Stem-and-Leaf-Displays — selbstgemacht

File: ms.rev
in: /home/wiwi/pwolf/R/aplpack
H. P. Wolf

Version: 2013-10-30, compiled: September 26, 2014, file: ms.rev

1 Einleitung

In diesem Papier wird eine eigene Umsetzung eines Stem-and-Leaf-Displays gewagt.[1] In der Tat enthielt der Weg der Programmierung einige Hürden, die inzwischen hoffentlich zum größten Teil übersprungen sind. Besondere Herausforderung sollte dabei in einem verständlichen Code sowie einer Auflistung von Tests zur Sicherstellung der gewünschten Funktionalität liegen.

2 Die Funktionsdefinition

2.1 Überblick

Der vorgestellte Vorschlag lehnt sich eng an *UREDA* (Hoaglin, Mosteller, Tukey, 1983: Understanding Robust and Exploratory Data Analysis) an. Haupteinsatzzweck wird in der Verwendung ohne weitere Parameter gesehen, jedoch sollten bei Unzufriedenheiten oder Sonderwünschen durch gezielte Setzungen Varianten erstellt werden können. Hierzu stehen folgende Argumente bereit:

1 ⟨*definiere Kurzkommentar* 1⟩ ≡ ⊂ 3

```
#####################################################################
#Description:                                                       #
#   stem.leaf  produces a stem-and-leaf-display of a data set       #
#                                                                   #
#Usage:                                                             #
#   stem.leaf(data)                                                 #
#   stem.leaf(data,unit=100,m=5,Min=50,Max=1000,                    #
#     rule.line=c("Dixon", "Velleman", "Sturges"),                  #
#     style=c("Tukey", "bare"), trim.outliers=TRUE, depths=TRUE,    #
#     reverse.negative.leaves=TRUE,na.rm=FALSE)                     #
#                                                                   #
#Arguments:                                                         #
#   data:      vector of input data                                 #
#   unit:      unit of leaves in: { ...,100,10,1,.1,.01,... }       #
#   m:         1, 2 or 5 -- 10/m=number of possible leaf digits     #
#   Min:       minimum of stem                                      #
#   Max:       maximum of stem                                      #
#   rule.line:   = "Dixon"    => number of lines <- 10*log(n,10)    #
#                = "Velleman" => number of lines <- 2*sqrt(n)       #
#                = "Sturges"  => number of lines <- 1 + log(n,2)    #
#   style:       = "Tukey"    => Tukey-like stem ( m = 2, 5 )       #
#   trim.outliers=TRUE        => outliers are printed absent  #
#   depths       =TRUE        => depths info is printed            #
#   reverse.negative.leaves=TRUE => neg.leaves are rev. sorted     #
```

[1] Hinweis von DT: *...aus Velleman/Hoaglin: ABC of EDA, Seite 15: It is easy to construct a Stem-and-Leaf-Display by hand... It is not nearly as easy to write a general computer program to produce Stem-and-Leaf-Displays.*

```
#Author:                                                            #
#   Peter Wolf 05/2003 (modified slightly by J. Fox, 20 July 03) #
#   rounding operation for comparing added 29 March 06           #
#   07/2008 NA-values are counted if na.rm==FALSE                #
#####################################################################
```

Das schwierigste Problem ist die Erstellung einer geeeigneten Skala. Ist die Skala gefunden, können die Daten als Blätter bzw. Extremwerte identifiziert und im Plot angebracht werden. Zum Schluß ist das Ergebnis geeignet auszugeben. Am 29.3.2006 wurde ein Rundungsproblem behoben.

2 ⟨*start* 2⟩ ≡
```
⟨definiere stem.leaf 3⟩
⟨definiere stem.leaf.backback 45⟩
```

3 ⟨*definiere* `stem.leaf` 3⟩ ≡ ⊂ 2, 45
```
## ms <-
stem.leaf <- function(data, unit, m, Min, Max,
     rule.line=c("Dixon", "Velleman", "Sturges"),
     style=c("Tukey", "bare"), trim.outliers=TRUE, depths=TRUE,
     reverse.negative.leaves=TRUE,na.rm=FALSE,printresult=TRUE){
  if(missing(data)){cat("Author:  Peter Wolf 05/2003,",
                        "(modified slightly by J. Fox, 20 July 03)",
                        "03/2006 additional rounding to prevent misclasification",
                        "07/2008 counting of NA's, 04/2009 improvement of rounding",
                        "syntax: stem.leaf(data.set)\n",sep="\n")
                    return("Warning: no data set found by stem.leaf")
  }
  ⟨checke Input 5⟩
  ⟨setze ggf. verb gemäß Debugging-Wunsch 33⟩
  ⟨definiere Kurzkommentar 1⟩
  ⟨generiere die Skala für den Plot 4⟩
  ⟨erstelle Stem-and-Leaf-Display 14⟩
  ⟨stelle Ergebnis zusammen 30⟩
}
```

2.2 Skala

Für die Skala wird zunächst gemäß der festgelegten Regel eine grobe Zeilenzahl für den Plot bestimmt. Dann wird der Bereich, den die Skala abdecken muß, grob mittels `boxplot` festgestellt, sofern keine Skalengrenzen beim Funktionsaufruf angegeben worden sind. Mit Hilfe des Skalenbereiches und der Zeilenzahl läßt sich die anzustrebende Größe des Bereiches ermitteln, den es mit einer Zeile abzudecken gilt. Diese Größe gilt es mittels passendem Stamm und passender Maserung umzusetzen. Da im Folgenden die Position des Dezimalpunktes für das Stem-and-Leaf-Display keine Rolle mehr spielen, können alle relevanten Variablen transformiert / normiert werden. Mit den groben Berechnungen und den verarbeiteten Sonderwünschen kann dann die endgültige Skala erstellt werden.

4 ⟨*generiere die Skala für den Plot* 4⟩ ≡ ⊂ 3
```
⟨stelle gemäß rule.line maximale Zeilenanzahl fest 6⟩
⟨ermittle mittels boxplot groben Skalenbereich 7⟩
⟨bestimme Intervalllänge und ggf. Faktor factor 9⟩
⟨berechne aus zeilen.intervall.laenge und factor Tickabstand 10⟩
⟨bestimme ggf. Maserung m 11⟩
⟨transformiere Daten 12⟩
⟨bestimme Skalenbereich 13⟩
```

Zunächst gilt es den Input zu checken. 080711: Zeile mit NA-Zählung `n.na` eingebaut.

5 ⟨*checke Input* 5⟩ ≡ ⊂ 3, 48
```
rule.line <- match.arg(rule.line)
style <- match.arg(style)
n.na  <- sum(is.na(data))
if(0<n.na){
```

```
  data <- data[!is.na(data)]
  if(na.rm){ # data<-data[!is.na(data)]
    print("Warning: NA elements have been removed!!")
  }else{
#   data[is.na(data)] <- mean(data,na.rm=TRUE)
#   print("Warning: NA elements have been exchanged by the mean value!!")
  }
}
```

Zeilenanzahl Nach UREDA sind drei Regeln für die Anzahl der Zeilen einsetzbar, die auch zur Definition der Klassenanzahl von Histogrammen herangezogen werden. Die erste, die auf Dixon zurückgeht, gilt als bewährt, die zweite (von Velleman) empfiehlt sich besonders bei kleineren Stichprobenumfängen, die dritte (Sturges) findet weniger Unterstützung.

Zunächst berechnen wir nach der gewählten Regel die Zeilenanzahl des Plots. Dazu wird der Stichprobenumfang auf `n` abgelegt und zusätzlich werden die Daten sortiert.

6 ⟨*stelle gemäß* `rule.line` *maximale Zeilenanzahl fest* 6⟩ ≡ ⊂ 4, 48

```
n <- length(data <- sort(data))
row.max <- floor(  c(Dixon   =10*log(n,10),
                     Velleman=2*sqrt(n),
                     Sturges =1+log(n,2)        ))[rule.line]
```

Skalenbereich In der Regel werden beim Aufruf keine Grenzen für den Bereich der Skala angegeben werden. Das Maximum und das Minimum können untauglich sein, da eventuelle Ausreißer zu üblen Effekten führen können. Deshalb wird, falls `Min` oder `Max` nicht festgelegt sind, diese mittels `boxplot` ermittelt. Die Spannweite der nicht-Ausreißer wird auf `spannweite.red` abgelegt.

7 ⟨*ermittle mittels* `boxplot` *groben Skalenbereich* 7⟩ ≡ ⊂ 4, 48

```
stats <- boxplot(data,plot=FALSE)
if(missing(Min)) Min <- if (trim.outliers) stats$stats[1,1] else min(data, na.rm=TRUE)
if(missing(Max)) Max <- if (trim.outliers) stats$stats[5,1] else max(data, na.rm=TRUE)
spannweite.red<-Max - Min
```

Normierungsfaktor Zur Darstellung muß eine geeignete Normierung der Daten erfolgen. Hierzu wird intern ein Skalierungsfaktor `factor` ermittelt. Der Faktor zeigt an, mit welcher 10-er Potenz der Stamm multipliziert werden muß, damit er den Bereich der Input-Daten abdeckt. Das Maximum der Daten reicht nicht zu seiner Bestimmung aus, da Inputs aus [1,989] zu einem anderen Stamm als aus [980,989] führen. Besser ist die Spannweite als Ausgangspunkt. Diese erbringt im ersten Fall 998 und im zweiten 9. Im ersten Fall könnte sich ein Faktor von 100 ergeben und die Zeilenstruktur `0 | xyz` bis `10 | xyz`, im zweiten ein Faktor von 1 bei Zeilen der Form: `980 | xyz` bis `990 | xyz`. Weiter betrachten wir Daten aus einem Intervall [980,982]: Wenn wenige Daten vorliegen, werden sich die Stämme `980, 981, 982` ergeben. Steigt die Anzahl Daten an, steigt durch eine feinere Maserung die Zeilenanzahl. Bei 1000 Werten werden nach der ersten Regel ca. 30 Klassen benötigt, was zu einer Faktorveränderung führen muß: `9800, 9801, ..., 9802` mit Faktor 1/10. Nach Regel 2 benötigen wir dann 63 Klassen, nach der dritten 10. Im Fall von 5 Werten liefern die Regeln 6, 4 und 3. Hier ist eine Übersicht:

8 ⟨*zeige Beziehung Werteanzahl Zeilenanzahl gemäß Regel* 8⟩ ≡

```
anz <- rbind(dixon=floor(10*log(n,10)),
             velleman=floor(2*sqrt(n)),
             sturges=floor(1+log(n,2)))
colnames(anz) <- paste("n=",n,sep="")
print(anz)
```

```
         n=2 n=4 n=8 n=16 n=32 n=64 n=128 n=256 n=512 n=1024 n=2048
dixon      3   6   9   12   15   18    21    24    27     30     33
velleman   2   4   5    8   11   16    22    32    45     64     90
sturges    2   3   4    5    6    7     8     9    10     11     12
```

Wir erkennen, daß gemäß der ersten und der dritten Regel der Unterschied der Zeilenanzahlen eine Zehnerpotenz umfaßt, nach der zweiten differiert die Klassenanzahl um 2 Zehnerpotenzen.

Wir wollen ausgehend von der Regel die Länge des Intervalls bestimmen, das zu einer Zeile gehört. Dann versuchen wir dieser Länge durch Kombination von Faktor und Maserung möglichst nahe zu kommen. Ausreißer dürfen dabei natürlich nicht berücksichtigt werden.

Eine grobe Länge für das Zeilenintervall erhalten wir durch Division der gesammten Länge durch die anzustrebende Zeilenanzahl. Wenn eine Einheit angegeben worden ist, ergibt sich der Normierungsfaktor mittels `unit*10` sowie zur Erzielung einer 10-er Potenz durch einen Rundungsprozeß. Ist keine Einheit angegeben, ergibt sich diese aus der zur Zeilenintervalllänge nächst größeren Zehnerpotenz.

9 ⟨*bestimme Intervalllänge und ggf. Faktor* `factor` 9⟩ ≡ ⊂ 4, 48

```
zeilen.intervall.laenge <- spannweite.red / row.max
if(missing(unit)){
       factor <- 10^ceiling(log(zeilen.intervall.laenge,10))
} else factor <- 10^round(log(unit*10,10))
debug.show("factor")
```

Zeilenintervalllänge Nun werden aufgrund der ermittelten Intervalllänge (im Zweifelsfall eher etwas groessere) Intervalle (und dadurch weniger Klassen) definiert: `delta.tick`. `z` zeigt schon eine normierte Länge an, die mit Länge der Größe `0,.1,.2,.5` verglichen werden. Der Vergleich mit 0 dient nur der Absicherung gegenüber pathologische Fällen. Als Resultat wird eine normierte Zeilenintervalllänge aus `.2,.5,1` ausgewählt.

10 ⟨*berechne aus* `zeilen.intervall.laenge` *und* `factor` *Tickabstand* 10⟩ ≡ ⊂ 4, 48

```
z <- zeilen.intervall.laenge/factor  # z in (0.1 ,1]
delta.tick <- c(.2,.2,.5,1)[sum(z>c(0,.1,.2,.5))]
```

Maserung Nach der hier implementierten Auffassung gibt es nur Maserungen aus der Menge: $\{1, 2, 5\}$. Die Maserung `m` ist der Kehrwert des normierten Tickabstands, so daß Tickabstand .2 zur Maserung 5 führt, in einer Zeile können dann 2 verschiedene Ziffern auftauchen. Wird jedoch die Maserung über das Input-Argument `m` festgelegt, muß `delta.tick` angepaßt werden. Hierdurch lassen sich übrigens auch Maserungen wie `m=10` erzwingen.

11 ⟨*bestimme ggf. Maserung* `m` 11⟩ ≡ ⊂ 4, 48

```
if(missing(m)) m <- round(1/delta.tick) else delta.tick <- 1/m
debug.show("delta.tick"); debug.show("m")
```

Datennormierung Im weiteren Verlauf wollen wir mit normierten Werten weiterarbeiten. Deshalb transformieren wir Werte wie auch die Extremwerte der Skalen.

12 ⟨*transformiere Daten* 12⟩ ≡ ⊂ 4

```
data.tr <- data/factor
Min.tr  <- Min/factor
Max.tr  <- Max/factor
```

Skalenkonstruktion Die Skala ist wie folgt zu interpretieren: im positiven Bereich bezeichnet eine Eintragung x im Stem-and-Leaf-Display das Intervall $[x, x+1)$, im negativen $(x-1, x]$. In der folgenden Tabelle lassen sich einige Beispiele ablesen:

Min-Eintrag	Max-Eintrag	Wertebereich	Spannweite
2	6	[2.000, 6.999]	4.999
-2	2	[-2.999, 2.999]	5.998
-6	-2	[-6.999,-2.000]	4.999

Zur Ermittlung des Skalenbereiches runden wir zunächst die transformierten Extremwerte ab bzw. auf: Der erste Skaleneintrag ist wie der letzte eine ganze Zahl. Die Produktion einer Skala ist mit `seq` kein Problem. Jedoch müssen wir für die gewünschte Interpretation eventuell noch zwei Modifikationen vornehmen. Denn im positiven bezeichnet ein Stamm-Skalenwert die Untergrenze der Werte, die in der Zeile eingetragen werden sollen. Im negativen wechselt die Skala die Bezeichnung: `-2, -1, 0, 1, 2, ...` wird zu `-1, -0, 0, 1, 2, ....` Um bei einem gewünschten `Min==-2` dieses noch unterzubringen, müssen wir eine entsprechende Zeile ergänzen, die später bei `m=1` Werte von `-2.9999` bis `-2.0` aufnehmen kann. Entsprechend kann es vorkommen, daß als Maximum `-2` geplant ist. Dann wird ohne Korrektur, wie am kleinen Beispiel zu sehen ist, aus der Obergrenze `sk.max` von `seq` der Eintrag `-1` werden, der jedoch überflüssig ist.

13 ⟨*bestimme Skalenbereich* 13⟩ ≡ ⊂ 4

```
spannweite.red <- Max.tr - Min.tr
sk.min <- floor(Min.tr)
sk.max <- ceiling(Max.tr)
skala  <- seq(sk.min,sk.max,by=delta.tick)
if(sk.min<0) skala <- c(sk.min-delta.tick,skala)
if(sk.max<0) skala <- skala[-length(skala)]
debug.show("skala")
```

2.3 Displayerstellung

Jetzt sind die Vorarbeiten abgeschlossen: `unit`, `m` und `skala` sind definiert, es liegen transformierte Werte vor und der Erstellungsprozeß kann beginnen.

Für die Erstellung werden zunächst Ausreißer erkannt und entfernt. Die verbleibenden Daten werden im zentralen Plot eingetragen und zum Schluß für die Legende einige Infos zusammengefaßt.

14 ⟨*erstelle Stem-and-Leaf-Display* 14⟩ ≡ ⊂ 3

```
⟨merke Ausreißer 15⟩
⟨konstruiere zentralen Teil des Plots 16⟩
⟨erstelle Interpretationshilfen 29⟩
```

Ausreißer Ein Wert, der außerhalb des Bereiches der Skala liegt, ist ein Ausreißer. Ist der erste Skalenwert positiv, so sind das alle Werte, die kleiner als der Skalenwert sind. Ist `skala[1]` negativ, dann wird schon ein Wert genau von der Größe `skala[1]` nicht aufgenommen und gilt als `LO`. Für positive Maxima sind Werte Ausreißer, die größer gleich `skala[n.sk]+delta.tick` sind. Falls das Maximum unter Null ist, wird ein Wert der Größe `skala[n.sk]+delta.tick` gerade noch eingetragen.

Für die Tiefenberechnung ist es günstig, die Anzahl der Ausreißer zu vermerken. Die Ausreißer selbst werden auf `lower.line` bzw. `upper.line` abgelegt. Zum Schluß werden die Daten ohne Ausreißer auf `data.tr.red` abgelegt.

15 ⟨*merke Ausreißer* 15⟩ ≡ ⊂ 14

```
lo.limit <- if (trim.outliers) skala[1] else -Inf
lo.log   <- if(skala[1   ] <  0) data.tr <= lo.limit else data.tr <  lo.limit
n.sk <- length(skala)
hi.limit <- if (trim.outliers) skala[n.sk] + delta.tick else Inf
hi.log   <- if(skala[n.sk] >= 0) data.tr >= hi.limit else data.tr >  hi.limit

n.lower.extr.values <- sum(lo.log); n.upper.extr.values <- sum(hi.log)
if(0<n.lower.extr.values){
  lower.line <- paste("LO:", paste(data[lo.log],collapse=" "))
}
if(0<n.upper.extr.values){
```

```
  upper.line <- paste("HI:", paste(data[hi.log],collapse=" "))
 }
 data.tr.red <-data.tr[(!lo.log)&(!hi.log)]
```

2.3.1 Zentraler Stem-and-Leaf-Display

Für den zentralen Plot müssen zu den verbleibenden Daten Stämme und Blätter gefunden werden. Dann werden die Blätter auf die Zeilen verteilt. Die Blätter müssen zu Ästen zusammengefaßt und aus `skala` ein Baumstamm erstellt werden. Zum Schluß ist die Tiefeninformation zu ermitteln und anzubringen.

16 ⟨*konstruiere zentralen Teil des Plots* 16⟩ ≡ ⊂ 14
⟨*zerlege Zahlen in Stamm und Blatt* 17⟩
⟨*verteile Blätter auf passende Klassen* 20⟩
⟨*ermittle Äste mit Blättern* 21⟩
⟨*konstruiere Skala und füge sie an den zentralen Plot an* 22⟩
⟨*ermittle Tiefen und füge sie an zentralen Plot an* 28⟩

Zerlegung der Werte Stämme werden durch Abschneiden gebildet. Für negative Werte geschieht das durch Aufrunden, für positive durch Abrunden. Die Blätter ergeben sich über Differenzbildung von um eine Stelle nach links geshifteten Daten und Stämmen. Die Differenzen negativer Werte sind dann aufzurunden, die anderen abzurunden. Übrigens führte `ceiling((data.tr.red-stem)*10)` zu Fehlern.

17 ⟨*zerlege Zahlen in Stamm und Blatt* 17⟩ ≡ ⊂ 16
```
 stem  <- ifelse(data.tr.red<0, ceiling(data.tr.red), floor(data.tr.red) )
 # eps <- 1e-12; leaf <- floor(abs(data.tr.red*10-stem*10)+eps)
 leaf  <- floor(10*abs(signif(data.tr.red-stem,10)))
 debug.show("leaf"); debug.show("stem")
```

Der Vergleich von Dezimalzahlen wird problematisch. wenn wir an die Grenzen von Dezimalzahlendarstellungen herankommen. Shiften wir 0.95 durch Multiplikation mit 0.1 um eine Stelle nach links, erhalten wir scheinbar 9.5. Jedoch stellt sich 9.5 nicht ganz genau ein, wie nach einer Restbildung mit 1 oder durch Subtraktion von 9.0 deutlich wird:

18 ⟨* 18⟩ ≡
```
 print(rbind("      .95/0.1-9.0  ="=as.character( .95/0.1-9.0),
 #                                   as.character((.95/0.1)%%1),
             "      .95/0.1-9.5  ="=as.character( .95/0.1-9.5),
             "floor(.95/0.1-9.5) ="=as.character(floor(.95/0.1-9.5))))
```

```
                   [,1]
     .95/0.1-9.0  = "0.499999999999998"
     .95/0.1-9.5  = "-1.77635683940025e-15"
floor(.95/0.1-9.5) = "-1"
```

Die Differenz ist relativ klein, kann jedoch wirkungsvoll sein. Deshalb können / sollten wir vor einem Abrundungsprozess ein Sicherheits-Epsilon addieren, um dann auf der sicheren Seite zu sein.

19 ⟨* 18⟩+ ≡
```
 eps<-1e-12
 print(rbind("      .95/0.1-9.0+eps  ="=as.character( .95/0.1-9.0+eps),
 #                                      as.character((.95/0.1)%%1+eps),
             "      .95/0.1-9.5+eps  ="=as.character( .95/0.1-9.5+eps),
             "floor(.95/0.1-9.5+eps) ="=as.character(floor(.95/0.1-9.5+eps))))
```

```
                     [,1]
     .95/0.1-9.0+eps  = "0.500000000000998"
     .95/0.1-9.5+eps  = "9.982236431606e-13"
floor(.95/0.1-9.5+eps) = "0"
```

Die Technik, etwas Schmutz zu addieren, erscheint nicht so richtig überzeugend zur Lösung der Diskretisierungsungenauigkeit. Besser gefällt die Idee, die Differenz von `data.tr.red` und `stem` nach 10 Stellen abzuschneiden und dann erst abzurunden. Falls sich weitere Probleme einstellen, kann ggf. auf die erste Lösung mit Addition eines `eps` zurückgegriffen werden, denn die entsprechende Zeile ist als Kommentarzeile noch existent.

Blätterzuordnung Die Blätter werden gemäß der Größe der Daten auf Klassen aufgeteilt. Die Klassen für nicht-negative Werte werden durch Zählen der Skalenwerte, die kleiner gleich sind, gefunden. Hier ist es für die Vorstellung praktisch, daß die Werte sortiert sind. Negative Werte werden nach der selben Logik zugeordnet, jedoch wird dazu vom Maximum aus operiert.

Damit leere Klassen keine Probleme bereiten, wird in jede Klasse zwischenzeitlich ein Dummyelement plaziert. Anhand von `class.of.data.tr` werden die Blätter gesplittet und die Dummyelemente wieder entfernt.

6.8 ist nicht gleich 6.8. Deshalb wurden am 29.3.2006 Rundungen mit `signif` in den Vergleichsprozess eingebaut – nach einem Fehlerhinweis von Dietrich Trenkler. Betrachten wir bspw. den Wert 68, dann kann Folgendes passieren: Es wird eine Skala erzeugt mit Schrittweite: 0.2 durch `seq(4,7,by=0.2)`. Dieses liefert selbst bei `options(digits=22)` die Werte: `[1] 4.0 4.2 4.4 4.6 4.8 5.0 5.2 5.4 5.6 5.8 6.0 6.2 6.4 6.6 6.8 7.0` Aus dem Wert 68 wird durch Transformation `68/10` also 6.8. Leider gilt nicht `seq(4,7,by=0.2)[15]==(68/10)`, sondern: `(seq(4,7,by=0.2)[15]-(68/10))==8.881784197001252e-16`. Der Skalenwert ist also etwas größer und die 68 wir in die Klasse vor der richtigen einsortiert. Dadurch führte mit der alten Version der Aufruf von `stem.leaf(2*(24:34),m=5,depths=FALSE)` zu einem Fehler. Ein Runden auf 10 Stellen neutralisiert die Ungenauigkeit.

20 ⟨*verteile Blätter auf passende Klassen* 20⟩ ≡ ⊂ 16

```
class.of.data.tr<-unlist(c(
   sapply(signif(data.tr.red[data.tr.red< 0],10),
     function(x,sk)length(sk)-sum(-sk<=-x),signif(skala,10))
  ,sapply(signif(data.tr.red[data.tr.red>=0],10),
     function(x,sk)sum( sk<= x),signif(skala,10))
))
debug.show("class.of.data.tr")
class.of.data.tr  <- c(1:length(skala),class.of.data.tr)
leaf.grouped      <- split(c(rep(-1,length(skala)),leaf),class.of.data.tr)
leaf.grouped      <- lapply(leaf.grouped, function(x){ sort(x[-1]) })
# debug.show("leaf.grouped")
```

`paste` regelt die Astbildung problemlos. Jedoch ist zu berücksichtigen, wenn die negativen Blätter in absteigender Reihenfolge eingetragen werden sollen.

21 ⟨*ermittle Äste mit Blättern* 21⟩ ≡ ⊂ 16

```
⟨merke negative Klassen und Klasse, die bei −1 beginnt 23⟩
⟨spiegele ggf. Blätter im negativen Bereich 24⟩
leaf.grouped.ch <- paste("|",unlist(lapply(leaf.grouped,paste,collapse="")))
# debug.show("leaf.grouped")
```

Display-Skala Die Konstruktion der Bezeichnung für die Skalen verläuft in zwei Schritten.

22 ⟨*konstruiere Skala und füge sie an den zentralen Plot an* 22⟩ ≡ ⊂ 16

```
⟨ermittle Zeilennamen für den Stamm 25⟩
⟨modifiziere Zeilennamen gemäß Maserung 26⟩
```

Für die Bezeichnung der Zeilen werden negative und -0-Klassen gemerkt.

23 ⟨*merke negative Klassen und Klasse, die bei −1 beginnt* 23⟩ ≡ ⊂ 21

```
class.negative <- skala < 0
class.neg.zero <- floor(skala) == -1
```

24 ⟨*spiegele ggf. Blätter im negativen Bereich* 24⟩ ≡ ⊂ 21

```
if (reverse.negative.leaves){
        for (i in seq(class.negative))
```

```
            if (class.negative[i]) leaf.grouped[[i]] <- rev(leaf.grouped[[i]])
}
```

Die Zeilennamen ergeben sich aus der Skala, indem negative Werte um 1 verschoben werden, die Klassen `class.neg.zero` bekommt den korrekten Namen `-0`.

25 ⟨*ermittle Zeilennamen für den Stamm* 25⟩ ≡ ⊂ 22

```
line.names <- skala
line.names[class.negative] <- line.names[class.negative]+1
line.names <- as.character(floor(line.names))
line.names[class.neg.zero] <- "-0"
```

Tukey-Stil Bei `style="Tukey"` werden spezielle Symbole zur Stammverschönerung angebracht. Wieder führen negative Werte zu Fallunterscheidungen.

26 ⟨*modifiziere Zeilennamen gemäß Maserung* 26⟩ ≡ ⊂ 22

```
if(style=="Tukey"){
  switch(as.character(m),
  "1"={},
  "2"={
        h<-round(2*(skala%%1)) #; line.names[h!=0] <- ""
        line.names<-paste(line.names,
                ifelse(skala<0,c(".","*")[1+h],c("*",".")[1+h]),sep="")
      },
  "5"={
        h<-round(5*(skala%%1)); line.names[h>0 & h<4] <- ""
        line.names<-paste(line.names, ifelse(skala<0,
                        c(".","s","f","t","*")[1+h],
                        c("*","t","f","s",".")[1+h]), sep="")
      }
  )
}
⟨definiere Funktion ragged.left 27⟩
line.names <- ragged.left(line.names)
```

Damit hinterher die |-Trennstriche untereinander stehen, ist eine Auffüllung mit Leerzeichen erforderlich. Dieses leistet die Funktion `ragged.left`.

27 ⟨*definiere Funktion* `ragged.left` 27⟩ ≡ ⊂ 26

```
ragged.left <- function(ch.lines){
  max.n <- max(n.lines<-nchar(ch.lines))
  h     <- paste(rep(" ",max.n),collapse="")
  ch.lines <- paste( substring(h,1,1+max.n-n.lines), ch.lines)
  ch.lines
}
```

Tiefenermittlung Die Tiefenermittlung geschieht über zwei Zählprozesse. Dabei müssen ggf. die Anzahlen der Ausreißer (`n.lower.extr.values` und `n.upper.extr.values`) beachtet werden.

Die Stelle des Medians liegt dort, wo die Tiefenvektoren, entstanden durch Kumulation von `n.class`, sich – graphisch gesprochen – schneiden. Dort kommen zwei Zeilen infrage. Die mit der kleineren Differenz zwischen den Zählvektoren ist die gesuchte.

Der jeweils kleinste Wert der Tiefenvektoren ist festzuhalten und das entstandene Objekt mit passend vielen Leerzeichen zu füllen. Weiter sind Tiefeneinträge in Zeilen ohne Blätter zu löschen. Nebenbei werden die Positionen leerer Zeilen vermerkt `select==FALSE`.

28 ⟨*ermittle Tiefen und füge sie an zentralen Plot an* 28⟩ ≡ ⊂ 16

```
n.class <- unlist(lapply(leaf.grouped,length))
select <- (cumsum(n.class) > 0) & rev((cumsum(rev(n.class)) > 0))
depth     <-     cumsum(n.class)          + n.lower.extr.values
```

```
depth.rev <- rev(cumsum(rev(n.class))     + n.upper.extr.values)
debug.show("depth")

uplow <- depth>=depth.rev
pos.median <- which(uplow)[1] + (-1:0)
h <- abs(depth[pos.median]-depth.rev[pos.median])
pos.median <- pos.median[1]+(h[1]>h[2])
debug.show("pos.median")

depth[uplow] <- depth.rev[uplow]
depth <- paste(depth,"")
depth[pos.median] <- paste("(",n.class[pos.median],")",sep="")
depth[n.class==0] <- " "
depth <- if (depths) ragged.left(depth) else ""
```

Zur Information werden die wesentlichen Infos in der Variablen `info` zusammengefaßt.

29 ⟨*erstelle Interpretationshilfen* 29⟩ ≡ ⊂ 14

```
info <- c(  paste("1 | 2: represents",1.2*factor),
         #  paste("    m:",m     ),
            paste(" leaf unit:",factor/10),
            paste("            n:",n     ))
```

2.4 Ausgabe

Zum Schluß werden die Ergebnisse in einem Objekt zusammengebunden bzw. ausgegeben. 080711: Zeile mit NA-Zähler eingebaut.

30 ⟨*stelle Ergebnis zusammen* 30⟩ ≡ ⊂ 3

```
stem <- paste(depth, line.names, leaf.grouped.ch)
stem <- if((m!=5)||sum(select)>4) stem[select] else stem
result <- list(display=stem)
if(exists("lower.line")) result<-c(lower=lower.line,result)
if(exists("upper.line")) result<-c(result,upper=upper.line)
if(0<n.na&&!na.rm) result<-c(result,NAs=paste("NA's:",n.na,collapse=" "))
result <- c(list( info=info), result)
if(printresult){ for(i in seq(result)) cat(result[[i]],sep="\n") }
result <- c(result, list(depths=depth, stem=line.names, leaves=leaf.grouped.ch))
invisible(result)
```

3 Demos

Für Demonstrationen bietet sich Chambers, Cleveland, Kleiner, Tukey (1983): *Graphical Methods for Data Analysis*, S.27, an. Dort wird ein Teil eines im Buch abgedruckten Ozon-Datensatzes mit verschiedenen m-Werten dargestellt:

31 ⟨*teste* `stem.leaf` 31⟩ ≡

```
# Chambers, Cleveland, Kleiner, Tukey (1983), p27
oz<-c( 60+c(0,1,1,4,4,4,4,6,6,8,8,8,9),
       70+c(1,1,1,1,1,1,1,2,2,3,5,5),
       80+c(0,0,0,0,0,0,2,2,3,5,6,6,7,7,7,9) )
data(co2)
"bd384" <- c(2.968, 2.097, 1.611, 3.038, 7.921, 5.476, 9.858,
             1.397, 0.155, 1.301, 9.054, 1.958, 4.058, 3.918, 2.019, 3.689,
             3.081, 4.229, 4.669, 2.274, 1.971, 10.379, 3.391, 2.093,
             6.053, 4.196, 2.788, 4.511, 7.3, 5.856, 0.86, 2.093, 0.703,
             1.182, 4.114, 2.075, 2.834, 3.698, 6.48, 2.36, 5.249, 5.1,
             4.131, 0.02, 1.071, 4.455, 3.676, 2.666, 5.457, 1.046, 1.908,
             3.064, 5.392, 8.393, 0.916, 9.665, 5.564, 3.599, 2.723, 2.87,
```

```
            1.582, 5.453, 4.091, 3.716, 6.156, 2.039)
repeat{
  cat("Wahl des Tests:\n")
  h<-menu(c( "Ozon  - m=1", "Ozon  - m=2", "Ozon  - m=5", "co2   - m=1",
             "co2   - m=2", "co2   - m=5", "bd384 - m=1", "bd384 - m=2",
             "bd384 - m=5"))
  switch(h, stem.leaf(oz,m=1), stem.leaf(oz,m=2), stem.leaf(oz,m=5),
            stem.leaf(co2,m=1), stem.leaf(co2,m=2), stem.leaf(co2,m=5),
            stem.leaf(bd384,m=1), stem.leaf(bd384,m=2), stem.leaf(bd384,m=5)  )
  if(h==0) break
}
```

4 RD-File

John Fox wrote the first version of the following RD-File (some small changes are done by Peter Wolf).

32 ⟨*definiere Hilfe zu* `stem.leaf` 32⟩ ≡

```
\name{stem.leaf}
\alias{stem.leaf}
\alias{stem.leaf.backback}

\title{stem and leaf display and back to back stem and leaf display}
\description{
  Creates a classical ("Tukey-style") stem and leaf display / back-to-back stem and leaf display.
}

\usage{
stem.leaf(data, unit, m, Min, Max, rule.line = c("Dixon", "Velleman", "Sturges"),
    style = c("Tukey", "bare"), trim.outliers = TRUE, depths = TRUE,
    reverse.negative.leaves = TRUE, na.rm = FALSE, printresult = TRUE)
stem.leaf.backback(x,y, unit, m, Min, Max, rule.line = c("Dixon", "Velleman",
    "Sturges"), style = c("Tukey", "bare"), trim.outliers = TRUE,
    depths = TRUE, reverse.negative.leaves = TRUE, na.rm = FALSE,
    printresult=TRUE, show.no.depths = FALSE, add.more.blanks = 0,
    back.to.back = TRUE)
}
\arguments{
  \item{data}{a numeric vector of data}
  \item{x}{first dataset for \code{stem.leaf.backback}}
  \item{y}{first dataset for \code{stem.leaf.backback}}
  \item{unit}{leaf unit, as a power of 10 (e.g., \code{100}, \code{.01});
    if \code{unit} is missing \code{unit} is choosen by \code{stem.leaf}.}
  \item{m}{number of parts (1, 2, or 5) into which each stem will be separated;
  if \code{m} is missing the number of parts/stem
  (\code{m}) is choosen by \code{stem.leaf}.}
  \item{Min}{smallest non-outlying value; omit for automatic choice.}
  \item{Max}{largest non-outlying value; omit for automatic choice.}
  \item{rule.line}{the rule to use for choosing the desired number of lines
    in the display; \code{"Dixon"} = 10*log10(n); \code{"Velleman"} = 2*sqrt(n);
    \code{"Sturges"} = 1 + log2(n); the default is \code{"Dixon"}.}
  \item{style}{\code{"Tukey"} (the default) for "Tukey-style" divided stems;
    \code{"bare"} for divided stems that simply repeat the stem digits.}
  \item{trim.outliers}{if \code{TRUE} (the default), outliers are placed on \code{LO} and
    \code{HI} stems.}
  \item{depths}{if \code{TRUE} (the default), print a column of "depths" to the left of the
    stems; the depth of the stem containing the median is the stem-count enclosed in
    parentheses.}
  \item{reverse.negative.leaves}{if \code{TRUE} (the default), reverse direction the leaves on negative
    stems (so, e.g., the leaf 9 comes before the leaf 8, etc.).}
  \item{na.rm}{ if TRUE "NA" values are removed otherwise the number of NAs are counted.}
  \item{printresult}{ if TRUE output of the stem and leaf display by \code{cat}.}
  \item{show.no.depths}{ if TRUE no depths are printed.}
  \item{add.more.blanks}{ number of blanks that are added besides the leaves.}
  \item{back.to.back}{ if FALSE two parallel stem and leaf displays are constructed.}
}

\details{
  Unlike the \code{stem} function in the \code{base} package, \code{stem.leaf} produces
```

```
  classic stem-and-leaf displays, as described in Tukey's \emph{Exploratory Data Analysis}.
  The function \code{stem.leaf.backback} creates back-to-back stem and leaf displays.
}

\value{
  The computed stem and leaf display is printed out.
  Invisibly \code{stem.leaf} returns the stem and leaf
  display as a list containing the elements
  \code{info} (legend), \code{display} (stem and leaf display as character vecter),
  \code{lower} (very small values), \code{upper} (very large values), \code{depths} (vector of depths),
  \code{stem} (stem information as a vector), and \code{leaves} (vector of leaves).
}
\references{
    Tukey, J.
    \emph{Exploratory Data Analysis.}
    Addison-Wesley, 1977.
    }

\author{Peter Wolf, the code has been slightly modified by John Fox \email{jfox@mcmaster.ca}
    with the original author's permission, help page written by John Fox,
    the help page has been slightly modified by Peter Wolf.}

\seealso{\code{\link[graphics]{stem}}}

\examples{
stem.leaf(co2)
stem.leaf.backback(co2[1:120],co2[121:240])
stem.leaf.backback(co2[1:120],co2[121:240], back.to.back = FALSE)
stem.leaf.backback(co2[1:120],co2[121:240], back.to.back = FALSE,
                   add.more.blanks = 3, show.no.depths = TRUE)
stem.leaf.backback(rivers[-(1:30)],rivers[1:30], back.to.back = FALSE, unit=10, m=5,
                   Min=200, Max=900, add.more.blanks = 20, show.no.depths = TRUE)
}

\keyword{misc}
```

5 Test

Testen ist eine schwierige Sache. Systematische Aufrufe werden sich hier besser als Zufallsaufrufe zu eignen. Zunächst empfiehlt es sich schon während des Entwicklungsprozesses, an bestimmten Punkten Öffnungen einzubauen, die bei Bedarf Auskunft über die Innereien während der Bearbeitung, also des Prozesses, geben. Dieses ist im Code umgesetzt durch `debug.show("xyz")`-Konstruktionen. Jetzt gilt es die Funktion `debug.show` geeignet zu definieren.

33 ⟨*setze ggf.* `verb` *gemäß Debugging-Wunsch* 33⟩ ≡ ⊂ 3, 48

```
if(0<length(h<-find("debug.cond")) && ".GlobalEnv" %in% h){
  debug.cond<-get("debug.cond",envir=.GlobalEnv)
} else debug.cond<-""
debug.show<-function(name){
  if(!exists("debug.cond")) return()
  if(debug.cond=="all"|| (name %in% debug.cond) ){
    cat(name,":\n"); obj<-eval(parse(text=name))
    if(is.vector(obj)){ print(obj) }
    return()
  }
}
```

Mit Hilfe dieser Testunterstützungsfunktion werden im Folgenden einige wichtige Tests absolviert. Zur Erinnerung hier noch einmal die Argumente. `unit,m,Min,Max,rule.line="Dixon",style="Tukey"`

5.1 Code-Erzeugung

34 ⟨* 18⟩+ ≡

```
tangleR("ms.rev",expand.roots = "", expand.root.start = TRUE)
```

5.2 Diverse Tests

5.2.1 Fehlersituation von DT

Dietrich Trenkler hat einen Fehler gefunden, der auf Rundungsprobleme zurückgeführt werden konnte. In *verteile Blätter auf passende Klassen* wurde 2.8 mit 2.8 verglichen mit dem Ergebnis, dass 2.8 größer als 2.8 ist. Deshalb wurde am 29.3.2006 Rundungen mit `signif` in den Vergleichsprozess eingebaut. 10 Stellen sollten reichen.

35 ⟨* 18⟩+ ≡
```
debug.cond<<-""
"a" <- structure(c(12, 29, 49, 280, 78, 41, 49, 308, 70, 57,
 41, 37, 275, 33, 267, 37, 33, 57, 37, 41, 25, 41, 53, 74,
57, 53, 37, 49, 66, 70, 134, 33, 57, 45, 62, 250, 37, 271,
37, 41, 12, 70, 25), .Names = c("Acerola", "Ananas", "Apfel",
 "Apfel, getrocknet", "Apfelmus", "Apfelsine, Orange", "Aprikose",
 "Aprikose, getrocknet", "Banane", "Birne", "Brombeeren",
 "Clementine", "Datteln, getrocknet", "Erdbeeren", "Feigen, getrockne",
 "Granatapfel", "Grapefruit", "Heidelbeeren", "Himbeeren",
 "Holunderbeeren", "Honigmelone", "Johannisbeeren, rot",
 "Johannisbeeren, schwarz", "Kaki", "Kirsche, sÃijÃü", "Kiwi",
 "Mandarine", "Mango", "Mirabellen", "Nektarine", "Oliven, mariniert",
 "Papaya","Passionsfrucht", "Pfirsich", "Pflaumen",
 "Pflaumen,getrocknet", "Preiselbeeren","Rosinen", "Satsuma",
 "Stachelbeeren", "Wassermelone", "Weintrauben","Zitrone"))
 names(a)<-NULL; aa<-c(rev(sort(a))[1:5],sort(a)[1:5])
⟨definiere test 36⟩
test('stem.leaf(a,Min=0,Max=300)')
```

```
stem.leaf(a,Min=0,Max=300)
1 | 2: represents 120
 leaf unit: 10
            n: 43
    2    0* | 11
   14     t | 222333333333
  (15)    f | 4444444445555555
   14     s | 6677777
        0. |
        1* |
    7    t | 3
         f |
         s |
        1. |
        2* |
         t |
    6    f | 5
    5    s | 677
    2   2. | 8
    1   3* | 0
```

5.2.2 Erfolgreiche Tests

Als Datensätze wollen wir `oz` wie auch `co2` verwenden. Für den Test bietet sich eine kleine Unterstützungsfunktion an:

36 ⟨*definiere* `test` 36⟩ ≡ ⊂ 35, 44
```
oz<-c( 60+c(0,1,1,4,4,4,4,6,6,8,8,8,9), 70+c(rep(1,7),2,2,3,5,5),
       80+c(rep(0,6),2,2,3,5,6,6,7,7,7,9) )
if(exists("data")) data(co2)
test<-function(what) {
  cat(what,"\n"); eval(parse(text=what)); return()
}
```

Mit `test` lassen sich bequem einige Tests erledigen. Von hinten beginnend testen wir, ob `style` für `m=2` und `m=5` wirksam wird, sofern es auf `"Tukey"` gesetzt ist. Damit ist auch gleich ein erster Test für `m` beschrieben.

37 ⟨*Test von* `style` 37⟩ ≡ ⊂ 44, 62, 63, 64
```
cat("style-Test-start\n")
```

```
test('stem.leaf(oz,m=1,style="Tukey")')
test('stem.leaf(oz,m=2,style="Tukey")')
test('stem.leaf(oz,m=5,style="Tukey")')
test('stem.leaf(oz,m=1,style="bare")')
test('stem.leaf(oz,m=2,style="bare")')
test('stem.leaf(oz,m=5,style="bare")')
cat("style-Test-end\n")
```

```
style-Test-start
stem.leaf(oz,m=1,style="Tukey")
1 | 2: represents 12
 leaf unit: 1
            n: 41
   13    6 | 0114444668889
  (12)   7 | 111111122355
   16    8 | 0000002235667779
stem.leaf(oz,m=2,style="Tukey")
1 | 2: represents 12
 leaf unit: 1
            n: 41
    7    6* | 0114444
   13    6. | 668889
  (10)   7* | 1111111223
   18    7. | 55
   16    8* | 000000223
    7    8. | 5667779
stem.leaf(oz,m=5,style="Tukey")
1 | 2: represents 12
 leaf unit: 1
            n: 41
    3    6* | 011
          t |
    7     f | 4444
    9     s | 66
   13    6. | 8889
   20    7* | 1111111
   (3)    t | 223
   18     f | 55
          s |
         7. |
   16    8* | 000000
   10     t | 223
    7     f | 5
    6     s | 66777
    1    8. | 9
stem.leaf(oz,m=1,style="bare")
1 | 2: represents 12
 leaf unit: 1
            n: 41
   13    6 | 0114444668889
  (12)   7 | 111111122355
   16    8 | 0000002235667779
stem.leaf(oz,m=2,style="bare")
1 | 2: represents 12
 leaf unit: 1
            n: 41
    7    6 | 0114444
   13    6 | 668889
  (10)   7 | 1111111223
   18    7 | 55
   16    8 | 000000223
    7    8 | 5667779
stem.leaf(oz,m=5,style="bare")
1 | 2: represents 12
 leaf unit: 1
            n: 41
    3    6 | 011
         6 |
    7    6 | 4444
    9    6 | 66
   13    6 | 8889
   20    7 | 1111111
   (3)   7 | 223
   18    7 | 55
         7 |
         7 |
   16    8 | 000000
   10    8 | 223
    7    8 | 5
    6    8 | 66777
    1    8 | 9
style-Test-end
```

Test der verschiedenen Regeln. Wir probieren sowohl Datensatz `oz` wie auch `co2`

38 ⟨*Test von* `rule.line` 38⟩ ≡ ⊂ 44, 62, 63, 64

```
cat("rule-Test-start\n")
test('stem.leaf(oz,rule.line="Dixon")')
test('stem.leaf(oz,rule.line="Velleman")')
test('stem.leaf(oz,rule.line="Sturges")')
test('stem.leaf(co2,rule.line="Dixon")')
test('stem.leaf(co2,rule.line="Velleman")')
test('stem.leaf(co2,rule.line="Sturges")')
cat("rule-Test-end\n")
```

```
rule-Test-start
stem.leaf(oz,rule.line="Dixon")
1 | 2: represents 12
 leaf unit: 1
           n: 41
   3    6* | 011
         t |
   7     f | 4444
   9     s | 66
  13    6. | 8889
  20    7* | 1111111
  (3)    t | 223
  18     f | 55
         s |
        7. |
  16    8* | 000000
  10     t | 223
   7     f | 5
   6     s | 66777
   1    8. | 9
stem.leaf(oz,rule.line="Velleman")
1 | 2: represents 12
 leaf unit: 1
           n: 41
    7    6* | 0114444
   13    6. | 668889
  (10)   7* | 1111111223
   18    7. | 55
   16    8* | 000000223
    7    8. | 5667779
stem.leaf(oz,rule.line="Sturges")
1 | 2: represents 12
 leaf unit: 1
           n: 41
    7    6* | 0114444
   13    6. | 668889
  (10)   7* | 1111111223
   18    7. | 55
   16    8* | 000000223
    7    8. | 5667779
stem.leaf(co2,rule.line="Dixon")
1 | 2: represents 12
 leaf unit: 1
           n: 468
    8    31* | 33344444
   70    31. | 55555556666666666666667777777777788888888888888999999999999
  135    32* | 000000000000000011111111111222222222222223333333333444444444
  187    32. | 555555556666666667777777777777778888888889999999
  233    33* | 00000000111111111222222222223333333334444444
  (40)   33. | 555555666666666777777777888889999999999
  195    34* | 0000000111111122222222223333334444444
  156    34. | 55555555666666667777777888888899999
  119    35* | 000001111111222222222333333333444444444
   78    35. | 555555555666666667777777788888899999999999
   33    36* | 00000011111122223333333444444
    5    36. | 55566
stem.leaf(co2,rule.line="Velleman")
1 | 2: represents 12
 leaf unit: 1
           n: 468
    3     t | 333
   15     f | 444445555555
   41     s | 666666666666667777777777
   70    31. | 888888888888889999999999999
   98    32* | 00000000000000011111111111
  126     t | 22222222222222233333333333
  143     f | 444444444555555555
  168     s | 6666666667777777777777777
  187    32. | 8888888889999999999
  205    33* | 000000001111111111
  225     t | 22222222223333333333
  (14)    f | 44444444555555
  229     s | 666666666777777777
  210    33. | 888889999999999
  195    34* | 00000001111111
  181     t | 222222222233333333
  163     f | 444444455555555
  148     s | 666666667777777
  133    34. | 88888889999999
  119    35* | 000001111111
  107     t | 22222222223333333333
   87     f | 4444444455555555555
   69     s | 66666666777777777
   53    35. | 88888899999999999999
   33    36* | 000000111111
   21     t | 2222333333
   11     f | 444444555
    2     s | 66
stem.leaf(co2,rule.line="Sturges")
1 | 2: represents 12
 leaf unit: 1
           n: 468
   70    31 | 3334444455555556666666666666677777777777888888888888888999999999999999
  187    32 | 000000000000000011111111111222222222222223333333333344444444455555555666666666777777777777777788888888899999999
  (86)   33 | 00000000111111111122222222222333333333444444445555556666666666777777777788888999999999
  195    34 | 0000000011111112222222222333333334444444555555566666666677777777888888899999
  119    35 | 000001111111222222222233333333344444444455555555666666667777777788888899999999999999
   33    36 | 00000011111122223333334444445566
rule-Test-end
```

Test von `unit`

39 ⟨*Test von* `unit` 39⟩ ≡ ⊂ 44, 62, 63, 64

```
cat("unit-Test-start\n")
test('stem.leaf(oz,unit=10)')
# test('stem.leaf(c(oz,-oz),unit=10)'); oz
test('stem.leaf(oz,unit=1)')
test('stem.leaf(oz,unit=.1)')
```

```
 cat("unit-Test-end\n")
```

```
unit-Test-start
stem.leaf(oz,unit=10)
1 | 2: represents 120
 leaf unit: 10
            n: 41
         0* |
          t |
          f |
  (25)    s | 66666666666667777777777
   16    0. | 8888888888888888
         1* |
stem.leaf(oz,unit=1)
1 | 2: represents 12
 leaf unit: 1
            n: 41
   3    6* | 011
         t |
   7     f | 4444
   9     s | 66
  13    6. | 8889
  20    7* | 1111111
  (3)    t | 223
  18     f | 55
         s |
        7. |
  16    8* | 000000
  10     t | 223
   7     f | 5
   6     s | 66777
   1    8. | 9
stem.leaf(oz,unit=.1)
1 | 2: represents 1.2
 leaf unit: 0.1
            n: 41
   1    60 | 0
   3    61 | 00
        62 |
        63 |
   7    64 | 0000
        65 |
   9    66 | 00
        67 |
  12    68 | 000
  13    69 | 0
        70 |
  20    71 | 0000000
  (2)   72 | 00
  19    73 | 0
        74 |
  18    75 | 00
        76 |
        77 |
        78 |
        79 |
  16    80 | 000000
        81 |
  10    82 | 00
   8    83 | 0
        84 |
   7    85 | 0
   6    86 | 00
   4    87 | 000
        88 |
   1    89 | 0
unit-Test-end
```

Test der Extremwertsetzungen.

40 ⟨*Test von* `Min/Max` 40⟩ ≡ ⊂ 44, 62, 63, 64

```
 cat("Max-Min-Test-start\n")
 test('stem.leaf(oz,Min=65,Max=83,unit=.1,m=1)')
 test('stem.leaf(oz,Min=65,Max=83,unit=1,m=1)')
 test('stem.leaf(-oz,Min=-83,Max=-65,unit=.1,m=1)')
 test('stem.leaf(-oz,Min=-83,Max=-65,unit=1,m=1)')
 test('stem.leaf(1:12,Min=5,Max=8,unit=.1,m=1)')
 test('stem.leaf(.5+(-7:6),Min=-3,Max=3,unit=.1,m=1)')
 cat("Max-Min-Test-end\n")
```

```
Max-Min-Test-start
stem.leaf(oz,Min=65,Max=83,unit=.1,m=1)
1 | 2: represents 1.2
 leaf unit: 0.1
            n: 41
LO: 60 61 61 64 64 64 64
   9    66 | 00
        67 |
  12    68 | 000
  13    69 | 0
        70 |
  20    71 | 0000000
  (2)   72 | 00
  19    73 | 0
        74 |
  18    75 | 00
        76 |
        77 |
```

```
       78 |
       79 |
  16   80 | 000000
       81 |
  10   82 | 00
   8   83 | 0
HI: 85 86 86 87 87 87 89
stem.leaf(oz,Min=65,Max=83,unit=1,m=1)
1 | 2: represents 12
 leaf unit: 1
           n: 41
   13    6 | 0114444668889
  (12)   7 | 111111122355
   16    8 | 0000002235667779
stem.leaf(-oz,Min=-83,Max=-65,unit=.1,m=1)
1 | 2: represents 1.2
 leaf unit: 0.1
           n: 41
LO: -89 -87 -87 -87 -86 -86 -85
   8   -83 | 0
  10   -82 | 00
       -81 |
  16   -80 | 000000
       -79 |
       -78 |
       -77 |
       -76 |
  18   -75 | 00
       -74 |
  19   -73 | 0
  (2)  -72 | 00
  20   -71 | 0000000
       -70 |
  13   -69 | 0
  12   -68 | 000
       -67 |
   9   -66 | 00
HI: -64 -64 -64 -64 -61 -61 -60
stem.leaf(-oz,Min=-83,Max=-65,unit=1,m=1)
1 | 2: represents 12
 leaf unit: 1
           n: 41
   16   -8 | 9777665322000000
  (12)  -7 | 553221111111
   13   -6 | 9888664444110
stem.leaf(1:12,Min=5,Max=8,unit=.1,m=1)
1 | 2: represents 1.2
 leaf unit: 0.1
           n: 12
LO: 1 2 3 4
   5    5 | 0
  (1)   6 | 0
   6    7 | 0
   5    8 | 0
HI: 9 10 11 12
stem.leaf(.5+(-7:6),Min=-3,Max=3,unit=.1,m=1)
1 | 2: represents 1.2
 leaf unit: 0.1
           n: 14
LO: -6.5 -5.5 -4.5
   4   -3 | 5
   5   -2 | 5
   6   -1 | 5
  (1)  -0 | 5
   7    0 | 5
   6    1 | 5
   5    2 | 5
   4    3 | 5
HI: 4.5 5.5 6.5
Max-Min-Test-end
```

Klassenzuordnungstest:

41 ⟨*Klassenzuordnungstest* 41⟩ ≡ ⊂ 44, 62, 63, 64

```
# debug.cond<-"skala"
cat("Klassen-Test-start\n")
test('stem.leaf(c(.7+(1:12),4.999,5.0,5.001,7,7.001,7.999,
      8,8.001,8.999,9,9.001,9.999),Min=5,Max=8,unit=.1,m=1)')
test('stem.leaf(-c(.7+(1:12),4.999,5.0,5.001,7,7.001,7.999,
      8,8.001,8.999,9,9.001,9.999),Min=-8,Max=-5,unit=.1,m=1)')
test('stem.leaf(c(.7+(-5:5),-4.001,-4,-3.999, -3, 0, 3, 3.999,
      4, 4.001),Min=-3,Max=3,unit=.1,m=1)')
cat("Klassen-Test-end\n")
```

```
Klassen-Test-start
stem.leaf(c(.7+(1:12),4.999,5.0,5.001,7,7.001,7.999,
      8,8.001,8.999,9,9.001,9.999),Min=5,Max=8,unit=.1,m=1)
1 | 2: represents 1.2
 leaf unit: 0.1
           n: 24
LO: 1.7 2.7 3.7 4.7 4.999
   8    5 | 007
   9    6 | 7
  (4)   7 | 0079
  11    8 | 0079
HI: 9 9.001 9.7 9.999 10.7 11.7 12.7
stem.leaf(-c(.7+(1:12),4.999,5.0,5.001,7,7.001,7.999,
      8,8.001,8.999,9,9.001,9.999),Min=-8,Max=-5,unit=.1,m=1)
1 | 2: represents 1.2
 leaf unit: 0.1
           n: 24
LO: -12.7 -11.7 -10.7 -9.999 -9.7 -9.001 -9
  11    -8 | 9700
```

```
  (4)   -7 | 9700
   9    -6 | 7
   8    -5 | 700
HI: -4.999 -4.7 -3.7 -2.7 -1.7
stem.leaf(c(.7+(-5:5),-4.001,-4,-3.999, -3, 0, 3, 3.999,
      4, 4.001),Min=-3,Max=3,unit=.1,m=1)
1 | 2: represents 1.2
 leaf unit: 0.1
            n: 20
LO: -4.3 -4.001 -4
   6    -3 | 930
   7    -2 | 3
   8    -1 | 3
   9    -0 | 3
  (2)    0 | 07
   9     1 | 7
   8     2 | 7
   7     3 | 079
HI: 4 4.001 4.7 5.7
Klassen-Test-end
```

Jim Albert mailte am 12.4.09 folgende Fehlersituation:

42 ⟨*Example of Jim Albert* 42⟩ ≡ ⊂ 44, 62, 63, 64

```
#debug.cond<<-"all"
⟨definiere ms NA⟩
y<-c(0.99,0.96,0.98,0.94,0.98,0.98,0.97,0.97,0.98,0.96,0.94,0.96,
0.96,0.96,0.96,
0.97,0.96,0.92,0.98,0.95,0.97,0.92,0.92,0.96,0.95,0.96,0.93,0.96,
0.94,0.96,
0.94,0.90,0.92,0.92,0.94,0.92,0.94,0.94,0.91,0.94,0.87,0.91,0.93,
0.84,0.90,
0.93,0.85,0.87,0.91,0.88,0.89,0.89)
print(table(y))
test('stem.leaf(y,unit=.01,m=5)')
```

```
y
0.84 0.85 0.87 0.88 0.89  0.9 0.91 0.92 0.93 0.94 0.95 0.96 0.97 0.98 0.99
   1    1    2    1    2    2    3    6    3    8    2   11    4    5    1
stem.leaf(y,unit=.01,m=5)
1 | 2: represents 0.12
 leaf unit: 0.01
            n: 52
    2      f | 45
    4      s | 77
    7     8. | 899
   12     9* | 00111
   21      t | 222222333
  (10)     f | 4444444455
   21      s | 666666666667777
    6     9. | 888889
```

alter fehlerhafter Output:

```
y
0.84 0.85 0.87 0.88 0.89  0.9 0.91 0.92 0.93 0.94 0.95 0.96 0.97 0.98 0.99
   1    1    2    1    2    2    3    6    3    8    2   11    4    5    1
1 | 2: represents 0.12
 leaf unit: 0.01
            n: 52
    2      f | 35
    4      s | 77
    7     8. | 799
   12     9* | 00111
   21      t | 222222333
  (10)     f | 3333333344
   21      s | 666666666667777
    6     9. | 777778
```

In der Tat muss in der letzten Zeile des Displays `9. | 888889` stehen. Der Fehler geht auf die ungenaue Darstellung von Dezimalzahlen zurück, welche durch Addition eines kleines Epsilons `eps` vor Rundungprozessen behoben werden kann, siehe Chunk: *zerlege Zahlen in Stamm und Blatt*. Nach Integration dieser Verbesserung erhalten wir das richtige Ergebnis, s.o.

Negative-Reverse-Test:

43 ⟨*reverse-Test* 43⟩ ≡ ⊂ 44, 62, 63, 64

```
# debug.cond<-"skala"
cat("Reverse-Test-start\n")
test('stem.leaf(-c(.7+(1:12),4.999,5.0,5.001,7,7.001,7.999,
      8,8.001,8.999,9,9.001,9.999),Min=-8,Max=-5,unit=.1,m=1,
      reverse.negative.leaves=TRUE)')
test('stem.leaf(-c(.7+(1:12),4.999,5.0,5.001,7,7.001,7.999,
      8,8.001,8.999,9,9.001,9.999),Min=-8,Max=-5,unit=.1,m=1,
      reverse.negative.leaves=!TRUE)')
```

```
test('stem.leaf(c(.7+(-5:5),-4.001,-4,-3.999, -3, 0, 3, 3.999,
      4, 4.001),Min=-3,Max=3,unit=.1,m=1,
      reverse.negative.leaves=TRUE)')
test('stem.leaf(c(.7+(-5:5),-4.001,-4,-3.999, -3, 0, 3, 3.999,
      4, 4.001),Min=-3,Max=3,unit=.1,m=1,
      reverse.negative.leaves=!TRUE)')
cat("Reverse-Test-end\n")
```

```
Reverse-Test-start
stem.leaf(-c(.7+(1:12),4.999,5.0,5.001,7,7.001,7.999,
      8,8.001,8.999,9,9.001,9.999),Min=-8,Max=-5,unit=.1,m=1,
      reverse.negative.leaves=TRUE)
1 | 2: represents 1.2
 leaf unit: 0.1
            n: 24
LO: -12.7 -11.7 -10.7 -9.999 -9.7 -9.001 -9
  11    -8 | 9700
  (4)   -7 | 9700
   9    -6 | 7
   8    -5 | 700
HI: -4.999 -4.7 -3.7 -2.7 -1.7
stem.leaf(-c(.7+(1:12),4.999,5.0,5.001,7,7.001,7.999,
      8,8.001,8.999,9,9.001,9.999),Min=-8,Max=-5,unit=.1,m=1,
      reverse.negative.leaves=!TRUE)
1 | 2: represents 1.2
 leaf unit: 0.1
            n: 24
LO: -12.7 -11.7 -10.7 -9.999 -9.7 -9.001 -9
  11    -8 | 0079
  (4)   -7 | 0079
   9    -6 | 7
   8    -5 | 007
HI: -4.999 -4.7 -3.7 -2.7 -1.7
stem.leaf(c(.7+(-5:5),-4.001,-4,-3.999, -3, 0, 3, 3.999,
      4, 4.001),Min=-3,Max=3,unit=.1,m=1,
      reverse.negative.leaves=TRUE)
1 | 2: represents 1.2
 leaf unit: 0.1
            n: 20
LO: -4.3 -4.001 -4
   6    -3 | 930
   7    -2 | 3
   8    -1 | 3
   9    -0 | 3
  (2)    0 | 07
   9     1 | 7
   8     2 | 7
   7     3 | 079
HI: 4 4.001 4.7 5.7
stem.leaf(c(.7+(-5:5),-4.001,-4,-3.999, -3, 0, 3, 3.999,
      4, 4.001),Min=-3,Max=3,unit=.1,m=1,
      reverse.negative.leaves=!TRUE)
1 | 2: represents 1.2
 leaf unit: 0.1
            n: 20
LO: -4.3 -4.001 -4
   6    -3 | 039
   7    -2 | 3
   8    -1 | 3
   9    -0 | 3
  (2)    0 | 07
   9     1 | 7
   8     2 | 7
   7     3 | 079
HI: 4 4.001 4.7 5.7
Reverse-Test-end
```

Hier noch einmal die Testaufrufe zusammengefaßt:

44 ⟨ *Testaufrufe* 44⟩ ≡
⟨*definiere ms* NA⟩
⟨*definiere* `test` 36⟩
⟨*Test von* `style` 37⟩
⟨*Test von* `rule.line` 38⟩
⟨*Test von* `unit` 39⟩
⟨*Test von* `Min/Max` 40⟩
⟨*Klassenzuordnungstest* 41⟩
⟨*Example of Jim Albert* 42⟩
⟨*reverse-Test* 43⟩

6 Back-to-back-stem-and-leaf-Plots

Naheliegend ist die Konstruktion zweier Stem-and-leaf-displays, bei denen die Blätter zweier Stichproben zu den beiden Seiten eines gemeinsamen Stammes eingetragen werden. Es entstehen dann Plots, die wie

Bevölkerungspyramiden aussehen. Mit Hilfe der Funktion `stem.leaf` müsste sich schnell eine Umsetzung finden lassen. Um leicht auf einzelne Einheiten zugreifen zu können, ist zunächst die Ausgabe der `stem.leaf()`-Funktion gegenüber älteren Versionen erweitert worden.

Wir gehen, wie folgt, vor:

1. Kombiniere beide Stichproben und ermittele mit den gemeinsamen Daten einen geeigneten Stamm, der zu beiden Stichproben passt.
2. Berechne für beide Stichproben getrennt die stem and leaf displays.
3. Extrahiere aus beiden Displays die notwendigen Größen.
4. Füge aus den Einzelteilen das neue Display zusammen.

6.1 Der grobe Aufbau der Funktion `stem.leaf.backback()`

Auf der groben Beschreibungsebene ergibt sich aus dem Vorgehen folgender Funktionaufbau. Wir definieren hier die Funktion `stem.leaf()` als lokale Funktion, da sich möglicherweise jemand die Funktion `stem.leaf.backbak()` kopieren und ohne das Paket `aplpack` verwenden möchte.

45 ⟨*definiere* `stem.leaf.backback` 45⟩ ≡ ⊂ 2, 65, 66, 67, 68, 69, 70

```
stem.leaf.backback <- function(x,y, unit, m, Min, Max, rule.line = c("Dixon", "Velleman",
    "Sturges"), style = c("Tukey", "bare"), trim.outliers = TRUE,
    depths = TRUE, reverse.negative.leaves = TRUE, na.rm = FALSE,
    printresult=TRUE, show.no.depths = FALSE, add.more.blanks = 0,
    back.to.back = TRUE){
  ⟨checke Input-Daten stem.leaf.backback() 46⟩
  ⟨definiere stem.leaf 3⟩
  ⟨suche gemeinsamen Stamm in stem.leaf.backback() 47⟩
  ⟨ermittle Blätter u.ä. der beiden Datensätze in stem.leaf.backback() 49⟩
  ⟨synchronisiere stems, leaves und depths der beiden Datensätze stem.leaf.backback() 50⟩
  ⟨komponiere Output von stem.leaf.backback() 51⟩
}
```

6.2 Input-Check

Wenn nur ein Datensatz eingegeben wird, wird er doppelt verwendet. Es werden die Namen und die Anzahlen der fehlenden Werte festgestellt.

46 ⟨*checke Input-Daten* `stem.leaf.backback()` 46⟩ ≡ ⊂ 45

```
x.name <- paste(deparse(substitute(x)),collapse="")
if(missing(y)){ y <- x; y.name <- x.name }
else y.name <- paste(deparse(substitute(y)),collapse="")
n.na.x <- sum(is.na(x)); n.na.y <- sum(is.na(y))
```

6.3 Kombination der beiden Datensätze

Es ist kein Problem die Datensätze zusammenzufassen. Jedoch gilt es ggf. geeignete Parameter-Werte für `m`, `Min` usw. zu ermitteln, damit diese dann explizit für die weiteren Berechnungen zur Verfügung stehen. Dann erstellen wir virtuell einen gemeinsamen stem and leaf display und können über `sl.xy` auf alle wesentlichen Informationen zugreifen. Insbesondere benötigen wir die Variable `unit`, die die Dezimalstelle festlegt.

47 ⟨*suche gemeinsamen Stamm in* `stem.leaf.backback()` 47⟩ ≡ ⊂ 45

```
⟨ermittle geeignete Parameterwerte, sofern nicht festgelegt 48⟩
sl.xy <- stem.leaf(c(x,y),unit=unit,m=m,Min=Min, Max=Max, rule.line=rule.line,
                   style=style,trim.outliers=trim.outliers,depths=depths,
                   reverse.negative.leaves = reverse.negative.leaves,
                   na.rm = na.rm, printresult=FALSE)
h <- grep(" leaf unit: ",sl.xy$info,value=TRUE)
```

```
unit <- as.numeric(sub(" leaf unit: ","",h))
```

Zur Parameterermittlung führen wir die ersten Schritte einer stem and leaf display-Konstruktion durch und können danach die notwendigen Größen zugreifen. Hiernach liegt eine Vorstellung über Stamm und Maserung vor.

48 ⟨*ermittle geeignete Parameterwerte, sofern nicht festgelegt* 48⟩ ≡ ⊂ 47

```
data <- c(x,y)
```

⟨*checke Input* 5⟩
⟨*setze ggf.* `verb` *gemäß Debugging-Wunsch* 33⟩
⟨*stelle gemäß* `rule.line` *maximale Zeilenanzahl fest* 6⟩
⟨*ermittle mittels* `boxplot` *groben Skalenbereich* 7⟩
⟨*bestimme Intervalllänge und ggf. Faktor* `factor` 9⟩
⟨*berechne aus* `zeilen.intervall.laenge` *und* `factor` *Tickabstand* 10⟩
⟨*bestimme ggf. Maserung* `m` 11⟩

6.4 Berechnung der Display-Elemente und Synchronisation der Stämme

Die Blätter der einzelnen Stichproben finden wir durch getrennte Aufrufe von `stem.leaf()`.

49 ⟨*ermittle Blätter u.ä. der beiden Datensätze in* `stem.leaf.backback()` 49⟩ ≡ ⊂ 45

```
sl.x  <- stem.leaf(x,     unit=unit,m=m,Min=Min, Max=Max, rule.line=rule.line,
                   style=style,trim.outliers=trim.outliers,depths=depths,
                   reverse.negative.leaves = reverse.negative.leaves, na.rm = na.rm,
                   printresult=FALSE)
sl.y  <- stem.leaf(y,     unit=unit,m=m,Min=Min, Max=Max, rule.line=rule.line,
                   style=style,trim.outliers=trim.outliers,depths=depths,
                   reverse.negative.leaves = reverse.negative.leaves, na.rm = na.rm,
                   printresult=FALSE)
```

Das Zusammenfügen ist mehr Textverarbeitung als Berechnung, also Fließarbeit. Zunächst extrahieren wir die relevanten Infos aus den Displays `sl.x` und `sl.y`. In den Stämmen suchen wir dann einen Eintrag mit Ziffern, der in beiden vorkommt. Anhand dieses Eintrags müssen die Stämme ggf. geeignet verlängert werden. Entsprechend zu den Anfügungen müssen auch die Vektoren mit den Blättern expandiert werden. Es kann notwendig sein, den ersten Datensatz vorn und oder oder hinten zu verlängern. Gleiches gilt für den zweiten Datensatz. Hierdurch erklären sich die vier If-Konstruktionen. Nach diesem Chunk müssen `x.stem` und `y.stem` identisch sein.

50 ⟨*synchronisiere stems, leaves und depths der beiden Datensätze* `stem.leaf.backback()` 50⟩ ≡ ⊂ 45

```
x.stem   <- gsub(" ","",sl.x$stem);          y.stem   <- gsub(" ","",sl.y$stem)
x.leaves <- substring(sl.x$leaves,3);        y.leaves <- substring(sl.y$leaves,3)
x.depths <- substring(sl.x$depths,3);        y.depths <- substring(sl.y$depths,3)
x.digits <- grep("[0-9]",x.stem,value=TRUE); y.digits <- grep("[0-9]",y.stem,value=TRUE)
h <- match(y.digits, x.digits); h <- h[!is.na(h)][1]
# take the first ->[1] only otherwise an error occurs, see mail from John Fox 10/2013:
x.pos <- which(x.stem==x.digits[h])[1]; y.pos <- which(y.stem==x.digits[h])[1]
LZ <- rep(" ",d <- abs(y.pos - x.pos))
if(x.pos < y.pos) { # x vorn verlÃďngern
  x.stem <- c(y.stem[1:d],x.stem); x.leaves <- c(LZ,x.leaves); x.depths <- c(LZ,x.depths)
}
if(y.pos < x.pos) { # y vorn verlÃďngern
  y.stem <- c(x.stem[1:d],y.stem); y.leaves <- c(LZ,y.leaves); y.depths <- c(LZ,y.depths)
}
x.l <- length(x.stem); y.l <- length(y.stem)
LZ <- rep(" ",d <- abs(y.l-x.l))
if(x.l < y.l) { # x hinten verlÃďngern
  x.stem <- c(x.stem,y.stem[-(1:x.l)]); x.leaves <- c(x.leaves,LZ)
  x.depths <- c(LZ,x.depths,LZ)
}
if(y.l < x.l) { # y hinten verlÃďngern
  y.stem <- c(y.stem,x.stem[-(1:y.l)]); y.leaves <- c(y.leaves,LZ)
  y.depths <- c(LZ,y.depths,LZ)
}
```

6.5 Output-Generierung

Der Output ergibt sich aus den gesammelten Zwischenergebnissen, die alle textlicher Natur sind. Es gilt also Textvektoren vertikal und horizontal zusammenzufügen. Hierzu werden eine Reihe von Hilfsfunktionen definiert, die auch an anderer Stelle hilfreich sein können. Zuerst passen wir die Variablen, die Stämme und Blätter der beiden Datensätze merken, in ihrer Größe an. Dabei werden auch die Blätter des ersten Datensatz gedreht. Dann fügen wir Tiefen, Stamm und Blätter zusammen. Im dritten Schritt komponieren wir die Informationszeilen, die uns Anzahlen, fehlende Werte, Extremwerte und Datensatznamen zeigen. Zum Schluss wird das entstandene Gebilde per `cat()` und als explizites Ergebnis ausgegeben.

51 ⟨*komponiere Output von* `stem.leaf.backback()` 51⟩ ≡ ⊂ 45
```
⟨definiere einige Funktionen zur Anordnung von Textobjekten 53⟩
⟨ergänze Leerzeichen zur Vereinheitlichung der Blatt-Variablen 52⟩
⟨kombiniere depths, stem and leaves 56⟩
⟨komponiere Zeilen mit Infos, Namen, Längen und NA-Anzahlen 58⟩
⟨compose info of HI values 59⟩
⟨compose info of LO values 60⟩
# komponiere Ergebnisvektor
line   <- paste(rep("_",max(nchar(result))),collapse="")
result <- c(line,info.line,name.line,lower.line,line,result,line,upper.line,n.line,
            if(substring(na.line,1,1)=="N") na.line,line)
result <- result[grep("[^ ]",result)]
# zeige Output
if(printresult){ cat(result,sep="\n") }
invisible(list(info=sl.x$info, display=result,lower.x=sl.x$lower,upper.x=sl.x$upper,
               lower.y=sl.y$lower,upper.y=sl.y$upper,
               x.depths=x.depths,y.depths=y.depths,stem=x.stem,x.leaves,y.leaves))
```

Die Blätteranzahl an jeder Stammstelle werden unterschiedlich sein. Gleiches gilt evtl. für den Stamm ebenfalls. Deshalb werden zur Erzielung einer rechteckigen Struktur Leerzeichen angefügt. Danach enthalten alle Elemente eines so vereinheitlichten Vektors gleich viele Zeichen. Zusätzlich werden gemäß Anwenderwunsch weitere `add.more.blanks` den Ästen mit den Blättern hinzugefügt. Zum Schluss wird sichergestellt, dass die beiden Objekte mit den Blättern wirklich gleich groß sind.

52 ⟨*ergänze Leerzeichen zur Vereinheitlichung der Blatt-Variablen* 52⟩ ≡ ⊂ 51
```
x.leaves <- expand.text(x.leaves,fill.right=TRUE,O=add.more.blanks)
x.leaves <- rotate.string(x.leaves)
x.stem   <- expand.text(x.stem,fill.right=FALSE)
y.leaves <- expand.text(y.leaves,fill.right=TRUE,O=add.more.blanks)
# Zweige egalisieren
x.leaves <- expand.a.to.b(x.leaves,y.leaves,fill.right=FALSE)
y.leaves <- expand.a.to.b(y.leaves,x.leaves)
```

Für die Umsetzung der Angleichung finden Hilfsfunktionen Verwendung: `rotate.string()`, `expand.text()` und `expand.a.to.b()`. Die erste rotiert einen String, die zweite macht die Elemente gleichgroß bzgl. der Zeichenzahl, die dritte expandiert einen Vektor auf die Größe eines zweiten Vektors. In `expand.text()` werden erst die Verlängerungswünsche umgesetzt und dann eine Angleichung vorgenommen. Das Auffüllen mit Leerzeichen kann prinzipiell rechts wie auch links erwünscht sein.

53 ⟨*definiere einige Funktionen zur Anordnung von Textobjekten* 53⟩ ≡ ⊂ 51
```
expand.text <- function(x,N=0,O=0,S=0,W=0,fill.right=TRUE,sep=" "){
  # bringt Elemente auf gleiche Laenge und ergaenzt Leerzeichen in den Himmelsrichtungen
  if(0<O) x <- paste(x,paste(rep(sep,O),collapse=""),sep="")
  if(0<W) x <- paste(  paste(rep(sep,W),collapse=""),x,sep="")
  if(0<S) x <- c(x,rep(sep,S)); if(0<N) x <- c(rep(sep,N),x)
  maxch <- max(nchar(x));  LZ <- paste(rep(sep,maxch),collapse="")
  ch <- substring(LZ,1,maxch-nchar(x))
  x <- if(fill.right) paste(x,ch,sep="") else paste(ch,x,sep="")
  return(x)
} # test: cbind(expand.text("asdf",3,2,1,4,fill.right=FALSE,sep=" "))
```

`expand.a.to.b` geht von dem Ansatz aus, dass das erste Objekt anhand der Verhältnisse des zweiten Objekts angepasst und dabei aber nur vergrößert werden soll. Unterscheiden sich die Längen, kann eine Verlängerung hinten (`fill.tail=TRUE`) aber auch vorn erwünscht sein. Der Fall einer Zentrierung wird noch nicht berücksichtigt. Hat Objekt `a` nun mindestens die Länge von `b`, wird in einem zweiten Schritt die maximale Zeichenzahl der Elemente beider Vektoren festgestellt und dann werden entweder rechts (`fill.right=TRUE`) oder links so viele Leerzeichen ergänzt, dass alle Elemente von `a` die maximale Zeichenzahl besitzen.

54 ⟨*definiere einige Funktionen zur Anordnung von Textobjekten* 53⟩+ ≡ ⊂ 51

```
expand.a.to.b <- function(a,b,fill.right=TRUE,fill.tail=TRUE){
  # expandiert Vektor a auf die Dimensionen von Vektor b
  d <- length(b) - length(a)
  if(0 < d) a <- if(fill.tail) c(a, rep(" ",d)) else c(rep(" ",d), a)
  n.LZ <- max(nchar(b),nchar(a)); d <- n.LZ - nchar(a)
  LZ <- paste(rep(" ",n.LZ),collapse="")
  a <- ifelse( d <= 0, a, if(fill.right) paste(a,substring(LZ,1,d),sep="")
                          else paste(substring(LZ,1,d),a,sep=""))
  return(a)
}
```

`rotate.string()` ist wird mit einer impliziten Schleife in Form von `sapply()` umgesetzt.

55 ⟨*definiere einige Funktionen zur Anordnung von Textobjekten* 53⟩+ ≡ ⊂ 51

```
rotate.string <- function(x){
    x <- sapply(x,function(y) { h <- nchar(y); paste(substring(y,h:1,h:1),collapse="") })
}
```

Jetzt können wir die Kombination der Tiefen, Blätter und des Stammes ins Auge fassen. Falls ein richtiger Back-To-Back-Display entstehen soll, ist das Zusammenfügen etwas anders, als wenn der Anwender lieber zwei parallele Halbbäume mit einem Stamm ganz links bekommen möchte. Weiterhin ergibt sich ein Unterschied dadurch, wenn auf die Tiefeninformation verzichtet werden soll. Je nach den verschiedenen Fällen merken wir uns ein paar markante Stellen, um für die Platzierung weiterer Informationen eine Orientierung zu haben. Außerdem legen wir einen String aus Leerzeichen auf `LZ` ab, dessen Zeichenzahl mit denen der Elemente von `result` übereinstimmt.

56 ⟨*kombiniere depths, stem and leaves* 56⟩ ≡ ⊂ 51

```
if(back.to.back){
  if(show.no.depths){
    result <- vecpaste("",x.leaves,x.stem,y.leaves,"",sep=c("  ","","| "," |","","  "))
  } else {
    result <- vecpaste(x.depths,x.leaves,x.stem,y.leaves,y.depths,
                       sep=c("  ","  ","| "," |","  ","  "))
  }
  end.of.x.leaves <- sum(result$widths[1:4])
  end.of.x.attributes <- sum(result$widths[1:4])
  start.of.y.attributes <- sum(result$widths[1:7]) # falls x.name sehr lang
  space.x.to.y   <- sum(result$widths[5:7])
  end.of.left.stem <- 0
  extr.width <- end.of.x.leaves
} else { # parallel displays
  if(show.no.depths){
    result <- vecpaste(x.stem,rotate.string(x.leaves),y.leaves,sep=c("  "," |","  |","  "))
    end.of.x.leaves <- sum(result$widths[1:4])
    end.of.x.attributes <- sum(result$widths[1:4])
    start.of.y.attributes <- sum(result$widths[1:4]) # falls x.name sehr lang
    space.x.to.y   <- sum(result$widths[5])
  } else {
    result <- vecpaste(x.stem,rotate.string(x.leaves),x.depths,y.leaves,y.depths,
                       sep=c("  "," |","  "," |","  "," "))
    end.of.x.leaves <- sum(result$widths[1:6])
    end.of.x.attributes <- sum(result$widths[1:6])
    start.of.y.attributes <- sum(result$widths[1:7]) # falls x.name sehr lang
    space.x.to.y   <- sum(result$widths[7]) #- 5
  }
  end.of.left.stem <- sum(result$widths[1:3])
  extr.width <- end.of.x.leaves-end.of.left.stem
```

```
}
result <- result$result
LZ <- paste(rep(" ",max(nchar(result))),collapse="")
```

Für die Operation "Zusammenfügen" wurde die Funktion `vecpaste()` verwendet. Dieser Funktion kann eine Liste von zu verschmelzenden Objekten übergeben werden. Diese Liste ist zunächst zu identifizieren. An den Verschmelzungsstellen lässt sich eine Fuge einbringen, die über das Argument `sep` beschrieben wird. Sollen alle Fugen oder Verbindungsstellen gleich aussehen, reicht die Angabe eines skalaren Wertes aus. Sonst muss `sep` ein Vektor sein, der einen Wert mehr als die Objektzahl umfasst. Denn die Fugen werden nicht nur zwischen den Objekten eingefügt, sondern auch am Rand. Ist `sep` nummerisch, wird die Angabe als gewünschte Anzahl von Leerzeichen interpretiert. Zur Verschmelzung werden sicherheitshalber die Elemente eines Objekts vereinheitlicht und dann mit der zugehörigen Fuge auf dem Ergebnisvektor abgelegt. Damit später die Information über die Größen der Teilblöcke verfügt werden kann, werden auf `widths` diese Breiten (in Zeichenanzahlen) gesammelt und mit ausgegeben.

Um eine zweite Situation abzubilden, können für die einzelnen Blöcke, die verschmolzen werden sollen auch Breiten vorgegeben werden. Für diesen Fall müssen an das Argument `widths` die Bereiten übergeben werden. Dann erhalten Zwischenräume eine Ausdehnung von 0, die Blöcke werden um Leerzeichen erweitert und dann aneinandergefügt.

57 ⟨*definiere einige Funktionen zur Anordnung von Textobjekten* 53⟩+ ≡ ⊂ 51

```
vecpaste <- function(..., sep=" ", widths=NULL, fill.right = TRUE){  ## ???
  # vecpaste pastet Vektoren zusammen
  xyz <- list(...); n <- length(xyz)
  LZ <- paste(rep(" ",200),collapse="")
  if(0 < length(widths)) sep <- "" else widths <- rep(0,n)
  if(is.numeric(sep)) sep <- substring(LZ,1,sep)
  if(length(sep)==1)  sep <- rep(sep,n+1)
  if(length(fill.right)==1)  fill.right <- rep(fill.right,n)
  result <- ""; w <- NULL
  for(i in 1:n){ #print(fill.right[i]); cat("n",n,"i",i)
    h <- as.character(xyz[[i]]); h <- expand.text(h,fill.right=fill.right[i])
    if(nchar(h[1]) < widths[i])
      if(fill.right[i]){
        h <- paste(h,substring(LZ,1,widths[i]-nchar(h[1])),sep="")
      }else{
        h <- paste(substring(LZ,1,widths[i]-nchar(h[1])),h,sep="")
      }
    result <- paste(result,h,sep=sep[i])
    w <- c(w, nchar(sep[i]), nchar(h[1]))
  }
  if(0 == length(widths)) widths <- c(w,nchar(sep[n+1])) else widths <- w
  result <- paste(result,"",sep=sep[n+1])
  return(list(result=result,widths=widths))
}
```

Mit Hilfe von `vecpaste()` und den Blockbreiten ist die Erstellung der Informationszeilen nicht weiter problematisch. Es muss immer aufgezählt werden, was zu kombinieren ist, und es müssen die Breiten in einem Vektor zusammengefasst angegeben werden. Falls der erste Datensatzname zu lang ist, soll der zweite in einer gesonderten Zeile eingetragen werden.

58 ⟨*komponiere Zeilen mit Infos, Namen, Längen und NA-Anzahlen* 58⟩ ≡ ⊂ 51

```
# compose info line
info.line <- vecpaste(sl.x$info[1],sl.x$info[2],sep=c("  ","," "," "))$result
end.of.attr <- max(end.of.left.stem,4)
# compose name line
if( nchar(x.name) < end.of.x.leaves) {
   name.line <- vecpaste("    ",x.name," ",y.name,
        widths=c(end.of.attr, end.of.x.leaves-end.of.attr, space.x.to.y, end.of.x.leaves-4),
        fill.right=c(TRUE,!back.to.back,TRUE,TRUE))$result
} else {
   name.line <- c(vecpaste(x.name,sep=c(end.of.x.leaves - nchar(x.name),0))$result,
                  vecpaste(y.name,sep=c(start.of.y.attributes,0))$result)
}
# compose length line
```

```
n.line <- vecpaste("n:  ",length(x)," ",length(y),
        widths=c(end.of.attr, end.of.x.leaves-end.of.attr, space.x.to.y, end.of.x.leaves-4),
        fill.right=c(TRUE,!back.to.back,TRUE,TRUE))$result
# compose NA line
na.line <- vecpaste("NAs:",n.na.x," ",n.na.y,
        widths=c(end.of.attr, end.of.x.leaves-end.of.attr, space.x.to.y, end.of.x.leaves-4),
        fill.right=c(TRUE,!back.to.back,TRUE,TRUE))$result
if(0==length(grep("[1-9]",na.line))) na.line <- "    "
```

Jetzt müssen nur noch die Extremwerte abgehandelt werden. Da sich eine ganze Reihe einstellen können, sollte man diese ggf. auf mehrere Zeilen aufteilen. Die Textvektoren der jeweiligen Extremwerte sind dann an passenden Stellen aufzusplitten. Diesen Job übernimmt die Funktion `line.to.textvec()`. Sowohl für die Lower- wie auch für die Upper-Werte müssen die sich ergebenden Textvariablen zum Schluss wieder mit `expand.a.to.b()` geeignet expandiert werden.

59 ⟨*compose info of HI values* 59⟩ ≡ ⊂ 51

```
# compose info of HI values
## split HI info of x in smaller parts
upper.x.res <- line.to.textvec(sl.x$upper, extr.width)
## split HI info of y in smaller parts
upper.y.res <- line.to.textvec(sl.y$upper, extr.width)
## unify x and y HI values
upper.x.res <- expand.a.to.b(upper.x.res, upper.y.res)
upper.y.res <- expand.a.to.b(upper.y.res, upper.x.res)
## compose HI line
if(back.to.back){
  upper.line <- vecpaste(upper.x.res,upper.y.res,sep=c(0,space.x.to.y,0))$result
} else {
  upper.line <- vecpaste(" ",upper.x.res," ",upper.y.res,
     widths=c(end.of.attr,end.of.x.leaves-end.of.attr,space.x.to.y,end.of.x.leaves-4),
     fill.right=c(TRUE,TRUE,TRUE,TRUE))$result
}
```

Für die extrem kleinen Werte sind die selben Überlegungen wie zu den extrem großen Werten anzustellen.

60 ⟨*compose info of LO values* 60⟩ ≡ ⊂ 51

```
# compose info of LO values
## width=space by depths, leaves and 2 outer and 2 between
## split LO info of x in smaller parts
lower.x.res <- line.to.textvec(sl.x$lower, extr.width)
## split LO info of y in smaller parts
lower.y.res <- line.to.textvec(sl.y$lower, extr.width)
## unify x and y LO values
lower.x.res <- expand.a.to.b(lower.x.res, lower.y.res)
lower.y.res <- expand.a.to.b(lower.y.res, lower.x.res)
## compose LO line
if(back.to.back){
  lower.line <- vecpaste(lower.x.res,lower.y.res,sep=c(0,space.x.to.y,0))$result
} else {
  lower.line <- vecpaste(" ",lower.x.res," ",lower.y.res,
     widths=c(end.of.attr,end.of.x.leaves-end.of.attr,space.x.to.y,end.of.x.leaves-4),
     fill.right=c(TRUE,TRUE,TRUE,TRUE))$result
}
```

Für die Aufspaltung eines Textvektors in Zeilen, wird der Input zunächst zu einem Element verschmolzen. Falls es keine Beschränkung der Zeilenbreite gibt, ist nichts zu tun. Wenn der Input in eine Zeile passt, wird er auf die angegebene Zeilenlänge mit Leerzeichen aufgefüllt. Das Vorgehen ist, wie folgt: Der Input wird an Stellen des Separators `sep` in Einzelstücke zerlegt. Dann werden immer so viele Einzelstücke genommen, wie in eine Zeile passen. Passt ein sehr langes Stück nicht in eine Zeile, wird dieses gesplittet und eine Warnung ausgegeben.

61 ⟨*definiere einige Funktionen zur Anordnung von Textobjekten* 53⟩+ ≡ ⊂ 51

```
line.to.textvec <- function(vec,width,sep=" "){
  # wandelt Textzeile in Textvektor vorgegebener maximaler Zeichenanzahl um
  vec <- paste(vec,collapse=" "); if(width==Inf) return(vec)
  d <- width - nchar(vec); LZ <- paste(rep(" ",width),collapse="")
  if( 0 < d ) return(paste(vec,substring(LZ,1,d),sep=""))
```

```
  vec.sep <- unlist(strsplit(vec,sep)); result <- NULL; imax <- 200
  while(0 < length(vec.sep)){        if( (imax<-imax-1) < 0 ) break
    h <- sum( cumsum(nchar(vec.sep)+1) <= width ) # +1 fuer Trennzeichen
    if( h==0 ) {
      h <- 1; cat("Warning: word '",vec.sep[1],"' too long ...",sep="")
      vec.sep <- c(substring(vec.sep[1],1,width),substring(vec.sep[1],1+width),vec.sep[-1])
      h <- 1; cat("  and has been split to '",vec.sep[1],"' and '",vec.sep[2],"'",sep="")
    }
    new.line <- paste(vec.sep[1:h],collapse=sep)
    d <- width - nchar(new.line)
    if( d < 0 ) new.line <- substring(new.line,1,width)
    if( 0 < d ) new.line <- paste(new.line,substring(LZ,1,d),sep="")
    result <- c(result, new.line); vec.sep <- vec.sep[-(1:h)]
  }
  return(result)
}
```

6.6 Some Tests

Wir definieren eine Reihe von Tests ganz einfach dadurch, dass wir die Test-Aufrufe von `stem.leaf()` abändern. Hierzu definieren wir zunächst die Funktion `test()` so, dass sie statt `stem.leaf()` die Funktion `stem.leaf.backback()` mit doppelter Verwendung der Daten aufruft.

62 ⟨* 18⟩+ ≡

```
test <- function(what) {
  what <- sub("stem.leaf","stem.leaf.backback",what)
  cat(what,"\n"); eval(parse(text=what)); return()
}
```

⟨*Test von* `style` 37⟩
⟨*Test von* `rule.line` 38⟩
⟨*Test von* `unit` 39⟩
⟨*Test von* `Min/Max` 40⟩
⟨*Klassenzuordnungstest* 41⟩
⟨*Example of Jim Albert* 42⟩
⟨*reverse-Test* 43⟩

Als nächstes schauen wir, ob die Veränderung weiterer Parameter klappt. Paralleles Blätterwerk erhalten wir mit Hilfe von `back.to.back=FALSE`.

63 ⟨* 18⟩+ ≡

```
test <- function(what) {
  what <- sub("stem.leaf.","stem.leaf.backback(back.to.back=FALSE,",what)
  cat(what,"\n"); eval(parse(text=what)); return()
}
```

⟨*Test von* `style` 37⟩
⟨*Test von* `rule.line` 38⟩
⟨*Test von* `unit` 39⟩
⟨*Test von* `Min/Max` 40⟩
⟨*Klassenzuordnungstest* 41⟩
⟨*Example of Jim Albert* 42⟩
⟨*reverse-Test* 43⟩

Wie sehen die Displays aus, wenn auf die Tiefeninformation verzichtet werden soll?

64 ⟨* 18⟩+ ≡

```
test <- function(what) {
  what <- sub("stem.leaf.",
              "stem.leaf.backback(show.no.depths=TRUE,add.more.blanks=10,",what)
  cat(what,"\n"); eval(parse(text=what)); return()
}
```

⟨*Test von* `style` 37⟩
⟨*Test von* `rule.line` 38⟩
⟨*Test von* `unit` 39⟩

⟨*Test von* `Min/Max` 40⟩
⟨*Klassenzuordnungstest* 41⟩
⟨*Example of Jim Albert* 42⟩
⟨*reverse-Test* 43⟩

Für singuläre Tests kann folgender Chunk verwendet werden:

65 ⟨*doit* 65⟩ ≡
```
⟨definiere stem.leaf.backback 45⟩
"a" <- structure(c(12, 29, 49, 280, 78, 41, 49, 308, 70, 57,
 41, 37, 275, 33, 267, 37, 33, 57, 37, 41, 25, 41, 53, 74,
57, 53, 37, 49, 66, 70, 134, 33, 57, 45, 62, 250, 37, 271,
37, 41, 12, 70, 25))
stem.leaf(a); stem.leaf(a,Max=70)
dat1 <- c(-2000,co2[113:130],NA,NA,500,600,700,1000)
dat2 <- c(-1000,-10000,-5000,co2[170:140],NA,NA,NA,500,600,700,10000)
dat1 <- c(co2[113:130],NA,NA,500,600,700,1000)#; dat2 <- dat1
dat1 <- dat2
a <- stem.leaf.backback(dat1, c(dat2,dat2,dat2) ,m=2,unit=.1)
a <- stem.leaf.backback(c(co2[113:130],NA,NA,500),c(co2[170:140],NA),
                        m=2,unit=.1)
stem.leaf(c(co2[1:50],NA,400),m=5,unit=1)
stem.leaf.backback(co2)
NULL
```

66 ⟨* 18⟩+ ≡
```
⟨definiere stem.leaf.backback 45⟩
a <- stem.leaf.backback(dat1, c(dat2,dat2,dat2) ,m=2,unit=.1,
                        show.no.depths=TRUE,add.more.blanks=0)
a <- stem.leaf.backback(c(dat2,dat2), c(dat2,dat2) ,m=2,unit=.1,
                        show.no.depths=!TRUE,add.more.blanks=10)
NULL
```

67 ⟨* 18⟩+ ≡
```
⟨definiere stem.leaf.backback 45⟩
a <- stem.leaf.backback(dat1, c(dat2,dat2,dat2) ,m=2,unit=.1,
                        show.no.depths=TRUE,add.more.blanks=0)
a <- stem.leaf.backback(c(dat2,dat2), c(dat2,dat2) ,m=2,unit=.1,
                        show.no.depths=TRUE,add.more.blanks=10)
a <- stem.leaf.backback(c(dat2,dat2), c(dat2,dat2) ,m=2,unit=.1,
                        show.no.depths=TRUE,add.more.blanks=10,back.to.back=!TRUE)
NULL
```

68 ⟨* 18⟩+ ≡
```
⟨definiere stem.leaf.backback 45⟩
a <- stem.leaf.backback(dat1, c(dat2,dat2,dat2) ,m=2,unit=.1,
                        show.no.depths=TRUE,add.more.blanks=0)
a <- stem.leaf.backback(c(dat2), 1.02*c(dat2,dat2) ,m=2,unit=.1,
                       show.no.depths=TRUE,add.more.blanks=0)
a <- stem.leaf.backback(c(co2[1:284]), c(co2[1:100]) ,m=1,unit=1,
                       show.no.depths=!TRUE,add.more.blanks=2,rule.line="Sturges")
a <- stem.leaf.backback(co2,co2 ,m=2,unit=1,
                        show.no.depths=!TRUE,add.more.blanks=2,rule.line="Sturges")
NULL
```

Beispiele aus der Hilfe:

69 ⟨* 18⟩+ ≡
```
⟨definiere stem.leaf.backback 45⟩
stem.leaf(co2)
stem.leaf.backback(co2[1:120],co2[121:240])
stem.leaf.backback(co2[1:120],co2[121:240], back.to.back = FALSE)
stem.leaf.backback(co2[1:120],co2[121:240], back.to.back = FALSE,
                   add.more.blanks = 3, show.no.depths = TRUE)
stem.leaf.backback(rivers[-(1:30)],rivers[1:30], back.to.back = FALSE, unit=10, m=5,
                   Min=200, Max=900, add.more.blanks = 20, show.no.depths = TRUE)
```

```
NULL
```

Error mailed by John Fox, 30.Okt.2013 in `stem.leaf.backback`:

70 ⟨*jferror* 70⟩ ≡

```
# style <- "Tukey"
style <- "bare"
⟨definiere stem.leaf.backback 45⟩
library(car,lib.loc="/home/pwolf/lib")
data(Moore, package="car")
with(Moore, stem.leaf.backback(conformity[partner.status == "low"],
     conformity[partner.status == "high"],style=style))            # ok
"ok"
```

Here is the correct output:

```
------------------------------
  1 | 2: represents 12, leaf unit: 1
conformity[partner.status == "low"]
                conformity[partner.status == "high"]
------------------------------
            | 0 |
            | 0 |
   3      444| 0 |
   8    77766| 0 |77       2
  (5)   99888| 0 |89       4
   9        0| 1 |01       6
   8   333222| 1 |223      9
            | 1 |44455   (5)
            | 1 |66777    9
            | 1 |9        4
            | 2 |01       3
   2        3| 2 |3        1
   1        4| 2 |
            | 2 |
            | 2 |
            | 3 |
------------------------------
n:          22     23
------------------------------
```

Up to October 2013 an error occurs because x.pos and y.pos are vectors.

```
Error in rep(" ", d <- abs(y.pos - x.pos)) : ung\"ultiges 'times' Argument
sorry, evaluation not successful!!!
```

71 ⟨* 18⟩+ ≡

```
x <- Moore[Moore$partner.status == "low", "conformity"]
y <- Moore[Moore$partner.status == "high", "conformity"]
y <- y[y<9]
stem.leaf.backback(x,x,style=style,unit=1) # ok
""
```

7 Anhang

7.1 Chunk-Liste

Code Chunk Index

⟨*definiere* `stem.leaf` 3⟩ ⊂ 2, 45 ..... p2
⟨*definiere* `stem.leaf.backback` 45⟩ ⊂ 2, 65, 66, 67, 68, 69, 70 ..... p19
⟨*definiere* `test` 36⟩ ⊂ 35, 44 ..... p12
⟨*doit* 65⟩ p26
⟨*ergänze Leerzeichen zur Vereinheitlichung der Blatt-Variablen* 52⟩ ⊂ 51 ..... p21
⟨*ermittle Äste mit Blättern* 21⟩ ⊂ 16 ..... p7
⟨*ermittle Blätter u.ä. der beiden Datensätze in* `stem.leaf.backback()` 49⟩ ⊂ 45 ..... p20
⟨*ermittle geeignete Parameterwerte, sofern nicht festgelegt* 48⟩ ⊂ 47 ..... p20
⟨*ermittle mittels* `boxplot` *groben Skalenbereich* 7⟩ ⊂ 4, 48 ..... p3
⟨*ermittle Tiefen und füge sie an zentralen Plot an* 28⟩ ⊂ 16 ..... p8
⟨*ermittle Zeilennamen für den Stamm* 25⟩ ⊂ 22 ..... p8
⟨*erstelle Interpretationshilfen* 29⟩ ⊂ 14 ..... p9
⟨*erstelle Stem-and-Leaf-Display* 14⟩ ⊂ 3 ..... p5
⟨*Example of Jim Albert* 42⟩ ⊂ 44, 62, 63, 64 ..... p17
⟨*generiere die Skala für den Plot* 4⟩ ⊂ 3 ..... p2
⟨*jferror* 70⟩ p27
⟨*Klassenzuordnungstest* 41⟩ ⊂ 44, 62, 63, 64 ..... p16
⟨*kombiniere depths, stem and leaves* 56⟩ ⊂ 51 ..... p22
⟨*komponiere Output von* `stem.leaf.backback()` 51⟩ ⊂ 45 ..... p21
⟨*komponiere Zeilen mit Infos, Namen, Längen und NA-Anzahlen* 58⟩ ⊂ 51 ..... p23
⟨*konstruiere Skala und füge sie an den zentralen Plot an* 22⟩ ⊂ 16 ..... p7
⟨*konstruiere zentralen Teil des Plots* 16⟩ ⊂ 14 ..... p6
⟨*merke Ausreißer* 15⟩ ⊂ 14 ..... p5
⟨*merke negative Klassen und Klasse, die bei* −1 *beginnt* 23⟩ ⊂ 21 ..... p7
⟨*modifiziere Zeilennamen gemäß Maserung* 26⟩ ⊂ 22 ..... p8
⟨*reverse-Test* 43⟩ ⊂ 44, 62, 63, 64 ..... p17
⟨*setze ggf.* `verb` *gemäß Debugging-Wunsch* 33⟩ ⊂ 3, 48 ..... p11
⟨*spiegele ggf. Blätter im negativen Bereich* 24⟩ ⊂ 21 ..... p7
⟨*start* 2⟩ p2
⟨*stelle Ergebnis zusammen* 30⟩ ⊂ 3 ..... p9
⟨*stelle gemäß* `rule.line` *maximale Zeilenanzahl fest* 6⟩ ⊂ 4, 48 ..... p3
⟨*suche gemeinsamen Stamm in* `stem.leaf.backback()` 47⟩ ⊂ 45 ..... p19
⟨*synchronisiere stems, leaves und depths der beiden Datensätze* `stem.leaf.backback()` 50⟩ ⊂ 45 ..... p20
⟨*Testaufrufe* 44⟩ p18
⟨*teste* `stem.leaf` 31⟩ p9
⟨*Test von* `Min/Max` 40⟩ ⊂ 44, 62, 63, 64 ..... p15
⟨*Test von* `rule.line` 38⟩ ⊂ 44, 62, 63, 64 ..... p13
⟨*Test von* `style` 37⟩ ⊂ 44, 62, 63, 64 ..... p12
⟨*Test von* `unit` 39⟩ ⊂ 44, 62, 63, 64 ..... p14
⟨*transformiere Daten* 12⟩ ⊂ 4 ..... p4
⟨*verteile Blätter auf passende Klassen* 20⟩ ⊂ 16 ..... p7
⟨*zeige Beziehung Werteanzahl Zeilenanzahl gemäß Regel* 8⟩ p3
⟨*zerlege Zahlen in Stamm und Blatt* 17⟩ ⊂ 16 ..... p6
⟨*Zugabe* 74 ∪ 75 ∪ 76 ∪ 77⟩ ..... p**??**
⟨*Zugabe: expand.A.to.B* 72⟩ ..... p**??**
⟨*Zugabe: line.to.textmat* 73⟩ ..... p**??**

7.2 Objekt-Index

Object Index