Zipped

CSC 252 – Sections A and B

**Assigned:** 7 August 2014 **Due**: 14 August 2014

**There are Java files needed to complete this lab! Please download them, if you haven’t already.**

**Overview**

In compression, every byte counts. Many systems integral to our daily lives, including Internet, rely on quick and effective compression algorithms to do their jobs well and make our jobs easier.

**Outline**

*HuffmanCompressor.java*

For this lab, you will be implementing Huffman compression. You can read more about it in Section 9.4 of the book. The interface for HuffmanCompressor is as follows:

**public** **byte**[] compress(HuffmanTree tree, **byte**[] b);

**public byte**[] decompress(HuffmanTree tree, **int** uncompressedLength, **byte**[] b);

The interface should be able to take in any array of bytes, calculate a good compression strategy via a Huffman coding tree, and compress the array into a smaller array. The following pseudocode must **always** be true excepting the worst case:

b == decompress(compress(b)) && b.length > compress(b).length

*HuffmanTree.java*

The HuffmanTree is a representation of a Huffman coding tree. In its constructor is a byte array of data on which to calculate a probability distribution. The API is as follows:

**public** HuffmanTree(**byte**[] b) { /\* constructor stuff \*/ }

**public byte** toByte(Bits bits);

**public void** fromByte(**byte** b, Bits bits);

**Thinking Questions**

How is this like Morse Code without spaces? Why is a message written with a Huffman coding tree unambiguous?

How could I transmit a Huffman tree as part of the message? If I didn’t, what other ways might there be to figure out how to parse the message?

What is the write data type to store the Huffman tree in memory? What about the right data type for tracking progress in the algorithm?

What happens to the resulting trees that a generated from the same data but with different tie breaking rules?

**Approach**

Huffman compression can be realized in three simple steps:

*Frequency Distribution.* First, say we have the following byte array to compress:

byte[] b = new byte[] { 45, 56, 67, 78, 89, 12, 23, 34, 45, 23, 45, 67, 45 };

The algorithm counts how many it sees and records that information for later use. ***Note:*** *If you already know Java fairly well, you may be tempted to use a HashMap here. There is a simpler way. Consider this: If we are encoding bytes, don’t we already know the maximum size of our alphabet?*

The frequency distribution for the above array is as follows (in no particular order):

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 45 | 23 | 67 | 78 | 89 | 12 | 56 | 34 |
| 4 | 2 | 2 | 1 | 1 | 1 | 1 | 1 |

*Forming the tree.* After the distribution is created, the Huffman algorithm creates nodes for each and places them into a priority queue, prioritizing by ascending frequency and breaking ties in a consistent way. The algorithm takes the two first nodes off of the priority queue, joins them into a composite node and places that composite node back into the priority queue. The composite node’s frequency is the sum of the frequencies of its two components.

*Compression.* For each byte in the message to be compressed, the algorithm walks the generated tree to find out the associated code. For the message above, these are the codes generated (ties were broken by order of appearance in the sample data):

45 = 10  
 23 = 110  
 67 = 11  
 78 = 1001  
 89 = 0111  
 12 = 1110  
 56 = 1111  
 34 = 010

From this, we can infer that the algorithm created a list bit bits like this:

1011111110010111111011001010110101110

Given that originally the byte array was 13 long, meaning 13 bytes, simple path will show that the now 5 bytes of data above has earned us a 39% compression ratio!

*Padding.* Because we must ultimately pick a data type in most languages, padding is necessary to fill up any remaining space in the last byte. So, the above, being 37 characters long, becomes:

1011111110010111111011001010110101110000

For the purposes of this lab, you are not storing the length nor the Huffman tree into the message itself (see Stretch Goals), but there are algorithms for doing so efficiently.

*Decompression.* For decompression, the compressed message is read in one bit at a time, each bit advancing a reference pointer to a location in the Huffman tree. Each time the algorithm reaches a leaf, it adds the discovered by to an output.

**Gotchas**

When working with bytes in Java, there are a couple of pitfalls to keep in mind:

1. Binary operators (+, \*, &, >>) default to ints in Java. So, **byte** asdf = 1; **byte** sdfg = asdf >> 1; will throw a compiler error because the compiler promoted the “asdf >> 1” to an integer before assignment. Casting the above expression to a byte before assignment will fix the error.
2. Bit shifting is a very efficient and nice way to complete this lab. Bit shifting has a small issue when shifting a negative number to the left. Instead of moving the sign bit in addition to the other seven bits in a byte, it only moves the seven bits, maintaining the sign bit. To get around this, simply pad the byte to an integer and then take the smallest 8 bits out:

String asString = String.*format*("%8s", Integer.*toBinaryString*(b & 0xFF)).replace(' ', '0');

These can also be avoided by not doing bit shifting, but, of course, bit shifting is super-fast. ☺

**What to Study**

Read Sections 5.3, 6.3, and 9.4 in the Levitin text.

Wikipedia has a nice article on [Huffman coding](http://en.wikipedia.org/wiki/Huffman_coding).

**There are Java files needed to complete this lab! Please download them, if you haven’t already.**

**What to Hand In**

Pass-offs will be done via one or more unit tests; however, please still check in your source code to your git repo along with a jar of your compiled files. Anything that doesn’t compile after the due date will receive **1** point off their total lab grade.

You should be checking in at least **HuffmanTree.java** and **HuffmanCompressor.java**.

**How You Will Be Graded**

Of the points possible, **30%** is for a verbal defense. **30%** is for a working compressor, **30%** for a working decompressor, and **10%** for telling me my secret message.

**Stretch Goals**

* The Huffman trees itself would obviously need to be stored as well. There is a nice Wikipedia article outlining how to do this efficiently. Change your byte[] to store the Huffman tree as well as either the length of the original message or some kind of indication of how much padding there is on the end. **(+10%)**