Klausur „Algorithmen und Datenstrukturen (AuD) I“
(WP zur Vorlesung WS 2009/10)

13. Juli 2010, 10:00 – 12:00

Aufgabe 1 [Gesamtpunktzahl: 10]

a) Gesucht ist eine Haskell-Funktion `remMultOccs :: Eq a => [a] -> [a]` , die aus beliebigen Listen Mehrfachvorkommen von Elementen dann entfernt, wenn diese *unmittelbar benachbart* vorkommen.
Beispiel:

```
Main> remMultOccs [1,2,2,3,4,4,4,2]
[1,2,3,4,2]
```

Überprüfen Sie, ob die folgende Implementation korrekt ist, und begründen Sie Ihre Antwort:

```
remMultOccs [] = []
remMultOccs [x] = [x]
remMultOccs (x:y:xs) = if x == y then y:remMultOccs xs
                       else x:remMultOccs (y:xs)
```

b) Definieren Sie ein Haskell-Prädikat `noReps :: Eq a => [a] -> Bool`, das genau dann zutrifft, wenn eine Liste *keine unmittelbar benachbarten* Wiederholungen von Elementen hat.
Beispiel:

```
Main> noReps [1,2,3,4]
True
Main> noReps [1,2,2,3,4]
False
```

c) Beschreiben Sie Ihr Vorgehen zunächst in Worten und implementieren Sie dann möglichst effizient in Haskell die folgende Verallgemeinerung der im Standard-Prelude vordefinierten Funktion `span`:

Die Funktion `span2 :: (a -> a -> Bool) -> [a] -> ([a],[a])` erwartet ein *zweistelliges Prädikat* und eine Liste und gibt ein Paar zurück aus zwei Listen: Die erste enthält diejenigen Elemente vom Anfang der Liste, für die alle jeweils das Prädikat *für das Element und das unmittelbar nachfolgende Element* (sofern letzteres existiert) zutraf. Die zweite enthält diejenigen Elemente der Liste ab dem Element einschliesslich, für welches das Prädikat *zum ersten Mal nicht mehr* für das Element und das unmittelbar nachfolgende Element zutraf.
Beispiele:

```
Main> span2 (<) [1,2,3,4,4,2,0]
([1,2,3],[4,4,2,0])
Main> span2 (<=) [1,2,3,4,4,2,0]
([1,2,3,4],[4,2,0])
Main> span2 (>=) [1,2,3,4,4,2,0]
([],[1,2,3,4,4,2,0])
Main> span2 (/=) [1,2,3,4,2,0]
([1,2,3,4,2,0],[])
```

Was ist ein schlechtester Fall für Ihren Algorithmus (Begründung)? Welche Komplexität ergibt sich daraus ($\mathcal{O}$-Notation)?

Aufgabe 2 [Gesamtpunktzahl: 10]

a) Bauen Sie schrittweise von Hand mit den Elementen der folgenden Liste (in der Reihenfolge von links nach rechts) einen (gewöhnlichen) **binären Suchbaum** auf und skizzieren Sie das Ergebnis:

```
[5,10,7,3,1,8,9]
```

b) Bauen Sie dann schrittweise von Hand mit den Elementen der Liste aus a) (in der Reihenfolge von links nach rechts) einen **Rot-Schwarz-Baum** auf. Erläutern Sie dabei insbesondere jeweils mit Skizzen, wann (d.h. bei welchen Einfügungen) und warum Sie welche Rotation verwenden und zeigen Sie deren Auswirkung auf die Gestalt des Baumes und die Färbung der Knoten.

c) Für die Navigation in einem binären Suchbaum sei zu einem Element, sofern dieses in einem Knoten des Baumes enthalten, der *Elternknoten* dieses Knotens zu bestimmen.
c1) Erläutern Sie zunächst mit Worten (und ggf. Skizzen), welche verschiedenen Fälle Sie behandeln müssen.
c2) Implementieren Sie dann **in Haskell** die Funktion:

```
parent :: (Show a, Ord a) => a ->  BinTree a -> (Bool, BinTree a)
```

Der Bool-Wert gibt dabei an, ob der Knoten mit dem als erstem Argument übergebenen Wert im Baum existiert. Falls ja, ist das zweite Element des Paars der gesuchte Elternknoten (falls existent), sonst ist es immer der leere Baum. Ist das Element der Wurzelknoten, so soll das zweite Element des Paars ebenfalls der leere Baum sein.
Zur Erinnerung:

```
data (Ord a) => BinTree a = EmptyBT
                          | NodeBT a (BinTree a) (BinTree a)
    deriving (Show,Eq)

emptyTree = EmptyBT

inTree v' EmptyBT                  = False
inTree v' (NodeBT v lf rt) | v==v' = True
                           | v'<v  = inTree v' lf
                           | v'>v  = inTree v' rt
```

```
addTree v' EmptyBT                          = NodeBT v' EmptyBT EmptyBT
addTree v' (NodeBT v lf rt) | v'==v         = NodeBT v lf rt
                            | v' < v        = NodeBT v (addTree v' lf) rt
                            | otherwise = NodeBT v lf (addTree v' rt)
```

Aufgabe 3 [Gesamtpunktzahl: 8]

Gegeben seien die folgenden Definitionen der **Haskell-Funktionen** `map` und `foldr`:

```
map :: (a -> b) -> [a] -> [b]
map f []     =  []                      -- map.1
map f (x:xs) =  f x : map f xs          -- map.2

foldr:: (a -> b -> b) -> b -> [a] -> b
foldr g z []     = z                    -- foldr.1
foldr g z (x:xs) = g x (foldr g z xs)   -- foldr.2
```

Beweisen Sie durch strukturelle Induktion über die Listenlänge, dass **für beliebige Funktionen** `f` (mit passender Signatur) und **für alle endlichen Listen** `xs` gilt:

```
map f xs = foldr ((:).f ) [] xs
```

Erinnerung: Funktionskomposition ist definiert durch: `(f . g ) x = f (g x)`

Wichtiger Hinweis: Geben Sie bitte bei allen Umformungen in Beweisschritten jeweils die verwendeten Gleichungen oder sonstigen Begründungen an.

Aufgabe 4 [Gesamtpunktzahl: 12]

Zur Erinnerung: Wir haben den ADT Heap mit Hilfe sog. 'leftist trees' (dt. 'linkslastiger Bäume') implementiert (s.u.). Ein Baum heisst (rangbasiert) 'linkslastig', wenn in jedem Knoten der Rang des linken Kinds grösser oder gleich dem Rang des rechten Kinds ist. Der **Rang** eines Knotens ist die kleinste Anzahl an Knoten, die vom aktuellen Knoten aus – diesen eingeschlossen – durchlaufen werden müssen, um zu einem leeren Knoten zu kommen.
a) Fügen Sie, ausgehend von einem leeren Heap, nacheinander die Elemente der Liste `[5,9,2,7,4,6]` von links nach rechts jeweils mit `insHeap` in den im vorausgehenden Schritt entstandenen Heap ein. Geben sie in den Knoten jeweils den Rang an und machen Sie jeweils deutlich, wo es zur Erhaltung der Linkslastigkeit zur Vertauschung von Teilbäumen kommt.

b) Eine Funktion zum Erzeugen eines Heap aus den Elementen einer Liste `list` ist `fromList`:

```
fromList list = foldr insHeap emptyHeap list
```

Leiten Sie her, wie der Heap aussieht, der sich mit folgendem Funktionsaufruf ergibt:

```
fromList [5,9,2,7,4,6]
```

c) Gegeben sei ein Heap `h` mit Länge des rechten Rückgrats gleich `n`. Zur Erinnerung: Das rechte Rückgrat (engl. 'right spine') eines binären Baums ergibt sich, wenn man vom Wurzelknoten ausgehend solange jeweils rechte Kindknoten verfolgt, bis man auf einen leeren Baum stösst. Die Länge des rechten Rückgrats ist die Anzahl der (nicht-leeren) Knoten dieses Pfads.

c1) Welche unterschiedlichen Werte kann die Länge des neuen rechten Rückgrats nach *Hinzufügen eines einzelnen weiteren Elements* zum Heap annehmen? Begründen Sie Ihre Antwort.

c2) Geben Sie ein Beispiel für einen Heap und unterschiedliche Elemente an, so dass in einem Fall der kleinste und im anderen Fall der grösste mögliche Wert für die Länge des neuen rechten Rückgrats nach Hinzufügen angenommen wird.

d) Schreiben Sie eine Haskell-Funktion `rs :: Ord a => Heap a -> [a]` (für right-spine), die zu einem Heap die Werte in den nichtleeren Knoten des rechten Rückgrats (vom Wurzelknoten ausgehend) als Liste zurückgibt.

Zur Erinnerung:

```
data (Ord a) => Heap a = EmptyHP
                       | HP a Int (Heap a) (Heap a)
    deriving Show

emptyHeap = EmptyHP

heapEmpty EmptyHP = True
heapEmpty _       = False

findHeap EmptyHP      = error "findHeap:empty heap"
findHeap (HP x _ a b) = x

insHeap x h = merge (HP x 1 EmptyHP EmptyHP) h

delHeap EmptyHP      = error "delHeap:empty heap"
delHeap (HP x _ a b) = merge a b

rank :: (Ord a) => Heap a -> Int
rank EmptyHP      = 0
rank (HP _ r _ _) = r

makeHP :: (Ord a) => a -> Heap a -> Heap a -> Heap a
makeHP x a b | rank a >= rank b = HP x (rank b + 1) a b
             | otherwise        = HP x (rank a + 1) b a

merge ::(Ord a) =>  Heap a -> Heap a -> Heap a
merge h EmptyHP = h
merge EmptyHP h = h
merge h1@(HP x _ a1 b1) h2@(HP y _ a2 b2)
      | x <= y    = makeHP x a1 (merge b1 h2)
      | otherwise = makeHP y a2 (merge h1 b2)
```