一、代码结构概要说明；

代码的主体结构分为实现哈夫曼结点和哈夫曼树功能的 huffman.h 和实现压缩解压功能的 function.h 两大类。

```
//哈夫曼结点
class HuffmanNode{
private:
    unsigned char ascii;
    long long int weight;
    string code;
    HuffmanNode* lchild;
    HuffmanNode* rchild;
public:
    HuffmanNode(){};
    HuffmanNode(unsigned char _ascii,long long int weight);//构造函数
    void set_ascii(int _ascii);//设置ascii值
    unsigned char get_ascii();//获取ascii值
    void set_weight(long long int _weight);//设置权重
    void add_weight();//权重加一
    long long int get_weight();//获取权重
    void set_code(string _code);//设置哈夫曼编码
    string get_code();//获取哈夫曼编码
    void set_lchild(HuffmanNode* _lchild);//设置左孩子节点
    HuffmanNode* get_lchild();//获取左孩子节点
    void set_rchild(HuffmanNode* _rchild);//设置右孩子节点
    HuffmanNode* get_rchild();//获取右孩子节点
    friend bool operator < (HuffmanNode A,HuffmanNode B){//利用重载的友元函数定义优先数列的优先级
        return A.get_weight()>B.get_weight();
    }
};
```

```
//哈夫曼树
class HuffmanTree{
private:
    priority_queue<HuffmanNode> HuffmanQueue;//优先队列实现哈夫曼结点的排序
public:
    HuffmanNode* root;//根节点
    HuffmanNode HuffmanArray[256];//用来存储256个ascii码相关的信息
    HuffmanTree();//构造函数
    void getweight(string path);//获得ascii的权重
    void creatHuffmanTree();//创建哈夫曼树
    void getHuffmanCode(HuffmanNode* node,string code);//获得哈夫曼编码
    void HuffmanCompress(string path,ofstream &fout);//哈夫曼压缩
    void HuffmanDecompress(string filename,ifstream& fin,long long int len);//哈夫曼解压
};
```

```
class function{
public:
    function(){};
    void getfilepath(string filename,vector<string> &filepaths);//获得文件夹中所有文件或空文件夹的路径
    void filecompress(string filename);//单个文件的压缩
    void foldercompress(string filename);//文件夹的压缩
    void compress(string filename);//压缩
    void decompress(string filename);//解压
};
```

对应函数的具体实现分别位于对应的 cpp 文件中。

二、项目“核心需求”与“其他需求”中每个评分项的设计、实现思路的大致描述；

“核心需求”的实现；

(1)　文件的压缩与解压

对于单个文件的压缩，实现思路为根据文件的字节流统计每个 ascii 码出现的次数作为权重，根据 256 个 ascii 码的权重构建哈夫曼树，获得每个 ascii 码对应的哈夫曼编码，再重新读入文件的字节流，将对应的字节的哈夫曼编码写入压缩文件中即可。此外，需要将文件名和文件是否为空的信息也写入压缩文件中，这里的做法是，将文件名的长度和文件名以及压缩后文件的字节数写入压缩后的文件中。

```
void function::filecompress(string filename){
    //自定义压缩后的文件名
    string newfilename;
    cout<<"please enter the new filename after compression!"<<endl;
    cin>>newfilename;
    newfilename+=".huf";
    ofstream fout(newfilename,ios::binary);
    //以sign为标志，若sign=0则为文件的压缩；若sign=1则为文件夹的压缩
    int sign=0;
    fout.write((char*)&sign,sizeof(int));
    //写入文件名的长度以及文件名
    int filenamelength=filename.size();
    fout.write((char*)&filenamelength,sizeof(int));
    for(int i=0;i<filenamelength;i++){
        fout.write((char*)&filename[i],sizeof(char));
    }
    //写入文件的长度、ascii数组以及文件的哈夫曼编码
    HuffmanTree tree;
    tree.HuffmanCompress(filename,fout);
    fout.close();
    cout<<"End of compression!"<<endl;
}
```

解压时，先读取根据文件的长度读取原来的文件名，然后读取压缩文件的字节数，如果其为 0，利用 ofstream 创建一个空文件即可，否则，先读取 256 个 ascii 码的权重来构建哈夫曼树，然后读取压缩文件的字节流来获取对应的 ascii 并写入解压后的文件中。

```
//sign=0,解压的是单个文件
if(sign==0){
    //获得文件的名字
    int filenamelength;
    fin.read((char*)&filenamelength,sizeof(int));
    string filename("");
    while(filenamelength--){
        char ch=fin.get();
        filename+=ch;
    }
    //读取哈夫曼编码的长度
    long long int filelength;
    fin.read((char*)&filelength,sizeof(long long int));
    HuffmanTree tree;
    //解压
    tree.HuffmanDecompress(filename,fin,filelength);
    fin.close();
}
```

(2) 文件夹的压缩与解压

对于文件夹的压缩，我们需要在压缩时写入一个标志来作为判断压缩的是文件还是文件夹，这里的做法是在压缩文件开头申请一块 sizeof(int)的空间，如果是文件则写入 0，是文件夹则写入 1，然后我们需要得到并统计文件夹中所有文件和空文件夹的路径并存入，最后按照单个文件的压缩做法利用遍历写入所有文件的相关信息即可。

```cpp
void function::foldercompress(string filename){
    //获得所有文件以及空文件夹的路径
    vector<string> filepaths;
    getfilepath(filename,filepaths);
    //自定义压缩后的文件夹名
    string newfilename;
    cout<<"please enter the new foldername after compression!"<<endl;
    cin>>newfilename;
    newfilename+=".huf";
    //sign=1作为文件夹的标志
    ofstream fout(newfilename,ios::binary);
    int sign=1;
    fout.write((char*)&sign,sizeof(int));
    //写入路径的总数以及各路径的长度以及其本身
    int filenum=filepaths.size();
    fout.write((char*)&filenum,sizeof(int));
    for(int i=0;i<filenum;i++){
        int len=filepaths[i].size();
        fout.write((char*)&len,sizeof(int));
        for(int j=0;j<len;j++){
            fout.write((char*)&filepaths[i][j],sizeof(char));
        }
    }
    //以for循环来压缩各文件
    for(int k=0;k<filenum;k++){
        int j=k+1;
        cout<<j<<"\\"<<filenum<<endl;
        //判断是否为文件
        if(filepaths[k].find('.')!=filepaths[k].npos){
            HuffmanTree tree;
            tree.HuffmanCompress(filepaths[k],fout);
        }
    }
    fout.close();
    cout<<"End of compression!"<<endl;
}
```

对于解压的做法是先判断解压的是文件还是文件夹，如果是文件夹，则读取所有路径的信息，然后遍历所有路径，如果是空文件夹，则创建该路径上的所有文件夹，如果是文件，则参考单个文件的做法。

```
//sign=1,解压的是文件夹
else if(sign==1){
    //读取各个路径
    int filenum;
    fin.read((char*)&filenum,sizeof(int));
    vector<string> filepaths;
    while(filenum--){
        int filepath_length;
        fin.read((char*)&filepath_length,sizeof(int));
        string filepath("");
        while(filepath_length--){
            char ch=fin.get();
            filepath+=ch;
        }
        filepaths.push_back(filepath);
    }
    //对每个路径做解压
    int num=filepaths.size();
    for(int i=0;i<num;i++){
        int j=i+1;
        cout<<j<<"\\"<<num<<endl;
        //文件
        if(filepaths[i].find('.')!=filepaths[i].npos){
            //创建文件路径上的文件目录
            int m=filepaths[i].find_last_of('\\');
            string str=filepaths[i].substr(0,m);
            filesystem::path entry(str);
            if(!filesystem::exists(entry)){
                filesystem::create_directories(entry);
            }
            long long int filelength=0;
            fin.read((char*)&filelength,sizeof(long long int));
            HuffmanTree tree;
            tree.HuffmanDecompress(filepaths[i],fin,filelength);
        }
        //空文件夹
        else{
            filesystem::path entry(filepaths[i]);
            filesystem::create_directories(entry);
        }
```

“其他需求”的实现：

（1）使用 CLI 与用户交互

未能实现。（无奈）

（2）检验压缩包来源是否是自己的压缩工具

对于这个问题我最初的想法是可以在压缩文件的开头留出一块空间存储特定的标志来判断压缩包是否来源与自己的压缩工具，类似于判断文件还是文件夹的标志，由于 PJ 文档中只提到了后缀名错误的问题，因此我在处理时未做这一步，只是简化只在压缩时确保压缩后的文件名的后缀均为“.huf”来判断是否来源于自己的压缩工具。

（3）文件覆盖问题

对于这个问题，我的做法是对于所有要压缩和解压的文件名和文件路径利用 filesystem::exists()判断文件是否存在，若存在，则做一个交互，根据用户的输入

来判断继续压缩解压还是停止。

```
filesystem::path entry(filename);
if(filesystem::exists(entry)){
    cout<<filename<<" already exists, whether to overwrite the original file. enter 0 to overwrite 1 to stop."<<endl
    int flag;
    cin>>flag;
    if(flag==1){
        return;
    }
}
```

（4）压缩包预览

在压缩时已经将各文件和空文件夹的路径放在压缩文件的前面，类似的输出为：

```
PS D:\Desktop\PJ\build> ."D:/Desktop/PJ/build/main_cmake.exe"
testcase5NomalFolder\1\6937154.htm
testcase5NomalFolder\1\6957767.htm
testcase5NomalFolder\1\6967566.htm
testcase5NomalFolder\1\6970070.htm
testcase5NomalFolder\1\7006039.htm
testcase5NomalFolder\1\7009495.htm
testcase5NomalFolder\1\7009526.htm
testcase5NomalFolder\1\7010647.htm
testcase5NomalFolder\2\20030187787.htm
testcase5NomalFolder\2\5539775.htm
testcase5NomalFolder\2\5539775.htm.stzip
testcase5NomalFolder\2\5629981.htm
testcase5NomalFolder\2\5784686.htm
testcase5NomalFolder\2\5825806.htm
testcase5NomalFolder\2\5890199.htm
testcase5NomalFolder\2\5974078.htm
testcase5NomalFolder\2\6046683.htm
testcase5NomalFolder\2\6121544.htm
testcase5NomalFolder\2\6130623.htm
testcase5NomalFolder\2\6184841.htm
testcase5NomalFolder\2\6243012.htm
testcase5NomalFolder\2\6249227.htm
testcase5NomalFolder\2\6264106.htm
testcase5NomalFolder\2\6266362.htm
testcase5NomalFolder\2\6286762.htm
```

输出为这个样子。

三、开发环境/工具，以及如何编译/运行项目；

开发环境/工具为 C++和 VScode,通过编写 CMakeLists.txt 编译运行项目

CMakeLists.txt

```
1  cmake_minimum_required(VERSION 3.0)
2
3  project(PJ)
4
5  include_directories(include)
6
7  add_executable(main_cmake main.cpp src/huffman.cpp src/function.cpp)
```

四、性能测试结果（表格记录每个测试用例的初始大小、压缩后大小、压缩率、压缩时间、解压时间）；

	A	B	C	D	E	F	G	H	I	J	K	L
			压缩前大小		压缩后大小		压缩率		压缩时间		解压时间	
	testcase1		0		1kb		\		2ms		2ms	
	testcase2		22.8MB		16.7MB		73.40%		3.052s		2.722s	
	testcase3		1.02GB		676.6MB		64.77%		118.65s		115.57s	
	testcase4		0		1kb		\		2ms		2ms	
	testcase5		5.7MB		4.1MB		71.85%		0.922s		0.889s	
	testcase6		427MB		434.7MB		101.80%		116.758s		bug	
	testcase7		1.02GB		666MB		63.80%		116.48s		112.45s	
	testcase8		613MB		392.6MB		64.05%		72.498s		63.188s	
)	testcase9		412MB		264.2MB		64.13%		48.614s		43.128s	
L												

五、与其他压缩工具的压缩率和压缩时间比较；

	A	B	C	D	E	F	G	H	I	J	K
	testcase02NormalSingleFile			压缩前大小		压缩后大小		压缩率		压缩时间	
	huf			22.8MB		16.7MB		73.40%		12,796s	
	zip			22.8MB		10.4MB		45.80%			
	7z			22.8MB		8.6MB		37.60%			
	testcase5NormalFolder										
	huf			5.7MB		4.1MB		71.80%		0.922s	
	zip			5.7MB		2.2MB		38.10%			
	7z			5.7MB		2.0MB		35.10%			
0											
1	testcase07XlargeSubFolders										
2	huf			1.02GB		666MB		63.80%		112.45s	
3	zip			1.02GB		177.4MB		16.90%			
4	7z			1.02GB		122.3MB		11.70%			
5											

六、遇到的问题和解决方案

遇到的问题：

1.CLI 未能实现，对于时间的测量用 clock_t 实现

2.测试用例 6 在解压时出现问题