1. 哈夫曼树的实现方式：

成员变量及成员函数的用处：

hf\_Node:哈夫曼树的结点类，内含左子结点，右子结点，权值，父结点等信息

std::map<std::string, int> fre：存放叶结点与其对应权值

hf\_Node \*elem：存放哈夫曼树的所有结点

total\_length：最终结点总数

leaf\_length:叶结点数

get\_sin\_table(std::string text):得到SingleChar模式下的叶结点及其权值的映射

get\_dou\_table(std::string text):得到MutiChar模式下的叶结点及其权值的映射

encode\_build(int index)以elem[index]为根结点编码（采用递归）

void double\_treebuild() 两个模式下的建树

struct Compare:设置比较器，使map中一直按字典序进行排序

实现方式：先获得所有叶结点及其对应权值，然后将整个哈夫曼树放置于一个长度为叶结点数的两倍的数组中，数组的0位置不放置结点，将所有叶结点放在数组的后面（数组下标从leaf\_length~total\_length），然后数组下标从leaf\_length-1开始建树，找两个权值最小的合并，放入对应位置，更新节点的信息，重复此过程，直到数组下标为1，搭建完毕。

1. 三个自建文本的实验数据（自建文本在self-created-test文件夹中）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 压缩前(字节数) | 压缩后（single） | 压缩后（muti） | 压缩率 |
| test1 | 3 | 9 | 9 | 300% |
| test2 | 312 | 176 | 175 | 56% |
| test3 | 3506 | 1910 | 1907 | 54% |

根据实验数据可以看到：当文本数据非常小时，压缩的字节数比原始字节数还要大，起不到压缩效果，只有当文本书足够大时，压缩才会其效果。根据数据，两种压缩模式的效果没有显著差别，压缩后的字节数几乎一致。

1. 更优的压缩策略：

从实验中可以看出，有些字符组合出现频率并不高，但由于算法强制要求必须有出现频率前三的字符组合，原本频率不高的字符组合被采用，这样并不能提高效率，同时采用频率不高的字符组合可能会破坏本来频率较高的单个字符，我认为，应统计所有单个字符和双字符组合的权值，当字符组合的权值不小于组成字符组合的单独字符权值的一半时，才使用该字符组合，否则不使用该字符组合的权值，这样才能提高压缩效率