# Kapitel 3

# Daten Abstraktion

## 3.1 Datenstrukturen und Typen in Haskell

Bisher haben wir nur die eingebauten Basisdatentypen wie Zahlen und Wahrheitswerte benutzt. Der Vollständigkeit halber: Es gibt noch den Typ `Char` für Zeichen, die als `'a'` eingegeben werden müssen. Einfache Funktionen sind `ord`, die die Hex-Codierung eines Zeichens angeben und `chr`, die Umkehrfunktion von `ord` ist.

Wir werden nun größere, zusammengesetzte Datenobjekte als Abstraktion von Klassen von Objekten verwenden.

### 3.1.1 einfache Typen

*Beispiel 1.* **Rationale Zahlen** Eine rationale Zahl kann man als zusammengesetztes Objekt verstehen, das aus zwei Zahlen, dem Zähler und dem Nenner besteht. Normalerweise schreibt man $\frac{x}{y}$ für die rationale Zahl mit Zähler $x$ und dem Nenner $y$.

Die einfachste Methode, dies in Haskell darzustellen, ist als Paar von Zahlen $(x, y)$. Beachte, daß in Haskell 1.4 rationale Zahlen bereits vordefiniert sind. Z.B.:

```
>  (3%4)*(4%5)
3 % 5
```

Datenkonversionen macht man mit `toRational` bzw. `truncate`. Es gibt rationale Zahlen mit kurzen und beliebig langen ganzen Zahlen.

Paare von Objekten kann man verallgemeinern zu $n$-Tupel von Objekten:

$(t_1, \ldots, t_n)$ stellt ein *$n$-Tupel* der Objekte $t_1, \ldots, t_n$ dar.

Um mit Datenobjekten operieren zu können, benötigt man:

**Datenkonstruktor:** Funktion, die ein Datenobjekt neu konstruiert, wobei die Teile als Parameter übergeben werden

**Datenselektor:** Funktion, die gegeben ein Datenobjekt, bestimmte Teile daraus extrahiert.

Zum Beispiel konstruiert man ein Paar $(s, t)$ aus den beiden Ausdrücken $s$ und $t$. Da man die Ausdrücke wieder extrahieren können muß benötigt man die Selektoren `fst` und `snd`, für die gelten muß: $\texttt{fst}(s, t) = s$ und $\texttt{snd}(s, t) = t$.

Für ein $n$-Tupel benötigt man $n$ Datenselektoren, für jede Stelle des Tupels einen. Die Definition dieser Selektoren wird in Haskell syntaktisch vereinfacht durch sogenannte *Muster* (*pattern*). Ein einfaches Beispiel einer solchen Definition ist:

```
selektiere_3_von_5 (x1,x2,x3,x4,x5) = x3
```

Diese Muster sind syntaktisch überall dort erlaubt, wo formale Parameter (Variablen) neu eingeführt werden, d.h. in Funktionsdefinitionen, in Lambda-ausdrücken und in let-Ausdrücken.

*Bemerkung.* In Haskell kann man Typen und Konstruktoren mittels der data-Anweisung definieren.

```
data Strecke  = Streckenkonstruktor Punkt Punkt
data viertupel a b c d =  viertupelkons a1 a2 a3 a4
```

**Definition 1.** Ein `Muster` ist ein Ausdruck der nach folgender Syntax erzeugt ist: Muster ::= $\langle \text{Variable} \rangle \mid \langle \text{Konstruktor}_{(n)} \rangle\ \underbrace{\langle \text{Muster} \rangle \ldots \langle \text{Muster} \rangle}_{n}$

Eine Kontextbedingung ist, daß in einem Muster keine Variable doppelt vorkommt.

Die erlaubten Transformationen bei Verwendung von Mustern kann man grob umschreiben als: wenn das Datenobjekt wie das Muster aussieht und die Konstruktoren übereinstimmen, dann werden die Variablen an die entsprechenden Ausdrücke (Werte) gebunden. Diese `Musteranpassung` kann man rekursiv beschreiben als:

- `anpassen` $Kon\ Kon =$ ok.
- `anpassen var` $t$ =: `var` wird an $t$ gebunden.
- `anpassen` $(Kon\ s_1 \ldots\ s_n)\ (Kon\ t_1 \ldots t_n) =$
  `anpassen` $s_1\ t_1$ und …und `anpassen` $s_n\ t_n$

- `anpassen` $(Kon\ s_1 \ldots\ s_n)$ $(Kon'\ t_1 \ldots t_n)$ = Fail, wenn Kon $\neq$ Kon$'$

Dies bedeutet auch, daß die Musteranpassung erzwingt, daß die Datenobjekte, die an das Muster angepaßt werden sollen, zunächst ausgewertet werden müssen. Muster wirken wie ein let-Ausdruck mit Selektoren kombiniert.

Man kann Zwischenstrukturen in Mustern ebenfalls mit Variablen benennen:

Das Muster $(x, y@(z_1, z_2))$ wird bei der Musteranpassung auf $(1, (2, 3))$ folgende Bindungen liefern: $x = 1, y = (2, 3), z_1 = 2, z_2 = 3$.

*Beispiel 2.* Die Selektoren `fst` und `snd` für den Paarkonstruktor sind definiert als:

```
fst (x,y) = x
snd (x,y) = y
```

Jedes Datenobjekt in Haskell muß einen Typ haben. Ein Typ korrespondiert somit zu einer Klasse von Datenobjekten.

Syntaktisch wird für jeden Typ ein Name eingeführt. Die eingebauten arithmetischen Datenobjekte haben z.B. die Typen `Int, Integer, Float, Char, ...`. Komplexere Typen und zugehörige Datenobjekte werden durch ihre obersten Konstruktoren charakterisiert.

Datenkonstruktoren können explizit definiert werden und haben dann einen eigenen Status in der Sprache. Es gibt eine eigene Anweisung, die solche Daten definiert, wobei man auf die definierten Typnamen zurückgreift (auch rekursiv). Es genügt, die Datenkonstrukoren zu definieren, da die Selektoren durch Musteranpassung (Musterinstanziierung) definierbar sind.

*Beispiel 3.* Punkte und Strecken in der Zahlenebene dargestellt durch Koordinaten:

```
data Punkt    = Punktkonstruktor Int Int
data Strecke  = Streckenkonstruktor Punkt Punkt
data viertupel a b c d =  viertupelkons a1 a2 a3 a4
```

Hier ist `Punkt` ein neu definierter Typ, `Punktkonstruktor` der neu definierte Konstruktor für Punkte der Ebene, `Strecke` ist ein neuer Typ, der aus zwei Punkten besteht. Es ist unproblematisch, Typ und Konstruktor gleich zu benennen, da keine Verwechslungsgefahr besteht:

```
data Punkt    = Punkt Int Int
data Strecke  = Strecke Punkt Punkt
data Vektor  = Vektor Int Int
```

Haskell sorgt mit der Typüberprüfung dafür, daß z.B. Funktionen, die für Punkte definiert sind, nicht auf rationale Zahlen angewendet werden, die ebenfalls aus zwei Zahlen bestehen.

Einige Funktionen, die man jetzt definieren kann sind:

```
addiereVektoren (Vektor a1 a2) (Vektor b1 b2) = Vektor (a1 + b1) (a2 + b2)

streckenLaenge (Strecke (Punkt a1 a2) (Punkt b1 b2)) =
         sqrt (fromInteger ((quadrat (a1 - b1)) + (quadrat  (a2-b2))))

verschiebeStrecke s v =
        let  (Strecke (Punkt a1 a2) (Punkt b1 b2)) = s
             (Vektor v1 v2) = v
                  in  (Strecke (Punkt (a1+v1) (a2+v2)) (Punkt (b1+v1) (b2+v2)))

teststrecke = (Strecke (Punkt 0 0) (Punkt 3 4))

test_streckenlaenge = streckenLaenge
               (verschiebeStrecke teststrecke
                                   (Vektor 10 (-10)))

--------------------------------------------------------
streckenLaenge teststrecke      <CR>

> 5.0

test_streckenlaenge                     <CR>

> 5.0
```

Wenn wir die Typen der Funktionen überprüfen, erhalten wir, wie erwartet:

```
addierevektoren :: Vektor -> Vektor -> Vektor
streckenlaenge :: Strecke -> Float
test_streckenlaenge :: Float
verschiebe_strecke :: Strecke -> Vektor -> Strecke
```

### 3.1.2 Summentypen und Fallunterscheidung

Bisher können wir noch keine Entsprechung des Booleschen Datentyps selbst definieren. Dies ist einfach, denn wir können Klassen von Datenobjekten verschiedener Struktur in einer Klasse vereinigen, indem wir einem Typ verschiedene Konstruktoren zuordnen: Dies ist gleichzeitig die Deklaration (Erklärung) eines syntaktischen Datentyps.

```
data Wahrheitswerte = Wahr | Falsch
```

Die zugehörigen Typen nennt man auch Summentypen.

Da Funktionen für Objekte unterschiedlicher Struktur eine Fallunterscheidung machen müssen, gibt es eine einfache Möglichkeit dies in Haskell hinzuschreiben: Man schreibt die Funktionsdefinition für jeden Fall der verschiedenen Muster der Argumente hin.

```
und1  Wahr Falsch    = Falsch
und1  Wahr Wahr       = Wahr
und1  Falsch Falsch  = Falsch
und1  Falsch Wahr    = Falsch
```

oder

```
und2  Wahr x   = x
und2  Falsch x  = Falsch
```

oder

```
und3  Wahr x   = x
und3  Falsch _  = Falsch
```

Der Unterstrich ist eine anonyme Mustervariable (wildcard).

**Definition 2.** Es gibt ein weiteres syntaktisches Konstrukt, den `case`-*Ausdruck*, der zur Fallunterscheidung verwendet werden kann. Die Syntax ist:

`case` ⟨Ausdruck⟩ `of`{⟨ Muster⟩ −> ⟨Ausdruck⟩;...;⟨Muster⟩ −> ⟨Ausdruck⟩}

Die Kontextbedingung ist, daß die Muster vom Typ her passen. Die Bindungsbereiche der Variablen in den Mustern sind genau die zugehörigen Ausdrücke hinter dem Pfeil (−>).

*Beispiel 4.* Folgende Definition ist äquivalent zum logischen und: `&&`.

```
und4  x y  = case x of True -> y; False -> False
```

*Beispiel 5.* Folgende Definition ist äquivalent zum normalen `if .then . else`. D.h. `case`-Ausdrücke sind eine Verallgemeinerung des `if . then . else`.

```
mein_if  x y z  = case x of True -> y; False -> z
```

## 3.2 Rekursive Datenobjekte: z.B. Listen

Listen sind eine Datenstruktur für Folgen von gleichartigen Objekten. Da wir beliebig lange Folgen verarbeiten und definieren wollen, gibt es die Möglichkeit, rekursive Datentypen zu definieren. Der Typ der Objekte in der Liste ist nicht festgelegt, sondern wird hier als (Typ-) Variable in der Definition verwendet.

```
data Liste a = Leereliste | ListenKons a (Liste a)
```

Dies ergibt Datenobjekte, die entweder leer sind: `Leereliste`, oder deren Folgenelemente alle den gleichen Typ $a$ haben, und die aufgebaut sind als `ListenKons` $b_1$ (`ListenKons` $b_2 \ldots Leereliste$)).

Listen sind in Haskell eingebaut und werden syntaktisch bevorzugt behandelt. Aber: man könnte sie völlig funktionsgleich auch selbst definieren. Im folgenden werden wir die Haskell-Notation verwenden. Die Definition würde man so hinschreiben, (allerdings entspricht diese nicht der Syntax).

```
data [a] = []   |   a : [a]
```

Listenobjekte werden auch dargestellt als Folge in eckigen Klammern: `[1,2,3,4,5]` ist die Liste der zahlen von 1 bis 5; die leere Liste wird einfach durch `[]` dargestellt. Listentypen werden mit eckigen Klammern notiert: `[ Int]` ist der Typ Listen von `Int`. Nun können wir auch den Typ von Tupeln hinschreiben: Tupel haben einen impliziten Konstruktor: $(.,\ldots,.)$. Der Typ wird entsprechend notiert: Der Typ von $(1,1)$ ist z.B . $(Int, Int)$, der Typ einer Liste von Paaren von langen ganzen Zahlen ist $[(Integer, Integer)]$, Der Typ einer Liste von Listen von Zeichen ist: $[[Char]]$.

Wir können jetzt rekursive Funktionen auf Listenobjekten definieren:

```
--   Laenge einer Liste
laenge []     = 0
laenge (_:xs) = 1 + laenge xs

--   map wendet eine Funktion f  auf alle Elemente einer Liste an.
map f []          = []
map f (x:xs)      = f x : map f xs
```

Die erste Funktion berechnet die Anzahl der Elemente einer Liste, d.h. deren Länge, die zweite Funktion wendet eine Funktion auf jedes Listenelement an und erzeugt die Liste der Resultate.

Strings oder Zeichenketten in Haskell sind Listen von Zeichen und können auch genauso verarbeitet werden. D.h. der Typ `String` ist äquivalent zu $[Char]$.

Zwei allgemeine Funktionen (Methoden), die Folgen von Elementen verarbeitet sind `foldl` und `foldr`. Argument ist eine zweistellige Operation, ein Einheitselement und die Liste.

```
foldl            :: (a -> b -> a) -> a -> [b] -> a
foldl f z []      = z
foldl f z (x:xs)  = foldl f (f z x) xs
```

```
foldr            :: (a -> b -> b) -> b -> [a] -> b
foldr f z []      = z
foldr f z (x:xs)  = f x (foldr f z xs)
```

Man kann sich vorstellen, daß für einen Operator $\otimes$ und ein Anfangselement (Einheitselement) $e$ der Ausdruck `foldl` $\otimes$ $e$ $[a_1, \ldots, a_n]$ äquivalent zu $((\ldots((e \otimes a_1) \otimes a_2) \ldots) \otimes a_n)$. Analog entspricht `foldr` $\otimes$ $e$ $[a_1, \ldots, a_n]$ der umgekehrten Klammerung: $a_1 \otimes (a_2 \otimes (\ldots (a_n \otimes e)))$. Für assoziative Operatoren und wenn $e$ Rechts und Linkseins ist, ergibt sich derselbe Wert. Die Operatoren `foldl` und `foldr` unterscheiden sich bzgl. des Ressourcenbedarfs in Abhängigkeit vom Operator und dem Typ des Arguments.

Beispiele für die Verwendung, wobei die bessere Variante definiert wurde.

```
sum xs = foldl (+) 0 xs
produkt xs = foldl (*) 1 xs          --   (foldl' (*) 1 xs)

concat xs = foldr (++) [] xs
```

Weitere wichtige Funktionen auf Listen sind:

```
append [] ys      = ys
append (x:xs) ys  = x : (append xs ys)

filter f []          = []
filter f (x:xs)      = if (f x) then  x : filter f xs
                       else filter f xs

--  umdrehen einer Liste
reverse xs = foldl (\x y -> y:x) [] xs

--- verbindet eine Liste von Listen zu einer einzigen Liste:

concat xs = foldr append [] xs
```

```
--   nimmt die ersten n Elemente der Liste xs.
take 0 _ = []
take n [] = []
take n (x:xs) = x : (take (n-1) xs)

randomInts a b
--     liefert ein Liste von (Pseudo-) Zufallszahlen
--     wenn import Random im File steht.
```

*Übungsaufgabe 3.* Die Funktionen `concat` und `concatl`

```
concatl xs = foldl (++) [] xs
concat xs = foldr (++) [] xs
```

sind gleich als Funktionen, aber haben verschiedenen Ressourcenbedarf. Demonstriere dies experimentell an verschiedenen Beispielen. Was ist der Grund dafür?

### 3.2.1 Listenausdrücke, list comprehensions

Dies ist eine Spezialität von Haskell und erleichtert die Handhabung von Listen. Die Syntax ist analog zu Mengenausdrücken, nur daß eine Reihenfolge der Listenelemente festgelegt ist. Syntax:

$$[\langle\text{Ausdruck}\rangle \mid \langle\text{Generator oder Filter}^+\rangle].$$

Vor dem senkrechten Strich | kommt ein Ausdruck, danach eine mit Komma getrennte Folge von Generatoren der Form $v -> $ `liste` oder von Prädikaten. Wirkungsweise: die Generatoren liefern nach und nach die Elemente der Listen. Wenn alle Prädikate zutreffen, wird ein Element entsprechend dem Ausdruck links von | in die Liste aufgenommen. Hierbei können auch neue lokale Variablen eingeführt werden, deren Geltungsbereich rechts von der Einführung liegt.

*Beispiel 6.*

```
[x   | x <- xs]            ergibt die Liste selbst
[f x | x <- xs]            ist dasselbe wie    map f xs
[x   | x <- xs, p x]       ist dasselbe wie     filter p xs

[(x,y) | x <- xs, y <-ys]  kartesische Produkt der Listen xs und ys

[y | x <- xs, y <-x]       entspricht der Funktion concat
```

Spezielle Listenausdrücke sind:

- `[n..m]` erzeugt eine Liste `[n, n+1,..., m]`
- `[n,m..e]` erzeugt eine Liste `[n, m, n+2*(m-n), ..., e]`
- `[n..]` erzeugt die (unendliche) Liste `[n,n+1,n+2,...]`.

## 3.3 Kontrollstrukturen, Iteration

Jetzt können wir auch Kontrollstrukturen wie `while`, `until` und `for` definieren, wobei das Typsystem bestimmte Einschränkungen für den Rumpf macht. Generell ist Rekursion allgemeiner als Iteration, allerdings sind vom theoretischen Standpunkt aus beide gleichwertig: wenn man den Speicher erweitern kann (d.h. beliebig große Datenobjekte verwenden und aufbauen kann), dann kann Iteration die Rekursion simulieren.

Die Definition dieser Kontrollstrukturen benutzt einen Datentyp als "Umgebung", so daß jeder Iterationsschritt die alte Umgebung als Eingabe hat, und die neue Umgebung als Ausgabe, die dann wieder als Eingabe für den nächsten Iterationsschritt dient. Dieser Effekt kann man auch beschreiben als: die Umgebung wird in jedem Iterationsschritt geändert. Der Rumpf ist eine Funktion, die genau diesen Schritt beschreibt.

*Beispiel 7.*

```
while:: (a -> Bool) -> (a -> a) -> a -> a
--    while test f init
--     Typ der Umgebung:  a
--     test:  Test, ob While-bedingung erfuellt
--     f:: a -> a    Rumpf, der die Umgebung abaendert
--    init:     Anfangsumgebung

while test f init =
        if test init
        then while test f (f init)
        else init

untill :: (a -> Bool) -> (a -> a) -> a -> a
untill test f init =
       if test init
       then init
       else untill test f (f init)

for :: (Ord a, Num a) => a -> a -> a -> (b -> a -> b) -> b -> b
```

```
--    For laeuft von start bis end in Schritten von schritt
--    Umgebung:: b, Zaehler vom Typ a
--    f: Rumpf mit Argumenten:  Umgebung und aktueller Zaehler
for start end schritt f init =
     if start > end
     then init
     else let startneu = start + schritt
             in for startneu end schritt f (f init start)
```

Die Funktion $f$, die als Argument mit übergeben wird, erzeugt ein neues Datenobjekt.

Verwendung: Z.B.

```
dreinwhile n = while (> 1) dreinschritt n
dreinschritt x = if x == 1 then 1
               else if geradeq x then x `div` 2
                    else 3*x+1

--   berechnet 1 + 2+ ...  + n:
summebis n = for 1 n 1 (+) 0

--   berechnet fibonacci (n)
fib_for n = fst (for 1 n 1 fib_schritt   (1,1))
fib_schritt (a,b) _ = (a+b,a)
```

*Übungsaufgabe 4.* Definiere eine Kontrollstruktur `forjp`, die wie `for` arbeitet, aber es der Funktion `f` erlaubt, den Zähler innerhalb des Rumpfes zu verändern.

Ein Ausdruck mit `foldl` läßt sich als `while`-Ausdruck schreiben:

```
foldl f e xs
```

ist äquivalent zu:

```
fst (while (\(res,list) -> list /= [])
      (\(res,list) -> (f res (head list), tail list))
      (e,xs))
```

Ein Ausdruck mit `foldr` läßt sich nicht so ohne weiteres als while-Ausdruck schreiben. Für endliche Listen ist das (unter Effizienzverlust) möglich:

```
foldr f e xs
```

ist äquivalent [1] zu:

```
fst (while (\(res,list) -> list /= [])
      (\(res,list) -> (f (head list) res, tail list))
      (e,reverse xs))
```

## 3.4 Sortieren

Die Sortierung von Mengen von Datensätzen ist eine häufige algorithmische Operation auf Mengen bzw. Folgen von gleichartigen Datenobjekten (insbesondere in der betriebswirtschaftlichen Datenverarbeitung). Wir betrachten im folgenden das etwas vereinfachte Problem der Sortierung einer gegebenen Liste (Menge) von ganzen Zahlen. Normalerweise gibt es einen Schlüsselbegriff nach dem sortiert wird, und zugehörige Attribute, die dem Schlüsselbegriff zugeordnet sind.

Im allgemeinen braucht man eine Ordnungsrelation (transitiv, reflexiv), die die Schlüsselbegriffe sortiert. Diese ist nicht automatisch gegeben. Z.B. ist die Sortierung von Texten, die im englischen / amerikanischen einfach auf der internen Darstellung durchgeführt wird, problematisch bei der Verwendung von Umlauten, da diese normalerweise aufgrund der internen Darstellung falsch einsortiert werden. Für die richtige Sortierung benötigt man die Definition einer passenden Ordnung auf den Zeichen, die z.B. ü zum u dazusortiert, d.h. $t < u < ü < v$, oder auch $t < u \sim ü < v$, wie in einem Wörterbuch.

Wir betrachten hier vier typische Sortierverfahren.

**Sortieren durch Einfügen** fügt die restlichen Elemente nach und nach in die bereits sortierte Liste der abgearbeiteten Zahlen.

**Sortieren mit Blasensort** Betrachtet jeweils zwei benachbarte Elemente und bringt diese in die richtige Reihenfolge. Der Vergleich läuft in zwei Schleifen: Die innere startet mit den letzten beiden Elementen und schiebt das Vergleichsfenster nach vorne. Danach ist das kleinste Element am Anfang der Liste. Dieses Verfahren wird jeweils mit der kleineren Liste wiederholt. Man kann abbrechen, wenn sich in einem inneren Durchlauf nichts verändert hat.

**Quicksort** Rekursives Verfahren, das das erste Element als Pivot nimmt, dann die Liste zerlegt in eine Liste der kleineren und eine Liste der grösseren Elemente, diese rekursiv sortiert, und dann einfach hintereinander hängt mit dem Pivot dazwischen.

**Mischsort** Rekursives Verfahren, das die Liste in zwei Hälften zerlegt, diese rekursiv sortiert und dann wieder zusammenmischt.

[1] für endliche Listen

```
---Sortieren von Listen von Zahlen

--  einfuegen
sorteinfuegen xs = sorteinfuegenr xs []
sorteinfuegenr [] ys = ys
sorteinfuegenr (x:xs) [] = sorteinfuegenr xs [x]
sorteinfuegenr (x:xs) ys = sorteinfuegenr xs (sorteinfuegenr1 x ys)

sorteinfuegenr1 x [] = [x]
sorteinfuegenr1 x (y:ys) =
         if x <= y then x:y:ys
         else y : (sorteinfuegenr1 x ys)

-- Blasensort  (bubblesort)
bubblesort [] = []
bubblesort [x] = [x]
bubblesort xs =
      let y:resty = bubblesort1 xs
       in  y: (bubblesort resty)
bubblesort1 [x] = [x]
bubblesort1 (x:rest) =
      let (y:resty) = bubblesort1 rest
       in if x  > y then y:x: resty
          else    (x: y:resty)

-- Blasensort  mit Optimierung
bubblesorto [] = []
bubblesorto [x] = [x]
bubblesorto xs =
      let (aenderung,y:resty) = bubblesorto1 xs
       in  if aenderung then y: (bubblesorto resty)
           else xs
bubblesorto1 [x] = (False,[x])
bubblesorto1 (x:rest) =
      let (aenderung, y:resty) = bubblesorto1 rest
       in if x  > y then (True,y:x: resty)
          else    (aenderung,x: y:resty)

-- quicksort
quicks [] = []
quicks [x] = [x]
quicks [x,y] = if x <= y then [x,y] else [y,x]
quicks (x:xs) = let (llt,lge) = splitlist x xs
```

```
                  in (quicks llt)  ++ (x: (quicks lge))

splitlist x [] = ([],[])
splitlist  x (y:ys) =
        let (llt,lge) = splitlist x ys
          in if y < x
             then (y: llt,lge)
             else (llt,y:lge)

--   Versuch einer Optimierung durch variierte Wahl des Pivots.
quicks2 [] = []
quicks2 [x] = [x]
quicks2 [x,y] = if x <= y then [x,y] else [y,x]
quicks2 (x:xs@(y:z:rest)) =
      if x >= y && x <= z || x >= z && x <= y
      then let (llt,lge) = splitlist x xs
                  in (quicks2 llt)  ++ (x: (quicks2 lge))
      else quicks2 (z:x:y:rest)

---  merge-sort

mischsort xs = mergesort (length xs) xs
mergesort _ [] = []
mergesort _ [x] = [x]
mergesort _ [x,y] = if x <= y then [x,y] else [y,x]
mergesort len xs =
       let lenh = len `div` 2
         in mische (mergesort lenh (take lenh xs))
                  (mergesort (len -lenh) (drop lenh xs))

mische [] ys = ys
mische xs [] = xs
mische (x:xs) (y:ys) =
          if x <= y then  x: (mische xs (y:ys))
          else y: (mische (x:xs) ys)
```

Sortieren von $n$ Zahlen mittels Vergleich von Zahlen kann nicht schneller als $n * log(n)$ gemacht werden:

**Begründung**. Eine Menge von $n$ verschiedenen Zahlen hat $n!$ verschiedene Permutationen. Jeder Vergleich kann die Menge der verbleibenden Permutationen höchstens halbieren. Deshalb gilt für die Menge der erforderlichen Vergleiche $v(n)$ in Abhängigkeit von der Anzahl $n$ der Elemente im schlechtesten Fall: $n! < 2^{v(n)}$. Offenbar gilt

$n! \geq \underbrace{1 * \ldots * 1}_{n/2} * \underbrace{n/2 * \ldots * n/2}_{n/2} = (n/2)^{(n/2)}$. Anwenden des Logarithmus ergibt $v > log((n/2)^{(n/2)}) = 0.5 * n * log(n/2) = 0.5 * n * (log(n) - 1)$. D.h. $v(n) = \Omega(n * log(n))$. Eine genauere Abschätzung einer unteren Schranke der minimal notwendigen Anzahl der Vergleiche ist möglich mittels der Stirlingformel für $n!$.

Allerdings konnte man noch nicht nachweisen, daß dies eine untere Schranke ist, wenn man alle Algorithmen zuläßt.

Bemerkungen zu den Eigenschaften der Sortierverfahren

**Sortieren durch Einfügen** Ist ein einfaches Verfahren, das für kleinere Listen schnell genug ist. Die Anzahl Reduktionen (Zeitbedarf) ist im schlechtesten Fall quadratisch: man benötigt für eine Liste der Länge $n$ höchstens: $1 + 2 + \ldots + (n - 1)$ Vergleiche (und Operationen). Dies sind $(n - 1) * n/2$, d.h. $O(n^2)$.

**Sortieren mit Blasensort** Ebenfalls quadratisch im schlechtesten Fall: $(n - 1) + (n - 2) + \ldots + 1$ Operationen. Der beste Fall tritt ein, wenn die Liste schon sortiert ist, oder schon fast sortiert ist, da dann abgebrochen werden kann.

**Quicksort** Dieses Verfahren hat Vorteile, wenn die Listen groß sind und die Werte zufällig verteilt sind. Quicksort ist im schlechtesten Fall quadratisch. Im Mittel ist der Aufwand $n * log(n)$. Beim Experimentieren in Haskell mit zufällig erzeugten Listen erscheint es als das beste Verfahren.

**Mischsort** Ist im schlechtesten Fall $n * log(n)$. Beim Experimentieren in Haskell ist es nur dann besser als Quicksort, wenn Teile der Listen bereits sortiert sind. In prozeduralen Sprachen hat es den Nachteil, daß man die Listen jeweils neu aufbauen muß.

Wir zeigen, daß der Mischsort einen Zeitbedarf von $O(n * \texttt{log}(n))$ hat: Dazu müssen wir $red_{ms}$ abschätzen. $red_{ms}(n) = c + red_{mische}(n) + 2 * red_{ms}(n/2) = c + n + 2 * red_{ms}(n/2)$ $\leq c + n + 2 * (c + n/2 + 1) + 4 * red_{ms}(n/4)$. Da dies $\texttt{log}(n)$ Schritte erfordert, ergibt sich als Abschätzung:

$c * n + n * \texttt{log}(n) + 2 * \texttt{log}(n)$.

Der Term in der Mitte ist asymptotisch am größten, d.h. der Mischsort ist $O(n * \texttt{log}(n))$.

*Übungsaufgabe 5.*

1. Was ist der best-case für den Einfügesort?
2. Für welche Eingabelisten ist der Quicksort quadratisch?
3. Wie ist der Aufwand im besten Fall für den Blasensort?

## 3.5 Abstraktionen mit Funktionen höherer Ordnung

Funktionen nennt man *von höherer Ordnung*, wenn diese selbst als Eingabe oder Ausgabe eine Funktion haben, oder selbst Funktionen wie Daten manipulieren.

*Beispiel 8.* Summe der Zahlen einer Folge $a_1, \ldots, a_n$ Wir nehmen an, daß die Folge in einer Liste dargestellt ist:

```
sum [] = 0
sum (x:xs) = x + (sum xs)
```

Die Summe der Kubikzahlen von $a$ bis $b$ kann dargestellt werden als:

```
kubik_summe a b = sum (map (\x->x^3) [a,a+1..b])
```

Der Ausdruck $[a, a+1..b]$ ist eine Liste, die mit $a$ startet, die Schrittweite 1 hat, und bei $b$ endet.

Damit kann man die mathematische Notation
$\sum\limits_{n=a}^{b} f(n)$ einer Summe als Funktion definieren:

```
summe_folge f a b = sum (map f [a,a+1..b])
```

*Beispiel 9.* Die Summendarstellung von $\frac{\pi}{8}$ als $\frac{1}{1*3} + \frac{1}{5*7} + \frac{1}{9*11} + \ldots$ ist mit dieser Methode leicht in eine Berechnung umzusetzen:

```
pisumme n = summe_folge pisumme_element 0 n
pisumme_element n = let m = 4*n+1 in 1.0 / fromInteger (m *(m+2))
```

## 3.6 Datenabstraktion, abstrakte Datentypen

Datenabstraktion ist eine Strukturierungsmethode, die unabhängig von einer Programmiersprache verwendet werden kann, um wiederverwendbare Teile und Verfahren zu entwerfen. Wichtiges Prinzip ist die Trennung von interner Implementierung der Daten, der Zugriffsfunktionen von deren Verwendung.

Wesentlich ist:

- Die Daten sind über klar definierte Zugriffsfunktionen verfügbar

- Die Algorithmen verwenden nur diese Zugriffsfunktionen
- Die Daten haben eine Semantik, die von der Implementierung respektiert wird.
- Die Implementierung der Daten und Zugriffe ist nicht sichtbar (Kapselung, information hiding).

Hierzu wird normalerweise ein sogenannter Modul verwendet, der i.a. einen bestimmten Datentyp zusammen mit den Zugriffsfunktionen zusammenfaßt und implementiert, aber nur ausgewählte Zugriffsfunktionen nach außen hin sichtbar macht (exportiert). In Haskell ist es auch erlaubt, interne Funktionen zu verwenden, wenn diese explizit mit dem Modulnamen qualifiziert sind (d.h. als Präfix den Modulnamen und einen "." vorgestellt bekommen). Werden auch interne Datenkonstruktoren extern verwendet, so ist die Implementierung nicht mehr verborgen. Dies kann man dadurch erkennen, daß man den Modul nicht mehr durch einen beliebigen Modul ersetzen kann, der die richtige Schnittstellenbedingungen erfüllt. Das kann zwar kurzfristig die Effizienz steigern oder bequemer beim Programmieren sein als der saubere Weg, aber kann langfristig in Projekten zu Problemen führen, wenn die interne Implementierung des Moduls geändert wird.

### 3.6.1 Axiomatische Semantik

Einen Datentyp kann man genau festlegen, indem man Axiome angibt, die die Konstruktoren und Selektoren erfüllen sollen. Verschiedene Implementierungen dieser Konstruktoren und Selektoren, die genau diese Axiome erfüllen, sind dann gleichwertig und können ausgetauscht werden. Im allgemeinen sind dies Axiome über Gleichheit und Ungleichheit von Ausdrücken.

Da wir eine getypte Programmiersprache betrachten, sind die Variablen über die quantifiziert wird, getypt. In Gleichungen sollten die rechten und linken Seiten gleichen Typ haben.

Hierzu nehmen wir als einfaches Beispiel die Listen. Wir nehmen an, daß Listen durch die Konstruktoren `cons` und `nil` und durch die Selektoren `head` und `tail` implementiert sind.

Folgende Gleichheitsaxiome müssen erfüllt sein:

$$
\begin{array}{l}
\forall\ a \\
\text{Die Elemente sind nur aus } a \text{, cons und nil aufgebaut} \\
\quad \begin{array}{lrcl}
\forall s :: a, t :: \text{List } a : & \text{head(cons } s\ t) & = & s \\
\forall s :: a, t :: \text{List } a : & \text{tail(cons } s\ t) & = & t \\
\forall s :: a, t :: \text{List } a : & \text{nil} & \neq & (\text{cons } s\ t) \\
 & \text{head nil} & = & \perp \text{ (undefiniert)}
\end{array}
\end{array}
$$

Diese Semantik gilt für unsere Listen:

```
head (x:y) = x
tail  (x:y) = y
```

```
cons x y = x:y
nil = []
```

Zunächst gilt für alle $s, t$: $[] \neq s : t$.

Zum Nachweis der Axiome erinnern wir uns, daß die Reduktionsregeln zwar die syntaktische Darstellung eines Ausdrucks ändern, aber nicht dessen Wert.

`head (cons s t )` reduziert zu `s`. `tail (cons s t)` reduziert zu `t`.

Wenn diese axiomatische Semantik erfüllt ist, dann kann man folgendes behaupten:

Wenn eine Implementierung nur die Zugriffsfunktionen `head,tail,cons` verwendet, dann kann man diese durch jede andere Implementierung der Zugriffsfunktionen ersetzen, die diese Axiome erfüllen.

Z.B:

*Beispiel 10.* Damit die Typen gleichen Namen haben, nehmen wir der Einfachheit halber an, es gäbe den Typ List noch nicht.

```
data List a = Exnil   | Excons a (List a) Int | Exddd a a
cons x y =   Excons x y 1
head (Excons x y i) = x
tail (Excons x y i) = y
nil = Exnil
```

Offenbar gelten die Axiome.

*Beispiel 11.* Wir betrachten den Datentyp positive ganze Zahlen. Der läßt sich mit Konstruktoren / Selektoren und einer axiomatischen Semantik folgendermaßen beschreiben: Es gibt die Konstruktoren `null, succ` und den Selektor `pred`.

$$\begin{array}{lrcl} \multicolumn{4}{l}{\text{Die Objekte sind nur aus succ null aufgebaut}} \\ \forall s :: \text{Nat}: & \text{pred}(\text{succ } s) & = & s \\ \forall s :: \text{Nat}: & \text{null} & \neq & (\text{succ } s) \\ & \text{pred}(\text{null}) & = & \bot \end{array}$$

Das sieht nicht sehr ausdrucksstark aus, aber damit kann man alle Funktionen auf natürlichen Zahlen definieren:

```
add x  y = if x == null then y
            else succ (add (pred x) y)
kleinergleich x y =
         if x == null then True
         else if y == null then False
         else kleinergleich (pred x) (pred y)
....
```

Eine mögliche und auch korrekte Implementierung ist:

```
data Nat = N | S Nat
null = N
succ x = S x
pred (S x) = x
```

Eine andere Implementierung kann man aufbauend auf den eingebauten Zahlen und deren Operationen angeben. Wir verwenden dazu die korrekten mathematischen Zahlen des Typs `Integer`.

```
null:: Integer
succ:: Integer-> Integer
pred:: Integer-> Integer
--  Nat sollen die nichtnegativen Zahlen vom Typ Integer sein.
null = 0
succ x = x+1
pred x = if x == 0 then error "pred 0??" else x-1
```

Jetzt kann man auch sehen, daß `null, succ, pred` wie eben definiert auf dem Typ `Integer` eine korrekte Implementierung von Nat ist:

- Betrachte nichtnegatives $s$:

  ```
  pred (succ s)  -->  if succ s == 0 then error "pred 0??" else (succ s)-1
  --> if s+1  == 0 then error "pred 0??" else (succ s)-1
  --> (succ s)-1 --> (s +1) - 1 --> s
  ```

- Offenbar gilt auch `null` $\neq$ `succ s` für nichtnegative Zahlen $s$.

- ```
  pred null  -->     if 0 == 0 then error "pred 0??" else 0-1
  --> error "pred 0??"
  ```

  Dies identifizieren wir mit undefiniert.

Man beachte, daß diese Funktionen auf dem Typ `Int` den Datentyp Nat nicht korrekt implementieren.

Da wir Beweise führen über "Implementierungen" ist es i.a. sinnvoll zwei Implementierungsbegriffe einzuführen:

- **theoretische Implementierung**: "auf Papier": die genau die Reduktionsregeln einhält und für die man keine praktische Implementierung braucht, um Ausdrücke auszuwerten.
  ```

- **praktische Implementierung**: Die in Hugs 1.4 eingegebene. Dies sollte bei der Auswertung die gleichen Ergebnisse liefern wie die Auswertung per Hand mit den Reduktionsregeln. Wenn dies nicht der Fall ist, dann ist die Hugs 1.4-Version selbst fehlerhaft.

Meist brauchen wir dies nicht zu unterscheiden, denn Hugs 1.4 führt genau dieselben Auswertungsschritte aus, die wir auch mit Hand ausgeführt hätten.

Beim Rechnen und Beweisen mit dem Symbol $\perp$ für "undefiniert" ist zu beachten, daß das Symbol in Ausdrücken vorkommen kann, ohne daß der ganze Ausdruck undefiniert ist. Das kommt daher, daß$\perp$ eher als nicht-terminierend zu interpretieren ist, denn als undefiniert. Z.B. ist in der Definition von `pred` ja immer ein Zweig "undefiniert", ohne daß `pred` selbst undefiniert ist.

*Übungsaufgabe 6.* Zeige, daß die folgende Implementierung von Nat korrekt ist:

```
null = []
succ x = 5:x
pred (_:x) = x
```

Der Datentyp $GZ$ hat die Konstruktoren `null, succ, neg`, den Selektor `pred`. Um die axiomatische Semantik anzugeben, brauchen wir noch die Meta-Notation: $\texttt{succ}^1(s) := \texttt{succ}\ s$ und $\texttt{succ}^{n+1}(s) := \texttt{succ}^n\ s$, wobei $n$ ein Zahl ist. Axiomatische Semantik von $GZ$:

$$
\begin{array}{lrcl}
\text{Die Objekte sind nur aus null, succ, neg aufgebaut} & & & \\
\forall s :: \mathrm{GZ}: & \mathrm{pred}(\mathrm{succ}\ s) & = & s \\
\forall s :: \mathrm{GZ}: & \mathrm{pred}(\mathrm{neg}\ s) & = & \mathrm{neg}(\mathrm{succ}\ s) \\
 & \mathrm{pred\ null} & = & \mathrm{neg\ (succ\ null)} \\
 & \mathrm{neg\ null} & = & \mathrm{null} \\
\forall s :: \mathrm{GZ}: & \mathrm{succ\ (neg\ (succ}\ s)) & = & \mathrm{neg}\ s \\
\forall s :: \mathrm{GZ}: & \mathrm{neg}(\mathrm{neg}\ s) & = & s \\
\forall s :: \mathrm{GZ}, n :: \text{positive Zahl}: & s & \neq & (\mathrm{succ}^n\ s)
\end{array}
$$

Eine mögliche und auch korrekte Implementierung ist:

```
--  Die Implementierung soll nur folgende Datenobjekte konstruieren:
--   N, S N, S ... (S N) und davon Neg ...
--   aber kein Neg N.
data GZ = N | S GZ | Neg GZ
null = N
succ N = S N
succ (S x) = S (S x)
succ (Neg (S x)) = neg x
neg (Neg x) = x
```

```
neg N = N
neg (S x) = Neg (S x)

pred (S x) = x
pred N = Neg (S N)
pred (Neg x) = Neg (S x)
```

Eine andere Implementierung ist:

```
null = 0
succ x = x+1
pred x =   x-1
neg x = - x
```

*Übungsaufgabe 7.* Gebe die Implementierung von `GZ` als Hugs-Datentyp `Integer` an. Zeige, daß `Int` mit der gleichen Implementierung die axiomatische Semantik von `GZ` nicht erfüllt.

Gebe die tatsächliche axiomatische Semantik des implementierten Datentyps `Int` von Hugs 1.4 an, indem Sie ein Axiom von `GZ` ersetzen. Verwenden Sie dabei entsprechende Abkürzungen für Zahlen.

### 3.6.2 Einige Datentypen

Beispiele hatten wir schon besprochen:

*Beispiel 12.* Rationale Zahlen: Es gibt Funktionen zum Erzeugen und Anzeigen, arithmetische Operationen: $*, /, +, -$. Als axiomatische Semantik kann man die mathematischen Eigenschaften für die arithmetischen Operationen fordern, und daß beim Drucken stets die gekürzte Darstellung erscheint. In Haskell gibt es sowohl ganzzahlige Brüche von kurzen ganzen Zahlen als auch von beliebig langen ganzen Zahlen. Die mathematisch adäquate axiomatische Semantik kann man nur von den aus beliebig langen Zahlen gebildeten rationalen fordern. Die Brüche aus kurzen ganzen Zahlen erfordern eine andere axiomatische Semantik (siehe auch Aufgabe 7).

Die Implementierungsdetails sollten unsichtbar sein. Man könnte rationale Zahlen wie 3/4 sowohl als Paar $(3, 4)$, oder auch als periodischen Dezimalbruch implementieren, der dann seinerseits als Paar dargestellt ist. Ebenso ist die Implementierung frei in der Wahl, ob und wann (intern) gekürzt wird.

*Beispiel 13.* Fließkommazahlen (Gleitkommazahlen): Es gibt Funktionen wie Erzeugen und Anzeigen, mathematische Operationen: $*, /, +, -$. Hier ist die Semantik nicht die mathematische für reelle Zahlen, da sich irrationale Zahlen i.a. nicht beliebig genau darstellen lassen. D.h. man muß Näherungen einführen. Die Semantik der implementierten arithmetischen Operationen garantiert dann, daß diese die richtigen Werte der Muliplikation möglichst gut annähern. Damit dies auf allen Rechnern eindeutig ist, gibt es Normierungen für die internen Berechnungen, je nach Genauigkeit.

Hier sieht man auch, daß man diese abstrakten Datentypen nicht völlig unabhängig voneinander definieren kann: man braucht oft sogenannte Datenkonversionen, die Datenobjekte von einem abstrakten Datentyp in einen anderen umwandeln (und wieder zurück). Z.B. ganze Zahlen in rationale Zahlen und umgekehrt.

*Beispiel 14.* Gleitkommazahlen in Intervallarithmetik: Es gibt Funktionen wie Erzeugen, Anzeigen, mathematische Operationen: $*, /, +, -$. Hier ist die axiomatische Semantik auch der Operationen mathematisch erfassbar: Die Ergebnisse müssen immer in dem angegebenen Intervall liegen. Allerdings gelten dann nicht mehr alle arithmetischen Gesetze. Division durch ein Intervall, das die Null enthält muß verboten sein (bzw, einen Fehler ergeben.)

*Beispiel 15.* Der abstrakte Datentyp (Modul) "Menge" (endliche Mengen mit Elementen von gleichem Typ) sollte folgende Funktionalität bereitstellen:

- Erzeugen von Mengen, gegeben die Elemente
- Test auf Enthaltensein eines Elementes in einer Menge
- Drucken einer Menge
- Kardinalität einer Menge
- Bildung von Schnitt, Vereinigung, Differenz, Potenzmenge
- Test auf Gleichheit von Mengen

*Beispiel 16.* Der abstrakte Datentyp (Modul) "Multimenge": endliche Multimengen mit Elementen von gleichem Typ,wobei Elemente mehrfach vorkommen dürfen. Die Funktionalitäten sind wie bei Mengen. Die Implementierung von Multimengen ist besonders einfach auf der Basis von Listen.

Als Beispiel nehmen wir `anzahl, einfuegen, cons, null`. Der Datentyp wäre *Multimenge a* mit den Konstruktoren `cons, null`. Zwei exemplarische Axiome sind dann:

`anzahl` $null = 0$

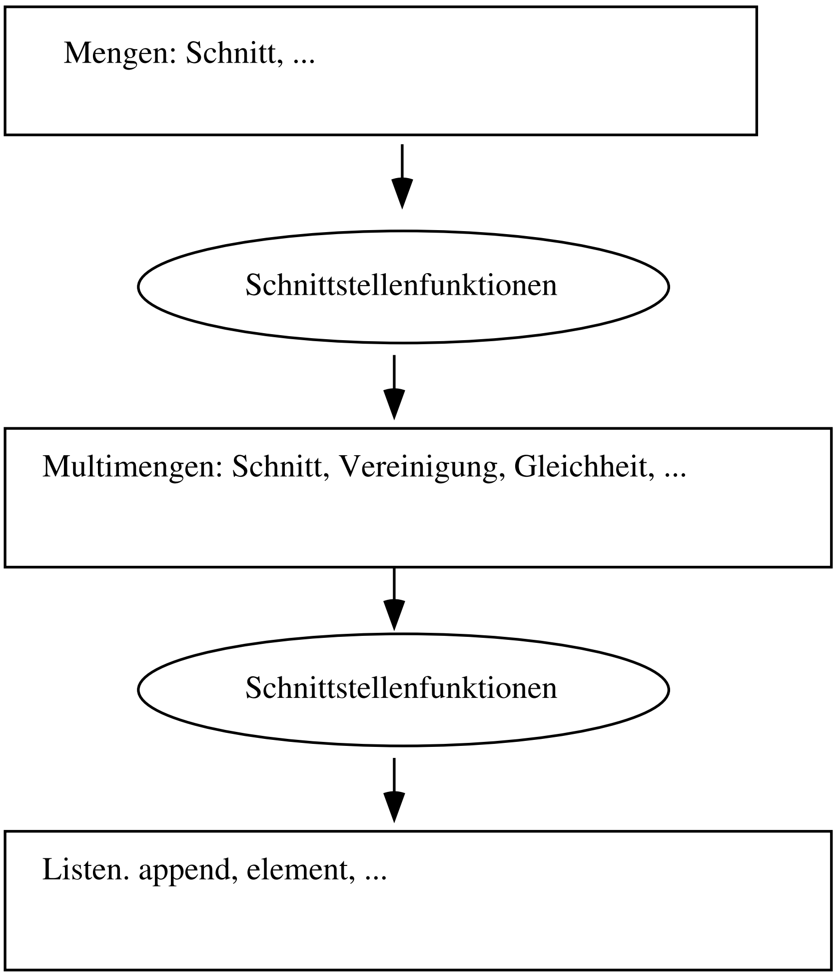
$\forall s :: a, t :: Multimenge\ a :$ `anzahl(einfuegen` $s\ t$`)` $=$ (`card` $s$) $+ 1$

Die Listenimplementierung mit `anzahl` = `length`, `einfuegen` = `cons` ist dann insoweit korrekt, denn:
`anzahl null` $\rightarrow$ `length` $[] \rightarrow 0$
`anzahl` (`einfuegen` s t) $\rightarrow$ `length` (`cons` s t) $\rightarrow$ (`length` $t$) $+ 1$.

Eine Implementierungsmöglichkeit für Mengen sind Listen von Elementen. Da man Schnitte von Mengen und die Kardinalität von Mengen ausrechnen will, muß man Gleichheit von Elementen testen können. Eine weitere Möglichkeit ist die Implementierung auf der Basis von Multimengen.

Ein Datentyp "unendliche Liste" oder "unendliche Menge" ist möglich, allerdings auch problematisch, denn nicht alle unendlichen Mengen sind endlich darstellbar. Ein Effekt ist, daß man den Elementtest dann nicht korrekt implementieren kann.

Abstraktionsbarrieren für die Beispiele Mengen / Multimengen / Listen sind:




Abstrakte Datentypen werden oft als *Modul* implementiert, wobei die erlaubten Funktionen in einer Export-Liste definiert werden. Oft wird auch für benutzte Module eine eigene Importliste von Funktionen angegeben.

Vorteile:

- Programme können leicht geändert und gewartet werden
- Die Implementierung der Zwischenschichten kann frei gewählt werden, und auch wieder völlig neu erstellt werden, ohne daß die benutzenden Funktionen geändert werden müssen.
- Die Überprüfung der Korrektheit von Implementierungen ist modularer und somit viel einfacher.

Problem: Der Ressourcenbedarf hängt ab von der Implementierung der Zwischenschichten und oft auch von dem Zusammenspiel der verschiedenen Zwischenschichten.

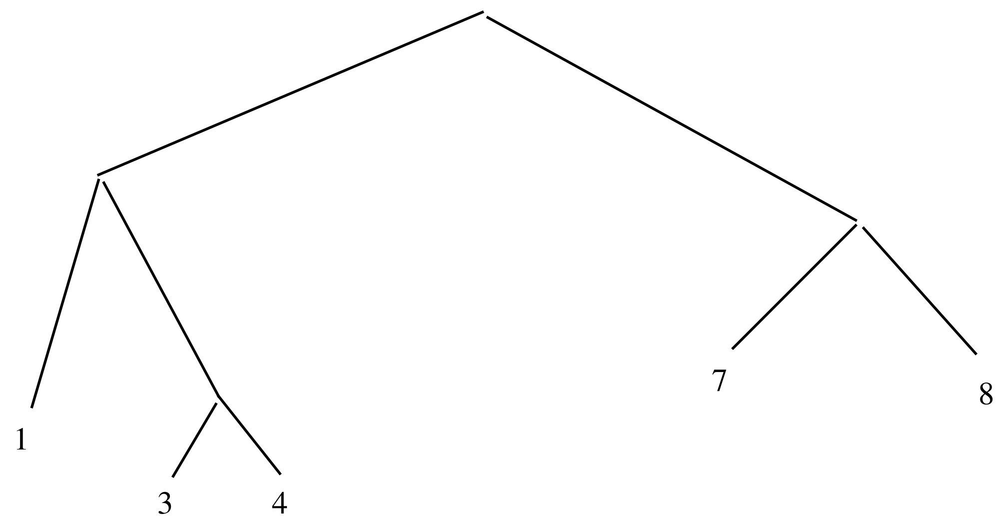
## 3.7 Bäume

Wir stellen binäre, geordnete Bäume in folgender Datenstruktur dar:

```
data Binbaum a = Bblatt  a | Bknoten (Binbaum a) (Binbaum a)

instance Functor Binbaum
    where map = b_map
```

*Beispiel 17.* Der folgende binäre Baum




hat eine Darstellung als

```
Bknoten (Bknoten (Bblatt 1)
                 (Bknoten (Bblatt 3) (Bblatt 4)))
        (Bknoten (Bblatt 7) (Bblatt 8))
```

Einige Verarbeitungsfunktionen sind:

```
--  Liste der Markierungen der Blaetter
b_rand (Bblatt x) = [x]
b_rand (Bknoten bl br) =
                  (b_rand bl)  ++ (b_rand br)

-- testet, ob Element im Baum ist
b_in x (Bblatt y) = (x == y)
b_in x (Bknoten bl br) = b_in x bl  ||  b_in x br

-- wendet eine Funktion auf alle Elemente des Baumes an,
--               Resultat: Baum der Resultate
b_map f (Bblatt x) = Bblatt (f x)
b_map f (Bknoten bl br) =
                  Bknoten (b_map f bl)  (b_map f br)

---   Groesse des Baumes:
b_size (Bblatt x) = 1
b_size (Bknoten bl br) =
                  (b_size bl) + (b_size br)

---   Summe aller Blaetter, falls Zahlen:
b_sum (Bblatt x) = x
b_sum (Bknoten bl br) =
                  (b_sum bl) + (b_sum br)


--  Erzeugung grosse Baeume
b_mkbt_test 0 k = Bblatt k
b_mkbt_test n k = Bknoten (b_mkbt_test (n-1) (k+1)) (b_mkbt_test (n-1) (2*k))

b_mkbt_testll [x] = Bblatt x
b_mkbt_testll (x:xs) = Bknoten (Bblatt x) (b_mkbt_testll xs)

b_mkbt_testlr [x] = Bblatt x
b_mkbt_testlr (x:xs) = Bknoten (b_mkbt_testlr xs) (Bblatt x)

--  schnelles fold "uber bin"are B"aume
foldbt :: (a -> b -> b) -> b -> Binbaum a -> b
foldbt op a (Bblatt x) = op x a
foldbt op a (Bknoten x y) = (foldbt op (foldbt op a y) x)
```

```
--- effizientere Version von b_rand

b_rand_eff = foldbt (:) []
```

Um zu begründen, warum `foldbt` relativ schnell ist, betrachte den Zwischenausdruck, der aus einem Baum `tr` mit der Struktur $(((1,2),3),(4,5))$ entsteht, wenn man `foldbt (+) 0 tr`auswertet.:

```
foldbt (+) 0 (((1,2),3),(4 ,5))
-->  foldbt (+) (foldbt (+) 0 (4 ,5))    ((1,2),3))
-->  foldbt (+) (foldbt (+) (foldbt (+) 0 (5)) (4)    ((1,2),3))
-->  foldbt (+) (foldbt (+) (5+0) (4)    ((1,2),3))
-->  foldbt (+) (4+ (5+0))   ((1,2),3))
-->  foldbt (+) (foldbt (+) (4+ (5+0))   (3))  (1,2))
-->  foldbt (+) (3+ (4+ (5+0)))   (1,2))
-->  foldbt (+) (foldbt (+) (3+ (4+ (5+0))) (2)  (1))
-->  foldbt (+)  (2+ (3+ (4+ (5+0))))  (1))
-->  1+ (2+ (3+ (4+ (5+0))))
```

D.h. Wenn ein binärer Baum $tr$ mit Randliste $[a_1, \ldots, a_n]$ gegeben ist, dann entspricht `foldbt f a tr` dem Ausdruck $f\ a_1\ (f\ a_2\ (\ldots (f\ a_n\ a) \ldots))$. D.h. es entspricht einem `foldr`, das für die `concat` Funktion schneller ist.

Man kann weitere Funktionen auf Bäumen definieren und auch etwas allgemeinere Bäume als Datenstruktur verwenden.

*Übungsaufgabe 8.* Benutze folgende Datenstruktur für Bäume mit mehreren Töchtern pro Knoten:

```
data Nbaum a = Nblatt a | Nknoten [Nbaum a]
```

1. Schreibe hierfür eine Funktion `bn_rand` und `bn_map` analog zu `b_rand` und `b_map`.
2. Schreibe eine Funktion `nb_generate xs n`, die eine Liste `xs` als Argument hat und die Elemente ungefähr gleichmäßig auf die Blätter verteilt, so daß jeder Knoten höchstens $n$ Töchter hat und die Tiefe nur logarithmisch mit der Anzahl der Elemente zunimmt.
3. Teste diese Funktion in sinnvoller Weise.

## 3.8 Felder, arrays

*Felder (arrays, Vektoren)* sind eine weitverbreitete Datenstruktur in Programmiersprachen, die normalerweise eine Folge $a_i, i = 0, \ldots, n$ modellieren.

Als Semantik eines Feldes $A$ mit den Grenzen $(a, b)$ und Elementen des Typs $\alpha$ kann man eine Funktion nehmen: $f_A : [a, b] \to \alpha$.

Die Implementierung als Liste ist möglich, hat aber als Nachteil, daß der Zugriff auf ein bestimmtes Element, wenn der Index bekannt ist, die Größe $O(laenge(liste))$ hat. Die interne Implementierung eines Feldes macht es möglich, eine (praktisch) konstante Zugriffszeit zu realisieren, wenn der Index bekannt ist. Allerdings ist die Handhabung eines Feldes im Programm etwas aufwendiger, da man bei Zugriffen stets den Index berechnen muß und da man im allgemeinen ein Feld nicht so ohne weiteres verkleinern oder erweitern kann.

In Haskell gibt es als eingebaute Datenstruktur ebenfalls arrays. Dies sind Felder von Elementen des gleichen Typs. Als Spezialität kann man beim Erzeugen verschiedene Typen des Index wählen, so daß nicht nur Zahlen, sondern auch Tupel von ganzen Zahlen möglich sind, womit man (mehrdimensionale) Matrizen modellieren kann. Einige Zugriffsfunktionen sind:

- `array x y` : erzeugt ein Feld (array) mit den Grenzen $x = (start, ende)$, initialisiert anhand der Liste $y$, die eine Liste von Index-Wert Paaren (eine Assoziationsliste) sein soll
- `listArray x y`: erzeugt ein array mit den Grenzen $x = (start, ende)$, initialisiert sequentiell anhand der Liste $y$.
- (!): Infix funktion: $ar!i$ ergibt das $i$-te Element des Feldes $ar$.
- (//): Infix-funktion $a//xs$ ergibt ein neues Feld, bei dem die Elemente entsprechend der Assoziationsliste $xs$ abgeändert sind.
- `bounds`: Erlaubt es, die Indexgrenzen des Feldes zu ermitteln.

*Beispiel 18.*
```
umdrehen_array  ar  =
    let (n,m) = bounds ar
        mplusn = m+n
      in ar // [(i,ar!(mplusn -i))   | i <- [n..m] ]
```

## 3.9 Suche und Zugriff

Wir betrachten das in indizierten Dateien und Datenbanken vorkommende Problem, eine Menge von gleichartigen Datenobjekten zu verwalten, Z.B. alle DiplomstudententInnen dieses

Jahrgangs, oder ein Lexikon von Worten mit zugehöriger Übersetzung. Wir wollen dieses Problem zunächst vereinfacht betrachten.

Die Objekte (Sätze) seien dargestellt als Paare, wobei das erste Argument eine Zahl ist (Integer), der sogenannte Schlüssel (key), und das zweite Argument der Inhalt. Die Datenstruktur, die dafür benötigt wird, entspricht in etwa einer Menge mit markierten Elementen: der Schlüssel bezeichnet das Element, während die Markierung den zugehörigen Daten entspricht.

Operationen: erzeugen, einfügen, suchen, drucken, entfernen.

```
data Maybe a = Nothing | Just a
data Satz a = Satz Integer  a
data Datenbank a = ???
intialisiereDb ::  Datenbank a
einfuegeDb :: Satz a -> Datenbank a -> Datenbank a
istinDb :: Integer -> Datenbank a -> Bool
sucheDb :: Integer -> Datenbank a -> Maybe a
druckeDb :: Datenbank a -> [(Integer,a)]
entferneDb :: Integer -> Datenbank a -> Datenbank a
```

Unter Benutzung dieser Operationen könnten wir jetzt spezifizieren, was wir von dieser Datenbank an Verhalten erwarten: Wir formulieren umgangssprachlich:

- Eine neu intialisierte Datenbank ist leer
- Ein eingefügter Satz ist nach dem Einfügen auch zu finden (mit gleichem Inhalt)
- Die Datenbank enthält genau die eingefügten Sätze.
- Nach dem Entfernen eines Satzes ist dieser nicht mehr zu finden.
- Ein Satz läßt sich nur einmal einfügen.

Eine weitergehende Funktionalität ist eine Anfrageschnittstelle, die logisch verknüpfte Anfragen beantworten kann. Als mögliche Antworten beschränken wir uns hier auf Mengen von Sätzen. Ruft man die Anfragen modular auf, so braucht man auf unterer Ebene eine schnelle Möglichkeit, Mengen von Sätzen zu schneiden, zu vereinigen, oder zu komplementieren bzw. Differenzen von (markierten) Mengen zu bilden

Wir betrachten verschiedene Möglichkeiten der internen Darstellung:
**Assoziationsliste**:

Wir beschreiben nur einige der notwendigen Funktionen.

```
data Satz a = Satz Int  a
```

```
data Db a = Dbl [Satz a]

intialisiereDb x = Dbl []
einfuegeDb (Satz x sx) (Dbl db) = Dbl ((Satz x sx): db)
druckeDb (Dbl db) = map (\(Satz x y) -> (x,y)) db
istinDb (Satz x y) (Dbl []) = False
istinDb (Satz x y) (Dbl ((Satz a _): restDb)) =
                                (x == a) || istinDb (Satz x y) (Dbl restDb)
sucheDb (Satz x y) (Dbl []) = Nothing
sucheDb (Satz x y) (Dbl ((Satz k sk): restDb)) =
              if x == k then  Just sk
              else sucheDb (Satz x y)  (Dbl restDb)
entferneDb (Satz x _) (Dbl []) = (Dbl [])
entferneDb (Satz k ky) (Dbl ((Satz x y) : restDb)) =
               if k == x
               then entferneDb (Satz k ky) (Dbl restDb)
               else let Dbl restDblist =  entferneDb (Satz k ky) (Dbl restDb)
                      in Dbl ((Satz x y) : restDblist)
```

Eine Schwäche dieser Implementierung ist die Laufzeit $O(n)$ für die Suche (`istinDb`) eines Elementes, was bei einer kleineren Menge von Sätzen akzeptabel ist.

*Übungsaufgabe 9.* Beschreibe die Stärken, Schwächen und (den) Fehler? dieser Implementierung und bestimme die Größenordnung des Ressourcenbedarf für einzelne Funktionen. Korrigiere die Implementierung.

**sortierte Liste** Betrachte jetzt eine aufsteigend sortierte Liste, sortiert nach Zahlen:

```
---   sortierte Liste

einfuege2Db (Satz x sx) (Dbl ((Satz a sa): restDb)) =
            if x < a then  Dbl ((Satz x sx) : ((Satz a sa): restDb))
            else if x == a then error "bereits vorhanden"
            else let Dbl restDblist = einfuege2Db (Satz x sx) (Dbl restDb)
                   in Dbl ((Satz a sa): restDblist)
istin2Db (Satz x y) (Dbl ((Satz a _): restDb)) =
              if x < a then False else
              if (x == a) then True
              else istin2Db (Satz x y) (Dbl restDb)
```

Diese Methode hat bereits erhebliche Vorteile bei der Bildung von Vereinigungen, Schnitten und Differenzen. Die folgenden Funktionen sind die Schnittbildung, Vereinigung und

Differenz auf sortierten Sätzen.

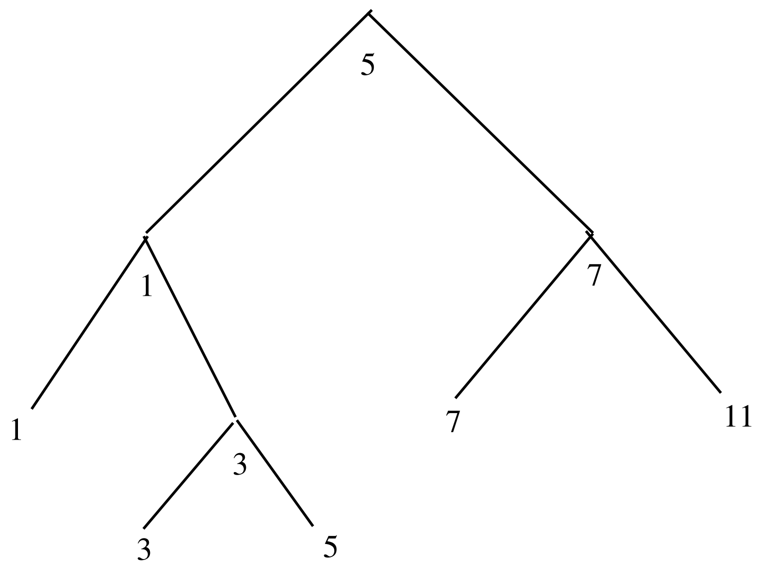
```
intersect2Dbtop (Dbl xs) (Dbl ys) = Dbl (intersect2Db   xs ys)
intersect2Db   [] ys = []
intersect2Db   xs []   = []
intersect2Db   ((Satz k1 d1): xs1)  ((Satz k2 d2): xs2)   =
         if k1 == k2
         then (Satz k1 d1): (intersect2Db xs1 xs2)
         else if k1 < k2
                then  intersect2Db   xs1  ((Satz k2 d2): xs2)
                else  intersect2Db   ((Satz k1 d1): xs1)   xs2

union2Dbtop (Dbl xs) (Dbl ys) = Dbl (union2Db   xs ys)
union2Db   [] ys = ys
union2Db   xs []   = xs
union2Db   ((Satz k1 d1): xs1)  ((Satz k2 d2): xs2)   =
         if k1 == k2
         then (Satz k1 d1): (union2Db xs1 xs2)
          else if k1 < k2
                then  (Satz k1 d1): union2Db   xs1  ((Satz k2 d2): xs2)
                else  (Satz k2 d2): union2Db   ((Satz k1 d1): xs1)   xs2

--   xs  ohne ys
diff2Dbtop (Dbl xs) (Dbl ys) = Dbl (diff2Db   xs ys)
diff2Db   [] ys = []
diff2Db   xs []   = xs
diff2Db   ((Satz k1 d1): xs1)  ((Satz k2 d2): xs2)   =
         if k1 == k2
         then (diff2Db xs1 xs2)
          else if k1 < k2
                then  (Satz k1 d1): diff2Db   xs1  ((Satz k2 d2): xs2)
                else   diff2Db   ((Satz k1 d1): xs1)   xs2
```

Man kann leicht nachprüfen, daß diese Funktionen einen Zeitbedarf haben, der linear in der Anzahl der Summe der Elemente der beteiligten Listen ist.

**Suchbaum** Diese Implementierung ist komplizierter, allerdings gibt es Vorteile gegenüber der Assoziationsliste beim Zugriff und Auffinden der schon eingetragenen Sätze. Jeder Knoten enthält als Markierung den größten Schlüssel des Teilbaumes mit den kleineren Werten.




```
---   Suchbaum


data Suchbaum a = Sblatt  Int a | Sknoten (Suchbaum a) Int (Suchbaum a)
                           | Suchbaumleer

data DbS a = Suchbaum (Satz a)
intialisiereDbS x = Suchbaumleer
einfuegeDbS (Satz x sx)  Suchbaumleer = Sblatt x (Satz x sx)
einfuegeDbS (Satz x sx)  (Sblatt  k satzk) =
                    if x < k
                    then Sknoten (Sblatt x (Satz x sx)) x (Sblatt k satzk)
                    else if x == k then error " schon eingetragen"
                    else Sknoten  (Sblatt k satzk) k (Sblatt x (Satz x sx))
einfuegeDbS (Satz x sx)  (Sknoten l k r) =
           if x < k then Sknoten (einfuegeDbS (Satz x sx)  l) k r
           else if x == k then  error " schon eingetragen"
          else Sknoten l k (einfuegeDbS (Satz x sx)  r)


istinDbS (Satz x y) Suchbaumleer = False
istinDbS (Satz x y) (Sblatt k _)   =  (x == k)
istinDbS (Satz x y) (Sknoten  bl k br) =
           if x < k then  istinDbS (Satz x y) bl
           else if x == k then True
           else  istinDbS (Satz x y) br
```

```
erzeugeDbS [] = Suchbaumleer
erzeugeDbS (x:xs) = einfuegeDbS (Satz x x)  (erzeugeDbS xs)
druckeDbS sb = blaetterSB

foldSB :: (a -> b -> b) -> b -> Suchbaum a -> b
foldSB op a Suchbaumleer = a
foldSB op a (Sblatt _ x) = op x a
foldSB op a (Sknoten x _ y) = (foldSB op (foldSB op a y) x)

--- effiziente Version

blaetterSB = foldSB (:) []
```

*Übungsaufgabe 10.* Schreibe weitere Funktionen auf den Suchbäumen

Analyse des Ressourcenbedarfs der verschiedenen Zugriffsfunktionen:

`einfuegeDb` hat linearen Ressourcenbedarf abhängig von der Tiefe des Suchbaumes. D.h. wenn der Suchbaum einigermaßen balanciert aufgebaut ist, dann ist der Ressourcenbedarf $O(log(n))$, wobei $n$ die Anzahl der Einträge in die Datenbank ist.

`istinDb` hat linearen Ressourcenbedarf abhängig von der Tiefe des Suchbaumes. Ist ebenfalls logarithmisch

*Beispiel 19.* Wenn der Suchbaum nicht balanciert ist, dann ist der Zeitbedarf der Enthaltenseinstest von der Größenordnung $O(n)$. Betrachte einen Baum der Form:

```
Sknoten (Sblatt (Satz 1 "eins")  1
       Sknoten (Sblatt (Satz 2 "zwei")  2  ....
```

Dann benötigt die Suche darin offenbar genau soviele Weiterschaltungen wie Blätter in diesem Baum sind. Das Problem ist, daß diese Bäume durchaus auch durch normales Einfügen entstehen können.

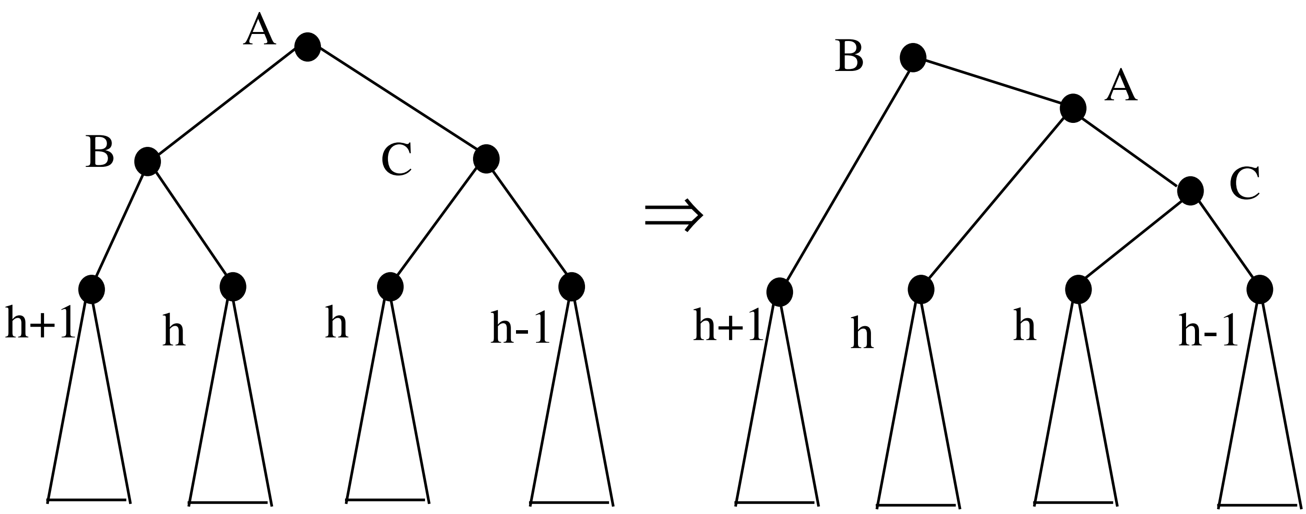
Ein Ausweg sind *balancierte Bäume*, mit der charakterisierenden Eigenschaft, daß die Tiefe des Baumes nicht zu groß wird. Optimal verteilt sind die Blätter, wenn die Tiefe am kleinsten ist. Dies ist allerdings beim Einfügen in beliebiger Reihenfolge nicht so einfach und kann eine völlige Reorganisation des Baumes beim Einfügen eines einzigen Elementes erzwingen.

Eine Datenstruktur mit Operationen, die eine Balance-Bedingung erfüllt, aber nicht optimal balanciert sind, sind *AVL-Bäume* (nach Adelson-Velskii und Landis)

Die Bedingung ist, daß in jedem Knoten die Differenz der Höhe des rechten und linken Teilbaumes höchstens 1 ist.

Man kann zeigen, daß in AVL-Bäumen die Höhe logarithmisch ist, d.h. die Funktion $f(n) = max\{h \mid h \text{ ist die Höhe eines AVL-Baumes mit } n \text{ Blättern}\}$ ist $O(log\ n)$.

Man kann durch Anreichern der Datenstruktur um Balancefaktoren $-1, 0, +1$ den Aufwand für die beim Einfügen und Löschen erforderlichen Rotationen gering halten. Natürlich muß dann auch jede Veränderung für den richtigen Wert dieser Faktoren sorgen.




Die Programmierung in Haskell von leicht abgewandelten AVL-Bäumen sieht folgendermaßen aus:

```
---     AVL-Suchbaum nur fuer Zahlen
---         Ablatt Wert
---         Aknoten linke-Hoehe linker-Baum  rechte-Hoehe rechter-Baum,

data AVLsuchbaum = Ablatt  Int | Aknoten Int AVLsuchbaum Int AVLsuchbaum
                           | Abaumleer

intialisiereAVL x = Abaumleer
--  Resultat:   (Tiefe erh"oht, Baum)
einfuegeAVL x  Abaumleer = Ablatt x
einfuegeAVL x  (Ablatt  k) =
                    if x < k
                    then (Aknoten 1 (Ablatt x) x (Ablatt k))
                    else if x == k then error " schon eingetragen"
```

```
                    else Aknoten  1 (Ablatt k) k (Ablatt x)


einfuegeAVL x  (Aknoten _ l k r)
    | x < k  = let lneu = (einfuegeAVL x l)
                   ldl = adepth lneu
                   ldr = adepth r
                   baumneu = Aknoten (1+ (max ldl ldr)) lneu k r
                        in  if ldl - ldr > 1 then arotate baumneu
                            else baumneu
    | x == k =   error " schon eingetragen"
    | otherwise = let rneu = (einfuegeAVL x r)
                      ldl = adepth l
                      ldr = adepth rneu
                      baumneu = Aknoten (1+ (max ldl ldr)) l k rneu
                        in  if ldr - ldl > 1 then arotate baumneu
                            else baumneu

adepth (Ablatt x) = 0
adepth (Aknoten d _ _ _) = d

arotate (Aknoten d0 (Ablatt s1) k0
                         (Aknoten d01 s01 k01 t01@(Aknoten _ _ _ _))) =
    let dneu1 = (adepth s01) +1
        dneu0 = 1+ (max dneu1 (adepth t01))
        baumneu = (Aknoten dneu0 (Aknoten dneu1 (Ablatt s1) k0 s01)
                              k01 t01)
      in if abs (dneu1 - adepth t01) > 1
          then arotate baumneu else baumneu

arotate (Aknoten d0 (Aknoten d01 s01@(Aknoten _ _ _ _)
                                   k01 t01) k0 (Ablatt s1)) =
    let dneu1 = (adepth t01) +1
        dneu0 = 1+ (max dneu1 (adepth s01))
        baumneu = (Aknoten dneu0 s01 k01
                             (Aknoten dneu1 t01 k0 (Ablatt s1)))
       in if abs(dneu1 - adepth s01) > 1
           then arotate baumneu else baumneu

arotate (Aknoten d0 s0@(Ablatt s1) k0
                 (Aknoten d01 s01@(Aknoten _ s011 k011 s012)
                   k01 t01@(Ablatt t011))) =
     arotate4 s0 k0 s011 k011 s012 k01 t01
```

```
arotate (Aknoten d0 (Aknoten d01 s01@(Ablatt _) k01
                  (Aknoten _ t11 k11 t12)) k0 s11@(Ablatt s1)) =
       arotate4 s01 k01 t11 k11 t12 k0 s11

arotate (Aknoten d0 s1@(Aknoten d01 s01 k01 t01) k0 t1
           @(Aknoten d11 s11 k11 t11)) =
     if d01 > d11 then arotate3 s01 k01 t01 k0 t1
     else arotate3 s1 k0 s11 k11 t11

--  5 faelle: 544, 454 445, 554,455
arotate3 b1 k1 b2 k2 b3 =
   let d1 = adepth b1
       d2 = adepth b2
       d3 = adepth b3
       dd1 = d1-d2
       dd2 = d2 -d3
       (Aknoten _ b21 k21 b22) = b2
     in
      if dd1 == 1 && dd2 == 0  || dd1 == 0 && dd2 == 1
      then Aknoten (1+ (max d1 (1 + (max d2 d3))))
                           b1 k1 (Aknoten (1+ (max d2 d3)) b2 k2 b3)
      else if dd1 == 0 && dd2 == -1 || dd1 == -1 && dd2 == 0
       then Aknoten (1+ (max d3 (1+ (max d1 d2))))
                            (Aknoten (1+ max d1 d2) b1 k1 b2) k2 b3
       else arotate4 b1 k1 b21 k21 b22 k2 b3

--   3 faelle: 4444, 4344, 4434
arotate4 b1 k1 b2 k2 b3 k3 b4 =
      let d1 = adepth b1
      in Aknoten (d1+2) (Aknoten (d1+1) b1 k1 b2) k2
                                   (Aknoten (d1+1) b3 k3 b4)

istinAVL x Abaumleer = False
istinAVL x (Ablatt k)   =  (x == k)
istinAVL x (Aknoten  _ bl k br) =
           if x < k then  istinAVL x bl
           else if x == k then True
           else  istinAVL x br

erzeugeAVL [] = Abaumleer
erzeugeAVL (x:xs) = einfuegeAVL x (erzeugeAVL xs)

foldAVL op a Abaumleer = a
```

```
foldAVL op a (Ablatt x) = op x a
foldAVL op a (Aknoten _ x _ y) = (foldAVL op (foldAVL op a y) x)

blaetterAVL = foldAVL (:) []
druckeAVL sb = blaetterAVL

testvarAVL = (erzeugeAVL ([100,99..50] ++ [1..49]))
testmischeqAVL = (blaetterAVL (erzeugeAVL ([1..30]++[60,59..31]++[61..100])))
                 == mischsort ([1..30]++[60,59..31]++[61..100])
```

### 3.9.1 binäre Suche

Diese Suche hat als Ausgangspunkt eine Feld (oder eine Liste) von sortierten Elementen. Gegeben ein Element, finde den Index. Implementiert man dies als Liste, so ist die Suche von linearer Zeitbedarf, da man die Liste durchlaufen muß, um an die Elemente zu kommen. Dies ist verbesserbar, wenn die Elemente in einem Feld sind: Man kann dann die Suche nach der Methode der Intervallhalbierung durchführen.

```
binaere_suche x ar = let (m,n) = (bounds ar) in binaere_suche_r x ar m n
binaere_suche_r x ar m n =
        if m == n then
           if (ar!m) == x then (True,m)
           else (False, error "binsuche")
        else let mid = (m+n) `div` 2
                in if ar ! mid > x binaere_suche_r x ar m mid
                    else  binaere_suche_r x ar mid n
```

## 3.10 Schnitt und Vereinigung von Mengen

Die interessanten bisher noch fehlenden Operationen der Datenstruktur Menge sind Schnitt, Vereinigung, Differenz und Potenzmenge.

**unsortierte Listen** In diesem Fall ist es zur Berechnung des Schnitts zweier Mengen $A, B$ notwendig, alle Elemente der Liste $A$ mit allen Elementen der Liste zu $B$ zu vergleichen. D.h. die Anzahl der Reduktionen ist $|A| * |B|$, d.h. $O((|A| + |B|)^2)$ bzw. $O(n^2)$, wenn $n$ die Gesamtgröße der Eingabelisten ist.

**sortierte Listen** Die Berechnung des Schnittes oder der Vereinigung erfolgt durch Mischen (Merge). Hierbei kann man sogar ohne viel Extra-Aufwand dafür sorgen, daß das Resultat aufsteigend sortiert ist.

```
merge_schnitt [] xb = []
merge_schnitt (x:xs) [] = []
merge_schnitt (x:xs) (y:ys) =
          if x < y then  merge_schnitt xs (y:ys)
          else if x == y then  x : merge_schnitt xs ys
          else merge_schnitt (x:xs) ys
```

Offenbar wird bei jedem Schritt ein Element verarbeitet, das danach nicht mehr angefaßt wird. D.h.der Zeitbedarf ist $O((|A| + |B|))$.

**balancierte Suchbäume** Wir gehen vereinfacht davon aus, daß wir Mengen von ganzen Zahlen haben. Die Idee zur Berechnung des Schnitts von $A$ und $B$ ist einfach: man nimmt jedes Element des Suchbaumes für $A$ und testet, ob es im Suchbaum für $B$ ist.

```
schnittSB xa xb = schnittSBliste (blaetterSB xa) xb

schnittSBliste []  _ = Suchbaumleer
schnittSBliste (x:xs)  sb =
    if istinDb x sb
    then einfuegeDbS x (schnittSBliste xs  sb
   else  schnittSBliste xs  sb
```

Der Aufwand berechnet sich folgendermaßen:
Zeit $= |A| * log(|B|)$, d.h. $n * log(n)$ im schlechtesten Fall, wenn $n = |A| + |B|$.

Die zweite Möglichkeit ist es, die Misch-idee auch für Suchbäume zu verwenden:

```
--   Schneiden von Mengen mit Suchbaeumen durch Mischen
--   !! Diese haelt nicht die Bedingung ein, dass
---   Knotenmarkierung = Maximum der linken
--    Schluessel, sondern nur noch die Bedingung: ist in der Mitte
mschnittSB Suchbaumleer _ = Suchbaumleer
mschnittSB _ Suchbaumleer  = Suchbaumleer
mschnittSB (Sblatt x a) (Sblatt y _) =
            if x == y then (Sblatt x a) else Suchbaumleer
mschnittSB (Sknoten lta ka rta) (Sblatt y b) =
       if      y == ka then (Sblatt y b)
       else if y < ka then mschnittSB lta (Sblatt y b)
              else  mschnittSB rta (Sblatt y b)
mschnittSB tl@(Sknoten lta ka rta) tr@(Sknoten ltb kb rtb) =
       if      ka == kb
       then mappendSB (mschnittSB lta ltb) ka  (mschnittSB rta rtb)
```

```
        else if ka < kb
              then mappendSB (mschnittSB lta ltb) ka (mschnittSB rta tr)
              else  mappendSB (mschnittSB lta tr) ka (mschnittSB rta rtb)

mschnittSB x y = mschnittSB y x             --   Sblatt  + Sknoten

mappendSB Suchbaumleer _ t = t
mappendSB t _ Suchbaumleer  = t
mappendSB l k r  = Sknoten l k r
```

Ein Analyseversuch bei balancierten Bäumen ergibt eine Gleichung $\tau(m,n) = \tau(m/2,n/2)+\tau(m/2,n)$ für den Zeitbedarf $\tau$. Dies deutet darauf hin, daß diese Methode keinen linearen Zeitbedarf hat, sondern etwas mehr, vermutlich sogar quadratisch. Nimmt man an, daß bei einem weiteren Schritt der linke und rechte Baum beide zerlegt werden, so ergibt sich eine Rekursionsgleichung von $\tau(m,n) = 3 * \tau(m/2,n/2)$. Deren Lösung ist $O(n^{log 3}) = O(n^{1{,}585})$.

D.h., diese Methode ist offenbar langsamer als das einzelne Testen.

Asymptotisch und im worst-case am besten ist folgende Methode:

- zunächst erzeuge die Liste der Blätter (linear),
- Verarbeite diese mit einer Misch-operation Schnitt, Vereinigung, Differenz (linear),
- und danach wandle die Liste in ein Feld um (linear)
- Baue den Baum aus dem Feld wieder auf (linear)

*Übungsaufgabe 11.* Argumentiere: die Berechnung des Schnitts von Mengen, die als Suchbäume vorliegen, kann mit der naheliegenden Methode nicht mit linearem Zeitbedarf durchgef/ührt werden.

*Übungsaufgabe 12.* Schreibe eine Funktion `merge_vereinigung` und `merge_differenz` für sortierte Listen .

*Übungsaufgabe 13.* Schreibe eine Funktion, die die Potenzmenge, d.h. die Menge aller Teilmengen einer Menge berechnet.

## 3.11 Hash-Tabellen

Dieses Verfahren erlaubt unter bestimmten Umständen eine lineare Verwaltung von Suchtabellen und einen konstanten Aufwand beim Suchen und Einfügen. Die Idee ist es, ein Feld bereitzustellen und aus einem gegebenen Element anhand einer festen Funktion einen Index auszurechnen.

Diese sogenannte *Hash-Funktion* sollte schnell auszurechnen sein und möglichst gleichmäßig über den Indexbereich streuen. Da man die Grenzen des Feldes vorher festlegen muß, gibt es beim Einfügen immer die Möglichkeit, daß verschiedene Elemente den gleichen Index ergeben. Diesen Fall nennt man *Kollision*. Man hat verschiedene Möglichkeiten, dann fortzufahren:

- Erweiterung des Feldes und neue Hash-Funktion
- sekundäre Hash-Funktion für einen zweiten Versuch.
- Ablegen des Elementes im nächsten freien Index.
- Erzeugen einer Liste der Elemente mit gleichem Index.

Die Suche nach einem Element in einer Hash-tabelle geschieht so, daß der Index mit der Hash-funktion berechnet wird. Danach wird in der Hash-tabelle unter diesem Index gesucht. Je nach Hash-Verfahren ist dort ein Element (oder eine Liste) abgelegt, die danach durchsucht werden muß.

Hashtabellen haben gegenüber Suchbäumen nur dann Vorteile, wenn die Hashtabelle genügend viele freie Plätze hat. Wenn sie voll ist und viele Kollisionen entstehen, dann wird das Zeitverhalten schlechter.

Hash-Tabellenfunktionen für Zahlen:

```
inithashoffen f n = (f,array (0, (n-1)) (zip [0..n-1] (repeat [])))

einfuegenhash k (f,ht) =
      let hk = f k
       in (f,ht // [(hk,k:(ht!hk))])

einfuegenhashliste [] h = h
einfuegenhashliste (x:xs) h = einfuegenhash x (einfuegenhashliste xs h)

istinhash n (f,ht) =
      elem n (ht!(f n))

-- einfache hashfunktion:
naivehashf n i = i `mod` n
testhash = let h = inithashoffen (naivehashf 100) 100
               hh = einfuegenhashliste [70..120] h
                in (istinhash 10 hh, istinhash 110 hh)
```