bäume in Haskell

Alexander Meirowski & Patrick Schmidt

Bei: Prof. Dr. Schmidt, FH Wedel
Freitag, 26. November 2010

http://de.flash-screen.com/free-wallpaper/uploads/200703/imgs/1173789191_1024x768_leaf-shadow.jpg

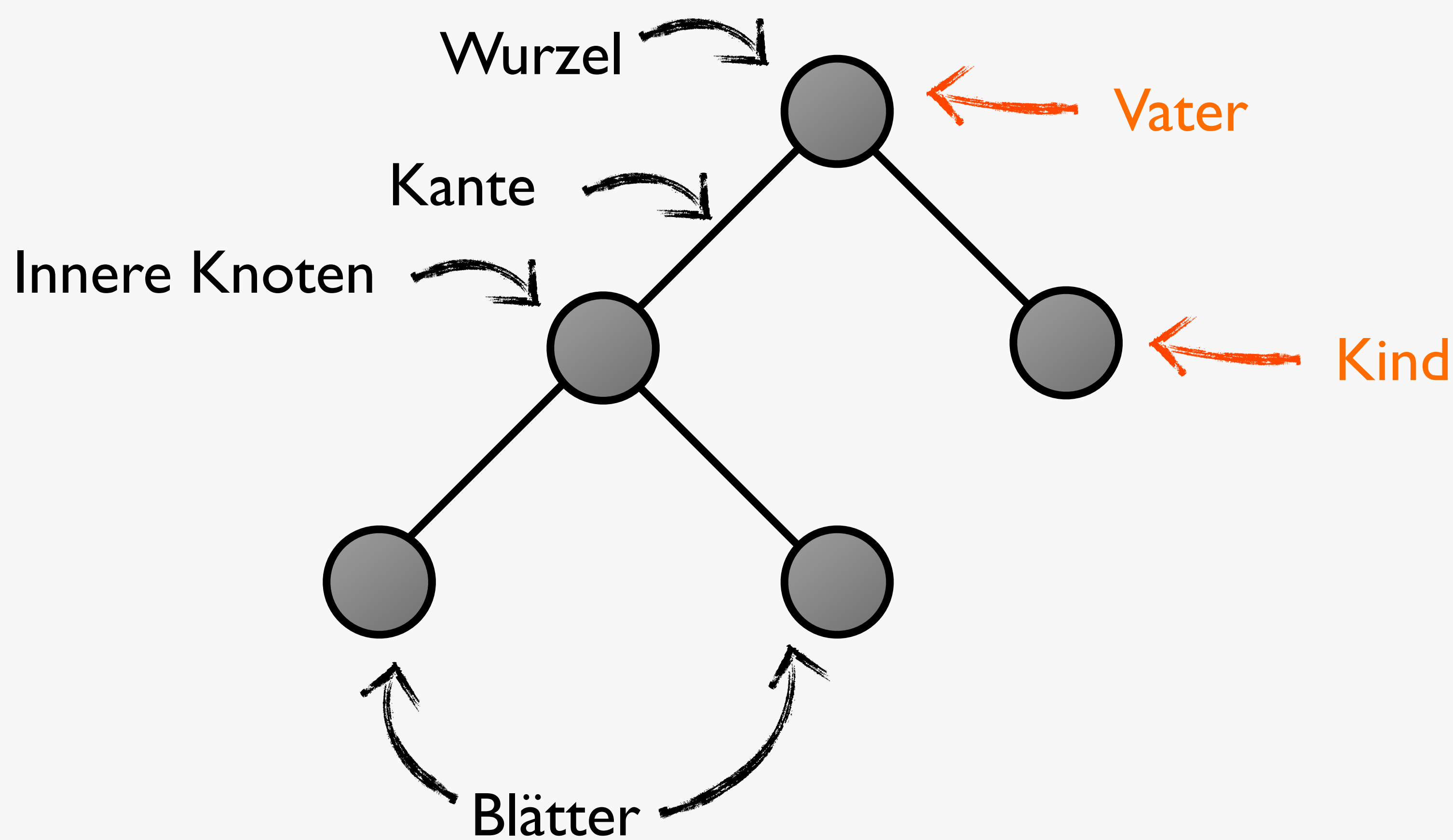
- Bäume
  - Binärbäume
    - erweiterte Binärbäume
    - binäre Suchbäume
  - Rose Trees (n-äre Bäume)

Bäume Binärbäume erweiterte

Bäume bestehen aus Knoten- und Kantenmenge

G = (V,E) mit |V| = n Knoten und |E| = m Kanten

Hat folgende Eigenschaft: G ist azyklisch & es gilt $m = n - 1$



Bäume

Binärbäume

erweiterte Binärbäume

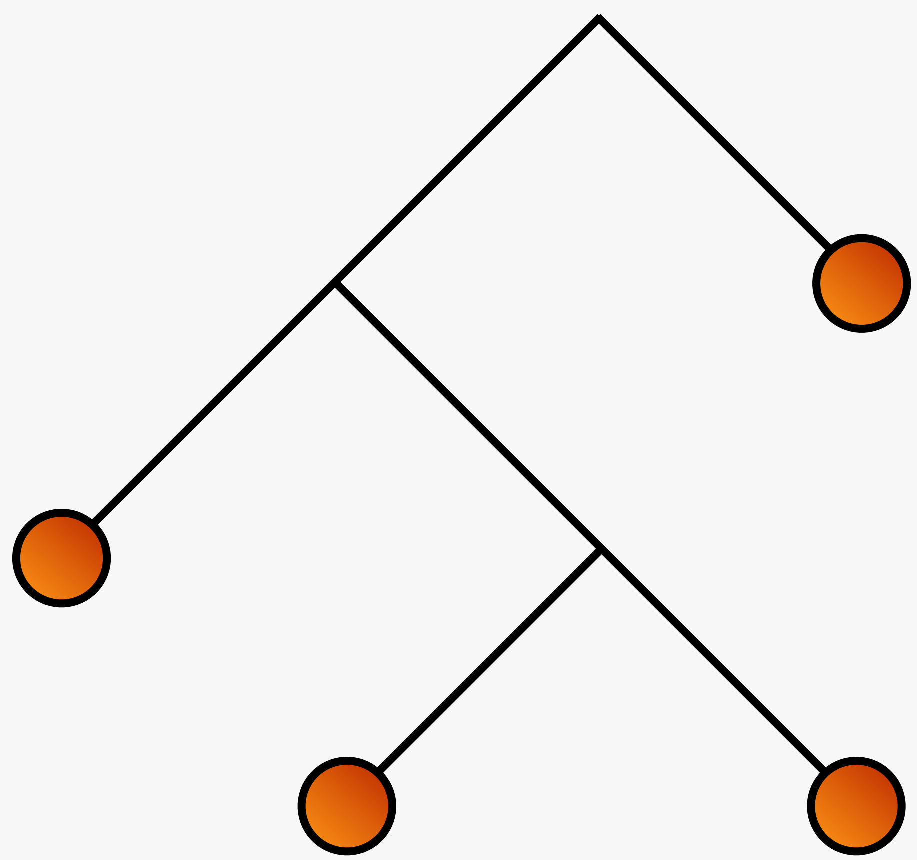
binäre Suchbäume

Rose Trees (n-äre Bäume)

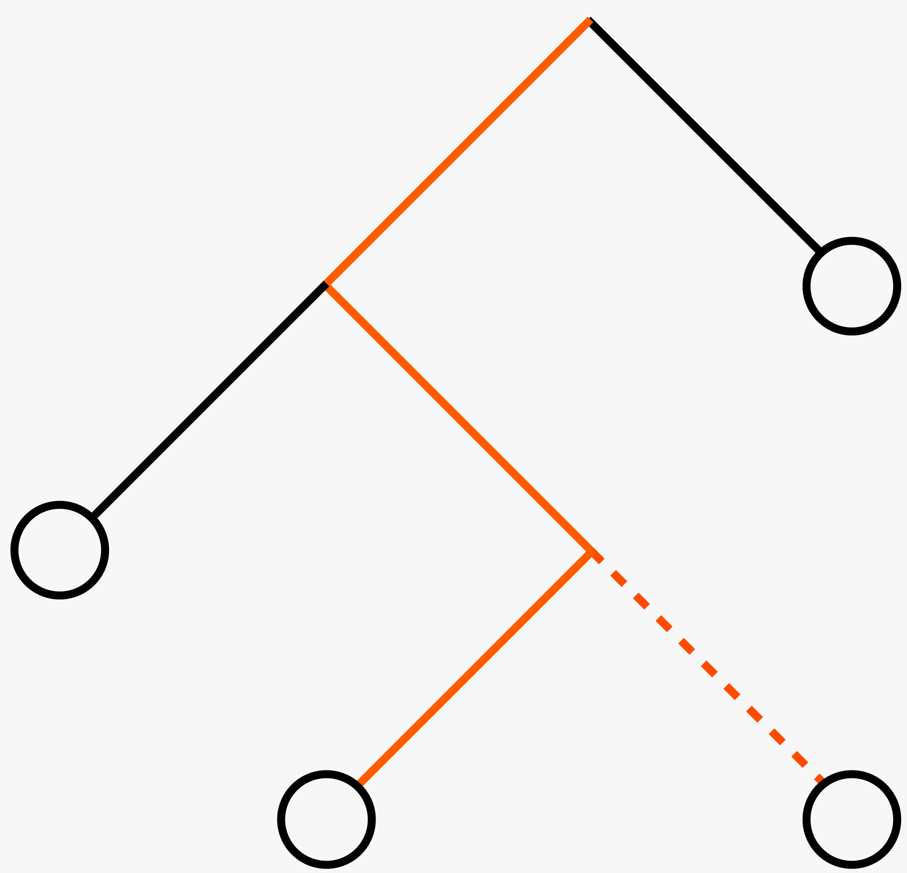
Binärbaum besteht aus einem Blatt oder aus einer Verzweigung, die wiederum zwei Binärbäume enthält.

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

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

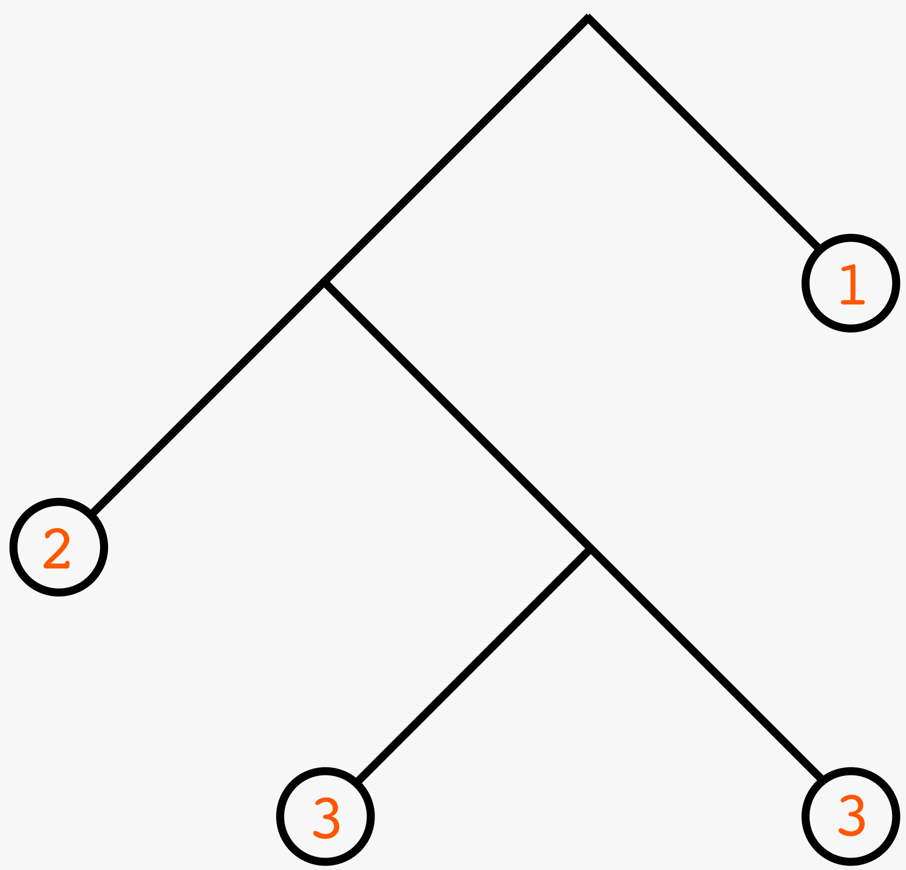


```
height                :: BTree a -> Int
height (Leaf x)       =  0
height (Fork xt yt) =  1 + (max (height xt) (height yt))
```

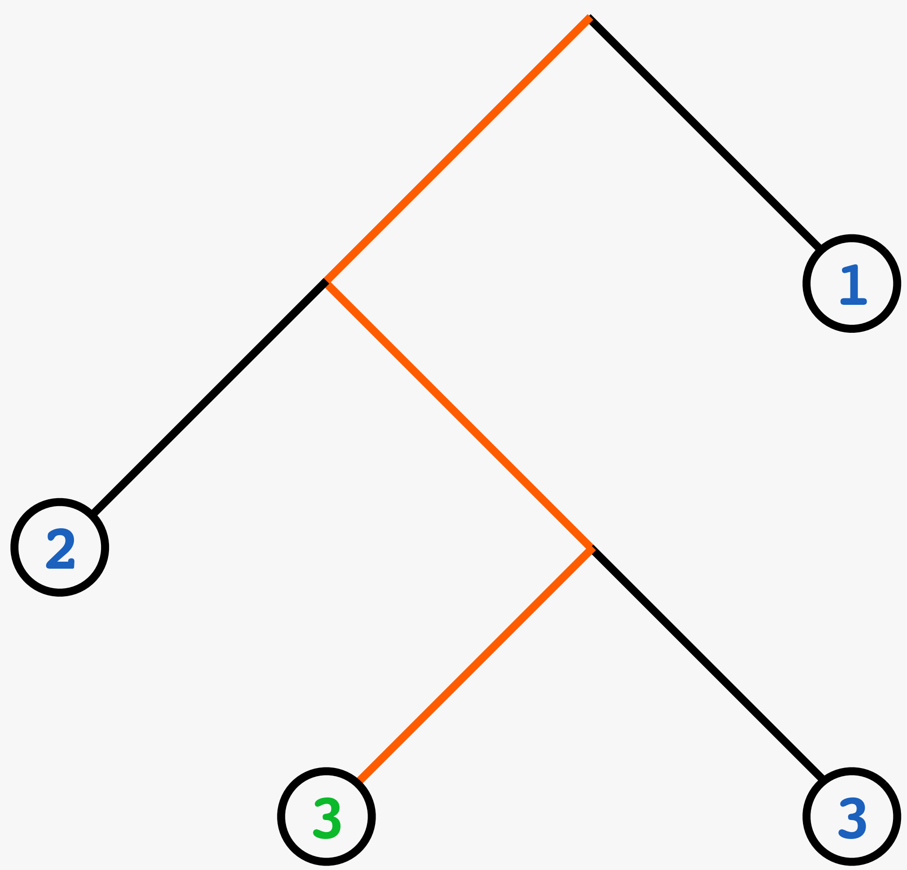


```
depths                :: BTree a -> BTree Int
depths                =  down 0

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)
```

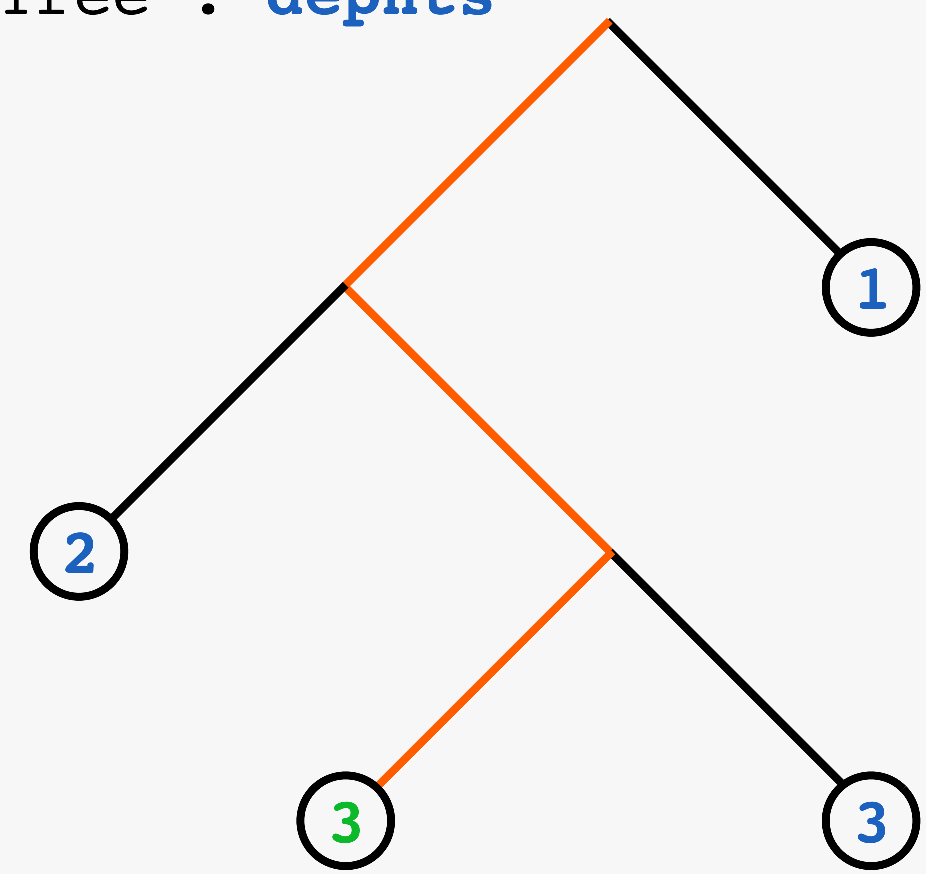


```
maxBTree                  :: (Ord a) => BTree a -> a
maxBTree (Leaf x)         = x
maxBTree (Fork xt yt) = (maxBTree xt) `max` (maxBTree yt)
```

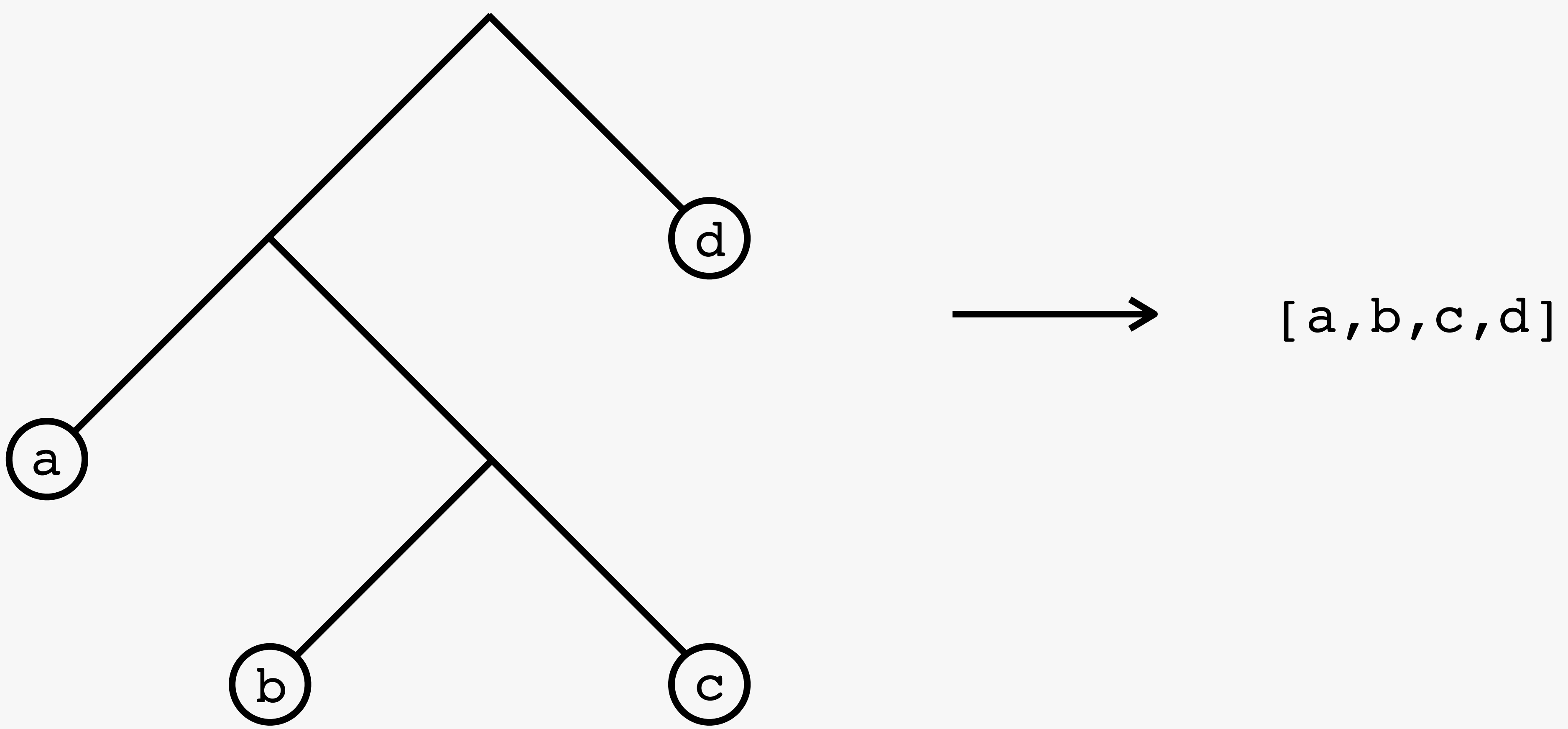


```
maxBTree                :: (Ord a) => BTree a -> a
maxBTree (Leaf x)       = x
maxBTree (Fork xt yt) = (maxBTree xt) `max` (maxBTree yt)
```

```
height = maxBTree . dephts
```

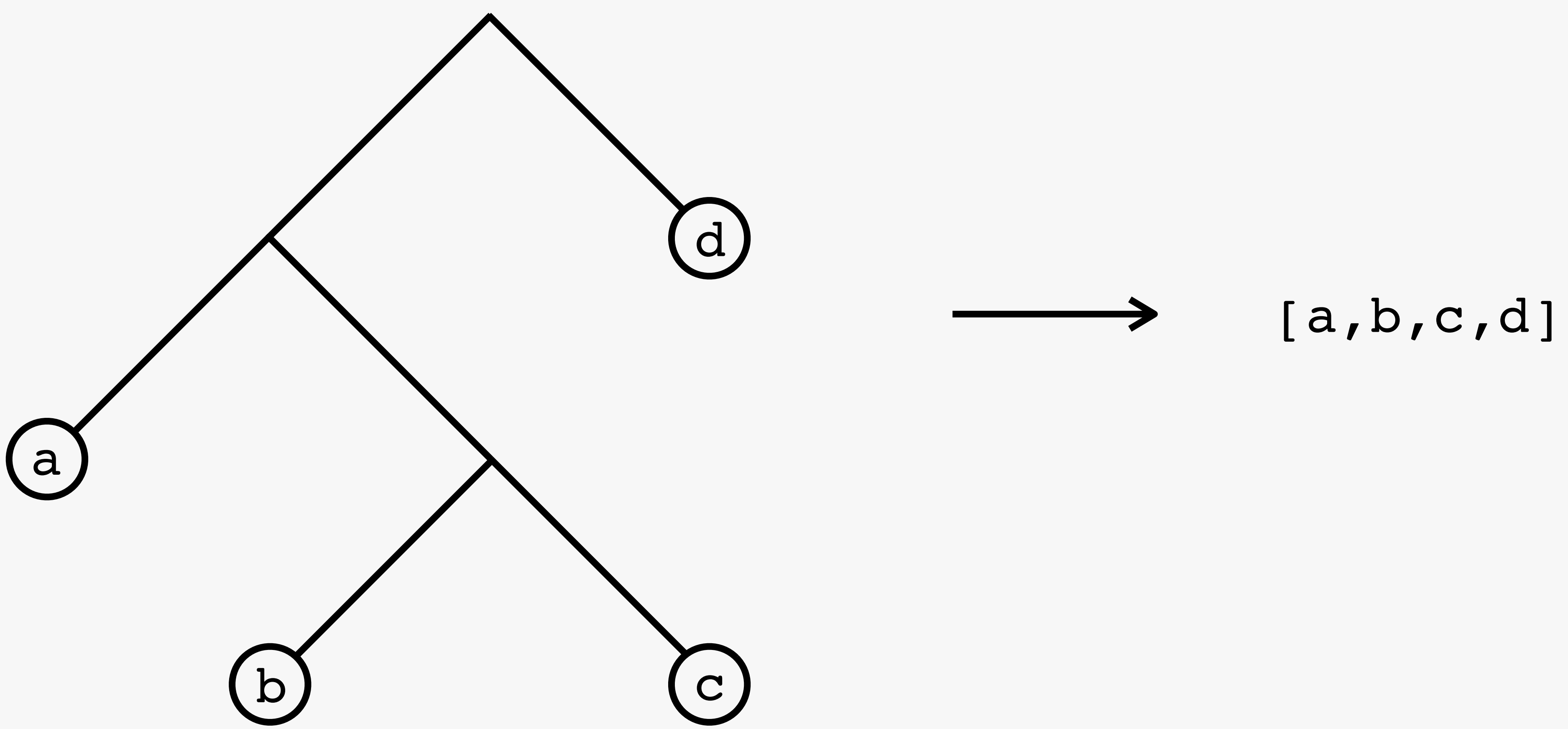


```
flatten                :: BTree a -> [a]
flatten (Leaf x)       =  [x]
flatten (Fork xt yt) =  flatten xt ++ flatten yt
```



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

```
size = lenght . flatten
```



Zusammenhänge:

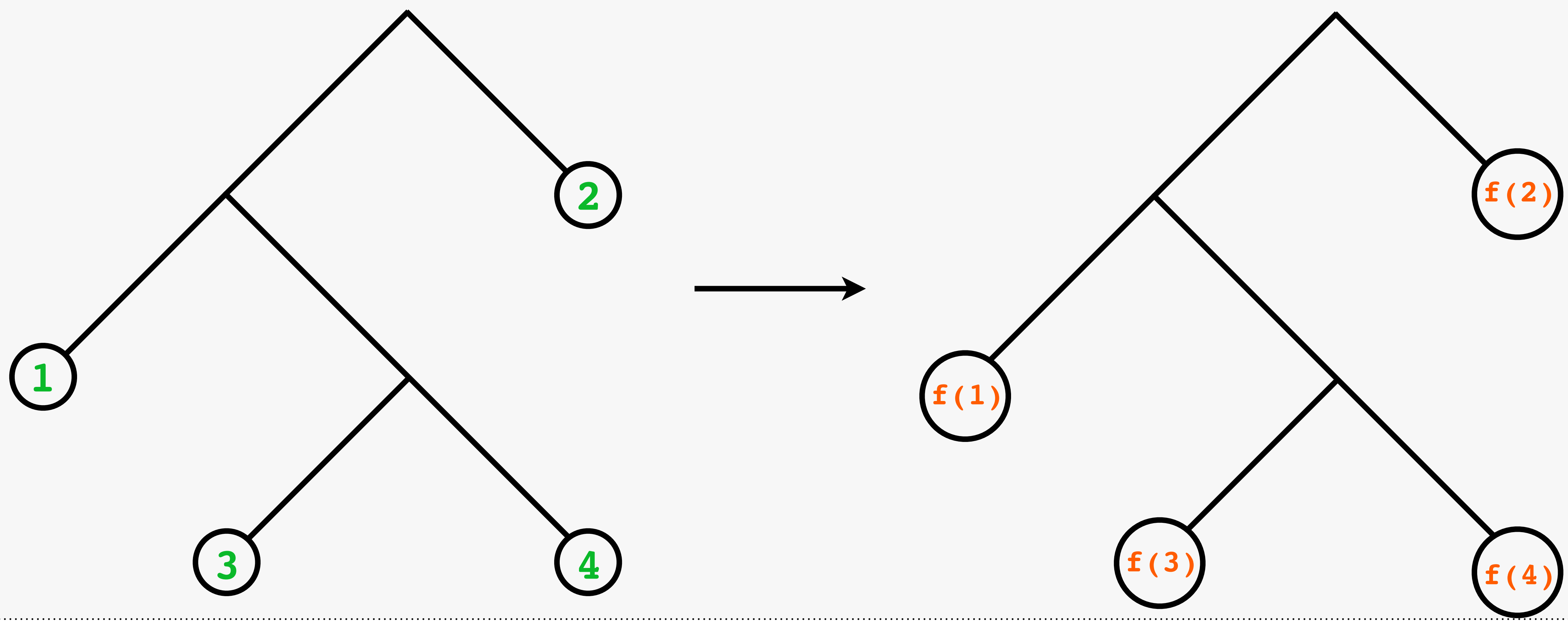
generell: $\log_2$ (size xt) $\leq$ height xt $<$ size xt

hier: $\log_2$ (length xs) – height xt minimal

```
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

-- mkBTree . flatten xt     (xt in Baum mit minimaler Höhe konvertieren)
```

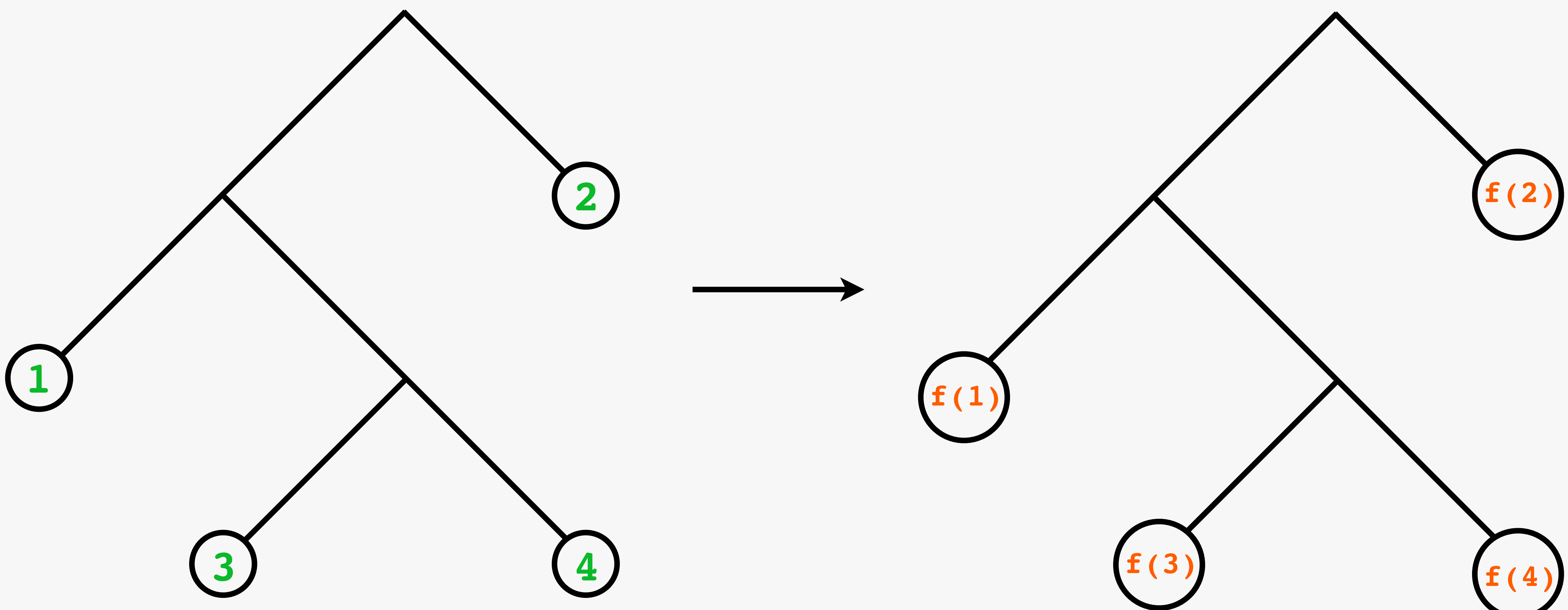
```
mapBTree                  :: (a -> b) -> BTree a -> BTree b
mapBTree f (Leaf x)       =  Leaf (f x)
mapBTree f (Fork xt yt)   =  Fork (mapBTree f xt) (mapBTree f yt)
```



```
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)
```

```
mapBTree id      = id;
mapBTree (f . g) = mapBTree f . mapBTree g
```

```
map f . flatten = flatten . mapBTree f
```



```
Leaf :: a -> BTree a
Fork :: BTree a -> BTree a -> BTree a

f    :: a -> b
g    :: b -> b -> b
```

foldBtree muss beide Konstruktoren behandeln, daher benötigen wir zwei Funktionen

```
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)
```

```
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)
```

```
size   = foldBTree (const 1) (+)
height = foldBTree (const 0) func
         where func xt yt = 1 + (max xt yt)
```

- Bäume
  - Binärbäume
    - erweiterte Binärbäume
    - binäre Suchbäume
  - Rose Trees (n-äre Bäume)

Speichern zusätzlich Information in den inneren Knoten.

Vorteil: Damit ist es möglich effizientere Algorithmen zu entwickeln

```
data ATree a = Leaf a
             | Fork Int (ATree a) (ATree a)
```

Idee: Innerer Knoten speichert Gesamtgröße des linken Teilbaumes

Vorraussetzung: Angegebene Gesamtgröße muss stimmen

Lösung: Erzeugen von Fork-Nodes durch Funktion fork

```
fork         :: ATree a -> ATree a -> ATree a
fork xt yt =  Fork (lsize xt) xt yt

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

Statt Konstruktor Fork wird jetzt Funktion fork zum Erzeugen des erweiterten Binärbaums genutzt.

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

Ähnlich zum indexierten Zugriff auf Listen mit !! Operator soll nun auch auf die Blätter eines erweiterten Binärbaums zugegriffen werden.

Daraus ergibt sich folgende Spezifikation:

```
retrieve        :: ATree a -> Int -> a
retrieve xt k =  (flatten xt) !! k
```

Unter Ausnutzung der Eigenschaft (1) lässt sich ein effizienter Algorithmus zum indexierten Zugriff auf Bäume realisieren (2).

(k – m) im else-Zweig, da sich die m Knoten im linken Teilbaum befinden.

```
(1)  (xs ++ ys) !! k = if k < m then xs !! k else ys !! (k - m)
                       where m = length xs


(2)  retrieve                      :: ATree a -> Int -> a
     retrieve (Leaf x) 0           =  x
     retrieve (Fork m xt yt) k =
         if k< m then retrieve xt k else retrieve yt (k - m)
```

erweiterte Binärbäume

Unter Ausnutzung der Eigenschaft (1) lässt sich ein effizienter Algorithmus zum indexierten Zugriff auf Bäume realisieren (2).

(k – m) im else-Zweig, da sich die m Knoten im linken Teilbaum befinden.

```
(1)   (xs ++ ys) !! k = if k < m then xs !! k else ys !! (k - m)
                        where m = length xs


(2)   retrieve                          :: ATree a -> Int -> a
      retrieve (Leaf x) 0               =  x
      retrieve (Fork m xt yt) k =
          if k< m then retrieve xt k else retrieve yt (k - m)
```

Bäume

- Binärbäume
  - erweiterte Binärbäume
  - binäre Suchbäume
- Rose Trees (n-äre Bäume)

Auch „labelled binary trees“ genannt.

Eigent sich für effiziente Suche.

Blätter sind leer, Informationen stecken in den Knoten.

```
data (Ord a) => STree a = Null
                        | Fork (STree a) a (STree a)
```

Eigenschaft eines binären Suchbaums:

flatten liefert eine Liste, deren Elemente aufsteigend sortiert sind.

Es gilt: xt < x < yt.

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

```
member :: 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
```

Im schlechtesten Fall ist die Laufzeit proportional zur Höhe des Baumes.

Beziehung zwischen Höhe und Größe:

$$\text{height xt} \leq \text{size xt} < 2^{\text{height xt}}$$

```
mkSTree          :: (Ord a) => [a] -> STree a
mkSTree []       =  Null
mkSTree (x:xs)   =  Fork (mkSTree yt) x (mkSTree zt)
                    where (yt,zt) = partition (<= x) xs

partition        :: (Ord a) => (a -> Bool) -> [a] -> ([a],[a])
partition p xs   =  (filter p xs, filter (not . p ) xs)
```

Mit dieser Implementierung ist nicht sichergestellt, dass Suchbaum minimale Höhe hat.

Im schlechtesten Fall gilt:

size xt = height xt

```
mkSTree          :: (Ord a) => [a] -> STree a
mkSTree []       =  Null
mkSTree (x:xs)   =  Fork (mkSTree yt) x (mkSTree zt)
                    where (yt,zt) = partition (<= x) xs

partition        :: (Ord a) => (a -> Bool) -> [a] -> ([a],[a])
partition p xs   =  (filter p xs, filter (not . p ) xs)

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

Mit dieser Implementierung ist nicht sichergestellt, dass Suchbaum minimale Höhe hat.

Im schlechtesten Fall gilt:

size xt = height xt

```
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 x yt
    | (x > y)  = Fork xt y (insert x yt)
```

Werte, die bereits im Baum gespeichert sind werden ersetzt.

Idee: Beim Löschen entstehen zwei Subtrees. Diese müssen zusammengefügt werden. Der rechte Subtree wird beim linken Subtree an den rechtesten leeren Ast gehängt.

```
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)

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)
```

Problem: Es können Bäume entstehen, die höher sind, als nötig.

Dies führt zu einer ineffektiven Suche.

```
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)

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)
```

Verbesserte Version von Join

Lösung: Suche im rechten Subtree das Element, welches am weitesten links ist und nutze dies als "Verbindungsknoten“

```
join          :: (Ord a) => STree a -> STree a -> STree a
join xt yt =
    if empty yt then xt else Fork xt ((fst.splitT) yt) ((snd.splitT) yt)

empty                    :: (Ord a) -> STree a -> Bool
empty Null               =  True
empty (Fork xt x yt) =  False

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

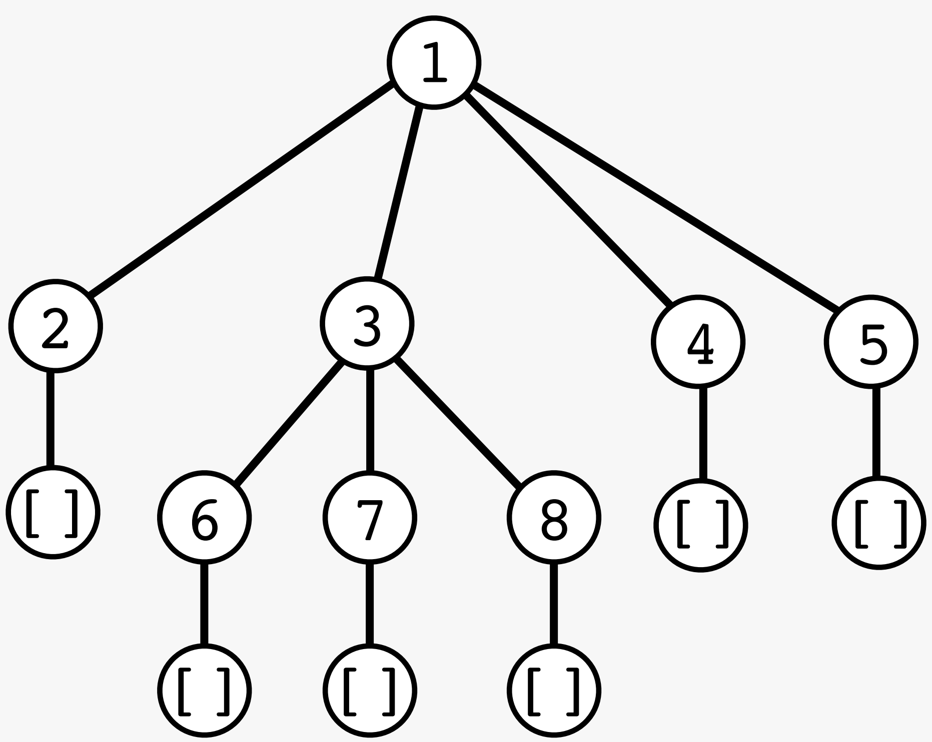
- Bäume
  - Binärbäume
    - erweiterte Binärbäume
    - binäre Suchbäume
  - Rose Trees (n-äre Bäume)

rose trees

Besteht aus Labelled Nodes und einer Liste weiterer Subtrees.

Externe Knoten haben keine Subtrees. Interne Knoten haben mindestens einen Subtree

```
data RTree a = Node a [RTree a]
```



rose trees

Endliche Bäume: Jeder Knoten hat eine endliche Anzahl von Subknoten.

⚠ Endliche Bäume können unendliche Größe (Tiefe) haben

```
roseFinite :: RTree Int
roseFinite =  Node 0 [roseFinite]

roseInfinite :: RTree Int
roseInfinite =  Node 0 [Node n [] | n <- [1..]]
```

rose trees

Ein Baum hat die Breite w, wenn alle Knoten des Baums nicht mehr als w unmittelbare Subtrees haben.

Alle endlichen Bäume haben eine endliche Breite.

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

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

rose trees


```
depths :: RTree a -> RTree Int
depths =  down 1


down                 :: Int -> RTree a -> RTree Int
down n (Node x xts)  =  Node n (map (down (n+1)) xts)


maxRose              :: (Ord a, Num a) => RTree a -> a
maxRose (Node x xts) =  x `max` maxlist (map maxRose xts)
                        where maxlist = foldl (max) 0
```

rose trees

Im Gegensatz zu BTree existiert in RTree nur ein Konstruktor, daher wird nur eine Ersatz-Funktion benötigt

```
Node :: a -> [RTree a] -> RTree a
f    :: a -> [b] -> b


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

rose trees

Jetzt können wir size und maxRose über foldRose realisieren

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


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

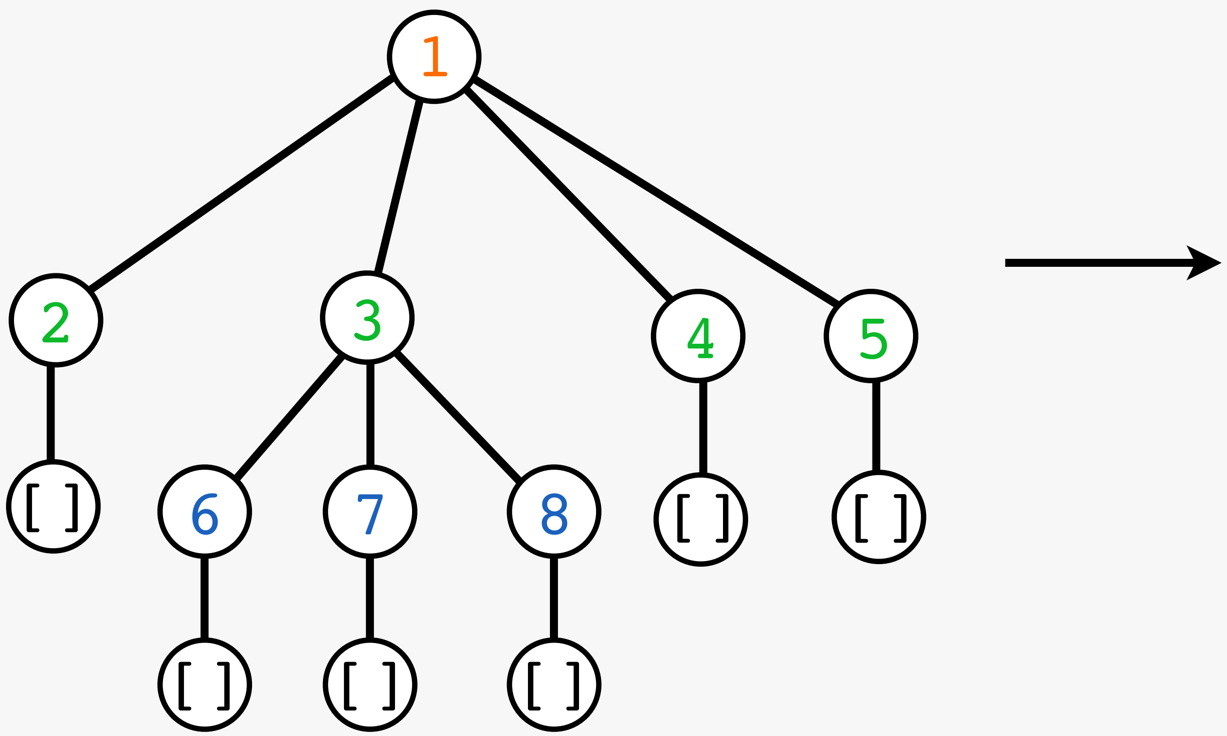
rose trees

... ebenso kann jetzt *map* für RoseTrees über foldRose realisiert werden

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

rose trees


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



[1,2,3,6,7,8,4,5]

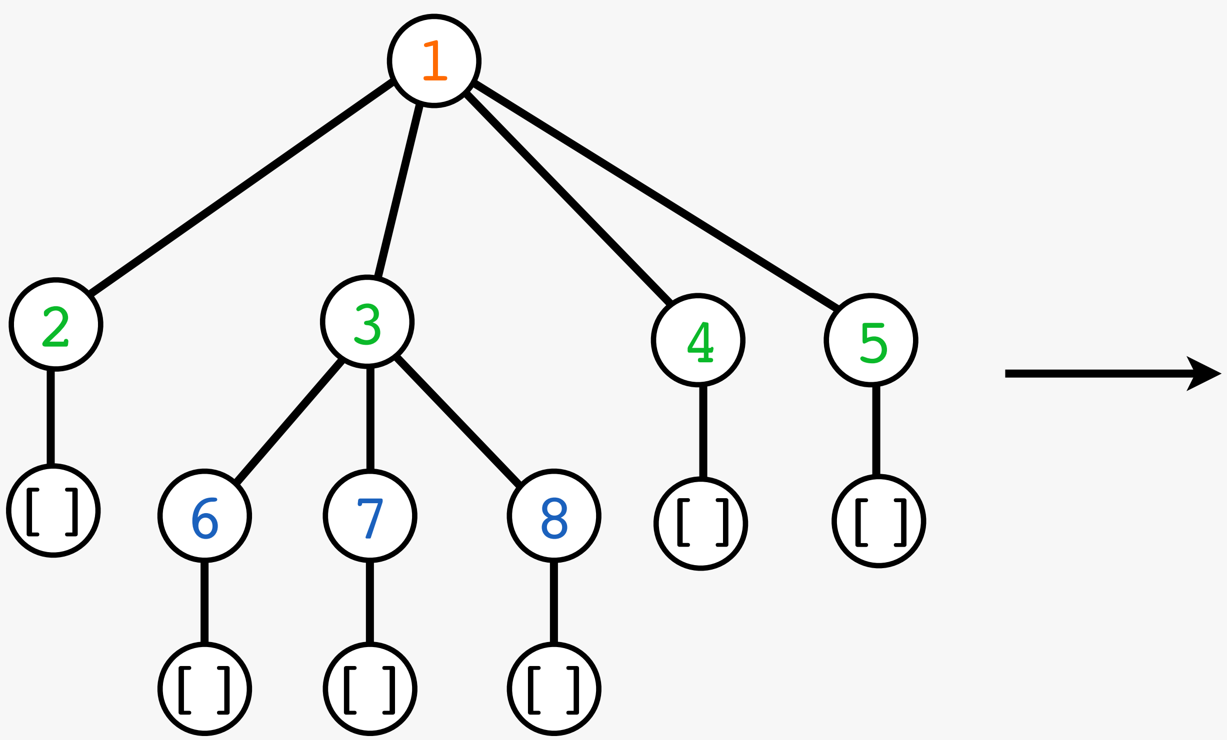
Problem: Depth-first order.

Problem ähnlich KI der Tiefensuche. Ist beispielsweise jeweils der linke äußere Subtree unendlich tief, so gibt der Algorithmus nur den Pfad entlang des Gabelung links außen aus.

rose trees

Lösung: Breadth-first order.

Baum wird Level für Level abgearbeitet



[1,2,3,4,5,6,7,8]

rose trees


```
level :: RTree a -> [a]
level =  concat . levels

levels                :: RTree a -> [[a]]
levels (Node x xts) =  [x] : combine (map levels xts)

combine :: [[[a]]] -> [[a]]
combine = foldr (##) []

(##)                    :: [[a]] -> [[a]] -> [[a]]
[] ## yss               =  yss
(xs:xss) ## []          =  xs:xss
(xs:xss) ## (ys:yss) =  (xs ++ ys) : (xss ## yss)
```

rose trees


```
roseFinite :: RTree Int
roseFinite =  Node 0 [roseFinite, RTree 1]

> elem 1 (flatten roseFinite)
> elem 1 (level roseFinite)
```

Der erste Funktionsaufruf würde unendlich tief in den linken Sub-Tree absteigen, während der zweite Funktionsaufruf unmittelbar auf der 2. Ebene das gesuchte Element findet thanks grazie شكر danke 謝謝 tak ευχαριστώ merci

Alexander Meirowski & Patrick Schmidt

Bei: Prof. Dr. Schmidt, FH Wedel
Freitag, 26. November 2010