CSCI 2110 Data Structures and Algorithms
Assignment No. 5

Date Given: Monday, March 22nd, 2021
Date Due: Monday, April 5th, 2021, 11.59 PM

This assignment has just one exercise. It is designed to help you get familiar with the binary tree data structure by implementing the Huffman coding algorithm. Download the example code/files provided along with this assignment document. You will need the following files to complete your work:

BinaryTree.java (Generic Binary Tree Class)
Pokemon.txt (Sample text file for input)

Submission Requirements:

- No submission other than a single ZIP file will be accepted.
- You MUST SUBMIT .java files that are readable by the TA. If you submit files that are unreadable such as .class file, the lab will be marked 0.
- Please additionally comment out package specifiers.

What to submit:

Submit one ZIP file containing all source code (files with .java suffixes) and a txt file containing console outputs. Your final submission should include the following files: Huffman.java, **Pair.java**, **HuffmanDemo.java**, **BinaryTree.java**, Pokemon.txt, Huffman.txt, Encoded.txt, Decoded.txt.

Problem Summary:

To complete your submission you will have to write a program that implements 2 methods:

- An encode method that
  (1) reads in an ASCII text file,
  (2) counts the number of occurrences of each non-whitespace character,
  (3) converts those frequencies to probabilities,
  (4) builds a Huffman tree consisting of characters and their probabilities,
  (5) derives Huffman codes, and
  (6) outputs a text file containing an encoded version of the original file as well as a text file containing the Huffman codes used in the encoding process.

- A decode method that
  (1) reads in an encoded ASCII text file and a file containing Huffman codes, and
  (2) decodes the first file according to coding scheme described on page 5, outputting a single decoded text file.

Note: If your methods are executed consecutively, with the outputs from the encode method passed to the decode method, the file output by the decode method should be **identical** to the original input file passed to the encode method.

Problem Specification

You will write a class that provides methods to encode and decode text files according to the Huffman coding algorithm. Call your class file **Huffman.java**. The method is described below.

I. Encoding

The encode method in **Huffman.java** will implement the Huffman coding algorithm. The method header should be:

```
public static void encode()throws IOException
```

- First, your method should **accept input from a user** (specifying the name of an input file) and read an ASCII text file. The file will contain only characters in the extended ASCII set (characters corresponding to decimal values 0-255). Your method should initialize a String variable to reference the text captured from the input file. The sample input file you are provided looks like this:

```
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!
```

- Next, your method should count the number of occurrences of each non-whitespace character in the text. You may find the following code snippet useful.

```
int[] freq = new int[256];
char[] chars = text.replaceAll("\\s", "").toCharArray();

for(char c: chars)
    freq[c]++;
```

- Having counted frequencies, you should be able to derive the relative probabilities of the characters. For the purpose of this assignment, a rounding operation shown below will provide acceptable accuracy.

  ```
  Math.round(freq[i]*10000d/chars.length)/10000d
  ```

- You will need to keep track of characters and their relative probabilities in order to build your Huffman tree. To do this, you will likely find it useful to create a class called **Pair.java**.

  ```
  public class Pair implements Comparable<Pair>{
      // declare all required fields
      private char value;
      private double prob;

      //constructor
      //getters
      //setters
      //toString

      /**
      The compareTo method overrides the compareTo method of the
      Comparable interface.
      */
      @Override
      public int compareTo(Pair p){
          return Double.compare(this.getProb(), p.getProb());
      }
  }
  ```

  Note: You can create an Arraylist of Pair Objects to keep track of your Pairs as you create them from your array of character frequencies.

- Next, you will build a Huffman tree. To build a Huffman tree you will require two queues of type **BinaryTree<Pair>**. You may use structures from the Java Standard Libraries or you may choose to trivially implement your queues using ArrayLists (appending new elements and removing at index 0). Both options are acceptable.

- Below we refer the lecture notes (pages 17-19) of Module 6 as Module 6.

- Enqueue BinaryTrees with data fields referencing the Pair Objects from the previous step into your first queue (Queue S from the algorithm in Module 6). Be sure to enqueue Pairs in sorted order (ascending), with the lowest probabilities at the head or the queue. Your second queue (Queue T from the algorithm in Module 6) should remain empty for the time being.

- Now you can implement the rest of the Huffman algorithm from Module 6.

1) Pick the two smallest weight trees, say A and B, from queues 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.

2) 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. (You may find it useful to assign a character value to P that is outside of the range of extended ASCII values. Something like '⁂' could work well.)

3) Enqueue tree P to queue T.

4) Repeat the previous steps until queue S is empty.

5) If the number of elements in queue T is greater than 1, dequeue two nodes at a time, combine them (see strep 2) and enqueue the combined tree until queue T's size is 1. The last node remaining in the queue T will be the final Huffman tree.

- You're now ready to derive the Huffman codes. The following methods can be used to find the encoding:

```
private static String[] findEncoding(BinaryTree<Pair> bt){
    String[] result = new String[256];
    findEncoding(bt, result, "");
    return result;
}
private static void findEncoding(BinaryTree<Pair> bt, String[] a, String
prefix){
    // test is node/tree is a leaf
    if (bt.getLeft()==null && bt.getRight()==null){
        a[bt.getData().getValue()] = prefix;
    }
    // recursive calls
    else{
        findEncoding(bt.getLeft(), a, prefix+"0");
        findEncoding(bt.getRight(), a, prefix+"1");
    }
}
```

- Print your derived codes to an output file called **Huffman.txt**. You can use a PrintWriter to create your output file and capture Strings list this:

```
output = new PrintWriter("Huffman.txt");
output.println(// text to print to file);
```

The resulting file should be a structured like this:

```
Symbol     Prob.      Huffman  Code
O          0.1077     001
T          0.1121     011
E          0.1165     100
…
…
```

Note: Your actual values/codes may differ slightly.

- Finally, encode the original text String from the input file. Encode each character in the text using the Huffman codes you have derived. Print your encoded String to an output file called **Encoded.txt**. Do not encode spaces, new line characters, or other whitespace characters. Simply copy these to your output whenever they are encounter in the input.

II. Decoding

The Huffman class's decode method will decode text that has been encoded using the Huffman coding algorithm. The method header should be:

```
public static void decode()throws IOException
```

- First, your method should **accept input from a user** (specifying the name of an encoded input file and the name of a file containing huffman codes) and read a pair of ASCII text files. The first file will contain a message encoded according to a scheme described by the second.
- Your method should initialize a String variable to reference the text captured from the encoded file, and build a structure (or pair of structures) from the second to permit you to easily work with the codes. You may find the following code snippet useful:

```
Scanner ls = new Scanner(codes);
// consume/discard header row and blank line
ls.nextLine();
ls.nextLine();

while(ls.hasNextLine()){
    char c = ls.next().charAt(0);
    ls.next(); // consume/discard probability
    String s = ls.next();
    // put the character and code somewhere useful
    // like a Map or pair of ArrayLists
```

}

- Finally, decode the encoded text String from the encoded file. Replace each coded String with its corresponding character using the codes you read in. Print your decoded String to an output file called **Decoded.txt**. Do not attempt to decode spaces, new line characters, or other whitespace characters. Simply copy these to your output whenever they are encountered.

III. Demo Program

Write a short demo program called **HuffmanDemo.java**. Your demo should call each of the methods you have written (encode and decode) once. A sample run of your demo program should look like this:

```
Enter the filename to read from/encode: Pokemon.txt
Printing codes to Huffman.txt
Printing encoded text to Encoded.txt

* * * * *

Enter the filename to read from/decode: Encoded.txt
Enter the filename of document containing Huffman codes: Huffman.txt
Printing decoded text to Decoded.txt
```

After your program has executed the files **Encoded.txt** and **Huffman.txt**, **Decoded.txt** should be present in your working directory. These along with Pokemon.txt will be part of your submission for this assignment.