CSCC24 2020 Winter – Assignment 1
Due: February 10 Monday midnight
This assignment is worth 10% of the course grade.

In this assignment, you will work in Haskell with an algebraic data type and given abstract data types, and you will write interesting recursive functions over the algebraic data type.

As usual, you should also aim for reasonably efficient algorithms and reasonably organized, comprehensible code.

Huffman Tree, Huffman Code

Huffman code compresses text by using shorter bit strings to represent more frequently occurring characters. To ensure unambiguity during decompression, it also ensures that the bit string for a character cannot be a prefix of the bit string for a different character; on the downside, infrequent characters may need fairly long bit strings.

For example, suppose there are only 4 possible characters, and their frequencies (numbers of occurrences) are:

a	50
b	84
c	10
d	6
total	150

Without Huffman code, we would use 2 bits per character, totalling to 300 bits. With Huffman code, we use these bit strings:

a	01
b	1
c	001
d	000

totalling to $50 \times 2 + 84 \times 1 + 10 \times 3 + 6 \times 3 = 232$ bits.

Here is how to arrive at the bit strings above: Build a Huffman tree.

A Huffman tree is a binary tree with the characters in the leaf nodes. (For debugging convenience, leaf nodes also store frequencies, and internal nodes also store frequency sums.) We represent this by

```
data HuffmanTree = Leaf Int Char | Branch Int HuffmanTree HuffmanTree
```

The weighted-average path length (using character frequencies for weights) should also be minimized, and this can be achieved by the construction algorithm below:

1. We will use a min-priority queue, in which jobs are intermediate Huffman trees, priorities are frequencies.

2. Start with a queue consisting of leaf nodes, one per character.

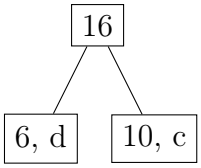
3. extract-min twice so you get two trees, make them children of a new parent node, put this new tree back into the queue. The new priority, and the new frequency stored in the new parent node, are the sum of the two children frequencies. (It should not matter who becomes the left child and who the right child, but to simplify autotesting, let's all agree to put the lower-frequency child on the left.)

   Example: If you get these two trees:

   ```
   Leaf 6 'd'
   Leaf 10 'c'
   ```

   then the new tree is

   ```
   Branch 16 (Leaf 6 'd') (Leaf 10 'c')
   ```



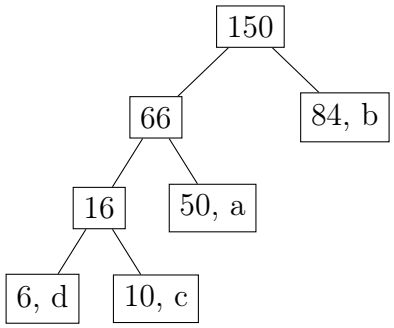
   and its new priority is also 16 when inserting into the queue.

   Repeat this process, until...

4. Until you can only extract-min once, not twice. That gives you the final Huffman tree!

Example: This should be the final Huffman tree for the example frequencies:

```
Branch 150 (Branch 66 (Branch 16 (Leaf 6 'd')
                                 (Leaf 10 'c'))
                      (Leaf 50 'a'))
           (Leaf 84 'b')
```



From the Huffman tree, we obtain the bit strings by noting which path from root leads to which character, using 0 when going left, 1 when going right. For example, 'a' is reached from root by going left (0) then right (1), so its bit string is 01.

For simplicity in this assignment, we use `[Bool]` for bit strings, with `False` for 0, `True` for 1. (This is very impractical of course.)

The Problems

1. [30%] Implement decoding given a Huffman tree and a boolean list. Assume that the boolean list is valid.

   ```
   decode :: HuffmanTree -> [Bool] -> [Char]
   ```

   The test cases used during marking do not necessarily conform to the promised character frequencies.

2. [30%] Implement construction of a Huffman tree given a list of characters and their frequencies. Assume that the list is non-empty, the frequencies are positive, the characters are distinct, and they are the only characters you need to handle.

   ```
   huffmanTree :: [(Char, Int)] -> HuffmanTree
   ```

   For simpler autotesting: When merging two trees, put the less frequent one on the left. On the other hand, there is no need to worry about two trees having the same frequency.

   You will need a priority queue. An implementation is included (module LeftistHeap).

3. [40%] To help with encoding, convert a Huffman tree to a lookup dictionary (the `Map` type—see the documentation of Data.Map.Strict) that maps a character to its Huffman code (as a boolean list).

   ```
   buildDict :: HuffmanTree -> Map Char [Bool]
   ```

   Then encoding is just looking up individual characters and concatenating boolean lists, as in the provided `encode` function.

End of questions.