Assignment 6: Huffman Coding

Start Assignment

Due Mar 20 by 11:59pm **Points** 100 **Submitting** a file upload

The goal of this assignment is to create a program that can compress and decompress plain text data using Huffman coding. For this assignment, you have to implement two modules and a test program.

Module BTree

This module defines a new data type for the binary tree data structure. This module should allow a user to create a binary tree of anything type. The new data type should be named `BTree a` where `a` is a type variable and contains two value constructors, `EmptyBTree` (for the empty binary tree) and `BTree` (for a non-empty binary tree). The following functions must be a part of this module:

- `getRootData :: BTree a -> a`
- `getHeight :: BTree a -> Int`
- `getNumberOfNodes :: BTree a -> Int`
- `showBTree :: Show a => BTree a -> String`

Only export the data type `BTree` and its value constructors, and the above four functions. You are allowed you create helper functions but they should not be visible to users. For the `BTree` data type, you can simply make it an instance of `Show`, `Eq`, and `Read` type classes using the `deriving` keyword. For the `showBTree` function, it must return a string representation of a binary tree which can be pretty printed on the console screen. For example, imagine the following binary tree of characters:

```
aTree = BTree 'a' (BTree 'b' (Btree 'c' EmptyBTree EmptyBTree) (BTree 'd' (BTree 'e' EmptyBTree EmptyBTree) EmptyBTree)) (BTree 'f' EmptyBTree (Btree 'g' EmptyBTree EmptyBTree))
```

The expression `showBTree aTree` should return the following string:

```
"'a'\n+--'f'\n|  +--'g'\n|  +\n+--'b'\n   +--'d'\n   |  +\n   |  +--'e'\n    +--'c'"
```

The expression `putStrLn (showBTree aTree)` should give us the following output:

```
'a'
+--'f'
|  +--'g'
|  +
```

```
+--'b'
   +--'d'
   |  +
   |  +--'e'
   +--'c'
```

It shows the structure of the binary tree by showing the data in the root node, structure of the root **right** sub-tree, and structure of the root **left** sub-tree in a pretty-printing style. Note that if left and right sub-trees of a node are empty trees (leaf node), only print the data in the root node. Leaf nodes in the above example are nodes containing characters `'g'`, `'e'`, and `'c'`. If only one of the sub-tree of a node is empty, we need to print the + symbol to indicate which sub-tree is empty. In the above example, nodes containing `'f'` and `'d'` only contain one sub-tree.

Module HuffmanCoding

This module defines a new data type called `HBit` which only consists of two value constructors, `L` (for left) and `R` (for right). Note that the `HBit` data type is just an enumerated type. For simplicity, you are allowed to use automatic deriving for the `Show`, `Eq`, and `Read` type classes. With the `HBit` data type, a Huffman coding is simply a list of `HBit` (`[HBit]`).

Recall that a Huffman tree is simply a binary tree. In theory, a non-leaf node of a Huffman tree will not contain a data. Each leaf node will contain a unique character. However, from an implementation point-of-view, each node will contain a character and a frequency of the character. In other word, in this assignment, a leaf node will contain a data `(c, f)` where `c` is a character and `f` if the frequency of the character `c` in a given message to be compressed. A non-leaf node should contain a data `('\NUL', f)` where `f` is the summation of the frequencies of its children. Note that `'\NUL'` in Haskell is the null character. These pair `(c, f)` will allow you to easily construct a Huffman tree. For this assignment, a Huffman tree will have type `BTree (Char, Int)`.

To construct a Huffman tree for a message, first we need to start with the frequency of each character in the message, sort them, and turn them into a list of binary trees. Each binary tree will only contain one node containing a character and the frequency of the character. Then perform the following repeatedly until there is only one binary tree on the list:

1. Take the least frequent trees, make them the left child and the right child of a new root node that contains the character `'\NUL'` and the frequency equal to the sum of frequencies of its children.
2. Remove those two least frequent trees from the list
3. Insert the new tree into the list in the position where the list is still a sorted list of binary trees.
4. Go back to step 1.

At each iteration, we take two trees out and put one back in. So, in the end, you should end up with a single tree on the list. That will be our Huffman tree.

The `HuffmanCoding` will only export these two functions:

- `toHuffmanCode :: [Char] -> ([Char], [Char])`
  The `toHuffmanCode` function take a message and returns a string representation of the Huffman tree of the given message and the string representation of the Huffman code of the given message. For example, the expression `toHuffmanCode "Dog"` should return the following pair of strings:
  `("BTree ('\\NUL',3) (BTree ('D',1) EmptyBTree EmptyBTree) (BTree ('\\NUL',2) (BTree ('o',1) EmptyBTree EmptyBTree) (BTree ('g',1) EmptyBTree EmptyBTree))","[L,R,L,R,R]")`
  Note that your program may not generate the same tree as shown above because we may deal with identical frequency in a different way. However, the length of the Huffman code should be the same.
- `fromHuffmanCode :: [Char] -> [Char] -> [Char]`
  The fromHuffmanCode function takes a string representation of a Huffman tree and a string representation of a Huffman code, and returns the uncompressed message. For example, the expression
  `fromHuffmanCode "BTree ('\\NUL',3) (BTree ('D',1) EmptyBTree EmptyBTree) (BTree ('\\NUL',2) (BTree ('o',1) EmptyBTree EmptyBTree) (BTree ('g',1) EmptyBTree EmptyBTree))" "[L,R,L,R,R]"`
  should return the string `"Dog"`.

Again, this `HuffmanCoding` module should only export functions `toHuffmanCode` and `fromHuffmanCode`.

A tester program named `HuffmanCodingTester.hs` is given. Compile it to an executable file and run it. This program use the `BTree` and `HuffmanCoding` modules. If you implement those modules correctly, you should see the structure of a pretty big tree and you should be able to read a message on the console screen.