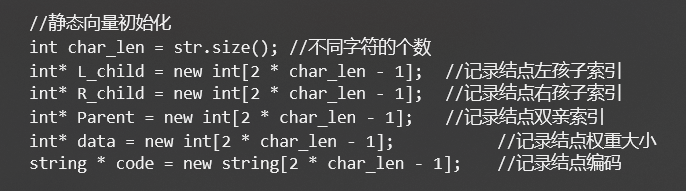
【问题描述与分析】

该问题要求实现哈夫曼编码。其核心在于通过哈夫曼算法实现最优二元树，然后通过构建的二元树对各叶子节点进行编码，由于该编码方式中权重越大的字符采用越短的编码且各字符的编码均不是其他字符编码的前缀，因此可以实现文件的压缩。

【数据结构选择】

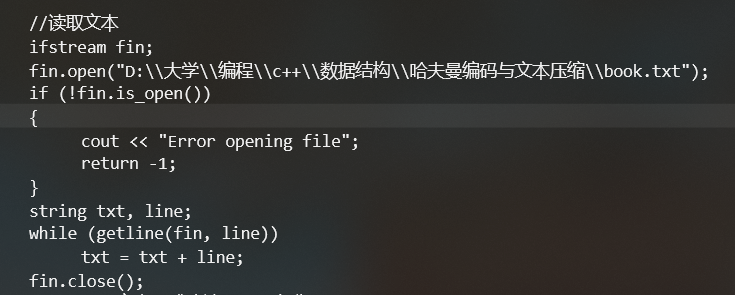
采用树结构作为算法设计的基础，分别用五个向量存储各结点信息：



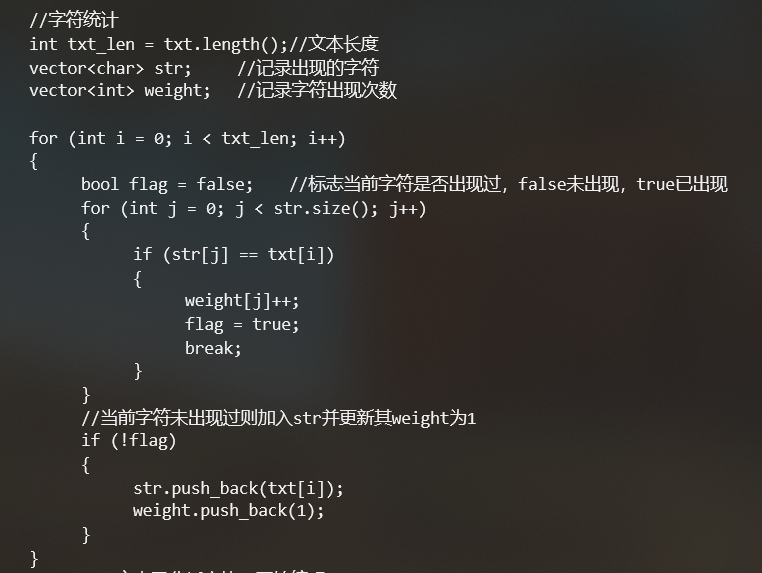
（此处也可以构造可用于各节点的结构，包含结点的双亲、孩子、权重及编码等属性，然后将各结点存放在一个向量中）

【算法设计】

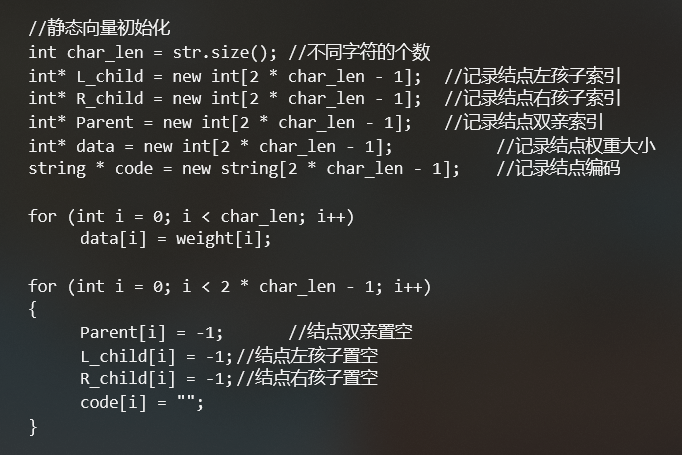
首先通过fstream读取文本，以文件夹下book.txt为例：



然后对读取到的文本进行分析即字符统计，遍历文本的每个字符，如果该字符存在于str向量中则将其权重加1；若该字符不存在则加入str向量中，并将其权重初始化为1，算法实现如下：

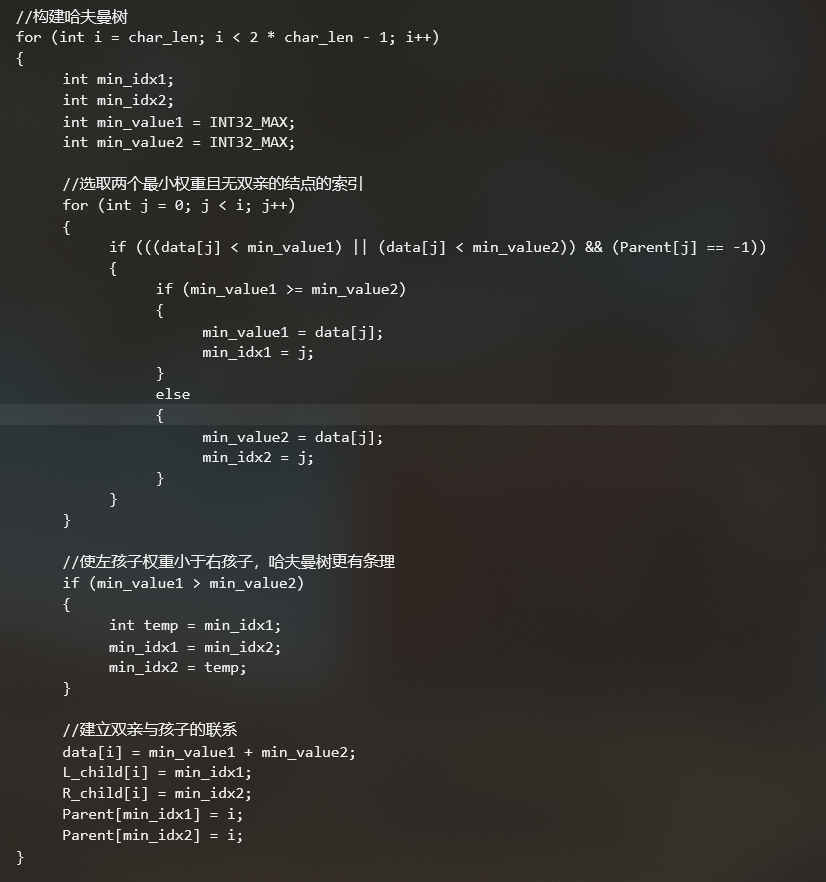


此时我们已经得到该文本所有不同字符及其权重，下一步则对存储树数据结构的各向量进行初始化：

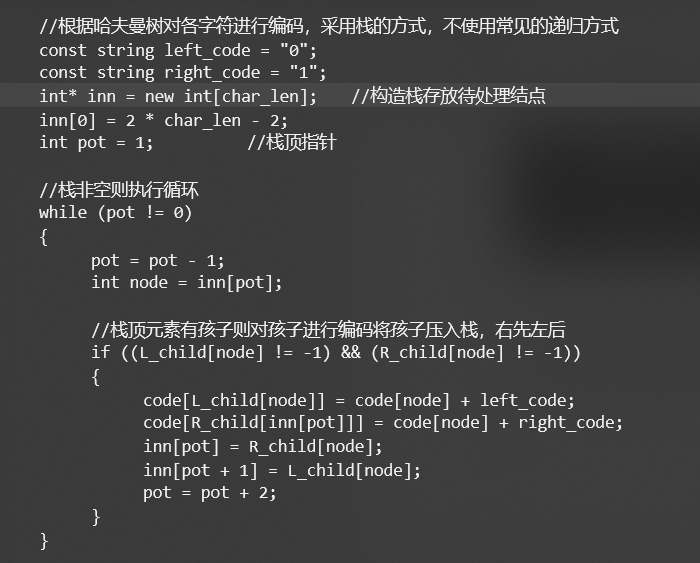


（-1代表该结点无该属性）

在完成这些准备工作后可以进行哈夫曼树的构造，每一次在结点队列中选取两个最小权重且尚无双亲的叶子，构造新的结点作为他们的双亲，其权重为两片叶子权重之和，并将该节点加入结点队列。循环执行上述操作直到队列中只剩一个不存在双亲的结点为止。该算法的难点在于寻找结点队列中权重最小且不含双亲的两个结点，整个构建过程代码如下：

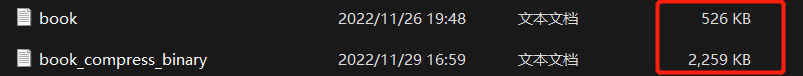


然后通过构建好的哈夫曼树对各结点进行编码，此处可以采用递归的方式但效率不高，因此采用人工栈的方式实现，具体算法实现如下：



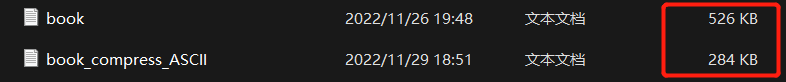
至此，对各字符我们已经完成编码，为了方便后续解码，将整个文本的分析结果以树的数据结构保存在Dict.txt中。往后只需根据编码对文本进行压缩即可。

此时遇到一些困难，如果将文本中每一个字符直接用二进制编码替换，输出压缩文本book\_compress\_binary.txt空间占用反而会变大，效果如下：



经过分析我才明白这是必然结果，因为将字符转化为二进制编码，其本意是每一位用位（bit）来保存，而我们将其输出为文本相当于以字节的形式存储，每一个字符反而变大了。举个例子，假设字符“a”的编码是1001，以此方式保存相当于将本来只用1个字节保存的字符用4个字节去保存，输出文件自然会变大。

于是我们考虑将二进制编码每八位转化为一个ASCII码进行输出，这样的方式理论上会使文件有明显的压缩效果，结果也在意料之中：



该种方式得到的压缩文件book\_compress\_ASCII.txt明显变小，不过由于ASCII码中一些字符本身无法打印，因此压缩文件中存在乱码现象。

值得一提的是，将二进制编码转化为ASCII码输出在代码实现上也存在一定挑战，首先是二进制编码的长度不一定是8的整数倍，其次将整个文本转化为二进制编码再进行ASCII的转化运行很慢，因此特意设计了一种方法解决以上两个问题，具体代码可以查看源代码文件。

【测试用例】

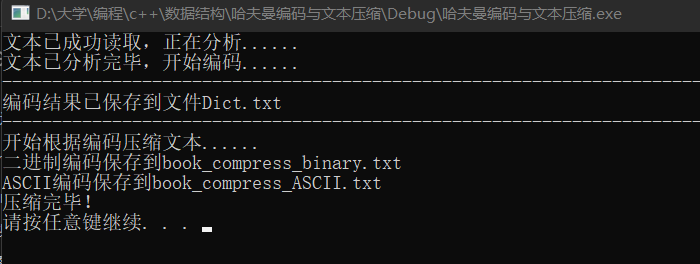
首先将输入文本放置在指定路径下：



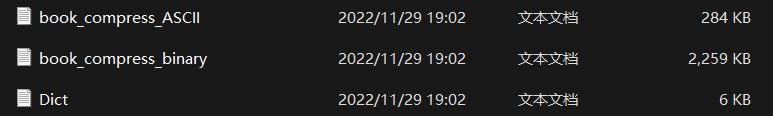


然后运行程序（由于在其他电脑环境这个路径搜索不到，因此可能需要对.cpp文件进行修改，作业包里的exe文件也无法直接使用）：

测试结果：



输出文件保存在相应目录：



【评价与心得】

哈夫曼编码十分巧妙，其本质是一个动态规划算法，每一步都是当前状态下的最优解，因此最后得到的树也是最优二元树，将其应用在编码与压缩领域也是十分值得称赞的实践。

通过实现哈夫曼算法以及将其应用在文本压缩问题上，让我对树这个数据结构有了更深层的理解。