Jan Martin (8899)
Wolfgang Schmoller (8780)

Abstrakte Datentypen in Haskell

Gliederung

- **1. Einführung**
  - 1.1 Grundidee
  - 1.2 Vorteile
  - 1.3 Vorgehensweise
- **2. Warteschlangen**
  - 2.1 Merkmale und Operationen
  - 2.2 Algebraische Spezifikation
  - 2.3 Implementierungen

Gliederung

- **3. Anwendung**
  - 3.1 Module
  - 3.2 Praxisbeispiel
- **4. Mengen**
  - 4.1 Merkmale und Operationen
  - 4.2 Algebraische Spezifikation
  - 4.3 Implementierung
  - 4.4 Komplexitäten

Gliederung

- **5. Bags**
  - 5.1 Merkmale und Operationen
  - 5.2 Algebraische Spezifikation
  - 5.3 Implementierung

Gliederung

- **6. Flexible Arrays**
  - 6.1 Merkmale und Operationen
  - 6.2 Algebraische Spezifikation
  - 6.3 Implementierung
- **7. Quellen**

1. Einführung

1.1 Grundidee

- Bisher: Konkrete Datentypen
  - Definition von Typen über explizite Werte
  - Beispiele:
    - Integer, Bool
    - data Foobar = Foo | Bar
  - Probleme:
    - Datenstruktur muss bekannt sein
    - Implementierungsänderungen u. U. mühselig

1. Einführung

1.1 Grundidee

- Idee der abstrakten Datentypen
  - Definition von Typen über Operationen
  - Operationen definieren eine Schnittstelle
  - Lediglich die Operationen sind bekannt
  - Vergleiche:
    - Interfaces in Java
    - Module
  - Werte sind Folgen von Operationen

1. Einführung

1.2 Vorteile

- **Universalität:** In beliebigen Programmen importierbar
- **Präzise Beschreibung:** Durch Operationenmenge exakt und eindeutig
- **Einfachheit:** Kein Wissen über interne Implementierung nötig
- **Kapselung:** Interne Implementierung bleibt verborgen
- **Geschütztheit:** Kein Zugriff auf interne Implementierung
- **Modularität:** Förderung guter Programmstrukturierung
- **Implementierungsunabhängigkeit:** Durch Schnittstellen-beschreibung gewährleistet

1. Einführung

1.3 Vorgehensweise

- Definition der Schnittstelle
  - Bezeichnung von Operationen
  - Definition des Typs dieser Operationen
- Festlegung der algebraischen Spezifikation (Axiome)
  - Regeln, welche die Implementierung einhalten soll
  - Hinweise auf Bedeutung der Operationen (Semantik)
- Wahl einer geeigneten Implementierung
  - Wichtigstes Kriterium: Effizienz/Laufzeit

2. Warteschlangen

2.1 Merkmale und Operationen

- Merkmale
  - Liste von Elementen willkürlichen Typs
  - Neue Elemente werden an das Ende angehängt
  - Abgreifen von Elementen am Anfang der Liste

2. Warteschlangen

2.1 Merkmale und Operationen

- Operationen
  - empty :: Queue a
  - join :: a → Queue a → Queue a
  - front :: Queue a → a
  - back :: Queue a → Queue a
  - isEmpty :: Queue a → Bool
- Menge von Operationen zusammen mit deren Typen ergibt die sogenannte Signatur des abstrakten Datentyps

2. Warteschlangen

2.2 Algebraische Spezifikation

- Axiomatische Grundlage für die Implementierung
  - isEmpty empty = True
  - isEmpty (join x xq) = False
  - front (join x empty) = x
  - front (join x (join y xq)) = front (join y xq)
  - back (join x empty) = empty
  - back (join x (join y xq)) = join x (back (join y xq))

2. Warteschlangen

2.3 Implementierungen

- Einfache Implementierung für...?
  - empty :: Queue a
  - join :: a → Queue a → Queue a
  - front :: Queue a → a
  - back :: Queue a → Queue a
  - isEmpty :: Queue a → Bool

2. Warteschlangen

2.3 Implementierungen

- Erste Implementierung: endliche Listen
  - empty :: [a]
  - empty = []
  - join :: a → [a] → [a]
  - join x xs = xs ++ [x]
  - front :: [a] → a
  - front (x:xs) = x

2. Warteschlangen

2.3 Implementierungen

- Erste Implementierung: endliche Listen
  - back :: [a] → [a]
  - back (x:xs) = xs
  - isEmpty :: [a] → Bool
  - isEmpty xs = null xs

2. Warteschlangen

2.3 Implementierungen

- Umwandlung zwischen Abstraktion und Repräsentation
  - abstr :: [a] → Queue a
  - abstr = foldr join empty . reverse

  - reprn :: Queue a → [a]
  - reprn empty = []
  - reprn (join x xq) = reprn xq ++ [x]

2. Warteschlangen

2.3 Implementierungen

- Laufzeiten...?
  - empty = [] O(?)
  - join x xs = xs ++ [x] O(?)
  - front (x:xs) = x O(?)
  - back (x:xs) = xs O(?)
  - isEmpty xs = null xs O(?)

2. Warteschlangen

2.3 Implementierungen

- Laufzeiten
  - empty = [] — O(1)
  - join x xs = xs ++ [x] — O(n)
  - front (x:xs) = x — O(1)
  - back (x:xs) = xs — O(1)
  - isEmpty xs = null xs — O(1)
- Problem: Häufige Auswertung der wenig effizienten Operation join

2. Warteschlangen

2.3 Implementierungen

- Bessere Implementierung: Paare von Teillisten
  - Queue a repräsentiert durch ([a], [a])
  - Warteschlange (xs, ys) repräsentiert in korrekter Reihenfolge die Elemente von (xs ++ reverse ys)
  - Constraint für alle (xs, ys): Wenn null xs, null ys
  - Beispiel für ungültige Warteschlange: ([], [x])
  - Beispiele für identische Warteschlangen:
    - ([1, 2, 3], []), ([1, 2], [3]), ([1], [3, 2])

2. Warteschlangen

2.3 Implementierungen

- Bessere Implementierung: Paare von Teillisten
  - Prüfung der Gültigkeit
    - valid :: ([a], [a]) → Bool
    - valid (xs, ys) = not (null xs) ∨ null ys
  - Abstraktion
    - abstr :: ([a], [a]) → Queue a
    - abstr (xs, ys) = (foldr join empty . reverse)
      (xs ++ reverse ys)

2. Warteschlangen

2.3 Implementierungen

- Bessere Implementierung: Paare von Teillisten
  - empty :: ([a], [a])
  - empty = ([], [])
  - join :: a → ([a], [a]) → ([a], [a])
  - join x (ys, zs) = mkValid (ys, x:zs)
  - front :: ([a], [a]) → a
  - front (x:xs, ys) = x

2. Warteschlangen

2.3 Implementierungen

- Bessere Implementierung: Paare von Teillisten
  - back :: ([a], [a]) → ([a], [a])
  - back (x:xs, ys) = mkValid (xs, ys)
  - isEmpty :: ([a], [a]) → Bool
  - isEmpty (xs, ys) = null xs
  - mkValid :: ([a], [a]) → ([a], [a])
  - mkValid (xs, ys) = if null xs then (reverse ys, [])
    else (xs, ys)

2. Warteschlangen

2.3 Implementierungen

- Laufzeiten...?
  - empty = ([], []) O(?)
  - join x (ys, zs) = mkValid (ys, x:zs) O(?)
  - front (x:xs, ys) = x O(?)
  - back (x:xs, ys) = mkValid (xs, ys) O(?)
  - isEmpty (xs, ys) = null xs O(?)
  - mkValid (xs, ys) = if null xs then (reverse ys, [])
    else (xs, ys)

2. Warteschlangen

2.3 Implementierungen

- Laufzeiten
  - empty = ([], []) O(1)
  - join x (ys, zs) = mkValid (ys, x:zs) O(1)
  - front (x:xs, ys) = x O(1)
  - back (x:xs, ys) = mkValid (xs, ys) *
  - isEmpty (xs, ys) = null xs O(1)

  * O(n) bei einelementiger Liste xs für das Umkehren von ys zur Validierung des Resultats, sonst O(1)

3. Anwendung

3.1 Module

- unterstützt Realisierung des Konzepts der abstrakten Datentypen in Haskell
- Schlüsselwort „module“, gefolgt von einer Liste von öffentlich Funktionen und Datentypen und „where“
- Definition eines Datentypen (Konstruktor nur im Modul verwendbar!)
- Modulnamen beginnen mit Großbuchstaben
- Modulname == Dateiname
- Einbinden über Schlüsselwort „import“

3. Anwendung

3.2 Praxisbeispiel

- Beispiel Modul

4. Sets

4.1 Merkmale und Operationen

- Sets (Mengen) bestehen aus Elementen aus einem bestimmten Wertebereich, deren Reihenfolge unerheblich ist, z.B. {1, 3, 4} == {4,1,3}
- verschiedene Arten der Implementierung: Liste, Liste ohne Duplikate, geordnete Liste, Bäume, Boolesche Funktionen, etc.
- Keine „beste“ Implementierung

4. Sets

4.1 Merkmale und Operationen

- dictionary-Operationen
  - empty :: Set a
  - isEmpty :: Set a → Bool
  - member :: Set a → a → Bool
  - insert :: a → Set a → Set a
  - delete :: a → Set a → Set a

4. Sets

4.1 Merkmale und Operationen

- non-dictionary-Operationen
  - union :: Set a → Set a → Set a
  - meet :: Set a → Set a → Set a
  - minus :: Set a → Set a → Set a
- hier: strikte Verarbeitung, also keine unendlichen Listen

4. Sets

4.2 Algebraische Spezifikation

- Axiomatische Grundlage für die Implementierung
  - isEmpty empty = True
  - isEmpty (insert x xs) = False
  - insert x (insert x xs) = insert x xs
  - insert x (insert y xs) = insert y (insert x xs)
  - member empty y = False
  - member (insert x xs) y = (x == y) || member xs x

4. Sets

4.2 Algebraische Spezifikation

- Axiomatische Grundlage für die Implementierung
  - delete x empty = empty
  - delete x (insert y xs) = if (x == y) then delete x xs
    else insert y (delete x xs)
  - union xs empty = xs
  - union xs (insert y ys) = insert y (union xs ys)

4. Sets

4.3 Implementierungen

- Umwandlung von Listen in Sets
  - abstr :: [a] → Set a
  - abstr = foldr insert empty
- Umwandlung von Sets in Listen
  - reprn [] = []
  - reprn (x:xs)

```
    | null xs   = [x]
    | otherwise = [x] ++ reprn xs
```

4. Sets

4.3 Implementierungen

- zwei gültige Repräsentationen
  - valid xs = True (Listen mit Dulplikaten)
  - valid xs = nonduplicated xs (Listen ohne Duplikate)

4. Sets

4.3 Implementierungen

- Listen mit Duplikaten
  - member xs x = some (==x) xs
  - insert x xs = x:xs
  - delete x xs = filter (/=x) xs
  - union xs ys = xs ++ ys
  - minus xs ys = filter (not . member ys) xs
  - some :: (a → Bool) → [a] → Bool
  - some p = or . map p

4. Sets

4.3 Implementierungen

- sortierte Listen ohne Duplikate
  - insert x xs = ys ++ [x] ++ (filter (/=x) zs)

    where (ys, zs) = span (<x) xs
  - member xs x = if null ys then False

    else (x == head ys)

    where ys = dropWhile (<x) xs

4. Sets

4.3 Implementierungen

- sortierte Listen ohne Duplikate
  - union [] ys = ys
  - union xs [] = xs
  - union (x:xs) (y:ys)

```
| (x < y)  = x:union xs (y:ys)
| (x == y) = x:union xs ys
| (x > y)  = y:union (x:xs) ys
```

4. Sets

4.3 Implementierungen

- sortierte Listen ohne Duplikate
  - Beispiel „minus“ für geordnete Listen ohne Duplikate

4. Sets

4.3 Implementierungen

- Bäume
  - data TSet a = Null

    | Fork (TSet a) a (TSet a)
- Ein Knoten besteht aus zwei weiteren Knoten, die Null sein können, sowie einem Wert aus dem Wertebereich a

4. Sets

4.3 Implementierungen

- Bäume
  - empty :: TSet a
  - empty = Null
  - isEmpty :: TSet a → Bool
  - isEmpty Null = True
  - isEmpty (Fork xt y zt) = False

4. Sets

4.3 Implementierungen

- Bäume
  - member :: (Ord a) => TSet a → a → Bool
  - member Null x = False
  - member (Fork xt y zt) x
  - | (x < y) = member xt x
  - | (x == y) = True
  - | (x > y) = member zt x

4. Sets

4.3 Implementierungen

- Bäume
  - insert :: (Ord a) => a → TSet a → TSet a
  - insert x Null = (Fork Null x Null)
  - insert x (Fork xt y zt)

    | (x < y) = Fork (insert x xt) y zt

    | (x == y) = Fork xt y zt

    | (x > y) = Fork xt y (insert x zt)

4. Sets

4.3 Implementierungen

- Bäume
  - delete :: (Ord a) => a → TSet a → TSet a
  - delete x Null = Null
  - delete x (Fork xt y zt)

```
    | (x < y)  = Fork (delete x xt) y zt
    | (x == y) = join xt zt
    | (x > y)  = Fork xt y (delete x zt)
```

4. Sets

4.3 Implementierungen

- Bäume
  - join :: TSet a → TSet a → TSet a
  - join xt yt = if isEmpty yt then xt else Fork xt y zt

    where (y,zt) = splitTree yt
  - splitTree :: TSet a → (a, TSet a)
  - splitTree (Fork xt y zt) = if isEmpty xt then (y, zt)

    else (u, Fork vt y zt)

    where (u, vt) = splitTree xt

4. Sets

4.3 Implementierungen

- Problem mit bisherigen Bäumen: im schlimmsten Fall nicht schneller als Listen
- Lösung: balancierte Bäume
  - beide Teilbäume eines Knotens unterscheiden sich in der Höhe um maximal 1

4. Sets

4.3 Implementierungen

- balancierte Bäume
  - data BTSet a = Null | Fork Int (BTSet a) a (BTSet a)
- Ein Knoten enthält wie vorher zwei Knoten, sowie den eigentlichen Wert, sowie als Integer-Label die maximale Höhe der beiden Teilbäume

4. Sets

4.3 Implementierungen

- balancierte Bäume
  - fork :: BTSet a → a → BTSet a → BTSet a
  - fork xt y zt = Fork h xt y zt

    where h = 1 + (max (ht xt) (ht zt))
  - ht :: BTSet a → Int
  - ht Null = 0
  - ht (Fork h xt y zt) = h

4. Sets

4.3 Implementierungen

- balancierte Bäume
- neue Funktion spoon (statt Aufruf von Fork in den baumverändernden Funktionen insert, delete, join und splitTree), um balancierte Bäume zu erzeugen

4. Sets

4.3 Implementierungen

- balancierte Bäume
  - spoon :: BTSet a → a → BTSet a → BTSet a
  - spoon xt y zt

```
|(hz + 1 < hx) && (bias xt < 0) = rotr(fork (rotl xt) y zt)
|(hz + 1 < hx) = rotr(fork xt y zt)
|(hx + 1 < hz) && (0 < bias zt) = rotl(fork xt y (rotr zt))
|(hx + 1 < hz) = rotl(fork xt y zt)
|otherwise = fork xt y zt
```

4. Sets

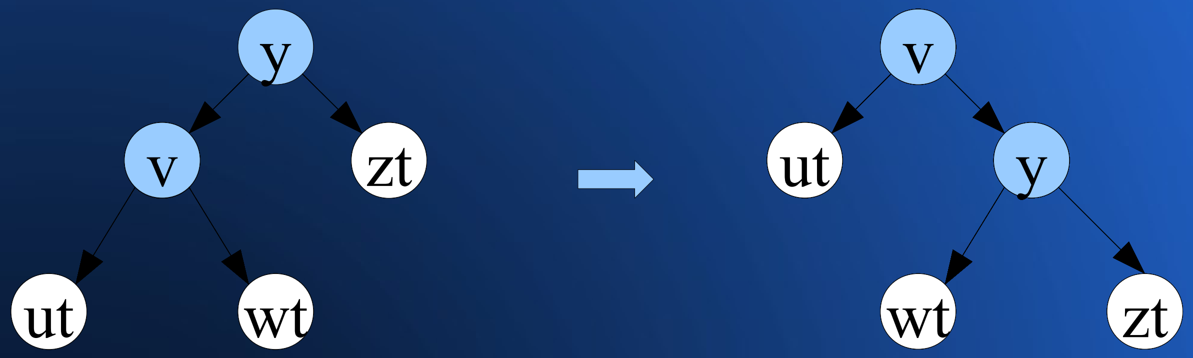
4.3 Implementierungen

- balancierte Bäume
  - hx = ht xt
  - hz = ht zt
  - bias :: BTSet a → Int
  - bias (Fork h xt y zt) = ht xt - ht zt
  - rotr :: BTSet a → BTSet a
  - rotr (Fork m (Fork n ut v wt) y zt) = fork ut v (fork wt y zt)
  - rotl :: BTSet a → BTSet a
  - rotl (Fork m ut v (Fork n rt s tt)) = fork (fork ut v rt) s tt

4. Sets

4.3 Implementierungen

rotr (Fork m (Fork n ut v wt) y zt) = fork ut v (fork wt y zt)

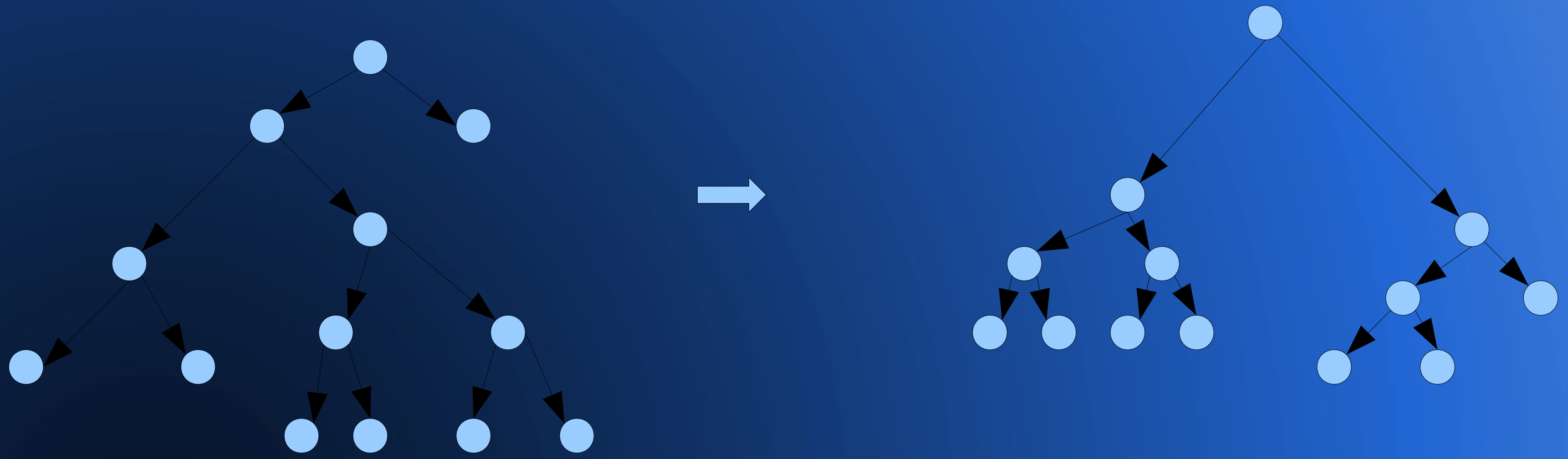


4. Sets

4.3 Implementierungen

spoon xt y zt

|(hz + 1 < hx) && (bias xt < 0) = rotr(fork (rotl xt) y zt)



4. Sets

4.4 Komplexitäten

	Liste	ohne Dupl.	ohne Dupl., sortiert	Baum	Bal. Baum
member	$O(n^2)$	$O(n^2)$	$O(N)$	$O(\log n)$	$O(\log n)$
insert	$O(1)$	$O(N)$	$O(N)$	$O(\log n)$	$O(\log n)$
delete	$O(n)$	$O(n)$	$O(N)$	$O(\log n)$	$O(\log n)$
union	$O(n)$	$O(N^2)$	$O(N)$	-	-
meet	$O(n)$	$O(n)$	$O(N)$	-	-
minus	$O(n^2)$	$O(n^2)$	$O(N)$	-	-

5. Bags

5.1 Merkmale und Operationen

- Merkmale
  - Reihenfolge wie bei Mengen irrelevant
  - Gleiche Elemente können wie bei Listen mehrfach enthalten sein
  - Beispiele:
    - $\{[1, 2, 2, 3]\} = \{[3, 2, 1, 2]\}$
    - $\{[1, 2, 2, 3]\} \neq \{[1, 2, 3]\}$

5. Bags

5.1 Merkmale und Operationen

- Operationen
  - mkBag :: [a] → Bag a
  - isEmpty :: Bag a → Bool
  - union :: Bag a → Bag a → Bag a
  - minBag :: Bag a → a
  - delMin :: Bag a → Bag a
- minBag und delMin erfordern, dass der Typ a eine Instanz von Ord ist

5. Bags

5.2 Algebraische Spezifikation

- Beispielsweise unter Nutzung zweier Konstruktoren empty und insert möglich
  - insert x (insert y xb) = insert y (insert x xb)
  - mkBag = foldr insert empty
  - union xb empty = xb
  - union xb (insert y yb) = insert y (union xb yb)
  - minBag (insert x empty) = x
  - minBag (insert x (insert y xb)) = x \`min\` minBag (insert y xb)

5. Bags

5.2 Algebraische Spezifikation

- Axiome für...?
  - delMin :: Bag a → Bag a

5. Bags

5.2 Algebraische Spezifikation

- Lösung
  - delMin (insert x empty) = empty
  - insert (minBag xb) (delMin xb) = xb

5. Bags

5.3 Implementierung

- Laufzeiten bei Umsetzung mit sortierten Listen
  - mkBag :: [a] → Bag a $O(n \log n)$
  - isEmpty :: Bag a → Bool $O(1)$
  - union :: Bag a → Bag a → Bag a $O(m + n)$
  - minBag :: Bag a → a $O(1)$
  - delMin :: Bag a → Bag a $O(1)$
- Das Erzeugen und Vereinen von Listen ist ineffizient!

5. Bags

5.3 Implementierung

- Leftist size-augmented binary heap trees
  - data Htree a = Null | Fork Int a (Htree a) (Htree a)
  - fork :: a → Htree a → Htree a → Htree a
  - fork x yt zt = if m < n then Fork p x zt yt else Fork p x yt zt

```
fork x yt zt = if m < n then Fork p x zt yt
                        else Fork p x yt zt
    where m = size yt
          n = size zt
          p = m + n + 1
```

5. Bags

5.3 Implementierung

- size :: Htree a → Int
- size Null = 0
- size (Fork n x yt zt) = n

- isEmpty :: Htree a → Bool
- isEmpty Null = True
- isEmpty (Fork n x yt zt) = False

5. Bags

5.3 Implementierung

- minBag :: Htree a → a
- minBag (Fork n x yt zt) = x

- delMin :: Htree a → Htree a
- delMin (Fork n x yt zt) = union yt zt

5. Bags

5.3 Implementierung

- union :: Htree a → Htree a → Htree a
- union Null yt = yt
- union (Fork m u vt wt) Null = Fork m u vt wt

- union (Fork m u vt wt) (Fork n x yt zt)

  | (u ≤ x) = fork u vt (union wt (Fork n x yt zt))

  | (x < u) = fork x yt (union (Fork m u vt wt) zt)

5. Bags

5.3 Implementierung

- Laufzeiten bei Umsetzung mit Heaps
  - mkBag :: [a] → Bag a — O(n)
  - isEmpty :: Bag a → Bool — O(1)
  - union :: Bag a → Bag a → Bag a — O(log m + log n)
  - minBag :: Bag a → a — O(1)
  - delMin :: Bag a → Bag a — O(1)
- Das Erzeugen von Bags hat nur noch lineare Komplexität
- Das Vereinen von Bags geschieht nicht mehr in linearer, sondern in logarithmischer Laufzeit

6. Flexible Arrays

6.1 Merkmale und Operationen

- spezielle Art von Listen
- Anfügen und Löschen von Elementen am Anfang und am Ende
- Lesen und Schreiben von Werten auch mitten im Array

6. Flexible Arrays

6.1 Merkmale und Operationen

- empty :: Flex a
- isEmpty :: Flex a → Bool
- access :: Flex a → Int → a
- update :: Flex a → Int → a → Flex a
- hiext :: a → Flex a → Flex a
- hirem :: Flex a → Flex a
- loext :: a → Flex a → Flex a
- lorem :: Flex a → Flex a

6. Flexible Arrays

6.2 Algebraische Spezifikationen

- Axiomatische Grundlage für die Implementierung
  - hiext x . loext y = loext y . hiext x
  - hirem empty = error
  - hirem (hiext x xf) = xf
  - hirem (loext x empty) = empty
  - hirem (loext x (hiext y xf)) = loext x xf
  - hirem (loext x (loext y xf)) =

    loext x (hirem (loext y xf))

6. Flexible Arrays

6.2 Algebraische Spezifikationen

- Axiomatische Grundlage für die Implementierung
  - access empty k = error
  - access (loext x xf) 0 = x
  - access (loext x xf) (k + 1) = access xf k
  - access (hiext x xf) k

    | (k < n) = access xf k

    | (k == n) = x

    | (k > n) = error

    where n = length xf

6. Flexible Arrays

6.2 Algebraische Spezifikationen

- Axiomatische Grundlage für die Implementierung
  - length empty = 0
  - length (hiext x xf) = length xf + 1
  - length (loext xf) = length xf + 1

6. Flexible Arrays

6.3 Implementierung

- Ziel: Bilden eines quasi-perfect size-augmented binary tree
  - data Flex a = Null

    | Leaf a

    | Fork Int (Flex a) (Flex a)
- Das Integer-Label speichert die Größe der beiden Teilbäume (Anzahl der Leafs)
- Speichern der Werte nur in den Blättern

6. Flexible Arrays

6.3 Implementierung

- Datentyp-Invariante 1 (Anzahl Knoten)
  - size Null = 0
  - size (Leaf x) = 1
  - size (Fork n xt yt) = n

6. Flexible Arrays

6.3 Implementierung

- Datentyp-Invariante 2
- Bilden von leaf-trees: entweder leerer Baum oder nicht-leerer Baum, in welchem der leere Baum nicht auftaucht
  - isLeafTree xt = isEmpty xt || isLeafy xt

```
where  isLeafy Null               = False
       isLeafy (Leaf x)           = True
       isLeafy (Fork n xt yt) =
         isLeafy xt && isLeafy yt
```

6. Flexible Arrays

6.3 Implementierung

- Datentyp-Invariante 3 (Höhe des Baumes)
  - height Null = 0
  - height (Leaf x) = 0
  - height (Fork n xt yt) =

    1 + (height xt `max` height yt)

6. Flexible Arrays

6.3 Implementierung

- statische Array-Operationen
  - access (Leaf x) 0 = x
  - access (Fork n xt yt) k = if k < m then access xt k

    else access yt (k - m)

    where m = size xt

6. Flexible Arrays

6.3 Implementierung

- statische Array-Operationen
  - update :: Flex a → Int → a → Flex a
  - update (Leaf y) 0 val = Leaf val
  - update (Fork n xt yt) pos val

```
    | (pos < m)   = Fork n (update xt pos val) yt
    | otherwise   = Fork n xt (update yt (pos - m) val)
          where m    = size xt
```

6. Flexible Arrays

6.3 Implementierung

- quasi-perfekte Bäume
  - der linke Teilbaum jedes rechten Teilbaums ist perfekt
  - der linke Teilbaum ist nicht zwingend perfekt
  - hiext x Null = Leaf x
  - hiext x (Leaf y) = Fork 2 (Leaf y) (Leaf x)
  - hiext x (Fork n xt yt) = Fork (n + 1) xt (hi x yt)

6. Flexible Arrays

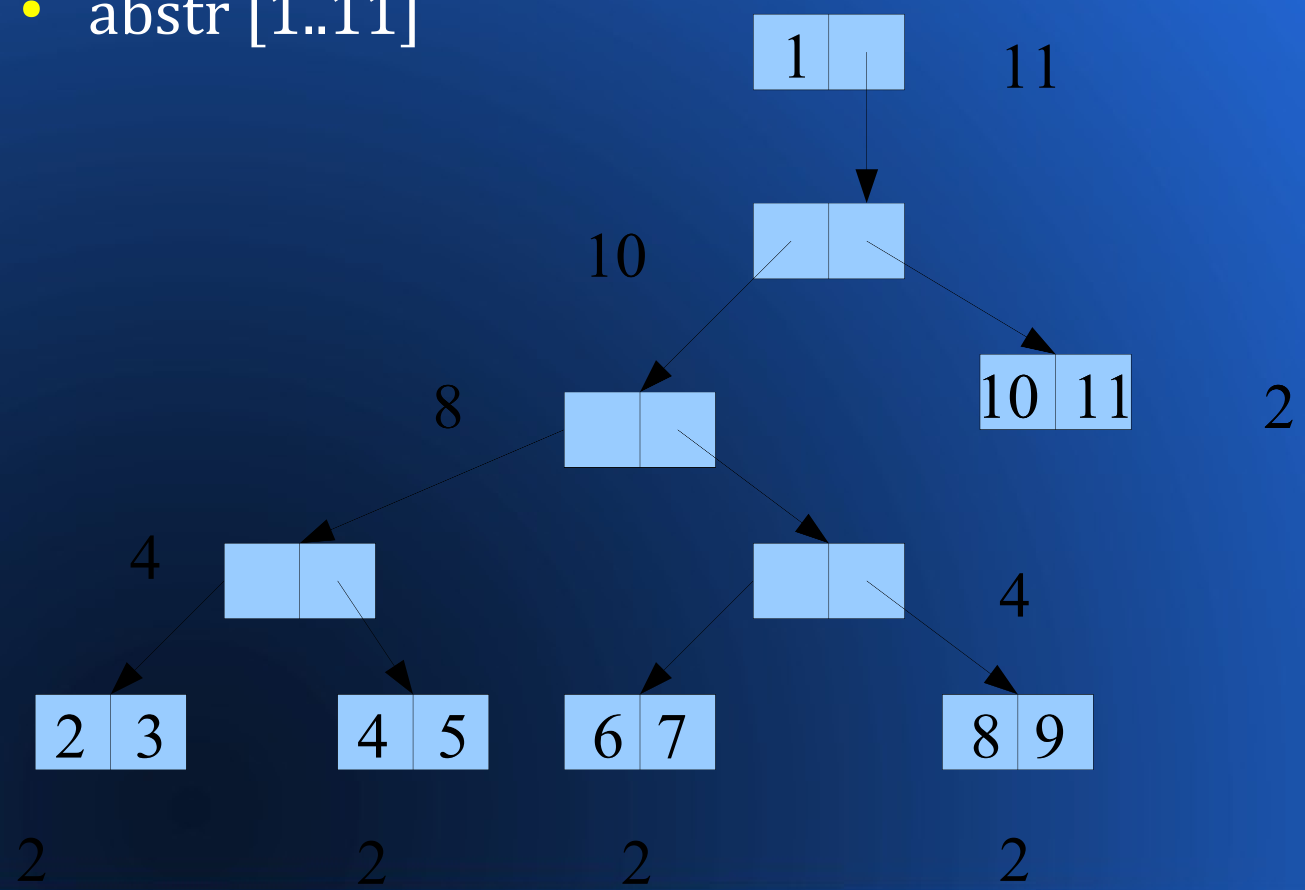
6.3 Implementierung

- hi x (Leaf y) = Fork 2 (Leaf y) (Leaf x)
- hi x (Fork n xt yt)
- | (size xt == size yt) = Fork (n + 1) (Fork n xt yt) (Leaf x)
- | otherwise = Fork (n + 1) xt (hi x yt)

6. Flexible Arrays

6.3 Implementierung

- abstr [1..11]



6. Flexible Arrays

6.4 Komplexität

	Liste	Baum
access	O(n)	O(h)
update	O(n)	O(h)
hiext	O(n)	O(h)
hirem	O(n)	O(h)
loext	O(1)	O(h)
lorem	O(1)	O(h)

7. Quellen

- Introduction to Functional Programming
  - von Richard Bird
  - Kapitel 8: Abstract datatypes
- Haskell Project Homepage
  - http://www.haskell.org/
- Wikibooks.org
  - http://en.wikibooks.org/wiki/Haskell