LZW compression, named after its three founders, Abraham Lempel, Jacob Ziv, and Terry Welch showcases a modern means of shrinking a file, despite being discovered shortly after the personal computer. Despite its rather long existence, it still impacts contemporary technology through the GIF, TIFF, and PDF file formats. Its simple and straightforward implementation, along with its lossless format, has assisted in its success throughout the years.

First, LZW compression relies heavily on an underlying symbol table. The symbol table’s implementation is trivial, but the quickest access and insert times possible are highly recommended. This table is initialized with all 256 (28 because of 8-bits) ASCII characters; these will be the initial “substrings” for the current uncompressed file. Then, the program loops through the file. For every iteration, the beginning of the file is scanned for the longest codeword contained within the codebook, or symbol table; this is called the longest prefix. So, for example, if the file began as “a little boy once was […]”, with the initialized symbol table, the longest prefix in this case would be “a”. Then, that prefix, concatenated with its neighbor character to the right, are placed into the symbol table with its appropriate index; in this case, that new codeword is “a “ with the codeword value 257 because a space is the next character in line. Next, the file, or ‘input’ string, is shortened and moved past the encoded codeword (in this situation, “a”) by utilizing the substring() method in Java. Now, the input string reads “ little boy once was […]”. One critical step in this compression is actually writing out the compressed data to a file. Because the codeword found in this example is “a”, which happens to be the value 97 in ASCII (the initialized codebook), that value is written to a new file in binary with N-bits. In a fixed-length encoding, where N is the number of encoded bits, the codebook size never exceeds 2N, and every codeword written to the compressed file is N-bits. However, in variable-length encoding, one may start with 8-bit codewords and move up to 16-bits, allowing for 216 (65,536) codewords.

Due to its lossless quality, expansion returns the file back to its original contents; this is key in files such as PDFs as mentioned earlier. When expanding, read in N bits at a time (expansion must be fixed-length or variable-length based on what was used in compression), adding codewords in a similar manner as compression. If the value 97 was read in (going back to the compression example), then go to the value in the symbol table at index 97 (“a”), write it to the uncompressed file, and append the first character of that string to the last string read in. In the first iteration of expansion, the empty string will be concatenated (since nothing had been read in previously). But, in the example with “a”, in the second iteration, the program would read in “ “, write it to the uncompressed file, then append it to “a” , producing “a “, and insert it into the codebook. Now, that new code, or longest prefix, can be used later on in the expansion of the file. In this project, these basic principles will be expanded to support a few more forms of compression and expansion.

For example, variable-length encoding was mentioned earlier. Currently, LZW.java uses fixed-length encoding of 12-bit codewords. To improve on this, MyLZW.java will begin with 8-bits and progress to 16-bit codewords. Variable-length encoding showcases its usefulness with an initially smaller codebook, allowing for fewer bits written and better compression early on but having the capacity to hold a larger amount of codewords which boosts compression later. However, when a file contains a significant amount of random noise, with no patterns whatsoever, LZW’s symbol table can end up being largely wasted space. This proposed change will be tested and examined on its practical utility. The final couple of changes to LZW.java are a reset and monitor mode. In reset mode, when the codebook fills up (contains 216 codewords), then reset the entire codebook. This helps by allowing new codewords to be introduced as one iterates through the file, potentially compressing the file even further, but it clears out the already added codewords, which may remove some of the most common words/phrases in the file, worsening compression. For monitor mode, compress normally, but when the codebook fills up, monitor the compression ratios of the file. Take the old compression ratio, which was calculated before the symbol table filled up, and divide it by a new compression ratio, which is calculated for every iteration while the table is full. If this result is greater than a specific threshold, in other words the old compression method would result in a significantly larger file than just resetting the codebook, reset the codebook. These three major proposed changes are simple, but have the potential to offer dramatic improvements to LZW.java.

Implementing these modifications, despite rather tiny amounts of code, is complex. First step, start small and aim for the variable-length encoding, then advance to completing the modes. When implementing variable-length encoding, the number of bits output to a file changes as the algorithm continues. Because the number of bits changes, so does the size of the codebook. So, the first immediate change is ridding of the ‘final’ keywords in LZW.java for both the variables W (bit-length) and L (size of codebook); their values now dynamically update throughout the program. For example, W starts at the value eight and every time the codebook fills up, will increment by one, until it reaches the value sixteen; every time W updates, L = 2W, indicating the max size of the codebook. Next, to make the program more modular, the resetting/initialization functionality for compress() and expand() will be moved to their own methods; their contents came with LZW.java, so no new code will be added, just abstracted away.

Now, for the first large chunk of additional functionality, the compress method. The program currently checks if the codeword value is less than the size of the codebook; in other words, it checks if there is still room to add another codeword. If so, it places the calculated longest prefix into the symbol table. Because MyLZW.java will use variable-length encoding, when the codebook is full (that previous if statement is false), the program must check if W is less than the max bit width (16). If it is, there is still room to expand the codebook for our liking, so increment W, recalculate L, and place the new codeword into the table. But what happens if W is equal to the max bit width? That is where the reset and monitor modes come into play. If the codebook has reached its absolute maximum capacity and W equals sixteen, then check the mode. If the mode is reset, then instantiate a new symbol table object, call the method initializing the codebook, reset all of the necessary values (W to 8, L to 28, and the codeword value to 257). Lastly, place the new codeword into the newly formed codebook. Then, the program continues as usual until the table fills up once again. As for monitor mode, when the codebook is completely full with 216 codewords, calculate the amount of uncompressed data in bits divided by the amount of compressed data, this is the current compression ratio of the file. Then, for every iteration after that, this program will check an else if statement to determine that the program is in monitor mode. When that condition is inevitably true, calculate a new compression ratio, constantly updating the number of bits read in and the number of bits written to the compressed file. Then, divide the old compression ratio by the new compression ratio in every iteration. If that result is greater than 1.1, indicating that the old codebook produces a significantly larger file size than a newly reset table would, reset everything in a similar fashion as reset mode, adding a new codeword immediately. Then, the process will restart with an essentially newly created codebook. These operations are done for every iteration; the old compression ratio is calculated for every iteration when the codebook is not full, the new compression ratio is calculated for every iteration while the codebook is completely full, and the ratio of the two ratios is taken for every iteration that the codebook is full as well. In order to calculate the amount of data read and written, for every longest prefix found in the input string, multiply its length by 16 (Java chars are UTF-16) and add that onto a variable storing the uncompressed data size; this represents the size of the prefix in bits. As for the written data, just increment a variable by the value of W, which is the number of bits written to the file for every iteration. Finally, the compression function should be finished.

The second significant section of changes occurs in the expansion method. Since the compress() function stores the compression mode as the first character in the file, the expansion function must immediately read in the character to determine if it has to reset or monitor in the future. Overall, expansion is nearly identical to compression in MyLZW.java. Maintain a value i, already supplied by LZW.java, which keeps track of the current codeword value. Then, for every value j read in from the uncompressed file, find the codeword corresponding to that number at table[j], and write it to the uncompressed file. Similar to compression, if the value is within the bounds of the codebook (max of 216 words), then read in the next value, find its String representation in the codebook, and append its first character to the previously gathered codeword; add this joined result to the codebook similarly to what happened in compression. As for reset mode, it follows nearly the same steps as compression; create a new String array and initialize it to all of the ASCII characters, reset W to the minimum bit width (8), along with L to 28, and i to the next index of the table not yet filled (257). And monitor mode, for every iteration when the codebook is not filled, calculate the old compression ratio, otherwise, calculate the new compression ratio, keeping track of the data read in as well as the data written using the same procedures as compression. Because expansion adds codewords in the opposite manner as compression (the former appends the first character of the current codeword to the previously encountered codeword while the latter adds the next character in the file onto the current codeword; think back to the original example with “a” and “a “), the condition checking if i is greater than the size of the codebook must occur before the if statement checking if i is less than L, allowing for new codewords to be immediately added to the codebook if a reset occurs in reset or monitor mode.

Finally, a small change must occur in the main function. For compression, since the user has the ability to run it in “do nothing”, “reset”, or “monitor” mode, an if statement must be created to parse that command-line argument, check that it is a valid mode, and store it in a global variable to be utilized by compress() method later on.

As one can see, this implementation of MyLZW.java does not imply massive additions of code, but rather precise and articulate tiny bits of code. With a small mistake, compression may corrupt or drop off several bytes of data, which in turn confuses the expansion function, effectively nullifying the lossless quality of LZW compression. And this is not in merely one mode, but rather three separate modes (do nothing, reset, monitor), so the implementation requires detail on all occasions. With the aforementioned proposed modifications, this altered version of LZW.java will allow for greater adaptability in codewords through variable-length encoding, as well as more efficient compression options by reset and monitor mode, both of which will be ensure all codewords are given an equal opportunity to be entered into the codebook and the compressed file’s size is the smallest it can be.