Lab 6：Huffman 编码问题

PB21020718 曾健斌

一、实验内容

编程实现 Huffman 编码问题,并理解其核心思想

- 对字符串进行 01 编码,输出编码后的 01 序列，并比较其相对于定长编码的压缩率
- 对文件`orignal.txt`中所有的大小写字母、数字(0-9)以及标点符号（即除空格 换行符之外的所有字符）按照 Huffman 编码方式编码为 01 序列，将编码表输出至`table.txt`文件，并在控制台打印压缩率

二、算法实现

首先定义结构体

```
struct Node {
    char data;
    int freq;
    Node* left;
    Node* right;
};

struct Compare {
    bool operator()(Node* a, Node* b) {  // 重载()用于优先队列的比较
        return a->freq > b->freq;
    }
};
```

`Node`为 Huffman 树的节点，由于本实现使用 STL 库中的 priority_queue 实现，故定义了包含重载运算符`()`的结构体`Compare`

以下为`Huffman`树的类声明

```
class Huffman {
public:
string text;
    unordered_map<char, int> freqm;  // 统计各字符频次
    unordered_map<char, string> codem;  // 记录各字符编码
    unordered_map<string, char> decodem;  // 记录各编码对应字符
    Node* root;
    string encoded;
    string decoded;
    int bitlength;   // 定长编码所需每字符码长
    float compressionRatio;   // 压缩率

    Huffman(string text) :
    text(text), root(nullptr), compressionRatio(-1), bitlength(8) {};  // 初始令bitlength为 8 (ascii)
    ~Huffman() {};
```

```
    void buildHuffmanTree();
    void buildCodeTable(Node* root, string code);
    void encode();
    void decode();
    void writeEncoded(string filename);
    void writeDecoded(string filename);
    void printCodeTable(int base = 1);
    void getCompressionRatio();
};
```

本实现使用了 STL 中的 unordered_map 来记录键值对

以下为类方法的具体实现：

统计字符频次，并使用一个优先队列构建 Huffman 树

```
void
Huffman::buildHuffmanTree()
{
    priority_queue<Node*, vector<Node*>, Compare> pq;
    unordered_map<char, int> freqmtmp;

    for (char c : text) {
        // Only need a-z, A-Z, 0-9, and punctuation(all characters without space, tab, newline) for this project
        if (c == ' ' || c == '\t' || c == '\n') {   // 不考虑这些字符
            continue;
        }
        freqmtmp[c]++;
    }
    for (auto it : freqmtmp) {   // 每个字符 push 进入优先队列 pq
        Node* node = new Node;
        node->data = it.first;
        node->freq = it.second;
        node->left = nullptr;
        node->right = nullptr;
        pq.push(node);
    }

    while (pq.size() > 1) {      // 利用优先队列的有序特性建立 Huffman 树
        Node* left = pq.top();
        pq.pop();
        Node* right = pq.top();
        pq.pop();
        Node* parent = new Node;
        parent->data = '\0';
        parent->freq = left->freq + right->freq;
        parent->left = left;
        parent->right = right;
        pq.push(parent);
    }
    freqm = freqmtmp;
    root = pq.top();
}
```

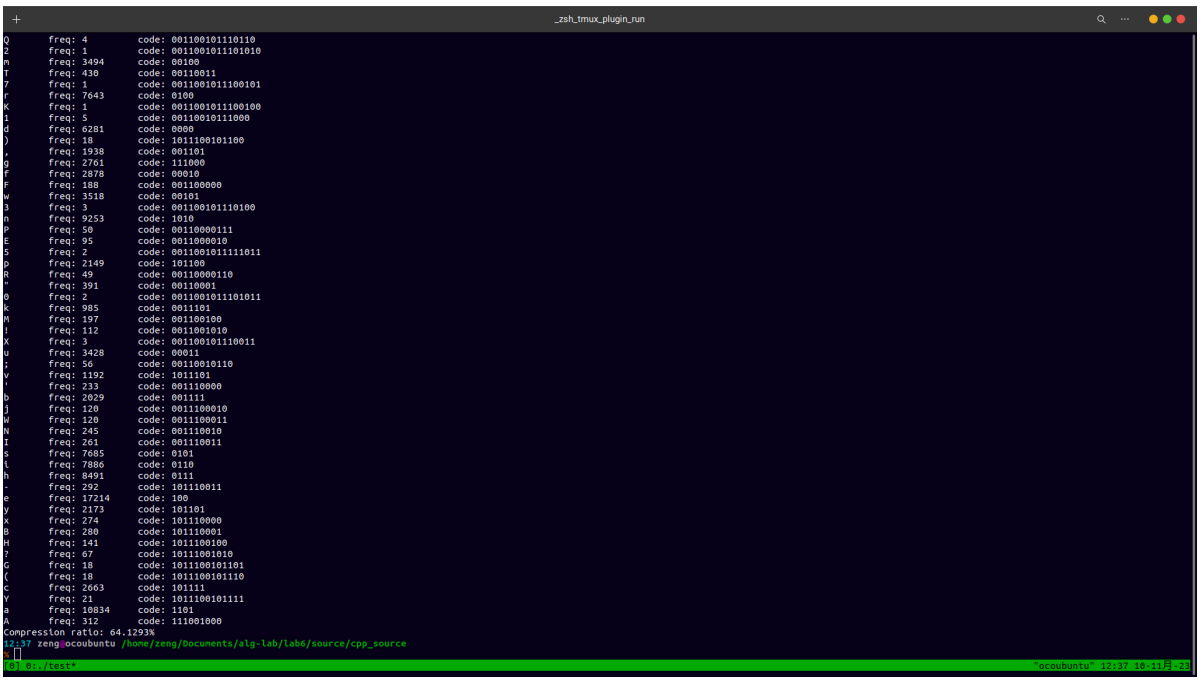
使用递归的手段来遍历 Huffman 树以得到编码

```
void
Huffman::buildCodeTable(Node* root, string code)
{
    if (root == nullptr) {     // 递归临界条件
        return;
    }
    if (root->data != '\0') {   // 遇到字符则记录编码
        codem[root->data] = code;
        decodem[code] = root->data;
    }
    buildCodeTable(root->left, code + "0");
    buildCodeTable(root->right, code + "1");
}
```

计算压缩率

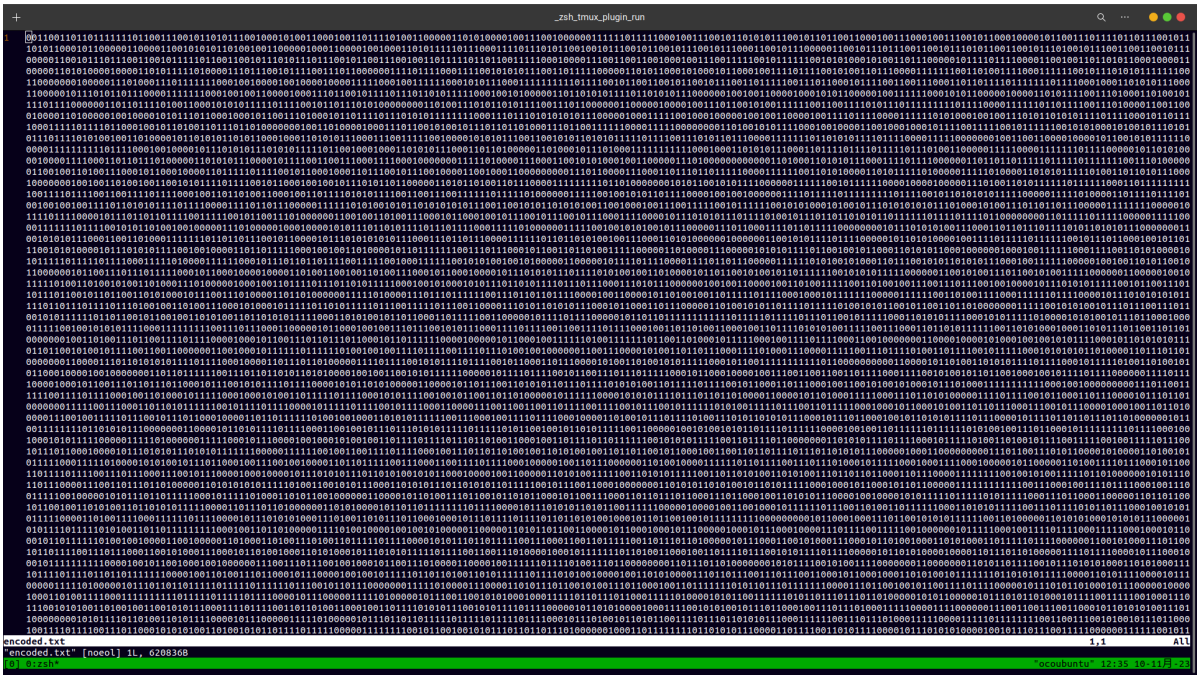
```
void
Huffman::getCompressionRatio()
{
    bitlength = ceil(log2(codem.size()));     // 计算定长编码长度
    int originalsize = 0;
    int compressedsize = 0;
    for (auto it : codem) {
        originalsize += freqm[it.first] * bitlength;
        compressedsize += freqm[it.first] * size(it.second);
    }
    compressionRatio = (float) compressedsize / originalsize * 100;
}
```

三、实验结果

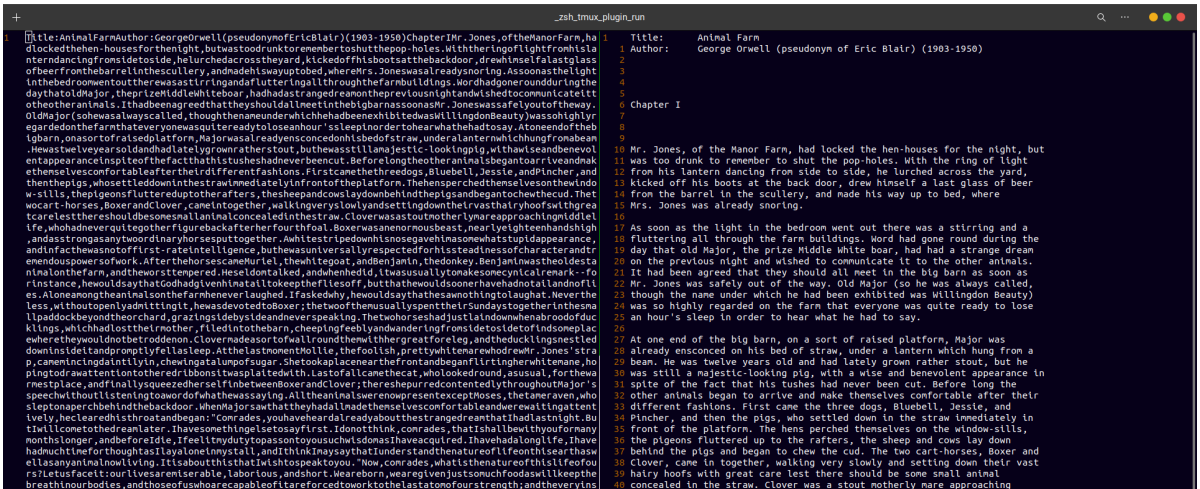


输出各字符对应出现频次以及对应的 Huffman 码

同时输出压缩率，得到64.1293%



输出编码结果



解码后与原文本进行对比

四、实验总结

本次实验旨在实现 Huffman 编码，总体较为简单

- 切身体会到了 Huffman 码作为一种贪心得到的前缀码的最优性
- priority_queue 和 unordered_map 的使用很有趣