Lab0:哈夫曼压缩(书面报告)

哈夫曼树实现

哈夫曼树成员变量

本次实验中，考虑到哈夫曼树中，非叶结点个数恰好为叶结点个数减一，故哈夫曼树使用一个数组实现，前半部分记录非叶结点，后半部分记录叶结点。节点结构体如下：

```
struct Node{
    std::string data;
    uint64_t weight; //权值
    uint64_t parent,left,right;
    std::string dictionary; //存放字典序
};
```

在节点中存放着节点的父结点及左右子节点的数组下标；同时为了方便比较，还存放着字典序，记录子树中最小的字典序，这意味着建树时，字典序需要随时更新。

整棵树的结构体如下：

```
struct OpTree{
    Node *Tree;
    uint64_t Length;
    std::map<std::string,uint64_t> frequecyMap; //记录出现频率
};
```

该结构体存放着节点构成的数组，数组长度。在建树前，需要记录所有字符元素和出现次数构成的键值对，故使用std::map来保存所有的键值对。

哈夫曼树建树过程

在建树前，需要遍历整个文本，并使用std::map记录所有的频率键值对，而不同的编码方式对应的遍历过程也不一样。

单字符文本遍历

```
if(op == Option::SingleChar){ //构造单字符树
    for(char c : text){ //记录所有字符的频率
        optree->frequecyMap[std::string(1,c)]++;
    }
}
```

当编码方式为单字符时，遍历十分简单，只需要遍历所有单字符，在std::map中增加对应的出现频率就可以了。

组合字符文本遍历

当编码方式为组合字符时，为了找到出现频率最高的前三个组合字符。因此，可以先遍历一次文本，将所有字符和紧随其后的字符构成的组合字符全部记录一遍，以便找到前三个组合字符。如下：

```
for(uint64_t i = 0; i < text.length(); i++){
    if(i + 1 < text.length()){
        std::string multi = text.substr(i,2);
        optree->frequecyMap[multi]++;
    }
}
```

接下来需要遍历std::map，从而找到前三个组合字符，为此使用max1,max2,max3记录最大、次大、第三大的出现频率，并使用multi1,multi2,multi3记录对应的组合字符(这里也可以用std::pair来记录)。每个新的组合字符需要依次与他们比较频率，或在相等时比较字典序，以插入对应的位置。如下：

```
std::string multi1,multi2,multi3;
uint64_t max1,max2,max3;
max1 = max2 = max3 = 0;
for(std::pair<std::string,int> p : optree->frequecyMap){ //找到前三个
    if(p.second > max1 || (p.second == max1 && p.first < multi1)){
        max3 = max2;
        max2 = max1;
        max1 = p.second;
        multi3 = multi2;
        multi2 = multi1;
        multi1 = p.first;
    }
    else if(p.second > max2 || (p.second == max2 && p.first < multi2)){
        max3 = max2;
        max2 = p.second;
        multi3 = multi2;
        multi2 = p.first;
    }
    else if(p.second > max3 || (p.second == max3 && p.first < multi3)){
        max3 = p.second;
        multi3 = p.first;
    }
}
```

最后重新遍历文本，先检查是否为之前记录的组合文本，如果是，则跳过下一字符(因为已被组合字符吸收)；如果不是，再以单字符的形式记录。当然，在此之前需要先清空std::map.如下：

```
optree->frequecyMap.clear(); //清空树，重新遍历

for(uint64_t i = 0; i < text.length(); i++){
    if(i + 1 < text.length()){
        std::string cc = text.substr(i,2);
        if(cc == multi1 || cc == multi2 || cc == multi3){
            optree->frequecyMap[cc]++;
            i++;
            continue;
        }
    }
    std::string c = text.substr(i,1);
    optree->frequecyMap[c]++;
}
```

建树

建树过程就不需要区分编码方式了。首先需要申请长度为map.size()两倍的节点数组，并将记录字符与权值的叶结点放在数组的后半部分：

```
uint64_t size = optree->frequecyMap.size();
uint64_t i = size;
optree->Length = 2 * size;
optree->Tree = new Node[optree->Length];

for(std::pair<std::string,uint64_t> p : optree->frequecyMap){
    optree->Tree[i].data = optree->Tree[i].dictionary = p.first;
    optree->Tree[i].weight = p.second;
    optree->Tree[i].parent = optree->Tree[i].left = optree->Tree[i].right = 0;
    i++;
}
```

接下来按照权值和字典序，每次找到最小的两个节点，合并，并从右向左依次放在数组的前半部分。数组下标为零的节点置为哨兵：

```
uint64_t min1,min2;
uint64_t x,y;

for(i = size - 1;i > 0;i--){
    min1 = min2 = UINT64_MAX;
    x = y = 0;

    for(uint64_t j = i + 1; j < optree->Length; j++){
        if(optree->Tree[j].parent == 0){
            if(optree->Tree[j].weight < min1 || (optree->Tree[j].weight == min1 && (optree-
                min2 = min1;
                min1 = optree->Tree[j].weight;
                x = y;
                y = j;
            }
            else if(optree->Tree[j].weight < min2 || (optree->Tree[j].weight == min2 && (opt
                min2 = optree->Tree[j].weight;
                x = j;
            }
        }
        if(min1 == min2) //权值相等，比较字典序
            if(optree->Tree[y].dictionary > optree->Tree[x].dictionary)
                std::swap(x,y);
    }

    optree->Tree[i].weight = min1 + min2;
    optree->Tree[i].left = y;
    optree->Tree[i].right = x;
    optree->Tree[i].parent = 0;
    optree->Tree[i].dictionary = (optree->Tree[y].dictionary > optree->Tree[x].dictionary)
    optree->Tree[y].parent = optree->Tree[x].parent = i;
}
```

编码表实现

检查数组的后半部分(叶子节点)，自下而上寻找父结点直至哨兵，如果是父节点的左孩子，就在编码的开头压入'0'；右孩子则压入'1'.代码如下：

```
std::map<std::string, std::string> hfTree::getCodingTable()
{
    // TODO: Your code here
    for(uint64_t i = optree->frequecyMap.size();i < optree->Length; i++){
        std::string code = "";
        uint64_t parent = optree->Tree[i].parent;
        uint64_t now = i;

        while(parent){
            if(optree->Tree[parent].left == now){ //左子树
                code = '0' + code;
            }
            else{ //右子树
                code = '1' + code;
            }
            now = parent;
            parent = optree->Tree[parent].parent;
        }
        hfCode.insert(std::make_pair(optree->Tree[i].data,code));
    }

    return hfCode;
}
```

自选文本比较压缩方式

为了进行测试，额外自选三组文本来进行比较，这三组文本长短有明显差异，每个文本字符出现频率也有一定参差，文本以及测试后的实验数据如下：

文本1

> The quick brown fox jumps over the lazy dog.

编码方式	文本字节数	压缩后文本字节数	压缩率
单字符	45	34	75.56%
组合字符	45	33	73.33%

文本2

> Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed pulvinar nisl at justo eleifend, a tincidunt dolor consectetur. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Sed augue lorem, blandit in nisl ac, consequat pellentesque quam. Nullam mollis metus nec nunc fermentum, quis tempor mauris pellentesque. Fusce efficitur ante sit amet tellus commodo, eu posuere purus efficitur. Suspendisse potenti. Donec vulputate aliquet orci, eget lobortis nunc. Nam id velit ut nunc hendrerit porttitor. In eget ipsum sed sem semper fringilla. Sed auctor sollicitudin tortor non pulvinar. Sed eget lacus vel est tempus finibus. Nulla facilisi.

编码方式	文本字节数	压缩后文本字节数	压缩率
单字符	679	368	54.20%
组合字符	679	364	53.61%

文本3

> It is a truth universally acknowledged, that a single man in possession of a good fortune must be in want of a wife.
>
> However little known the feelings or views of such a man may be on his first entering a neighbourhood, this truth is so well fixed in the minds of the surrounding families, that he is considered the rightful property of some one or other of their daughters.
>
> "My dear Mr. Bennet," said his lady to him one day, "have you heard that Netherfield Park is let at last?"
>
> Mr. Bennet replied that he had not.
>
> "But it is," returned she; "for Mrs. Long has just been here, and she told me all about it."
>
> Mr. Bennet made no answer.
>
> "Do you not want to know who has taken it?" cried his wife impatiently.
>
> "_You_ want to tell me, and I have no objection to hearing it."
>
> This was invitation enough.
>
> "Why, my dear, you must know, Mrs. Long says that Netherfield is taken by a young man of large fortune from the north of England; that he came down on Monday in a chaise and four to see the place, and was so much delighted with it, that he agreed with Mr. Morris immediately; that he is to

> take possession before Michaelmas, and some of his servants are to be in the house by the end of next week."

编码方式	文本字节数	压缩后文本字节数	压缩率
单字符	1209	674	55.74%
组合字符	1209	671	55.50%

综合三次比较，压缩率整理如下：

编码方式	文本1(45)	文本2(679)	文本3(1209)
单字符	75.56%	54.20%	55.74%
组合字符	73.33%	53.61%	55.50%

观察实验数据可知，当文本量较小时，虽然哈夫曼编码较短，但还有用于记录长度的8个字节，导致压缩率反而较大；当文本量较大时，出现频率小的字符对应的编码可能还超过了一个字节，但仍趋于稳定。另一方面，相较于单字节编码，组合字节编码压缩率略有优势，但差异可以说是非常小，这可能是文本选的不够好，没有出现频率很大的组合字符。

更优的压缩策略

本实验使用的组合字符压缩之所以效果有限，我认为有两点原因：其一是只使用了双字符组合，其二是只采用前三的字符组合，无法根据文本长度进行调整。另一种方法是使用多进制的哈夫曼算法，来代替现在的二进制哈夫曼算法。本实验的哈夫曼树建树时，每次只取最小的两个节点来构造新的节点。而多进制的哈夫曼算法，可以一次取多个节点来进行合并。这样构建出来的树高度更低，相较于二进制的哈夫曼算法，具有占用内存空间更小，解码速度快的优点。