Bäume in Haskell

Vortrag

Christoph Forster Thomas Kresalek

Fachhochschule Wedel – University of Applied Sciences

27. November 2009

Agenda

fhwedel
UNIVERSITY OF APPLIED SCIENCES

Agenda

Motivation

- Natürliche Repräsentation hierarchischer Strukturen
- Ordnung von Mengen durch Hierarchisierung
- Bessere Laufzeiten als Liste
- Implizit Sortierung

verschiedene Baumtypen

- Binäre Bäume (binary tree)
  - Daten nur in den Blättern
  - Daten in Blättern und Knoten
    - jeweils gleiche Typen
    - unterschiedliche Typen, z.B. Int in Knoten für jeweilige Blätter-Anzahl
  - Daten nur in den Knoten
    - Datentyp hat Ordnungsrelation

    $\Rightarrow$ Daten vor Unterbäumen (binary heap)

    $\Rightarrow$ Daten zwischen Unterbäumen (binary search tree)
- N-äre Bäume (Rose Trees)

Agenda

1. Einleitung
2. Binäre Bäume
   - Datenstruktur
   - Operationen
   - Kontrollstruktur
   - Erweiterung
3. Binäre Suchbäume
4. Rose Trees
5. Zusammenfassung & Ausblick

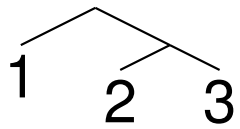
Definition: Datentyp

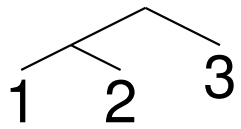
```
{- Datenstruktur Binärbaum  -}

data Btree a = Leaf a
             | Fork (Btree a ) (Btree a)
```

- Hier zunächst einfachste Art: Daten in den Blättern

Beispiel

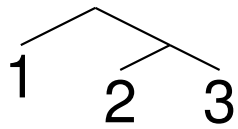
```
    {- Beispiel 1 -}
Fork (Leaf 1) (Fork (Leaf 2) (Leaf 3))

    {- Beispiel 2 -}
Fork (Fork (Leaf 1) (Leaf 2)) (Leaf 3)
```



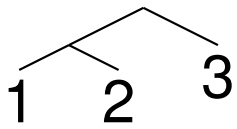


Beispiel

```
    {- Beispiel 1 -}
Fork (Leaf 1) (Fork (Leaf 2) (Leaf 3))

    {- Beispiel 2 -}
Fork (Fork (Leaf 1) (Leaf 2)) (Leaf 3)
```



Beispiel 1



Beispiel 2

Wichtige Maße eines Baumes

```
    {- Größe des Baums  -}

size :: Btree a -> Int
size (Leaf x) = 1
size (Fork xt yt) = size xt + size yt
```

Wichtige Maße eines Baumes

```
    {- Größe des Baums  -}

size :: Btree a -> Int
size (Leaf x) = 1
size (Fork xt yt) = size xt + size yt

    {- Höhe des Baums  -}

height :: Btree a -> Int
height (Leaf x) = 0
height (Fork xt yt) = 1 +
                      (height xt `max` height yt)
```

- Was sagen uns diese Maße?

Wichtige Maße eines Baumes

```
    {- Größe des Baums  -}

size :: Btree a -> Int
size (Leaf x) = 1
size (Fork xt yt) = size xt + size yt

    {- Höhe des Baums  -}

height :: Btree a -> Int
height (Leaf x) = 0
height (Fork xt yt) = 1 +
                      (height xt `max` height yt)
```

- Was sagen uns diese Maße?
  - Größe entspricht der Länge einer Liste
  - Höhe entspricht der maximalen Tiefe

Operationen

```
    {- Tiefe des Baumes -}

depths :: Btree a -> Btree Int
depths = down 0

down :: Int -> Btree a -> Btree Int
down n (Leaf x) = Leaf n
down n (Fork xt yt) = Fork (down (n + 1) xt)
                           (down (n + 1) yt)
```

Operationen

```
    {- Tiefe des Baumes -}

depths :: Btree a -> Btree Int
depths = down 0

down :: Int -> Btree a -> Btree Int
down n (Leaf x) = Leaf n
down n (Fork xt yt) = Fork (down (n + 1) xt)
                           (down (n + 1) yt)

    {- Baum in Liste umwandeln -}

flatten :: Btree a -> [a]
flatten (Leaf x) = [x]
flatten (Fork xt yt) = flatten xt ++ flatten yt
```

- *depths* ersetzt die Blätter durch die Tiefe
- Reihenfolge der durch *flatten* erzeugten Liste?

Operationen

```
    {- Tiefe des Baumes -}

depths :: Btree a -> Btree Int
depths = down 0

down :: Int -> Btree a -> Btree Int
down n (Leaf x) = Leaf n
down n (Fork xt yt) = Fork (down (n + 1) xt)
                           (down (n + 1) yt)

    {- Baum in Liste umwandeln -}

flatten :: Btree a -> [a]
flatten (Leaf x) = [x]
flatten (Fork xt yt) = flatten xt ++ flatten yt
```

- *depths* ersetzt die Blätter durch die Tiefe
- Reihenfolge der durch *flatten* erzeugten Liste?
  - $\Rightarrow$ von Links nach Rechts

Erstellung von Bäumen

```
    {- Erstellt Bäume aus einer Liste  -}

mkBtree :: [a] -> Btree a
mkBtree xs
  | (m == 0)  = Leaf (unwrap xs)
  | otherwise = Fork (mkBtree ys) (mkBtree zs)
  where         m = (length xs) `div` 2
              (ys, zs) = splitAt m xs
          unwrap [x] = x
```

Erstellung von Bäumen

```
    {- Erstellt Bäume aus einer Liste  -}

mkBtree :: [a] -> Btree a
mkBtree xs
  | (m == 0)  = Leaf (unwrap xs)
  | otherwise = Fork (mkBtree ys) (mkBtree zs)
  where         m = (length xs) `div` 2
              (ys, zs) = splitAt m xs
          unwrap [x] = x
```

- Erstellt Baum minimaler Höhe
- Rekursives Aufbauen von optimalen Teilbäumen
- Halbierung der Ausgangsliste in jedem Schritt
- Laufzeit?

Erstellung von Bäumen

```
    {- Erstellt Bäume aus einer Liste  -}

mkBtree :: [a] -> Btree a
mkBtree xs
  | (m == 0)  = Leaf (unwrap xs)
  | otherwise = Fork (mkBtree ys) (mkBtree zs)
  where         m = (length xs) `div` 2
              (ys, zs) = splitAt m xs
          unwrap [x] = x
```

- Erstellt Baum minimaler Höhe
- Rekursives Aufbauen von optimalen Teilbäumen
- Halbierung der Ausgangsliste in jedem Schritt
- Laufzeit? O(n log n)

Maximum Ermittlung

```
    {- Maximum der gespeicherten Werte -}
maxBtree :: (Ord a) => Btree a -> a
maxBtree (Leaf x) = x
maxBtree (Fork xt yt)=(maxBtree xt)
                      `max` (maxBtree yt)
```

Maximum Ermittlung

```
    {- Maximum der gespeicherten Werte -}
maxBtree :: (Ord a) => Btree a -> a
maxBtree (Leaf x) = x
maxBtree (Fork xt yt)=(maxBtree xt)
                      `max` (maxBtree yt)
```

Gesetze

```
    {- Gesetze über bisherige Operationen -}
size = length . flatten

height = maxBtree . depths

    {- Stimmts oder nicht? -}
mkBtree . flatten = id
flatten . mkBtree = id
```

Maximum Ermittlung

```
    {- Maximum der gespeicherten Werte -}
maxBtree :: (Ord a) => Btree a -> a
maxBtree (Leaf x) = x
maxBtree (Fork xt yt)=(maxBtree xt)
                      `max` (maxBtree yt)
```

Gesetze

```
    {- Gesetze über bisherige Operationen -}
size = length . flatten

height = maxBtree . depths

    {- Stimmts oder nicht? -}
mkBtree . flatten = id
flatten . mkBtree = id
```

$\Rightarrow$ Stimmt!

Verarbeitung mit map

```
    {- Map für den Binären Baum  -}

mapBtree :: (a -> b) -> Btree a -> Btree b
mapBtree f (Leaf x) = Leaf (f x)
mapBtree f (Fork xt yt) = Fork (mapBtree f xt)
                               (mapBtree f yt)
```

Verarbeitung mit map

```
    {- Map für den Binären Baum  -}

mapBtree :: (a -> b) -> Btree a -> Btree b
mapBtree f (Leaf x) = Leaf (f x)
mapBtree f (Fork xt yt) = Fork (mapBtree f xt)
                               (mapBtree f yt)
```

- Analogon zu *map* auf Listen
- Anwendung von Funktionen auf die Elemente des Baums
- *mapBtree* ist wie *map* für Listen ein Funktor für Bäume

Gesetze für map

```
    {- Gesetze für map auf binärem Baum  -}

mapBtree id = id

mapBtree (f . g) = mapBtree f . mapBtree g

map f . flatten = flatten . mapBtree f
```

- Was ist ein Funktor?

Gesetze für map

```
    {- Gesetze für map auf binärem Baum  -}

mapBtree id = id

mapBtree (f . g) = mapBtree f . mapBtree g

map f . flatten = flatten . mapBtree f
```

- Was ist ein Funktor?
  - $\Rightarrow$ mapBtree, genau wie map bei Listen

Gesetze für map

```
    {- Gesetze für map auf binärem Baum  -}

mapBtree id = id

mapBtree (f . g) = mapBtree f . mapBtree g

map f . flatten = flatten . mapBtree f
```

- Was ist ein Funktor?
  - $\Rightarrow$ mapBtree, genau wie map bei Listen

Definition: Funktor

Ein Operator, der aus einem oder mehreren singulären Termen wiederum einen singulären Term erzeugt.

Quelle: angelehnt an CARNAP, 1954

Kontrollstruktur fold

```
    {- Fold für den Binären Baum -}

foldBtree :: (a->b) -> (b->b->b) -> Btree a->b
foldBtree f g (Leaf x) = f x
foldBtree f g (Fork xt yt) = g( foldBtree f g xt)
                                (foldBtree f g yt)
```

- Analogon zu fold auf Listen
- *fold* ermöglicht Verarbeitung der Datenstruktur

Kontrollstruktur fold

```
    {- Fold für den Binären Baum -}

foldBtree :: (a->b) -> (b->b->b) -> Btree a->b
foldBtree f g (Leaf x) = f x
foldBtree f g (Fork xt yt) = g( foldBtree f g xt)
                                (foldBtree f g yt)
```

- Analogon zu fold auf Listen
- *fold* ermöglicht Verarbeitung der Datenstruktur
- Funktion f bearbeitet Blätter
- Funktion g arbeitet in den Knoten

Nutzung von fold für Operationen

```
size' = foldBtree (const 1) ( + )
```

Nutzung von fold für Operationen

```
size' = foldBtree (const 1) ( + )

height' = foldBtree (const 0) maxsubs
     where m `maxsubs` n = 1 + m `max` n
```

Nutzung von fold für Operationen

```
size' = foldBtree (const 1) ( + )

height' = foldBtree (const 0) maxsubs
     where m `maxsubs` n = 1 + m `max` n

flatten' = foldBtree wrap ( ++ )
```

Nutzung von fold für Operationen

```
size' = foldBtree (const 1) ( + )

height' = foldBtree (const 0) maxsubs
     where m `maxsubs` n = 1 + m `max` n

flatten' = foldBtree wrap ( ++ )

maxBtree' :: (Ord a) => Btree a -> a
maxBtree' = foldBtree id (max)
```

Nutzung von fold für Operationen

```
size' = foldBtree (const 1) ( + )

height' = foldBtree (const 0) maxsubs
     where m `maxsubs` n = 1 + m `max` n

flatten' = foldBtree wrap ( ++ )

maxBtree' :: (Ord a) => Btree a -> a
maxBtree' = foldBtree id (max)


mapBtree' f = foldBtree (Leaf . f) Fork
```

Definition: Datentyp

```
    {- Datenstruktur Erweiterter Binärbaum  -}
data ATree a = ALeaf a
             | AFork Int (ATree a) (ATree a)
```

- Zusätzlicher Wert im Knoten
- Speichert Größe des linken Teilbaums

Konstruktor

```
    {- Konstruktor AFork -}
fork :: ATree a -> ATree a -> ATree a
fork xt yt = AFork n xt yt
             where n = lsize xt

    {- Konstruktur ATree-}
mkATree :: [a] -> ATree a
mkATree xs
   | (m==0) = ALeaf (unwrap xs)
   | otherwise = fork (mkATree ys) (mkATree zs)
   where       m = (length xs) `div` 2
         (ys,zs) = splitAt m xs
     unwrap [x] = x
```

Indizierter Zugriff

```
   {- indizierter Zugriff -}
retrieve :: ATree a -> Int -> a
retrieve (ALeaf x) 0 = x
retrieve (AFork m xt yt) k =
   if k < m
   then retrieve xt k
   else retrieve yt (k-m)
```

- Nutzt gespeicherte Informationen in Knoten
- Effizienter als Zählen in Listen, bei großen n

Definition: Datentyp

```
    {- Datenstruktur Binärer Suchbaum  -}
data (Ord a) Stree a = Null
                     | Fork (Stree a ) a (Stree a)
```

Eigenschaften

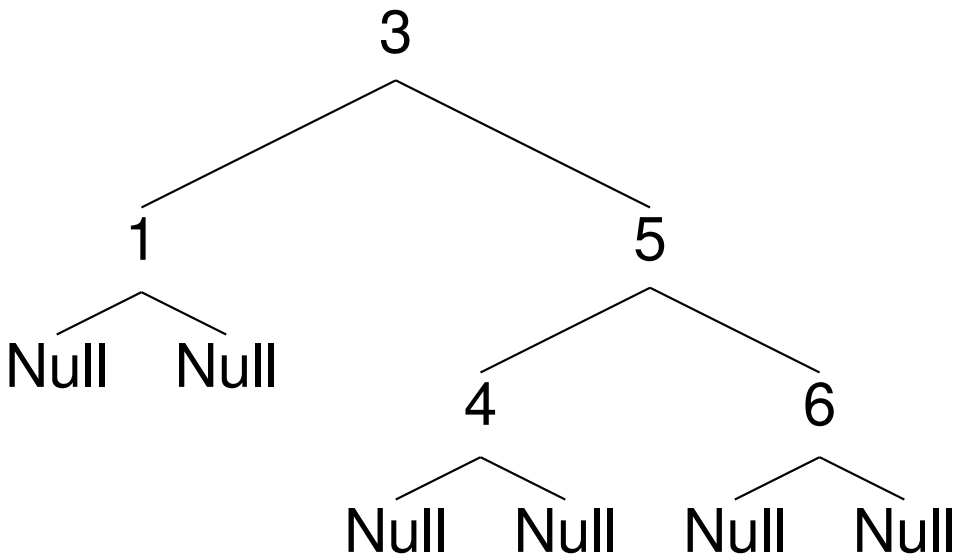
- Die Daten befinden sich in den Knoten zwischen den Subbäumen
- Für den Datentyp a muss eine Ordnungsrelation definiert sein

$\Rightarrow$ Fork xt e yt $\Rightarrow \forall x : x \in xt \rightarrow (x \leq e)$, $x \in yt \rightarrow (x > e)$

$\Rightarrow$ Gute Repräsentationsmöglichkeit für Multimengen mit Ordnungsrelation

Beispiel

```
{- Beispiel 1 -}
  Fork (Fork Null 1 Null) 3 (Fork (Fork Null 4 Null)
      5 (Fork Null 6 Null))
```



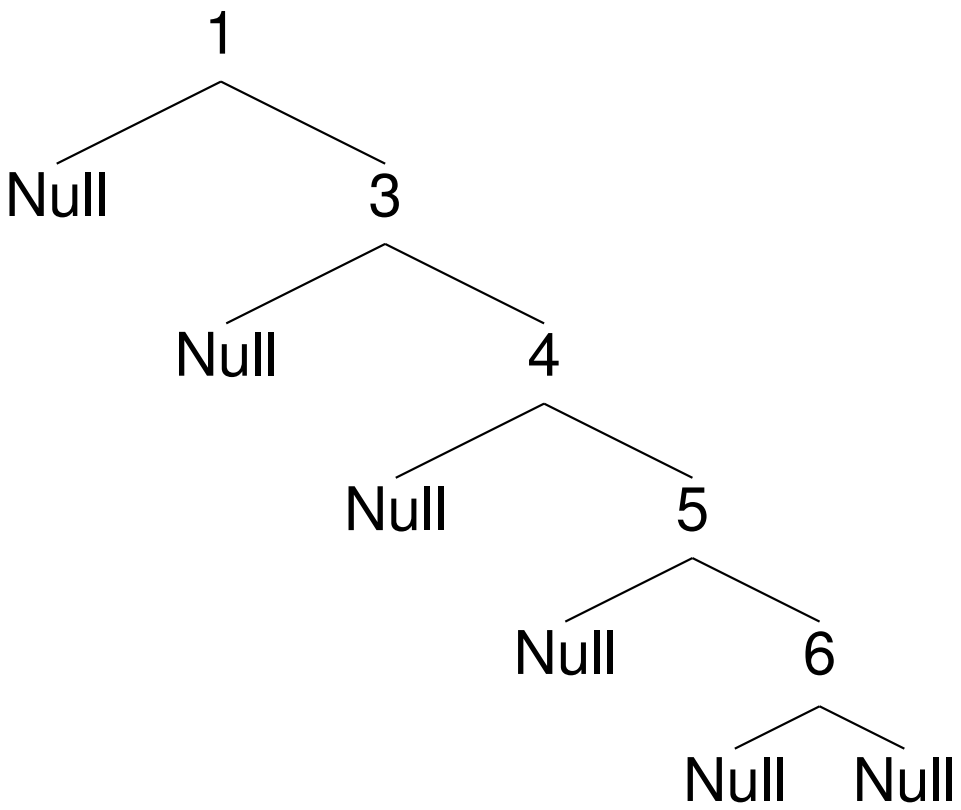
Erzeugung

```
{- Erzeugung eines Suchbaums aus einer Liste -}
mkStree :: (Ord a) => [a] -> Stree a
mkStree [] = Null
mkStree (x:xs) = Fork (mkStree ys) x (mkStree zs)
          where (ys,zs) = partition (<= x) xs

{- Aufteilung zweier Listen an Hand von Prädikat -}
partition :: (a -> Bool) -> [a] -> ([a],[a])
partition p xs = (filter p xs,filter (not . p) xs)
```

- Aus sortierten Listen erzeugte Bäume sind leider sehr unbalanciert

Beispiel

```
{- Beispiel 2 -}
{- mkStree [1,3,4,5,6] => -}
Fork Null 1 (Fork Null 3 (Fork Null 4 (Fork Null 5
    (Fork Null 6 Null))))
```



Erzeugung einer Liste aus einem Stree

```
flatten :: (Ord a) => Stree a -> [a]
flatten Null = []
flatten (Fork xt x yt) = flatten xt ++ (x:flatten yt)
```

Sortieren

```
sort' :: (Ord a) => [a] -> [a]
sort' = flatten . mkStree
```

- entspricht einer möglichen Implementierung des Quicksort-Algorithmus

Höhe

```
heightST :: (Ord a) => Stree a -> Integer
heightST Null = 0
heightST (Fork xt x yt) = 1 + (max (heightST xt)
    (heightST yt))
```

- Die Höhe eines Suchbaums ergibt sich über die Knoten

Membership

```
member :: (Ord a) => a -> Stree a -> Bool
member x Null = False
member x (Fork xt y yt)
     | (x < y) = member x xt
     | (x == y) = True
     | (x > y) = member x yt
```

- besonders effizient, wenn der Baum perfekt balanciert ist

Einfügen

```
insert :: (Ord a) => a -> Stree a -> Stree a
insert x Null = Fork Null x Null
insert x (Fork xt y yt)
     | (x < y) = Fork (insert x xt) y yt
     | (x == y) = Fork xt y yt
     | (x > y) = Fork xt y (insert x yt)
```

Entfernen von Elementen

```
delete :: (Ord a) => a -> Stree a -> Stree a
delete x Null = Null
delete x (Fork xt y yt)
     | (x < y) = Fork (delete x xt) y yt
     | (x == y) = join xt yt
     | (x > y) = Fork xt y (delete x yt)
```

- Teilbäume müssen an der Stelle des gelöschten Elements zusammengeführt werden
- Das Ergebnis soll perfekt balanciert sein.

Join

```
join :: (Ord a) => Stree a -> Stree a -> Stree a
join Null yt = yt
join (Fork ut x vt) yt = Fork ut x (join vt yt)
```

- Einfache Implementierung
- Aber Ergebnis ist nicht gut balanciert

Join

```
join :: (Ord a) => Stree a -> Stree a -> Stree a
join xt yt =
   if empty yt
   then xt
   else Fork xt (headTree yt) (tailTree yt)
```

- komplexere Implementierung
- empty, headTree und tailTree müssen implementiert werden

Empty

```
empty :: (Ord a) => Stree a -> Bool
empty Null = True
empty (Fork xt y yt) = False
```

Head und Tail

```
{- Head -}
headTree :: (Ord a) => Stree a -> a
headTree = fst . splittree
{- Tail -}
tailTree :: (Ord a) => Stree a -> Stree a
tailTree = snd . splitTree
```

splitTree

```
splitTree :: (Ord a) => Stree a -> (a,Stree a)
splitTree (Fork xt y yt) =
     if empty xt
     then (y,yt)
     else (x, Fork wt y yt)
             where (x,wt) = splitTree xt
```

Join

```
join' :: (Ord a) => Stree a -> Stree a -> Stree a
join' xt yt =
    if empty yt
    then xt else
    Fork xt headOfTree tailOfTree
           where (headOfTree,tailOfTree) = splitTree yt
```

$\Rightarrow$ splitTree wird nur einmal aufgerufen.

Verarbeitung mit map

```
    {- Map für den Binären Suchbaum  -}

mapStree :: (a -> b) -> Stree a -> Stree b
mapStree f Null = Null
mapStree f (Fork xt x yt) =
    Fork (mapBtree f xt) (f x) (mapBtree f yt)
```

- Analog zur Implementierung für Btree

fold

```
foldStree :: (Ord a) => (a -> b) -> (b -> b -> b
    -> b) -> b -> Stree a -> b
foldStree f g v Null = v
foldStree f g v (Fork xt x yt) =
    g (foldStree f g v xt)
      (f x) (foldStree f g v yt)
```

Nutzung von fold für Operationen

```
{- Höhe -}
heightST' ::  (Ord a) => STree a -> Integer
heightST' =
   foldSTree (const 1) (\ x y z -> (max x z) + y) 0

{- Bildung einer Liste -}
flatten' :: (Ord a) => STree a -> [a]
flatten' =
   foldSTree (\ x -> [x])
     (\ x y z -> x ++ (y ++ z)) []
```

- FoldStree nicht direkt für member, insert und delete einsetzbar
- allerdings auch hier Strukturübereinstimmungen
- Schaffung einer weiteren Kontrollstruktur sinnvoll?

Erweitertes fold

```
foldSTree2 :: Ord a => (STree a -> a -> STree a -> c)
                    -> (c -> a -> STree a -> c)
                    -> (STree a -> a -> c -> c)
                    -> (a -> c)
                    -> a
                    -> STree a
                    -> c
foldSTree2 fm fl fr fn y Null = fn y
foldSTree2 fm fl fr fn x (Fork xt y yt)
    | (x < y)  = fl (foldSTree2 fm fl fr fn x xt) y yt
    | (x == y) = fm xt x yt
    | (x > y)  = fr xt y (foldSTree2 fm fl fr fn x yt)
```

Nutzung des erweiterten fold

```
member' :: (Ord a) => a -> Stree a -> Bool
member' = foldStree2 (\ x y z -> True) (\ x y z -> x)
                     (\ x y z -> z) (\ x -> False)


insert' :: (Ord a) => a -> Stree a -> Stree a
insert' = foldStree2 (\ xt y yt -> Fork xt y yt)
                     (\ xt x yt -> Fork xt x yt)
                     (\ xt x yt -> Fork xt x yt)
                     (\ x -> Fork Null x Null)


delete' :: (Ord a) => a -> Stree a -> Stree a
delete' = foldStree2 (\ xt y yt -> join' xt yt)
                     (\ xt x yt -> Fork xt x yt)
                     (\ xt x yt -> Fork xt x yt)
                     (\ x -> Null)
```

Nutzung des erweiterten fold

```
buildFork xt x yt = Fork xt x yt

insert'' :: (Ord a) => a -> STree a -> STree a
insert'' = foldSTree2 buildFork
                      buildFork
                      buildFork
                      (\ x -> Fork Null x Null)

delete'' :: (Ord a) => a -> STree a -> STree a
delete'' = foldSTree2 (\ xt y yt -> join' xt yt)
                      buildFork
                      buildFork
                      (\ x -> Null)
```

Agenda

- Angelehnt an Verzweigungen des Rhododendron Busches

Rose Tree

Definition: Datentyp

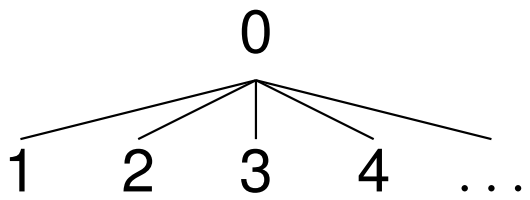
```
{- Datenstruktur Rose Trees  -}

data Rose a = Node a [Rose a]
```

- Allgemeiner definiert
- Beliebig viele Kinder
- Binäre Bäume $\equiv$ Rose Tree 2. Ordnung
- Blätter sind durch leere Listen dargestellt

Unendliche Bäume

Beispiel

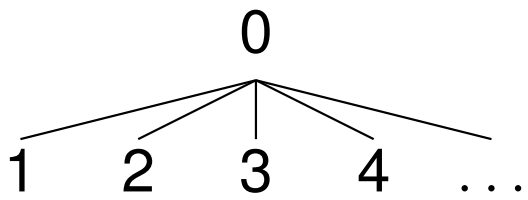
```
      {- Unendlicher Rose Tree  -}

  Node 0 [ Node n [] | n <- [1..]]
```



Unendliche Bäume

Beispiel

```
    {- Unendlicher Rose Tree  -}

Node 0 [ Node n [] | n <- [1..]]
```



- Spezielle Verarbeitung notwendig (breath-first)
- Analogon zu unendlichen Listen
- Im Folgenden: Endliche Rose Trees

Wichtige Maße eines Baumes

```
    {- Anzahl der Elemente  -}

size :: Rose a -> Int
size (Node x xts) = 1 + sum (map size xts)
```

Wichtige Maße eines Baumes

```
    {- Anzahl der Elemente  -}

size :: Rose a -> Int
size (Node x xts) = 1 + sum (map size xts)

    {- Höhe des Baums  -}

height :: Rose a -> Int
height (Node x xts) = 1 + maxlist (map height xts)
       where maxlist = foldl (max) 0
```

Wichtige Maße eines Baumes

```
    {- Anzahl der Elemente  -}

size :: Rose a -> Int
size (Node x xts) = 1 + sum (map size xts)

    {- Höhe des Baums  -}

height :: Rose a -> Int
height (Node x xts) = 1 + maxlist (map height xts)
       where maxlist = foldl (max) 0
```

- Höhe und Größe eines Rose Tree immer größer 0

Plätten eines Rosenbaumes

```
    {- Plätten des Baums  -}

flatten :: Rose a -> [a]
flatten (Node x xts) = x:concat(map flatten xts)
```

Plätten eines Rosenbaumes

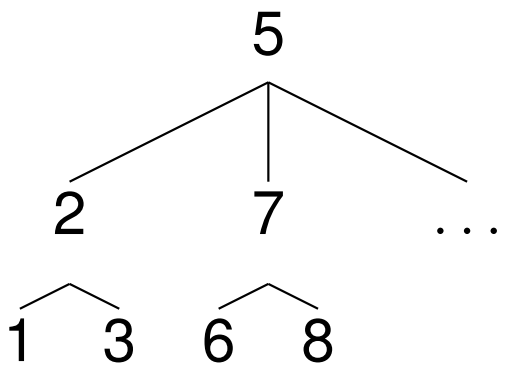
```
    {- Plätten des Baums  -}

flatten :: Rose a -> [a]
flatten (Node x xts) = x:concat(map flatten xts)
```

- Plätten nur bei endlichen Bäumen (depth-first)
- Plätten effizient O(n log n)
- Reihenfolge?

Plätten eines Rosenbaumes

```
    {- Plätten des Baums  -}

flatten :: Rose a -> [a]
flatten (Node x xts) = x:concat(map flatten xts)
```

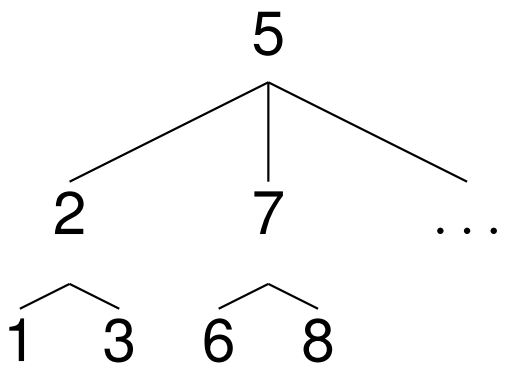
- Plätten nur bei endlichen Bäumen (depth-first)
- Plätten effizient O(n log n)
- Reihenfolge?



Plätten eines Rosenbaumes

```
    {- Plätten des Baums  -}

flatten :: Rose a -> [a]
flatten (Node x xts) = x:concat(map flatten xts)
```

- Plätten nur bei endlichen Bäumen (depth-first)
- Plätten effizient O(n log n)
- Reihenfolge?



Zuerst Knoten, dann Kinder von Links nach Rechts
$\Rightarrow$5,2,1,3,7,6,8

Kontrollstruktur fold

```
{- Fold für den Rosenbaum -}

foldRose :: (a -> [b] -> b) -> Rose a -> b
foldRose f (Node x xts) = f x (map (foldRose f) xts)
```

Kontrollstruktur fold

```
    {- Fold für den Rosenbaum -}

foldRose :: (a -> [b] -> b) -> Rose a -> b
foldRose f (Node x xts) = f x (map (foldRose f) xts)
```

Element Verarbeitung mit map

```
    {- map für den Rosenbaum -}

mapRose :: (a -> b) -> Rose a -> Rose b
mapRose f = foldRose (Node . f)
```

Beispiele mit fold

```
    {- Größe eines Rosenbaums -}

size :: Rose a -> Int
size = foldRose f
   where f x ns = 1 + sum ns
```

Beispiele mit fold

```
    {- Größe eines Rosenbaums -}

size :: Rose a -> Int
size = foldRose f
   where f x ns = 1 + sum ns

    {- Maximum eines Rosenbaumes -}

maxRose :: (Ord a, Num a) =>  Rose a -> a
maxRose = foldRose f
   where f x ns = x `max` maxlist ns
         maxlist = foldl (max) 0
```

Beispiele mit fold

```
    {- Größe eines Rosenbaums -}

size :: Rose a -> Int
size = foldRose f
   where f x ns = 1 + sum ns

    {- Maximum eines Rosenbaumes -}

maxRose :: (Ord a, Num a) =>  Rose a -> a
maxRose = foldRose f
   where f x ns = x `max` maxlist ns
         maxlist = foldl (max) 0

    {- Plätten eines Rosenbaumes -}

flatten :: Rose a -> [a]
flatten = foldRose f
   where f x ns = x:concat ns
```

Agenda

Bäume

- 3 1/2 verschiedene Spezialfälle
  - Binäre Bäume (Btrees / Atrees)
  - Binäre Suchbäume (Strees)
  - N-äre Bäume (Rose trees)

Bäume

- 3 1/2 verschiedene Spezialfälle
  - Binäre Bäume (Btrees / Atrees)
  - Binäre Suchbäume (Strees)
  - N-äre Bäume (Rose trees)
- Datenstrukturen
- Operationen
- Kontrollstrukturen

Bäume

- Ausblick
  - Binäre Halden

Bäume

- Ausblick
  - Binäre Halden
  - Anwendungsfälle
    - Merteens' number
    - Huffman trees

Vielen Dank für die Aufmerksamkeit!