Inhaltsverzeichnis

1 Gloss

Um in Haskell graphische Anzeigen machen zu können, gibt es sehr viele Bibliotheken. Manche sind optimiert auf Geschwindigkeit oder manche speichern einfach nur Bilder. Wir haben uns für das Paket Gloss entschieden. Gloss hat den Vorteil, dass es im Vergleich zu vielen anderen Paketen sehr einfach zu benutzen ist. Dafür ist Gloss nicht ganz so performant in großen Projekten und ist von der Funktionalität beschränkt. Für unsere Zwecke ist es allerdings sehr gut geeignet.
Um externe Bibliotheken im Code benutzen zu können, müssen wir diese importieren. Für Gloss benötigen wir die folgenden zwei Zeilen:

```
1  import Graphics.Gloss
2  import Graphics.Gloss.Interface.Pure.Game
```

1.1 Grundlegendes

Gloss funktioniert für einfache Bilder so, dass man eine Funktion hat, die das gewünschte Objekt zeichnet. Diese hat Signatur:

```
1  display :: Display -> Color -> Picture -> IO ()
```

Die Funktion will als Parameter ein `Display`, eine `Color` und ein `Picture`.

1.1.1 `Display`

`Display` ist dabei ein Parameter, der angibt wie die Form der Ausgabe sein soll. Es gibt hierfür zwei Möglichkeiten:

```
1  data Display = InWindow String (Int, Int) (Int, Int)
2               | FullScreen
```

Die erste Möglichkeit ist `InWindow`. Der `String` ist dabei der Fenstername. Das erste Tupel gibt an wie groß das Fenster in x- und y-Richtung sein soll. Das zweite Tupel die Position auf eurem Bildschirm.
Die zweite Möglichkeit ist `FullScreen`. Hier entsteht einfach eine Ausgabe im Vollbildmodus.

1.1.2 `Color`

Um die Funktion `display` ausführen zu können fehlt noch der Parameter `color`. Dieser gibt die Hintergrundfarbe eures Fensters an. Standardmäßig sind viele Farben bereits definiert, ihr könnt sie jedoch mit der passenden Funktion auch mischen! Die bereits existierenden Farben sind: `black`, `white`, `red`, `green`, `blue`, `yellow`, `cyan`, `magenta`, `rose`, `violet`, `azure`, `aquamarine`, `chartreuse`, `orange`.
Weiterführende Informationen zu den Farben gibt es unter: `https://hackage.haskell.org/package/gloss-1.1.0.0/docs/Graphics-Gloss-Color.html`

1.1.3 `Picture`

Zu Guter letzt fehlt noch der `Picture` Parameter an. Dieser gibt letztendlich an, was ihr zeichnen wollt. Ein einfaches `Picture` wäre zum Beispiel:

```
1  x = circleSolid 100
```

`x` ist nun ein schwarz ausgefüllter Kreis mit einem Radius von 100 Pixeln. Natürlich gibt es nicht nur ausgefüllte Kreise, sondern auch nicht ausgefüllte Kreise, Rechtecke, Polygone, Linien und so weiter. Man muss wissen, dass jedes dieser Objekte immer auf den Punkt (0,0) im Koordinatensystem zentriert ist.

Position eines `Picture` Für die Positionen muss man zuerst folgendes Bild verinnerlichen. Der schwarze Kreis ist die Variable `x` von oben.

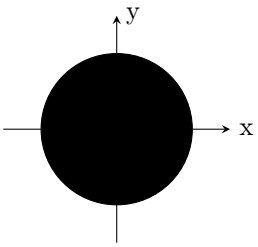


Abbildung 1: Gloss Koordinatensystem

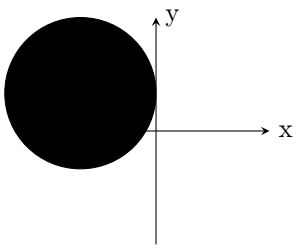
Möchten wir nun den Kreis, der den Typ `Picture` hat, an einer anderen Stelle im Koordinatensystem haben, so können wir ihn einfach bewegen. Zum Beispiel

```
1    xMoved = translate (-100) 50 x
```

Wobei gilt:

```
1    translate :: Float -> Float -> Picture -> Picture
```

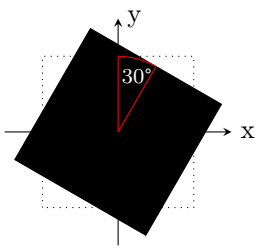
Würden wir `xMoved` nun zeichnen, ergäbe sich folgendes Bild:



`translate` verändert beim Kreis nur die Position innerhalb unseres Windows, alle anderen Eigenschaften bleiben gleich. Natürlich kann man auch Rechtecke zeichnen, aber wie sieht es mit rotierten Rechtecken aus? Zum Glück geht auch das recht einfach.

```
1    rec = rectangleSolid 100 100
2    rotRec = rotate 30 rec
```

Zeigt man `rotRec` an, so erhält man ein Rechteck der Seitenlänge 100 um 30° im Uhrzeigersinn gedreht.



Zusammengefasst ist `rotate` nur eine Funktion, die ein `picture` und einen Winkel nimmt und einfach nur ein neues `picture` zurückgibt. Sie ist also sehr ähnlich zu `translate`.

Mehrere Objekte Um mehrere Objekte in einem Bild anzuzeigen, verwenden wir folgende Funktion:

```
1    pictures :: [Picture] -> Picture
```

Wir packen also einfach ganz viele Objekte in eine Liste und diese Funktion generiert daraus ein einzelnes `Picture`, das wir in `display` reinstecken können.

```
1    kreis1 = translate (-100) 100 $ circleSolid 10
2    kreis2 = color green $ circleSolid 50
3    rec1 = translate 42 42 $ rectangleSolid 50 50
4    p = pictures [ kreis1 , kreis2 , rec1]
5
6    d = display FullScreen red $ p
```

Hier erhalten wir Abbildung 2.

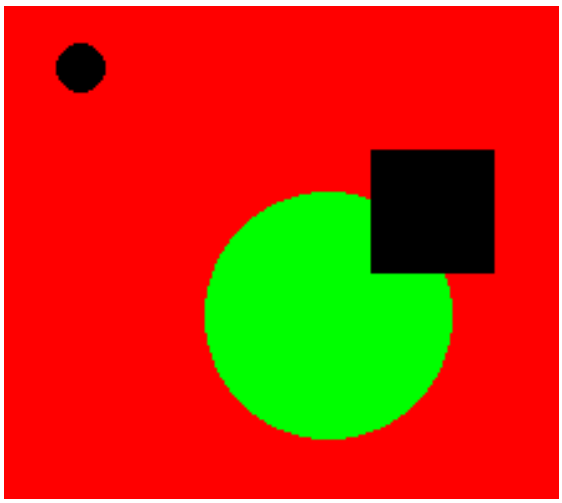


Abbildung 2: Screenshot von Gloss

Natürlich könnten wir auch das neue `p` wieder durch die Gegend schieben, denn es ist ja wieder nur ein `Picture`.

In Zeile 2 habt ihr eine neue Funktion kennengelernt. Denn genauso wie man mit `translate` oder `rotate` Objekte schieben und rotieren kann, so kann man mit `color` die Farben verändern:

```
1    color :: Color -> Picture -> Picture
```

Dabei taucht der Datentyp `Color` von oben wieder auf.
Es gibt noch einige weitere Funktionen und Formen, die ihr gerne wieder auf der folgenden Seite nachschlagen könnt
`https://hackage.haskell.org/package/gloss-1.13.0.1/docs/Graphics-Gloss-Data-Picture.html`.

1.2 Weitere Funktionen

In Gloss gibt es noch viele weitere Funktionen, viele findest du auf: `https://hackage.haskell.org/package/gloss-1.13.0.1/docs/Graphics-Gloss-Data-Picture.html`
Dort ist zum Beispiel erklärt, wie man Text darstellt oder Sachen skaliert.

1.3 Interaktion mit Gloss

Bisher haben wir nur besprochen, wie man in Gloss Bilder zeichnen kann und wie diese modifiziert werden. Um einer interaktiven Anwendung näher zu kommen, müssen wir lernen wie wir regelmäßig unser Bild updaten wenn zum Beispiel Tastatureingaben erfolgen. Möchten wir einen Spieler haben, der sich nach rechts bewegt, wenn die Pfeiltaste gedrückt wird, so müssen wir einerseits die Tastatureingabe verarbeiten und die Position des Spielers ändern. Dazu gehört auch, ob der Spieler 5 mal pro Sekunde bewegt werden soll, oder 50 mal.
Die Hauptfunktion, um eine interaktive Anwendung in Haskell zu realisieren ist `play`. Die Signatur ist:

```
1    :: Display                    -- Vollbild oder Fenstermodus
2    -> Color                      -- Gibt die Hintergrundfarbe an
3    -> Int                        -- Gibt die Framerate pro Sekunde an.
4    -> WorldTyp                      -- Gibt den State aller Objekte an. Dies wird ein eigener Datentyp
5    -> (WorldTyp -> Picture)         -- Funktion: produziert aus WorldTyp das zu malende Picture
6    -> (Event -> WorldTyp -> WorldTyp) -- Funktion: verarbeitet Tastatur- und Mauseingaben
7    -> (Float -> WorldTyp -> WorldTyp) -- Eine Funktion, die eine Kommazahl bekommt. Diese Kommazahl gibt
8                                   -- die Zeit an, die seit dem letzten Update vergangen ist. Diese Funktion
9                                   -- verändert also regelmäßig den Zustand von WorldTyp
10   -> IO                         -- Rückgabetyp
```

Wir werden nun im Detail die jeweiligen Parameter der `play` Funktion besprechen. Habt ihr diese verstanden, könnt ihr euer Spiel bereits programmieren.

1.3.1 `WorldTyp`

Wie ihr bereits wisst, gibt es in funktionalen Sprachen, also auch Haskell, keinen globalen Zustand. Alles was wir für eine Funktion als Eingabe brauchen, müssen wir übergeben. Daher müssen wir also alles was für unser Spiel benötigt wird in Variablen packen. Diese Variablen können wir dann immer an die richtigen Funktionen übergeben. Da selbst kleine Spiele bereits sehr viele Variablen haben, packen wir alle diese Variablen zusammen. Dies könnten wir mit Tupeln, Listen oder Records machen. Da wir möglichst einfach auf bestimmte Variablen zugreifen wollen, verwenden wir hierfür Records. Diese Variable, die alles speichert, nennen wir `WorldTyp`. Für `WorldTyp` erstellen wir unseren eigenen Datentypen. Für ein einfaches Spiel, bei dem ein roter Kreis in vier Richtungen bewegt werden soll, brauchen wir nur folgende Informationen in `WorldTyp`:

```
1        data WorldTyp = World
2        { xPos   :: Float,
3         yPos   :: Float,
4         breite :: Float,
5         lPressed :: Bool,
```

```
6         rPressed :: Bool
7       }
```

`xPos` und `yPos` geben dabei die Position des Balls an. Mit `breite` werden wir nachher einen Bereich spezifizieren, den der Ball nicht verlassen darf. Wollen wir also unseren Ball nachher bewegen, müssen wir diesen Record immer wieder aktualisieren und neu an andere Funktionen übergeben. `lPressed` und `rPressed` werden wir nachher erklären. Der `play` Funktion müssen wir ganz am Anfang einen Initialwert übergeben. In unserem Fall können wir einfach wählen:

```
1       initialWorld :: WorldTyp
2       initialWorld = World
3           {xPos = 0,
4            yPos = 0,
5            breite = 100,
6            lPressed = False,
7            rPressed = False
8           }
```

1.3.2 `draw :: WorldTyp -> Picture`

`draw` baut wie im statischen Fall einfach ein `picture`. Wir müssen dieser Funktion, also nur beibringen, dass sie mit den Informationen aus `WorldTyp` unser Bild zeichnet.

```
1       draw :: WorldTyp -> Picture
2       draw world = ball
3           where
4               x =  xPos world
5               y =  yPos world
6               ball = Color red $ translate x y $ circleSolid 5
```

Wir zeichnen den Ball also und schieben ihn an die richtige Stelle. Dann könnte man noch die Farbe zu rot geben.

Beachte, dass die `draw` Funktion nur die Aufgabe des Zeichnens übernimmt. In dieser Funktion ist keine Logik des Spiels implementiert. Nachdem dort außer dem `Picture` auch keine `World` zurückgegeben wird, ist dies auch nicht möglich!

1.3.3 `handleKeys :: Event -> WorldTyp -> WorldTyp`

Die Funktion `handleKeys` soll nun unsere Tasteneingabe verarbeiten. Diese Funktion wird immer dann aufgerufen, sobald eine Taste gedrückt wird. Da sich aber unsere Kugel nicht nur bewegen soll, wenn die Taste gedrückt wird, sondern sich solange bewegen soll wie die Taste nach unten gedrückt ist, benötigen wir in `WorldTyp` die Zustande `lPressed` und `rPressed`. `lPressed` speichert also in `WorldTyp`, ob denn derzeit die linke Pfeiltaste gedrückt ist. `rPressed` speichert dies für die rechte Pfeiltasten. Nun schauen wir uns an, wie die `handleKeys` Methode für unseren Fall hier aussieht:

```
1     handleKeys :: Event -> WorldTyp -> WorldTyp
2     handleKeys (EventKey (SpecialKey KeyLeft ) Down _ _ ) world = world {lPressed = True}
```

```
handleKeys (EventKey (SpecialKey KeyLeft ) Up _ _ ) world = world {lPressed = False}
handleKeys (EventKey (SpecialKey KeyRight) Down _ _ ) world = world {rPressed = True}
handleKeys (EventKey (SpecialKey KeyRight) Up _ _ ) world = world {rPressed = False}
handleKeys _ world = world
```

Letztendlich machen wir also in `handleKeys` direkt ein Pattern Matching. Der Teil `EventKey (SpecialKey KeyLeft) Down _ _` erstellt dabei ein Event, bei dem die *Spezialtaste Pfeil-links* gedrückt wird. `down` gibt an, dass sie nach unten gedrückt wurde. `up` gibt an, dass die Taste von unten nach oben losgelassen wurde. Die beiden folgenden Parameter benötigen wir erstmal nicht, weshalb sie mit `_` ausgedrückt wurden. Möchte man zum Beispiel eine einzelne Taste einlesen, so ginge das mit:

```
handleKeys :: Event -> WorldTyp -> WorldTyp
handleKeys (EventKey (Char "a") Down _ _ ) world = undefined
```

Falls wir nachher Mauseingaben benötigen, können wir das nachlesen:
`https://hackage.haskell.org/package/gloss-1.1.1.0/docs/Graphics-Gloss-Game.html#t:Event`

1.3.4 `update :: Float -> WorldTyp -> WorldTyp`

Kommen wir nun zur letzten wichtigen Funktion, zu `update`. Der erste Parameter ist dabei ein `Float` der angibt, wie lange seit dem letzten Aufruf von `update` vergangen ist. Wählen wir zum Beispiel eine FPS Zahl von 25, so wird das Bild 25 mal die Sekunde aktualisiert. Idealerweise würde diese `Float` Zahl also immer den Wert $\frac{1}{25} = 0.04$ haben. Allerdings sind diese 25 FPS nur ein grober Richtwert. Gloss versucht das einzuhalten, schafft es allerdings nicht immer perfekt. Deshalb wird dieser Methode die real vergangene Zeit übergeben. Der zweite Parameter ist wieder `WorldTyp`. Unsere Update Methode aktualisiert diesen wieder. Um nun eine Bewegung des Balls zu realisieren, könnte die Funktion so aussehen:

```
update :: Float -> WorldTyp -> WorldTyp
update _ world = case (lPressed world, rPressed world) of
            (True, False) -> if x > - breite then world { xPos = x - 2 } else world
            (False, True) -> if x < breite then world { xPos = x + 2 } else world
            _   -> world
  where
    x = xPos world
    breite = breite world
```

Wie ihr seht, benötigen wir die Zeitdifferenz hier gar nicht. Wir addieren bzw. subtrahieren in jedem Frame einen Offset zur aktuellen Position, wenn die jeweilige Taste gedrückt ist. Aber nur dann, wenn der Ball unseren Korridor, der durch `breite` bestimmt ist, nicht verlässt. Die `yPos` lassen wir unverändert.

1.3.5 `play`

Am Ende müssen wir nur noch Alles zusammenbringen und wir können unsere minimale Anwendung starten.

```
main = do
  play FullScreen white 25 initialWorld draw handleKeys update
```

2 Binärbaume

Bisher haben wir als Datenstruktur zu Speicherung von vielen Objekten Listen verwendet. Listen sind in der Praxis sehr einfach zu benutzen, haben allerdings den Nachteil einer langen Zugriffszeit, wenn die Listen sehr lange werden. Um diesen Missstand zu beheben, werden wir nun *Binärbäume* einführen. Einen Binärbaum könnte zum Beispiel wie in Abbildung 3 aussehen.

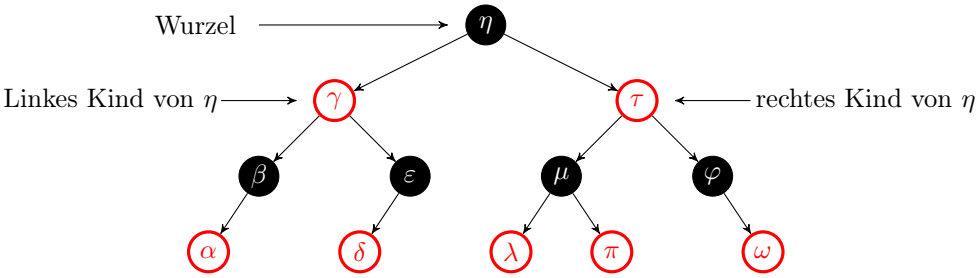


Abbildung 3: Ein Binärbaum mit einigen griechischen Buchstaben

2.1 Regeln für Binärbaume

2.1.1 Wurzel

Jeder Binärbaum hat eine Wurzel. Dies ist der oberste Knoten in der Mitte.

2.1.2 Knoten

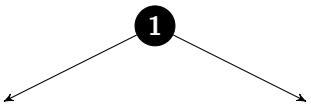
Jeder Knoten hat maximal zwei Kinder. Kinder sind dabei die Knoten, die direkt mit einem Knoten weiter oben verbunden sind. Daher auch der Name *Binärbaum*.

2.1.3 Sortiertheit

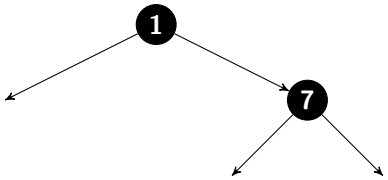
Alle Knoten links von einem Knoten sind immer *kleiner oder gleich* als der betrachtete Knoten. Zum Beispiel sind in unserem griechischen Binärbaum die linken Buchstaben in der Reihenfolge des Alphabets weiter vorne. Alle Knoten rechts sind immer größer bzw. im Alphabet weiter hinten.

2.2 Einfügen

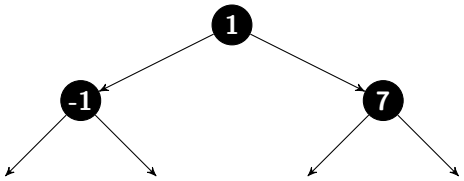
Mit den bekannten Regeln können wir nun Objekte in einen Binärbaum einfügen. Fangen wir mit dem Binärbaum an, der nur Integer Zahlen speichern kann.



Wenn wir nun z.B. die Zahl 7 einfügen wollen, müssen wir nur beachten, dass Zahlen die größer als die 1 sind rechts abgespeichert werden.



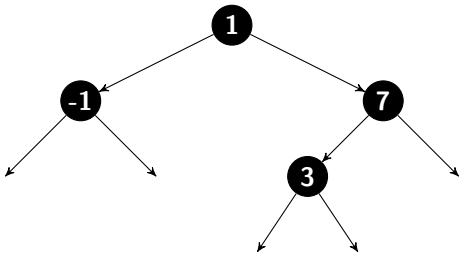
Die Zahl (-1) wird zum Beispiel links eingefügt.



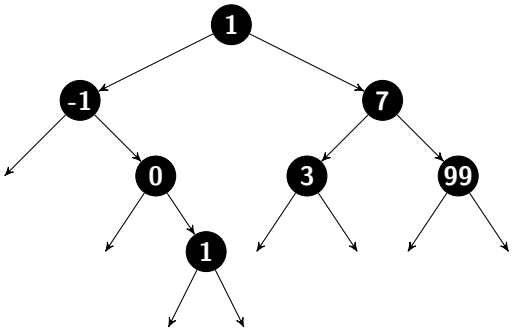
Komplizierter wird es, wenn die Bäume größer werden. Was passiert zum Beispiel nun, wenn die 3 eingefügt wird? Klar ist, dass sie im rechten Teilbaum von der 1 abgespeichert werden muss. Es gibt nun viele Möglichkeiten die 3 so einzufügen, dass die Regeln nicht gebrochen werden. Wir entscheiden uns dafür, dass wir den Baum so wenig wie möglich verändern wollen. Daher geht man wie folgt vor:

- Starte am Knoten, muss links oder rechts eingefügt werden?
- Gehe zum nächsten Knoten, muss nach links oder rechts eingefügt werden?
- Prozedere solange durchführen bis man auf einen Pfeil stößt, der keinen Knoten mehr enthält, dort wird der neue Knoten eingefügt.

Für die 3 erhält man nun:



Wenn wir nun die Zahlen 0, 99, 1 einfügen ergibt sich folgender Baum:

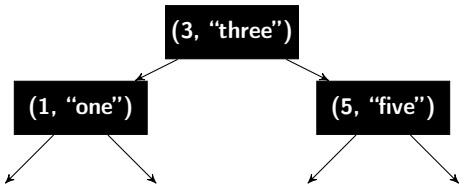


2.3 Binärbaum als Datentyp in Haskell

In Haskell stellen wir den Binärbaum als eigenen Datentypen dar:

```
data Tree = Leaf | Node Int Tree Tree deriving (Show, Eq)
```

Den Zusatz `Show` brauchen wir, damit wir den Baum auch in der Konsole ausgeben können. `Eq` fügt man hinzu, damit man Bäume auf Gleichheit prüfen kann.
Im Allgemeinen will man natürlich keinen Binärbaum haben, der nur Integer Zahlen speichern kann. Deshalb führt man das Prinzip eines key-value Paares ein. Man sagt, dass man zum Schlüssel 1 z.B. das englische Wort *one* speichern möchte. Ein Binärbaum für einige englische Zahlen mit key-value Paaren sieht z.B. so aus:



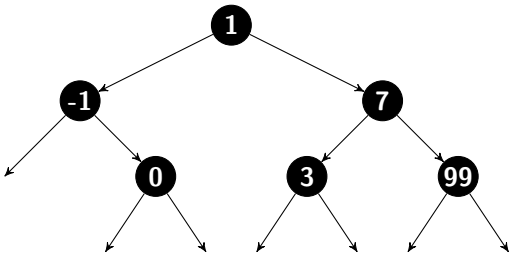
Der Haskell Datentyp ist:

```
data TreeKV = LeafKV | NodeKV (Int,String) TreeKV TreeKV deriving (Show, Eq)
```

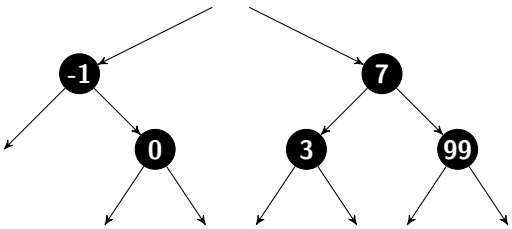
Anstelle des `Int` speichert nun jeder Knoten ein Tupel, wobei an erster Stelle der *key* liegt. Der *key* ist ein `Int`. An zweiter Stelle der *value*, in unserem Fall ein `String`. Das Einfügen funktioniert wie bisher, nur dass man nur die Schlüssel miteinander vergleicht, der zugeordnete Wert zum Schlüssel wird nicht beachtet.

2.4 Löschen

Das Löschen funktioniert leider nicht mehr ganz so einfach wie das Einfügen. Löschen wir bei unserem Zahlenbinärbaum von oben die Wurzel (also 1), so müssen wir die Struktur des Baumes offensichtlich ändern. Wenn man Knoten ohne Kinder löscht, können diese einfach aus dem Baum weggelassen werden.
Nehmen wir also erneut den Binärbaum mit Zahlen:

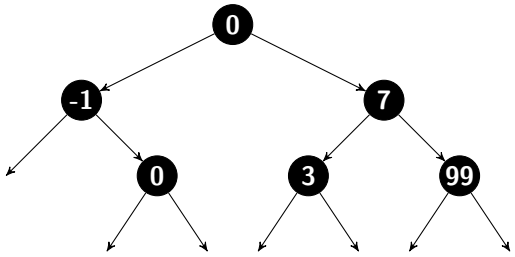


Löscht man nun die 1 aus dem Baum, so können wir den Knoten erstmal wegnehmen:

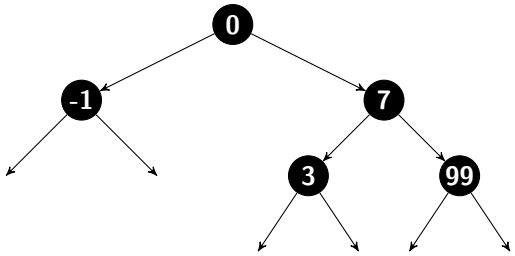


Dieser Baum ist natürlich kein gültiger Binärbaum und lässt sich auch in Haskell so nicht darstellen. Wir müssen nun die Lücke mit einer Zahl füllen, sodass die Regel für den Binärbaum

weiterhin erfüllt sind. Zum Beispiel könnten wir die größte Zahl im linken oder die kleineste Zahl im rechten Teilbaum nehmen. Wir entscheiden uns die größte Zahl im linken Teilbaum. Dann ergibt sich:



Jetzt haben wir aber die Anzahl der Knoten erhöht und den Inhalt des Baumes verändert, denn die 0 kommt nun doppelt vor. Dies wollen wir natürlich nicht haben. Nun können wir einfach im linken Teilbaum die Zahl 0 löschen. Da die Zahl 0 keine Kinder hat, kann man sie einfach entfernen und es ergibt sich:



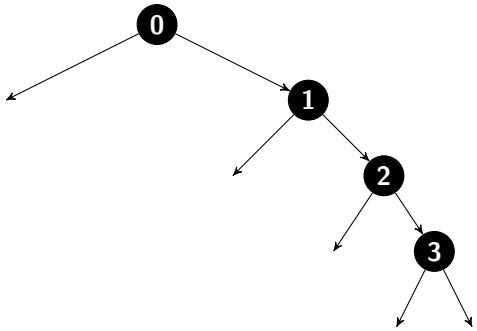
Falls der Knoten 0 ein oder zwei Kinder hätte, würde man das Prozedere wie bei Knoten 1 erneut durchführen.

2.5 Vergleich mit Listen

Der Vorteil von Binärbäumen gegenüber Listen ist die schnellere Zugriffszeit. Bei Listen muss man immer durch die Liste gehen umd beim richtigen Element stoppen. Bei Binärbaumen muss man durch die binäre Struktur sehr wenige Entscheidungen treffen, selbst für große Binärbäume mit mehreren tausend Elementen sind es oft nur 10-15 Entscheidungen die man treffen muss. Bei einer Liste müsste man im schlimmsten mal mehrere tausend Mal weitersuchen.
Desweiteren liegen die Element aufgrund der Struktur des Baumes immer sortiert vor, Listen müsste man eventuell noch in die richtige Reihenfolge bringen.

2.6 Nachteil

Ein einfacher Binärbaum, wie wir ihn hier eingeführt haben, hat einen entscheidenden Nachteil. Fügt man zum Beispiel eine sortierte Liste von Zahlen in einen Binärbaum ein, so wird aus dem Baum eher eine Liste. Das sieht dann zum Beispiel so aus:



Daher gibt es Techniken wie AVL-Bäume oder Red-Black-Trees die den Baum immer wieder verändern, sodass gewisse Regeln erfüllt sind und die Bäume eher *breiter* sind.