Data structures and Algorithms Assignment 2 – Text Compression: Huffman Coding

Due Date: Thursday 20th December by 12noon

Demo Date: Thursday 20th and Friday 21st December –

A schedule will be published on Moodle

Introduction

Huffman coding is a scheme that assigns variable-length bit-codes (binary strings) to characters, such that the lengths of the codes depend on the frequencies of the characters in a typical message. As a result, encoded messages take less space (as compared to fixed-length encoding such as ASCII or UNICODE) since the letters that appear more frequently are assigned shorter codes. This is performed by first building a Huffman coding **Tree** based on a given set of frequencies. From the tree, bit-codes for each character are determined and then used to encode a message. The tree is also used to decode an encoded message as it provides a way to determine which bit sequences translate back to a character. (See the text passage at the end of the assignment for more details).

You are to implement a Huffman coding system using the frequency table given below (this is also in a text file on my student share called LetterCount.txt in folder Assignment 1). Your final solution should be able to encode any message that uses the characters of the english alphabet or decode any bit sequence to the corresponding characters. There are three main parts to the system, *generating the huffman tree, encoding and decoding*.

Frequency Table

E	21912	M	4761
T	16587	F	4200
A	14810	Y	3853
O	14003	W	3819
I	13318	G	3693
N	12666	P	3316
S	11450	B	2715
R	10977	V	2019
H	10795	K	1257
D	7874	X	315
L	7253	Q	205
U	5246	J	188
C	4943	Z	128

Part 1: Generating the Huffman Tree (40 marks)

Here you are required to generate a binary tree (Huffman tree) representing the bit-codes of each character in the alphabet. The characters that occur more frequently will have shorter bit codes than those occurring less frequently. Traversing the huffman tree from root to leaf should reveal the bit-code of the characters, where a left branch encodes a 0 (zero) and a right branch a 1. In this way, the leaves of the tree will represent the characters in our alphabet and the paths from the root to the leaves will reveal the binary codes for the characters. See the tutorial at the end of the document for an example on how to create a Huffman tree from a frequency table. You should reuse ADTs that you have seen in class where you can. For example you could extend the BinaryTree class to create a HuffmanTree class. You could also use an ADT List to represent the frequency table above. Some changes or extensions may be needed to the Node classes, e.g. Node.java, TreeNode.java but these should become apparent during your analysis of the problem.

Part 2: Encode (20 marks)

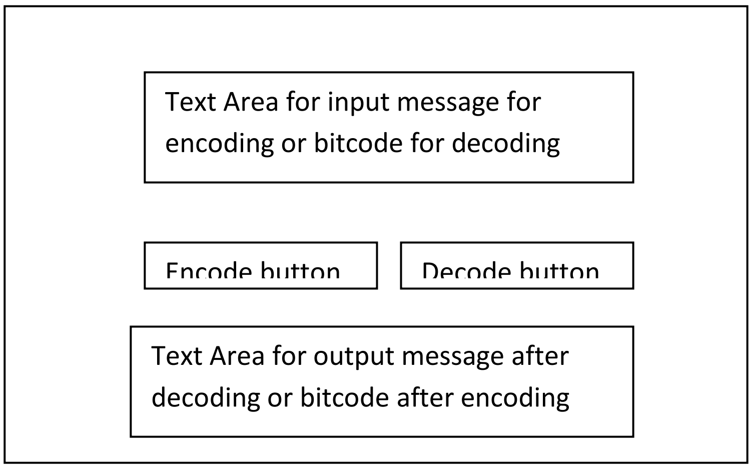
This part of the assignment will involve generating the binary code for a message from the alphabet. This will be obtained by traversing the Huffman tree from the root to the leaf for each character in the message. There are a number of approaches that could be taken to this part. One might be to find the leaf node in the tree for a particular character and then back track from there to the root recording zero's and one's depending on whether you encounter a left or right branch. You would also need to reverse the code at the end as you went backwards up the tree. Display also the compression ratio for the encoded message by assuming that the fixed length representation for each character is 7 bits (the ASCII standard).

Part 3: Decode (20 marks)

This part of the assignment involves taking a binary string (the bit code) and regenerating the character message from the Huffman tree. To do this you will need to use the binary code to guide you down the right branches (remember, 0 for a left branch and 1 for a right branch) until you encounter a leaf and therefor e a character in the message.

Part 4: Program Interface (20 marks)

The interface for this program should be quite straightforward and resemble something like the following (with a bit more colour and dazzle ☺):


Text Area for input message for encoding or bitcode for decoding
Encode button
Decode button
Text Area for output message after decoding or bitcode after encoding

You will receive marks for:

1. All work attempted and submitted.
2. The quality of your solution design and your report describing this in detail.

3. The quality of your code and in-program documentation (comments).
4. You will receive marks for a bug-free, fully tested, application.
5. The quality of the test data.
6. The quality and professionalism of the design of the user graphical user interface. Ideally, this should be user friendly, realistic and have a well-designed Java look and feel while not permitting the entry of erroneous data.
7. Innovation and creativity in your deign and solution.

You are required to deliver:

1. A Java application including **source code and .class file**
2. A **description of your program design** stating the design problems that you needed to address and how you surmounted them. For each of these, state the design problem and them your solution.
3. **Screen shots** illustrating test cases
4. A professionally written and word-processed **report** that contains all of the above.
5. Signed copy of the plagiarism form

Extract from Data Structures and Algorithms in Java, Goodrich and Tamassia.

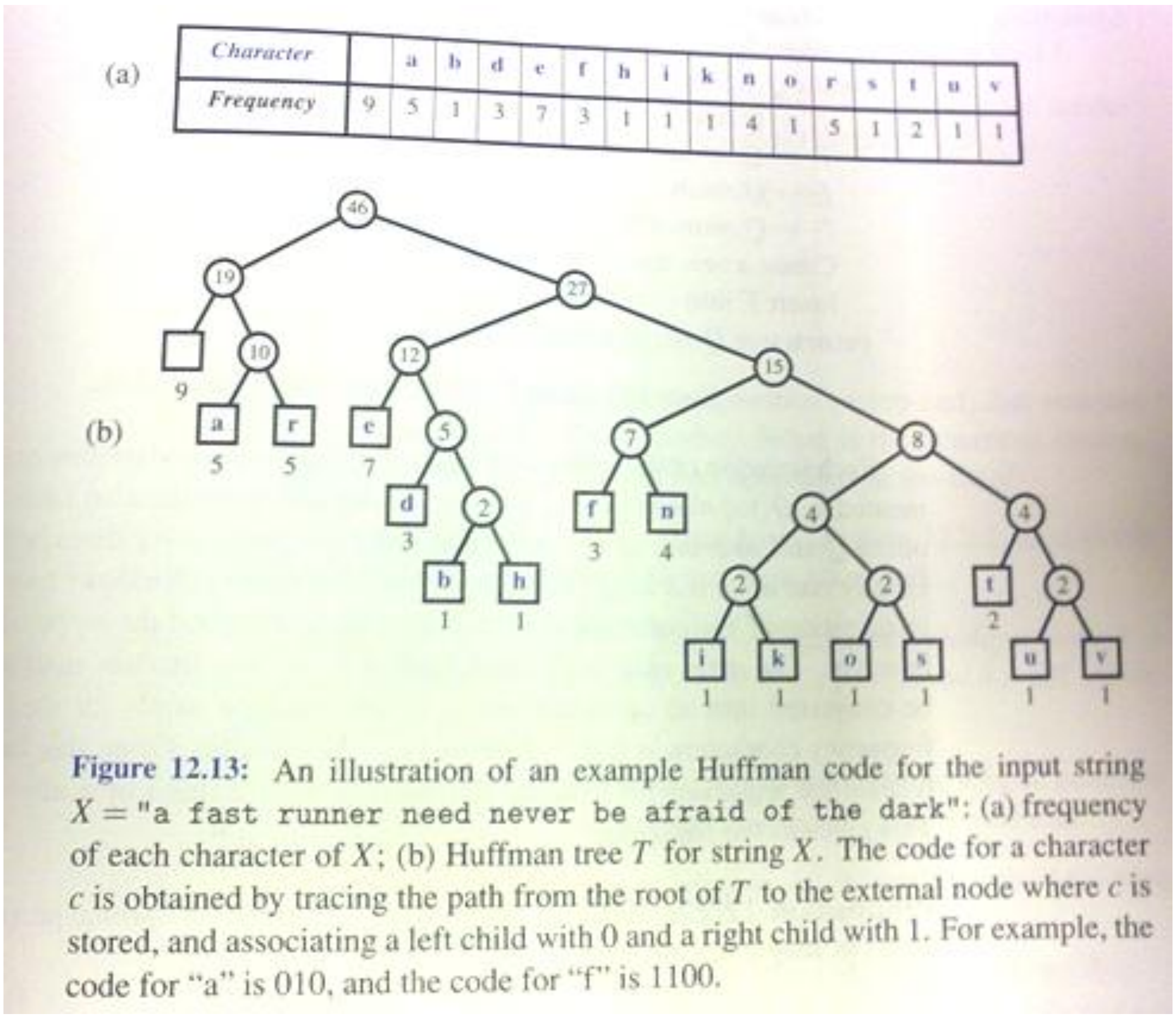
12.4.3 Huffman Coding

Let us consider another application of the greedy method, this time to a string compression problem (we give another string compression algorithm in Section 14.1.6). In this problem, we are given a string $X = x_1x_2\ldots x_n$ defined over some alphabet, such as the ASCII or Unicode character sets, and we want to efficiently encode X into a small binary string Y (using only the characters 0 and 1). The method we explore in this section is to use a ***Huffman code***. Standard encoding schemes, such as the ASCII and Unicode systems, use fixed-length binary strings to encode characters (with 7 bits in the ASCII system and 16 in the Unicode system). A Huffman code, on the other hand, uses a variable-length encoding optimized for the string X. The optimization is based on the use of character ***frequencies***, where we have, for each character c, a count $f(c)$ of the number of times c appears in the string X. The Huffman code saves space over a fixed-length encoding by using short code-word strings to encode high-frequency characters and long code-word strings to encode low-frequency characters.

To encode the string X, we convert each character in X from its fixed-length code word to its variable-length code word using a Huffman code optimized for X, and we concatenate all these characters in order to produce the encoding Y for X. In order to avoid ambiguities when using such a variable-length code, we insist that no code word in our encoding is a prefix of another code word in our encoding. Such a code is called a ***prefix code***, and it simplifies the decoding of Y so as to get back X. (See Figure 12.13.) Even with this restriction, the savings produced by using Huffman's method to define a variable-length prefix code to encode X into Y can be significant, particularly if there is a wide variance in character frequencies (as is the case for natural language text in almost every spoken language).

Huffman's algorithm for producing an optimal variable-length prefix code for X is based on the construction of a binary tree T that represents the code. Each edge in T represents a bit in a code word, with each left child edge representing a "0" and each right child edge representing a "1." Each external node v is associated with a specific character, and the string for that character is defined by the listing of edges in the path from the root of T to v. (See Figure 12.13.) Each external node v has a ***frequency*** $f(v)$, which is simply the frequency in X of the character associated with v. In addition, we give each internal node v in T a frequency, $f(v)$, that is the sum of the frequencies of all the external nodes in the subtree rooted at v.

Huffman's algorithm for building the tree T is based on the greedy method. It begins with each of the n given characters being the root node of a single-node tree, and proceeds in a series of rounds. In each round, the algorithm takes the two root nodes with smallest frequencies and merges them into a single tree. It repeats this process until only one node is left. (See Code Fragment 12.8.)



(a)

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1

(b)

Figure 12.13: An illustration of an example Huffman code for the input string $X =$ `"a fast runner need never be afraid of the dark"`: (a) frequency of each character of X; (b) Huffman tree T for string X. The code for a character c is obtained by tracing the path from the root of T to the external node where c is stored, and associating a left child with 0 and a right child with 1. For example, the code for "a" is 010, and the code for "f" is 1100.

A quick tutorial on generating a huffman tree – siggraph.org

Let’s say you have a set of numbers and their frequency of use and want to create a huffman encoding for them:

```
FREQUENCY       VALUES of ALPHABET
 ---------        -----
     5              1
      7              2
    10               3
    15               4
    20               5
    45               6
```

Creating a huffman tree is simple. Sort this list by frequency and make the two-lowest elements into leaves, creating a parent node with a frequency that is the sum of the two lower element's frequencies:

```
12:*
```

```
        /   \
      5:1   7:2
```
The two elements are removed from the list and the new parent node, with frequency 12, is inserted into the list by frequency. So now the list, sorted by frequency, is:
```
        10:3
        12:*
        15:4
        20:5
        45:6
```
You then repeat the loop, combining the two lowest elements. This results in:
```
        22:*
        /   \
     10:3   12:*
            /   \
          5:1   7:2
```
and the list is now:
```
        15:4
        20:5
        22:*
        45:6
```
You repeat until there is only one element left in the list.
```
        35:*
        /   \
      15:4  20:5

        22:*
        35:*
        45:6

            57:*
        ___/    \___
       /            \
     22:*          35:*
    /   \          /   \
 10:3   12:*     15:4   20:5
        /   \
      5:1   7:2

        45:6
        57:*

                                   102:*
                ___________________/    \__
               /                           \
            57:*                          45:6
        ___/    \___
       /            \
     22:*          35:*
    /   \          /   \
 10:3   12:*     15:4   20:5
        /   \
      5:1   7:2
```
Now the list is just one element containing 102:*, you are done.

This element becomes the root of your binary huffman tree. To generate a huffman code you traverse the tree to the value you want, outputing a **0** every time you take a lefthand branch, and a **1** every time you take a righthand branch. (normally you traverse the tree backwards from the code you want and build the binary huffman encoding string backwards as well, since the *first* bit must start from the top).

Example: The encoding for the value **4** (15:4) is **010**. The encoding for the value **6** (45:6) is **1**

Decoding a huffman encoding is just as easy : as you read bits in from your input stream you traverse the tree beginning at the root, taking the left hand path if you read a **0** and the right hand path if you read a **1**. When you hit a leaf, you have found the code.