Huffman Coding Trees

Assignment:

You will write a pair of file compression programs based on Huffman coding trees. Typical encoding schemes like ASCII use fixed-length coding schemes with each character represented by a fixed number of bits. Instead, Huffman coding trees define variable length codes, where more frequently occurring characters are encoded with fewer bits than characters that occur less frequently. As a result, files encoded using Huffman coding trees often require significantly less space than those that use fixed-length coding schemes.

Invocation and I/O:

The programs are invoked as `huffencode infile outfile` and `huffdecode infile outfile`

`huffencode infile outfile` encodes the contents of $infile$ using a Huffman coding tree and stores the tree and the encoded file in $outfile$. A Huffman coding tree should be created based on the contents of $infile$. The file must first be opened, its contents read, and a frequency count generated. The frequency count must then be used to generate a Huffman tree in memory. The basic encoding unit is a single byte. The input file might be, but need not be, an ASCII file. Thus, it will be necessary to represent all characters, including spaces, tabs, and non-printing characters. For verification purposes, your program should print to standard output the code and frequency for each character in the Huffman coding tree in ascending order by code value.

Upon completion, the output file $outfile$ should contain a representation of the code list as well as the encoded input file. The encoded output should be stored in binary format to make use of the space gains from the encoding.

`huffdecode infile outfile` decodes the contents of $infile$ and stores the decoded file in $outfile$. The format of $infile$ will be the same as the format of $outfile$ for `huffencode`: it will consist of the Huffman code list and the encoded version of the message. This program will first reconstruct in main memory the Huffman tree based on the code list in the file, then will decode the encoded information using the tree.

Upon completion, the output file should contain the decoded information. Of course, running `huffdecode` on the output file from `huffencode` should result in the original file.

Design and Implementation Considerations:

Before implementing Huffman coding trees, it would be wise to read Section 5.6 of the textbook. You may use the code from the textbook, though there is no requirement to do so.

Your Huffman coding trees should support binary data input, with the basic unit a single byte of data. In writing to disk, your program must store the encoded data in binary format as well. That is, don't store each 0 or 1 as a separate character.

To ensure that your program can decode files encoded by other programs (including the test files), the following standard ordering will be used for the output files:

- the number of codes (in decimal)
- the code values and their lengths
- the codes themselves (in binary)
- the number of coded characters (in decimal)

- the encoded message (in binary)

Note that the codes and the encoded message may overlap byte boundaries. Only the final byte should be padded with extra zeros.

You should use a four-byte integer to store the number of coded characters. For any condition where this is inadequate, an error message should be presented. The number of coded characters will be byte aligned, not word aligned.

See the sample input files and explanations for more details.