Effiziente funktionale Datenstrukturen

Alexander Nutz

3. Dezember 2007

Einführung

Rot-Schwarz-Bäume

Queues

Zusammenfassung

Einführung

- Aufgabe: die zeit- und platzeffiziente Implementierung von abstrakten Datentypen, z.B. eines Wörterbuches oder einer Zahlenmenge, für Programme
- bekannte Datenstrukturen: Listen, Bäume, Heaps, Queues, Stacks,...
- bisher in der Regel imperativ realisiert
- Zeigeroperationen werden nicht direkt unterstützt
  $\Rightarrow$ Übernahme imperativer Algorithmen i.d.R. ineffizient

Einführung

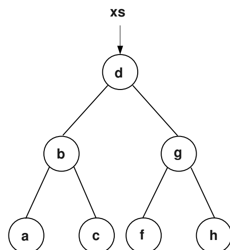
Rot-Schwarz-Bäume

Queues

Zusammenfassung

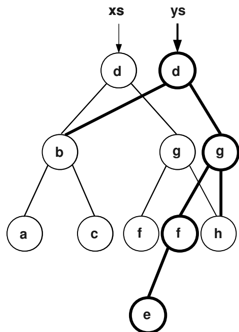
Persistenz

- keine zerstörerischen Updates (Wertzuweisungen)
- gleiche Daten werden nicht kopiert, sondern mehrfach verwendet (»Sharing«)



Persistenz

- keine zerstörerischen Updates (Wertzuweisungen)
- gleiche Daten werden nicht kopiert, sondern mehrfach verwendet (»Sharing«)



Binärsuchbaum in Haskell

```
data Tree e = E | T (Tree e) e (Tree e)

empty :: Tree e
empty = E

member :: Ord e => e -> Tree e -> Bool
member x E = False
member x (T a y b) | x < y    = member x a
                   | x == y   = True
                   | x > y    = member x b

insert :: Ord e => e -> Tree e -> Tree e
insert x E = (T E x E)
insert x (T a y b) | x < y    = T (insert x a) y b
                   | x == y   = T a y b
                   | x > y    = T a y (insert x b)
```

Nachteil: auf geordneten Daten ineffizient (Tiefe in $O(n)$)

Rot-Schwarz-Bäume

Binärsuchbäume mit zusätzlichen Eigenschaften:

- Jeder Knoten ist rot oder schwarz.
- Leere Knoten und die Wurzel sind schwarz.
- zwei Balance-Invarianten:
    - Invariante 1:
      Die Kind-Knoten eines roten Knotens sind schwarz.
    - Invariante 2:
      Jeder Pfad von der Wurzel zu einem Blatt enthält die gleiche Anzahl schwarzer Knoten.
      Diese Anzahl wird durch die »Schwarzhöhe« eines Knotens v, $Sh(v)$ beschrieben.

Tiefe von Rot-Schwarz-Bäumen (1)

Satz:
Die maximale Höhe eines Rot-Schwarz-Baumes mit Knotenzahl n beträgt $2\lfloor \log(n+1) \rfloor$.

Definitionen:
Sei $h(v)$ die Höhe des Teilbaums mit der Wurzel v,
sei $m(v)$ die Anzahl seiner innerer Knoten,
sei $Sh(v)$ seine Schwarzhöhe.

Lemma:

Es gilt: $m(v) \geq 2^{Sh(v)} - 1$ (Induktionsvorraussetzung)

Induktion über $h(v)$:

Induktionsanfang: $h(v) = 0 \Rightarrow Sh(v) = 0$

Es gilt: $m \geq 2^{Sh(v)} - 1 = 2^0 - 1 = 0$

Tiefe von Rot-Schwarz-Bäumen (2)

Induktionsschritt:

Sei v ein Knoten, v', v'' seine Kindknoten.

$h(v) = k \Rightarrow h(v') = h(v'') = k - 1$

$\Rightarrow Sh(v') = Sh(v) - 1$

oder $Sh(v') = Sh(v)$, analog für v'' $(*)$

verwende Induktionsvorraussetzung:

$\Rightarrow m(v') \geq 2^{Sh(v')} - 1$

mit $(*)$ folgt:

$\Rightarrow m(v) = m(v') + m(v'') + 1 \geq$

$(2^{Sh(v)-1} - 1) + (2^{Sh(v)-1} - 1) + 1 = 2^{Sh(v)} - 1$

fertig

Tiefe von Rot-Schwarz-Bäumen (3)

Beweis des Satzes:

(Die maximale Höhe eines Rot-Schwarz-Baumes mit Knotenzahl n beträgt $2\lfloor \log(n+1) \rfloor$.)
Da mindestens die Hälfte der Knoten auf jedem Pfad von der Wurzel w zu einem Blatt schwarz sind (Invariante 1), gilt:

$$Sh(w) \geq \frac{h(w)}{2}$$

wir benutzen das Lemma:

$$n \geq 2^{\frac{h(w)}{2}} - 1$$

$$\Leftrightarrow \log_2(n+1) \geq \frac{h(w)}{2}$$

$$\Leftrightarrow 2\log_2(n+1) \geq h(w)$$

$$\Leftrightarrow h(w) \leq 2\log_2(n+1)$$

□

Implementierung (1)

Typdefinitionen, member-Funktion

```
data Color = R | B
data Tree e = E | T Color (Tree e) e (Tree e)

empty :: Tree e
empty = E

member :: Ord e => e -> Tree e -> Bool
member x E = False
member x (T _ a y b) | x < y  = member x a
                     | x == y = True
                     | x > y  = member x b
```

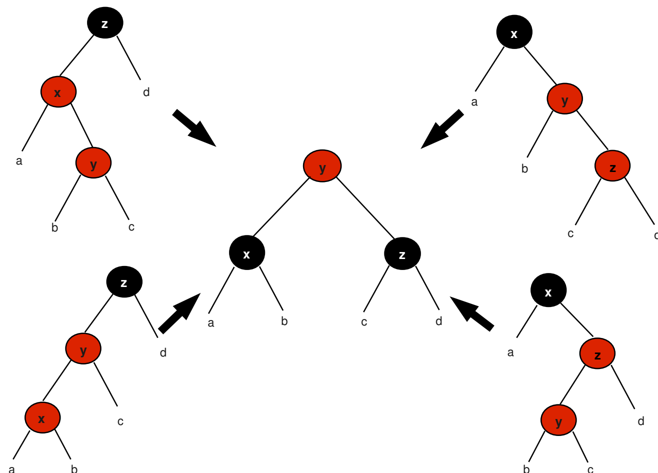
Implementierung (2)

insert-Funktion

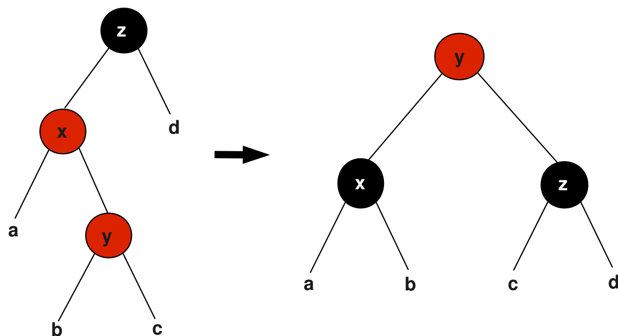
```
insert :: Ord e => e -> Tree e -> Tree e
insert x s = makeBlack (ins s) where
  ins E = T R E x E
  ins (T color a y b)
   | x < y  = balance color (ins a) y b
   | x == y = T color a y b
   | x > y  = balance color a y (ins b)
  makeBlack (T _ a y b)   = T B a y b

  balance B (T R (T R a x b) y c) z d
                = T R (T B a x b) y (T B c z d)
  balance B (T R a x (T R b y c)) z d
                = T R (T B a x b) y (T B c z d)
  balance B a x (T R (T R b y c) z d)
                = T R (T B a x b) y (T B c z d)
  balance B a x (T R b y (T R c z d))
                = T R (T B a x b) y (T B c z d)
  balance color a x b     = T color a x b
```

Analyse (1)

Vier Probleme, eine Lösung



Analyse (2)




```
balance B (T R a x (T R b y c)) z d
                        = T R (T B a x b) y (T B c z d)
```

Zusammenfassung

- effiziente Baum-Datenstruktur (alle Operationen in $O(\log(n))$
- sehr elegante funktionale Implementierung möglich
- noch weitere Verbesserungsmöglichkeiten bei der Effizienz (s.h. Übung)

Queues

- Queues (»Schlangen«, FIFO-Queues) arbeiten nach dem First-In-First-Out-Prinzip
- imperativ: mit Zeigern einfach zu implementieren, alle Operationen in $O(1)$
- drei Operationen:
  - *head*: auslesen des ersten Elements

    ```
    head :: Queue a -> a
    ```

  - *tail*: entfernen des ersten Elements, zurückliefern der restlichen Queue

    ```
    tail :: Queue a -> Queue a
    ```

  - *snoc*: einreihen eines neuen Elements in eine Queue (»cons von rechts«)

    ```
    snoc :: a-> Queue a -> Queue a
    ```

Naiver Algorithmus

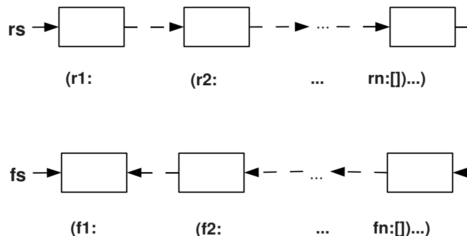
```
data Queue a  = Q [a]

head (Q (x:xs)) = x

tail (Q (x:xs)) = xs

snoc y (Q []) = Q (y:[])
snoc y (Q xs) = Q (xs ++ [y])
```

Problem: *snoc* in $O(n)$.

Effizienter Algorithmus - Idee

Implementierung durch zwei Listen *fs* (Front) und *rs* (Rear)

- *rs* wird invertiert $\Rightarrow$ Zugriff auf letztes Element in $O(1)$
- *fs* wird durch die *tail*-Operation leer $\Rightarrow$ mache *rs* zu *fs*, ersetze *rs* durch die leere Liste
- Invariante: Wenn *fs* leer ist, muss die Queue leer sein.



Effizienter Algorithmus - Code

```
import Prelude hiding (head,tail)

data Queue a = Q [a] [a]

check [] rs = Q (reverse rs) []
check fs rs  = Q fs rs

isEmpty (Q fs rs) = null fs

snoc (Q fs rs) x = check fs (x:rs)

head (Q [] _) = error "empty Queue"
head (Q (x:fs) rs) = x

tail (Q [] _) = error "empty Queue"
tail (Q (x:fs) rs) = check fs rs
```


Einführung

Rot-Schwarz-Bäume

Queues

Zusammenfassung

Amortisierte Kostenanalyse mittels Potentialmethode

Definiere die Potentialfunktion $\Phi(rs) = |rs|$ als die aktuelle Länge der hinteren Liste.

- *head*: immer Kosten von 1
- *snoc*: benötigt selbst Kosten von 1, erhöht das Potential um 1 $\Rightarrow$ amortisierte Kosten von 2
- *tail* :
  - falls *fs* nicht leer wird: Kosten 1
  - falls *fs* leer wird:
    Kosten $reverse(rs) + 1 - \Phi(rs) = |rs| + 1 - |rs| = 1$

Zusammenfassung

- effizienter funktionaler Algorithmus
- amortisiert gleiche Kosten wie bei imperativem Algorithmus
- Will man Persistenz nutzen (mehrmals auf dieselbe Version einer Queue *tail* anwenden), liegen die Kosten für *tail* dennoch in $O(n)$.

Gesamtzusammenfassung

- Persistenz ist immer gegeben
- keine Zeigeroperationen $\Rightarrow$ oft übersichtlichere Algorithmen, als in imperativen Sprachen
- amortisierte Analyse hilfreich, berücksichtigt aber erstmal keine Persistenz, kann aber angepasst werden (nicht Teil dieses Vortrags)
- neue Sichtweisen sind notwendig, erweisen sich aber oft als fruchtbar
- weiterführende Literatur: Chris Okasaki, Purely Functional Data Structures