Assignment #2

Due: October 27, 2023 11:59pm

Overview

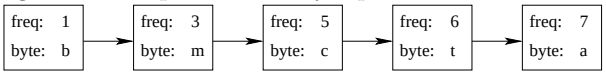
This assignment requires dynamically allocated data structures, file I/O, and bit manipulations. For this assigment you will implement two programs: a Huffman encoder and a Huffman decoder. The programs will be used to compress and decompress files. The algorithmic details are provided below. You should structure your code in such a way that each program can use the same data structures (as appropriate).

Algorithm

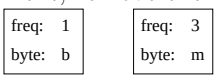
Encoding

To encode (compress) the contents of a file, first read the contents of the file and create a histogram – a count of the number of times that each byte appears in the file.

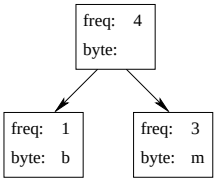
The next step is to build a binary tree based on the frequency of each byte (ignoring those that don't occur in the file). This can be done using a linked list ordered from least frequent to most frequent byte. First, build a list based on the histogram constructed previously. Assume the frequencies given in the following diagram (note, though the example shows only alphabetical characters, the stored byte can be any sequence of bits).



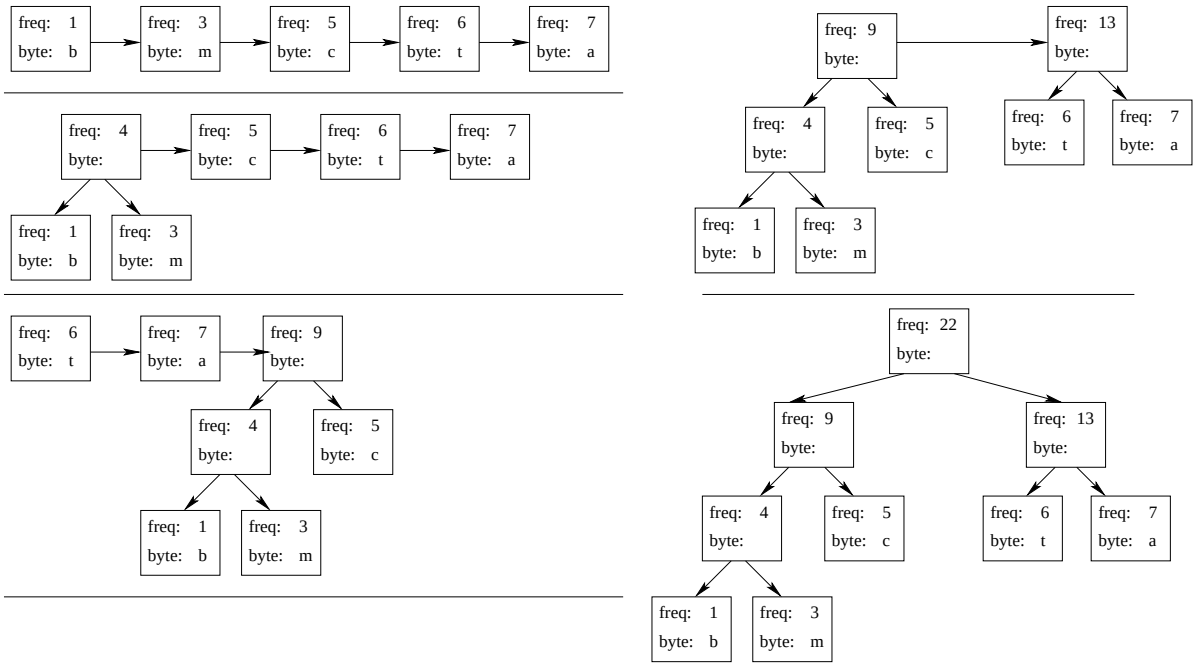
Next, remove the first two nodes from the list.



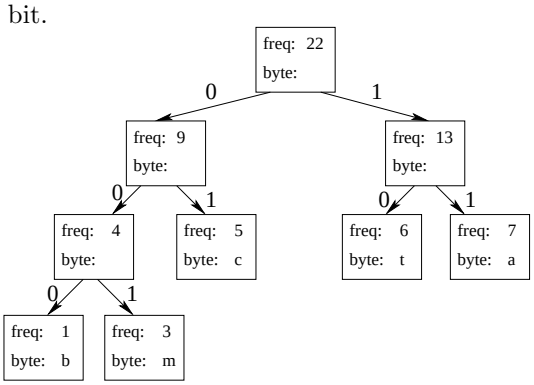
Construct a new node based on the previously removed nodes. The frequency stored in the new node is sum of the frequencies stored in the previously removed nodes. The new node will also point to the previously removed nodes as its children (forming a tree). The first node that was removed (the one with the lower frequency) should become the left child. The second node that was removed should become the right child.



Insert the newly created node back into the list and continue the above steps until there is only one node remaining in the list.



The resulting tree can be used to generate the bit encoding for each byte that appeared in the original input. When traversing the tree, each traversal of a left pointer denotes a `0` bit. Each traversal of a right pointer denotes a `1` bit.



Lookup Table

Perform a depth-first search of the tree to create a lookup table for each byte and its encoding. Using this table, read the input file again and output an encoded file. You may store the encoding as a character array.

At the beginning of the file you will need to print, on its own line, a "header" containing enough information to rebuild the tree from earlier. For consistency, print each byte that appeared in the input file and its frequency as determined earlier. For example, the "header" for the above example might be (with the number of characters given first)

```
5 a7 b1 c5 m3 t6
```

You can use `fprintf` to output the "header" and `fscanf` to read the "header". You may use `putc` and `getc` for printing and reading the actual contents of each file since the program essentially works on a single byte at a time.

Decoding

Decoding is a very similar process to that described above. The "header" information at the beginning of the file is read and used to reconstruct the tree as done for encoding. Each subsequent byte of the file is read and used to traverse the tree. A `0` bit indicates that the left child of the current node should be visited; a `1` indicates that the right child of the current node should be visited. When a leaf is reached, the byte stored at that leaf is printed to the output file and further searches begin at the root of the tree.

Binary Manipulation

For full credit, your encoding program must output a sequence of bits instead of the actual characters `0` and `1` and your decoding program must take a sequence of bits as input. Alternatively, your programs may output and read the characters `0` and `1`, but this will result in a point deduction as detailed in the Grading section below.

`huff-e`

Write a program named `huff-e` that takes a file as input and generates an encoded file using the algorithm described above. The name of the output file should be the same as the input file but with a `.huff` extension.

huff-d

Write a program named `huff-d` that takes a file as input and generates an decoded file using the algorithm described above. The name of the output file should be the same as the input file but with the `.huff` extension removed. If the given file does not have a `.huff` extension then report an error.

Submit

- All source code.
- A `Makefile` that can be used to construct your programs.
- A `README` file that specifies how to build your programs and that describes anything about your program that you feel I should know during grading.