3.12 Slices

Auf Seite 38 hatten wir eine Lösung mit `perl` vorgestellt, um die **Liste der „echten" Benutzer** aus /etc/passwd zu extrahieren:
▶▶ `view perl/passwd_realnames.pl` :

```
#! /usr/bin/perl -w -n
# The -n flag above loops over the input (= /etc/passwd) line by line.
use diagnostics;
# In order to avoid warnings about unused variables, we declare them
# explicitly. (The latter will be explained later.)
my ($login, $passwd, $uid, $gid, $realname, $rest);
($login, $passwd, $uid, $gid, $realname, $rest) = split /:/ ;
if($uid >= 500 and $uid <= 599) {
  print "$realname\n";
}
```

Nachteil: Vier **Dummy-Variablen** waren notwendig.

Man kann auch ohne die Dummy-Variablen auskommen. Wenn bei einer **Listen-Zuweisung** `undef` vorkommt, wird der entsprechende **Wert** einfach **ignoriert**:
▶▶ `view perl/passwd_realnames2.pl` :

```
#! /usr/bin/perl -w -n
# The -n flag above loops over the input (= /etc/passwd) line by line.
use diagnostics;
(undef, undef, $uid, undef, $realname, undef) = split /:/ ;
if($uid >= 500 and $uid <= 599) {
  print "$realname\n";
}
```

▶▶ `./passwd_realnames2.pl < ../sed/passwd`

Nachteil: Man muß ggf. **viele** `undef`s **abzählen**.

Besser: ein **Listen-Slice** (im **skalaren Kontext**):
▶▶ `view perl/passwd_realnames3.pl` :

```
#! /usr/bin/perl -w -n
# The -n flag above loops over the input (= /etc/passwd) line by line.
use diagnostics;
$uid = (split /:/)[2];
$realname = (split /:/)[4];
if($uid >= 500 and $uid <= 599) {
  print "$realname\n";
}
```

▶▶ `./passwd_realnames3.pl < ../sed/passwd`

Die **runden Klammern** um den Listen-liefernden Ausdruck sind hier **notwendig**.

Die resultierende **Liste** wird beim Indizieren **wie ein Array behandelt**.

Nachteil dieser Lösung: **Zwei Aufrufe** von `split`.

Noch **besser**: ein Listen-Slice im **Listen-Kontext**:

▶▶ `view perl/passwd_realnames4.pl` :

```
#! /usr/bin/perl -w -n
# The -n flag above loops over the input (= /etc/passwd) line by line.
use diagnostics;
($uid, $realname) = (split /:/)[2, 4];
if($uid >= 500 and $uid <= 599) {
  print "$realname\n";
}
```

▶▶ `./passwd_realnames4.pl < ../sed/passwd`

Man darf **bei** der **Indizierung alles verwenden**, was sonst auch erlaubt ist:

▶▶ `view perl/list-slice.pl` :

```
#! /usr/bin/perl -w
use diagnostics;
@namen = qw{ null eins zwei drei vier fuenf sechs sieben acht neun MZH };
@raum = ( @namen )[-1, 5, 3, 0, 0];
print "Der Kurs findet statt in @raum.\n";
```

▶▶ `./list-slice.pl`

Wenn man **Elemente** nicht aus einer allgemeinen Liste, sondern **speziell aus einem Array** holt, gibt es eine **einfachere Notation**, das **Array-Slice**:

▶▶ `view perl/array-slice.pl` :

```
#! /usr/bin/perl -w
use diagnostics;
@namen = qw{ null eins zwei drei vier fuenf sechs sieben acht neun MZH };
@raum = @namen[-1, 5, 3, 0, 0];
print "Der Kurs findet statt in @raum.\n";
```

▶▶ `./array-slice.pl`

Die **runden Klammern** können **weggelassen** werden.

Achtung: Diese Indizierungs-**Notation** ist **anders als** die Notation **Dollar-Name-Indexausdruck**! Jetzt haben wir einen **Klammeraffen statt** eines **Dollar**-Zeichens.

Regel:

- „**Dollar**-irgendwas“: Liefert **Skalar**
- „**Klammeraffe**-irgendwas“: Liefert **Liste**

Man darf auch auf Array-Slices **schreiben**:

▶▶ `view perl/passwd_realnames5.pl` :

```
#! /usr/bin/perl -w -n
# The -n flag above loops over the input (= /etc/passwd) line by line.
use diagnostics;
chomp;
@zeile = split /:/;
if ($zeile[4] eq 'B. Mueller') {
  @zeile[4, -1] = ('B. Mueller-Luedenscheid', '/bin/tcsh');
}
$zeile = join ':', @zeile;
print "$zeile\n";
```

▶▶ `./passwd_realnames5.pl < ../sed/passwd`

Dieses Programm kopiert die Eingabe in die Ausgabe und **ändert** dabei den **Nachnamen** von B. Müller **und** außerdem auch die **Login-Shell**.

3.13 Grundlagen der Ein- und Ausgabe

3.13.1 Der Diamant-Operator

Bisher hatten unsere Programme **immer von STDIN** gelesen. Die normalen Unix-Tools wie **cat, sed, grep** sind da **flexibler**: **Wenn** man **nichts** angibt, lesen sie auch von **STDIN,** aber **wenn** man einen **Dateinamen** angibt, lesen sie **aus** dieser **Datei**. Wenn man **mehrere Dateinamen** angibt, lesen sie **alle** diese Dateien nacheinander.

Beispiele:

▶▶ `./grep split`

▶▶ `./grep split split-ws.pl`

▶▶ `./grep split *`

▶▶ `./grep split -`

Wenn man ein **Minus** angibt, **liest** das von **STDIN**.

Das gleiche Verhalten wie diese Unix-Tools erreicht man in `perl` mit dem Diamant-Operator anstelle von `<STDIN>`:

▶▶ `view perl/grep-poor.pl` :

```
#! /usr/bin/perl -w
use diagnostics;
while (<>) {
  if ( /split/ ) {
    print;
  }
}
```

▶▶ `./grep-poor.pl`

▶▶ `./grep-poor.pl split-ws.pl`

▶▶ `./grep-poor.pl *`

▶▶ `./grep-poor.pl -`

Es wird ohne Unterbrechung aus allen Dateien nacheinander gelesen. Falls man wissen will, aus welcher Datei man gerade liest, steht dies in der Variablen `$ARGV`:
▶▶ `view perl/grep-poor2.pl` :

```
#! /usr/bin/perl -w
use diagnostics;
while (<>) {
  if ( /split/ ) {
    print "$ARGV: $_";
  }
}
```

▶▶ `./grep-poor2.pl *`

3.13.2 Die Aufruf-Parameter

Die **Aufruf-Parameter** eines Skripts werden **beim Programmstart** im **Array** `@ARGV` abgelegt. **Danach** kann man das Array **beliebig benutzen**.

Der **Diamant-Operator** benutzt z.B. `@ARGV`.

Wir können unser **obiges grep-Skript erweitern**, indem wir den **Suchausdruck** aus `@ARGV` holen: ▶▶ `view perl/grep-poor3.pl` :

```
#! /usr/bin/perl -w
use diagnostics;
$muster = shift @ARGV;
if (! defined $muster) {
  print "Aufruf: grep-poor <muster> [<datei> ...]\n";
  exit 1;
}
while (<>) {
  if ( /$muster/ ) {
    print "$ARGV: $_";
  }
}
```

▶▶ `./grep-poor3.pl split *`

Nachdem der **Suchausdruck entfernt** worden ist, wird der **Rest** der Liste dem **Diamant-Operator** übergeben.

Man kann auf diese Weise auch **Kommandozeilen-Optionen** realisieren:

▶▶ `view perl/grep-poor4.pl` :

```
#! /usr/bin/perl -w
use diagnostics;
$verbose = 0;
if (($#ARGV >= 0) and $ARGV[0] eq "-v") {
  $verbose = 1;
  shift;
}
$muster = shift @ARGV;
if (! defined $muster) {
  print "Aufruf: grep-poor <muster> [<datei> ...]\n";
  exit 1;
}
while (<>) {
  if ( /$muster/ ) {
    if ($verbose) {
      print "$ARGV: ";
    }
    print;
  }
}
```

▶▶ `./grep-poor4.pl split *`

▶▶ `./grep-poor4.pl -v split *`

Anmerkung: Wenn man viele Kommandozeilenoptionen hat, dann empfiehlt es sich, ein fertiges perl-Modul aus der Standard-Distribution zu nehmen (perldoc Getopt::Long und perldoc Getopt::Std).

3.13.3 Formatierte Ausgabe mit printf

Die **aus C bekannte Funktion** printf gibt es auch in perl.

Erster Parameter: **Formatieranweisung**.
Restliche Parameter: Auszugebende **Werte**

Es sind im wesentlichen **genau alle Formatieranweisungen** erlaubt, die es **auch in C** gibt. Eine Auswahl:

▶▶ view perl/printf.pl :

```
#! /usr/bin/perl -w
use diagnostics;
printf "Eine Dezimalzahl: %d\n", 42;
printf "Eine Fliesskommazahl: %f\n", 3.14159;
printf "Eine Fliesskommazahl im wiss. Format: %e\n", 3.14159;
printf "Eine Fliesskommazahl im passenden Format: %g\n", 3.14159;
printf "Eine Fliesskommazahl im passenden Format: %g\n", 798000000000000;
printf "Eine Integer-Zahl im passenden Format: %g\n", 42;
printf "Eine String: %s\n", "Hallo";
printf "Eine Zahl in fester Spaltenbreite : '%10d'\n", 42;
printf "Ein String in fester Spaltenbreite: '%10s'\n", "Hallo";
printf "Ein String linksbuendig          : '%-10s'.\n", "Hallo";
printf "Eine Fliesskommazahl mit Breite und Genauigkeit: '%10.2f'\n",
       3.14159;
printf "Ein einzelnes Prozentzeichen: %%\n";
```

▶▶ ./printf.pl

3.14 Assoziative Felder (Hashes)

3.14.1 Was ist das?

Ein **Hash** (assoziatives Feld) ist **eine Art Feld**, dessen **Indizes nicht Zahlen** sind, **sondern** beliebige **Strings**.

Intern werden assoziative Felder **mit einer Hash-Tabelle implementiert**, daher der **Name**. Die Implementation ist **sehr effizient**, auch große Hashes sind kein Problem. Wie bei Listen gibt es **keine Längenbeschränkung**.

Die **Eigenschaften** eines Hashes ergeben sich aus dem bereits gesagten: Die **Werte** sind beliebige perl-**Skalare**, und sie müssen **nicht vom selben Typ** sein, wie bei Listen auch.

Mehrere **Einträge dürfen** den **gleichen Wert** haben. Die **Indizes** müssen dagegen **eindeutig** sein, zu einem Index-String kann es nur einen Wert geben.

3.14.2 Wozu ist das gut?

Beispiele:

- Zuordnung von **Login-Namen** zu **Real-Namen**
- Zuordnung von **Host-Namen** zu **IP-Adressen**
- Zuordnung von **IP-Adressen** zu **Host-Namen**
  (Hashes gehen nur in *eine* Richtung)
- Zuordnung von **Variablennamen** zur **Häufigkeit,** mit der sie in einem Programm auftauchen

3.14.3 Syntax

Zugriff auf **ein Feldelement**:

```
$hashvar{$stringvar}
```

Jetzt mit **geschweiften** Klammern, **sonst wie** bei **Feldern**. Kann zum **Lesen und** zum **Zuweisen** benutzt werden.

Wird von einem **Index gelesen**, auf den **noch nicht geschrieben** wurde, gibt es `undef` zurück, analog **wie bei Listen**.

Zugriff auf ein **ganzes Hash**:

```
%hashvar
```

Hashes können **in Listen verwandelt** werden, und **umgekehrt**. Damit kann man **Listen-Literale** auch für Hashes verwenden:

```
%hashvar = ("foo",  35, "bar",  12.4, "baz",  "hallo");
```

Die Liste enthält **Paare** von Schlüsseln und Werten.

Die **Reihenfolge der Paare** ist naturgemäß **egal**.

Wegen der Möglichkeit der Konversion in Listen können wir **Hashes leicht invertieren**:

```
%ip_addr = reverse %host_name;
```

Damit das sinnvoll klappt, **sollte** das **Hash eindeutig** (injektiv) sein. Sonst **gewinnt das letzte Paar** in der Liste. Da die Reihenfolge nicht definiert ist, ist das Ergebnis also **nichtdeterministisch**.

Damit man bei Listen-Literalen **nicht** die **Kommas abzählen** muß, darf man auch den **großen Pfeil** verwenden:

```
%hashvar = ("foo" => 35, "bar" => 12.4, "baz" => "hallo");
```

Der große Pfeil ist **einfach** eine **andere Schreibweise für** ein **Komma**. Das **gilt** für **ganz** perl.

Damit man das **schön zeilenweise** schreiben kann, darf am **Ende einer Liste** ein **überflüssiges Komma** stehen:

```
%hashvar = (
  "foo" => 35,
  "bar" => 12.4,
  "baz" => "hallo",
);
```

Ein **Beispiel** für eine **Anwendung**: Ein **Cross-Referenz-Programm**, das die **Anzahl der Vorkommen** der Variablennamen **zählt**:

cross-reference2.pl:

```
#! /usr/bin/perl -w
use diagnostics;
use English;
undef $INPUT_RECORD_SEPARATOR;
$prog = <STDIN>;
$prog =~ s/\#.*$//m;
@vars = split /[^\w%\@\$]+/, $prog;
foreach $var (@vars) {
  if ($var =~ /^\$(\w+)$/ ) {
    $scalars{$1}++;
  } elsif ($var =~ /^\@(\w+)$/ ) {
    $arrays{$1}++;
  } elsif ($var =~ /^%(\w+)$/ ) {
    $hashes{$1}++;
  }
}
print "The alphanumeric scalar variables are:\n";
while ( ($name, $num) = each %scalars) {
  print "$name: $num\n";
}
print "\nThe alphanumeric array variables are:\n";
while ( ($name, $num) = each %arrays) {
  print "$name: $num\n";
}
print "\nThe alphanumeric hash variables are:\n";
while ( ($name, $num) = each %hashes) {
  print "$name: $num\n";
}
```

Anmerkungen:

Indem wir `$INPUT_RECORD_SEPARATOR` **un-definieren**, lesen wir die **ganze Datei** als eine Zeile ein. (Kurzform wäre `$/`).

Beim **ersten Auftreten** eines **Variablennamens** greifen wir auf einen noch nicht vorhandenen Eintrag zu. Das ergibt `undef`. Wenn wir eine `undef`-Variable **inkrementieren**, **ergibt** das **1**.

Wir haben die `each`-**Funktion** benutzt, die wir **gleich** noch **erkären** werden.

Wir haben die **Suche** auf **Hash-Variablen** erweitert.

►► `./cross-reference2.pl < cross-reference2.pl`

3.14.4 Funktionen auf assoziativen Feldern

Es gibt eine Reihe von **Funktionen,** die **auf** einem **gesamten Hash** arbeiten.

Die Funktionen `keys` und `values` liefern die **Liste der Index-Strings** bzw. die **Liste der Werte**:

`keys-values.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
%realnames = (
  "brederek" => "Jan Bredereke",
  "mueller"  => "B. Mueller",
  "meyer"    => "L. Meyer",
  "schulze"  => "G. Schulze",
);
@logins = keys %realnames;
@realnames = values %realnames;
$logins = join ', ', @logins;
$realnames = join ', ', @realnames;
print "Login-Namen: $logins\n";
print "Real-Namen: $realnames\n";
```

►► `./keys-values.pl`

Frage: Wie sind die **Ergebnisse geordnet? Gar nicht**, und **anders sortiert** als sie eingegeben wurden! **Grund**: **Effiziente** interne **Speicherung**. Aber beide **Listen** sind **trotzdem korreliert**.

Im **skalaren Kontext** liefern `keys` und `values` die **Anzahl der Schlüssel-Wert-Paare** im Hash:

`keys-values2.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
%realnames = (
  "brederek" => "Jan Bredereke",
  "mueller"  => "B. Mueller",
  "meyer"    => "L. Meyer",
  "schulze"  => "G. Schulze",
);
$login_num = keys %realnames;
$realname_num= values %realnames;
print "Login-Namen: $login_num\n";
print "Real-Namen: $realname_num\n";
```

►► `./keys-values2.pl`

Ein **Hash im skalaren Kontext** liefert die boolsche Information, **ob** es **voll** ist **oder leer**:

`hash-scalar.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
%realnames = ();
if (%realnames) { print "Voll\n"; } else { print "Leer\n"; }
%realnames = (
  "brederek" => "Jan Bredereke",
  "mueller"  => "B. Mueller",
  "meyer"    => "L. Meyer",
  "schulze"  => "G. Schulze",
);
if (%realnames) { print "Voll\n"; } else { print "Leer\n"; }
%realnames = ();
if (%realnames) { print "Voll\n"; } else { print "Leer\n"; }
```

►► `./hash-scalar.pl`

Die Funktion `each` erlaubt, **über ein Hash** zu **iterieren**:

`hash-each.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
%realnames = (
  "brederek" => "Jan Bredereke",
  "mueller"  => "B. Mueller",
  "meyer"    => "L. Meyer",
  "schulze"  => "G. Schulze",
);
while ( ($login, $realname) = each %realnames ) {
  printf "%-8s: %s\n", $login, $realname;
}
```

►► `./hash-each.pl`

Jeder **Aufruf** von `each` **liefert** ein **Schlüssel-Wert-Paar** als **zweielementige Liste**.

Diese **Liste** wird hier **als Bedingung** in der `while`-Schleife verwendet, also in einem **skalaren Kontext**. Daher **liefert** sie die **Anzahl der Listenelemente**. Beim ersten Mal ist dies **2**, also **True**. **Am Ende des Hashes** liefert `each` die **leere Liste**, die Anzahl der Elemente ist **0**, und das ist **False**.

Die Information, **wo** wir **bei der Iteration** gerade **stehen**, ist in einem **Iterator** gespeichtert, der **zu jedem Hash dazugehört**. Wir können also über verschiedene Hashes gleichzeitig iterieren.

Der **Iterator** wird **zurückgesetzt,** wenn:

- das **Ende erreicht** wurde,
- eine **neue Liste** an das Hash **zugewiesen** wurde, oder
- `keys` oder `values` **aufgerufen** wurde.

Das ist normalerweise genau das, **was man erwarten** würde.

Wenn die **Reihenfolge** der Iteration **wichtig** ist, dann iteriert man mit Hilfe von `keys` und sortiert die Schlüssel:
`hash-iter-sort.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
%realnames = (
  "brederek" => "Jan Bredereke",
  "mueller"  => "B. Mueller",
  "meyer"    => "L. Meyer",
  "schulze"  => "G. Schulze",
);
foreach $login (sort keys %realnames) {
  printf "%-8s: %s\n", $login, $realnames{$login};
}
```

▶▶ `./hash-iter-sort.pl`

Oft ist es auch wichtig, **nach der Reihenfolge der Werte**, nicht der Schlüssel, eines Hashes **zu sortieren**.

Dies ist ein **Vorgriff** auf das Kapitel über Unterprogramme, und es verwendet fortgeschrittene Eigenschaften der `sort`-Funktion. Trotzdem ist dieses Sortieren so nützlich, daß es hier bereits eingeführt werden soll.

Man kann der `sort`-Funktion als Parameter ein **Vergleichs-Unterprogramm** mitgeben, um eine **eigene Sortierreihenfolge** zu definieren.

Für den **Spezialfall eines Hashes** kann das folgendermaßen geschehen:
`hash-iter-sort-value.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
%scores = (
  "Jan Bredereke" => 42,
  "B. Mueller"  => 27,
  "L. Meyer"    => 1,
  "G. Schulze"  => 234,
);
# a sorting subroutine: first param in $a, second param in $b
sub by_score { $scores{$b} <=> $scores{$a} }
foreach $name (sort by_score keys %scores) {
  printf "%-16s: %s\n", $name, $scores{$name};
}
```

▶▶ `./hash-iter-sort-value.pl`

Das **Sortier-Unterprogramm** hat sinnvollerweise einen **Namen**, der an die **Sortierreihenfolge** erinnert. Die Parameter werden anders als sonst in `$a` und `$b` übergeben. Der **„Raumschiffoperator“** `<=>` gibt **-1, 0 oder 1** zurück, je nachdem wie die beiden Zahlen sich zueinander verhalten. (**Für Strings** heißt das Gegenstück `cmp`.) Da wir `$b` mit `$a` vergleichen, haben wir eine **umgekehrte Sortierreihenfolge**.

Mit der Funktion `exists` kann man prüfen, **ob** es **zu einem Schlüssel überhaupt** einen **Eintrag** im Hash gibt:
`hash-exists.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
%user_ids = (
  "root"     => 0,
  "brederek" => 501,
  "mueller"  => undef,
);
print "Login     if(\$hash{\$key}) if(defined \$hash{\$key}) ".
      "if(exists \$hash{\$key})\n";
foreach $login (keys %user_ids, "meyer") {
  printf "%-8s: %-15s %-23s %-22s\n",
    $login,
    ( $user_ids{$login}         ? "true" : "false" ),
    ( defined $user_ids{$login} ? "true" : "false" ),
    ( exists $user_ids{$login}  ? "true" : "false" );
}
```

►► `./hash-exists.pl`

Die **Nicht-Existenz** eines Schlüssels **ist etwas anderes** als ein `undef`-Wert für einen Schlüssel. (Und ein `undef`-Wert ist bekanntermaßen etwas anderes als ein False-Wert.)

Anmerkung: Wir haben eben den **bedingten Ausdruck** `( ? : )` verwendet, der **aus C** übernommen wurde. Er entspricht if-then-else, liefert aber einen Wert zurück.

Die Funktion `delete` **entfernt einen Schlüssel** aus einem Hash (und **ebenso** seinen **Wert**):
`hash-exists2.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
%user_ids = (
  "mueller"  => 502,
);
print "Login     if(defined \$hash{\$key}) if(exists \$hash{\$key})\n";
foreach $login ("brederek", "mueller") {
  printf "%-8s: %-23s %-22s\n",
    $login,
    ( defined $user_ids{$login} ? "true" : "false" ),
    ( exists $user_ids{$login}  ? "true" : "false" );
}
$user_ids{"brederek"} = undef;
delete $user_ids{"mueller"};
delete $user_ids{"schulze"};
print "Login     if(defined \$hash{\$key}) if(exists \$hash{\$key})\n";
foreach $login ("brederek", "mueller") { # NOT using (keys %user_ids) !
  printf "%-8s: %-23s %-22s\n",
    $login,
    ( defined $user_ids{$login} ? "true" : "false" ),
    ( exists $user_ids{$login}  ? "true" : "false" );
}
```

▶▶ `./hash-exists2.pl`

Man beachte, daß die Verwendung von `(keys %user_ids)` den **anderen Login** jeweils gar **nicht** mehr mit **ausgegeben** hätte.

Es gibt **keine Warnung**, **wenn** der **Schlüssel** gar **nicht** im Hash **vorhanden** war.

Interpolation in Doppel-Quote-Strings:
`hash-interp.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
%user_ids = (
  "mueller"  => 502,
);
$login = "mueller";
print "$login: $user_ids{$login}\n";
print "%user_ids\n";
```

▶▶ `./hash-interp.pl`

Einzelne Elemente eines Hashes werden **wie erwartet interpoliert**. Ein **ganzes Hash** wird **nicht interpoliert**, das würde auch fast nie Sinn machen.

3.15 Online-Aufgaben

3.15.1 Aufgabe: Einfaches Adreßbuch

Schreibe ein einfaches **Adreßbuch**. Man soll Namen und Telefonnummern **eingeben und abfragen** können, sowie Einträge auch **löschen** können. Außerdem soll man eine **sortierte Liste aller Einträge** ausgeben können.

3.16 Hausaufgaben

3.16.1 Aufgabe: Web-Log-Auswertung

In der Datei `httpd_access_log.txt` (auf den WWW-Seiten des Kurses) findet man ein (gekürztes) **Zugriffs-Log eines Web-Servers**. Sie enthält **nur Angriffsversuche** von Skripten und Würmern. Schreibe ein `perl`-Skript, das **für jeden zugreifenden Rechner** zählt, **wie oft** insgesamt zugegriffen wurde. Es soll eine **nach Häufigkeit sortierte Liste** ausgegeben werden.