Lab 7.2 – Huffman Compression[1]

Learning Goals

1) Develop skills using binary trees.
2) Practice traversing a binary tree using recursion in a variety of scenarios.
3) Learn to perform binary level file operations.
4) Integrate a broad array of computer science skills to complete a complex task.

Your Task

Extract the files from **huff.zip** into your project.

The package **huff** includes a number of Java classes. Here are the ones that you need to be most familiar with:

- **BitInputStream** – An object for reading from an input stream one bit at a time. The only method you will be calling is **int readBits(int numBits)** which reads the specified number of bits from the input stream and returns them as an int.

- **BitOutputStream** – An object for writing to an output stream one bit at a time. The only method you will be calling is **void writeBits(int numBits, int value)**. The lower order "numBits" from "value" will be written to the output. For example, **writeBits(5, 10)** would write the bits **01010** to the output.

  Note, in BitOutputStream, there is a constant DEBUG. When true, the output of the program is text. When false, the output is binary. **Once you complete testing your compression program, make sure to change this mode to FALSE.** You won't be able to decompress a file that was compressed with DEBUG turned on!

- **HuffNode** – Node used to build the encoding trie.

- **HuffProcessor** – All of your coding goes here.

Your task is to implement the **compression** and **decompression** helper methods in HuffProcessor. The controlling methods **compress** and **decompress** have been written for you. They call the helper methods, including:

- readForCounts
- makeTreeFromCounts
- makeCodingsFromTree
- writeHeader
- writeCompressedBits
- readHeader

[1] Assignment adapted from: http://www.cs.duke.edu/courses/cps100/current/assign/huff/src/

- readCompressedBits.

Detailed comments are included in HuffProcessor to describe the functionality of each function. Be sure to go back and review the paper and pencil algorithms that we completed in class as you work on this assignment.

There are several debugging methods called in **compression** and **decompression** to help you debug the helper methods: **printCounts printEncodingTree,** and **printCodingsArray**. You will likely get the same output for the encoding trie as I, but you may not. (Recall that there are many optimal encoding tries.) So, if you get a different trie, you will need to check your output by hand to verify that the trie makes sense. Likewise, the printed array-table of codings may differ if your trie differs.

Testing

Run HuffMain to test your compression and decompression. You can compress any kind of file, although the debugging methods will print oddly for non-ASCII text files.

The ultimate test of your program will be the compression and decompression of both a text file (**melville.txt**, 101,453 bytes) and an image file (**hackbca.bmp**, 1,001,078 bytes).

After compression, the resulting files (**melville.txt.hf** and **hackbca.bmp.hf**) should be have sizes ~57,195 bytes and ~257,254 bytes respectively.

After decompression, the new files (**melville.txt.dehf** and **hackbca.bmp.dehf**) must match the original file. (Again, make sure your compressed file is generated with DEBUG mode off! You won't be able to decompress a file generated with DEBUG turned on.)

A short test file called **short_test.txt** has also been included. Following is the output I receive when I run the program. You may get a different trie and output that is also correct – but they should be the same length (432 bits).

short_test.txt.hf (DEBUG = true)

```
0000 1001 1011 1010 0110 0001 1001 1101 0001 0011
0010 1100 1110 0110 0010 0110 1001 0100 1100 0111
0011 0110 1001 0011 0111 1100 1100 1110 1001 1010
0001 0011 0011 0110 0000 0000 0001 0001 0111 0100
1100 0101 0011 1001 0010 0111 1001 0100 1010 1001
0011 0010 0100 0100 0001 1011 0101 1010 0001 1111
1000 0111 1100 0111 1001 0100 1100 1111 1010 0101
1101 1100 1101 1001 0111 0101 0011 0101 1001 1010
1010 0000 1001 0110 1011 1110 0011 1001 1010 0000
1110 1111 0010 0001 0110 0110 0000 0010 1011 1101
1110 1001 1001 0101 0011 1100 0010 1111
```

Console print from compression: See below

```
//Counts
32 ( ): 8
46 (.): 1
84 (T): 1
97 (a): 3
98 (b): 1
99 (c): 2
100 (d): 1
101 (e): 6
102 (f): 1
103 (g): 2
104 (h): 2
105 (i): 3
109 (m): 2
110 (n): 2
111 (o): 2
114 (r): 2
115 (s): 6
116 (t): 5
121 (y): 2
```

```
//Trie – may differ
 L
   L
     L
       L
         'n'
       R
         'a'
     R
       't'
   R
     L
       'e'
     R
       's'
 R
   L
     L
       L
         'i'
       R
         L
           'c'
         R
           'm'
     R
       L
         L
           'o'
         R
           'g'
       R
         L
           'h'
         R
           L
             'f'
           R
             'EOF'
   R
     L
       L
         L
           L
             '.'
           R
             'b'
         R
           'r'
       R
         L
           'y'
         R
           L
             'T'
           R
             'd'
     R
       ' '
```

```
//Codings – may differ
32 ( ): 111
46 (.): 110000
84 (T): 110110
97 (a): 0001
98 (b): 110001
99 (c): 10010
100 (d): 110111
101 (e): 010
102 (f): 101110
103 (g): 10101
104 (h): 10110
105 (i): 1000
109 (m): 10011
110 (n): 0000
111 (o): 10100
114 (r): 11001
115 (s): 011
116 (t): 001
121 (y): 11010
256 (EOF): 101111

Header out.  Total bits:
219
Body out.  Total bits:
213
```

Compression Results (in the user interface):
Total original: 52 bytes
Total new: 54 bytes

Question: Why is the compressed file larger than the original?