Inhaltsverzeichnis

1 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 1 aussehen.

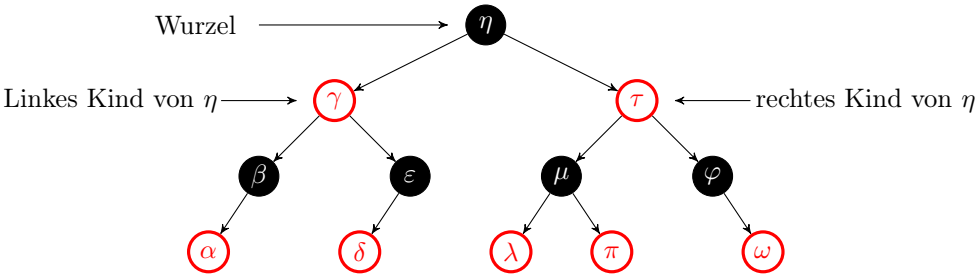


Abbildung 1: Ein Binärbaum mit einigen griechischen Buchstaben

1.1 Regeln für Binärbaume

1.1.1 Wurzel

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

1.1.2 Knoten

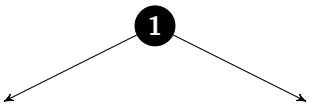
Jeder Knoten hat maximal zwei Kinder. Kinder sind dabei die Knoten, die direkt mit einem Knoten weiter oben verbunden sind. Daher auch der Name *Binärbaum*.

1.1.3 Sortiertheit

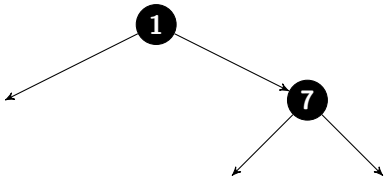
Alle Knoten links von einem Knoten sind immer *kleiner oder gleich* als der betrachtete Knoten. Zum Beispiel sind in unserem griechischen Binärbaum die linken Buchstaben in der Reihenfolge des Alphabets weiter vorne. Alle Knoten rechts sind immer größer bzw. im Alphabet weiter hinten.

1.2 Einfügen

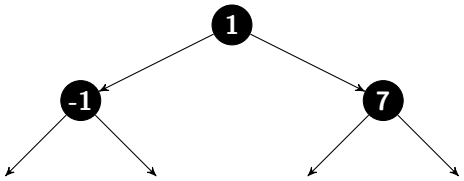
Mit den bekannten Regeln können wir nun Objekte in einen Binärbaum einfügen. Fangen wir mit dem Binärbaum an, der nur Integer Zahlen speichern kann.



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.



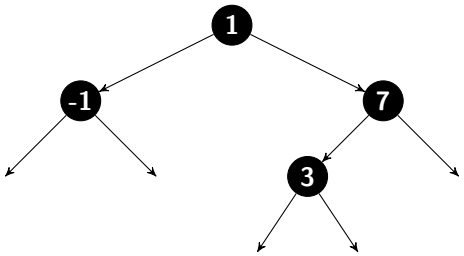
Die Zahl (-1) wird zum Beispiel links eingefügt.



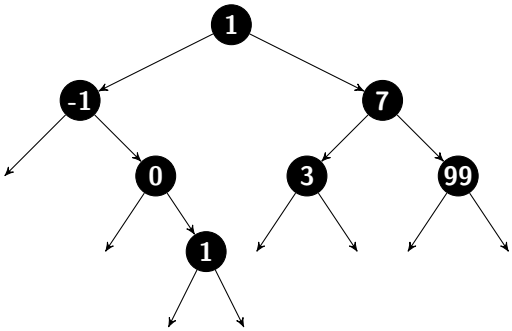
Komplizierter wird es, wenn die Bäume größer werden. Was passiert zum Beispiel nun, wenn die 3 eingefügt wird? Klar ist, dass sie im rechten Teilbaum von der 1 abgespeichert werden muss. Es gibt nun viele Möglichkeiten die 3 so einzufügen, dass die Regeln nicht gebrochen werden. Wir entscheiden uns dafür, 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:

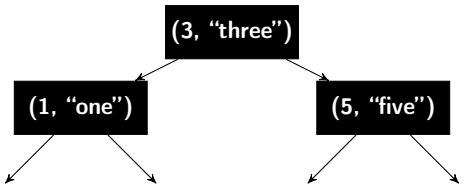


1.3 Binärbaum als Datentyp in Haskell

In Haskell stellen wir den Binärbaum als eigenen Datentypen dar:

```
1 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:



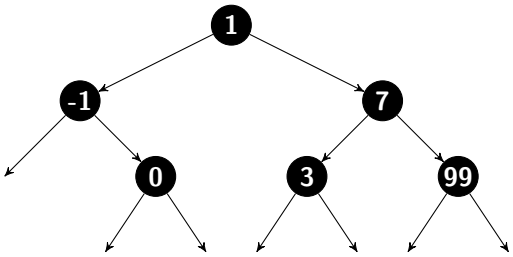
Der Haskell Datentyp ist:

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

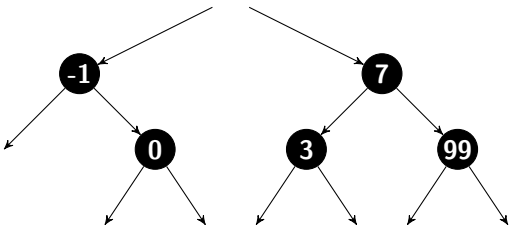
Anstelle des `Int` speichert nun jeder Knoten ein Tupel, wobei an erster Stelle der *key* liegt. Der *key* ist ein `Int`. An zweiter Stelle der *value*, in unserem Fall ein `String`. Das Einfügen funktioniert wie bisher, nur dass man nur die Schlüssel miteinander vergleicht, der zugeordnete Wert zum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

1.4 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:

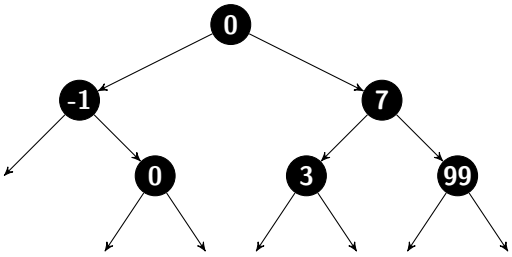


Löscht man nun die 1 aus dem Baum, so können wir den Knoten erstmal wegnehmen:

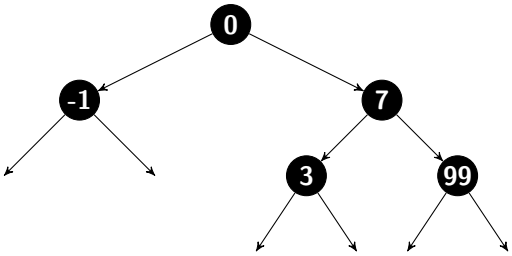


Dieser Baum ist natürlich kein gültiger Binärbaum und lässt sich auch in Haskell so nicht darstellen. Wir müssen nun die Lücke mit einer Zahl füllen, sodass die Regel für den Binärbaum

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:



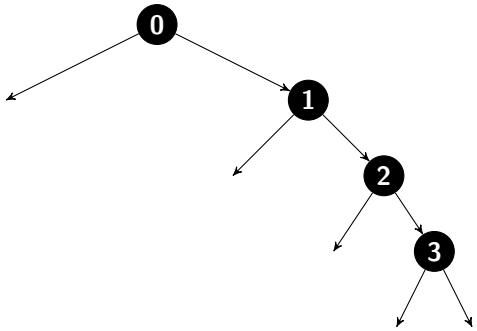
Falls der Knoten 0 ein oder zwei Kinder hätte, würde man das Prozedere wie bei Knoten 1 erneut durchführen.

1.5 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.

1.6 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.