同济大学计算机系

OOP 文件压缩大作业实验报告

学　　号 2152118

姓　　名 史君宝

专　　业 计算机科学与技术（计科 1 班）

完成时间 2023.11.22

一、设计思路与功能描述

1. 得分点

（1）压缩比
（2）程序执行时间
（3）高阶的核心算法
（4）算法/程序的通用性

2. 设计思路

（1）采用哈夫曼树进行文件字符的编码

在文件压缩中有很多的压缩方式，我们主要讨论无损压缩。其中主要有游程编码、熵编码和字典编码。字典编码是很多压缩程序使用的方式，比如著名的 LZ77 算法和相关衍生算法。

今天我们主要使用的是在算法中经常遇到的哈夫曼编码，通过对 ASCALL 码字符的频率进行 01 编码，将原文件转化为新字符，并实行解码。

（2）解压缩和程序的通用性

在文件压缩之后必然有文件的解压过程，如何正确的解码，实现文件前后的对比，确保整个过程的完整性，是比较重要的。所以在程序设计中实现了文件的压缩，并将解码的帮助信息放入了压缩文件中，在之后的解码过程中通过读取这些帮助信息，可以顺利的重建哈夫曼树并实现解码。

3. 功能描述

（1）函数声明：

```
//定义哈夫曼树的结点
struct HaffmanNode{ ... };

//对于给定的链表，建立哈夫曼树。
HaffmanNode CreateHaffmanTree(HaffmanNode head){ ... }

HaffmanNode CreateNewHaffmanTree(HaffmanNode new_head, string str_node[]){ ... }

//字符串翻转函数
string reserve(string str){ ... }

//确定各字符的编码，求得对应的字符编码
void DFS(HaffmanNode* node, string ch_code[], string str){ ... }

//在建树过程中使用了动态内存申请，需要DFS查找删除结点
void DFS_delete(HaffmanNode* node){ ... }

//压缩文件
int zip(string in_file, string out_file){ ... }
//解压文件
int unzip(string in_file, string out_file){ ... }

//主函数
int main(int argc, char* argv[]){ ... }
```

（2）main 函数内容：

功能:实现一个简单的程序选择结构，根据命令行的输入指令，转向对应的压缩或者解压程序：

```
cout << "开始运行" << endl;
if (argc != 4) {
    cerr << "请确保输入的参数个数是否正确" << endl;
    return -1;
}
if (!strcmp(argv[3], "zip")||!strcmp(argv[3], "unzip")) {
    if (!strcmp(argv[3], "zip"))
    {
        int number = 0;
        number = zip(argv[1], argv[2]);
        if (number == -1)
            return 0;
    }
    else if (!strcmp(argv[3], "unzip"))
    {
        int number = 0;
        number = unzip(argv[1], argv[2]);
        if (number == -1)
            return 0;
    }
}
else {
    cerr << "Unknown parameter!\nCommand list:\nzip\nunzip" << endl;
    return -1;
}

cout << "Complete!" << endl;
return 0;
```

（3）zip 函数内容：

功能:程序的主体，需要将原文件实现一个压缩。
参数列表：int zip(string in_file, string out_file)
其中 in_file 是压缩前的文件，out_file 是压缩后的文件。

函数实现过程：

```
LARGE_INTEGER frequency;
LARGE_INTEGER start;
LARGE_INTEGER end;

// 获取时钟频率
QueryPerformanceFrequency(&frequency);

// 获取开始时间
QueryPerformanceCounter(&start);
```

上述是对压缩过程进行一个计时。

```
long long frequent[256] = { 0 };

//读文件
ifstream fin(in_file, ios::binary); // 以二进制方式打开文件
if (!fin) {
    cerr << "Can not open the input file!" << endl; // 输出错误信息并退出
    return -1;
}

double originalFileSize = fin.tellg();

istreambuf_iterator<char> begi(fin), endi; // 设置两个文件指针，指向开始和结束，以 char(一字节) 为步长
string content(begi, endi); // 将文件全部读入 string 字符串
fin.close(); // 操作完文件后关闭文件句柄是一个好习惯
```

首先先申请一个 256 的数组（因为需要计算各字符的频率，可能比较大，我们采用 lon long 类型）。
之后打开并读取文件，将内容储存在 content 中，并计算文件的大小 originalFileSize，用于后面计算压缩率。

```
int ch;
// 遍历字符串中的每个字符，计算频率
for (int i = 0; i < content.size(); ++i) {
    ch = int(content[i]);
    if (ch < 0)
        ch += 256;
    frequent[ch]++;
}
```

遍历整个文件的字符，计算频率并储存。

```
//根据频率建立相应的结点和链表，并进行排序
//链表的头结点
HaffmanNode head;
head.c = '0';
head.freq = 0;
head.flag = 0;
head.left = NULL;
head.right = NULL;
head.next = NULL;
```

```
for (int i = 0; i < 256; ++i)
{
    if (frequent[i] == 0)
        continue;

    HaffmanNode* new_node = new HaffmanNode;
    HaffmanNode* curNode = head.next;

    new_node->c = char(i);
    new_node->flag = 0;
    new_node->freq = frequent[i];
    new_node->left = NULL;
    new_node->right = NULL;
    new_node->next = NULL;

    if (!curNode) {
        head.next = new_node;
        continue;
    }
    else if (curNode->freq > new_node->freq) {
        new_node->next = head.next;
        head.next = new_node;
        continue;
    }

    while (curNode->next && (curNode->next->freq < new_node->freq))
        curNode = curNode->next;

    new_node->next = curNode->next;
    curNode->next = new_node;
}
```

上面是利用动态内存申请，建立一个链表，实现一个简单的按照频率排序，这样，也能简便我们之后的建树过程。

```
//根据链表可以建立哈夫曼树
head = CreateHaffmanTree(head);


//定义每个字符的string数组
string ch_code[256];
//通过DFS获得每个字符的编码
DFS(head.next, ch_code, "");
```

上面首先根据建立的链表实现哈夫曼树的建立（函数具体之后讲到）。
之后我们定义一个 256 的 string 数组，帮助我们储存每个字符的编码。
然后通过 DFS 深度优先搜索遍历真个哈夫曼树，将字符编码储存起来。

```
//根据前面获得的字符编码的数组，创建一个二进制的结果
string number_result = ""; //储存文件按照哈夫曼编码的二进制
string true_result = "";   //储存最终的文件编码结果
for (int i = 0; i < content.size(); ++i) {
    ch = content[i];
    number_result += ch_code[int(ch)];
}
```

遍历文件中的字符，通过上面获得的编码数组，将所有字符转化成对应的编码，即二进制形式，记为 number_result

```
//由于上面的二进制不一定是8的倍数，所以我们末尾补零，来帮助我们转化成字符
int disappear;
if (number_result.size() % 8 == 0)
    disappear = 0;
else
    disappear = 8 - number_result.size() % 8;

for (int i = 0; i < disappear; ++i)
    number_result += '0';
```

由于上面的二进制结果最后需要作为 bit 位来重新变成字符，所以如果上述二进制结果不是 8 的倍数，我们会在末尾补‘0’作为处理。

```
//将上面的二进制转换成对应的字符
for (int i = 0, number = 0; i < number_result.size(); i += 8)
{
    number = 0;
    for (int j = 7; j >= 0; j--)
    {
        if (number_result[i + 7 - j] == '0')
            number += 0;
        else
            number += pow(2, j);
    }
    true_result += char(number);
}
```

补齐位数后我们就可以将上面的二进制结果转化为对应的字符了，每 8 位加和作为一个字节进行转化。

```
//下面记录辅助信息
int help_number = 0;       //有编码的字符数统计
for (int i = 0; i < 256; ++i)
{
    if (ch_code[i] != "")
        help_number++;
}
```

下面是另一个辅助信息，帮助我们确定多少字符有编码，尽可能节省空间，简便程序。

之后就是将输入到压缩文件中：

```
fstream fout(out_file, std::ios::binary | std::ios::in | std::ios::out); // 打开输出文件
if (!fout) {
    cerr << "Can not open the output file!" << endl;
    return -1;
}

//下面是输出辅助信息
fout << disappear << " " << help_number << "\n";
for (int i = 0; i < 256; ++i)
{
    if (ch_code[i] != "")
        fout << i << " " << ch_code[i] << "\n";
}

DFS_delete(head.next);


fout << true_result;
```

我们以二进制读写打开文件，然后首先输入辅助信息，分别是 help_disappear 缺失位（即前面二进制结果补的 0 的个数），help_number（多少字符有编码）
然后输入转化后的新字符。
中间我们回收动态申请的内存，并不影响。

```
fout << true_result;
double compressedFileSize = fout.tellg();

fout.close();
cout << "文件压缩已经完成" << endl;

double compressionRate = (compressedFileSize / originalFileSize) * 100;
cout << "压缩率为: " << compressionRate << "%" << std::endl;

// 获取结束时间
QueryPerformanceCounter(&end);

// 计算程序执行时间
double duration = static_cast<double>(end.QuadPart - start.QuadPart) / frequency.QuadPart;

// 输出程序执行时间（以秒为单位）
cout << "压缩执行时间: " << duration << " 秒" << endl;

return 0;
```

最后计算压缩率和压缩执行时间就可以了。

（4）unzip 函数：

功能:程序的主体，需要将压缩文件实现解压。
参数列表：int unzip(string in_file, string out_file)
其中 in_file 是压缩文件，out_file 是解压后的文件。

函数实现过程：

```
LARGE_INTEGER frequency;
LARGE_INTEGER start;
LARGE_INTEGER end;

// 获取时钟频率
QueryPerformanceFrequency(&frequency);

// 获取开始时间
QueryPerformanceCounter(&start);
```

上述是对解压过程进行一个计时。

```
ifstream finin(in_file, ios::binary); // 打开输出文件
if (!finin) {
    //cerr << "Can not open the input file!" << endl;
    cout << "Can not open the input file!" << endl;
    return -1;
}
```

打开文件

```
int help_disappear = 0;
int help_num = 0;

//读取辅助信息
finin >> help_disappear >> help_num;
int str_num = 0;
string str;
string str_code[256];
for (int i = 0; i < help_num; ++i)
{
    finin >> str_num >> str; // 读取字符和对应的编码
    str_code[str_num] = str;
}

finin.get();
//读取重要的文本信息放入information的string中
istreambuf_iterator<char> begin(finin),
    endin; // 设置两个文件指针，指向开始和结束，以 char(一字节) 为步长
string information(begin, endin); // 将文件全部读入 string 字符串
```

读取全部的辅助信息和文本信息，分别是缺失位，有编码字符数，对应字符和对应编码。

```
HaffmanNode new_head;
new_head.c = '0';
new_head.freq = 0;
new_head.flag = 0;
new_head.left = NULL;
new_head.right = NULL;
new_head.next = NULL;

HaffmanNode* new_node = new HaffmanNode;
new_node->c = '0';
new_node->freq = 0;
new_node->flag = 1;
new_node->left = NULL;
new_node->right = NULL;
new_node->next = NULL;

new_head.next = new_node;

new_head = CreateNewHaffmanTree(new_head, str_code);
```

根据上面读取到的字符编码信息，我们重新创建哈夫曼树。

```
//下面两个string，分别储存二级制数据和解压结果
string number_key = "";
string true_key = "";

for (int i = 0; i < information.size(); ++i)
{
    int number = int(information[i]);
    if (number < 0)
        number += 256;
    for (int i = 7; i >= 0; i--)
    {
        if (number >= pow(2, i))
        {
            number_key += '1';
            number -= pow(2, i);
        }
        else
            number_key += '0';
    }
}
```

遍历文本信息，将所有字符转化为对应的二进制。

```
    char ch;
    HaffmanNode* curnode;
    curnode = new_head.next;
    for (int i = 0; i < number_key.size() - help_disappear; i++)
    {
        ch = number_key[i];
        if (ch == '0')
        {
            if (curnode->left)
                curnode = curnode->left;
            else
            {
                true_key += curnode->c;
                curnode = new_head.next->left;
            }
        }
        else if (ch == '1')
        {
            if (curnode->right)
                curnode = curnode->right;
            else
            {
                true_key += curnode->c;
                curnode = new_head.next->right;
            }
        }
    }
```

搜索哈夫曼树，将二进制信息转化成对应的字符。

```
    DFS_delete(new_head.next);

    ofstream foutin(out_file, ios::binary); // 打开输出文件
    if (!foutin) {
        cerr << "Can not open the output file!" << endl;
        return -1;
    }
    foutin << true_key; // 直接将操作好的字符串进行输出
    foutin.close();

    cout << "文件解压已经完成" << endl;

    // 获取结束时间
    QueryPerformanceCounter(&end);

    // 计算程序执行时间
    double duration = static_cast<double>(end.QuadPart - start.QuadPart) / frequency.QuadPart;

    // 输出程序执行时间（以秒为单位）
    cout << "解压执行时间: " << duration << " 秒" << endl;

    return 0;
```

DFS 删除结点，回收内存，输出解码信息并计算程序执行时间。

（5）结点结构体定义情况：

```
//定义哈夫曼树的结点
struct HaffmanNode {
    char c;                 //结点字符
    int flag;               //标志位，是否为最终字符，如果是建树过程中的结点会为1。
    long long freq;         //字符出现的频率
    HaffmanNode* left;      //左子结点
    HaffmanNode* right;     //右子结点
    HaffmanNode* next;      //下一结点(我们采用链表，来方便的进行排序)
};
```

（6）CreateHaffmanTree 函数：

功能:根据给定的链表，给出对应的哈夫曼树：
具体思路：就是由于上面的链表已经按照频率排序了，我们创建一个新结点，将头结点后面的两个结点（频率最小的两个）作为其左子结点和右子结点。然后将其重新排到链表中。
循环执行上面，直到链表中只有一个结点。

具体代码较长，粘贴出来意义也不大，在这里就不再展示了。

（7）CreateNewHaffmanTree 函数

功能:根据给定的 string 数组，给出对应的哈夫曼树：
具体思路：就是遍历 string 数组，根据字符的具体编码，字符‘0’就去左子结点，字符‘1’就去右子结点，没有结点就创建，一直执行就重建了哈夫曼树

具体代码较长，粘贴出来意义也不大，在这里就不再展示了。

（8）DFS 函数

功能:DFS 搜索哈夫曼树，然后给出每个字符的 01 编码

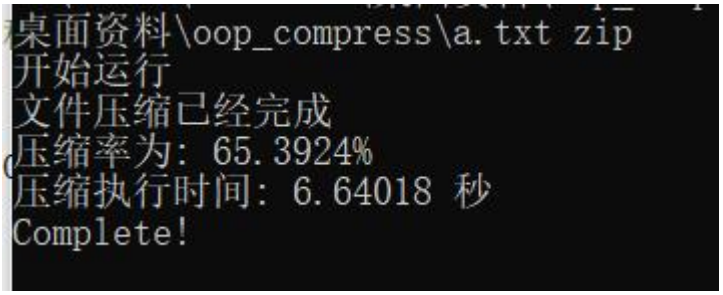
```
//确定各字符的编码，求得对应的字符编码
void DFS(HaffmanNode* node, string ch_code[], string str) {

    if (node->flag == 0)
    {
        ch_code[int(node->c)] = str;
    }

    if (node->left)
        DFS(node->left, ch_code, str + '0');

    if (node->right)
        DFS(node->right, ch_code, str + '1');

    return;
}
```

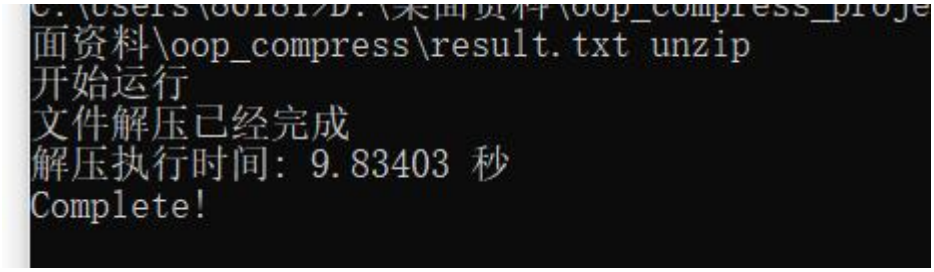
（9）DFS_delete 函数

功能:DFS 搜索哈夫曼树，删除动态申请的结点。

```
//在建树过程中使用了动态内存申请，需要DFS查找删除结点
void DFS_delete(HaffmanNode* node)
{
    if (node->left)
        DFS_delete(node->left);

    if (node->right)
        DFS_delete(node->right);

    else
        delete node;

    return;
}
```

（10）DFS_delete 函数
结果展示：





二、问题与解决方法

1. 遇到的问题

整体上项目并不难，只需要将原来写的东西略微修改就可以了，遇到的问题都是比较小的问题。
（1）数据转换的问题。
（2）位移符的优先级问题。

2. 解决的方法

（1）数据转换的问题。

在过程中我们会涉及到 int 和 char 的来回转换，在 char 转 int 的时候，对于 1111 1111 上面会转化为-1 而非 255。在编写过程中出现了这个小问题，稍微注意一下解决就可以了。

（2）位移符的优先级问题。

下面是现在的代码，

```
//读取辅助信息
finin >> help_disappear >> help_num;
int str_num = 0;
string str;
string str_code[256];
for (int i = 0; i < help_num; ++i)
{
    finin >> str_num >> str; // 读取字符和对应的编码
    str_code[str_num] = str;
}
```

下面是出错的代码：

```
finin >> help_disappear >> help_num;
int str_num = 0;
string str_code[256];
for (int i = 0; i < help_num; ++i)
{
    finin >> str_num >> str_code[str_num]; // 读取字符和对应的编码
}
```

我们是希望读取第一个字符数，并将其转换到对应 string 数组的，本来是想先读 str_num，再用 str_num 修改对应的 str_code[str_num]，但是在位移符的优先级时，是会先读后面的 str_code[str_num]，这里导致了错误。

上面的问题主要都是基础知识的遗忘，温故而知新。

三、心得体会

这次大作业总体来说并不难，使用的是 Haffman 树，在数据结构中也经常学到，在刚开始的时候并不困难，但是总的来说还是比较顺利的。

同时老师的准备也很充分。这次实验本就实用性，类似于一个文件压缩器。老师也提供了很多的资料，比较喜欢的是一个学长写的“压缩方法简介”，看着增长了很多见识。

希望各位助教手下留情，嘻嘻。

四、源代码

```
#include <fstream>
#include <iostream>
#include <string>
#include <cmath>
#include <Windows.h>
using namespace std;

//定义哈夫曼树的结点
struct HaffmanNode {
    char c;                 //结点字符
    int flag;               //标志位，是否为最终字符，如果是建树过程中
的结点会为1。
    long long freq;         //字符出现的频率
    HaffmanNode* left;      //左子结点
    HaffmanNode* right;     //右子结点
    HaffmanNode* next;      //下一结点(我们采用链表，来方便的进行排序)
};

//对于给定的链表，建立哈夫曼树。
HaffmanNode CreateHaffmanTree(HaffmanNode head)
{
    //如果链表中只有一个结点了，则说明已经完成了建树
    while (head.next->next)
    {
        //创建新结点
        HaffmanNode* new_node = new HaffmanNode;
        //结点值初始化
        new_node->c = char(0);
        new_node->flag = 1;
        new_node->freq = head.next->freq + head.next->next->freq;
        new_node->left = head.next;
        new_node->right = head.next->next;
        new_node->next = NULL;

        head.next = head.next->next->next;

        new_node->left->next = NULL;
        new_node->right->next = NULL;
```

```
        HaffmanNode* curNode = head.next;

        if (!curNode) {
            head.next = new_node;
            continue;
        }
        else if (curNode->freq > new_node->freq) {
            new_node->next = head.next;
            head.next = new_node;
            continue;
        }

        while (curNode->next && (curNode->next->freq < new_node->freq))
            curNode = curNode->next;

        new_node->next = curNode->next;
        curNode->next = new_node;
    }

    return head;
}

HaffmanNode CreateNewHaffmanTree(HaffmanNode new_head, string str_node[])
{
    HaffmanNode* curNode;
    for (int i = 0; i < 256; ++i)
    {
        if (str_node[i] != "")
        {
            curNode = new_head.next;

            for (int j = 0; j < str_node[i].size(); ++j)
            {
                if (str_node[i][j] == '0')
                {
                    if (curNode->left == NULL)
                    {
                        HaffmanNode* new_node = new HaffmanNode;
                        new_node->c = '0';
                        new_node->freq = 0;
                        new_node->flag = 1;
                        new_node->left = NULL;
                        new_node->right = NULL;
```

```
                    new_node->next = NULL;

                    curNode->left = new_node;
                    curNode = curNode->left;
                }
                else
                    curNode = curNode->left;
            }
            else if (str_node[i][j] == '1')
            {
                if (curNode->right == NULL)
                {
                    HaffmanNode* new_node = new HaffmanNode;
                    new_node->c = '0';
                    new_node->freq = 0;
                    new_node->flag = 1;
                    new_node->left = NULL;
                    new_node->right = NULL;
                    new_node->next = NULL;

                    curNode->right = new_node;
                    curNode = curNode->right;
                }
                else
                    curNode = curNode->right;
            }
        }
        curNode->c = char(i);
        curNode->flag = 0;
    }
}

    return new_head;
}

//字符串翻转函数
string reserve(string str)
{
    string new_str = "";
    for (int i = str.length() - 1; i >= 0; --i) {
        new_str += str[i];
    }
    return new_str;
}
```

```
//确定各字符的编码，求得对应的字符编码
void DFS(HaffmanNode* node, string ch_code[], string str) {

    if (node->flag == 0)
    {
        ch_code[int(node->c)] = str;
    }

    if (node->left)
        DFS(node->left, ch_code, str + '0');

    if (node->right)
        DFS(node->right, ch_code, str + '1');

    return;
}

//在建树过程中使用了动态内存申请，需要DFS查找删除结点
void DFS_delete(HaffmanNode* node)
{
    if (node->left)
        DFS_delete(node->left);

    if (node->right)
        DFS_delete(node->right);

    else
        delete node;

    return;
}

//压缩文件
int zip(string in_file, string out_file)
{
    LARGE_INTEGER frequency;
    LARGE_INTEGER start;
    LARGE_INTEGER end;

    // 获取时钟频率
    QueryPerformanceFrequency(&frequency);

    // 获取开始时间
```

```
    QueryPerformanceCounter(&start);

    long long frequent[256] = { 0 };

    //读文件
    fstream fin(in_file, ios::binary | ios::in | ios::out); // 以二进制方式打开文件
    if (!fin) {
        cerr << "Can not open the input file!" << endl; // 输出错误信息并退出
        return -1;
    }

    istreambuf_iterator<char> begi(fin), endi; // 设置两个文件指针，指向开始和结束，以 char(一字节) 为步长
    string content(begi, endi); // 将文件全部读入 string 字符串

    double originalFileSize = fin.tellg();

    fin.close(); // 操作完文件后关闭文件句柄是一个好习惯


    int ch;
    // 遍历字符串中的每个字符，计算频率
    for (int i = 0; i < content.size(); ++i) {
        ch = int(content[i]);
        if (ch < 0)
            ch += 256;
        frequent[ch]++;
    }

    //根据频率建立相应的结点和链表，并进行排序
    //链表的头结点
    HaffmanNode head;
    head.c = '0';
    head.freq = 0;
    head.flag = 0;
    head.left = NULL;
    head.right = NULL;
    head.next = NULL;

    for (int i = 0; i < 256; ++i)
    {
        if (frequent[i] == 0)
```

```
            continue;

        HaffmanNode* new_node = new HaffmanNode;
        HaffmanNode* curNode = head.next;

        new_node->c = char(i);
        new_node->flag = 0;
        new_node->freq = frequent[i];
        new_node->left = NULL;
        new_node->right = NULL;
        new_node->next = NULL;

        if (!curNode) {
            head.next = new_node;
            continue;
        }
        else if (curNode->freq > new_node->freq) {
            new_node->next = head.next;
            head.next = new_node;
            continue;
        }

        while (curNode->next && (curNode->next->freq < new_node->freq))
            curNode = curNode->next;

        new_node->next = curNode->next;
        curNode->next = new_node;
    }


    //根据链表可以建立哈夫曼树
    head = CreateHaffmanTree(head);


    //定义每个字符的 string 数组
    string ch_code[256];
    //通过 DFS 获得每个字符的编码
    DFS(head.next, ch_code, "");


    //根据前面获得的字符编码的数组，创建一个二进制的结果
    string number_result = ""; //储存文件按照哈夫曼编码的二进制
    string true_result = "";   //储存最终的文件编码结果
    for (int i = 0; i < content.size(); ++i) {
```

```
        ch = content[i];
        number_result += ch_code[int(ch)];
    }

    //由于上面的二进制不一定是8的倍数，所以我们末尾补零，来帮助我们转化成字符
    int disappear;
    if (number_result.size() % 8 == 0)
        disappear = 0;
    else
        disappear = 8 - number_result.size() % 8;

    for (int i = 0; i < disappear; ++i)
        number_result += '0';

    //将上面的二进制转换成对应的字符
    for (int i = 0, number = 0; i < number_result.size(); i += 8)
    {
        number = 0;
        for (int j = 7; j >= 0; j--)
        {
            if (number_result[i + 7 - j] == '0')
                number += 0;
            else
                number += pow(2, j);
        }
        true_result += char(number);
    }

    //下面记录辅助信息
    int help_number = 0;      //有编码的字符数统计
    for (int i = 0; i < 256; ++i)
    {
        if (ch_code[i] != "")
            help_number++;
    }

    fstream fout(out_file, ios::binary | ios::in | ios::out); // 打开输出文件
    if (!fout) {
        cerr << "Can not open the output file!" << endl;
        return -1;
    }
```

```
    //下面是输出辅助信息
    fout << disappear << " " << help_number << "\n";
    for (int i = 0; i < 256; ++i)
    {
        if (ch_code[i] != "")
            fout << i << " " << ch_code[i] << "\n";
    }

    DFS_delete(head.next);


    fout << true_result;
    double compressedFileSize = fout.tellg();

    fout.close();
    cout << "文件压缩已经完成" << endl;

    double compressionRate = (compressedFileSize / originalFileSize) *
100;
    cout << "压缩率为: " << compressionRate << "%" << std::endl;

    // 获取结束时间
    QueryPerformanceCounter(&end);

    // 计算程序执行时间
    double duration = static_cast<double>(end.QuadPart - start.QuadPart)
/ frequency.QuadPart;

    // 输出程序执行时间（以秒为单位）
    cout << "压缩执行时间: " << duration << " 秒" << endl;

    return 0;

}
//解压文件
int unzip(string in_file, string out_file)
{
    LARGE_INTEGER frequency;
    LARGE_INTEGER start;
    LARGE_INTEGER end;

    // 获取时钟频率
    QueryPerformanceFrequency(&frequency);
```

```
    // 获取开始时间
    QueryPerformanceCounter(&start);

    ifstream finin(in_file, ios::binary); // 打开输出文件
    if (!finin) {
        //cerr << "Can not open the input file!" << endl;
        cout << "Can not open the input file!" << endl;
        return -1;
    }

    int help_disappear = 0;
    int help_num = 0;

    //读取辅助信息
    finin >> help_disappear >> help_num;
    int str_num = 0;
    string str;
    string str_code[256];
    for (int i = 0; i < help_num; ++i)
    {
        finin >> str_num >> str; // 读取字符和对应的编码
        str_code[str_num] = str;
    }

    finin.get();
    //读取重要的文本信息放入 information 的 string 中
    istreambuf_iterator<char> begin(finin),
        endin;// 设置两个文件指针，指向开始和结束，以 char(一字节) 为
步长
    string information(begin, endin); // 将文件全部读入 string 字符串

    HaffmanNode new_head;
    new_head.c = '0';
    new_head.freq = 0;
    new_head.flag = 0;
    new_head.left = NULL;
    new_head.right = NULL;
    new_head.next = NULL;

    HaffmanNode* new_node = new HaffmanNode;
    new_node->c = '0';
    new_node->freq = 0;
    new_node->flag = 1;
    new_node->left = NULL;
```

```
new_node->right = NULL;
new_node->next = NULL;

new_head.next = new_node;

new_head = CreateNewHaffmanTree(new_head, str_code);

//下面两个 string，分别储存二级制数据和解压结果
string number_key = "";
string true_key = "";

for (int i = 0; i < information.size(); ++i)
{
    int number = int(information[i]);
    if (number < 0)
        number += 256;
    for (int i = 7; i >= 0; i--)
    {
        if (number >= pow(2, i))
        {
            number_key += '1';
            number -= pow(2, i);
        }
        else
            number_key += '0';
    }
}

char ch;
HaffmanNode* curnode;
curnode = new_head.next;
for (int i = 0; i < number_key.size() - help_disappear; i++)
{
    ch = number_key[i];
    if (ch == '0')
    {
        if (curnode->left)
            curnode = curnode->left;
        else
        {
            true_key += curnode->c;
            curnode = new_head.next->left;
        }
    }
```

```
        else if (ch == '1')
        {
            if (curnode->right)
                curnode = curnode->right;
            else
            {
                true_key += curnode->c;
                curnode = new_head.next->right;
            }
        }
    }

    DFS_delete(new_head.next);

    ofstream foutin(out_file, ios::binary); // 打开输出文件
    if (!foutin) {
        cerr << "Can not open the output file!" << endl;
        return -1;
    }
    foutin << true_key; // 直接将操作好的字符串进行输出
    foutin.close();

    cout << "文件解压已经完成" << endl;

    // 获取结束时间
    QueryPerformanceCounter(&end);

    // 计算程序执行时间
    double duration = static_cast<double>(end.QuadPart - start.QuadPart)
/ frequency.QuadPart;

    // 输出程序执行时间（以秒为单位）
    cout << "解压执行时间: " << duration << " 秒" << endl;

    return 0;
}

//主函数
int main(int argc, char* argv[]) {
    cout << "开始运行" << endl;
    if (argc != 4) {
        cerr << "请确保输入的参数个数是否正确" << endl;
        return -1;
    }
```

```
    if (!strcmp(argv[3], "zip")||!strcmp(argv[3], "unzip")) {
        if (!strcmp(argv[3], "zip"))
        {
            int number = 0;
            number = zip(argv[1], argv[2]);
            if (number == -1)
                return 0;
        }
        else if (!strcmp(argv[3], "unzip"))
        {
            int number = 0;
            number = unzip(argv[1], argv[2]);
            if (number == -1)
                return 0;
        }
    }
    else {
        cerr << "Unknown parameter!\nCommand list:\nzip\nunzip" << endl;
        return -1;
    }

    cout << "Complete!" << endl;
    return 0;
}
```