Huffman Coding

For this project you will write a Java program to encode and decode a character-based file using a canonical Huffman code.

Program Specification

In a package named `huffman`, create the public classes Encode and Decode. The usage will be as follows:

```
java huffman.Encode SOURCEFILE TARGETFILE
```

will create a Huffman encoding of the character-based ***`SOURCEFILE`*** and save the binary encoding in ***`TARGETFILE`***.

```
java huffman.Decode SOURCEFILE TARGETFILE
```

will decode the binary file ***`SOURCEFILE`*** and save character data in ***`TARGETFILE`***.

For example, the following two calls will encode and then decode the file sample.txt.

```
java huffman.Encode sample.txt sample.huf
java huffman.Decode sample.huf sample2.txt
```

Afterwards, the files `sample.txt` and `sample2.txt` will be identical.

```
bash% diff sample.txt sample2.txt
bash%
```

Note: If you are not familiar with `diff`, look it up! It will be very useful on this project.

Encoding Specification

Prefix Code

To encode the character file, we will use a Huffman prefix code. Recall, a prefix code is a set of binary codewords where no codeword is a prefix of another. Each character is replaced by its binary prefix codeword. If the codewords are created optimally, the overall number of bits to store the data will be reduced.

Huffman Tree

Huffman's algorithm builds a binary tree describing an optimal prefix code for the given input character data. The algorithm for building a Huffman tree is given in the book.

Canonical Huffman Tree

In order to decode the data later, the Huffman tree must also be saved with the file. In order to store the tree efficiently, we will place some restrictions on the structure of the tree. These restrictions won't affect the optimality of Huffman's algorithm.

These restrictions are based on the following observations:

- For a given set of character frequencies, there may be many optimal Huffman trees.
- In fact, there may be multiple optimal Huffman trees with the same codeword lengths
  - e.g. Four codewords of length 4, two codewords of length 3, two codewords of length 2

Our goal is to create a canonical Huffman tree. That is, given the codeword lengths for each character, there may be multiple Huffman trees (e.g. '`a`' could be 0010, 0001, 0000, etc), but there will only be one canonical Huffman tree.

The restrictions are as follows:

- Longer codes will be to the left of shorter codes
  - This restriction can also be viewed as: If we compare the numerical value of shorter codes with longer codes, by filling out the shorter codes by adding zeros on the right until the lengths match, the shorter codes will have higher numerical values
- For codewords of the same length, the numerical value of the codeword increases with the lexicographical order
  - based on character ordering (Unicode)
  - e.g. '`a`' = `0000` and '`b`' = `0001` is valid, but '`a`' = `0001` and '`b`' = `0000` is not.

In practice, the easiest way to create the canonical Huffman tree is to create an optimal Huffman tree, and then convert to the canonical version.

Encoding the Tree

Because we are now using a canonical Huffman tree, we can store the tree by simply remembering the codeword lengths for each character. We will store the tree as a header at the beginning of our binary file in the following format:

- First, 8 bits representing the number k of characters in the alphabet (unsigned)
- Then, k pairs of bytes representing
  - First Byte: the character value
  - Second Byte: the length of the codeword for that character

For example, for the canonical code: { `a` : `000`, `b` : `1`, `c` : `001`, `d` : `01` }, the header would be the following 9 bytes:

Decimal	4	97	3	98	1	99	3	100	2
Binary	0000 0010	0110 0001	0000 0011	0110 0010	0000 0001	0110 0011	0000 0011	0110 0100	0000 0010

End Of File

Binary files are organized into bytes. If the Huffman encoding uses a number of bits that is not a multiple of 8, then the data will not neatly fit into bytes, and we will have to add extra bits. For example, if the encoding uses 8001 bits, it will have to be written as 8008 bits, or 1001 bytes. This means we have to pad the end with 7 extra bits. This presents a problem when decoding, since those extra bits could be interpreted as characters, adding extra character data onto the end of the file.

To avoid this problem, add a pseudo-EOF character to the encoded data. We will use `0x00` (`'\u0'`) as our pseudo-EOF character. This character should be added to the tree-generation (with frequency 1) and encoded as the last character in the file. (Note, this means we can't actually encode character `'\u0'`, the null character, which is an assumption we make for this assignment)

When decoding, once the pseudo-EOF character is seen the decoding is finished.

Encoding Outline

The main steps in encoding are as follows:

1. Count character frequencies
2. Create Huffman tree
   - Follow the algorithm in the book
3. Canonize the Huffman tree
4. Efficiently store codes for lookup
5. Write canonical tree to output file
   - code lengths
6. Write coded data to output file

Decoding Outline

The main steps in decoding are as follows:

1. Read in and build Huffman tree
2. Efficiently store codes for lookup
3. Decode data and write character output

Note that these are just the main points. You will have to plan out the details and make sure your program is time & space efficient.

Comments

You must use full JavaDoc commenting throughout your program. Include descriptions of your classes and methods. Be sure to include your name as author of each file.

Extra Credit (10 points)

Graphviz is a graph visualization software package that performs layout and publishing of graphs and trees based on a DOT language file. For extra credit, add the optional command line parameters to save your Huffman tree and canonical Huffman tree as DOT files when encoding, and save the Huffman tree as a DOT file when decoding. See the following command line examples:

```
java huffman.Encode [-h HUFFMAN_TREE_FILE] SOURCEFILE TARGETFILE
java huffman.Encode [-c CANONICAL_TREE_FILE] SOURCEFILE TARGETFILE
java huffman.Decode [-c CANONICAL_TREE_FILE] SOURCEFILE TARGETFILE
```

Graphviz can be downloaded for free at http://www.graphviz.org. Find additional information on the DOT language here:

- http://en.wikipedia.org/wiki/DOT_language
- http://www.graphviz.org/content/dot-language

Also, find a sample DOT tree file under iLearn > Resources.

To receive extra credit for the graphviz component, you must include a readme.txt in your zipped submission explaining what functionality you implemented.

Submission

Save your Java files in a directory named `huffman` (for the package).

```
huffman/
    Encode.java
    Decode.java
    ... any other .java files
```

Zip this file using `zip` to the file `huffman.zip`. Use **only** zip (not gzip, rar, etc). Submit this file through iLearn. Do **not** submit any other files.

Grading

To receive credit, your program must, at a minimum, be able to encode or decode a simple file. Programs that do not compile or perform any encodings/decodings will receive no credit.

Deadlines

This project has two deadlines. You will submit the decode portion first, provide feedback before the full project is due.

1. Submit `huffman.zip` with decode functionality through iLearn by 11:00 pm on November 25.
2. Submit `huffman.zip` with the full project through iLearn by 11:00 pm on December 10.