprechenden Objekte zeigen. Da aber jedes Objekt eine Referenz auf das jeweils andere hat, wird der Referenzzähler nicht Null und die Objekte damit nicht speicherbereinigt. Als Ergebnis bleiben die Objekte im Speicher alloziiert, obwohl der Interpreter keine Möglichkeit mehr hat, darauf zuzugreifen, d.h. die Namen für den Zugriff darauf sind verschwunden.

Wenn ein Programm eine Zuweisung wie in `a = b` vornimmt, wird eine neue Referenz auf `b` erzeugt. Für einfache Objekte wie Zahlen und Strings erzeugt diese Zuweisung eine Kopie von `b`. Für veränderliche Objekte wie Listen und Dictionaries ist dieses Verhalten jedoch gänzlich verschieden:

```
b = [1, 2, 3, 4]
a = b             # a ist eine Referenz auf b.
a[2] = -100       # Ändere ein Element in 'a'.
print(b)          # Ergibt '[1, 2, -100, 4]'.
```

Da a und b in diesem Beispiel dasselbe Objekt referenzieren, macht sich eine Änderung der einen Variablen bei der anderen bemerkbar. Um dies zu vermeiden, muss man eine Kopie des Objektes anfertigen und nicht nur eine neue Referenz darauf.

Es gibt zwei verschiedene Möglichkeiten, Objekte wie Listen und Dictionaries zu kopieren: eine **flache und eine tiefe Kopie**. Eine flache Kopie erzeugt ein neues Objekt, füllt es jedoch mit Referenzen auf die Elemente des ursprünglichen Objektes. Beispiel:

```
b = [1, 2, [3, 4]]
a = b[:]           # Erzeuge eine flache Kopie von b.
a.append(100)      # Füge Element an a hinzu.
print(b)           # Ergibt '[1, 2, [3, 4]]'. b unverändert.
a[2][0] = -100     # Ändere ein Element von a.
print b            # Ergibt '[1, 2, [-100, 4]]'.
```

In diesem Fall sind `a` und `b` eigenständige Listenobjekte, aber beide teilen sich die Elemente darin. Daher wird bei jeder Änderung der Elemente von `a` auch ein Element in `b` verändert, wie man sieht.
Eine *tiefe Kopie* erzeugt ein neues Objekt und kopiert rekursiv alle darin befindlichen Objekte. Es gibt keine eingebaute Funktion, um tiefe Kopien von Objekten anzufertigen, aber die Funktion `copy.deepcopy()` aus der Standardbibliothek kann dazu wie folgt benutzt werden:

```
import copy
b = [1, 2, [3, 4]]
a = copy.deepcopy(b)
```

2 Übersicht: Aggregierte Datentypen

Allgemein sind zusammengesetzte (aggregierte) Datentypen Konstrukte, die aus einfacheren (elementaren) Datentypen (oder zusammengesetzten Datentypen) bestehen. Der Wert eines aggregierten Datentyps besteht also allgemein aus Komponenten, deren Datentyp elementar ist oder wiederum aggregiert sein kann.

Eine **Abgrenzung** des Begriffs „Datentyp" zu dem Begriff „Datenstruktur" ist schwierig und in der Fachsprache nicht einheitlich.. Allenfalls könnte man folgendes definieren:

Datenstruktur: Der Fokus der Diskussion liegt auf der Strukturierung/(der Kodierung und Anordnung) der gespeicherten Daten

Aggregierter Datentyp: Der Fokus der Diskussion liegt auf den Operationen (im Sinne eines abstrakten Datentyps).

Diese Unterscheidungen sind meistens leider nicht klar zu treffen; wir benutzen die Begriffe aggregierter / zusammengesetzter Datentyp und Datenstruktur synonym, zumal bei vielen Operationen der Ressourcenbedarf, also sowohl die **benötigte Laufzeit** für die Ausführung einer Operation als auch der **Speicherplatzbedarf** voneinander abhängen und deshalb eine Unterscheidung nicht ins Auge springt.

Die Definition von Datenstrukturen erfolgt im Allgemeinen durch die Angabe einer konkreten Spezifikation zur Datenhaltung und der Operationen an der Schnittstelle. Oft impliziert eine Datenstruktur auch gewisse Eigenschaften. Diese konkrete Spezifikation legt das allgemeine Verhalten der Operationen fest und abstrahiert damit von der konkreten Implementation der Datenstruktur.

Von den meisten Datenstrukturen gibt es neben ihrer Grundform viele Spezialisierungen, die eigens für die Erfüllung ganz bestimmter Aufgaben spezifiziert wurden.

Aggregierte Datentypen stehen in den meisten Programmiersprachen zur Verfügung. Programmiersprachen können aggregierte Datentypen auf zwei Ebenen unterstützen:

1. Sie stellt **Konstruktoren** zur Verfügung, mit deren Hilfe der Aufbau eines aggregierten Datentyps durch den Programmierer beschrieben werden kann.

2. Sie verfügen über **vordefinierte aggregierte Datentypen** (builtins).
   Einfache, uns schon bekannte Beispiele für aggregierte Datentypen sind Zeichenketten und komplexe Zahlen.

2.1 Bottom-up Strukturen – Array, Records

Je nach Ausgangspunkt einer Betrachtung aggregierter Datentypen kommt man durchaus zu verschiedenen Ergebnissen: Folgt man der Entwicklung der klassischen imperativen Programmierung, so war es zunächst das Ziel, mehrere elementare Daten unter **einem Namen** anzusprechen. Diese Betrachtungsweise ist gewissermaßen bottom-up. Wir unterscheiden dann folgende Konstruktoren, die bei statischem Typing bei der Deklaration benutzt werden:

- **ARRAY** (Reihung, Feld, Tabelle)
  - alle Komponenten (Elemente) eines Arrays besitzen stets den gleichen Datentyp, meist einen primitiven Datentyp
  - eine Komponente wird über einen Index bzw. eine Menge von Indizes angesprochen:
    ```
    array_name[index] bzw.           # eindimensional
    array_name[index_1]...[index_n] # n-dimensional
    ```
  - eindimensionale Arrays nennt man häufig auch Vektor
    Typische Operationen auf einem Vektor A sind
    A[i] gibt den Wert des i-ten Elements
    A[i] = x Zuweisung an das i-te Element
    Bei höherdimensionalen Arrays sind mehrere Indexwerte erlaubt (im folgenden Beispiel 2-dimensional)
    A[i,j] gibt den Wert des (i,j)-ten Elements
    A[i,j] = x Zuweisung an das (i,j)-te Element

- **RECORD** (Verbund, Struct, Variant)
  - die Komponenten eines Records können unterschiedliche Datentypen besitzen, auch wiederum zusammengesetzte
  - jede Komponente besitzt einen eigenen Namen. Angesprochen wird eine Komponente in folgender Weise:
    `record_name.komponenten_name` oder bei mehrfach zusammengesetzten `record_name.komponenten_name1.komponenten_name` 2

Eigenschaften dieser Aggregate sind:
(1) Alle Komponenten eines Werts des aggregierten Datentyps sind uneingeschränkt sichtbar.
(2) Alle Operationen, die gemäß des Datentyps einer Komponente in Frage kommen, sind auf diese Komponente uneingeschränkt anwendbar.

Struktur	Ordnung	Einzigartigkeit der Werte der Komponenten (Elemente)	Werte pro Komponente (Element)
Array	ja	nein	1
Record	nein	nein	n

"Ordnung" meint hier, dass die Reihenfolge der Eingabe (des Einfügens) bewahrt wird, respektive, dass ein Element an einer bestimmten Stelle eingeordnet werden kann, aber **nicht,** dass die Elemente geordnet nach ihren Werten vorliegen.

2.2 Zeiger (Pointer)

In einigen Programmiersprachen kam zu diesen Basis-Konstrukten noch das Konzept des Zeigers hinzu: Ein **Zeiger** (oder **Pointer**) bezeichnet eine besondere Art von Variablen, die auf einen anderen Speicherbereich verweist. Der referenzierte Speicherbereich enthält entweder Daten (Objekt, Variable) oder Programmcode (ggf. sind auch mehrfach hintereinander aufzulösende Referenzzierungen zugelassen). Array und Record sind lineare Strukturen: Durch Zeiger kann man dagegen beliebige verzweigte Strukturen realisieren.

Einerseits kann man mit Zeigern rechnen, sie inkrementieren, zuweisen, etc. Andererseits kann auf das verwiesene Element selbst zugegriffen werden. Dieses nennt man Dereferenzierung.

Zeiger kommen vor allem in maschinennahen Programmiersprachen wie z.B. Assembler, C oder C++ vor, während man den Gebrauch in streng typisierten Sprachen wie Modula-2 oder Ada stark einschränkt und sie in Sprachen wie Java, Eiffel oder Python zwar intern vorhanden, aber für den Programmierer verborgen sind (weil Programmierern bei der Arbeit mit Zeigern leicht schwerwiegende Programmierfehler unterlaufen, sicherlich die Hauptursache für „Pufferüberläufe" und „Abstürze" bei C oder C++ Programmen).
.
Pro und Cons für das Programmieren mit Zeigern:

Vorteile

- Bei der Verwendung von Feldern/Vektoren kann man mittels Zeigern schnell innerhalb des Feldes springen und navigieren. Mittels Zeigerinkrement wird dabei durch ein Feld hindurchgelaufen. Anstatt einen Index zu verwenden und so die Feldelemente über diesen anzusprechen, setzt man zu Beginn des Ablaufs einen Zeiger auf den Anfang des Feldes und inkrementiert diesen Zeiger bei jedem Durchlauf.

- Verweise auf Speicherbereiche können geändert werden, z.B. zur Sortierung von Listen, ohne die Elemente umkopieren zu müssen (dynamische Datenstrukturen).

- Mit Zeigern lassen sich Datenstrukturen wie „verkettete Listen" effektiv und (fast natürlich) realisieren.

- Bei Funktionsaufrufen kann durch die Übergabe eines Zeigers auf eine Variable vermieden werden, die Variable selbst zu übergeben, was eine in bestimmten Fällen sehr zeitaufwändige Anfertigung einer Kopie der Variablen erfordern würde.

- Anstatt Variablen jedes Mal zu kopieren und so jedes Mal erneut Speicherplatz zur Verfügung zu stellen, kann man in manchen Fällen einfach mehrere Zeiger auf ein und dieselbe Variable verweisen lassen.

Nachteile und Gefahren

- Der Umgang mit Zeigern ist relativ schwierig zu erlernen, kompliziert und fehleranfällig. Auch bei erfahrenen Programmierern kommen Flüchtigkeitsfehler im Umgang mit Zeigern noch relativ häufig vor.

- Programmierfehler bei der Arbeit mit Zeigern können schwere Folgen haben. So kommt z.B. zu Programmabstürzen, unbemerkter Beschädigung von Daten oder gar Programmteilen, Pufferüberläufen, etc.

- Die Effizienz des Prozessor-Caches leidet darunter, wenn eine Datenstruktur auf viele Speicherblöcke verweist, die im Adressraum weit auseinander liegen. Daher kann es sinnvoll sein, stattdessen Arrays zu verwenden, weil diese eine kompakte Darstellung im Speicher haben.

Viele moderne Sprachen verzichten bewusst auf den Einsatz von Zeigern, was jedoch zum Teil auf Kosten der Effizienz geht – aber „Programmiersicherheit" wird höher gewertet. In vielen Fällen bedeutet ein „effizientes" Programmieren mit Zeigern, dass man

die Implementierung einer Datenstruktur kennen muss, und das steht im Widerspruch zu den Konzepten des abstrakten Datentyps oder der objektorientierten Programmierung! Vielmehr sollen

- die Komponenten eines abstrakten Datentyps nach außen verborgen (gekapselt) werden und.
- auf die Komponenten eines abstrakten Datentyps nicht einzeln zugegriffen werden können, sondern nur mit Hilfe spezieller Methoden.

2.3 Top-down Strukturen – Mengen, Tupel, Abbildungen

Wir können die Betrachtungsweise auch anders gestalten (nicht nur ein Zusammenfassen von Daten) und uns fragen, welche Konstrukte sich in der Mathematik als besonders leistungsfähig erwiesen haben. Diese sollten wir dann in einer Programmiersprache geeignet nachbilden und als **vordefinierte aggregierte Datentypen** (built-in), bereitstellen. Siehe hierzu insbesondere den Anhang.

Wir finden als Programmkonstrukte:

- **Mengen:** *„Eine Menge ist eine Zusammenfassung bestimmter, wohlunterschiedener Objekte unsere Anschauung oder unseres Denkens zu einem Ganzen. Diese Objekte heißen Elemente der Menge.“*
  Charakteristische Operationen sind:
  is $\in$ $\cup$ $\cap$ $\setminus$ Δ

- **Tupel:** *Ein n-**Tupel** ist eine **geordnete** Zusammenstellung von Objekten* (im Gegensatz zu Mengen, deren Elemente keine Reihenfolge bewahren).
  Charakteristische Operationen sind:

T[i]	Wert des i-ten Elements
T[i] = x	Zuweisung an das i-te Element

  wenn es veränderlich lange Tupel sind auch:

S + T	Verkettung des Tupels S und T
T.append(x)	Füge Objekt x am Ende des Tupels T an
T.count(x)	

  Der Zugriff erfolgt über einen Indexwert vom Typ Integer in der Regel im Intervall [0,n-1] bei n Elementen.

T[i,j]	Wert des (i,j)-ten Elements
T[i,j] = x	Zuweisung an das (i,j)-te Element

  In der Regel ist eine Schachtelung der Indexwerte erlaubt

- **Funktionen/Abbildungen, partiell**: „Eine partielle Funkftion ist eine rechtseindeutige Relation R. R ist eine Menge von *n*-Tupeln. Objekte, die in der Relation *R* zueinander stehen, bilden ein *n*-Tupel, das Element von *R* ist.“

  Hierzu benutzt man 2-Tupel der Form (Schlüssel, Wert), geschrieben zum Beispiel als {Schlüssel:Wert}

D[k]	Indizierung mit Schlüssel *k*
D[k]=X	Wertzuweisung über Schlüssel *k*

del D[k]	Lösche Eintrag über Schlüssel *k*
len (D)	Länge (Anzahl der Einträge)
k in D D.has_key (k)	Test auf Enthaltensein des Schlüssels *k*
k not in D	
D.items()	Liste von D's (key, value)-Paaren, als 2-Tuple
D.keys()	Liste aller Keys

Struktur	**Ordnung**	**Einzigartigkeit der Elemente**	**Werte pro Element**
Set = Menge	nein	ja	1
List = Liste (Tupel)[2]	ja	nein	1
Map (Dictionary) = Funktion / Abbildung	nein	ja (für das erste Element: sichert Rechtseindeutigkeit, also die Eigenschaft, eine partielle Funktion zu sein)	2

Wie oben: "Ordnung" meint hier nicht, dass die Elemente geordnet nach ihren Werten vorliegen, sondern, das die Reihenfolge der Eingabe bewahrt wird, respektive, dass ein Element an bestimmten Stellen eingeordnet werden kann.

> Die Operation, z.B. eine Liste gemäß ihrer Werte und der darauf definierten Ordnung zu ordnen, ist etwas gänzlich anderes, was im Fachjargon „Sortieren" genannt wird. Hierzu bemerkt Donald Knuth allerdings: *[Knuth98, 3:1, Chap. 5]*
>
> *"... that this operation might be called "order". In standard English, "to sort" means to arrange by kind or to classify. The term "sort" came to be used in Computer Science because the earliest automated ordering procedures used punched card machines, which classified cards by their holes, to implement radix sort."*

In Python sind genau diese wichtigen Grundstrukturen als *builtins* realisiert. Gegenüber den bottom-up Strukturen geht hiermit einher, dass gleichzeitig mächtige Operatoren zur Verfügung stehen (und nicht nur „Einfügen" und „Auslesen").

Eine wichtige Grundfunktion für alle genannten Datenstrukturen ist die Iteration über alle Elemente oder über Teilbereiche, die häufig mittels Zählschleifen realisiert wird. Für statische (also während der Programmlaufzeit konstant große Aggregate) ist das trivial, hier kennt man die Grenzen. Insbesondere aber für alle dynamischen Strukturen, also solche, deren Größe sich während der Laufzeit ändern kann, ist dieses alles andere als trivial: Hier müssen die Programmiersprachen geeignete Konstrukte anbieten.

Zu den genannten Grundstrukturen kommen noch diverse spezielle Strukturen, die für die Informatik von hoher Bedeutung sind, namentlich:

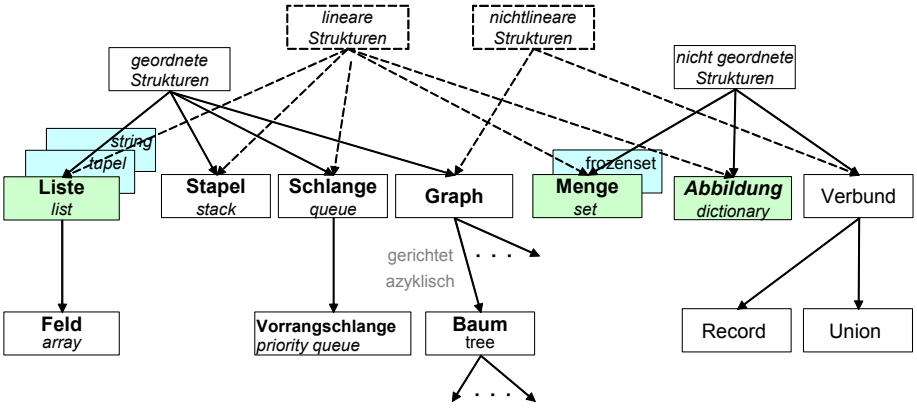
---

[2] In der Informatik wird anstelle von Tupel in der Regel der Begriff **Liste**. benutzt Python benutzt allerdings für unveränderliche Listen den Begriff Tupel. Tatsächlich kommt man damit einer häufig unterstellten Vorstellung nach (die nach der Definition allerdings nicht zwingend ist), dass die Anzahl der Elemente in einem n-Tupel zwar beliebig, aber fest ist.

Häufig trifft man den Begriff Tupel auch im Kontext von Datenbanksystemen an und zwar im Sinne von „Datensätzen", durch die einer endlichen Menge von Feldnamen Werte zugeordnet sind.

- Stapelspeicher (stacks)
- (Warte-)Schlangen (queues)
- Graphen und Baumstrukturen.

Die folgende Abbildung gibt einen ersten Überblick:



Die farblich markierten Strukturen sind solche, die von Python als *builtins* unterstützt werden, also sowohl geordnete Strukturen (solche, die eine Reihenfolge der Eingabe bewahrt) und nicht geordnete Strukturen : Grün: mutable – Blau: unmutable in Python.

Die weitere wichtige Unterscheidung ist lineare vs. nichtlineare Struktur: In linearen Strukturen gibt es immer eine eindeutige Vorgänger – Nachfolger – Relation in nichtlinearen Strukturen hingegen nicht! Man erkennt auch, dass das Array als Spezialfall der Liste (nämlich mit Typ-identischen Elementen) angesehen werden kann.

Wir wollen zunächst einige weitere Eigenschaften einiger Strukturen kennen lernen und auch Implementierungsmöglichkeiten betrachten. In einer späteren Vorlesung widmen wir uns dann speziell den Graph- und Baumstrukturen.

2.4 Listen (*list*)

Die Liste ist eine Datenstruktur zur i.d.R. dynamischen Speicherung von beliebig vielen Elementen. Die statische Variante kennen wir aus der Mathematik und wird Tupel genannt. Charakterisierend ist der Zugriff auf Elemente über einen Indexwert vom Typ Integer in der Regel im Intervall [0,n-1] bei n Elementen.

Implementiert wird eine solche dynamische Struktur dadurch, dass jedes Listenelement einen Verweis auf das nächste Element erhält, wodurch die Gesamtheit der Objekte **zu einer Verkettung** wird. Die zu einer Liste gehörenden Operationen sind relativ unspezifiziert. (Verkettete) **Listen** gehören zu den dynamischen Datenstrukturen, die eine Speicherung von einer im Vorhinein nicht bestimmten Anzahl von miteinander in Beziehung stehenden Werten einfacher oder zusammengesetzter Datentypen erlauben.

Im Gegensatz zu Arrays (siehe unten) müssen die einzelnen Speicherzellen nicht nacheinander im Speicher abgelegt sein, es kann also nicht mit einfacher Adress-Arithmetik gearbeitet wer-

den, sondern die Speicherorte müssen absolut referenziert werden. Im Gegensatz zu Bäumen sind Listen linear, d.h. ein Element hat genau einen Nachfolger und einen Vorgänger.

Operationen auf Listen sind sehr vielgestaltig und hängen sehr stark von der jeweiligen Programmiersprache ab (soweit diese überhaupt direkt unterstützt werden!). Typische Beispiele für Operationen sind:

- append(x) Füge ein Element x am Ende der Liste an
- extend(list2) Füge jedes Element einer beliebigen Liste list2 am Ende der Liste an
- sort Sortiere die Liste nach einer bestimmten Vergleichsfunktion
- reverse Drehe die Reihenfolge der Liste um
- count(x) Gib die Anzahl des Vorkommens des Elementes x in der List
- index(x) Gib den Index des ersten Auftritts von x in der Liste aus
- insert(i,x) Füge Element x an der Stelle i in die Liste ein
- remove(x) Lösche das erste Auftreten von x aus der Liste

Listen sind geordnete und lineare Datenstrukturen. Ein Spezialfall der Liste ist das Feld.

2.5 Feld (array)

Das **Array** ist eine einfache und in Programmiersprachen sehr häufig vorgefundene Datenstruktur. Es werden hierbei mehrere Variablen **vom selben Basisdatentyp** gespeichert. Ein Zugriff auf die einzelnen Elemente ist **über einen Index** möglich. In einer Implementierung entspricht dieser Index oft dem Wert, der zu der Startadresse des Arrays im Speicher hinzuaddiert wird, um die Adresse des Objektes zu erhalten. Die einzigen notwendigen Operationen sind das **indizierte Speichern** und das **indizierte Lesen**, die auf jedes Element des Arrays direkt und schnell zugreifen können. Im eindimensionalen Fall wird das Array häufig als **Vektor** und im zweidimensionalen Fall als **Tabelle oder Matrix** bezeichnet. Arrays sind aber keinesfalls nur auf zwei Dimensionen beschränkt, sondern werden beliebig mehrdimensional verwendet. Wegen ihrer Einfachheit und grundlegenden Bedeutung bieten die allermeisten Programmiersprachen eine konkrete Umsetzung dieser Datenstruktur als zusammengesetzten Datentyp Array im Grundsprachumfang an ... nicht aber Python!

Array (engl.: *Anordnung, Aufstellung, Reihe,* **Reihung**, **Feld**, *Bereich*)

- **Bezeichnung**: `ARRAY of` ... <Basis-Datentyp>
- **Wertebereich:** Abbildung einer endlichen Menge (Indexmenge) auf den Wertebereich eines (oft nur elementaren) Datentyps. Die Indexmenge muss dabei ordinal sein (meist Integer). Durch Anwenden mehrerer Indizes entsteht ein mehrdimensionales Array.
- **Operationen:** Zuweisung und Auslesen (folgend alle erlaubten Operationen des Basisdatentyps

Beispiel: `3D-Vektor is ARRAY(1..3) of FLOAT;`

Mit Hilfe eines Arrays können Daten eines einheitlichen Datentyps geordnet so im Speicher eines Computers abgelegt werden, dass ein Zugriff auf die Daten über einen Index möglich wird. Das (Standard-)Array verwendet im Gegensatz zum assoziativen Array einen ganzzahligen Index zur Adressierung

Adressierung eines Arrays. Die in einem Array gespeicherten Elemente müssen auf Speicheradressen in einem linearen Adressraum abgebildet werden. Dies erfolgt häufig mit Hilfe

eines so genannten Dope-Vektors. In einem n-dimensionalen Array $A[i_1{:}k_1,i_2{:}k_2,...,i_n{:}k_n]$ wird die Adresse eines Elements $a[j_1,j_2,..j_n]$ mit Hilfe folgender Formel berechnet.

$$\sum_{s=1}^{n}[(j_s - i_s)\cdot\prod_{t=s+1}^{n}(k_t - i_t + 1)]\sum_{s=1}^{n}(j_s - i_s)\cdot d_t$$

Da die Produkte $\prod_{t=s+1}^{n}(k_t - i_t + 1)]$ konstant sind, können sie einmalig berechnet werden und der daraus resultierende Dope-Vektor **d** ermöglicht dann mit der Formel

$$\sum_{s=1}^{n}(j_s - i_s)\cdot d_t$$

eine sehr schnelle Berechnung der Adresse eines jeden gespeicherten Elements.

2.6 Mengen (*set*)

Anschaulich gilt*: „Eine **Menge** ist eine Zusammenfassung bestimmter, wohlunterschiedener Objekte unsere Anschauung oder unseres Denkens zu einem Ganzen. Diese Objekte heißen **Elemente** der Menge.“.*

Die Elemente einer Menge müssen in den meisten Implementierungen vom gleichen Datentyp sein. Typischerweise sind die im Anhang angegebenen Operationen, wie

- Schnittmenge $\cap$ oft als & notiert,
- Vereinigung $\cup$ oft als | notiert,
- Differenz $\setminus$ oft als – notiert,
- Enthaltensein ist $\in$ oft mit dem Schlüsselwort *in* getestet.

Mengen sind nur in einigen Programmiersprachen implementiert.

2.7 Stapel (*stack*)

In einem so genannten **Stapelspeicher** (auch Kellerspeicher engl. ***Stack*)** kann eine Anzahl von Objekten gespeichert werden nur in umgekehrter Reihenfolge wieder gelesen werden. Dies entspricht dem LIFO-Prinzip (Last In First Out). Für die Definition und damit die Spezifikation des Stapelspeichers ist es unerheblich, welche Objekte in ihm gespeichert werden. Zu einem Stapelspeicher gehören **zumindest** die Operationen

- *push*, um ein Objekt im Stapelspeicher abzulegen und
- *pop*, um das zuletzt gespeicherte Objekt wieder zu lesen.

Ein Stapelspeicher ist mit dem Stapeln von Kisten vergleichbar. Es kann zu jeder Zeit eine neue Kiste oben auf den Stapel gepackt werden (entspricht *push*) oder eine Kiste von oben heruntergenommen werden (entspricht *pop*). Der verändernde Zugriff ist nur auf das oberste Element des Stapels möglich. Ein Hinzufügen oder Entfernen einer Kiste weiter unten im Stapel ist nicht möglich. Gelegentlich wird auch ein Befehl zum Vertauschen der beiden obersten Elemente angeboten (SWAP) oder ein Befehl zum Bestimmen der Länge des Stapels (LENGTH).

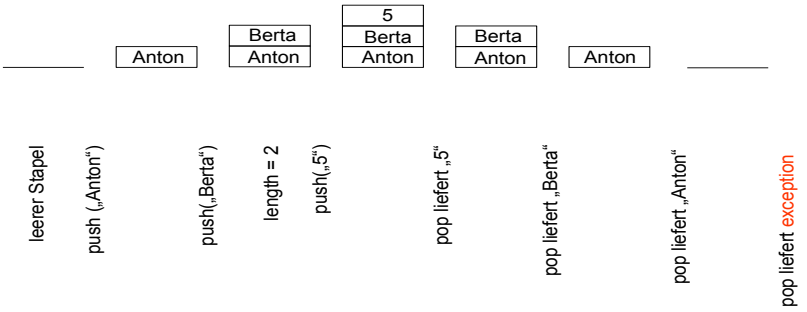


Abbildung 2 Operationen im Stapelspeicher

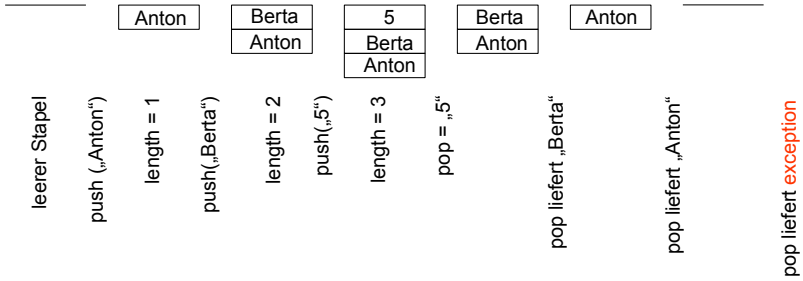


Abbildung 3 Operationen im Stackspeicher

Manchmal unterscheidet man genauer zwischen einem echten **Kellerspeicher**, bei dem kein Element außer dem obersten gelesen werden kann, und einem **Stapelspeicher**, bei dem ohne Veränderung der Daten jedes Element betrachtet werden kann (vergleichbar einem Lesezugriff auf einen Array). Diese Unterscheidung spielt jedoch in der Praxis keine Rolle. Ein Stapelspeicher wird gewöhnlich als Liste oder als Vektor (eindimensionales Feld) implementiert.

Die Verwendung eines Stapelspeichers zur Übersetzung von Programmiersprachen wurde 1957 von Friedrich Ludwig Bauer und Klaus Samelson unter dem Namen "Kellerprinzip" patentiert. Friedrich Ludwig Bauer erhielt später den Computer Pioneer Award (IEEE) für *Computer Stacks* (Samelson war inzwischen verstorben).

Stapelspeicher haben vielfältige Anwendungen: Hardwarenah wird ein Stapel oft genutzt, um die Rücksprungadresse bei einem Unterprogrammaufruf zu speichern. Hierzu haben viele Mikroprozessoren ein spezielles Register, den Stackpointer (Stapelzeiger). Dieses Register enthält eine Speicheradresse, die auf den gerade obersten Stapeleintrag zeigt. Wenn mit der Operation *push* ein weiteres Objekt auf dem Stapel abgelegt wird, dann erhöht sich der Wert des Stapelzeigers und zeigt so auf die nächste Adresse, in die der neue Stapeleintrag geschrieben wird. Bei *pop* wird der Eintrag der Adresse gelesen und anschließend der Stapelzeiger vermindert, so dass er auf den letzten Stapeleintrag zeigt.

Compiler nutzen gewöhnlich *push*- und *pop*-Operationen beim Aufruf eines Unterprogramms, um hier die aktuellen Parameter zu hinterlegen.

Zur Übersetzung des nutzen Compiler und Interpreter einen Parser, der bei der Textanalyse Syntax-Regeln auf einem Stapel ablegt und so vergleichend dem nachfolgenden Textelement eine angenommene Bedeutung (das oberste Stapelelement) zuordnen kann. Programmierspra-

chen, die auf eine virtuelle Maschine aufsetzen (zum Beispiel Java, C#, Pascal), optimieren den kompilierten Zwischencode für die Verwendung eines Stapels.

Stapelspeicher eignen sich auch zur Auswertung von Klammerausdrücken, z.B. bei Ausdrücken in der so genannten **Infixnotation** (der Operator steht zwischen den beteiligten Zahlwerten, die gewöhnliche Schreibweise, etwa a + b für die Operation *addiere*(a,b)). Hierbei werden zunächst Teilterme in einem Stapel zwischengelagert, bevor das Ergebnis durch Abarbeiten des Stapels errechnet wird.

Dabei wird z.B. zunächst für Operatoren und Operanden je ein Stapelspeicher initialisiert. Der zu verarbeitende Klammerausdruck wird zeichenweise eingelesen. Wird eine öffnende Klammer eingelesen, so ist keine Operation auszuführen. Wird ein Operand oder Operator eingelesen, so wird diese(r) auf dem jeweiligen Stapelspeicher abgelegt. Wird eine schließende Klammer eingelesen, so wird der oberste Operator vom Stapelspeicher für die Operatoren genommen und entsprechend diesem Operator eine geeignete Anzahl von Operanden, die zur Durchführung der Operation benötigt werden. Das Ergebnis wird dann wieder auf dem Stapelspeicher für Operanden abgelegt. Sobald der Stapelspeicher für die Operatoren leer ist, befindet sich im Stapelspeicher für die Operanden das Ergebnis.

Streng genommen wandelt diese Vorschrift eine Infixnotation in eine **Postfixnotation** um: diese macht eine Klammersetzung und Prioritätsregeln für die Operationen überflüssig. Operanden werden auf dem Stapel abgelegt. Binäre Operatoren (zum Beispiel +, −, *, /) holen die oberen beiden Werte, unäre Operatoren (zum Beispiel Vorzeichenwechel) einen Wert vom Stapel und legen anschließend das (Zwischen-)Ergebnis dort wieder ab.

2.8 (Warte-)Schlange (*queue*)

In einer Warteschlange (engl. queue) kann eine beliebige Anzahl von Objekten gespeichert werden, jedoch können die gespeicherten Objekte nur in der gleichen Reihenfolge wieder gelesen werden, wie sie gespeichert wurden. Dies entspricht dem FIFO-Prinzip (First In First Out). Für die Definition und damit die Spezifikation der Queue ist es unerheblich, welche Objekte in ihm gespeichert werden. Zu einer Queue gehören zumindest die Operationen:

- *enqueue*, um ein Objekt in der Warteschlange zu speichern und
- *dequeue*, um das zuerst gespeicherte Objekt wieder zu lesen und aus der Warteschlange zu entfernen.

Eine Warteschlange wird gewöhnlich als Liste implementiert, kann aber auch ein Vektor sein. Eine spezielle Implementierung ist eine als **Ringpuffer**. Die Besonderheit des Ringpuffers ist, dass er eine **feste Größe** besitzt. Dabei werden, wenn der Puffer voll ist, die ältesten Inhalte wieder überschrieben. Eine als Ringpuffer implementierte Warteschlange sollte daher in diesem Fall entweder einen Pufferüberlauf signalisieren. In manchen Fällen gehört der „Datenverlust“ zum Funktionsprinzip, z.B. in einem Überwachungssystem, bei dem die Ereignisse der letzten 5 Minuten hinterlegt wurden.

Eine Spezialisierung der Warteschlange ist die Vorrangwarteschlange, die auch Prioritätswarteschlange bzw. engl. *Priority Que*ue genannt wird. Dabei wird vom FIFO-Prinzip abgewichen. Die Durchführung der `enqueue`-Operation, die in diesem Fall auch meist `insert`-Operation genannt wird, sortiert das Objekt gemäß einer gegebenen Priorität, die jedes Objekt mit sich führt, in die Vorrangwarteschlange ein. Die `dequeue`-Operation liefert immer das Objekt mit der höchsten Priorität.

2.9 *Dictionaries (*Assoziatives Array)

Die Bezeichnung Assoziatives Array deutet auf die Hauptfunktion hin: Es handelt sich um einen „Inhaltsadressierten" Speicher. Es wird nicht über einen numerischen Index, sondern über einen **Schlüssel** auf die gespeicherten Daten zugegriffen. Als **Schlüssel** bezeichnet man eine Entität aus einer **geordneten Menge**, die man einem Objekt oder Element zuordnen kann. Der Schlüssel erlaubt es, Elemente einer Relation zu identifizieren. Über die zugeordneten Schlüssel erhalten die Objekte oder Elemente einer Menge dadurch selbst eine Ordnung. Assoziative Arrays speichern also Relationen.

Viele Programmiersprachen unterstützen assoziative Arrays, z. B. Perl, PHP, Ruby, Smalltalk, Tcl, C++ (als Klasse der Standardbibliothek), Java, Delphi (als Array-Property) und Visual Basic. Statt von einem assoziativen Array spricht man auch von einem **Dictionary** (Smalltalk, Python, Objective-C), einer **Map** (C++, Java) einem **Hash** (Perl, Ruby) oder einer **Hashtable/Hashmap** (Java).

Eine Spezialisierung des assoziativen Arrays ist die **Hashtabelle** bzw. Streuwerttabelle. Hashtabellen eignen sich also vor allem dazu, schnell Datenelemente in großen Datenbeständen aufzufinden. Das wesentliche Prinzip des Verfahrens beruht auf dem Einsatz einer Hash-Funktion. Diese Funktion berechnet aus dem (Such-)schlüssel einen Index in die Hashtabelle, die z.B. als Array implementiert ist. Die Hashfunktion bildet eine Menge von Suchschlüsseln auf denselben Index ab. Hierdurch kann es zu so genannten Kollisionen kommen, d.h. verschiedene Schlüssel ergeben den gleichen Index.

Wegen dieser möglichen Kollisionen werden die Daten i.d.R. nicht direkt im Array gespeichert, sondern z.B. in Behältern (Buckets), die mehrere Datenelemente mit dem gleichen Hash-Wert aufnehmen können. Das Array selbst enthält nur noch einen Verweis auf dieses Bucket. Um das gesuchte Element zu finden, muss eine weitere Suche über die Elemente im Bucket durchgeführt werden. Ist die Hashtabelle ausgewogen, befinden sich nur wenige Datensätze in einem Bucket und der letzte Schritt hat einen vergleichsweise geringen Aufwand.

Wichtig sind Hashtabellen z.B. in Datenbanksystemen. Hier werden sie als Index für Tabellen verwendet. Des Weiteren finden Hashtabellen Einsatz zur Implementierung von Mengen (Sets) oder eines Caches verwendet. Symboltabellen, die die Menge der benutzten Namen in einem Programm enthalten, werden meistens auch als Hashtabelle realisiert.

2.10 Verbund (struct, record, union)

- **Bezeichnung**: RECORD, STRUCT oder UNION

- **Wertebereich:** Ein Verbund enthält eine Folge verschiedener Komponenten, welche verschiedene Datentypen haben können. Als Komponententyp ist jeder Typ zulässig.

- **Operationen:** Zuweisung, Gleichheit (stark programmiersprachenabhängig!)
  Jede Komponente besitzt einen eigenen Namen. Angesprochen wird eine Komponente in folgender Weise:
  record_name.komponenten_name oder bei mehrfach zusammengesetzten Records
  record_name.komponenten_name1.komponenten_name 2

Beispiel:
```
type Prüfung is RECORD (Fach: STRING, Student: STRING,
        Punkte: INTEGER, Prüfer: STRING, Termin: DATUM)
```

Unter einem **Verbund** versteht man also die Zusammenfassung mehrerer elementarer oder aggregierter Datentypen zu einem neuen Typ.

In vielen Programmiersprachen existieren zusätzlich Möglichkeiten, den Speicherbereich eines Verbunds unterschiedlich zu interpretieren. Das wird **Variantenrecord** oder **UNION** genannt. Dabei ist jedoch keine Typsicherheit mehr gegeben.

2.11 Zusammenfassung

Die von einer Programmiersprache zur Verfügung gestellten Datentypen kennzeichnen diese in besonderer Weise. Wir müssen unterscheiden:

- einfache Speicherstrukturen – mächtige Datentypen
- statische – dynamische Datenstrukturen

Zu den **einfachen** Speicherstrukturen gehören das Array und der Record. Im wesentlichen realisieren diese „nur" eine strukturierte Ablage der Daten. Hauptoperationen sind Lese- und Schreib-Operationen, sowie einige Verwaltungsoperationen: Länge, etc.

Mächtige Datentypen realisieren neben diesen Grundoperationen Typ-entsprechend Verknüpfungen zwischen den Elementen, z.B. Mengenoperationen, Listenoperationen, etc.

Statische Datenstrukturen sind solche, deren (maximale) Größe beim Anlegen dieser Struktur (also beim Programmieren) bekannt sind. Dynamische Datenstrukturen sind solche, die ihre Größe während der Laufzeit beliebig verändern können. Für den Programmierer sind dynamische Strukturen bequem, weil er sich keine Gedanken über die maximale Größe machen muss und keine Spezialbehandlungen beim Überschreiten der angenommenen maximalen Größe, das wieder Freigeben von nicht mehr benötigten Speicherplatz, etc. machen muss. Erfahrungsgemäß sind solche Programmteile sehr fehleranfällig, z.B. in C.

In **dynamischen** Datenstrukturen übernimmt der Compiler oder Interpreter diese Verwaltungsaufgaben. Dies erfordert, verglichen mit statischen Strukturen, andere Implementierungen und zum Teil einen erheblichen Verwaltungs-Overhead – Speicherbedarf und Laufzeit sind fast immer deutlich größer.

Jede Programmiersprache stellt eine Auswahl der vorgestellten Strukturen als *builtins* zur Verfügung. Die Programmiererin kann hieraus dann entweder konstruktiv durch erlaubte Kombinationen dieser Strukturen oder selektiv durch Spezialisierung eigene schaffen. Dies ist das Haupthema der nächsten Vorlesung: Klassen. Hierdurch lassen sich dann auch Standard-Datenstrukturen erschaffen, wie Graphen und deren Spezialisierung Bäume. Wir werden uns in der übernächsten Vorlesung diesem Thema widmen.

3 Anhang

1. Mengen

Die so genannte **Naive Mengenlehre** geht auf Georg Cantor zurück. Nach seiner Definition von 1877 gilt:

*„Eine **Menge** ist eine Zusammenfassung bestimmter, wohlunterschiedener Objekte unsere Anschauung oder unseres Denkens zu einem Ganzen. Diese Objekte heißen **Elemente** der Menge.“*

Diese anschauliche (aber naive) Mengenauffassung erwies sich als nicht widerspruchsfrei. Am bekanntesten ist die Russellsche Antinomie. Deshalb wurde später die Cantorsche Mengenlehre als *naive* Mengenlehre bezeichnet.

Endliche Mengen können (insbesondere wenn sie relativ wenig Elemente haben) durch Aufzählen ihrer Elemente angegeben werden.

Beispiel: M = {rot, grün, blau}. Prinzipiell ist die Reihenfolge beliebig. Es ist unüblich (aber möglich), dabei Elemente mehrfach anzuführen. Da eine Menge dadurch bestimmt ist, welche Elemente sie enthält, ändert sich durch wiederholtes Anschreiben von Elementen oder durch Vertauschen nichts: M = {blau, grün, rot} = {rot, blau, grün, blau}.

In der Mathematik wird heute meist eine **axiomatische Mengenlehre** (z.B: die Zermelo-Fraenkel-Mengenlehre) benutzt:

Gleichheit: Zwei Mengen heißen gleich, wenn sie die selben Elemente enthalten.

Diese Definition ist **die** grundlegende Eigenschaft von Mengen. Formal:

$$A = B \Leftrightarrow \forall x(x \in A \leftrightarrow x \in B)$$

[$x \in A$ bedeutet: x ist Element von A . Ein Element bezeichnet also **genau ein** Objekt aus einer Menge].

Leere Menge: Die Menge, die kein Element enthält, heißt leere Menge. Sie wird mit ∅ oder auch {} bezeichnet.

Aus der Gleichheit der Mengen folgt, dass es nur ***eine*** leere Menge gibt: Jede „andere“ leere Menge enthält die selben Elemente (nämlich keine), ist also gleich.

Teilmenge: Eine Menge *A* heißt Teilmenge einer Menge *B*, wenn jedes Element von *A* auch Element von *B* ist.

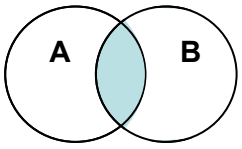
B wird dann zuweilen auch **Obermenge** von *A* genannt. Formal:

$$A \subseteq B \Leftrightarrow \forall x(x \in A \rightarrow x \in B)$$

> **Echte Teilmenge**: *A* ist echte Teilmenge von *B* (oder *B* ist echte Obermenge von *A*) notiert als $A \subset B$, wenn *A* Teilmenge von *B* ist, aber von *B* verschieden.

Schnittmenge: *A* geschnitten mit *B* (*oder*: **Der Durchschnitt von *A* und *B***) ist die Menge aller Elemente, die sowohl in *A* als auch in *B* enthalten sind.

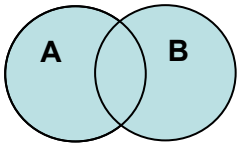
$$A \cap B := \{x \mid (x \in A) \wedge (x \in B)\}$$



Schnittmenge von A und B

Vereinigungsmenge: *A* vereinigt mit *B* (*oder*: Die Vereinigung von *A* und *B*) **ist die Menge aller Elemente, die in *A* oder in *B* enthalten sind.** Das „oder" ist hier nicht-ausschließend zu verstehen. Die Vereinigung umfasst auch die Elemente, die in *beiden* Mengen enthalten sind.

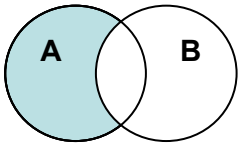
$$A \cup B := \{x \mid (x \in A) \vee (x \in B)\}$$



Vereinigungsmenge von A und B
Vereinigungsmenge ist der zu *Schnittmenge* duale Begriff:

Differenz und Komplement: *A ohne B* (Schreibweise A\B oder A-B) ist die Menge aller Elemente, die in *A* enthalten sind, aber nicht in *B*

$$A - B := \{x \mid (x \in A) \wedge (x \notin B)\} \qquad B - A := \{x \mid (x \notin A) \wedge (x \in B)\}$$



Differenzmenge A ohne B
Komplement B in A

Komplement: bezeichnet das Komplement von *B* in A, das ist die Menge aller Elemente von A, die nicht in *B* liegen.

Symmetrische Differenz: ist die Menge aller Elemente, die in einer aber nicht in beiden der gegebenen Mengen liegen

$$A \Delta B := (A - B) \cup (B - A)$$

Als **Mächtigkeit** (auch *Kardinalität*) der Menge *A* (Schreibweise $|A|$) bezeichnet man die Anzahl der Elemente von *A*. Für eine endliche Menge ist die Mächtigkeit eine natürliche Zahl;

bei unendlichen Mengen unterscheidet man nach „verschiedenen Graden der Unendlichkeit". Diese werden als Kardinalzahlen bezeichnet.

Die **Potenzmenge** P(A) ist die Menge aller Teilmengen von A.

Produktmenge oder kartesisches Produkt

Die zweistellige Produktmenge ist die Menge aller geordneten Paare, die sich aus den Mengen A und B bilden lassen.

$$A \times B := \{(a,b) \mid (a \in A) \wedge (b \in B)\}$$

2. Tupel

Das n-**Tupel** ist ein Begriff der Mathematik. Er bezeichnet eine **geordnete** Zusammenstellung von Objekten, im Gegensatz zu Mengen, deren Elemente keine festgelegte Reihenfolge haben.

n-Tupel werden üblicherweise durch runde Klammern angegeben (hier für $n = 3$, da drei Elemente im Tupel vorhanden sind), z.B. (a,b,c)

Die Objekte werden als **Elemente**, Komponenten oder Einträge des n-Tupels bezeichnet. Dadurch, dass bei einem n-Tupel jedem seiner Elemente ein eindeutiger Platz zugeordnet ist, kann es auch mehrfach dasselbe Element enthalten. n bezeichnet hierbei die Anzahl der Elemente des n-Tupels. Diese Anzahl muss endlich sein. Im Fall eines 2-Tupels spricht man auch von einem geordneten **Paar**, für $n = 3$ von einem **Tripel**. Die entsprechenden, selten gebrauchten Wörter **Quadrupel**, **Quintupel** usw. gaben dem n-Tupel den Namen.

Oft werden die Elemente eines n-Tupels mit Hilfe der natürlichen Zahlen indiziert: Sei A = (a,b,c), dann ist A(1) = a, A(2) =b, A(3) =c oder A_1 = a, A_2 =b, A_3 =c

Abgrenzung gegenüber Mengen

Ein n-Tupel ist von einer Menge zu unterscheiden. Bei einer Menge ist die Reihenfolge der Elemente unerheblich. Deswegen kann eine Menge ein und dasselbe Element niemals mehrfach enthalten. Sie kann es nur entweder enthalten, oder es nicht enthalten. Für die Menge stehen geschweifte Klammern, die kennzeichnen, dass die Elemente ungeordnet, d.h. ohne Reihenfolge, sind .

Beispiel: Ist $a \neq b$, so ist das Paar (a, b) verschieden von (b, a), dagegen ist die Menge $\{a, b\}$ dieselbe Menge wie $\{b, a\}$.

n-Tupel von Zahlen (oder anderen gleichartigen Objekten) nennt man je nach Kontext auch Vektoren, wie z.B. Elemente des $\mathbf{R}^3$ oder allgemeiner $\mathbf{R}^n$. Je nachdem, ob man sie horizontal (a,b,c) oder vertikal $\begin{pmatrix} a \\ b \\ c \end{pmatrix}$ schreibt, spricht man von *Zeilen-* oder *Spaltenvektoren*.Man beachte jedoch, dass die lineare Algebra einen abstrakteren Vektorbegriff verwendet: Vektoren sind definiert als Elemente eines Vektorraums. Zwar ist der $\mathbf{R}^n$ (mit der naheliegenden Struktur) ein Vektorraum, aber im Allgemeinen sind Vektoren im Sinne der linearen Algebra keine n-Tupel. Vektor im Sinne der linearen Algebra zu sein ist auch keine Eigenschaft eines einzelnen Objek-

tes, sondern nur sinnvoll für ein Objekt als Teil einer Gesamtheit mit algebraischer Zusatzstruktur, eines Vektorraums.

Der Begriff *n*-Tupel wird bei der Definition des kartesischen Produkts endlich vieler Mengen verwendet. In weiterer Folge wird dann der Begriff des kartesischen Produkts bei der Definition der Begriffe Relation, Funktion und Folge benötigt; *n*-Tupel ist daher ein sehr grundlegender Begriff der Mathematik.

Für *n*-Tupel wird vor allem gefordert, dass zwei *n*-Tupel **dann und nur dann gleich sind**, wenn sie in allen entsprechenden Komponenten übereinstimmen:

$$(a_1, a_2, ..., a_n) = (b_1, b_2, ..., b_n) \Leftrightarrow (a_1 = b_1) \wedge (a_2 = b_2) \wedge ... \wedge (a_n = b_n)$$

3. Relationen

Eine **Relation** ist allgemein eine Beziehung, die zwischen Objekten bestehen kann. Relationen im Sinne der Mathematik sind ausschließlich diejenigen Beziehungen, bei denen stets klar ist, ob sie bestehen oder nicht. Zwei Objekte können entsprechend nicht „zu einem gewissen Grade" in einer Relation zueinander stehen.

Damit ist eine einfache mengentheoretische Definition des Begriffs der Relation möglich: Eine Relation R ist eine Menge von *n*-Tupeln. Objekte, die in der Relation R zueinander stehen, bilden ein *n*-Tupel, das Element von R ist.

Wenn nicht ausdrücklich anderes angegeben ist, versteht man unter einer Relation eine „zweistellige" oder „binäre" Relation, also eine Beziehung zwischen je zwei Dingen. Die Elemente eines Paares (a,b) können aus verschiedenen Grundmengen A und B stammen; die Relation heißt dann *heterogen* oder „Relation *zwischen* den Mengen A und B". Wenn die Grundmengen übereinstimmen, $A = B$, heißt die Relation auch *homogen* oder „Relation *in* der Menge A". Wichtige Spezialfälle, zum Beispiel **Äquivalenzrelationen** und **Ordnungsrelationen**, sind Relationen *in* einer Menge.

Die vorstehenden Überlegungen erlauben uns nun folgende formale Definition: eine **binäre Relation R** ist eine Teilmenge des kartesischen Produkts zweier Mengen A und B:

$$R \subseteq A \times B \quad mit \quad A \times B := \{(a,b) \mid (a \in A) \wedge (b \in B)\}$$

Allgemeiner ist eine *n*-stellige Relation eine Teilmenge des kartesischen Produkts von n Mengen $A_1, \ldots, A_n$. Einige wichtige Eigenschaften von binären Relationen sind:

Die Relation heißt	**wenn gilt**	**und das bedeutet**
reflexiv $(A = B)$	$\forall a \in A : (a,a) \in R$	Jedes Element steht in Relation zu sich selbst.
irreflexiv	$\forall a \in A : (a,a) \notin R$	Kein Element steht in Relation zu sich selbst.
symmetrisch	$\forall (a,b) \in A^2 : (a,b) \in R \Rightarrow (b,a) \in R$	Die Relation ist ungerichtet, z.B.

($A = B$)		folgt aus $a=b$ stets $b=a$
asymmetrisch	$\forall (a,b) \in A^2 : (a,b) \in R \Rightarrow (b,a) \notin R$	Es gibt keine zwei Elemente, die in beiden Richtungen in Relation stehen, z.B. folgt aus $a<b$ stets, dass $b<a$ nicht gilt
antisymmetrisch *bzw.* identitiv ($A = B$)	$\forall (a,b) \in A^2 : (a,b) \in R \wedge (b,a) \in R \Rightarrow a = b$	Es gibt keine zwei *verschiedenen* Elemente, die in beiden Richtungen in Relation stehen, z.B. folgt aus $a \leq b$ und $b \leq a$ stets $a=b$
transitiv ($A = B$)	$\forall a,b,c \in A : (a,b) \in R \wedge (b,c) \in R \Rightarrow (a,c) \in R$	Anfang und Ende einer verbundenen Sequenz sind verbunden, z.B. folgt aus $a<b$ und $b<c$ stets $a<c$
intransitiv	$\exists a,b,c \in A : (a,b) \in R \wedge (b,c) \in R \Rightarrow (a,c) \notin R$	nicht bei jeder verbundenen Sequenz sind Anfang und Ende verbunden
antitransitiv	$\forall a,b,c \in A : (a,b) \in R \wedge (b,c) \in R \Rightarrow (a,c) \notin R$	bei keiner verbundenen Sequenz sind Anfang und Ende verbunden
total *bzw.* linear *bzw.* konnex ($A = B$)	$\forall a,b \in A : (a,b) \in R \vee (b,c) \in R$	Je zwei Elemente stehen in Relation, z.B. gilt stets $a \leq b$ oder $b \leq a$
linkstotal	$\forall a \in A : \exists b \in B : (a,b) \in R$	Jedes El. aus A hat mindestens einen Partner in B
surjektiv bzw. rechtstotal	$\forall b \in B : \exists a \in A : (a,b) \in R$	Jedes El. aus B hat mindestens einen Partner in A
injektiv bzw. linkseindeutig	$\forall a,c \in A : \forall b \in B : (a,b) \in R \wedge (c,b) \in R \Rightarrow a = c$	Kein El. aus B hat mehr als einen Partner in A
rechtseindeutig	$\forall a \in A : \forall b,c \in B : (a,b) \in R \wedge (a,c) \in R \Rightarrow a = c$	Kein El. aus A hat mehr als einen Partner in B

Wichtige Klassen von Relationen:

- Eine **Äquivalenzrelation** ist reflexiv, transitiv und symmetrisch.
- Eine **Funktion (Abbildung)** ist linkstotal und rechtseindeutig (d.h. N:1).

- Eine **partielle Funktion** ist rechtseindeutig.
- Eine **Quasiordnung** oder *Präordnung* ist reflexiv und transitiv. (Nicht notwendig symmetrisch oder antisymmetrisch.)
- Eine Halbordnung oder **partielle Ordnung** ist reflexiv, transitiv und antisymmetrisch. (Halbordnung = Präordnung + antisymmetrisch)
- Eine lineare Ordnung oder **totale Ordnung** ist reflexiv, transitiv, antisymmetrisch und total (linear). (lineare Ordnung = Halbordnung + total)
- Eine Wohlordnung ist eine lineare Ordnung, bei der jede nichtleere Teilmenge von A ein kleinstes Element besitzt.

4. Funktionen (Abbildungen)

Eine **Funktion** *f* weist *jedem* Element einer Definitionsmenge **A** (einem "x-Wert") *genau ein* Element einer Zielmenge **B** (einen "y-Wert") zu. Eine Funktion hat demnach die explizite Eigenschaft:

Jedem *x*-Wert aus dem Definitionsbereich A wird **genau ein** y-Wert aus B zugeordnet. In manchen Fällen kann man eine Zuordnungsvorschrift angeben, man nennt sie **Funktionsgleichung**.

Mengentheoretisch ist eine Funktion eine **linkstotale** und **rechtseindeutige Relation**, das heißt: Jedes Element aus *A* hat mindestens einen Partner in *B* **und** kein Element. aus A hat mehr als einen Partner in B.

Daneben gibt es noch den Begriff **partielle Funktion**, der besonders in der Informatik verwendet wird. Hier wird nicht verlangt, dass jedem Argument ein Wert zugeordnet wird, es wird lediglich verlangt, dass es **höchstens** einen zugeordneten Wert gibt.

Eine partielle Funktion ist keine Funktion im strengen oben definierten Sinn; zur Verdeutlichung nennt man diese dann in diesem Kontext **totale Funktion**. nennen.