Lossless compression algorithms have evolved greatly since they were first introduced in the form of Shannon-Faro and Huffman coding. Among the earliest of evolutions from Huffman coding came lossless LZW compression. Many compression algorithms in use today were derived from the central idea used in LZW compression, making it an extremely important algorithm for understanding data compression.

The general idea behind LZW compression is to compress the data in a file by representing variable-length portions of input data, using fixed-length codewords. Therefore, the implementation of the original LZW compression algorithm uses a fixed codeword length of 12 bits. Being that this uses a constant codeword width we can calculate the fixed size of our codebook to be 4,096 bits, which is taken from the idea that the size of the codebook should always be 2^(W), where W is the size (in bits) of our codewords. It is extremely crucial to realize that prior to compressing the file, the first 256 codewords added to the codebook will represent each individual extended ASCII character. This idea of a codebook, is really just a symbol table implementation, which originally maps ASCII characters (keys) to their distinct ASCII outputs (values). In the case of LZW, this symbol table is implemented using a Ternary Search Trie, because for LZW every prefix of an input-substring key is also a key. Once the codebook is initialized with these 256 ASCII characters compression is ready to begin.

In order to start the LZW compression process our program must read from the file to be compressed and do so until there are no unscanned input characters remaining in the original file. This works by reading the remaining bytes of data from standard input and returning these remaining bytes as a string. Our program then enters in a loop until there is no more bytes of data to be read from this string. During each iteration of this loop, we must find the longest string currently contained in symbol table, which serves as a prefix to the unscanned input. Once this longest prefix is obtained the LZW program must output the codeword value that is associated with this longest prefix, to a binary file, using a width of W bits.. After this codeword is outputted it takes this longest prefix, adds the next character in the file, and adds the result of this to the symbol table with a new codeword. One main advantage of LZW compression is that this symbol table doesn’t have to be stored anywhere for compression, because it can be automatically regenerated by expansion.

To expand our compressed file we first declare a string array to be the size of our codebook from compression, which in this case is L. This string array acts as the symbol table for expansion, utilizing an instance of perfect hashing to map codewords to output strings. Before beginning expansion our algorithm must initialize the symbol table (codebook) to all single characters, which maps ASCII values to their character. Once this codebook is initialized we read codewords of width W, which in this case is twelve, from the file generated during compression. After this compressed file is read we loop through the contents of the file until the end of the file is reached. Throughout each iteration of this loop our program reads the next codeword from the file, looks for this corresponding pattern in the codebook, and outputs the pattern to the uncompressed file. Each time after doing this, if the codebook has not yet been filled, expansion must add the previous pattern plus the first character of the current pattern and add this into the codebook. As larger files begin to use LZW compression, the use of fixed width codewords seems to be very sub-optimal, as our codebook would fill in a very short amount of time. Along with our codebook filling up early, it would be filled with relatively small patterns (strings), which greatly reduces the effectiveness of LZW compression. This is because LZW compression works best as it produces longer patterns (strings) from its codewords.

Being that the use of fixed-width codewords greatly reduces the performance of LZW compression, my first step to implementing MyLZW is to make it work with variable-width codewords. As stated previously the width of the codewords, is represented by W. In the previous LZW the codeword width was a constant 12 bits, but for variable-width codewords the first step is to set the original codeword width to 9 bits. Along with changing W to 9 bits we also have to change the codebook size (L) to 512 in order to uphold the rule that our number of codewords will always be 2^W. The next step I would take to implementing variable-width codewords would be in the compress method.

In the compress method if the current codebook has been filled, meaning that it has reached its capacity of 512 codewords, I would then increment W by one and double the size of the codebook (L). After this you would have a new codeword width set to ten and new codebook size set to 1,024. Each time the codebook fills up to L, I would repeat this process of incrementing W and doubling L, until W is equal to sixteen and L is equal to 65,536. Once this codebook has a codeword width of 16 and codebook size of 65,536, this means that our codebook has been completely filled. From here, we have three options for how to compress the rest of the file. First, we could “do nothing” and just use the existing codebook to complete compression. Second, we could “reset” the codebook to generate new sets of codewords for unscanned input remaining in the file. Finally, we could “monitor” the compression ratio and if it exceeds 1.1 we reset the codebook, otherwise we just stick with the existing codebook to carry out our compression.

Whatever mode the user wants to use for compression, must also be automatically used for expansion. To handle for this case, in my main method I would determine the value of argument 1 given on the command line and immediately write this as the first character to the binary output file, which is where we will be storing our compressed data (codewords). I would also pass whatever mode they chose as a parameter to my compress method. Passing in whatever mode the user chose allows me to then decide what operations to perform if the codebook were to completely fill, note that this case will only ever occur when the codebook has reached max capacity and max codeword width. Using a switch-case statement I would then handle the user’s compression preference by using the mode as the switch condition (mode would be n, m, r). If the user we to enter a ‘n’ at the command line, then my program would break from the do nothing case and continue to use the already generated codebook to complete compression.

In the case where the user enters ‘r’ at the command line, we know the user wants to reset the codebook and start generating new codewords if it were to fill. Therefore, in this case we have to set our codeword width (W) and codebook size (L) back to their original values of 9 and 512. However, before we do this we first have to instantiate a new symbol table instance so that we can store these newly generated codewords in the codebook. After instantiating a new symbol table instance I would add the individual ASCII characters to this symbol table, because this is always the first step when starting with a new codebook. Once we do both of these things we can set W and L back to their original size as mentioned previously. It is also important the we update the value of the next codeword to be added to 257, this way we will start adding codewords at the correct spot as we go through the loop again. Finally, before breaking from the reset case it is necessary to add the new codeword into this newly instantiated symbol table (codebook). In the situation that our compression were to completely fill the codebook to its max again, we would again reset just like explained previously. The case of monitor mode is intertwined with the functionality of both reset and do nothing mode.

With monitor mode, it will have the same function as do nothing mode if the compression ratio staya under 1.1. In this case, I would just check if the compression ratio is greater than 1.1 and as long as it isn’t then we would continue to use the full codebook as is, just like do nothing mode. However, in the situation where the compression ratio exceeds 1.1 we must reset the codebook just like we did for reset mode. This is the easy part of the implementation for monitor mode, the tricky part is figuring out where to measure the compressed and uncompressed size, as well as where to calculate both the past and current compression ratios. Ultimately, to calculate the compressed size for each iteration of the while loop you must add the size of the compressed data plus the size of W, which makes a lot of sense because the size of our compressed data will just be the amount of W bit codewords we have stored. To obtain the uncompressed size of the data we calculate the length of the string that is the longest prefix of and multiple this by 8(# of characters that is longest prefix \* 8). Once we have both these sizes and the codebook is full we calculate the old ratio by dividing the uncompressed size by the compressed size for each time the codebook does not fill. The new ratio is to be calculated in the same manner but only when the codebook completely fills. Combining these two ratios into a ratio of ratios we get the compression ratio for the current iteration of the loop. We then utilize this compression ratio for deciding whether to reset or do nothing from monitor mode, as described previously.

Throughout this explanation so far I have only really been talking about implementing these changes for compression, you may be wondering what about expansion? I went into great detail on compression because ultimately the process of to implement these extra feature in compression and decompression are very similar in concept, yet produce opposite results. Much like I did with compression I will first focus on implementing variable width keywords into the original expand method. To do so, as the first condition, I would check if our codebook is full (greater than or equal to L) and if our codeword width (W) is less than 16. If both of these conditions check out then we know that we have a full codebook, but it is not at max capacity. To incorporate variable width codewords here you would once again just increment W by one and increase the size of the codebook by double its original size. As long as W is less than 16 and the amount of codewords is greater than the size of the codebook we will continue to add variable-width codewords until the codebook reaches max capacity of 65,536. When our codebook reaches max capacity we than have three options, to do nothing, reset, or monitor. However, how this mode of decompression is decided is dependent completely on the mode chosen during compression. Recall earlier in the paper I talked about how in my main method I immediately wrote the mode selected to the compressed file. This is then grabbed automatically by expansion, by grabbing the first character from the compressed file as the first step. Once this mode had been obtained from the file it once again can be used to control a switch case statement, deciding which mode the user selected during compression.

To implement do nothing mode for expansion I would just break from the do nothing case allowing expansion to run like normal. Now for implementing reset mode this is almost identical to the implementation for compression, except that we instantiate a new string array as our symbol table as opposed to instantiating a new ternary search trie as our symbol table. Finally to execute monitor mode for expansion, I would calculate the compressed size and uncompressed size in the same manner that I did for compression. To calculate the compression ratio I would obtain the value for the old ratio whenever the codebook size is increased but also in the case that the codebook does not fill completely. Now to get the new ratio I would only calculate this when the codebook reaches max capacity and we are in monitor mode. Using these ratios I would then calculate the total compression ratio to use for monitor mode’s reset.