

Huffman 编码压缩/解压器实验报告



实验题目：Huffman 编码压缩/解压器

学生姓名：李骋昊

学生学号：PB20081583

完成时间：2021年12月15日

Huffman 编码压缩/解压器实验报告

实验目的

实验要求

实现功能

输入与输出

设计思路

总体思路

哈夫曼编码与哈夫曼树的生成

哈夫曼编码的建立

哈夫曼树的建立

压缩部分

解压部分

模块讲解

队列操作（来自课本）

哈夫曼树建立

解压缩操作

具体操作

关键代码讲解及其调试分析

队列部分

哈夫曼树建立

FindMinTree

SortTree

HuffmanNode(重点部分)

CreateHuffManTree（重点部分）

解压缩操作

GetCode

EnCode

ReadCode

GetLength

具体操作

compress（）

uncompress（）

代码测试

总结

使用到的数据结构

实验思考

附录（文件内容）>

实验目的

实验要求

基于 Huffman 编码实现一个压缩器和解压缩器（其中 Huffman 编码以字节作为统计和编码的基本符号单元），使其可以对任意的文件进行压缩和解压缩操作。针对编译生成的程序，要求压缩和解压缩部分可以分别独立运行。

实现功能

基础部分：实现了对当前目录下指定文件的压缩与解压。

自定义加分项：完成了对压缩率的测定以及哈夫曼编码的具现。

输入与输出

输入：输入你所需要的功能，然后选定目标文件进行执行。

输出：如果是压缩，输出你的文件形成的huffman编码，同时对压缩率进行计算并生成你的目标文件；如果是解压缩则会直接生成你的目标文件。

设计思路

总体思路

本次实验使用了两种数据结构，队列与树。树结构用于构建哈夫曼树，存储所需的数据。队列则是读取、写入编码（利用1字节总是8位的特点设计队长）。

```
//这里的ELEMENTYPE是unsigned char
//这里队长选用这种类型其实是为了方便计算空间（1bytes）以及对之后数据的将错就错
typedef struct
{
    ELEMENTYPE ch;//字符
    long weight;//权值
    int parent,lchild,rchild;//双亲和孩子节点
}HMTNode,*HMTTree;//哈夫曼树结构

typedef struct
{
    int status;//队伍状态
    int front,rear;
    ELEMENTYPE lenth;//队长
    char elem[MAXSIZE];//存入字符
}*queue,QNode;
```

在文件头我们依次写入文件长，读取编码最后不足8bit的位数（只用于中间过程计算方便，后同），叶子数，最长最短编码数，哈夫曼编码数，写入后不足8bit的位数（只用于中间过程计算方便）。

文件主体部分我们写入字符对应的编码及其对应的频率以及整个哈夫曼编码。

在解压时只需要将上述信息读取后逆序求解即可。

```
rewind(newfile);
fseek(newfile, sizeof(long)+sizeof(ELEMENTYPE), SEEK_SET);//开头留出空间
fwrite(&leaf, sizeof(int), 1, newfile);//叶子数
fwrite(&MaxLenth, sizeof(ELEMENTYPE), 1, newfile); //最长码串长度
fwrite(&MinLenth, sizeof(ELEMENTYPE), 1, newfile); //最短码串长度
fwrite(&codes, sizeof(int), 1, newfile); //哈夫曼编码数
fwrite(&lastlenth, sizeof(ELEMENTYPE), 1, newfile); //写入编码剩下的字长
```

.....

```
rewind(newfile);
fwrite(&Lenth, sizeof(long), 1, newfile);//压缩后文件长度
fwrite(&lastlenth, sizeof(ELEMENTYPE), 1, newfile);//读取编码最后不足8bit的位数
```

哈夫曼编码与哈夫曼树的生成

哈夫曼编码的建立

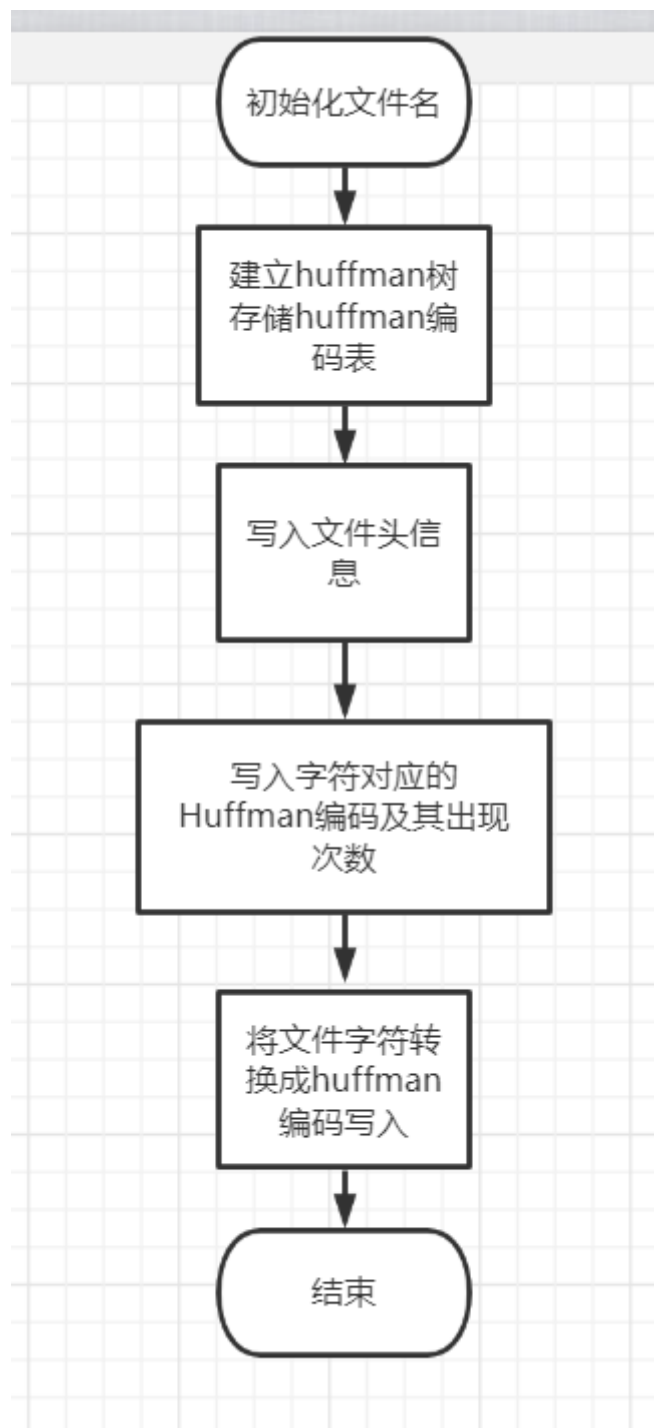
1. 根据叶子数申请空间 (leaf+1)
2. 根据哈夫曼树的左右孩子和双亲关系标记为“0”、“1”编码。
3. 返回得到的编码表，数据类型为char**

哈夫曼树的建立

1. 建立一个大小为512*unsigned char的树结构（叶子最多有256个，而节点最多有512-1个，0号作为头节点）
2. 初始化所有数据为0.
3. 读取目标文件，填入从文件中得到的字符所对应的频率（这是叶子节点，存储在1到256的空间内）
4. 对此时的树进行排序
5. 按照每次取最小两节点的原则建树。
6. 返回此huffman tree

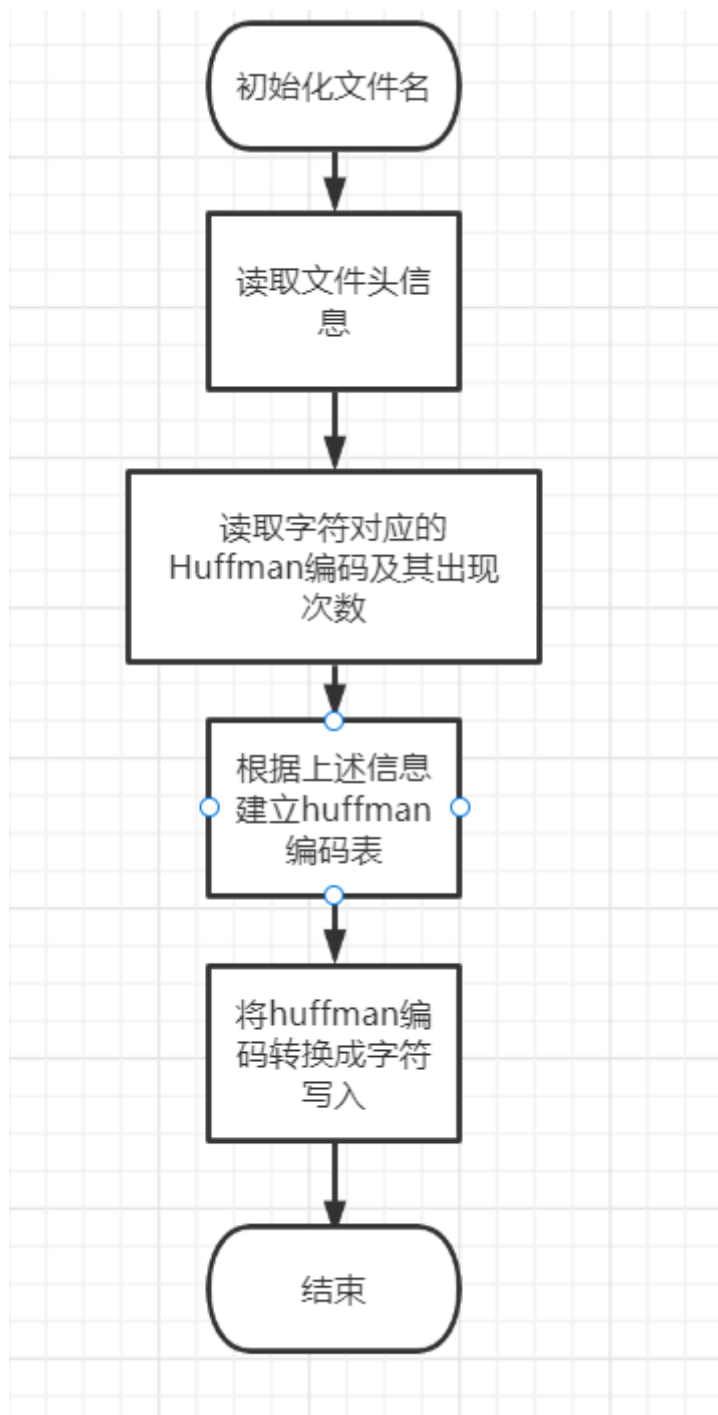
压缩部分

这里通过思维导图进行一个简单的描述。



解压部分

这里通过思维导图进行一个简单的描述。



模块讲解

主要有4部分代码（具体将在下一部分进行综合分析）

队列操作（来自课本）

```
int InitQueue(queue q)//创建空队，成功返回1

int EnQueue(queue q,char a)//入队操作，成功返回1

int DeQueue(queue q,char*a)//出队操作，返回队的状态
```

哈夫曼树建立

```
int FindMinTree(HMTree T,int n)//找到树最小节点

int SortHMT(HMTree T)//叶子节点排序，返回叶子数

char **HuffmanNode(HMTree T,int leaf)//生成huffman编码表

HMTree CreateHuffManTree(FILE* fp,long* FileLenth,int*leaf)//建立Huffman树
```

解压缩操作

```
ELEMENTYPE GetCode(queue Q)//生成编码

int EnCode(FILE *fp,queue Q,int leaf,char** text,ELEMENTYPE *lenth)
//text[]是每个叶子节点对应的编码，fp是对应文件，Q是存储队列，leaf是叶子节点数目
//功能是写入编码到新的文件中

int ReadCode(queue Q,ELEMENTYPE ch)
//读取编码的“译码操作”，作用是将压缩包中的字符转换为huffman编码，存入队列

int GetLength(HMTree T,ELEMENTYPE* MaxLenth,char**List,ELEMENTYPE* MinLenth,int
leaf,FILE *fp)
//得到最长和最短编码长度，并将字符和对应的编码长度写进文件
```

具体操作

```
void compress()//压缩

void uncompress()//解压

int main() {
    int sw;

    printf("          WELCOME TO USING MY PROGRAM          \n");
    while (1)
    {
        printf("          PLEACE TELL ME ABOUT YOUR CHOICE          \n");
        printf("          1:          COMPRASS          \n");
        printf("          2:          UNCOMPRASS          \n");
        printf("          else:          EXIT          \n");
        printf("          NOW YOU COULD MAKE YOUR CHOICE          \n");

        scanf("%d",&sw);

        switch (sw)
        {
            case 1:
                compress();
                break;

            case 2:
                uncompress();
                break;

            default:
```



```
        return 0;
    }
}

return 0;
} //主函数
```

关键代码讲解及其调试分析

队列部分

```
//由于这来自于我们的教材，此处只选择DeQueue操作来讲解自定义的标识符

int InitQueue(queue q)//创建空队，成功返回1

int EnQueue(queue q,char a)//入队操作，成功返回1

int DeQueue(queue q,char*a){
    if(q->status==EMPTY)                //EMPTY代表空队
        return ERROR;                  //ERROR代表失败
    *a=q->elem[q->front];
    q->front=(q->front+1)%MAXSIZE;        //考虑到满队列的情况，我使用了循环的思想，
MAXSIZE被我定义为200
    q->lenth--;
    if(q->lenth==0)
        q->status=EMPTY;
    else
        q->status=UNEMPTY;              //UNEMPTY代表队列不空，FULL代表满了
    return OK;
} //出队操作，返回队首字符
```

哈夫曼树建立

FindMinTree

```
int FindMinTree(HMTree T,int n){
    int temp,i;
    long weight;

    for ( i = 0; i <= n; i++)
    {
        if (T[i].parent ==0){
            weight=T[i].weight;
            temp=i;
            break;
        } //找到未选择节点
    }
    for (i++; i <= n; i++) {
        if (0 == T[i].parent && T[i].weight < weight) {
            weight = T[i].weight;
            temp = i;
        }
    }
    return temp;
} //找到（在0至n）权值最小的HUFFMAN树，返回值是权值最小的节点（这要求HuffmanTree是有序的）
```

这是对树的遍历，时间复杂度为 $O(n)$

SortTree

```
int SortHMT(HMTTree T){
    int i,j,k;
    HMTNode temp;
    for (i = SIZE-1; i >= 0; i--) {
        for (j = 0; j < i; j++)
            if (T[j].weight < T[j + 1].weight) {
                temp = T[j];
                T[j] = T[j + 1];
                T[j + 1] = temp;
            }
    }
    for (i = 0; i < SIZE; i++)
        if (0 == T[i].weight)
            return i; //有权值代表它是叶子节点，无权值代表它所对应的字符并没有出现
    return i;
} //从大到小排序,返回值是叶子节点的数量
```

冒泡排序，时间复杂度为 $O(n^2)$

HuffmanNode(重点部分)

```
char **HuffmanNode(HMTTree T,int leaf){
    for (i = 0; i < leaf; i++) {
        start=leaf;end=i;
        //开始构建该字符所对应的哈夫曼编码
        for(papa=T[i].parent;papa!=0;papa=T[papa].parent){
            //papa是双亲节点
            if (T[papa].lchild==end)
            {
                ctemp[--start]='0'; //左孩子标记为0
            }
            else {
                ctemp[--start] = '1'; //右孩子标记为1
            }
            end=papa; //更新上一节点
        }
        //开始构建编码表
        temp[i]=(char*)malloc((leaf-start)*sizeof(char)); //有几位分配多少个空间
        strcpy(temp[i],&ctemp[start]); //从第start开始复制（因为前面都是空的）
        printf("%3d号 %3c 编码长度为:%2d;编码:%s\n", T[i].ch, T[i].ch, leaf - start,
            &ctemp[start]);
    }
    free(ctemp);
    return temp;
} //leaf是叶子节点的个数，也就是sorthuffman的值
```

CreateHuffManTree (重点部分)

```
HMTTree CreateHuffManTree(FILE* fp,long* FileLenth,int*leaf){
    HMTTree TempTree=NULL;
    TempTree=(HMTTree)malloc(2*SIZE*sizeof(HMTNode)); //一棵树的极限节点数为2*size-1
    int i,j,k;
    int n; //n是huffman的节点总数
    ELEMENTYPE word; //所读取的字符
```

```

//省略掉了树的初始化部分
*leaf=SortHMT(TempTree); //出现的字符种类,也就是huffmantree的叶子节点数
n=2*(*leaf)-1; //节点总数
if(n-1==0){
    TempTree[0].parent=1;
    return TempTree;
} //只存在根节点
else if(*leaf)
    return NULL; //空树
for ( i = n-1; i >=0; i-- )
{
    TempTree[i].parent=0;
    TempTree[i].lchild=TempTree[i].rchild=0;
} //此时TempTree已经排序, 数组前n个元素都是含有字符的非空节点。我们将它初始化为双亲和左右孩子都为0的节点。
for ( i = *leaf; i < n; i++ ) {
    //找到最小节点
    j=FindMinTree(TempTree,i-1);
    TempTree[j].parent = i;
    TempTree[i].lchild = j;
    //找到次小节点
    k=FindMinTree(TempTree, i-1);
    TempTree[k].parent = i;
    TempTree[i].rchild = k;
    TempTree[i].weight = TempTree[j].weight+TempTree[k].weight;
} //构建huffman树
return TempTree;
} //fp是目标文件

```

时间复杂度为 $O(n^2)$, 空间复杂度为 $O(n)$

解压缩操作

GetCode

见注释

```

ELEMENTYPE GetCode(queue Q){
    char c;
    ELEMENTYPE code=0;
    for (int i = 0; i < 8; i++){
        if(DeQueue(Q,&c)!=EMPTY){
            if (c=='0')
                code=code<<1; //左移一位
            else
                code=(code << 1) | 1; //左移一位然后补1
        }
        else break;
    }
    return code;
} //代码的核心思想就是8个8个读取01编码, 然后以unsigned char形式输出 (1bytes)

```

EnCode

```
int EnCode(FILE *fp, queue Q, int leaf, char** text, ELEMENTYPE *lenth){
    char* wTemp;
    int i;
    int count=0;
    ELEMENTYPE code, j;
    for ( i = 0; i < leaf; i++)
    {
        for(wTemp=text[i]; *wTemp!='\0'; wTemp++)
            EnQueue(Q, *wTemp);

        while (Q->lenth>8){
            code=GetCode(Q); //读字符
            fputc(code, fp);
            count++;
        }
    } //编码
    //这是对不足8bit情况分析，由于过分雷同，之后不会单独列出
    i=8-Q->lenth;
    code = GetCode(Q);
    *lenth=Q->lenth; //lenth是余下的编码长
    for(j=0; j<i; j++)
        code=code<<1;
    //printf(" %u ", code);
    fputc(code, fp);
    count++;
    InitQueue(Q);
    //printf("\n");
    return count;
} //text[]是每个叶子节点对应的编码，fp是对应文件，Q是存储队列，leaf是叶子节点数目
```

ReadCode

```
int ReadCode(queue Q, ELEMENTYPE ch){
    int i;
    ELEMENTYPE Ctemp;
    for (i = 0; i < 8; i++) {
        Ctemp = ch << i;
        Ctemp = Ctemp >> 7;
        //将想要位数通过位运算从高到低获取入队
        if (Ctemp==1)
            EnQueue(Q, '1');
        else
            EnQueue(Q, '0');
    }
    return OK;
} //译码
```

GetLength

```
int GetLength(HMTTree T, ELEMENTYPE* MaxLenth, char**List, ELEMENTYPE* MinLenth, int
leaf, FILE *fp){
    int i;
    ELEMENTYPE temp;
    *MaxLenth=*MinLenth=strlen(List[0]); //初始化最大最小的编码
```

```

    for (i = 0; i < leaf; i++){
        temp=strlen(List[i]);
        fwrite(&T[i].ch, sizeof(ELEMENTYPE), 1, fp);
        fwrite(&temp, sizeof(ELEMENTYPE), 1, fp); //字符和对应的编码长度写进文件
        if (temp>*MaxLenth)
            *MaxLenth = temp;
        else if(temp<*MinLenth)
            *MinLenth = temp;
    }
    return OK;
} //求编码的上下限长度，将字符和对应的编码长度写进文件

```

具体操作

compress ()

此处将几个重要操作列出即可。由于初始化和文件头操作已经列出，这里不多赘述。

首先进行GetLength和EnCode。之后就是压缩的主体部分。

```

while (count<FileLenth){
    ch=fgetc(originfile);
    count++; //count是统计字符读取是否见底
    for (cp=Dictionary[ch]; *cp!='\0'; cp++)
        EnQueue(Q,*cp);
    while (Q->lenth > 8) // printf("OutQueue: ");
    {
        bit = GetCode(Q); //出队8个元素,合成一个字节
        fputc(bit, newfile); //fwrite(&bits,1,1,compressFile);
        Lenth++;
    }
}

```

由于字符读取不一定是整8个，所以会有剩下的。

```

lastlenth = Q->lenth; //还剩多少字节

bit = GetCode(Q);
for (i = 0; i < 8-Q->lenth; i++)
    bit = bit << 1;
//printf(" %u ",bit);
fwrite(&bit, sizeof(ELEMENTYPE), 1, newfile); //把最后的字符写上去
Lenth++;

```

这就是关键的压缩操作。

整个函数时间复杂度为 $O(n^2)$ ，空间复杂度为 $O(n)$

uncompress ()

与上文一致，这里的头文件读取我们不多赘述。

首先我们需要读取GetLength和EnCode所写入的内容，计入表中

```

Q=(queue)malloc(sizeof(QNode));//初始化
InitQueue(Q);
words=(ELEMENTYPE*)malloc(leaf*sizeof(ELEMENTYPE));
wordslenth=(ELEMENTYPE*)malloc(leaf*sizeof(ELEMENTYPE));
for ( i = 0; i < leaf; i++)
{
    fread(&words[i],sizeof(ELEMENTYPE),1,originfile);
    fread(&wordslenth[i],sizeof(ELEMENTYPE),1,originfile);
    //printf(" %u %u \n",words[i],wordslenth[i]);
}
//读出每个字符以及它们对应的编码长度
List=(char**)malloc((leaf) *sizeof(char*));

```

```

for (i =j= 0; i < codes;i++){
    ch=fgetc(originfile);
    //printf(" %u ",ch);
    ReadCode(Q,ch);
    while (Q->lenth>maxlenth){
        text=List[j]=(char*)malloc(1+wordslenth[j]);
        for (ch = 0; ch < wordslenth[j]; ch++) {
            DeQueue(Q,&c);
            *text++ = c;
        }
        *text = '\0';
        j++;
    }
}

```

然后我们对新文件进行写入。

```

minlenth--;
while (curr_lenth<lenth){
    ch=fgetc(originfile);//printf(" %u ",ch);
    ReadCode(Q,ch);
    curr_lenth++;
    while (Q->lenth>maxlenth) {
        for ( i = 0; i < minlenth; i++)
        {
            DeQueue(Q,&c);
            content[i]=c;
        }//出来一个不完整字符编码，minlenth减1保证下一个for循环可以正常进行
        for (; i < maxlenth;i++) {
            DeQueue(Q,&c);
            content[i] = c;
            content[i+1] = '\0';
            for(j=0;j<leaf;j++){
                if (i+1 == wordslenth[j] && 0 == strcmp(List[j],content)){
                    ch = words[j];
                    //printf(" %u ",ch);
                    fputc(ch,newfile);//判断是否是字符
                    break;
                }
            }
            if(j<leaf)
                break;
        }
    }
}
}

```





```
}
```

通过将01编码与编码表比对得到我们想要的文件。

注意：上述操作均省略了对小于8情况下的修正，因为在compress和encode中都有过描述。

代码测试


这里选用一张图片和一份压缩包进行测试。初始状态如下。

 5.zip	2021/12/18 20:01
 huffman.c	2021/12/23 10:12
 huffman.exe	2021/12/23 10:13
 tina.jpg	2021/12/19 0:16









其中压缩包来自于助教的检查样例，tina.jpg是一张可爱的壁纸。



我们分别把它们压缩命名为1和2如下。

 1	2021/12/23 16:39	文件	31,175 KB
 2	2021/12/23 16:39	文件	93 KB

然后解压得到1.zip和Diana.jpg

 1	2021/12/23 16:39	文件	31,175 KB
 1.zip	2021/12/23 16:41	压缩(zipped)文件...	31,175 KB
 2	2021/12/23 16:39	文件	93 KB
 5.zip	2021/12/18 20:01	压缩(zipped)文件...	31,175 KB
 Diana.jpg	2021/12/23 16:41	JPG 文件	92 KB
 huffman.c	2021/12/23 10:12	C 源文件	26 KB
 huffman.exe	2021/12/23 10:13	应用程序	71 KB
 tina.jpg	2021/12/19 0:16	JPG 文件	92 KB

打开1.zip

1.文本测试.txt	文本文档	3 KB	否	131 KB	98%	2018/11/18 18:50
2.图片测试.jpg	JPG 文件	1,512 KB	否	1,628 KB	8%	2018/11/18 18:54
3.音频测试.aif	AIFF 格式声音	13,072 KB	否	14,737 KB	12%	2018/11/18 18:58
4.视频测试.flv	FLV 文件	16,588 KB	否	17,018 KB	3%	2018/11/18 18:57

1.文本测试.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

《数据结构》实验课内容和要求

实验一、线性表的应用：稀疏一元多项式运

实验目的:

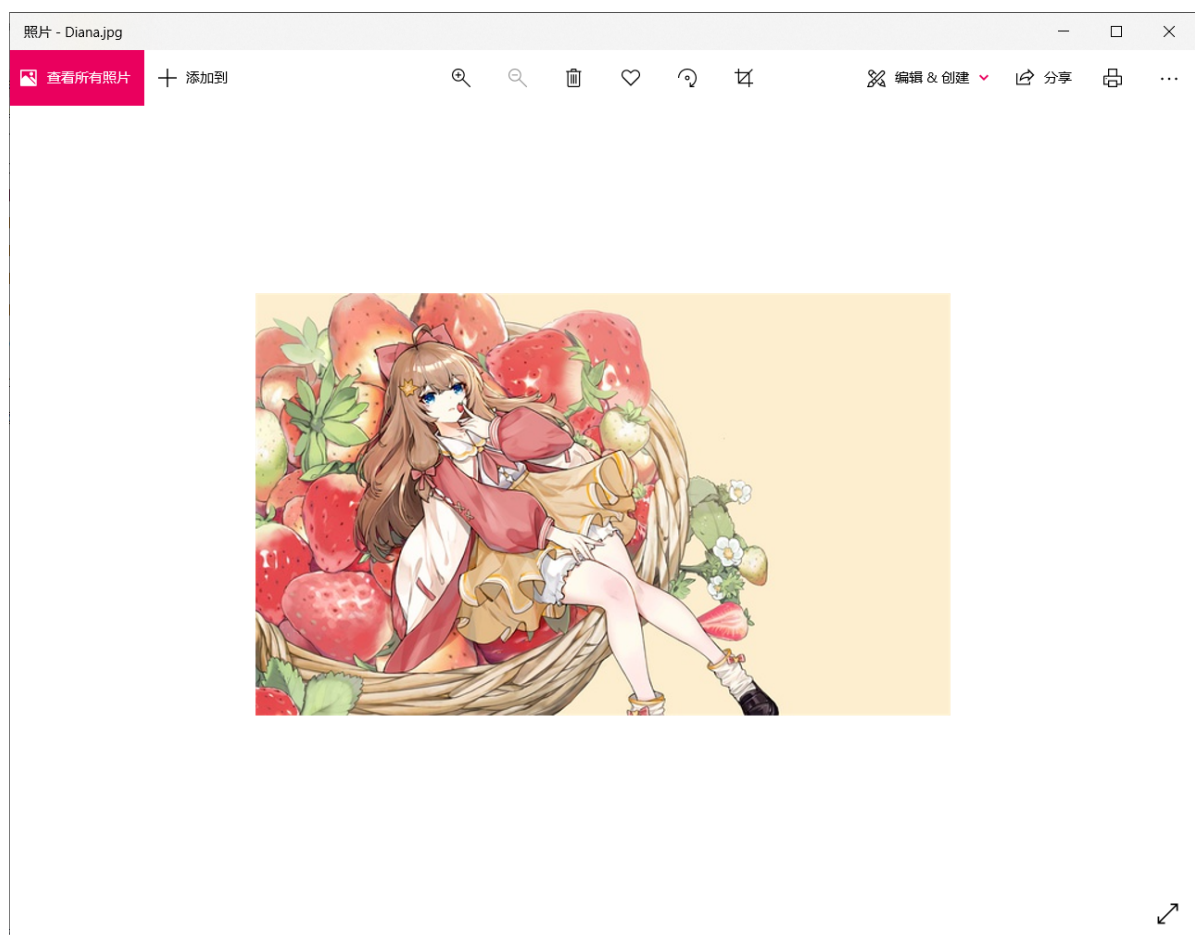
- ? 熟练掌握指针和链表操作的基本功
- ? 熟练掌握数组操作的基本功
- ? 模块化程序设计 (程序的分层结
- ? 人机交互界面设计 (界面美观, 1
- ? 源程序的书写风格 (缩进式, 加
- ? 对程序健壮性的处理
- ? 程序的调试技术训练(debug方法
- ? 时空效率

实验学时:

12学时 (第1,2,3次实验)

实验内容: (部分内容参考习题库21和15)

打开Diana.jpg



均无误。

总结

使用到的数据结构

本次实验使用了二叉树和队两种数据结构类型。

实验思考

有的函数存在一定的纰漏，所以在解压函数里通过了一个if语句来强行解决bug(此bug在检查时才发现)

由于使用纯C语言，C++的类无法运用，也是代码过长的原因之一。

通过做这次实验加深了对于课本前几章所学内容的印象与这几种基本数据结构类型的应用。同时通过对较长代码的debug，也加深了我对养成良好编程习惯的深刻认识。需要进行适当的注释以及恰当的缩进，同时粗糙的复制粘贴可能是许多bug的成因。

附录（文件内容）

>

- 源代码huffman.c
- 实验报告