**Huffman Coding Assignment**

**For CS211, Bellevue College, Fall 2013**

(original from Marty Stepp, UW CSE 143, modified for Bellevue College)

For this last CS211 assignment, you are to submit the following Java code for a Huffman Tree data structure, along with the Huffman Node structure that you have used.

For the Huffman Nodes, read through the description of Huffman compression below and the following organization will become apparent. We’re building a binary tree, where the “data” is a count of the frequency of each character.

public class HuffmanNode implements Comparable<HuffmanNode> { public int frequency; public char character; public HuffmanNode left; public HuffmanNode right;

The Huffman Node class will also need a boolean isLeaf() method, plus a static method to actually provide a count of the characters in an input file, and place those counts into a Map, with character as the unique key mapped into the integer counts of that character:

public static Map<Character, Integer> getCounts(FileInputStream input)

Your Huffman Tree class must have the following defined:

public HuffmanTree(Map<Character, Integer> counts) // Constructor public StringBuilder compress(InputStream inputFile) // inputFile is a text file public StringBuilder decompress(StringBuilder inputString) //inputString 1’s & 0’s public String printSideways() // as per the method presented in Chapter 17.

Note that the input file here can actually be any format, but plain text will allow us to see what we’re doing. The compress method returns a string of 1’s and 0’s (bits) as per the description below. The decompress method can take that string of 1’s and 0’s, use the Huffman tree structure (different for every text file) and recreate the text file from those bits. Finally, the printSideways here is similar to what is presented in Chapter 17 of Building Java Programs, but in this case we build a string to return into a graphical text area. You should actually use the String Builder class, as it is mutable.

We normally don’t use System.out calls in a GUI. But I understand they are useful during debugging. You must comment out all System.out calls before submitting your work. The client program (HuffmanGUI.java) is provided, and you need to meet these specifications listed above. I expect two files to be submitted (HuffmanNode.java and HuffmanTree.java) which I will copy into my Eclipse folder and run the provided GUI for testing.

Huffman coding is an algorithm devised by David A. Huffman of MIT in 1952 for compressing text data to make a file occupy a smaller number of bytes. This relatively simple compression algorithm is powerful enough that variations of it are still used today in computer networks, fax machines, modems, HDTV, and other areas.

Normally text data is stored in a standard format of 8 bits per character, commonly using an encoding called ASCII that maps every character to a binary integer value from 0-255. The idea of Huffman coding is to abandon the rigid 8-bits-percharacter requirement and use different-length binary encodings for different characters. The advantage of doing this is that if a character occurs frequently in the file, such as the letter 'e', it could be given a shorter encoding (fewer bits), making the file smaller. The tradeoff is that some characters may need to use encodings that are longer than 8 bits, but this is reserved for characters that occur infrequently, so the extra cost is worth it.

The table below compares ASCII values of various characters to possible Huffman encodings for the text of Shakespeare's *Hamlet*. First step is to count the number of occurrences of each character in the text. Frequent characters such as space and 'e' should have short encodings, while rarer ones like 'z' have longer ones.

|  |  |  |  |
| --- | --- | --- | --- |
| **Character** | **ASCII value** | **ASCII (binary)** | **Huffman (binary)** |
| ' ' | 32 | 00100000 | 10 |
| 'a' | 97 | 01100001 | 0001 |
| 'b' | 98 | 01100010 | 0111010 |
| 'c' | 99 | 01100011 | 001100 |
| 'e' | 101 | 01100101 | 1100 |
| 'z' | 122 | 01111010 | 00100011010 |

The steps involved in Huffman coding a given text source file into a destination compressed file are the following:

1. Examine the source file's contents and count the number of occurrences of each character.
2. Place each character and its frequency (count of occurrences) into a sorted "priority" queue.
3. Convert the contents of this priority queue into a binary tree with a particular structure.
4. Traverse the tree to discover the binary encodings of each character.
5. Re-examine the source file's contents, and for each character, output the encoded binary version of that character to the destination file.

# Encoding a File:

For example, suppose we have a file named example.txt with the following contents:

ab ab cab

In the original file, this text occupies 10 bytes (80 bits) of data. The 10th is a special "end-of-file" (EOF) byte.

**byte** 1 2 3 4 5 6 7 8 9 10

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **char** | 'a' | 'b' | ' ' | 'a' | 'b' | ' ' | 'c' | 'a' | 'b' | EOF |
| **ASCII** | 97 | 98 | 32 | 97 | 98 | 32 | 99 | 97 | 98 | 256 |
| **binary** | 01100001 | 01100010 | 00100000 | 01100001 | 01100010 | 00100000 | 01100011 | 01100001 | 01100010 | N/A |

In Step 1 of Huffman's algorithm, a count of each character is computed. (i.e. the getCounts method). The counts are represented as a map:

{' '=2, 'a'=3, 'b'=3, 'c'=1, EOF=1}

Step 2 of the algorithm places these counts into binary tree nodes, each storing a character and a count of its occurrences.

The nodes are put into a priority queue, which keeps them in sorted order with smaller counts at the front of the queue.

(The priority queue is somewhat arbitrary in how it breaks ties, such as 'c' being before EOF and 'b' being before 'a').

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 |  | 1 |  | 2 |  | 3 |  | 3 |
| 'c' | EOF | ' ' | 'b' | 'a' |

 front

back

Now the algorithm repeatedly removes the two nodes from the front of the queue (the two with the smallest frequencies) and joins them into a new node whose frequency is their sum. The two nodes are placed as children of the new node; the first removed becomes the left child, and the second the right. The new node is re-inserted into the queue in sorted order:



3



'

a

'



3



'

b

'



1



EOF



1

'

n

'

a

'

b

'



'

c

'



**2**



front



back



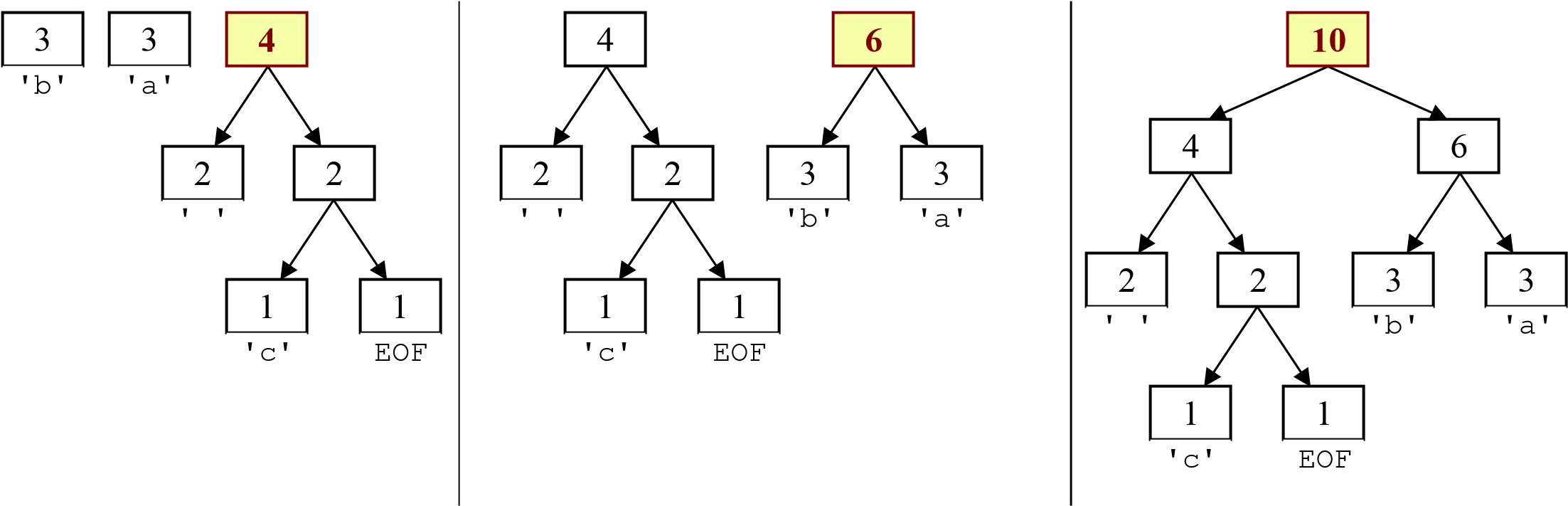
2



'

'

This process is repeated until the queue contains only one binary tree node with all the others as its children. This will be the root of our finished Huffman tree. The following diagram shows this process:



Notice that the nodes with low frequencies end up far down in the tree, and nodes with high frequencies end up near the root of the tree. This structure can be used to create an efficient encoding. The Huffman code is derived from this tree by thinking of each left branch as a bit value of 0 and each right branch as a bit value of 1:



2



'

'



1



EOF



1



'

c

'



2



4



3



'

a

'



3



'

b

'



6



10



**0**



**1**



**0**



**0**



**0**



**1**



**1**



**1**

The code for each character can be determined by traversing the tree. To reach ' ' we go left twice from the root, so the code for ' ' is 00. The code for 'c' is 010, the code for EOF is 011, the code for 'b' is 10 and the code for 'a' is 11. By traversing the tree, we can produce a map from characters to their binary representations. For this tree, it would be:

{' '=00, 'a'=11, 'b'=10, 'c'=010, EOF=011}

Using this map, we can encode the file into a shorter binary representation. The text ab ab cab would be encoded as:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **char** | 'a' | 'b' | ' ' | 'a' | 'b' | ' ' | 'c' | 'a' | 'b' | EOF |
| **binary** | 11 | 10 | 00 | 11 | 10 | 00 | 010 | 11 | 10 | 011 |

Use of the word “map” above is not random. Implementing this in code, you might find it useful to create a Map data structure that maps characters into the right string of bits. This really simplifies things, but still requires the building of a Huffman Tree to complete the assignment.

The overall encoded contents are 1110001110000101110011, which is 22 bits, or almost 3 bytes, compared to the original file which was 10 bytes. (Many Huffman-encoded text files compress to about half their original size.) Modern encoding algorithms are faster and more efficient, but this is where it all started over 50 years ago, and has often been elevated into other technology (<http://en.wikipedia.org/wiki/Huffman_coding>).

**byte** 1 2 3

|  |  |  |  |
| --- | --- | --- | --- |
| **char** | a b a | b c a b EOF | |
| **binary** | 11 10 00 11 | 10 00 010 1 | 1 10 011 00 |

Since the character encodings have different lengths, often the length of a Huffman-encoded file does not come out to an exact multiple of 8 bits. Files are stored as sequences of whole bytes, so in cases like this the remaining digits of the last bit are filled with 0s. You do not need to worry about this in the assignment; it is part of the underlying file system.

It might worry you that the characters are stored without any delimiters between them, since their encodings can be different lengths and characters can cross byte boundaries, as with 'a' at the end of the second byte above. But this will not cause problems in decoding the compressed file, because Huffman encodings have a *prefix property* where no character's encoding can ever occur as the start of another's encoding. This is important when you decode the file later.

# Decoding a File:

You can use a Huffman tree to decode text that was compressed with its encodings. The decoding algorithm is to read each bit from the file, one at a time, and use this bit to traverse the Huffman tree. If the bit is a 0, you move left in the tree. If the bit is 1, you move right. You do this until you hit a leaf node. Leaf nodes represent characters, so once you reach a leaf, you output that character. For example, suppose we are asked to decode a file containing the following bits:

101101000110111011

Using the Huffman tree, we walk from the root until we find characters, then we output them and go back to the root.



'

'



EOF



'

c

'



'

a

'



'

b

'



**0**



**1**



**0**



**0**



**0**



**1**



**1**



**1**

* First we read a 1 (right), then a 0 (left). We reach 'b' and output b. Back to the root. 101101000110111011
* We read a 1 (right), then a 1 (right). We reach 'a' and output a. 101101000110111011
* We read a 0 (left), then a 1 (right), then a 0 (left). We reach 'c' and output c. 101101000110111011
* We read a 0 (left), then a 0 (left). We reach ' ' and output a space. 101101000110111011
* We read a 1 (right), then a 1 (right). We reach 'a' and output a. 101101000110111011
* We read a 0 (left), then a 1 (right), then a 0 (left). We reach 'c' and output c. 101101000110111011
* We read a 1 (right), then a 1 (right). We reach 'a' and output a. 101101000110111011

So the overall decoded text is bac aca. (The input source reports when we reach an EOF character of 011, so we stop.)

When the selected input files get very large (try Moby.txt) the display will take forever (maybe 20 minutes) to load. This is a well know problem with the Java set text methods, as they require constant refresh, rebuffer, and reload. So I provide a check box on the GUI to skip the text displays, so you can see your compression can really run in under a second even for huge files. In real compression algorithms, the 100011100… bit stream would actually go out over the wire as binary data, and not be converted into character, strings, and graphics (so slow).

Deliverables: 1) HuffmanNode.java 2) HuffmanTree.java 3) QA document with screenshots showing your working program.