数据结构 PJ 开发文档

课　程：数据结构
项　目：基于哈夫曼编码的压缩解压缩程序
实验者：曹 琦
学　号：22307110076
日　期：2023 年 12 月 2 日

一、引言

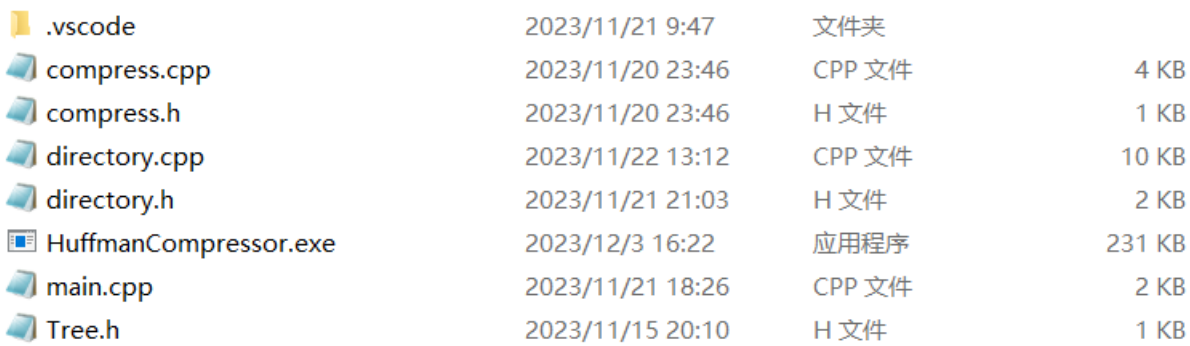
本文档旨在完整展示我的课程 Project（复旦大学软件学院 23 秋数据结构）的完成情况。该项目是一个使用 c++语言，基于霍夫曼算法，实现了任意文件（夹）的无损压缩与解压缩功能的小程序，通过控制台与用户交互。

文档有以下几个部分：**代码结构简述**（介绍各项目文件的关系，各个类的功能与关系）、**具体功能实现思路**（描述项目需求每个评分项的设计与实现思路）、**开发环境与项目部署**（介绍开发项目的平台与工具，如何编译运行该项目）、**性能测试**（记录给出的测试用例的测试参数）、**与其他工具比较**（与市面上成熟的压缩解压工具比较性能，并尝试分析原因）、**问题与解决**（介绍开发过程中遇到的部分问题与解决思路）。

在开始文档的正式内容前，我要衷心感谢数据结构课程的袁助教和吴助教在本项目完成过程提供的指导、帮助与支持。助教老师在项目的各个阶段提供了耐心的帮助与建议，对项目完成起到的相当重要的作用。

二、代码结构简述

（一）文件构成：

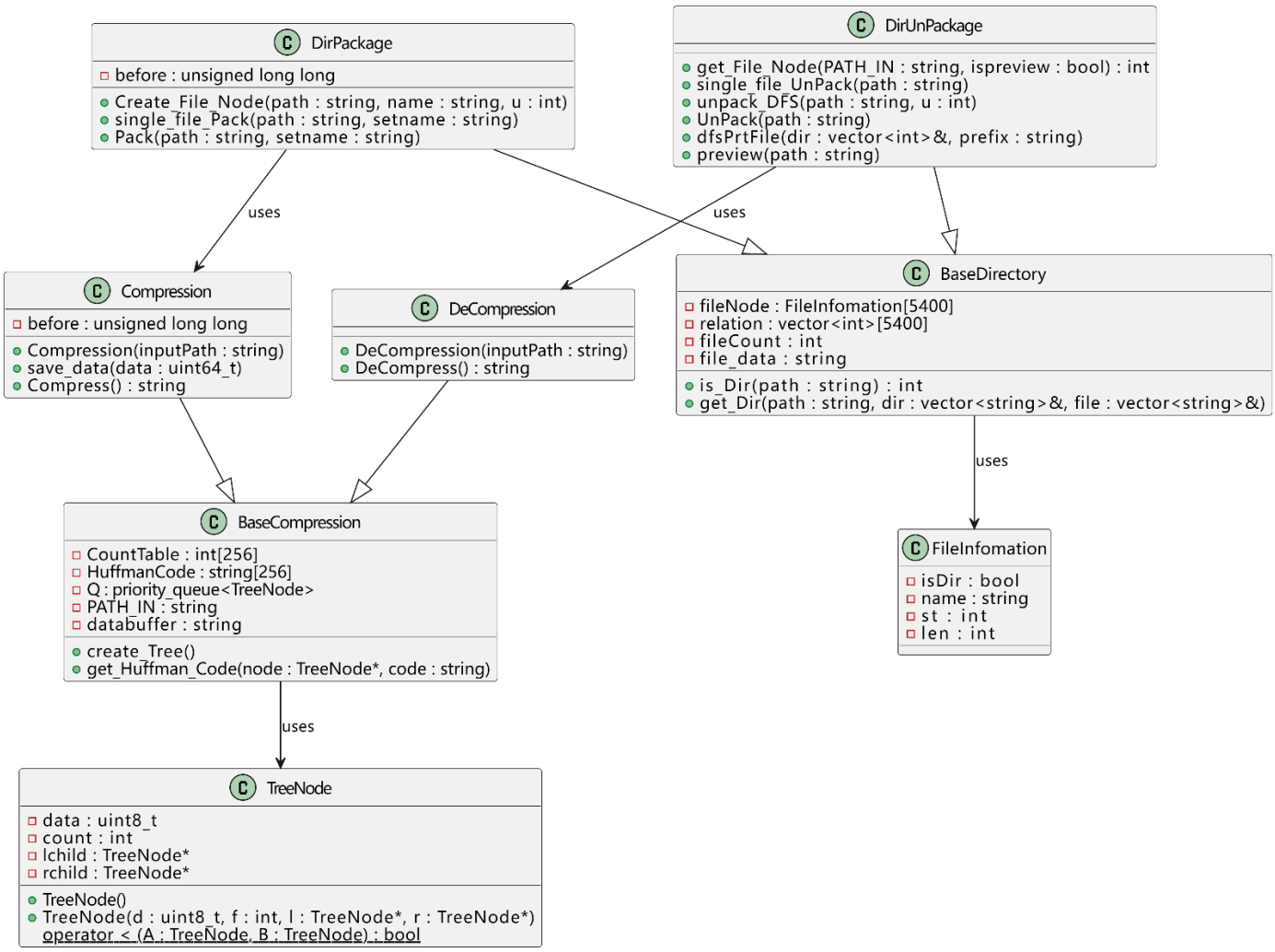


.vscode	2023/11/21 9:47	文件夹	
compress.cpp	2023/11/20 23:46	CPP 文件	4 KB
compress.h	2023/11/20 23:46	H 文件	1 KB
directory.cpp	2023/11/22 13:12	CPP 文件	10 KB
directory.h	2023/11/21 21:03	H 文件	2 KB
HuffmanCompressor.exe	2023/12/3 16:22	应用程序	231 KB
main.cpp	2023/11/21 18:26	CPP 文件	2 KB
Tree.h	2023/11/15 20:10	H 文件	1 KB

- **.vscode**：Visual Studio Code 编辑器的一个特殊文件夹，用于存放与项目相关的编辑器配置和设置。在 VSCode 中打开一个文件夹作为项目时，会自动加载其中的配置文件来定制编辑器的行为，包括编译时各个文件的依赖关系。包含此文件夹后可在 VSCode 中直接一键编译运行该项目。
- **Tree.h**：定义了 TreeNode 类。
- **Compress.h**：定义了 BaseCompression 类和由其派生出的 Compression 类和 DeCompression 类。
- **Directory.h**：定义了 BaseDirectory 类和由其派生出的 DirPackage 类和 DirUnPackage 类。
- **Compress.cpp**：定义了 compress.h 头文件中声明的一系列函数。
- **Directory.cpp**：定义了 directory.h 头文件中声明的一系列函数。
- **Main.cpp**：主函数，实现了通过控制台窗口与用户交互的功能。
- **HuffmanCompressor.exe**：该项目的可执行程序。

（二）类的构成与简介：

类图：



类的简介：

（简要介绍了每个类的具体作用，解释了一部分重要的成员变量及每个成员函数的作用）

（1） **TreeNode 类**：

定义了哈夫曼树的节点。类重载了节点的小于号，便于使用优先队列构造哈夫曼树；uint_8 data 为特定字符的 8 位,count 为该字符出现的次数。

（2） **BaseCompression 类**：

文件压缩与解压缩类的基类。定义了一些压缩与解压缩单个文件共同的数据与操作。其中 CountTable 数组用于记录文件中每个字符的出现次数，最后会存入文件；Huffmancode 记录每个字符的哈夫曼编码；Q 是构造哈夫曼树时使用的优先队列，最后用于保存哈夫曼树的根节点；databuffer 是用于记录文件数据的缓冲区。

函数 create_Tree()函数根据 counttable 的数据构造一个哈夫曼树，其根节点保存在 Q 中；

函数 get_Huffman_Code()函数根据刚构造的哈夫曼树构造哈夫曼编码表，保存在 Huffmancode 数组中。

（3） Compression 类：

单文件哈夫曼压缩类，由 DirPackage 类调用使用。定义了压缩单个文件需要的函数。unsigned long long before 记录未压缩前文件长度，用于后续计算压缩率。

save_data()函数用于向 databuffer 中写入经过哈夫曼编码后的数据。

Compress ()函数，返回压缩后文件的完整数据 databuffer，后续由 DirPackage 类写入文件。

（4） DeCompression 类:

单文件哈夫曼解压类，由 DirUnPackage 类调用使用。

DeCompress()函数，返回解压后的完整数据 databuffer，后由 DirUnPackage 类将数据写入文件。

（5） Fileinformation 类：

记录文件夹和子文件的信息。用于在解压时还原原本的文件夹结构。包含是否是文件夹、文件名、文件长度、文件在数据区的起始地址。

（6） BaseDiretory 类：

文件夹处理基类。定义了一些压缩与解压缩文件夹共同的操作与数据。filenode 数组记录所有文件的关键信息；relation 使用 vector 数组实现了一个邻接表结构记录文件夹的树状结构。file_data 是用于写入最终压缩文件的缓冲区。

is_Dir()函数用于判断输入的路径是否是一个文件夹，是则返回 0，不是则返回 1，当前路径不存在则返回-1；

get_Dir()函数用于获取当前文件夹下的内容，读取当前文件夹，把其中的子文件夹的名字存入 vector<string>&dir，把子文件的名字存入 vector<string>&file；

（7） DirPackage 类：

文件夹打包类。实现了文件夹以及单个文件的打包压缩功能。unsigned long long before 记录未压缩前文件夹下所有文件的总长度，用于后续计算压缩率。

Create_File_Node()函数根据输入的路径递归的遍历整个文件夹，获取文件中每个文件的信息存入成员变量 filenode 数组和 relation 邻接表中，再调用 Compression 类压缩每个文件。

single_file_Pack()函数，当 Pack()函数发现要压缩是的单个文件而不是文件夹时会调用这个函数单独处理，以压缩单个文件。

Pack()函数压缩整个文件夹，函数通过递归使用 create_file_node()函数，一遍存储文件夹结构信息，一边压缩文件，并把数据输出到指定的.hfmp 文件（压缩包文件）中。

（8） DirUnPackage 类：

文件夹解包类。实现了文件夹以及单个文件的解压缩功能以及文件不解压的前提下实现预览功能。

int get_File_Node()函数读取压缩包文件头的信息，并把信息存入成员变量 filenode 数组和 relation 邻接表中，在后续的解压缩或预览时发挥作用。返回值情况：0:单文件 1:多文件 2:未知来源文件 3:预览模式；

void single_file_UnPack()函数，当 UnPack()函数发现要解压是的单个文件而不是文件夹时会调用这个函数单独处理，以解压单个文件；

void unpack_DFS()函数使用深度优先遍历的算法，递归地解压子目录中的所有文件，解压时调用 DeCompression 类;

void UnPack()函数用于解压整个压缩包文件，先调用 get_File_Node 函数获取压缩包文件夹结构的信息，再调用 unpack_DFS()函数递归的解压整个文件;

void dfsPrtFile()函数，使用深度优先遍历的递归算法，输出邻接表中存储的文件夹结构，递归时不断改变输出前缀来实现树状结构的视觉效果;

void preview()函数用于预览文件夹结构，先调用 get_File_Node 函数获取压缩包文件夹结构的信息，再调用 dfsPrtFile()函数递归的输出所有内容。

三、具体功能实现思路简述

(一)文件的压缩与解压

单个文件压缩与解压：首先用二进制打开文件（确保该路径是一个存在的单文件），逐个 byte 读取文件的每一个信息，把 256 种字符的出现次数记录在一个数组中，再记录文件一共有多少个字符。然后，根据每个字符出现的次数构造一颗哈夫曼树，使用优先队列的数据结构，每次弹出频率最低的两个节点。根据构造出的哈夫曼树使用递归的方法得出每个字符对应的哈夫曼编码，并把编码存在一个数组中。

完成了以上前置工作，可以开始文件的压缩。首先向压缩包文件中写入一行校验码 114514（用于检查文件来源），再向压缩文件中写入一个 0（说明该压缩包是单个文件压缩来的而不是文件夹），再向压缩文件中写入原文件名（便于解压缩时恢复），再向其中写入原文件字符总数和记录每个字符出现次数的数组。以上信息构成了压缩包文件的文件头，再来写入原文件的压缩后数据。再次打开原文件，逐个读取每个字符，将当前字符的对应哈夫曼编码通过掩码和位运算（核心代码见下图）的方法写入一个 8bit 的缓冲区，当缓冲区满 8 位时便向压缩包文件写入一个字符，读到最后一个字符时用 0 补全 8bit 输出最后一个字符。关闭文件，单个文件压缩完成。

```
while(!sourcefile.eof()){
    int length=HuffmanCode[data].length();
    for(int i=0; i<length; i++){
        if(HuffmanCode[data][i]=='1')
            temp=temp|(1<<index);
        index++;
        if(index==8){
            buffer.write((char*)&temp,1);
            index=0;
            temp=0;
        }
    }
    data=sourcefile.get();
}
```

解压缩时，首先打开压缩包文件，读取文件头中的信息，出现任何异常时将返回并打印错误信息。再由读出的原文件名，在输入路径下创建一个新文件，后续将往此文件中写入解析出来的信息。根据文件头中读出的每个字符出现的频率的数组，用压缩文件时构造霍夫曼树相同的算法构造一棵与压缩时完全相同的霍夫曼树，再根据霍夫曼树得出每个字符对应的哈夫曼编码。完成前置工作，开始解压缩数据区，逐个 byte 读取压缩包文件的数据区，根据其每一位的 0 和 1 向下查找霍夫曼树，直到找

到叶子节点便向目标文件中写入一个该叶子节点对应的字符，直到文件结束。关闭文件，单个文件解压缩完成。

处理空文件：按照上述方法处理空文件不存在特殊性，故不需要特殊处理。实际实现时发现空文件在构造霍夫曼树时会出现错误，原因是将出现次数为 0 的字符也当做霍夫曼树的叶子节点加入优先队列会造成一些难以预测问题，在构造哈夫曼树的函数中，在创建叶子节点并入队的时候加一条判断语句后问题就不再出现。

指定压缩包名称，解压时还原原本文件名：在压缩文件时，会向压缩包文件的头部写入一些信息，只要在这部分信息中记录原文件的名字，在解压缩时读取出来并根据这个名字创建新文件即可实现还原文本文件名的功能。由于这个过程不依赖压缩包文件的文件名，故可以有用户指定压缩包文件名，甚至之后对压缩包文件的文件名多次修改，也不会影响。

以上单个文件的压缩与解压缩需要的数据和函数被封装在了 compression 类和 decompression 类中，后续处理文件夹时会直接调用者两个类。

(二)文件夹的压缩与解压

压缩解压深度不确定的文件夹：首先判断当前路径指定的是一个文件夹还是单个文件，若是单个文件则直接调用单个文件的压缩类，若是一个文件夹则进行后续操作。对于一个文件夹，首先要读取并保存其树状逻辑结构，通过广度优先遍历的递归算法先扫描一遍文件夹下的内容，通过邻接表保存下来这种树状结构。此外要将整个文件夹下的所有内容保存到一个压缩包中，就要记录好每个文件在数据区的起始地址和长度，这样才能把混在一起的数据分离开来，故创建了一个结构数组，结构体重保存了一个文件（夹）的如下信息：是否是文件夹、文件名、文件数据长度、在压缩包数据区的起始位置。以上数据都会被写入压缩文件的文件头部分。再采用广度优先遍历的算法，遇到文件则压缩，遇到文件夹则进入，逐个调用 compression 类中单个文件的压缩函数，把压缩后的数据暂时存入缓冲区，完成压缩后一并输入到压缩文件包中。

解压缩时，先读出压缩包文件的文件头信息，还原出 file_node 数组和 relation 邻接表，根据这些这些数据递归地创建文件夹与其下的文件。由于 file_node 中保存了文件名和其在数据区的起始位

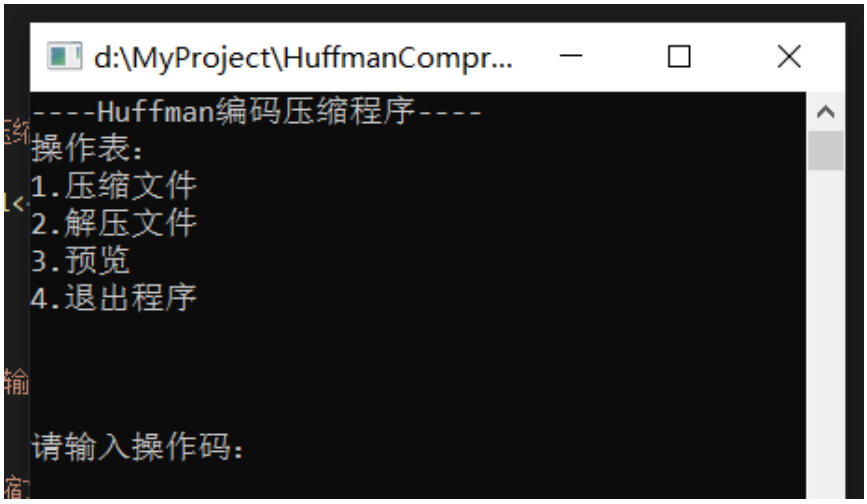
置，故可以找到文件数据所在位置，调用 DeCompression 类对这些霍夫曼编码解析并把解析后的内容输出到目标文件中。

处理空文件夹：按照上述方法处理空文件夹不存在特殊性，故不需要特殊处理。

还原原本的文件名与文件夹名：同理，由于已经在文件头的部分中 file_node 数组中存储了每个文件夹和文件的名字，故可以在解压时完整还原出来。

(三)用户交互

使用控制台交互：



进入程序后现在屏幕上打印操作表，然后进入一个 while(1)的死循环等待用户输入 1,2,3,4 四种指令，接受其他任何指令都会输出“无效操作码”作为提示，当用户输入操作码后，程序会等待用户输入要操作的文件路径，以及指定目标文件名。操作完成后输出成功提示和一些信息。然后进入下一轮循环，等待用户操作。

不断等待用户输入：使用一个死循环实现，除非用户输入退出程序的指令，不然程序完成一轮操作后就会进入下一轮操作，等待用户输入。

显示压缩时间：通过调用 c++标准库<chrono>头文件中提供的计时函数来实现。把用户完成输入，开始压缩的时间点，作为计时的起点；结束压缩，输出完成提示的时间点作为计时的终点。在输出完成提示的后面，输出经过的时间间隔，最小单位为 ms。

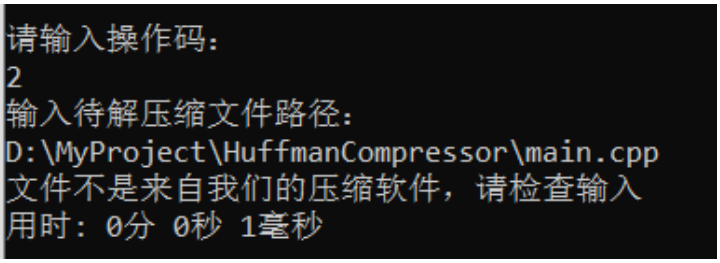
显示压缩率：在压缩文件前会先遍历整个文件获取其长度（字符总数），将这个长度保存下来。压缩完成后，调用函数读取压缩后文件包的总长度，将压缩前后的文件字符总数做除法再输出即可。使用 unsigned long long 来保存数据，故只要文件大小不超过 16777216TB 就不会有溢出的风险。

(四)检查压缩包来源

创建压缩包文件时，程序会在用户指定的文件名后添加后缀“.hfmp”，这仅起到一个标识作用。在压缩文件时，程序会向目标文件的第一行写入一个 int 型的常量“114514”作为验证码，在解压文件时，首先读取该验证码，若不是“114514”则说明文件不是来源于我们的压缩软件。此时解压函数会输出“文件不是来自我们的压缩软件，请检查输入

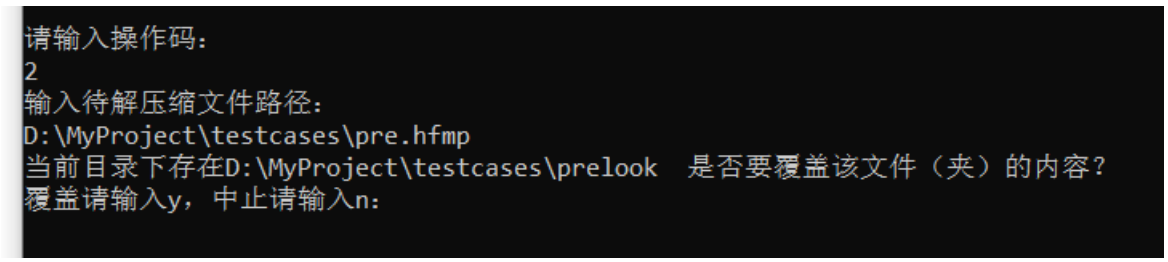
”并直接返回，进入下一次循环，等待用户继续输入指令。故即使是用户修改了“.hfmp”的后缀名，程序依然可以正常解压来自该程序的压缩文件。

虽然不排除其他文件的开头一行也会出现 114514 导致程序错误的把它认为是来自于我们的程序，但这种概率实在是太小太小了，以至于完全可以当做不可能事件。



(五)文件覆盖问题

压缩文件时，产生压缩包名字由用户指定，在创建新文件前，先调用函数判断该目录下是否已经存在同名文件，若已经存在则输出信息，询问用户是否要覆盖，并等待用户输入决定，若确定覆盖则继续执行程序，若不覆盖则压缩函数直接返回，并输出停止信息。



解压文件时，程序会还原出原文件的文件名，故在创建新文件时，先调用函数判断该路径是否存在同名文件夹，逻辑与上述过程相同。

```
void DirPackage::Pack(string path,string setname){
    if(is_Dir(path)==-1){cout<<"路径不存在，请检查输入"<<endl;return;}
    if(is_Dir(path)==0){single_file_Pack(path,setname);return;}
    stringstream buffer;
    string PATH_OUT=path;
    PATH_OUT.replace(PATH_OUT.find_last_of('\\')+1,std::string::npos, setname+".hfmp");//设置输出路径
    if(is_Dir(PATH_OUT)!=-1){
        cout<<"当前目录下存在"<<PATH_OUT<<"  是否要覆盖该文件（夹）的内容？"<<endl<<"覆盖请输入y，中止请输入n: ";
        char op;
        cin>>op;
        if(op!='y') {cout<<"中止成功"<<endl;return;}
    }
```

(六)压缩包预览

在压缩文件夹时，程序向压缩包文件的开头部分写入一些信息，预览时只需要读取出这一部分信息，并不用读取压缩包文件后续的数据区域。再根据这些信息，递归的输出文件名，即可实现预览，但要实现树状结构的视图，则需要修改递归的函数，使之在输出前文件名前先输出一段前缀，具体方法后述。

```
221 void DirUnPackage::preview(string path){
222     int whatcase=get_File_Node(path,1);
223     cout<<"————————————————————————"<<endl;
224 
225     //单个文件情况
226     if(whatcase==2){return;}
227     if(whatcase==0){
228     ifstream sourcefile(path,ios::binary);
229     string name;
230     sourcefile>>name;
231     sourcefile>>name;
232     sourcefile>>name;
233     sourcefile.close();
234     cout<<"单文件: "<<name<<endl;
235     cout<<"————————————————————————"<<endl;
236     return;
237     }
238 
239     //文件夹情况 循环输出树状结构
240     cout<<fileNode[1].name<<endl;
241     dfsPrtFile(relation[1],"├──");
242     cout<<endl;
243     cout<<"————————————————————————"<<endl;
244     return;
245 }
```

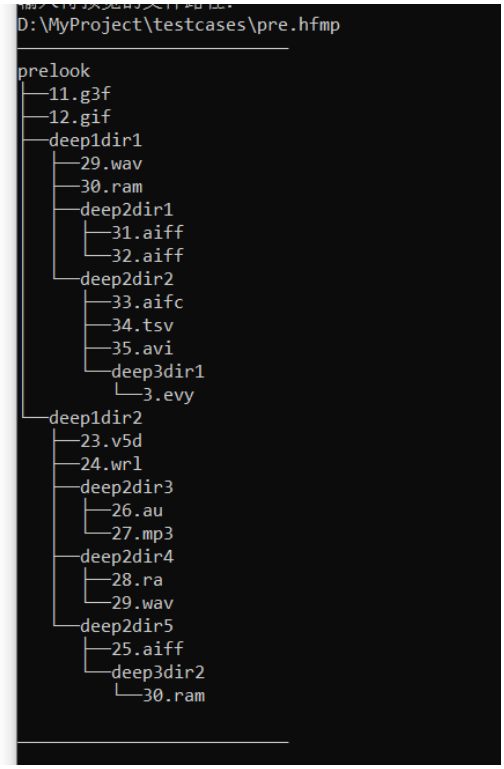
```
247 void DirUnPackage:: dfsPrtFile(vector<int> &dir,string prefix){
248     for(int i=0;i<dir.size();i++){
249         if(i==dir.size()-1){
250             prefix.erase(prefix.size()-9);
251             prefix=prefix+"└──";
252         }
253         cout<<prefix+fileNode[dir[i]].name<<endl;
254         if(fileNode[dir[i]].isDir) {
255             string nextprefix=prefix;
256             size_t pos1=nextprefix.find("├──");
257             while(pos1!=string::npos){
258                 nextprefix.replace(pos1,9,"│   ");
259                 pos1=nextprefix.find("├──");
260             }
261             size_t pos2=nextprefix.find("└──");
262             while(pos2!=string::npos){
263                 nextprefix.replace(pos2,9,"    ");
264                 pos2=nextprefix.find("└──");
265             }
266             nextprefix=nextprefix+"├──";
267             dfsPrtFile(relation[dir[i]],nextprefix);
268         }
269     }
270     return;
271 }
272 //为当前目录最后一个文件时"├──"变成"└──"
273 //递归层数变深时修改前缀,前缀的"├──"变成"│   ";前缀的"└──"变成"    "
```

如上所示代码，首先判断输入的路径是一个文件夹还是一个文件，对于两种情况由于压缩包结构不同，故分开处理。dfsPrtFile()函数接受两个参数，一个是记录文件从属关系的邻接表的一项（vector），另一个则是显示树状结构的前缀，通过观察理想的树状结构视图，发现前缀有以下变化规律，利用以下规律编写递归函数即可：

1.初始前缀为"├──";

2.当该文件为当前目录最后一个文件时"├──"变成"└──";

3.当递归层数变深时修改前缀,前缀的"├──"变成"│ ";前缀的"└──"变成" "



四、开发环境与项目部署

开发环境：

Windows 10 基于 x86-64 的处理器

编程语言：C++

开发工具：

Visual Studio Code：该项目使用 Visual Studio Code 作为主要的集成开发环境。

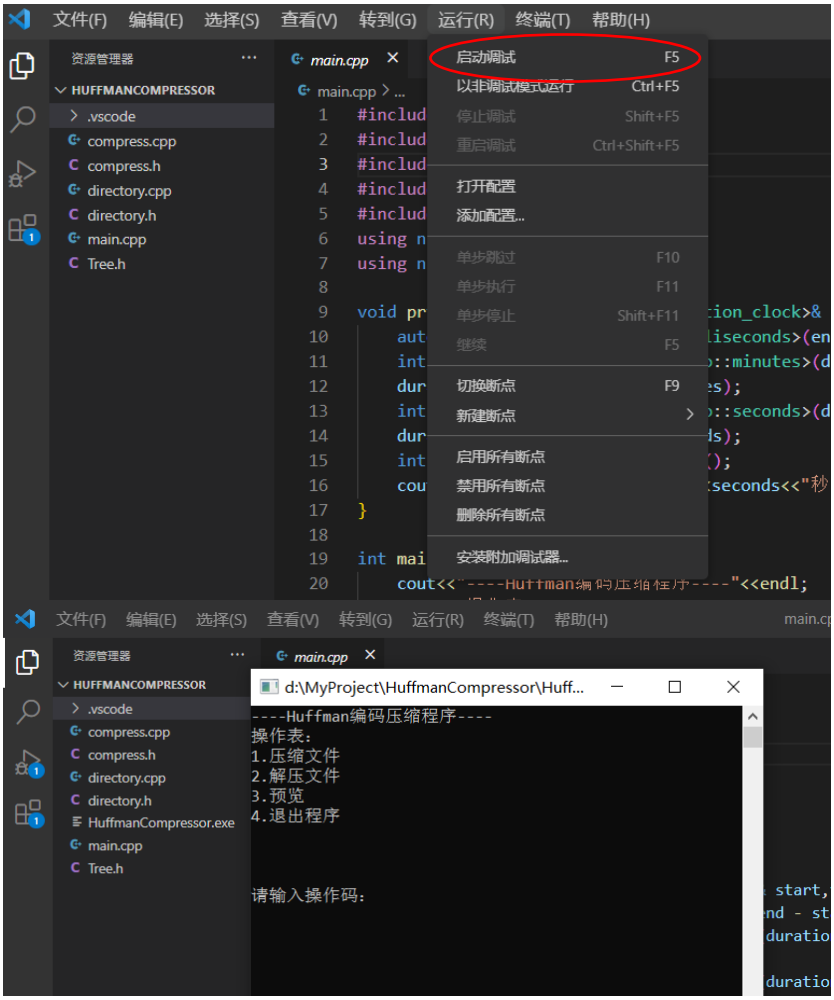
MinGW 编译器：gcc version 8.1.0 (x86_64-posix-sjlj-rev0)

项目部署：

提供两种方式在你的电脑上部署该项目：

（1） 若已经安装的 Visual Studio Code 和 MinGW 编译器

项目中已经配置好.vscode 文件，只需要确保已经安装 g++编译器，在 vscode 中打开该项目所在的文件夹，再编译运行 main.cpp 文件即可。

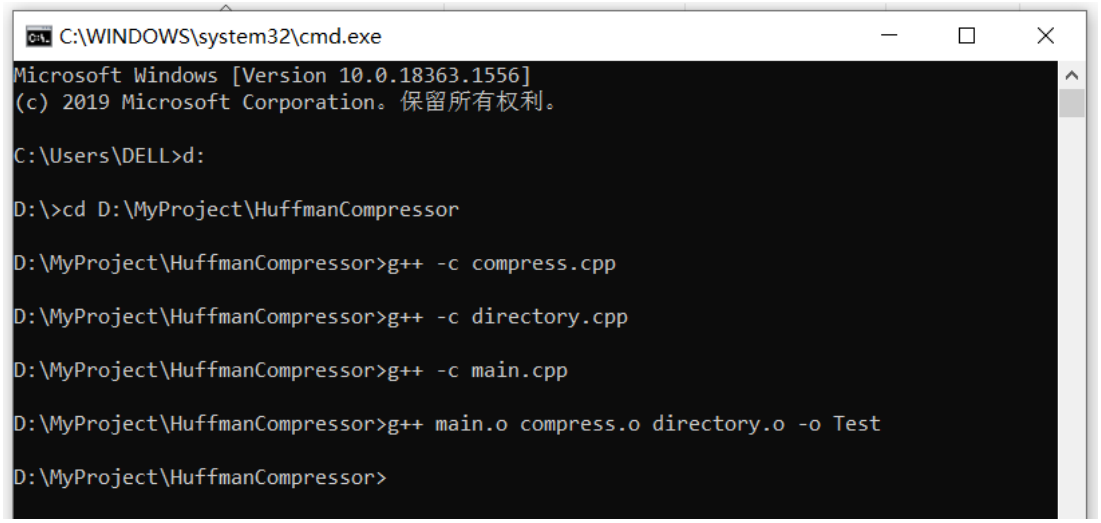


（2） 若未安装 Visual Studio Code，可以直接通过 cmd 编译运行

首先要确保你的 Windows 已经安装 g++编译器。先在 win+R 输入 cmd 打开 Windows cmd 窗口。再通过 cd 进入项目所在的文件夹。然后依次输入以下指令，手动编译链接生成可执行程序。成功后，双击可执行程序 HuffmanCompressor.exe 即可运行。

命令：

```
g++ -c compress.cpp
g++ -c directory.cpp
g++ -c main.cpp
g++ main.o compress.o directory.o -o HuffmanCompressor
```



五、性能测试

说明：

以下测试数据均在我的个人电脑上运行得出（配置参数后附）并未测试在不同硬件配置下的数据。

测试时是直接压缩整个 testcase0x 的文件夹。数据均为三次测量取平均值得出。

测试机配置参数：

Windows 版本： Windows 10 家庭中文版

处理器：Intel(R) Core(TM) i5-8300H CPU @2.30GHz

GPU:NVIDIA GeForece GTX 1050 Ti

RAM:16.0GB

测试数据表：

测试数据

测试用例	压缩前大小/字节	压缩后大小/字节	压缩率	压缩时间	解压时间
testcase01EmptyFile	0	57	—	3ms	2ms
testcase02NormalSingleFile	23903670(22.7 MB)	17520792(16.7 MB)	73%	7s 580ms	4s 983ms
testcase03XLargeSingleFile	1,105,931,880(1.02 GB)	709,459,025(676 MB)	64%	5min 30s 821ms	3min 29s 229ms
testcase04EmptyFolder	0	61	—	2ms	1ms
testcase05NomalFolder	5945430(5.66 MB)	4,224,633(4.02 MB)	71%	2s 496ms	1s 246ms
testcase06SubFolders	448,515,860(427 MB)	450,607,150(429 MB)	100%	6min 12s 344ms	2min 4s 708ms
testcase07XlargeSubFolders	1,099,970,162(1.02 GB)	698,224,051(665 MB)	63%	5min 31s 136ms	3min 36s 716ms
testcase08Speed	643,412,034(613 MB)	411,716,170(329 MB)	63%	3min 14s 25ms	1min 59s 54ms
testcase09Ratio	441,771,600(421 MB)	277,051,720(264 MB)	62%	2min 15s 247ms	1min 21s 532ms

六、与其他工具比较

说明：

本次比较测试与 Winrar，Bandizip，7z 三个主流压缩软件进行比较。由于三个压缩软件均没有自动统计具体压缩时间并显示的功能，故用人工计时测试测试，其中小文件的压缩时间非常短，故误差较大，统计数据仅供参考，大文件压缩时间数据相对误差较小。

Winrar，Bandizip，7z 三个软件都可以选择压缩模式，在测试统一选择“标准模式”。

测试用例：

测试用例介绍	后缀名	大小	简介
小型文件1	.txt	1.89 MB (1,985,016 字节)	纯英文文本文件
小型文件2	.jpg	20.8 KB (21,331 字节)	有损压缩格式的图片文件
小型文件夹3	文件夹	14.4 MB (15,163,296 字节)	含txt avi tsv aifc 四种格式的文件
大型文件1	.mp4	39.7 MB (41,692,746 字节)	手机录制的mp4格式视频文件
大型文件2	.csv	421 MB (441,771,600 字节)	数据量极大的表格文件
大型文件夹3	文件夹	390 MB (409,247,126 字节)	内含大量子文件夹和jpg格式文件

测试数据：

压缩时间比较表

压缩软件\文件大小	小型文件1	小型文件2	小型文件夹3	大型文件1	大型文件2	大型文件夹3
My Compressor	584ms	11ms	4s 919ms	14s 512ms	2min 15s 241ms	2min 29s 910ms
WinRaR	220ms	(极短)	1s 120ms	2s 350ms	16s 290ms	17s 100ms
7Z	670ms	(极短)	4s 500ms	4s 190ms	1min 12s 130ms	24s 180ms
Bandzip	(极短)	(极短)	(极短)	(极短)	4s 120ms	11s 380ms

压缩率比较表

压缩软件\文件大小	小型文件1	小型文件2	小型文件夹3	大型文件1	大型文件2	大型文件夹3
My Compressor	56%	104%	73%	100%	62%	100%
WinRaR	31%	100%	30%	100%	17%	99%
7Z	28%	101%	24%	100%	15%	99%
Bandzip	37%	100%	45%	100%	23%	99%

数据分析：

该项目直接使用霍夫曼算法，对数据进行压缩，由于没有精心设计缓冲区大小等优化，使**压缩速度**显著慢于其他压缩软件。Winrar 使用 Roshal Archive 算法，7z 使用 LZMA2 压缩算法，bandzip 使用 Deflate 算法，以上几种算法基本都是基于 LZ77 算法和霍夫曼编码的组合实现的压缩。其中 lZ77 算法能有效的压缩重复出现的数据，故而上述压缩软件的**压缩率**较我们的项目会略好一些。

此外上图展示的数据中还有一些值得解释的点。小型文件 2 和大型文件夹 3 的内部文件的格式是.jpg，这种文件已经通过有损压缩算法对图片进行过压缩，并且 jpeg 算法也是使用了哈夫曼编码对其中数据进行了压缩，故而再通过哈夫曼编码进行压缩长度并不会减少，反而会因为压缩文件时需要记录一些新的信息导致文件变大。而大型文件 1 是 MP4 格式，同理，MP4 文件本身也是通过哈夫曼编码了信息，使数据长度在数学上已经达到了最优，故经过压缩软件处理大小并不会显著改变。

七、问题与解决

该项目的开发还是比较顺利的。以下记录一些，开发过程中遇到的问题与解决的方法。

（1） 文件没有压缩：

最初写完单文件压缩函数后进行测试，发现运行过程一切正常，但是压缩后的文件总是变大。后来发现是文件的数据虽然全部经过了霍夫曼编码，但霍夫曼编码还是存在一个字节（8bit）中，如某字符经过霍夫曼编码后位 10，则向目标文件中写入一个 00000010 的字符，这样做仅仅只是对数据进行了一个加密而没有进行真正意义上的压缩。后续重构了这一部分的代码，由于 write 函数写入数据的最小单位是 8bit，故只能通过位运算把多个编码后数据存入一个字节中，才解决了这个问题。

（2） 文件压缩率显示问题

开始定义文件压缩率时使用了 int 型保存文件总字符数，稍大点的文件就会导致溢出，并且当文件为空文件时，会直接报错退出程序。后来更换成 unsigned long long 后就防止了溢出。而空文件闪退是因为程序计算压缩率时用 0 作为除法的分母，在计算压缩率的代码上添加一条判断即可。

（3） 写完文件夹压缩发现不能很好的兼容单文件压缩

由于想不到一套合适的方法来实现用相同的函数处理文件夹和单文件的压缩与解压缩，所以后续写完文件夹压缩函数后又单独写两个处理单文件的压缩与解压缩的函数。在读取当前需要压缩的路径或压缩包文件时，先判断该路径是文件夹还是文件，再进入不同的函数进行操作。这使整个压缩解压缩函数的代码多了很多 if 和很多路 return，看起来更加的屎山了，修改的成本也变得很高，但好在能完成当前的需求。