Assignment 5b: Huffman Coding

Data Structures, CSCI-UA 102, Section 7
Instructor: Max Sklar

Encode and decode a document from a specified Huffman Tree.

Overview

In part A, you built a Huffman tree from frequency data on symbols. In this project, you will be using your Huffman Tree to compress files, which you will be able to encode and decode through a new class called HuffmanConverter. You will need to copy over and make changes to `HuffmanTree.java`, and you may create any other class that will help you complete the project. Included in full is `BitConverter.java` which converts a string of 1s and 0s into hexadecimal, base64, and binary (byte) formats and vice versa.

Huffman Converter

Your primary goal is to complete a new class called HuffmanConverter (`HuffmanConverter.java`). It takes two inputs, and also reads from stdin.

The first input is the mode, which is either, "encode", "decode", or "analyze". The second input is the tree specification file, which is created by your HuffmanTree class in "spec" mode. Piped into the HuffmanConverter should be the file that you want to encode (or decode).

As an example, suppose that we want to encode `just_to_say.txt` (included) and we want to use the tree specified in `just_to_say_spec.txt` which we can create using our original `HuffmanTree` in spec mode. In order to do the encoding the command woud be:

```
cat just_to_say.txt | java HuffmanTree encode spec > just_to_say_spec.txt
cat just_to_say.txt | java HuffmanConverter encode  just_to_say_spec.txt
```

(On windows machines, use type instead of cat.)

To store this encoding in `just_to_say_bits.txt`, you would use this command:

```
cat just_to_say.txt | java HuffmanConverter encode just_to_say_spec.txt
> just_to_say_bits.txt
```

The HuffmanConverter should also be able to decode the text, so if you use:

```
cat just_to_say_bits.txt | java HuffmanConverter decode  just_to_say_spec.txt
```

You should retrieve the original contents of `just_to_say.txt`.

The third mode, which is "analyze" will take your original file and encode it, but instead of printing out the encoding it will analyze how effective the message can be compressed by providing the following pieces of information:

- **Encoded Bits**: The total number of bits in the encoded file.
- **Original Character Count**: The total number of characters in the original file.
- **Average Bits Per Character**: The number of bits per character it took to encode the file. This is the first number (encoded bits) divided by the second (the character count) which will tell you how effective your huffman tree was at storing this file.

The interpretation of the mode argument is already done for you, as well as pulling in the file from standard input. You will just need to fill in the 3 modes (encode, decode, and analyze) in HuffmanConverter's main function.

Loading a Tree Specification

To load a tree specification file as a Huffman Tree, implement a static function in `HuffmanTree.java`:

```
public static HuffmanTree loadTree(String storedTree) { ... }
```

How to read a tree specification file

The tree specification file consists of symbols and pipes. Some symbols have a special escape sequence using the backslash (\).

- The escape sequences must eventually be converted into their original character. This could happen when it is put into the `HuffmanNode`, or when it is retrieved for encoding and decoding. To see the standard escaped encodings that should be handled, look at the function `asciiToSymbol` in `FrequencyCounter.java`. You'll need to create a reversed version of that. You will also have to ensure the correct behaviors for the sequence\\ which should be changed to \ and \| which should changed to |. These are non-standard escape sequences that we are using to ensure the tree specification works properly. Finally, \e does not represent a character but an end-of-message signal.
- | (unescaped) is a delimiter indicating the combination of two nodes.
- Any other character is its own symbol, representing a node.

As you read in each symbol, create a new HuffmanNode (frequency is irrelevant and can be set to 0). Then, push that node onto a stack, which starts out empty. The only exception to this is the unescaped pipe character |. The pipe symbol indicates that you need to pop the top 2 nodes from the stack, combine them into a new HuffmanNode, and push it back onto the stack! Be sure to remember that the first node popped out goes on the right side of the new `HuffmanNode`, and the second one goes on the left.

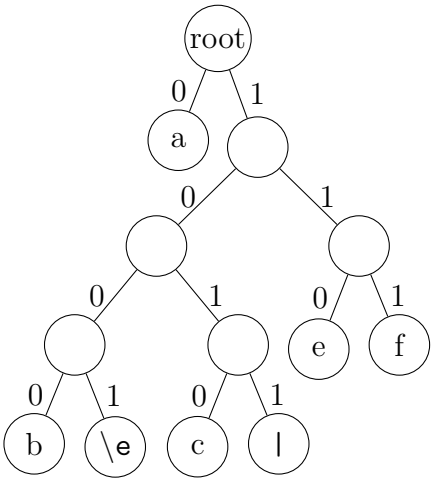
After processing all the characters, check if there is exactly 1 node on the stack. If there is more than 1 node, then apply the pipe operation (combining the top 2) until the stack only has a single node left. Pop the final node from the stack; this is the root of the reconstructed Huffman Tree.

Example: Tree Specification and Reconstruction

Here is an example of a tree specification file:

```
ab\e|c\|||ef
```

This specification corresponds to the following Huffman Tree:



You can test to ensure that the input above produces the correct tree by printing the legend after creating the tree.

Encoding

In part A, you printed out a legend from a HuffmanTree. In order to encode a document, you will need the legend returned as an array. There are 128 ascii characters and an end-of-message signal `\e` that could be in your spec. Build a method inside HuffmanTree that returns this array so that you can encode any character. You'll also need the encoding for `\e`. One possibility is to make an array of length 129 with the end-of-message code in spot 128.

Once you get the legend as an array, encoding is easy. Just read each character 1-by-1 and print the bits that you find in the array lookup. When you are done, print out the bits for EOM (end of message).

Remember to use System.out.print and not System.out.println! Your output should be on a single line.

Analyze

You can copy much of your encoding code, but instead of printing out bits, you should count them. When you are done, you'll have the number of encoded bits (don't forget `\e`!). The rest should be straightforward to calculate.

There are no automated tests for the analysis script, so you don't need to worry about the labels matching exactly. Instead, you'll use this to learn about the effectiveness of your Huffman encodings and report back in your readme file.

Decoding

In order to decode a document, turn the HuffmanTree class into an iterator by keeping track of a `current` node as a field in the `HuffmanTree` and initialize to `root`. Then, have a function called `advanceCurrent` which accepts a bit and returns a string. That bit will in most cases advance the `current` node right or left (depending on the bit) and return `null`. If the current node is a leaf, then the symbol for that leaf should be returned and current should be set back to the root.

Your decoder should go through all the bits, passing it into `advanceCurrent`. When `advanceCurrent` returns a symbol, that should be printed out except when the symbol is `\e`, in which case it should end the decoding process.

Testing with the BitConverter

Your HuffmanConverter will print out a stream of 0s and 1s as characters. BitConverter.java is designed to convert your bits into 3 possible formats: hexadecimal, base 64, and binary. If you were able to successfully encode `just_to_say.txt` into `just_to_say_bits.txt`, you can try these 3 commands (after compiling BitConverter.java) to convert them into different formats:

```
cat just_to_say_bits.txt | java BitConverter encode hex
cat just_to_say_bits.txt | java BitConverter encode base64
cat just_to_say_bits.txt | java BitConverter encode binary
```

These encodings do the following:

- hex: This is the hexadecimal system (the digits + a through f). Each hex character represents 4 bits.
- base64: This is a 6-bit encoding which uses all of the alphanumerics (lower case, upper case, and numbers) which amounts to 62 characters, plus the `+` and `\\` symbols to make 64. This is a more efficient compression method than hexadecimal, and is useful for sending certain documents over the web when the data is formatted in such a way (such as in JSON) where its helpful to ensure no illegal characters are used
- binary: This actually packs your bits into bytes (8-bits), and outputs a binary-formatted file. This is the most compressed format.

Note that these encodings require that the number of bits be divisible by a certain number. For example, the binary encoder requires that each byte contain 8 bits. If your bit list is not divisible by 8, then it will append zeros onto the end. Try the following:

```
cat just_to_say_bits.txt | java BitConverter encode binary |
  java BitConverter decode binary
```

Here, your original bit list will be returned, but a few extra 0s will be included. This is why it is important that your `HuffmanConverter` handles the special end-of-message character correctly!

In order to ensure that your `HuffmanConverter` works end-to-end, you can pipe your original file all the way through each stage: encoding to bits, encoding the bits further, decoding back to bits, and then decoding the message like so:

```
cat just_to_say.txt | java HuffmanConverter encode  just_to_say_spec.txt |
  java BitConverter encode binary | java BitConverter decode binary |
  java HuffmanConverter decode  just_to_say_spec.txt
```

This should just output your original file and match exactly!

Testing on the Corpus

A sample from the national archives has been provided in the folder `national-archives-sample`. The entire corpus has been used to create a frequency file using the following command:

```
cat national-archives-sample/*.txt | java FrequencyCounter > archives_freq.txt
```

Then manually `archives_freq.txt` to give `\e` a frequency of 11 instead of 1, because there are 11 text files in that collection. The next command created the tree specification:

```
cat archives_freq.txt| java HuffmanTree > archives_spec.txt
```

You can use your code to analyze specific documents using the following command to see how many

```
cat national-archives-sample/jay-treaty.txt |
  java HuffmanConverter analyze archives_spec.txt
```

Final Analysis

You may include some additional analysis in your readme file for bonus points. Here are some potential projects:

- Try encoding, decoding, and analyzing the archive, and report back what you have found.
- Try encoding your own java code, so see if the source code itself can be compressed using Huffman Codes. Report back what you have found.

Submission

The entire submission should be a zip file containing all relevant files and folders and should be uploaded to Brightspace by the due date. It should be zipped up in file with your netid as the name. This includes:

- Your Java code, which should include one or more Java files and may or may not be organized into packages.
- Your `readme.txt` file (described below).

README

Included in the `readme.txt` file should be the following:

NOTES: Include the difficulties and roadblocks encountered while working on this project, including notable bugs that needed to be overcome. Are there any quirks in your solution that we should be aware of? Was the outcome what you expected?

RESOURCES AND ACKNOWLEDGEMENTS: Disclose the sources you consulted for help on this project. Possible human sources include: NYU Staff (the instructor, TA, tutor), other students, and anyone who helped you with the project. Sources that you consulted, including code that you found online, in the book, and output from LLMs. Describe the nature of your LLM usage.

ANALYSIS: Include your findings and analysis when you applied your Huffman Converter to real files. What was possbile / not possible? How effective was the encoding?

There will never be a grading penalty for a complete and accurate accounting of resources and acknowledgments.