华南师范大学本科生实验报告

姓名 邓智超　　学号 20192121026

院系 计算机学院 专业 计算机科学与技术（师范）

年级 19 级　　班级 1 班

实验时间　2020 年 12 月 25 日

实验名称　综合应用

课程名称 数据结构实验

华南师范大学教务处编印

华南师范大学本科生实验报告

实验课程：数据结构实验课

实验名称：哈夫曼树

一、实验内容

- **问题描述**

利用哈夫曼编码进行通信可以大大提高利用率，缩短信息传输时间，降低传输成本。但是，这就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（复原），对于双工通信（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写一个哈夫曼的编/译码系统。

- **必做内容**

◆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。

◆编码（Encoding）。利用已建好的哈夫曼树（如不在内存，则从 hfmTree 中读入），对文件 ToBeTran 中的正文进行编码，然后将结果存入文件 CodeFile 中。

◆译码（Decoding）。利用已建好的哈夫曼树将文件 CodeFile 中的代码进行译码，结果存入文件 TextFile 中。

◆打印代码文件（Print）。将文件 CodeFile 以紧凑格式显示在屏幕上，每行 50 个二进制代码。同时将此字符形式的编码文件写入 CodePrin 中。

◆打印哈夫曼树(TreePrinting)。将已在内存中的哈夫曼树以直观的方式(树或凹入形式）显示在屏幕上，同时将此字符形式的哈夫曼树写入文件 TreePrin 中。

◆其他辅助功能：实现各个转换操作的源/目标文件，均由用户在选择此操作时指定。

选做内容

用 MFC 的单文档窗口和菜单设计界面。

- **已完成内容**

必做部分：

✓ 在 Visual Studio 平台使用 c++语言完成系统的所有功能，并且对用户交互界面做了一定的优化，使得控制台界面看起来更加简洁舒适。所有功能

可正常使用且具有一定的健壮性。

- ✓ 根据文档给出的字符集和实际统计数据建立好了哈夫曼树，并实现了“THIS PROGRAM IS MY FAVORITE”的编码和译码。
- ✓ 实现了对我完成的系统的源程序进行编码和译码。（因为我写的代码含有中文字符，所以中文字符部分没有被编码和译码）
- ✓ 搜索了 10 篇英文文章并统计了出现频率，并将他们进行编码和译码

二、实验目的

- 加深对哈夫曼树实现算法的理解。
- 加深对二叉树遍历算法的理解。
- 加深对数据结构中链表、顺序表、堆、文件等基本操作以及实现算法的理解，以便在解决实际问题中灵活运用它们。
- 加深对各种课程各种基本概念、基本结构和基本操作的理解。

三、实验文档：

必做部分：

说明

通过检查 CodeFile 文件容量来进行分析，为提高压缩效率，我前前后后一共写了四个版本的系统。

version1： 对 01 编码的字符串不做处理直接存入编码文件，能够正确编码，解码，但是压缩后的文件大小反而比压缩前更大。如 17kb 的未压缩文件压缩后变成了 20kb 以上。

version2：在 version1 基础上，对 01 编码的字符串转成 10 进制数存储，因为刚开始做的时候不太搞清楚整个过程，为了方便看见读写结果，所以在读写文件的时候采用了 ASCLL 的读写方式，总的来看，压缩、解压结果是正确的，压缩率比 version1 有所提升，但是因为 ASCLL 读写方式需要写入大量的空格，所以压缩后文件大小还是比压缩前略微更大。

version3:在 version2 的基础上,将 01 编码的字符串转成了十进制数字存储，且采用二进制文件读写方式。进一步提升了压缩率，压缩后文件大小比压缩前有所减小，但是因为使用了 short 的整形数据，将原先 8 个二进制串当成 2 字节写入文件，导致压缩效率没有明显提升，但相比 version2，version3 已经实现了压缩后文件小于压缩前文件。

version4：考虑到 8 位二进制码最大表示数为 255，所以将 version3 中写入压缩文件的数据类型改成了 1 字节的 unsigned char，以及对其他细节进行了优化相比与两个字节的 short，这极大的提升了压缩率，对于 6000 字的英文文章，压缩前的文件大小为 34k，压缩后文件大小为 17k，压缩率提升效果显著。

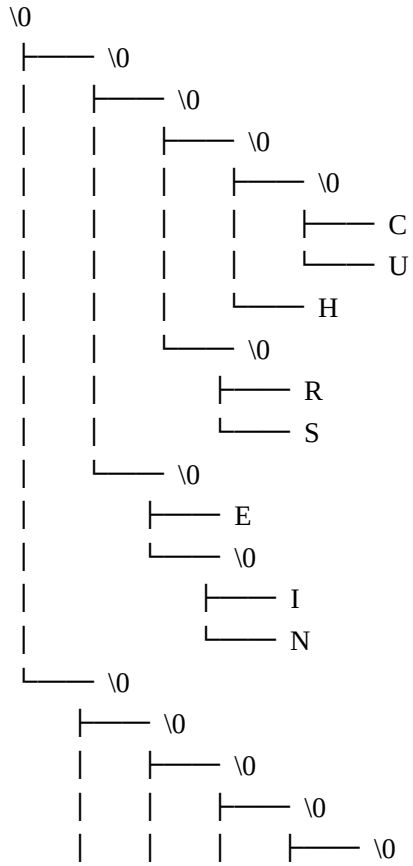
本实验报告将按照 version4 来撰写。

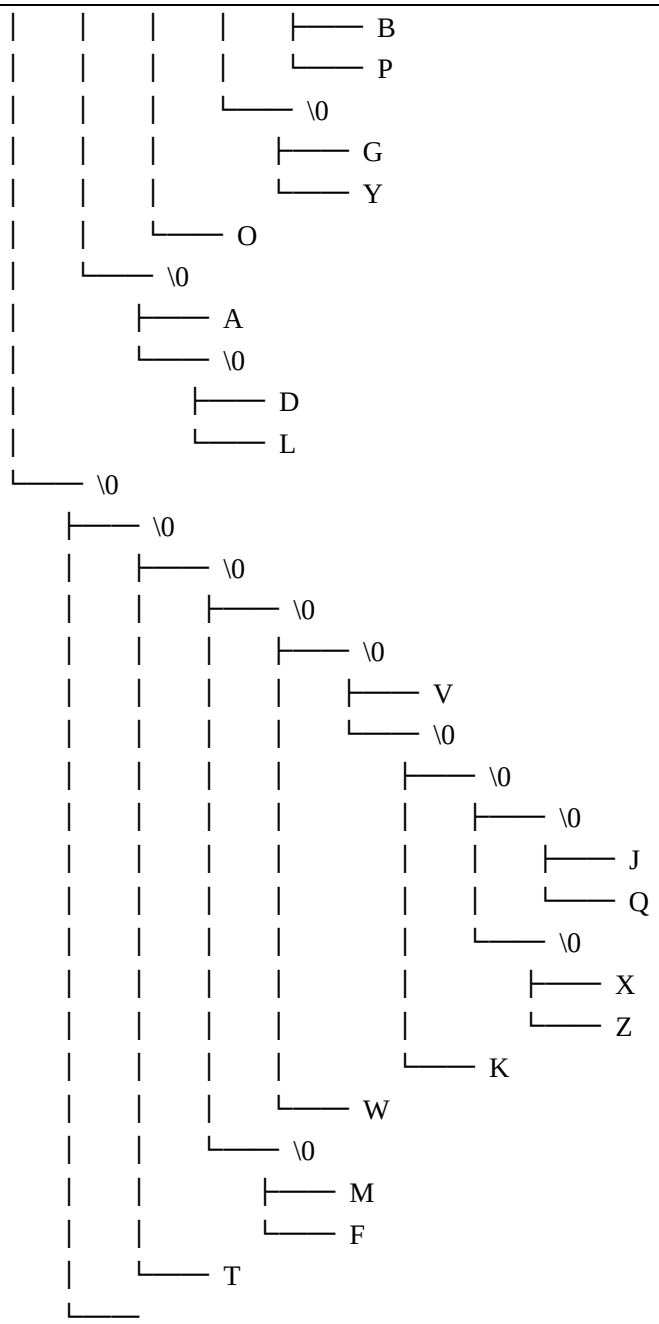
数据测试结果

- 根据文档给出的字符集和实际统计数据建立好了哈夫曼树，并实现了“THIS PROGRAM IS MY FAVORITE”的编码和译码。

字符		A	B	C	D	E	F	G	H	I	J	K	L	M
频度	186	64	13	22	32	103	21	15	47	57	1	5	32	20
字符	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	
频度	57	63	15	1	48	51	80	23	8	18	1	16	1	

✓ 根据以上字符集和频度，产生以下形状的哈夫曼树（非叶子结点的关键字默认输出\0）：





✓ 按照生成的哈夫曼树，”THIS PROGRAM IS MY FAVORITE ”产生的二进制编码如下：（相关的文件存储在 test 相关文件中）
01101000001100010111100001001010011000100100101000
11101100011111010010100011111100011101011000000001
00100110110100
实际上在压缩文件中是按照十进制存储的

- 实现了对我完成的系统的源程序进行编码和译码。

✓ 统计结果如下,共出现 81 种字符：

空格	!	"	#	%	&	'	(	)	*
1377	27	138	15	10	59	46	321	321	63
+	,	-	.	/	0	1	2	3	4
74	107	115	158	853	104	66	30	7	4
5	6	7	8	:	;	<	=	>	?
11	6	2	19	73	432	265	212	94	6
A	C	D	E	F	H	I	L	M	N
12	63	13	14	13	130	7	33	18	98
O	P	R	S	T	U	W	Y	[	\
4	21	8	31	228	18	18	2	95	19
]	_	a	b	c	d	e	f	g	h
95	19	415	49	294	323	1237	431	86	159
i	j	k	l	m	n	o	p	r	s
667	19	51	273	443	586	542	207	668	325
t	u	v	w	x	y	z	{	|	}
690	394	81	69	35	247	63	113	6	113
~									
2									

✓ 相关的编码，译码文件已经存储在 myProgma 相关文件夹中

- 搜索了 10 篇英文文章并统计了出现频率，并将他们进行编码和译码。相关的编码译码文件都在作业包中，以下列出文件内容变化。

	原文件大小	压缩后大小	二进制串大小	解压后大小
第 1 篇	35kb	20kb	161kb	35kb
第 2 篇	44kb	25kb	200kb	43kb
第 3 篇	55kb	31kb	252kb	55kb
第 4 篇	38kb	21kb	172kb	38kb
第 5 篇	38kb	21kb	174kb	37kb
第 6 篇	56kb	32kb	262kb	56kb
第 7 篇	80kb	44kb	366kb	79kb
第 8 篇	71kb	39kb	322kb	71kb
第 9 篇	87kb	49kb	403kb	87kb
第 10 篇	46kb	26kb	212kb	46kb

可见，按照我的编码方式，压缩率可以达到 50%左右，若是直接用二进制串存储压缩文件，压缩后文件不但没有变小，反而变大不少。

实现思路

主要的实现思路是首先接受终端输入的字符集和频度，分别用 keys[]和 times[]

数组存储后，建立哈夫曼树，获得每个字符对应的编码，将编码按顺序存储在字符串数组 codes[]中。

进行编码时，按照字符的输入，先将代编码的文章转换成 01 二进制编码存储在 binaryCode[]数组中，然后根据哈夫曼树的 wpl，从 binaryCode[]取 8 位 01 码（需要考虑最后是否有 8 位的问题，根据 wpl 能否被 8 整除即可）转换成十进制写入编码文件。需要提一下的是，8 位二进制码最大表示数字是 255，完全可以用 unsigned char 类型的变量来存储，unsigned char 只占用 1 字节，比 int 的 4 字节能节省很多空间，提升压缩效率。

解码的时候，从编码文件读取十进制数字，并将十进制数字转成二进制的 01 字符串存储在 binaryCode[]数组中，然后对整个 01 字符串，从哈夫曼树的根开始走，遇见 0 往左走，遇见 1 往右走。当走到叶子结点，酒吧叶子结点输出，之后从树根重新走即可。

全局变量声明

```
//声明全局类对象
//声明最小堆的最大结点树，此处设置为130
const int DefaultSize =130;
```

最小堆 MyHeap 的声明和实现

使用类模板，其数据成员包括指向 T 类型数组的指针 heap，currentSize(当前元素数量)，maxHeapSize(最大元素数量)，其中 heap 是 T*指针，currentSize 和 maxHeapSize 是 int 变量。

方法包括缺省构造函数和参数构造函数 Myheap()，析构函数~MyHeap()，插入函数 Insert，删除函数 Remove()，判断是否为空函数 IsEmpty(),判断堆是否位满函数 IsFull()，清空堆函数 MakeEmpty()，向下调整算法 siftDown()，向上调整算法 siftUp()，因为最小堆是课程中详细讲解过的，实现方法较为简单，此处不做详细叙述。以下直接写出实现代码。

```
//////////////////////////////
//接下来是最小堆myHeap类的定义
//////////////////////////////

template<class T>
class MyHeap
{
```

```
        //heap数组
        T* heap;
        //当前元素数量
        int currentSize;
        //最大元素数量
        int maxHeapSize;
    public:
        //缺省构造函数
        MyHeap(int sz = DefaultSize);
        //由数组生成最小堆
        MyHeap(T arr[], int n);
        //析构函数
        ~MyHeap() { delete[]heap; }
        //最小堆插入算法
        bool Insert(T& x);
        //最小堆删除算法
        bool Remove(T& x);
        //判断堆是否为空
        bool IsEmpty()const
        {
            return currentSize == 0;
        }
        //判断堆是否已满
        bool IsFull()const
        {
            return currentSize == maxHeapSize;
        }
        //清空堆
        void MakeEmpty()
        {
            currentSize = 0;
        }
    private:
        //最小堆向下调整算法
        void siftDown(int start, int m);
        //最小堆向上调整算法
        void siftUp(int start);
    };
```

缺省构造函数

函数原型:

```
    MyHeap<T>::MyHeap(int sz)
```

代码：

```
//构造函数，根据给定大小maxSize，建立堆对象
//函数功能：创建堆数组，执行初始化操作
//函数参数：数组大小sz
//返回值：无
template<class T>
MyHeap<T>::MyHeap(int sz)
{
    maxHeapSize = (DefaultSize < sz) ? sz : DefaultSize;
    heap = new T[maxHeapSize];//创建堆数组
    currentSize = 0;
}
```

参数构造函数

函数原型：

```
template <class T>
MyHeap<T>::MyHeap(T arr[], int n)
```

代码：

```
//函数参数：包含字符出现次数的数组arr，数组长度n
//返回值：无
template <class T>
MyHeap<T>::MyHeap(T arr[], int n)
{
    maxHeapSize = (DefaultSize < n) ? n : DefaultSize;
    heap = new T[maxHeapSize];
    //对数组赋值
    for (int i = 0; i < n; i++)
    {
        heap[i] = arr[i];
    }
    //对数组进行调整，使之成为最小堆
    currentSize = n;
    int currentPos = (currentSize - 2) / 2;//最后非叶子结点
    while (currentPos >= 0)
    {
        //从currentPos调整到currentSize
        siftDown(currentPos, currentSize - 1);
        currentPos--;
    }
```

```
}
```

最小堆插入算法

函数原型：

```
template <class T>
bool MyHeap<T>::Insert(T& x)
```

代码：

```
//函数功能：最小堆插入算法，将元素x插入最小堆
//函数参数：要插入堆的元素x
//返回值：逻辑值true或false
template <class T>
bool MyHeap<T>::Insert(T& x)
{
    //若堆已满
    if (currentSize == maxHeapSize)
    {
        cout << "堆已满！" << endl;
        return false;
    }
    //未满
    heap[currentSize] = x;
    siftUp(currentSize);   //向上调整
    currentSize++;           //当前元素数加1
    return true;
}
```

最小堆删除算法

函数原型：

```
template <class T>
bool MyHeap<T>::Remove(T& x)
```

代码：

```
//函数功能：最小堆删除算法，删除堆顶元素
//函数参数：将堆顶元素赋值给x
//返回值：逻辑值true或false
template <class T>
bool MyHeap<T>::Remove(T& x)
```

```
{
    //若堆为空
    if (!currentSize)
    {
        cout << "堆为空" << endl;
        return false;
    }
    //堆不空
    x = heap[0];                    // 返回最小元素
    heap[0] = heap[currentSize - 1]; //最后元素填补到根结点
    currentSize--;                   //当前元素数减1
    siftDown(0, currentSize - 1); //自上向下调整为堆
    return true;
}
```

最小堆向上调整算法

函数原型：

```
template<class T>
void MyHeap<T>::siftUp(int start)
```

代码：

```
//函数功能：最小堆向上调整算法
//函数参数：起点位置start
//返回值：无
template<class T>
void MyHeap<T>::siftUp(int start)
{
    //从start开始向上调整到0
    int j = start, i = (j - 1) / 2;//i是j的双亲
    T temp = heap[j];
    while (j > 0)
    {
        if (heap[i] <= temp)break;
        else
        {
            heap[j] = heap[i];
            j = i;
            i = (i - 1) / 2;//往上走
        }
    }
    heap[j] = temp;
}
```

最小堆向下调整算法

函数原型:

```
template<class T>
void MyHeap<T>::siftDown(int start, int m)
```

代码:

```
//函数功能：最小堆向下调整算法
//函数参数：起点位置start，终点位置m
//返回值：无
template<class T>
void MyHeap<T>::siftDown(int start, int m)
{
    int i = start, j = 2 * i + 1;
    T temp = heap[i];
    while (j <= m)
    {
        //两子女选小的
        if (j<m && heap[j]>heap[j + 1])
            j++;
        if (temp <= heap[j])break;
        else
        {
            heap[i] = heap[j];
            i = j;
            j = 2 * j + 1;//往下走
        }
    }
    heap[i] = temp;
}
```

HuffmanNode 结构体声明与实现

HuffmanNode:

该结构作为哈夫曼树的结点。其数据成员包括字符 key,字符出现的次数 times，该字符对应的编码 code，指向子节点的指针 leftChild，rightChild，指向双亲的指针 parent，构造函数 HuffmanNode(),重载<=运算符，重载>运算符，实现思路很简单，此处直接列出代码:

代码:

```
//////////////////////////////////
//接下来是HuffmanNode结构体的定义
//////////////////////////////////
struct HuffmanNode
{
    char key;//字母
    int times;//出现的次数
    string code;//该字符对应的编码
    HuffmanNode* leftChild, * rightChild;
    HuffmanNode* parent;
    //缺省构造函数
    HuffmanNode() :leftChild(NULL), rightChild(NULL), parent(NULL)
    {
        key = '\0';
        times = 0;
        code = "";
    }
    //参数构造函数
    HuffmanNode(char c,int t=0, HuffmanNode* left = NULL, HuffmanNode* right =
NULL,HuffmanNode*par=NULL)
    {
        key = c;
        times = t;
        leftChild = left;
        rightChild = right;
        parent = par;
        code = "";
    }
    //重载<=运算符，排序依据是该符号出现的次数times
    bool operator<=(HuffmanNode& R) { return times <= R.times; }
    //重载>运算符，排序依据是该符号出现的次数times
    bool operator>(HuffmanNode& R) { return times > R.times; }

};
```

HuffmanTree 类声明

studentDataBase 类用于构建哈夫曼树，并执行相关的操作。

公有属性包括*times(指向数组，数组存储字符出现次数)，wpl(二进制编码长度)，typeNum(出现的字符种类数)，*codes(指向字符串数组，存储每个字符对应的编码)，*keys(指向字符数组，存储出现的字符)，树的根结点地址*root。

公有方法包括构造函数 HuffmanTree()，析构函数~HuffmanTree()，由已知数组生成 Huffman 树的函数 createTree()，将两棵树合成一棵的函数 mergeTree()，将文件编码并将所得编码写入文件的函数 encoding()，将 fileName 文件译码并将保存译码文件的函数 decode()，将生成的 HuffmanTree 写入到 hfmTree.txt 文件的函数 save()，检测哈夫曼树是否为空的函数 isEmpty()，统计 wpl，wpl 是所有非叶子结点的权值之和函数 setWpl()，对 HuffmanTree 的每个叶子结点进行编码函数 encode()，编写 codes[]字符串数组的内容函数 setCodeArray()，统计文章中各字符出现次数并显示函数 countTimes()，画出并存储 huffman 树函数 displayHuffman(),删除树结点函数 deleteTree()。

在接下来的内容会逐个说明每个函数的实现方法。

HuffmanTree 类声明代码如下：

```
//////////////////////////////
//接下来是HuffmanTree类的定义
//////////////////////////////

class HuffmanTree
{
public:
    HuffmanTree();
    //析构函数，释放HuffmanTree的空间
    ~HuffmanTree()
    {
        deleteTree(root);
        delete[]times;
        delete[]codes;
        delete[]keys;
    }
    unsigned short *times;//数组，存储各字符出现的次数
    int wpl;//总编码长度
    short typeNum;//出现的字符种类数
    string* codes;//字符串数组，用于存储每个字符的编码
    char* keys;//字符数组，存储出现的字符
    HuffmanNode* root;//根结点
    //由已知数组生成HuffmanTree
    void createTree(char keys[], unsigned short w[], int n);
    //将两棵树合并成一棵
    void mergeTree(HuffmanNode* bt1, HuffmanNode* bt2, HuffmanNode*& parent);
```

```
        //将文件编码并将所得编码写入文件
        void encoding(string fileName);
        //将fileName文件译码并将保存译码文件
        void decode(string fileName);
        //将生成的HuffmanTree写入到hfmTree.txt文件
        void save();
        //检测哈夫曼树是否为空
        bool isEmpty() { return root == 0; }
        //统计wpl，wpl是所有非叶子结点的权值之和
        void setWpl(HuffmanNode* root);
        //功能是对HuffmanTree的每个叶子结点进行编码
        void encode(HuffmanNode* root, string s);
        //编写codes[]字符串数组的内容
        void setCodeArray(HuffmanNode* root);
        //子函数，功能是统计文章中各字符出现次数并显示
        void countTimes();
        //画出并存储树
        void displayHuffman();
    private:
        //子函数，辅助打印生成的HuffmanTree
        void displayHuffman(HuffmanNode* root, int ident, ofstream& out);
        //子函数，后序遍历删除树结点
        void deleteTree(HuffmanNode* root);
    };
```

HuffmanTree 类实现

接下来将逐个介绍 deleteTree()， HuffmanTree()， createTree()，mergeTree()， setWpl()， encode()， setCodeArray()， save()， encoding()，decode()，displayHuffman()，countTimes()的实现方法（根据需要会适度调整说明顺序，并不是严格按照以上顺序的）。

deleteTree()

函数原型：

```
void HuffmanTree::deleteTree(HuffmanNode* root)
```

返回值：

无

函数功能：

后序遍历方式释放二叉树空间

函数参数：

HufmanTree 的根结点地址 root

实现方式：

采用后续遍历的方式遍历哈夫曼树，对每一个结点，先释放其左子树结点的空间，再释放其右子树结点的空间，最后释放根结点地址的空间。

代码：

```
//函数功能：后序遍历方式释放二叉树空间
//函数参数：HuffmanTree的根结点root
//返回值：void
void HuffmanTree::deleteTree(HuffmanNode* root)
{
    if (root == 0)return;
    else
    {
        deleteTree(root->leftChild);
        deleteTree(root->rightChild);
        delete root;
    }

}
```

HuffmanTree()

函数原型：

```
HuffmanTree::HuffmanTree()
```

返回值：

无

函数功能：

构造函数，初始化各类数据

函数参数：

无

实现方式：

初始化各类数据即可。

代码：

```
//函数功能：构造函数，初始化各类数据
//函数参数：无
//返回值：无
HuffmanTree::HuffmanTree()
{
    codes = 0;
    times = 0;
    root = 0;
    typeNum = 0;
    keys = 0;
    wpl = 0;

}
```

mergeTree()

函数原型：

```
void HuffmanTree::mergeTree(HuffmanNode* bt1, HuffmanNode* bt2, HuffmanNode*& parent)
```

返回值：

无

函数功能：

将两棵树合成一棵树

函数参数：

两棵树的根结点 bt1,bt2,以及新树的根结点 parent

实现方式：

动态新建一个 HuffmanNode，将其地址赋值给 parent。之后使 parent 结点的 leftChild 指向 bt1，rightChild 指向 bt2，让 parent 的 parent 指向自己，parent

的权值 times 等于 bt1 和 bt2 的权值之和。因为 parent 结点肯定不是叶子结点，所以将 parent 的关键字设置成‘\0’。

代码：

```
//函数功能：将两棵树合成一棵树
//函数参数：两棵树的根结点bt1,bt2,以及新树的根结点parent
//返回值：void
void HuffmanTree::mergeTree(HuffmanNode* bt1, HuffmanNode* bt2, HuffmanNode*& parent)
{
    parent = new HuffmanNode;
    parent->leftChild = bt1;
    parent->rightChild = bt2;
    parent->parent = parent;
    parent->times = bt1->times + bt2->times;
    parent->key = '\0';//不是叶子结点的值设置成'\0'
    bt1->parent = bt2->parent = parent;
}
```

createTree()

函数原型：

```
void HuffmanTree::createTree(char keys[], unsigned short w[], int n)
```

返回值：

无

函数功能：

根据已有数组，建立 HuffmanTree，并建立结构体数组用于存储字符和出现次数的信息

函数参数：

存储各字符出现频率的数组 w，存储字符种类的数组 keys，字符种类总数 n

实现方式：

建立 n 个 HuffmanNode 结点，依次赋值后插入最小堆。之后对最小堆执行 n-1 此操作，每次操作，取出权值最小的两棵树，并将这两颗树合成为一棵新树。最后所得的 parent 值就是 HuffmanTree 的根结点地址 root。

代码：

```
//函数功能：根据已有数组，建立HuffmanTree，并建立结构体数组用于存储字符和出现次数的信息
//函数参数：存储各字符出现频率的数组w，存储字符种类的数组keys，以及字符种类总数n
//返回值：void
void HuffmanTree::createTree(char keys[], unsigned short w[], int n)
{
    MyHeap<HuffmanNode>heap1;
    HuffmanNode* parent = 0, * first, * second, * work, temp;
    for (int i = 0; i < n; i++)//逐个插入最小堆
    {
        work = new HuffmanNode();
        work->leftChild = NULL;//一定记得初始化！！
        work->rightChild = NULL;
        work->parent = work;
        work->key = keys[i];
        work->times = w[i];
        heap1.Insert(*work);
    }
    //执行n-1次操作，以建立哈夫曼树
    for (int i = 1; i <= n - 1; i++)
    {
        heap1.Remove(temp);
        first = temp.parent;
        heap1.Remove(temp);
        second = temp.parent;
        mergeTree(first, second, parent);//合并
        heap1.Insert(*parent);//新节点插入堆中
    }
    root = parent;

}
```

setWpl()

函数原型：

```
void HuffmanTree::setWpl(HuffmanNode*root)
```

返回值：

无

函数功能：

前序遍历递归计算 wpl

函数参数：

HuffmanTree 的根结点 root

实现方式：

Wpl 是所有非叶子结点的权值之和，所以只需要前序遍历哈夫曼树，若当前结点不是叶子结点，则加上该结点的权值，直到遍历完整棵树，所得的权值之和就是 wpl。

代码：

```
//函数功能：前序遍历递归计算wpl
//函数参数：根结点root
//返回值：void
void HuffmanTree::setWpl(HuffmanNode*root)
{
    if (!root)return;
    else
    {
        if (root->leftChild != 0 || root->rightChild != 0)//判断是否为叶子结点
            wpl += root->times;//wpl值是所有非叶子结点权值之和
        setWpl(root->leftChild);
        setWpl(root->rightChild);
    }

}
```

encode()

函数原型：

```
void HuffmanTree::encode(HuffmanNode* root, string s)
```

返回值：

无

函数功能：

对 HuffmanTree 的每个结点进行编码，规定往左走加'0'，往右走加‘1’

函数参数：

哈夫曼树的根结点 root，当前的编码 s。

实现方式：

从树根开始前序遍历，初始的编码 s 设置为”“，对于每一个结点，先往左走，需要把 s 末尾加上’0’，之和往右走，需要把之前加上的’0’改成 1。（其实就是先序遍历的一种形式）。

代码：

```
//函数功能：对HuffmanTree的叶子结点进行编码，规定往左走加'0'，往右走加 '1'
//函数参数：HuffmanTree的根结点root，以及当前的编码s
//返回值：void
void HuffmanTree::encode(HuffmanNode* root, string s)
{
    if (root == NULL)
        return;
    //先访问左子树
    //s需要加上'0'
    s = s + '0';
    if (root->leftChild != NULL)
    {
        root->leftChild->code = s;
    }
    encode(root->leftChild, s);//遍历左子树
    //往右走之前，先把最后一个编码换成1
    s[s.size() - 1] = '1';
    if (root->rightChild != NULL)
        root->rightChild->code = s;
    encode(root->rightChild, s);//遍历右子树

}
```

setCodeArray()

函数原型：

```
void HuffmanTree::setCodeArray(HuffmanNode* root)
```

返回值：

无

函数功能：

根据哈夫曼树叶子结点的编码，设置 codes[]字符串数组的内容

函数参数：

哈夫曼树的根结点 root

实现方式：

按照遍历的方式遍历哈夫曼树，若当前结点是叶子结点，则将该结点的编码写入到 codes[]数组的对应单元中去。

代码：

```
//函数功能：根据哈夫曼树叶子结点的编码，设置codes[]字符串数组的内容
//函数参数：根结点root
//返回值：void
void HuffmanTree::setCodeArray(HuffmanNode* root)
{
    if (root == 0)return;
    if (root->leftChild == 0 && root->rightChild == 0)
    {
        for (int i = 0; i < typeNum; i++)//找到对应的编号
            if (keys[i] == root->key)codes[i] = root->code;
    }
    setCodeArray(root->leftChild);
    setCodeArray(root->rightChild);
}
```

save()

函数原型：

```
void HuffmanTree::save()
```

返回值：

无

函数功能：

将生成的 HuffmanTree 存入 hfmTree.txt 文件，实际上存储的是每个字符以及字符出现的次数。

函数参数：

无

实现方式：

先写入字符集的大小 typeNum,之后将 keys[]和 times[]的内容写入文件即可。

代码：

```
//函数功能：将生成的HuffmanTree存入hfmTree.txt文件
//          实际上存储的是每个字符以及字符出现的次数
//函数参数：无
//返回值：void
void HuffmanTree::save()
{
    ofstream out("hfmTree.txt", ios::binary);
    if (!out)
    {
        cout << "文件打开失败！" << endl;
        return;
    }
    //先写入种类数
    out.write((char*)&typeNum, sizeof(short));
    //写入各字符和频度
    for (int i = 0; i < typeNum; i++)
    {
        out.write((char*)&keys[i], sizeof(keys[i]));
        out.write((char*)&times[i], sizeof(times[i]));
    }
    out.close();

}
```

encoding()

函数原型：

```
void HuffmanTree::encoding(string fileName)
```

返回值：

Void

函数功能：

将文件编码并将编码结果写入到 CodeFile.txt 文件

函数参数：

待编码的文件名

实现方式：

建立存储二进制编码的字符数组 binaryCode，数组长度为 wpl，并将数组初值设置为空字符’\0’，声明一个整形变量 flag 用于标记数组下标。之和开始从文件中读取字符，每读入一个字符，现在 codes[]中找到其对应的编码，将对应的编码写入 binaryCode，这里要注意的是 codes[i]是字符串，而 binaryCode 是字符数组，所以在写入时要使用到一个 for 循环。读取结束后 flag 的值就是最后 01 字符串的长度。读取完成后，一次从 binaryCode 中读取八个 01 字串，转换成十进制，以 unsigned char 的数据类型写入目标文件（8 位 01 字串最大表示的数字为 255，用 unsigned char 足够了，能够减少存储空间）。若 wpl 不能被 8 整除，则在最后几位 01 字串要做处理，此处默认在最后补 0 使其成为八位 01 字串。

代码：

```
//函数功能：将文件编码并将编码结果写入到CodeFile.txt文件
//函数参数：无
//返回值：void
void HuffmanTree::encoding(string fileName)
{
    //建立字符数组临时存储二进制编码
    char* binaryCode = new char[wpl];
    //初始元素设置为\0
    memset(binaryCode, '\0', sizeof(binaryCode));
    ofstream out("CodeFile.txt", ios::binary);
    ifstream in(fileName, ios::in);
    if (!in || !out)
    {
        cout << "打开文件失败！" << endl;
        return;
    }
    char c;//从文件读出来的字符
    //字符数组下标标记
    int flag = 0;
    while (in.peek() != EOF)
    {
        c = in.get();
        //找到c字符对应codes数组的哪个下标
        for(int u=0;u<typeNum;u++)
        {
```

```
                if (keys[u] == c)//找到
                {
                    for (int i = 0; i < codes[u].length(); i++)//将c的编码写到binaryCode
中
                    {
                        binaryCode[flag] = codes[u][i];
                        flag++;
                    }
                    break;
                }
            }
        }
        out.write((char*)&flag, sizeof(flag));
        //接下来开始转化成十进制并写入文件
        //此处的flag是二进制串的总长度
        short num = 0;
        int i = 0;
        for (i = 0; i < flag-flag%8; i++)
        {
            if(binaryCode[i]=='1')
                num += pow(2, 7 - i % 8);
            if ((i + 1) % 8 == 0)
            {
                unsigned char c = num;//因为num的最大值是255，用unsigned char即可存储，可
以节约存储空间，提升压缩率
                out.write((char*)&c,sizeof(c));
                num = 0;
            }
        }
        int temp = 128;
        for (i = flag-flag%8; i < flag ; i++)
        {
            num += (binaryCode[i]-'0')*temp;
            temp = temp / 2;
        }
        unsigned char numToWrite = num;
        out.write((char*)&numToWrite, sizeof(numToWrite));
        out.close();
        in.close();
        cout << "编码已经写入文件CodeFile.txt！" << endl;

    }
```

decode()

函数原型：

```
void HuffmanTree::decode(string fileName)
```

返回值：

无

函数功能：

将编码好的文件译码并将译码结果写入到 CodeFile.txt 文件。

函数参数：

待译码文件名 fileName。

实现方式：

先声明一个字符数组 binaryCode[]，并初始化为‘\0’用于存储译码产生的 01 二进制串。接下来从文件中读入字符串长度 WPL(此时的 WPL 并不是哈夫曼树的 wpl)，然后开始读取十进制数（注意，因为写入的时候时 unsigned char 类型），所以读取的时候也用 unsigned char 类型，之后赋值给 int 型变量即可。对于每次读入的十进制数，将其转成 8 位二进制码并存储在 binaryCode 中。最后若长度超过了 WPL，说明最后多出来的几位是之前补上去的，需要将补上去的去掉，之后才能得到正确的 01 编码。最后，从下标 0 开始，遍历 binaryCode，若值为 0，则哈夫曼树向左走，若为 1，就向右走。每当遇到叶子节点，就把叶子结点的值输入来并写入目标文件，然后下次就从哈夫曼树的根结点开始走，直到遍历完 binaryCode 为止。

代码：

```
//函数功能：将编码好的文件译码并将译码结果写入到CodeFile.txt文件
//函数参数：无
//返回值：void
void HuffmanTree::decode(string fileName)
{
    ifstream in(fileName, ios::binary);
    if (!in)
    {
        cout << "请先初始化哈夫曼树!" << endl;
```

```
        return;
    }
    char* binaryCode = new char[wpl];
    memset(binaryCode, '\0', sizeof(binaryCode));
    short num;//存储读进来的十进制数
    unsigned char c;
    int flag = 0;//字符数组下标
    int WPL;
    in.read((char*)&WPL, sizeof(WPL));
    //将读取进来的十进制数转成二进制并存入binaryCode数组
    for (int i = 1; i <= WPL / 8; i++)
    {
        in.read((char*)&c, sizeof(c));
        num = c;
        int temp = 128;
        for (int i = 0; i < 8; i++, flag++)
        {
            if (num >= temp)
            {
                num = num - temp;
                binaryCode[flag] = '1';
            }
            else
                binaryCode[flag] = '0';
            temp = temp / 2;
        }
    }
    //对最后一个十进制数是否译码为8为二进制数的判断
    if (WPL % 8 != 0)
    {
        in.read((char*)&c, sizeof(c));
        num = c;
        int temp = 128;
        for (int i = 0; i < WPL % 8; i++, flag++)
        {
            if (num >= temp)
            {
                num = num - temp;
                binaryCode[flag] = '1';
            }
            else
                binaryCode[flag] = '0';
            temp = temp / 2;
        }
```

```
        }
        //接下来对文件写入译码结果
        HuffmanNode* current = root;
        ofstream out("TextFile.txt", ios::out);
        if (!out)
        {
            cout << "文件打开失败！" << endl;
            return;
        }
        for (int i = 0; i < WPL; i++)
        {
            char c = binaryCode[i];
            if (c == '0')
            {
                current = current->leftChild;
            }
            else if (c == '1')
            {
                current = current->rightChild;
            }
            if (current->leftChild == NULL && current->rightChild == NULL)
            {
                c = current->key;
                out.write((char*)&c, sizeof(char));
                current = root;//复位，从树根开始往下走
            }
        }
        in.close();
        out.close();
        cout << "译码文件已经存入TextFile文件中！";
        cout << endl << endl;

    }
```

displayHuffman()

函数原型：

```
void HuffmanTree::displayHuffman()
void HuffmanTree::displayHuffman(HuffmanNode* root, int ident,ofstream&out)
```

返回值：

无

函数功能：

绘制生成的 huffman 树并写入文件

实现方式：

有两个函数，第一个函数用于打开文件，并传参数给第二个文件。第二个函数用于绘制和存储哈夫曼树。对于绘制操作，其实也是先序遍历哈夫曼树的过程。

代码：

```
//函数功能：绘制生成的huffman树并写入文件
//函数参数：无
//返回值：void
void HuffmanTree::displayHuffman()
{
    ofstream out("TreePrin.txt", ios::out);
    if (!out)
    {
        cout << "文件打开失败！" << endl;
        return;
    }
    displayHuffman(root, 0, out);
    out.close();
}

//函数功能：子函数打印生成的HuffmanTree
//函数参数：根结点root，数组下标ident
int vec_left[100] = { 0 };
void HuffmanTree::displayHuffman(HuffmanNode* root, int ident,ofstream&out)
{
    if (ident > 0)
    {
        for (int i = 0; i < ident - 1; ++i)
        {
            cout<<(vec_left[i] ? "│   " : "    ");//输出到屏幕
            out << (vec_left[i] ? "│   " : "    ");//输出到文件
        }
        cout<<(vec_left[ident - 1] ? "├──" : "└──");
        out << (vec_left[ident - 1] ? "├──" : "└──");
    }
    if (!root)
    {
        cout<<"(null)"<<endl;
        out << "(null)"<<endl;
```

```
            return;
        }
        if (root->key == '\0')//非叶子结点的关键字默认输出\0
        {
            cout << "\\0" << endl;
            out << "\\0" << endl;
        }
        else
        {
            cout << root->key << endl;
            out << root->key << endl;
        }
        if (!root->leftChild && !root->rightChild)
        {
            return;
        }
        vec_left[ident] = 1;
        displayHuffman(root->leftChild, ident + 1,out);//遍历左子树
        vec_left[ident] = 0;
        displayHuffman(root->rightChild, ident + 1,out);//遍历右子树
    }
```

countTimes()

函数原型：

```
void HuffmanTree::countTimes()
```

返回值：

无

函数功能：

统计文章中的字符出现的种类数和频率

函数参数：

无

实现方式：

开一个整型数组，长度为 130。值初始化为 0。0-129 号单元依次存储 ASCLL 值为 0-129 的字符出现次数。打开文件后读入字符，在相应数组位置的值加 1 即可。最后统计整个数组中值不为 0 的个数，这就是字符种类数了，之后输出次数和对应

的字符即可。

代码：

```
////函数功能：统计文章中的字符出现的种类数和频率
//函数参数：无
//返回值：无
void HuffmanTree::countTimes()
{
    int counts[128];
    memset(counts, 0, sizeof(counts));
    cout << "请输入要统计的文章名称" << endl;
    string fileName;
    cin >> fileName;
    ifstream in(fileName, ios::in);
    if (!in)
    {
        cout << "文件打开失败！";
        return;
    }
    char c;
    while (in.peek() != EOF)
    {
        c=in.get(); //读入一个字符
        counts[c - '\0']++;
    }
    int num = 0;//统计出现的字符种类
    for (int i = 31; i <128; i++)
    {
        if (counts[i] != 0)
        {
            c = i - 0;
            cout << c << " " << counts[i] << " " << endl;
            num++;
        }
    }
    cout << "字符集格个数为" << num << endl;
    cout << "(默认不统计非英文字符和除空格外的不可见字符)" << endl;
    in.close();

}
```

main.cpp 实现

为实现更快速度的交互过程，采用了函数指针数组来实现相应函数的调用。在函数 func0-func6 中，分别执行统计、初始化、编码、译码、打印代码、打印哈夫曼树、退出的操作。函数指针数组声明如下：

```
//函数指针数组的内容
//利用函数指针数组，避免使用了switch结构来实现相应函数的调用，提升效率
void func0(HuffmanTree &myTree);//统计
void func1(HuffmanTree &myTree);//初始化
void func2(HuffmanTree &myTree);//编码
void func3(HuffmanTree &myTree);//译码
void func4(HuffmanTree &myTree);//打印代码文件
void func5(HuffmanTree &myTree);//打印哈夫曼树
void func6(HuffmanTree& myTree);//退出

//函数指针数组，用于根据用户输入调用对应函数
void(*functionPointer[7])(HuffmanTree &myTree) =
{ func0,func1,func2,func3,func4,func5,func6 };
```

之后，在 main 函数中，根据用户输入，即可完成相应函数的调用，代码如下：
(说明：menu()函数实现的是显示菜单的功能，此处不做介绍)

```
//主函数
//函数功能：实现用户交互，调用对应函数
int main()
{
    int input;
    menu();
    cin >> input;
    HuffmanTree myTree;
    while (true)
    {
        if (input >= 0 && input <= 6)
        {
            (*functionPointer[input])(myTree);//调用对应函数
            menu();//显示菜单
        }
        else
            cout << "请输入正确的指令！" << endl;
        cin >> input;
    }
    return 0;
}
```

接下来介绍 func0-func6 的实现过程

func0()

函数原型：

```
void func0(HuffmanTree& myTree)
```

返回值：

无

函数功能：

统计文章中的字符出现的种类数和频率

函数参数：

HuffmanTree 类对象 myTree

实现方式：

执行 myTree.countTimes()函数即可

代码：

```
//函数功能：统计文章中字符出现的种类和次数
//函数参数：HuffmanTree类对象myTree
//返回值：void
void func0(HuffmanTree& myTree)
{
    myTree.countTimes();
}
```

func1()

函数原型：

```
void func1(HuffmanTree &myTree)
```

返回值：

无

函数功能：

执行初始化操作，从终端读入字符集大小 n 以及 n 个字符和权值，根据这些建立哈夫曼树。

函数参数：

HuffmanTree 类对象 myTree

实现方式：

先读入字符集大小 n，之后读入 n 个字符和 n 个权值，因为读入的字符可能包含空格，所以读入字符的时候要使用 getChar()函数。读取完成后，调用 createTree()函数建立哈夫曼树，并统计 wpl，对每个结点进行编码并将叶子结点的编码存在 codes[]数组中

代码：

```
//函数功能：执行初始化操作
//函数参数：HuffmanTree类对象myTree
//返回值：void
void func1(HuffmanTree &myTree)
{
  unsigned short num,times;
  char c;
  cout << "请输入字符集的个数" << endl;
  cin >> num;
  myTree.keys = new char[num+1];
  myTree.times = new unsigned short[num];
  myTree.codes = new string[num];

  for (int i = 0; i < num; i++)
  {
      cout << "请输入第"<<i+1<<"个字符" << endl;
      getchar();//去掉\n和\t
      myTree.keys[i] = getchar();
      cout << "请输入该字符的频度" << endl;
      getchar();//去掉\n和\t
      cin >> myTree.times[i];
  }
  myTree.typeNum = num;//字符种类数
  myTree.createTree(myTree.keys, myTree.times, num);//建立树
  myTree.setWpl(myTree.root);//统计wpl
  myTree.encode(myTree.root,"");//对每个字符进行相应的编码
```

```
        myTree.setCodeArray(myTree.root);
        myTree.save();//保存
    }
```

func2()

函数原型：

```
void func2(HuffmanTree &myTree)
```

返回值：

无

函数功能：

执行编码操作，并把编码存在 CodeFile.txt 文件中。

函数参数：

HuffmanTree 类对象 myTree

实现方式：

判断哈夫曼树是否在内存中，若不在，就先从 hfmTree 文件读入，若 hfmTree 文件不存在，就提醒用户先初始化哈夫曼树。读取完成后，执行编码操作。

代码：

```
//函数功能：执行编码操作
//函数参数：HuffmanTree类对象myTree
//返回值：void
void func2(HuffmanTree &myTree)
{
    if (myTree.isEmpty())
    {
        ifstream in("hfmTree.txt", ios::binary);
        if (!in)
        {
            cout << "请先初始化哈夫曼树！" << endl;
            return;
        }
        //先读入种类数
        in.read((char*)&myTree.typeNum, sizeof(myTree.typeNum));
        myTree.keys = new char[myTree.typeNum + 1];
```

```
            myTree.times = new unsigned short[myTree.typeNum];
            myTree.codes = new string[myTree.typeNum];
            //接着读入字符和频度
            for (int i = 0; i < myTree.typeNum; i++)
            {
                in.read((char*)&myTree.keys[i], sizeof(myTree.keys[i]));
                in.read((char*)&myTree.times[i], sizeof(myTree.times[i]));
            }
            myTree.createTree(myTree.keys, myTree.times, myTree.typeNum);//建立树
            myTree.setWpl(myTree.root);//统计wpl
            myTree.encode(myTree.root, "");//对每个字符进行相应的编码
            myTree.setCodeArray(myTree.root);
            myTree.save();//保存
        }
        //接下来开始编码
        string fileName;
        cout << "请输入你要编码的文件名" << endl;
        cin >> fileName;
        myTree.encoding(fileName);

    }
```

func3()

函数原型：

```
void func3(HuffmanTree &myTree)
```

返回值：

无

函数功能：

执行译码操作，并把译码存在 TextFile.txt 文件中。

函数参数：

HuffmanTree 类对象 myTree

实现方式：

判断哈夫曼树是否在内存中，若不在，就先从 hfmTree 文件读入，若 hfmTree 文件不存在，就提醒用户先初始化哈夫曼树。读取完成后，执行译码操作。

代码：

```
//函数功能：执行译码操作
//函数参数：HuffmanTree类对象myTree
//返回值：void
void func3(HuffmanTree &myTree)
{
    if (myTree.isEmpty())
    {
        ifstream in("hfmTree.txt", ios::binary);
        if (!in)
        {
            cout << "请先初始化哈夫曼树！" << endl;
            return;
        }
        //先读入种类数
        in.read((char*)&myTree.typeNum, sizeof(myTree.typeNum));
        //开始建立哈夫曼树
        myTree.keys = new char[myTree.typeNum + 1];
        myTree.times = new unsigned short[myTree.typeNum];
        myTree.codes = new string[myTree.typeNum];
        //先读入字符和频度
        for (int i = 0; i < myTree.typeNum; i++)
        {
            in.read((char*)&myTree.keys[i], sizeof(myTree.keys[i]));
            in.read((char*)&myTree.times[i], sizeof(myTree.times[i]));
        }
        myTree.createTree(myTree.keys, myTree.times, myTree.typeNum);//建立树
        myTree.setWpl(myTree.root);//统计wpl
        myTree.encode(myTree.root, "");//对每个字符进行相应的编码
        myTree.setCodeArray(myTree.root);//对编码数组codes[]进行赋值，以便后续操作
        myTree.save();//保存
    }
    //接下来开始解码
    cout << "请输入要解码的文件名称" << endl;
    string fileName;
    cin >> fileName;
    myTree.decode(fileName);
}
```

func4()

函数原型：

```
void func4(HuffmanTree &myTree)
```

返回值：

无

函数功能：

输出二进制码并将二进制码存在 CodePrin.txt 中。

函数参数：

HuffmanTree 类对象 myTree

实现方式：

先从编码文件中读入wpl,然后建立长度为wpl的字符数组binaryCode,然后读取编码文件中的十进制数，将十进制数转成 01 字符串存入 binaryCode 中，之后输出即可。设置一个计数器 temp，每输出一个字符，temp 就加 1，当 temp%50 是 0 时，就换行。

代码：

```
//函数功能：输出并存储二进制码操作
//函数参数：HuffmanTree类对象myTree
//返回值：void
void func4(HuffmanTree &myTree)
{
    //为省去建树的时间，考虑将wpl直接写入文件，则读取时可不依赖Huffman树即可输出二进制码
    string fileName;
    cout << "请输入要显示二进制码的文件名" << endl;
    cin >> fileName;
    ifstream in(fileName, ios::binary);
    ofstream out("CodePrin.txt", ios::out);
    if (!in||!out)
    {
        cout << "文件打开失败" << endl;
        return;
    }
    int wpl;
    in.read((char*)&wpl, sizeof(int));
    unsigned char c;
    int temp=0;//计数器
    //将读入的十进制数转换成01二进制代码
    for (int i = 0; i < wpl / 8; i++)
```

```
        {
            in.read((char*)&c, sizeof(c));
            int num = c;
            for (int i = 8; i > 0; i--)
            {
                if (c & (1 << i))
                {
                    cout << "1";
                    out << '1';
                }
                else
                {
                    cout << "0";
                    out << '0';
                }
                temp++;
                if (temp % 50 == 0)//保证每行50个
                {
                    cout << endl;
                    out << endl;
                }
            }
        }
        //对最后一位十进制数是否转成8位二进制代码的处理
        if (wpl % 8 != 0)
        {
            in.read((char*)&c, sizeof(c));
            int num = c;
            for (int i = wpl%8; i > 0; i--)
            {
                if (c & (1 << i))
                {
                    cout << "1";
                    out << '1';
                }
                else
                {
                    cout << "0";
                    out << '0';
                }
                temp++;
                if (temp % 50 == 0)//保证每行50个
                {
                    cout << endl;
```

```
                    out << endl;
                }
            }
        }
        cout << endl;
    }
```

func5()

函数原型：

```
void func5(HuffmanTree &myTree)
```

返回值：

无

函数功能：

打印 huffman 树并显示在屏幕上，并且写入到文件 TreePrin.txt 中。

函数参数：

HuffmanTree 类对象 myTree

实现方式：

判断哈夫曼树是否在内存中，若不在，就先从 hfmTree 文件读入，若 hfmTree 文件不存在，就提醒用户先初始化哈夫曼树。读取完成后，调用 myTree.displayHuffman()函数执行打印和写入文件操作。

代码：

```
//函数功能：打印并存储huffman树
//函数参数：HuffmanTree类对象myTree
//返回值：void
void func5(HuffmanTree &myTree)
{
    if (myTree.isEmpty())
    {
        ifstream in("hfmTree.txt", ios::binary);
        if (!in)
        {
            cout << "请先初始化哈夫曼树！" << endl;
            return;
        }
```

```
            //先读入种类数
            in.read((char*)&myTree.typeNum, sizeof(myTree.typeNum));
            myTree.keys = new char[myTree.typeNum + 1];
            myTree.times = new unsigned short[myTree.typeNum];
            myTree.codes = new string[myTree.typeNum];
            //接着读入字符和频度
            for (int i = 0; i < myTree.typeNum; i++)
            {
                in.read((char*)&myTree.keys[i], sizeof(myTree.keys[i]));
                in.read((char*)&myTree.times[i], sizeof(myTree.times[i]));
            }
            myTree.createTree(myTree.keys, myTree.times, myTree.typeNum);//建立树
            myTree.setWpl(myTree.root);//统计wpl
            myTree.encode(myTree.root, "");//对每个字符进行相应的编码
            myTree.setCodeArray(myTree.root);
            myTree.save();//保存
            myTree.displayHuffman();
        }
        else
            myTree.displayHuffman();
        cout << "哈夫曼树已经写入文件TreePrin.txt，非叶子结点的关键字默认显示\\0" << endl
    << endl;
        }
```

func6()

函数原型：

```
void func6(HuffmanTree &myTree)
```

返回值：

无

函数功能：

退出系统

函数参数：

HuffmanTree 类对象 myTree

实现方式：

执行 exit(0)即可。

代码：

```
//函数功能：退出程序
//函数参数：HuffmanTree类对象myTree
//返回值：void
void func6(HuffmanTree &myTree)
{
    cout << "祝您生活愉快，再见！" << endl;
    exit(0);
}
```

至此，实现部分介绍完成。

- 测试数据和 10 篇文章的编码解码产生的文件包在作业提交的压缩包中。
- 压缩前后文件大小变化的表格数据在本报告前面部分已经说明了。可以看到，使用二进制转十进制的压缩存储方式使得压缩率达到了 50%，效果还是很可观的。

四、实验总结（心得体会）

① 要注意实验时间的规划，若是堆在几天完成，容易手忙脚乱，实现思路不清晰。

② 吸收实验一的经验，考虑到用户隐私和数据安全，本次文件读取不再使用 ascll 文件，而是采用二进制文件读写方式。

③ 当待压缩的文件长度越大，压缩率越高。但是系统在读取二进制串的耗时会明显增加。

④ 重新审视我写的代码，发现有些能够复用代码的地方我没有做到复用，导致代码较为复杂，此外，对于各个函数的代码实现也不够精简。究其原因，我认为是我在写代码之前，没有做好系统的具体规划，而是想到什么就写什么。下次注意。

五、参考文献：

1、 《数据结构：用面向对象方法和 C++语言描述》殷人昆主编