Huffman Compression

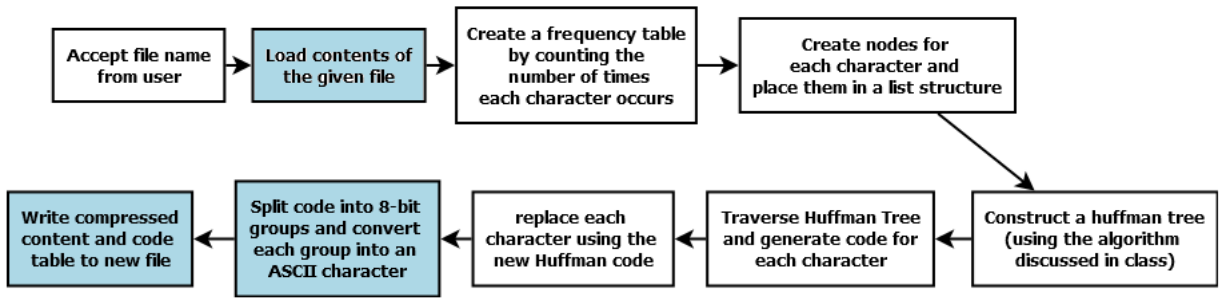
(This document was originally prepared by Salessawi Ferede (Ph.D.))

Instructions

To complete this assignment you need to implement the blocks shown in the pictures below**. The code for blocks shown in blue is already provided to save you time.** Therefore, you do not have to implement them yourself.

1. Compression

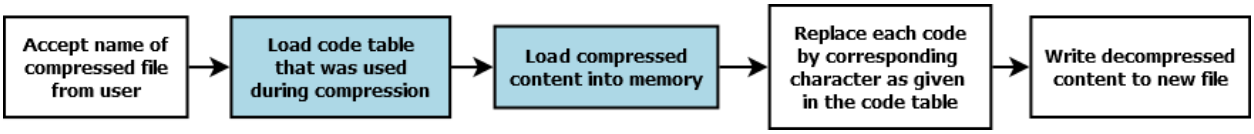
A Huffman compressor generally follows the steps shown below:



- Your program should start by accepting the full path of the file to be compressed from the standard I/O. An example input might be something like: *C:\Users\abebe\Desktop\myfile.txt*.
- After getting the file name you must load the entire content of the file. You can use the method **readOneFile** in **fileManagement** class.
- Next, you have to create the frequency table of characters not words. You may store the frequency table in an array (or any other structure you find suitable)
- Based on the frequency table you need to create nodes that will be part of the Huffman tree. You may store these nodes in a priority queue (or any other structure you find suitable).
- The next steps involve implementing the Huffman Coding algorithm discussed in class.
    - After constructing the Huffman tree, you should generate the appropriate code table by traversing the tree.
    - Using that code table, you should generate a string of ones and zeros.
- The code table and the string of ones and zeros must be written to the compressed file. The code for doing this is given as part of the starter code (**ManageCompressedFile**.java). More specifically, it is given in the method **writeACompressedFile.** So you do not need to write it. The details of this class is discussed in the section *"Understanding the Starter Code"*

2. Decompression

The decompression program reverses what was done above



- The first step is to accept the full path for compressed file.
- In addition to the bit string that was saved to file, we need to have the code table that was used for encoding. The code for extracting this information from the compressed file is given as part of the starter code. More specifically, it is given in the method, **readCompressedFile.** So you do not need to write it. The details of this class is discussed in the section *"Understanding the Starter Code"*
- After having the bit string and code table at hand, you need to replace the zeros and ones with the appropriate symbol.

Understanding the Starter Code

Let's assume the string to be compressed is: **SUSIE SAYS IT IS EASY.**

The frequency and possible Huffman Codes for each symbol can be given as: (**these are your output after compression to give to the starter code**)

Character	Frequency	Code (from Huffman Coding)
A	2	010
E	2	1111
I	3	110
S	6	10
T	1	0110
U	1	01111
Y	2	1110
[Space]	4	00
. (Full Stop)	1	01110

1. Compression

To create a compressed file you may use the method:

```
Boolean writeACompressedFile(String bitString,char[] symbols,String[] codeForSymbols,
String outputFile)
```

The parameters for the writeACompressedFile method have the following meanings:

- **String bitString**: this is the string of ones and zeros you generated using the Huffman coding algorithm. From the above example, the bitString can be seen to be "10011111011011110010010111010001100110001101000111101010111001110"

- **char[] symbols**: this array should contain the list of characters in the text. In our example, it would contain the characters {‘A’, ‘E’, ‘I’, ‘S’, ‘T’, ‘U’, ‘Y’, ‘ ’, ‘.’}
- **String[] codeForSymbols**: this array should contain the corresponding code of each of the characters stored in the symbols array. In our case the array becomes: {"010","1111","110","10","0110","01111","1110","00","01110"}
- **String outFileName**: this parameter is used to pass the full path of the compressed file. For example *C:\Users\abebe\Desktop\myfile.cmp*

The method returns true if the file is successfully created.

Example Usage:

```
String bitString="1001111101101111001001011101000110011000110100011111010101110011110";

char[] symbols = {'A', 'E', 'I', 'S', 'T', 'U', 'Y', ' ', '.'};

String[] codeForSymbols={"010","1111","110","10","0110","01111","1110","00","01110"};

String outputFile="output.txt";

manageCompressedFile compressedFile = new manageCompressedFile();

compressedFile.writeACompressedFile(bitString,symbols,codeForSymbols, outputFile);
```

2. Decompression

To extract information from the decompressed file, you can use the method:

```
CompressedContent readCompressedFile(String InputFile);
```

The method returns a **CompressedContent** object that has the following attributes:

- **CompressedContent.bitString**: A ***string*** containing the original string of zeros and ones
- **CompressedContent.symbols:** A **character array** containing the list of characters in the original text. It can be considered as the first column of the code table
- **CompressedContent.**codeForSymbols**:** A **string array** containing the binary codes corresponding to the symbols. It can be considered as the second column of the code table.

Example Usage:

```
        manageCompressedFile compressedFile = new manageCompressedFile();
       String InputFile="output.txt";
       CompressedContent data = compressedFile.readCompressedFile(InputFile);
       String bitString= data.bitString;
       char[] symbols = data.symbols;
       String[] codeForSymbols = data.codeForSymbols;
```

How Does the ManageCompressedFile class work anyways? (Optional Reading)

Even if you don't need to understand the internals the ManageCompressedFile class in order to use it, it is beneficial for you to know how it works.

1. Writing a Compressed File

- Suppose the bit string given below is generated after the Huffman Coding Step:
  "10011111011011110010010111010001100110001101000111101010111001110"
- We must arrange the bits into groups of eight (this is because a single character is represented by eight bits in the ASCII table): so
  [1][00111110][11011110][01001011][10100011][00110001][10100011][11010101][11001110]
- Each group is then converted to a base ten integer: [1][62][222][75][163] [49][163][213][206]
- In the next step, each of the integers are converted to the corresponding ASCII character:
  [ ][>][Þ][K][£][1][£][Õ][Î]
- After compression, the text "**SUSIE SAYS IT IS EASY."** becomes " **>ÞK£1£ÕÎ"**
- When the compressed file is created, the code table and the bit length of the codes are indicated in the header. As this header will take up some space itself, Huffman Coding is not a good algorithm for compressing files that are already small.

2. Reading a compressed file

The steps taken while reading a compressed file are the inverse of the ones discussed above.