Lab 6 Huffman Algorithm

Assignment Overview

Write a program that implements Huffman algorithm and test it by encoding and decoding a set of files.

See files in the assignment folder for the partial pseudocode that you may use to implement the program. You have to devise the complete algorithms yourself or find them elsewhere, for example, in the textbook.

The assignment folder also contains input files and correspondent expected output files. Note that different input files may be used for the grading.

Submit your program source code, input files, and output files into **Lab6** folder.

Byte Type

The pseudocode uses *Byte* type to store character values. For the actual program, select appropriate type that can hold values between 0 and 255 (or more).

In the case of Java, it can be **char**, **short**, **int**, or **long**. In the case of C++, it would be, probably, **unsigned char**, but can be **signed** or **unsigned short**, **int**, or **long**. In the case of Python, it would be, probably, **int**.

Some languages have **byte** type. You can use it, but it may contain values between -128 and 127. Similar to any signed type, you have to be careful when using it as an index in the frequency or encoding tables. You can work around by never using such values directly and by converting them to 0..255 range.

Encoding Bits Storage and Representation.

For the purpose of this assignment, it is acceptable to store encoding bits in a data type that is not actually a bit. For example, as a boolean, an integer, or even as a character. The similar way, a container of bits can be implemented as a simple wrapper around vector of booleans, vector of integers, vector of characters, or just as a string.

When doing input/output to files, represent the encoding bits as characters ‘0’ and ‘1’.

Unacceptable Language Tricks that Lead to Zero Grade

There is no place in this assignment for hashes, maps, associative arrays, or similar data types. All arrays (vectors) have to use integer indexes, so using strings as indexes is not acceptable.

Also, byte values, when needed, have to be used as indexes directly. Searching through arrays, strings, and other data structures is not acceptable.

There is no need for text processing. So, no regular expressions, text search, text splitting, text filtering, or similar tricks. No string manipulation, except that strings can be used while printing debug messages.

Note that these issues often happen when copying/pasting code from the Internet.

General Program Design

The program has to consist of at least 4 modules/classes

- A driver (main) that opens/closes all input and output files and calls appropriate **HuffmanCodec** class functions
- **HuffmanCodec** class that does input/output and calls **HuffmanAlgorithm** class functions.
- **HuffmanAlgorithm** class that does actual encoding and decoding
- **HuffmanTree** class that represents Huffman tree

See appropriate **-pseudocode.txt* files for a pseudocode of these classes. You may use the pseudocode to implement the classes in a language of your choice.

While following the pseudocode is not required, there are two things to consider

- The most important part is the separation of duties - the four modules/classes mentioned above have to be implemented separately
- The debug output shall resemble the provided examples

HuffmanAlgorithm class does not deal with input/output at all. It receives bytes, encode/decodes them, and returns encoded/decoded bytes and strings.

HuffmanCodec class is a wrapper around **HuffmanAlgorithm** class. **HuffmanCodec** class reads data from an input stream, encodes or decodes the data, and writes results to an output stream. To encode and decode the data, the class calls appropriate **HuffmanAlgorithm** methods to calculate *Byte* frequencies, to build a Huffman binary tree, to assign bit sequences to *Bytes*, and to actually encode and decode *Bytes*.

The driver opens and closes all input and output files. **HuffmanCodec** class uses input and output streams that are opened by the driver.

HuffmanAlgorithm Class

HuffmanAlgorithm class must not do any input/output except printing debugging messages. To have the expected debug output format, implement functions similar to **printFrequencyTable()**, **printEncodingTable()**, and **printQueue()** in the pseudocode.

Building Huffman Tree

To build the Huffman tree, you may use a priority queue of Huffman trees. In the beginning, for each *Byte* that has non0zero frequency, create one node tree and add it to the queue. Then, for each two least frequent trees, remove them from the queue, combine them in one new tree, and place the new tree back in the queue.

One tree is less than another tree if its root node frequency is less than another tree root node frequency. You will, probably, need to define an appropriate tree comparison lambda closure and use it to setup the priority queue.

If your language does not have a priority queue class, use a heap. The priority queue is essentially a wrapper around the heap.

Decoding

To decode incoming bytes, the decoding function has to travers the Huffman tree.

During decoding you have to take care of some edge cases

- When the Huffman tree root is null (an empty input file)
- When the Huffman tree consists of exactly one node (the input file consists of one repeating byte)

HuffmanTree Class

HuffmanTree class is a simple binary tree class that stores *Bytes* in each terminal node and frequencies in each node.

Traversing the tree from the root to a terminal node generates a sequence of bits matching the terminal node character. Going to the left child node generates 0. Going to the right child node generates 1.

The class has two constructors, first to create a tree consisting of one terminal node, second to create a tree consisting of two subtrees. If your language does not support constructor overloading, implement them as one constructor with appropriate parameters, or as factory methods.

HuffmanTree class must not do any input/output except printing debugging messages.

Utility Functions

There are few utility objects and functions that are used by the pseudocode. You may implement them as fields and methods of **Utilities** class or as global variables and functions, as appropriate for your language. See *Utilities-pseudocode.txt* for the pseudocode.

debugStream object represent some output stream that is opened and closed by the driver. Methods in other classes use **debugStream** to print debugging information.

bytePresentation() functions returns text representation of a given character. If the character is printable, the function quotes it (like ‘a’). Otherwise, the function returns either some common representation (like \n) or a hexadecimal representation (like 0xf7).

Driver (main)

Write main driver class/function that uses the mentioned above classes, tests encoding and decoding functionality, and prints test results, utilizing various input files. See *Driver-pseudocode.txt* file for the driver pseudocode.

The driver has to open and close all required input and output files.

There are eleven input files that are named *N*_**in.txt**, where *N* is the file number between 1 and 11. For each input file, the driver has to create three output files, specifically

- *N*_**encoded.txt** containing encoded data that is a result of applying Huffman encoding to the input data
- *N*_**decoded.txt** containing decoded data that is a result of applying Huffman decoding to the encoded data
- *N*_**debug.txt** containing debugging output

After the decoded file is written, the driver has to compare the input file and the decoded file, byte by byte. If the files are the same, an "*OK*" message has to be printed to stdout. Otherwise, an exception indicating the files difference has to be printed to stdout and the program has to be terminated.

Input and Output Files

The pseudocode reads and writes bytes one by one. If you think that is not efficient, you may read and write multiple bytes in one operation.

Do not read all input data at once into one string or array. Do not collect all output data before outputting it. Read and write data in pieces.

While copying the files from the assignment to your workspace, you have to use copy methods that preserve all bytes in the files. Double check the file sizes in the assignment and in your workspace. Note that if you are using copy/paste, some operating systems can replace whitespaces (spaces, tabs, new lines) and other control characters incorrectly.

Input and Output Data Files are Binary

The input file data is arbitrary and can contain any byte codes, including non-printable codes. All characters have to be read and processed until the end of file.

The decoded output data is supposed to be the same as the input data. So, it is also arbitrary.

If your language or operating system require it, open such files in binary mode. Opening files in text mode (that is often by default) will read or write corrupted data. Do not attempt to set a text encoding, for example, "UTF8" as binary I/O does not use text encoding options.

While doing input/output to such files, the program has to use appropriate methods to read and write raw bytes as is. Such methods are often called **read()** and **write()**. Search on the Internet for language specific rules regarding binary input/output.

The encoded output file will be a sequence of '0' and '1' characters. The debug output file is supposed to contain printable characters only. You can write to these files using regular language facilities.

Input Files

The input file names are *N*_**in.txt**. The *N*_**in.txt** files listed below shall be used as sample input files to test your program. Note that different input files may be used for the grading purposes.

When doing input from these files, the program has to read raw bytes. If your language or operating system require it, open the files in binary mode. You may also need to use non-default functions to read bytes.

Input filename	**Input file content**	**Input file size in bytes**
1_in.txt	Empty file, size is 0	0
2_in.txt	Contains character 'a' repeated 3 times. No new line or other control characters.	3
3_in.txt	Contains character 'a' repeated 6 times and 'b' repeated 2 times. No new line or other control characters.	8
4_in.txt	Contains characters 'a', 'b', \r, and \n. No other control characters.	12
5_in.txt	Contains character 'a', 'b', 'c', \r, and \n. No other control characters.	20
6_in.txt	Contains short text (including \r and \n)	59
7_in.txt	Contains few text statements (including \r and \n)	402
8_in.txt	Contains a document (including control characters besides \r and \n)	13,115
9_in.txt	Contains a PDF document (including various control characters)	241,514
10_in.txt	Contains some English text (including \r and \n)	1,100
11_in.txt	One text line terminated by \r\n	14

Output Files

Encoded Output File

The encoded output file name is *N*_**encoded.txt**. The bits have to printed as characters '0' and '1'. No line delimiters, like **\r** or **\n**, are allowed.

The *N*_**encoded.txt** file in the assignment folder contains encoded output expected after running the program using the *N*_**in.txt** file.

When writing to these files, do not flush after every single byte is written. It will seriously impact performance. The file will be flushed automatically when its closed.

Decoded Output File

The decoded output file name is *N*_**decoded.txt**. The *N*_**decoded.txt** file in the assignment folder contains decoded output expected after running the program using the *N*_**in.txt** file.

When doing output to these files, the program has to write raw bytes. If your language or operating system require it, open the files in binary mode. You may also need to use non-default functions to write bytes.

When writing to these files, do not flush after every single byte is written. It will seriously impact performance. The file will be flushed automatically when its closed.

Note that *N*_**in.txt and** *N*_**decoded.txt** files have to be the same for the assignment to be graded.

Debugging Output File

The debugging output file name is *N*_**debug.txt**. The *N*_**debug.txt** file in the assignment folder contains debugging output expected after running the program using the *N*_**in.txt** file.

Your debugging output may differ from the expected debugging output if you use an unstable priority queue or an unstable heap.

The program shall produce debugging output format that is similar to the format of provided *N*_**debug.txt** files**.**

When writing to these files, flush them regularly, when appropriate. As you are not flushing after every single byte, it will not impact performance visibly. The flushing is useful as the debugging messages may be left unprinted if the program unexpectedly terminates (crashes).

Console Output (stdout)

The console output (stdout) shall to be printed in the same format as the pseudocode prints it. Here is the expected console output for your program.

```
*** Testing file 1_in.txt, debug output file 1_debug.txt ***

OK

*** Testing file 2_in.txt, debug output file 2_debug.txt ***

OK

…

*** Testing file 11_in.txt, debug output file 11_debug.txt ***

OK
```