Basic part:

```
void initialize_table() {
    table[4095] = { -1, EOF };
    for (next_entry = 0; next_entry < 256; ++next_entry)
        table[next_entry] = { -1, next_entry };
}
```

Initialize the first 256 entries and the last entry.

Compression part in main():

```
/* ADD CODES HERE */
            initialize_table();
            for(int i = 0; i < no_of_file; ++i){
                FILE* input_file = fopen(input_file_names[i], "rb");
                compress(input_file, lzw_file);
                fclose(input_file);
            }
            //flush the remaining 8 bits
            write_code(lzw_file,0,8);
            fclose(lzw_file);
```

First, initialize the LZW table before compressing the input files. Then, iterate through the list of input files and compress them one by one. After that, write 8 bits of zeros to the output file to ensure that any remaining bits in the output buffer are flushed.

Decompression part in main():

```
/* ADD CODES HERE */
            initialize_table();
            char * files = strtok(output_file_names, "\n");
            for(int i = 0; i < no_of_file; ++i){
                FILE* output_file = fopen(files, "wb");
                decompress(lzw_file, output_file);
                fclose(output_file);
                files = strtok(NULL,"\n");
            }
            fclose(lzw_file);
            free(output_file_names);
```

First, set up the LZW table. Then, strtok() to extract file names from the output_file_names. Decompress it one by one.

Compress():

```
struct Entry {
    int prev_char;
    int current_char;
};

Entry table[4096];
int next_entry;

void compress(FILE *input, FILE *output)
{
    /* ADD CODES HERE */
    int prev, cur;
    // read the first character from input file
    prev = fgetc(input);
    // return if file is empty
    if (prev == EOF) return;

    // loop until end of input file
    while ((cur = fgetc(input))!= EOF){
        // find if the char is in table
        int char_index = -1;
        // search the table for the current combination of previous and
current char
        for (int i = 0; i < next_entry; ++i) {
            if (table[i].prev_char == prev && table[i].current_char ==
cur) {
                char_index = i; // current index of entry e
                break;
            }
        }
        // if not found, output the code for the previous char and add
them to the table
        if (char_index == -1) {
            write_code(output, prev, CODE_SIZE);
```

```
            table[next_entry++] = { prev, cur };
        }
        // set the previous char to the current char if not found.
Otherwise, set the previous char to the found pair in table.
        prev = char_index != -1 ? char_index : cur;

        // set the next entry to index 256 if running out of entries
        if (next_entry == 4096) {
            next_entry = 256;
            table[4095] = { -1, EOF };
        }

    }
    // output the remaining chars and EOF
    write_code(output, prev, CODE_SIZE);
    write_code(output, 4095, CODE_SIZE);
}
```

It starts by reading the first character from the input file and setting it as the **prev**. If the input file is empty, the function returns.

Then, it reads the next character until the end of the file. It searches the table to see if the combination of previous and current chars already exists. If not found, outputs the code for the previous char and adds the pair{**prev, cur**} to the table. It sets **prev = cur** if the pair was not found in the table. Otherwise, **prev = char_index**(the index of the pair in the table). Once the table is full, it resets the table and **next_entry** to index 256.

Finally, when the EOF is reached, outputs the code for the last **prev** character and EOF(4095).

Decompress():

```
void write_code(FILE* output, int code) {
    vector<int> stack;
    while (code != -1) {
        stack.push_back(code);
        code = table[code].prev_char;
    }

    for (auto i = stack.rbegin(); i != stack.rend(); ++i) {
        fputc(table[*i].current_char, output);
```

```
    }
}

void decompress(FILE *input, FILE *output)
{
    /* ADD CODES HERE */
    int code, temp;
    int cur_code, prev_code;

    // read and write the first code
    cur_code = read_code(input, CODE_SIZE);
    write_code(output, cur_code);

    // loop code until 4095 (EOF)
    while ((code = read_code(input, CODE_SIZE)) != 4095) {
        prev_code = cur_code;
        cur_code = code;
        // find the index of the current code in the table
        // if not, use the previous code instead.
        int code_index = cur_code < next_entry ? cur_code : prev_code;
        while (table[code_index].prev_char != -1) {
            code_index = table[code_index].prev_char; // code_index
points to the index of the last character
        }
        // write the current code to output file and update temp
        write_code(output, cur_code < next_entry ? cur_code :
prev_code);

        // if current code >= next entry, write the temp to output
        temp = table[code_index].current_char;
        if(cur_code >= next_entry) fputc(temp, output);

        // add new entry to the table
        table[next_entry++] = { prev_code, temp };

        // set the next entry to index 256 if running out of entries
        if (next_entry == 4096) {
            next_entry = 256;
```

```
            table[4095] = { -1, EOF };
        }
    }
}
```

It reads and writes the first code from the input first. The while loop runs until it reads the EOF (4095). Within the loop, **prev_code** is set to **cur_code** and **cur_code** is set to **code**. Then, it checks whether the current code is in the table by finding its index in the table. If not, the previous code and the last character are written to the output file using the **write_code()** function which is defined by myself. It traverses the table backwards, adding the index of each character to the stack until it reaches EOF in prev_char. After writing to output file, A new entry is added to the table with the previous code and the last character. If the current code is greater than or equal to the next entry, the last character for the current code is written to the output file. Again, if the table is full, it is reset to the initial state.

Sample Commands:
cl lzw.cpp
lzw -c *output.lzw test_files/Windows.txt test_files/CSE.txt test_files/lena.bmp*
lzw -d output.lzw

Enhancement part:

LZW with unordered_map as dictionary(support different file formats):

Global variables:

```
int next_entry;
unordered_map<string, int> dict_compress;
unordered_map<int, string> dict_decompress;
```

For the compression and decompression, the data structure has changed into using unordered map as dictionary for efficient lookup with key-value pairs, since unordered_map is implemented using Hash Table internally.

Libraries:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <ctype.h>
#include <vector>
```

```
#include <iostream>
#include <unordered_map>
#include <string>
```

Function defined:

```
void initialize_dict(char c) {
    next_entry = 256;
    if(c=='c'){
        for (int i = 0; i < next_entry; i++) {
            dict_compress[string(1, (char)i)] = i;
        }
    }
    else {
        for (int i = 0; i < next_entry; i++) {
            dict_decompress[i] = string(1, (char)i);
        }
    }
}
```

Compression:

```
void compress(FILE *input, FILE *output)
{
    int cur, prev;
    prev = fgetc(input);
    if (prev == EOF) return;
    //first char
    string str = string(1, (char)prev);
    while ((cur = fgetc(input)) != EOF) {
        string str_plus_cur = str + (char)cur;

        // str+cur is in the dictionary
        if (dict_compress.count(str_plus_cur))
            str = str_plus_cur;

        else {
            // output the code for str
```

```
                write_code(output, dict_compress[str], CODE_SIZE);
                // add str+cur to the dictionary
                if (next_entry < 4096) {
                     dict_compress[str_plus_cur] = next_entry++;
                }
                // set str to cur
                str = string(1, (char)cur);
           }
           // reset dictionary if it is full
           if (next_entry == 4096) {
                dict_compress.clear();
                initialize_dict('c');
           }
     }
     // output the code for the last str
     write_code(output, dict_compress[str], CODE_SIZE);
     // output the end-of-file marker
     write_code(output, 4095, CODE_SIZE);
}
```

The whole process is almost the same as the basic part besides the loop.

Process of the loop:

1. find longest matching sequence in the dictionary
2. write the code for the matched sequence to the output file
3. add the next char to the dictionary as a new sequence

Original compress(): time complexity = O(n) where n is the size of input file

Space complexity= O(n*e) where e is the size of entry

Enhanced compress(): time complexity = O(n), Space complexity= O(e)

Decompression:

```
void decompress(FILE *input, FILE *output)
{
     int prev_code = -1, cur_code;
     string str;
     while ((cur_code = read_code(input, CODE_SIZE)) != 4095) {
          // decode the current code
          string s;
          // output the string if the code is in dictionary
```

```
          if (dict_decompress.count(cur_code)) {
               s = dict_decompress[cur_code];
          }
          // current string == previous string + its first character
          else if (cur_code == next_entry) {
               s = str + str[0];
          }
          fwrite(s.data(), 1, s.length(), output);
          // new entry = concatenation of the previous with the first char of
the current str
          if (prev_code != -1 && next_entry < 4096) {
               dict_decompress[next_entry++] = dict_decompress[prev_code] +
s[0];
          }
          // update previous code and str
          prev_code = cur_code;
          str = s;
          // start over if the dictionary is full
          if (next_entry == 4096) {
               dict_decompress.clear();
               initialize_dict('d');
          }
     }
}
```

The whole process is almost the same as the basic part besides the loop.

Process of the loop:

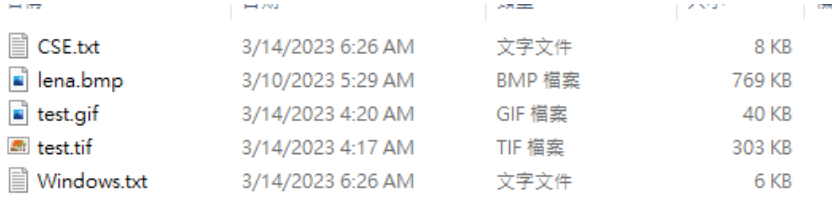
1. read the code from the file.
2. if the code is in the dictionary, output the corresponding string to the file.
3. if the code is not in the dictionary, create a new string by concatenating the last output string with the first character of the last output string, output this new string to the file, and add it to the dictionary.
4. If the dictionary becomes full, reset and clear it.

Original decompress(): time complexity = O(n) where n is the size of compressed file
Space complexity= O(n*e) where e is the size of entry
Enhanced decompress(): time complexity = O(n), Space complexity= O(e)

Also, the code can also compress and decompress some image formats like GIF and TIFF. In decompression, treating the input data as a sequence of bits and converting it back into codes.

Sample input and output:

Original input files in *test_files*



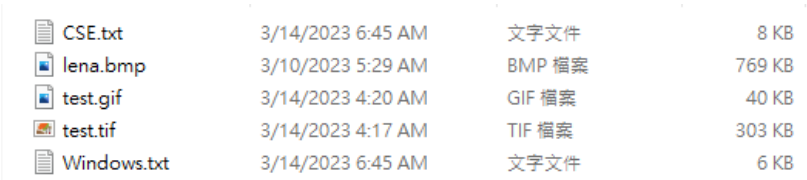
>cl lzw_enhancement.cpp

Compression:

>lzw_enhancement -c output_enhancement.lzw test_files/Windows.txt test_files/CSE.txt test_files/lena.bmp test_files/test.gif test_files/test.tif

Decompression:

> lzw_enhancement -d output_enhancement.lzw



Run-length encoding (RLE) implementation (support different file formats):

This is also a form of lossless data compression algorithm. However, RLE works best when there are long runs of repeated characters in the file, such as image file. If the input file has random or evenly distributed characters, the algorithm may not result in efficient compression.

Compression and decompression code:

```
void RLEcompress(FILE* input, FILE* output) {
    int prev = fgetc(input);
    if (prev == EOF) return;
    int count = 1;
```

```
    int cur;
    while ((cur = fgetc(input)) != EOF) {
        // max count = 4095 - 1 (12 bit)
        if (cur == prev && count < 4094)
            ++count;

        else {
            write_code(output, prev, CODE_SIZE);
            write_code(output, count, CODE_SIZE);
            prev = cur;
            count = 1;
        }
    }
    write_code(output, prev, CODE_SIZE);
    write_code(output, count, CODE_SIZE);
    //EOF
    write_code(output, 4095, CODE_SIZE);
}

void RLEdecompress(FILE* input, FILE* output) {
    int cur;
    while ((cur = read_code(input, CODE_SIZE)) != 4095) {
        int code = cur;
        int count = read_code(input, CODE_SIZE);
        for (int i = 0; i < count; ++i) {
            fputc(code, output);
        }
    }
}
```

For the compression, Let's say the input is `WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWWWWWWWBWWWWWWWWWWWW` , then the compressed data will be `W12B1W12B3W24B1W14……(prev)(count)……` However, the maximum count for every char will be 4094, since it is the maximum of 12-bit number – 1 (EOF is 4095).

For the decompression, it just prints out the code ***count*** times.

Time complexity for both functions: O(n)

Space complexity for both functions: O(1)

For RLE compression and RLE decompression, the flag is ***-RLEC*** and ***-RLED***.

This has been handled in main():

```
...
if (strcmp(argv[1], "-RLEC") == 0) {
              /* compression */
              lzw_file = fopen(argv[2], "wb");

              /* write the file header */
              input_file_names = argv + 3;
              no_of_file = argc - 3;
              writefileheader(lzw_file, input_file_names, no_of_file);

              for (int i = 0; i < no_of_file; ++i) {
                   FILE* input_file = fopen(input_file_names[i], "rb");
                   RLEcompress(input_file, lzw_file);
                   fclose(input_file);
              }
              fclose(lzw_file);
         }else
         if (strcmp(argv[1], "-RLED") == 0) {
              /* decompress */
              lzw_file = fopen(argv[2], "rb");

              /* read the file header */
              no_of_file = 0;
              readfileheader(lzw_file, &output_file_names, &no_of_file);

              char* files = strtok(output_file_names, "\n");
              for (int i = 0; i < no_of_file; ++i) {
                   FILE* output_file = fopen(files, "wb");
                   RLEdecompress(lzw_file, output_file);
                   fclose(output_file);
                   files = strtok(NULL, "\n");
              }
              fclose(lzw_file);
              free(output_file_names);
```

```
        }
...
```

Testing:

> cl lzw_enhancement.cpp

> lzw_enhancement -RLEC output_enhancement.lzw test_files/Windows.txt test_files/CSE.txt test_files/lena.bmp test_files/test.gif test_files/test.tif

>lzw_enhancement -RLED output_enhancement.lzw