目录

一．课程设计题目

设计一个哈夫曼编码、译码系统。对一个文本文件中的字符进行哈夫曼编码，生成编码文件；反过来，可将编码文件译码还原为一个文本文件。

(1) 读入一篇英文短文(文件扩展名为 txt)；

(2) 统计并输出不同字符在文章中出现的频率(空格、换行、标点等也按字符处理)；

(3）根据字符频率构建哈夫曼树，并给出每个字符的哈夫曼编码，其中求最小权值要求用堆实现；

(4）利用已建好的哈夫曼树，将文本文件进行编码，生成编码文件(编码文件后缀名为.huf)；

(5）根据相应哈夫曼编码，对编码后的文件进行译码，将 huf 文件译码为 txt 文件，与原 txt 文件进行比较。

二．算法设计思想

（一）Huffman 树

1.Huffman 树简介

（1）基本概念

路径：树中一个结点到另一个结点之间的分支构成这两个结点之间的路径。

路径长度：路径上的分支数目称作路径长度。

树的路径长度：从树根到每一个结点的路径长度之和。

结点的带权路径长度：在一棵树中，如果其结点上附带有一个权值，通常把该结点的路径长度与该结点上的权值的乘机称为节点的带权路径长度。

树的带权路径长度：树中所有叶子结点的带权路径长度之和，通常记作：

$$WPL=\sum_{k=1}^{n} w_k l_k$$

（2）Huffman 树的概念

假设有 n 个权值$\{w_1，w_2，...，w_n\}$，试构造一颗有 n 个叶子结点的二叉树，每个叶子结点带权为 w_i，则其中带权路径长度 WPL 最小的二叉树称做最优二叉树或 Huffman 树。

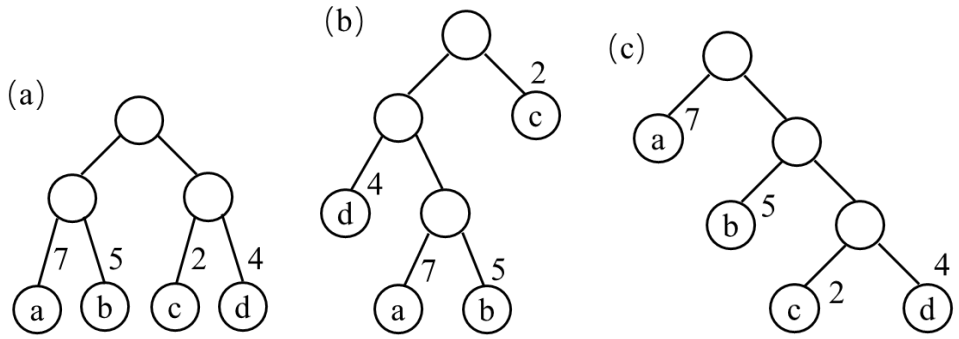
Huffman 树是一类带权路径最短的树，又称最优二叉树。

例如：（a）、（b）、（c）为 3 棵二叉树，都有 4 个叶子结点 a、b、c、d，分别带权 7、5、2、4，他们的带权路径长度分别为：

（a）WPL=7×2+5×2+2×2+4×2=36

（b）WPL=7×3+5×3+2×1+4×2=46

（c）WPL=7×1+5×2+2×3+4×3=35



2.最小堆排序简介

（1）最小堆的定义

①堆是一颗完全二叉树；

②堆中某个节点的值总是不大于或不小于其孩子节点的值；

③堆中每个节点的子树都是堆；

④当父节点的键值总是大于或等于任何一个子节点的键值时为最大堆。 当父节点的键值总是小于或等于任何一个子节点的键值时为最小堆。

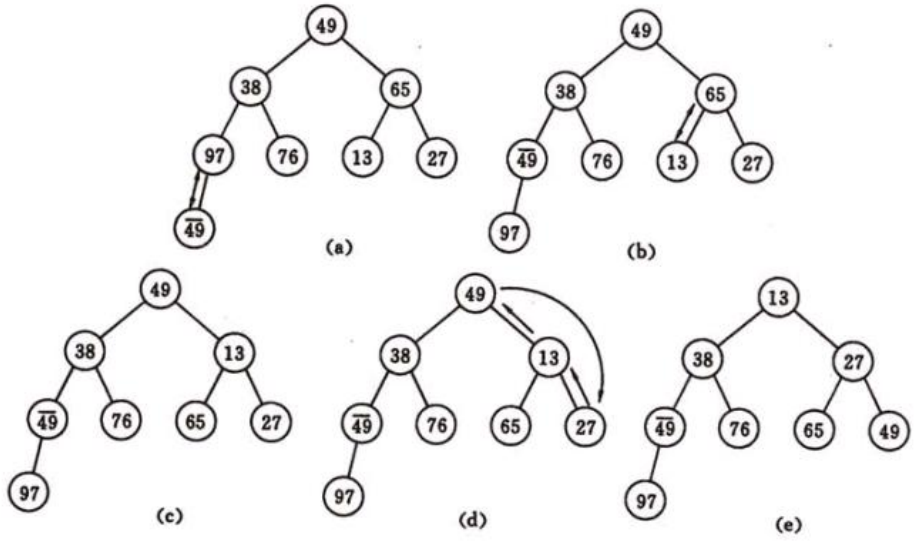
（2）最小堆的构造

元素下沉思想：从无序数组的最后往前遍历，堆树从下层到上层依次搭建完成：

①从叶子节点（无序数组的最后一个元素）出发，把当前的 parent 设置为无序数组中的最后一个数（也就是叶子节点），依次往前遍历，当 parent 是叶子节点，就不做操作；

②当 parent 遍历到非叶子节点，此时，它就有左子节点和右子节点或者只有左子节点。需要做的操作：先在左右子节点中找到最小元素所在的索引，然后将左右子节点中的最小元素与 parent 的节点，进行比较，如果小于，则交换，同时更新 parent 和 child 的位置。否则不交换，退出当前循环；

③“元素下沉”的终止条件就是父节点的元素值不小于其任意左右子节点的元素值，或者当前父节点无子节点（即当前节点为叶子节点）。



建初始堆过程示例

(a) 无序序列；(b) 97 被筛选之后的状态；(c) 65 被筛选之后的状态；

(d) 38 被筛选之后的状态；(e) 49 被筛选之后建成的堆

（二）Huffman 编码

1.Huffman 编码简介

目前，进行快速远距离通信的主要手段是电报，即将需传送的文字转换成由二进制的字符组成的字符串。在设计编码时需要遵守两个原则：

①发送方传输的二进制编码，到接收方解码后必须具有唯一性，即解码结果与发送方发送的电文完全一样；

②发送的二进制编码尽可能地短。下面介绍两种编码的方式：

a.等长编码

这种编码方式的特点是每个字符的编码长度相同（编码长度就是每个编码所含的二进制位数）。假设字符集只含有 4 个字符 A，B，C，D，用二进制两位表示的编码分别为 00，01，10，11。若现在有一段电文为：ABACCDA，则应发送二进制序列：00010010101100，总长度为 14 位。当接收方接收到这段电文后，将按两位一段进行译码。这种编码的特点是译码简单且具有唯一性，但编码长度并不是最短的。

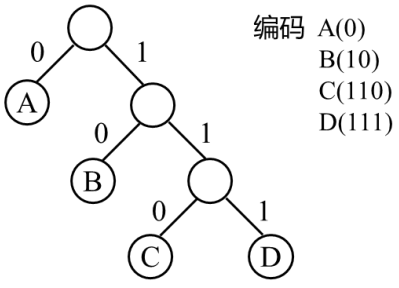
b.不等长编码

在传送电文时，为了使其二进制位数尽可能地少，可以将每个字符的编码设计为不等长的，使用频度较高的字符分配一个相对比较短的编码，使用频度较低的字符分配一个比较长的编码。例如，可以为 A，B，C，D 四个字符分别分配 0，00，1，01，并可将上述电文用二进制序列：000011010 发送，其长度只有 9 个二进制位，但随之带来了一个问题，接收方接到这段电文后无法进行译码，因为无法断定前面 4 个 0 是 4 个 A，1 个 B、2 个 A，还是 2 个 B，即译码不唯一，因此这种编码方法不可使用。

因此，若要设计长短不等的编码，则必须是任一个字符的编码都不是另一个字符的编码的前缀，这种编码称做前缀编码。

2.Huffman 编码的实现

可以利用二叉树来设计二进制的前缀编码。假设有一棵如图（2a）所示的二叉树，其 4 个叶子结点分别表示 A、B、C、D 这 4 个字符，且约定左分支表示字符’0’，右分支表示字符’1’，则可以从根节点到叶子结点的路径上分支字符组成的字符串作为该叶子结点字符的编码。可以证明，如此得到的必为二进制前缀编码。如图（2a）所得 A、B、C、D 的二进制前缀编码分别为 0、10、110 和 111。



(2a)

假设每种字符在电文中出现的次数为 wi，其编码长度为 li，电文中只有 n 种字符，则电文总长为。对应到二叉树上，若置 wi 为叶子结点的权，li 恰为从根到叶子的路径长度。则恰为二叉树上带权路径长度。由此可见，设计电文总长最短的二进制前缀编码即为以 n 种字符出现的频率作权，设计一棵 Huffman 树的问

题，由此得到的二进制前缀编码便称为 Huffman 编码。

由于 Huffman 树中没有度为 1 的结点，则一棵有 n 个叶子结点的 Huffman 树共有 2n-1 个结点，可以存储在一个大小为 2n-1 的一维数组中。由于在构成 Huffman 树之后，为求编码需从根结点出发走一条从跟到叶子的路径；而为译码也需从根出发走一条从根到叶子的路径。则对每个结点而言，不需知双亲的信息，只需知孩子结点的信息。由此，设定下述存储结构：

```
//定义哈夫曼树结构
typedef struct TreeNode* HuffmanTree;
struct TreeNode
{
    int Weight;         //权值
    char ch;        //ASCii 字符
    HuffmanTree Lchild;
    HuffmanTree Rchild;
};
```

（二）Huffman 译码

1.Huffman 译码简介

哈夫曼树译码是指由给定的代码求出代码所表示的结点值。

从文本中，依次读入原先编码的 Huffman 编码，再利用已经存储的 Huffman 树结构，根据编码信息往下进行搜索，则可得该 Huffman 编码所对应的字符。

2.Huffman 译码的实现

Huffman 译码的基本思想是：从根结点出发，逐个读入电文中的二进制代码；若代码为 0 则走向左孩子，否则走向右孩子；一旦到达叶子结点，便可译出代码所对应的字符。然后又重新从根结点开始继续译码，直到二进制电文结束。

操作步骤如下：

①读入一个字符，判断是’1’还是’0’；

②从根节点出发，出发走到该字符所对应的分支；

③若该分支节点没有左孩子和右孩子，则输出该节点所对应的字符；若有，则继续执行步骤②直至找到；

④若已经译出一个字符，则将数组清空，重新开始步骤①。

```
while(1)/*从根到叶子节点解码*/
        {
            if ((temp->Lchild==NULL) && (temp->Rchild==NULL))
                break;
            ch = fgetc(fp_code);
            if (ch == '0')
            {
                temp = temp->Lchild;
            }
            else
            {
                temp = temp->Rchild;
            }
```

```
            cnt++;   //计数已解码的哈夫曼码长度
            if (cnt >= code_length)
                break;
        }
        if((temp->Lchild==NULL)&&(temp->Rchild==NULL))          //找到叶子节点
            fputc(temp->ch, fp_decode);
```

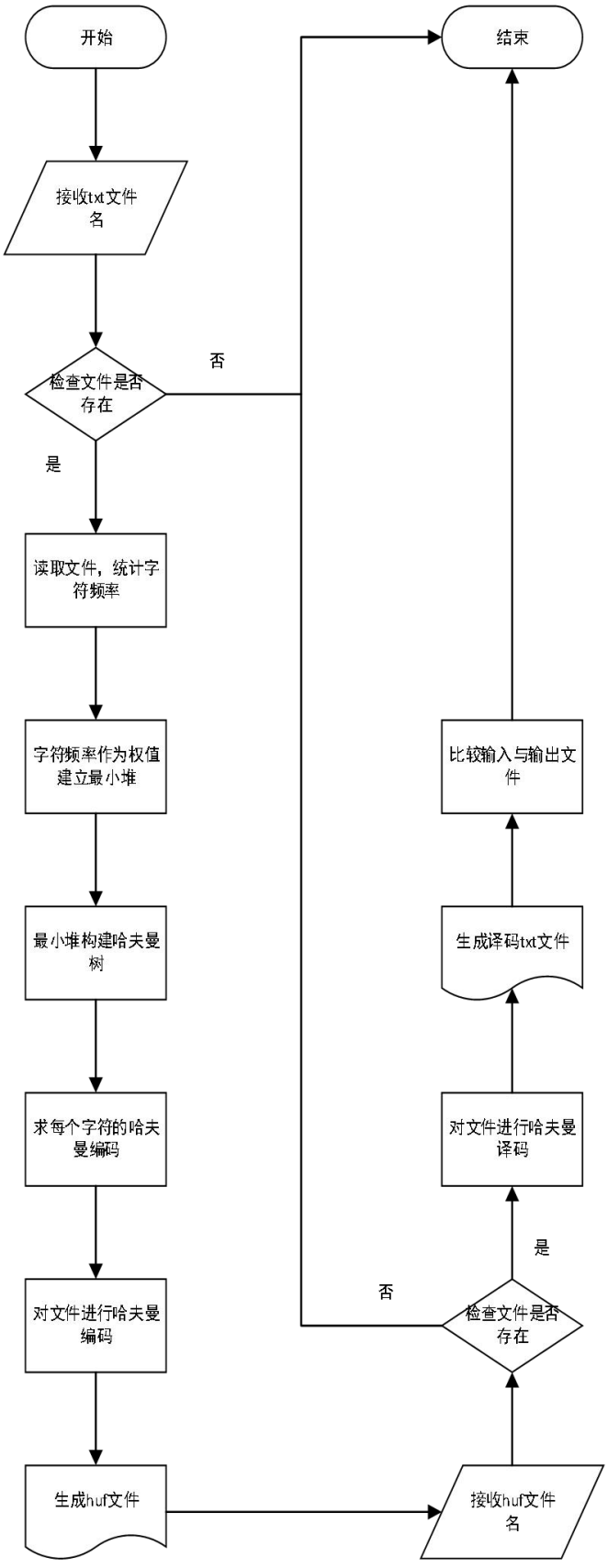
三．程序结构

（一）程序执行结构

1.程序实现功能

①读取 txt 文档；
②统计字符频率；
③创建 Huffman 树；
④求每个字符的 Huffman 码；
⑤对文件进行 Huffman 编码；
⑥根据编码生成 huf 文件；
⑦根据编码翻译 huf 文件；
⑧对 huf 翻译文件进行解码；
⑨比较输入文件与输出文件；

2.程序执行流程


开始
接收txt文件名
检查文件是否存在
否
是
读取文件，统计字符频率
字符频率作为权值建立最小堆
最小堆构建哈夫曼树
求每个字符的哈夫曼编码
对文件进行哈夫曼编码
生成huf文件
结束
比较输入与输出文件
生成译码txt文件
对文件进行哈夫曼译码
是
否
检查文件是否存在
接收huf文件名

（二）函数说明

1 哈夫曼树结构体

```
typedef struct TreeNode* HuffmanTree;
struct TreeNode
{
    int Weight;         //权值
    char ch;        //ASCii 字符
    HuffmanTree Lchild;   //左子树
    HuffmanTree Rchild;   //右子树
};
```

2 堆结构体

```
typedef struct HeapStruct* MinHeap;
struct HeapStruct {
    HuffmanTree* Data;   //存储堆元素的数组   存储时从下标 1 开始
    int Size;   //堆的当前元素的个数
    int Capacity;   //堆的最大容量
};
```

3 读取文件，统计各个字符出现频率,将频率作为权值生成堆

```
MinHeap MeasuringFrequency(char FileName[200],int& text_length);
```

4 构造新的哈夫曼树

```
HuffmanTree NewHuffmanNode();
```

5 创建容量为 MaxSize 的最小堆

```
MinHeap CreateMinHeap(int MaxSize);
```

6 将元素 item 插入到最小堆 H 中

```
Status Insert(MinHeap H, HuffmanTree item);
```

7 从最小堆 H 中取出权值为最小的元素，并删除一个结点

```
HuffmanTree DeleteMin(MinHeap H);
```

8 将 H->data[]按权值调整为最小堆

```
MinHeap BuildMinHeap(MinHeap H);
```

9 最小堆构造哈夫曼树

```
HuffmanTree Huffman(MinHeap H);
```

10 各个字符 Huffman 编码

```
Status HuffmanCode(HuffmanTree BST, int depth,int code[][100],int codelength [128]);
```

11 生成.huf 文件

```
Status FileHuffmancodeCreate(char FileName[200],int code[][100], int codelength [128]);
```

12 对.huf 文件进行译码

```
Status FileHuffmanEncode(HuffmanTree BST, char decode_name[200]);
```

13 文档对比

```
Status FileComparison(int text_length, char FileName[200], char decode_name[200]);
```

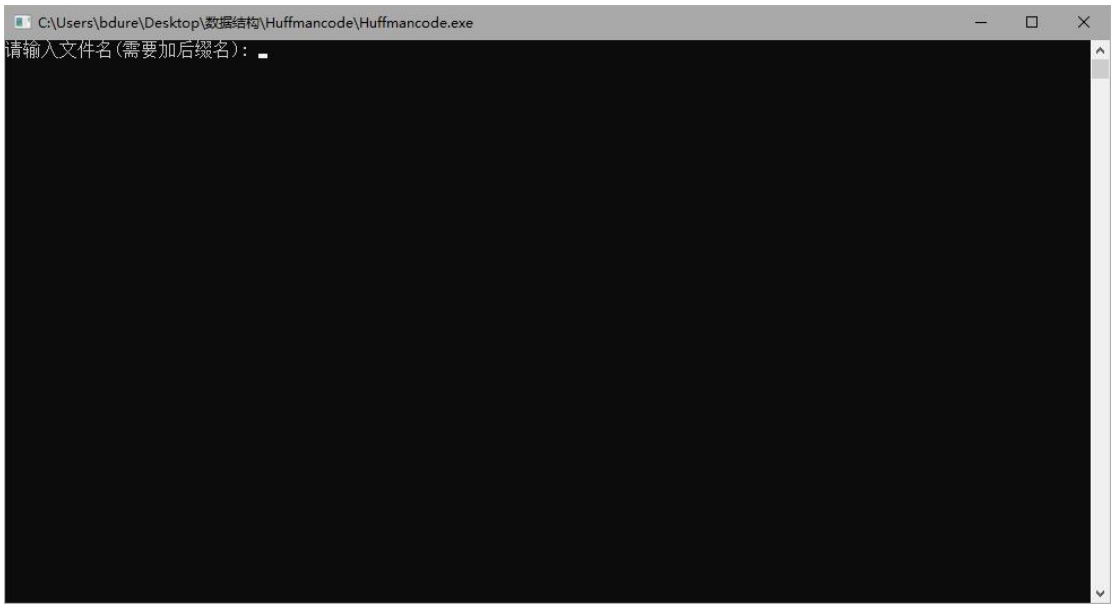
四．实验结果与分析

（一）程序执行结果

测试开始前请确保进行编解码的文件格式为.txt，且与可执行程序“Huffmancode.exe”位于同一目录下。

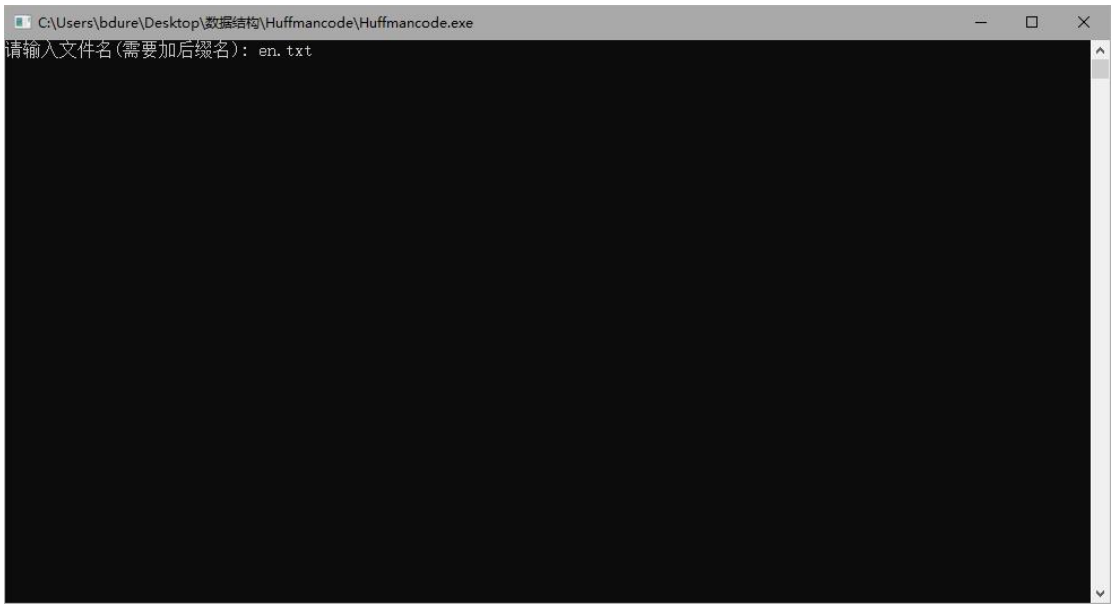
1.运行可执行程序

打开“Huffmancode”工程文件夹，点击根目录下的“Huffmancode.exe”可执行程序，程序开始运行；



2.读取 txt 文档

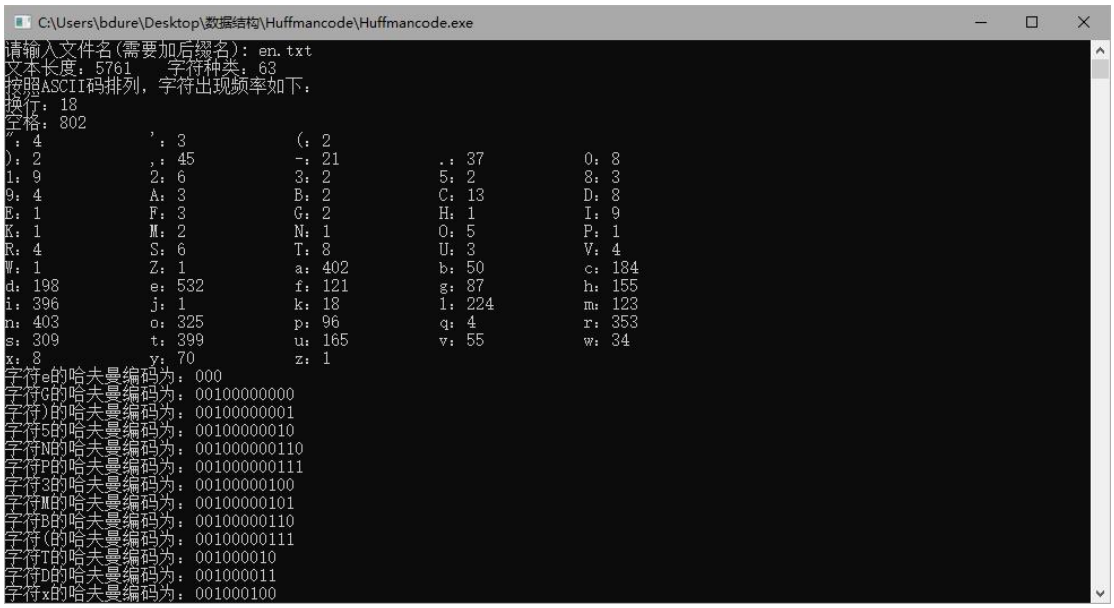
输入准备好的 txt 文件名，程序准备读取文档；



3.统计字符频率并编码

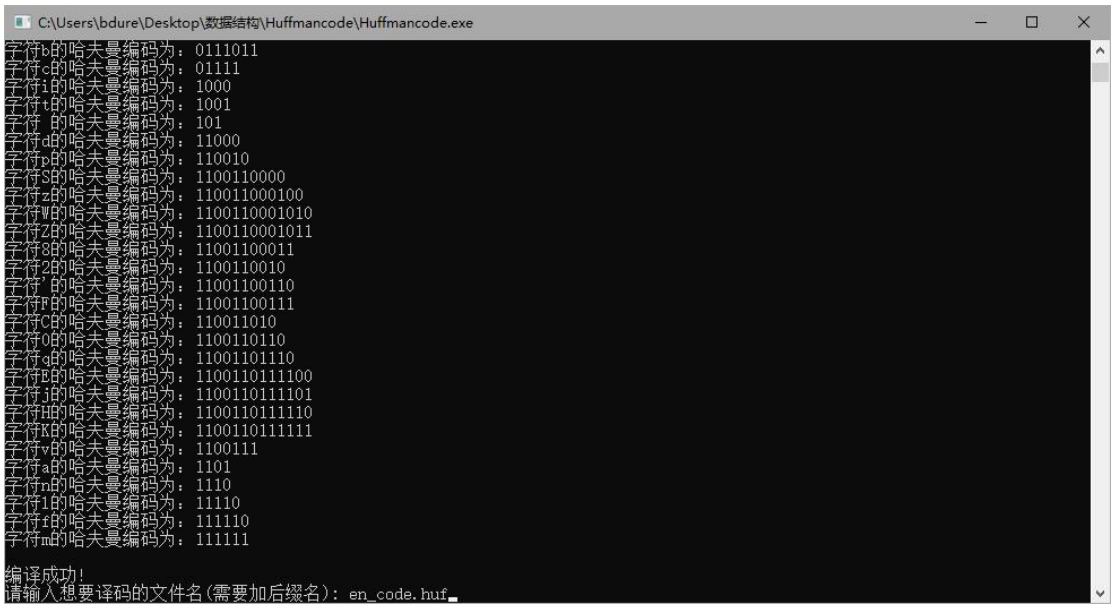
敲入回车键，程序将读取文档内的 ASCII 字符，统计字符种类及频率并输出

到控制台。将字符频率作为权值生成哈夫曼树，利用生成的哈夫曼树对字符进行编码。最后根据编码生成 huf 文件；



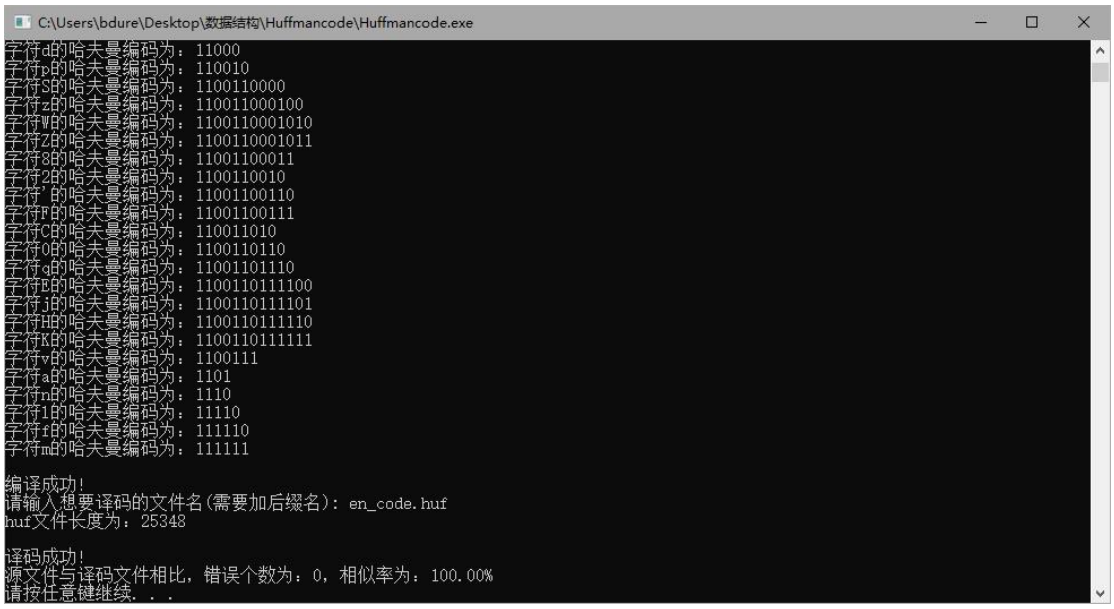
4.读取 huf 文件

输入由 txt 文档生成的编码后文件名，程序准备读取文件；



5.huf 文件解码并对比

敲入回车键，程序将读取 huf 文件编码信息，利用生成的哈夫曼树对文件进行解码，并将解码后的信息保存在“文件名_decode.txt”中。最后，程序对比输入和输出文件，统计解码后出错的数量并计算文件相似率。



五．总结

该程序完成了题目的全部要求。

经过测试，发现该程序能够实现 Huffman 编码/译码功能。程序能够实现的功能如下：

1.读入文档，该文档位于可执行程序根目录下。

2.统计并输出不同字符在文章中出现的频率。在处理空格、换行、标点等特殊字符时，为了让显示准确，程序会自动识别并作出标识，让用户能够更加清晰

的了解。

3.根据字符频率构建 Huffman 树，并给出每个字符的 Huffman 编码。

4.输出 Huffman 编码。程序会自动显示每个字符对应的 Huffman 编码。

5.利用已建好的 Huffman 编码，将文本文件进行编码，生成编码后文件“文件名_code.huf”。

6.进行译码，将 huf 文件译码为 txt 文件，与原 txt 文件进行比较。

然而，程序在设计时，仍有一些不足，如：

1.由于 Windows 系统特性，中文的编码方式与字符不同，无法使用 Huffman 算法进行编码，译码。

2.因时间与精力有限，没有对所有的 ASCII 字符进行测试。

3.因设备限制，没有将程序在不同的设备上运行，难以保证程序跨平台的兼容性、稳定性、准确性。

4.没有为程序编写 GUI 界面。

5.对于程序的执行逻辑、错误及异常处理没有很好的优化。

六．源程序

1.Huffman.h

```
#define _CRT_SECURE_NO_WARNINGS
#define OK 1;
#define ERROR 0;
#define MinData -10        /*随着堆元素的具体值而改变*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <direct.h>

typedef int Status;
//定义哈夫曼树结构
typedef struct TreeNode* HuffmanTree;
struct TreeNode
{
    int Weight;         //权值
    char ch;        //ASCii 字符
    HuffmanTree Lchild;
    HuffmanTree Rchild;
};
//定义堆结构
typedef struct HeapStruct* MinHeap;
struct HeapStruct {
    HuffmanTree* Data;   //存储堆元素的数组    存储时从下标 1 开始
    int Size;   //堆的当前元素的个数
    int Capacity;   //堆的最大容量
};
```

```
//读取文件，统计各个字符出现频率,将频率作为权值生成堆
MinHeap MeasuringFrequency(char FileName[200],int& text_length);
//构造新的哈夫曼树
HuffmanTree NewHuffmanNode();
//创建容量为 MaxSize 的最小堆
MinHeap CreateMinHeap(int MaxSize);
//将元素 item 插入到最小堆 H 中
Status Insert(MinHeap H, HuffmanTree item);
//从最小堆 H 中取出权值为最小的元素，并删除一个结点
HuffmanTree DeleteMin(MinHeap H);
//将 H->data[]按权值调整为最小堆
MinHeap BuildMinHeap(MinHeap H);
//最小堆构造哈夫曼树
HuffmanTree Huffman(MinHeap H);
//各个字符 Huffman 编码
Status HuffmanCode(HuffmanTree BST, int depth,int code[][100],int codelength
[128]);
//生成.huf 文件
Status FileHuffmancodeCreate(char FileName[200],int code[][100], int codelength
[128]);
//对.huf 文件进行译码
Status FileHuffmanEncode(HuffmanTree BST, char decode_name[200]);
//文档对比
Status FileComparison(int text_length, char FileName[200], char decode_name[200]);
```

2.MinHeap.cpp

```
#include "Huffman.h"

HuffmanTree Huffman(MinHeap H)
{/*假设 H->Size 个权值已经存在 H->data[]->Weight 里*/
    int i, num;
    HuffmanTree T;
    BuildMinHeap(H);   //将 H->data[]按权值调整为最小堆
    /*此处必须将 H->Size 的值交给 num,因为后面做 DeleteMin()和 Insert()函数
会改变 H->Size 的值*/
    num = H->Size;
    for (i = 1; i < num; i++)
    {   //做 H->Size-1 次合并     //此处教科书有问题！
        T = NewHuffmanNode();   //建立一个新的根结点
        T->Lchild = DeleteMin(H);   //从最小堆中删除一个节点，作为新 T 的左
子结点
        T->Rchild= DeleteMin(H);   //从最小堆中删除一个节点，作为新 T 的右子
结点
        T->Weight = T->Lchild->Weight + T->Rchild->Weight;   //计算新权值
```

```
            Insert(H, T);    //将新 T 插入到最小堆
        }
        T = DeleteMin(H);
        return T;
}

HuffmanTree NewHuffmanNode()
{
        HuffmanTree BST = (HuffmanTree)malloc(sizeof(TreeNode));
        BST->Weight = 0;
        BST->Lchild= BST->Rchild= NULL;
        return BST;
}

MinHeap CreateMinHeap(int MaxSize)
{   /*创建容量为 MaxSize 的最小堆*/
        MinHeap H = (MinHeap)malloc(sizeof(struct HeapStruct));
        H->Data = (HuffmanTree*)malloc((MaxSize + 1) * sizeof(HuffmanTree));
        H->Size = 0;
        H->Capacity = MaxSize;
        HuffmanTree T = NewHuffmanNode();
        T->Weight = MinData;    /*定义哨兵-为小于堆中所有可能元素权值的值，便于
以后更快操作*/
        T->ch = '\0';
        H->Data[0] = T;
        return H;
}

//插入新增节点
bool    IsFull(MinHeap H)
{
        return (H->Size == H->Capacity);
}

bool IsEmpty(MinHeap H)
{
        return (H->Size == 0);
}

Status Insert(MinHeap H, HuffmanTree item)
{               //将元素 item 插入到最小堆 H 中
        int i;
        if (IsFull(H))
        {
```

```
        printf("最小堆已满\n");
        return ERROR;
    }
    i = ++H->Size;   //i 指向插入后堆中的最后一个元素的位置
    for (; H->Data[i / 2]->Weight > item->Weight; i /= 2)   //无哨兵，则增加判决条件 i>1
        H->Data[i] = H->Data[i / 2];   //向下过滤结点
    H->Data[i] = item;     //将 item 插入
    return OK;
}

HuffmanTree DeleteMin(MinHeap H)
{/*从最小堆 H 中取出权值为最小的元素，并删除一个结点*/
    int parent, child;
    HuffmanTree MinItem, temp = NULL;
    if (IsEmpty(H))
    {
        printf("最小堆为空\n");
        return NULL;
    }
    MinItem = H->Data[1];   //取出根结点-最小的元素-记录下来
    //用最小堆中的最后一个元素从根结点开始向上过滤下层结点
    temp = H->Data[H->Size--];   //最小堆中最后一个元素，暂时将其视为放在了根结点
    for (parent = 1; parent * 2 <= H->Size; parent = child)
    {
        child = parent * 2;
        if ((child != H->Size) && (H->Data[child]->Weight > H->Data[child + 1]->Weight))
        {
            /*有右儿子，并且左儿子权值大于右儿子*/
            child++; //child 指向左右儿子中较小者
        }
        if (temp->Weight > H->Data[child]->Weight)
        {
            H->Data[parent] = H->Data[child];   //向上过滤结点-temp 存放位置下移到 child 位置
        }
        else
        {
            break;   //找到了合适的位置
        }
    }
    H->Data[parent] = temp;   //temp 存放到此处
```

```
    return MinItem;
}



MinHeap BuildMinHeap(MinHeap H)
{
    int i, parent, child;
    HuffmanTree temp;
    for (i = H->Size / 2; i > 0; i--)
    {   //从最后一个父结点开始，直到根结点
        temp = H->Data[i];
        for (parent = i; parent * 2 <= H->Size; parent = child)
        {
            /*向下过滤*/
            child = parent * 2;
            if ((child != H->Size) && (H->Data[child]->Weight > H->Data[child +
1]->Weight))
            {/*有右儿子，并且左儿子权值大于右儿子*/
                child++; //child 指向左右儿子中较小者
            }
            if (temp->Weight > H->Data[child]->Weight)
            {
                H->Data[parent] = H->Data[child];   //向上过滤结点-temp 存放位
置下移到 child 位置
            }
            else
            {
                break;   //找到了合适的位置
            }
        }/*结束内部 for 循环对以 H->data[i]为根的子树的调整*/
        H->Data[parent] = temp;   //temp 存放到此处
    }
    return H;
}
```

3.HuffmanCode.cpp

```
#include "Huffman.h"

Status HuffmanCode(HuffmanTree BST, int depth,int code[][100], int codelength
[128])   //depth 为目前编码到哈夫曼树的深度（层次）
{
    static int cd[100];            /*临时存储哈夫曼编码*/
    if (BST) /*从根到叶子节点编码*/
```

```
    {
        if ((BST->Lchild == NULL) && (BST->Rchild== NULL))
        {   //找到了叶结点
            if(BST->Weight!=0)
            {
                codelength[BST->ch] = depth;
                printf("字符%c 的哈夫曼编码为：", BST->ch);
                for (int i=0; i < depth; i++)
                    code[BST->ch][i] = cd[i];
                for (int i=0; i < depth; i++)
                    printf("%d", code[BST->ch][i]);
                printf("\n");
            }
        }
        else
        {
            cd[depth] = 0 ;   //往左子树方向编码为 0
            HuffmanCode(BST->Lchild, depth + 1, code,codelength);
            cd[depth] = 1 ;   //往右子树方向编码为 1
            HuffmanCode(BST->Rchild, depth + 1, code,codelength);
        }
    }
    return OK;
}

MinHeap MeasuringFrequency(char FileName[200],int& text_length)
{
    printf("请输入文件名(需要加后缀名): ");
    scanf_s("%s", FileName,200);
    FILE* fp1 = fopen(FileName, "r");
    FILE* fp2 = fopen(FileName, "r");
    int char_num[128] = { 0 };   //根据 ASCII 表，有 128 个字符
    int char_number = 0;
    int i, l = 0,n=0;
    unsigned char c;
    HuffmanTree T, BT = NULL;
    //初始化文件，若不存在则退出
    if (fp1 == NULL)
    {
        printf("读取文件失败或文件不存在！请重试！\n");
        return ERROR;
    }
    //统计字符频率及字符串长度
    while (!feof(fp1))
```

```
{
    c = fgetc(fp1);
    char_num[c]++;
    text_length++;
}
rewind(fp1);
//统计字符串中有多少种不同字符
for (i = 0; i < 128; i++)
{
    if (char_num[i] != 0)
        char_number++;
}
//创建以字符频率为权值、以字符种类为数量的哈夫曼堆
MinHeap H = CreateMinHeap(2*char_number);
for (i = 1; i <2*char_number; i++)
{
    T = NewHuffmanNode();
    T->Weight = 0;
    H->Data[i] = T;
}
n = 1;
for ( i = 0; i <=128; i++)
{
    if (char_num[i] != 0)
    {
        H->Data[n]->ch = i;
        H->Data[n]->Weight = char_num[i];
        n++;
    }
}
H->Size = char_number;
fclose(fp1);
//结果输出
printf("文本长度：%d      字符种类：%d\n", text_length, char_number);
printf("按照 ASCII 码排列，字符出现频率如下：\n");
for (i = 0; i < 128; i++)
{
    if (char_num[i] != 0)
    {
        if (i == 32)
            printf("空格：%d\n", char_num[i]);
        else if (i == 13)
            printf("回车：%d\n", char_num[i]);
        else if (i == 10)
```

```
                printf("换行：%d\n", char_num[i]);
            else
                printf("%c：%d\t\t", i, char_num[i]);
            l++;
            if (l % 5 == 0)
                printf("\n");
        }
    }
    printf("\n");
    return H;
}

Status FileHuffmancodeCreate(char FileName[200],int code[][100], int codelength
[128])
{
    int ch;
    char code_name[200];
    int i;
    for (i = 0; i < 200; i++)
    {
        code_name[i] = FileName[i];
        if (code_name[i] == '.')
            break;
    }
    code_name[i++] = '_';
    code_name[i++] = 'c';
    code_name[i++] = 'o';
    code_name[i++] = 'd';
    code_name[i++] = 'e';
    code_name[i++] = '.';
    code_name[i++] = 'h';
    code_name[i++] = 'u';
    code_name[i++] = 'f';
    code_name[i++] = '\0';
    //向.huf 文件中写入编码后的数据
    FILE* fp_souce = fopen(FileName, "r");
    FILE* fp_code = fopen(code_name, "w");
    if (fp_souce == NULL)
    {
        printf("文件打开失败!!!\n");
        return ERROR;
    }
    while ((ch = fgetc(fp_souce)) != EOF)
    {
```

```
            for (int i = 0; i < codelength[ch]; i++)
            {
                fprintf(fp_code, "%d", code[ch][i]);    //将 int 类型的哈夫曼编码写入
文件
            }
        }
        printf("\n 编译成功!\n");
        fclose(fp_souce);
        fclose(fp_code);
        return OK;
}

Status FileHuffmanEncode(HuffmanTree BST, char decode_name[200])
{
        char ch;
        int i,code_length = 0;
        volatile int cnt = 0;
        char code_name[200];
        HuffmanTree temp;
        printf("请输入想要译码的文件名(需要加后缀名): ");
        scanf_s("%s", code_name,200);
        for (i = 0; i < 200; i++)
        {
            decode_name[i] = code_name[i];
            if (code_name[i] == '_')
            {
                i++;
                break;
            }
        }
        decode_name[i++] = 'd';
        decode_name[i++] = 'e';
        decode_name[i++] = 'c';
        decode_name[i++] = 'o';
        decode_name[i++] = 'd';
        decode_name[i++] = 'e';
        decode_name[i++] = '.';
        decode_name[i++] = 't';
        decode_name[i++] = 'x';
        decode_name[i++] = 't';
        decode_name[i++] = '\0';
        //向_decode.txt 文件中写入解码后的数据
        FILE* fp_code = fopen(code_name, "r");
        FILE* fp_decode = fopen(decode_name, "w");
```

```
    i = 0;
    while (!feof(fp_code))
    {
        ch = fgetc(fp_code);
        code_length++;
    }
    printf("huf 文件长度为：%d\n", code_length);
    rewind(fp_code);
    while (1)
    {
        if (cnt >=code_length)
            break;
        temp = BST;
        while(1)/*从根到叶子节点解码*/
        {
            if ((temp->Lchild==NULL) && (temp->Rchild==NULL))
                break;
            ch = fgetc(fp_code);
            if (ch == '0')
            {
                temp = temp->Lchild;
            }
            else
            {
                temp = temp->Rchild;
            }
            cnt++;   //计数已解码的哈夫曼码长度
            if (cnt >= code_length)
                break;
        }
        if((temp->Lchild==NULL)&&(temp->Rchild==NULL))          //找到叶子
节点
            fputc(temp->ch, fp_decode);
    }
    fclose(fp_code);
    fclose(fp_decode);
    printf("\n 译码成功!\n");
    free(BST);
    return OK;
}

Status FileComparison(int text_length, char FileName[200],char decode_name[200])
{
    int i;
```

```
    char ch1, ch2;
    float error_num = 0.0, percentage = 0.0;
    FILE* source = fopen(FileName, "r");
    FILE* decode = fopen(decode_name, "r");
    //逐个字符进行比较，不一样则计数，一样则跳过看下一个字符
    for (i = 0; i < text_length; i++)
    {
        ch1 = fgetc(source);
        ch2 = fgetc(decode);
        if (ch1 != ch2)
            error_num++;
    }
    percentage = (text_length - error_num) / text_length;
    printf("源文件与译码文件相比，错误个数为：%d，相似率为：%.2f%%\n",
(int)error_num, percentage * 100);
    return OK;
}
```

4.main.cpp

```
#include "Huffman.h"

int main()
{
    static char FileName[200];     //源文件名（.txt）
    static int code[256][100]; //哈夫曼编码存储数组
    static int codelength[128];//ASCii 码对应的哈夫曼编码长度
    static char decode_name[200];//解码生成文件名(_decode.txt)
    int text_length = 0;          //源文件长度

    HuffmanTree hf=Huffman(MeasuringFrequency(FileName, text_length));// 根 据
字符频率生成对应的哈夫曼树
    HuffmanCode(hf, 0, code,codelength);                              //根据哈夫
曼树从根到叶子方向生成哈夫曼编码
    FileHuffmancodeCreate(FileName,code,codelength);                  // 根 据
哈夫曼编码生成.huf 文件
    FileHuffmanEncode(hf,decode_name);                                // 根 据
哈夫曼树解码并生成解码后文件
    FileComparison(text_length,FileName,decode_name);                 // 解 码
前后文件对比

    system("pause");
    return 0;
}
```