Programmierung und Modellierung mit Haskell

Weitere Algorithmen und Laufzeitbetrachtungen

Martin Hofmann Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

26. Mai 2014

Vorne- oder hinten Anhängen?

Noch einmal die beiden Versionen von `reverse`, vgl. Hausaufgabe.

```
revAcc :: [a] -> [a] -> [a]
revAcc [] acc = acc
revAcc (x:l) acc = revAcc l (x:acc)

revSpec :: [a] -> [a]
revSpec [] = []
revSpec (x:l)  = revSpec l ++ [x]
```

Wir wollen die Laufzeit und den Platzverbrauch empirisch untersuchen.

Laufzeitmessung

```
testlist n = [n,n-1..1]

test :: ([Int] -> [a]) -> Int -> Int
test f n = let l = f (testlist n) in length l
```

Experimenteller Vergleich:

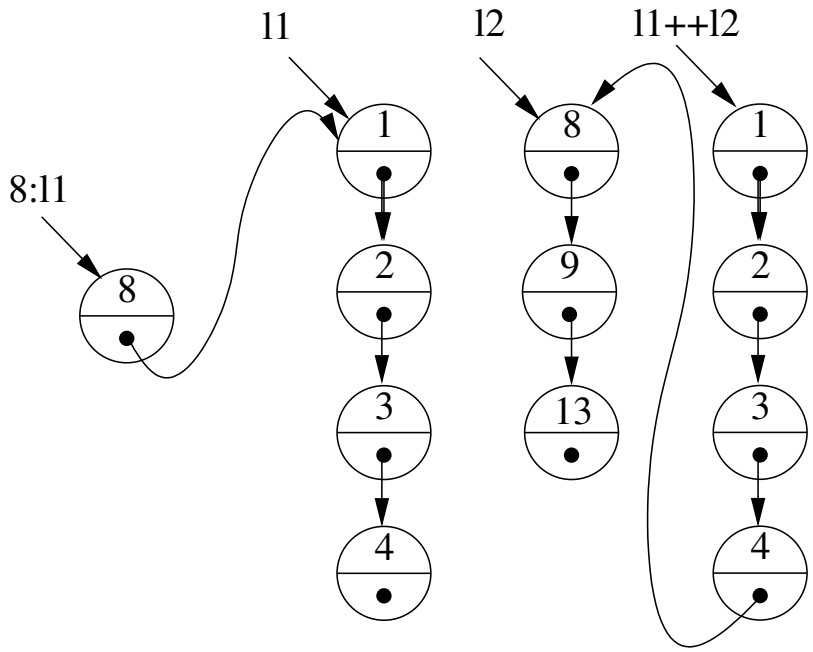
```
*Laufzeit> test (\l -> revAcc l []) 10000
10000
(0.04 secs, 3031908 bytes)
*Laufzeit> test (revSpec) 10000
10000
(3.27 secs, 2060530908 bytes)
```

Hängt es mit der Endrekursion zusammen?

```
filterAcc :: (a->Bool) -> [a] -> [a] -> [a]
filterAcc p [] acc = acc
filterAcc p (x:l) acc | p x = filterAcc p l (acc++[x])
                      | otherwise = filterAcc p l acc

filterSpec :: (a->Bool) ->  [a] -> [a]
filterSpec p [] = []
filterSpec p (x:l) | p x = x:
filterSpec p l
                   | otherwise = filterSpec p l
*Laufzeit> test (\l-> filterAcc (\_ ->True) l []) 10000
10000
(3.31 secs, 2318824116 bytes)
*Laufzeit> test (filterSpec (\_ -> True)) 10000
10000
(0.04 secs, 4467588 bytes)
```

Anmerkungen zur Laufzeit

- Listen sind im Rechner als Ketten von Einträgen gespeichert und können vom Kopf her angesehen werden
- Ein "cons", also `x:l` benötigt konstante Zeit
- Abgleich gegen ein Muster der Form `kopf:rumpf` benötigt konstante Zeit (ein paar Zykeln)
- Eine Verkettung `l1 ++ l2` benötigt Zeit proportional zur Länge von `l1`.



Anmerkungen zur Laufzeit

- Zeitaufwand "`reverse`":
  $T_{\texttt{revSpec}}(n+1) = T_{\texttt{revSpec}}(n) + O(n) \leadsto T_{\texttt{revSpec}(n)} = O(n^2)$
- Zeitaufwand "`revAcc`": $T_{\texttt{revAcc}}(m+1, n) =$
  $T_{\texttt{revAcc}}(m, n) + O(1) \leadsto T_{\texttt{revAcc}}(m, n) = O(m)$

Erinnerung: $O(f(n))$ bezeichnet eine Funktion die durch $c \cdot f(n)$ für festes c beschränkt werden kann. Also $O(n)$ (höchstens) linear; $O(n^2)$ (höchstens) quadratisch.

Wdh.: Sortieren durch Einfügen

Eine Liste $[x_1, \ldots, x_n]$ von heißt **sortiert**, wenn $x_1 \leq x_2 \leq \cdots \leq x_n$.

```
insertel :: Ord a => a -> [a] -> [a]
insertel x [] = [x]
insertel x (y:l) = if x <= y then x:y:l
                             else y:insertel x l

inssort :: Ord a => [a] -> [a]
inssort [] = []
inssort (x:l) = insertel x  (inssort l)
```

Ist l sortiert, so auch `insertel` a l und es enthält dieselben Elemente wie $a : l$.
Für beliebiges l ist `inssort`(l) sortiert und enthält dieselben Elemente wie l.

Beispiel: Sortieren durch Mischen

```
mergesort :: Ord a => [a] -> [a]
mergesort [] = []
mergesort [a] = [a]
mergesort l = merge (mergesort l1)  (mergesort l2)
      where (l1,l2) = split l
            split [] = ([],[])
            split [a] = ([a],[])
            split (a:b:u) = (a:u1,b:u2)
                  where (u1,u2) = split u
            merge [] [] = []
            merge [] u = u
            merge u [] = u
            merge (x:u)(y:v) = if x<=y
                  then x:merge u (y:v)
                  else y:merge (x:u) v
```

Experimenteller Vergleich

```
*Laufzeit> test (inssort) 3000
3000
(3.87 secs, 697815456 bytes)
*Laufzeit> test (mergesort) 3000
3000
(0.07 secs, 8646792 bytes)
*Laufzeit> test (mergesort) 1000000
1000000
(41.24 secs, 4410681016 bytes)
```

Analytische Abschätzung der Laufzeit

- Zeitaufwand
  $T_{\texttt{insertel}}(n+1) = T_{\texttt{insertel}}(n)+O(1) \rightsquigarrow T_{\texttt{insertel}}(n) = O(n)$
- Zeitaufwand
  $T_{\texttt{inssort}}(n+1) = T_{\texttt{inssort}}(n) + O(n) \rightsquigarrow T_{\texttt{inssort}}(n) = O(n^2)$
- Zeitaufwand $T_{\texttt{mergesort}}(n) = 2\,T_{\texttt{mergesort}}(n/2) + O(n) \rightsquigarrow$
  $T_{\texttt{mergesort}}(n) = O(n \log(n))$

Effizienzproblem bei breadthForest

```
breadthForest :: [Tree a] -> [a]
breadthForest [] = []
breadthForest (Empty : frst) = breadthForest frst
breadthForest (Node x l r : frst) =
                  x : breadthForest (frst ++ [l, r])
```

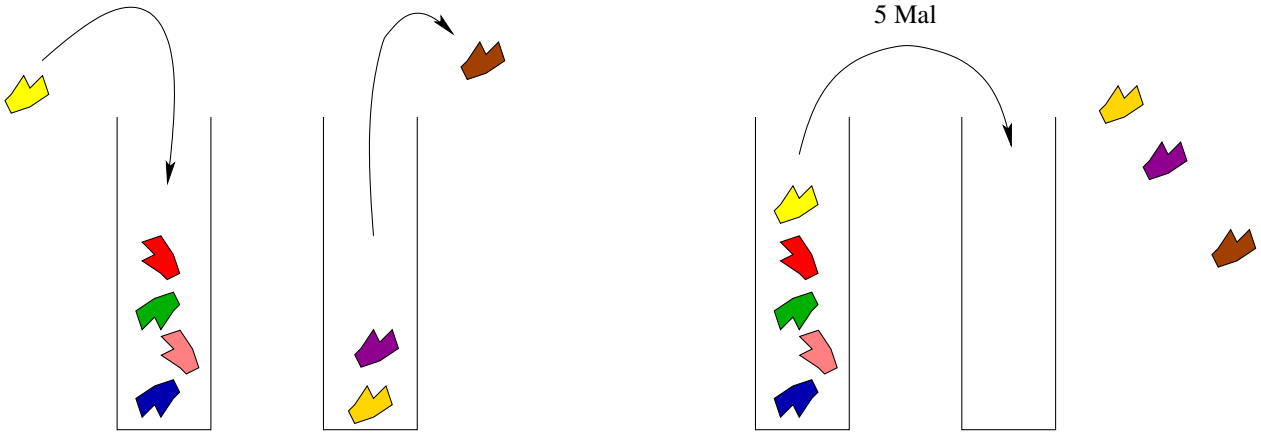
Einhängen von l, r braucht Zeit $O(n)$, wenn n die Größe des Waldes ist.

```
fulltree 0 = Empty
fulltree n = let l = fulltree (n-1) in Node 0 l l
*Tree> length (breadthForest [fulltree 14])
16383
(8.57 secs, 5890705980 bytes)
*Tree> length (breadthForest [fulltree 16])
65535
(506.69 secs, 94519577072 bytes)
```

Amortisierte Schlange

Verkettete Listen wie in Haskell unterstützen sehr effizient die Datenstruktur Stapel (Stack, last-in-first-out, LIFO), haben aber ein Effizienzproblem, wenn sie direkt als Schlange (first-in-first-out, FIFO) eingesetzt werden.

Abhilfe bietet die Implementierung einer Schlange durch zwei Stapel: Eingangsstapel ("in tray") und Ausgangsstapel.
Einfügen in den Eingangsstapel, Entnehmen vom Ausgangsstapel.
Wird der Ausgangsstapel leer, so wird der gesamte Eingangsstapel en-bloc in den Ausgangsstapel verschoben.



Anwendung und Laufzeit

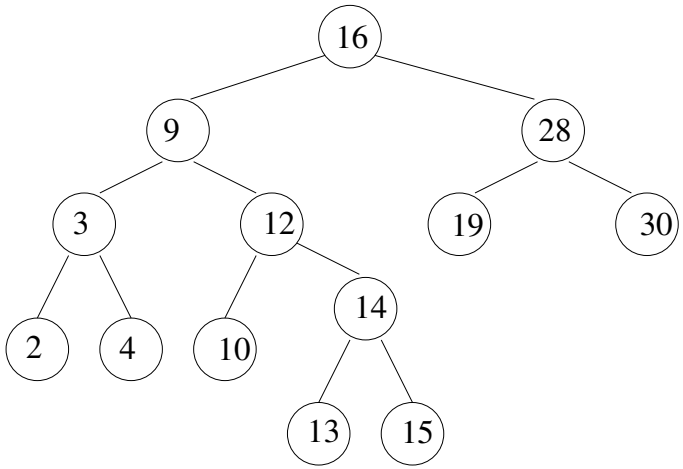
```
breadthForest2 :: [Tree a] -> [Tree a] ->  [a]
breadthForest2 [] [] = []
breadthForest2 [] in_tray =
           breadthForest2 (reverse in_tray) []
breadthForest2 (Empty : frst) in_tray =
           breadthForest2 frst in_tray
breadthForest2 (Node x l r : frst) in_tray =
           x:breadthForest2 frst (r:l:in_tray)

*Tree> length (breadthForest2 [fulltree 16][])
65535
(0.14 secs, 13648716 bytes)
*Tree> length (breadthForest2 [fulltree 20][])
1048575
(2.36 secs, 214595844 bytes)
```

Analyse der Laufzeit

- Die **`reverse`** Operation benötigt auch lineare Zeit, tritt aber nur selten auf.
- Jeder Baum wird dreimal bewegt. Beim Einhängen, beim Hinüberkopieren, beim Aushängen.
- Potentialmethode: Man berechnet beim Einhängen zusätzlich zu den tatsächlichen Kosten von, sagen wir, 1EUR, noch fiktive Kosten von 1EUR um für die spätere **`reverse`** Operation "vorzusorgen".
- Das Einhängen in die Schlange verursacht so Kosten von 2EUR, das Aushängen, gleich ob **`reverse`** erforderlich ist, oder nicht, kostet 1EUR, da das **`reverse`** aus dem angesparten Kapital bezahlt werden kann. Mehr dazu in A&DS

Binärer Suchbaum

In einem binären Suchbaum (BST) sind die Knotenmarkierungen des linken Teilbaums kleiner oder gleich der Wurzelmarkierung und die Knotenmarkierungen des rechten Teilbaums größer oder gleich der Wurzelmarkierung. Alle Teilbäume sind selbst wieder (BST)



Für jeden Knoten mit Markierung x gilt:

- Die Markierungen des linken Teilbaumes sind $\leq x$;
- Die Markierungen des rechten Teilbaumes sind $\geq x$;

Suchen in BST

```
mem_BST :: Ord a => a -> Tree a -> boolean
mem_BST x Empty = False
mem_BST x (Node y l r) | x==y = True
                       | x< y = mem_BST x l
                       | x> y = mem_BST x r
search_BST :: Ord a => a -> Tree (a,b) -> Maybe b
search_BST x Empty = Nothing
search_BST x (Node (k,v) l r) | x==k = Just v
                              | x< k = search_BST x l
                              | x> k = search_BST x r
```

- Der Vergleich mit der Wurzel erlaubt es, die Suche auf einen der Teilbäume zu beschränken.
- BST können auch zur effizienten Speicherung von Schlüssel-Wert-Paaren (key-value-pairs), also endlichen Abbildungen, verwendet werden. `search_BST` beschreibt das Aufsuchen eines Wertes, der zu einem gegebenen Schlüssel gehört.

Einfügen in BST

```
ins_BST :: Ord a => a -> Tree a -> Tree a

ins_BST x Empty  = leaf x
ins_BST x (Node y l r) | x<=y =
            Node y (ins_BST x l) r
                   | otherwise =
            Node y l (ins_BST x r)
```

Je nach Größe rekursiv links oder rechts einfügen.

Löschen aus BST

```
del_BST :: Ord t => t -> Tree t -> Tree t
del_BST x Empty = Empty
del_BST x (Node y l r) | x<y  = Node y (del_BST x l) r
del_BST x (Node y l r) | x>y  = Node y l (del_BST x r)
del_BST x (Node y Empty r) | x==y  = r
del_BST x (Node y l r) | x==y  =
              let (z,l1)=del_largest l
                          in Node z l1 r
              where
```

- `(z,t1) = del_largest t` entfernt den größten Eintrag `z` aus `t`, der resultierende Baum ist `t1`. Er steht ganz rechts in `t`.
- Muss man die Wurzel eines BST löschen, so kann man sie durch den größten Eintrag des linken Teilbaumes ersetzen ohne die BST Eigenschaft zu verletzen.
- Ist kein linker Teilbaum vorhanden, so kann man die Wurzel direkt entfernen (Ergebnis ist der rechte Teilbaum).

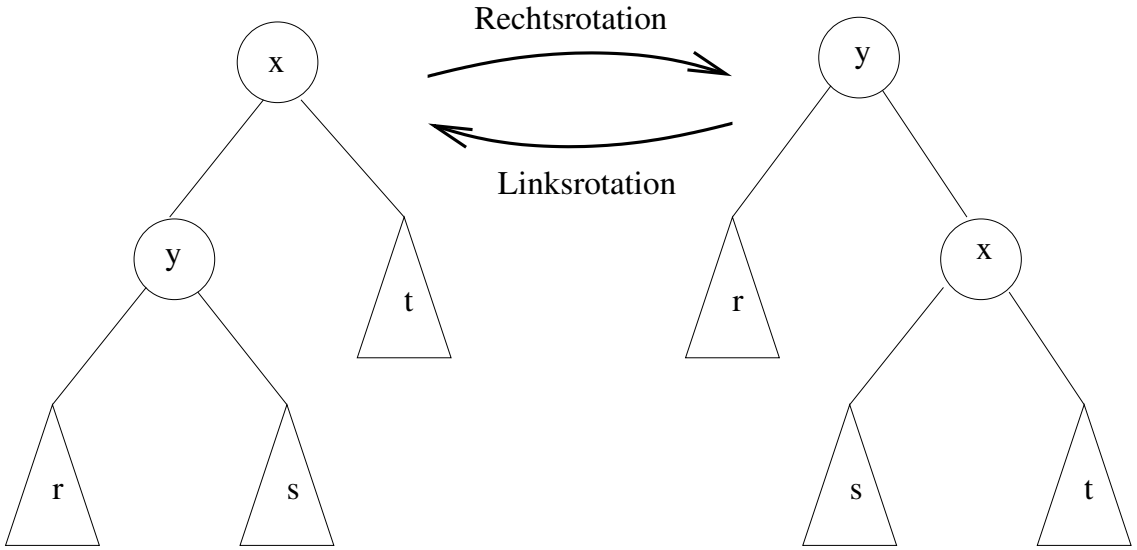
Löschen des grössten Eintrags

```
...
del_largest (Node x l Empty) = (x,l)
del_largest (Node x l r) =
      let (z,r1)=del_largest r in
                  (z,Node x l r1)
```

Laufzeitbetrachtungen für BST

- Suchen und Löschen in einem BST benötigt Zeit $O(d)$, wobei d die Höhe des Baumes ist.
- Ist der Baum gut ausgeglichen, so ist die Tiefe ungefähr gleich dem Zweierlogarithmus der Knotenzahl.
- Besteht der Baum nur aus einem Ast, so ist die Tiefe gleich der Knotenzahl.
- Durch geeignete Umstrukturierungen kann man erreichen, dass ein BST im wesentlichen ausgeglichen bleibt, gleich welche Einträge in ihn eingefügt und aus ihm gelöscht werden und in welcher Reihenfolge.
- Solche BST bilden eine effiziente Implementierung von endlichen Abbildungen.

Rotationen




```
rot_right (Node x (Node y r s) t) =
                          Node y r (Node x s t)
rot_left  (Node y r (Node x s t)) =
                          Node x (Node y r s) t
```

Verwaltung der Rotationen

Um festzustellen, wann rotiert werden muss, möchte man nicht global den ganzen Baum analysieren.

Stattdessen führt man in den Knoten geeignete Verwaltungsinformation mit:

- Höhendifferenz zwischen linkem und rechten Teilbaum abspeichern und jeweils im Bereich $\{-1, 0, 1\}$ halten. *(AVL-Bäume)*
- Ein Bit ("rot", "schwarz") abspeichern, sodass auf jedem Pfad von einem Knoten zu einem Blatt dieselbe Zahl schwarzer Knoten liegt und auf jeden schwarzen Knoten höchstens ein roter Knoten folgt. *(Rot-Schwarz-Bäume)*
- Man kann auch ganz ohne Verwaltungsinformation auskommen, wenn man auch bei den Suchoperationen rotiert. *(Splay-Bäume)*

Zusammenfassung

- Empirische Laufzeitmessung für Listenfunktionen
- Erklärung der beobachteten Laufzeit anhand eines Speichermodells für Listen: Laufzeit der Konkatenation `l++k` linear in der Länge von `l`
- Sortieren durch Mischen als Beispiel eines effizienten ($O(n \log n)$) Sortierverfahrens
- Effizientere Version der Breitensuche durch "amortisierte" Schlange und entsprechende Laufzeitanalyse.
- Binäre Suchbäume als effiziente Implementierung von Mengen und endlichenAbbildungen.