DS_PJ_Report

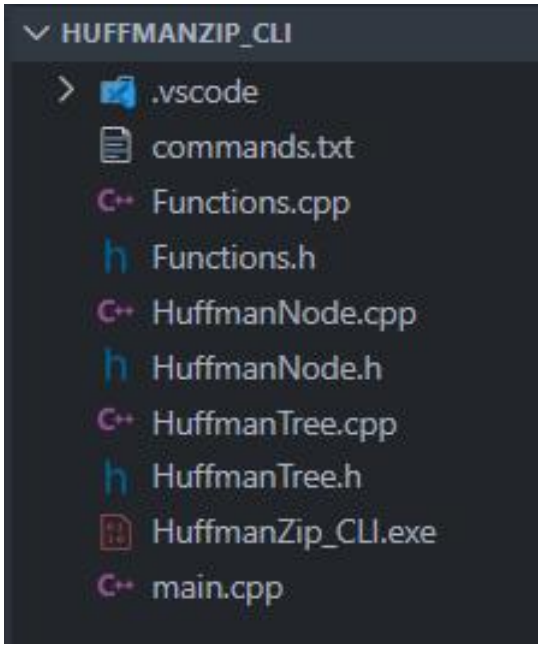
21302020042
侯斌洋

注：关于本次 PJ 的代码也可访问网站：
https://gitee.com/hby_star/Code/tree/master/DataStructure/project/HuffmanZip_CLI
本次 PJ 共做了两个版本，上面的网站显示的是命令行版本的（HuffmanZip_CLI）。此外，还有一个图形界面版本的（HuffmanZip_GUI），可以访问以下网站：
https://gitee.com/hby_star/Code/tree/master/DataStructure/project/HuffmanZip_GUI
此图形界面版本仅供参考，由于邻近期末，故该图形界面版本只实现了主要功能，对于文件路径检查以及文件覆盖等细节功能并没有完全实现，也就是说此版本只保证在正确的输入下能够正常运行，不能对错误的输入进行提示。而命令行版本则实现了 PJ 文档要求的所有细节，此次提交的也是命令行版本的代码。

下面根据文档要求分块对项目进行介绍：

一、代码结构概要

项目文件夹如下：



其中：
①.vscode 文件夹存储 vscode 的配置文件。
②main.cpp 为主程序，里面有项目文件夹中唯一的 main 函数。
③Functions.h 和 Functions.cpp 文件存储 main.cpp 中要用到的函数的声明和实现。
④HuffmanNode.h 和 HuffmanNode.cpp 文件存储哈夫曼树节点的类及其成员函数的声明与实现。

⑤HuffmanTree.h 和 HuffmanTree.cpp 文件存储哈夫曼树的类及其成员函数的声明与实现。
⑥HuffmanZip_CLI.exe 文件为可执行文件，将其与要压缩的文件或文件夹放于同一目录下便可直接通过命令行参数运行，压缩文件也放于该目录下。
⑦commands.txt 文件为运行时的命令指南，对各种命令都有详细的介绍，且文件中还有测试用例相关的所有命令。
项目中比较重要的几个函数如下：

main.cpp:

```
int main(int argc, char **argv);
```

HuffmanTree.cpp:

```
HuffmanTree(priority_queue<HuffmanNode,vector<HuffmanNode>,
            greater<HuffmanNode>> &queue);
```

Functions.cpp:

```
void FileCompress(const fs::path &file_path, const fs::path &zip_path);

void FileUncompress(const fs::path &zip_path, const fs::path &folder_path);

void FolderCompress(const fs::path &folder_path, const fs::path &zip_path);

void FolderUncompress(const fs::path &zip_path, const fs::path &folder_path);

void zipPreview(const fs::path &zip_path, string &preview_graph);
```

项目开发步骤为：首先实现哈夫曼树的构建和哈夫曼编码的获取，然后实现单个文件的压缩与解压缩功能，之后考虑到对于文件夹的压缩与解压缩实际上就是对文件夹目录进行递归然后再逐个压缩或解压缩其中的单个文件，故在单个文件压缩与解压缩功能的基础上增加一个对于文件夹树形结构的保存以及对于文件夹目录的递归遍历即可。至此主要功能已实现，之后再增加压缩包预览以及错误路径检查等细节就完成了整个项目。最后再为每个文件添加注释。
关于项目实现：列出的几个重要函数包含了上述的开发步骤的大部分代码实现，代码中已有较为详细的注释，故详细实现请参考具体的代码文件，代码不在此列出。

二、项目需求的实现

核心需求：
（1） 文件的压缩与解压
压缩：首先遍历一遍文件，找出每个字符出现的频率并将其保存在一个优先队列（按频率大小排序）中，之后每次取出队列中的频率最小的两个元素并增加一个根节点构造一个子树，根节点的频率为两个最小节点的频率之和，然后再让根节点入队。不断执行上述操作直至队列中只有一个节点，即构造了以此节点为根节点的哈夫曼树，然后遍历哈夫曼树来获取哈夫曼编码，并将其存放在一个数组中。之后循环读取文件，每次读取文件中的一个字节并转化为对应的哈夫曼编码，暂时存放在一个 8 位的字符中，等该字符存放的哈夫曼编码满 8 位时

再写入压缩文件。这样就完成了文件的压缩。由于使用的是 fstream 流，故无需刻意设置缓冲区。

解压：解压操作正好相反，先从文件头中获取文件大小以及字符频率信息，之后构造哈夫曼树并获取哈夫曼编码。（注：对于本次 PJ 我并没有将哈夫曼树存储到文件中，而是存储了文件的字符频率，然后在解压的时候通过读取字符频率再构建哈夫曼树进行解压操作，由于压缩和解压的时候构造哈夫曼树用的是相同的方法，故构造出的哈夫曼树总是一样的。此外只存储字符频率相当于只存储了哈夫曼树的叶节点，故理论上更省空间）之后循环读取文件，每次读取一个字节即 8 位，然后按位进行遍历，同时在哈夫曼中寻找对应的叶节点，找到叶节点则直接写入该叶节点对应的字符，当之前记录的文件大小与已解压的字符数相等时解压操作完成。

（2） 文件夹的压缩与解压

对于文件夹的压缩与解压缩实际上就是对文件夹目录进行递归然后再逐个压缩或解压缩其中的单个文件。这里由于使用了<filesystem>类，故对于文件夹的递归遍历比较简单，如下为一个示例。

```
//文件夹压缩操作的内部递归函数
void inFolderCompressDir(const fs::path &folder_path, ofstream &output)
{
    for (const fs::directory_entry &entry : fs::directory_iterator(folder_path))
    {
        if (entry.is_directory())
        {
            //若为文件夹则递归遍历
            inFolderCompressDir(entry.path(), output);
        }
        else
        {
            //若为文件则执行压缩操作
            inFolderCompressFile(entry.path(), output);
        }
    }
}
```

此外关于文件夹树形结构的压缩与解压，思路与 N 叉树的序列与反序列化相同，均为深度优先遍历并做标记，思路也比较简单，不过在实现时要注意细节。

（3） 性能

见文档后面“四、性能测试结果”

（4） 代码风格

不同类的定义，函数的实现，函数的声明以及 main 函数均放在不同文件中。整体结构可参考前面的项目结构的介绍，此外整个项目中除一些较简单的函数外，代码基本上都有比较详细的注释。项目整体的结构也调整了几次，但函数确实比较多，而且有很多要递归处理的函数不得不把递归函数单拿出来，因此 Function.h 和 Function.cpp 中函数比较多，但在 Function.h 中也用注释给函数分了类。其他的文件个人认为以及非常简洁了。

其他需求：

（1） 使用 CLI 与用户交互

有关命令行的处理详见 main 函数，只是简单的字符串处理，关于此功能请直接运行来进行

测试，关于具体的运行命令，请仔细阅读 commands.txt 文件，其对各种命令都有详细的介绍，并且文件中还有测试用例相关的所有命令。如下是一个运行示例（在 cmd 中）：

```
F:\testcases\testcases> .\HuffmanZip_CLI --h
---------------------------------Help---------------------------------
| Read commands.txt for details                                       |
| Compress:   .\HuffmanZip_CLI  zip_hby  anyname.hby  file_or_folder  |
| Uncompress:.\HuffmanZip_CLI  unzip_hby  zip_file                    |
| Preview:    .\HuffmanZip_CLI  prev_hby   zip_file                   |
|---------------------------------End---------------------------------|
```

（2） 检验压缩包来源是否是自己的压缩工具

这里我是通过识别文件名后缀来检测的，压缩包后缀应为.hby。

检查后缀代码如下：

```
1.      //识别压缩文件后缀是否正确
2.      string zip_str_name = zip_path.filename().string();
3.      string post_fix = zip_str_name.substr(zip_str_name.rfind('.'));
4.      if (post_fix != ".hby")
5.      {
6.          cout << post_fix << " is not the right suffix." << endl;
7.          cout << "The file suffix should be .hby" << endl;
8.          system("pause");
9.          exit(1);
10.     }
```

如下是一个运行示例：

```
F:\testcases\testcases>.\HuffmanZip_CLI unzip_hby test.pptx
.pptx is not the right suffix.
The file suffix should be .hby
Press any key to continue . . .
```

（3） 文件覆盖问题

这里主要是利用了<filesystem>中的 exist()函数来检查文件或文件夹是否已存在。若检测到文件已存在则进行询问是否覆盖，若不存在则正常进行操作。如下是具体实现

```
1.  char c_or_q = 'c';
2.  if (fs::exists(file_path))
3.  {
4.      cout << "There is already a file: " << file_path.filename() << endl;
5.      cout << "Press c to cover the old file, any other key to quit" << endl;
6.      cin >> c_or_q;
7.  }
8.  if (c_or_q == 'c')
9.  {
10.     if (fs::exists(file_path))
11.     {
12.         fs::remove_all(file_path);
```

```
    }
    cout << "Please wait a moment..." << endl;
    Uncompress(zip_path, folder_path);
    cout << "Finish!" << endl;
}
else
{
    exit(0);
}
```

下面是一个运行示例：

```
F:\testcases\testcases> .\HuffmanZip_CLI zip_hby testcase01EmptyFile.hby testcase01EmptyFile
There is already a file: "testcase01EmptyFile.hby"
Press c to cover the old file, any other key to quit
c
Please wait a moment...
Finish!

F:\testcases\testcases>.\HuffmanZip_CLI zip_hby testcase01EmptyFile.hby testcase01EmptyFile
There is already a file: "testcase01EmptyFile.hby"
Press c to cover the old file, any other key to quit
q

F:\testcases\testcases>
```

（4） 压缩包预览

由于测试用例生成的文件夹树形结构实在是不太美观，故自己建了一个文件夹进行测试，下面是运行示例。详细代码见函数：

```
void zipPreview(const fs::path &zip_path, string &preview_graph)
```

代码中已有详细的注释，主要思路还是递归遍历文件夹目录，同前面的文件夹树形结构的反序列化。要注意的一点就是递归函数中需要一个 static int level 来记录文件所处的深度，并根据 level 的大小来决定文件名前面的修饰符的多少，这样就可以呈现出有层次的树形结构。

```
F:\>.\HuffmanZip_CLI   zip_hby   test.hby  test
Please wait a moment...
Finish!

F:\>.\HuffmanZip_CLI   prev_hby   test.hby
test
  |----test1.txt.txt
  |----test2.txt.txt
  |----test_test1
  |       |----test_test1.txt.txt
  |       |----test_test_test
  |       |       |----test_empty
  |       |       |----test_test_test1.txt.txt
  |----test_test2
  |       |----1.txt.txt
  |       |----2.txt.txt
Press any key to continue . . .
```

（5） 与其他压缩工具的压缩率与压缩时间比较

见文档后面 “五、与其他压缩工具的比较”

（6） 开发文档

略...

三、开发环境及如何编译运行项目

开发环境：

系统：Win10

语言：C++

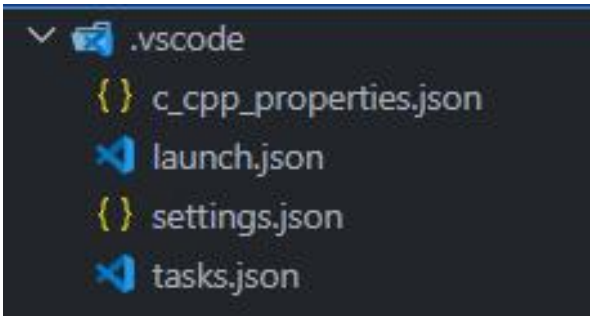
编译器：gcc

IDE：VS Code

如何编译运行项目：

编译：

项目文件夹已有自带的配置文件如下：



若要在不同的机器上编译，要注意修改上面配置文件中 gcc 以及 gdb 的位置，要修改的地方已经用注释以以下形式标注出来了，请仔细检查每个配置文件，避免遗漏。

```
/*这里需要修改为本地gcc位置*/
"compilerPath": "D:/mingw64/bin/gcc.exe",
/*************************/
```

最后按下图操作即可：在 vscode 中打开 HuffmanZip_CLI 文件夹，选择最左侧图标，在右面的配置中选择 Project(GDB)，此即上述.vscode 配置文件的配置。最后点击绿色三角形即可以 debug 状态进行运行和调试。

运行：

上传的文件夹中已含本地编译好的文件 HuffmanZip_CLI.exe，也可以通过上面的方法重新生成可执行文件。

关于具体的运行命令，请仔细阅读 commands.txt 文件，其对各种命令都有详细的介绍，且文件中还有测试用例相关的所有命令，故不在此重复列出。

此外，关于编译运行有任何问题请联系我，谢谢。

四、性能测试结果

	初始大小/MB	压缩后大小/MB	压缩率	压缩时间/s	解压时间/s
testcase01EmptyFile	0	0	——	0.026	0.035
testcase02NormalSingleFile	22.7	16.7	73.6%	2.04	1.285
testcase03XLargeSingleFile	1044	676	64.8%	78.28	44.799
testcase04EmptyFolder	0	0	——	0.035	0.027
testcase05NomalFolder	5.66	4	70.7%	0.561	0.364
testcase06SubFolders	427	430	100.7%	57.55	40.678
testcase07XlargeSubFolders	1044	665	63.7%	77.23	44.067
testcase08Speed	613	392	63.9%	43.658	24.504
testcase09Ratio	421	264	62.7%	32.352	18.487

压缩率 | 压缩时间 | 解压时间	testcase02NormalSingleFile	testcase06SubFolders	testcase07XlargeSubFolders	testcase08Speed	testcase09Ratio
My_PJ	73.6% | 2s | 1s	100.7% | 58s | 41s	63.7% | 77s | 44s	63.9% | 44s | 25s	62.7% | 32s | 18s
Bandizip	48% | 0s | 0s	99% | 12s | 6s	17% | 3s | 2s	12% | 1s | 1s	24% | 1s | 1s
7-z	34% | 5s | 0s	97% | 14s | 6s	11% | 61s | 1s	8% | 30s | 0s	15% | 35s | 0s

五、与其他压缩工具的比较

上图中第一列为小文件，大小为 22.7MB，第 2-5 列均为大文件，超过 200MB。
分析：可以看出在大多数情况下无论是压缩率还是压缩效率 My_PJ 与其他主流压缩软件还有很大的差距。首先是压缩率，本次 PJ 使用的是哈夫曼编码，在压缩率上应该是比较固定的，而其他压缩软件可能根据不同的文件类型有不同的编码方式，其他编码方式在不同的文件类型上可能比哈夫曼编码更省空间，因此压缩率上比其他软件更低也在意料之中。此外通过上图可以看出 7-z 的压缩率明显比 Bandizip 低，而随之而来的是 Bandizip 相对更快的压缩速度，因此不同的压缩算法有不同的优缺点，在选择压缩算法时要在压缩率与效率上进行抉择。其次是压缩时间，这里我发现 Bandizip 可以选择 CPU 线程数，上面的数据是在线程数为自动的情况下测出的。线程数为自动时，testcase08Speed 压缩时间为 1s，而手动将线程数改为 1 时，testcase08Speed 压缩时间为 9s，差了将近 10 倍。在多核 CPU 占主流的今天，使用多核进行并行压缩无疑可以极大地提高效率。而本次 PJ 并不支持多核并行运算，这是其中一个差距所在；还有重要的差距应该便是在算法上，由于对其他压缩算法了解比较

少，故在此直接带过。此外，在 CPU 算力剩余的情况下磁盘的读取速度也是需要重点关注的，不过这方面的差异主要体现在不同的机器上。（字数稍多，见谅）

六、一些问题及解决方案

（1）由于哈夫曼树涉及很多的指针操作，故程序经常会跑着就报段错误了，在调试的过程中虽然 vscode 的 debug 程序很方便，但其只能看到当前作用域中的变量，在调试递归程序时就显得不太方便了，这时在合适的位置插入 cout 可以更快地锁定错误的地方。

（2）之前对 main 函数的命令行参数、STL 中的优先队列以及<filesystem>类并不熟悉，写代码的时候翻了很多博客才一点点摸索学会，期间也遇见了不少 bug，花了很多时间，但总算收获颇丰，这些标准库函数都是学起来难受，用起来真香。

（3）此外正如开头提到的，本次 PJ 我还做了另一个图形界面版本，这也是我写的第一个带图形界面的程序。在实现完主要功能后我就准备开始写图形界面了，本来是想用 Easyx 的，但 Easyx 图形库并不支持 Mingw，因此用 vscode 很不方便，于是便用了另一个图形库 EGE。调用这些库函数并不困难，但是想要做好界面就要花很大功夫，甚至只是一个圆角矩形都要花不少时间来做。不过做完之后就能发现这比命令行版本要方便非常多。毕竟鼠标点一下比输一个命令要方便多了。

https://gitee.com/hby_star/Code/tree/master/DataStructure/project/HuffmanZip_GUI

（4）对于本次 PJ 来说，文件的打开形式一定要是二进制，不然会出很多问题，并且这些问题很难被调试出来。

（5）对于本次 PJ 的效率优化，有一个未经检验的猜想：本次 PJ 是每次以 8 位为单位进行读写的，如果将 8 位改为 64 位（即将一次读取 1 个字节改为一次读取 8 个字节）是否有可能提高效率？不过若这样做则对于文件末尾的检查也会变得更加复杂。

七、总结

本次 PJ 确实工作量比较大，写了有上千行代码，花了两个星期左右，但也学到了很多东西，总而言之收获颇丰。此外，专注做一件事的过程也很令人享受。