

## 第 2.1 题：哈夫曼编/译码器

### 实验报告

题目：为信息收发站写一个哈夫曼码的编/译码系统。

班级：F1702120 学号：517021910526 姓名：陈聪

#### 一、需求分析

1. 一个完整的系统应具有以下功能：

(1) I: 初始化 (Initialization)。从终端读入字符集大小  $n$ ，以及  $n$  个字符和  $n$  个权值，建立哈夫曼树，并将它存于文件 `hfmTree` 中。

(2) E: 编码 (Encoding)。利用已建好的哈夫曼树，对文件 `plainFile` 中的正文进行编码，然后将结果存入文件 `codeFile` 中。

(3) D: 译码 (Decoding)。利用已建好的哈夫曼树，对文件 `codeFile` 中的代码进行译码，然后将结果存入文件 `textFile` 中。

(4) P: 打印代码文件 (code Printing)。将文件 `codeFile` 显示在终端上，每行 50 个代码。同时将此字符形式的编码文件写入文件 `codePrint` 中。

(5) T: 打印哈夫曼树 (Tree printing)。将哈夫曼树以直观的方式 (树或凹入表形式) 显示在终端中，同时将此字符形式的哈夫曼树写入文件 `treePrint` 中。

2. 程序亮点：在真正的译码编码使用过程中，各个功能相对独立。比如，可能非常久才会初始化一次哈夫曼树，或某天突然需要查看哈夫曼树结构等。这种独立性的需求要求本程序中各模块相互独立，任何模块的完成不依赖于其他模块的完成。笔者程序对独立性的实现、各个函数的封装完成得较好。

#### 二、概要设计

本程序基本概要非常清晰——基于已实现的 Huffman 树类，相对独立地实现上述五个功能。其中，Huffman 树的实现参考并优化了数据结构课本的实现方式，其他部分的具体实现在后续会有详细说明。

##### 1. hftree 类

对象：

```
int length;           //哈夫曼树的节点数
struct Node           //存放哈夫曼树节点的详细信息的节点结构体
{
    char data;         //存放节点所对应的数据，若不为叶节点则为空
    double weight;     //每个节点对应的权重，用于后续的哈夫曼树构建
    int parent, left = -1, right = -1; //父节点、左右孩子在 elem 中的索引值
};
Node *elem;           //哈夫曼树的所有节点数组，存放所有节点
struct hfCode         //用于提取哈夫曼树编码对的结构体 hfcode
{
    char data;         //编码对象字符
```

```
std::string code;} //编码
```

基本操作:

```
hfTree(const std::string &input, const double *weight, const int input_length);  
//初始化函数, 初始化一棵 Huffman 树  
void getCode(hfCode result[]); //用于得到 Huffman 树的编码对  
~hfTree(){delete [] elem;} //析构函数, 返还开辟的空间
```

## 2. 其他函数

```
int counting(const std::string &input, double *numbers, std::string &output)  
//服务于选做题 (initialize 函数), 用于将输入的字符串中的所有元素去重并统计每种元素  
//出现的频次  
//这里使用 double 的原因是为了能和 I 的原始功能重合, 具体解释详见后面的代码注释  
// (位置: P8L24)
```

```
void initialize()  
//I 函数, 其中有两种操作, 一种是输入字符串, 另一种是输入字符和频次
```

```
void translation(const std::string encoder, char* characters, std::string* codes, int iter_string)  
//Initialize 将哈夫曼树存入了 txt 文件中, 这个函数用于将 txt 文件中的数据提取成符号-编码  
//对
```

```
void Encoding()  
//E 函数, 利用已建好的哈夫曼树的编码对, 对文件 plainFile 中的正文进行编码, 然后将结  
//果存入文件 codeFile 中。
```

```
void Decoding()  
//D 函数, 利用已建好的哈夫曼树的编码对, 对文件 codeFile 中的代码进行译码, 然后将结  
//果存入文件 textFile 中。
```

```
void Printing()  
//P 函数, 将文件 codeFile 显示在终端上, 每行 50 个代码。同时将此字符形式的编码文件  
//写入文件 codePrint 中。
```

```
void Table(int current_hierachy, std::string current_string, std::string* codes, char* characters, int  
numbers, int side, std::ofstream &file1)  
//递归实现树打印的函数。
```

其中, `current_hierachy` 用于表示当前的层级以输出足够多的 `/t` 来得到凹表。

`current_string` 用于表示当前层级的字符串, 用于和 `translation` 得到的字符串进行比对以确认是否达到了想要到达的叶子节点

`codes` 和 `characters` 是 `translation` 函数的结果, 用于表示之前构建的哈夫曼树的符号-编码对  
`numbers` 为 `codes` 和 `characters` 的元素个数, 便于编程中 `for` 函数的设计

`side` 用于确认当前是左叶子还是右叶子节点

`File1` 即为需要写入的文件

本函数比较复杂，后续会有更详细的介绍（page 14）

```
void Tree()
```

//T 函数，用于输出树结构，笔者选用了凹表来表示树状结构，该函数主要用于调用 Table 函数前的准备工作。

### 三、 详细设计

以下列出两段在笔者程序中反复出现的代码，这两段代码因为自身特性难以进行封装故笔者选择了重用代码，为方便阅读代码，对这两段代码单独列出，使用时则在注释中标出：

第一段：

```
std::cin.clear();  
std::cin.sync();
```

这段代码用于对输入缓冲区的清除，对 char 类型变量赋值时常出现缓冲区有多余元素的情况，故本段代码经常出现在各程序中。

第二段：

```
std::ifstream file;  
file.open("hfmTree.txt");  
std::string encoder;  
getline(file, encoder);  
std::cout<<encoder;  
file.close();  
//读取文件中的所有内容  
int iter_string = 0, valid_length = 0;  
while(encoder[iter_string]>= '0' && encoder[iter_string] <= '9')  
{  
    valid_length = encoder[iter_string] - '0' + 10 * valid_length;  
    iter_string += 1;  
}  
//I 函数将编码个数存在了 txt 文件中，先读取得到这个长度（page8line31）  
std::cout<<"encoder"<<std::endl<<"length is : "<<valid_length<<std::endl;  
char *characters = new char[valid_length];  
std::string *codes = new std::string[valid_length];  
//利用长度定义符号和编码的数组  
translation(encoder, characters, codes, iter_string);  
//利用 translation 函数为符号和编码数组赋值
```

这段代码用于将 I 函数写入 txt 文件的内容翻译成 符号-编码对（characters-codes 对应元素即为符号与其对应编码），因为其中涉及变量的初始化等操作，难以封装。

//笔者在报告完成后在程序中每个打开文件操作后都加了判断是否成功打开文件的操作，但因报告已写成，未能在报告中体现，特此说明

## 1. 哈夫曼树类

*hftree.h*

```
class hfTree{
private:
    int length;                //哈夫曼树的节点数
    struct Node                //存放哈夫曼树节点的详细信息的节点结构体
    {
        char data;            //存放节点所对应的数据，若不为叶节点则为空
        double weight;        //每个节点对应的权重，用于后续的哈夫曼树构建
        int parent, left = -1, right = -1; //父节点、左右孩子在 elem 中的索引值
    };
public:
    Node *elem;                //哈夫曼树的所有节点数组，存放所有节点
    struct hfCode              //用于提取哈夫曼树编码对的结构体 hfcode
    {
        char data;            //编码对象字符
        std::string code;     //编码
    };
    hfTree(const std::string &input, const double *weight, const int input_length);
    //初始化函数，初始化一棵 Huffman 树
    void getCode(hfCode result[]); //用于得到 Huffman 树的编码对
    ~hfTree(){delete [] elem;}     //析构函数，返还开辟的空间
};
```

hftree.cpp

//初始化哈夫曼树

//基本思路：后 length 个 node 用于存放叶子节点（也即输入的字符和权重），复制即可

//前 length 个节点用于存放非叶子节点，这时需要每次对后面的所有节点遍历，找到最小的两个节点，作为下一个父节点的左右孩子（也即哈夫曼树的基本思路）

hfTree::hfTree(const std::string &valid\_input, const double \*weight, const int input\_length)

```
{
    int length_2 = input_length;
    length = 2 * input_length;
    const double MAX_INT = 32767;
    double min1,min2;
    int x,y;
    elem = new Node[length];
    for (int i = length_2; i < length; ++i)
    {
        elem[i].weight = weight[i - length_2];
        elem[i].data = valid_input[i - length_2];
        elem[i].parent = elem[i].left = elem[i].right = 0;
    }
```

//到这里，后 length 项都已经复制成了输入的字符和权重，一下就是树的建立过程

```
for (int i = length_2 - 1; i > 0; --i)
```

```
{
    min1 = min2 = MAX_INT; x = y = 0;
```

//min1 是最小的节点，min2 是倒数第二小的节点。x、y 分别是这两个节点的下标

```
    for (int j = i + 1; j < length; ++j)
    {
        if(elem[j].parent == 0)
        {
            if(elem[j].weight < min1)
            {
                min2 = min1; min1 = elem[j].weight;
                x = y; y = j;
            }
            else if(elem[j].weight < min2)
            {
                min2 = elem[j].weight; x = j;
            }
        }
    }
```

```
    elem[i].weight = min1 + min2;
    elem[i].left = x; elem[i].right = y; elem[i].parent = 0;
    elem[x].parent = i; elem[y].parent = i;}
```

//哈夫曼树的建立过程，对两个最小的节点进行合并

```

void hfTree::getCode(hfCode result[])
//用于得到哈夫曼树每个节点对应的符号-编码对的函数，符号编码对存在 result 中
{
    int length_2 = length / 2;    //哈夫曼树的符号个数为树节点数的一半
    int parent, present;
    //parent 用于存放父节点的索引，用于确认当前节点是左孩子还是右孩子，present 为
    当前节点的索引
    for (int i = length_2; i < length; ++i)
    {    //每次遍历前先对各个变量初始化一下

        result[i-length_2].data = elem[i].data;
        result[i-length_2].code = "";
        parent = elem[i].parent;
        present = i;
        while(parent)
        {
            //如果是左孩子，编码加一个 0，右孩子则加一个 1
            if (elem[parent].left == present)
            {result[i - length_2].code = '0' + result[i-length_2].code;}
            else{result[i - length_2].code = '1' + result[i-length_2].code;}
            present = parent;
            parent = elem[parent].parent;}}}

```

//该函数主要用于实现选作部分，对字符串进行了去重和计数。思路比较简单。

```

int counting(const std::string &input, double *numbers, std::string &output)
{
    int length = input.length();
    int valid_length = 0;//nature length
    for (int iter = 0; iter < length; iter++)
    {
        for (int iter_valid = 0; iter_valid <= valid_length; iter_valid++)
        {
            //如果已经有记录（也即与前面值重复），对所记数目加一
            if(input[iter] == output[iter_valid])
            {
                numbers[iter_valid] ++;
                break;}
            //如是还未见过的值（与前面值不重复），添加一个新值进入去重表
            if(iter_valid == valid_length)
            {
                output[iter_valid] = input[iter];
                numbers[iter_valid] = 1;
                valid_length++;
                break:   }}}
    return valid_length;}

```

```

//1 函数，用于哈夫曼树的初始化和存储
void initialize()
{
    char switches;
    std::string rubbish;
    std::cout<<" ----- "<<std::endl;
    std::cout<<" |There are two legal operations you can choose from:
| "<<std::endl;
    std::cout<<" |1. type 1 : Input a sequence of words, which is simple and convenient. |
"<<std::endl;
    std::cout<<" |2. type 2 : Input the characters-frequency pairs. |
"<<std::endl;
    std::cout<<" ----- "<<std::endl;

    std::cin>>switches;
    std::cin.clear();
    std::cin.sync();
    //得到选用的模式
    int length_valid;
    double *numbers;
    std::string input_valid;
    //这一段没有用 switch 而用了 if，因为其中涉及到变量的定义和初始化，switch 不支持
    if(switches == '1')
    {
        std::string input;
        std::cout<<"Please input the words :\n";
        getline(std::cin,input);
        int length = input.length();
        numbers = new double [length]{0};
        std::string input_temp(length,' ');
        input_valid = input_temp;
        length_valid = counting(input, numbers, input_valid);
        //实现了选作部分
    }
    else
    {
        if(switches == '2')
        {
            std::cout<<"Please input the number of characters :\n";
            std::cin>>length_valid;
            numbers = new double [length_valid]{0};
            std::cout<<"Please input the characters-frequency pair :\n";
            for(int iter = 0; iter < length_valid; iter++)
            {

```

```

        std::cout<<"The "<<iter+1<<" character is :"<<std::endl;
        std::cin.clear();
        std::cin.sync();
        std::cin.get(switches);
        std::cin.clear();
        std::cin.sync();
        std::cout<<"The "<<iter+1<<" frequency is :"<<std::endl;
        std::cin>>numbers[iter];
        input_valid = input_valid + switches;
        //最原始的读取方式，不断 get 符号和频次/频率即可
    }
}
else
{
    std::cout<<"Fatal error: Wrong input! Please rear the instruction
carefully!"<<std::endl;
    return;
}
}
std::ofstream write;
write.open("hfmTree.txt",std::ios::out);
for (int i = 0; i < length_valid; i++){std::cout<<numbers[i]<<std::endl;}
hfTree tree1(input_valid, numbers, length_valid);
//这里就体现了之前把 numbers 设成 double 的好处了：
//题目要求的输入不仅有频次，还有可能是频率（小数），预先设成 double 可以免去重
//载/写模板的工作，增加代码效率和可读性
hfTree::hfCode *result = new hfTree::hfCode[length_valid];
tree1.getCode(result);
//得到符号-代码对应关系
write<<length_valid;    //txt 中先写入的是字符个数，方便后续操作
int store_min1 = 0;
int flag = 0, delete_it = 0;
write<<" ";
for (int i = 0; i < length_valid; i++)
{
    flag = 0;
    for (int k = 0; k < length_valid; k++)
    {
        if (result[k].code != "" && flag == 0)
        {
            store_min1 = result[k].code.length();
            flag = 1; delete_it = k;
        }
        if (result[k].code != "" && (result[k].code.length() < store_min1 ||

```



```
result[k].code.length() == store_min1 && result[k].code < result[delete_it].code ))
    {
        store_min1 = result[k].code.length();
        delete_it = k;
    }
}
write<<result[delete_it].data<<result[delete_it].code<<" ";
result[delete_it].code = "";
}
```

//这里在写入文件之前对字符串排了一下序，这个排序并不必要，笔者排序主要有两个原因，一是方便人工检查，如果是乱序，很难检查后续的 E、D 操作答案是否正确。二是为了文件内容美观。

```
write.close();
}
```

//translation 函数用于根据 I 函数保存的 txt 文件得到字符-编码对，并分别保存在 characters 和 codes 中，主要用于 E、D、T 三个函数中

```
void translation(const std::string encoder, char*characters, std::string* codes, int iter_string)
{
    //根据 I 函数保存的格式，对内容进行遍历即可。笔者的保存方式是空格+ 符号+编码，
    //本函数逻辑比较简单
    int length = encoder.length();
    int iter_char = -1, iter_code = -1;

    for (; iter_string < length; iter_string++)
    {
        if(encoder[iter_string] == ' ')
        {
            characters[iter_char] = encoder[iter_string + 1];
            iter_char++;
            iter_code++;
            iter_string++;
        }
        if(encoder[iter_string]>= '0' && encoder[iter_string] <= '9')
        {
            codes[iter_code] = codes[iter_code] + encoder[iter_string];
        }
    }
}
```

//E 函数，主要思路是遍历匹配

```
void Encoding()
{
    std::ifstream write;
    write.open("hfmTree.txt");
    std::string encoder;
    getline(write, encoder);
    std::cout<<encoder;
    //get the contents

    int iter_string = 0, valid_length = 0;
    while(encoder[iter_string]>= '0' && encoder[iter_string] <= '9')
    {
        valid_length = encoder[iter_string] - '0' + 10 * valid_length;
        iter_string += 1;
    }
    std::cout<<"encoder"<<std::endl<<"length is :"<<valid_length<<std::endl;
    char *characters = new char[valid_length];
    std::string *codes = new std::string[valid_length];
}
```

```

translation(encoder, characters, codes, iter_string);
write.close();
//以上部分为在之前列出的复用函数，用于提取得到字符-编码对
std::string target;
write.open("plainFile.txt");
getline(write, target);
//遍历找到相同的字符，添加对应的编码即可
int target_length = target.length();
std::string output = "";
for (int i = 0; i < target_length; ++i)
{
    for(int f = 0; f < valid_length; f++)
    {
        if(target[i] == characters[f])
        {
            output = output + codes[f];
            break;
        }
    }
}
std::ofstream write_wow;
write_wow.open("codeFile.txt",std::ios::out);
write_wow<<output;
write.close();
write_wow.close();
}

```

//D 函数，解码，主要思路是遍历找到相同的字符串，即为解码成功

```
void Decoding()
{
    std::ifstream file;
    file.open("hfmTree.txt");
    std::string encoder;
    getline(file, encoder);
    std::cout<<encoder;
    file.close();
    int iter_string = 0, valid_length = 0;
    while(encoder[iter_string]>= '0' && encoder[iter_string] <= '9')
    {
        valid_length = encoder[iter_string] - '0' + 10 * valid_length;
        iter_string += 1;
    }
    std::cout<<"encoder"<<std::endl<<"length is :"<<valid_length<<std::endl;
    char *characters = new char[valid_length];
    std::string *codes = new std::string[valid_length];
    translation(encoder, characters, codes, iter_string);
    //以上部分为在之前列出的复用函数，用于提取得到字符-编码对
    std::string decoded = "", detector = "";
    int start = 0, length_to_detect = 0;
    file.open("codeFile.txt");
    getline(file, decoded);
    file.close();
    //得到待解码的字符串
    std::ofstream file1;
    file1.open("textFile.txt");
    while(start != decoded.length())
    {
        //遍历找到相同的字符串即可，逻辑比较简单
        for (int i = 0; i < valid_length; i++)
        {
            length_to_detect = codes[i].length();
            detector = decoded.substr(start, length_to_detect);
            if(detector == codes[i])
            {
                file1<<characters[i];
                start += length_to_detect;
            } }
        file1.close();
    }
```

// P 函数，基本思路是每行输出 50 个即可，逻辑非常简单

```
void Printing()
{
    std::ifstream file;
    file.open("codeFile.txt");
    std::string content, sub;
    getline(file, content);
    file.close();
    std::ofstream file1;
    file1.open("codePrint.txt");
    int start = 0;
    int length = content.length();
    int cycle = length/50;
    for(int i = 0; i < cycle; i++)
    {
        sub = content.substr(start, 50);
        std::cout<<sub<<std::endl;
        file1<<sub<<std::endl;
        start = start + 50;
    }
    sub = content.substr(start, length - start);
    std::cout<<sub<<std::endl;
    file1<<sub;
    file1.close();
}
```

```

//递归实现输出树状结构的函数
void Table(int current_hierachy,std::string current_string, std::string* codes, char* characters, int
numbers,int side, std::ofstream &file1)
{
    for (int k = 0; k < current_hierachy; k++)
    {
        //根据当前所在的层级输出对应长度的/t 以得到凹表
        std::cout<<"    ";
        file1<<"    ";
    }
    for (int k = 0; k < numbers; k++)
    {
        if(current_string == codes[k])
        {
            //将当前叶子节点的字符串与 codes 中的字符串进行比较，如果相等则意味着
            //找到了对应的节点，输出
            std::cout<<characters[k]<<std::endl;
            file1<<characters[k]<<std::endl;
            return;
        }
    }
}

switch (side)
{
    //side 的 0 表示 left, 1 表示 right
    case (0):
        std::cout<<side;
        file1<<side;
        break;
    case (1):
        std::cout<<side;
        file1<<side;
        break;
    default:
        ;
}
std::cout<<std::endl;
file1<<std::endl;
//递归部分，每次递归层级+1，根据左右子树的不同对字符串加上 0 或 1
Table(current_hierachy + 1, current_string + '0',codes, characters, numbers, 0, file1);
Table(current_hierachy + 1, current_string + '1',codes, characters, numbers, 1, file1);
}

```

//T 函数，这个函数的核心是上面的 Table 函数，T 函数中主要是为 Table 函数做了一些准备工作

```
void Tree()
{
    std::ifstream file;
    file.open("hfmTree.txt");
    std::string encoder;
    getline(file, encoder);
    std::cout<<encoder;
    file.close();
    //get the contents

    int iter_string = 0, valid_length = 0;
    while(encoder[iter_string]>= '0' && encoder[iter_string] <= '9')
    {
        valid_length = encoder[iter_string] - '0' + 10 * valid_length;
        iter_string += 1;
    }
    std::cout<<"encoder"<<std::endl<<"length is :"<<valid_length<<std::endl;
    char *characters = new char[valid_length];
    std::string *codes = new std::string[valid_length];
    translation(encoder, characters, codes, iter_string);
    //以上部分为在之前列出的复用函数，用于提取得到字符-编码对
    std::ofstream file1;
    file1.open("treePrint.txt");
    int max_length = codes[valid_length - 1].length();
    int layer = 1;
    std::cout<<"\nroot";
    file1<<"\nroot";
    //调用 Table 函数
    Table(0, "", codes, characters, valid_length, 3, file1);
    file1.close();
}
```

## 2. 主函数部分

```
int main()
{
    while(true){
        char switches = 'Q';
        cout<<...(略去)
        cin>>switches;
        cin.clear();
        cin.sync();
        switch(switches)
        {
            case ('I'):
                initialize();
                break;
            case ('E'):
                Encoding();
                break;
            case ('D'):
                Decoding();
                break;
            case ('P'):
                Printing();
                break;
            case ('T'):
                Tree();
                break;
            case ('Q'):
                cout<<"Thanks for using, hope you have a nice day. :)";
                return 0;
            default:
                cout<<"Fatal error: Wrong input! Please rear the instruction carefully!"<<endl;
                return 1;
        }
    }
    return 0;
}
```

//主函数因为其他函数封装非常好的原因，非常简洁，可读性极强。



## 四、 调试分析

1. 在最开始涉及 I 函数和读取函数时，笔者忽略了一种情况：符号为 0 或 1。在这种情况下，如果在保存符号-编码对时不加注意的话，很有可能符号会被当成编码处理，导致 E、D 函数无法正常工作。在发现这一问题后，笔者通过加入空格的方式去掉了这一特殊情况。
2. Table 函数笔者在修改了近十版后确定为当前版本。这个函数的主要难点在于确定思路，不要犹豫。笔者最开始想通过树的遍历来实现，但苦于总体设计时“独立性”的要求，在 T 函数中难以重构 I 函数构造出的哈夫曼树，在尝试多次后遂放弃。在输出样式的选择上笔者也经历了多次犹豫，最终选择了更易懂也更容易实现的凹表来进行表示。
3. 本次大作业笔者最开始的设计过程中追求的“各函数之间完全独立”这一特点虽然好，但确实带来了许多麻烦。例如前述的复用函数第二段，假如能放弃函数间的独立性，那一段复用代码完全可以避免。
4. 很希望助教/老师可以专门分出来一节课教怎么调试，我感觉自己的调试方法（输出调试和逐步调试）在很多特定的情况下比如内存溢出时效率低的惊人...

### 算法复杂度分析

#### 1. 时间复杂度

(1) 在所有函数中，translation 和 Printing 两个函数都只涉及一次遍历，时间复杂度  $O(n)$

(2) 是  $O(n^2)$  的函数：

hfTree::hfTree	涉及到了两层遍历找最小值
Counting	两层遍历，去重+计数
initialize	调用了树的初始化等 $O(n^2)$ 函数
Encoding	两层遍历+查重
Decoding	和 Encoding 同理，两者原理相似

(3)  $O(n \log n)$ :

hfTree::getCode 遍历树+回溯树,由哈夫曼树高度  $O(\log n)$  可知  $O(n \log n)$

(4)  $O(2^n)$ :

Table	递归，对二叉树进行了遍历
Tree	其实它就是多了初始化的 Table 函数

#### 2. 空间复杂度

(1) hfTree::getCode, counting, translation 函数的空间复杂度为  $O(1)$

(2) hfTree::hfTree, initialize, Encoding, Encoding, Printing 这些函数都开辟了数组或新建了字符串(字符串看作  $n$  个字符组成)，复杂度为  $O(n)$

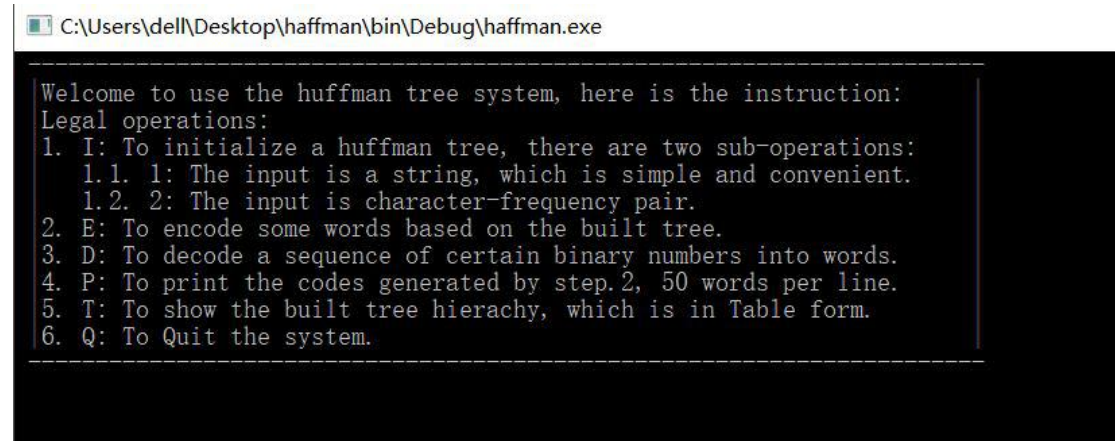
(3) 递归调用每次都要调用拷贝构造函数，因此空间复杂度与调用次数相关，而遍历二叉树共需要调用  $2^n$  次，所以空间复杂度为  $O(2^n)$

## 五、 用户手册

1. 本程序使用的 Code::Blocks 16.01 IDE，程序以项目（project）方式组织，如图 1 所示：



2. 依次点击菜单 “Build” -> “Build and run”，显示文本方式的用户界面，如图 2 所示：



之后只需按照需求输入自己想要进行的操作即可。

### NOTICE:

- (1) 本程序六个函数模块完全独立，所以可以任意时间、顺序调用以上六个函数。
  - (2) 程序各模块运行依赖文件夹中 txt 文件，请保证 txt 文件存在。
  - (3) 文件夹中 “test\_data.txt” 文件包含了 pdf 中列出的测试数据，win10 和 Ubuntu 系统下全选-复制-粘贴即可测试。
  - (4) 注意输入时的大小写和输入法是否切换。
3. 更详细的操作可阅读后面的样例测试。

## 六、 测试结果

以下的每一步都可以单独、随时进行，不需按照本测试顺序

```
C:\Users\dell\Desktop\haffman\bin\Debug\haffman.exe

Welcome to use the huffman tree system, here is the instruction:
Legal operations:
1. I: To initialize a huffman tree, there are two sub-operations:
    1.1. 1: The input is a string, which is simple and convenient.
    1.2. 2: The input is character-frequency pair.
2. E: To encode some words based on the built tree.
3. D: To decode a sequence of certain binary numbers into words.
4. P: To print the codes generated by step.2, 50 words per line.
5. T: To show the built tree hierachy, which is in Table form.
6. Q: To Quit the system.

I

There are two legal operations you can choose from:
1. type 1 : Input a sequence of words, which is simple and convenient.
2. type 2 : Input the characters-frequency pairs.
```

先输入 I，表示要初始化一棵树

因为想要通过频数来建立，再输入 2:

```
C:\Users\dell\Desktop\haffman\bin\Debug\haffman.exe

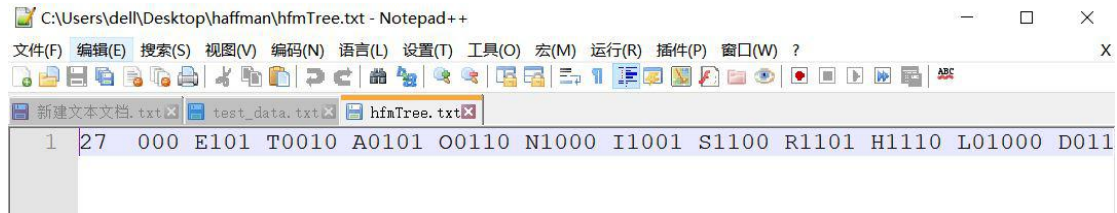
Welcome to use the huffman tree system, here is the instruction:
Legal operations:
1. I: To initialize a huffman tree, there are two sub-operations:
    1.1. 1: The input is a string, which is simple and convenient.
    1.2. 2: The input is character-frequency pair.
2. E: To encode some words based on the built tree.
3. D: To decode a sequence of certain binary numbers into words.
4. P: To print the codes generated by step.2, 50 words per line.
5. T: To show the built tree hierachy, which is in Table form.
6. Q: To Quit the system.

I

There are two legal operations you can choose from:
1. type 1 : Input a sequence of words, which is simple and convenient.
2. type 2 : Input the characters-frequency pairs.

2
Please input the number of characters :
27
Please input the characters-frequency pair :
The 1 character is :
```

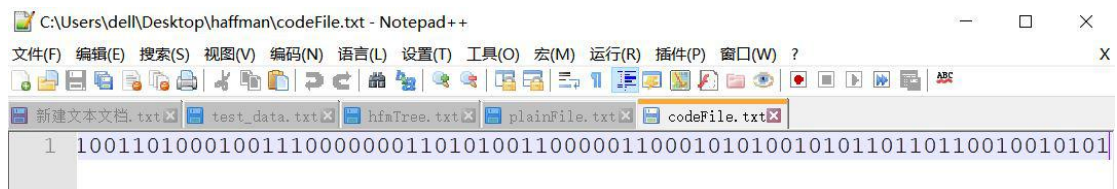
接下来依此输入字符和频数（建议直接复制粘贴 test\_data.txt，方便一些），回车后，本次操作结束。打开 hfmTree.txt:



文件很长没有截完，但可以看到已经根据设定得到了排序后的字符-编码对。

接下来在 **plainFile.txt** 中输入 **THIS PROGRAM IS MY FAVORITE**

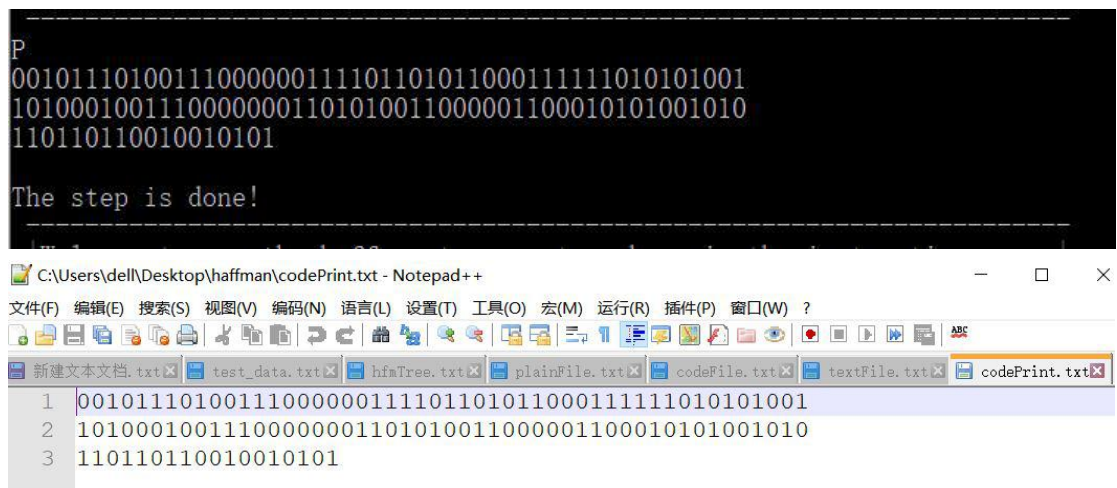
在命令行中输入 **E**，回车，打开 **codeFile.txt** 文件，可以看到上面那段话已经被编码了，文件很长没有截完：



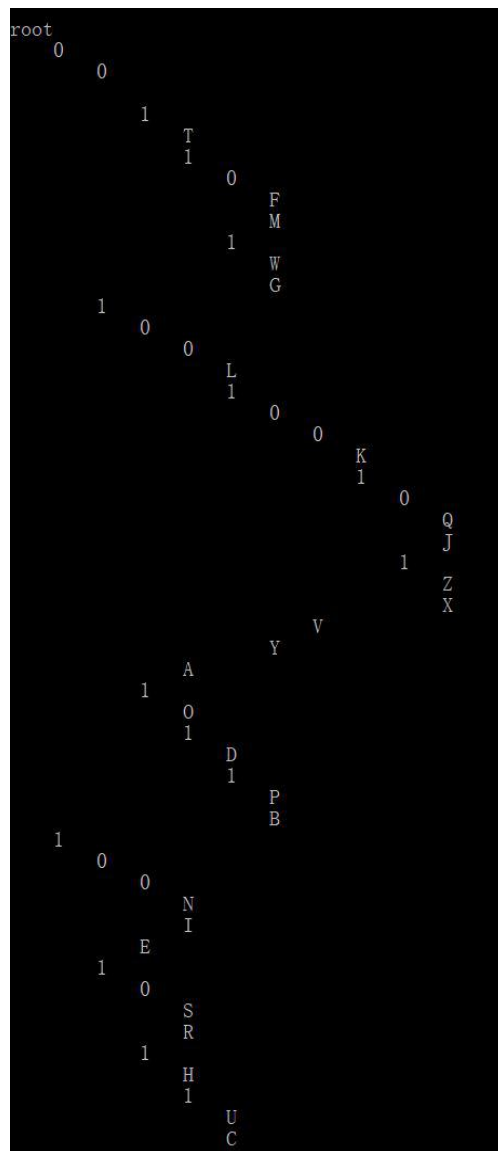
再输入 **D**，回车，打开 **textFile.txt**，可以看到上述编码又被解码了：



再输入 **P**，可以看到被分成 50 个每行的命令行和 **codeprint.txt**：

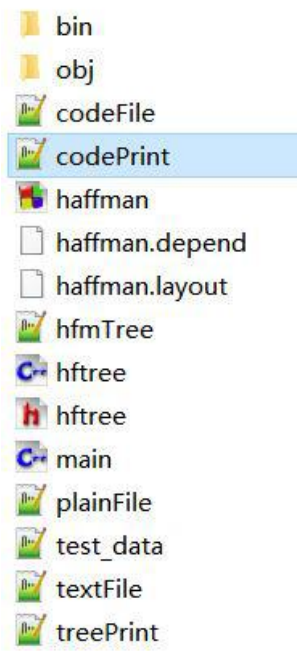


再输入 **T**，可以看到整个树的凹表模型：



最后可以输入 **q**，退出程序。

## 七、 附录



其中 test\_data 存放了测试数据