Vorbemerkungen

Was haben wir bisher besprochen? Was wissen wir schon?

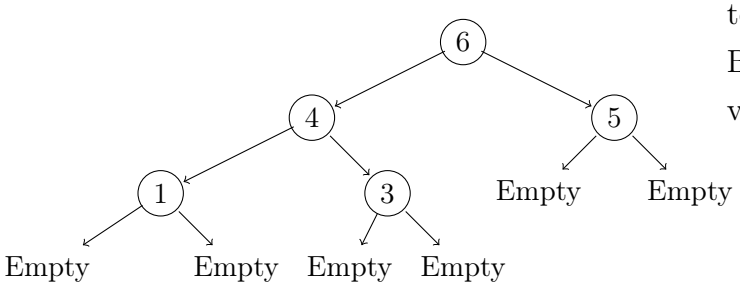
Bäume sind wichtige Datenstrukturen, die ein effizientes Verwalten von Daten ermöglichen.
Außerdem sind Bäume als Datentyp für Haskell interessant, weil es diese im System noch nicht gibt. Somit müssen wir darüber nachdenken, wie man Bäume in einer Computersprache definiert.
Die Vorgehensweise ist die, dass man überlegt, was Bäume für Eigenschaften haben. Und da fällt auf, dass Bäume *rekursiv, polymorph und sortiert* sind. Unsortierte Bäume sind nicht so interessant, weil das Arbeiten mit Daten in diesen Bäumen (z. B. das Suchen) nicht effizient ist. Daher betrachten wir Suchbäume.

Typdefinition und Eigenschaften

Aus diesen Überlegungen ergibt sich folgende Definition:

```
data Bintree a = Empty | Tree (Bintree a) a (Bintree a) deriving (Eq, Ord)
```

Ein Baum ist entweder leer (Empty) oder er hat immer die Struktur eines linken Unterbaums, eines Knoten und eines rechten Unterbaums. Diese Aufzählung wird nun direkt in Haskell als *algebraischer Datentyp* definiert.
Empty und *Tree* sind dabei die Typkonstruktoren des Typs und geben ein Pattern vor. Dieses Pattern ist eine große Stärke von Haskell (und allen funktionalen Sprachen), denn es lassen sich leicht ($\mathcal{O}(1)$) durch *Pattern Matching* auf einzelne Komponenten (z. B. Wurzeln, Unterbäume) zugreifen.



Wichtige Begriffe: Knoten, gerichtete Kante, Vater-/Kindknoten, Blatt, Höhe, Tiefe, geordnet, voll, vollständig, Pfad

Mit obiger Definition wird ein Baum wie folgt dargestellt:

```
tree2 = Tree ((Tree Empty 2 Empty) 5 (Tree (Tree Empty 6 Empty))) 8
        (Tree Empty 12 Empty)
```

Algorithmen auf Bäumen

Wichtige Algorithmen auf Bäumen (in zweifelsfreien Fällen wird nur der Name angegeben):
`anzahlknoten, anzahlblätter, hoehe, bal, istleer, maptree, listezubaum, baumzuliste, gueltig, einfuegen, suchen, loeschen, verschiedene Traversierungen (in-,post-,pre-, levelorder)`

Eine strukturgleiche Ausgabe eines Baumes könnte so aussehen:

```
-- Ausgabe des Baumes ist nicht sehr schön
-- vielleicht so:
instance (Show a) => Show (Bintree a)
         where show b = aus b 0
aus Empty n = " "
aus (Tree l w r) n = aus r (n+1) ++ replicate n '\t' ++ show w ++ "\n"
                     ++ aus l (n+1)
```

Beispielimplementierung

Es folgt ein Listing zu einer Funktion `levelorder`, die die Knoten eines Baums ebenenweise ausliest. Gleichzeitig wird das Kapseln von Methoden in einem eigenen Module gezeigt.

```
1  module ShowTrees (
2    Binbaum(Leer,Knoten), -- Binaerbaeume und ihre Konstruktoren
3    showbinbaum, -- Anzeigen eines Binaerbaums (mit terminierenden
         Leer's)
4   ) where
5
6  -- ============================================================
7  -- Ausgabefunktionen fuer Binaerbaeume
8  -- ============================================================
```

```
9  -- Anzeige einfach polymorpher Binaerbaeume
10 -- Zeigt die terminierenden leeren Baeume an.
11
12 data Binbaum a = Leer | Knoten (Binbaum a) a (Binbaum a)
       deriving (Show)
13
14 showbinbaum :: (a->[Char]) -> Binbaum a -> IO()
15 showbinbaum f t = putStr (showbaum f t 0)
16 showbaum f Leer acc = ""
17 showbaum f (Knoten l w r) acc
18   = showbaum f r (acc+1) ++
19     tabs acc++"/\n"++tabs acc++f w++"\n"++tabs acc++"\\\n"++
20     showbaum f l (acc+1)
21
22 -- ------------------------------------------------------------
23 -- Hilfsfunktion
24 tabs n = ['\t' | x<-[1..n]]
25 -- ------------------------------------------------------------
26 -- Testbaeume
27
28 baum1 = Knoten (Knoten Leer 3 Leer) 4 (Knoten Leer 5 Leer)
29 baum2 = Knoten (Knoten Leer 8 Leer) 15 (Knoten Leer 25 Leer)
30 baum3 = Knoten (Knoten baum1 6 Leer) 9 (Knoten Leer 11 baum2)
31 baum4 = Knoten (Knoten Leer 9 Leer) 14 (Knoten (Knoten Leer 26
       Leer)
32           39 (Knoten Leer 41 (Knoten Leer 55 Leer)))
33
34 -- Tests
35 t1 = showbinbaum show baum1;
36 t2 = showbinbaum show baum2;
37 t3 = showbinbaum show baum3;
38 t4 = showbinbaum show baum4;
39
40 hoehe::Binbaum a -> Int
41 hoehe Leer            = 0
42 hoehe (Knoten l w r) = 1 + max (hoehe l) (hoehe r)
43
44 -- anspruchsvolle Traversierung
```

```
45 levelorder:: Binbaum a -> [a]
46 levelorder a = levelo a (tiefe a)
47
48 levelo:: Binbaum a -> Int -> [a]
49 levelo Leer n = []
50 levelo a 0 = elemt a 0
51 levelo a n = levelo a (n-1) ++ elemt a n
52
53 elemt:: Binbaum a -> Int -> [a]
54 elemt Leer n = []
55 elemt (Knoten l w r) 0 = [w]
56 elemt (Knoten l w r) n = elemt l (n-1) ++ elemt r (n-1)
```

AVL-Bäume

Was fehlt noch?