CS 284: Makeup Assignment 6: Huffman Trees
Due: Tuesday 3 December, 11:55pm

1 Assignment Policies

Collaboration Policy. Homework will be done individually: each student must hand in their own answers. It is acceptable for students to collaborate in understanding the material but not in solving the problems or programming. Use of the Internet is allowed, but should not include searching for existing solutions.

Under absolutely no circumstances code can be exchanged between students. Excerpts of code presented in class can be used.

Assignments from previous offerings of the course must not be re-used. Violations will be penalized appropriately.

2 Assignment

This assignment consists in coding a series of methods required to complete the implementation of a Huffman Tree, and then using the Huffman Tree to encode and decode messages (text strings). You will be given a partial implementation of a class `HuffmanTree`. Each of the following sections explains the methods you have to implement.

2.1 Huffman Tree `toString`

Your first assignment is to implement the `toString` method of `HuffmanTree`. Your implementation should satisfy the following requirements:

- As seen in class, we will use a pre-order traversal.
- As seen in class, use new line characters after each node is printed.
- As seen in class, nodes should be *indented* according to their depth.
- **Leaf** node output should include the Character and its frequency. The class `LeafNode`, that models a leaf, is provided in the stub.
- **Internal** node output should include the subtree's frequency. The class `InternalNode`, that models an internal node, is provided in the stub.

Here is an example output of a Huffman tree:

```
(freq=15)
    (freq=11)
         [value=a,freq=9]
         [value=x,freq=2]
    [value=e,freq=4]
```

As depicted above, the frequency of an internal node is the sum of the frequencies of its children. Also, leaf nodes have, in addition to the frequency, a character.

2.2 Huffman Tree `bitsToString`

For this assignment, we will represent bit strings as arrays of booleans. Therefore, you will need to implement the following method:

```
public String bitsToString(Boolean[] coding) { ... }
```

For example, given an input array of `[true,false,true]` this method should return `"101"`.

2.3 Huffman Tree `decode`

The next part of the assignment is to implement an algorithm that uses the Huffman tree to decode a given sequence of bits. To this end, you will implement the following method:

```
public String decode(Boolean[] coding) { ... }
```

The input is an array of bits. Your implementation should decode the bits, collecting the output characters one-by-one and assembling them into an output String. If the input is not a valid encoding, you should throw an `IllegalArgumentException`.

2.4 Huffman Tree – Naïve Encoding

You will next implement the following method that encodes a string into an array of booleans:

```
public Boolean[] encode(String inputText) { ... }
```

This method takes an input text string (sequence of characters), looks up the bit sequence for each one, and returns an array that concatenates all of the bit sequences. If the input cannot be encoded, you should throw an `IllegalArgumentException`.

For example, if `t` is the Huffman Tree `new HuffmanTree("Some string you want to encode")`, then the statment

```
System.out.println(Arrays.toString(t.encode("string")));
```

will produce:

```
[false, true, true, false, false, false, false, false, true, true,
false, true, false, true, false, false, true, true, false, false,
true, true, true, false, true, true]
```

2.5 Huffman Tree – Efficient Encoding

In the previous encoding `encode`, you'll notice that you may have done a lot of redundant work. If the input string was "aa", you would search the tree twice looking for the same character each time.

For `efficientEncode`, your task is to make this more efficient. Your implementation should provide a way to reuse the results of previous lookups. For example, given the input string "aa", your implementation can traverse the tree at most once. This method has the same type as `encode`.

Hint: use hash tables (i.e. `HashMap`).

2.6 Testing

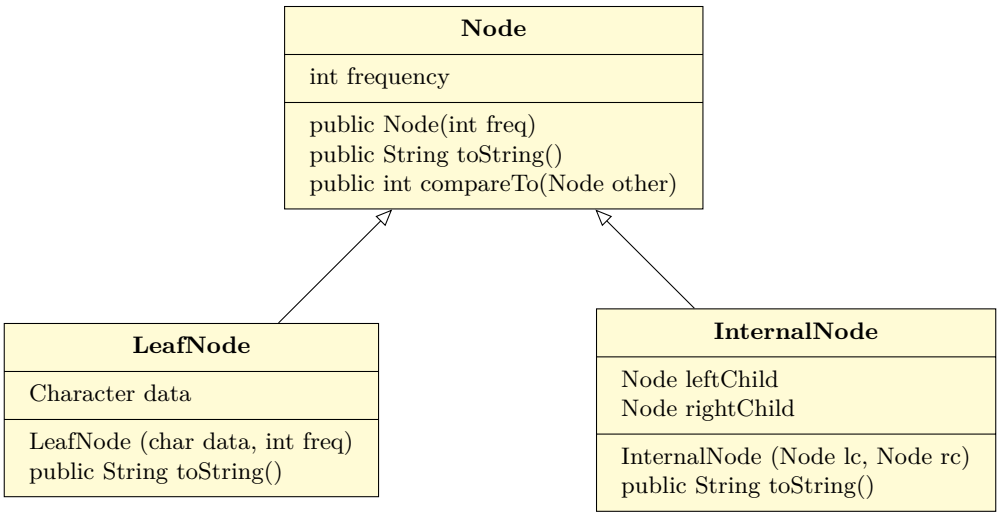
Test all of the methods you have implemented. You should test it on valid input as well as invalid input to ensure it correctly rejects invalid input.

3 Submission instructions

Submit a single file named `HuffmanTree.zip` through Canvas that includes `HuffmanTree.java` and `HuffmanTest.java` with your test cases. No report is required. Your grade will be determined as follows:

- You will get 0 if your code does not compile.
- The code must implement the following UML diagram precisely.
- We will try to feed erroneous and inconsistent inputs to all methods. All arguments should be checked.
- Partial credit may be given for style, comments and readability.

The private inner class `Node` should follow the UML diagram:



The class `HuffmanTree` should include the following operations:

HuffmanTree
private Node root
public HuffmanTree () public static int[] frequency(String s) private static Node buildHuffmanTree(String s) public String toString() public String decode(Boolean[] coding) public Boolean[] encode(String inputText) public Boolean[] efficientEncode(String inputText)