

**课程 实 验 报 告**

**课程名称： 数据结构上机实验报告**

**专业班级： 自实1901**

**学 号： U201916457**

**姓 名： 张皓然**

**指导教师： 朱明富**

**报告日期： 2020年11月20日**

**人工智能与自动化学院**

目录

[题目1 1](#_Toc56609936)

[题目 1](#_Toc56609937)

[设计思路 1](#_Toc56609938)

[基本思路 1](#_Toc56609939)

[相关数据结构的设计 1](#_Toc56609940)

[算法的简介 2](#_Toc56609941)

[函数流程图 3](#_Toc56609942)

[实验结果 5](#_Toc56609943)

[实验分析 5](#_Toc56609944)

[题目2 6](#_Toc56609945)

[题目 6](#_Toc56609946)

[设计思路 6](#_Toc56609947)

[基本思路 6](#_Toc56609948)

[相关数据结构的设计 6](#_Toc56609949)

[算法的简介 6](#_Toc56609950)

[函数流程图 7](#_Toc56609951)

[实验结果 8](#_Toc56609952)

[实验分析 8](#_Toc56609953)

[题目3 8](#_Toc56609954)

[题目 8](#_Toc56609955)

[设计思路 8](#_Toc56609956)

[基本思路 8](#_Toc56609957)

[相关数据结构的设计 9](#_Toc56609958)

[算法的简介 9](#_Toc56609959)

[函数流程图 10](#_Toc56609960)

[实验结果 16](#_Toc56609961)

[实验分析 18](#_Toc56609962)

[附录 19](#_Toc56609963)

[实验一代码 19](#_Toc56609964)

[实验二代码 23](#_Toc56609965)

[实验三代码 25](#_Toc56609966)

# 题目1

## 题目

走迷宫问题，利用栈的思想实现迷宫的求解。

## 设计思路

### 基本思路

走迷宫问题，主要需要解决三个问题。一，如何表示迷宫。二，如何得到解答迷宫路径。三，如何记录解答迷宫的路径。

对于第一个问题，我选择用二维数组来表示迷宫的信息。

|  |
| --- |
|  |
| 图1. 迷宫信息的表示 |

地图中1表示墙壁，0表示通路，3表示起点，2表示终点。其中数组每个元素表示地图的一个位置。例如，map[1][1]的值为3，表示地图上(1,1)这个点为起点。这种表示方法的优点是直观，易于理解。缺点是当迷宫中墙壁较多通路较少，或者通路较多墙壁较少时，这种迷宫的存储方法占用了更多的空间。

对于第二个问题，我使用类似栈的数据结构来实现迷宫的求解。将路径已有的点放进栈顶，如果走至死路，则将栈内的元素逐个抛出，相当于是原路返回，直至找到其他通路，重复类似的操作。最终走到终点时，栈内的元素实际就是迷宫的解答路径，至此，第三个问题也不攻自破了。

### 相关数据结构的设计

#### 迷宫信息的储存

虽上文已经提到，用二维数组来记录迷宫的信息。然而，二维数组中1，0可以表示位置能否通过，但却难以记录我们是否经过这个位置。于是我设计maze\_point结构，包含foot和way两个量。way储存该处能否通过的信息，foot储存该处是否已经走过。

|  |
| --- |
|  |
| 图2. maze\_point结构定义 |

#### 路径信息的储存

路径信息主要需要存储两方面的内容。一，当前路径的位置。二，来路的方向（方便在最后输出全部路径）。因此，我按照下述语句定义了stack结构。（仅指栈的一个元素）

|  |
| --- |
|  |
| 图3. stack结构定义 |

### 算法的简介

在初始化地图后，我们首先找到起点（maze[i][j].way=3）,随后，我们从这个点开始，进入一个while(1)循环开始寻路。cur\_way是存储当前路径的stack结构数组。cur\_y的值是当前位置的y坐标，cur\_x的值是当前位置的x坐标。

在每轮循环我们对上下左右四个方向做如下判断。对于上下两方向，我们先判断，如果朝该方向走，是否超出迷宫的范围，既cur\_x+1<=n2和cur\_x-1>=0，左右两方向类似判断。以向右为例，我们接着判断目前位置向右一格的迷宫情况，要求这一格可通过，且未曾走过（防止走回头路），即maze[cur\_x][cur\_y+1].way是否为1且maze[cur\_x][cur\_y+1].foot是否为0。如果以上判断均成立，则就向这个方向走一格。同时将新的一格入栈（记录cur\_way[top+1].x, cur\_way[top+1].y, cur\_way[top+1].pre\_dir的值），更新迷宫的信息（留下一个足迹，maze[cur\_x][cur\_y].foot=1）

|  |
| --- |
|  |
| 图4. 正常寻路算法 |

如果走至死路，则抛出一个元素，原路返回一格，并将这个位置用墙堵住，避免再次通过。

|  |
| --- |
|  |
| 图5. 原路返回 |

### 函数流程图

#### create\_maze函数

迷宫初始化

足迹初始化

foot全部定为0

地图初始化

way=1表示墙壁

way=0表示通路

way=3表示起点

way=2表示终点

#### solve\_maze函数

开始

逐行循环 i代表行数

逐列循环 j代表列数

maze[i][j].way是否为3

从此开始

确定起点

否

是

向左？

向右？

向下？

向上？

否

否

否

更新路径

是

是

是

是

原路返回一次

新位置为终点？

从栈中输出路径

是

否

## 实验结果

|  |
| --- |
|  |
| 图6. 程序运行结果 |

比照迷宫，寻找的路径没有问题，程序设计成功。

|  |
| --- |
|  |
| 图7. 迷宫 |

## 实验分析

设计的程序成功用栈的思想，寻找到了迷宫求解的路径。本程序的优点是可读性强，易于理解，思路简洁。缺点是因为思路简单，故而仅能找到路径，但却无法保证找到的路径是最优的路径。如图7所示，在一开始时，路径绕了一个无意义小弯。

# 题目2

## 题目

地图填色问题，用栈的思想解决地图的填色。

## 设计思路

### 基本思路

地图填色问题非常经典，我们可以用栈的方法解决它。当然，要用程序来解决这个问题，就需要首先解决这两个问题，一，如何表示地图上不同区域间的关系。二，如何确保相邻区域填充不同的颜色。

对于第一个问题，我选择用邻接矩阵来表示城市间的关系。若有n个城市，则构建一个n\*n的矩阵，其中第i行第j列的值，如果为1，则表示城市i与城市j相邻；如果为0，表示城市i与城市j不相邻。这种表示方法的优点是直观，简单，在表示较为丰富的城镇关系时充分利用空间。缺点是如果城镇关系较少，矩阵较稀疏，则会浪费许多空间。而且，i城镇与j城镇的关系实际在矩阵中出现了两次，这对计算机而言也是空间的浪费。

对于第二个问题，我选择构建一个堆栈。在为新城市填色时，先任意放入一个颜色，然后找到堆栈中与其相邻城市的颜色，如果颜色重复，就把栈顶颜色抛出，换成新的颜色，直至找到符合要求的颜色为止。

### 相关数据结构的设计

程序较简单，仅用普通的数组便可以完成栈的构建。

### 算法的简介

首先定义一个数组stack作为栈。在正式进行填色前，我们先对栈进行初始化。将stack的每一个元素都赋-1，表示该城市尚未着色。

|  |
| --- |
|  |
| 图8. 栈的初始化操作 |

对于每个新的城市，我们先随机给他一个颜色，然后进行判断。如果相邻城市和他撞色，则换下一种颜色，直至找到无撞色的颜色为止。

|  |
| --- |
|  |
| 图9. 着色过程代码 |

### 函数流程图

开始

给第i个城市赋一个新颜色

找到城市i的相邻城市

是否撞色

i++

否

是

## 实验结果

|  |
| --- |
|  |
| 图10. 着色结果图 |

对照给定的城市关系，城市1与城市2、4、5相邻，填色无问题。城市2与城市1、3、4相邻，填色无问题。城市3与城市2、4相邻，填色无问题。城市4与城市1、2、3、5相邻，填色无问题。城市5与城市1、4相邻，填色无问题。城市填色成功！

|  |
| --- |
|  |
| 图11. 城市关系邻接矩阵 |

## 实验分析

填色问题实际上比上一题，也就是迷宫求解问题更加简单。不必额外定义新的结构，只用数组就可以描述所需要的栈了。可以改进的地方，就是如何储存地图。邻接矩阵是最简单的方法，但并不是最省空间的方法，不过时间与空间的占用都是相对的，换一种新的存储方法，可能会节约一点空间，但会多占用一些时间。

# 题目3

## 题目

霍夫曼编码问题。使用霍夫曼的编码方式，对字符串进行编码与解码。

## 设计思路

### 基本思路

对字符进行霍夫曼编码，总共有三个问题需要解决。一，如何统计字符的权重。二，如何生成一个霍夫曼树。三，如何利用霍夫曼树，对霍夫曼码解码。

对于如何统计字符的频数，其中关键的问题便是，在新接收到一个字符后，我们如何判断这个字符是否已经出现过了。为了解决这个问题，需要编写几个函数，但都不难，只是略显繁琐。

对于第二个问题，如何生成一个霍夫曼树，我们首先需要理解霍夫曼树的原理。霍夫曼树根据数据的权重进行树的构建，每次挑选两个权重最小的节点，组合在一起，构成一个新的节点。新节点的权重为其两个子节点的权重之和。然后将已经用过一次的两个节点排除在外，重复上述操作。最终便可形成一个霍夫曼树。给定初始数据的个数，则霍夫曼树的节点数便已确定，于是我们可以通过数组的方式储存霍夫曼树。

对于第三个问题，我选择从根节点出发，按照1和0的关系，最终抵达对应的叶子节点，即可对霍夫曼码进行解码。

### 相关数据结构的设计

#### 存储字符信息

输入的字符有两方面信息需要存储，一是字符本身，二是他待求的频数。

|  |
| --- |
|  |
| 图12. 存储字符信息 |

#### 存储树的节点信息

树节点需要存储的信息有：树的父节点，树的左右孩子节点，以及这个节点的权重。我额外增加了condition量，在寻找最小的两个节点时使用。

|  |
| --- |
|  |
| 图13. 存储树的节点信息 |

### 算法的简介

霍夫曼编码，首先需要我们对现有的节点进行比较，从中挑选出两个权重最小的节点，把他们合并为一个新的节点。然后将新节点加入现有节点的数组中，并将已经用过的两个节点，再接下来的比较中剔除。以此循环，最终用光所有节点，得到霍夫曼树。从根节点出发，向左孩子移动，则为1，向右孩子移动记为0。

### 函数流程图

#### search\_ABC（在字典中寻找字符）

确定寻找目标

循环遍历字典

是否匹配

结束循环，返回对应指针

遍历是否结束

结束，返回-1

是

否

否

是

#### calculate\_ABC（统计字符频数）

新取一个字符

search\_ABC

得到指针

字典对应未知的num增加1

realloc

为字典增加一片存储区域

存储相关信息

得到-1

还有字符待统计

是

统计结束

#### sort\_ABC（为字符频数排序）

循环

确定第i个位置的元素

选出第i个元素后面最小的元素

将该元素放置到第i个位置

i++

i达到最大？

结束

是

否

#### Select（寻找两个最小节点）

进入循环

遍历，找最小节点，要求condition!=1。得到该元素位置，并令该元素condition=1

进入循环

遍历，找最小节点，要求condition!=1。得到该元素位置，并令该元素condition=1

#### make\_huffman（生成霍夫曼树）

Select函数，找到两个最小节点

生成新的父节点

左孩子右孩子分别为上一步找到的最小节点

权重为左右孩子的权重之和

把新的父节点涵盖进下一次的Select的比较范围

因为Select中已把condition置为1，故比较过的元素不再参与比较

所有叶节点使用完毕？

结束

否

是

#### show\_tree（展示霍夫曼编码）

从叶节点i向上

找到子节点的父节点

父节点的左孩子指向该子节点？

stack内添加1

stack内添加0

以父节点为新的子节点

抵达根节点？

否

结束，将stack内元素逐个输出

#### huf\_to\_human（将霍夫曼码转为人类的语言）

读取一个字符

从根节点出发

为‘1’？

走向左孩子

走向右孩子

新的节点无孩子？

输出对应字符

否

是

是

否

## 实验结果

运行程序后，输入任意一段字符。

|  |
| --- |
|  |
| 图14. 输入一段字符 |

随后，程序先统计各个字符的频数。

|  |
| --- |
|  |
| 图15. 统计词频，得到权重 |

同时程序将字符按照频数进行排序。这一步与生成霍夫曼树无关，只是将字符按频数排序后，在寻找字符的霍夫曼码时会更方便一点。

|  |
| --- |
|  |
| 图16. 对字符按照频数进行排序 |

同时，程序生成霍夫曼树，展示各个字符的编码。

|  |
| --- |
|  |
| 图17. 霍夫曼编码 |

随后我们可以以霍夫曼码的形式输入一段字符，程序会把将其转换为正常的字符。

|  |
| --- |
|  |
| 图18. 霍夫曼码的转换 |

又例如：

输出一段字符，作为编码的样本

huazhong\_university\_of\_science\_and\_technology

统计字符频数

word:h num:3

word:u num:2

word:a num:2

word:z num:1

word:o num:4

word:n num:5

word:g num:2

word:\_ num:5

word:i num:3

word:v num:1

word:e num:4

word:r num:1

word:s num:2

word:t num:2

word:y num:2

word:f num:1

word:c num:3

word:d num:1

word:l num:1

进行排序

word:z num:1

word:v num:1

word:r num:1

word:f num:1

word:d num:1

word:l num:1

word:s num:2

word:t num:2

word:y num:2

word:g num:2

word:a num:2

word:u num:2

word:c num:3

word:h num:3

word:i num:3

word:o num:4

word:e num:4

word:\_ num:5

word:n num:5

z code:000010

v code:000011

r code:000000

f code:000001

d code:000110

l code:000111

s code:1011

t code:1110

y code:1111

g code:1100

a code:1101

u code:00010

c code:0100

h code:0101

i code:1010

o code:0010

e code:0011

\_ code:011

n code:100

现在，你可以按上述规则，输入一段霍夫曼编码

01010001010111110011110110101101

hust\_aia

## 实验分析

本程序较为完整的实现了霍夫曼的编码与解码。首先统计任意字符串的字符频数，再按照字符频数去生成霍夫曼树，并将霍夫曼树展示出来。在程序设计的过程中，遇到了很多正常学习中没有发现的问题，通过实际编写程序，我对于数据结构的学习得到了明显的强化。

# 附录

## 实验一代码

1. /\*迷宫求解\*/
2. /\*用二维数组表示迷宫地图，0表示通道，1表示墙壁\*/
3. /\*创建栈来存储当前路径\*/
4. #include<stdio.h>
5. #include<stdlib.h>
6. #define n1 10
7. #define n2 10
8. /\*地图节点的信息\*/
9. **enum** dir{up, down, left, right};
10. **typedef** **struct**
11. {
12. **int** foot;//1表示已走过，0表示没走过
13. **int** way;//1表示墙壁，0表示通路，3表示起点，2表示终点
14. } maze\_point;
15. /\*当前路径的信息\*/
16. **typedef** **struct**
17. {
18. **int** x;
19. **int** y;
20. **int** pre\_dir;//来路的方向
21. }stack;

24. **void** create\_maze(maze\_point maze[][n2]);//初始化迷宫
25. **int** solve\_maze(maze\_point maze[][n2], stack cur\_way[]);//寻路

28. //初始化迷宫
29. **void** create\_maze(maze\_point maze[][n2])
30. {
31. **int** i,j;
32. **int** map[n1][n2]={
33. {1,1,1,1,1,1,1,1,1,1},
34. {1,3,0,1,0,0,0,1,0,1},
35. {1,0,0,1,0,0,0,1,0,1},
36. {1,0,0,0,0,1,1,0,0,1},
37. {1,0,1,1,1,0,0,0,0,1},
38. {1,0,0,0,1,0,0,0,0,1},
39. {1,0,1,0,0,0,1,0,0,1},
40. {1,0,1,1,1,0,1,1,0,1},
41. {1,1,0,0,0,0,0,0,2,1},
42. {1,1,1,1,1,1,1,1,1,1} };
44. //足迹初始化，为每一个位置的足迹maze[i][j].foot赋0
45. **for**(i=0;i<n1;i++)
46. {
47. **for**(j=0;j<n2;j++)
48. {
49. maze[i][j].foot = 0;//足迹初始化
50. }
51. }
52. //地图初始化，把map[n1][n2]的信息导入maze[n1][n2]中
53. **for**(i=0;i<n1;i++)
54. {
55. **for**(j=0;j<n2;j++)
56. {
57. maze[i][j].way = map[i][j];
58. }
59. }
61. }

64. //迷宫寻路
65. **int** solve\_maze(maze\_point maze[][n2], stack cur\_way[])
66. {
67. **int** i;
68. **int** j;
69. **int** top = 0;//指向栈顶
70. **int** cur\_x;
71. **int** cur\_y;
73. //找到起点(way=3)
74. **for**(i=0;i<n1;i++)
75. {
76. **for**(j=0;j<n2;j++)
77. {
78. **if**(maze[i][j].way == 3)//不断循环，知道way=3的位置
79. {
80. //从起点出发
81. cur\_way[0].x = i;
82. cur\_way[0].y = j;
83. cur\_way[0].pre\_dir = -1;
84. cur\_x = i;
85. cur\_y = j;
86. maze[i][j].foot = 1;
87. **break**;
88. }
89. }
90. **if**(maze[i][j].way == 3)//跳出两层循环
91. {
92. **break**;
93. }
94. }
95. printf("start at: x=%d y=%d\n",cur\_x,cur\_y);//找到初始点


99. //寻路
100. **while**(1)
101. {
102. /\*cur\_x，cur\_y为当前位置的坐标\*/
103. /\*此方向可走的条件是：
104. 该方向下一个位置的maze[][].way!=0(通路)
105. 且maze[][].foot=1(不走回头路)\*/
106. **if**((cur\_y+1<=n2) && (maze[cur\_x][cur\_y+1].way != 1) && (maze[cur\_x][cur\_y+1].foot == 0))//能否向右
107. {
108. //若可走，则让该元素入栈
109. cur\_way[top+1].x = cur\_x;
110. cur\_way[top+1].y = cur\_y+1;
111. cur\_way[top+1].pre\_dir = right;
112. maze[cur\_x][cur\_y+1].foot = 1;
113. top++;
114. //printf("right ");
115. }**else** **if**((cur\_y-1>=0) && (maze[cur\_x][cur\_y-1].way != 1) && (maze[cur\_x][cur\_y-1].foot == 0))//能否向左
116. {
117. cur\_way[top+1].x = cur\_x;
118. cur\_way[top+1].y = cur\_y-1;
119. cur\_way[top+1].pre\_dir = left;
120. maze[cur\_x][cur\_y-1].foot = 1;
121. top++;
122. //printf("left ");
123. }**else** **if**((cur\_x+1<=n1) && (maze[cur\_x+1][cur\_y].way != 1) && (maze[cur\_x+1][cur\_y].foot == 0))//能否向下
124. {
125. cur\_way[top+1].x = cur\_x+1;
126. cur\_way[top+1].y = cur\_y;
127. cur\_way[top+1].pre\_dir = down;
128. maze[cur\_x+1][cur\_y].foot = 1;
129. top++;
130. //printf("down ");
131. }**else** **if**((cur\_x-1>=0) && (maze[cur\_x-1][cur\_y].way != 1) && (maze[cur\_x-1][cur\_y].foot == 0))//能否向上
132. {
133. cur\_way[top+1].x = cur\_x-1;
134. cur\_way[top+1].y = cur\_y;
135. cur\_way[top+1].pre\_dir = up;
136. maze[cur\_x-1][cur\_y].foot = 1;
137. top++;
138. //printf("up ");
139. }**else**//遇到死胡同
140. {
141. maze[cur\_x][cur\_y].way = 1;//用墙堵住
142. top--;//相当于栈顶出栈
143. //printf("return ");
144. }
145. //一轮循环，走一步
146. cur\_x = cur\_way[top].x;//把cur\_x和cur\_y更新为栈顶的新元素
147. cur\_y = cur\_way[top].y;
149. **if**(maze[cur\_x][cur\_y].way == 2)//如果找到出口way=2，则退出循环
150. {
151. **break**;
152. }
153. }
155. **return** top;
156. }

159. **void** main()
160. {
161. maze\_point maze[n1][n2];
162. stack cur\_way[n1\*n2];
163. **int** top;
164. **int** i;
166. create\_maze(maze);//初始化迷宫
168. top = solve\_maze(maze, cur\_way);//寻找路径
170. //printf("%d",top);
171. printf("\nwe get a way to escape the maze!\n\n");
172. **for**(i=1;i<top;i++)
173. {
174. **if**(cur\_way[i].pre\_dir == up)
175. printf("up\n");
176. **else** **if**(cur\_way[i].pre\_dir == down)
177. printf("down\n");
178. **else** **if**(cur\_way[i].pre\_dir == left)
179. printf("left\n");
180. **else**
181. printf("right\n");
182. }
183. }

## 实验二代码

1. /\*地图填色问题\*/
2. /\*地图关系用邻接矩阵表示\*/
3. #include<stdio.h>
4. #include<stdlib.h>
5. #define n 5
7. **void** solve\_color(**int** map[][n], **int** stack[])
8. {
9. **int** top = 0;//标记栈顶
10. **int** Color;
11. **int** i,j;
12. **int** ret;
14. **for**(i=0;i<n;i++)
15. {
16. stack[i] = -1;//-1代表尚未着色
17. }

20. **for**(i=0;i<n;i++)//为第i个城市着色
21. {
22. top = i;
23. **for**(Color=0;Color<4;Color++)
24. {
25. //先给个颜色，然后判断是否重色
26. //若重色，则换下一种颜色
27. stack[top] = Color;
28. ret = 1;
29. **for**(j=0;j<n;j++)
30. {
31. **if**(map[i][j] == 1)//如果城市相邻（第i个城市（正在着色的城市）和第j个城市相邻）
32. {
33. **if**(stack[j] == stack[top])
34. {
35. ret = 0;//ret=0，代表此次着色失败
36. **break**;//这是一个“废色”，着新色
37. }
38. }
39. }
40. **if**(ret == 1)//意味着着色成功
41. {
42. **break**;
43. }
44. }
45. }
46. }

49. **int** main()
50. {
51. **int** map[n][n]={
52. {0,1,0,1,1},
53. {1,0,1,1,0},
54. {0,1,0,1,0},
55. {1,1,1,0,1},
56. {1,0,0,1,0} };
57. **int** stack[n];
58. **int** i;
59. solve\_color(map, stack);
61. **for**(i=0;i<n;i++)
62. {
63. printf("第%d城市的颜色", i+1);
64. **if**(stack[i] == 0)
65. printf("red\n");
66. **else** **if**(stack[i] == 1)
67. printf("blue\n");
68. **else** **if**(stack[i] == 2)
69. printf("green\n");
70. **else** **if**(stack[i] == 3)
71. printf("yellow\n");
72. }
73. **return** 0;
74. }

## 实验三代码

1. /\*用霍夫曼树对字符编码\*/
2. /\*左孩子1,右孩子0\*/
3. #include<stdio.h>
4. #include<stdlib.h>
6. //建立类字典类型
7. **typedef** **struct**
8. {
9. **char** letter;//字符
10. **int** num;//频数
11. } dir;

14. //二叉树节点
15. **struct** node
16. {
17. //char letter;
18. **int** weight;
19. **int** leftchl;    //表示0
20. **int** rightchl;   //表示1
21. **int** parent;
22. **int** condition;
23. };
24. **typedef** **struct** node node;

27. dir\* search\_ABC(**char** cur, dir\* dirs, **int** length);
28. dir\* calculate\_ABC(**char**\* sentence, dir\* dirs, **int**\* P\_len);
29. dir\* sort\_ABC(dir\* dirs, **int** len);
30. **void** Select(node\* tree, **int** len, **int**\* p1, **int**\* p2);
31. node\* make\_huffman(dir\* dirs, **int** len);
32. **void** show\_tree(node\* tree, **int** len, dir\* dirs);
33. **void** huf\_to\_human(node\* tree, dir\* dirs, **char**\* huffman\_code, **int** len);
35. /\*搜索dir中的字符，并返回对应位置\*/
36. dir\* search\_ABC(**char** cur, dir\* dirs, **int** length)
37. {
38. **if**(dirs==NULL)
39. {
40. //printf("there\n");
41. **return** 0;
42. }
43. **while**(length>=0)
44. {
45. **if**(dirs->letter == cur)
46. **break**;
47. dirs++;
48. length--;
49. }
51. **if**(length<0)
52. {
53. //printf("here\n");
54. **return** 0;
55. }**else**
56. {
57. //printf("%d",length);
58. //printf("%d",dirs->num);
59. **return** dirs;//若找到了该字符，则返回对应指针
60. }
61. }

64. /\*计算字符频数\*/
65. dir\* calculate\_ABC(**char**\* sentence, dir\* dirs, **int**\* P\_len)
66. {
67. **char** cur;
68. dir\* dir\_point = NULL;
69. **int** length = 0;
70. //计算频数
71. **while**(\*sentence != '\0')
72. {
73. cur = \*sentence;
75. dir\_point = search\_ABC(cur, dirs, length);//指向字典中cur所在的位置
77. //循环检索字典内是否已有该字符
78. **if**(dir\_point)
79. {
80. (dir\_point->num)++;
81. }**else**
82. {
83. length += 1;
84. dirs = (dir\*)realloc(dirs, **sizeof**(dir)\*(length+1));
85. (dirs+length-1)->letter = cur;
86. (dirs+length-1)->num = 1;
87. }
88. sentence++;
89. //printf("\n%c", cur);
90. //printf("频数%d\n", (dirs+length-1)->num);
91. }
93. \*P\_len = length;
95. **return** dirs;//因为realloc可能会改变指针的地址，所以必须把最终的dirs返回
96. }

99. /\*对字符频数排序\*/
100. /\*len为字符字典的长度，从1数起\*/
101. dir\* sort\_ABC(dir\* dirs, **int** len)
102. {
103. **int** i,j;
104. dir temp;
106. **for**(i=0; i<len-1; i++)
107. {
108. **for**(j=i+1; j<len; j++)
109. {
110. **if**((dirs+j)->num < (dirs+i)->num)
111. {
112. temp.letter = (dirs+i)->letter;
113. temp.num = (dirs+i)->num;
114. (dirs+i)->letter = (dirs+j)->letter;
115. (dirs+i)->num = (dirs+j)->num;
116. (dirs+j)->letter = temp.letter;
117. (dirs+j)->num = temp.num;
118. }
119. }
120. }
121. }
123. **void** Select(node\* tree, **int** len, **int**\* p1, **int**\* p2)
124. {
125. **int** i,j;
126. **int** s1=0, s2=0;
128. **for**(i=1;i<=len;i++)
129. {
130. **if**((tree[s1].weight>=tree[i].weight) && (tree[i].condition!=1) )
131. {
132. s1 = i;
133. }
134. }
135. tree[s1].condition = 1;
136. **for**(i=1;i<=len;i++)
137. {
138. **if**((tree[s2].weight>=tree[i].weight) && (tree[i].condition!=1) )
139. {
141. s2 = i;
142. }
143. }
144. tree[s2].condition = 1;
145. \*p1 = s1;
146. \*p2 = s2;
147. }

150. /\*构造一个哈夫曼树\*/
151. node\* make\_huffman(dir\* dirs, **int** len)
152. {
153. /\*思路：将两个频数最小的元素组合起来\*/
154. node\* p;
155. node\* tree;
156. **int** i;
157. **int** s1,s2;
158. dir\* p1;
159. tree = (node\*)malloc(**sizeof**(node)\*(2\*len));//为哈夫曼树分配空间，看待成数组,0不用
161. //节点初始化
162. tree->leftchl = 0;//0节点的初始化
163. tree->rightchl = 0;
164. tree->parent = 0;
165. tree->weight = 100;
166. tree->condition = 0;
168. **for**(p=tree+1,i=1,p1=dirs; i<=len; ++i,++p,++p1)
169. {
170. p->leftchl = 0;
171. p->rightchl = 0;
172. p->parent = 0;
173. p->condition = 0;
174. p->weight = p1->num;
175. }
176. **for**(; i<2\*len;++i,++p)
177. {
178. p->leftchl = 0;
179. p->rightchl = 0;
180. p->parent = 0;
181. p->condition = 0;
182. p->weight = 0;
183. }
185. **for**(i=len+1; i<=2\*len-1; ++i)
186. {
187. //在1~i-1中选择权重最小的两个节点，返回他们的序号
189. Select(tree, i-1, &s1, &s2);
190. tree[s1].parent = i;
191. tree[s2].parent = i;
192. tree[i].leftchl = s1;
193. tree[i].rightchl = s2;
194. tree[i].weight = tree[s1].weight + tree[s2].weight;
195. }
197. **return** tree;
198. }

201. /\*展示霍夫曼树\*/
202. **void** show\_tree(node\* tree, **int** len, dir\* dirs)
203. {
204. **int** i,j;
205. **int** root=1;
206. **int** top=0;
207. **int** stack[10];
208. **int** temp=-1;
209. **int** temp2;
210. //printf("len:%d",len);
211. **for**(i=1; i<=len; ++i)
212. {
213. top = 0;
214. temp = tree[i].parent;
215. temp2 = i;
216. printf("\n%c ",dirs[i-1].letter);
217. **while**(temp)
218. {
219. **if**(tree[temp].leftchl == temp2)
220. {
221. stack[top] = 1;
222. top++;
223. }**else** **if**(tree[temp].rightchl == temp2)
224. {
225. stack[top] = 0;
226. top++;
227. }
228. temp2 = temp;
229. temp = tree[temp].parent;
230. }
231. printf("code:");
232. **for**(;top>0;top--)
233. {
234. printf("%d",stack[top-1]);
235. }
236. printf("\n");
237. }
238. }
240. /\*把霍夫曼码转为人话\*/
241. **void** huf\_to\_human(node\* tree, dir\* dirs, **char**\* huffman\_code, **int** len)
242. {
243. **int** temp;
244. **int** tree\_len = 2\*len-1;//树的节点个数
246. temp = tree\_len;//根节点的位置
248. **while**(\*huffman\_code != '\0')
249. {//wzc清洁能源
250. **while**(tree[temp].leftchl != 0)
251. {
252. **if**(\*huffman\_code == '1')
253. {
254. temp = tree[temp].leftchl;
255. huffman\_code++;
256. }**else** **if**(\*huffman\_code == '0')
257. {
258. temp = tree[temp].rightchl;
259. huffman\_code++;
260. }**else**
261. **break**;
262. }
263. printf("%c",dirs[temp-1].letter);
264. temp = tree\_len;
265. }
266. }

269. **int** main()
270. {
271. **char** sentence[100];
272. **char** huffman\_code[100];
273. dir\* dirs = NULL;//词频字典
274. dir\* P\_dir = NULL;
275. node\* tree = NULL;//根节点
276. **int** len;
277. **int** i;
279. printf("输出一段字符，作为编码的样本\n");
280. scanf("%s",sentence);
281. dirs = calculate\_ABC(sentence, dirs, &len);
282. printf("统计字符频数\n");
283. i = len;
284. P\_dir = dirs;
285. **while**(i>0)
286. {
287. printf("word:%c num:%d\n",P\_dir->letter, P\_dir->num);
288. i--;
289. P\_dir++;
290. }

293. sort\_ABC(dirs, len);
294. printf("进行排序\n");
295. i = len;
296. P\_dir = dirs;
297. **while**(i>0)
298. {
299. printf("word:%c num:%d\n",P\_dir->letter, P\_dir->num);
300. i--;
301. P\_dir++;
302. }
304. tree = make\_huffman(dirs, len);
305. show\_tree(tree, len, dirs);
307. printf("现在，你可以按上述规则，输入一段霍夫曼编码\n");
308. scanf("%s",huffman\_code);
309. huf\_to\_human(tree, dirs, huffman\_code, len);//把霍夫曼码转化成人话
311. }