Informatik-Seminar 2003 - Thema 6: Bäume

Robin Brandt

14. November 2003

Übersicht

Beispiel: Huffman-Bäume (Binäre Bäume)

Definition eines Baumes

Eigenschaften:

- Menge von Punkten (Knoten) und Linien (Kanten)
- eine Kante verbindet zwei Knoten
- ein Baum besitzt 3 Eigenschaften:
  - einen hervorgehobener Knoten, die Wurzel
  - Jeder Knoten c außer der Wurzel ist durch eine Kante mit einem anderen Knoten p, dem Vater von c, verbunden
  - ein Baum ist zusammenhängend
- spezielle Knoten ohne “Kinder”: Blätter
- nicht-Blattknoten auch innere Knoten genannt

Eigenschaften

Binärbaum: Variante eines Baumes mit zwei "Stellen" für Kinder

Definition in Haskell (rekursive Datenstruktur):

```
data BTree a = Leaf a
             | Fork (BTree a) (BTree a)
```

Definierte Eigenschaften

- Größe: Anzahl der Blätter
- Höhe: Entfernung zum weitesten Punkt von der Wurzel aus
- Anzahl innere Knoten = Anzahl Blätter - 1

Eigenschaften

In Haskell:

- Verarbeitung mit Pattern Matching

```
size :: BTree a -> Int
size (Leaf x) = 1
size (Fork xt yt) = size xt + size yt

height :: BTree a -> Int
height (Leaf x) = 0
height (Fork xt yt) = 1 + max (height xt) (height yt)

nodes :: BTree a -> Int
nodes (Leaf x) = 0
nodes (Fork xt yt) = 1 + nodes xt + nodes yt
```

Glätten eines Bäumes

- Umwandlung des Inhalts des Baumes in eine Liste
- Reihenfolge der Listenelemente entspricht der Blätterfolge von links nach rechts

Umsetzung in Haskell:

```
flatten :: BTree a -> [a]
flatten (Leaf x) = [x]
flatten (Fork xt yt) = flatten xt ++ flatten yt
```

Erzeugen eines Baumes aus einer Liste

- Umkehroperation zum Glätten
- Idee: Liste in zwei Teile unterteilen und daraus neue Teilbäumen erzeugen (Rekursion)

```
mkBTree :: [a] -> BTree a
mkBTree xs
    | (m == 0) = Leaf (unwrap xs)
    | otherwise = Fork (mkBTree ys) (mkBTree zs)
    where m = (length xs) `div` 2
          (ys, zs) = splitAt m xs
          unwrap [x] = x
```

mapBtree-Operation

- analog zu map auf Listen
- erzeugt aus einem Baum und einer Funktion einen neuen Baum

```
mapBTree :: (a -> b) -> BTree a -> BTree b
mapBTree f (Leaf x) = Leaf (f x)
mapBTree f (Fork xt yt) = Fork (mapBTree f xt) (mapBTree f yt)
```

foldBtree

- analog zu fold auf Listen
- berechnet aus zwei Funktionen und einem Baum einen Wert
- 1. Funktion verarbeitet ein Blattelement
- 2. Funktion verarbeitet zwei verarbeitete (!) Teilbäume

```
foldBTree :: (a -> b) -> (b -> b -> b) -> BTree a -> b
foldBTree f g (Leaf x) = f x
foldBTree f g (Fork xt yt) = g (foldBTree f g xt) (foldBTree f g yt)
```

Zurückführen einfacher Operationen auf fold

- Motivation: Traversieren des Baumes ausgliedern und wiederverwenden

```
size :: BTree a -> Int
size (Leaf x) = 1
size (Fork xt yt) = size xt + size yt

height :: BTree a -> Int
height (Leaf x) = 0
height (Fork xt yt) = 1 + max (height xt) (height yt)

size2 = foldBTree (const 1) (+)
height2 = foldBTree (const 0) maxsubs
  where m `maxsubs` n = 1 + m `max` n
```

Erweiterte binäre Bäume

- bisher: Information nur in den Blattknoten
- Erweiterung: auch die inneren Knoten sollen Informationen aufnehmen können
- Ziel: Entwicklung effizienterer Algorithmen
- Beispiel: Größeninformation im inneren Knoten für einen schnellen indizierten Zugriff auf den Baum (vgl. (!!) bei Listen)

Datendefinition

```
data ATree a = Leaf a
             | Fork Int (ATree a) (ATree a)
```

- zusätzliche Int-Komponente speichert die Gesamtgröße des linken Teilbaum
- muß bei Baumerzeugung zugesichert werden (z.B. durch eigene Funktion zum Erzeugen)
- Beispiel an der Tafel

Einleitung

- auch “labelled binary trees” genannt
- verwendet zur effizienten Suche
- recht effizient für Mengenrepräsentationen ($\in$ - Test proportional zu h(t))
- Eigenschaften (Größe, Höhe) und Operationen (flatten, map) ähnlich zu einfachen binären Bäumen
- (sorted . flatten) searchtree == True
- keine Blattknoten mehr, alle Informationen in den inneren Knoten

Datendefinition

- Nur für Elemente aus Ord-Typklasse definiert
- Alle kleineren Elemente liegen im linken, alle größeren im rechten Teilbaum

```
data (Ord a) => Stree a = Null
                        | Fork (Stree a) a (Stree a)
```

Erzeugung aus einer Liste

- ähnlicher Ablauf wie bei mkBTree
- Aufteilung der Liste in eine kleinere und eine größere Hälfte

```
mkStree :: (Ord a) => [a] -> Stree a
mkStree [] = Null
mkStree (x:xs) = Fork (mkStree smaller) x (mkStree larger)
                 where (smaller,larger) = partition (<= x) xs

partition :: (a->Bool) -> [a] -> ([a],[a])
partition p xs = ([ x | x <- xs, p x],
                  [ y | y <- xs, (not . p) y])
```

Sortieren mit Hilfe von Suchbäumen

- es gilt: (sorted . flatten) searchtree == True
- Idee: Umwandlung einer Liste in einen Baum und anschließendes Glätten ergibt eine sortierte Liste

```
sort :: (Ord a) => [a] -> [a]
sort = flatten . mkStree
```

- Entfernen des "Zwischenbaums" ergibt eine Implementierung von Quicksort

Einfügen und Löschen aus Suchbäumen

- Grundfunktionen für Mengenverarbeitung
- Ablauf an der Tafel

Einleitung

- struktureller Aufbau nahezu identisch zu Suchbäumen
- Daten werden im Baum jedoch anders angeordnet: Ein Datum eines Knotens ist kleinergleich allen Daten *aller* Unterbäume
- für andere Anwendungen als Suchbäume geeignet:
  - kleinstes Element ist in konstanter Zeit gefunden
  - Zusammenfügen zweier Heap-Trees ist sehr schnell
- Heap Trees werden in einem späteren Vortrag genauer behandelt

Einleitung

- Baumtyp mit Mehrfachverzweigungen

```
data Rose a = Node a [Rose a]
```

- bei externen Knoten ist die Liste der Unterbäume leer
- binäre Bäume sind strukturgleich zu "Rose Trees" 2. Ordnung
- Möglich: Erzeugung von "unendlichen Bäumen"

```
r1 = Node 1 [ Node n [] | n <- [1..]]
```

Möglichkeiten der Verarbeitung

- Depth-First
  - Datum am Knoten verarbeiten
  - anschließend Daten des linkesten Teilbaumes
  - dann der zweit-linke Teilbaum ...
- Breadth-First
  - Daten am Knoten verarbeiten
  - dann alle Knoten der Teilbäume (erste Ebene)
  - weiter mit unteren Ebenen ...

Einleitung

- werden benutzt zur Datenkomprimierung
- Idee: Umkodierung des zu komprimierenden Textes (Bildes,...) mit Hilfe Codes variabler Länge (EDV 1)
- häufig auftauchende Zeichen werden kürzer kodiert als selten auftauchende
- Auftrittswahrscheinlichkeit h(x) $\sim$ 1/Codelänge

Aufteilung des Problems in Unterprobleme

- Informationen über Auftrittshäufigkeiten sammeln
- Aufbau eines binären Huffman-Baums
- Kodierung eines Textes
- Dekodierung

Dekodieren

```
decode :: BTree Char -> [Bit] -> [Char]
decode t cs
    = if null cs then [] else decode1 t cs
    where decode1 (Leaf x) cs = x : decode t cs
          decode1 (Fork xt yt) (L:cs) = decode1 xt cs
          decode1 (Fork xt yt) (R:cs) = decode1 yt cs
```

- Funktion erzeugt aus dem Huffman-Baum und einer Bitfolge den Ursprungstext
- Beispiel an der Tafel

Kodieren

```
encode :: BTree Char -> [Char] -> [Bit]
```

- Vorgehensweise: Suche im Baum nach dem zu kodierenden Buchstaben und merken des Pfades von der Wurzel
- leider ineffizient!
- Zwischenschritt, der aus Baum eine Codetabelle baut verbessert die Laufzeit

Analyse

```
sample :: [Char] -> [(Char,Int)]
```

- Erzeugt Liste aus Buchstaben und ihren Auftrittswahrscheinlichkeiten
- Idee: Text nach Buchstaben sortieren und mehrfaches Auftreten zusammenfassen

```
sample = sortby freq . collate . sortby id
```

Baumerzeugung

```
mkHuff :: [(Char,Int)] -> HuffTree
mkHuff = unwrap . until singleton combine . map mktip
```

- erzeugt aus der Liste mit den Buchstabenhäufigkeiten einen Huffman-Baum

```
data HuffTree = Tip Int Char
              | Node Int HuffTree HuffTree
              deriving(Show)
```

- zusätzliches Datum beinhaltet Gewicht (Optimierung)

Zusammenfassung Algorithmus

1. aus Analyse-Tupeln Blätter (*Tips*) erzeugen
2. die ersten beiden Bäume der Liste zu einem Knoten kombinieren (Gewicht des Knotens ist die Summe der Gewichte der kombinierten Bäume)
3. erzeugten Knoten in die nach Gewichten aufsteigend sortierte Liste einsortieren
4. gehe zu Schritt 2 bis die Liste nur noch ein Element besitzt
5. abschließend Gewichte aus Baum entfernen