Background

File compression is widely used in modern computing systems. For example, many image and video formats, such a JPEG or MP3, are compressed. Data, document and program files are also often compressed; for example using the zip format. Compressing these files allows them to be stored, streamed, downloaded and/or transferred more efficiently.

There are two primary forms of compression, lossy compression and lossless compression. With lossy compression, information is discarded during the compression process. Thus, the full original file cannot be reproduced exactly from the compressed file. In the case of images or video the loss of information is often acceptable because the compressed image or video looks nearly indistinguishable from the original but takes less storage or arrives more quickly. However, in the case of data, document files or programs, lossy compression would destroy the file contents and is thus unacceptable. A lossless compression scheme must be used for these types of files.

Huffman coding is a lossless data compression scheme. Huffman Coding is not usually used directly to compress files in modern systems, however, it or a related technique is used as part of many commonly used lossy and lossless compression schemes (e.g. zip, JPEG and MP3).

Huffman Coding Algorithm

The premise for Huffman Coding is that sequences of bits are used to represent information in the computer. For example, each ASCII character is typically represented by one byte (e.g. 0100 00012 = 6410 = 'A' in ASCII). What Huffman Coding does is use shorter sequences of bits for information that occurs frequently in the file and longer sequences for information that is rare.

The process that we will use to find a Huffman Code for the characters in an ASCII text is as follows:

* 1. First the frequency of each character that appears in the text is computed. For example:

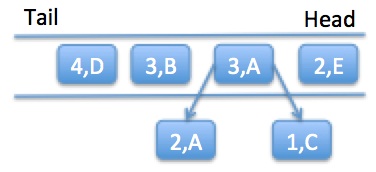
|  |  |
| --- | --- |
| **Text** | **Frequency** |
| AABBBCDDDDEE | http://users.dickinson.edu/~skalakm/cs232s19/labs/lab08files/letfreq.jpg |

* 1. A single node binary tree (i.e. only a root node) is created for each letter, with the letter as the value and the frequency as the key. These binary trees are then inserted into a priority queue. The priority queue orders the trees such that the minimum key appears at the head of the queue. Ties between the keys are broken using the values with smaller ASCII values coming first.\* (Note: the ASCII codes can be compared directly or the Character.compareTo method, which compares the ASCII codes, can be used.) The queue of trees for the letters above is shown here:



Notice that (2,A) comes before (2,E) in the queue due to the tie breaking criterion.

* 1. Until there is only one tree left in the priority queue, the first two trees are removed from the queue, joined into a single tree and replaced into the queue. The two trees are joined by making a new root node and the two trees becoming its left and right sub-trees. The tree with the larger key becomes the left sub-tree and the tree with the smaller key becomes the right sub-tree. The root of the new tree has its key given by the sum of the keys from the roots of the two trees being joined. The value at the root of the new tree is the smaller character (by ASCII value) from the roots of the trees being joined. Thus, the value of the root of each tree will always be the character with the smallest ASCII value in the tree. After removing and joining the first two trees the queue appears as shown here:



Notice that the location of the new tree in the queue is determined by both is key (3, tied with the 3 for B) and its value (A smaller than B, by ASCII code and the Character.compareTo) method.

The next two steps of this process are shown below:

|  |  |
| --- | --- |
| **i.** | **ii.** |
| http://users.dickinson.edu/~skalakm/cs232s19/labs/lab08files/queue3.jpg | http://users.dickinson.edu/~skalakm/cs232s19/labs/lab08files/queue4.jpg |

* 1. The final step is to compute the Huffman Code for the characters. This is done by imagining each left branch being labeled with a 0 and each right branch being labeled with a 1. The code for a letter is then simply the sequence of 0's and 1's that are encountered on a path from the root to the leaf with the character as its value. For example, the tree below has the edges labeled with 0's and 1's. The table to the right of the tree gives the Huffman Code for the characters.

|  |  |
| --- | --- |
| http://users.dickinson.edu/~skalakm/cs232s19/labs/lab08files/finaltree.jpg | http://users.dickinson.edu/~skalakm/cs232s19/labs/lab08files/charcodes.jpg |

* 1. Another description of this algorithm along with an example can be found in Section 5.6 in our text.

Problem

Ultimately you will compute a Huffman Code for a given text and use it to compress that text. However, this is a complex process, so this problem also requires that your program be able to display several intermediate results.

The first line of input will contain one character indicating the result to display as follows:

* + F: Display the frequency count for the characters in the text.
  + T: Display the key,value pairs of the nodes in the final tree in level order.
  + H: Display the Huffman Code table for the characters in the text.
  + M: The original text encoded using the Huffman Code.

The exact format in which to display each of these results is given below in the Output Format section.

Scoring

The scoring for correctness portion of this lab will be divided based on the intermediate results above. The correctness score will be a weighted average of the correctness for each type of result. The weights for each result are as follows:

* + F: 25%
  + T: 50%
  + H: 15%
  + M: 10%

Input Format

The first line of the input will contain a single character from [F,T,H,M].

The remaining lines of text are the text from which to compute the Huffman Code. There is no specified limit to the number of lines of text or to the length of the lines. However, each line can be read using a Scanner. Each line, except for the final line of text will be terminated with a newline (\n) character. The text will contain only characters with [ASCII values](http://www.asciitable.com/) of 9 (tab), 10 (newline) and between 32-126 inclusive.

Output Format

The output format required for each type of result is given below.

* + F: Frequency Table

The frequency table will be displayed with one character and one frequency per line separated by a colon (e.g. A:27). The characters will be displayed in ASCII order with those with smaller ASCII values appearing before those with larger ASCII values. Characters not appearing in the text should not be displayed.

* + T: Tree Nodes

The tree nodes will be displayed one per line with the value and key separated by a colon (e.g. D:12).

* + H: Huffman Code

The Huffman Code will be displayed with one character and one code per line separated by a colon (e.g. A:100). The characters will be displayed in ASCII order with those with smaller ASCII values appearing before those with larger ASCII values. Characters not appearing in the text should not be displayed.

* + M: The Huffman Coded Text

The Huffman Coded text will be displayed as a single line of 0's and 1's.

Anytime a newline character appears as part of the output (e.g. in the frequency table to indicate the number of newlines in the text), it should be displayed as *\n* (e.g. \n:27 *if there are 27 lines*). Similarly, Tab character should be displayed as *\t* (e.g. \t:7).

Sample Input 1

F  
AABBBCDDDDEE

Sample Output 1

A:2  
B:3  
C:1  
D:4  
E:2

Sample Input 2

T  
AABBBCDDDDEE

Sample Output 2

A:12  
B:7  
A:5  
D:4  
B:3  
A:3  
E:2  
A:2  
C:1

Sample Input 3

H  
AABBBCDDDDEE

Sample Output 3

A:100  
B:01  
C:101  
D:00  
E:11

Sample Input 4

M  
AABBBCDDDDEE

Sample Output 4

100100010101101000000001111

**Notes**

\* This tie breaking condition is not necessary to get a valid and optimal Huffman Code. But agreeing on how ties will be broken ensures that all correct solutions will give the same code for a given input.