Universität Hamburg
Fachbereich Informatik

Proseminar: Programmierung in R

# Datenstrukturen

*Dominik Scheinert*

betreut von
Dr. Julian Kunkel

2. September 2016

# Inhaltsverzeichnis

# 1 Einleitung

Im Rahmen des Proseminars *Programmierung in R* im Sommersemester 2016 beschäftigten sich die Studenten und Studentinnen mit der Programmiersprache ***R*** und lernten deren wichtigste Funktionalität sowie weitere Pakete zur Erweiterung kennen.

Die folgende Ausarbeitung behandelt das Thema **Datenstrukturen in R**. Nach Klärung der wichtigsten Begriffe für das Erreichen eines Basisverständnisses wird die Bedeutung von Datenstrukturen in der Informatik erläutert, bevor sich dann konkret der Programmiersprache R gewidmet wird. Hier werden alle Datenstrukturen in R beschrieben und anhand von sinnvollen Code-Beispielen veranschaulicht. Ergänzt wird dies abschließend durch einen Ausblick auf ausgewählte Pakete, die importiert werden können um die Funktionalität zu erweitern.

# 2 Motivation

## 2.1 Daten

*„Daten sind in erkennungsfähiger Form dargestellte Elemente einer Information, die in Systemen verarbeitet werden können.“*
[ITW]

Dieser gängigen Definition von *Daten* zufolge geht es konkret um Zeichenketten, für die bestimmte Abmachungen gelten, um sie, unabhängig von Ort oder technischer Ausstattung, auf die gleiche Weise interpretieren zu können. Wiederum werden interpretierte Daten als *Informationen* definiert. Damit sind Daten **das** zentrale Thema der Informatik, denn die Informatik ist die *„Wissenschaft der systematischen Verarbeitung von Informationen, insbesondere der automatischen Verarbeitung mit Hilfe von Digitalrechnern.”* [HE93] Ein Ziel dabei ist es, diese Verarbeitung effizient in Bezug auf Bearbeitungszeit und Speicher-/Rechenaufwand zu gestalten. Eine sinnvolle Organisation der Daten scheint deshalb erstrebenswert.

## 2.2 Datenstrukturen

Geht es um das Thema *Datenstrukturen*, so begegnen einem meist zwei Begriffe, zwischen denen es zu differenzieren gilt: **Abstrakter Datentyp** und **Datenstruktur**.

1. *„Ein abstrakter Datentyp (ADT) ist ein Verbund von Daten zusammen mit der Definition aller zulässigen Operationen, die auf sie zugreifen.”* [Wika] Ein ADT sagt also noch in keinster Weise etwas darüber aus, *wie* bestimmte Operationen mit den Daten arbeiten, sondern beschreibt nur *was* die Operationen tun. Im Kontext von objektorientierten Programmiersprachen wie Java ist ein ADT ein Interface, welches eine Klasse spezifiziert, aber keine Implementationsdetails besitzt.

2. Eine Datenstruktur hingegen ist die konkrete Implementation und somit die Umsetzung des theoretischen Konzeptes. So definiert man eine Datenstruktur auch als *”das Konstrukt in einem Programm (bzw. Im Speicher [...]), das Daten auf eine gewisse Weise speichert.”* [Kir03]

Die formale Unterscheidung ist an dieser Stelle notwendig, um beide Begriffe im Folgenden griffig und ohne weitere Erläuterungen benutzen zu können.

# 3 Datenstrukturen in der Informatik

Ungeachtet der Tatsache, dass die Informatik eine noch recht junge Wissenschaft ist, existieren heute diverse abstrakte Datentypen und noch viel mehr Implementationen, begründet u.a. durch die Vielzahl an Programmiersprachen, die meist eine auf ihre Syntax und Semantik angepasste Implementation benötigen. Die vielen Datenstrukturen lassen sich dann wiederum kategorisieren, denn manche sind nur für bestimmte Anforderungen geeignet. Meist werden Datenstrukturen durch die drei wichtigsten Operationen **Einfügen, Löschen** und **Suchen** klassifiziert.*Arrays* beispielsweise benötigen für jede dieser Operationen im Schnitt die gleiche Zeit, doch ist diese wiederum abhängig von der Anzahl der Elemente im Array. Andere Datenstrukturen benötigen dagegen vielleicht für eine der Operationen stets nur konstant viel Zeit, aber dafür ist der Zeitaufwand eventuell für die anderen beiden Operationen enorm. Der gewiefte Informatiker ist deshalb gut beraten, sich mit möglichst vielen verschiedenen Datenstrukturen vertraut zu machen und deren Vor- und Nachteile zu kennen.

# 4 Datenstrukturen in R

Im Folgenden werden alle Datenstrukturen in R vorgestellt, deren Eigenschaften erläutert und anhand von Code-Beispielen verdeutlicht. In Bezug auf die jeweiligen Operationen der unterschiedlichen Datenstrukturen wird verständlicherweise nur ein Ausschnitt präsentiert. Bei Interesse oder komplexeren Anforderungen sei an dieser Stelle auf die offizielle Seite **https://www.r-project.org/** verwiesen, welche das Projekt R betreut.

## 4.1 Vektoren

In der Programmiersprache R stellen *Vektoren* die Basis-Datenstruktur dar. Sehr oft werden sie als Eingabeparameter von Funktionen erwartet oder bilden das interne Fundament komplexerer Datenstrukturen. Ein Vektor lässt sich mit der *vector()*-Funktion erzeugen und kann, je nach Belieben, zusätzlich an eine Variable gebunden werden. Ein solcher Aufruf sieht wie folgt aus:

```
> a = vector()
# Der erzeugte Vektor wird der Variable a zugewiesen.
```

Natürlich lassen sich bei der Vektorerzeugung auch Parameter mit übergeben, die beispielsweise den Typ oder die Länge des zu erzeugenden Vektors spezifizieren.

```
> a = vector("numeric", length = 10)
> length(a)
[1] 10
> class(a)
[1] "numeric"
# Der erzeugte Vektor ist vom Typ "numeric" und hat die Laenge 10.
```

Ein Vektor lässt sich mit Elementen **eines** Datentyps füllen. Werden beim Funktionsaufruf zur Erzeugung Elemente verschiedener Datentypen übergeben, findet eine Umwandlung in den jeweils flexibelsten Datentyp statt. Dabei gilt: $logical -> integer -> numeric -> complex -> character$.
Das untenstehende Beispiel verdeutlicht dies. An dieser Stelle sei gesagt, dass es noch einen alternativen Weg der Vektorerzeugung gibt: Die *c()*-Funktion (wobei *c* in diesem Kontext für *combine* oder *concatenate* steht) nimmt Elemente entgegen und erzeugt aus ihnen einen Vektor, wobei der Erzeugung ggf. wie gerade beschrieben eine Umwandlung vorher geht.

```
> a = c(1L,2.2,TRUE,'h')
> a
[1] "1"    "2.2"  "TRUE" "h"
# Der erzeugte Vektor nach Umwandlung einiger Elemente.
```

Da ein Vektor in letzter Instanz also ein Array von Elementen eines Datentyps ist, ist auch der Zugriff auf die Elemente mittels Indexierung realisiert. Vektoren sind eindimensional, für die Lokalisierung eines Elementes in einem Vektor mittels Indexierung benötigt man demnach nur *eine Positionsangabe*.

```
< a = c(1,2,3)
# Mit a[1] wird das erste Element in dem Vektor zurueckgegeben.
> a[1]
[1] 1
```

Vektoren bieten darüber hinaus eine Fülle von Operationen an. Insbesondere bei komplexen, numerischen oder Integer-Vektoren ist dies sehr nützlich. Neben den klassischen Rechenoperationen gibt es auch solche, die das Maximum in einem Vektor bestimmen oder den Durchschnitt bilden. Dabei gilt grundsätzlich, dass *Vektor1* immer mindestens gleich lang, auf jedenfall aber immer ein Vielfaches von *Vektor2* sein muss.

```
> Vektor1 = c(1,2,3)
> Vektor2 = c(4,5,6)
> Vektor1 + Vektor2
[1] 5 7 9

# Dies funktioniert jedoch nicht:
> Vektor1 = c(1,2,3)
> Vektor2 = c(4,5)
> Vektor1 + Vektor2
[1] 5 7 7
Warnmeldung:
In Vektor1 + Vektor2 : Laenge des laengeren Objektes
ist kein Vielfaches der Laenge des kuerzeren Objektes
```

## 4.2 Matrizen

Matrizen stellen neben Vektoren eine weitere Datenstruktur in R dar. Dabei handelt es sich im Grunde nur um *spezielle* Vektoren, da Matrizen im Gegensatz zu einfachen Vektoren nicht eindimensional, sondern zweidimensional sind. Ansonsten haben Matrizen die gleichen Charakteristika wie Vektoren, sie können folglich auch nur Elemente eines Datentyps aufnehmen und der Zugriff auf Elemente wird durch Indexierung realisiert.
Matrizen werden mit der *matrix()*-Funktion erzeugt, wobei erforderlich ist, durch Parameter die Anzahl der Spalten und Zeilen festzulegen. Standardmäßig werden Matrizen spaltenweise befüllt, dies lässt sich jedoch durch einen zusätzlichen Parameter ändern.

```
# Eine einfache 2 x 2 -Matrix wird erzeugt.
> m = matrix(nrow = 2 , ncol = 2)
> m
     [,1] [,2]
[1,]   NA   NA
[2,]   NA   NA

# Zugriff: Einfuegen
> m[2,2] = 5
> m
     [,1] [,2]
[1,]   NA   NA
[2,]   NA    5

# Zugriff: Auslesen
> m[2,1]
[1] NA

# Zeilen oder Spalten lassen sich auch komplett befuellen / auslesen:
> m[1,] = c(3,3)
> m
     [,1] [,2]
[1,]    3    3
[2,]   NA    5

> m[1,]
[1] 3 3
>
```

Oft ist es erforderlich, eine Matrix zu transponieren, also die Zeilen und Spalten zu vertauschen. Diese Möglichkeit ist durch das Bereitstellen der *t()*-Funktion gegeben.

```
> t(m)
     [,1] [,2]
[1,]    3   NA
[2,]    3    5
```

In bestimmten Fällen kann es vorkommen, dass eine Matrix nachträglich um Spalten oder Zeilen erweitert werden muss. Um dies möglichst einfach durchzuführen, bietet es sich an die *cbind*-Funktion (für Spaltenkombinationen) oder die *rbind*-Funktion (für Zeilenkombinationen) zu nutzen.Hier bei gilt es besonders darauf zu achten, dass die anzufügenden Spalten / Zeilen die gleiche Länge haben wie die Spalten / Zeilen der Ausgangsmatrix. Die folgenden Beispiele verdeutlichen dies.

```
# Bei Zeilenkombinationen bedarf es der gleichen Anzahl an Spalten.
> Matrix1 = matrix(1:6, nrow = 2, ncol = 3)
> Matrix2 = matrix(1:15, nrow = 5 , ncol = 3)
> rbind(Matrix1, Matrix2)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
[3,]    1    6   11
[4,]    2    7   12
[5,]    3    8   13
[6,]    4    9   14
[7,]    5   10   15

# Bei Spaltenkombinationen bedarf es der gleichen Anzahl an Zeilen.
> Matrix3 = matrix(7:16, nrow = 2 , ncol = 5)
> cbind(Matrix1,Matrix3)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1    3    5    7    9   11   13   15
[2,]    2    4    6    8   10   12   14   16
```

## 4.3 Faktoren

Der Nutzen von Faktoren ist zu Anfang oft nicht gleich ersichtlich, tatsächlich sind sie jedoch sehr nützlich. Faktoren eignen sich sehr gut, um Daten zu kategorisieren oder auch zu filtern. Dabei wird ein Faktor mit der *factor()*-Funktion erzeugt und erwartet einen Eingabevektor vom Typ *character* oder *numeric*.

```
> fac = factor(c("blue","green","yellow","green"))
> fac
[1] blue   green  yellow green
Levels: blue green yellow
```

Im obigen Beispiel wurde ein Faktor mit einem Vektor vom Typ *character* gespeist. Der entstehende Faktor besitzt die gleiche Länge wie der Vektor, ist aber zusätzlich die Elemente des Vektors Schritt für Schritt durchgegangen und hat gleichartige Elemente zu einem *Level*, zu einer Kategorie, zusammengefasst. Da beim Funktionsaufruf die *Levels* nicht spezifiziert wurden, hat die Funktion sich selbst aus dem Eingabevektor die *Levels* gebildet. Soll nur nach bestimmten *Levels* kategorisiert werden, ist dies wie folgt möglich:

```
> fac = factor(c("blue","green","yellow","green"), levels = c("green"))
> fac
[1] <NA>  green <NA>  green
Levels: green
```

Jedes Element, welches nun nicht eindeutig einem *Level* zugeordnet werden kann, wird mit <**NA**> markiert - der Faktor "kennt"dieses Element nicht. Intern werden die Kategorien effizient auf Integer abgebildet. Bei Betrachtung des ersten Beispiels würde dies entsprechend bedeuten:
"*blue*" $\rightarrow 1$
"*green*" $\rightarrow 2$
"*yellow*" $\rightarrow 3$
"*green*" $\rightarrow 2$

Um den Nutzen von Faktoren weiter zu verdeutlichen, sei folgendes Beispiel genannt. Angenommen, man ziehe aus einem Topf zufällig eine Karte mit einer Zahl von 1 bis 10, notiert die Zahl und legt die Karte wieder zurück in den Topf. Dies wiederholt man 1000 mal, hat daraufhin sämtliche Ziehungen in einem Vektor vorliegen und interessiert sich nun für die Verteilung. Hier können Faktoren helfen:

```
> fac = factor(sample(1:10, 1000, replace = TRUE), levels = c(1:10))
> table(fac)
fac
  1   2   3   4   5   6   7   8   9  10
 90  97 107  91 101 100 121  77 113 103
```

Eine weitere Eigenschaft von Vektoren ist die Ordnung. Damit lässt sich aufzeigen, dass es zwischen den einzelnen *Level* eine Größenordnung gibt.
Verständlich wird dies, wenn man sich vorstellt Daten über T-Shirt Größen zu besitzen. *L* kommt vor *M* und *S* im Alphabet, folglich würde der Faktor die *Levels* so anordnen, obwohl dies im Kontext der T-Shirt Größen falsch ist. Natürlich lässt sich die Reihenfolge durch das explizite Angeben der *Levels* ändern, aber damit wird immer noch nicht ausgedrückt, dass es eben auch einen Unterschied zwischen den Größen gibt. Helfen tut hier ein sogenannter **ordered factor**.

```
> sizes = c("S","M","L","M","S","L","M","L","S")
> fac = factor(sizes, levels = c("S","M","L"), ordered = TRUE)
> fac
[1] S M L M S L M L S
Levels: S < M < L
```

Daraus ergibt sich nun die Möglichkeit, logische Abfragen bezüglich des Größenverhältnisses zweier Elemente zu machen:

```
# Ist M kleiner als S?
> fac[2] < fac[1]
[1] FALSE
```

## 4.4 Data Frames

Data Frames sind eine grundlegende und oft benutzte Datenstruktur in R. Man kann sich einen Data Frame als eine Matrix mit benannten Spalten und Zeilen vorstellen. Demnach ist ein Data Frame also zweidimensional.
Was Data Frames nun von den bisher vorgestellten Datenstrukturen unterscheidet ist die Tatsache, dass sie mehrere Datentypen erlauben - spaltenweise. Daten lassen sich so gut strukturieren und darstellen.

Um einen Data Frame zu erzeugen, greift man auf die *data.frame()*-Funktion zurück und übergibt als Parameter Vektoren mit entsprechenden Daten. Dabei ist das erste Argument die erste Spalte des Data Frames, das zweite Argument die zweite Spalte usw.

```
# Zuerst werden Eingabeargumente benoetigt:
> integer = c(1L,2L,3L,4L)
> character = c("a","b","c","d")
> logical = c(TRUE,FALSE,TRUE,FALSE)

# Data Frame erzeugen mit entsprechenden Eingabeargumenten:
> df = data.frame(integer, character, logical)
> df
  integer character logical
1       1         a    TRUE
2       2         b   FALSE
3       3         c    TRUE
4       4         d   FALSE
```

Manchmal ist es sinnvoll, die Beschriftung der Spalten und Zeilen nicht dem Automatismus der *data.frame()*-Funktion zu überlassen. Abhilfe schaffen hier zwei dafür zuständige Funktionen.

```
> colnames(df) = c("c1","c2","c3")
> rownames(df) = c("r1","r2","r3","r4")
> df
   c1 c2    c3
r1  1  a  TRUE
r2  2  b FALSE
r3  3  c  TRUE
r4  4  d FALSE
```

Ähnlich wie bei Matrizen kann es auch bei Data Frames manchmal die Anforderung geben, den jeweiligen Data Frame um eine Spalte oder eine Zeile zu erweitern. Die Spaltenerweiterung ist trivial, denn eine Spalte hat immer nur Elemente eines Datentyps und die Eigenschaften des Vektors erzwingen diesen Umstand.

```
# Die cbind-Funktion realisiert die Spaltenerweiterung.
> df = cbind(df, test2 = c(10,11,12,13))
> df
  ich bin  test test2
5   1   a  TRUE    10
6   2   b FALSE    11
7   3   c  TRUE    12
8   4   d FALSE    13
```

Die Zeilenerweiterung ist etwas kniffliger, sind doch unter Umständen in einer Zeile Elemente verschiedener Datentypen. Es hat sich als praktikabel erwiesen, für jenen Fall eben einen neuen Data Frame zu erzeugen, die benötigten Daten als einelementige Vektoren zu übergeben und mit der *rbind()*-Funktion an das bestehende Data Frame zu heften. Außerdem muss darauf geachtet werden, dass die Spaltenbeschriftungen identisch sind.

```
# Die rbind-Funktion realisiert die Zeilenerweiterung.
> neu = data.frame(ich = 5, bin ="e",test =TRUE,test2 = 14)
> df = rbind(df, neu)
> df
      ich  bin  test test2
5       1    a  TRUE    10
6       2    b FALSE    11
7       3    c  TRUE    12
8       4    d FALSE    13
5       5    e  TRUE    14
# Nun muesste man noch ggf. den Zeilennamen der fuenften Zeile anpassen.
```

Es ist nun leicht vorstellbar, dass auch das Auslesen, Löschen und Einfügen von Daten in den Data Frame anders zu handhaben ist als bisher. Die Adressierung ist jedoch, wie auch zuvor, durch Indexierung realisiert.

An dieser Stelle wir der Übersichtlichkeit halber ein neuer Data Frame erzeugt.
Mit dem Argument **stringsAsFactors = FALSE** wird sichergestellt, dass *characters* eben nicht als Faktor abgespeichert werden. Das ermöglicht die spätere Modifikation von Spalten, Zeilen oder einzelnen Feldern des Data Frames, und zwar auch mit *characters*, die bei der Data-Frame-Erzeugung der Funktion noch nicht bekannt waren und durch die sonst übliche Faktorisierung von *character*-Symbolen nicht als *Level* geführt werden würden und demnach unbekannt wären.

```
> df = data.frame(integer = c(7,8,9),
character = c("a","b","c"),
logical = c(TRUE,FALSE,TRUE),
complex = c(2+2i, 4-3i, 1+1i),
stringsAsFactors = FALSE)
> df
  integer character logical complex
1       7         a    TRUE    2+2i
2       8         b   FALSE    4-3i
3       9         c    TRUE    1+1i

# Adressieren und Modifizieren von Zeilen:
> df[1,]
  integer character logical complex
1       7         a    TRUE    2+2i

# Auf diese Weise lassen sich ganze Zeilen ansprechen.
# Nun kann ganz einfach der Inhalt veraendert werden:
> df[1,] = list(10,"m",FALSE,3-3i)
> df
  integer character logical complex
1      10         m   FALSE    3-3i
2       8         b   FALSE    4-3i
3       9         c    TRUE    1+1i

# Adressieren und Modifizieren von Spalten:
> df[,1]
[1] 10  8  9

# Auf diese Weise lassen sich ganze Spalten ansprechen.
# Nun kann ganz einfach der Inhalt veraendert werden:
> df[,1] = c(11,12,13)
> df
  integer character logical complex
1      11         m   FALSE    3-3i
2      12         b   FALSE    4-3i
3      13         c    TRUE    1+1i
```

Da Data Frames sehr oft benutzt werden existieren unzählige Funktionen. Hier wurden nur grundsätzliche Operationen beschrieben, die ein tieferes Verständnis überhaupt erst ermöglichen.

## 4.5 Arrays

Ein Array ist eine Datenstruktur, deren Reichweite sich nicht auf die Programmiersprache R beschränkt, sondern die nahezu in allen Programmiersprachen genutzt wird und dementsprechend im Allgemeinen ein Thema der Informatik ist. Gängigen Definitionen zufolge ist ein Array eine zusammenhängende Sequenz von Elementen, wobei der Zugriff auf die Elemente durch Indizes erfolgt und alle Elemente den gleichen Datentyp haben. Auffällig ist, dass diese Eigenschaften auch auf Vektoren und Matrizen zutreffen, und

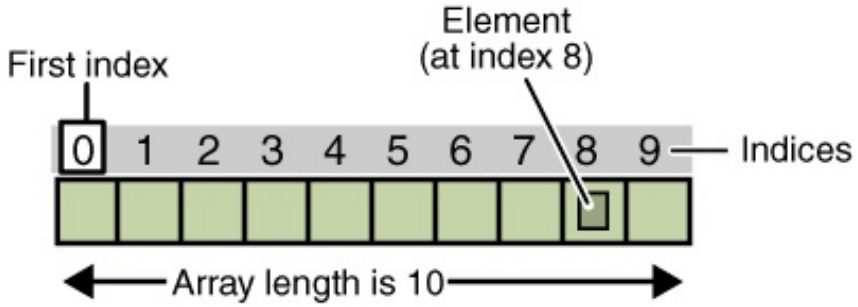


Abbildung 1: Aufbau eines Arrays [Pau]

tatsächlich gibt es bis auf eine ggf. andere interne Verarbeitung keinen Unterschied zwischen diesen Datenstrukturen. Vereinfacht lässt sich auch sagen, dass Vektoren und Matrizen konkrete Ausprägungen von Arrays sind. Ein Array hat eine theoretische Dimensionalität von $\boldsymbol{n}$, wobei der Lesbarkeit wegen meist nicht allzu viele Dimensionen benutzt werden.
Um ein Array zu erzeugen, bedient man sich der *array()*-Funktion und kann durch Parameterübergabe noch die Daten und die Dimensionalität spezifizieren.

```
> array = array(1:24  , dim = c(6,2,2))
> array
, , 1

     [,1] [,2]
[1,]    1    7
[2,]    2    8
[3,]    3    9
[4,]    4   10
[5,]    5   11
[6,]    6   12

, , 2

     [,1] [,2]
[1,]   13   19
[2,]   14   20
[3,]   15   21
[4,]   16   22
[5,]   17   23
[6,]   18   24

# Als Eingabe erhaelt dieses Array die Zahlen 1 bis 24.
# Das Array hat 6 Zeilen, 2 Spalten und eine "Tiefe" von 2.

# Zugriff ueber Indizes:
> array[2,1,1]
[1] 2
```

Bedingt durch die bereits oben beschriebene Gleichheit gelten für Arrays auch alle Eigenschaften und Operationen von Vektoren und Matrizen.

## 4.6 Listen

Auch für Listen gilt, dass sie sich nicht nur auf die Programmiersprache R beschränken, sondern konzeptuell auch weiter verbreitet sind. In R stellen sie die wohl komplexeste Datenstruktur dar. Listen können Vektoren, Matrizen, Faktoren, Data Frames, Arrays oder auch wiederum Listen aufnehmen. Unterschiedliche Datentypen spielen für Listen ebenfalls keine Rolle. Für die Erzeugung einer Liste greift man auf die *list()*-Funktion zurück und übergibt, je nach Belieben, gleich die jeweiligen Elemente oder Datenstrukturen.

```
# Erstellen einiger Beispiel-Daten:
> a = c(1,2,3,4)
> b = matrix(1:4, nrow = 2, ncol = 2)
> c = array(1:8, dim = c(2,2,2))

# Nun erzeugen wir unsere Liste:
> list = list(a,b,c,10)
> list
[[1]]
[1] 1 2 3 4

[[2]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

[[3]]
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8


[[4]]
[1] 10

## Zugriff
# Der Zugriff wird durch Indizes ermoeglicht.
# Beispiel: Zugriff auf die Matrix.
> list[[2]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

# Sollen Elemente in der Matrix geaendert werden,
# so muessen darueberhinaus die jeweiligen Indizes angeben werden.
> list[[2]][1,1]
[1] 1
```

```
# An diese Stelle soll nun eine 10 geschrieben werden:
> list[[2]][1,1] = 10
> list
[[1]]
[1] 1 2 3 4

[[2]]
     [,1] [,2]
[1,]   10    3
[2,]    2    4

[[3]]
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8


[[4]]
[1] 10
```

Listen bieten folglich umfassende Möglichkeiten, um Daten, die sich aus den verschiedensten Datentypen oder Datenstrukturen zusammensetzen, zueinander zu führen und in einer großen Struktur zu bündeln.

## 4.7 Ergänzungen durch Pakete

Neben den offiziellen Datenstrukturen in R existieren noch weitere Möglichkeiten, Daten in R zu organisieren. Zahlreiche Erweiterungen, im folgenden Pakete genannt, wurden entwickelt und verfügbar gemacht, so dass es mitunter verschiedenste nutzbare Implementationen weiterer abstrakter Datentypen gibt. Im Folgenden werden Implementationen der ADT's Stack, Queue und Tree anhand eines ausgewählten Paketes vorgestellt und deren Möglichkeiten aufgezeigt.

### 4.7.1 Stack

Der *Stack*, im deutschen auch als *Stapelspeicher* bezeichnet, ist eine Datenstruktur, welche nach dem LIFO-Prinzip (Last in, First out) arbeitet. Es können Elemente auf den Stack gelegt werden, das oberste Element betrachtet oder gar entfernt werden. Charakteristika des Stacks ist die begrenzte Sichtweite und der eingeschränkte Zugriff. Liegt ein benötigtes Element weiter unten im Stack, müssen zuvor alle darüber liegenden Elemente entfernt werden. Implementieren lässt sich ein Stack zum Beispiel mit einer

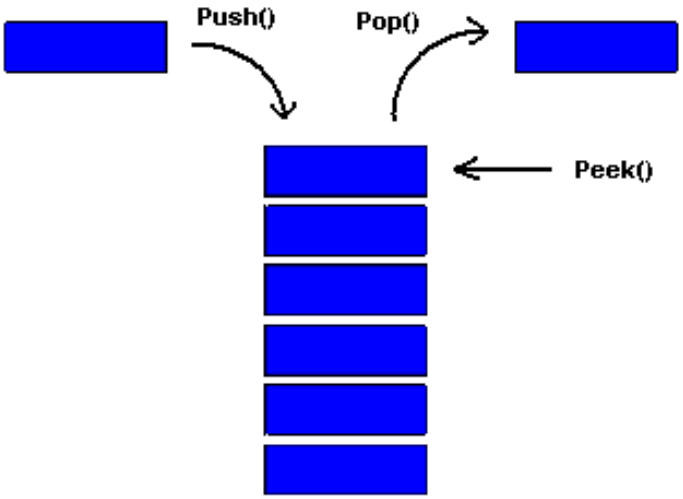


Abbildung 2: Funktionsweise eines Stacks [Wir]

verketteten Liste, wo die einzelnen Elemente jeweils Referenzen auf die Elemente über bzw. unter ihnen haben.
Es wurde die Entscheidung getroffen, die Implementation ***rstackdeque*** zu benutzen, da sie leicht zu bedienen ist und intern den Stack als Liste repräsentiert. Nach Import in die jeweilige R-Entwicklungsumgebung lassen sich die grundlegenden Funktionen des Stacks schnell nachstellen:

```
# Erzeugung
stack <- rstack()
# Ein leerer Stack wurde erzeugt.

# Einfuegen von Elementen:
> stack <- insert_top(stack, 1)
> stack <- insert_top(stack, 2)
> stack <- insert_top(stack, "a")
> stack
An rstack with 3 elements.
: chr "a"
: num 2
: num 1
```

```
# Loeschen von Elementen:
> stack <- without_top(stack)
> stack
An rstack with  2  elements.
: num 2
: num 1

# Das oberste Stack-Element in eine Variable kopieren, aber nicht loeschen:
> top <- peek_top(stack)
> stack
An rstack with  2  elements.
: num 2
: num 1
> top
[1] 2

# Empty-Funktion, um zu pruefen, ob der Stack leer ist:
> empty(stack)
[1] FALSE
> stack <- without_top(stack)
> stack <- without_top(stack)
> empty(stack)
[1] TRUE

# Alle elementaren Stack-Operationen sind mit diesem Paket realisiert.
```

### 4.7.2 Queue

Die klassische *Queue*, im deutschen auch als *Warteschlange* bezeichnet, ist eine Datenstruktur, welche nach dem FIFO-Prinzip (First in, First out) arbeitet. So ist es möglich das erste Element aus der Warteschlange zu betrachten oder gar zu entfernen sowie neue Elemente ans Ende der Warteschlange einzureihen. Auch Warteschlangen werden häufig als verkettete Listen implementiert, wo die einzelnen Elemente jeweils Referenzen auf ihre Nachbarn in der Warteschlange besitzen.
Das Paket ***rstackdeque*** bietet nicht nur eine Stack-Implementation an, sondern auch eine bzw. mehrere

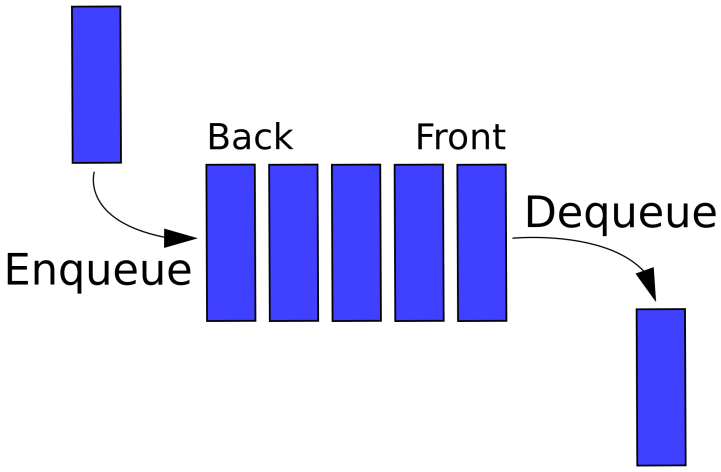


Abbildung 3: Funktionsweise einer Queue [Wikb]

Implementationen der Queue. Im Folgenden wird eine einfache Queue vorgestellt, wo also jeweils das Einfügen und das Löschen von Elementen nur an **einem** Ende der Warteschlange möglich ist.

```
# Erzeugung:
queue = rpqueue()

# Moechte man eine doppelt-verkettete Queue, sollte man rdeque() nutzen.

# Einfuegen von Elementen:
> queue = rpqueue()
> queue <- insert_back(queue, "a")
> queue <- insert_back(queue, "b")
> queue <- insert_back(queue, "c")
> queue
A queue with 3 elements.
Front:
$: chr "a"
$: chr "b"
$: chr "c"

# Loeschen von Elementen:
> queue <- without_front(queue)
> queue
A queue with 2 elements.
Front:
$: chr "b"
$: chr "c"

# Das vorderste Element einer Queue in eine Variable kopieren,
# aber nicht loeschen:
> front <- peek_front(queue)
> queue
A queue with 2 elements.
Front:
$: chr "b"
$: chr "c"
> front
[1] "b"

# Empty-Funktion, um zu pruefen, ob die Queue leer ist:
> empty(queue)
[1] FALSE
> queue <- without_front(queue)
> queue <- without_front(queue)
> empty(queue)
[1] TRUE

# Alle elementaren Queue-Operationen sind mit diesem Paket realisiert.
```

### 4.7.3 Tree

Ein *Baum* (eng. *tree*) ist eine zentrale Datenstruktur in der Informatik, von der es diverse Ausprägungen für die unterschiedlichsten Anforderungen gibt. Allgemein eignen sich Bäume gut, um hierarchische Strukturen abzubilden. Ein sinnvoll aufgebauter Baum bietet darüber hinaus durch seine speziellen Eigenschaften die Möglichkeit, Elemente wesentlich schneller zu finden oder an der richtigen Stelle einzufügen als andere Datenstrukturen. Die Elemente des Baumes bezeichnet man als *Knoten*, wobei Knoten mit Referenzen auf andere Knoten die *Eltern* dieser Knoten sind, und folglich die referenzierten Knoten als die *Kinder* der Eltern bezeichnet werden. Einen Knoten ohne ein Kind nennt man ein *Blatt*, ein Knoten ohne Eltern wird *Wurzel* genannt - die Definition des Baumes fordert überdies, dass es in einem Baum nur eine Wurzel geben darf. Die Verbindungen zwischen den Knoten werden als *Kanten* bezeichnet. Spezielle

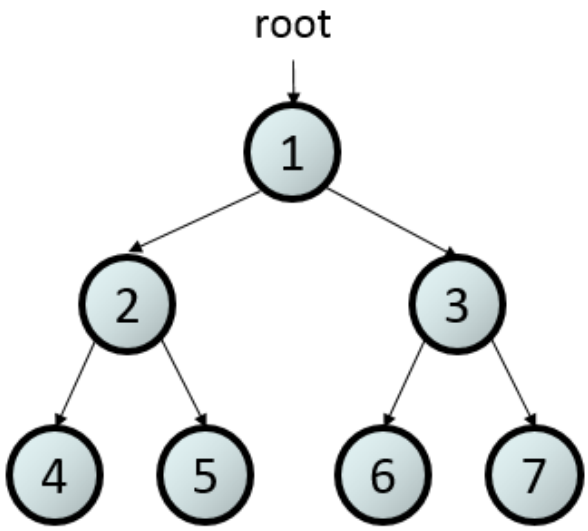


Abbildung 4: Aufbau eines Baumes [Ste]

Ausprägungen des Baumes sind zum Beispiel binäre Suchbäume. In Programmiersprachen werden Bäume meist als verkettete Listen implementiert.
Eine gute Implementation des Baumes findet sich mit dem Paket ***data.tree***, weshalb es im Folgenden vorgestellt wird. Nach Import in die jeweilige R-Entwicklungsumgebung lassen sich die grundlegenden Funktionen des Baumes schnell nachstellen:

```
# Knoten erzeugen:
> fbinformatik <- Node$new("Fachbereich Informatik")

# Der Knoten fbinformatik soll nun Kinder haben.
# Unterknoten erzeugen:
> bachelor <- fbinformatik$AddChild("Bachelorstudiengaenge")
> master <- fbinformatik$AddChild("Masterstudiengaenge")

# Dies laesst sich nun beliebig weiterfuehren.
# Mit der print-Funktion kann man sich anzeigen lassen,
# welchen Baum ein bestimmer Knoten aufspannt:
> print(fbinformatik)
levelName
1 Fachbereich Informatik
2  --Bachelorstudiengaenge
3  --Masterstudiengaenge
```

```
# Zu Knoten lassen sich beliebig viele Attribute zuweisen:
> bachelor$studentenanzahl <- 500
> master$studentenanzahl <- 100

# Man kann der Print-Funktion neben dem Knoten auch die Attribute uebergeben
# welche auch angezeigt werden sollen:
> print(fbinformatik, "studentenanzahl")
levelName studentenanzahl
1 Fachbereich Informatik                NA
2  --Bachelorstudiengaenge              500
3  --Masterstudiengaenge                100

# Auf Knoten lassen sich auch Funktionen anwenden.
# In diesem Beispiel scheint es sinnvoll, dass die Studentenanzahl
# des Fachbereichs die Summe von Master- und Bachelorstudenten ist:
> fbinformatik$studentenanzahl <- Aggregate(node = fbinformatik,
attribute = "studentenanzahl",
aggFun = sum)
> print(fbinformatik, "studentenanzahl")
levelName studentenanzahl
1 Fachbereich Informatik                600
2  --Bachelorstudiengaenge              500
3  --Masterstudiengaenge                100

# Darueberhinaus existieren noch etliche weitere Funktionen, z.B.:
# Suchen von Knoten in einem Baum
# boolsche Abfragen wie isLeaf / isRoot

# Diese finden sich alle ausfuehrlich beschrieben
# in der Anleitung zu diesem Paket.
```

# 5 Zusammenfassung

Nach Vorstellung aller Datenstrukturen in R sowie einiger, durch Pakete importierter Implementationen elementarer abstrakter Datentypen lässt sich ein Fazit ziehen und auf wichtige Punkte hinweisen. Zweifelsohne sind Vektoren und Data Frames viel benutzte Datenstrukturen in R. Vektoren werden nicht

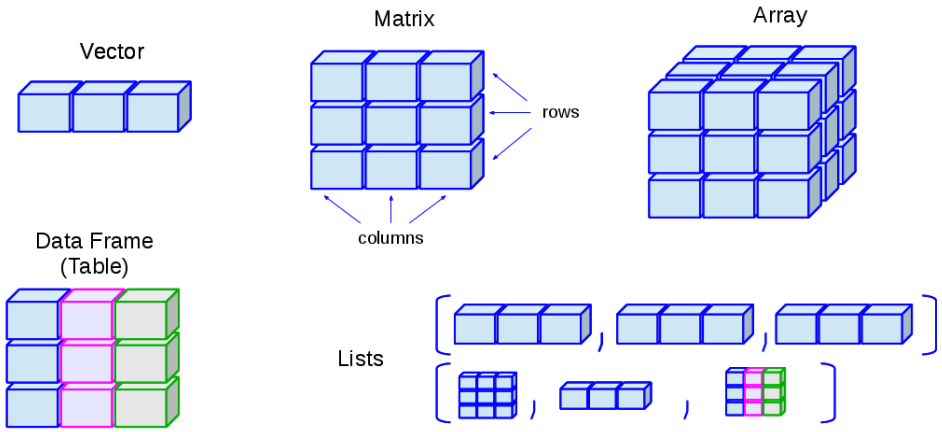


Abbildung 5: Übersicht über die Datenstrukturen in R [CC]

nur zur simplen Verwaltung von Elementen benutzt, sondern oft auch als Funktionsargumente erwartet, während Data Frames insbesondere im Kontext der Bearbeitung von Datensätzen unerlässlich sind und eine komfortable Daten-Handhabung ermöglichen. Es ist auch aufgefallen, dass viele Funktionen als Argumente bestimmte Datentypen voraussetzen. Dies gilt es bei der Benutzung zu beachten.
Ferner stellen die Bordmittel von R noch lange nicht das Ende der Möglichkeiten dar, denn durch unzählige Pakete lässt sich weitere Funktionalität importieren und der Workflow verbessern.

# Literatur


[Bla] Black, Kelly: *Basic Operations and Numerical Descriptions.* http://www.cyclismo.org/tutorial/R/basicOps.html

[Car] Carpentry, Software: *Understanding basic data types in R.* http://nicercode.github.io/2014-02-13-UNSW/lessons/01-intro_r/data-structures.html

[CC] Ceballos, Maite ; Cardiel, Nicolás: *Data structure: Data structure types.* http://venus.ifca.unican.es/Rintro/_images/dataStructuresNew.png

[Glu16] Glur, Christoph: *Introduction to data.tree.* https://cran.r-project.org/web/packages/data.tree/vignettes/data.tree.html. Version: 2016

[HE93] Hermann Engesser, Volker C.: *Informatik : ein Sachlexikon für Studium und Praxis.* Mannheim ; Leipzig ; Wien ; Zürich : Dudenverl., 1993

[IDR] IDRE: *R Learning Module: Factor variables.* http://www.ats.ucla.edu/stat/r/modules/factor_variables.htm

[ITW] ITWissen: *Allg. IT-Grundlagen: Daten.* http://www.itwissen.info/definition/lexikon/Daten-data.html

[Kir03] Kirk, Alexander: *Datenstrukur: Die Art, wie Daten im Speicher abgelegt werden.* http://www.computerlexikon.com/was-ist-datenstruktur. Version: 2003

[Mal15] Malecki, Mike: *Stylized concatenation of data.frames or ffdfs.* https://cran.r-project.org/web/packages/Stack/Stack.pdf. Version: 2015

[O'N15] O'Neil, Shawn T.: *Persistent Fast Amortized Stack and Queue Data Structures.* https://cran.r-project.org/web/packages/rstackdeque/rstackdeque.pdf. Version: 2015

[Pau] Paul, Javin: *How to Compare Two Arrays in Java - String, Integer Array Example.* http://www.java67.com/2014/05/how-to-compare-two-arrays-in-java-string-int-example.html

[RP] R-Project: *An Introduction to R.* https://cran.r-project.org/doc/manuals/r-release/R-intro.html

[Ste] Stepp, Marty: *Lecture Preview: Binary Trees.* https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1162/preview-binarytree.shtml

[Wika] Wikipedia: *Abstrakter Datentyp.* http://tinyurl.com/jccgsyj

[Wikb] Wikipedia: *Queue (abstract data type).* https://en.wikipedia.org/wiki/Queue_(abstract_data_type)#/media/File:Data_Queue.svg

[Wir] Wirtschaftsingenieur, Der: *Stack Datenstruktur in C-Sharp.* http://www.der-wirtschaftsingenieur.de/index.php/stack-datenstruktur-in-c/

[Yaua] Yau, Chi: *Data Frame.* http://www.r-tutor.com/r-introduction/data-frame

[Yaub] Yau, Chi: *Matrix Construction.* http://www.r-tutor.com/r-introduction/matrix/matrix-construction