CS119 – Module 10: Trees

Purpose: Not all data can or should be represented linearly in a list. Hierarchical data abounds and is represented in the form of a tree. Examples from real life are family trees, the table of contents of a book, computer files system folders and subfolders, etc. We will look at a particular example involving data compression where a tree representation is useful.

Knowledge: This module will help you become familiar with the following content knowledge:

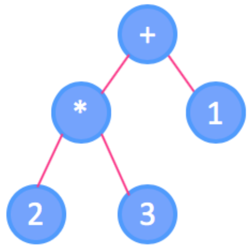
- Manipulation of trees
- Uses of trees

Activity: With your group perform the following tasks and answer the questions. You will be reporting your answers back to the class in 40 minutes.

1. Datatypes in Haskell can be recursive and this creates hierarchical data structures. Consider the data type:

   ```
   data ExprTree = Leaf Float | Add ExprTree ExprTree | Sub ExprTree ExprTree |
                   Mul ExprTree ExprTree | Div ExprTree ExprTree deriving Show
   ```

   The expression `Add (Mul (Leaf 2) (Leaf 3)) (Leaf 1)`
   represents 2*3 + 1 and can be viewed graphically as the tree:

   Write expressions using the ExprTree data type to represent the following:

   1 + 3 * 5 ______________________________

   (1 + 3) * 5 ______________________________

2. This shows that the tree can represent the precedence of the operators, or in other words it describes which operator gets evaluated first.
   Draw a tree diagram that illustrates each of the expressions above.

3. Examine the function:

```
evaluate :: ExprTree -> Float
evaluate (Leaf x) = x
evaluate (Add e1 e2) = evaluate e1 + evaluate e2
evaluate (Sub e1 e2) = evaluate e1 - evaluate e2
evaluate (Mul e1 e2) = evaluate e1 * evaluate e2
evaluate (Div e1 e2) = evaluate e1 / evaluate e2
```

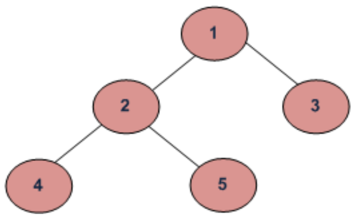
   For each of the nodes in the tree, Add, Sub, Mul, and Div, we recursively call evaluate on the two "children" trees e1 and e2.

   Write up the substitution models for `evaluate` on the two expressions that you wrote up above. Do you see the "tree recursion"? Tree recursion just means that we do the recursion for each of the children and combine the two results in some way.

4. Consider the data type:

```
data Tree a = Empty | Node a (Tree a) (Tree a) deriving Show
```

   Write an expression using this data type for the following tree:

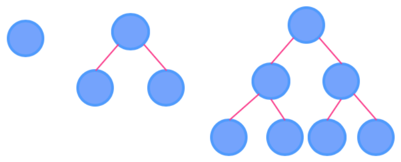


5. Let's define a complete tree as one in which every node either is a leaf or that node has two children. That would mean that the tree above is complete but if you removed any one of the three leaves it would not be.

   Write a function which returns true if a tree is complete and false otherwise:

```
complete :: Tree a -> Bool
complete Empty = True
complete (Node x Empty Empty) = ____________________
complete (Node x t1 Empty) = ____________________
complete (Node x Empty t2) = ____________________
complete (Node x t1 t2) = ________________________________________
```

6. If we take a look at complete trees of height 0, 1, and 2 below it looks like the number of leaves are powers of 2. We can prove by induction that the number of leaves of a complete tree of height h is 2^h.



What would be the base case? Verify that the number of leaves is correct for the base case.

Assume the number of leaves is 2^k for a tree of height $k < h$. How would you finish this induction proof?

Complete the following assignments to be submitted for grading. Each should be done individually but you can consult with a classmate to discuss your strategies or if you get an error message that you do not understand.

Write all of your functions in the file `Huffman.hs` contained in the `lab10` directory.

An illustration of the use of binary trees is in the problem of data compression. Ordinarily, each character is represented by an 8-bit code. We can reduce the total number of bits required to code a text by replacing this fixed-length code with a coding scheme based on the frequency of occurrence of characters in the text. Characters that appear most frequently should have short codes whereas characters that appear infrequently can have longer codes. For example in the word "text" we could encode $t \to 0$, $e \to 10$, and $x \to 11$. Then the encoding of "text" would be 010110.

We have to be careful, however, that we choose the code so that it can be uniquely decoded. If we had chosen $t \to 0$, $e \to 10$, and $x \to 1$, then both "text" and "tee" would have the same code! Not good. To prevent this from happening we must choose the codes so that no code is a proper prefix of any other.

To construct an optimal code satisfying the prefix property, we will use a technique called Huffman coding (named after David Huffman). Each character is stored as the leaf in a binary tree in such a way that more frequently used characters are of lesser depth in the tree than less frequently used ones. The code of a character is a sequence of 0's and 1's describing the path in the tree to the character, where a 0 represents a left branch and a 1 a right branch. Consider the following tree structure and tree:

```
data HTree = Leaf Char | Branch HTree HTree deriving Show

Branch (Branch (Leaf 'x') (Leaf 'e')) (Leaf 't')
```

In this tree, the character 'x' is coded by 00, 'e' by 01, and 't' by 1.

To build a Huffman tree we start with a list of characters along with their frequencies. For example:

```
[('g',8),('r',9),('a',11),('t',13),('e',17)]
```

We convert this list of pairs into a list of trees and then repeatedly combine the trees with the lightest weights until just one tree remains. The weight of a single leaf will be the weight of the character at that leaf. The weight of a binary node is the sum of the weights of its two subtrees. We will need another tree data type for this weighted tree:

```
data WeightedTree = Tip Int Char | Node Int WeightedTree WeightedTree
                    deriving Show
```

After the weighted tree is constructed we can simply remove the weights and get our HTree.

The code for construction of the weighted tree is given to you. Look through the code and trace through it with the given frequencies. Test it out and see if you get the weighted tree that you expect with:

```
> makeWeightedTree frequencies
```

Assignment 1:
We will make our HTree as follows:

```
        makeHTree :: [(Char,Int)] -> HTree
        makeHTree x = unweight (makeWeightedTree x)
```

Write the function `unweight` which takes a weighted tree and converts it to an HTree by stripping off the weights.

Hint: You will need to use pattern matching with the two cases for the WeightedTree.

Criteria for Success: Test it out by creating the `huffTree` which uses the weights in the example above. Draw out the tree from the expression that is printed and verify that it is a tree that you would expect to get.

Assignment 2:
Now that we have our HTree we can decode a message by traversing the tree making left branches for 0's and right branches for 1's until you get a leaf. The given character is produced and if you have more bits, repeat the process again starting at the root for the next character.

Write the function `decode :: HTree -> [Bit] -> [Char]`.

Hint: Start by writing a helper function that has an extra parameter which is an HTree whose root is the current position while traversing the tree. You will want to consider the cases where a) the current position is a Leaf, b) the bits are an empty list, and c) the current position is a Branch.

Criteria for Success Test out your function with the HTree from Assignment 1 and the bit string [1,1,0,1,1,1,1,0,0,0,0,1]. You should be able to figure out whether the text produced is correct or not.

Assignment 3:
Encoding is not as direct since the tree is good for finding a character associated with a bit string but poor at finding the bit string associated with a given character. So we will write a function `transform` which will transform the tree into a table where we can look up the code for a given character. This table and transform function will be

```
type CodeTable = [(Char,[Bit])]

transform :: HTree -> CodeTable
transform (Leaf x) = [(x,[])]
transform (Branch t1 t2) = hufmerge (transform t1) (transform t2)
```

Write the function `hufmerge :: CodeTable -> CodeTable -> CodeTable` which takes two code tables and merges them, adding a zero bit to the front of all the codes coming from the first table, and a one bit to the front of all the codes coming from the second table. Test it out by doing a transform of your HTree.

Hint: Consider the `map` function to add a bit to the front of all the codes in the list.

Criteria for Success: Use the `transform` function on the tree that you created previously. Take a look at the table that it is produced and verify that each letter has the correct code.

Assignment 4:
Write the function `codeLookup :: Char -> CodeTable -> [Bit]` which looks up the bit string for a given character in the CodeTable.

Criteria for Success: Perform `codeLookup 'e' (transform huffTree)` and verify that you get the correct bit string for that letter.

Assignment 5:
Write the function `encode :: HTree -> [Char] -> [Bit]` which uses a CodeTable to encode a string into a bits.

Criteria for Success: Encode the string "great" and verify that you get the bit string in Assignment 2.

Submit your `Huffman.hs` file in Canvas for grading.