Funktionales Programmieren
(Praktische Informatik 3)

Berthold Hoffmann
Studiengang Informatik
Universität Bremen

Winter 03/04

Vorlesung vom 01.12.2003: Abstrakte Datentypen

Inhalt

- Letzte VL:
  - Datenabstraktion durch algebraische Datentypen
  - information hiding fehlt
- Heute: abstrakte Datentypen (ADTs) in Haskell
- Beispiele für bekannte ADTs:
  - `Store`
  - Stapel und Schlangen: `Stack` und `Queue`
  - Mengen: `Set`

Abgeleitete Klasseninstanzen

- Wie würde man Gleichheit auf `Shape` definieren?

```
Circ p1 i1 == Circ p2 i2 = p1 == p2 && i1 == i2
Rect p1 q1 == Rect p2 q2 = p1 == p2 && q1 == q2
Poly ps    == Poly qs    = ps == qs
_          == _          = False
```

- Schematisierbar:
  - Gleiche Konstruktoren mit gleichen Argumenten sind gleich,
  - alles andere ungleich.
  - Automatisch generiert mit `data Shape = .. deriving Eq`
  - Ähnlich `deriving (Ord, Show, Read)`

Wohlbekannte algebraische Datentypen

- Wahrheitswerte sind ein Aufzählungstyp

```
data Bool = False | True
```

- Tupel sind Produkttypen

```
data (a,b) = (a,b)
data (a,b,c) = (a,b,c)
data (a,b,c,d) = (a,b,c,d)
...
```

- Listen sind rekursive algebraische Datentypen

```
data [a] = [] | a : [a]
```

Abstrakte Datentypen

Ein abstrakter Datentyp besteht aus
einem Typen und Operationen darauf.

- Beispiele:
  - Speicher, mit Operationen lesen und schreiben;
  - Stapel, mit Operationen `push`, `pop`, `top`;
  - Schlangen, mit Operationen `enq`, `enq`, `deq`;
  - Mengen, mit leerer Menge, einfügen, löschen;
- Die Darstellung des Typen wird versteckt.

Modul in Haskell

- Einschränkung der Sichtbarkeit durch Verkapselung
- Modul: Kleinste verkapselbare Einheit
- Ein Modul umfaßt:
  - Definitionen von Typen, Funktionen, Klassen
  - Angabe der nach außen sichtbaren Definitionen
- Syntax:

  `module` *Name* [`(`*sichtbare Bezeichner*`)`] `where` *Rumpf*

  - `(`*sichtbare Bezeichner*`)` kann fehlen
  - Gleichzeitig: Übersetzungseinheit (getrennte Übersetzung)

Beispiel: ein einfacher Speicher (`Store`)

- Typ `Store a b`, parametrisiert über
  - Indextyp `a` (muß Gleichheit, oder besser Ordnung, zulassen)
  - Werttyp `b`
- Konstruktor: leerer Speicher `initial :: Store a b`
- Wert lesen: `value :: Store a b-> a-> Maybe b`
  - Der Wert ist möglicherweise undefiniert.
- Wert schreiben:
  `update :: Store a b-> a -> b-> Store a b`

Moduldeklaration

- Moduldeklaration

```
module Store(
    Store,    -- Eq a => Store a b
    initial,  -- Store a b
    value,    -- Store a b-> a-> Maybe b
    update,   -- Store a b-> a-> b-> Store a b
    ) where
```

- Die Signaturen sind nicht nötig, aber sehr hilfreich.

Erste Implementierung

- Speicher als Liste von Paaren (als `data`, nicht `type`)

```
data Store a b = St [(a, b)]
```

- Leerer Speicher: leere Liste

```
initial  = St []
```

- Lookup

```
value (St          []  ) x = Nothing
value (St ((a,b):st)) x = if a == x then Just b
                          else value (St st) x
```

- Update: neues Paar vorne anhängen

```
update (St ls) a b  = St ((a, b): ls)
```

Test

Zweite Implementierung

- Speicher als Funktion

```
data Store a b = St (a-> Maybe b)
```

- Leerer Speicher: konstant undefiniert

```
initial  = St (const Nothing)
```

- Lookup: Funktion anwenden

```
value (St f) a  = f a
```

- Update: punktweise Funktionsdefinition

```
update (St f) a b
  = St (\x-> if x== a then Just b else f x)
```

Ein Interface, zwei mögliche Implementierungen.

Export von Datentypen

- `...,T(..),...` exportiert auch die Konstruktoren von `T`
  - Darstellung und Implementierung sichtbar
  - Pattern matching möglich
- `...,T,...` exportiert nur den Typ `T`
  - Implementierung nicht sichtbar
  - Kein pattern matching möglich
- Typsynonyme sind immer sichtbar
- Kritik an Haskell:
  - Exportsignatur erscheint nicht im Kopf (außer als Kommentar)
  - Schnittstelle und Implementierung in der gleichen Datei!

Benutzung eines ADTs — Import.

`import [qualified]` *Name* `[hiding]` (*Bezeichner*)

- Ohne (*Bezeichner*) wird alles importiert
- `qualified`: importierte Namen müssen qualifiziert werden

```
import Store2 qualified
f = Store2.initial
```

- `hiding`: Liste der (*Bezeichner*) wird versteckt

```
import Prelude hiding (foldr)
foldr f e ls = ...
```

- Mit `qualified` und `hiding` Namenskonflikte auflösen.

Schnittstelle vs. Semantik

- Stacks
  - Typ: `St a`
  - Initialwert:

    ```
    empty :: St a
    ```

  - Wert ein/auslesen

    ```
    push :: a-> St a-> St a
    top  :: St a-> a
    pop  :: St a-> St a
    ```

  - Test auf Leer

    ```
    isEmpty :: St a-> Bool
    ```

  - Last in, first out.

Gleiche Signatur

- Queues
  - Typ: `Qu a`
  - Initialwert:

    ```
    empty :: Qu a
    ```

  - Wert ein/auslesen

    ```
    enq   :: a-> Qu a-> Qu a
    first :: Qu a-> a
    deq   :: Qu a-> Qu a
    ```

  - Test auf Leer

    ```
    isEmpty :: Qu a-> Bool
    ```

  - First in, first out.

unterscheidliche Semantik

Implementation von `Stack`: Liste

- Sehr einfach wg. last in, first out
- `empty` ist `[]`
- `push` ist `(:)`
- `top` ist `head`
- `pop` ist `tail`
- `isEmpty` ist `null`

Implementation von `Queue`

- Mit einer Liste?
  - Problem: am Ende anfügen oder abnehmen ist teuer.
- Deshalb zwei Listen:
  - Erste Liste: zu entnehmende Elemente
  - Zweite Liste: hinzugefügte Elemente rückwärts
  - Invariante: erste Liste ist leer gdw. die Queue ist leer

- Beispiel:

  Operation Queue Darstellung

- Beispiel:

Operation	Queue	Darstellung
`empty`		`([], [])`

- Beispiel:

Operation	Queue	Darstellung
`empty`		`([], [])`
`enq 9`	9	`([9], [])`

- Beispiel:

Operation	Queue	Darstellung
`empty`		`([], [])`
`enq 9`	9	`([9], [])`
`enq 4`	$9 \longrightarrow 4$	`([9], [4])`

- Beispiel:

Operation	Queue	Darstellung
`empty`		`([], [])`
`enq 9`	9	`([9], [])`
`enq 4`	$9 \longrightarrow 4$	`([9], [4])`
`deq`	4	`([4], [])`

- Beispiel:

Operation	Queue	Darstellung
`empty`		`([], [])`
`enq 9`	9	`([9], [])`
`enq 4`	$9 \longrightarrow 4$	`([9], [4])`
`deq`	4	`([4], [])`
`enq 7`	$4 \longrightarrow 7$	`([4], [7])`

- Beispiel:

Operation	Queue	Darstellung
`empty`		`([], [])`
`enq 9`	9	`([9], [])`
`enq 4`	$9 \longrightarrow 4$	`([9], [4])`
`deq`	4	`([4], [])`
`enq 7`	$4 \longrightarrow 7$	`([4], [7])`
`enq 5`	$4 \longrightarrow 7 \longrightarrow 5$	`([4], [5, 7])`

- Beispiel:

Operation	Queue	Darstellung
`empty`		`([], [])`
`enq 9`	9	`([9], [])`
`enq 4`	$9 \longrightarrow 4$	`([9], [4])`
`deq`	4	`([4], [])`
`enq 7`	$4 \longrightarrow 7$	`([4], [7])`
`enq 5`	$4 \longrightarrow 7 \longrightarrow 5$	`([4], [5, 7])`
`deq`	$7 \longrightarrow 5$	`([7, 5], [])`

- Beispiel:

Operation	Queue	Darstellung
`empty`		`([], [])`
`enq 9`	9	`([9], [])`
`enq 4`	$9 \longrightarrow 4$	`([9], [4])`
`deq`	4	`([4], [])`
`enq 7`	$4 \longrightarrow 7$	`([4], [7])`
`enq 5`	$4 \longrightarrow 7 \longrightarrow 5$	`([4], [5, 7])`
`deq`	$7 \longrightarrow 5$	`([7, 5], [])`
`deq`	7	`([5], [])`

- Beispiel:

Operation	Queue	Darstellung
`empty`		`([], [])`
`enq 9`	9	`([9], [])`
`enq 4`	$9 \longrightarrow 4$	`([9], [4])`
`deq`	4	`([4], [])`
`enq 7`	$4 \longrightarrow 7$	`([4], [7])`
`enq 5`	$4 \longrightarrow 7 \longrightarrow 5$	`([4], [5, 7])`
`deq`	$7 \longrightarrow 5$	`([7, 5], [])`
`deq`	7	`([5], [])`
`deq`		`([], [])`

- Modulkopf und Typdefinition

```
module Queue(Qu, empty, isEmpty,
             enq, first, deq) where


data Qu a = Qu [a] [a]
empty = Qu [] []
```

- Invariante: erste Liste leer gdw. Queue leer

```
isEmpty (Qu xs _) = null xs
```

- Erstes Element steht vorne in erster Liste

```
first (Qu  []    _) = error "Qu: first of empty Q"
first (Qu (x:_) _) = x
```

- Bei `enq` und `deq` wird die Invariante geprüft

```
enq x (Qu xs ys) = check xs (x:ys)

deq (Qu [] _ )      = error "Qu: deq of empty Q"
deq (Qu (_:xs) ys) = check xs ys
```

- `check` bewahrt die Invariante

```
check [] ys = Qu (reverse ys) []
check xs ys = Qu xs ys
```

Mengen

- Eine Menge ist Sammlung von Elementen so dass
  - kein Element doppelt ist, und
  - die Elemente nicht angeordnet sind.
- Operationen:
  - leere Menge und Test auf leer,
  - Element einfügen,
  - Element löschen,
  - Test auf Enthaltensein,
  - Elemente aufzählen.

Mengen — Schnittstelle

- Typ `Set a`
  - Typ `a` mit Gleichheit, besser Ordnung.

```
module Set (Set,
  empty,    -- Set a
  isEmpty,  -- Set a-> Bool
  insert,   -- Ord a=> Set a-> a-> Set a
  remove,   -- Ord a=> Set a-> a-> Set a
  elem,     -- Ord a=> a-> Set a-> Bool
  enum      -- Ord a=> Set a-> [a]
  ) where
```

Mengen als Geordnete Bäume

- Voraussetzung:
  - Ordnung auf `a` (`Ord a`)
  - Es soll für alle `Node a l r` gelten:

    $$\texttt{member x l} \Rightarrow \texttt{x} < \texttt{a} \wedge \texttt{member x r} \Rightarrow \texttt{a} < \texttt{x}$$

    Jeder Eintrag kommt höchstens einmal vor

```
type Set a = Tree a
data Tree a = Null
            | Node (Tree a) a (Tree a)
```

- Die leere Menge, und Test:

```
empty =  Null

isEmpty Null = True
isEmpty _    = False
```

- Test auf Enthaltensein:

```
elem Null _ = False
elem (Node l a r) b
   | b < a  = elem l b
   | a == b = True
   | b > a  = elem r b
```

- Ordnungserhaltendes Einfügen

```
insert Null a = Node Null a Null
insert (Node l a r) b
  | b < a  = Node (insert l b) a r
  | b == a = Node l a r
  | b > a  = Node l a (insert r b)
```

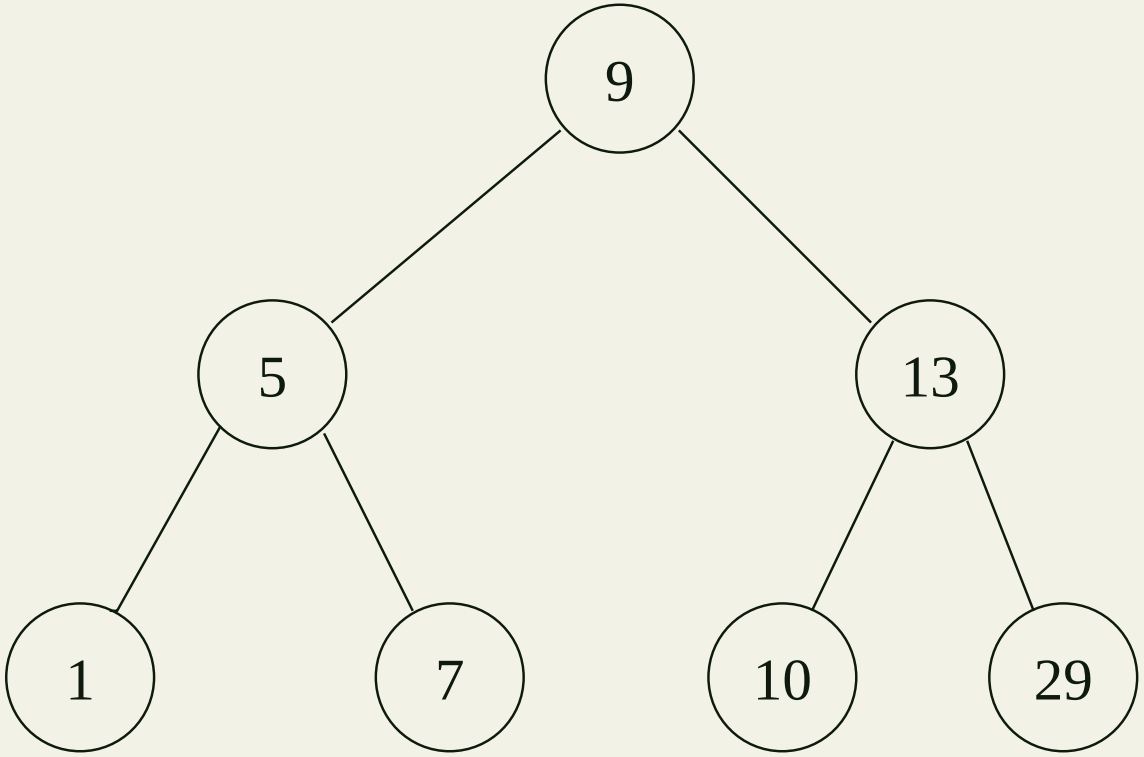
- Problem: `insert` funktioniert nur für geordnete Bäume.
  - Geordnete Bäume dürfen nur mit `insert` erzeugt werden.
    Nicht mit dem Konstruktor `Node`!

- Löschen:

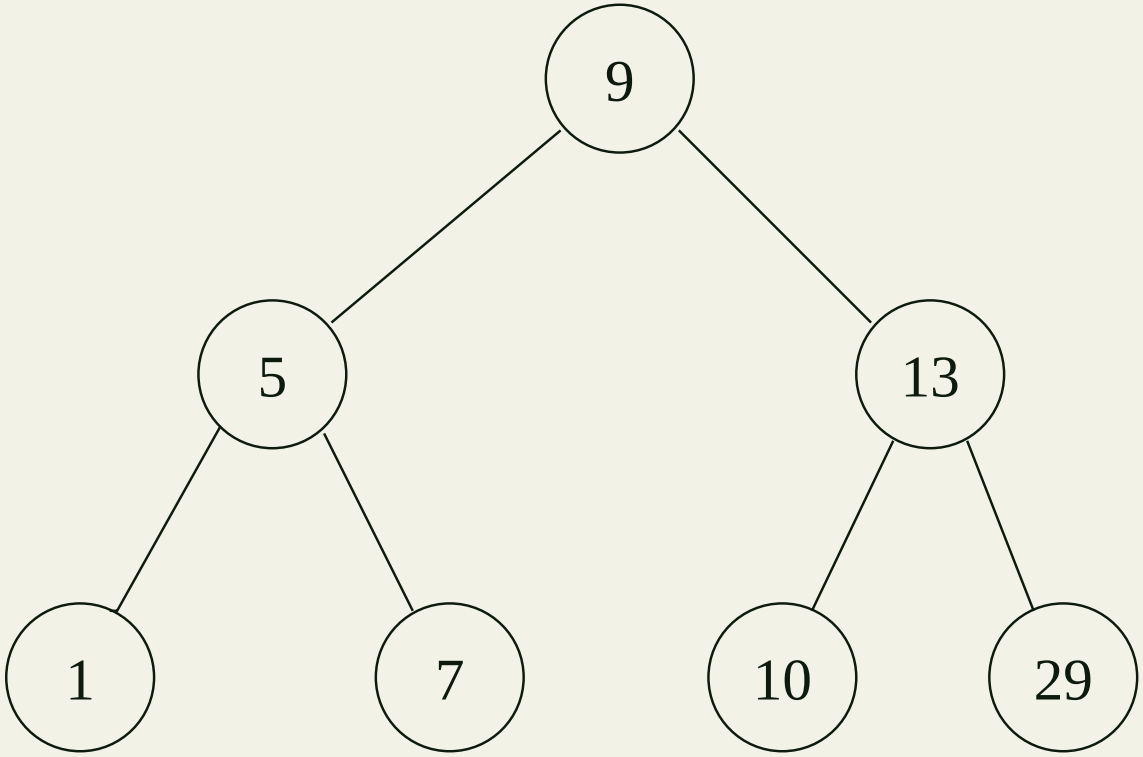
```
remove  Null          x = Null
remove (Node l y r) x
   | x < y  = Node (remove l x) y r
   | x == y = join l r
   | x > y  = Node l y (remove r x)
```

- `join` fügt zwei Bäume ordnungserhaltend zusammen.

- Beispiel: Gegeben folgender Baum, dann `remove t 9`

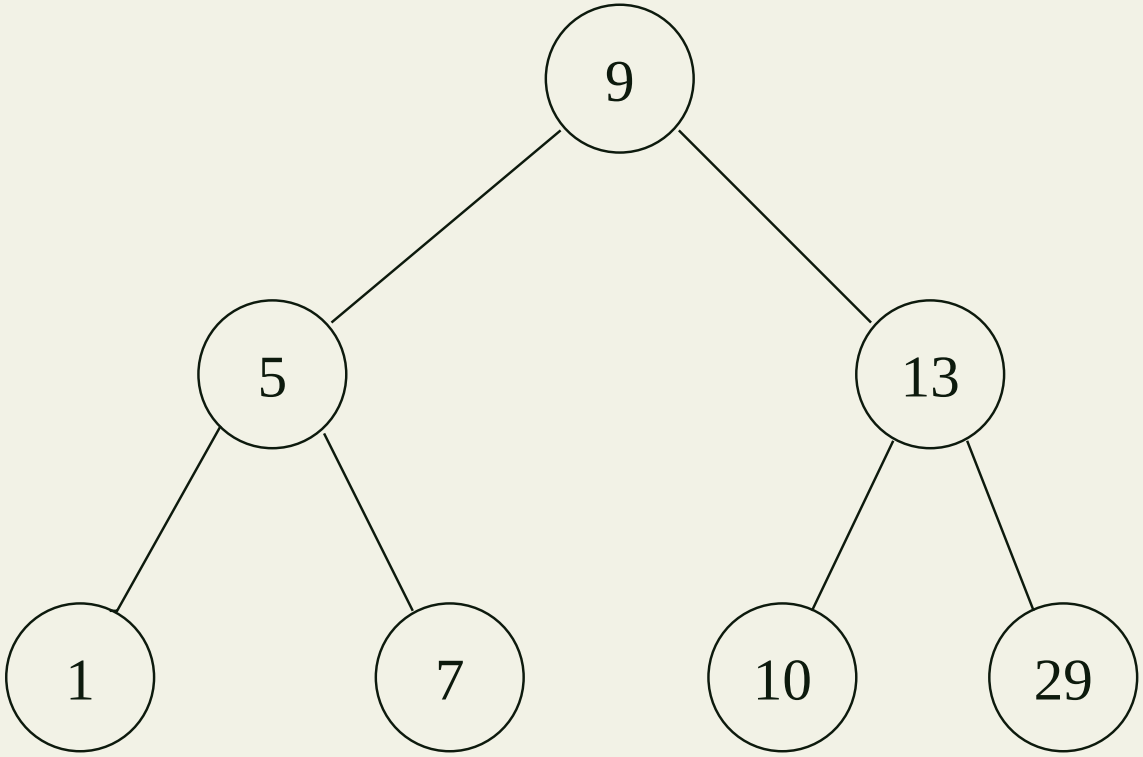


- Beispiel: Gegeben folgender Baum, dann `remove t 9`



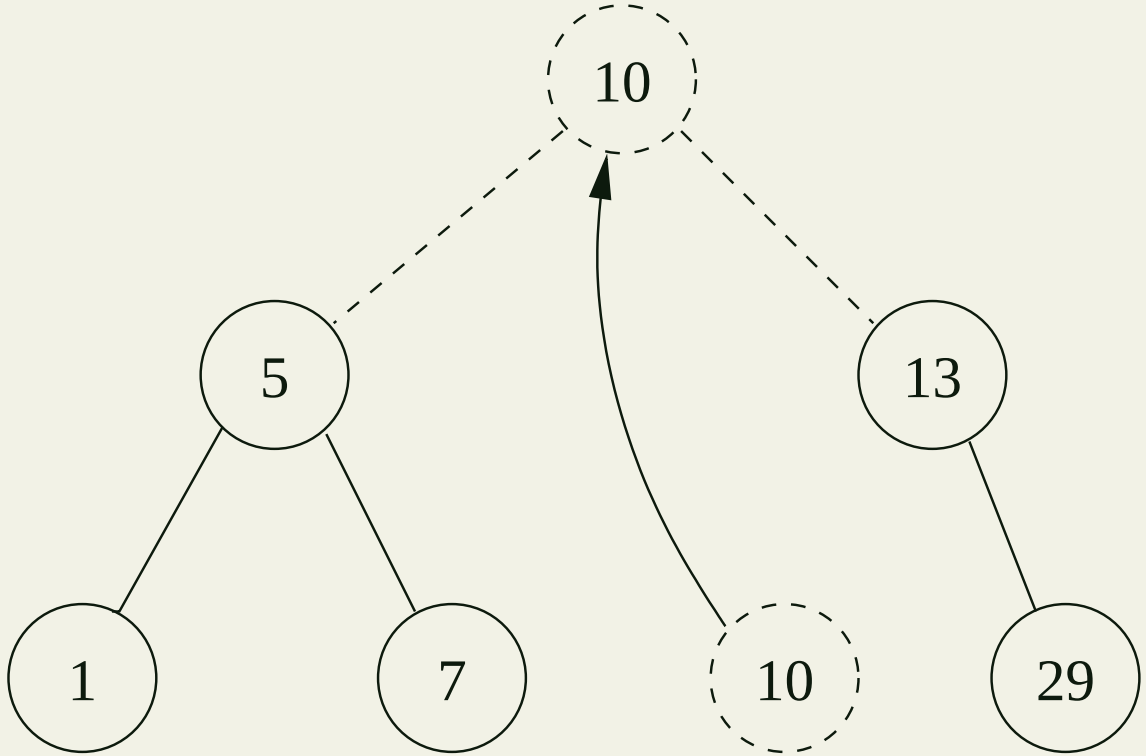
  - Wurzel wird gelöscht

- Beispiel: Gegeben folgender Baum, dann `remove t 9`



  - Wurzel wird gelöscht
  - `join` muß Bäume ordnungserhaltend zusammenfügen

- `join` fügt zwei Bäume ordnungserhaltend zusammen.



  - Wurzel des neuen Baums: Knoten links unten im rechten Teilbaum (oder Knoten rechts unten im linken Teilbaum)

- Implementierung:
  - `splitTree` spaltet Baum in Knoten links unten und Rest.

```
join ::  Tree a-> Tree a-> Tree a
join xt Null = xt
join xt yt   = Node xt u nu where
  (u, nu) = splitTree yt
  splitTree :: Tree a-> (a, Tree a)
  splitTree (Node Null a t) = (a, t)
  splitTree (Node lt a rt) =
    (u, Node nu a rt) where
       (u, nu) = splitTree lt
```

- `enum` wandelt einen geordneten Baum in eine Liste um.

```
enum  Null          = []
enum (Node l a r) = (enum l) ++ [a] ++ (enum r)
```

- Testen

Vertiefung: Mengen als AVL-Bäume

- Implementation durch balancierte Bäume
  - AVL-Bäume =

```
type Set a = AVLTree a
```

- Ein Baum ist ausgeglichen, wenn
  - alle Unterbäume ausgeglichen sind, und
  - der Höhenunterschied zwischen zwei Unterbäumen höchstens eins beträgt.

- Im Knoten Höhe des Baumes an dieser Stelle merken

```
data AVLTree a = Null
               | Node Int
                      (AVLTree a)
                      a
                      (AVLTree a)
```

- Ausgeglichenheit ist Invariante.
- Alle Operationen müssen Ausgeglichenheit bewahren.
- Bei Einfügen und Löschen ggf. rotieren.

Simple Things First

- Leere Menge = leerer Baum

```
empty :: AVLTree a
empty = Null
```

- Test auf leere Menge:

```
isEmpty :: AVLTree a-> Bool
isEmpty Null = True
isEmpty _    = False
```

Hilfsfunktionen

- Höhe eines Baums
  - Aus Knoten selektieren, nicht berechnen.

```
ht :: AVLTree a-> Int
ht Null              = 0
ht (Node h _ _ _) = h
```

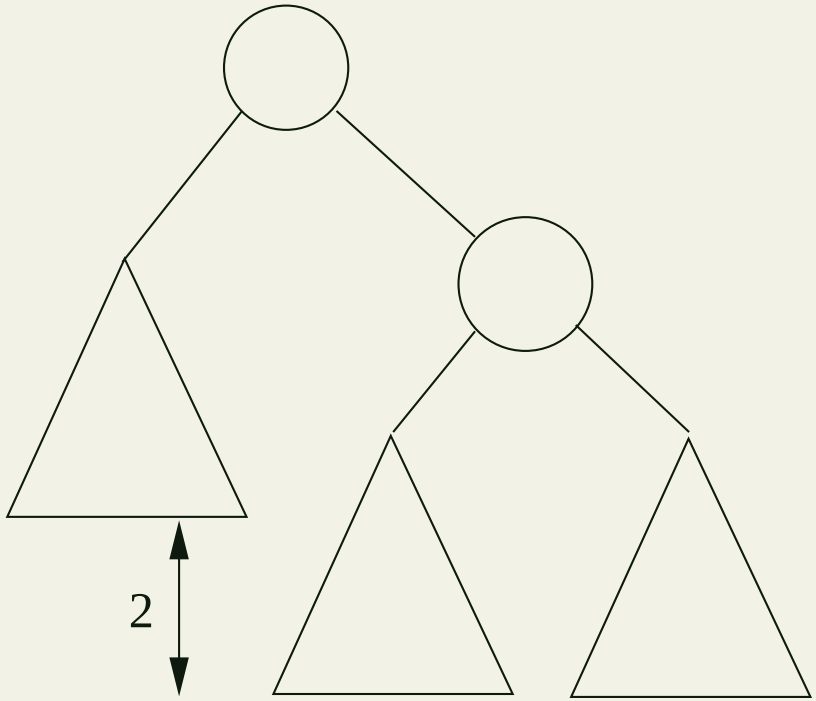
- Neuen Knoten anlegen, Höhe aus Unterbäumen berechnen.

```
mkNode :: AVLTree a-> a-> AVLTree a-> AVLTree a
mkNode l n r = Node h l n r where
                    h = 1+ max (ht l) (ht r)
```

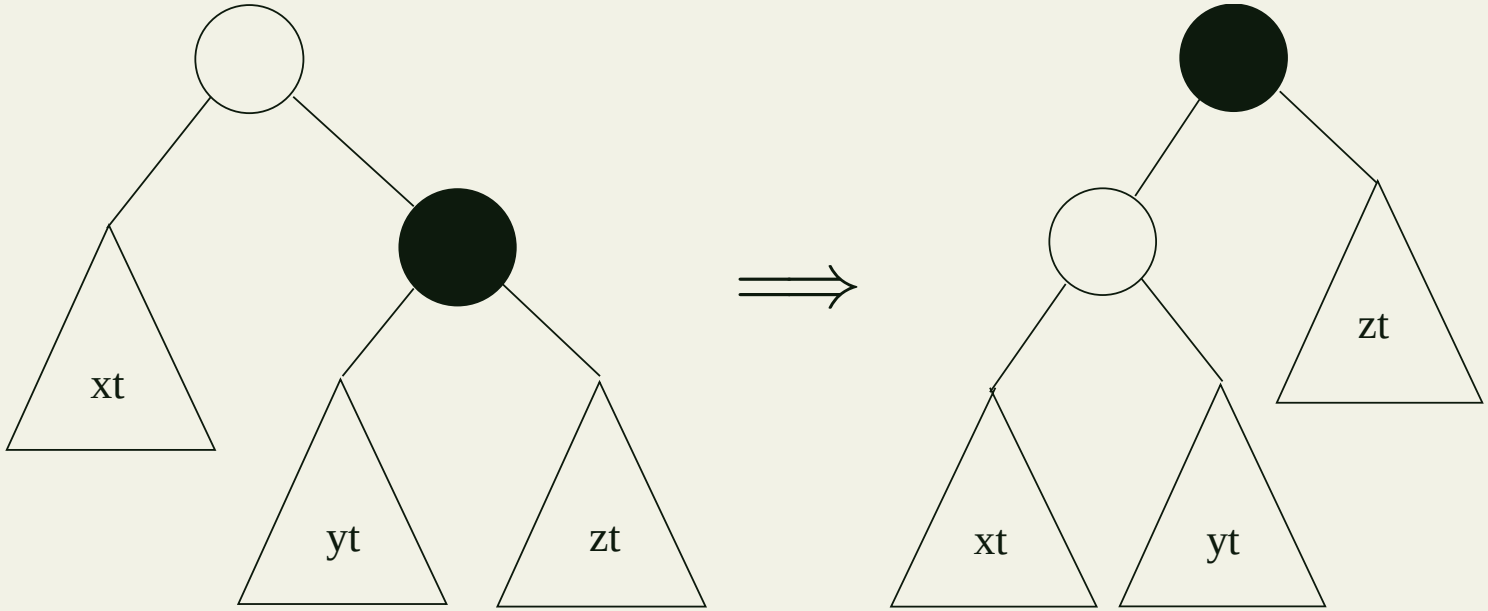
Ausgeglichenheit sicherstellen

- Problem:
  Nach Löschen oder
  Einfügen zu großer
  Höhenunterschied

- Lösung:
  Rotieren der Unterbäume

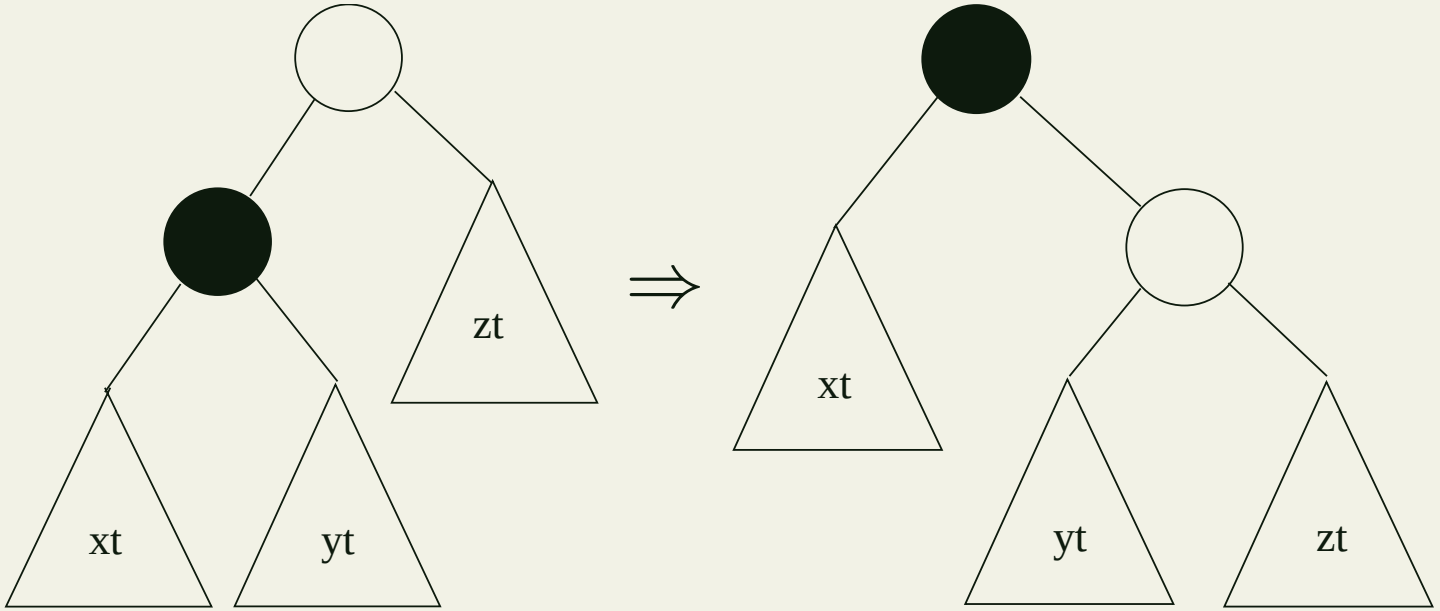


Linksrotation




```
rotl :: AVLTree a-> AVLTree a
rotl (Node _ xt y (Node _ yt x zt)) =
  mkNode (mkNode xt y yt) x zt
```

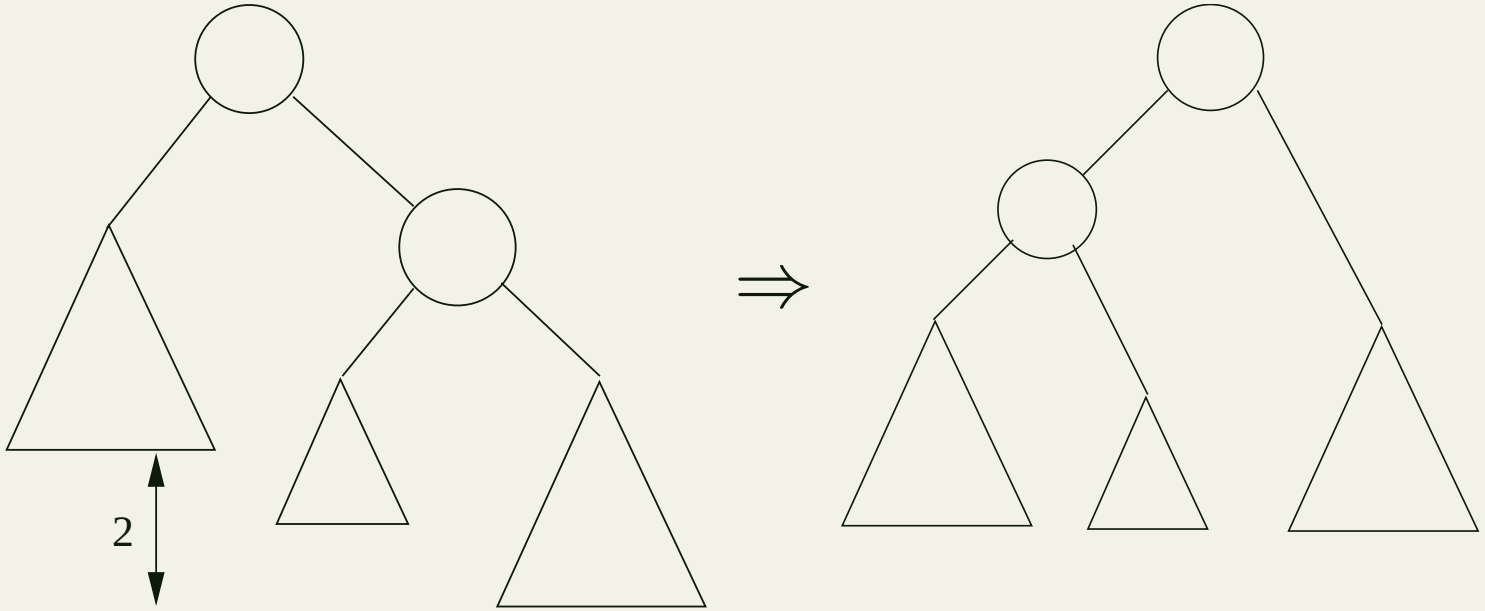
Rechtsrotation




```
rotr :: AVLTree a-> AVLTree a
rotr (Node _ (Node _ xt y yt) x zt) =
  mkNode xt y (mkNode yt x zt)
```

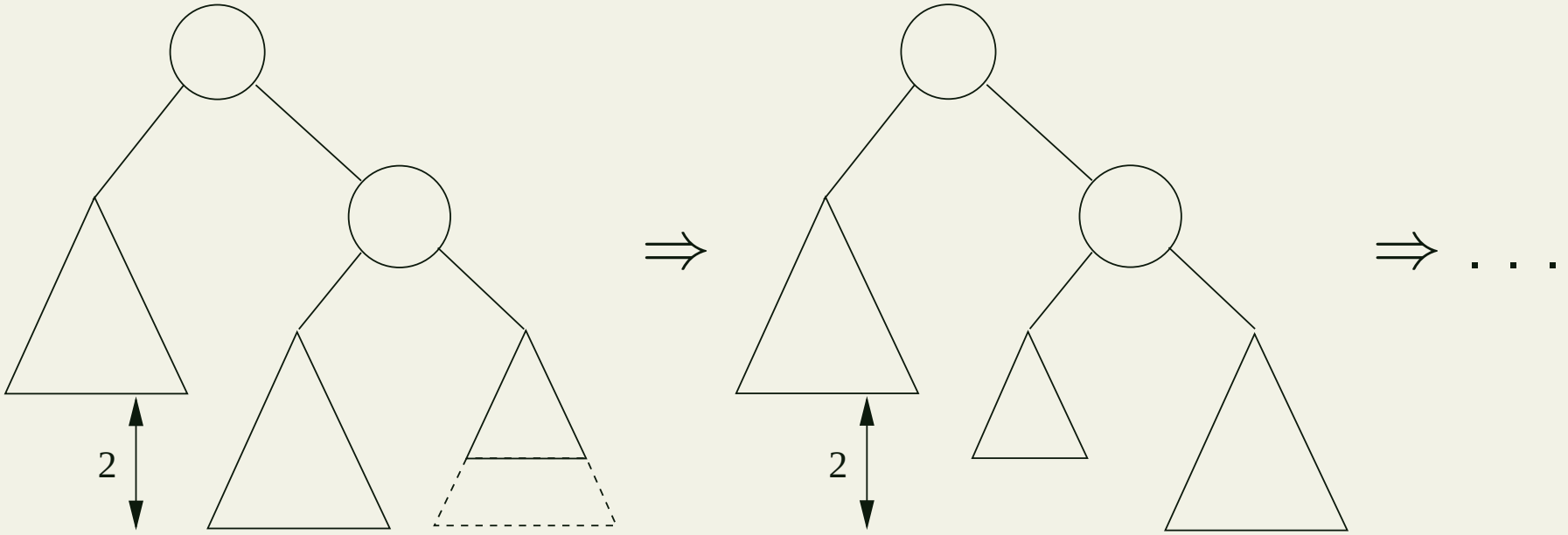
Ausgeglichenheit sicherstellen

- Fall 1: Unterbaum rechts außen zu hoch
- Lösung: Linksrotation



Ausgeglichenheit sicherstellen

- Fall 2: Unterbaum rechts innen zu hoch oder gleich hoch
- Ausgleichen des rechten Unterbaumes (durch Rechtsrotation)



Ausgeglichenheit sicherstellen

- Hilfsfunktion: Balance eines Baumes

```
bias :: AVLTree a-> Int
bias (Node _ lt _ rt) = ht lt - ht rt
```

- Unterscheidung der Fälle
  - Fall 1: `bias` < 0
  - Fall 2: `bias` ≥ 0
- Symmetrischer Fall (linker Teilbaum zu hoch)
  - Fall 1: `bias` > 0
  - Fall 2: `bias` ≤ 0

- `mkAVL xt y zt`:
  - Voraussetzung: Höhenunterschied `xt`, `zt` höchstens zwei;
  - Konstruiert neuen AVL-Baum mit Knoten `y`.

```
mkAVL :: AVLTree a-> a-> AVLTree a-> AVLTree a
mkAVL xt y zt
   | hx+1< hz &&
     0<= bias zt = rotl (mkNode xt y (rotr zt))
   | hx+1< hz    = rotl (mkNode xt y zt)
   | hz+1< hx &&
     bias xt<= 0 = rotr (mkNode (rotl xt) y zt)
   | hz+1< hx    = rotr (mkNode xt y zt)
   | otherwise   = mkNode xt y zt
     where hx= ht xt; hz= ht zt
```

Ordnungserhaltendes Einfügen

- Erst Knoten einfügen, dann ggf. rotieren:

```
insert :: Ord a=> AVLTree a-> a-> AVLTree a
insert Null a = mkNode Null a Null
insert (Node n l a r) b
   | b < a  = mkAVL (insert l b) a r
   | b == a = Node n l a r
   | b > a  = mkAVL l a (insert r b)
```

- `mkAVL` garantiert Ausgeglichenheit.

Löschen

- Erst Knoten löschen, dann ggf. rotieren:

```
remove :: Ord a=> AVLTree a-> a-> AVLTree a
remove Null x = Null
remove (Node h l y r) x
   | x < y  = mkAVL (remove l x) y r
   | x == y = join l r
   | x > y  = mkAVL l y (remove r x)
```

- `mkAVL` garantiert Ausgeglichenheit.

- `join` fügt zwei Bäume ordnungserhaltend zusammen (s. letzte VL)

```
join ::  AVLTree a-> AVLTree a-> AVLTree a
join xt Null = xt
join xt yt   = mkAVL xt u nu where
  (u, nu) = splitTree yt
  splitTree :: AVLTree a-> (a, AVLTree a)
  splitTree (Node h Null a t) = (a, t)
  splitTree (Node h lt a rt) =
    (u, mkAVL nu a rt) where
       (u, nu) = splitTree lt
```

Menge aufzählen

```
foldAVL :: (Int-> b-> a-> b-> b)-> b-> AVLTree a-> b
foldAVL f e Null = e
foldAVL f e (Node h l a r) =
   f h (foldAVL f e l) a (foldAVL f e r)
```

- Aufzählen der Menge: Inorder-Traversion (via `fold`)

```
enum :: Ord a=> AVLTree a-> [a]
enum = foldAVL (\h t1 x t2-> t1++ [x]++ t2) []
```

- Enthaltensein: Testen.

```
elem :: Ord a=> a-> AVLTree a-> Bool
elem e = foldAVL (\h b1 x b2-> e == x || b1 || b2)
```

Zusammenfassung

- Abstrakter Datentyp: Datentyp plus Operationen.
- Module in Haskell:
  - Module begrenzen Sichtbarkeit
  - Importieren, ggf. qualifiziert oder nur Teile
- Beispiele für ADTs:
  - Speicher: mehrere Implementierungen
  - Stapel und Schlangen: gleiche Signatur, andere Semantik
  - Implementierung von Schlangen durch zwei Listen
  - Mengen: Implementierung durch (ausgeglichene) Bäume