Introduction

Bitwise Compression is an algorithm created 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, burned directly into chips found in computer networks, modems, HD and 4K TV signal processors, and other areas.

Normally text data is stored in a standard format of 8 bits per character, commonly using ASCII encoding to map every character to a binary integer value from 0-255. The idea of Bitwise Compression is to abandon the rigid 8-bits-per-character 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 Bitwise encodings for the text of Shakespeare's *Hamlet*. Frequent characters such as space and 'e' have short encodings, while rarer ones like 'z' have longer ones. ASCII 32 is the ***space***. We will use ***sp.*** To be clearer but in the actual implementations it will be the actual space character.

Character	ASCII decimal	ASCII (binary)	Bitwise (binary)
'sp.'	32	00100000	10
'a'	97	01100001	0001
'b'	98	01100010	0111010
'c'	99	01100011	001100
'e'	101	01100101	1100
'z'	122	01111010	00100011010

You may wonder if a file written with these Bitwise character encodings will actually be smaller than a file written with ASCII or Unicode encodings. There are a couple factors to consider.

1. How many different characters are in the file. If there are a small number of different characters, then all Bitwise encodings will be less than 8 bits.
2. How high is the frequency variation between characters? If the characters all have the same frequency, then there will be a high number of characters that are longer than 8 bits. But if some characters are quite common, then the savings from shorter encodings for those characters will be high. If you look at a character frequency chart such as the one at https://en.wikipedia.org/wiki/Letter_frequency, you will see that the most common letter, e, is 165 times as common as the least common letter, z. Although it's not in the chart, the space character is even more common than the letter e.

Another concern could be how to determine when where each character ends and if we need to store a delimiter. As you will see below, no character encoding is a prefix of another. As an example, if the character e is encoded as 10, there will be no other character encodings which start with 10. Hence when 10 is found in a compressed file, we know that it represents the letter e.

The restriction on prefixes is also why some characters will take more than 8 bits. In the above example, all 3 bit or longer combinations which begin with 10 are disallowed, so there are less than the typical 256 possible character encodings in 8 bits.

Compression Steps

The steps involved in encoding and compressing with a Bitwise Compression algorithm for a given ***source*** text file into a ***destination*** compression 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 calculation into a tree node object of your creation (an inner class is fine).
3. Place these nodes into a priority queue of tree nodes.
4. Convert the contents of this priority queue into a binary tree with a particular structure.

5. Traverse the tree to define the binary encodings of each character. Create a map using the tree, with the character as the key, and the Bitwise encoding as the value. Save this map to a file.
6. Re-examine the ***source*** file's contents, and for each character, output the encoded binary version of that character to the ***destination*** file.

BitTree Project

Encoding a File

For example, suppose we have a file named *example2.txt* with the following contents:

```
ab ab cab
```

In the original file, this text occupies 9 bytes (80 bits) of data. The last is a special "end-of-file" (EOF) byte. This is a special case that needs to be added at the end of parsing your input file. We will use the character End Of Text, whose value is 3. But will call it EOF for clarity.

Byte	1	2	3	4	5	6	7	8	9	10
Char	'a'	'b'	' sp.'	'a'	'b'	' sp.'	'c'	'a'	'b'	EOF
ASCII	97	98	32	97	98	32	99	97	98	3
Binary	01100001	01100010	00100000	01100001	01100010	00100000	01100011	01100001	01100010	00000011

Step 1

Compute a frequency count of each character is computed. The counts are represented as a map:

MAP:

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

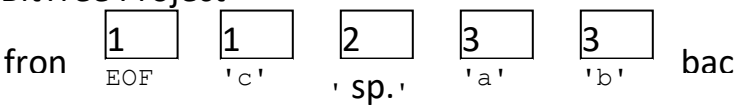
Step 2

Place these counts into tree nodes, each storing a character and a count of its occurrences. Note that these nodes are not put into a tree during this step.

Step 3

Place the tree nodes in a min-priority queue. In other words the nodes with **<u>smaller</u>** counts are at the front of the queue. Ties are broken by the ascii value of the characters being compared. Non-leaf nodes do not have ascii values, so do not break ties for them.

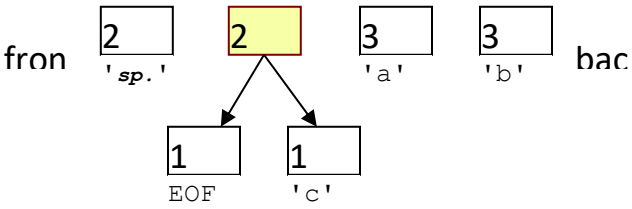
PRIORITY QUEUE:


fron
1
EOF
1
'c'
2
' sp.'
3
'a'
3
'b'
bac

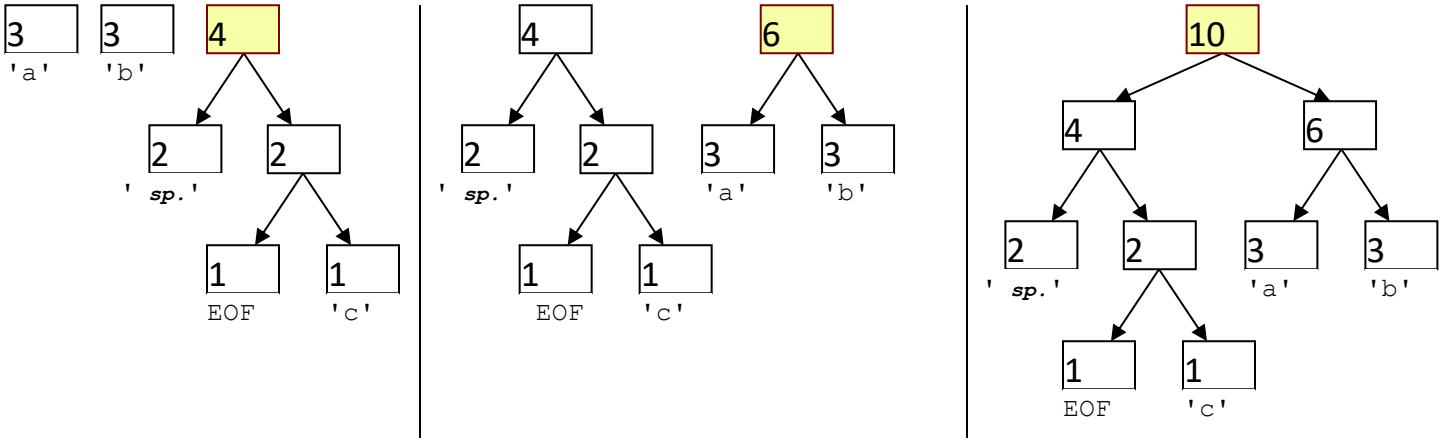
Step 4

Repeatedly remove the two nodes from the front of the queue (the two with the smallest frequencies) and joins them into a new 3-node binary tree whose root is a new node with frequency equal to their sum. The two nodes are placed as children of the new node with the first removed as the left child and the second as the right child. The new parent node has no character, only pointers to children, and the frequency total. The new 3-node tree is reinserted into the priority queue, again in sorted order, and again based on the frequency field.

BINARY TREE:



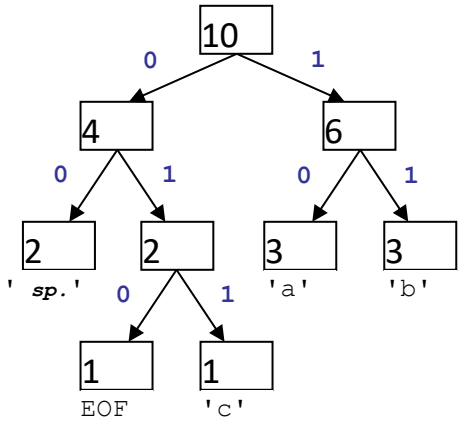
This process is repeated until the queue contains only one binary tree node, with all the other nodes as its children. This will be the root of our Bitwise tree. The following diagram shows this process:



Notice that all characters are located in leaves, only, and 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 Bitwise 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.

BitTree Project

BITWISE BINARY TREE:



Step 5

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

MAP:

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

Save this map as a file using HashMap.toString(). Examine your map file for example2, and you should see a text file similar to the above.

Step 6

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

char	'a'	'b'	'***sp.***'	'a'	'b'	'***sp.***'	'c'	'a'	'b'	EOF

binary	10	11	00	10	11	00	011	10	11	010

The overall encoded contents of the file are `10110010110001110110 10`, which is 22 bits, or almost 3 bytes, compared to the original file which was 9 bytes.

byte	1	2	3
char	a b *sp*. a	b *sp*. c a	b EOF
binary	10 11 00 10	11 00 011 1	0 11 010 00

A Detail: Originally you will not actually manipulate bits in your code. You will use the Strings "1" and "0" to represent your bits. This will allow you to focus on the algorithm, and gives you an easy way to print and debug your bits. Because there are 8 bits in a character byte, your "compressed" file will be 8 times larger than it could be. Once you are confident, proceed to Step 7. Meanwhile, simply divide your file sizes by 8 if you want to monitor your compression ratio.

Step 7

Actually Compress Your File.

It can be difficult to tell whether you have compressed/decompressed a file correctly. If you open **an actual** Bitwise-compressed binary file in a text editor the appearance will look like gibberesh because the text editor will try to interpret the bytes as ASCII encodings, which is not the way the data is stored. So far we have been having you write your bits as strings so you can visually verify that your algorithm is working during development. However, this defeats the purpose of compression, because the "compressed" file is actually larger than the original.

To compress/decompress files, you will want to read and write binary data one bit at a time. Java's built-in input/output streams read an entire byte at a time, which makes it difficult to examine each bit. Therefore we are providing two java files – BitOutputStream and BitInputStream with writeBit and readBit methods to make it easier. Every constructor and method of these classes throws IOException if something fails during the input/output process.

BitInputStream(String file) – Creates a BitInputStream reading input from the file
readBit() – Reads next bit from input (-1 if at the end of the file)
close() – closes the input

BitOutputStream(String file) – Create a BitOutputStream sending output to the file

writeBit() – Writes given bit to output. Note, if your final bits do not form a complete byte it will be padded with zeros.
close() – Writes any final bits and closes the output.

Once you feel fairly confident with your alogirthm, modify your compress method to additionaly write the ".bwt.bits" file as actual bits rather than strings of 1's and 0's. You will still output the string version as ".bwt.bits.txt"

Decompression

Decompression, aka decoding, is the process of converting a compressed file back to its original uncompressed from.
This process requires compressed file and the codes map file which was saved in compression step 5.

Step 1

Create the codes tree from the codes map file.
You will first need to read in and parse the codes map file. In our implementation the codes map file is a text file with the output from HashMap.ToString(). Here is the codes map file for example 2.
{ =00, a=10, b=11, ————————————=010, c=011}

Knowing the format, you can read in the first '{', then read each character to code mapping. The mappings are delimited by ", ". Be careful to allow for ',' and ' ' as characters.

As each character code mapping is read, create the tree. For example, when the first pair is read in, you will create a root node, a left child node, and a left grandchild node with the ' ' character.
For the 'a' character, the root will already be created, but you will need to create the right child, and a left grandchild under that for the mapping for the 'a'.

Continue this process until the codes map tree has been fully populated.

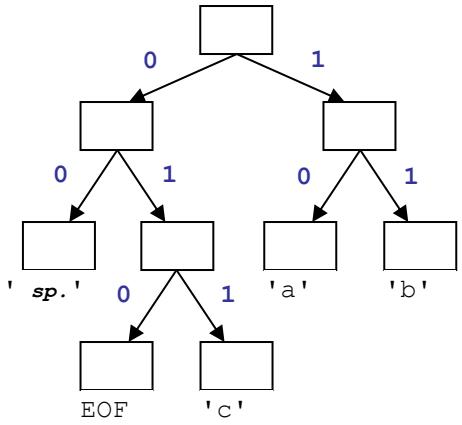
Step 2

Decode the file.
Read each bit from the compressed file one at a time and use this bit to traverse the Bitwise tree. If the bit is a 0, you move left in the tree. If the bit is 1, you move right.
Once you reach a leaf node you output the character that is stored there.

For example, suppose we are asked to decode a file containing the following bits.

`101100101100011101101000`

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



- First, we read a `1` (right), then a `0` (left). We reach `'a'` and output `a`. Back to the root.
- We read a `1` (right), then a `1` (right). We reach `'b'` and output `b`. Back to the root.
- We continue this process until we reach an `EOF` character (in this case `010`) and stop. This will keep any padded zeros at the end of our file from corrupting our decoded file.

Deliverables

In this assignment, you will create a class BitTree to both compress the source file and decode the compressed file ***back*** into a readable text file. All functional methods, data structures, and file output will occur in your BitTree class. Do not rely on static methods. Use a test class called BitsMain with a static main method to create your input file names, then make a BitTree object, passing that complete file name to the BitTree constructor.

All input files must be accessed from a folder named input and you must write all maps and compressed files to a folder named output. Your code should access them using a path that starts with ./input/ and ./output/ respectively. Your code should use try/catch to handle any potential file I/O exceptions. You may not use throws in your method headers.

Scalability is important. Your solution must support large files.

You will also create a class called BitTreeNode where each node stores information about one character. It will hold left and right pointers, a single character, and a freqency integer. Note that your BitTreeNode class will need to construct leaf nodes differently than it constructs branch nodes, so 2 constructors will be needed. A key attribute of your BitTreeNode class is that it must implement the Comparable interface. The reason for this is so you can make a priority queue that will sort the node elements for you.

The other contents of the BitTreeNode class are up to you, but it should not perform a large share of the overall algorithm.

Turn in a zipped folder called BitTree.zip. It must include java files named BitsMain.java, BitTree.java, and BitTreeNode.java.

Development Strategy and Hints

We suggest that you first focus on building your Bitwise tree properly from the given map of character counts. Then work on creating the map of char->String encodings from your Bit tree. Then work on using your encodings to compress files and calculate your compression ratio. Consider writing a toString method in your BitTreeNode so you can easily print nodes or a priority queue of nodes.

Implementation Details

Your BitsMain class must be easily edited for different input file names and use proper folder structure.

Your BitTreeNode class must have simple data structures and pointers declared as fields. It should implement the Comparable interface. It should have the following public constructors and methods:
public BitTreeNode(Character chr, int freq)
public BitTreeNode(BitTreeNode left, BitTreeNode right)
public int compareTo(BitTreeNode other)
In this method you should decide how to compare 2 nodes and return proper integer values so you can properly build your priority queue.
public String toString()

Your BitTree class must have all major data structures declared as fields. It should have the following public constructor and methods:
public BitTree(String fName, String fExt)
In this constructor you are passed an input file name and file extension. All fields should be initialized. Passing the name and extension seperately make it easier to construct your output files.
public void encode()
In this method you should implement steps 1-5 of the algorithm for encoding as detailed above. Your encode method should write the histoMap file using extension

".bwt.histoMap.txt". Your encode method should also write the Bitwise code map file using extension ".bwt.codesMap.txt".

Use HashMap for the histoMap and codesMap and use the HashMap.toString() to generate the output file. This is necessary so that the output format will match what the test cases expect.

public void compress()

In this method you should implement step 6 of the algorithm by reopening the input file and use your Bitwise code map to write a Bitwise-compressed version of this data to the output file using the extension ".bwt.bits.txt" (string version) and ".bwt.bits" (actual bits version)

public void decompress()

In this method you should read the compressed file and your Bitwise code map, then write a decompressed text file. Use the "`.bwt.decoded.txt`" label in your final output file name. Note that you will need to read in the map file and rebuild your binary tree structure before decoding can begin. You will need your `codesMap` file for the tree reconstruction.

public void print()

This method should print the name of your input text file, the input file size, the compressed file size, and your compression ratio to the console.

Filename: <filename> File Size: <size> Compressed File Size: <size> Compression Ratio: <ratio>

public int inputFileSize()

This method returns the number of characters in the input file. Does not include EOF.

public int compressedFileSize()

This method returns the number of bits in the input file. Does include EOF.

public double compressionRatio()

This method returns compressedFileSize/inputFileSize as a double.

You may have additional methods, so long as they are private. Methods that traverse your tree should be implemented recursively using public/private method pairs, whenever practical.