



数学与计算机学院

# 哈夫曼编（译）码系统 设计报告

杨 鑫

计算机 11902 班

2021 年 2 月 24 日

本作品采用 CC-BY-NC-SA 协议进行许可



# 1 需求分析

## 1.1 问题描述

利用哈夫曼编码进行通信可以大大提高信道利用率，缩短信息传输时间，降低传输成本。但是，这要求在发送端通过一个编码系统对待传数据预先编码，在接收端将传来的数据进行译码（解码）。对于双工信道（即可以双向传输信息的信道），每端都需要一个完整的编/译码系统。试为这样的信息收发站设计一个哈夫曼编译码系统。

## 1.2 基本要求

### (1) 初始化 (Initialization)

从数据文件 DataFile.data 中读入字符及每个字符的权值，建立哈夫曼树 HuffTree；

### (2) 编码 (EnCoding)

用已建好的哈夫曼树，对文件 ToBeTran.data 中的文本进行编码形成报文，将报文写在文件 Code.txt 中；

### (3) 译码 (DeCoding)

利用已建好的哈夫曼树，对文件 CodeFile.data 中的代码进行解码形成原文，结果存入文件 Textfile.txt 中；

### (4) 输出 (Output)

输出 DataFile.data 中出现的字符以及各字符出现的频度（或概率）；

输出 ToBeTran.data 及其报文 Code.txt；

输出 CodeFile.data 及其原文 TextFile.txt。

# 2 概要设计

## 2.1 数据结构

### 2.1.1 哈夫曼树结点

本系统的关键实现取决于构造一棵哈夫曼树。哈夫曼树的一个结点包括权值，父结点，左右孩子结点。假设由  $n$  个字符来构造一棵哈夫曼树，则共有结点  $2n - 1$  个。我们以线性结构实现哈夫曼树，详见下文描述。

哈夫曼树结点的数据成员：

- ch：当前结点代表的字符，根据哈夫曼树的特点，非叶子结点值为空
- weight：当前结点的权值
- parent：当前结点的父结点序号，若无父结点，设其为 0

- lchild: 当前结点的左孩子结点序号, 若无左孩子, 设其为 0
- rchild: 当前结点的右孩子结点序号, 若无右孩子, 设其为 0

HTNode
+ ch: string
+ weight: size_t <sup>1</sup>
+ parent: size_t
+ lchild: size_t
+ rchild: size_t

1

### 2.1.2 系统综合结构

实现哈夫曼编(译)码系统, 主要依据一棵哈夫曼树, 以及其生成的哈夫曼编码表。将上述多个数据结构, 以及相应的构造、编码、译码算法封装, 形成一个“哈夫曼编码树”类。这些算法将在下节论述。

Huffman
- tree: HTNode*
- size: size_t
- code: string*
- totalSize(): size_t
- select(n: size_t, s1: size_t&, s2: size_t&)
- init(n: size_t, _ch: string*, _weight: size_t*)
+ Huffman()
+ Huffman(n: size_t, _ch: string*, _weight: size_t*)
+ Huffman(is: istream&)
+ operator=(h: Huffman&&): Huffman&
~ Huffman()
+ printTree()
- enCode()
+ enCode(is: istream&): string
+ enCode(str: const string&): string
+ printCode()
+ deCode(is: istream&): string
+ deCode(str: const string&): string

Huffman 类数据成员:

---

<sup>1</sup>size\_t 是无符号超长整型, 即 unsigned long long。

- tree: 一棵以动态数组实现的哈夫曼树, 数组下标即代表了当前的序号, 0 号单元不使用
- size: 字符数, 也代表叶子结点数, 假定  $n$  个, 则对应  $1 \sim n$  单元结点的字符值非空
- code: 存储各字符编码后的二进制编码, 因为 tree 的 0 号单元不使用, 所以表长为字符数加 1

Huffman 类成员函数:

- totalSize: 计算 tree 的总大小, 等同于  $2 * size + 1$ , 即 tree 结点总数
- select: 将两个双亲域为 0 且权值最小的结点的下标赋给参数  $s1, s2$
- init: 初始化哈夫曼树
- Huffman: 构造哈夫曼树, 可以多种方式构造 (从字符和权值数组、从输入流)
- operator=: 移动复制操作符, 将右值引用复制给当前 Huffman 对象
- printTree: 输出当前 Huffman 对象的关键参数 (各结点的字符值、权值、父子结点序号)
- enCode: 编码, 可以是对系统所有字符编码, 或者, 对输入的一串字符串进行编码;  
在构造哈夫曼树时, 就经历了初始化 (init) 和编码 (enCode) 阶段。
- printCode(): 输出系统各字符的二进制编码
- deCode: 译码, 翻译二进制字符串

## 2.2 算法

### 2.2.1 构造哈夫曼树

假如由  $n$  个字符来构造一棵哈夫曼树, 则共有结点  $2n - 1$  个。

构造哈夫曼树的操作是, 将  $1 \sim n$  号结点作为叶子结点, 输入各结点的字符、权值。并将他们的 parent、lchild、rchild 均赋值为 0。

然后, 在当前哈夫曼树中选择权值最小且双亲域为 0 的两个结点作为下一结点的子结点, 新结点的权值为这两个结点的权值之和。

将上述操作称哈夫曼树的初始化算法, 见算法描述??。

然后进行对各字符的编码操作 (见下一小节)。

### 2.2.2 编码

编码操作一, 生成构造哈夫曼树的各个字符<sup>2</sup>的编码表。

编码操作一的思想是“逆序编码”, 从叶子结点出发, 向上回溯, 如果该结点是回溯到上一个结点的左孩子, 则在记录编码的字符串里追加“0”, 否则追加“1”, 注意是倒着追加; 直到

---

<sup>2</sup>此处为方便系统实现, 用"(space)" (不含引号) 表示在编码表中的空格字符。

遇到根结点（结点双亲为 0），每一次循环编码到根结点，把编码存在编码表中，然后开始编码下一个字符（叶子）。

详见算法描述??。

编码操作二，对输入的一串字符串编码，此时，只需遍历该字符串，在编码表寻找每个字符对应的二进制编码，然后将这些二进制编码“串接”在一块。

### 2.2.3 译码

译码的思想是循环读入一串哈夫曼序列，读到“0”从根结点的左孩子继续读，读到“1”从右孩子继续，如果读到一个结点的左孩子和右孩子是否都为 0，说明已经读到了一个叶子（字符），翻译一个字符成功，把该叶子结点代表的字符追加到一个字符串中，然后继续从根结点开始读，直到读完这串哈夫曼序列。如果读到最后哈夫曼序列的最后一个位置，却不是叶子结点，说明不是一个有效的哈夫曼序列。详见算法描述??。

---

**算法 1: 初始化哈夫曼树**

---

输入: 字符数 n, 字符数组 ch, 权值数组 weight

```

1 if  $n \leq 1$  then
2   | 字符数过少, 无法初始化;
3   | STOP;
4 else
5   |  $size \leftarrow n$ 
6 end
7 为 tree 分配  $2 * size - 1$  的存储空间;
8 for  $i \leftarrow 1$  to  $2 * size - 1$  do
9   | if  $i \leq size$  then
10    |   |  $tree[i].ch \leftarrow ch[i - 1]$ ;
11    |   |  $tree[i].weight \leftarrow weight[i - 1]$ ;
12   | end
13   |  $tree[i].lchild \leftarrow 0$ ;
14   |  $tree[i].rchild \leftarrow 0$ ;
15   |  $tree[i].parent \leftarrow 0$ ;
16 end
17 for  $i \leftarrow size + 1$  to  $2 * size - 1$  do
18   | 选择权值最小且双亲域为 0 的结点 s1, s2;
19   |  $tree[s1].parent \leftarrow i$ ;
20   |  $tree[s2].parent \leftarrow i$ ;
21   |  $tree[i].lchild \leftarrow s1$ ;
22   |  $tree[i].rchild \leftarrow s2$ ;
23   |  $tree[i].weight \leftarrow tree[s1].weight + tree[s2].weight$ ;
24 end

```

---

---

**算法 2: 编码**

---

输出: 编码表

```

1 为编码表 code 分配 size + 1 的存储空间;
2 for  $i \leftarrow 1$  to size do
3   current  $\leftarrow i$ ;
4   father  $\leftarrow tree[current].parent$ ;
5   while father! = 0 do
6     if tree[father].lchild == c then
7       在 code[i] 的初始位置插入字符'0';
8     else
9       在 code[i] 的初始位置插入字符'1';
10    end
11   current  $\leftarrow father$ ;
12   father  $\leftarrow tree[father].parent$ ;
13 end
14 end

```

---



---

**算法 3: 译码**

---

输入: 二进制哈夫曼序列 str

输出: 翻译后的字符串 ans

```

1  $q \leftarrow totalSize()$ ;
2 for  $i \leftarrow 0$  to str 的长度 - 1 do
3   while father  $\neq 0$  do
4     if str[i] == '0' then
5        $q \leftarrow tree[q].lchild$ ;
6     else if str[i] == '0' then
7        $q \leftarrow tree[q].rchild$ ;
8     else
9       抛出异常;
10    end
11 end
12 if tree[q] 为叶子结点 then
13   ans += tree[q].ch;
14   q  $\leftarrow totalSize()$ ;
15 else if  $i == str.size() - 1$  then
16   抛出异常;
17 end
18 end

```

---

### 3 详细设计

#### 3.1 类的函数实现

##### 3.1.1 HTNode 类

HTNode 类的函数实现见代码??

```

1 struct HTNode {
2     string ch{" "};                                // 字符
3     size_t weight{0};                             // 权值
4     size_t lchild{0}, rchild{0}, parent{0}; // 父、子结点序号
5 };

```

代码 1: HTNode 类的实现

##### 3.1.2 Huffman 类

Huffman 类的函数实现见代码??

```

1 class Huffman {
2 private:
3     HTNode *tree; // 以动态数组存储哈夫曼树
4     size_t size; // 字符数
5     string *code; // 哈夫曼编码表
6
7     // 构造函数
8 public:
9     // 默认构造一棵空树
10    Huffman() {
11        tree = nullptr;
12        size = 0;
13        code = nullptr;
14    }
15
16    // 从字符数组和权值数组中构造一棵哈夫曼树
17    Huffman(int n, string *_ch, size_t *_weight) {
18        init(n, _ch, _weight);
19        enCode();
20    }
21
22    // 从输入流中构造哈夫曼树

```

```
23     Huffman(std::istream &is) {
24         size_t n;
25         is >> n;
26         auto _ch = new string[n];
27         auto _weight = new size_t[n];
28         for (size_t i = 0; i < n; i++)
29             is >> _ch[i] >> _weight[i];
30         init(n, _ch, _weight);
31         delete[] _ch;
32         delete[] _weight;
33         enCode();
34     }
35
36 // 移动复制
37 Huffman &operator=(Huffman &&h) noexcept {
38     delete[] tree;
39     delete[] code;
40     tree = h.tree;
41     size = h.size;
42     code = h.code;
43     h.tree = nullptr;
44     h.code = nullptr;
45     h.size = 0;
46     return *this;
47 }
48
49 // 析构函数
50 ~Huffman() {
51     delete[] tree;
52     delete[] code;
53     tree = nullptr;
54     code = nullptr;
55 }
56
57 // 输出哈夫曼树的参数
58 void printTree() {
59     cout << "Node-ID "
60             << "Character "
61             << "Weight "
```

```
62         << "Parent-ID "
63         << "lChild-ID "
64         << "rChild-ID"
65         << endl;
66     for (size_t i = 1; i <= size * 2 - 1; i++) {
67         auto _parent = tree[i].parent ? std::to_string(tree[i]
68             ].parent) : "(null)";
69         auto _lchild = tree[i].lchild ? std::to_string(tree[i]
70             ].lchild) : "(null)";
71         auto _rchild = tree[i].lchild ? std::to_string(tree[i]
72             ].rchild) : "(null)";
73         cout << std::left;
74         cout << std::setw(8) << i
75             << std::setw(10) << tree[i].ch
76             << std::setw(7) << tree[i].weight
77             << std::setw(10) << _parent
78             << std::setw(10) << _lchild
79             << std::setw(9) << _rchild << endl;
80     }
81 }
82
83 // 从输入流中编码字符
84 string enCode(std::istream &is) {
85     string str;
86     std::getline(is, str);
87     return enCode(str);
88 }
89
90 // 对一串字符编码
91 string enCode(const string &str) {
92     string ans;
93     for (auto i : str) {
94         bool isFound{false};
95         for (size_t j = 1; j <= size; j++) {
96             if ((i == ' ' && tree[j].ch == "(space)") || i ==
tree[j].ch[0]) {
97                 ans += code[j];
98                 isFound = true;
99                 break;
```

```
97         }
98     }
99     if (!isFound)
100        throw std::invalid_argument("Can not encode this
101           string!");
102    }
103 }
104
105 // 输出编码表
106 void printCode() {
107     cout << "Node-ID "
108           << "Character "
109           << "Weight "
110           << "Code"
111           << endl;
112     for (size_t i = 1; i <= size; i++) {
113         cout << std::left;
114         cout << std::setw(8) << i
115             << std::setw(10) << tree[i].ch
116             << std::setw(7) << tree[i].weight
117             << code[i] << endl;
118     }
119 }
120
121 // 对一串二进制字符进行译码
122 string deCode(const string &str) {
123     size_t q = totalSize();
124     string ans;
125     for (size_t i = 0; i < str.size(); i++) {
126         if (str[i] == '0')
127             q = tree[q].lchild;
128         else if (str[i] == '1')
129             q = tree[q].rchild;
130         else {
131             throw std::invalid_argument("Can not decode this
132               string!");
133         }
```

```

134     if (tree[q].lchild == 0 && tree[q].rchild == 0) {
135         ans += (tree[q].ch == "(space)" ? " " : tree[q].
136                 ch);
137     } else if (i == str.size() - 1) {
138         throw std::invalid_argument("Can not decode this
139                         string!");
140     }
141
142     return ans;
143 }
144
145 // 从输入流中译码
146 string deCode(std::istream &is) {
147     string str;
148     std::getline(is, str);
149     return deCode(str);
150 }
151
152 // 内部函数
153 private:
154     // 结点总数
155     inline size_t totalSize() {
156         return 2 * size - 1;
157     }
158
159     // 将两个双亲域为0且权值最小的结点的下标赋给参数 s1, s2
160     void select(size_t n, size_t &s1, size_t &s2) {
161         //***** 前两个循环找s1 *****/
162         // 找出一个双亲为0的结点
163         for (size_t i = 1; i <= n; i++) {
164             if (tree[i].parent == 0) {
165                 s1 = i; // s1 初始化为i
166                 break;
167             }
168         }
169         // 找到权值最小的结点, 且双亲为0
170         for (size_t i = 1; i <= n; i++) {

```

```

171         if (tree[i].weight < tree[s1].weight && tree[i].
172             parent == 0)
173             s1 = i;
174     }
175
176     //*****后两个循环找s2 *****/
177     // 找出一个双亲为0的结点，并且不是s1
178     for (size_t i = 1; i <= n; i++) {
179         if (tree[i].parent == 0 && i != s1) {
180             s2 = i; // s2 初始化为i
181             break;
182         }
183     }
184     // 找到权值第二小的结点，且双亲为0
185     for (size_t i = 1; i <= n; i++) {
186         if (tree[i].weight < tree[s2].weight && tree[i].
187             parent == 0 && i != s1)
188             s2 = i;
189     }
190
191     // 初始化
192     void init(size_t n, string *_ch, size_t *_weight) {
193         if (n <= 1) {
194             size = 0;
195             return;
196         } else {
197             size = n;
198         }
199         tree = new HTNode[totalSize() + 1]; // 0号单元未使用
200         // 初始化树
201         for (size_t i = 1; i <= totalSize(); i++) {
202             if (i <= size) {
203                 tree[i].ch = _ch[i - 1];
204                 tree[i].weight = _weight[i - 1];
205             }
206             tree[i].lchild = 0;
207             tree[i].rchild = 0;
208             tree[i].parent = 0;

```

```

208     }
209     // 建成哈夫曼树
210     for (size_t i = size + 1; i <= totalSize(); i++) {
211         // 在 tree[1~i-1] 中选择 parent 为 0, 且 weight 最小的两个结
212         // 点
213         size_t s1, s2;
214         select(i - 1, s1, s2);
215         tree[s1].parent = tree[s2].parent = i;
216         // 将 s1, s2 的双亲改为 i
217         tree[i].lchild = s1;
218         // 将 i 的左孩子改为 s2
219         tree[i].rchild = s2;
220         // i 的权值为左右孩子权值之和
221         tree[i].weight = tree[s1].weight + tree[s2].weight;
222     }
223 }

224
225     // 对所有字符进行编码
226     void enCode() {
227         code = new string[size + 1];
228         for (size_t i = 1; i <= size; i++) {
229             auto c = i; // 当前结点
230             auto f = tree[c].parent; // 指向当前的父结点
231             while (f != 0) { // 从叶子结点开始回溯, 直到
232                 // 根结点
233                 if (tree[f].lchild == c)
234                     code[i].insert(code[i].begin(), '0');
235                 else
236                     code[i].insert(code[i].begin(), '1');
237                 c = f;
238                 f = tree[f].parent; // 向上回溯
239             }
240         }
241     };

```

代码 2: HTNode 类的实现

## 3.2 程序模块

main、menu、showFile、importFile、writeToFile 函数，用于组织程序结构：

```
1 int main() {
2     cout << "Welcome to Huffman Encoding and Decoding System!" <<
3         endl;
4     Hf h;           // 实例化一个哈夫曼编码树对象
5     ifstream is; // 实例化一个输入文件流对象
6     while (true) {
7         switch (menu()) {
8             case 1: {
9                 // 尝试导入文件
10                try {
11                    importFile(is);
12                    h = Hf(is); // 匿名构造，然后复制之
13                    is.close();
14                } catch (std::invalid_argument &e) {
15                    std::cerr << e.what() << endl;
16                    break;
17                }
18                cout << "\nInit succeeded!\n\n"
19                     << "Here are the Overview of the Huffman
20                         Tree:\n\n";
21                h.printTree();
22            } break;
23            case 2: {
24                // 尝试导入文件
25                try {
26                    importFile(is);
27                } catch (std::invalid_argument &e) {
28                    std::cerr << e.what() << endl;
29                    break;
30                }
31                string code; // 用以存储二进制哈夫曼编码
32                // 尝试编码
33                try {
34                    code = h.enCode(is);
35                } catch (std::invalid_argument &e) {
```

```
35         std::cerr << e.what() << endl;
36         is.close();
37         break;
38     }
39     is.close();
40     cout << "\nEnCode succeeded!\n\n"
41         << "The string has been transformed to:\n"
42         ""
43         << code << endl
44         << endl;
45     // 将编码写入Code.txt
46     writeToFile("Code.txt", code);
47 } break;
48 case 3: {
49     // 尝试导入文件
50     try {
51         importFile(is);
52     } catch (std::invalid_argument &e) {
53         std::cerr << e.what() << endl;
54         break;
55     }
56     string str; // 用以存储译码后的字符串
57     try {
58         str = h.deCode(is);
59     } catch (std::invalid_argument &e) {
60         std::cerr << e.what() << endl;
61         is.close();
62     }
63     is.close();
64     cout << "\nDeCode succeeded!\n\n"
65         << "The string has been translated to:\n"
66         << str << endl
67         << endl;
68     // 将译码后的字符串写入Textfile.txt
69     writeToFile("Textfile.txt", str);
70 } break;
71 case 4: {
72     cout << "\nCharacter Table:" << endl;
73     h.printCode(); // 显示Datafile.data中
```

```

    的字符频率

73     showFile("ToBeTran.data"); // 显示 ToBeTran.data
74     showFile("Code.txt");      // 显示 Code.txt
75     showFile("CodeFile.data"); // 显示 CodeFile.data
76     showFile("TextFile.txt");  // 显示 TextFile.txt
77 } break;
78 case 0:
79     cout << "Thanks for using!" << endl;
80     exit(0);
81 }
82 }
83 return 0;
84 }
```

代码 3: main 函数

```

1 int menu() {
2     auto sn{0};
3     cout << endl
4         << "----- Menu -----" << endl
5         << "1. Init system" << endl
6         << "2. Encoding" << endl
7         << "3. Decoding" << endl
8         << "4. Output" << endl
9         << "0. Exit" << endl
10        << "----- Menu ----->" << endl
11        << "Pleas input 0-4: ";
12     while (true) {
13         cin >> sn;
14         if (sn < 0 || sn > 4) {
15             cout << "Error Number! Please Input 0-4: ";
16         } else
17             break;
18     }
19     return sn;
20 }
```

代码 4: menu 函数

```

1 void showFile(const string &filename) {
2     cout << endl;
```

```

3     cout << filename << ":" << endl;
4     ifstream is;
5     is.open(filename);
6     if (!is.is_open()) {
7         cout << "Fail to open file " << filename << "!" << endl;
8         return;
9     }
10    string str;
11    getline(is, str);
12    cout << str << endl;
13 }
```

代码 5: showFile 函数

```

1 ifstream &importFile(ifstream &is) {
2     string filename;
3     cout << "Input filename (with '.data'): ";
4     cin >> filename;
5     is.open(filename);
6     if (!is.is_open()) {
7         is.close();
8         throw std::invalid_argument("Fail to open file " +
9             filename + "!");
10    }
11    return is;
12 }
```

代码 6: importFile 函数

```

1 void writeToFile(const string &filename, const string &str) {
2     ofstream os;
3     os.open(filename);
4     if (!os.is_open()) {
5         cout << "Fail to write the result to file " << filename
6             << "!" << endl;
7         return;
8     }
9     os << str << endl;
10    os.close();
11    cout << "The result has been written to " << filename << endl
12        ;
```

12

}

代码 7: writeToFile 函数

## 4 测试分析

测试文件 1: CodeFile.data, 为构造哈夫曼树的字符源, 包含了字符及其权值。

```

27
(space) 186
a 64
b 13
c 22
d 32
e 103
f 21
g 15
h 47
i 57
j 1
k 5
l 32
m 20
n 57
o 63
p 15
q 1
r 48
s 51
t 80
u 23
v 8
w 18
x 1
y 16
z 1

```

第 1 行, 27 表示有 27 个字符。

第 2~28 行, 指出了各字符及其权值。其中空格字符用 “(space)” (不含引号) 替代。

测试文件 2: ToBeTran.data, 为待编码的字符串。

```
hello world
```

测试文件 3: CodeFile.data, 为待译码的二进制哈夫曼编码

```
0000010011100101000100000111010100010
```

## 4.1 构造哈夫曼树

通过 DataFile.data 构造哈夫曼树，在构造完成后，自动展示哈夫曼树的各结点参数。

```
PowerShell > .\Huffman_System.exe
Welcome to Huffman Encoding and Decoding System!

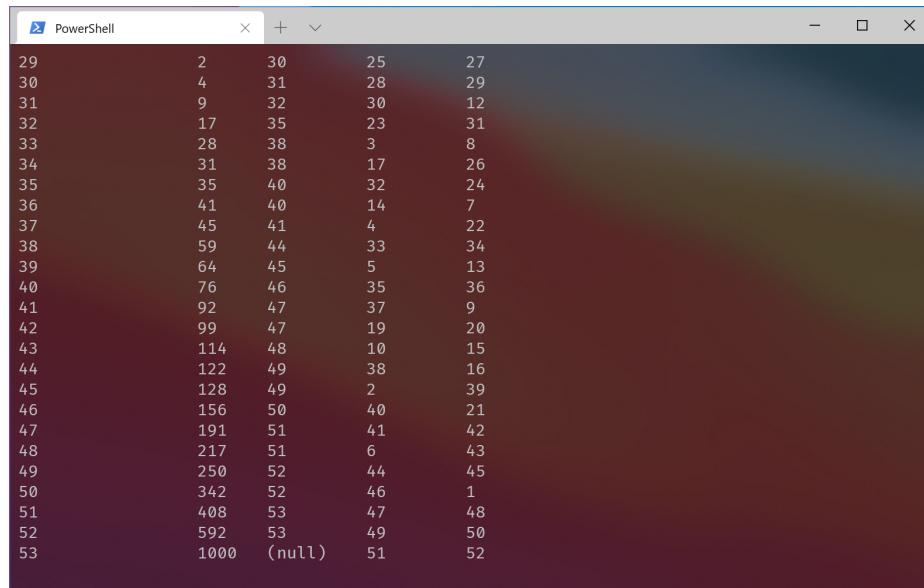
←————— Menu —————→
1. Init system
2. Encoding
3. Decoding
4. Output
0. Exit
————— Menu —————→
Pleas input 0-4: 1
Input filename (with '.data'): DataFile.data

Init succeeded!

Here are the Overview of the Huffman Tree:

Node-ID Character Weight Parent-ID lChild-ID rChild-ID
1      (space)   186    50      (null)   (null)
2      a          64     45      (null)   (null)
3      b          13     33      (null)   (null)
4      c          22     37      (null)   (null)
5      d          32     39      (null)   (null)
```

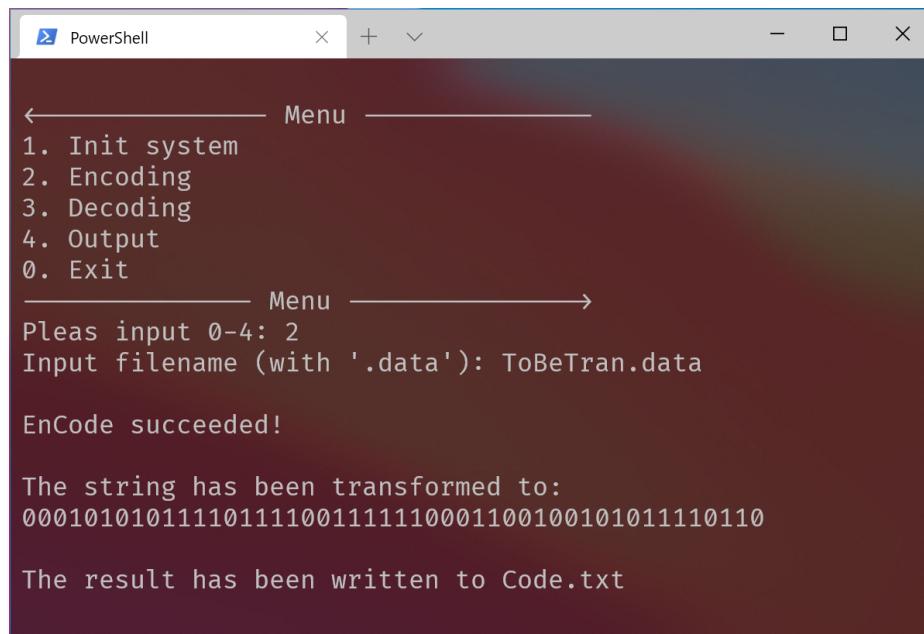
```
PowerShell >
6      e          103   48      (null)   (null)
7      f          21    36      (null)   (null)
8      g          15    33      (null)   (null)
9      h          47    41      (null)   (null)
10     i          57    43      (null)   (null)
11     j          1     28      (null)   (null)
12     k          5     31      (null)   (null)
13     l          32    39      (null)   (null)
14     m          20    36      (null)   (null)
15     n          57    43      (null)   (null)
16     o          63    44      (null)   (null)
17     p          15    34      (null)   (null)
18     q          1     28      (null)   (null)
19     r          48    42      (null)   (null)
20     s          51    42      (null)   (null)
21     t          80    46      (null)   (null)
22     u          23    37      (null)   (null)
23     v          8     32      (null)   (null)
24     w          18    35      (null)   (null)
25     x          1     29      (null)   (null)
26     y          16    34      (null)   (null)
27     z          1     29      (null)   (null)
28
```



29	2	30	25	27
30	4	31	28	29
31	9	32	30	12
32	17	35	23	31
33	28	38	3	8
34	31	38	17	26
35	35	40	32	24
36	41	40	14	7
37	45	41	4	22
38	59	44	33	34
39	64	45	5	13
40	76	46	35	36
41	92	47	37	9
42	99	47	19	20
43	114	48	10	15
44	122	49	38	16
45	128	49	2	39
46	156	50	40	21
47	191	51	41	42
48	217	51	6	43
49	250	52	44	45
50	342	52	46	1
51	408	53	47	48
52	592	53	49	50
53	1000	(null)	51	52

## 4.2 编码字符串

对 ToBeTran.data 中的字符串进行编码，并将结果写入 Code.txt 中。



```
←———— Menu —————→
1. Init system
2. Encoding
3. Decoding
4. Output
0. Exit
————— Menu —————→
Pleas input 0-4: 2
Input filename (with '.data'): ToBeTran.data

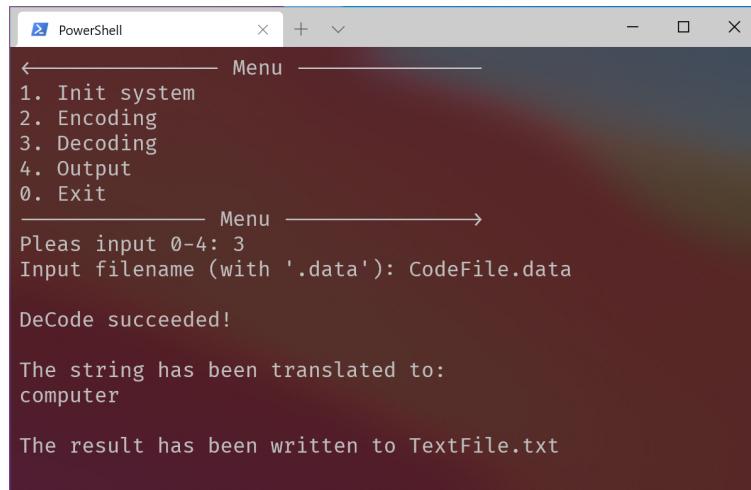
EnCode succeeded!

The string has been transformed to:
000101010111101111001111110001100100101011110110

The result has been written to Code.txt
```

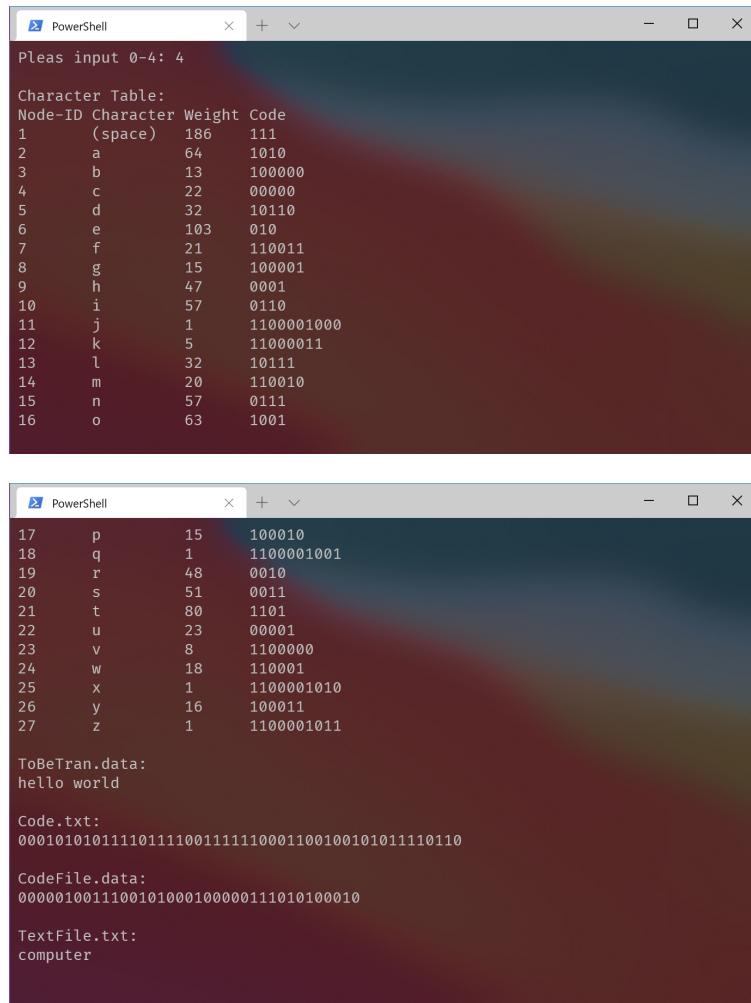
## 4.3 译码二进制哈夫曼编码

对 CodeFile.data 中的字符串进行译码，并将结果写入 TextFile.txt 中。



## 4.4 输出

展示 DataFile.data 中出现的字符及其频率；输出 ToBeTran.data 及其报文 Code.txt；输出 CodeFile.data 及其原文 TextFile.txt。



## 5 总结

通过本次项目实践，理解了哈夫曼树的特征及应用，掌握了哈夫曼树的构造算法设计。熟悉了字符的哈夫曼编码和译码，并能够通过哈夫曼树进行编码和译码操作。通过哈夫曼程序的编写，对计算机处理编码问题有了感性的理解，并将树、哈夫曼等知识得到实际应用。

## 6 附录

本项目实例使用 CMake 构建，并要求编译器为 G++，使用的操作系统是 Windows。

项目主要文件清单：

/.....	.....	项目根目录
└── bin.....	.....	输出文件夹
└── CodeFile.data.....	.....	用以译码的示例文件
└── DataFile.data .....	.....	用以构造哈夫曼树的示例文件
└── Huffman_System.exe .....	.....	已经编译的可执行文件
└── ToBeTran.data.....	.....	用以编码的示例文件
└── CMakeLists.txt.....	.....	CMake 项目配置文件
└── docs .....	.....	项目文档目录
└── images.....	.....	文档使用的图片资源
└── project3.pdf .....	.....	项目文档
└── project3.tex.....	.....	项目文档 LATEX 源文件
└── src .....	.....	源代码
└── main.cpp .....	.....	主程序
└── Huffman.h .....	.....	Huffman 类头文件

这些源文件可以在 <https://github.com/DianDengJun/Course-Design/tree/main/Winter/Project3> 中查看。

构建本实例的命令 (Bash 或 Powershell) 如下：

进入根目录，执行：

```
mkdir build
cd build
cmake -G "MinGW Makefiles" ..
make # 或者是 mingw32-make
```

然后进入 bin 目录运行 Huffman\_System.exe。

```
cd ../bin
./Huffman_System
```