Lab 0 Report

一、哈夫曼树类的实现

1. 哈夫曼树节点：如下图，一个简单的结构体，包含了节点的权重（词频）、值（在这里是节点的字典序）以及左右子节点指针。其中为了实现节点的优先级比较，构造一个自定义函数 compare，表示词频较小、词频相等时字典序较小的节点的优先级更高。

```
struct Node
{
    std::string data;//order of the characters
    int freq;
    Node *left;
    Node *right;
    Node(std::string data, int freq, Node *left, Node *right) : data(data), freq(freq), left(left), right(right) {}
};
struct compare
{
    bool operator()(Node *l, Node *r)
    {   if(l->freq==r->freq) return (l->data > r->data);
        return (l->freq > r->freq);
    }
};
Node *root;
```

2. 哈夫曼树的构建：如下图，先实现单字符哈夫曼编码树的构建。先遍历文本建立词频数组，然后依次塞入优先级队列，最后从堆中不断取出优先级最高的节点来构建哈夫曼树。其中需要注意节点值 data 应该更新为子节点的较小值（字典序）。

再实现多字符编码树的构造函数。同样建立词频数组，但是是连续 2 字符的词频。然后进行排序，选出词频最大的三个词放进数组 top3 中。最后根据这三个词构建最终词频，基本思路就是对于每一个字符位，先搜索 2 字符是否存在编码，没有的话就建立单字符的词频。最后的建树过程和单字符树一样，不再赘述。

```
hfTree::Node* hfTree::buildSingleCharHfTree (const std::string &text)
{
    std::map<char, int> frequencyMap;
    for (char c : text) {
        if(frequencyMap.find(c)!=frequencyMap.end()) frequencyMap[c]++;
        else frequencyMap[c]=1;
    }

    std::priority_queue<Node*, std::vector<Node*>, compare> pq;

    for (auto it = frequencyMap.begin(); it != frequencyMap.end(); ++it) {
        Node* newNode = new Node(std::string(1, it->first), it->second, nullptr, nullptr);
        pq.push(newNode);
    }

    while (pq.size() > 1) {
        Node* left = pq.top();
        pq.pop();
        Node* right = pq.top();
        pq.pop();

        Node* merged = new Node(std::min(left->data,right->data), left->freq + right->freq, left, right);
        merged->left = left;
        merged->right = right;

        pq.push(merged);
    }

    root = pq.top();
    return root;
}
```

```cpp
hfTree::Node* hfTree::buildMultiCharHfTree (const std::string &text)
{
   std::map<std::string, int> frequencyMapof2;
    for (int i = 0; i < text.size(); ++i) {
       if (i < text.size() - 1) {
           frequencyMapof2[text.substr(i, 2)]++;
       }
    }
    std::vector<std::pair<std::string, int>> sortedMap(frequencyMapof2.begin(), frequencyMapof2.end());
    std::sort(sortedMap.begin(), sortedMap.end(), [](std::pair<std::string, int> a, std::pair<std::string, int> b)
       if(a.second==b.second) return a.first < b.first;
       return a.second > b.second;
    });
    std::map<std::string, int> top3;
    for (int i = 0; i < 3; ++i) {
       top3[sortedMap[i].first] = sortedMap[i].second;
    }

    std::map<std::string, int> frequencyMap;
    for(int i=0;i<text.size();++i){
       if(i<text.size()-1){
           std::string temp=text.substr(i,2);
           if(top3.find(temp)!=top3.end()){
               frequencyMap[temp]++;
               i++;
           }
           else frequencyMap[std::string(1,text[i])]++;
       }
       else frequencyMap[std::string(1,text[i])]++;
    }
```

3. 各功能函数的实现

见下图。Parsetext 函数大致思路为：用二进制格式读取文件，一个一个字符读取然后转换为 string；

Output 函数大致思路为：用文件流直接输出到文件里；

Codingtable2string 函数大致思路为：遍历编码表，一个一个按格式添加到字符串尾部；

Loadcodingtable 函数大致思路为：先把文件读成字符串，然后遍历。i 指向了每一个编码的空格位置，j 指向了末尾的换行符位置，然后用 substr 函数读入键和值加入编码表；

```cpp
std::string parseText(const std::string &input){
    std::ifstream file(input, std::ios::in | std::ios::binary); // 以二进制模式打开文件
    std::ostringstream oss;
    char buffer;
    while (file.get(buffer)) {
        oss << buffer; // 将读取的二进制数据写入字符串流
    }
    std::string content = oss.str();// 从字符串流中获取内容
    file.close(); // 关闭文件
    return content;
}
void output(const std::string &output, const std::string &data){
    std::ofstream file(output);
    if (file.is_open()) {
        file << data;
        file.close();
    }
}
std::string codingTable2String(const std::map<std::string, std::string> &codingTable){
    std::string result = "";
    for (auto it = codingTable.begin(); it != codingTable.end(); it++) {
        result += it->first + " " + it->second + "\n";
    }
    return result;
}
void loadCodingTable(const std::string &input, std::map<std::string, std::string> &codingTable){
    std::string text=parseText(input);
    int i=0,j=-1;
    while(j!=text.size()-1){
        i=text.find(' ',j+2);
        if(text[i+1]==' ') i++;
        std::string key=text.substr(j+1,i-j-1);
        j=text.find('\n',i+1);
        std::string value=text.substr(i+1,j-i-1);
        codingTable[key]=value;
    }
}
```

Compress 函数大致思路为：字符串遍历一次转换为 string 形式的编码，然后对齐到 8 位，再一次遍历把字符串形式转换为 char 形式的字节流。最后把字节流大小这个值用小端法转换成字节流，合并然后返回。

```
std::string compress(const std::map<std::string, std::string> &codingTable, const std::string &text)
{
    std::string compressed;
    compressed = "";
    for (int i = 0; i < text.length(); i++) {
        std::string key=text.substr(i,2);
        if(codingTable.find(text.substr(i,2))!=codingTable.end()){
            compressed += codingTable.at(key);
            i++;
        }
        else compressed += codingTable.at(key.substr(0,1));
    }
    std::string compressedBits = compressed;
    int numBits = compressedBits.length();
    int padding = (8 - numBits % 8) % 8;
    compressedBits.append(padding, '0');
    compressed="";
    for(int i=0;i<compressedBits.size();i+=8){
        std::string temp=compressedBits.substr(i,8);
        char c=static_cast<char>(std::stoi(temp,nullptr,2));
        compressed+=c;
    }
    std::bitset<64> numBitsBitSet(numBits);
    std::string numBitsStr = numBitsBitSet.to_string();
    std::string numBitsLE;
    for (int i = 0; i < 64; i += 8) {
        std::bitset<8> bits(numBitsStr.substr(i, 8));
        numBitsLE.insert(0, 1, (char)bits.to_ulong());
    }
    compressed = numBitsLE + compressed;
    return compressed;
}
```

二、实验结果分析

见下表。其中我的三个测试文件分别为：**lorem.txt, essay.txt, ebook.txt**，lorem 文本是一段乱数假文，文本较短；essay 文本是一篇完整的学术论文，文本较长；ebook 文本是一本完整的小说作品，文本最长。测试文件的文本都比较贴近生活实际，字符分布较均匀随机，没有过于重复的词频。

测试文件	原字节数	单字符压缩后字节数	多字符压缩后字节数	单字符压缩率	多字符压缩率
test_a	11	5	3	45.45%	27.27%
test_b	106	56	54	52.83%	50.94%
test_c	140	80	77	57.14%	55.00%
test_d	1238	674	662	54.44%	53.47%
lorem	578	305	305	52.77%	52.77%
essay	53,235	33,330	32,701	62.61%	61.43%
ebook	623,869	352,589	351,614	56.52%	56.36%

由表可知：在不同的文本大小下，多字符压缩方式比单字符压缩效果要好。且随着文本大小逐渐增大，两种方式的压缩率逐渐趋于一致。从理论上分析，当文本字数很大时，出现的两字符串的种类也趋于均匀随机（如 52 个字母只能组成 2500 左右个字符串），此时两字符的字符串与单字符没有本质上的区别，压缩率也就趋于一致。另外，由于我们始终固定了词频最高的 3 个两字符进行编码，3 这个量在文本量极大时几乎可以忽略，也就几乎不会对压缩效果产生影响。经试验，若把 3 改为和文本量相关的数值，可以一定程度上提高多字符的压缩效果。

三、可能更优的压缩策略

经相关资料的查阅，发现被 7zip 软件所采用的 **LZMA 算法**是一种较优的压缩策略。LZMA 算法的大致流程为初始化、字典压缩、预测分析、范围编码、压缩、解压。LZMA 具有低压缩率、数据通用性、较快的解压速度等优势。

除此之外，**上文所提到的方式**也可以显著提高压缩率。如：将选择个数由 3 改为 min(text.size()/10,1000)后，ebook 多字节压缩后字节数降低为 311.2k，essay 字节数降为 28.6k，lorem 降为 288。a,b,c,d 文件因为文本量较小没有什么显著差异。这说明合适的选择策略会提高哈夫曼压缩的效率。