

《数据结构》实验报告

姓 名：	何杰煊	学 号：	14020032011
年 级：	2014 级	专 业：	保密管理
实验题目：	Huffman 树及 Huffman 编码的算法实现	实验时间：	第八~九周
任课老师：	高云	实验成绩：	

一、实验目的：

- 1、了解该树的应用实例，熟练掌握 Huffman 树的构造方法及 Huffman 编码的应用，
- 2、了解 Huffman 树在通信、编码领域的应用过程。

二、实验要求：

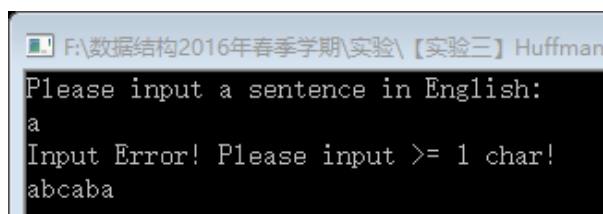
- 1、输入一段 100—200 字的英文短文，存入一文件 a 中。
- 2、写函数统计短文出现的字母个数 n 及每个字母的出现次数
- 3、写函数以字母出现次数作权值，建 Huffman 树（n 个叶子），给出每个字母的 Huffman 编码。
- 4、用每个字母编码对原文进行编码，码文存入文件 b 中。
- 5、用 Huffman 树对 b 中码文进行译码，结果存入文件 c 中，比较 a,c 是否一致，以检验编码、译码的正确性。

三、实验内容和实验步骤：

1、需求分析：

（1）输入形式：

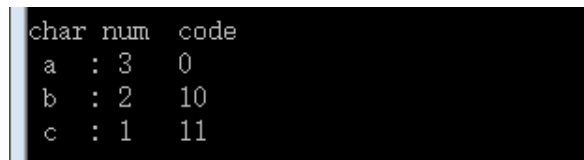
一串英文字符，大于 1 个字符，最好在 100~150 个字符内，可以超出该范围，允许有空格，但不能出现标点符号和数字。



```
F:\数据结构2016年春季学期\实验\【实验三】 Huffman
Please input a sentence in English:
a
Input Error! Please input >= 1 char!
abcaba
```

（2）输出形式：

- a) 句子中出现的字符，以及各个字符的数量，和对应 Huffman 编码。



```
char num  code
a  : 3   0
b  : 2  10
c  : 1  11
```

- b) 程序根目录 a.txt 文件中存放着该英文句子。



```
a - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
abcaba
```

- c) b.txt 文件存放利用该 Huffman 编码对英文句子的编码。



d) c.txt 文件存放利用该 Huffman 编码对 b.txt 文件的解码得到的英文字符串。



(3) 实现功能：对输入的字符串进行分析，统计出现的字符和各个字符出现的次数，把次数作为该字符的权值进行 Huffman 编码，并利用该规则对 Huffman 编码进行译码，对比译码和原文的结果。

2、概要设计：

(1) 数据结构：

- 哈夫曼数结点结构体，记录该节点的权值和左、右子树指针、双亲指针。
- 字符的结构体，记录该字符和该字符的权值。

```
typedef struct{
    unsigned int weight;
    unsigned int parent,lchild,rchild;
}HTNode, HuffmanTree; //动态分配数组存储哈夫曼树

typedef struct{
    char ch;
    int wei;
}Letter;
```

(2) 主函数流程：

- 为各个指针变量开辟空间。
- 输入字符串；将该字符串写入文件“a.txt”；并统计不同字符的个数以及每个字符个数出现的次数，作为其权值。
- 对各字符进行哈夫曼编码，并把各字符的编码结果输出到显示屏；然后按照该编码对字符串编码，并把编码结果写入文件“b.txt”。
- 从文件“b.txt”中读出编码，按照编码规则进行译码，把译码结果写入文件“c.txt”中。
- 释放各内存空间。

(3) 各程序模块之间的调用关系

```

//在HT[1..i-1]选择parent为0且weight最小的两个结点，其序号分别为s1和s2
+ void Select(HuffmanTree *HT, int i, int *s1, int *s2){

//设置一个哈夫曼树结点
+ void SetHTNode(HuffmanTree *T, int w, int p, int l, int r){

//存放n个字符的权值（均>0），构造哈夫曼树HT，并求出n个字符的哈夫曼编码HC
+ void HuffmanCoding(HuffmanTree *HT, HuffmanCode HC, Letter *w, int n){

//输入字符串，返回字符串的长度
+ int InputWord(char *str, int len){

//统计部分，计算各字母权值，返回字母（即权值）的个数
+ int CountWeight(Letter *let, char *str, int count, int n){

//输入一段100-200字的英文短文，存入一文件a中
+ void WriteText(char *str, int len){

//用每个字母编码对原短文进行编码，码文存入文件b中
+ int WriteCode(char *str, Letter *let, HuffmanCode HC, int num, int len){

//从文件b中读出编码
+ void ReadCode(char *s, int n){

//用Huffman树对b中码文进行译码，结果存入文件c中
+ void TranslateCode(char *s, Letter *let, HuffmanTree *HT, int num){

```

3、详细设计（主要展示 HuffmanCoding 函数和 TranslateCode 函数）

（1）HuffmanCoding 函数

```

//存放n个字符的权值（均>0），构造哈夫曼树HT，并求出n个字符的哈夫曼编码HC
void HuffmanCoding(HuffmanTree *HT, HuffmanCode HC, Letter *w, int n){
    int i, j; //用于循环
    int *s1, *s2; //用于寻找权值最小的两个结点的序号
    s1 = (int*)malloc(sizeof(int));
    s2 = (int*)malloc(sizeof(int));
    int m;

    HuffmanTree *p = NULL;

    if(n <= 1)
        exit(0);
    m = 2 * n - 1;

    //初始化哈夫曼树
    for(p = HT + 1, i = 1; i <= n; ++i, ++p, ++w)
        SetHTNode(p, w->wei, 0, 0, 0);
    for(; i <= m; ++i, ++p)
        SetHTNode(p, 0, 0, 0, 0);

    //建哈夫曼树
    for(i = n + 1; i <= m; ++i){
        //在HT[1..i-1]中选择parent为0且weight最小的两个结点，其序号分别为s1和s2
        Select(HT, i - 1, s1, s2);
        HT[*s1].parent = i;
        HT[*s2].parent = i;
        HT[i].lchild = *s1;
        HT[i].rchild = *s2;
        HT[i].weight = HT[*s1].weight + HT[*s2].weight;
    } //for

    /*打印哈夫曼树表
    printf("i weight parent lchild rchild\n");
    for(i = 1; i <= m; i++)
        printf("%3d%5d%6d%7d%8d\n", i, HT[i].weight, HT[i].parent, HT[i].lchild, HT[i].rchild);
    */
}

```

```

//从叶子到根逆向求每个字符的哈夫曼编码
int c, f, start; //中间变量
char *cd = NULL;
cd = (char*)malloc(n * sizeof(char)); //分配求编码的工作空间
cd[n - 1] = '\0'; //编码结束符
for(i = 1; i <= n; ++i){ //逐个字符求哈夫曼编码
    start = n - 1; //编码结束符位置
    for(c = i, f = HT[i].parent; f != 0; c = f, f = HT[f].parent) //从叶子到根逆向求编码
        if(HT[f].lchild == c)
            cd[--start] = '0';
        else
            cd[--start] = '1';
    HC[i] = (char*)malloc((n - start) * sizeof(char)); //为第i个字符编码分配空间
    strcpy(HC[i], &cd[start]); //从cd复制编码（串）到HC
} //for

//释放工作空间
free(cd);
free(s1);
free(s2);
} //HuffmanCoding

```

(2) TranslateCode 函数

思路：从哈夫曼树的根节点开始，根据读取的字符（‘0’或‘1’）寻找下一个结点，当当前结点没有

左右孩子，即找到叶子结点，将对应的译码字符存入文件。

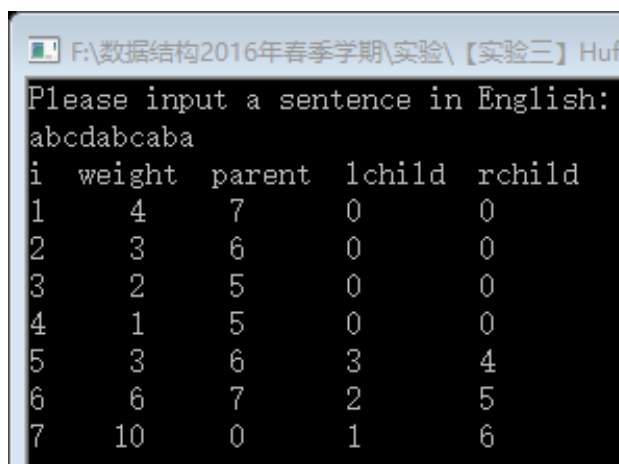
```
//用Huffman树对b中码文进行译码，结果存入文件c中
void TranslateCode(char *s, Letter *let, HuffmanTree *HT, int num){
    FILE *fp3 = NULL;
    if((fp3 = fopen("c.txt", "wb")) == NULL){
        printf("cannot open file\n");
        return;
    }
    int i, j = 2 * num - 1;
    for(i = 0; s[i] != '\0'; i++){
        //思路：从哈夫曼树的根节点开始，根据读取的字符寻找下一个结点。当当前结点没有左右孩子，即找到叶子结点，存入文件
        if(s[i] == '0')
            j = HT[j].lchild;
        else
            j = HT[j].rchild;

        if(HT[j].lchild == 0 && HT[j].rchild == 0)
            if(fwrite(let + j - 1, sizeof(char), 1, fp3) != 1)
                printf("file write error\n");
            else
                j = 2 * num - 1;
    }
    fclose(fp3);
}
```

4、调试分析：

(1) 通过哈夫曼树表判断编码是否正确？

实际上对于同字符串，哈夫曼编码是不唯一的。可以通过 a、c 文件对比来判断哈夫曼编码是否唯一，但是在程序设计时，进行分块设计，先设计编码功能，该模块正确后再（加上）完善对文件读写的功能。在命令窗口无法图形化输出，为了方便，将采取书上的讲解哈夫曼编码过程所用的表格作为核对标准。加入输出提示。（在上述 HuffmanCoding 函数截图中有注释，以下是输出截图）



i	weight	parent	lchild	rchild
1	4	7	0	0
2	3	6	0	0
3	2	5	0	0
4	1	5	0	0
5	3	6	3	4
6	6	7	2	5
7	10	0	1	6

实际上，左右孩子的序号顺序可变，例如上图中 i=5 的那行，可以是 lchild=4 而 rchild=5，为了方便核对，在 Select 函数中（在 HT[1..i-1] 选择 parent 为 0 且 weight 最小的两个结点，其序号分别为 s1 和 s2）s1 指向序号（不是权重，是序号）较小的，s2 指向序号较大的。

(2) Select 函数（在 HT[1..i-1] 选择 parent 为 0 且 weight 最小的两个结点，其序号分别为 s1 和 s2）。

- 思路大致有两种，一是新建数组让权值升序排序存放，选取下标前两位，但是这样空间复杂度较高，而且排序算法的时间复杂度不低，因为是对数组的内容排序而返回的是数组的下标。二是先寻找数组中权值最小的，再寻找除去最小权值外最小的。第二种方法优于第二种，尤其是对于权值个数比较多的情况。

- b. 思路二中设置一变量（Wmin）记录当前最大权值，当找到比它小的权值则替换。所以该变量一开始的值是比所有权值都大的数。一开始调试时该变量记录的是权值最大的数，而实际上这是有问题的，如果恰好最大权值有两个（权值相同），那么寻找就会出错，所以应当在时权值最大的数+1。
- c. 上述 b 问题想通后，打算在寻找时就+1，但其实应该寻找完后+1，即原本是在 if 里就进行 m+1，但是因为这样就会改变下一次 m 对比的值，举例子，权值是 1112，当前 m=1，则按照错误版本，m=1+1，那么下次比较是 m 就是 2，并且不会有比 2 更大的数了最终循环完后 m=2，实际上我想要的结果是对比完后，m=3。

```
//在HT[1..i-1]选择parent为0且weight最小的两个结点，其序号分别为s1和s2
void Select(HuffmanTree *HT, int i, int *s1, int *s2){
    int j, Wmin, m = 0;

    //寻找最大权重最大的值，方便之后寻找最大值时设置初始比较值。
    for(j = 1; j <= i; j++)
        if(HT[j].parent == 0 && HT[j].weight > m)
            m = HT[j].weight;

    //寻找最小的值
    Wmin = m + 1; //加1的原因：设置一个比数列中所有数都大的数。
    for(j = 1; j <= i; j++)
        if(HT[j].parent == 0 && HT[j].weight < Wmin){
            Wmin = HT[j].weight;
            *s1 = j;
        } //if

    //寻找第二小的值
    Wmin = m + 1;
    for(j = 1; j <= i; j++)
        if(HT[j].parent == 0 && j != *s1 && HT[j].weight < Wmin){
            //此处不加<=是因为如果碰到两个权值相等的第二小，我们选择的是序号较小的那个
            Wmin = HT[j].weight;
            *s2 = j;
        } //if

    if(*s1 > *s2){ //把序号小的靠前
        j = *s1;
        *s1 = *s2;
        *s2 = j;
    } //if
} //Select
```

（3）realloc 函数的使用（空间分析）

为了平衡内存空间与容错性的矛盾，良好的测试结果是输入少于但接近 100 个字符的字符串，但是实际上允许用户输入更多的字符，这时候就需要 realloc 重新分配空间来存储字符串。于是使用到 realloc 函数。通过网上查询资料得出以下知识点：

- a. 如果当前内存段后面有需要的内存空间，则直接扩展这段内存空间，realloc()将返回原指针。
- b. 如果当前内存段后面的空闲字节不够，那么就使用堆中的第一个能够满足这一要求的内存块，将目前的数据复制到新的位置，并将原来的数据块释放掉，返回新的内存块位置。
- c. 如果申请失败，将返回 NULL，此时，原来的指针仍然有效。

```

//输入字符串,返回字符串的长度
int InputWord(char *str, int len){
    char ch;
    int count = 0; //实际字符数
    while((ch = getchar()) != '\n'){
        count++;
        if(count >= len)
            str = (char*)realloc(str, sizeof(char) * (len++));
        str[count - 1] = ch;
    } //while
    str[count] = '\0';

    return count;
} //InputWord

```

平时用这个函数是为了将分配的内存扩大，实际上这个函数是重新分配的意思，可以是缩小原来的内存，例如用户只输入了几个字符，或者，考虑到只统计大小写英文和空格共 53 个字符，所以一开始则开辟 53 个存放字符及其权值的结构体内存，实际上有些字符个数是 0 则没必要进行编码，这时为了节省空间，则应当缩小空间。（如下图最后一个 if 语句）

```

//统计部分 计算各字母权值,返回字母(即权值)的个数
int CountWeight(Letter *let, char *str, int count, int n){
    int i, j, num = 0; //n为默认不同字母的个数的初始值, num为实际不同字母的个数
    for(i = 0; i < n; i++){
        let[i].wei = 0;
    }
    for(i = 0; i < count; i++){
        if(('a' <= str[i] && str[i] <= 'z') || (str[i] == ' ') || ('A' <= str[i] && str[i] <= 'Z')){
            //考虑到小写字母出现的频率普遍比大写高, 为了提高效率, 所以先判断是否小写再判断是否大写
            for(j = 0; j < num; j++){
                if(let[j].ch == str[i]){
                    let[j].wei++;
                    break;
                } //if
            }
            if(j == num){
                let[j].ch = str[i];
                let[j].wei++;
                num++;
            } //if
        } //if(a~z || A~Z)
    } //for
    if(num < n) //节省内存空间
        let = (Letter*)realloc(let, sizeof(Letter) * (num));
    return num;
} //CountWeight

```

(4) 容错性考虑:

对于只有一种字符的情况视为输入不合法，因为没有编码的必要，所以要求重新输入。分两种情况，一是只输入一个字符；二是虽然输入多个字符，但都是同一个字符。

```

printf("Please input a sentence in English:\n");
len = InputWord(str, len); //输入英文短文并返回其的长度
num = CountWeight(let, str, len, num); //计算英文短文中各字母的出现的次数(即权值)并返回权值的个数
while(len == 1 || num == 1){
    printf("Input Error! Please input >= 1 char!\n");
    len = InputWord(str, len);
    num = CountWeight(let, str, len, num);
}

```

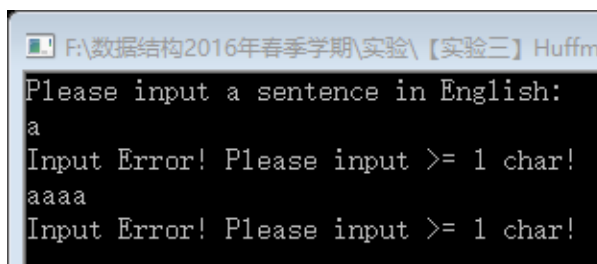
(5) 对于条件中或运算的优化

多个判断条件且条件间的逻辑关系是“或”，那么只要有一个条件为真，整个判别式值为真。为了减少判断，可以将为真可能性更大的条件放前面。例如在统计字符时，一般书写习惯中小写字母最多，空格其次，大写字母较少，所以判别条件则可以按照该顺序排放，能一定程度上提高运行效率。

```
for(i = 0; i < count; i++){
    if(('a' <= str[i] && str[i] <= 'z') || (str[i] == ' ') || ('A' <= str[i] && str[i] <= 'Z')){
        //考虑到小写字母出现的频率普遍比大写字高，为了提高效率，所以先判断是否小写再判断是否大写
        for(j = 0; j < num; j++){
            if(let[j].ch == str[i]){
                let[j].wei++;
                break;
            }
        }
        if(j == num){
            let[j].ch = str[i];
            let[j].wei++;
            num++;
        }
    }
}
}
}
```

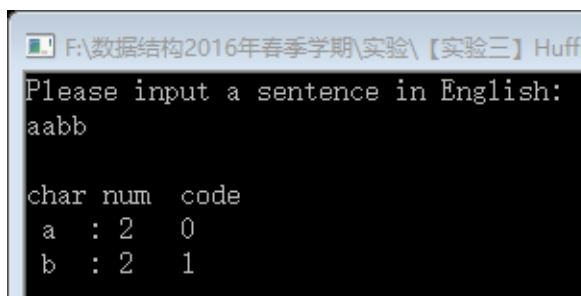
四、实验结果：

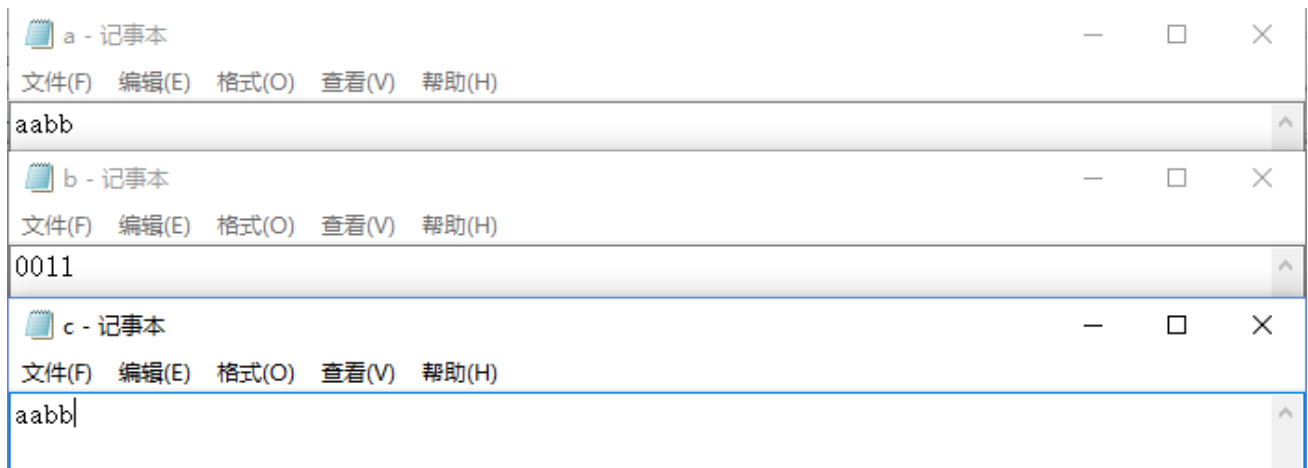
1、只有一个字符或一种字符的情况：



(需要重新输入)

2、只有两个字符，且权值相同：





3、两个或以上字符，且权值都相同：

```

F:\数据结构2016年春季学期\实验\【实验三】Huff
Please input a sentence in English:
abcdabcdabcd

char num  code
a  : 3    00
b  : 3    01
c  : 3    10
d  : 3    11

```



4、正常情况：（不带标点符号和数字，允许带空格）

测试语句：One day a rabbit was in the garden eating carrots When he was about to go home he saw the farmer

```
F:\数据结构2016年春季学期\实验\【实验三】 Huffman树及 Huffman 编码的算法实现\Huffman.exe
Please input a sentence in English:
One day a rabbit was in the garden eating carrots When he was about to go home he saw the farmer

char num  code
O  : 1   1111100
n  : 5   0110
e  :10   1110
   :20   00
d  : 2   111100
a  :11   010
y  : 1   1111101
r  : 6   1000
b  : 3   10010
i  : 3   10110
t  : 7   1100
w  : 3   10111
s  : 4   11010
h  : 6   1010
g  : 3   11011
c  : 1   1111110
o  : 5   0111
W  : 1   1111111
u  : 1   100110
m  : 2   111101
f  : 1   100111
```

a - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

One day a rabbit was in the garden eating carrots When he was about to go home he saw the farmer

b - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

111110001101110001111000101111101000100010000101001010010101101100001011101011010001
011001100011001010111000110110101000111100111001100011100101100101100110110110011111
10010100010000111110011010001111111010111001100010101110001011101011010000101001001
111001101100001100011100110110111001010011111110111100010101110001101001010111001100
1010111000100111010100011110111101000

c - 记事本

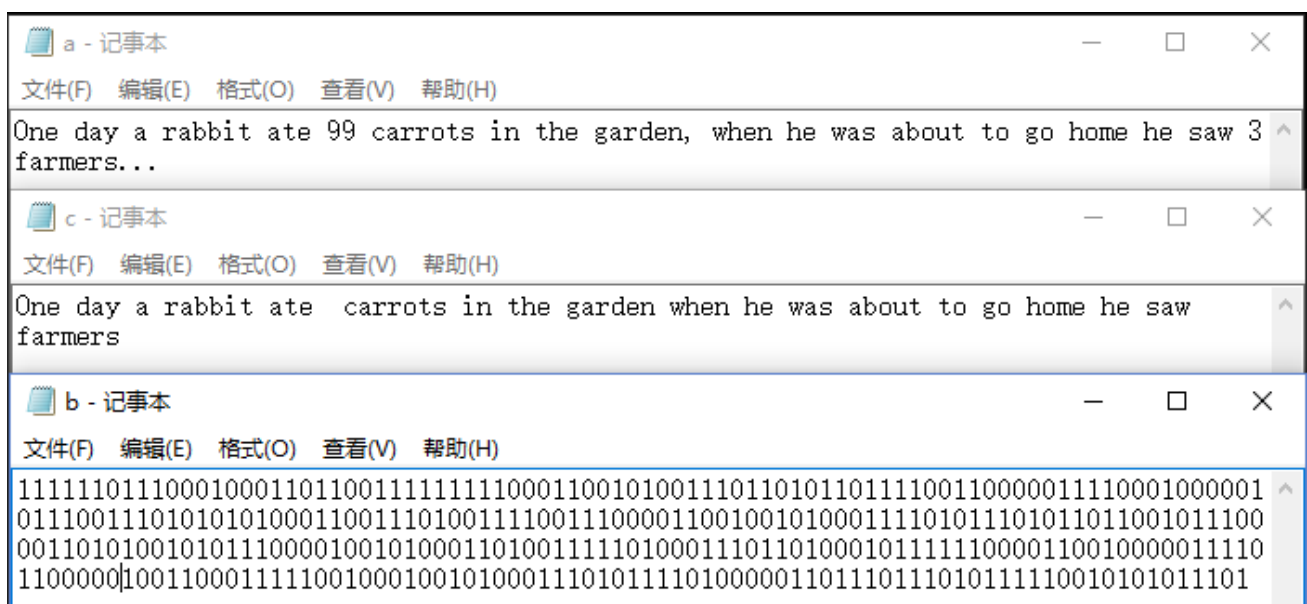
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

One day a rabbit was in the garden eating carrots When he was about to go home he saw the farmer

5、带标点符号或数字的（则还原 c 文件将缺失 a 文件中的所有标点符号和数字）

```
F:\数据结构2016年春季学期\实验\【实验三】 Huffman树及Huffman编码的算法实现\Huffman.exe
Please input a sentence in English:
One day a rabbit ate 99 carrots in the garden, when he was about to go home he saw 3 farmers...

char num  code
O : 1    1111110
n : 4    11100
e : 9    010
:20     00
d : 2    110110
a :10    011
y : 1    1111111
r : 6    1010
b : 3    10110
i : 2    111100
t : 6    1100
c : 1    101110
o : 5    1000
s : 4    11101
h : 5    1001
g : 2    111101
w : 3    11010
u : 1    101111
m : 2    111110
f : 1    110111
```



五、实验总结:

- 1、只要是指针，就要养成定义时指向 NULL，使用前为其开辟空间，使用后释放空间的习惯。
- 2、通过实验更加了解哈夫曼编码的过程。
- 3、强化了对 c 语言中文件的操作的认识。
- 4、realloc 函数既可以用来扩大已分配的内存，也能缩小已经分配的内存，对于它的扩大机制，也有一定的了解与认识。在需要动态内存分配且对内存有一定要求时，这个函数很好用。
- 5、对于同一英文串，哈夫曼编码不唯一，但最后的长度应该是唯一的。因为一个带有左右孩子的结点，

它左右孩子可以互换，这个动作不会影响编码的长度。而这一步的关键是 **Select** 函数对 **s1** 和 **s2** 的先后顺序处理。但是无论怎样，用同一哈夫曼树编码和译码，则译码后的文字一定和原来的文字是一致的。

附：源代码

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
```

```
typedef struct{
    unsigned int weight;
    unsigned int parent,lchild,rchild;
}HTNode, HuffmanTree;//动态分配数组存储哈夫曼树
```

```
typedef struct{
    char ch;
    int wei;
}Letter;
```

```
typedef char **HuffmanCode;//动态分配数组存储哈夫曼编码表
```

```
//在 HT[1..i-1]选择 parent 为 0 且 weight 最小的两个结点，其序号分别为 s1 和 s2
```

```
void Select(HuffmanTree *HT, int i, int *s1, int *s2){
    int j, Wmin, m = 0;
```

```
    //寻找最大权重最大的值，方便之后寻找最大值时设置初始比较值。
```

```
    for(j = 1; j <= i; j++)
```

```
        if(HT[j].parent == 0 && HT[j].weight > m)
```

```
            m = HT[j].weight;
```

/*错误记录：原本是在 if 里就进行 m+1，但是因为这样就会改变下一次 m 对比的值，举例子，权值是 1112，当前 m=1，则按照错误版本，m=1+1

那么下次比较是 m 就是 2，并且不会有比 2 更大的数了最终循环完后 m=2，实际上我想要的结果是对比完后，m=3。*/

```
    //寻找最小的值
```

```
    Wmin = m + 1;//加 1 的原因：设置一个比数列中所有数都大的数。
```

```
    for(j = 1; j <= i; j++)
```

```
        if(HT[j].parent == 0 && HT[j].weight < Wmin){
```

```
            Wmin = HT[j].weight;
```

```
            *s1 = j;
```

```
        }//if
```

```
    //寻找第二小的值
```

```
    Wmin = m + 1;
```

```

for(j = 1; j <= i; j++)
    if(HT[j].parent == 0 && j != *s1 && HT[j].weight < Wmin){
        //此处不加<=号是因为如果碰到两个权值相等的第二小，我们选择的是序号较小的那个
        Wmin = HT[j].weight;
        *s2 = j;
    }//if

if(*s1 > *s2){//把序号小的靠前
    j = *s1;
    *s1 = *s2;
    *s2 = j;
} //if
} //Select

```

```

//设置一个哈夫曼树结点
void SetHTNode(HuffmanTree *T, int w, int p, int l, int r){
    T->weight = w;
    T->parent = p;
    T->lchild = l;
    T->rchild = r;
} //SetHTNode

```

//w 存放 n 个字符的权值（均>0），构造哈夫曼树 HT，并求出 n 个字符的哈夫曼编码 HC。

```

void HuffmanCoding(HuffmanTree *HT, HuffmanCode HC, Letter *w, int n){
    int i, j; //用于循环
    int *s1, *s2; //用于寻找权值最小的两个结点的序号
    s1 = (int*)malloc(sizeof(int));
    s2 = (int*)malloc(sizeof(int));
    int m;

    HuffmanTree *p = NULL;

    if(n <= 1)
        exit(0);
    m = 2 * n - 1;

    //初始化哈夫曼树
    for(p = HT + 1, i = 1; i <= n; ++i, ++p, ++w)
        SetHTNode(p, w->wei, 0, 0, 0);
    for(; i <= m; ++i, ++p)
        SetHTNode(p, 0, 0, 0, 0);

    //建哈夫曼树
    for(i = n + 1; i <= m; ++i){

```

```

        //在 HT[1..i-1]选择 parent 为 0 且 weight 最小的两个结点，其序号分别为 s1 和 s2
        Select(HT, i - 1, s1, s2);
        HT[*s1].parent = i;
        HT[*s2].parent = i;
        HT[i].lchild = *s1;
        HT[i].rchild = *s2;
        HT[i].weight = HT[*s1].weight + HT[*s2].weight;
    }//for

/*打印哈夫曼树表
    printf("i weight parent lchild rchild\n");
    for(i = 1; i <= m; i++)
        printf("%-2d%5d%6d%7d%8d\n",i,HT[i].weight,HT[i].parent,HT[i].lchild,HT[i].rchild);
*/

//从叶子到根逆向求每个字符的哈夫曼编码
int c, f, start;//中间变量
char *cd = NULL;
cd = (char*)malloc(n * sizeof(char));//分配求编码的工作空间
cd[n - 1] = '\0';//编码结束符
for(i = 1; i <= n; ++i){//逐个字符求哈夫曼编码
    start = n - 1;//编码结束符位置
    for(c = i, f = HT[i].parent; f != 0; c = f, f = HT[f].parent)//从叶子到根逆向求编码
        if(HT[f].lchild == c)
            cd[--start] = '0';
        else
            cd[--start] = '1';
    HC[i] = (char*)malloc((n - start) * sizeof(char));//为第 i 个字符编码分配空间
    strcpy(HC[i], &cd[start]);//从 cd 复制编码（串）到 HC
}

//释放工作空间
free(cd);
free(s1);
free(s2);
}

//输入字符串,返回字符串的长度
int InputWord(char *str, int len){
    char ch;
    int count = 0;//实际字符数
    while((ch = getchar()) != '\n'){
        count++;
        if(count >= len)

```

```

        str = (char*)realloc(str, sizeof(char) * (len++));
        str[count - 1] = ch;
    }//while
    str[count] = '\0';

    return count;
} //InputWord

//统计部分，计算各字母权值,返回字母（即权值）的个数
int CountWeight(Letter *let, char *str, int count, int n){
    int i, j, num = 0; //n 为默认不同字母的个数的初始值，num 为实际不同字母的个数
    for(i = 0; i < n; i++)
        let[i].wei = 0;
    for(i = 0; i < count; i++){
        if(('a' <= str[i] && str[i] <= 'z') || (str[i] == ' ') || ('A' <= str[i] && str[i] <= 'Z')){
            //考虑到小写字母出现的频率普遍比大写高，为了提高效率，所以先判断是否小写再判断是否大
写
            for(j = 0; j < num; j++)
                if(let[j].ch == str[i]){
                    let[j].wei++;
                    break;
                } //if
            if(j == num){
                let[j].ch = str[i];
                let[j].wei++;
                num++;
            } //if
        } //if(a~z || A~Z)
    } //for
    if(num < n) //节省内存空间
        let = (Letter*)realloc(let, sizeof(Letter) * (num));
    return num;
} //CountWeight

//输入一段 100—200 字的英文短文，存入一文件 a 中。
void WriteText(char *str, int len){
    FILE *fp1 = NULL;
    if((fp1 = fopen("a.txt", "wb")) == NULL){
        printf("cannot open file\n");
        return;
    }
    if(fwrite(str, sizeof(char) * len, 1, fp1) != 1)
        printf("file write error\n");
    fclose(fp1);
}

```

```
}//WriteText
```

//用每个字母编码对原短文进行编码，码文存入文件 b 中。

```
int WriteCode(char *str, Letter *let, HuffmanCode HC, int num, int len){
    FILE *fp2 = NULL;
    int i,j;
    int n = 0;//文件中的编码的长度
    if((fp2 = fopen("b.txt", "wb")) == NULL){
        printf("cannot open file\n");
        return;
    }//if
    for(i = 0; i < len; i++)
        if(('a' <= str[i] && str[i] <= 'z') || (str[i] == ' ') || ('A' <= str[i] && str[i] <= 'Z'))
            for(j = 0; j < num; j++)
                if(str[i] == let[j].ch){
                    if(fputs(HC[j + 1],fp2) != 0)
                        printf("file write error\n");
                    else
                        n += strlen(HC[j + 1]);
                }//if

    fclose(fp2);
    return n + 1;//算上'\0'
}//WriteCode
```

//从文件 b 中读出编码

```
void ReadCode(char *s, int n){
    FILE *fp2;
    if((fp2 = fopen("b.txt","rb")) == NULL){
        printf("cannot open file\n");
        return;
    }//if
    fgets(s,n,fp2);//函数默认读入 n-1 个字符，最后自动添加'\0'，n 已经预留了
    fclose(fp2);
}//ReadCode
```

//用 Huffman 树对 b 中码文进行译码，结果存入文件 c 中

```
void TranslateCode(char *s, Letter *let, HuffmanTree *HT, int num){
    FILE *fp3 = NULL;
    if((fp3 = fopen("c.txt", "wb")) == NULL){
        printf("cannot open file\n");
        return;
    }//if
```



```

int i, j = 2 * num - 1;
for(i = 0; s[i] != '\0'; i++){
    //思路：从哈夫曼树的根节点开始，根据读取的字符寻找下一个结点，当当前结点没有左右孩子，即找到叶子结点，存入文件
    if(s[i]=='0')
        j = HT[j].lchild;
    else
        j = HT[j].rchild;

    if(HT[j].lchild == 0 && HT[j].rchild == 0)
        if(fwrite(let + j - 1, sizeof(char), 1, fp3) != 1)
            printf("file write error\n");
        else
            j = 2 * num - 1;
} //for
fclose(fp3);
} //TranslateCode

int main(){
    int len = 100; //英文短文长度的初始值
    int num = 53; //权值个数（53为初始值）
    int i;
    char *str = (char*)malloc(len * sizeof(char)); //存放输入的英文短文
    Letter *let = (Letter*)malloc(num * sizeof(Letter)); //存放各字母的权值

    HuffmanTree *HT;
    HuffmanCode HC;

    printf("Please input a sentence in English:\n");
    len = InputWord(str, len); //输入英文短文并返回其的长度
    num = CountWeight(let, str, len, num); //计算英文短文中各字母的出现次数（即权值）并返回权值的个数
    while(len == 1 || num == 1){
        printf("Input Error! Please input >= 1 char!\n");
        len = InputWord(str, len);
        num = CountWeight(let, str, len, num);
    }

    WriteText(str, len); //存入文件 a

    HC = (HuffmanCode)malloc((num + 1) * sizeof(char*)); //分配 n 个字符编码的头指针向量
    HT = (HuffmanTree*)malloc((2 * num) * sizeof(HTNode)); //0 号单元未用

    HuffmanCoding(HT, HC, let, num); //对各字母进行哈夫曼编码

```

```

//输出各字母权值和编码
printf("\nchar num  code\n");
for(i = 0; i < num; i++){
    printf("%2c  :%2d  ",let[i].ch,let[i].wei);
    puts(HC[i + 1]);
}

int n = WriteCode(str, let, HC, num, len); //将原文编码写入文件 b，并返回编码的长度（已经算上
'\0'）
char s[n];
ReadCode(s, n); //从文件 b 中读出编码
TranslateCode(s, let, HT, num); //用 Haffman 树对 b 中码文进行译码，结果存入文件 c 中

//释放空间
free(str);
free(let);
free(HC);
free(HT);
return 0;
}

```