武 汉 大 学

学 生 实 验 报 告 书

实验课程名称	数据结构实验
开课学院	国家网络安全学院
指导教师姓名	董红斌
学生姓名	王祥国
学生专业班级	计算机类 四班

2019 -- 2020 学年 第 2 学期

实验课程名称：数据结构实验

实验项目名称	二叉树与哈夫曼图片压缩			报告成绩	
实验者	王祥国	专业班级	计算机类四班	完成日期	2020/5/31

第一部分：实验分析与设计

一、 实验目的和要求

1. 目的

掌握树的存储结构
掌握二叉树的三种遍历方法
掌握 Huffman 树、Huffman 编码等知识和应用
使用 C/C++、文件操作和 Huffman 算法实现“图片压缩程序”专题编程。

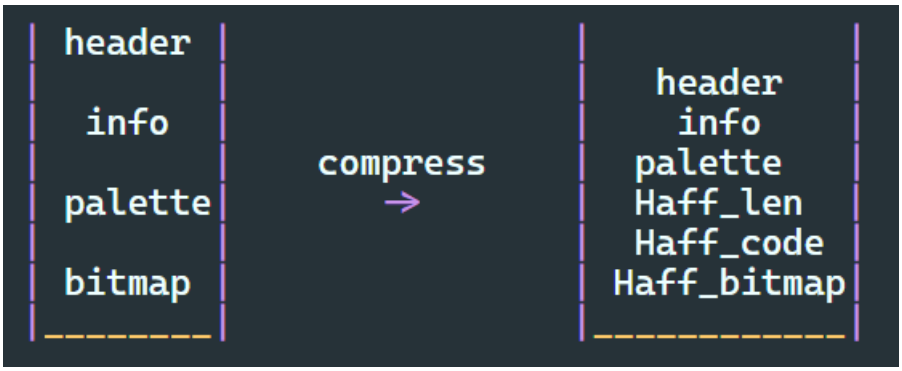
2. 要求

针对一幅 BMP 格式的图片文件，统计 256 种不同字节的重复次数，以每种字节重复次数作为权值，构造一颗有 256 个叶子节点的哈夫曼二叉树。利用上述哈夫曼树产生的哈夫曼编码对图片文件进行压缩。压缩后的文件与原图片文件同名，加上后缀.huf（保留原后缀），如 pic.bmp 压缩后 pic.bmp.huf

依据上述的实验目的与要求，可导出实现的二叉树与赫夫曼图片压缩软件的流程为：

① 读取图片文件、统计权值
② 生成 Huffman 树
③ 生成 Huffman 编码
④ 压缩图片文件
⑤ 保存压缩的文件

二、 分析与设计



1. 数据结构的设计

① 创建储存bmp文件的数据结构。

要对bmp格式图片进行解析处理，先弄清楚bmp格式文件主要分为四个部分。

Ⅰ：文件头BitmapHeader：四个部分，其中包含整个文件的的大小。

Ⅱ：信息头BitmapInfo：比较重要的有图像的宽高，调色板索引数。

Ⅲ：调色板palette_ptr：当位深度为8的时，有2^8=256种颜色，则调色板有256*4(RAGB四通道)=1024

字节。可看作一个数组，下表从0到255，每个元素占四字节代表一种颜色。

Ⅳ：像素数据content_ptr：该块大小Bytes=长*宽，每一个像素一个字节，该字节值作为调色板的下标取颜色。

尽管bmp图片不全是这样的格式，比如对于24位深度或者32位深度无调色板，像素数据直接表示颜色。bmp图片本身也存在压缩方式，比如RLE算法压缩，其实bmp图像原来也有叫做Huffman 1D的压缩格式，但现在似乎被弃用了。

根据以上的内容，定义bmp图片数据结构如下：

```
typedef struct BMP {
    unsigned raw_size;
    unsigned palette_size;
    BitmapHeader *header;
    BitmapInfo *info;
    unsigned *palette_ptr;
    unsigned char *content_ptr;
    unsigned int (*getPixel)(struct BMP *, int, int);
    void (*bitmapDestroy)(struct BMP *);
} Bitmap;
```

raw_size表示整个图的大小，palette_size表示调色板大小，一般是256，header是文件头指针，info是信息头指针，palette_ptr指向调色板字符串，content_ptr指向像素数据。其中，header的类型和info的类型基本按照其本身内容定义，因在程序中没太用到，不做过多描述，详见源代码。另外为了增强代码模块化，将两个对此结构体很重要的函数并入了该结构体，getPixel是获得指定位置的像素值，bitmapDestroy销毁对象。为了方便后面的Haffman编码，有如下数据结构定义：

```
typedef struct HUFFBMP {
    Bitmap *bitmap;
    unsigned int weight[256];
    unsigned int code_length[256];
    unsigned char huffcodes[256][256];
    void (*huffmanEncode)(struct HUFFBMP *encoding_huffmap);
    int (*writeFile)(struct HUFFBMP *, char *filename);
    void (*huffmapDestroy)(struct HUFFBMP *);
} Huffmap;
```

其中，bitmap就是上面定义的Bitmap结构体的指针类型，表示一副bmp图像的完整内容。前面讲过，对于位深度为8的情况，有256种颜色。weight[]数组存放像素数据区256种颜色对应的每种字节出现的次数，作为Haffman编码过程中的依据。huffcodes[][]数组存放huffman编码。code_length存放编码长度，以方便写入文件。huffmanEncode函数可以对该结构体所代表的bmp图像进行haffman编码，writeFile函数将编码后的图像写入新文件，huffmapDestroy函数用在最后销毁该结构体指针。

② 建立哈夫曼树的结构体。

这里采用数组的方式构建huffman树。huffman树的结点的数据结构定义如下：

```
typedef struct {
    unsigned color_data:8;
    char huffcodes[256];
    int code_length;
    int weight;
```

```
    int parent;
    int left_child;
    int right_child;
} HuffNode;
```

color_data表示该结点对应的颜色类型，huffcodes是它的huffman编码，weight表示权值，parent父节点，left_child和right_child表示左右孩子。

2. 核心算法设计（核心算法模块，细节可以用代码描述）

第一步：读取 bmp 图片文件。

对应 main.c 主函数第一句：

```
Huffmap *test = initHuffmap((char *)("image.bmp"));
```

以要进行 Haffman 编码压缩的图片名为参数，调用函数 inithuffmap，返回值为 Huffmap 指针类型，赋值给 test。函数如下：

```
Huffmap *initHuffmap(char *filename) {
    Huffmap *huffmap = (Huffmap *) malloc(sizeof(Huffmap));
    huffmap->bitmap = _openBitmap(filename);
    huffmap->huffmapDestroy = _huffmapDestroy;
    huffmap->writeFile = _writeFile;
    huffmap->huffmanEncode = _huffmanEncode;

    int x_max = huffmap->bitmap->info->biWidth;
    int y_max = huffmap->bitmap->info->biHeight;
    unsigned int index = 0;

    for (int i = 0; i < y_max; ++i) {
        for (int j = 0; j < x_max; ++j) {
            index = huffmap->bitmap->getPixel(huffmap->bitmap, j, i);
            huffmap->weight[index]++;
        }
    }
    return huffmap;
}
```

函数流程就是依照图片文件数据，给要创建的 huffmap 结构体指针变量的每一个成员赋值。首先是 bitmap 成员，由于 bitmap 本身也是结构体，内部成员也很多，再调用一个函数_openBitmap()返回 Bitmap 指针类型的值给 huffmap 的 bitmap 成员变量。然后后面的二重循环的作用是利用 getPixel()函数获取每种颜色出现的数量当作权值，作为后面哈夫曼编码的依据。下面描述 initHuffmap()函数中各个函数的作用。

_openBitmap：真正读取图片，返回给 huffmap->bitmap

```
Bitmap *_openBitmap(char *filename) {
    Bitmap *map = (Bitmap *) malloc(sizeof(Bitmap));
    map->bitmapDestroy = _bitmapDestroy;
    map->getPixel = _getPixel;
    map->header = (BitmapHeader *) malloc(sizeof(BitmapHeader));
    map->info = (BitmapInfo *) malloc(sizeof(BitmapInfo));
```

```
    // the pointer to extire file structure

    FILE *file = fopen(filename, "rb");
    // file describer for our bmp file
    assert(file != NULL);
    fread(map->header, 1, HEADER_SIZE, file);
    // read data from file for further analyse
    fread(map->info, 1, INFO_SIZE, file);

    printf("Got Image, Analyzing...\n");
    printf("Image Pixel Depth: %d\n", map->info->biBitCount);
    printf("Height: %d\nWidth: %d\n", map->info->biHeight, map->info->biWidth);
    printf("Clr Found, Avalibile Clr Number: %d\n", map->info->biClrUsed);

    map->palette_size = (unsigned int) pow(2, map->info->biBitCount);
    map->palette_ptr = (unsigned *) malloc(map->palette_size * sizeof(int));

    if (map->info->biSize > INFO_SIZE) {
        map->info->biOtherDataSize = map->info->biSize - INFO_SIZE;
        map->info->biOtherData = (char *) malloc(map->info->biOtherDataSize);
        fread(map->info->biOtherData, 1, map->info->biOtherDataSize, file);
    } else {
        map->info->biOtherDataSize = 0;
    }

    fread(map->palette_ptr, 1, 4 * map->palette_size, file);
    // read pallette to memory

    map->raw_size = map->info->biHeight * inter2real(map->info->biWidth);
    assert(map->raw_size == map->info->biSizeImage);
    printf("Raw Total Size in Bytes: %d (0x%x)\n", map->raw_size,map->raw_size);

    map->content_ptr = (unsigned char *) malloc(map->raw_size);
    if (fread(map->content_ptr, 1, map->raw_size + 1, file) != map->raw_size) {
        printf("File corrupt.\n");
        map->bitmapDestroy(map);
        exit(1);
    }
    printf("Image Successfully Loaded.\n");
    return map;
}
```

该函数在生成 bitmap 的同时会检测 bmp 图像是否正确，在图像数据异常的时候(比如像素数据长度对不上等)，会结束程序并放出提示。

接下来生成哈夫曼编码过程中必须的权值数组 huffman->weight[]。该过程对每一个像素点使用

_getPixel()函数，返回该点的颜色值对应的索引。

```
unsigned int _getPixel(Bitmap *map, int X, int Y) {
    assert(X < map->info->biWidth && Y < map->info->biHeight && X
    >= 0 && Y >= 0);
    int real_line = inter2real(map->info->biWidth);
    unsigned int res = *(map->content_ptr + X + Y * real_line);
    return res & 0xff;
}
```

这里比较重要的是 inter2real()这个宏，它的定义为：

```
#define realrow(x) (4 * ceil((double)(x) * 8 / 32))
```

这样做是因为 bmp 图片文件中的像素数据是以行存的，为了方便索引，一定是 4 的倍数，就是说假设图片每行只有 7 个像素，但存的时候还是以 8 个存，不过最后一个特殊处理了。所以我们为了得到真实的行数，需要用这个宏转换一下。

第二步：实现哈夫曼编码

对应 main.c 主函数第二句：

```
test->huffmanEncode(test);
```

对应的函数为：

```
void _huffmanEncode(Huffmap *encoding_huffmap) {
    HuffNode huffnode[512];
    for (int i = 0; i < 256; i++) {
        huffnode[i].color_data = i;
        huffnode[i].weight = encoding_huffmap->weight[i];
        huffnode[i].left_child=huffnode[i].right_child=huffnode[i].parent = -1;
    }
    for (int i = 256; i < 511; i++) {
        int minimum_1 = 0x7fffffff;
        int minimum_2 = 0x7fffffff;
        int leftnode = 0;
        int rightnode = 0;

        for (int j = 0; j < i; j++) {
            if (huffnode[j].parent == -1) {
                if (huffnode[j].weight < minimum_1) {
                    minimum_2 = minimum_1;
                    rightnode = leftnode;
                    minimum_1 = huffnode[j].weight;
                    leftnode = j;
                }
                else if (huffnode[j].weight < minimum_2) {
                    minimum_2 = huffnode[j].weight;
                    rightnode = j;
                }
            }
```

```
        }
        huffnode[i].parent = -1;
        huffnode[i].left_child = leftnode;
        huffnode[i].right_child = rightnode;
        huffnode[i].weight=huffnode[leftnode].weight+huffnode[rightnode].weight;
        huffnode[leftnode].parent = huffnode[rightnode].parent = i;
    }
    huffnode[510].code_length = 0;

    for (int i = 510; i >= 256; i--) {
        int fatherlength = huffnode[i].code_length;
        int leftchild = huffnode[i].left_child;
        int rightchild = huffnode[i].right_child;
        huffnode[leftchild].code_length = huffnode[rightchild].code_length =
 fatherlength + 1;
        for (int j = 0; j < fatherlength; j++) {
            huffnode[leftchild].huffcodes[j] = huffnode[rightchild].huffcodes[j]
             = huffnode[i].huffcodes[j];
        }
        huffnode[leftchild].huffcodes[fatherlength] = 0;
        huffnode[rightchild].huffcodes[fatherlength] = 1;
    }

    for (int i = 0; i < 256; i++) {
        encoding_huffmap->code_length[i] = huffnode[i].code_length;
        for (int j = 0; j < huffnode[i].code_length; j++) {
            encoding_huffmap->huffcodes[i][j] = huffnode[i].huffcodes[j];
        }
    }
}
```

Huffman 编码的思路：每次在可选结点中选择两个权值最小的点，合并起来，成为一个结点，并将这个结点加入可选的结点中，直到最后只剩下一个点。在树建好后，选择从根节点开始，向左为 0，向右为 1，遍历一遍，就能得到每个叶结点的 Huffman 编码了。最后还要把 Huffman 编码的长度也统计一下，方便写入文件。

第三步：写入新文件

对应 main.c 主函数第三句：

```
test->writeFile(test, (char *)("image.bmp.huf"));
```

这里不仅仅是将 bitmap data 区的数据进行哈夫曼编码，另外还在 bitmap data 区之间加了两个区，分别为从第一种颜色到最后一种颜色每种颜色哈夫曼编码的长度和对应的哈夫曼编码，这样做是考虑到了进行解码的方便。

第四步：释放内存

对应 main.c 主函数第四句：

```
test->huffmapDestroy(test);
```

第二部分：实验过程和结果

一、 实现说明（软硬件开发环境、开发过程说明）

在 ubuntu 18.04 和 windows10 下都进行了测试。

```
gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04)
gcc version 8.1.0 (x86_64-posix-sjlj-rev0, Built by MinGW-W64 project)
```

开发流程：主要分为四个部分。bmp 文件读取->Haffman 编码->写入文件->释放内存

二、 调试说明（调试手段、过程及结果分析）

gdb 9.1 with peda

调试过程解决了三个问题。

一是在windows下，若用“r”方式读取文件，会导致读取到1A字节时意外中止，改成“rb”方式就行了。

```
FILE *file = fopen(filename, "rb");
```

二是在写文件时，没有考虑到哈夫曼编码后的长度问题，在考虑整体长度时估计小了，malloc()分配内存不够，导致越界出现Segment Fault。

三是在进行Huffman编码时，仅仅对编码数组前256为进行了初始化，导致在每次寻找两个最小的权值结点时候出现了偏差。

三、 软件测试（测试效果、界面、综合分析和结论）

随便找了几张图进行测试，运行截图如下：

图一： 

图二： 

```
me@ubuntu:~/Documents$ gcc main.c -lm -o main
me@ubuntu:~/Documents$ ./main
Welcome!This is a compressor for BMP file
Filename:image
Target File:image.bmp
Analyzing...
          IMAGE ATTRIBUTION
-----------------------------------
|       Image Pixel Depth:  8      |
-----------------------------------
|  Height:  3541  |  Width:  2508  |
-----------------------------------
|  palette Found, Clr Number: 256  |
-----------------------------------
|  BitMap Size: 8880828, 0x8782bc  |
-----------------------------------
Image Successfully Loaded.
Huffman encoding...
File size(after compression): 5748909(0x57b8ad)
compression efficiency:%35.27
Output filename:image.bmp.huf
-------------------END-------------------
Press "ENTER" to continue...
```

```
me@ubuntu:~/Documents$ ./main
Welcome!This is a compressor for BMP file
Filename:123
Target File:123.bmp
Analyzing...
          IMAGE ATTRIBUTION
-----------------------------------
|       Image Pixel Depth:  8      |
-----------------------------------
|  Height:   718  |  Width:  1457  |
-----------------------------------
|  palette Found, Clr Number: 256  |
-----------------------------------
|  BitMap Size: 1048280, 0xffed8   |
-----------------------------------
Image Successfully Loaded.
Huffman encoding...
File size(after compression): 134814(0x20e9e)
compression efficiency:%87.15
Output filename:123.bmp.huf
-------------------END-------------------
Press "ENTER" to continue...
```

```
PS D:\my programs\suibianxiexie> cd "d:\my programs\suibianxiexie\" ; if ($?) { gcc main.c -o main } ; if ($?) { .\main }
Welcome!This is a compressor for BMP file
Filename:image
Target File:image.bmp
Analyzing...
          IMAGE ATTRIBUTION
-----------------------------------
|       Image Pixel Depth:  8      |
-----------------------------------
|  Height:  3541  |  Width:  2508  |
-----------------------------------
|  palette Found, Clr Number: 256  |
-----------------------------------
|  BitMap Size: 8880828, 0x8782bc  |
-----------------------------------
Image Successfully Loaded.
Huffman encoding...
File size(after compression): 6221355(0x5eee2b)
compression efficiency:%29.96
Output filename:image.bmp.huf
-------------------END-------------------
Press "ENTER" to continue...
```

第三部分：实验小结、收获与体会

根据运行例子所示，压缩时候的压缩效率从百分之二十多到百分之八十多不等，效率还是非常可观的。现在尝试从理论上进行分析：

在原来的编码中，每一种像素数据都是按照一字节(8 bits)进行编码的。Huffman 编码过程中，进行的优化方式是针对不同像素色值出现的次数进行编码的再分配，给出现次数较多的色值较短的编码，以此达到缩短编码的效果。考虑极端情况：当整张图片只有一种颜色，原编码方式给这种颜色一字节(8 bit)的编码长度，但经过 Huffman 编码后，可以就用 1(1 bit)表示，因此最大压缩率可以达到压缩 7/8。

最坏的情况：如果原图片中 256 种颜色出现次数都是平均的，那么 Huffman 编码时形成的 Huffman 树会是一个满二叉树，这样一来，每种颜色的编码都是 8 bit，就跟原来的 1 字节没区别，也就没体现出压缩的优点了。总结以下就是说，颜色越少，出现次数方差越大，压缩效果越好。

第四部分：附完整源代码

main.c 文件源码。

```c
//
//  main.c
//
//    国家网络安全学院
//    2019302180119
//   王祥国 2020/5/30

#include "bitmap.h"

int main() {
    printf("Welcome!This is a compressor for BMP file\n");
    printf("Filename:");
    char *filename = (char *)malloc(20 * sizeof(char));
    _startInput(filename);
    // input process

    Huffmap *test = initHuffmap(filename);
    test->huffmanEncode(test);
    test->writeFile(test, strcat(filename, ".huf"));
    test->huffmapDestroy(test);
    // open file and Huffman encode

    free(filename);
    getchar();
    return 0;
}
```

bitmap.h 文件源码

```c
//
//    bitmap.h
//
//    国家网络安全学院
//    2019302180119
//    王祥国 2020/5/30

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>

#define HEADER_SIZE 14
#define INFO_SIZE 40
#define realrow(x) (4 * ceil((double)(x) * 8 / 32))

typedef struct {
    unsigned bfType:16;
    unsigned bfSize;
    unsigned bfReserved1:16;
    unsigned bfReserved2:16;
    unsigned bfOffBits;
} BitmapHeader;

typedef struct {
    unsigned biSize;
    unsigned biWidth;
    int biHeight;
    unsigned biPlanes:16;
    unsigned biBitCount:16;
    unsigned biCompression;
    unsigned biSizeImage;
    unsigned biXPelsPerMeter;
    unsigned biYPelsPerMeter;
    unsigned biClrUsed;
    unsigned biClrImportant;
    unsigned biOtherDataSize;
    char *biOtherData;
} BitmapInfo;
```

```c
typedef struct BMP {
    unsigned raw_size;
    // the total size(including padding) of data field
    BitmapHeader *header;
    // the header structure
    BitmapInfo *info;
    // the information structure
    unsigned palette_size;
    unsigned *palette_ptr;
    // to pointer to palette
    unsigned char *content_ptr;
    // the pointer to pure data
    unsigned int (*getPixel)(struct BMP *, int, int);
    void (*bitmapDestroy)(struct BMP *);
} Bitmap;

typedef struct HUFFBMP {
    Bitmap *bitmap;
    unsigned int weight[256];
    unsigned int code_length[256];
    unsigned char huffcodes[256][256];
    void (*huffmanEncode)(struct HUFFBMP *encoding_huffmap);
    int (*writeFile)(struct HUFFBMP *, char *filename);
    void (*huffmapDestroy)(struct HUFFBMP *);
} Huffmap;

typedef struct {
    unsigned color_data:8;
    char huffcodes[256];
    int code_length;
    int weight;
    int parent;
    int left_child;
    int right_child;
} HuffNode;

void _startInput(char *filename);
Huffmap *initHuffmap(char *filename);
Bitmap *_openBitmap(char *filename);
unsigned int _getPixel(Bitmap *, int X, int Y);
int _writeFile(struct HUFFBMP *, char *filename);
void _huffmanEncode(Huffmap *encoding_huffmap);
void _concatBinary(unsigned char *src, unsigned char *bin, unsigned length,
```

```
unsigned orign_length);
void _setBit(unsigned char *src, unsigned bit, unsigned pos);
void _huffmapDestroy(struct HUFFBMP *);
void _bitmapDestroy(Bitmap *);

Huffmap *initHuffmap(char *filename) {
    Huffmap *huffmap = (Huffmap *) malloc(sizeof(Huffmap));
    huffmap->bitmap = _openBitmap(filename);
    huffmap->huffmapDestroy = _huffmapDestroy;
    huffmap->writeFile = _writeFile;
    huffmap->huffmanEncode = _huffmanEncode;

    int x_max = huffmap->bitmap->info->biWidth;
    int y_max = huffmap->bitmap->info->biHeight;
    unsigned int index = 0;

    for (int i = 0; i < y_max; ++i) {
        for (int j = 0; j < x_max; ++j) {
            index = huffmap->bitmap->getPixel(huffmap->bitmap, j, i);
            huffmap->weight[index]++;
        }
    }
    return huffmap;
}

Bitmap *_openBitmap(char *filename) {
    Bitmap *map = (Bitmap *) malloc(sizeof(Bitmap));
    map->bitmapDestroy = _bitmapDestroy;
    map->getPixel = _getPixel;
    map->header = (BitmapHeader *) malloc(sizeof(BitmapHeader));
    map->info = (BitmapInfo *) malloc(sizeof(BitmapInfo));
    // the pointer to extire file structure

    FILE *file = fopen(filename, "rb");
    // file describer for our bmp file
    assert(file != NULL);
    fread(map->header, 1, HEADER_SIZE, file);

    // read data from file for further analyse
    fread(map->info, 1, INFO_SIZE, file);

    printf("Target File:%s\nAnalyzing...\n", filename);
    printf("          IMAGE ATTRIBUTION\n");
    printf("-----------------------------------\n");
```

```
    printf("|       Image Pixel Depth: %2d      |\n", map->info->biBitCount);
    printf("-------------------------------------\n");
    printf("|  Height:%6d  |  Width:%6d  |\n", map->info->biHeight,
    map->info->biWidth);
    printf("-------------------------------------\n");
    printf("|  palette Found, Clr Number: %3d  |\n", map->info->biClrUsed ==
     0 ? 1 << map->info->biBitCount : map->info->biClrUsed);
    printf("-------------------------------------\n");

    map->palette_size = (unsigned int) pow(2, map->info->biBitCount);
    map->palette_ptr = (unsigned *) malloc(map->palette_size * sizeof(int));

    if (map->info->biSize > INFO_SIZE) {
        map->info->biOtherDataSize = map->info->biSize - INFO_SIZE;
        map->info->biOtherData = (char *) malloc(map->info->biOtherDataSize);
        fread(map->info->biOtherData, 1, map->info->biOtherDataSize, file);
    } else {
        map->info->biOtherDataSize = 0;
    }

    fread(map->palette_ptr, 1, 4 * map->palette_size, file);
    // read pallette to memory

    map->raw_size = map->info->biHeight * realrow(map->info->biWidth);
    assert(map->raw_size == map->info->biSizeImage);
    printf("|  BitMap Size:%8d, 0x%-8x|\n", map->raw_size, map->raw_size);
    printf("-------------------------------------\n");

    map->content_ptr = (unsigned char *) malloc(map->raw_size);
    if (fread(map->content_ptr, 1, map->raw_size + 1, file) != map->raw_size) {
        printf("File corrupt.\n");
        map->bitmapDestroy(map);
        exit(1);
    }
    map->header->bfSize = ftell(file);
    printf("Image Successfully Loaded.\n");
    return map;
}

unsigned int _getPixel(Bitmap *map, int X, int Y) {
    assert(X < map->info->biWidth && Y < map->info->biHeight && X >= 0&&Y>=0);
    int real_line = realrow(map->info->biWidth);
    unsigned int res = *(map->content_ptr + X + Y * real_line);
    return res;
```

```
}

void _huffmanEncode(Huffmap *encoding_huffmap) {
    printf("Huffman encoding...\n");
    HuffNode huffnode[512];
    for (int i = 0; i < 256; i++) {
        huffnode[i].color_data = i;
        huffnode[i].weight = encoding_huffmap->weight[i];
        huffnode[i].left_child = huffnode[i].right_child=huffnode[i].parent=-1;
    }
    for (int i = 256; i < 511; i++) {
        int minimum_1 = 0x7fffffff;
        int minimum_2 = 0x7fffffff;
        int leftnode = 0;
        int rightnode = 0;

        for (int j = 0; j < i; j++) {
            if (huffnode[j].parent == -1) {
                if (huffnode[j].weight < minimum_1) {
                    minimum_2 = minimum_1;
                    rightnode = leftnode;
                    minimum_1 = huffnode[j].weight;
                    leftnode = j;
                }
                else if (huffnode[j].weight < minimum_2) {
                    minimum_2 = huffnode[j].weight;
                    rightnode = j;
                }
            }
        }
        huffnode[i].parent = -1;
        huffnode[i].left_child = leftnode;
        huffnode[i].right_child = rightnode;
        huffnode[i].weight=huffnode[leftnode].weight+huffnode[rightnode].weight;
        huffnode[leftnode].parent = huffnode[rightnode].parent = i;
    }
    huffnode[510].code_length = 0;

    for (int i = 510; i >= 256; i--) {
        int fatherlength = huffnode[i].code_length;
        int leftchild = huffnode[i].left_child;
        int rightchild = huffnode[i].right_child;
        huffnode[leftchild].code_length=huffnode[rightchild].code_length=
fatherlength + 1;
```

```
        for (int j = 0; j < fatherlength; j++) {
            huffnode[leftchild].huffcodes[j]=huffnode[rightchild].huffcodes[j]
= huffnode[i].huffcodes[j];
        }
        huffnode[leftchild].huffcodes[fatherlength] = 0;
        huffnode[rightchild].huffcodes[fatherlength] = 1;
    }

    for (int i = 0; i < 256; i++) {
        encoding_huffmap->code_length[i] = huffnode[i].code_length;
        for (int j = 0; j < huffnode[i].code_length; j++) {
            encoding_huffmap->huffcodes[i][j] = huffnode[i].huffcodes[j];
        }
    }
}

int _writeFile(struct HUFFBMP *map, char *filename) {
    FILE *res = fopen(filename, "w");
    const char *hufname = "HUF";
    fwrite(map->bitmap->header, 1, HEADER_SIZE, res);
    fwrite(map->bitmap->info, 1, INFO_SIZE, res);
    if (map->bitmap->info->biOtherDataSize) {
        fwrite(map->bitmap->info->biOtherData, 1,
map->bitmap->info->biOtherDataSize, res);
    }
    fwrite(map->bitmap->palette_ptr, 4, map->bitmap->palette_size, res);
    // write all auxillary data

    fwrite(hufname, 1, 3, res);
    // write 'huf as sign

    for (int i = 0; i < 256; i++) {
        fwrite(&map->code_length[i], 1, 1, res);
    }


    unsigned char *after = 0;
    unsigned int after_length = 0;
    unsigned char *out = (unsigned char *) calloc(256, 256);
    unsigned out_length = 0;

    for (int i = 0; i < 256; ++i) {
        after = map->huffcodes[i];
        after_length = map->code_length[i];
```

```
        _concatBinary(out, after, after_length, out_length);
        out_length += after_length;
    }
    fwrite(out, 1, out_length / 8 + 1, res);
    free(out);



    out = (unsigned char *) calloc(map->bitmap->raw_size, 1);
    out_length = 0;
    unsigned char before = 0;

    for (int i = 0; i < map->bitmap->raw_size; ++i) {
        before = map->bitmap->content_ptr[i];
        after = map->huffcodes[before];
        after_length = map->code_length[before];
        _concatBinary(out, after, after_length, out_length);
        out_length += after_length;
    }
    fwrite(out, 1, out_length / 8 + 1, res);
    printf("File size(after compression): %ld(0x%lx)\n",ftell(res),ftell(res));
    printf("compression efficiency:%%%.2lf\n",
    (1-((double)ftell(res)/map->bitmap->header->bfSize))*100);
    fclose(res);
    free(out);
    printf("Output filename:%s\n", filename);
    return 0;
}

void _startInput(char *filename) {
    fgets(filename, 20, stdin);
    int file_len = strlen(filename) - 1;
    filename[file_len] = '\0';
    if (file_len < 5 || filename[file_len - 4] != '.') {
        strcat(filename, ".bmp");
    }
}

void _concatBinary(unsigned char *dest, unsigned char *src,
 unsigned src_length, unsigned dest_length) {
    unsigned char *dest_base = dest + dest_length / 8;
    unsigned dest_offset = dest_length % 8;
```

```
    for (int i = 0; i < src_length; ++i) {
        _setBit(dest_base, src[i], dest_offset);
        dest_offset++;
        dest_base += dest_offset / 8;
        dest_offset %= 8;
    }
}

void _setBit(unsigned char *src, unsigned bit, unsigned pos) {
    pos = 7 - pos;
    if (bit) {
        *src = *src | (1 << pos);
    } else {
        *src = *src & ~(1 << pos);
    }
}

void _bitmapDestroy(Bitmap *map) {
    if (map->info->biOtherDataSize) {
        free(map->info->biOtherData);
    }

    free(map->header);
    free(map->content_ptr);
    free(map->info);
    free(map->palette_ptr);

    free(map);
}

void _huffmapDestroy(Huffmap *map) {
    _bitmapDestroy(map->bitmap);
    free(map);
    printf("--------------------END--------------------\n");
    printf("Press \"ENTER\" to continue...");
}
```