

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 8 vom 04.12.2012: Abstrakte Datentypen

Universität Bremen
Wintersemester 2012/13

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ Teil II: Funktionale Programmierung im Großen
 - ▶ Abstrakte Datentypen
 - ▶ Signaturen und Eigenschaften
 - ▶ Spezifikation und Beweis
 - ▶ Aktionen und Zustände
- ▶ Teil III: Funktionale Programmierung im richtigen Leben

- ▶ Abstrakte Datentypen
 - ▶ Allgemeine Einführung
 - ▶ Realisierung in Haskell
 - ▶ Beispiele

Printed by Christoph Loh		Page 23	
Page 13 to 14	Shogun's	Page 14	Shogun's
<pre> 1 #include <iostream> 2 using namespace std; 3 4 // ... 5 6 // ... 7 8 // ... 9 10 // ... 11 12 // ... 13 14 // ... 15 16 // ... 17 18 // ... 19 20 // ... 21 22 // ... 23 24 // ... 25 26 // ... 27 28 // ... 29 30 // ... 31 32 // ... 33 34 // ... 35 36 // ... 37 38 // ... 39 40 // ... 41 42 // ... 43 44 // ... 45 46 // ... 47 48 // ... 49 50 // ... 51 52 // ... 53 54 // ... 55 56 // ... 57 58 // ... 59 60 // ... 61 62 // ... 63 64 // ... 65 66 // ... 67 68 // ... 69 70 // ... 71 72 // ... 73 74 // ... 75 76 // ... 77 78 // ... 79 80 // ... 81 82 // ... 83 84 // ... 85 86 // ... 87 88 // ... 89 90 // ... 91 92 // ... 93 94 // ... 95 96 // ... 97 98 // ... 99 100 // ... 101 102 // ... 103 104 // ... 105 106 // ... 107 108 // ... 109 110 // ... 111 112 // ... 113 114 // ... 115 116 // ... 117 118 // ... 119 120 // ... 121 122 // ... 123 124 // ... 125 126 // ... 127 128 // ... 129 130 // ... 131 132 // ... 133 134 // ... 135 136 // ... 137 138 // ... 139 140 // ... 141 142 // ... 143 144 // ... 145 146 // ... 147 148 // ... 149 150 // ... 151 152 // ... 153 154 // ... 155 156 // ... 157 158 // ... 159 160 // ... 161 162 // ... 163 164 // ... 165 166 // ... 167 168 // ... 169 170 // ... 171 172 // ... 173 174 // ... 175 176 // ... 177 178 // ... 179 180 // ... 181 182 // ... 183 184 // ... 185 186 // ... 187 188 // ... 189 190 // ... 191 192 // ... 193 194 // ... 195 196 // ... 197 198 // ... 199 200 // ... 201 202 // ... 203 204 // ... 205 206 // ... 207 208 // ... 209 210 // ... 211 212 // ... 213 214 // ... 215 216 // ... 217 218 // ... 219 220 // ... 221 222 // ... 223 224 // ... 225 226 // ... 227 228 // ... 229 230 // ... 231 232 // ... 233 234 // ... 235 236 // ... 237 238 // ... 239 240 // ... 241 242 // ... 243 244 // ... 245 246 // ... 247 248 // ... 249 250 // ... 251 252 // ... 253 254 // ... 255 256 // ... 257 258 // ... 259 260 // ... 261 262 // ... 263 264 // ... 265 266 // ... 267 268 // ... 269 270 // ... 271 272 // ... 273 274 // ... 275 276 // ... 277 278 // ... 279 280 // ... 281 282 // ... 283 284 // ... 285 286 // ... 287 288 // ... 289 290 // ... 291 292 // ... 293 294 // ... 295 296 // ... 297 298 // ... 299 300 // ... 301 302 // ... 303 304 // ... 305 306 // ... 307 308 // ... 309 310 // ... 311 312 // ... 313 314 // ... 315 316 // ... 317 318 // ... 319 320 // ... 321 322 // ... 323 324 // ... 325 326 // ... 327 328 // ... 329 330 // ... 331 332 // ... 333 334 // ... 335 336 // ... 337 338 // ... 339 340 // ... 341 342 // ... 343 344 // ... 345 346 // ... 347 348 // ... 349 350 // ... 351 352 // ... 353 354 // ... 355 356 // ... 357 358 // ... 359 360 // ... 361 362 // ... 363 364 // ... 365 366 // ... 367 368 // ... 369 370 // ... 371 372 // ... 373 374 // ... 375 376 // ... 377 378 // ... 379 380 // ... 381 382 // ... 383 384 // ... 385 386 // ... 387 388 // ... 389 390 // ... 391 392 // ... 393 394 // ... 395 396 // ... 397 398 // ... 399 400 // ... 401 402 // ... 403 404 // ... 405 406 // ... 407 408 // ... 409 410 // ... 411 412 // ... 413 414 // ... 415 416 // ... 417 418 // ... 419 420 // ... 421 422 // ... 423 424 // ... 425 426 // ... 427 428 // ... 429 430 // ... 431 432 // ... 433 434 // ... 435 436 // ... 437 438 // ... 439 440 // ... 441 442 // ... 443 444 // ... 445 446 // ... 447 448 // ... 449 450 // ... 451 452 // ... 453 454 // ... 455 456 // ... 457 458 // ... 459 460 // ... 461 462 // ... 463 464 // ... 465 466 // ... 467 468 // ... 469 470 // ... 471 472 // ... 473 474 // ... 475 476 // ... 477 478 // ... 479 480 // ... 481 482 // ... 483 484 // ... 485 486 // ... 487 488 // ... 489 490 // ... 491 492 // ... 493 494 // ... 495 496 // ... 497 498 // ... 499 500 // ... 501 502 // ... 503 504 // ... 505 506 // ... 507 508 // ... 509 510 // ... 511 512 // ... 513 514 // ... 515 516 // ... 517 518 // ... 519 520 // ... 521 522 // ... 523 524 // ... 525 526 // ... 527 528 // ... 529 530 // ... 531 532 // ... 533 534 // ... 535 536 // ... 537 538 // ... 539 540 // ... 541 542 // ... 543 544 // ... 545 546 // ... 547 548 // ... 549 550 // ... 551 552 // ... 553 554 // ... 555 556 // ... 557 558 // ... 559 560 // ... 561 562 // ... 563 564 // ... 565 566 // ... 567 568 // ... 569 570 // ... 571 572 // ... 573 574 // ... 575 576 // ... 577 578 // ... 579 580 // ... 581 582 // ... 583 584 // ... 585 586 // ... 587 588 // ... 589 590 // ... 591 592 // ... 593 594 // ... 595 596 // ... 597 598 // ... 599 600 // ... 601 602 // ... 603 604 // ... 605 606 // ... 607 608 // ... 609 610 // ... 611 612 // ... 613 614 // ... 615 616 // ... 617 618 // ... 619 620 // ... 621 622 // ... 623 624 // ... 625 626 // ... 627 628 // ... 629 630 // ... 631 632 // ... 633 634 // ... 635 636 // ... 637 638 // ... 639 640 // ... 641 642 // ... 643 644 // ... 645 646 // ... 647 648 // ... 649 650 // ... 651 652 // ... 653 654 // ... 655 656 // ... 657 658 // ... 659 660 // ... 661 662 // ... 663 664 // ... 665 666 // ... 667 668 // ... 669 670 // ... 671 672 // ... 673 674 // ... 675 676 // ... 677 678 // ... 679 680 // ... 681 682 // ... 683 684 // ... 685 686 // ... 687 688 // ... 689 690 // ... 691 692 // ... 693 694 // ... 695 696 // ... 697 698 // ... 699 700 // ... 701 702 // ... 703 704 // ... 705 706 // ... 707 708 // ... 709 710 // ... 711 712 // ... 713 714 // ... 715 716 // ... 717 718 // ... 719 720 // ... 721 722 // ... 723 724 // ... 725 726 // ... 727 728 // ... 729 730 // ... 731 732 // ... 733 734 // ... 735 736 // ... 737 738 // ... 739 740 // ... 741 742 // ... 743 744 // ... 745 746 // ... 747 748 // ... 749 750 // ... 751 752 // ... 753 754 // ... 755 756 // ... 757 758 // ... 759 760 // ... 761 762 // ... 763 764 // ... 765 766 // ... 767 768 // ... 769 770 // ... 771 772 // ... 773 774 // ... 775 776 // ... 777 778 // ... 779 780 // ... 781 782 // ... 783 784 // ... 785 786 // ... 787 788 // ... 789 790 // ... 791 792 // ... 793 794 // ... 795 796 // ... 797 798 // ... 799 800 // ... 801 802 // ... 803 804 // ... 805 806 // ... 807 808 // ... 809 </pre>			

- ▶ Übersichtlichkeit der Module
 - Lesbarkeit (*human consumption*)
- ▶ Getrennte Übersetzung
 - technische Handhabbarkeit
- ▶ Verkapselung
 - konzeptionelle Handhabung, Invarianten

Definition (ADT)

Ein **abstrakter Datentyp** (ADT) besteht aus einem (oder mehreren) **Typen** und **Operationen** auf diesem.

- ▶ Werte des Typen können nur über die bereitgestellten Operationen erzeugt werden
- ▶ Eigenschaften von Werten des Typen werden nur über die bereitgestellten Operationen beobachtet
- ▶ Einhaltung von **Invarianten** über dem Typ kann garantiert werden

- ▶ Module
- ▶ Objekte

- ▶ Alg. Datentypen
 - ▶ Frei erzeugt
 - ▶ Vordefinierte Invarianten
 - ▶ Insbes. keine Gleichheiten
- ▶ ADTs:
 - ▶ Einschränkungen und Invarianten möglich
 - ▶ Gleichheiten möglich

- ▶ Einschränkung der Sichtbarkeit durch **Verkapselung**
- ▶ **Modul**: Kleinste verkapselbare **Einheit**
- ▶ Ein **Modul** umfaßt:
 - ▶ **Definitionen** von Typen, Funktionen, Klassen
 - ▶ **Deklaration** der nach außen **sichtbaren** Definitionen
- ▶ Gleichzeitig: $\text{Modul} \triangleq \text{Übersetzungseinheit (getrennte Übersetzung)}$

Module: Syntax

► Syntax:

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

- Bezeichner können leer sein (dann wird alles exportiert)
- Bezeichner sind:
 - Typen: $T, T(c_1, \dots, c_n), T(\dots)$
 - Klassen: $C, C(f_1, \dots, f_n), C(\dots)$
 - Andere Bezeichner: Werte, Felder, Klassenmethoden
 - Importierte Module: **module** M
- Typsynonyme und Klasseninstanzen bleiben sichtbar
- Module können **rekursiv** sein (*don't try at home*)

9 [31]

Beispiel: Das Lager

► Export als **abstrakter Datentyp**:

```
module Lager( Lager , leer , suche , einlagern ) where
```

- Typ Lager extern sichtbar
- Konstruktoren versteckt

► Export als **konkreter Datentyp**:

```
module Lager( Lager(..) , leer , suche , einlagern ) where
```

- Konstruktoren von Lager extern sichtbar
- Pattern Matching ist möglich
- Erzeugung von inkonsistentem Lager möglich

10 [31]

Benutzung von ADTs

- Operationen und Typen müssen **importiert** werden
- Möglichkeiten des Imports:
 - Alles importieren
 - Nur bestimmte Operationen und Typen importieren
 - Bestimmte Typen und Operationen **nicht** importieren

11 [31]

Importe in Haskell

► Syntax:

```
import [ qualified ] M [ as N ] [ hiding ] [ ( Bezeichner ) ]
```

- *Bezeichner* geben an, **was** importiert werden soll:
 - Ohne Bezeichner wird **alles** importiert
 - Mit **hiding** werden Bezeichner **nicht** importiert
- Für jeden exportierten Bezeichner *f* aus *M* wird importiert
 - *f* und qualifizierter Bezeichner *M.f*
 - **qualified**: nur qualifizierter Bezeichner *M.f*
 - Umbenennung bei Import mit **as** (dann *N.f*)
 - Klasseninstanzen und Typsynonyme werden immer importiert
- Alle Importe stehen immer am **Anfang** des Moduls

12 [31]

Beispiel

```
module A(x, y) where ...
```

Import(e)	Bekannte Bezeichner
import A	x, y, A.x, A.y
import A()	(nothing)
import A(x)	x, A.x
import qualified A	A.x, A.y
import qualified A()	(nothing)
import qualified A(x)	A.x
import A hiding ()	x, y, A.x, A.y
import A hiding (x)	y, A.y
import qualified A hiding ()	A.x, A.y
import qualified A hiding (x)	A.y
import A as B	x, y, B.x, B.y
import A as B(x)	x, B.x
import qualified A as B	B.x, B.y

Quelle: Haskell98-Report, Sect. 5.3.4

13 [31]

Schnittstelle vs. Implementation

- Gleiche **Schnittstelle** kann unterschiedliche **Implementationen** haben
- Beispiel: (endliche) Abbildungen

14 [31]

Endliche Abbildungen

- Eine Sichtweise: Ersatz für Hashtables in imperativen Sprachen. **Sehr nützlich!**
- Abstrakter Datentyp für **endliche Abbildungen**:
 - Datentyp

```
data Map  $\alpha$   $\beta$ 
```
 - Leere Abbildung:

```
empty :: Map  $\alpha$   $\beta$ 
```
 - Abbildung auslesen:

```
lookup :: Eq  $\alpha$   $\Rightarrow$   $\alpha \rightarrow$  Map  $\alpha$   $\beta \rightarrow$  Maybe  $\beta$ 
```
 - Abbildung ändern:

```
insert :: Eq  $\alpha$   $\Rightarrow$   $\alpha \rightarrow$   $\beta \rightarrow$  Map  $\alpha$   $\beta \rightarrow$  Map  $\alpha$   $\beta$ 
```
 - Abbildung löschen:

```
delete :: Eq  $\alpha$   $\Rightarrow$   $\alpha \rightarrow$  Map  $\alpha$   $\beta \rightarrow$  Map  $\alpha$   $\beta$ 
```

15 [31]

Eine naheliegende Implementation

- Modellierung als Haskell-Funktion:

```
data Map  $\alpha$   $\beta$  = Map ( $\alpha \rightarrow$  Maybe  $\beta$ )
```

- Damit einfaches lookup, insert, delete:

```
empty = Map ( $\lambda x \rightarrow$  Nothing)
```

```
lookup a (Map s) = s a
```

```
insert a b (Map s) =  
  Map ( $\lambda x \rightarrow$  if  $x == a$  then Just b else s x)
```

```
delete a (Map s) =  
  Map ( $\lambda x \rightarrow$  if  $x == a$  then Nothing else s x)
```

- Instanzen von Eq, Show **nicht möglich**
- **Speicherleck**

16 [31]

Endliche Abbildungen: Anwendungsbeispiel

- ▶ Artikel im Lager:

```
newtype Lager = Lager (M.Map Artikel Menge)
```

- ▶ Artikel

```
suche art (Lager m) = M.lookup art m
```

- ▶ Ins Lager hinzufügen:

```
einlagern a m (Lager l) =  
  case preis a m of  
    Nothing → Lager l  
    _ → let m' = maybe m (addiere m) (M.lookup a l)  
        in Lager (M.insert a m' l)
```

- ▶ Für Inventur fehlt Möglichkeit zur **Iteration**
- ▶ Daher: Map als **Assoziativliste**

17 [31]

Bewertung

- ▶ Map als **Assoziativliste** bietet
 - ▶ Instanzen von Eq und Show
 - ▶ Iteration (fold)
 - ▶ ... ist aber **ineffizient** (Zugriff/Löschen in $\mathcal{O}(n)$)
- ▶ Deshalb: **balancierte Bäume**

18 [31]

AVL-Bäume und Balancierte Bäume

AVL-Bäume

Ein Baum ist **ausgeglichen**, wenn

- ▶ alle Unterbäume ausgeglichen sind, und
- ▶ der Höhenunterschied zwischen zwei Unterbäumen höchstens eins beträgt.

Balancierte Bäume

Ein Baum ist **balanciert**, wenn

- ▶ alle Unterbäume balanciert sind, und
- ▶ für den linken und rechten Unterbaum l, r gilt:

$$\text{size}(l) \leq w \cdot \text{size}(r) \quad (1)$$

$$\text{size}(r) \leq w \cdot \text{size}(l) \quad (2)$$

w — **Gewichtung** (Parameter des Algorithmus)

19 [31]

Implementation von Balancierten Bäumen

- ▶ Der Datentyp

```
data Tree  $\alpha$  = Null  
          | Node Weight (Tree  $\alpha$ )  $\alpha$  (Tree  $\alpha$ )
```

- ▶ Gewichtung:

```
type Weight = Int
```

```
weight :: Weight
```

- ▶ Hilfskonstruktor, setzt Größe (l, r balanciert)

```
node :: Tree  $\alpha$  →  $\alpha$  → Tree  $\alpha$  → Tree  $\alpha$   
node l n r = Node h l n r where  
    h = 1 + size l + size r
```

- ▶ Hilfskonstruktor, balanciert ggf. neu aus:

```
mkNode :: Tree  $\alpha$  →  $\alpha$  → Tree  $\alpha$  → Tree  $\alpha$ 
```

20 [31]

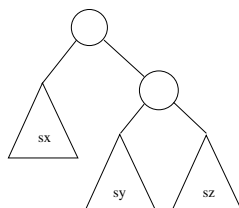
Balance sicherstellen

- ▶ Problem:

Nach Löschen oder Einfügen zu großes Ungewicht

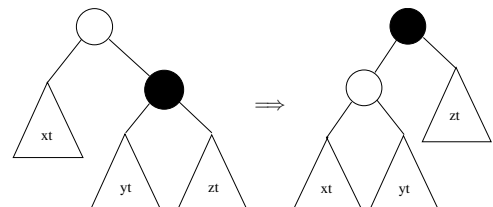
- ▶ Lösung:

Rotieren der Unterbäume



21 [31]

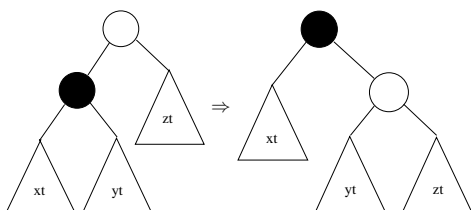
Linksrotation



```
rotr :: Tree  $\alpha$  → Tree  $\alpha$   
rotr (Node _ xt y (Node _ yt x zt)) =  
  node (node xt y yt) x zt
```

22 [31]

Rechtsrotation

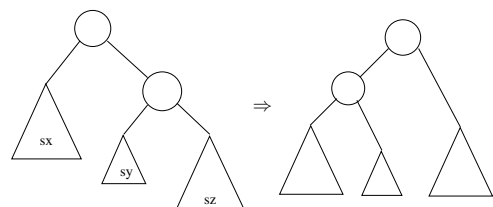


```
rotr :: Tree  $\alpha$  → Tree  $\alpha$   
rotr (Node _ (Node _ ut y vt) x rt) =  
  node ut y (node vt x rt)
```

23 [31]

Balanciertheit sicherstellen

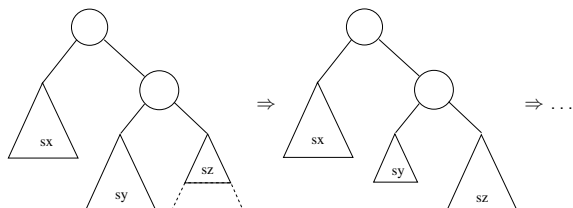
- ▶ Fall 1: Äußerer Unterbaum zu groß
- ▶ Lösung: Linksrotation



24 [31]

Balanciertheit sicherstellen

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



25 [31]

Balance sicherstellen

- ▶ Hilfsfunktion: **Balance** eines Baumes

```
bias :: Tree α → Ordering
bias Null = EQ
bias (Node _ lt _ rt) = compare (size lt) (size rt)
```

- ▶ Zu implementieren: mkNode lt y rt
 - ▶ Voraussetzung: lt, rt balanciert
 - ▶ Konstruiert neuen balancierten Baum mit Knoten y
- ▶ Fallunterscheidung:
 - ▶ rt zu groß, zwei Unterfälle:
 - ▶ Linker Unterbaum von rt kleiner (Fall 1): bias rt == LT
 - ▶ Linker Unterbaum von rt größer/gleich groß (Fall 2): bias rt == EQ, bias rt == GT
 - ▶ lt zu groß, zwei Unterfälle (symmetrisch).

26 [31]

Konstruktion eines ausgeglichenen Baumes

- ▶ Voraussetzung: lt, rt balanciert

```
mkNode lt x rt
| ls + rs < 2 = node lt x rt
| weight* ls < rs =
    if bias rt == LT then rotl (node lt x rt)
    else rotl (node lt x (rotr rt))
| ls > weight* rs =
    if bias lt == GT then rotr (node lt x rt)
    else rotr (node (rotl lt) x rt)
| otherwise = node lt x rt where
    ls = size lt; rs = size rt
```

27 [31]

Balancierte Bäume als Maps

- ▶ Endliche Abbildung: Bäume mit (key, value) Paaren

```
type Map α β = Tree (α, β)
```

- ▶ insert fügt neues Element ein:

```
insert :: Ord α ⇒ α → β → Map α β → Map α β
insert k v Null = node Null (k, v) Null
insert k v (Node n l a@(kn, _) r)
| k < kn = mkNode (insert k v l) a r
| k == kn = Node n l (k, v) r
| k > kn = mkNode l a (insert k v r)
```

- ▶ lookup liest Element aus
- ▶ remove löscht ein Element
 - ▶ Benötigt Hilfsfunktion join :: Tree α → Tree α → Tree α

28 [31]

Zusammenfassung Balancierte Bäume

- ▶ Einfügen und löschen logarithmischer Aufwand ($\mathcal{O}(\log n)$)
- ▶ Fold hat linearen Aufwand ($\mathcal{O}(n)$)
- ▶ Guten durchschnittlichen Aufwand
- ▶ Auch in der Haskell-Bücherei: Data.Map (mit vielen weiteren Funktionen)

29 [31]

ADTs vs. Objekte

- ▶ ADTs (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
 - ▶ Java: packages für Sichtbarkeit

30 [31]

Zusammenfassung

- ▶ **Abstrakte Datentypen** (ADTs):
 - ▶ Besteht aus **Typen** und **Operationen** darauf
- ▶ Realisierung in Haskell durch **Module**
- ▶ Beispieldatentypen: endliche Abbildungen
- ▶ Nächste Vorlesung: ADTs durch **Eigenschaften** spezifizieren

31 [31]