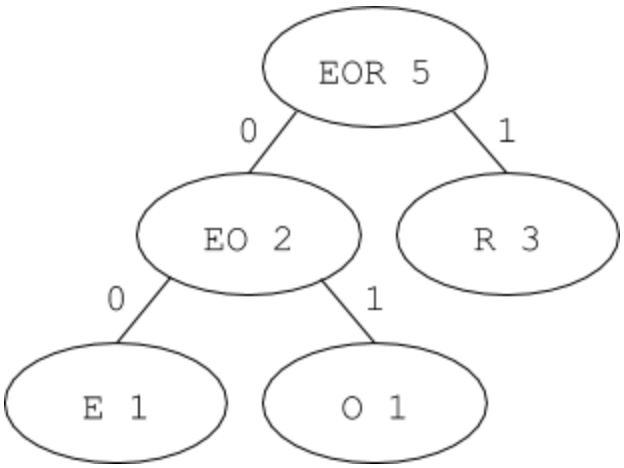
Huffman Encoder Part 1: Building a Huffman Tree

In this two part assignment, we will implement an efficient scheme for compressing a text message called Huffman coding. A simple method to encode text as a string of 0s and 1s is to represent each character as a unique string of 8 binary digits (bits). A text message is thus translated into a string of bits. By exploiting the fact that not all characters appear with the same frequency in the text, we can encode rarely used characters with long codes and frequently used ones with short codes.

In part 1, we'll use a given a set of characters and their corresponding frequencies to create an optimal coding scheme: a special binary tree where the path to each leaf represents a different character. In part 2, we'll take a text file and, using a Huffman Tree, encode/decode its message.

Huffman Tree

A Huffman Tree stores elements based on their frequency such that the higher frequency characters have shorter paths. For example, let's take the word "ERROR". The character/frequency legend is `E 1 R 3 O 1` and here's a corresponding Huffman Tree:



The leaf nodes are the characters with their frequencies, and the internal nodes are combinations of its children's characters with the sum of their frequencies. You should also notice the labels on the paths from node to node (each left child is a 0 and each right child is a 1); these are the basis for encoding.

If you follow the path from the root to each leaf, concatenating the edge labels, you get a unique string for each character. `E` is encoded as `00`, `O` is `01`, and `R` is simply `1`. The more frequent character `R` has a shorter string and so we can encode the word "ERROR" with only 7 bits (the encoding is `0011011`). This is far fewer than ASCII.

Implementation

We'll use a binary heap to generate a Huffman Tree from the character frequencies.

Set up a project/package with the following classes:

```
public class BinaryHeap // the heap class as posted
{
  // public int getSize()
}

public class HuffmanNode implements Comparable
{
   public String letter;
   public Double frequency;
   public HuffmanNode left, right;

   // public HuffmanNode(String letter, Double frequency)
   // public HuffmanNode(HuffmanNode left, HuffmanNode right)
   // public int compareTo(Object o)
   // public String toString()
}

public class HuffmanTree
{
   HuffmanNode root;

   // public HuffmanTree(HuffmanNode huff)
   // public void printLegend()
   // public static BinaryHeap legendToHeap(String legend)
   // public static HuffmanTree createFromHeap(BinaryHeap b)
   // public static void main(String[] args)
}
```

Here's information about the above methods:

BinaryHeap

`public int getSize()`. This method returns the number of elements in the heap.

HuffmanNode

`public HuffmanNode(String letter, Double frequency)`. This constructor creates a new HuffmanNode where letter is set to *this.letter*, frequency is set to *this.frequency*, and *left* and *right* are set to null.

`public HuffmanNode(HuffmanNode left, HuffmanNode right)`. This constructor creates a new HuffmanNode from its two children (i.e. the two nodes passed as parameters should become children of the new node), setting the *letter* variable to the concatenation of *left.letter* & *right.letter*, and the frequency variable to the sum of *left.frequency* & *right.frequency*.

`public int compareTo(Object o)`. This method casts Object *o* into a HuffmanNode object called *huff*. We then return `this.frequency.compareTo(huff.frequency)`. This allows us to make a heap of HuffmanNodes where the frequency determines which node is larger than which (This is the primary reason we make *frequency* of type Double rather than double).

`public String toString()`. It returns a string of form `"<"+letter+", "+frequency+">"`. There's no need to recursively iterate left/right pointers in this method. It may help you to better debug your assignment if you also add toString() methods in the BinaryHeap and HuffmanTree classes. I'll let you figure out on your own how these methods should behave.

HuffmanTree

`public HuffmanTree(HuffmanNode huff)`. This constructor sets *this.root* to *huff*. Before calling this constructor, we make a BinaryHeap of HuffmanNodes where each node has its *left* & *right* pointers set appropriately via the HuffmanNode constructor with node parameters above. Once the final HuffmanNode (containing all the others) is removed from the heap, we make that into a HuffmanTree object by calling this constructor.

`public void printLegend()`. This calls `printLegend(root, "")`, which calls private `void printLegend(HuffmanNode t, String s)`, a recursive method that works as follows: If `(t.letter.length() > 1)` i.e., *t* contains multiple characters, then *t* is NOT a leaf node, so we recursively call printLegend() on its left child using `string s + "0"`, and recurse on *t*'s right child using `string s + "1"`. If *t.letter* is a single character, then *t* is a leaf node, and we print out `(t.letter+"="+s);`

`public static BinaryHeap legendToHeap(String legend)` takes a String for the legend containing our input (letter & frequency data). The letters and frequencies are all one line with spaces as separators. (You may assume each separator is a single space).

`public static HuffmanTree createFromHeap(BinaryHeap b)`. We run the Huffman algorithm here. When we have only one element left in the heap, we remove it, and create a new HuffmanTree object with *root* set to the removed object.

`public static void main(String[] args)` calls `legendToHeap()` on the legend string and returns a BinaryHeap (*bheap*, for example). We then call `bheap.printHeap()` on the heap. Next, we call `createFromHeap(bheap)` on the heap to run our Huffman algorithm which returns a HuffmanTree, called, here, *htree*. Finally, we call `htree.printLegend()` on this HuffmanTree object to print the binary encodings for each of the letters in our input file.

The Algorithm

The input is a legend of characters and their corresponding frequencies. The output is a Huffman Tree, built using a Binary Heap.

1. Create a single HuffmanNode for each letter and its frequency, and insert each of these into a new BinaryHeap. (Hint: Call the BinaryHeap constructor that takes an array of Comparables. Don't forget that Binary Heaps ignore the zeroth entry!).
2. While the Binary Heap has more than one element:
    a. Remove the two nodes with minimum frequency.
    b. Create a new HuffmanNode with those minimum frequency nodes as children (using the the HuffmanNode constructor with left and right nodes as parameters) and insert that node into the BinaryHeap.
3. The BinaryHeap's only element will be the root of the Huffman Tree. Pass this node into the HuffmanTree constructor and return the result.

Legend

The test data for part 1 of this program is:

```
A 20 E 24 G 3 H 4 I 17 L 6 N 5 O 10 S 8 V 1 W 2
```

It's ok to hardcode this string. You can and should try out your program with other character/frequency legends.

Submission

You will submit your HuffmanNode, HuffmanTree, and adjusted BinaryHeap classes with the detailed methods. You should also submit the unchanged UnderflowException file.

Grading Rubric

Criteria

Programs that do not compile successfully will get no credit for the assignment

Proper implementations of the detailed classes and methods (4 points)

Project successfully creates a Huffman tree from a string legend (3 points)

Good style demonstrated in the code; sensible formatting (1 point)