



华南理工大学

# 课程设计报告

课程名称：数据结构

学生姓名：雷皓兰

学生学号：202130320717

学生专业：数学与应用数学

开课学期：3

**数学系**  
**2023 年 3 月**

**数据结构课程设计报告**

——文件的哈夫曼编码与解码

**一. 语言:**

C++语言

**二. 算法思想:**

1. 将任意一个指定的文本文件中的字符进行哈夫曼编码，生成一个编码文件（压缩文件）和哈夫曼树文件；反过来，可将一个编码文件（压缩文件）解码还原为一个文本文件。
2. 利用哈夫曼树的特殊结构对文件进行操作，完成文件的编码、解码、显示等功能的实现。
3. 在对文件的操作过程中，先通过查找文件的储存路径来访问文件。在将文件的内容（即字符）进行编码，构造哈夫曼树，哈夫曼树的权重为每个字符分得的比特。用 num 来储存编码前后字符串的长度。
4. 功能模块图:

E----Encode a file
D----Decode a file
L----List Text file
Q----Quit

选择 E 时:

输入一个待编码的文本文件名称（可带路径）。如: D:\lu\lu.txt

打开该文件，统计文本文件中各字符的个数作为权值，生成哈夫曼树；  
将哈夫曼树存入磁盘文件中。

将文本文件利用哈夫曼树进行编码，生成压缩文件。

压缩文件名称=文本文件名.ecd      如: D:\lu\lu.ecd

哈夫曼树文件名称=文本文件名.hfm    如: D:\lu\lu.hfm

选择 D 时:

输入一个待解码的编码文件名称（可带路径）。如: D:\lu\lu.ecd      从  
对应的哈夫曼文件（如 D:\lu\lu.hfm）中读出哈夫曼树。

利用哈夫曼树将编码文件中的编码序列解码；生成（还原）文本文件。

文件名称=编码文件名+ “\_new.txt”      如: D:\lu\lu\_new.txt

选择 L 时:

输入一个要显示的文本文件名称（可带路径）。如: D:\lu\lu\_new.txt

显示出该文本文件的内容。

### 三. 实现过程:

1. 初始化字符、出现的次数、分配的比特, 用 weight 来记录各个字符的权重和在哈夫曼树中这个节点被分配到的比特

```
typedef struct char_count* count_node;
struct char_count {
    char ch; // 字符
    long count; // 出现的次数
    short bit; // 分配的比特
};
count_node weight[126];
```

2. 构造哈夫曼树的节点、双亲节点、左孩子、右孩子, 创建一个 p 指针用来记录哈夫曼树的树根, 同时也是工作指针。

```
struct HNodeType {
    count_node node;
    HNode parent;
    HNode lchild;
    HNode rchild;
};
HNode p;
```

3. 初始化编码以及编码的长度, 用 code 记录各字符编码的信息。
4. 由于输入不同字符的个数, num 会改变, 所以用 n 来保存 num 的值。初始化编码前字符串的长度、编码前的字符串、编码后的比特。

```
struct charCode {
    char ch;
    short codeNum[1000];
    int codeLen;
};
code char_code[125];

int n, num;
int sumStrLen = 0;
char str[100000];
short codeStr[100000000];
```

5. 译码 tranCode(int len):
  - a. 输入文件名称 (可带路径), 以读模式打开文件
  - b. 打开文件, 进入循环, 构建哈夫曼树。
  - c. 若字符的比特为 0, 则进入左子树添加字符; 若字符的比特为 1, 则进入右子树添加字符。

```

using namespace std;
string name;
void get_file() {

    cout << "请输入文件名称（可带路径）：";
    cin >> name;
    ifstream fin(name+".txt");
    fin >> str;
}

void tranCode(int len) {
    ofstream ofile(name+"_new.txt", std::ios::app);
    HNode q;
    q = p;
    ofile << "\n译码结果：" << endl;
    for (int i = 0; i < len; i++) {
        if (codeStr[i] == 0) {
            q = q->lchild; // 进入左子树
            if (q->node->ch) {
                ofile << q->node->ch;
                q = p;
            }
        }
        if (codeStr[i] == 1) {
            q = q->rchild; // 进入右子树
            if (q->node->ch) {
                ofile << q->node->ch;
                q = p;
            }
        }
    }
    ofile << endl;
}

```

6. 计算压缩效率 efficiency(int len):

打开文件，计算编码后字符的长度 sumCodeLen 等于个字符编码长度 codeLen 的总和。计算压缩效率为 sumCodeLen\*1.0/sumStrLen。

```

void efficiency(int len) {

    ofstream ofile(name+".hfm", std::ios::app);
    int sumCodeLen = 0;
    for (int i = 0; i <= len; i++) {
        sumCodeLen += char_code[i]->codeLen;
    }
    ofile << "\n压缩效率：" << sumCodeLen << "/" << sumStrLen << "=" << sumCodeLen * 1.0 / sumStrLen << endl;
}

```

7. 把字符串编码，返回编码后数组的长度 finishCode():

以输入的方式打开文件，遍历文件，数组的长度等于字符串的总长度。

```

int finishCode() {
    ofstream ofile(name + ".ecd", std::ios::app);
    int m = 0;
    ofile << "\n编码结果: " << endl;
    for (int i = 0; i < sumStrLen; i++) {
        for (int j = 0; j < sumStrLen; j++) {
            if (char_code[j]->ch == str[i]) {
                for (int k = 0; k < char_code[j]->codeLen; k++) {
                    codeStr[m] = char_code[j]->codeNum[k];
                    ofile << codeStr[m];
                    m++;
                }
                break;
            }
        }
    }
    ofile << endl;
    return m;
}

```

8. 获取各个字符的编码 getcCode() :

- a. 左子叶：打开文件，初始化，遍历所有左子叶，若左孩子不为空进入循环，新建字符为 q1 左孩子节点的字符，设 j 为 0、cache 为左孩子节点的比特，j++，q2=q1。若 q2 节点的比特不为 0，则 cache 为 q2 节点的比特，j++，q2 指向它的双亲节点。M 置为 0，K 从 j-1 开始循环，字符的编码等于 cache[k]，m++。最终编码的长度等于 m。若 q1 的右孩子不为空，则 q1 指向 q1 点右孩子。
- b. 右子叶：若右孩子不为空进入循环，新建字符为 q1 右孩子节点的字符，设 j 为 0、cache 为右孩子节点的比特，j++，q2=q1。若 q2 节点的比特不为 0，则 cache 为 q2 节点的比特，j++，q2 指向它的双亲节点。M 置为 0，K 从 j-1 开始循环，字符的编码等于 cache[k]，m++。最终编码的长度等于 m。

```

void getCode() {
    ofstream ofile(name+".hfm", std::ios::app);
    // 初始化
    for (int i = 0; i < 125; i++) {
        char_code[i] = new charCode();
        for (int j = 0; j < 1000; j++) {
            char_code[i]->codeNum[j] = -1;
        }
    }
}

```

```

int cache[1000] = { 0 }, i = 0, j = 0;
HNode q1, q2;
q1 = p;
// 所有左叶子
while (q1->lchild) {
    char_code[i]->ch = q1->lchild->node->ch;
    j = 0;
    cache[j] = q1->lchild->node->bit;
    j++;
    q2 = q1;
    while (q2->node->bit) {
        cache[j] = q2->node->bit;
        j++;
        q2 = q2->parent;
    }
    int m = 0;
    for (int k = j - 1; k >= 0; k--) {
        char_code[i]->codeNum[m] = cache[k];
        m++;
    }
    char_code[i]->codeLen = m;
    i++;
    if (q1->rchild) {
        q1 = q1->rchild;
    }
    else {
        break;
    }
}

// 右叶子
q1 = p;
if (q1->rchild) {
    j = 0;
    int cache2[1000] = { 0 };
    while (q1->rchild) {
        cache[j] = q1->rchild->node->bit;
        j++;
        if (q1->rchild->node->ch) {
            char_code[i]->ch = q1->rchild->node->ch;
        }
        q1 = q1->rchild;
    }
    int m = 0;
    for (int k = j - 1; k >= 0; k--) {
        char_code[i]->codeNum[m] = cache[k];
        m++;
    }
    char_code[i]->codeLen = m;
}

// output
ofile << "\n得出各字符的编码: " << endl;
for (int k = 0; k <= i; k++) {
    if (char_code[k]->ch) {
        ofile << "char: " << char_code[k]->ch << " code: ";
    }
}

```

9. 输出：打开文件，将得到的各字符的编码写入文件。

```

ofile << "\n得出各字符的编码：" << endl;
for (int k = 0; k <= i; k++) {
    if (char_code[k]->ch) {
        ofile << "char: " << char_code[k]->ch << " code: ";
        int k2 = 0;
        while (char_code[k]->codeNum[k2] != -1) {
            ofile << char_code[k]->codeNum[k2];
            k2++;
        }
        ofile << endl;
    }
}

// 把字符串进行编码
int codeLen = finishCode();

// 统计压缩效率
efficiency(i);

```

10. 把字符串进行编码。

11. 统计压缩效率 efficiency()

12. 分配比特 coder(HNode p):

若 p 不为空，若 p 的左孩子不为空，则左孩子节点的比特为 0；若右孩子不为空，则右孩子 比特为 1 若 p 的左孩子不为空，则 p 指向它 左孩子；若 p 的右孩子不为空，则它指向它的右孩子。

```

void coder(HNode p) {
    if (p) {
        if (p->lchild) {
            p->lchild->node->bit = 0;
        }
        if (p->rchild) {
            p->rchild->node->bit = 1;
        }
        if (p->lchild) {
            coder(p->lchild);
        }
        if (p->rchild) {
            coder(p->rchild);
        }
    }
}

```

13. 前序遍历哈夫曼树 outputHafftree(HNode p):

若节点的字符不为空，则显示节点的字符和权重，否则显示[新增节点]的权重。若左孩子不为空，递归左孩子；若右孩子不为空，递归右孩子。关闭文件。

```

void outputHafftree(HNode p) {
    ofstream ofile(name + ".hfm", std::ios::app);
    ofile << "\n前序遍历哈夫曼树:" << endl;
    if (p) {
        // root
        if (p->node->ch != NULL) {
            ofile << "字符:" << p->node->ch << " 权重:" << p->node->count << endl;
        }
        else if (p->node->count) {
            ofile << "[新增节点] 权重:" << p->node->count << endl;
        }
        // left
        if (p->lchild) {
            outputHafftree(p->lchild);
        }
        // right
        if (p->rchild) {
            outputHafftree(p->rchild);
        }
    }
    ofile.close();
}

```

#### 14. 构建哈夫曼树 HaffmanTree():

将该节点（将成为左孩子）的左右子树置为空，该节点（将成为左孩子）的核心节点置为 weight 中未使用的最小 count 的节点。

- a. 只有一个不同字符的情况：右孩子置 NULL，左孩子和右孩子的父亲置为一个新的节点，并使工作指针 p 指向它。新节点的左孩子置为旧的这个左孩子，新节点的右孩子置为 NULL，给新父亲分配空间，核心节点的 ch 置为 NULL。父亲权重为两个孩子权重的和，置空哈夫曼树根节点中父亲节点。
- b. 剩余情况：该节点（将成为左孩子）的核心节点置为 weight 中未使用的最小 count 的节点。左孩子和右孩子的父亲置为一个新的节点，并使工作指针 p 指向它。新节点的左孩子置为旧的这个左孩子，新节点的右孩子置为旧的这个右孩子，给新父亲分配空间，核心节点的 ch 置为 NULL。父亲权重为两个孩子权重的和。给旧的左孩子（将再次成为一个左孩子）重新分配空间，旧的左孩子（将再次成为一个左孩子）的核心节点置为 weight 中未使用的最小 count 的节点，左孩子（将再次成为一个左孩子）的左右树置为 NULL，旧的右孩子成为父亲。



```

void HaffmanTree() {
    HNode hNl, hNr;
    hNl = new HNodeType();
    hNr = new HNodeType();
    hNl->lchild = hNl->rchild = NULL; // 该节点(将成为左孩子)的左右子树置为NULL
    hNl->node = weight[1]; // 该节点(将成为左孩子)的核心节点置为weight中未使用的最小count的节点

    hNr->lchild = hNr->rchild = NULL; // 该节点(将成为右孩子)的左右子树置为NULL

    if (n == 1) { // 只有一个不同字符的情况
        hNr->node = NULL; // 没有右孩子, 右孩子置NULL

        p = hNl->parent = new HNodeType(); // 左孩子和右孩子的父亲置为一个新的节点, 并使工作指针p指向它

        p->lchild = hNl; // 新节点的左孩子置为旧的这个左孩子
        p->rchild = NULL; // 新节点的右孩子置为NULL
        p->node = new char_count(); // 给新父亲分配空间
        p->node->ch = NULL; // 新增的父亲的核心节点的ch置NULL
        p->node->count = p->lchild->node->count; // 父亲权重为两个孩子权重的和
        p->parent = NULL; // 置空哈夫曼树根节点中的父亲节点
    }
    else {
        hNr->node = weight[2]; // 该节点(将成为右孩子)的核心节点置为weight中未使用的最小count的节点

        for (int i = 3; i <= n; i++) {
            p = hNl->parent = hNr->parent = new HNodeType(); // 左孩子和右孩子的父亲置为一个新的节点, 并使工作指针p指向它

            p->lchild = hNl; // 新节点的左孩子置为旧的这个左孩子
            p->rchild = hNr; // 新节点的右孩子置为旧的这个右孩子
            p->node = new char_count(); // 给新父亲分配空间
            p->node->ch = NULL; // 新增的父亲的核心节点的ch置NULL
            p->node->count = p->lchild->node->count + p->rchild->node->count; // 父亲权重为两个孩子权重的和

            hNl = new HNodeType(); // 给旧的左孩子(将再次成为一个左孩子)重新分配空间
            hNl->node = weight[i]; // 旧的左孩子(将再次成为一个左孩子)的核心节点置为weight中未使用的最小count的节点
            hNl->lchild = hNl->rchild = NULL; // 旧的左孩子(将再次成为一个左孩子)的左右子树置为NULL

            hNr = p; // 旧的右孩子成为父亲
        }
    }
}

```

- c. 生成哈夫曼树的根节点：左孩子和右孩子的父亲置为一个新的节点，并使工作指针 p 指向它。新节点的左孩子置为旧的这个左孩子，新节点的右孩子置为旧的这个右孩子，给新父亲分配空间，核心节点的 ch 置为 NULL。父亲权重为两个孩子权重的和。置空哈夫曼树根节点中的父亲节点。

15. 打印 outputHafftree(p)。

16. 分配比特 coder(p)。

```

p = hNl->parent = hNr->parent = new HNodeType(); // 左孩子和右孩子的父亲置为一个新的节点, 并使工作指针p指向它

p->lchild = hNl; // 新节点的左孩子置为旧的这个左孩子
p->rchild = hNr; // 新节点的右孩子置为旧的这个右孩子
p->node = new char_count(); // 给新父亲分配空间
p->node->ch = NULL; // 新增的父亲的核心节点的ch置NULL
p->node->count = p->lchild->node->count + p->rchild->node->count; // 父亲权重为两个孩子权重的和
p->parent = NULL; // 置空哈夫曼树根节点中的父亲节点
}
// 打印

outputHafftree(p);

//分配比特
coder(p);
}

```

17. 交换堆中两个元素 swap(int x, int y), x, y 为需要交换位置的元素之一的下标。

18. 向下调整 sifttdown(int i): i 为需要向下调整的元素的下标。若  $i*2 \leq \text{num}$  且  $\text{flag} == 0$ , 若 i 的权重小于  $i*2$  的权重, 则  $t = i*2$ , 否则  $t = i$ 。若  $i*2+1 \leq \text{num}$ , 若 t 的权重小于  $i*2+1$  的权重, 则  $t = i*2+1$ ; 若 t 不等于 i, 则交换 t 和 i,  $i = t$ ; 否则  $\text{flag} = 1$ 。

```

void swap(int x, int y) {
    count_node t;
    t = weight[x];
    weight[x] = weight[y];
    weight[y] = t;
}

void siftDown(int i) {
    int t, flag = 0;
    while (i * 2 <= num && flag == 0) {
        if (weight[i]->count < weight[i * 2]->count) t = i * 2;
        else t = i;
        if (i * 2 + 1 <= num)
            if (weight[t]->count < weight[i * 2 + 1]->count) t = i * 2 + 1;
        if (t != i) {
            swap(t, i);
            i = t;
        }
        else flag = 1;
    }
}

```

19. 建立堆 creat()。

20. 堆排序 heapsort()。

```

void creat() {
    for (int i = num / 2; i >= 1; i--) siftDown(i);
}

void heapsort() {
    while (num > 1) {
        swap(1, num);
        num--;
        siftDown(1);
    }
}

```

21. 统计从文件当中获取的字符串中每个字符出现的次数

get\_str\_weight():

遍历文件的字符，计算各个字符出现的次数。

```

void get_str_weight1() {
    ofstream ofile(name + ".hfm", std::ios::app);
    int i = 0;
    long count[126] = { 0 };
    while (str[i] != '\0') {
        count[str[i] - 32]++;
        sumStrLen++;
        i++;
    }
    sumStrLen *= 8;
    int j = 1;
    for (int i = 1; i < 126; i++) {
        if (count[i] > 0) {
            weight[j]->ch = i + 32;
            weight[j]->count = count[i];
            ofile << weight[j]->ch << " : 出现 " << weight[j]->count << " 次" << endl;
            j++;
            num++;
        }
    }
    n = num;
}

```

22. 获取文件中所有字符，执行任务 1 task1(): 获取文件 get\_file(), 获取权重 get\_str\_weight1(), 建立堆 creat(), 堆排序 (建立最小堆, 权重小的先出) heapsort(), 输出权重, 构建哈夫曼树 HaffmanTree(), 输出“编码成功!”。

```
void task1() {
    get_file();
    get_str_weight1();
    creat();
    heapsort();
    ofstream ofile(name + ".hfm", std::ios::app);
    ofile << "\n建立最小堆 (权重小的先出): " << endl;
    for (int i = 1; i <= n; i++) {
        ofile << weight[i]->ch << " 权重: " << weight[i]->count << endl;
    }
    HaffmanTree();
    getCode();
    cout << "编码成功!" << endl;
}
```

23. 主函数，设计主菜单：


E (Encode a file): 执行任务 1。

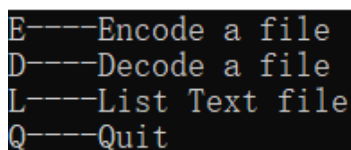
D (Decode a file): 执行 finishCode(), tranCode(), “解码成功”

L (List Text file): 执行 fin(path), getline(fin, a), 输出 a。

```
int main() {
    for (int i = 0; i < 126; i++) {
        weight[i] = new char_count();
    }
    char choose;
    cout << "E----Encode a file\nD----Decode a file\nL----List Text file\nQ----Quit" << endl;
    while (cin >> choose) {
        if (choose == 'E') { task1(); }
        if (choose == 'D') { int codeLen = finishCode(); tranCode(codeLen); cout << "解码成功!" << endl; }
        if (choose == 'L') {
            char path1[1000];
            cout << "请输入文件路径: ";
            cin >> path1;
            ifstream fin(path1);
            string a;
            while (getline(fin, a)) { cout << a << endl; };
        }
        if (choose == 'Q') break;
        cout << "E----Encode a file\nD----Decode a file\nL----List Text file\nQ----Quit" << endl;
    }
}
```

#### 四. 实现结果：

 D:\ConsoleApplication1\x64



```
E----Encode a file
D----Decode a file
L----List Text file
Q----Quit
```

为方便演示先选择 L 列出文件内容：

```
L
请输入文件路径: D:/23.txt
jfjdskljfls
```

选择 E 时:


输入一个待编码的文本文件名称 (可带路径)。如: D:\23

```
E----Encode a file
D----Decode a file
L----List Text file
Q----Quit
E
请输入文件名称 (可带路径): D:/23
编码成功!
```

生成压缩文件。

 23.ecd	2023/3/11 21:25	ECD 文件
--	-----------------	--------

文件内容如下:

 23.ecd - 记事本


文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
|
编码结果:
0100111101110111111100101101110
```

同时生成哈夫曼树文件:

 23.hfm	2023/3/11 21:25	HFM 文件
--	-----------------	--------

文件内容:

 23.hfm - 记事本

文件(F) 编辑(E) 格式(O)

```
d: 出现 1 次
f: 出现 2 次
j: 出现 3 次
k: 出现 1 次
l: 出现 2 次
s: 出现 2 次
```

建立最小堆 (权重小的先出):

```
d 权重: 1
k 权重: 1
s 权重: 2
l 权重: 2
f 权重: 2
j 权重: 3
```

得出各字符的编码:

```
char: j code: 0
char: f code: 10
char: l code: 110
char: s code: 1110
char: d code: 11110
char: k code: 11111
```

压缩效率:  $20/88=0.227273$

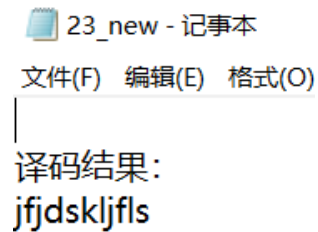
并在编码成功后显示编码成功。

选择 D 时:

将原来压缩的文件进行解码生成新文件:

 23_new	2023/3/11 21:35	文本文档
--	-----------------	------

内容不变:



选择 Q 时，系统退出

```
E----Encode a file
D----Decode a file
L----List Text file
Q----Quit
Q
D:\ConsoleApplication1\x64\Debug\ConsoleApplication1.exe (进程 18060)已退出，代码为 0。
按任意键关闭此窗口. . .
```

## 五. 小组分工：

我主要负责的任务是：

参与选题讨论，查找哈夫曼编码相关资料。

主要参与搭建哈夫曼树基本框架的搭建、结构体的声明、根据文本内容生成哈夫曼树节点等和计算压缩效率等。

## 六. 心得体会：

哈夫曼编码是我们上课时已经接触过的知识点，了解到是树在实际生活中的应用之一。加之我们也比较感兴趣，便选择了这个题目做课程设计。虽然一开始时对文件的哈夫曼编码和解码有了清晰的思路 and 想法，但是在完成程序设计的过程中我们仍然遇到了不少问题，幸运的是几乎都找到了解决方案。仍然存在的问题是译码，没有设置读取压缩文件和哈夫曼树文件的步骤，以致于生成新文件时只能生成上一个压缩的文件而无法任意选择压缩文件生成。

总而言之，通过这次课程设计，我更深刻地认识到了小组合作的重要性，不仅融会贯通了课本上的知识，还学会了利用网上的资料，学到了许多，也收获了学习编码的好的方法和技巧，很有成就感。并且认识到如果找到了正确的方法，很多事情就能够事半功倍。这次机会，也为我们今后的编程学习提供了动力。非常感谢老师和合作的同学！