CSCI 2110 Computer Science III
Data Structures and Algorithms
ASSIGNMENT NO. 3

Date given: Friday, October 26, 2018

Due: Saturday, November 10, 2018, 11.55 PM

The objective of this assignment is to implement the Huffman coding algorithm using the binary tree data structure.

Download BinaryTree.java, Frequency.txt and Pokemon.txt files given next to the Assignment link.

Problem Summary: You are given a table of letters of the English alphabet and their frequencies. Build a Huffman tree with the alphabet symbols and their probabilities. Derive the Huffman codes. Using the codes, encode a given text file with the codes. Decode the encoded text file and show that it is the same as the input text file.

Problem in Detail:
In order to help you with the assignment, here's the Huffman algorithm step-by-step procedure (as discussed in the lectures).

Step 1: Read the text file frequency.txt. Its link is given next to this lab document. It contains the frequency of letters in the English alphabet based on a sample of 40,000 words as shown below. (The file actually contains each letter and its frequency on two separate lines).
(Source: http://pi.math.cornell.edu/~mec/2003-2004/cryptography/subs/frequencies.html)

E	12.02
T	9.10
A	8.12
O	7.68
I	7.31
N	6.95
S	6.28
R	6.02
H	5.92
D	4.32
L	3.98
U	2.88
C	2.71
M	2.61
F	2.30
Y	2.11
W	2.09
G	2.03
P	1.82
B	1.49
V	1.11
K	0.70
X	0.17
Q	0.11
J	0.10
Z	0.07

To do this step, you will find it useful to create a class called Pair.java that defines the letter and its probability as an object.

```
public class Pair
{
   private char letter;
   private double prob;

   //constructor
   //get and set methods
   //toString method
}
```

You can create an Arraylist of Pair objects

ArrayList<Pair> freqs = new ArrayList<Pair>();

and store the items into the Arraylist as you read them. Of course, you will need other variables and methods to count the frequencies and convert them into probabilities.

Step 2: Using this set of letters and frequencies, build the Huffman tree.

Step 3.1: Create a queue of Binary Tree nodes. Each Binary Tree node is of type Pair. The queue can be implemented as another simple Arraylist, where enqueue means adding an item to the end of the Arraylist and dequeue means removing the item at index 0. That is, the queue is an **Arraylist of type <BinaryTree<Pair>>.** The queue contains these sorted according to the increasing order of their frequencies. This is your **Queue S**. This is done by checking the **Arraylist freqs** for values in increasing order, creating the binary tree nodes and enqueueing them in the queue.

If you enumerate the Queue S, it should have the Pair objects in increasing order of their frequencies, something like this:

('Z', 0.07) ('J', 0.10), etc.

Step 3.2 : Now initialize another queue T (another Arraylist) of type <BinaryTree<Pair>>.

Step 3.3 : Build the Huffman tree according to the algorithm discussed in the lectures.

For instance, in the above example, first ('Z', 0.07) and ('J', 0.10), will be dequeued from S. Create a node with the combined frequency. What do you put as the character for the combined node? You can put a dummy character, say '&'. So ('&',0.17) will be the parent node, and ('Z', 0.07) and ('J', 0.10), will be the left and right children. This tree will be enqueued to Queue T.

You keep repeating the above procedure and building the Huffman tree according to the algorithm given below:

Pick the two smallest weight trees, say A and B, from S and T, as follows:

a) If T is empty, A and B are respectively the front and next to front entries of S. Dequeue them from S.

b) If T is not empty,
   i) Find the smaller weight tree of the trees in front of S and in front of T. This is A. Dequeue it.
   ii) Find the smaller weight tree of the trees in front of S and in front of T. This is B. Dequeue it.

3. Construct a new tree P by creating a root and attaching A and B as the subtrees of this root. The weight of the root is the combined weights of the roots of A and B.

4. Enqueue P to T.

5. Repeat steps 2 to 4 until S is empty.

6. After step 5, if T's size is > 1, dequeue two nodes at a time, combine them and enqueue the combined tree until T's size is 1. The last node remaining in the queue T will be the final Huffman tree.

Step 4: Derive the Huffman codes.

The following methods can be used for finding the encoding. They use a String array of 26, one spot for each letter.

```
  public static void findEncoding(BinaryTree<Pair> t, String[] a, String prefix)
 {
    if (t.getLeft()==null&& t.getRight()==null)
    {
       a[((byte)(t.getData().getValue()))-65]= prefix;
    }
    else
    {
      findEncoding(t.getLeft(), a, prefix+"0");
      findEncoding(t.getRight(), a, prefix+"1");
    }

  }

  public static String[] findEncoding(BinaryTree<Pair> t)
  {
    String[] result = new String[26];
    findEncoding(t, result, "");
    return result;
  }
```

Step 5: Read the sample text file Pokemon.txt which is shown below:

```
POKEMON TOWER DEFENSE
YOUR MISSION IN THIS FUN STRATEGY TOWER DEFENSE GAME IS TO
HELP PROFESSOR OAK TO STOP ATTACKS OF WILD RATTATA
SET OUT ON YOUR OWN POKEMON JOURNEY TO CATCH AND TRAIN ALL
POKEMON AND TRY TO SOLVE THE MYSTERY BEHIND THESE ATTACKS
```

```
YOU MUST PLACE POKEMON CHARACTERS STRATEGICALLY ON THE
BATTLEFIELD SO THAT THEY STOP ALL WAVES OF ENEMY ATTACKER
DURING THE BATTLE YOU WILL LEVEL UP AND EVOLVE YOUR POKEMON
YOU CAN ALSO CAPTURE OTHER POKEMON DURING THE BATTLE AND ADD
THEM TO YOUR TEAM
USE YOUR MOUSE TO PLAY THE GAME
GOOD LUCK
```

Encode each letter using the Huffman codes that you have determined. Do not encode spaces and newline characters. Leave them as they are. Write the encoded file into another text file, Encoded.txt.

Step 6: Read the encoded text file and decode it. Write the decoded file into yet another text file, Decoded.txt. If you have done everything correctly, then Decoded.txt must be the same as Pokemon.txt.

Submit a zip file containing all the source codes (.java files), Frequency.txt, Pokemon.txt, Encoded.txt and Decoded.txt.