Praktische Informatik 3 WS 18/19

Christoph Lüth
Thomas Barkowsky
Andreas Kästner
Gerrit Marquardt
Tobias Haslop
Matz Habermann
Berthold Hoffmann

8. Übungsblatt

Ausgabe: 27.11.18
Abgabe: 18.12.18 12:00

Dieses ist ein Gruppenübungsblatt, und sollte in einer Gruppe mit drei Studierenden bearbeitet werden.

In diesem Übungsblatt geht es um Kodierung und gleich zwei Baum-Datentypen. Zur (De-)Kodierung bzw. Kompression von Texten ist die *Huffman-Kodierung* eine populäre Wahl. Dabei wird ein Huffman-Baum erstellt, der häufigen Buchstaben kürzere Bit-Codes zuweist. Zur Konstruktion des Huffman-Baums benutzen wir außerdem einen *Linksbaum* (Leftist-Heap) als PriorityQueue.

8.1 *Linksbaum* — *12 Punkte*

Ein Linksbaum ist ein Binärbaum mit einer starken Tendenz nach links. Ein Linksbaum besitzt zwei wesentliche Invarianten. Dazu brauchen wir zuerst etwas Notation: das *rechte Rückgrat* eines Baumes ist der Pfad über ausschließlich rechte Kinder bis zu einem Blatt, und der *Rang* eines Knotens ist die Länge des rechten Rückgrats.

Ein Linksbaum erfüllt die folgenden zwei Invarianten:

(1) Die *Ordnungseigenschaft*: Ein Element an einem Knoten darf nicht größer sein als die Elemente an einem Kinderknoten. Damit ist der kleinste Wert immer an der Wurzel.

(2) Die *Links*-Eigenschaft: Der Rang des rechten Knotens ist höchstens so groß wie der Rang des linken Knotens.

Abb. 1 zeigt ein Beispiel für einen Linksbaum. Das rechte Rückgrat der Wurzel sind die Knoten `'a'`,`'x'`, sein Rang ist damit 2. Der Rang der restlichen Knoten ist 1, der Blätter 0. Damit ist die Linkseigenschaft erfüllt. Die Knoten erfüllen die Ordnungseigenschaft: es gilt `'a' < 'b'`, `'a' < 'x'` und `'x' < 'y'`.

Um die Links-Eigenschaft einzuhalten, müssen wir bei der Modikation eines Linksbaums den Rang der jeweiligen Knoten kennen. Um den Rang nicht jedesmal neu berechnen zu müssen, fügen wir ein Feld für den vorberechneten Rang im Datentyp hinzu:

```
data LeftistHeap α = Leaf
                   | Fork
                       Int              -- Rang
                       α                -- Knotenmarkierung
                       (LeftistHeap α)  -- linkes Kind
                       (LeftistHeap α)  -- rechtes Kind
```

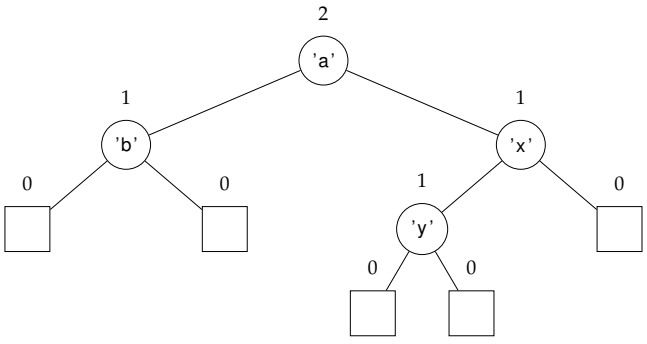


Abbildung 1: Beispiel für einen Linksbaum.

Um den Rang vorzuberechnen, implementieren wir zwei Hilfsfunktionen:

```
rank     :: Ord α ⇒ LeftistHeap α → Int
makeFork :: Ord α ⇒ α → LeftistHeap α → LeftistHeap α → LeftistHeap α
```

Die Funktion rank gibt den vorberechneten Rang eines Linksbaums zurück, wobei der Rang eines Blatts 0 ist.

Die Funktion makeFork wird anstelle des Konstruktors Fork benutzt. Es werden die Knotenmarkieruing und die beiden Kinderbäume übergeben (wobei wir annehmen, dass die Kinderbäume die Linksbaumeigenschaft erfüllen). Der Rang berechnet sich aus dem rechten Teilbaum. Als rechter Teilbaum wird derjenige der beiden Kinderbäume gewählt, so dass die Links-Eeigenschaft gilt.

Jetzt implementieren Sie die folgende Schnittstelle nach außen für den Linksbaum:

```
empty     :: Ord α ⇒ LeftistHeap α                                    -- O(1)
merge     :: Ord α ⇒ LeftistHeap α → LeftistHeap α → LeftistHeap α     -- O(log n)
insert    :: Ord α ⇒ α → LeftistHeap α → LeftistHeap α                 -- O(log n)
viewMin   :: Ord α ⇒ LeftistHeap α → Maybe α                           -- O(1)
extractMin :: Ord α ⇒ LeftistHeap α → Maybe (a, LeftistHeap α)         -- O(log n)
```

Offensichtlich wird Ord α benötigt, um die Ordnungseigenschaft einzuhalten.

Die Funktion merge vereinigt zwei Linksbäume. Dazu identifiziert sie den Linksbaum mit dem kleineren Wurzelelement als neue Wurzel. Der andere Linksbaum wird rekursiv mit der rechten Seite der neuen Wurzel vereinigt. Aufgrund der Links-Eigenschaft gilt für einen Linksbaum h mit n Elementen $\mathsf{rank}(h) \leq \lfloor \log(n+1) \rfloor$. Da wir immer rechts vereinigen ist der Aufwand von merge logarithmisch.

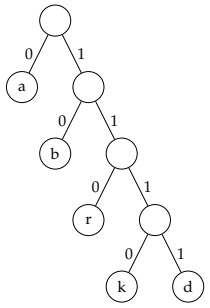
Bei insert wird ein neuer Knoten mit dem Baum vereinigt. Bei viewMin und extractMin wird das kleinste Element (am Wurzelknoten) zurückgegeben. Bei extractMin wird das kleinste Element entfernt, d.h. es wird ebenfalls ein Linksbaum ohne das vorherige Wurzelelement zurückgegeben. Dazu werden die beiden Kinder der Wurzel mit logarithmischen Aufwand vereinigt.

Die Vorlage enthält außerdem die zwei Funktionen toList und fromList, die ihr in den Tests und der Dokumentation benutzen könnt. Dazu könnt ihr euch z.B. an der Dokumentation von Data.Map[1] orientieren:

```
toList   :: Ord α ⇒ LeftistHeap α → [α]
fromList :: Ord α ⇒ [α] → LeftistHeap α
```

8.2 *Huffman Tree* — *8 Punkte*

In der deutschen Sprache kommen manche Buchstaben wie z.B. e oder n besonders häufig vor, während Buchstaben wie x oder q sehr viel seltener vorkomen. Huffman-Kodierungen dienen dazu, bei einer Kodierung von Texten (oder anderen Informationen mit ungleicher Häufigkeitsverteilung) zu einer Bitsequenz die Anzahl der benötigten Bits gering halten.



Kodierungstabelle:

Buchstabe	Codewort
a	0
b	10
r	110
k	1110
d	1111

Wir nehmen den sehr kurzen Text "abrakadabra" als Beispiel, um die Funktionsweise der Huffman-Kodierung nachzuvollziehen. Wir bemerken, dass der Buchstabe a besonders häufig vorkommt. Wir wollen nun jedem Buchstaben einem Codewort zuordnen. Ein Codewort ist eine Bitsequenz. Dabei wollen wir für besonders häufige Buchstaben kurze Codewörter wählen und für weniger häufige Buchstaben kürzere Codewörter. Ziel ist es, die durchschnittliche Codewort-Länge zu minimieren.

Der zugehörige Huffman-Baum (links) ist ein Binärbaum. Die Kanten von einem Knoten zu seinen linken bzw. rechten Kind sind jeweils mit 0 bzw. 1 markiert. An den Blättern befinden sich die Buchstaben. Ein Pfad von der Wurzel zu einem Buchstaben-Blatt ist dabei das Codewort des Buchstaben.

Aus dem Huffman-Baum ergibt sich die Kodiertabelle rechts daneben, mit der wir das Wort "abrakadabra" zu der Sequenz 0|10|110|0|1110|0|1111|0|10|110|0 aus 23 Bits kodieren.

[1] https://hackage.haskell.org/package/containers-0.6.0.1/docs/Data-Map-Lazy.html

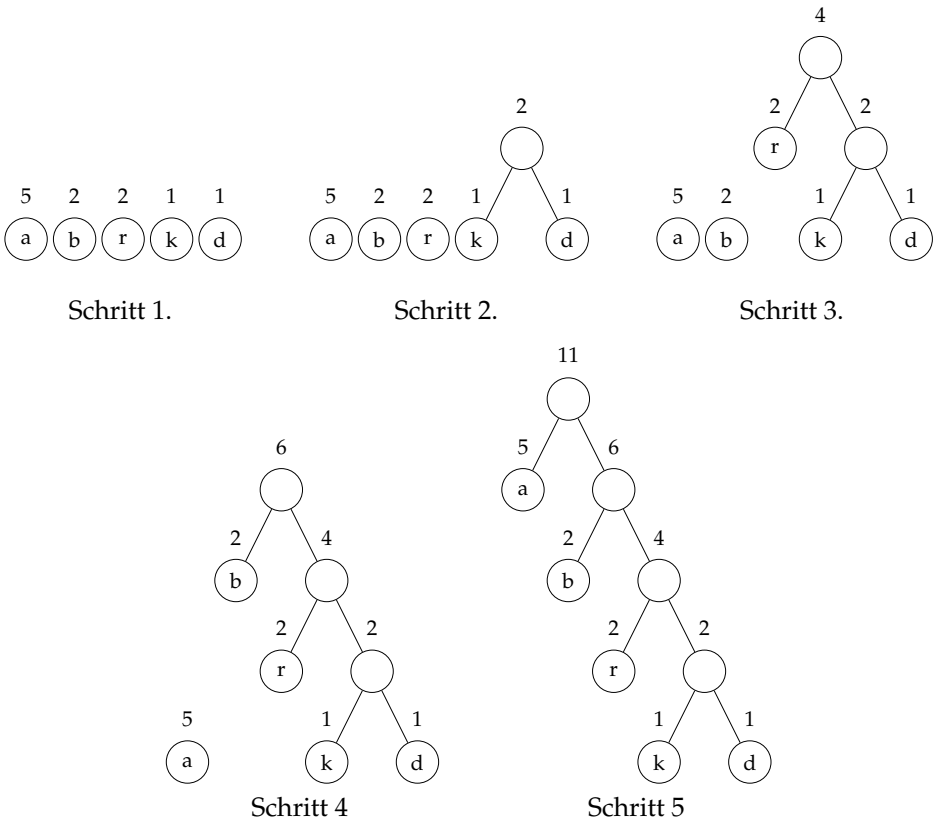


Abbildung 2: Konstruktion des Huffman-Baumes

Zur Dekodierung nutzen wir die Kantenmarkierung. Wir starten bei der Wurzel und gehen bei einer 0 zum linken Kind bzw. zum rechten Kind bei einer 1. Wenn wir das letzte Bit des Codeworts verarbeitet haben, sind wir einen Pfad von der Wurzel zu dem richtigen Buchstaben-Blatt gegangen.

Im Huffman-Baum werden gleiche Codewort-Präfixe mit demselben Knoten zusammengefasst. Damit ist der Huffman-Baum ein sog. *Präfix-Baum*, auch *Trie* genannt.

Außerdem ist bei Huffman kein Codewort ein Präfix eines anderen Codeworts. Codes mit dieser Eigenschaft werden Präfixcode bzw. präfixfreier Code genannt. Im Allgemeinen ist bei einer Codewortsequenz nicht klar, wann ein einzelnes Codewort vollständig dekodiert wurde und ein neues Codewort anfängt. Denn das bisher dekodierte Codewort könnte tatsächlich nur der Präfix des eigentlichen Codeworts sein. Um diese Unklarheit zu beseitigen, kann man z.B. ein zusätzliches Codewort für ein Trennsymbol einführen. Präfixfreie Codes haben den Vorteil, dass keine extra Bits für Trennsymbole o.Ä. benötigt werden.

Die Konstruktion des Huffman-Baums ist in Abb. 2 skizziert. Anfangs kennen wir nur die Buchstaben und ihre Häufigkeiten. Wir beginnen deshalb mit einem Wald aus Blättern, d.h. jeder Baum im Wald besteht aus einem Buchstaben-Blatt. Um die Häufigkeit zu berücksichtigen, werden die beiden Bäume mit den geringsten Wurzel-Häufigkeiten zu Geschwistern unter einer neuen Wurzel mit den addierten Häufigkeiten. Auf diese Weise werden wiederholt zwei Bäume im Wald zu einem Baum vereinigt bis nur noch ein Baum übrig bleibt. Dieser ist der Huffman-Baum.

Es kann passieren, dass mehr als zwei Bäume valide Kandidaten zur Vereinigung sind. Im Beispiel haben der zusammenfassende Wurzel von `k` und `d` sowie die Wurzeln `b` und `r` jeweils die geringste Häufigkeit 2. Deshalb ist auch der Baum aus Abb. 3 ein möglicher Huffman-Baum.

Jeder Huffman-Baum resultiert in einer Kodierung mit der minimal nötigen Anzahl von Bits (23 Bits im Fall von `"abrakadabra"`).

Die folgenden Funktionen sind zu implementieren:

(1) **type** Frequency = Int

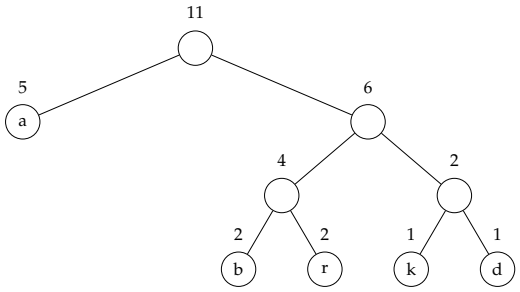


Abbildung 3: Alternativer Huffman-Baum

```
frequencies :: Ord a ⇒ [a] → [(a, Frequency)]
```

Die Funktion frequencies zählt die Häufigkeit von jedem Symbol in einer Liste von Symbolen. Alle Symbole im Ergebnis haben eine Häufigkeit > 0. Zur Implementierung empfehlen sich die Funktionen sort und group aus Data.List.[2]

(2) `frequenciesAscii :: String → [(Char, Frequency)]`

Alleradings arbeitet sort in $O(n \cdot \log n)$. Das geht besser, schließlich zählen wir doch nur die Häufigkeiten. Die Funktion frequenciesAscii bewältigt dieselbe Aufgabe wie frequencies für eine Sequenz von ASCII-Symbolen. Dafür soll ein Array mit einem Feld für jedes der 128 ASCII-Symbole erstellt werden. Die Felder werden mit 0 initialisiert. Für jedes Vorkommen eines Symbols erhöhen wir den Feldeintrag um 1. Schließlich wandeln wir das Array wieder in eine Liste von Häufigkeitseinträgen um. Der Aufwand bleibt dabei linear. Zur Implementierung empfiehlt sich die Funktion accumArray aus Data.Array.[3]

(3) `huffmanTree :: Ord a ⇒ [(a, Frequency)] → Maybe (HuffmanTree a)`

Die Funktion huffmanTree konstruiert aus einer Liste von Häufigkeiten einen Huffman-Baum. Für einen schnellen Zugriff auf die beiden Bäume mit der geringsten Wurzel-Häufigkeit sollt ihr alle Bäume im Wald in einer Min-Priority-Queue verwalten.

Ein Linksbaum kann als Min-Priority-Queue benutzt werden, indem die Elemente mit einer Priorität versehen werden:

```
type PriorityQueue p a = LeftistHeap (Entry p a)

data Entry p a = Entry { payload :: a, priority :: p }

instance Ord p ⇒ Ord (Entry p a) where
    compare = comparing priority
```

In der Codevorlage sind die Funktionen encode und decode bereits implementiert. Für decode benötigt man den Huffman-Baum. Für encode braucht man ein Codewörterbuch. Die ebenfalls schon implementierte Funktion codewords erstellt ein Codewörterbuch aus einem Huffman-Baum.

```
newtype Bit = Bit Bool
type Codebook a = [(a, [Bit])]

codewords :: HuffmanTree a → Codebook a
encode    :: Eq a ⇒ [a] → Codebook a → Maybe [Bit]
decode    :: [Bit] → HuffmanTree a → Maybe [a]
```

Änderungen:

- *Version 1.0 Ausgegebene Version*
- *Version 1.1 Linkseigenschaft korrigert.*

[2] https://hackage.haskell.org/package/base-4.12.0.0/docs/Data-List.html
[3] https://hackage.haskell.org/package/array-0.5.2.0/docs/Data-Array.html#v:accumArray