Vorlesung vom 06.12.2004
Abstrakte Datentypen

Inhalt

- Letzte VL:
  - Datenabstraktion durch algebraische Datentypen
  - Problem mit geordneten Bäumen: Konstruktoren sichtbar
- Heute: abstrakte Datentypen (ADTs) in Haskell
- Beispiele für bekannte ADTs:
  - Stapel und Schlangen: `Stack` und `Queue`
  - Endliche Mengen: `Set`
  - Assoziationslisten (endliche Abbildungen): `Finite Map`
  - Graphen

Abstrakte Datentypen

Ein abstrakter Datentyp besteht aus
einem Typ und Operationen darauf.

Beispiele:

- geordnete Bäume, mit leerer Baum, einfügen, löschen, suchen;
- Speicher, mit Operationen lesen und schreiben;
- Stapel, mit Operationen `push`, `pop`, `top`;
- Schlangen, mit Operationen einreihen, Kopf, nächstes Element;

Repräsentation des Typen versteckt.

Module in Haskell

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

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

  - *sichtbare Bezeichner* können leer sein
  - Gleichzeitig: Übersetzungseinheit (getrennte Übersetzung)

Beispiel: ein einfacher Speicher (`Store`)

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

Moduldeklaration

- Moduldeklaration

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

- Signaturen nicht nötig, aber sehr hilfreich.

Erste Implementation

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

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

- Leerer Speicher: leere Liste

```
initial  = Store []
```

- Lookup (Neu: `case` für Fallunterscheidungen):

```
value (Store ls) a = case filter ((a ==).fst) ls of
                       (_, x):_ -> Just x
                       []       -> Nothing
```

- Update: neues Paar vorne anhängen Test.

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

Zweite Implementation

- Speicher als Funktion

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

- Leerer Speicher: konstant undefiniert

```
initial  = Store (const Nothing)
```

- Lookup: Funktion anwenden

```
value (Store f) a  = f a
```

- Update: punktweise Funktionsdefinition

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

Ein Interface, zwei Implementierungen:

```
module Store (Store, initial, value, update) where

data Store a b =
  Store [(a, b)]
initial  =
  Store []
value (Store ls) a = case
  filter ((a ==).fst)ls of
    (_, x):_ -> Just x
    []       -> Nothing

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

```
data Store a b =
  Store (a-> Maybe b)
initial  =
  Store (const Nothing)
value (Store f) a  = f a


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

Export von Datentypen

- `Store(..)` exportiert Konstruktoren
  - Implementation sichtbar
  - Pattern matching möglich
- `Store` exportiert nur den Typ
  - Implemtation nicht sichtbar
  - Kein pattern matching möglich
- Typsynonyme immer sichtbar
- Kritik Haskell:
  - Exportsignatur nicht im Kopf (nur als Kommentar)
  - Exportsignatur nicht von Implementation zu trennen (gleiche Datei!)

Benutzung eines ADTs — Import.

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

- Ohne Bezeichner wird alles importiert
- `qualified`: qualifizierte Namen

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

- `hiding`: Liste der Bezeichner wird nicht importiert

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

- Selektiver Import: nur einiges Bezeichner importieren

```
import List (find)
```

- Miteinander kombinierbar.

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.
- 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.
- Gleiche Signatur, 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`
- Zeigen.

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 leer gdw. Queue leer

Repräsentation von `Queue`

Operation	Resultat	Queue	Repräsentation
`empty`			([], [])
`enq 9`		9	([9], [])
`enq 4`		4 $\rightarrow$9	([9], [4])
`enq 7`		7 $\rightarrow$4 $\rightarrow$9	([9], [7, 4])
`deq`	9	7 $\rightarrow$4	([4, 7], [])
`enq 5`		5 $\rightarrow$7 $\rightarrow$4	([4, 7], [5])
`deq`	4	5 $\rightarrow$7	([7], [5])
`deq`	7	5	([5], [])
`deq`	5		([], [])
`deq`	`error`		([], [])

Implementation

- Modulkopf, Exportliste:

```
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 mt Q"
first (Qu (x:xs) _) = x
```

- Bei `enq` und `deq` Invariante prüfen

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

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

- `check` prüft die Invariante

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

Endliche Mengen

- Eine endliche 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.

Endliche 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
```

Endliche Mengen — Implementation

- Implementation durch balancierte Bäume (AVL-Bäume)
  - Andere Möglichkeiten: 2-3 Bäume, Rot-Schwarz-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

```
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)
```

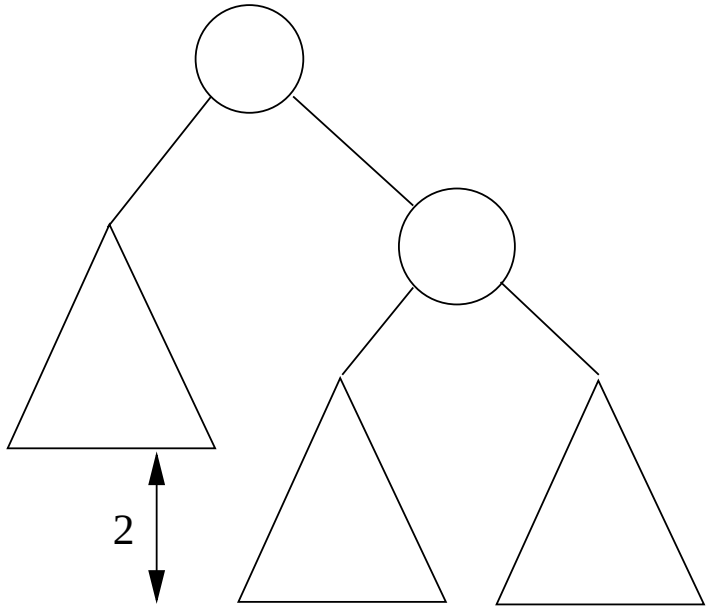
Ausgeglichenheit sicherstellen

- Problem:

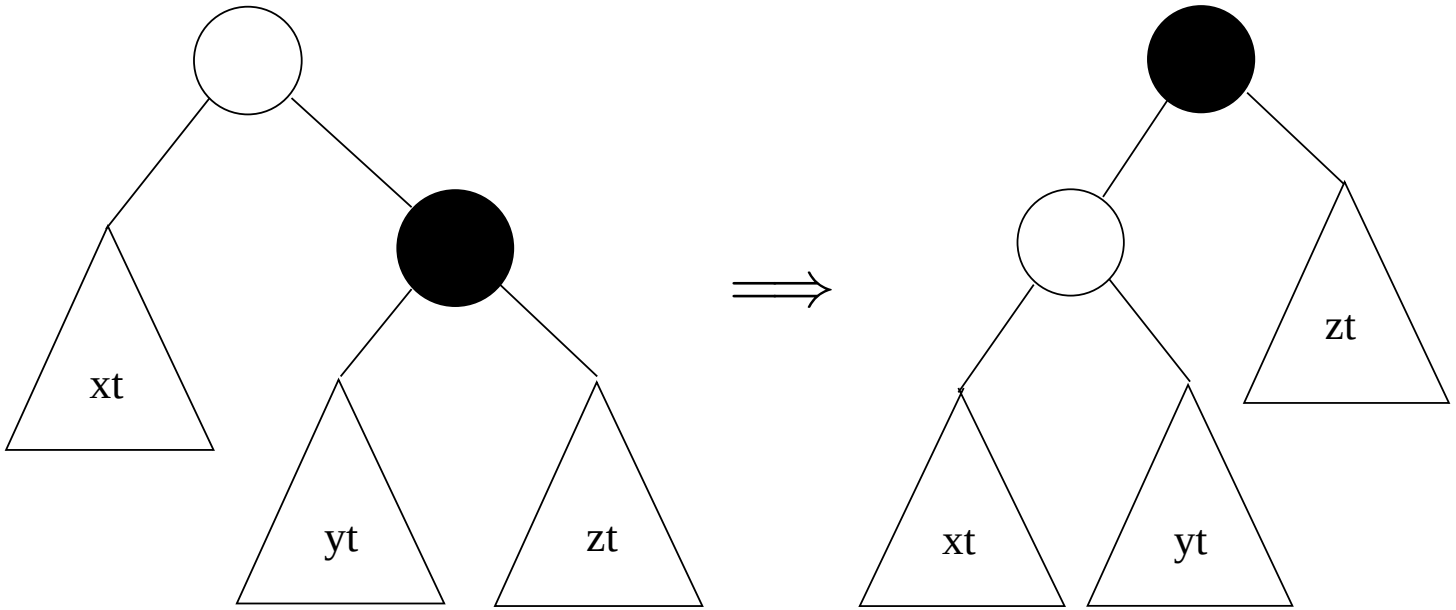
  Nach Löschen oder Einfügen zu großer Höhenunterschied

- Lösung:

  Rotieren der Unterbäume

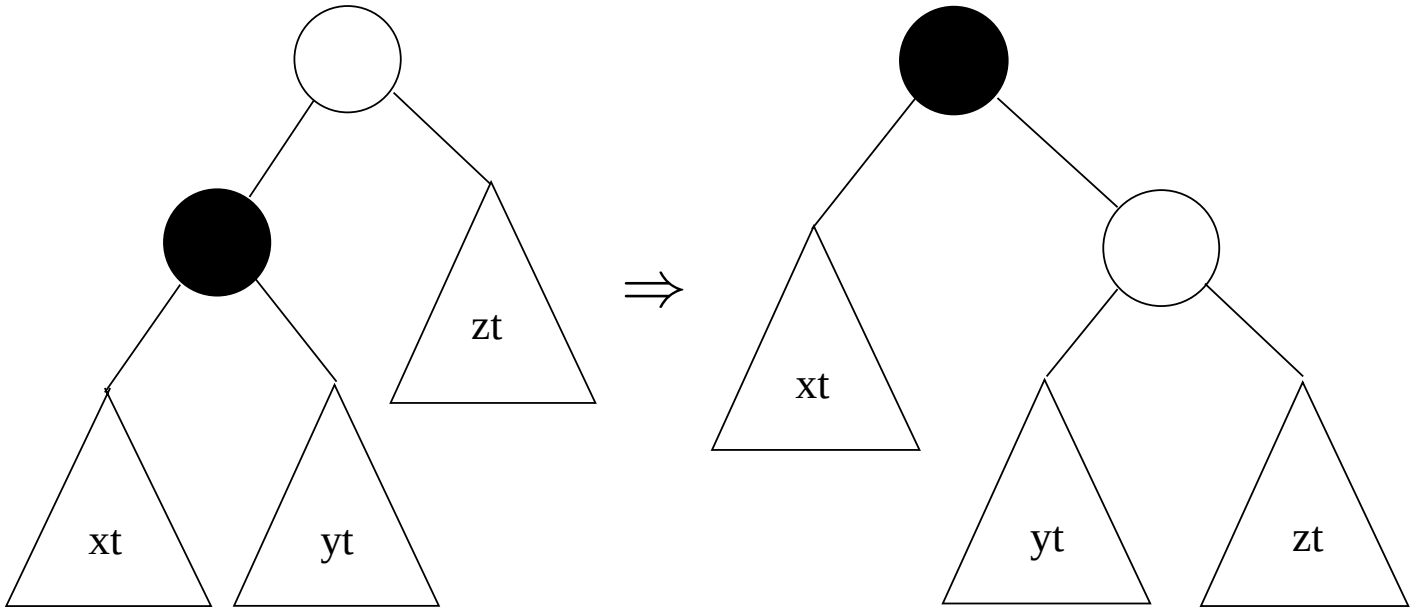


Linksrotation




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

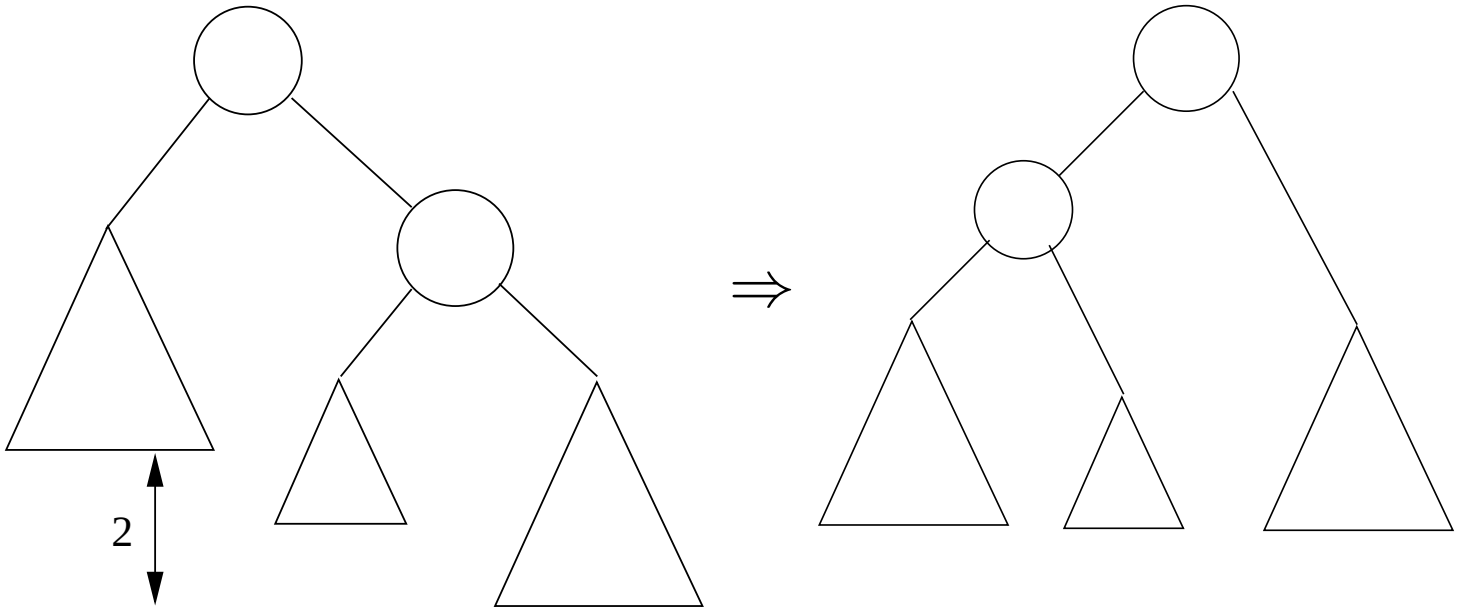
Rechtsrotation




```
rotr :: AVLTree a-> AVLTree a
rotr (Node _ (Node _ ut y vt) x rt) =
  mkNode ut y (mkNode vt x rt)
```

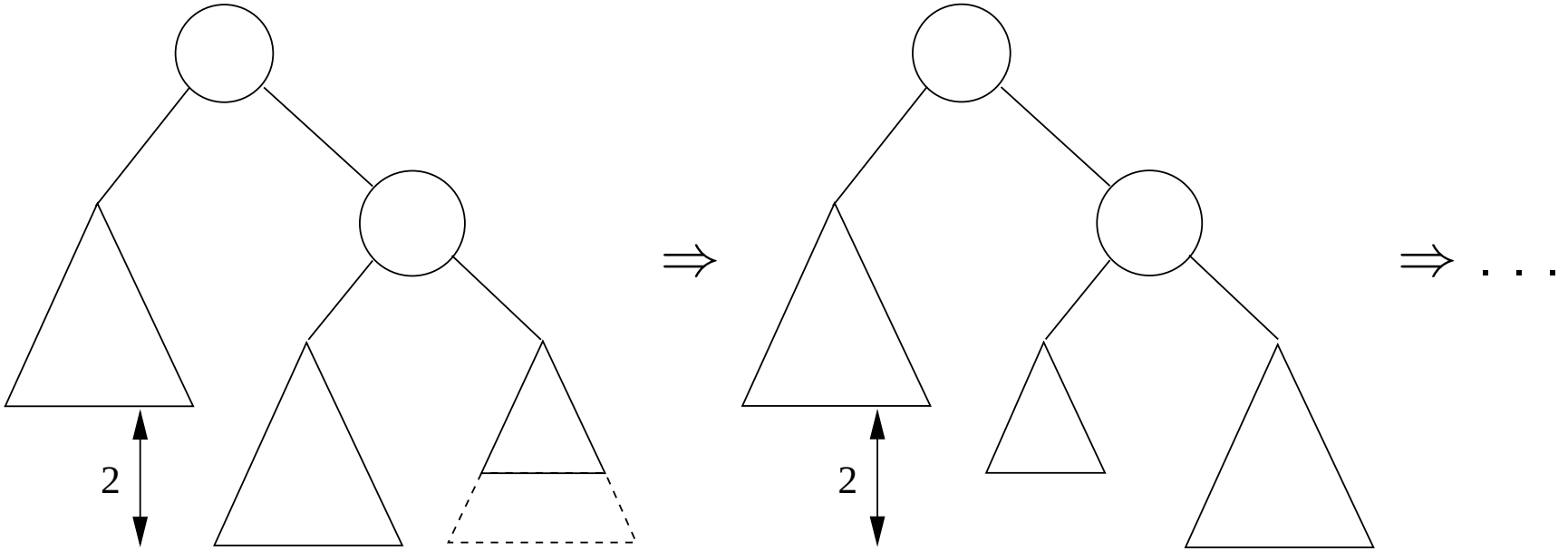
Ausgeglichenheit sicherstellen

- Fall 1: Äußerer Unterbaum zu hoch
- Lösung: Linksrotation



Ausgeglichenheit sicherstellen

- Fall 2: Innerer Unterbaum zu hoch oder gleich hoch
- Reduktion auf vorherigen Fall durch Rechtsrotation des Unterbaumes



Ausgeglichenheit sicherstellen

- Hilfsfunktion: Balance eines Baumes

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

- Zu implementieren: `mkAVL xt y zt`
  - Voraussetzung: Höhenunterschied `xt`, `zt` höchstens zwei;
  - Konstruiert neuen AVL-Baum mit Knoten `y`.
- Fallunterscheidung:
  - `zt` zu `hoch`, zwei Unterfälle:
    - ▷ Rechter Unterbaum von `zt` höher (Fall 1): `bias zt` < 0
    - ▷ Linker Unterbaum von `zt` höher/gleich hoch (Fall 2): `bias zt` ≥ 0
  - `xt` zu hoch, zwei Unterfälle (symmetrisch).

- Konstruktion eines neuen AVL-Bäumes mit Knoten `y`
  - Voraussetzung: Höhenunterschied `xt`, `zt` höchstens zwei;

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

- Enthaltensein:

```
elem :: Ord a=> a-> AVLTree a-> Bool
elem x Null = False
elem x (Node _ lt a rt) | x == a  = True
                        | x <  a  = x `elem` lt
                        | x >  a  = x `elem` rt
```

- Mengen aufzählen:

```
foldT :: (a-> b-> b-> b)-> b-> AVLTree a-> b
foldT f e Null = e
foldT f e (Node _ l a r) =
   f a (foldT f e l) (foldT f e r)
```

- Aufzählen der Menge: Inorder-Traversion (via `foldT`) Testen.

ADTs vs. Objekte

- ADTs (z.B. Haskell): Typ plus Operationen
- Objekte (z.B. Java): Interface, Methoden.
- Gemeinsamkeiten: Verkapselung (information hiding) der Implementation
- Unterschiede:
  - Objekte haben internen Zustand, ADTs sind referentiell transparent;
  - Objekte haben Konstruktoren, ADTs nicht (Konstruktoren nicht unterscheidbar)
  - Vererbungsstruktur auf Objekten (Verfeinerung für ADTs)
  - Java: `interface` eigenes Sprachkonstrukt, Haskell: Signatur eines Moduls nicht (aber z.B. SML).

Zusammenfassung

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