## 2. Question two (60 Marks)

In this question you need to implement a basic data compression / decompression program using the popular Huffman compression algorithm.

**Background knowledge:**

1) Prefix codes:

A prefix code is most easily represented by a binary tree in which the external nodes are labeled with single characters that are combined to form the message. The encoding for a character is determined by following the path down from the root of the tree to the external node that holds that character: a ‘0’ bit identifies a left branch in the path, and a ‘1’ bit identifies a right branch. In the following tree, the code for ‘c’ is ‘1111’, because the external node holding b is reached from the root by taking 4 consecutive right branches. An example of prefix codes is given in the table below.

```
          ( )
      0 /       \ 1
   (a)          ( )
            0 /       \ 1
          ( )          ( )
      0 /    \ 1   0 /     \ 1
     (d)    (b)   (r)      ( )
                       0 /    \ 1
                       (!)    (c)
```

| Sample character encoding | |
|---|---|
| ! | 1110 |
| a | 0 |
| b | 101 |
| c | 1111 |
| d | 100 |
| r | 110 |

In the example above, the character 'a' is encoded with a single bit, while the character 'c' is encoded with 4 bits. This is a fundamental property of prefix codes. In order for this encoding scheme to reduce the number of bits in a message, we use short encodings for frequently used characters, and long encodings for infrequent ones. A second fundamental property of prefix codes is that messages can be formed by simply stringing together the code bits from left to right. For example, from the above encoding tree the bit string:

```
0101110011110100010111001110
```

encodes the message 'abracadabra!'. The first 0 must encode 'a', then the next three '101' must encode 'b', then '110' must encode 'r', and so on as follows:

```
|0|101|110|0|1111|0|100|0|101|110|0|1110
 a  b   r  a   c  a  d  a  b   r  a  !
```

The codes can be run together because no encoding is a prefix of another one. This property defines a prefix code, and it allows us to represent the character encodings with a binary tree, as shown above.

2) Huffman encoding

A Huffman Code is an optimal prefix code that guarantees unique decodability of a file compressed using the code. The code was devised by Huffman as part of a course assignment at MIT in the early 1950s. The goal of an optimal code is to assign the minimum number of bits to each symbol (letter) in the alphabet. ASCII is an example of a fixed length code. There are 100 printable characters in the ASCII character set, and a few non printable characters, giving 128 total characters. Since $lg_2 128 = 7$, ASCII requires 7 bits to represent each character. The ASCII character set treats each character in the alphabet equally, and makes no assumptions about the frequency with which each character occurs.

A variable length code is based on the idea that for a given alphabet, some letters occur more frequently than others. This is the basis for much of information theory, and this fact is exploited in compression algorithms to use as few bits as possible to encode data without 'losing' information. The Huffman coding algorithm produces an optimal variable length prefix code for a given alphabet in which frequencies are pre-assigned to each letter in the alphabet. Symbols that occur more frequently have a shorter code-word than symbols that occur less frequently. The two symbols that occur least frequently will have the same code-word length.

**Algorithms and tasks**

For this question, your program `TextZip` should implement the following six tasks defined in four sub-questions.

(1) Given a default prefix code tree and its compressed file ('a.txz'), decompress the file and save the uncompressed file into a text file ('a.txt'), also output the size of the file and compression ratio;

Possible algorithm:

- ❖ Use the `BitReader` class to read each bit of the compressed file ('a.txz') and decompress it. The algorithm for decoding characters is as follows.
  - ▪ Start at the root of the prefix tree.
  - ▪ Repeat until you reach an external leaf node.
    - Read one message bit.
    - Take the left branch in the tree if the bit is '0'; take the right branch if it is '1'.
  - ▪ Read the character in that leaf node and write it into the output file
  - ▪ Repeat the above process until reach the end of the compressed file
- ❖ Read the file size of both the compressed and decompressed file, calculate the compression ratio = (compressed file size / uncompressed file size)

Your program should have similar input/outputs as follows:

```
c:\105\A4> java TextZip –a
a.txz decompressed by abdc001
Size of the compressed file: 5 bytes
Size of the original file: 12 bytes
Compressed ratio: 41.66666666666667%
```

As from the above result, the original message contains 12 bytes which would normally require 96 bits of storage (8 bits per character). The compressed message uses only 5 bytes, or 41.67% of the space required without compression. The compression factor depends on the frequency of characters in the message, but ratios around 50% are common for English text. Note that for large messages the amount of space needed to store the description of the tree is negligible compared to storing the message itself, so we have ignored this quantity in the calculation.

Note that you also need to save the decompressed file into a text file ('a.txt'). You should implement the `decompress` method in the code frame given with the following method header.

```
public static void decompress(BitReader br, TreeNode huffman,
                              FileWriter fw)
```

Where `BitReader` is a file reader that will read the compressed file bit by bit; `huffman` is the prefix code tree; `FileWriter` is a standard Java class for output to files. Note that the data object in the TreeNode is an instance of the `CharFreq` Class (definition provided).

(2) Given a text file ('file.txt'), calculate the frequencies of each characters in the file and save the character frequency table into a file ('file.freq'); based on the character frequencies, build up the prefix code tree using Huffman algorithm; print out the prefix codes;

(2.1) calculate the character frequencies in a text file

Possible algorithm:

❖ Using an array of integers to store the frequency for each character

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| char | | | a | b | | e | | s | t | |
| frequency | 0 | … | 10 | 0 | … | 15 | … | 3 | 4 | … |
| index | 0 | | 97 | 98 | | 101 | | 115 | 116 | |

when you read a character from the file, lookup the corresponding indexed (ASCII code) items in the array and increment its frequency.

❖ Save the non-zero frequencies and their character into a file ('file.freq')

❖ Create and return an array list of tree node from the non-zero frequencies characters in the frequency array. Note that the tree node list is originally arranged according to their ASCII order from left to right.

In this sub-question you should implement the `countFrequencies` method in the code frame given with the following method header.

```
public static ArrayList countFrequencies(FileReader fr,
                                         PrintWriter pw)
```

where `FileReader` is a standard Java class to read in files, and `PrintWriter` is a standard Java class to output to files. This method return an ArrayList of tree node, where the data object in the TreeNode is an instance of the `CharFreq` Class (definition provided).

(2.2) build up the prefix code tree using Huffman algorithm

Possible algorithm:

- The Huffman algorithm works by constructing a prefix code tree from the bottom up, using the frequency counts of the symbols to repeatedly merge subtrees together. Intuitively, the symbols that are more frequently occurred should appear higher in the tree structure and the symbols that are less frequent should be lower in the tree. Let's use an example character table to illustrate this:

| Character | Frequency | Sum(bits) |
|---|---|---|
| ! | 1 | 8 |
| a | 5 | 40 |
| b | 2 | 16 |
| c | 1 | 8 |
| d | 1 | 8 |
| r | 2 | 16 |
| Total | | 96 |

Now we simply create a 'forest' of six (one for each character) trees. This should be obtained by passing an array list of tree node into the method call.

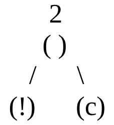
| | | | | | |
|---|---|---|---|---|---|
| 1 | 5 | 2 | 1 | 1 | 2 |
| (!) | (a) | (b) | (c) | (d) | (r) |

Weight of the tree (here: frequency)

Next we remove the least frequency element (tree) in the list from the left to right twice. Then merge these two minimum-weight trees, where the latest extracted element will be the right subtree. The new merged tree will have a weight of sum of its two subtrees, append the merged tree onto the end of the tree node list.

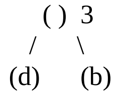
The sum of the weights

| | | | | |
|---|---|---|---|---|
| 5 | 2 | 1 | 2 | 2 |
| (a) | (b) | (d) | (r) | ( ) |



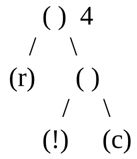
In the next step we again merge the trees of minimum weight:

| | | | |
|---|---|---|---|
| 5 | 2 | 2 | 3 |
| (a) | (r) | ( ) | ( ) |



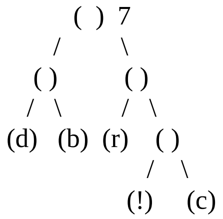
Step 3:

| | | |
|---|---|---|
| 5 | 3 | 4 |
| (a) | ( ) | ( ) |



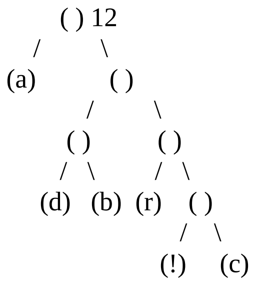
Step 4:

| | |
|---|---|
| 5 | 7 |
| (a) | ( ) |



Final step:

| |
|---|
| 12 |
| ( ) |



Thus our optimal prefix code using Huffman algorithm is as follows.

| Character | Code | Frequency | Sum(bits) |
|---|---|---|---|
| ! | 1110 | 1 | 4 |
| a | 0 | 5 | 5 |
| b | 101 | 2 | 6 |
| c | 1111 | 1 | 4 |
| d | 100 | 1 | 3 |
| r | 110 | 2 | 6 |
| Total | | | 28 |

In this sub-question you should implement the `buildTree` method in the code frame given with the following method header.

```
public static TreeNode buildTree(ArrayList trees)
```

where `trees` is an ArrayList of tree node, where the data object in the TreeNode is an instance of the `CharFreq` Class containing character and its frequency. This method returns the root of the prefix code tree generated using Huffman algorithm.

(2.3) print out the prefix code for each character from prefix codes tree generated.

Possible algorithm

- ❖ Modify the common tree traversal method, traverse the prefix code tree and print out the prefix codes. The following recursive algorithm might be useful.
  - ▪ If it is a leaf node, print out the character and its prefix code
  - ▪ Otherwise go to the left subtree and keep track of '0'; go to the right subtree and keep track of '1'

In this sub-question you should implement the `traverse` method in the code frame given with the following method header.

```
public static void traverse(TreeNode t, String code)
```

where `t` is a tree node in the prefix code tree, `code` is a string which keeps the track of the route taken

Your program should have similar input/outputs as follows:

```
c:\105\Asst5> java TextZip -f a.txt a.freq
a.txt prefix codes by abdc001
character code:
a : 0
d : 100
b : 101
r : 110
! : 1110
c : 1111
```

Note that you also need to save the character frequency into a file ('file.freq'), which has the format as follows.

```
! 1
a 5
b 2
c 1
d 1
r 2
```

(3) Given a text file (‘file.txt’), create the prefix code tree using the Huffman algorithm and save its frequency table (‘file.freq’); using the prefix codes generated to compress the text file and save the compressed file into (‘file.txz’), also output the compression ratio.

Possible algorithm:

- ❖ Using the methods implemented in previous questions (such as `countFrequency`, `BuildTree`) to generate the prefix code tree using Huffman algorithm;
- ❖ Modify the `traverse` method implemented in the previous question to store the character and its prefix code in to a table structure for future lookup;
- ❖ using the prefix code table to lookup the code for each character in the text file, perform the compression by replacing each character with its prefix code, use the `BitWrite` class to write each bit into a compressed file (‘file.txz’);
- ❖ since you using the `countFrequency` method, it will also save the character frequency into a file (‘file.freq’) for future decompression usage.

Your program should have similar input/outputs as follows:

```
c:\105\Asst5> java TextZip -c file.txt file.freq file.txz
file.txt compressed by abdc001
Size of the compressed file: 932302 bytes
Size of the original file: 1749990 bytes
Compressed ratio: 53.27470442688244%
```

(4) Given a compressed file (‘file.txz’) and its frequency table (‘file.freq’), create the prefix code tree using the Huffman algorithm; using the prefix codes tree to decompress the ‘file.txz’ file and save the uncompressed file into a text file (‘file.txt’), also output the compression ratio.

Possible algorithm:

- ❖ Using the methods implemented in previous questions (such as `countFrequencies BuildTree`) to generate the prefix code tree using Huffman algorithm. You may need to modify the `countFrequency` method in order to generate frequency array list from the saved frequency table file (‘file.freq’) instead of counting them from a text file;
- ❖ Once you correctly obtained the frequency array list from the frequency table, you can use the `BuildeTree` and `decompress` method implemented in the previous question to perform decompression and save the result back to a text file (‘file.txt’).

Your program should have similar input/output as follows:

```
c:\105\Asst5> java TextZip -d file.txz file.freq file.txt
file.txz decompressed by abdc001
Size of the compressed file: 932302 bytes
Size of the original file: 1749990 bytes
Compressed ratio: 53.27470442688244%
```

***NOTE:***

- Your UPI must appear in the output, as shown in the example above