# Programmierung und Modellierung mit Haskell

## Teil 7: Bäume und Funktoren

Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

16. Mai 2018

# Teil 7: Bäume und Funktoren

1. **Binärbäume**
   - Binärbäume in der Mathematik
   - Binärbäume in Haskell
2. **Funktoren**
   - Motivation
   - Kinds
   - Typklasse Funktor
3. **Monoide**
   - Halbgruppen
   - Monoide
4. **Algorithmen auf Bäumen**
   - Linearisierung von Bäumen
   - Typklasse Foldable
   - Generischer Tiefendurchlauf
   - Breitendurchlauf
   - Arithmetische Ausdrücke
   - Parsing
5. **Laufzeitbetrachtungen**
   - Listen
   - Warteschlange
   - Binäre Suchbäume

# Binärbäume

- Ist $A$ eine Menge, so definieren wir die Menge $A^\triangle$ der Binärbäume mit Knotenmarkierungen aus $A$ (auch: **Binärbäume über $A$**) induktiv durch:
  1. Der leere Baum $\varepsilon$ ist in $A^\triangle$.
  2. Sind $l$ und $r$ in $A^\triangle$ und $x$ in $A$, so ist das Tripel $(x, l, r)$ in $A^\triangle$.
- Alternativ kann man die Mengen $A_n^\triangle$ der **Binärbäume mit Höhe (Tiefe) kleiner $n$** rekursiv definieren durch:
  1. $A_0^\triangle = \{\,\}$ kein Baum hat negative Höhe
  2. $A_{n+1}^\triangle = \{\varepsilon\} \cup \{(x, l, r) \mid x \in A,\ l, r \in A_n^\triangle\}$.
- Man beweist leicht $A_n^\triangle \subseteq A_{n+1}^\triangle$ (Kumulativität).
- $A^\triangle = \bigcup_{n \in \mathbb{N}} A_n^\triangle$ ist der Limes der Folge $A_0^\triangle, A_1^\triangle, \ldots$
- Die **Höhe (Tiefe)** eines Baums $t$ ist das kleinste $n$, so dass $t \in A_{n+1}^\triangle$.

# Binärbäume

- Ist $A$ eine Menge, so definieren wir die Menge $A^\triangle$ der Binärbäume mit Knotenmarkierungen aus $A$ (auch: **Binärbäume über $A$**) induktiv durch:
  1. Der leere Baum $\varepsilon$ ist in $A^\triangle$.
  2. Sind $l$ und $r$ in $A^\triangle$ und $x$ in $A$, so ist das Tripel $(x, l, r)$ in $A^\triangle$.
- Alternativ kann man die Menge $A_n^\triangle$ der **Binärbäume mit Höhe (Tiefe) kleiner $n$** rekursiv definieren durch:

  Höhe

Den Satz *„ein Binärbaum ist leer, oder ein Tripel aus einem $x \in A$ und zwei Binärbäumen“* können wir direkt in eine Datentypdeklaration übersetzen:

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

- $A^\triangle = \bigcup_{n\in\mathbb{N}} A_n^\triangle$ ist der Limes der Folge $A_0^\triangle, A_1^\triangle, \ldots$
- Die **Höhe (Tiefe)** eines Baums $t$ ist das kleinste $n$, so dass $t \in A_{n+1}^\triangle$.

# Terminologie für Bäume

- $x$ heißt **Wurzel(-beschriftung)** des Baumes $(x, l, r)$.
- $l$ heißt **linker** und $r$ **rechter Unterbaum** von $(x, l, r)$.
- $y$ heißt **linker Kind**-knoten des **Elternknoten** $x$ für $(x, (y, ll, lr), r)$.
- Ein Binärbaum der Form $(x, \varepsilon, \varepsilon)$ heißt **Blatt**.
- $\varepsilon$ heißt **leerer Baum**, jeder andere Baum ist **nichtleer**.
- Die **Knoten(-markierungen)** und die **Teilbäume** eines Binärbaums sind rekursiv definiert:
  1. Der leere Baum $\varepsilon$ hat keine Knoten und nur sich selbst als Teilbaum.
  2. Die Knoten von $(x, l, r)$ sind $x$ plus die Knoten von $l$ und $r$. Die Teilbäume von $(x, l, r)$ sind $(x, l, r)$ plus die Teilbäume von $l$ und $r$.

Beispiel Sei $\langle x \rangle$ definiert als Abkürzung für ein Blatt $(x, \varepsilon, \varepsilon)$ und

$$t = (6, (3, \langle 2 \rangle, (8, \langle 5 \rangle, \varepsilon)), (8, \varepsilon, \langle 4 \rangle))$$

dann hat $t$ die Knoten $\{2, 3, 5, 8, 6, 4\}$ und Teilbäume
$\{t,\ ((3, \langle 2 \rangle, (8, \langle 5 \rangle\, \varepsilon)),\ (8, \varepsilon, \langle 4 \rangle)),\ (8, \langle 5 \rangle, \varepsilon),\ \langle 2 \rangle,\ \langle 4 \rangle,\ \langle 5 \rangle,\ \varepsilon\}$.

# Terminologie für Bäume

- $x$ heißt **root** des Baumes $(x, l, r)$.
- $l$ heißt **left** und $r$ **right subtree** von $(x, l, r)$.
- $y$ heißt **left child** des **parent** $x$ für $(x, (y, ll, lr), r)$.
- Ein Binärbaum der Form $(x, \varepsilon, \varepsilon)$ heißt **leaf**.
- $\varepsilon$ heißt **empty tree**, jeder andere Baum ist **nonempty**.
- Die **labels** und
  die **subtees** eines Binärbaums sind rekursiv definiert:
  1. Der leere Baum $\varepsilon$ hat keine Knoten und nur sich selbst als Teilbaum.
  2. Die Knoten von $(x, l, r)$ sind $x$ plus die Knoten von $l$ und $r$. Die Teilbäume von $(x, l, r)$ sind $(x, l, r)$ plus die Teilbäume von $l$ und $r$.

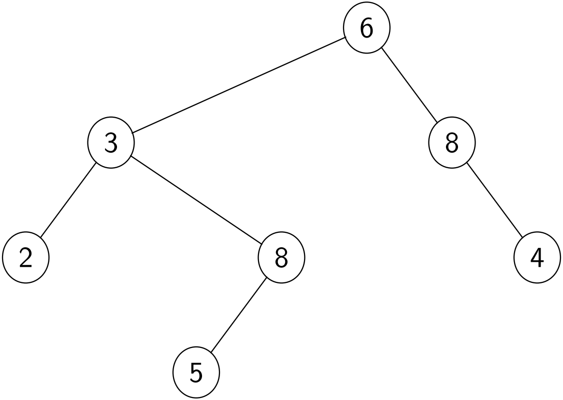
Beispiel Sei $\langle x \rangle$ definiert als Abkürzung für ein Blatt $(x, \varepsilon, \varepsilon)$ und

$$t = (6, (3, \langle 2 \rangle, (8, \langle 5 \rangle, \varepsilon)), (8, \varepsilon, \langle 4 \rangle))$$

dann hat $t$ die Knoten $\{2, 3, 5, 8, 6, 4\}$ und Teilbäume
$\{t,\ ((3, \langle 2 \rangle, (8, \langle 5 \rangle \varepsilon)),\ (8, \varepsilon, \langle 4 \rangle)),\ (8, \langle 5 \rangle, \varepsilon),\ \langle 2 \rangle,\ \langle 4 \rangle,\ \langle 5 \rangle,\ \varepsilon\}$.

# Typische Darstellung eines Binärbaumes

Sei $t = (6, (3, \langle 2 \rangle, (8, \langle 5 \rangle, \varepsilon)), (8, \varepsilon, \langle 4 \rangle))$ mit $\langle x \rangle := (x, \varepsilon, \varepsilon)$.
Dann wäre die übliche graphische Darstellung von $t$ wie folgt:




- Wurzel: 6
- Blätter: $2, 5, 4$
- Elternknoten 3 hat rechtes Kind 8

# Binärbäume in Haskell 

```
data Tree a = Empty | Node a (Tree a) (Tree a)

leaf :: a -> Tree a
leaf a = Node a Empty Empty
```

oder äquivalent mit Record Syntax (Folie 4.18), wodurch automatisch nützliche partielle Projektionen definiert werden:

```
data Tree a = Empty
            | Node { label :: a,  left,right :: Tree a }

> let t= Node 6 (Node 3 (leaf 2) (Node 8 (leaf 5) Empty))
                (Node 8 Empty (leaf 4))
> :type left
left :: Tree a -> Tree a
> :type label
label :: Tree a -> a
> label (left t)
3
```

# Darstellung als String

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

## Darstellung eines Baumes als String:

Durch `deriving Show` wird ungefähr folgender Code eingefügt:

```
instance (Show a) => Show (Tree a) where
  show (Empty)       = "Empty"
  show (Node x l r) = "Node " ++ (show x)
                         ++ " (" ++ show l ++ ") "
                         ++  "(" ++ show r ++ ")"
```

*Beachte:* Damit wir den Ausdruck `(show x)` ausführen können, müssen wir wissen, dass `x` ein Typ aus der Typklasse `Show` ist!

D.h., dies ist wieder eine überladene Instanzdeklaration: *Wenn* `a` *in der Typklasse* `Show` *ist, dann ist auch* `Tree a` *in der Typklasse* `Show`.

# Rekursion auf Bäumen

```
data Tree a = Empty | Node {label::a, left,right::Tree a}
```

## Höhe eines Baumes:

```
height :: Tree a -> Integer
height (Empty)       = 0
height (Node _ l r) = 1 + max (height l) (height r)
```

## mapTree: Eine Funktion `f` auf alle Beschriftungen anwenden

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree _ (Empty)          = Empty
mapTree f (Node x l r)     = let x' = f x
                                 l' = mapTree f l
                                 r' = mapTree f r
                             in Node x' l' r'
```

- *Beachte:* `mapTree f t` erzeugt einen komplett *neuen* Baum!
- `mapTree (\x -> x) t` liefert Kopie von `t`. Bringt nichts!

# Verallgemeinerung von map

Wir haben kennengelernt:

```
map     :: (a -> b) ->     [a]->     [b]
mapTree :: (a -> b) -> Tree a -> Tree b
```

In beiden Fällen wenn wir eine Funktion auf alle Inhalte einer Datenstruktur an, ohne die Datenstruktur selbst zu verändern. (Länge der Liste bleibt gleich, Baumstruktur bleibt gleich, etc.)

## Fragen:

- Können wir dies verallgemeinern?
- Können wir den Namen `map` überladen?

# Verallgemeinerung von map

Wir haben kennengelernt:

```
mapList :: (a -> b) -> List a -> List b
mapTree :: (a -> b) -> Tree a -> Tree b
```

In beiden Fällen wenn wir eine Funktion auf alle Inhalte einer Datenstruktur an, ohne die Datenstruktur selbst zu verändern. (Länge der Liste bleibt gleich, Baumstruktur bleibt gleich, etc.)

## Fragen:

- Können wir dies verallgemeinern?
- Können wir den Namen `map` überladen?

# Verallgemeinerung von map

Wir haben kennengelernt:

```
mapList :: (a -> b) -> List a -> List b
mapTree :: (a -> b) -> Tree a -> Tree b
```

In beiden Fällen wenn wir eine Funktion auf alle Inhalte einer Datenstruktur an, ohne die Datenstruktur selbst zu verändern. (Länge der Liste bleibt gleich, Baumstruktur bleibt gleich, etc.)

Fragen:

- Können wir dies verallgemeinern?
- Können wir den Namen `map` überladen?

Antwort: Natürlich! Mit Hilfe einer Typklasse!

Diese Typklasse muss jedoch anstelle eines Typen (wie `Tree Int`) mit Typkonstruktoren (wie `Tree`) arbeiten!

# Typparameter WDH

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

*Wir erinnern uns:* `Tree` ist kein Typ, aber `Tree Int` ist ein Typ!

`Tree` ist „ein Typ mit einem Loch drin“, d.h. `Tree` ist ein Typkonstruktor.
`Tree` bildet Typen auf Typen ab, z.B. Typ `Int` auf den Typ der Binärbäume mit ganzzahligen Knotenbeschriftungen: `Tree Int`.

Weitere Beispiele: `Maybe`, `Either`, `Map`, `[ ]` (Listen), ...
`Map` aus Modul `Data.Map` (Folie 5.45)

Tipp: Mit dem GHCI-Befehl `:kind` kann man sich anzeigen lassen, wie viele Typparameter benötigt werden:

```
> :kind Tree
Tree :: * -> *                       -- 1 Typparameter
> :kind Data.Map.Map
Data.Map.Map :: * -> * -> *          -- 2 Typparameter
```

"Kinds" sind quasi "Typen" von Typkonstruktoren

# Kind

Der **Kind** (engl. für Sorte) eines Typen beschreibt die *Art des Typen*, also die Anzahl und Art der Argumente eines Typkonstruktors.

Ein Kind ist entweder `*` oder aus zwei Kinds per Pfeil zusammengesetzt:

$$\kappa ::= * \mid \kappa \to \kappa$$

`(*)` steht für alle konkreten Datentypen, z.B. `Int`, `Bool`, `Double` und auch `[Int]`, `Maybe Bool`, `Either String Double` und auch vollständige Funktionstypen wie `Int -> Int`

`(* -> *)` steht für alle Typkonstruktoren mit genau einem Argument, z.B. `[]`, `Maybe` und auch `Map String`.

`(* -> (* -> *))` steht für alle Typkonstruktoren mit genau zwei Argumenten, z.B. `Map`.

Wie bei Funktionstypen ist die Rechtsklammerung implizit, d.h.

`* -> (* -> *)` = `* -> * -> *` ≠ `(* -> *) -> *`

# KIND

Der **Kind** (engl. für Sorte) eines Typen beschreibt die *Art des Typen*, also die Anzahl und Art der Argumente eines Typkonstruktors.

Ein Kind ist entweder `*` oder aus zwei Kinds per Pfeil zusammengesetzt:

```
data kind = Stern | Pfeil kind kind
```

`(*)` steht für alle konkreten Datentypen, z.B. `Int`, `Bool`, `Double` und auch `[Int]`, `Maybe Bool`, `Either String Double` und auch vollständige Funktionstypen wie `Int -> Int`

`(* -> *)` steht für alle Typkonstruktoren mit genau einem Argument, z.B. `[]`, `Maybe` und auch `Map String`.

`(* -> (* -> *))` steht für alle Typkonstruktoren mit genau zwei Argumenten, z.B. `Map`.

Wie bei Funktionstypen ist die Rechtsklammerung implizit, d.h.

`* -> (* -> *)` = `* -> * -> *` ≠ `(* -> *) -> *`

# Kind

Der **Kind** (engl. für Sorte) eines Typen beschreibt die *Art des Typen*, also die Anzahl und Art der Argumente eines Typkonstruktors.

Ein Kind ist entweder `*` oder aus zwei Kinds per Pfeil zusammengesetzt:

$$\kappa ::= * \mid \kappa \to \kappa$$

- `(*)` steht für alle konkreten Datentypen, z.B. `Int`, `Bool`, `Double` und auch `[Int]`, `Maybe Bool`, `Either String Double` und auch vollständige Funktionstypen wie `Int -> Int`
- `(* -> *)` steht für alle Typkonstruktoren mit genau einem Argument, z.B. `[]`, `Maybe` und auch `Map String`.
- `(* -> (* -> *))` steht für alle Typkonstruktoren mit genau zwei Argumenten, z.B. `Map`.

Wie bei Funktionstypen ist die Rechtsklammerung implizit, d.h.

`* -> (* -> *)` $=$ `* -> * -> *` $\neq$ `(* -> *) -> *`

# Kind

GHCI kann den Kind eines Typen mit `:kind` anzeigen:

```
> :kind [Char]
[Char] :: *

> :k []
[] :: * -> *

> :k Maybe
Maybe :: * -> *

> :k Either
Either :: * -> * -> *
```

`Maybe` und `Either` haben unterschiedliche Kinds, da diese Typkonstruktoren unterschiedliche viele Parameter verlangen.

# Typklasse Functor

Im Modul `Data.Functor` findet sich folgende Definition:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Der Parameter `f` der Typklasse `Functor` steht also nicht für einen konkreten Typ wie z.B. `Tree Int`, sondern für einen Typkonstruktor wie z.B. `Tree`.
*Genauer:* `f` hat Kind `* -> *`

Die Typklasse `Functor` ist also die Klasse aller "Container"-Typen, welche es erlauben Ihre Inhalte auf andere abzubilden.

Die Deklaration ist dabei unabhängig von dem beinhalteten Typ!

# Beispiel: Functor Tree

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Bäume können wir leicht zur Typklasse `Functor` hinzufügen:

```
instance Functor Tree where
-- fmap :: (a -> b) -> Tree a -> Tree b
   fmap _ (Empty)      = Empty
   fmap f (Node a l r) = Node (f a) (fmap f l) (fmap f r)
```

## Beispiele

```
> fmap even (Node 2 (Leaf 1) (Leaf 4))
Node True (Leaf False) (Leaf True)

> fmap (*2) (Node 2 (Leaf 1) (Leaf 4))
Node 4 (Leaf 2) (Leaf 8)

> :type fmap
fmap :: Functor f => (a -> b) -> f a -> f b
```

# Beispiel: Functor Liste

Listen können wir leicht zur Typklasse `Functor` hinzufügen:

```
instance Functor [] where
--   fmap :: (a -> b) -> [a] -> [b]
     fmap = map
```

Der Typ `map :: (a -> b) -> [a] -> [b]` passt genau!

## Beispiele

```
> fmap (*2) [1..5]
[2,4,6,8,10]

> fmap even [1..5]
[False,True,False,True,False]

> map even [1..5]
[False,True,False,True,False]
```

# Beispiel: Functor Liste

Listen können wir leicht zur Typklasse `Functor` hinzufügen:

```
instance Functor [] where
--  fmap :: (a -> b) -> [a] -> [b]
    fmap _ []     = []
    fmap f (h:t) = (f h):(fmap f t)
```

Der Typ `map :: (a -> b) -> [a] -> [b]` passt genau!

## Beispiele

```
> fmap (*2) [1..5]
[2,4,6,8,10]

> fmap even [1..5]
[False,True,False,True,False]

> map even [1..5]
[False,True,False,True,False]
```

# Noch ein Dollar mehr

Die Standardbibliothek definiert auch ein Infix für Funktoren:

```
infixl 4 <$>
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

So wie man mit `f $ x` eine Funktion auf einen Wert anwendet, kann man mit `f <$> t` eine Funktion auf eine Datenstruktur anwenden:

```
> even <$> [1..5]
[False,True,False,True,False]

> even <$> t
(True,(False,<True>,(True,<False>,)),(True,,<True>))

> (+1) <$> t
(7,(4,<3>,(9,<6>,)),(9,,<5>))
```

# Beispiel: Functor Maybe

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Eine Funktion auf ein Maybe-Wert anwenden:

```
data Maybe a = Nothing | Just a

instance Functor Maybe where
  fmap g (Just x) = Just (g x)
  fmap _ Nothing  = Nothing
```

`Nothing` bleibt `Nothing`; aber auf Inhalte von `Just` wird die gegebene Funktion angewendet und das Ergebnis wieder verpackt:

```
> fmap even (Just 42)
Just True
> fmap even Nothing
Nothing
```

# Beispiel: Functor Maybe

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Eine Funktion auf ein Maybe-Wert anwenden:

```
data Maybe a = Nothing | Just a

instance Functor Maybe where
  fmap g (Just x) = Just (g x)
  fmap _ Nothing  = Nothing
```

`Nothing` bleibt `Nothing`; aber auf Inhalte von `Just` wird die gegebene Funktion angewendet und das Ergebnis wieder verpackt:

```
> even <$> (Just 42)
Just True
> even <$> Nothing
Nothing
```

# Generische Funktor Instanzen

GHC hat auch eine Erweiterung, welche `Functor`-Instanzen für viele Container-Datentypen automatisch generieren kann:

```
{-# LANGUAGE DeriveFunctor #-}

data Tree a = Leaf a | Node a (Tree a) (Tree a)
  deriving (Functor )
```

- Mit `{-# ... #-}` am Anfang der Datei kann der Kompiler beeinflusst werden. Solche speziellen Kommentar nennt man auch **Pragma**.
- Mit dem Pragma `LANGUAGE` werden Spracherweiterung gegenüber dem Haskell Standard aktiviert
- Mehrere Spracherweiterung werden mit Komma getrennt.
- `{-# LANGAUGE InstanceSigs -#}` erlaubt z.B. Typsignaturen in Instanzdeklaration

# Funktor Gesetze

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
  (<$) =  fmap . const
```

```
const::a->b->a
const  x  y = x
```

`fmap f` wendet Funktion `f` punktweise auf eine Datenstruktur an.

Gesetze Es *sollten* folgende Gesetze erfüllt werden:

1. Identität: `fmap id == id`     `id = \x -> x`
2. Komposition: `fmap f . fmap g == fmap (f . g)`

GHC prüft dies nicht, Programmierer muss dies alleine sicherstellen!

*Hinweis:* Der Begriff "Funktor" kommt aus der Mathematik: ein Funktor ist *strukturerhaltende Abbildung* zwischen zwei Kategorien

# Zusammenfassung Funktoren

Funktoren sind ein Programmierschema für die punktweise Anwendung einer Funktion auf eine Datenstruktur

- *Identität:* Funktoren verändern nie die Form einer Datenstruktur, sondern nur deren Inhalt
- *Komposition:* Es ist egal, ob wir mehrfach über die Datenstruktur gehen oder nur einmal und dabei gleich mehrere Funktionen hintereinander punktweise anwenden.

$\Rightarrow$ Änderungen durch Funktoren innerhalb einer Datenstruktur sind immer lokal und voneinander unabhängig!

# Typklasse Semigroup

Modul `Data.Semigroup` der Standardbibliothek definiert:

```
class Semigroup a where
  (<>)    :: a -> a -> a        -- gesprochen ‘mappend‘

  stimes :: Integral b => b -> a -> a
  stimes n x = foldl1 (<>) $ replicate (fromIntegral n) x
    ...
```

Gesetze Die binäre Operation `(<>)` sollte **assoziativ** sein, d.h.

`x <> (y <> z)  ==  (x <> y) <> z`

Mathematiker bezeichnen eine Menge mit einer 2-stelligen inneren Abbildung, welche assoziativ ist, als **Halbgruppe**.

Für uns kommuniziert die Typklasse die Eigenschaft eines Typs, solch eine *assoziative Operation* `a -> a -> a` zu besitzen.

# Typklasse Semigroup

Modul `Data.Semigroup` der Standardbibliothek definiert:

```
class Semigroup a where
  (<>)   :: a -> a -> a        -- gesprochen ‘mappend‘

  stimes :: Integral b => b -> a -> a
  stimes n x = foldl1 (<>) $ replicate (fromIntegral n) x
    ...
```

**Gesetze** Die binäre Operation `(<>)` sollte **assoziativ** sein, d.h.

```
x <> (y <> z)  ==  (x <> y) <> z
```

Mathematiker bezeichnen eine Menge mit einer 2-stelligen inneren Abbildung, welche assoziativ ist, als **Halbgruppe**.

Für uns heißt das einfach: `a -> a -> a`

Für uns kommuniziert die Typklasse die Eigenschaft eines Typs, solch eine *assoziative Operation* `a -> a -> a` zu besitzen.

# Typklasse Semigroup

Modul `Data.Semigroup` der Standardbibliothek definiert:

```
class Semigroup a where
  (<>)    :: a -> a -> a          -- gesprochen ‘mappend‘

  stimes :: Integral b => b -> a -> a
  stimes n x = foldl1 (<>) $ replicate (fromIntegral n) x
  ...
```

> `(+)` ist assoziativ, `(-)` ist nicht assoziativ:
>
> *Beispiel:* $(3+4)+5 = \;\; 12 \qquad = 3+(4+5)$
>
> *Gegenbeispiel:* $(3-4)-5 = -6 \quad \neq 4 = 3-(4-5)$

**Gesetze** Die binäre Operation `(<>)` sollte **assoziativ** sein, d.h.

```
x <> (y <> z)  ==  (x <> y) <> z
```

Mathematiker bezeichnen eine Menge mit einer 2-stelligen inneren Abbildung, welche assoziativ ist, als **Halbgruppe**.

Für uns kommuniziert die Typklasse die Eigenschaft eines Typs, solch eine *assoziative Operation* `a -> a -> a` zu besitzen.

# Typklasse Semigroup

Modul `Data.Semigroup` der Standardbibliothek definiert:

```
class Semigroup a where
  (<>)   :: a -> a -> a          -- gesprochen ‘mappend‘

  stimes :: Integral b => b -> a -> a
  stimes n x = foldl1 (<>) $ replicate (fromIntegral n) x
  ...
```

D.h.: Reihenfolge in der mehrere aufeinanderfolgende Anwendungen von `(<>)` ausgeführt werden ist egal.

*Nicht verwechseln:* Reihenfolge der Argumente muss aber gleich bleiben: `x <> y` $\neq$ `y <> x`

Gesetze Die binäre Operation `(<>)` sollte **assoziativ** sein, d.h.

```
x <> (y <> z)  ==  (x <> y) <> z
```

Mathematiker bezeichnen eine Menge mit einer 2-stelligen inneren Abbildung, welche assoziativ ist, als **Halbgruppe**.

Für uns kommuniziert die Typklasse die Eigenschaft eines Typs, solch eine *assoziative Operation* `a -> a -> a` zu besitzen.

# Typklasse Monoid

Modul `Data.Monoid` der Standardbibliothek definiert:

```
class Semigroup a => Monoid a where
  mempty  :: a                           -- eine Konstante

  mappend :: a -> a -> a
  mappend = (<>)                     -- immer noch assoziativ

  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

Gesetze Konstante `mempty` ist neutral zu `mappend`:

```
mempty <> x   ==   x   ==   x <> mempty
```

Mathematiker bezeichnen mit **Monoid** eine Halbgruppe mit Eins, d.h. eine Halbgruppe, welche ein neutrales Element besitzt.

# Typklasse Monoid

Modul Data.Monoid der Standardbibliothek definiert:

```
class Semigroup a => Monoid a where
  mempty  :: a                         -- eine Konstante

  mappend :: a -> a -> a
  mappend = (<>)                  -- immer noch assoziativ

  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

*Hinweis:* Monoid ist erst ab GHC 8.4.x eine Unterklasse von Semigroup. Vorher waren beide Typklassen unabhängig voneinander und die Typklasse Monoid forderte zusätzlich, dass mappend assoziativ ist.
In der Vorlesung verwenden wir GHC 8.2.2; bei Monoid-Instanzen schreiben wir `import Data.Semigroup` und `mappend = (<>)` hin, dann geht es mit beiden Versionen.

# Beispiel: Listen sind Halbgruppen

Das für uns wichtigste Monoid sind vielleicht Listen:

```
instance Semigroup [a] where
  -- (<>) :: [a] -> [a] -> [a]
  (<>)   = (++)
instance Monoid [a] where
  -- mempty :: [a]
  mempty  = []
```

Das die geforderten Gesetze gelten sollte für uns inzwischen offensichtlich sein:

```
([1,2,3] ++ [4,5,6]) ++ [7,8,9] == [1,2,3,4,5,6,7,8]
[1,2,3] ++ ([4,5,6] ++ [7,8,9]) == [1,2,3,4,5,6,7,8]

[] ++ [1,2,3] == [1,2,3]
[1,2,3] ++ [] == [1,2,3]
```

# Beispiel: Listen sind Halbgruppen

Das für uns wichtigste Monoid sind vielleicht Listen:

```
import Data.Semigroup          -- nur für GHC <8.4.x
instance Semigroup [a] where
  -- (<>) :: [a] -> [a] -> [a]
  (<>)   = (++)
instance Monoid [a] where
  -- mempty :: [a]
  mempty  = []
  mappend = (<>)               -- nur für GHC <8.4.x
```

Das die geforderten Gesetze gelten sollte für uns inzwischen offensichtlich sein:

```
([1,2,3] ++ [4,5,6]) ++ [7,8,9] == [1,2,3,4,5,6,7,8]
[1,2,3] ++ ([4,5,6] ++ [7,8,9]) == [1,2,3,4,5,6,7,8]

[] ++ [1,2,3] == [1,2,3]
[1,2,3] ++ [] == [1,2,3]
```

# Beispiel: Listen sind Halbgruppen

Das für uns wichtigste Monoid sind vielleicht Listen:

```
import Data.Semigroup          -- nur für GHC <8.4.x
instance Semigroup [a] where
  -- (<>) :: [a] -> [a] -> [a]
  (<>)   = (++)
instance Monoid [a] where
  -- mempty :: [a]
  mempty  = []
  mappend = (<>)               -- nur für GHC <8.4.x
```

Das die geforderten Gesetze gelten sollte für uns inzwischen offensichtlich sein:

```
([1,2,3] ++ [4,5,6]) ++ [7,8,9] == [1,2,3,4,5,6,7,8]
[1,2,3] ++ ([4,5,6] ++ [7,8,9]) == [1,2,3,4,5,6,7,8]

[] ++ [1,2,3] == [1,2,3]
[1,2,3] ++ [] == [1,2,3]
```

Beispiele sind aber kein Beweis!

# Beispiel: Listen sind Halbgruppen

Das für uns wichtigste Monoid sind vielleicht Listen:

```
import Data.Semigroup           -- nur für GHC <8.4.x
instance Semigroup [a] where
  -- (<>) :: [a] -> [a] -> [a]
  (<>)   = (++)
i
                                               4.x
```

Dies ist ein Beispiel für eine *nicht-kommutative* Halbgruppe, denn

$$\texttt{[1,2]++[3,4,5]} = \texttt{[1,2,3,4,5]}$$
$$\neq \texttt{[3,4,5,1,2]} = \texttt{[3,4,5]++[1,2]}$$

Das d

offensichtlich sein:

```
([1,2,3] ++ [4,5,6]) ++ [7,8,9] == [1,2,3,4,5,6,7,8]
[1,2,3] ++ ([4,5,6] ++ [7,8,9]) == [1,2,3,4,5,6,7,8]

[] ++ [1,2,3] == [1,2,3]
[1,2,3] ++ [] == [1,2,3]
```

Beispiele sind aber kein Beweis!

# Beispiel: Listen sind Halbgruppen

Das für uns wichtigste Monoid sind vielleicht Listen:

Beweis von (x++y)++z = x++(y++z) mit Induktion über die Länge von x an der Tafel ausgeführt.

```
-- (<>) :: [a] -> [a] -> [a]
(<>)    = (++)
```

Dies ist ein Beispiel für eine *nicht-kommutative* Halbgruppe, denn

```
[1,2]++[3,4,5] =[1,2,3,4,5]
               ≠[3,4,5,1,2] = [3,4,5]++[1,2]
```

Das d… offensichtlich sein:

```
([1,2,3] ++ [4,5,6]) ++ [7,8,9] == [1,2,3,4,5,6,7,8]
[1,2,3] ++ ([4,5,6] ++ [7,8,9]) == [1,2,3,4,5,6,7,8]

[] ++ [1,2,3] == [1,2,3]
[1,2,3] ++ [] == [1,2,3]
```

Beispiele sind aber kein Beweis!

FRAGE: Welchen Sinn hat es, `[]` und `(++)` neue Namen zu geben?

ANTWORT: Verallgemeinerung! `(++)` funktioniert nur auf Listen, aber `(<>)` funktioniert auf allen Halbgruppen/Monoiden. *Beispiele:*

```
> [1,2,3] <> mempty <> [4,5,6]
[1,2,3,4,5,6]

> Just [1,2,3]  <>  Just [4,5,6]
Just [1,2,3,4,5,6]
> Nothing  <>  Just [4,5,6]
Just [4,5,6]

> ([1,2,3],"abc") <> ([4,5,6],"def")
([1,2,3,4,5,6],"abcdef")
```

Nur das erste Beispiel hier würde auch mit `[]` und `(++)` gehen. Für eine Bibliothek ist das sehr nützlich: Der Nutzer entscheidet, ob er `[a]`, `Maybe [a]` oder `([a],[b])`, usw. haben möchte!

# Beispiel: Maybe als Monoid

Wenn der Inhalt ein Monoid bildet, dann bildet auch die Verpackung `Maybe` ein Monoid:

```
instance Semigroup a => Semigroup (Maybe a) where
  Nothing <> b       = b
  a       <> Nothing = a
  Just a  <> Just b  = Just (a <> b)

instance Semigroup a => Monoid (Maybe a) where
  mempty = Nothing
```

Das die Gesetze gelten, kann man hier leicht nachrechnen!

Interessanterweise reicht als Voraussetzung für die Monoid-Instanz hier bereits die Halbgruppe aus. Dass dies hier gut geht muss man nachrechnen, oder man weiß es bereits aus der Mathematik.

# Beispiel: Maybe als Monoid

Wenn der Inhalt ein Monoid bildet, dann bildet auch die Verpackung `Maybe` ein Monoid:

```
instance Semigroup a => Semigroup (Maybe a) where
  Nothing <> b       = b
  a       <> Nothing = a
  Just a  <> Just b  = Just (a <> b)
```

```
ins                                          re
  n
```

Das d

Intere -Instanz hier b uss man nachrechnen, oder man weiß es bereits aus der Mathematik.

Beweis von `(x<>y)<>z` $=$ `x<>(y<>z)` für `Semigroup (Maybe a)` an der Tafel teilweise ausgeführt.

Alle Fälle mit $x =$ `Nothing` oder $y =$ `Nothing` oder $z =$ `Nothing` sind eigentlich trivial, so dass wir nur $x =$ `Just a` und $x =$ `Just b` und $z =$ `Just c` betrachtet haben.

# Beispiel: Tupel als Monoide

Wenn die Inhalte ein Monoid bilden, dann bilden auch Paare davon wieder ein Monoid:

```
instance (Semigroup a, Semigroup b) =>
  Semigroup (a, b) where
    (a,b) <> (a',b') = (a<>a', b<>b')

instance (Monoid a, Monoid b) => Monoid (a,b) where
    mempty = (mempty, mempty)
```

*Beachte:* In der Definition von `(<>)` taucht `a<>a'` und `b<>b'` auf: Dies sind keine rekursiven Aufrufe! Stattdessen werden die Definition von `Semigroup a` und `Semigroup b` verwendet — was auch immer diese sind!

# Beispiel: Zahlen bilden Monoide

**Additives Monoid** $(+, 0)$: Es gilt $(x + y) + z = x + (y + z)$
**Multiplikatives Monoid** $(\cdot, 1)$: Es gilt $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

In der Haskell Standardbibliothek wurde entschieden, dass man sich explizit entscheiden muss, welches Monoid man meint:

```
> getSum     $ Sum     3 <> mempty <> Sum     4
7
> getProduct $ Product 3 <> mempty <> Product 4
12
```

Realisiert wird das mit Hilfe von `newtypes`:

```
newtype Sum a     = Sum     { getSum :: a }
newtype Product a = Product { getProduct :: a }
instance Num a => Semigroup (Sum a)     where
  Sum     x <> Sum     y = Sum     (x+y)
instance Num a => Semigroup (Product a) where
  Product x <> Product y = Product (x*y)
instance Num a => Monoid (Sum a)     where mempty = Sum 0
instance Num a => Monoid (Product a) where mempty = Product 1
```

# Beispiel: Zahlen bilden Monoide

**Additives Monoid** $(+, 0)$: Es gilt $(x + y) + z = x + (y + z)$
**Multiplikatives Monoid** $(\cdot, 1)$: Es gilt $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

In der Haskell Standardbibliothek wurde entschieden, dass man sich explizit entscheiden muss, welches Monoid man meint:

```
> getSum     $ mconcat $ map Sum     [3,4,5]
12
> getProduct $ mconcat $ map Product [3,4,5]
60
```

Realisiert wird das mit Hilfe von `newtypes`:

```
newtype Sum a     = Sum     { getSum :: a }
newtype Product a = Product { getProduct :: a }
instance Num a => Semigroup (Sum a)     where
  Sum     x <> Sum     y = Sum     (x+y)
instance Num a => Semigroup (Product a) where
  Product x <> Product y = Product (x*y)
instance Num a => Monoid (Sum a)     where mempty = Sum 0
instance Num a => Monoid (Product a) where mempty = Product 1
```

# Beispiel: Komposition als Monoid

Funktionen mit Typ `a->a` bilden ein Monoid unter Komposition:

```
instance Semigroup (a->a) where
  f <> g = f . g

instance Monoid (a->a) where
  mempty  = id
```

**Geht so nicht:**
Instanzdeklaration dürfen in Standard-Haskell keine Gleichheit zwischen Typvariablen erzwingen.
`instance Monoid (a->b)` wäre aber erlaubt.
**Abhilfe:** `newtype`

# Beispiel: Komposition als Monoid

Funktionen mit Typ `a->a` bilden ein Monoid unter Komposition:

```
newtype Endo a = Endo { appEndo :: a -> a }

instance Semigroup (Endo a) where
  Endo f <> Endo g = Endo (f . g)

instance Monoid (Endo a) where
  mempty  = Endo id
```

Beispiel:

```
> let f= mconcat $ map Endo [(4+),(10*),succ,max 1,\n -> n*n+1]
> appEndo f 1
34
```

Bemerkung: Dies ist wieder ein nicht-kommutatives Monoid, denn

`(\x->x*x) . (\y->y+1)` ≠ `(\y->y+1) . (\x->x*x)`

# Beispiel: Komposition als Monoid

Funktionen mit Typ `a->a` bilden ein Monoid unter Komposition:

```
newtype Endo a = Endo { appEndo :: a -> a }

instance Semigroup (Endo a) where
  Endo f <> Endo g = Endo (f . g)

instance Monoid (Endo a) where
  mempty  = Endo id
```

„Endo“ ist griechisch für „innerhalb“; wir haben es hier ja mit 1-stelligen *inneren* Abbildung zu tun.

Beispiel:

```
> let f= mconcat $ map Endo [(4+),(10*),succ,max 1,\n -> n*n+1]
> appEndo f 1
34
```

Bemerkung: Dies ist wieder ein nicht-kommutatives Monoid, denn

`(\x->x*x) . (\y->y+1)` ≠ `(\y->y+1) . (\x->x*x)`

# Zusammenfassung Halbgruppen und Monoide

- Halbgruppe: hat *assoziative* binäre Operation `(<>)::a->a->a`
- Monoid: hat *neutrales* Element `mempty` bezüglich `(<>)`
- Instanzen von `Monoid` sollten zu Instanzen von `Semigroup` passen (Pflicht ab GHC 8.4.x) und die Gesetze beachten.
- Wer GHC älter als 8.4.$x$ verwendet, muss anstatt `(<>)` immer `` `mappend` `` schreiben; oder `Data.Semigroup` importieren.
- Viele Instanzen in Standardbibliothek vordefiniert
- Erlaubt sehr starke verallgemeinerte Programmierung
  $\Rightarrow$ erhöht Wiederverwendbarkeit, erleichtert Wartung

# Linearisierung

Mit Hilfe eines Baum**durchlaufs** (engl. **tree traversal**) sammeln wir alle Knotenmarkierungen in einer Liste auf.

Die Reihenfolge der Listenelemente hängt dabei von der Art des gewählten Durchlaufs ab:

1. Vorordnung (engl. **preorder**)
   Zuerst Markierung, dann linker, dann rechter Teilbaum.

2. Symmetrische Ordnung (engl. **inorder**)
   Zuerst linker Teilbaum, dann Markierung, dann rechter Teilbaum.

3. Nachordnung (engl. **postorder**)
   Zuerst linker, dann rechter Teilbaum, dann Markierung.

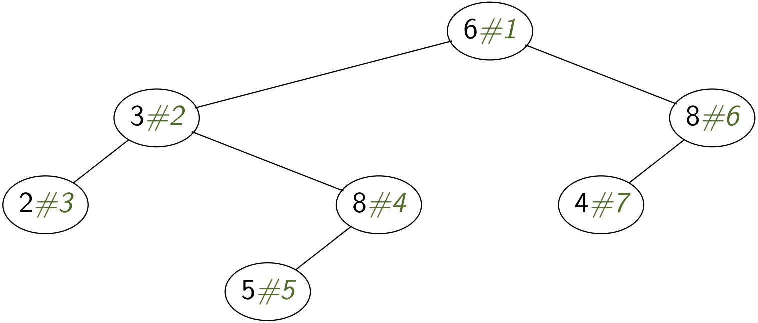
Alle Durchläufe bearbeiten die Teilbäume unabhängig voneinander.

# Vorordnung

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Zuerst Markierung, dann linker, dann rechter Teilbaum.

```
preorder :: Tree a -> [a]
preorder (Empty)      = []
preorder (Node x l r) = [x] ++ preorder l ++ preorder r
```




vgl. auch Folie 4.21

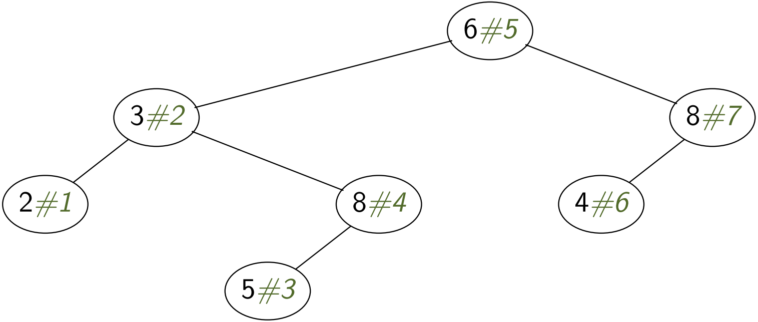
```
> preorder t
[6,3,2,8,5,8,4]
```

# Symmetrische Ordnung

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Zuerst linker Teilbaum, dann Markierung, dann rechter Teilbaum.

```
inorder :: Tree a -> [a]
inorder (Empty)      = []
inorder (Node x l r) = inorder l ++ [x] ++ inorder r
```



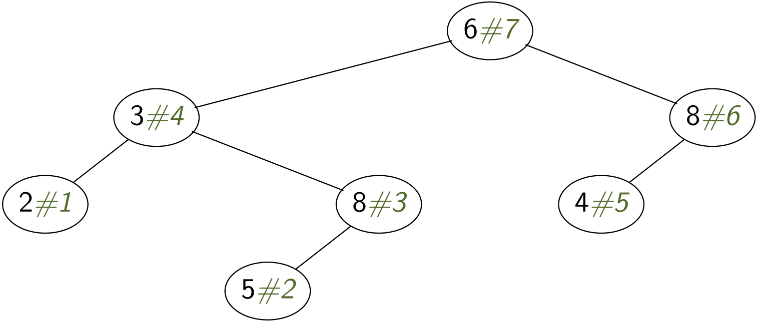

```
> inorder t
[2,3,5,8,6,4,8]
```

# Nachordnung

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Zuerst linker, dann rechter Teilbaum, dann Markierung.

```
postorder :: Tree a -> [a]
postorder (Empty)      = []
postorder (Node x l r) = postorder l ++ postorder r ++ [x]
```




```
> postorder t
[2,5,8,3,4,8,6]
```

# Modul Data.Foldable

```
class Foldable t where          {-# MINIMAL foldMap | foldr #-}
    fold    :: Monoid m => t m -> m
    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldr   :: (a -> b -> b) -> b -> t a -> b
    foldl   :: (b -> a -> b) -> b -> t a -> b
    toList  :: t a -> [a]
    null    :: t a -> Bool
    length  :: t a -> Int
    elem    :: Eq a => a -> t a -> Bool
    sum, product :: Num a => t a -> a
```

Verallgemeinert Zusammenfalten von Listen auf andere Typen

## Gesetze

- Identität: `fold == foldMap id`
- Gesetze für `foldr`/`foldl`:
  `foldr f z t = appEndo (foldMap (Endo . f) t) z`
- Funktoren-Komposition: `foldMap f = fold . fmap f`
  damit auch `foldMap f . fmap g == foldMap (f . g)`

# Instanzen für Foldable

```haskell
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Bei der Instanzdeklaration für `Foldable` müssen wir uns für eine Variante des Durchlaufs entscheiden:

```haskell
instance Foldable Tree where
  -- foldMap :: Monoid m => (a -> m) -> Tree a -> m
  foldMap f Empty = mempty
  foldMap f (Node x l r)
    = f x <> foldMap f l <> foldMap f r    -- preorder
```

Das ist auch okay, denn die Durchlauf-Reihenfolge lässt sich schwer verallgemeinern. Was würde z.B. „inorder“ bei einem Baum mit drei Kindern bedeuten?

```haskell
data Tree3 a = Empty3
             | Node3 a (Tree a) (Tree a) (Tree a)
```

# Instanzen für Foldable

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Bei der Instanzdeklaration für `Foldable` müssen wir uns für eine Variante des Durchlaufs entscheiden:

```
instance Foldable Tree where
  -- foldMap :: Monoid m => (a -> m) -> Tree a -> m
  foldMap f Empty = mempty
  foldMap f (Node x l r)
    = foldMap f l <> f x <> foldMap f r     -- inorder
```

Das ist auch okay, denn die Durchlauf-Reihenfolge lässt sich schwer verallgemeinern. Was würde z.B. „inorder“ bei einem Baum mit drei Kindern bedeuten?

```
data Tree3 a = Empty3
             | Node3 a (Tree a) (Tree a) (Tree a)
```

# Instanzen für Foldable

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Bei der Instanzdeklaration für `Foldable` müssen wir uns für eine Variante des Durchlaufs entscheiden:

```
instance Foldable Tree where
  -- foldMap :: Monoid m => (a -> m) -> Tree a -> m
  foldMap f Empty = mempty
  foldMap f (Node x l r)
    = foldMap f l <> foldMap f r <> f x  -- postorder
```

Das ist auch okay, denn die Durchlauf-Reihenfolge lässt sich schwer verallgemeinern. Was würde z.B. „inorder“ bei einem Baum mit drei Kindern bedeuten?

```
data Tree3 a = Empty3
             | Node3 a (Tree a) (Tree a) (Tree a)
```

# Instanzen für Foldable

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Bei der Instanzdeklaration für `Foldable` müssen wir uns für eine Variante des Durchlaufs entscheiden:

```
instance Foldable Tree where
  -- foldMap :: Monoid m => (a -> m) -> Tree a -> m
  foldMap f Empty = mempty
  foldMap f (Node x l r)
    = foldMap f l <> f x <> foldMap f r     -- inorder
```

Man kann natürlich jeweils eigene Datentypen und Instanzen für jede benötigte Variante deklarieren:

```
data TreePre  a = EmptyPre
                | NodePre  a (TreePre  a) (TreePre  a)
data TreePost a = EmptyPost
                | NodePost a (TreePost a) (TreePost a)
```

# Instanzen für Foldable

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Bei der Instanzdeklaration für `Foldable` müssen wir uns für eine Variante des Durchlaufs entscheiden:

```
instance Foldable Tree where
  -- foldMap :: Monoid m => (a -> m) -> Tree a -> m
  foldMap f Empty = mempty
  foldMap f (Node x l r)
    = foldMap f l <> f x <> foldMap f r      -- inorder
```

Die generischen Typklassen liefern automatisch viele nützliche Funktionalitäten unter verständlichen Namen:

```
> t
(6,(3,<2>,(8,<5>,ε)),(8,ε,<4>))
> foldMap Sum t
Sum {getSum = 36}
> foldMap Product t
Product {getProduct = 46080}
```

# Instanzen für Foldable

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Bei der Instanzdeklaration für `Foldable` müssen wir uns für eine Variante des Durchlaufs entscheiden:

```
instance Foldable Tree where
  -- foldMap :: Monoid m => (a -> m) -> Tree a -> m
  foldMap f Empty = mempty
  foldMap f (Node x l r)
    = foldMap f l <> f x <> foldMap f r     -- inorder
```

Die generischen Typklassen liefern automatisch viele nützliche Funktionalitäten unter verständlichen Namen:

```
> sum t
36
> product t
46080
> length t
7
```

# Instanzen für Foldable

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Bei der Instanzdeklaration für `Foldable` müssen wir uns für eine Variante des Durchlaufs entscheiden:

```
instance Foldable Tree where
  -- foldMap :: Monoid m => (a -> m) -> Tree a -> m
  foldMap f Empty = mempty
  foldMap f (Node x l r)
    = foldMap f l <> f x <> foldMap f r     -- inorder
```

Die generischen Typklassen liefern automatisch viele nützliche Funktionalitäten unter verständlichen Namen:

```
> foldMap show t
"2358684"
> foldr (\a b -> "("++show a++b++")")  "1"  t
"(2(3(5(8(6(8(41)))))))"
> foldl (\b a -> "("++b++show a++")")  "1"  t
"(((((((12)3)5)8)6)8)4)"
```

# Instanzen für Foldable

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Bei der Instanzdeklaration für `Foldable` müssen wir uns für eine Variante des Durchlaufs entscheiden:

```
instance Foldable Tree where
  -- foldMap :: Monoid m => (a -> m) -> Tree a -> m
  foldMap f Empty = mempty
  foldMap f (Node x l r)
    = foldMap f l <> f x <> foldMap f r      -- inorder
```

*Achtung:* Für GHC $< 8.4.x$ *muss* man hier `mappend` anstatt `(<>)` verwenden.

# Generischer Tiefendurchlauf

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Bei den gezeigten Linearisierung handelt es sich jeweils um einen **Tiefendurchlauf** (engl. **depth-first traversal**). Merkmal ist dabei, dass alle Teilbäume unabhängig voneinander bearbeitet werden.

Einen generischen Tiefendurchlauf erhalten wir, wenn wir für *jeden Konstruktor* einen Parameter einführen, der alle Argumente des jeweiligen Konstruktors auf ein Ergebnis abbildet:

```
foldTree1 :: ( b, (a,b,b)->b ) -> Tree a -> b
foldTree1 (fe, fn)  Empty = fe
foldTree1 (fe, fn) (Node x l r)
  = fn (x, foldTree1 (fe,fn) l, foldTree1 (fe,fn) r)
```

Die drei Baumlinearisierungen erhalten wir dann mit:

```
preorder  t = foldTree1 [] (\(l,x,r) -> [x]++l++r) t
inorder   t = foldTree1 [] (\(l,x,r) -> l++[x]++r) t
postorder t = foldTree1 [] (\(l,x,r) -> l++r++[x]) t
```

# Gen. Tiefendurchlauf (Pointfree & Curried)

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Bei den gezeigten Linearisierung handelt es sich jeweils um einen **Tiefendurchlauf** (engl. **depth-first traversal**). Merkmal ist dabei, dass alle Teilbäume unabhängig voneinander bearbeitet werden.

Einen generischen Tiefendurchlauf erhalten wir, wenn wir für *jeden Konstruktor* einen Parameter einführen, der alle Argumente des jeweiligen Konstruktors auf ein Ergebnis abbildet:

```
foldTree2 :: b -> (a -> b -> b -> b) -> Tree a -> b
foldTree2 fe fn = fTaux
  where fTaux  Empty        = fe
        fTaux (Node x l r) = fn x (fTaux l) (fTaux r)
```

Die drei Baumlinearisierungen erhalten wir dann mit:

```
preorder    = foldTree2 [] (\l x r -> [x]++l++r)
inorder     = foldTree2 [] (\l x r -> l++[x]++r)
postorder   = foldTree2 [] (\l x r -> l++r++[x])
```

# Gen. Tiefendurchlauf (Pointfree & Curried)

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Bei den gezeigten Linearisierung handelt es sich jeweils um einen **Tiefen**… …rkmal ist dabei, dass a… …itet werden.

Einen … …n wir für *jeden Konstruktor* einen Parameter einführen, der alle Argumente des jeweiligen Konstruktors auf ein Ergebnis abbildet:

> Wem das immer noch nicht allgemein genug ist: GHC kann Funktionen wie `foldTree` und `mapTree` auch automatisch erzeugen
> ⇒ Generic Programming.

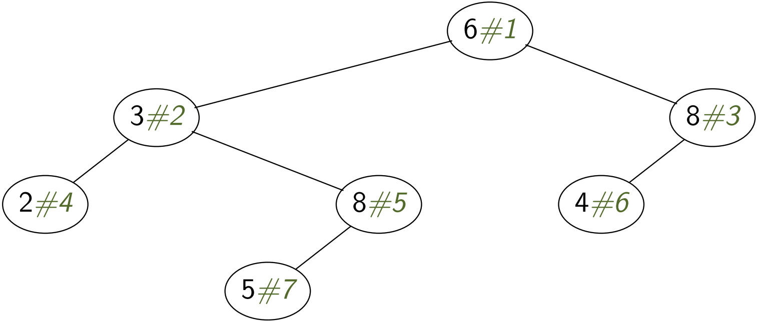
```
foldTree2 :: b -> (a -> b -> b -> b) -> Tree a -> b
foldTree2 fe fn = fTaux
  where fTaux  Empty        = fe
        fTaux (Node x l r) = fn x (fTaux l) (fTaux r)
```

Die drei Baumlinearisierungen erhalten wir dann mit:

```
preorder    = foldTree2 [] (\l x r -> [x]++l++r)
inorder     = foldTree2 [] (\l x r -> l++[x]++r)
postorder   = foldTree2 [] (\l x r -> l++r++[x])
```

# Breitendurchlauf

Beim **Breitendurchlauf** (engl. **breadth-first traversal**) werden die Teilbäume nicht unabhängig behandelt.




Der Baum wird schichtenweise abgearbeitet, d.h. es werden zuerst Knoten mit geringerer Tiefe bearbeitet:
Linearisierung hier zu `[6,3,8,2,8,4,5]`

# Breitendurchlauf

## Implementierung durch Verallgemeinerung:

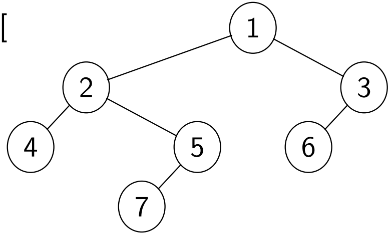
Breitendurchlauf eines **Waldes** (engl. **forest**), d.h. Liste von Bäumen.

```
breadthForest :: [Tree a] -> [a]
breadthForest [] = []
breadthForest ( Empty        : forest)
  =       breadthForest  forest
breadthForest ((Node x l r) : forest)
  =  x : breadthForest (forest ++ [l,r])
```

- Zuerst alle Wurzeln, dann alle Knoten der Tiefe 1, 2, 3, . . . .
- `breadthForest` erlaubt eine elegante rekursive Formulierung

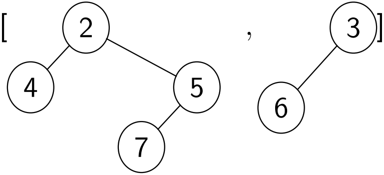
Für einen Breitendurchlauf eines einzelnen Baums `t` ruft man `breadthForest [t]` auf.

`breadthForest[`



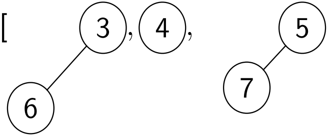
`] =`

`1:breadthForest[`



`] =`

`1:2:breadthForest[`



`] = ...`

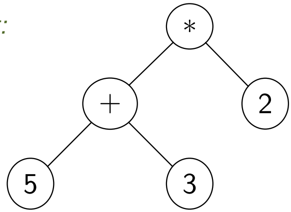
`[1,2,3,4,5,6,7]`

# Arithmetische Ausdrücke als Bäume

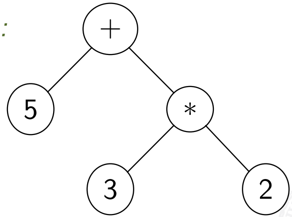
**Beispiel** Ein Mathe-Nachhilfe-Programm für Grundschüler.
Es druckt zufällig einen **arithmetischen Ausdruck** wie $(5 + 3) * 2$, und vergleicht den Lösungsvorschlag mit dem **Wert** des Ausdrucks (hier: 16). Die Zahlen sollen dabei immer einstellig und die Operationen nur $+$ und $*$ sein.

**Beobachtung**
Den Ausdruck können wir intern als Binärbaum repräsentieren.

*Richtig:*




*Falsch:*




rechter Baum entspricht $5 + (3 * 2)$

# Arithmetische Ausdrücke als Binärbäume

- Es gibt drei Arten von Knotenmarkierungen: Zahl, "+", "*".

```
data Label = Const Integer
           | Plus
           | Times

type Expr1 = Tree Label
```

Die Typabkürzung `Expr1` reicht hier, denn es ist ja ein Baum.

- *Beispiel*

```
a1 = Node Times ( Node Plus (leaf (Const 5))
                            (leaf (Const 3)) )
                ( leaf (Const 2) )
```

# Ausdrücke ausdrucken

Wegen `type Expr1 = Tree Label` können wir keine eigene Instanz für `Show Expr1` definieren.

Nur mit `newtype Expr1 = Expr1 (Tree Label)` möglich.

Da wir bereits `instance Show a => Show (Tree a)` haben, reicht:

```
instance Show Label where
  show (Const i) = show i
  show Plus  = "+"
  show Times = "*"
```

Damit erhalten wir den arithmetischen Ausdruck in Präfix-Notation:

```
> show a1
"(*,(+,<5>,<3>),<2>)"
```

. . . brauchbar, aber nicht so wirklich hübsch.

# Ausdrücke ausdrucken

Eine hübschere Repräsentation des Ausdrucks als Zeichenkette erhalten wir z.B. wie folgt:

```
exprToString :: Expr1 -> String
exprToString  Empty                 =  ""
exprToString (Node (Const n) _  _)  =  show n
exprToString (Node  Plus     l  r)  =
  "("++ exprToString l ++" + "++ exprToString r ++")"
exprToString (Node  Times    l  r)  =
  "("++ exprToString l ++" * "++ exprToString r ++")"

exprToString a1  ==  "((5 + 3) * 2)"
```

## Übungen

- Implementieren Sie `exprToString` durch `foldTree`

# Auswertung

Den **Wert** des Ausdrucks können wir rekursiv berechnen (H5-1), oder mit Hilfe von `foldTree`:

```
eval1 :: Expr1 -> Integer
eval1 t = foldTree1 (0, enode) t
  where
    enode :: (Label, Integer, Integer) -> Integer
    enode (Const n,_,_) = n
    enode (Plus    ,l,r) = l + r
    enode (Times   ,l,r) = l * r
```

Wir sagen, `eval` (dt. *auswerten*) **interpretiert** den Ausdruck.

```
> eval a1
16
```

Bemerkung: `foldMap`/`foldr` nur möglich, wenn Teilbäume unabhängig mit *gleicher Operation* `(<>)` verknüpft werden. Hier muss aber manchmal `(+)`,`(*)` oder `const` verwendet werden.

# Auswertung

Den **Wert** des Ausdrucks können wir rekursiv berechnen (H5-1), oder mit Hilfe von `foldTree`:

```
eval2 :: Expr1 -> Integer  -- Curried & Pointfree
eval2 = foldTree2 0 enode
  where
    enode :: Label -> Integer -> Integer -> Integer
    enode (Const n) = const . const n
    enode (Plus)    = (+)
    enode (Times)   = (*)
```

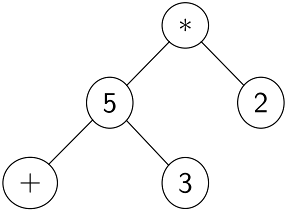
Wir sagen, `eval` (dt. *auswerten*) **interpretiert** den Ausdruck.

```
> eval a1
16
```

Bemerkung: `foldMap`/`foldr` nur möglich, wenn Teilbäume unabhängig mit *gleicher Operation* `(<>)` verknüpft werden.
Hier muss aber manchmal `(+)`,`(*)` oder `const` verwendet werden.

# Ungültige Ausdrücke

Die Binärbaum-Repräsentation erlaubt ungültige Ausdrücke (engl. *malformed expressions*).




```
bad = Node Times (Node (Const 5)
                       (leaf Plus) (Const 3))
                 (leaf (Const 2))
```

- Zahlen sollen nur als Blattknoten auftreten!
- Operationen nur als innere Knoten!

# Arithmetische Ausdrücke als Datentyp

Wir definieren einen speziellen Baumtyp `expr`:

```
data Expr = Const Integer
          | Plus  Expr Expr
          | Times Expr Expr
  deriving (Eq)

a2 = Times (Plus (Const 5) (Const 3)) (Const 2)
```

- Die Ausdrucksrepräsentation ist ökonomischer.
- Die Funktionsdefinitionen sind auch klarer:

```
eval :: Expr -> Integer
eval (Const n)     =  n
eval (Plus  l r)   =  eval l + eval r
eval (Times l r)   =  eval l * eval r
```

# Arithmetische Ausdrücke als Datentyp

Wir definieren einen speziellen Baumtyp `expr`:

```
data Expr = Const Integer
          | Plus  Expr Expr
          | Times Expr Expr
  deriving (Eq)
```

**Hinweis:**
Im behandelten Beispiel würde auch ein generischer Baum funktionieren:

```
data Tree2 a b = Leaf a | Node b (Tree2 a b) (Tree2 a b)
data Op = Plus | Times
type Expr2 = Tree2 Integer Op
```

Bei Operationen mit anderer Stelligkeit (z.B. ein unäres Minus) bräuchte man entsprechend Bäume mit Knoten mit 1 oder 3 Kindern, etc.

Diese sinnvolle Alternative haben wir bereits in H5-1 behandelt!

# Lexing

Jetzt wollen wir Ausdrücke einlesen! Wir verwenden dazu Datentyp

```
data Token = CONST Integer
           | LPAREN | RPAREN | PLUS | TIMES
```

Als String notierte Ausdrücke entsprechen Token-Listen, z.B.

```
s1 = [CONST 3, TIMES, LPAREN, CONST 8, PLUS, CONST 3,
      RPAREN, PLUS, CONST 5, TIMES, CONST 4]
```

entspricht dem String `"3 * ( 8 + 3 ) + 5 * 4"`

Die Aufgabe, aus einer Zeichenkette solch eine Token-Liste zu erzeugen, heißt **lexikalische Analyse**, oder auch kurz **lexing**.

### Übung

Implementieren Sie eine Funktion `lexer :: String -> [Token]` die genau das leistet!

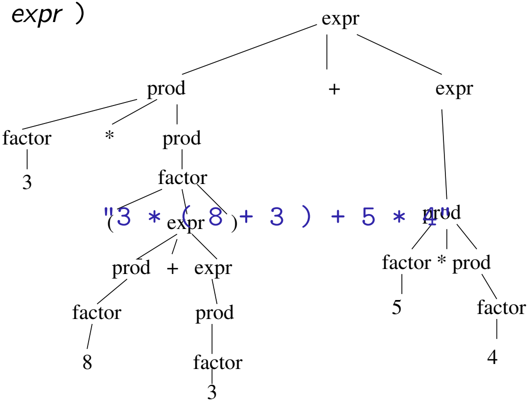
# Parsing durch rekursiven Abstieg

Die Aufgabe, aus einer Token-Liste einen Syntaxbaum zu erzeugen, heißt **Parsing**, oder **Syntaxanalyse**.

Beschreibung Arithmetischer Ausdrücke durch BNF-Grammatik:

| | | | | |
|---|---|---|---|---|
| *expr* | ::= | *prod* | \| | *prod* + *expr* |
| *prod* | ::= | *factor* | \| | *factor* * *prod* |
| *factor* | ::= | *const* | \| | ( *expr* ) |

Punkt-vor-Strich und Klammerung wird von dieser Grammatik richtig behandelt.



# Parsing durch rekursiven Abstieg

Die Syntaxanalyse orientiert sich an dieser Grammatik. Wir schreiben drei wechselseitig rekursive Funktionen

```
parseExpr   :: [Token] -> (Expr,[Token])
parseProd   :: [Token] -> (Expr,[Token])
parseFactor :: [Token] -> (Expr,[Token])
```

wobei `parseExpr l` versucht, ein möglichst großes Anfangsstück von `l` als *expr* zu interpretieren. Eventuell unbenutzte Tokens am Ende werden zurückgegeben. Ebenso `parseProd`, `parseFactor`.

```
parseExpr l =
  let (summand1,rest1) = parseProd l in
  case rest1 of
     PLUS:rest2 -> let (summand2,rest3) = parseExpr rest2
                   in  (Plus summand1 summand2, rest3)
     _other     -> (summand1,rest1)
```

# Zusammenfassung Bäume

- Bäume als induktiv definierte mathematische Objekte.
- Bäume als rekursiver Datentyp
- Rekursive Funktionen auf Bäumen
- Linearisierungen: Vor-, In-, Nachordnung
- Tiefen- und Breitensuche für Bäume
- Bäume als Repräsentation von Syntax:
  Lexikalische und Syntaxanalyse.

# Vorne- oder hinten Anhängen?

Hier sind zwei Implementierung zum Umdrehen einer Liste:

```
revAcc :: [a] -> [a] -> [a]
revAcc acc []     =  acc
revAcc acc (x:l) =  revAcc (x:acc) l

revSpec :: [a] -> [a]
revSpec    []     =  []
revSpec    (x:l) =  revSpec l ++ [x]
```

Wir wollen die Laufzeit und den Platzverbrauch empirisch untersuchen.

```
test :: ([Int] -> [a]) -> Int -> Int
test f n = length $ f [n,n-1..1]
```

# Laufzeitmessung

```
revAcc acc []    =  acc
revAcc acc (x:l) =  revAcc (x:acc) l

revSpec    []    =  []
revSpec    (x:l) =  revSpec l ++ [x]

test f n = length $ f [n,n-1..1]
```

## Experimenteller Vergleich:

```
> :set +s +r
> test (revAcc []) 30000
30000
(0.01 secs, 6,318,544 bytes)
> test  revSpec    30000
30000
(12.63 secs, 39,653,291,824 bytes)
```

# Laufzeitmessung

Beachte: Zeit- & Speichermessungen mit GHCI können manchmal sehr trügerisch sein — besser GHC mit Option -O2 nehmen.

- `:set +s` schaltet Anzeige der Statistik ein
- `:set +r` verhindert Speicherung von Ergebnissen zwischen zwei Aufrufen, damit nächste Anfrage alles komplett neu auswertet

## Experimenteller Vergleich:

```
> :set +s +r
> test (revAcc []) 30000
30000
(0.01 secs, 6,318,544 bytes)
> test  revSpec    30000
30000
(12.63 secs, 39,653,291,824 bytes)
```

# Hängt es mit der Endrekursion zusammen?

```
filterAcc :: [a] -> (a->Bool) -> [a] -> [a]
filterAcc acc p [] = acc
filterAcc acc p (x:l) | p x       = filterAcc (acc++[x]) p l
                      | otherwise = filterAcc  acc       p l

filterSpec ::       (a->Bool) -> [a] -> [a]
filterSpec p [] = []
filterSpec p (x:l)    | p x       = x:filterSpec p l
                      | otherwise =   filterSpec p l
```

`filterAcc` ist endrekursiv, `filterSpec` ist nicht endrekursiv.

```
> test (filterAcc [] odd) 50000
25000
(12.99 secs, 27,481,723,576 bytes)
> test (filterSpec   odd) 50000
25000
(0.02 secs, 18,877,528 bytes)
```

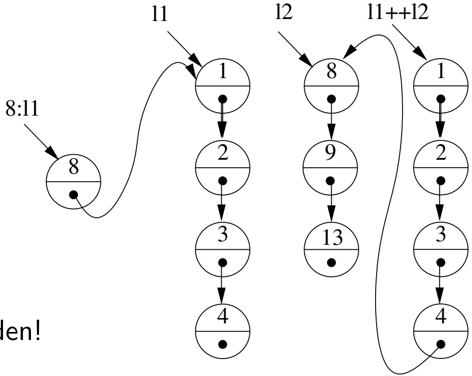
An der Endrekursion liegt der Unterschied hier also nicht!

# Anmerkungen zur Laufzeit

Listen werden als Ketten gespeichert: Jeder Listen-Knoten speichert einen Eintrag (hier `Int`-Wert) und einen **Verweis**(=Speicheradresse) auf den nächsten Knoten.

- Ein "cons", also `(x:l)` benötigt konstante Zeit (ein paar Taktzykeln)
- Abgleich gegen ein Muster der Form `(kopf:rumpf)` benötigt konstante Zeit
- Eine Verkettung `l1 ++ l2` benötigt Zeit proportional zur Länge von `l1`, denn die erste Liste muss kopiert werden!

```
(++) []     ys = ys
(++) (x:xs) ys = x : (xs ++ ys)
```



# Anmerkungen zur Laufzeit

Listen werden als Ketten gespeichert: Jeder Listen-Knoten speichert einen Eintrag (hier `Int`-Wert) und einen **Verweis**(=Speicheradresse) auf den nächsten Knoten.

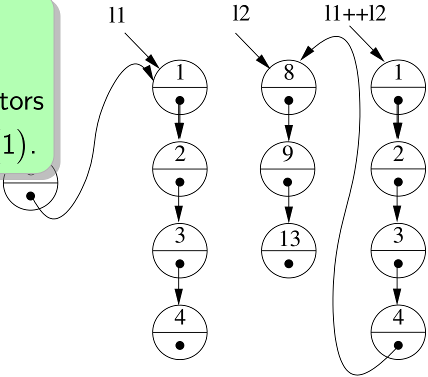
- Ein "cons", also `(x:l)`

Grundsätzlich gilt

- einzelne Konstruktor Applikation
- Pattern-Matching eines Konstruktors

haben konstanten Zeit/Platzbedarf $O(1)$.

benötigt konstante Zeit



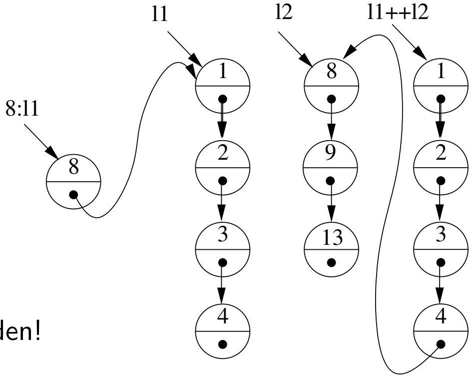

- Eine Verkettung `l1 ++ l2` benötigt Zeit proportional zur Länge von `l1`, denn die erste Liste muss kopiert werden!

```
(++) []     ys = ys
(++) (x:xs) ys = x : (xs ++ ys)
```

# Anmerkungen zur Laufzeit

Listen werden als Ketten gespeichert: Jeder Listen-Knoten speichert einen Eintrag (hier `Int`-Wert) und einen **Verweis**(=Speicheradresse) auf den nächsten Knoten.

- Ein "cons", also `(x:l)` benötigt konstante Zeit (ein paar Taktzykeln)
- Abgleich gegen ein Muster der Form `(kopf:rumpf)` benötigt konstante Zeit
- Eine Verkettung `l1 ++ l2` benötigt Zeit proportional zur Länge von `l1`, denn die erste Liste muss kopiert werden!




```
(++) []     ys = ys
(++) (x:xs) ys = x : (xs ++ ys)
```

$$T_{(++)}(n+1,m) = T_{(++)}(n,m) + O(1) \leadsto T_{(++)}(n,m) = O(n)$$

# Anmerkungen zur Laufzeit

- Zeitaufwand `revSpec`:

```
revSpec     []     =  []
revSpec     (x:l)  =  revSpec l ++ [x]
```

$$T_{\texttt{revSpec}}(n+1) = T_{\texttt{revSpec}}(n) + O(n) \rightsquigarrow T_{\texttt{revSpec}}(n) = O(n^2)$$

- Zeitaufwand `revAcc`:

```
revAcc acc []     =  acc
revAcc acc (x:l)  =  revAcc (x:acc) l
```

$$T_{\texttt{revAcc}}(m, n+1) = T_{\texttt{revAcc}}(m+1, n) + O(1) \rightsquigarrow T_{\texttt{revAcc}}(m, n) = O(n)$$

Hinweis: $O(f(n))$ bezeichnet eine Funktion, welche durch $c \cdot f(n)$ für ein festes $c$ beschränkt werden kann. Es gilt $n \cdot O(1) = O(n)$ und für Konstanten $a, b, c \in \mathbb{R}$ gilt $O(a \cdot n + b) = O(n)$ und $O(a \cdot n^2 + b \cdot n + c) = O(n^2)$.

Also $O(n)$ (höchstens) linear; und $O(n^2)$ (höchstens) quadratisch.

# Beispiel: Sortieren durch Einfügen 

Eine Liste $[x_1, \ldots, x_n]$ von heißt **sortiert**, wenn $x_1 \leq x_2 \leq \cdots \leq x_n$.

```
insertel :: Ord a => a -> [a] -> [a]
insertel x []     = [x]
insertel x (y:l) = if x <= y then x:y:l
                               else y:insertel x l

inssort :: Ord a => [a] -> [a]
inssort [] = []
inssort (x:l) = insertel x  (inssort l)
```

Ist `l` sortiert, so auch `insertel a l` und es enthält dieselben Elemente wie `(a:l)`.

Für beliebiges `l` ist `inssort l` sortiert und enthält dieselben Elemente wie `l`.

# Beispiel: Quicksort

Wdh. A3-2

```
quicksort :: [Int] -> [Int]
quicksort    []   = []
quicksort (h:t) =
    (quicksort smaller) ++ [h] ++ (quicksort bigger)
  where
    (smaller,bigger) = splitBy h t

splitBy :: Int -> [Int] -> ([Int],[Int])
splitBy _ []     = ([],[])
splitBy p (h:t) | h <= p     = (h:smaller,   bigger)
                | otherwise = (  smaller, h:bigger)
  where
    (smaller,bigger) = splitBy p t
```

# Beispiel: Sortieren durch Mischen

```
mergesort :: Ord a => [a] -> [a]
mergesort []  = []
mergesort [a] = [a]
mergesort l   = merge (mergesort l1)  (mergesort l2)
  where (l1,l2) = split l
        split :: [a] -> ([a],[a])
        split []  = ([],[])
        split [a] = ([a],[])
        split (a:b:u) = (a:u1,b:u2)
              where (u1,u2) = split u
        merge :: Ord a => [a] -> [a] -> [a]
        merge   []    []   = []
        merge   []     u   = u
        merge    u    []   = u
        merge (x:u) (y:v)
          | x<= y      = x:merge    u   (y:v)
          | otherwise = y:merge (x:u)    v
```

# Experimenteller Vergleich

```
> :set +r +s
> test (inssort) 3333
(1.41 secs, 1,737,891,384 bytes)
> test (inssort) 12345
(20.87 secs, 24,400,496,712 bytes)

> test (quicksort) 3333
(54.81 secs, 2,789,277,408 bytes)

> test (mergesort) 3333
(0.03 secs, 23,177,688 bytes)
> test (mergesort) 123456
(1.65 secs, 1,224,082,864 bytes)
> test (mergesort) 1234567
(22.90 secs, 14,573,141,296 bytes)
```

# Analytische Abschätzung der Laufzeit

- Zeitaufwand `insertel`
  $T_{\texttt{insertel}}(n+1) = T_{\texttt{insertel}}(n) + O(1) \leadsto T_{\texttt{insertel}}(n) = O(n)$
- Zeitaufwand `inssort`
  $T_{\texttt{inssort}}(n+1) = T_{\texttt{inssort}}(n) + O(n) \leadsto T_{\texttt{inssort}}(n) = O(n^2)$

- Zeitaufwand `splitBy`
  $T_{\texttt{splitBy}}(n+1) = T_{\texttt{splitBy}}(n) + O(1) \leadsto T_{\texttt{mergesort}}(n) = O(n)$
- Zeitaufwand `quicksort` $T_{\texttt{quicksort}}(n+1) = 2T_{\texttt{quicksort}}(n) + 3O(n) \leadsto T_{\texttt{mergesort}}(n) = O(n^2)$

- Zeitaufwand `mergesort` $T_{\texttt{mergesort}}(n) = 2T_{\texttt{mergesort}}(n/2) + O(n) \leadsto T_{\texttt{mergesort}}(n) = O(n \log(n))$

# Anmerkung zu Quicksort

Quicksort hat eine quadratische Laufzeitkomplexität, aber diese tritt nur auf, wenn bei der Aufteilung der Eingabeliste immer alle Elemente in einer der beiden Hälften landen und die andere Hälfte leer bleibt.

In unserem Test war dies der Fall, da die Eingabeliste vorsortiert war! Ist dies nicht der Fall, so läuft der Algorithmus schneller:

```
> test quicksort 3330
3330
(5.29 secs, 2,784,256,328 bytes)
> length $ quicksort $ concat $ replicate 333 [1..10]
3330
(0.36 secs, 252,303,248 bytes)
> length $ quicksort $ concat $ replicate 1234 [1..20]
24680
(11.29 secs, 7,560,814,008 bytes)
```

⇒ Es wird eine probabilistische Analyse benötigt!

# Effizienzproblem bei breadthForest

```
breadthForest :: [Tree a] -> [a]
breadthForest [] = []
breadthForest (Empty      : forest) = breadthForest forest
breadthForest (Node x l r : forest)
  = x : breadthForest (forest ++ [l, r])
```

Einhängen von `l`, `r` braucht Zeit $O(n)$, wenn $n$ die Größe des Waldes ist.

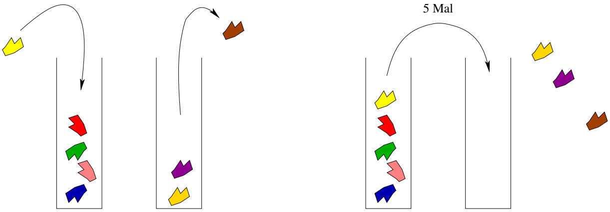
```
fulltree 0 = Empty
fulltree n = let l = fulltree (n-1) in Node 0 l l
*Tree> length (breadthForest [fulltree 14])
16383
(8.57 secs, 5890705980 bytes)
*Tree> length (breadthForest [fulltree 16])
65535
(506.69 secs, 94519577072 bytes)
```

# Amortisierte Schlange

Verkettete Listen wie in Haskell unterstützen sehr effizient die Datenstruktur Stapel (Stack, last-in-first-out=**LIFO**), haben aber ein Effizienzproblem, wenn sie direkt als Warteschlange (first-in-first-out=**FIFO**) eingesetzt werden.

Abhilfe bietet die Implementierung einer Schlange durch zwei Stapel: Eingangsstapel ("in tray") und Ausgangsstapel wie in H5-3!

Einfügen in den Eingangsstapel, Entnehmen vom Ausgangsstapel. Wird der Ausgangsstapel leer, so wird der gesamte Eingangsstapel en-bloc in den Ausgangsstapel verschoben und dabei umgedreht.



# Anwendung und Laufzeit

```
breadthForest2 :: [Tree a] -> [Tree a] ->  [a]
breadthForest2 [] [] = []
breadthForest2 [] in_tray =
          breadthForest2 (reverse in_tray) []
breadthForest2 (Empty : forest) in_tray =
          breadthForest2 forest in_tray
breadthForest2 (Node x l r : forest) in_tray =
          x:breadthForest2 forest (r:l:in_tray)

*Tree> length (breadthForest2 [fulltree 16][])
65535
(0.14 secs, 13648716 bytes)
*Tree> length (breadthForest2 [fulltree 20][])
1048575
(2.36 secs, 214595844 bytes)
```

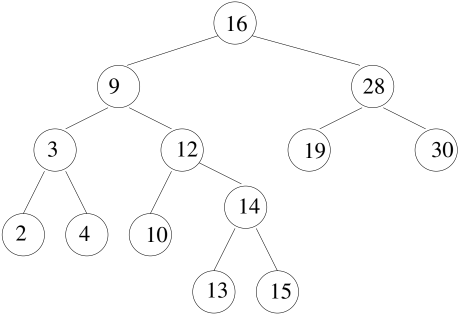
# Analyse der Laufzeit

- Die `reverse` Operation benötigt auch lineare Zeit, tritt aber nur selten auf.
- Jeder Baum wird dreimal bewegt: Beim Einhängen, beim Hinüberkopieren, beim Aushängen.
- Potentialmethode: Man berechnet beim Einhängen zusätzlich zu den tatsächlichen Kosten von, sagen wir, 1€, noch fiktive Kosten von 1€ um für die spätere `reverse` Operation "vorzusorgen".
- Das Einhängen in die Schlange verursacht so Kosten von 2€, das Aushängen, gleich ob `reverse` erforderlich ist, oder nicht, kostet 1€, da das `reverse` aus dem angesparten Kapital bezahlt werden kann.

Mehr dazu in Algorithmen & Datenstrukturen

# Binärer Suchbaum

In einem binären Suchbaum (BST) sind die Knotenmarkierungen des linken Teilbaums kleiner oder gleich der Wurzelmarkierung und die Knotenmarkierungen des rechten Teilbaums größer oder gleich der Wurzelmarkierung. Alle Teilbäume sind selbst wieder (BST)




Für jeden Knoten mit Markierung $x$ gilt:

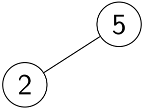
- Die Markierungen des linken Teilbaumes sind $\leq x$;
- Die Markierungen des rechten Teilbaumes sind $\geq x$;

# Beispiel: Einfügen/Suche in einem Suchbaum

Für fügen der Reihe nach folgende Elemente in einen binären Suchbaum ein: $[5, 2, 3, 7, 6, 8, 1]$

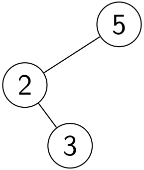
# Beispiel: Einfügen/Suche in einem Suchbaum

Für fügen der Reihe nach folgende Elemente in einen binären Suchbaum ein: $[5, 2, 3, 7, 6, 8, 1]$



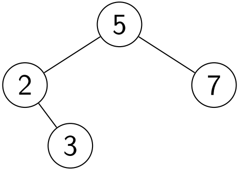
# Beispiel: Einfügen/Suche in einem Suchbaum

Für fügen der Reihe nach folgende Elemente in einen binären Suchbaum ein: $[5, 2, 3, 7, 6, 8, 1]$



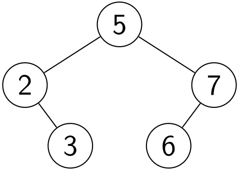
# Beispiel: Einfügen/Suche in einem Suchbaum

Für fügen der Reihe nach folgende Elemente in einen binären Suchbaum ein: $[5, 2, 3, 7, 6, 8, 1]$



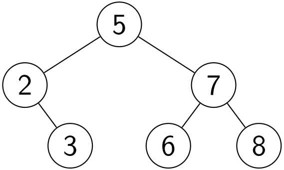
# Beispiel: Einfügen/Suche in einem Suchbaum

Für fügen der Reihe nach folgende Elemente in einen binären Suchbaum ein: $[5, 2, 3, 7, 6, 8, 1]$



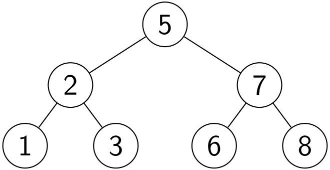
# Beispiel: Einfügen/Suche in einem Suchbaum

Für fügen der Reihe nach folgende Elemente in einen binären Suchbaum ein: $[5, 2, 3, 7, 6, 8, 1]$



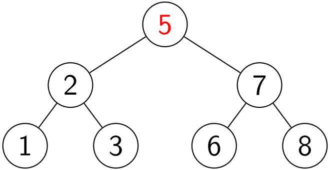
# Beispiel: Einfügen/Suche in einem Suchbaum

Für fügen der Reihe nach folgende Elemente in einen binären Suchbaum ein: $[5, 2, 3, 7, 6, 8, 1]$



# Beispiel: Einfügen/Suche in einem Suchbaum

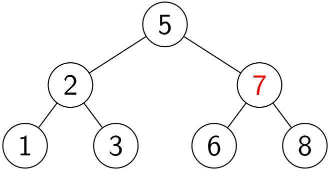
Für fügen der Reihe nach folgende Elemente in einen binären Suchbaum ein: $[5, 2, 3, 7, 6, 8, 1]$




Suche: Um Zu prüfen, ob die Zahl 6 im BST enthalten ist, müssen wir nur der Reihe nach die Elemente $5, 7, 6$ prüfen.
In der Liste hätten wir anstatt 3 Vergleiche 5 benötigt.

# Beispiel: Einfügen/Suche in einem Suchbaum

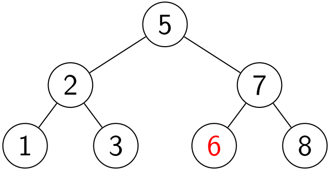
Für fügen der Reihe nach folgende Elemente in einen binären Suchbaum ein: $[5, 2, 3, 7, 6, 8, 1]$




Suche: Um Zu prüfen, ob die Zahl 6 im BST enthalten ist, müssen wir nur der Reihe nach die Elemente $5, 7, 6$ prüfen. In der Liste hätten wir anstatt 3 Vergleiche 5 benötigt.

# Beispiel: Einfügen/Suche in einem Suchbaum

Für fügen der Reihe nach folgende Elemente in einen binären Suchbaum ein: $[5, 2, 3, 7, 6, 8, 1]$




Suche: Um Zu prüfen, ob die Zahl 6 im BST enthalten ist, müssen wir nur der Reihe nach die Elemente $5, 7, 6$ prüfen.
In der Liste hätten wir anstatt 3 Vergleiche 5 benötigt.

# Einfügen in BST

```
ins_BST :: Ord a => a -> Tree a -> Tree a

ins_BST x Empty = leaf x
ins_BST x (Node y l r)
  | x<=y      = Node y (ins_BST x l)            r
  | otherwise = Node y              l  (ins_BST x r)
```

Je nach Größe rekursiv links oder rechts einfügen.
Das neue Element wird hier immer einfach als Blatt angefügt.

```
leaf x = Node x Empty Empty
```

# Suchen in BST

```
elem_BST :: Ord a => a -> Tree a -> Bool
elem_BST x Empty = False
elem_BST x (Node y l r) | x==y = True
                        | x< y = elem_BST x l
                        | x> y = elem_BST x r
```

Der Vergleich mit der Wurzel erlaubt es, die Suche auf einen der Teilbäume zu beschränken.

Je nach Größe rekursiv links oder rechts weitersuchen.

# Löschen aus BST

```
del_BST :: Ord t => t -> Tree t -> Tree t
del_BST x Empty = Empty
del_BST x (Node y l r)     | x<y  = Node y (del_BST x l) r
del_BST x (Node y l r)     | x>y  = Node y l (del_BST x r)
del_BST x (Node y Empty r)| x==y = r
del_BST x (Node y l r)     | x==y = let (z,l1) = del_largest l
                                     in Node z l1 r
  where del_largest (Node x l Empty) = (x,l)
        del_largest (Node x l r) = let (z,r1) = del_largest r
                                     in (z,Node x l r1)
```

- `(z,t1) = del_largest t` entfernt den größten Eintrag `z` aus `t`, der resultierende Baum ist `t1`. Er steht ganz rechts in `t`.
- Muss man die Wurzel eines BST löschen, so kann man sie durch den größten Eintrag des linken Teilbaumes ersetzen ohne die BST Eigenschaft zu verletzen.
- Ist kein linker Teilbaum vorhanden, so kann man die Wurzel direkt entfernen (Ergebnis ist der rechte Teilbaum).

# Laufzeitbetrachtungen für BST

Wir betrachten einen Baum mit $n$ Koten der Höhe $h$:

- Einfügen, Suchen , Löschen in BST benötigt Zeit $O(h)$
- Ist der Baum gut **balanciert**, so ist die Tiefe ungefähr gleich dem Zweierlogarithmus der Knotenzahl $O\big(\log n\big)$
- Besteht der Baum nur aus einem Ast, so ist die Tiefe gleich der Knotenzahl.
- Durch geeignete Umstrukturierungen kann man erreichen, dass ein BST im wesentlichen ausgeglichen bleibt, gleich welche Einträge in ihn eingefügt und aus ihm gelöscht werden und in welcher Reihenfolge.
- Solche BST bilden eine effiziente Implementierung von **Mengen** (`Data.Set`) und **endlichen Abbildungen** (`Data.Map`).

# Datenstruktur Menge

Im Vergleich zu eine Liste besitzt eine **Menge** keine Reihenfolge; Elemente können nicht doppelt vorkommen.

## Funktionalität

- Elemente einfügen
- Elemente suchen
- Elemente löschen
- Elemente verarbeiten (`Foldable`)

Implementierung direkt durch binäre Suchbäume!
Inorder Tiefendurchlauf liefert alle Elemente der Größe nach.

## Menge vs Liste

| | Liste | Menge |
|---|---|---|
| Einfügen | $O(1)$ | $O(\log n)$ |
| Suchen | $O(n)$ | $O(\log n)$ |
| Löschen | $O(n)$ | $O(\log n)$ |

# Datenstruktur Abbildung

Endliche Abbildung sind eine sehr wichtige Datenstruktur, welche wir in Modul `Data.Map` schon kennengelernt haben.

## Funktionalität

- Schlüssel-Wert Paare einfügen
- Schlüssel suchen, Wert liefern
- Schlüssel löschen
- Elemente verarbeiten (`Foldable`)

Funktionalität, Effizient und Implementierung praktisch wie bei Mengen:

```
lookup_BST :: Ord a => a -> Tree (a,b) -> Maybe b
lookup_BST x Empty = Nothing
lookup_BST x (Node (k,v) l r) | x==k = Just v
                              | x< k = search_BST x l
                              | x> k = search_BST x r
```

# Datenstruktur Abbildung

Endliche Abbildung sind eine sehr wichtige Datenstruktur, welche
wir in

Funk

- S
- S
- S

Funkti

Menge

looku
looku
looku

Alternativ kann man auch einen speziellen Datentyp für Schlüssel-Wert-Paare einführen:

```
newtype KV k v = KV (k,v)
instance Eq k => Eq (KV k v) where
  KV (k1,_) == KV (k2,_) = k1 == k2
instance Ord k => Ord (KV k v) where
  KV (k1,_) <= KV (k2,_) = k1 <= k2

type Map k v = Tree (KV k v)

insert_BST :: Ord k => k -> v -> Map k v -> Map k v
insert_BST k v t = ins_BST (KV (k,v)) t

lookup_BST :: Ord k => k -> Map k v -> Maybe v
lookup_BST _ Empty = Nothing
lookup_BST x (Node (KV (k,v)) l r)
  | x==k = Just v
  | x< k = lookup_BST x l
  | x> k = lookup_BST x r
```

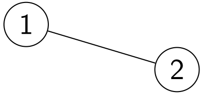
l
r

# Beispiel: Einfügen in einem Suchbaum (Teil 2)

Für fügen der Reihe nach folgende Elemente in einen binären Suchbaum ein: $[1, 2, 3, 4, 5, 6, 7]$

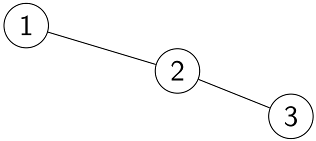
# Beispiel: Einfügen in einem Suchbaum (Teil 2)

Für fügen der Reihe nach folgende Elemente in einen binären Suchbaum ein: $[1, 2, 3, 4, 5, 6, 7]$



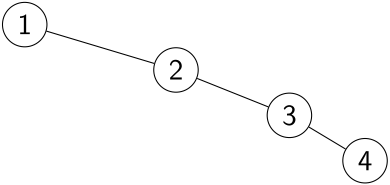
# Beispiel: Einfügen in einem Suchbaum (Teil 2)

Für fügen der Reihe nach folgende Elemente in einen binären Suchbaum ein: $[1, 2, 3, 4, 5, 6, 7]$



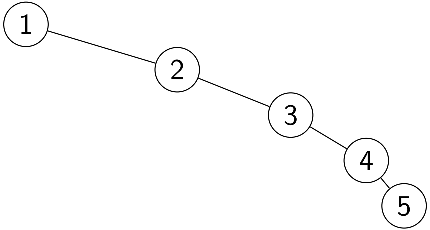
# Beispiel: Einfügen in einem Suchbaum (Teil 2)

Für fügen der Reihe nach folgende Elemente in einen binären Suchbaum ein: $[1, 2, 3, 4, 5, 6, 7]$



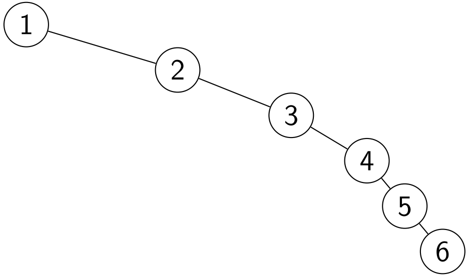
# Beispiel: Einfügen in einem Suchbaum (Teil 2)

Für fügen der Reihe nach folgende Elemente in einen binären Suchbaum ein: $[1, 2, 3, 4, 5, 6, 7]$



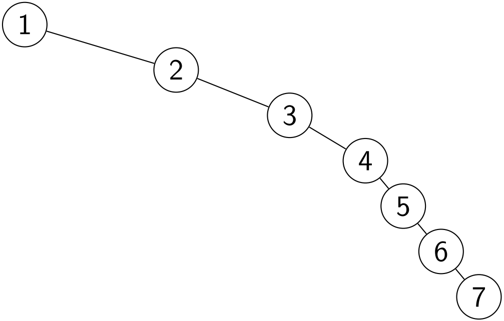
# Beispiel: Einfügen in einem Suchbaum (Teil 2)

Für fügen der Reihe nach folgende Elemente in einen binären Suchbaum ein: $[1, 2, 3, 4, 5, \textcolor{red}{6}, 7]$



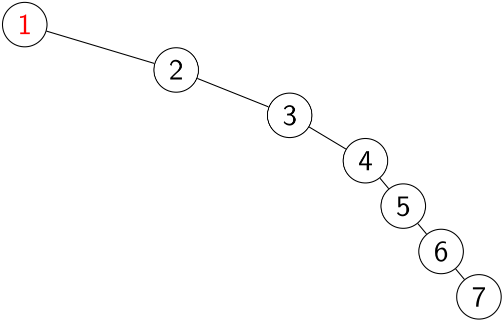
# Beispiel: Einfügen in einem Suchbaum (Teil 2)

Für fügen der Reihe nach folgende Elemente in einen binären Suchbaum ein: $[1, 2, 3, 4, 5, 6, 7]$



# Beispiel: Einfügen in einem Suchbaum (Teil 2)

Für fügen der Reihe nach folgende Elemente in einen binären Suchbaum ein: $[1, 2, 3, 4, 5, 6, 7]$

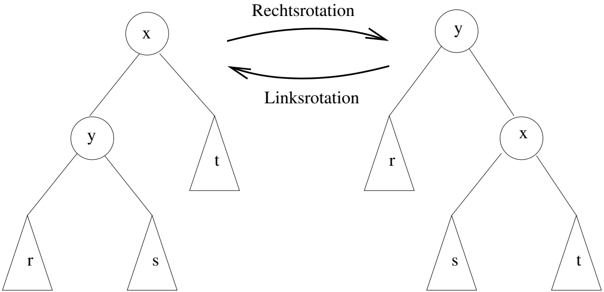



Problem: Höhe des Baumes ist gleich zur Anzahl der Elemente

$\Rightarrow$ Keine Einsparungen bei windschiefen Bäumen!
Suchbäume sollten daher immer **balanciert** sein!

# Rotationen

Geschickte Rotationen erhalten die Suchbaumeigenschaft:




```
rot_right (Node x (Node y r s) t) =
                        Node y r (Node x s t)
rot_left  (Node y r (Node x s t)) =
                        Node x (Node y r s) t
```

# Verwaltung der Rotationen

Um festzustellen, wann rotiert werden muss, möchte man nicht global den ganzen Baum analysieren. Zu teuer!

Stattdessen führt man in den Knoten geeignete Verwaltungsinformation mit:

- Höhendifferenz zwischen linkem und rechten Teilbaum abspeichern und jeweils im Bereich $\{-1, 0, 1\}$ halten. *(AVL-Bäume)*
- Ein Bit ("rot", "schwarz") abspeichern, sodass auf jedem Pfad von einem Knoten zu einem Blatt dieselbe Zahl schwarzer Knoten liegt und auf jeden schwarzen Knoten höchstens ein roter Knoten folgt. *(Rot-Schwarz-Bäume)*
- Man kann auch ganz ohne Verwaltungsinformation auskommen, wenn man auch bei den Suchoperationen rotiert. *(Splay-Bäume)*

# Zusammenfassung Laufzeitbetrachtungen

- Empirische Laufzeitmessung für Listenfunktionen
- Erklärung der beobachteten Laufzeit anhand eines Speichermodells für Listen: Laufzeit der Konkatenation `l++k` linear in der Länge von `l`
- *Sortieren durch Mischen* als Beispiel eines effizienten ($O(n \log n)$) Sortierverfahrens
- *Warteschlange:* Effizientere Variante einer Liste, bei der beide Enden bearbeitet werden können.
- Effizientere Version der Breitensuche durch "amortisierte" Warteschlange und entsprechende Laufzeitanalyse.
- *Menge:* Effizientere Operation für "enthalten-sein", im Gegensatz zur List keine Reihenfolge
- *Binäre Suchbäume* als effiziente Implementierung von Mengen und endlichen Abbildungen, so fern diese balanciert sind.