

《数据结构与算法分析》

实验报告

实验名称	基于哈夫曼树的数据压缩算法
姓名	曾庆文
学号	23060218
院系	自动化学院
专业	人工智能
班级	23061011
实验时间	2024.5

一、实验目的

1. 掌握哈夫曼树的构造算法。
2. 掌握哈夫曼编码的构造算法。

二、实验设备与环境

编辑器：Visual Studio Code

编译器：gcc

三、实验内容与结果

基本操作的输入输出结果如下：

1. 测试样例：

```
aaaaaaabbbbbccdddd
```

输入样例1

```
a:7 b:5 c:2 d:4
```

```
1 7 7 0 0
```

```
2 5 6 0 0
```

```
3 2 5 0 0
```

```
4 4 5 0 0
```

```
5 6 6 3 4
```

```
6 11 7 2 5
```

```
7 18 0 1 6
```

```
a:0 b:10 c:110 d:111
```

```
0000000101010101011011011111111111
```

```
aaaaaaabbbbbccdddd
```

输出结果1

```
aabccc
```

输入样例2

```
a:2 b:1 c:3
```

```
1 2 4 0 0
```

```
2 1 4 0 0
```

```
3 3 5 0 0
```

```
4 3 5 2 1
```

```
5 6 0 3 4
```

```
a:11 b:10 c:0
```

```
111110000
```

```
aabccc
```

输出结果2

```
0
```

"0" 终止程序

请按任意键继续 . . . |

2. 鲁棒性测试:

2.1 不按照字母表顺序的输入:

输入	输出
aws ^l	a:1 l:1 s:1 w:1 1 1 5 0 0 2 1 5 0 0 3 1 6 0 0 4 1 6 0 0 5 2 7 1 2 6 2 7 3 4 7 4 0 5 6 a:00 l:01 s:10 w:11 00111001 aws ^l

2.2 多个字符不按字母表顺序的输入:

输入	输出
yyds	d:1 s:1 y:2 1 1 4 0 0 2 1 4 0 0 3 2 5 0 0 4 2 5 1 2 5 4 0 3 4 d:10 s:11 y:0 001011 yyds

实验全部代码如下：

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. #include<string.h>
4. #define MAXLEN 100
5.
6. //哈夫曼树的存储表示
7. typedef struct
8. {
9.     int weight;
10.    int parent,lchild,rchild;
11. }HTNode,*HuffmanTree;
12.
13. //哈夫曼编码表的存储表示
14. typedef char **HuffmanCode;
15.
16. //字符类别和出现频率
17. typedef struct
18. {
19.     char ch;
20.     int numOfLetter;
21. }Letter;
22.
23. //四个函数声明
24. void Select(HuffmanTree HT,int n,int *s1,int *s2);
25. void CreateHuffmanTree(HuffmanTree *HT,int n,Letter *letter
    );
26. void CreateHuffmanCode(HuffmanTree HT,HuffmanCode *HC,int n
    );
27. void Statistic(char *input,int *n,int *frequency);
28. int Conserve(Letter *letter,char ch,int n);
29.
30. //主函数
31. int main()
32. {
33.     char input[MAXLEN][MAXLEN];
34.     for(int i=0;i<=MAXLEN;i++)
35.     {
36.         //1. 输入并处理输入结果
37.         scanf("%s",input[i]);
38.         if(input[i][0]=='\0') //退
            出循环条件
39.             break;
```

```

40.         int n=0;                                //字
           符类别个数
41.         int length=strlen(input[i]);            //每
           行字符串的字符个数
42.         int frequency[26]={0};                  //每
           个字符出现频率
43.         Statistic(input[i],&n,frequency);         //统
           计字符类别个数和出现频率
44.         Letter letter[n+1];                      //存
           在的字符及其频率
45.         for(int i=0,j=1;i<26;i++)
46.         {
47.             if(frequency[i]!=0)                  //将
           存在的字符统计好并赋给 letter
48.             {
49.                 letter[j].ch='a'+i;
50.                 letter[j].numOfLetter=frequency[i];
51.                 j++;
52.             }
53.         }
54.
55.         //2. 构建哈夫曼树及其编码
56.         HuffmanTree HT;
57.         CreateHuffmanTree(&HT,n,letter);          //创
           建哈夫曼树
58.         HuffmanCode HC;
59.         CreateHuffmanCode(HT,&HC,n);              //构
           建哈夫曼编码
60.
61.         //3. 输出结果
62.         for(int i=1;i<=n;i++)
63.         {
64.             //输出存在的字符及其出现频率
65.             printf("%c:%d ",letter[i].ch,letter[i].numOfLet
           ter);
66.             if(i==n)
67.                 printf("\n");
68.         }
69.         for(int i=1;i<=2*n-1;i++)
70.         {
71.             //输出 HT 表终态
72.             printf("%d %d %d %d %d\n",i,HT[i].weight,HT[i].
           parent,HT[i].lchild,HT[i].rchild);
73.         }

```

```

74.         for(int i=1;i<=n;i++)
75.         {
76.             //输出每个字符的哈夫曼编码
77.             printf("%c:%s ",letter[i].ch,HC[i]);
78.             if(i==n)
79.                 printf("\n");
80.         }
81.         for(int j=0;j<length;j++)
82.         {
83.             //输出编码后的字符串
84.             printf("%s",HC[Conserve(letter,input[i][j],n)])
85.             ;
86.             if(j==length-1)
87.                 printf("\n");
88.             //输出解码后的字符串
89.             printf("%s\n",input[i]);
90.         }
91.     }
92.
93. //在 i-1 个结点中选择双亲域为0 且权值最小的两个结点
94. void Select(HuffmanTree HT,int n,int *s1,int *s2)
95. {
96.     int min=1000;
97.     for(int i=1;i<=n;i++)
98.         for(int j=1;j<=n;j++)
99.             if(HT[i].parent==0 && HT[j].parent==0 && i!=j)
100.                 if(HT[i].weight+HT[j].weight<min)
101.                 {
102.                     min=HT[i].weight+HT[j].weight;
103.                     *s1=i;
104.                     *s2=j;
105.                     if(HT[*s1].weight > HT[*s2].weight)
106.                         *s1^=*s2^=*s1^=*s2;
107.                     //防止 s1 所代表的结点的权值>s2 的
108.                     //交换数值
109.                 }
110. //构造哈夫曼树
111. void CreateHuffmanTree(HuffmanTree *HT,int n,Letter *letter)
112. {

```

```

113.     if(n<=1)
114.         exit(0);
115.     int m=2*n-1;
116.     *HT = (HuffmanTree)malloc((m + 1) * sizeof(HTNode));
117.     for(int i=1;i<=m;i++)
118.     {
119.         (*HT)[i].parent=0;
120.         (*HT)[i].lchild=0;
121.         (*HT)[i].rchild=0;
122.         if(i<n+1)
123.             (*HT)[i].weight=letter[i].numOfLetter;    //
            将统计好的字符频率赋值给HT 的权值
124.     }
125.     /*----- 初始化结束，开始创建哈夫曼树-----*/
126.     int s1,s2;
127.     for(int i=n+1;i<=m;i++)
128.     {
129.         Select(*HT,i-1,&s1,&s2);    //
            规定 s1 所代表的结点的权值<=s2 的
130.         (*HT)[s1].parent=i;
131.         (*HT)[s2].parent=i;
132.         (*HT)[i].lchild=s1;
133.         (*HT)[i].rchild=s2;
134.         (*HT)[i].weight=(*HT)[s1].weight+(*HT)[s2].weight
            ;
135.     }
136. }
137.
138. // 构造哈夫曼编码
139. void CreateHuffmanCode(HuffmanTree HT,HuffmanCode *HC,int
    n)
140. {
141.     *HC=(char **)malloc((n+1) * sizeof(char *));
            //分配 n+1 个 char* 的空间
142.     char *cd=(char *)malloc(n * sizeof(char));
            //分配 n 个 char 的空间
143.     cd[n-
        1]='\0';    //在 cd
            结尾添加结束符
144.     for(int i=1;i<=n;i++)
145.     {
146.         int start=n-
            1;    //因为要倒序给 cd 赋
            值，所以 start 初值为 n-1

```

```

147.         int c=i;
148.         int f=HT[i].parent;
           //f 代表结点 c 的双亲结点的序号
149.         while (f!=0)
150.         {
151.             start--;
152.             if(HT[f].lchild == c)
           //左结点
153.                 cd[start]='0';
154.             else
           //右结点
155.                 cd[start]='1';
156.             c=f;
157.             f=HT[f].parent;
158.         }
159.         (*HC)[i]=(char *)malloc((n-
           start) * sizeof(char)); //仅分配所需要的空间, 不浪费存储
160.         strcpy((*HC)[i],cd+start);
           //将求得的编码赋值到*HC 当前行
161.     }
162.     free(cd);
           //释放空间
163. }
164.
165. //统计字符类别个数和出现频率
166. void Statistic(char *input,int *n,int *frequency)
167. {
168.     (*n)=0;
169.     int ascii;
170.     for(int i=0;input[i]!='\0';i++)
171.     {
172.         ascii=input[i]-
           'a'; //当前字符与'a' 的距离
173.         frequency[ascii]++;
           //当前字符出现频率++
174.     }
175.     for(int i=0;i<26;i++)
176.         if(frequency[i]!=0)
177.             (*n)++;
           //出现的字符种类++
178. }
179.
180. //将此时处理的字符转化为该字符在 letter 数组中的下标
181. int Conserve(Letter *letter,char ch,int n)

```



```
182. {  
183.     for(int i=1;i<=n;i++)  
184.         if((letter+i)->ch == ch)  
185.             return i;  
186. }
```

四、实验总结及体会

本次实验对我个人来说有以下几个难点：

1. 输入数据的处理：由于输入的字符串可能出现多个字符类别、不同种类字符交替出现等情况，因此我定义了一个结构体数组，用来存放该行字符串中出现的字符及其出现次数，这样就保证了下面的哈夫曼树能够更加方便快捷地创建。

2. Select 函数的代码：该函数要求在 $i-1$ 个结点中选择双亲域为 0 且权值最小的两个结点。我使用的是双层 for 循环去遍历哈夫曼树，找出两个结点的下标。由于实验规定了 $s1 \leq s2$ ，因此为保证代码的严谨性，我又加了一段代码，并且使用异或交换变量，实现单行代码交换变量，提高了程序的整洁性。

```
1.         if(HT[*s1].weight > HT[*s2].weight)  
2.             *s1^=*s2^=*s1^=*s2;
```

3. CreateHuffmanCode 函数的存储空间分配：为了不浪费存储空间，要先每一个字符的编码，再根据编码长度申请相对应的空间。由于申请空间是对指针进行操作，因此要注意多级指针的处理。一开始我是对二级指针去申请空间，直到程序报错才让我意识到应该对一级指针去申请空间，才能得到正确结果。
4. 结果的正确输出：由于每一类字符在哈夫曼编码数组中都有一段编码，那么如何通过一个字符，反向找到该字符所对应的编码是输出的关键。这里我定义了一个 Conserve 函数，用来将当前处理的字符，并转化成该字符在编码数组中对应的下标，这样便成功实现了输出该字符对应的编码。
5. 代码的分块书写：我将本次实验的代码分成三个部分，从而更快速的完成代码。第一部分是输入数据的处理，将每行字符串的出现字符及其频率保存在结构体数组中；第二部分是哈夫曼树和哈夫曼编码的创建，将处理好的字符转化成编码；第三部分是结果输出，要注意输出的顺序和换行处理。