Data Structures: Homework on Huffman encodings, part 2

November 30, 2023

Last week you wrote code that takes a table of characters and their frequencies to build a Huffman encoding. The purpose of the Huffman encoding was to represent a chunk of text efficiently by encoding common characters using fewer bits. This week you will use the code you wrote last week to encode and decode text.

You will start by defining a new class called `HuffmanConverter` which will have a constructor taking as input a string which will be stored in a variable called `contents`.

Your first step is to calculate the frequencies of each character including punctuation and whitespaces. Each ASCII character corresponds to a number between 0 and 256 which you can get by casting a character `c` into an integer – `int i = (int)c`. You can then get the character back as `c = (char)i`. We will store the frequencies of the characters in an integer array, `count`, of size 256 such that `count[(int)c]=` the count of character `c`. `HuffmanConverter` will have a method `public void recordFrequencies()` that stores the counts of the characters in `contents` in an attribute `count`. **Print the table of frequencies you've created**.

Second you will build a Huffman tree from `count` using the code you've already written. Your code should build a heap using `count` and then call `HuffmanTree.createFromHeap`. The Huffman tree should be stored in an attribute `huffmanTree`. This should occur in a method `public void frequenciesToTree()`.

Third, you will extract a code from the tree. You will want to store the code in a string array attribute called `code` such that `code[(int)c]=` the Huffman encoding of character `c`. This will be done in a method `public void treeToCode()`. You will also need to write a private method `private void treeToCode(HuffmanNode t, String encoding)`; this private method recursively calls itself on the children of the Huffman node `t`, keeping track of its encoding `encoding`; once it reaches a leaf, it adds the character at that leaf and the encoding to `code`. In `treeToCode()`, first set every element of code to the empty string `""`, then call `treeToCode` at the root of the Huffman tree. **Print the code you have created**; this can also be done using a call to `huffmanTree.printLegend()`.

Fourth, once you've built `code`, you can encode `contents` into a string of bits in a method `public String encodeMessage()`. **Print the encoded message**. Also print the message size in the ASCII encoding (8 bits for each letter) and the Huffman encoding.

Finally, you will write a method `public String decodeMessage(String encodedStr)` to decode a given bit string using `huffmanTree`. To do so, you will take in one bit at a time to navigate through the `huffmanTree` (0 means go left, 1 means go right). Once you reach a leaf, you should store the character at that leaf and return to the root of the Huffman tree. Call `decodeMessage` on your encoded message and **print the decoded message**, which should be identical to your original message.

We will call the main method of `HuffmanConverter` from the command line, passing the path to a file of text. To recap, the output should be: the list of characters and their frequencies, the Huffman encodings, the encoded message, the number of bits needed to encode your message in a Huffman encoding vs using ASCII, and the decoding of your encoded message (which should be the same as the initial message).

You are provided a file with a template of `HuffmanConverter` with code to import a text file. You are also provided with two example input and output files. The inputs are two love poems taken totally randomly from http://www.lovepoemsandquotes.com. You should also use the files provided to you in homework 1.