README

It conists of two main programs:

lzw.cpp (lzw compression of the basic requirements)

lzw_c2.cpp (lzw compression of the colored bmp images)

Note: Before running lzw_c2.cpp

1) Put it with bmp.h and bmp.cpp

2) Compile the program in the command prompt of visual studio in the following command line:

> cl lzw_c2.cpp bmp.cpp

Basic part analysis

For lzw.cpp, it is same as the requirements in the basic part. The input is totally same as the requirement as well, ie:

Compression:

> lzw –c <lzw filename> <a list of files>

Decompression:

> lzw –d <lzw filename>

For both compression and decompression commands, it will output the time used for the execution of the program.

Data structure analysis

In this cpp file, I have implemented two data structure to perform lzw compression and decompression separately, which is hash table and array.

In compression, I implement hash table with separate chaining as dictionary.

```
class Hash {

public:
    // size of hash table
    int size;

    // declare table of linked lists of two values: string and code
    list< pair< string, unsigned int> > * table;

    // declare hash table of required size
```

```
    Hash (int size);

    // insert elements in the hash table : Best : O(1) , Worst : O(n)
    void insert(string data, unsigned int count);

    // find the string with corresponding code : Best : O(1) , Worst :
O(n)
    unsigned int find(string data);

    // initialize the hash table with 256 ascii characters inserted : O
(n)
    void initialize();
};
    Hash :: Hash(int size){
        this->size = size;
        table = new list< pair<string , unsigned int > >[size];
    }
    void Hash :: insert(string data, unsigned int count) {
        unsigned int key;
        // deal with the probelm that a signed char has value between -
128 to 127
        if (data[0] >= 0)
            key = data[0];
        else key = data[0] + ascii_size;

        table[key].push_back(make_pair(data, count));
    }
    unsigned int Hash :: find(string data) {
        unsigned int key;
        // deal with the probelm that a signed char has value between -
128 to 127
        if (data[0] >= 0)
            key = data[0];
        else key = data[0] + ascii_size;

        // search the string and return the respective code, if can't,
return npos
        list < pair < string , unsigned int> > :: iterator i;
```

```
        for ( i = table[key].begin(); i != table[key].end();i++) {
            if ( (* i).first == data) {
                return (* i).second;
            }
        }
        return -1;
    }
    void Hash :: initialize() {

        // free all the memory in the table if it is not empty
        for (int i = 0; i < ascii_size; i++) {
          if (!table[i].empty()) table[i].clear();
         }

        // initialize the table with 256 ascii inserted
        string c = "";
        for (int i = 0; i < ascii_size; i++){
            c = "";
            c += i;
            table[i].push_back(make_pair (c ,  i));
        }
    }
```

Each table is a linked list with two elements, one is the string, another is the corresponding code. So the hash table only needs size of 256. When the string is found, the corresponding code stored could be output. The time complexity of the searching the string in the dictionary is only O(1) for the best case and O(n) for the worst case. As we only need to obtain the corresponding ascii code of the first character to locate the row of dictionary. Then only a simple scan through the row of linked list to obtain the result. Adding new entry in the hash table is similar to searching, so it has same time complexity. The space complexity of hash table is flexible, it will only occupy the size of an array of 256 (ascii code) at first. Later it will change as the insertion of new wordings. Only in the worst case (ie. all the words of the file is the same characters), the space complexity will be O(n). Overall, hash table with separate chaining has low space complexity and high speed in searching and inserting. As a result, the speed of the lzw compression which requires frequently searching and inserting is very high when hash is implemented.

For the lzw decompression, I implement array of string of dictionary size 4096 as data structure.

```
string dict[dict_size];

// initialization of array dictionary with 256 ascii inserted
void initializeDict(string * dict) {
    int i;
    for (i = 0; i < ascii_size; i++) {
        dict[i] = i;
    }
    for (; i < dict_size; i++) {
        dict[i] = "";
    }
}
```

I use the index of the dictionary as the code which represents the string. As the code is read from the archive in increasing order. For array structure, the searching can be done by comparing the current code size. If it is current code is smaller than the array index position. Then the dictionary does not have that word. The searching and insertion time complexity is O(1) in both the best and the worst case. So the all the lzw decompressions operation are only O(1), which means it only nearly uses the time of running through the whole compressed file, which is O(n). But the space complexity of array is O(n). It will use up all the dictionary whenever any string is starting input or not. Overall , using array in decompression is one of the fastest way to implement lzw decompression.

Enhancement additional features (lzw_c2.cpp)

Usage: It can compress multiple colored bmp files into one file. And through the program, it can extract bmp files from the archive. Before the program runs, it needs a library of bmp (bmp.h) and compile with bmp.cpp in the command prompt of visual studio, ie :

cl lzw_c2 bmp.cpp

The input format will be as follow:

Compression:

> lzw_c2 –c <lzw_c2 filename> <a list of bmp files>

Decompression:

> lzw_c2 –d <lzw_c2 filename>

It is very similar to the basic part of lzw compression. Only the format of the list of file changed. The sample output format of the program is placed at the end the this file.

The data structure of this program is same as lzw.cpp mentioned before. In this program, I make a small amendment on the writefileheader to save the height and width of

the bmp files, ie:

```
void writefileheader(FILE *lzw_file,char** input_file_names,int no_of_f
iles)
{
    int i;
    /* write the file header */
    for ( i = 0 ; i < no_of_files; i++)
    {
        Bitmap image(input_file_names[i]);
        int h = image.getHeight();
        int w = image.getWidth();
        fprintf(lzw_file,"%s\n",input_file_names[i]);
        fprintf(lzw_file,"%d\n",h);
        fprintf(lzw_file,"%d\n",w);
    }
    fputc('\n',lzw_file);

}
```

The algorithm of this program is straightforward. I just go through all the pixels of the required bmp file three times. For each time it saves the R,G and B of the color pixel. The RGB values are treated as an ascii code ranged from 0 to 255. Then I use the same procedure to output the compressed code to the lzw file.

```
void compress(char * name, FILE *output)
{

    /* ADD CODES HERE */
    Bitmap image_data(name);
    cout << name << endl;
    Hash h(ascii_size);
    h.initialize();
    string Prefix = "";
    unsigned int X;
    int end;
    char C;
    int pos = ascii_size;
    int found;
    int temp;
```

```
    if (image_data.getData() == NULL) {
        cout << "cannot load data" << endl;
    }
    //     FILE * d = fopen("debug.txt", "w");
    //     fprintf(d, "%s" , name);
    //     fclose(d);
    // }
    int height = image_data.getHeight();
    int width = image_data.getWidth();
    printf("Height : %d Width : %d \n", height, width);
    // write the height and width of the image
    // write_code(output, height, CODE_SIZE);
    // write_code(output, width, CODE_SIZE);
    unsigned char rgb[3]; // R = 0 , G = 1 , B = 2
    for (int i = 0; i < 3; i++) {
       for (int r = 0; r < height; r++) {
            for (int c = 0; c < width; c++) {
                image_data.getColor(c , r , rgb[0] , rgb[1] , rgb[2]);

                temp = rgb[i];

                if (temp >= ascii_size/2) temp = temp - ascii_size;
                C = (char) temp;
                // printf("rgb : %d r : %d c: %d\n", temp,r,c);
                found = h.find(Prefix + C);
                if (found != npos) {
                    Prefix = Prefix + C;
                } else
                {
                    X = h.find(Prefix);
                    write_code(output, X , CODE_SIZE);
                    h.insert(Prefix + C , pos);
                    Prefix = C;
                    pos++;
                }

                // initialize the hash table when the dictionary size i
s full
```

```
                if (pos == dict_size-1) {
                    h.initialize();
                    pos = ascii_size;
                }

            }
        }
    }
    X = h.find(Prefix);
    write_code(output, X , CODE_SIZE);

    // Add indication of end of file
    write_code(output , dict_size -1 , CODE_SIZE);

}
```

For the decompression part, I separate it to two parts: 1. decompress the code, 2. extract the bmp file.

For the decompression of the code, I extract the string from the code in the file by the algorithm mentioned above. Then they are stored in a char array (named pixel). Which means all required color information is in this array of character.

```
void decompress(FILE *input ,int h, int w )
{

    /* ADD CODES HERE */
    int count = 0;
    initializeDict(dict);
    unsigned int PW = read_code(input, CODE_SIZE);
    char C = dict[PW][0];
    unsigned int CW;
    int end;
    pixel = (char * )malloc(sizeof(char) * h*w*3);
    pixel[count] = C;
    count++ ;
    string S,P;

    int pos = ascii_size;
```

```
    // FILE * tfile = fopen("temp.txt", "w");
    // read the code of file whenever there is no indication of the end
 of file
    while ((end = read_code(input,CODE_SIZE) ) != dict_size-1) {

        CW = end;
        if (CW < pos) {
            C = dict[CW][0];
            S = dict[CW];
        } else {
            C = dict[PW][0];
            S = dict[PW] + C;
        }
        for (int i = 0; i < S.length(); i++){
            pixel[i+count] = S.at(i);
        }
        count += S.length();
        P = dict[PW];
        dict[pos] = P + C;
        pos++;
        PW = CW;
        if (pos == dict_size-1) {
            initializeDict(dict);
            pos = ascii_size;
        }
    }
    // fclose(tfile);
}
```

After I get the desired bmp color information, I start reconstructing the bmp file. As the name of bmp file, height and width are known by reading the file header. I just need to go through the column and the height once, setting the color by the char array(named pixel) one by one and finally save it as bmp.

```
void extractbmp(int width, int height , char * name)
{
    Bitmap image_data(width, height);
    int h = image_data.getHeight();
    int count = 0;
    int w = image_data.getWidth();
```

```
    char temp;
    FILE * tfile = fopen("temp.txt", "r");
    short * cr = (short *)malloc(sizeof(short) * h * w);
    short * cg = (short *)malloc(sizeof(short) * h * w);
    short * cb = (short *)malloc(sizeof(short) * h * w);
    int rgb[3];
    int tint;
    // for (int i = 0; i < 3; i++) {
        for (int r = 0; r < h; r++) {
            for (int c = 0; c < w; c++) {
            // fscanf(tfile , "%d", &temp);

            temp = pixel[count];
            // if (i == 0)
            *(cr+r*w+c) = temp;
            // else if (i == 1)
            temp = pixel[count + h*w];
                *(cg+r*w+c) = temp;
                // else
            temp = pixel[count + h*w*2];
                *(cb+r*w+c) = temp;
                // printf("rgb : %d r : %d c: %d\n", rgb,r,c);
            image_data.setColor(c,r,*(cr+r*w+c),*(cg+r*w+c),*(cb+r*w+c)
);
            count++;
        }
        fclose(tfile);
    }
    // }
    // for (int r = 0; r < h; r++) {
    //         for (int c = 0; c < w; c++) {
    //          image_data.setColor(c,r,*(cr+r*w+c),*(cg+r*w+c),*(cb+r*
w+c));
    //      }
    // }
    free(cr);
    free(cb);
    free(cg);
```

```
    image_data.save(name);
}
```

Analysis

The reason of why I am reading R then G then B of color pixels is that the value of RGB of a pixel is hard to same as the other. So separate reading R, G and B can increase the chances of finding a string in the dictionary, which means the compressed file size would be smaller.

As the color pixels need to be run 3 times. The time complexity is O(n^3). The char array (named pixel) will store in the memory depending on the bmp file size. So it will occupy a certain amount of space. For the decompression, the time of extracting the colors from code would be O(n), depending on the size of the images. And the building of the bmp image is also O(n), depending on the dimension of the images.

Here is some of the running results:

Reference :

https://stackoverflow.com/questions/5248915/execution-time-of-c-program
https://www.geeksforgeeks.org/c-program-hashing-chaining/
https://www.cplusplus.com/reference/list/list/
https://www.geeksforgeeks.org/pair-in-cpp-stl/