

Grundlagen der Programmierung 4

Haskell: Bäume

Prof. Dr. Manfred Schmidt-Schauß

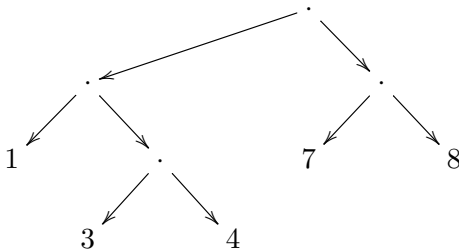
Sommersemester 2018

Binäre, geordnete Bäume in Haskell:

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

- Daten (Markierungen) sind an den Blättern des Baumes
- Jeder (innere) Knoten hat genau zwei Tochterknoten
- es gibt linken und rechten Tochterknoten (geordnet)

Der folgende binäre Baum

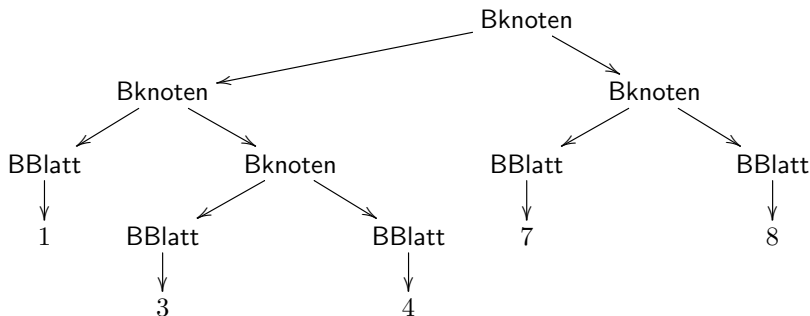


hat eine Darstellung als

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

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

Als Syntaxbaum:



Rand: Liste der Markierungen der Blätter:

```
b_rand (Bblatt x)          = [x]
b_rand (Bknoten bl br) = (b_rand bl) ++ (b_rand br)
```

testet, ob eine Markierung existiert:

```
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)
```

Größe des Baumes:

```
b_size (Bblatt x)          = 1  
b_size (Bknoten bl br) = 1+ (b_size bl) +(b_size br)
```

Summe aller Blätter, falls Zahlen:

```
b_sum (Bblatt x)          = x  
b_sum (Bknoten bl br) = (b_sum bl) + (b_sum br)
```

Tiefe des Baumes:

```
b_depth (Bblatt x) = 0  
b_depth (Bknoten bl br) =  
    1 + (max (b_depth bl) (b_depth br))
```

divide-and-conquer fold

```
foldb :: (a -> a -> a) -> Binbaum a -> a
foldb op (Bblatt x)      = x
foldb op (Bknoten x y) = (foldb op x) 'op' (foldb op y)
```

- für binäre Bäume,
- assoziativer Operator
- kein Initialwert für leere Bäume

```
foldb_summe  = foldb (+)  
foldb_max    = foldb max  
foldb_rand   = foldb (++)
```

foldb (++) nicht optimal

sequentielles fold über binäre Bäume:

```
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)
```

foldbt mit optimiertem Stackverbrauch:

```
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 und b_sum

```
b_rand_bt  = foldbt  (:) []
b_sum_bt'  = foldbt' (+) 0
```

Sei tr binärer Baum mit Randliste $[a_1, \dots, a_n]$

$(\text{foldbt } f \ a \ tr)$ entspricht
 $(f \ a_1 \ (f \ a_2 \ (\dots (f \ a_n \ a) \ \dots s)))$.

D.h. foldbt entspricht foldr

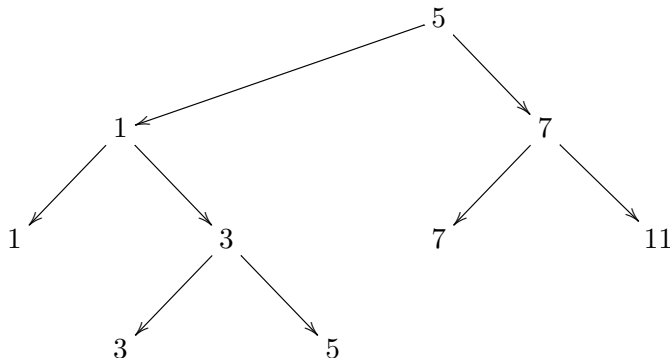
Effizienzsteigerung beim Zugriff auf Elemente einer Menge:

sortierte Liste als Baum

Jeder Knoten enthält als Markierung den größten Schlüssel seines linken Teilbaumes

Laufzeit des Such-Zugriffs:

logarithmisch in der Anzahl der Elemente



```
data Satz a      = Satz Integer a
data Suchbaum a =
    Sblatt Integer a
  | Sknoten (Suchbaum a) Integer (Suchbaum a)
  | Suchbaumleer
```

Verwendung: Suchbaum (Satz a)
(Haskell-Implementierung eines Suchbaum: siehe Skript)

Elementweises Einfügen : sortiert die Eingabe

Vorsicht: Der naiv erzeugte Baum kann
 unbalanciert sein!
 das bewirkt Verschlechterung der Laufzeit
 von $O(\log(n))$ auf $O(n)$

Abhilfe: Balancieren des Baumes durch Rotationen
 zusammen mit dem Einfügen








Laufzeit des Baumaufbaus
mit Balancieren: $O(n * \log(n))$

Data.Map

implementiert balancierte Suchbäume in Haskell:
(Schlüssel,Wert) - Paare, so dass pro Schlüssel nur ein Paar vorkommt.

Funktionen:

singleton	erzeugt Suchbaum mit einem (Schlüssel, Wert)-Paar
insert	fügt ein Element ein.
delete	löscht ein Element zu gegebenem Schlüssel.
lookup	findet ein Element zu gegebenem Schlüssel
adjust	ändert Wert zu gegebenem Schlüssel

```
> :m Data.Map   
Data.Map> let m1 = singleton 1 'A'   
Data.Map> m1   
fromList [(1,'A')]  
Data.Map> let m2 = insert 5 'E' m1   
Prelude Data.Map> m2   
fromList [(1,'A'),(5,'E')]  
Prelude Data.Map> let m3 = insert 3 'C' m2   
Prelude Data.Map> m3   
fromList [(1,'A'),(3,'C'),(5,'E')]
```


mit beliebiger Anzahl Töchter.

Die Programmierung ist analog zu einfachen Bäumen

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

```
nbaumrand :: Nbaum a -> [a]
nbaumrand  (Nblatt x) = [x]
nbaumrand  (Nknoten xs) = concatMap nbaumrand xs
```

```
sumNbaum :: Num t => Nbaum t -> t
sumNbaum (Nblatt x) = x
sumNbaum (Nknoten ts) = sum (map sumNbaum ts)
```