

**数**

**据**

**结**

**构**

**课**

**程**

**设**

**计**

**专业班级： 软件工程2003班**

**学 号： U202010783**

**姓 名： 刘铭宸**

**软件工程**

**目 录**

**1基于磁盘的带替换选择的合并排序 3**

1.1 问题描述和分析 3

1.2 设计与实现 3

1.3 代码清单 4

1.4 测试用例与结果分析 9

**2 AVL树的插入 12**

2.1 问题描述和分析 12

2.2 设计与实现 12

2.3 代码清单 13

2.4 测试用例与结果分析 16

**3 图的拓扑排序 19**

2.1 问题描述和分析 19

2.2 设计与实现 19

2.3 代码清单 19

2.4 测试用例与结果分析 21

**4最小编辑距离 24**

2.1 问题描述和分析 24

2.2 设计与实现 24

2.3 代码清单 25

2.4 测试用例与结果分析 26

**5霍夫曼编码 27**

2.1 问题描述和分析 27

2.2 设计与实现 27

2.3 代码清单 27

2.4 测试用例与结果分析 31

**6有向图最短路径的Dijsktra算法 34**

2.1 问题描述和分析 34

2.2 设计与实现 34

2.3 代码清单 34

2.4 测试用例与结果分析 37

**7快速排序 39**

2.1 问题描述和分析 39

2.2 设计与实现 39

2.3 代码清单 39

2.4 测试用例与结果分析 40

**8最小生成树 42**

2.1 问题描述和分析 42

2.2 设计与实现 42

2.3 代码清单 42

2.4 测试用例与结果分析 45

# 1基于磁盘的带替换选择的合并排序

## 1.1 问题描述和分析

外部排序基本上由两个相对独立的阶段组成。首先，按可用内存大小，将外存上含n个记录的文件分成若干长度为x的子文件或段，依次读入内存并利用有效的内部排序方法对它们进行排序，并将排序后得到的有序子文件重新写入外存；然后，对这些归并段进行逐趟归并，使归并段逐渐由小到大，直至得到整个有序文件为止。一般情况下，外部排序所需总的时间=内部排序（产生初始归并段）所需的时间+外存信息读写的时间+内部归并所需的时间。进行外存读写的时间要比内部归并所需时间大得多，因此，提高外排的效率应主要着眼于减少外存信息读写的次数。在一般情况下，对m个初始归并段进行k-路平衡归并时，归并的趟数为⌊⌋，可见若增加k或减少m便能减少归并趟数。

假设所得初始归并段为m个，则内部归并过程中进行比较的总次数为⌊⌋(k-1)(n-1)tmg 其中tmg为进行内部归并所需时间。由于随k的增长而增长，则内部归并时间亦随k的增长而增长，这将抵消由于增大k而减少外存信息读写时间所得效益，这是我们不希望的。然而，若在进行k-路归并时利用“败者树”，则可使在k个记录中选出关键字最小的记录时仅需进行⌊⌋次比较，从而使总的归并时间变为⌊⌋(n-1)tmg，显然，这个式子和k无关，它不再随k的增长而增长。

另外，归并的趟数不仅和k成反比，也和m成正比，因此，减少m是减少归并趟数的另一条途径。所以采用置换—选择排序。它的特点是：在整个排序（得到所有初始归并段）的过程中，选择最小（或最大）关键字和输入、输出交叉或平行进行。

## 1.2 设计与实现

“败者树”是树形选择排序的一种变型，即在双亲结点中记下刚进行完的这场比赛中的败者，而让胜者去参加更高一层的比赛。在选得最小关键字的记录之后，只要修改叶子节点中的值，使其为同一归并段中的下一个记录的关键字，然后从该结点向上和双亲结点所指的关键字进行比较，摆着留在该双亲结点，胜者继续向上直至树根的双亲。为了防止在归并过程中某个归并段变空，可以在每个归并段中附加一个关键字为最大值的记录。当选出的“冠军”记录的关键字为最大值时，表明此次归并已完成。由于实现k-路归并的败者树的深度为⌈⌉+1，则在k个记录中选择最小关键字仅需进行⌈⌉次比较。

得到初始归并段的过程用置换—选择排序来完成：假设初始待排文件为输入文件FI，初始归并段文件为输出文件FO，内存工作区为WA，FO和WA的初始状态为空，并设内存工作区WA的容量可容纳w个记录，则置换—选择排序的操作过程为：

(1)从FI输入w个记录到工作区WA。

(2)从WA中选出其中关键字取最小值的记录，记为MINIMAX记录。

(3)将MINIMAX记录输出到FO中去。

(4)若FI不空，则从FI输入下一个记录到WA中。

(5)从WA中所有关键字比MINIMAX记录的关键字大的记录中选出最小关键字记录，作为新的MINIMAX记录。

(6)重复(3)-(5)，直至在WA中选不出新的MINIMAX记录为止，由此得到一个初始归并段，输出一个归并段的结束标志到FO中去。

(7)重复(2)-(6)，直至WA为空。由此得到全部初始归并段。

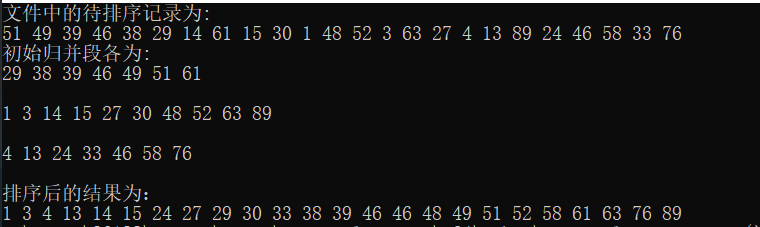
## 1.3 代码清单

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <time.h>
4. #define w             6     //内存工作区可容纳的记录个数
5. #define MAXKEY        (~(0x1 << 31)) //最大关键字标志
6. #define RUNEND\_SYMBOL (~(0x1 << 31)) //归并段结束标志
7. #define N             24    //输入文件的预设记录个数
8. #define MINKEY        -(~(0x1 << 31))//最小关键字标志
9. **typedef** **int** KeyType;        //关键字类型
10. **typedef** **struct** {
11. KeyType key;
12. }RcdType;                   //记录类型
13. **typedef** **int** LoserTree[w];  //败者树是完全二叉树且不含叶子，可采用顺序存储结构
14. **typedef** **struct** {
15. RcdType rec;        //记录
16. KeyType key;        //从记录中抽取的关键字
17. **int** rnum;           //所属归并段的段号
18. }RcdNode,WorkArea[w];   //内存工作区，容量为w
19. **int** t0 = 0, t1 = 0, t2 = 0, m = 0;//标志变量
21. /\*输入、输出、中间暂存文件指针的定义
22. f0、f1、f2为中间暂存文件，存放初始归并段
23. ff为最终输出文件
24. \*/
25. **FILE**\* f0 = fopen("f0", "wb");
26. **FILE**\* f1 = fopen("f1", "wb");
27. **FILE**\* f2 = fopen("f2", "wb");
28. **FILE**\* f3 = fopen("f0", "rb");
29. **FILE**\* f4 = fopen("f1", "rb");
30. **FILE**\* f5 = fopen("f2", "rb");
31. **FILE**\* ff = fopen("ff", "wb");
33. //从wa[q]起到败者树的根比较选择MINIMAX记录，并由q指向它所在的归并段
34. **void** Select\_MiniMax(LoserTree& ls, WorkArea wa, **int** q) {
35. **int** p, s, t;
36. **for** (t = (w + q) / 2, p = ls[t]; t > 0; t /= 2, p = ls[t]) {
37. //段号小者或段号相同且关键字更小者为胜者
38. **if** (wa[p].rnum < wa[q].rnum || wa[p].rnum == wa[q].rnum && wa[p].key < wa[q].key) {
39. s = q;
40. q = ls[t];  //q指向新的胜者
41. ls[t] = s;
42. }
43. }
44. ls[0] = q;  //冠军
45. }
47. //沿从叶子结点wa[q]到根结点ls[0]的路径调整败者树
48. **void** Adjust(LoserTree& ls, WorkArea wa, **int** q) {
49. **int** t = (q + w) / 2;                        //ls[t]是wa[q]的双亲结点
50. **while** (t > 0) {
51. **if** (wa[q].key > wa[ls[t]].key) {
52. **int** swap = q;
53. q = ls[t];                          //q指示新的胜者
54. ls[t] = swap;
55. }
56. t = t / 2;
57. }
58. ls[0] = q;
59. }
60. **int** a0[N], a1[N], a2[N], count0 = 0, count1 = 0, count2 = 0;
62. /\*已知wa[0]到wa[w-1]为完全二叉树ls的叶子结点存有w个关键字，
63. 沿从叶子到根的w条路径将ls调整为败者树
64. \*/
65. **void** CreateLoserTree(LoserTree& ls, WorkArea& wa) {
66. **int** i;
67. **for** (i = 0; i < w; i++) {
68. ls[i] = w;              //设置ls中败者的初值
69. }
70. **for** (i = w - 1; i >= 0; i--) {
71. Adjust(ls, wa, i);      //依次从wa[w-1],wa[w-2],...,wa[0]出发调整败者
72. }
73. }
75. /\*从存放着初始归并段中的三个文件f0, f1, f2中读取下一个记录的关键字，
76. 如果读到一个归并段结束则读取最大关键字MAXKEY\*/
77. **void** input(LoserTree& ls, WorkArea& wa, **int** i) {
79. **switch** (i) {
80. **case** 0:
81. **if** (t0 < count0) {
82. fread(&wa[i].key, **sizeof**(KeyType), 1, f3);
83. t0++;
84. }
85. **else** wa[i].key = MAXKEY;
86. **break**;
87. **case** 1:
88. **if** (t1 < count1) {
89. fread(&wa[i].key, **sizeof**(KeyType), 1, f4);
90. t1++;
91. }
92. **else** wa[i].key = MAXKEY;
93. **break**;
94. **case** 2:
95. **if** (t2 < count2) {
96. fread(&wa[i].key, **sizeof**(KeyType), 1, f5);
97. t2++;
98. }
99. **else** wa[i].key = MAXKEY;
100. **break**;
101. **default**:wa[i].key = MAXKEY;
102. }
103. }
105. /\*利用败者树ls将编号从0到k-1的k个输入归并段中的记录归并到输出归并段
106. wa[0]至wa[w-1]为败者树上的w个叶子结点，
107. 分别存放w个输入归并段中当前记录的关键字
108. \*/
109. **void** K\_Merge(LoserTree& ls, WorkArea& wa) {
110. **for** (**int** i = 0; i < w; i++) {
111. input(ls, wa, i); //分别从w个输入归并段读入当前该段第一个记录的关键字到外结点
112. }
113. CreateLoserTree(ls,wa);      //建败者树ls，选得最小关键字为wa[ls[0]].key
114. **while** (wa[ls[0]].key != MAXKEY) {
115. fprintf(ff, "%d", wa[ls[0]].key);   //将最小记录的关键字写入输出文件
116. printf("%d ", wa[ls[0]].key);
117. input(ls, wa, ls[0]);               //从编号为ls[0]的输入归并段中读入下一个记录的关键字
118. Adjust(ls, wa, ls[0]);              //调整败者树，选择新的最小关键字
119. }
120. fclose(ff);
121. }
122. //输入w个记录到内存工作区wa，建得败者树ls，选出关键字最小的记录并由s指示
123. **void** Construct\_Loser(LoserTree& ls, WorkArea& wa, **FILE**\* fi) {
124. **int** i;
125. **for** (i = 0; i < w; i++)
126. wa[i].rnum = wa[i].key = ls[i] = 0;             //工作区初始化
127. **for** (i = w - 1; i >= 0; i--) {
128. fread(&wa[i].rec, **sizeof**(RcdType), 1, fi);      //输入一个记录
129. wa[i].key = wa[i].rec.key;                      //提取关键字
130. wa[i].rnum = 1;                                 //其段号为"1"
131. Select\_MiniMax(ls, wa, i);                      //调整败者
132. }
133. }
135. //求得一个初始归并段，fi为输入文件指针，f0为输出文件指针
136. **void** get\_run(LoserTree& ls, WorkArea& wa, **FILE**\* fi, **FILE**\* fo, **int** rc, **int**\* rmax) {
137. **int** q;
138. KeyType minimax;
139. **while** (wa[ls[0]].rnum == rc) {          //选得的MINIMAX记录属当前段时
140. q = ls[0];                          //q指示MINIMAX记录在wa中的位置
141. minimax = wa[q].key;
142. fwrite(&wa[q].rec, **sizeof**(RcdType), 1, fo);   //将刚选好的MINIMAX记录写入输出文件
143. //输入文件结束，虚设记录（属"rmax+1"段）
144. fread(&wa[q].rec, **sizeof**(RcdType), 1, fi);//从输入文件读入下一记录
145. **if** (feof(fi)) {
146. wa[q].rnum = \*rmax + 1;
147. wa[q].key = MAXKEY;
148. }
149. **else** {                                    //输入文件非空时
150. wa[q].key = wa[q].rec.key;            //提取关键字
151. **if** (wa[q].key < minimax) {            //新读入的记录属下一段
152. \*rmax = rc + 1;
153. wa[q].rnum = \*rmax;
154. }
155. **else**
156. wa[q].rnum = rc;                  //新读入的记录属当前段
157. }
158. Select\_MiniMax(ls, wa, q);                //选择新的MINIMAX记录
159. }
160. }
162. /\*
163. 在败者树ls和内存工作区wa上用置换 - 选择排序求初始归并段，
164. fi为输入文件（只读文件）指针，fo为输出文件（只写文件）指针，
165. 两个文件均已打开
166. \*/
167. **void** Replace\_Selection(LoserTree& ls, WorkArea& wa, **FILE**\* fi, **FILE**\* fo) {
168. **int** rc, rmax;
169. RcdType j;
170. j.key = RUNEND\_SYMBOL;
171. Construct\_Loser(ls, wa, fi);              //初建败者树
172. rc = rmax = 1;                            //rc指示当前生成的初始归并段的段号，rmax指示wa中关键字所属初始归并段的最大段号
173. **while** (rc <= rmax) {                   //"rc==rmax+1"标志输入文件的置换-选择排序已完成
174. get\_run(ls, wa, fi, fo, rc, &rmax);   //求得一个初始归并段
175. fwrite(&j, **sizeof**(RcdType), 1, fo);   //将段结束标志写入输出文件
176. rc = wa[ls[0]].rnum;                  //设置下一段的段号
177. }
178. }
180. **void** print(RcdType t) {
181. printf("%d ", t.key);
182. }
183. **int** main() {
184. **int** i, k, j = 0;
185. RcdType a[N] = { 51,49,39,46,38,29,14,61,15,30,1,48,52,3,63,27,4,13,89,24,46,58,33,76 };
186. RcdType b;
187. **FILE**\* fi, \* fo;                     //输入输出文件
188. LoserTree ls;                       // 败者树
189. WorkArea wa;                        // 内存工作区
190. fo = fopen("ori", "wb");            //准备对 ori 文本文件进行写操作
191. fwrite(a, **sizeof**(RcdType), N, fo);  //将数组 a 写入大文件ori
192. fclose(fo);                         //关闭指针 fo 表示的文件
193. fi = fopen("ori", "rb");            //准备对 ori 文本文件进行读操作
194. printf("文件中的待排序记录为:\n");
195. **for** (i = 1; i <= N; i++) {           // 依次将文件ori的数据读入并赋值给b
196. fread(&b, **sizeof**(RcdType), 1, fi);
197. print(b);
198. }
199. printf("\n");
200. rewind(fi);                         // 使fi的指针重新返回大文件ori的起始位置，以便重新读入内存，产生有序的子文件。
201. fo = fopen("out", "wb");            // 用置换－选择排序求初始归并段
202. Replace\_Selection(ls, wa, fi, fo);
203. fclose(fo);
204. fclose(fi);
205. fi = fopen("out", "rb");
206. printf("初始归并段各为:\n");
207. **do** {
208. k = fread(&b, **sizeof**(RcdType), 1, fi); //读 fi 指针指向的文件，并将读的记录赋值给 b，整个操作成功与否的结果赋值给 k
209. **if** (k == 1) {
210. **if** (b.key == MAXKEY) { //当其值等于最大值时，表明当前初始归并段已经完成
211. m++;
212. printf("\n\n");
213. **continue**;
214. }
215. print(b);
216. }
217. } **while** (k == 1);
218. fclose(fi);
219. fi = fopen("out", "rb");
220. //将置换-选择排序所得到的初始归并段存放进三个暂存文件中
221. **while** (1) {
222. fread(&b, **sizeof**(RcdType), 1, fi);
223. **if** (b.key == MAXKEY)**break**;
224. a0[count0++] = b.key;
225. }
226. fwrite(a0, **sizeof**(RcdType), count0, f0);
227. fclose(f0);
228. **while** (1) {
229. fread(&b, **sizeof**(RcdType), 1, fi);
230. **if** (b.key == MAXKEY)**break**;
231. a1[count1++] = b.key;
232. }
233. fwrite(a1, **sizeof**(RcdType), count1, f1);
234. fclose(f1);
235. **while** (1) {
236. fread(&b, **sizeof**(RcdType), 1, fi);
237. **if** (b.key == MAXKEY)**break**;
238. a2[count2++] = b.key;
239. }
240. fwrite(a2, **sizeof**(RcdType), count2, f2);
241. fclose(f2);
242. printf("排序后的结果为：\n");
243. K\_Merge(ls, wa);
244. **return** 0;
245. }

## 1.4 测试用例与结果分析

|  |  |  |
| --- | --- | --- |
|  | 内存工作区可容纳的记录个数 | 6 |
| 用例1 | 输入文件中的记录个数 | 24 |
|  | 输入文件中记录的关键字 | 51,49,39,46,38,29,14,61,15,30,1,48,52,3,63,27,4,13,89,24,46,58,33,76 |
|  | 初始归并段个数 | 3 |
| 预期结果 | 初始归并段内容 | 29 38 39 46 49 51 61  1 3 14 15 27 30 48 52 63 89  4 13 24 33 46 58 76 |
|  | 排序结果 | 1 3 4 13 14 15 24 27 29 30 33 38 39 46 46 48 49 51 52 58 61 63 76 89 |

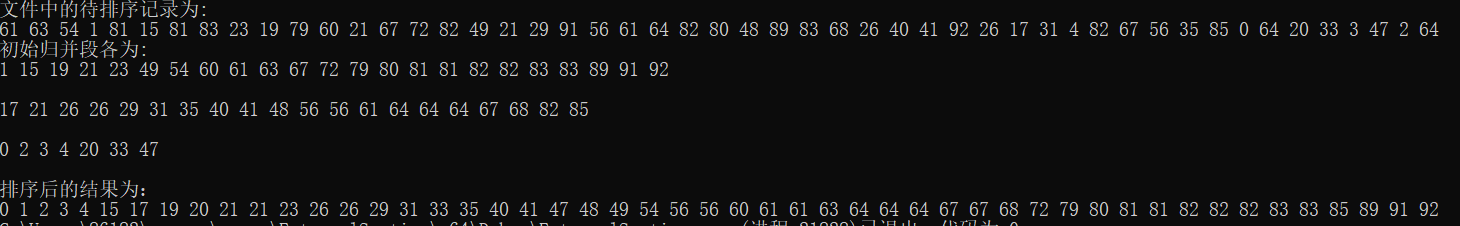
运行结果如图1-1



**图1-1 用例一的测试结果**

|  |  |  |
| --- | --- | --- |
|  | 内存工作区可容纳的记录个数 | 12 |
| 用例2 | 输入文件中的记录个数 | 50 |
|  | 输入文件中记录的关键字 | 61 63 54 1 81 15 81 83 23 19 79 60 21 67 72 82 49 21 29 91 56 61 64 82 80 48 89 83 68 26 40 41 92 26 17 31 4 82 67 56 35 85 0 64 20 33 3 47 2 64 |
|  | 初始归并段个数 | 3 |
| 预期结 果 | 初始归并段内容 | 1 15 19 21 23 49 54 60 61 63 67 72 79 80 81 81 82 82 83 83 89 91 92  17 21 26 26 29 31 35 40 41 48 56 56 61 64 64 64 67 68 82 85  0 2 3 4 20 33 47 |
|  | 排序结果 | 0 1 2 3 4 15 17 19 20 21 21 23 26 26 29 31 33 35 40 41 47 48 49 54 56 56 60 61 61 63 64 64 64 67 67 68 72 79 80 81 81 82 82 82 83 83 85 89 91 92 |

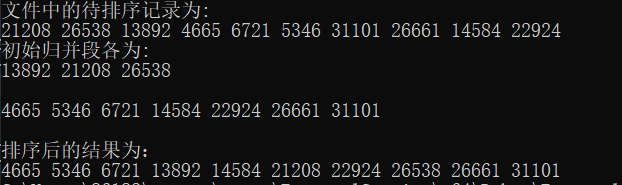
运行结果如图1-2



**图1-2 用例二的测试结果**

|  |  |  |
| --- | --- | --- |
|  | 内存工作区可容纳的记录个数 | 3 |
| 用例3 | 输入文件中的记录个数 | 10 |
|  | 输入文件中记录的关键字 | 21208,26538,13892,4665,6721,5346,31101,26661,14584,22924 |
|  | 初始归并段个数 | 2 |
| 预期结 果 | 初始归并段内容 | 13892 21208 26538  4665 5346 6721 14584 22924 26661 31101 |
|  | 排序结果 | 4665 5346 6721 13892 14584 21208 22924 26538 26661 31101 |

运行结果如图1-3



**图1-3 用例三的测试结果**

# 2 AVL树的插入

## 2.1 问题描述和分析

平衡二叉树又称AVL树，它的左子树和右子树都是平衡二叉树，且左子树和右子树的深度之差的绝对值不超过1。若将二叉树上结点的平衡因子BF定义为该结点的左子树的深度减去它的右子树的深度，则平衡二叉树上所有结点的平衡因子只可能是-1、0和1。只要二叉树上有一个结点的平衡因子的绝对值大于1，则该二叉树就是不平衡的。所以在向AVL树中插入新的数据时，若造成了某个结点的平衡因子的绝对值大于1，就需要通过单向左旋平衡处理、单向右旋平衡处理、双向旋转（先左后右）平衡处理、双向旋转（先右后左）平衡处理等操作对树进行调整，使其重新成为一棵平衡二叉树。

## 2.2 设计与实现

在平衡的二叉排序树BBST上插人一个新的数据元素e的递归算法可描述如下：

(1) 若BBST为空树，则插入一个数据元素为e的新结点作为BBST的根结点，树的深度增l;

(2) 若e的关键字和BBST的根结点的关键字相等，则不进行插人；

(3) 若e的关键字小于BBST的根结点的关键字，而且在BBST的左子树中不存在和e有相同关键字的结点，则将e插入在BBST的左子树上，并且当插入之后的左子树深度增加(+1)时，分别就下列不同情况处理之：

①BBST的根结点的平衡因子为-1（右子树的深度大于左子树的深度）：则将根结点的平衡因子更改为O,BBST的深度不变；

②BBST 的根结点的平衡因子为0（左、右子树的深度相等）：则将根结点的平衡因子更改为1,BBST的深度增1;

③BBST的根结点的平衡因子为1（左子树的深度大于右子树的深度）：若BBST的左子树根结点的平衡因子为1，则需进行单向右旋平衡处理，并且在右旋处理之后，将根结点和其右子树根结点的平衡因子更改为0，树的深度不变；

若BBST的左子树根结点的平衡因子为-1，则需进行先向左、后向右的双向旋转平衡处理，并且在旋转处理之后，修改根结点和其左、右子树根结点的平衡因子，树的深度不变；

(4) 若e的关键字大于BBST的根结点的关键字，而且在BBST的右子树中不存在和e有相同关键字的结点，则将e插入在BBST的右子树上，并且当插入之后的右子树深度增加(+1)时，分别就不同情况处理之。其处理操作和（三）中所述相对称。

## 2.3 代码清单

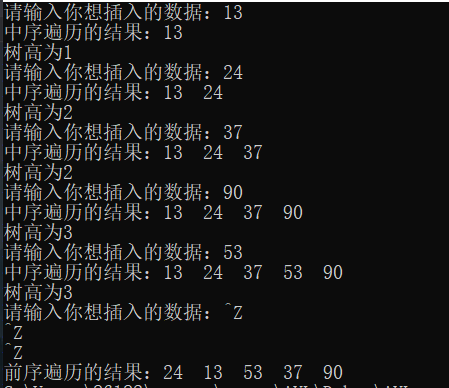
1. #include<stdio.h>
2. #include<stdlib.h>
3. **typedef** **struct** node {
4. **int** data;
5. **int** bf;     //balance flag 平衡因子的缩写
6. **struct** node\* lchild, \* rchild;
7. }BitNode, \* BiTree;
9. //之所以要用\*\* 是因为参数传递情况下要更改指针指向内容的需要地址传递
10. //右调整：对以p为根的二叉树右旋   该树 左子树深度大于右子树
11. //调整后：根节点为p节点的左孩子，同时将p节点左孩子的右孩子接在p节点的左孩子
12. //然后将p节点作为新根的右孩子
13. **void** rightRotate(BiTree\* p) {
14. BiTree L = (\*p)->lchild;
15. (\*p)->lchild = L->rchild;
16. L->rchild = (\*p);
17. \*p = L;
18. }
20. //左调整：同理，只是将以p为根的二叉树左旋，该树的右子树深度大于左子树
21. //调整好：根节点为p的右孩子，同时将p节点的右孩子的左孩子加在p节点的右孩子
22. //然后将p节点作为新的根节点的左孩子
23. **void** leftRotate(BiTree\* p) {
24. BiTree R = (\*p)->rchild;
25. (\*p)->rchild = R->lchild;
26. R->lchild = (\*p);
27. \*p = R;
28. }
30. //处理LL 和LR 两种类型的旋状  LL型旋转只需要右旋一次 LR型又分三种情况
31. //但是LR型的处理方法都是先根据根节点的左孩子左旋转一次变为LL型然后再又旋转
32. **void** LL\_LRBalance(BiTree\* p) {
33. BiTree L = (\*p)->lchild;
34. BiTree R;
35. **switch** (L->bf) {
36. **case** 1://若为1,则表示新节点插在左孩子的左子树上，为LL型
37. (\*p)->bf = L->bf = 0;
38. rightRotate(p);
39. **break**;
40. **case** -1://若为-1,则表示插入到左孩子的右子树上，为LR型
41. R = L->rchild;
42. **switch** (R->bf) {
43. **case** 0:
44. (\*p)->bf = L->bf = 0; **break**;
45. **case** 1:
46. (\*p)->bf = -1; R->bf = L->bf = 0; **break**;
47. **case** -1:
48. (\*p)->bf = R->bf = 0; L->bf = 1; **break**;
50. }
51. leftRotate(&((\*p)->lchild));
52. rightRotate(p);
53. **break**;
54. }
55. }

58. **void** RR\_RLBalance(BiTree\* p) {
59. BiTree R = (\*p)->rchild;
60. BiTree L;
61. **switch** (R->bf) {
62. **case** -1:
63. (\*p)->bf = R->bf = 0;
64. leftRotate(p);
65. **break**;
66. **case** 1:
67. L = R->lchild;
68. **switch** (L->bf) {
69. **case** 0: (\*p)->bf = R->bf = 0; **break**;
70. **case** 1: (\*p)->bf = L->bf = 0; R->bf = -1; **break**;
71. **case** -1: (\*p)->bf = 1; L->bf = R->bf = 0; **break**;
72. }
73. rightRotate(&(\*p)->rchild);
74. leftRotate(p);
75. **break**;
76. }
77. }
79. //输入参数p为根节点指针 key为被插入的关键字 chain表示插入节点后是否引起调整
80. //初始为0
81. **int** insertAVL(BiTree\* p, **int** key, **int**\* chain) {
82. //表示未在树中找到key，直接生成新的节点，用于存储key
83. **if** ((\*p) == NULL) {
84. (\*p) = (BitNode\*)malloc(**sizeof**(BitNode));
85. (\*p)->bf = 0;
86. (\*p)->lchild = (\*p)->rchild = NULL;
87. (\*p)->data = key;
88. \*chain = 1;
89. }
90. **else** {
91. //表示树中有相同的关键字，表示插入失败
92. **if** (key == (\*p)->data) {
93. \*chain = 0;
94. printf("有相同关键字，插入失败！\n");
95. **return** 0;
96. }
97. //表示插入值小于当前的结点key，则递归在当前结点的左子树插入
98. **if** (key < (\*p)->data) {
99. //如果插入不成功
100. **if** (!insertAVL(&(\*p)->lchild, key, chain)) **return** 0;
102. **if** (\*chain) {       //此处在递归调用退出时调用，表示新节点在p左子树插入
103. **switch** ((\*p)->bf) {
105. //if p的平衡因子为0 那么在其左子树插入，将变为1
106. //树的高度增加，会产生平衡引子的变化  不用调整
107. **case** 0:(\*p)->bf = 1; \*chain = 1; **break**;
109. //if p的平衡因子为1,那么在其左子树插入，平衡被破坏，
110. //需要进行调整，调整之后，不会导致树的高度增加，则结束
112. **case** 1: LL\_LRBalance(p); \*chain = 0;  **break**;
114. //if p的平衡因子为-1，那么在其左孩子插入节点，则bf变为0,
115. //树的平衡没有破坏，反应结束
116. **case** -1:(\*p)->bf = 0; \*chain = 0;  **break**;
117. }
118. }
119. }
120. **else**
121. {
122. **if** (!insertAVL(&(\*p)->rchild, key, chain)) **return** 0;
123. **if** (\*chain) {
124. {
125. **switch** ((\*p)->bf) {
126. **case** 0: (\*p)->bf = -1; \*chain = 1; **break**;
127. **case** 1: (\*p)->bf = 0; \*chain = 0; **break**;
128. **case** -1:RR\_RLBalance(p); \*chain = 0; **break**;
129. }
130. }
131. }
132. }
133. }
134. **return** 1;
135. }
137. //中序遍历
138. **void** InOrder(BiTree t) {
139. **if** (t != NULL) {
140. InOrder(t->lchild);
141. printf("%d  ", t->data);
142. InOrder(t->rchild);
143. }
144. }
146. //前序遍历
147. **void** PreOrder(BiTree t) {
148. **if** (t != NULL) {
149. printf("%d  ", t->data);
150. PreOrder(t->lchild);
151. PreOrder(t->rchild);
152. }
153. }
154. **int** main(**void**) {
155. BiTree T = NULL;
156. **int** chain, test, count = 0, height = 0;
157. printf("请输入你想插入的数据：");
158. **while** ((scanf("%d", &test)) != EOF) {
159. count++;
160. insertAVL(&T, test, &chain);
161. **if** (chain)height++;
162. printf("中序遍历的结果：");
163. InOrder(T);
164. printf("\n树高为%d", height);
165. printf("\n请输入你想插入的数据：");
166. }
167. printf("前序遍历的结果：");
168. PreOrder(T);
169. **return** 0;
170. }

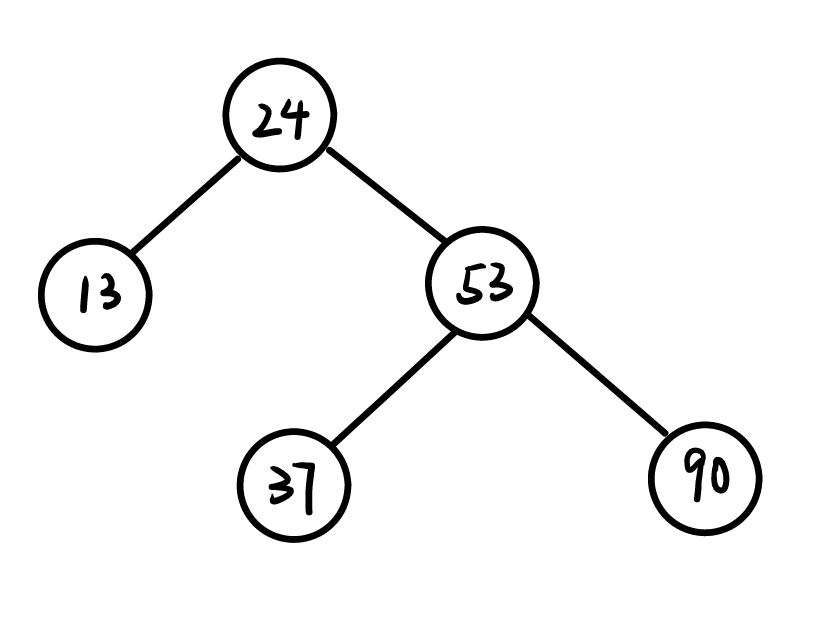
## 2.4 测试用例与结果分析

|  |  |  |
| --- | --- | --- |
|  | 依次输入13、24、37、90、53 | |
| 用例1 |
|  |
|  | 树高 | 3 |
| 预期结 果 | 中序遍历结果 | 13 24 37 53 90 |
|  | 前序遍历结果 | 24 13 53 37 90 |

运行结果如图2-1，建立的AVL树如图2-2



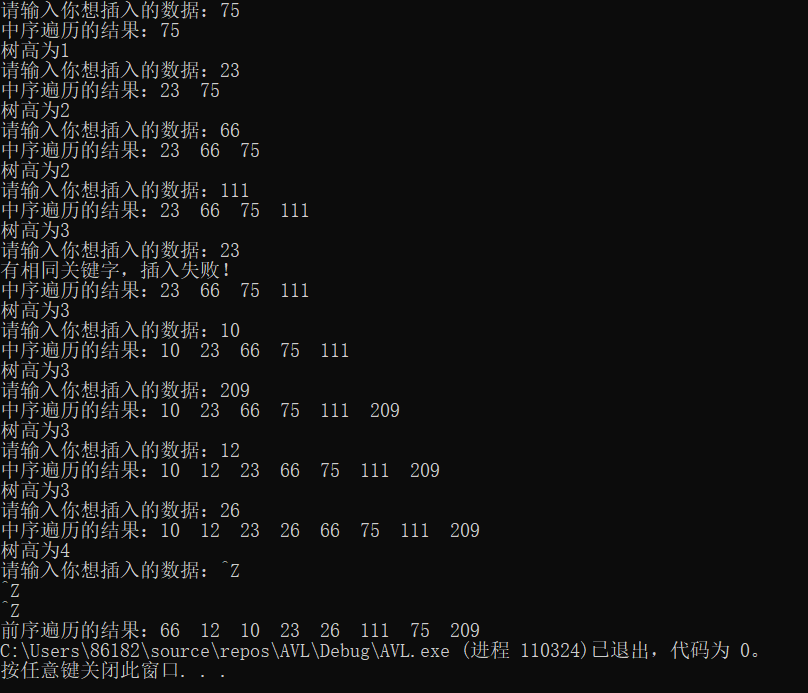
**图2-1 用例一的测试结果**



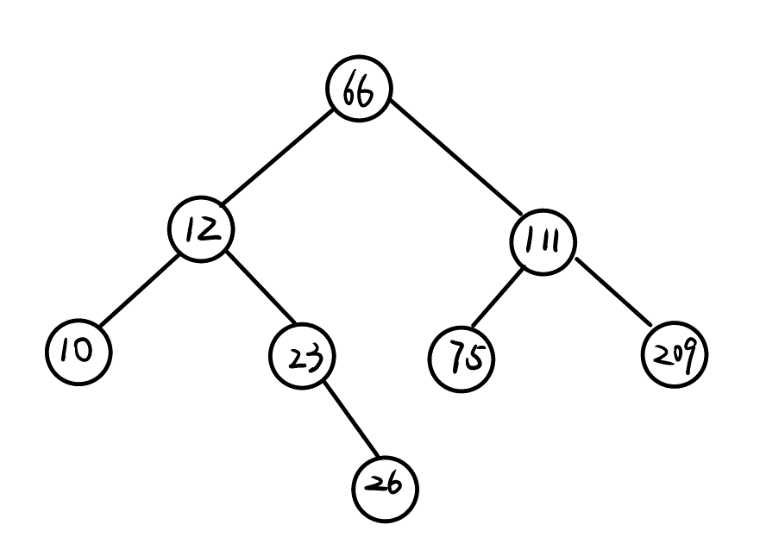
**图2-2 用例一建立的AVL树**

|  |  |  |
| --- | --- | --- |
|  | 依次输入75、23、66、111、23、10、209、12、26 | |
| 用例2 |
|  |
|  | 树高 | 4 |
| 预期结 果 | 中序遍历结果 | 10 12 23 26 66 75 111 209 |
|  | 前序遍历结果 | 66 12 10 23 26 111 75 209 |

运行结果如图2-3，建立的AVL树如图2-4



**图2-3 用例二的测试结果**



**图2-4 用例二建立的AVL树**

# 3图的拓扑排序

## 3.1 问题描述和分析

由某个集合上的一个偏序得到该集合上的一个全序，这个含有环的有向图操作称之为拓扑排序。

图的拓扑排序是对有向无环图来说的，无向图和有环图没有拓扑排序，或者说不存在拓扑排序，对于一个有向无环图G进行拓扑排序，是将G中所有顶点排成一个线性序列，使得图中任意一对顶点u和v,若图G存在边<u,v>，则u在线性序列中出现在v之前。对一个有向无环图进行拓扑排序产生的线性序列称为满足拓扑次序的序列。一个有向无环图可以表示某种动作或者方案或者状态,而有向无环图的拓扑序列通常表示某种某案切实可行或者各个成员之间的某种关系。

## 3.2 设计与实现

进行拓扑排序的方法：

(1)在有向图中选一个没有前驱的顶点且输出之。

(2)从图中删除该顶点和所有以它为尾的弧。

重复上述两步，直至全部顶点均已输出，或者当前图中不存在无前驱的顶点为止。后一种情况则说明有向图中存在环。

采用邻接表作为有向图的存储结构，且在头节点中增加一个存放顶点入度的数组。入度为零的顶点即为没有前驱的顶点，删除顶点及以它为尾的弧的操作，换作以弧头顶点的入度减1来实现。

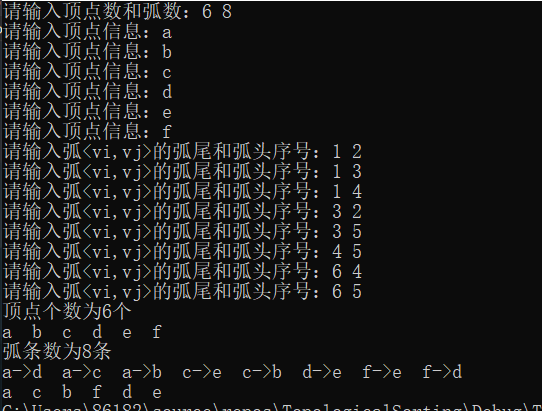
## 3.3 代码清单

1. #include <stdio.h>
2. #include <stdlib.h>
3. #define MaxVertexNum 100
4. **typedef** **char** VertexType;
5. **typedef** **int** ArcType;
6. **typedef** **struct** ArcNode {        //弧结点
7. **int** adjvex;     //该顶点对应的下标
8. **struct** ArcNode\* nextarc;        //指向下一条弧的指针
9. }ArcNode;
10. **typedef** **struct** VertexNode {     //顶点
11. VertexType data;        //顶点信息
12. **int** indegree=0;     //入度
13. **int** flag = 0;       //是否被访问过
14. ArcNode\* firstarc;      //指向第一条弧的指针
15. }VertexNode,AdjList[MaxVertexNum];
16. **typedef** **struct** {
17. AdjList vertex;     //顶点数组
18. **int** vexnum, arcnum;     //图的当前顶点数和弧数
19. }ALGraph;
21. //建立有向图的邻接表
22. **void** CreateALGraph(ALGraph \*G) {
23. **int** i, j, k;
24. ArcNode\* e;
25. printf("请输入顶点数和弧数：");
26. scanf("%d%d", &G->vexnum, &G->arcnum);
28. **for** (i = 1; i <= G->vexnum; i++) {
29. printf("请输入顶点信息：");
30. getchar();
31. scanf("%c", & G->vertex[i].data);
32. G->vertex[i].firstarc = NULL;
33. }
34. **for** (k = 0; k < G->arcnum; k++) {
35. printf("请输入弧<vi,vj>的弧尾和弧头序号：");
36. scanf("%d%d", &i, &j);
37. G->vertex[j].indegree++;     //弧头顶点的入度加1
38. e = (ArcNode\*)malloc(**sizeof**(ArcNode));
39. e->adjvex = j;
40. e->nextarc = G->vertex[i].firstarc;       //向以弧尾顶点为头节点的邻接表中添加弧头顶点
41. G->vertex[i].firstarc = e;
42. }
43. }
45. //输出图
46. **void** PrintGraph(ALGraph G) {
47. printf("顶点个数为%d个\n", G.vexnum);
48. **for** (**int** i = 1; i <= G.vexnum; i++)
49. printf("%c  ", G.vertex[i].data);
50. printf("\n弧条数为%d条\n", G.arcnum);
51. **for** (**int** j = 1; j <= G.vexnum; j++) {
52. **if** (G.vertex[j].firstarc) {
53. ArcNode\* p = G.vertex[j].firstarc;
54. **while** (p) {
55. printf("%c->%c  ", G.vertex[j].data, G.vertex[p->adjvex].data);
56. p = p->nextarc;
57. }
58. }
59. }
60. printf("\n");
61. }
63. //对有向图G拓扑排序
64. **void** TopoSort(ALGraph G) {
65. **int** count = 1;
66. **int** vt[MaxVertexNum];       //存放每次入度为0的顶点序号
67. **for**(**int** m=1;m<=G.vexnum;m++) {
68. **int** i;
69. **for** (i = 1; i <= G.vexnum; i++) {
70. **if** (!G.vertex[i].indegree && !G.vertex[i].flag) {       //该顶点的入度为0且未被访问过
71. vt[count] = i;
72. count++;
73. **break**;
74. }
75. }
76. **if** (i <= G.vexnum) {
77. ArcNode\* p = G.vertex[i].firstarc;
78. **if** (!G.vertex[i].flag) {
79. **while** (p) {
80. G.vertex[p->adjvex].indegree--;     //弧头顶点的入度减1
81. p = p->nextarc;      //删除以此顶点为弧尾的弧
82. }
83. G.vertex[i].flag = 1; //将此顶点标志为已被访问，下次将不再访问
84. }
85. }
86. }
87. **if** (count <= G.vexnum)printf("该有向图存在回路，错误\n");
88. **else** {
89. **for** (**int** k = 1; k < count; k++)
90. printf("%c  ", G.vertex[vt[k]].data);       //输出拓扑排序的结果
91. }
92. }
93. **int** main(**void**) {
94. ALGraph G;
95. CreateALGraph(&G);
96. PrintGraph(G);
97. TopoSort(G);
98. **return** 0;
99. }

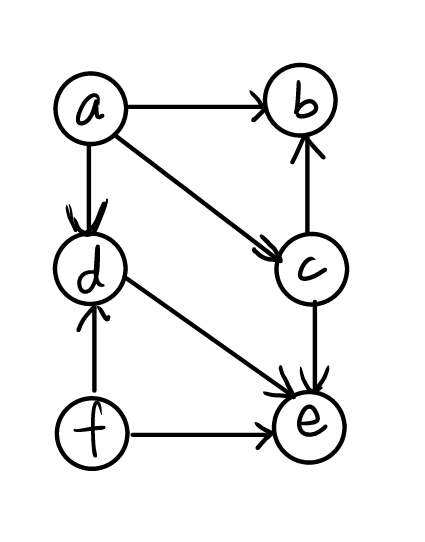
## 3.4 测试用例与结果分析

|  |  |
| --- | --- |
|  | 依次输入6,8,a,b,c,d,e,f,1,2,1,3,1,4,3,2,3,5,4,5,6,4,6,5 |
| 用例1 |
|  |
|  | a c b f d e |
| 预期结 果 |
|  |

运行结果如图3-1，初始建立的有向图如图3-2



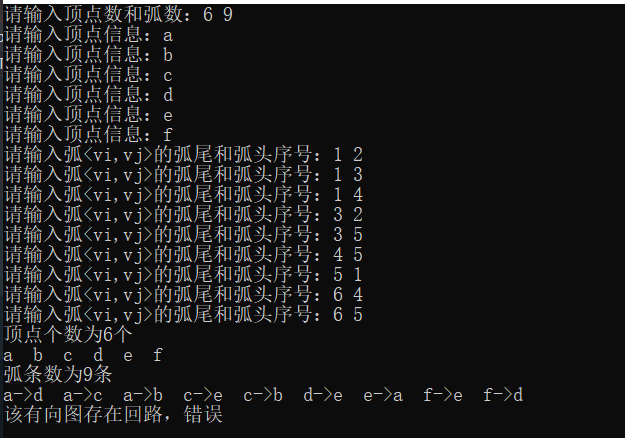
**图3-1 用例一的测试结果**



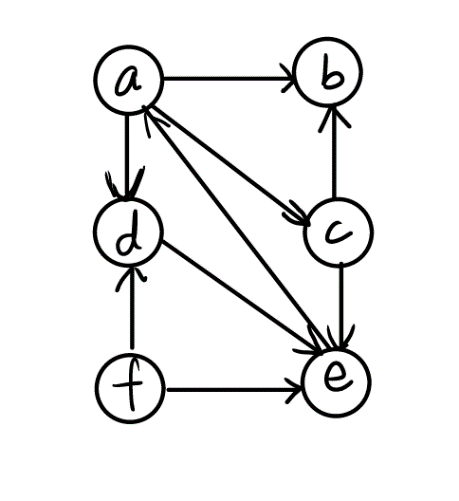
**图3-2 用例一初始建立的有向图**

|  |  |
| --- | --- |
|  | 依次输入6,9,a,b,c,d,e,f,1,2,1,3,1,4,3,2,3,5,4,5,5,1,6,4,6,5 |
| 用例2 |
|  |
|  | 该有向图存在回路，错误 |
| 预期结 果 |
|  |

运行结果如图3-3，初始建立的有向图如图3-4



**图3-3 用例二的测试结果**



**图3-4 用例二初始建立的有向图**

# 4最小编辑距离

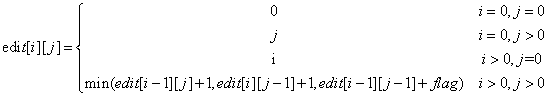
## 4.1 问题描述和分析

编辑距离（Edit Distance），又称Levenshtein距离，是指两个字串之间，由一个转成另一个所需的编辑操作次数。最小编辑距离，是指所需最小的编辑操作次数。编辑操作包含：插入、删除和替换三种操作。

一般来说，两个字符串的编辑距离越小，则它们越相似。如果两个字符串相等，则它们的编辑距离为0（不需要任何操作）。两个字符串的编辑距离肯定不超过它们的最大长度（可以通过先把短串的每一位都修改成长串对应位置的字符，然后插入长串中的剩下字符）。

## 4.2 设计与实现

用edit[i][j]表示A串和B串的编辑距离。edit[i][j]表示A串从第0个字符开始到第i个字符和B串从第0个字符开始到第j个字符这两个字串的编辑距离。字符串的下标从1开始。

dis[0][0]表示word1和word2都为空的时候，此时他们的Edit Distance为0。很明显可以得出的，dis[0][j]就是word1为空，word2长度为j的情况，此时他们的Edit Distance为j，也就是从空添加j个字符转换成word2的最小Edit Distance为j；同理dis[i][0]就是，word1长度为i，word2为空时，word1需要删除i个字符才能转换成空，所以转换成word2的最小EditDistance为i。

上式中的min（）函数中的三个部分，对应三种字符操作方式：

edit[i-1][j]+1相当于给word2的最后插入了word1的最后的字符，插入操作使得edit+1，之后计算edit[i-1][j]；

edit[i][j-1]+1相当于将word2的最后字符删除，删除操作edit+1，之后计算edit[i][j-1];

edit[i-1][j-1]+flag相当于通过将word2的最后一个字符替换为word1的最后一个字符。flag标记替换的有效次数。

## 4.3 代码清单

1. #include <stdio.h>
2. #include <string.h>
3. #include <algorithm>
4. **int** dp[1000][1000];
5. **char** a[1000], b[1000];
6. **int** EditDis(**char**\* a,**char**\* b,**int** dp[][1000])
7. {
8. **int** len1 = strlen(a);
9. **int** len2 = strlen(b);
11. **for** (**int** i = 1; i <= len1; i++)
12. dp[i][0] = i;       //a长度为i,b为空时，a转换成b的最小编辑距离为i
13. **for** (**int** j = 0; j <= len2; j++)
14. dp[0][j] = j;       //a为空，b长度为j时，a转换成b的最小编辑距离为j
15. dp[0][0] = 0;           //a和b都为空时，最小编辑距离为0
16. **for** (**int** i = 1; i <= len1; i++)
17. {
18. **for** (**int** j = 1; j <= len2; j++)
19. {
20. **int** flag;
21. **if** (a[i-1] == b[j-1])
22. flag = 0;
23. **else**
24. flag = 1;
25. dp[i][j] = fmin(dp[i - 1][j] + 1, fmin(dp[i][j - 1] + 1, dp[i - 1][j - 1] + flag));
26. //dp[i-1][j]+1表示删掉字符串a最后一个字符a[i]
27. //dp[i][j-1]+1表示给字符串添加b最后一个字符
28. //dp[i-1][j-1]+flag表示改变,相同则不需操作次数,不同则需要,用flag记录
29. }
30. }
31. **return** dp[len1][len2];
32. }
33. **int** main(**void**) {
34. gets\_s(a);
35. gets\_s(b);
36. **int** dis=EditDis(a, b, dp);
37. printf("%d\n", dis);
38. }

## 4.4 测试用例与结果分析

|  |  |
| --- | --- |
|  | 输入café,coffee |
| 用例1 |
|  |
|  | 3 |
| 预期结 果 |
|  |

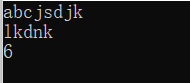
运行结果如图4-1



**图4-1 用例一的测试结果**

|  |  |
| --- | --- |
|  | 输入abcjsdjk,lkdnk |
| 用例2 |
|  |
|  | 6 |
| 预期结 果 |
|  |

运行结果如图4-2



**图4-2 用例二的测试结果**

# 5霍夫曼编码

## 5.1 问题描述和分析

霍夫曼编码(Huffman Coding)是一种编码方法，霍夫曼编码是可变字长编码(VLC)的一种。霍夫曼编码使用变长编码表对源符号（如文件中的一个字母）进行编码，其中变长编码表是通过一种评估来源符号出现机率的方法得到的，出现机率高的字母使用较短的编码，反之出现机率低的则使用较长的编码，这便使编码之后的字符串的平均长度、期望值降低，从而达到无损压缩数据的目的。

## 5.2 设计与实现

求霍夫曼编码的大致过程如下：

(1)统计用户输入的文本中每个字母出现的个数，作为每个结点的权重；

(2)以输入的各个字母为顶点建立霍夫曼树。霍夫曼树采用数组的存储方式，每次选择两个没有双亲结点且权值最小的两个结点构造新的二叉树，根结点的权值为左右结点权值之和；

(3)从叶子结点出发走一条从叶子到根的路径，求出每个字符的霍夫曼编码；

(4)译码的过程则是从根结点出发走一条从根到叶子的路径，按字符’0’或’1’确定找左孩子或右孩子，直至叶子结点，便求得该子串相应的字符。

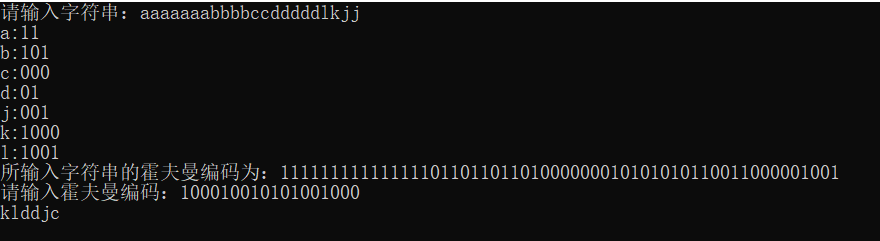
## 5.3 代码清单

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #define  MAX         10000      //最大字符个数
5. #define  MAXVALUE    20000      //权值最大值
6. **typedef** **struct** HuffNode {
7. **char** data;                  //结点的数据，在此为用户输入的字符
8. **int** weight;                 //结点的权值，在此为各个字符出现的次数
9. **int** parent, lchild, rchild; //定义双亲结点、左孩子和右孩子
10. }HuffNode, \* HuffTree;          //动态分配数组存储霍夫曼树
11. **typedef** **char**\*\* HuffCode;        //动态分配数组存储霍夫曼编码表
13. //统计每个字母出现的次数
14. **int** CalWeight(**char**\* str, **char**\* data, **int**\* weight) {
15. **int** i, j, index = 0;
16. **int** arr[26] = { 0 };
17. **for** (i = 0; str[i]; i++) {
18. **if** (str[i] >= 'a' && str[i] <= 'z')
19. arr[str[i] - 'a']++;
20. **else** **if** (str[i] >= 'A' && str[i] <= 'Z')
21. arr[str[i] - 'A']++;
22. }
23. **for** (j = 0; j < 26; j++) {
24. **if** (arr[j]) {
25. data[index] = (**char**)(j + 'a');      //存储所有叶子结点的数据
26. weight[index++] = arr[j];           //存储所有叶子结点的权值
27. }
28. }
29. **return** index;       //返回不同字母的个数
30. }
32. //选出两个没有双亲结点且权值最小的两个结点
33. **void** select(HuffTree HT, **int** k, **int**\* p, **int**\* q) {
34. **int** i, j, index=1;
35. **int** min = MAXVALUE, lmin = MAXVALUE;
36. **for** (i = 1; i <= k; i++) {
37. **if** (HT[i].weight < min && !HT[i].parent) {       //选出最小者
38. min = HT[i].weight;
39. \*p = i;
40. }
41. }
42. **for** (j = 1; j <= k; j++) {
43. **if** (HT[j].weight < lmin && j!=\*p && !HT[j].parent) { //选出次小者
44. lmin = HT[j].weight;
45. \*q = j;
46. }
47. }
49. }
51. /\*建霍夫曼树并求出编码表
52. n为叶子结点的个数
53. d为存放叶子结点数据的数组
54. w为存放叶子结点权值的数组
55. \*/
56. **void** HuffmanCoding(HuffTree& HT, HuffCode& HC, **char**\* d, **int**\* w, **int** n) {
57. **if** (n <= 1)**return**;
58. **int** i, s1, s2,f;
59. **int** m = 2 \* n - 1;      //霍夫曼树的结点个数为叶子结点个数乘2减1
60. **char**\* cd;
61. HT = (HuffTree)malloc((m + 1) \* **sizeof**(HuffNode));  //为霍夫曼树分配存储空间
62. //初始化叶子结点
63. **for** (i = 1; i <= n; i++) {
64. HT[i].data = d[i-1];
65. HT[i].weight = w[i - 1];
66. HT[i].parent = 0;
67. HT[i].lchild = 0;
68. HT[i].rchild = 0;
69. }
70. //初始化非叶子结点
71. **for** (i; i <= m; i++) {
72. HT[i].weight = 0;
73. HT[i].parent = 0;
74. HT[i].lchild = 0;
75. HT[i].rchild = 0;
76. }
77. //建霍夫曼树
78. **for** (i = n + 1; i <= m; i++) {
79. select(HT, i - 1, &s1, &s2);        //选出两个权值最小且没有双亲结点的结点，序号为s1,s2
80. HT[s1].parent = i;
81. HT[s2].parent = i;
82. HT[i].weight = HT[s1].weight + HT[s2].weight;
83. HT[i].lchild = s1;
84. HT[i].rchild = s2;
85. }
86. HC = (HuffCode)malloc((n + 1) \* **sizeof**(**char**\*));
87. cd = (**char**\*)malloc(n \* **sizeof**(**char**));   //cd暂时存放单个字母的霍夫曼编码
88. cd[n - 1] = '\0';
89. **for** (i = 1; i <= n; i++) {
90. **int** start = n - 1;
91. **for** (**int** c = i, f = HT[i].parent; f != 0; c = f, f = HT[f].parent) {
92. **if** (HT[f].lchild == c) cd[--start] = '0';       //从叶子结点向上寻找到一条到根结点的路径
93. **else** cd[--start] = '1';
94. }
95. HC[i] = (**char**\*)malloc((n - start) \* **sizeof**(**char**));
96. strcpy(HC[i], &cd[start]);      //将每个字母的霍夫曼编码复制到编码表中
97. }
98. **for** (i = 1; i <= n; i++)
99. printf("%c:%s\n", HT[i].data, HC[i]);       //输出每个字母的编码
100. }
102. /\*译码
103. code为用户输入的编码
104. letter存放译码的结果
105. \*/
106. **void** HuffmanDecoding(HuffTree& HT, **int** n, **char**\* code, **char**\* letter) {
107. **int** i=0, j=0;
108. HuffNode p = HT[2 \* n - 1];
109. **char**\* q = code;
110. **while** (code[i]) {
111. **while** (p.lchild || p.rchild) {  //从根结点出发按'0'或'1'寻找叶子结点
112. **if** (code[i] == '0')
113. p = HT[p.lchild];
114. **else** **if** (code[i] == '1')
115. p = HT[p.rchild];
116. **else** {
117. printf("输入错误！\n");  //未找到对应的叶子结点
118. **return**;
119. }
120. i++;
121. }
122. letter[j++] = p.data;
123. p = HT[2 \* n - 1];
124. }
125. letter[j] = '\0';
126. printf("%s\n", letter);
127. }
128. **int** main(**void**) {
129. **char** a[MAX],b[MAX],letter[MAX];
130. printf("请输入字符串：");
131. gets\_s(a);
132. **char** data[MAX] = { 0 };
133. **int** weight[MAX] = { 0 };
134. **int** count = CalWeight(a, data, weight);
135. **if** (count <= 1) {
136. printf("输入有误！");
137. **return** -1;
138. }
139. HuffTree HT;
140. HuffCode HC;
141. HuffmanCoding(HT, HC, data, weight, count);
142. printf("所输入字符串的霍夫曼编码为：");
143. **for** (**int** i = 0; a[i]; i++) {        //将用户输入的字符串编码后输出
144. **for** (**int** j = 1; j <= count; j++) {
145. **if** (a[i] == HT[j].data)
146. printf("%s", HC[j]);
147. }
148. }
149. printf("\n请输入霍夫曼编码：");
150. gets\_s(b);
151. HuffmanDecoding(HT, count, b, letter);
152. **return** 0;
153. }

## 5.4 测试用例与结果分析

|  |  |  |
| --- | --- | --- |
| 用例1 | 用户输入的需要编码的字符串 | aaaaaaabbbbccdddddlkjj |
| 用户输入的需要译码的编码串 | 100010010101001000 |
| 预期结 果 | 编码表 | a:11  b:101  c:000  d:01  j:001  k:1000  l:1001 |
| 编码结果 | 11111111111111101101101101000000010101010110011000001001 |
| 译码结果 | klddjc |

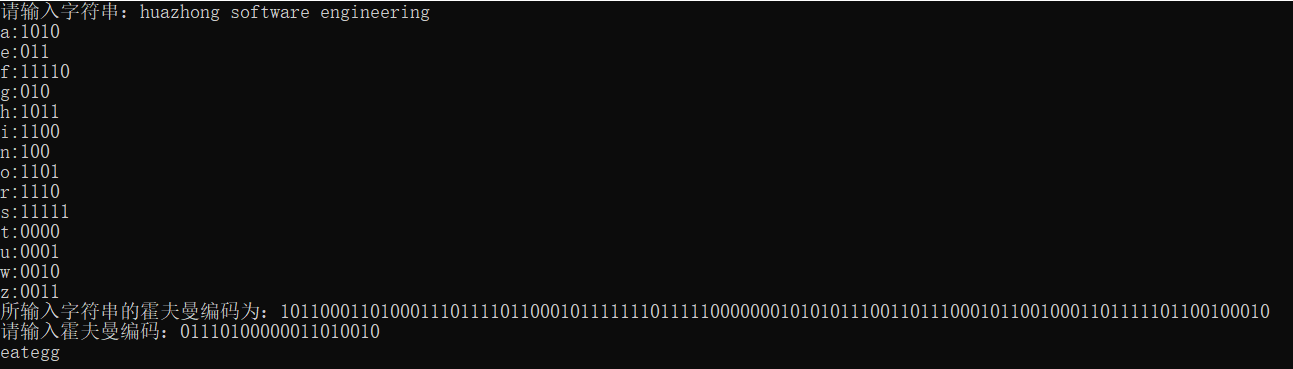
运行结果如图5-1



**图5-1 用例一的测试结果**

|  |  |  |
| --- | --- | --- |
| 用例2 | 用户输入的需要编码的字符串 | huazhong software engineering |
| 用户输入的需要译码的编码串 | 01110100000011010010 |
| 预期结 果 | 编码表 | a:1010  e:011  f:11110  g:010  h:1011  i:1100  n:100  o:1101  r:1110  s:11111  t:0000  u:0001  w:0010  z:0011 |
| 编码结果 | 10110001101000111011110110001011111110111110000000101010111001  1011100010110010001101111101100100010 |
| 译码结果 | eategg |

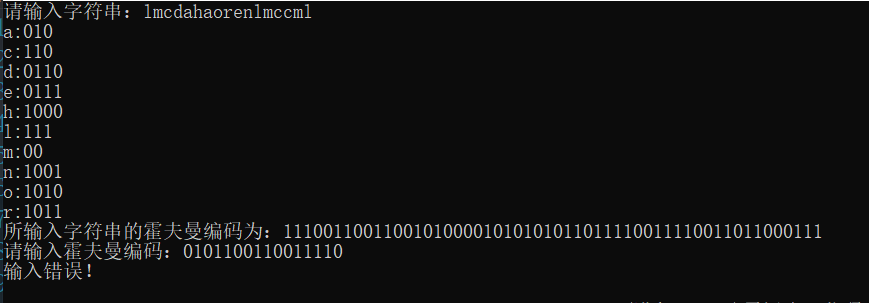
运行结果如图5-2



**图5-2 用例二的测试结果**

|  |  |  |
| --- | --- | --- |
| 用例3 | 用户输入的需要编码的字符串 | lmcdahaorenlmccml |
| 用户输入的需要译码的编码串 | 0101100110011110 |
| 预期结 果 | 编码表 | a:010  c:110  d:0110  e:0111  h:1000  l:111  m:00  n:1001  o:1010  r:1011 |
| 编码结果 | 111001100110010100001010101011011110011110011011000111 |
| 译码结果 | 输入错误！ |

运行结果如图5-3



**图5-3 用例三的测试结果**

# 6有向图最短路径的Dijsktra算法

## 6.1 问题描述和分析

狄克斯特拉算法解决图中一点到其余各点到最短路径的问题。其基本思想为：图G=(V,E)是一个有权有向图，把顶点V分成两组，第一组为已求出最短路径的点的集合(用S表示，初始时S中只有一个源点，以后每求得一条最短路径v,…k，就将k加入到集合S中，直到全部到点都加入S集合中，算法结束)，第二组为其余未确定最短路径的点的集合(用U表示)，按最短路径长度的递增次序依次把第二组的顶点加入到S中。

## 6.2 设计与实现

算法大致如下：

(1) 假设用带权的邻接矩阵arcs 来表示带权有向图，arcs[i][j]表示弧<vi,vj>上的权值。若<vi,vj>不存在，则置arcs[i][j]为INF（在计算机上可用允许的最大值代替）。S为已找到从v出发的最短路径的终点的集合，它的初始状态为空集。那么，从v 出发到图上其余各顶点（终点）vi，可能达到的最短路径长度的初值为：

D[i]=arcs[Locate Vex(G,v) [i] ；

(2) 选择vj，使得D[j] = Min{D[i]}，vj就是当前求得的一条从v出发的最短路径的终点。令s = s ∪ {j}；

(3) 修改从v 出发到集合V-S 上任一顶点Vk可达的最短路径长度。如果

D[j] + arcs [j][k] < D[k]，则修改D[k]为D[k] = D[j] + arcs[j][k]

(4) 重复操作(2)、(3) 共n-1次。由此求得从v 到图上其余各顶点的最短路径是依路径长度递增的序列。

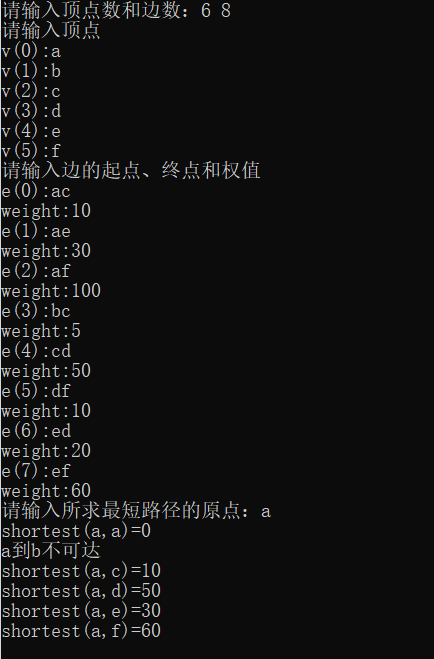
## 6.3 代码清单

1. #include <stdio.h>
2. #include <stdlib.h>
4. #define MAX   100
5. #define INF   (((unsigned int)(-1)) >> 1)
7. //邻接矩阵定义
8. **typedef** **struct** graph {
9. **char** vexs[MAX];         //顶点集合
10. **int** vexnum, edgnum;     //定点数和边数
11. **int** matrix[MAX][MAX];   //邻接矩阵
12. }Graph, \* PGraph;
14. //边定义
15. **typedef** **struct** EdgeData{
16. **char** start, end;        //边的起点和终点
17. **int** weight;             //边的权重
18. }Edata;
20. //返回ch在matrix矩阵中的位置
21. **static** **int** get\_position(Graph G, **char** ch){
22. **int** i;
23. **for** (i = 0; i < G.vexnum; i++)
24. **if** (G.vexs[i] == ch)
25. **return** i;
26. **return** -1;
27. }
29. //创建图
30. Graph\* CreatGraph() {
31. **char** c1, c2;
32. **int** vex, edg;
33. **int** i, j, weight, p1, p2;
34. Graph\* graph;
35. graph = (Graph\*)malloc(**sizeof**(Graph));
36. //输入顶点数和边数
37. printf("请输入顶点数和边数：");
38. scanf("%d%d", &vex, &edg);
39. getchar();
40. **if** (vex < 1 || edg < 1 || (edg > (vex \* (vex - 1)))) {
41. printf("输入不符合要求！\n");
42. **return** NULL;
43. }
44. graph->vexnum = vex;
45. graph->edgnum = edg;
46. //输入顶点
47. printf("请输入顶点\n");
48. **for** (i = 0; i < graph->vexnum; i++) {
49. printf("v(%d):",i);
50. scanf("%c", &graph->vexs[i]);
51. getchar();
52. }
53. //初始化边的权值
54. **for** (i = 0; i < graph->vexnum; i++) {
55. **for** (j = 0; j < graph->vexnum; j++) {
56. **if** (i == j)
57. graph->matrix[i][j] = 0;
58. **else**
59. graph->matrix[i][j] = INF;
60. }
61. }
62. //输入边的起点、终点和权值
63. printf("请输入边的起点、终点和权值\n");
64. **for** (i = 0; i < graph->edgnum; i++) {
65. printf("e(%d):", i);
66. scanf("%c%c", &c1, &c2);
67. getchar();
68. printf("weight:");
69. scanf("%d", &weight);
70. getchar();
71. p1 = get\_position(\*graph, c1);
72. p2 = get\_position(\*graph, c2);
73. **if** (p1 == -1 || p2 == -1){
74. printf("边不符合要求！\n");
75. free(graph);
76. **return** NULL;
77. }
78. graph->matrix[p1][p2] = weight;
79. }
80. **return** graph;
82. }
84. /\*
85. Dijkstra
86. v0 -- 起始顶点。即计算"顶点v0"到其它顶点的最短路径。
87. pre -- 前驱顶点数组,即pre[i]的值是"顶点v0"到"顶点i"的最短路径所经历的全部顶点中，位于"顶点i"之前的那个顶点。
88. dijk -- 长度数组,即dijk[i]是"顶点v0"到"顶点i"的最短路径的长度。
89. \*/
90. **void** Dijkstra(Graph graph,**char** v0,**int** pre[],**int** dijk[]) {
91. **int** i, j, k= get\_position(graph, v0), min, tmp, flag[MAX];      //flag[i]==1表示v0到vi的最短路径已成功获取
92. **int** p0 = get\_position(graph, v0);
93. //初始化
94. **for** (i = 0; i < graph.vexnum; i++) {
95. flag[i] = 0;
96. pre[i] = 0;
97. dijk[i] = graph.matrix[p0][i];
98. }
99. flag[p0] = 1;
100. dijk[p0] = 0;
102. **for** (i = 0; i < graph.vexnum; i++) {
103. min = INF;
104. **for** (j = 0; j < graph.vexnum; j++) {
105. **if** (!flag[j] && dijk[j] < min) {
106. min = dijk[j];
107. k = j;
108. }
109. }
110. flag[k] = 1;
111. **for** (j = 0; j < graph.vexnum; j++) {
112. tmp = graph.matrix[k][j] == INF ? INF : (min + graph.matrix[k][j]);
113. **if** (!flag[j] && tmp < dijk[j]) {
114. dijk[j] = tmp;
115. pre[j] = k;
116. }
117. }
118. }
120. **for** (j = 0; j < graph.vexnum; j++) {
121. **if** (dijk[j] != INF)
122. printf("shortest(%c,%c)=%d\n", v0, graph.vexs[j], dijk[j]);
123. **else** printf("%c到%c不可达\n", v0, graph.vexs[j]);
124. }
125. }
127. **int** main(**void**) {
128. **if** (Graph\* graph = CreatGraph()) {
129. **char** v0;
130. **int** pre[MAX] = { 0 }, dijk[MAX] = { 0 };
131. printf("请输入所求最短路径的原点：");
132. scanf("%c", &v0);
133. getchar();
134. Dijkstra(\*graph, v0, pre, dijk);
135. }
136. **return** 0;
137. }

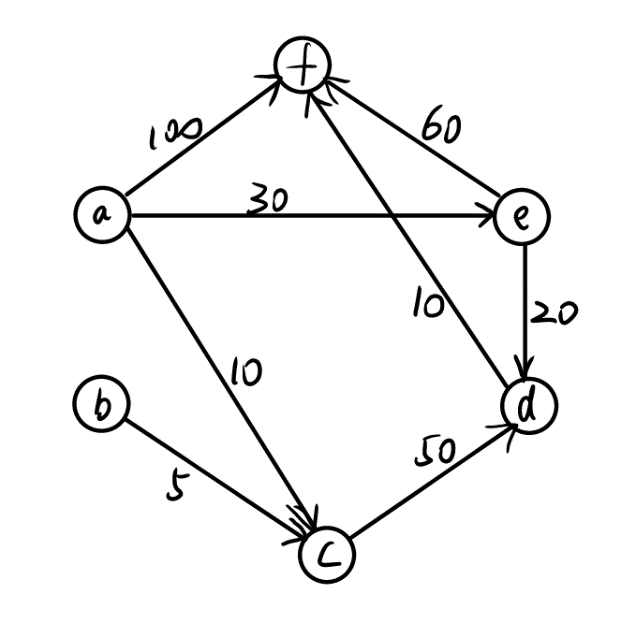
## 6.4 测试用例与结果分析

|  |  |
| --- | --- |
| 用例1 | 依次输入6,8,a,b,c,d,e,f,ac,10,ae,30,af,100,bc,5,cd,50,df,10,ed,20,ef,60,a  所求最短路径的原点为a |
| 预期结 果 | shortest(a,a)=0  a到b不可达  shortest(a,c)=10  shortest(a,d)=50  shortest(a,e)=30  shortest(a,f)=60 |

运行结果如图6-1，初始建立的有向图如图6-2



**图6-1 用例一的测试结果**



**图6-2 用例一初始建立的有向图**

# 7快速排序

## 7.1 问题描述和分析

快速排序(Quick Sort)是对起泡排序的一种改进。它的基本思想是，通过一趟排序将待排记录分割成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

## 7.2 设计与实现

快速排序使用分治法策略来把一个序列分为两个子序列。

算法思路如下：

① 从待排序列中挑出一个元素，称为“基准”（pivot）。

② 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。

③ 递归地把小于基准值元素的子数列和大于基准值元素的子数列排序。

递归到最底部时，数列的大小是零或一，也就是已经排序好了。这个算法一定会结束，因为在每次的迭代中，它至少会把一个元素摆到它最后的位置去。

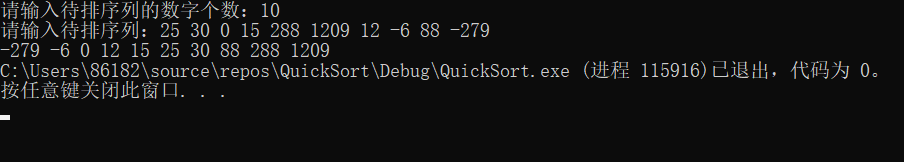
## 7.3 代码清单

1. #include <stdio.h>;
2. **void** quicksort(**int**\* arr, **int** low, **int** high);
3. **int** main(**void**) {
4. **int** arr[10000],N;
5. printf("请输入待排序列的数字个数：");
6. scanf\_s("%d", &N);      //输入待排序列的数字个数
7. printf("请输入待排序列：");
8. **for**(**int** i=0;i<N;i++)
9. scanf\_s("%d", &arr[i]);     //输入待排序列，存放在数组arr中
10. quicksort(arr, 0, N-1);
11. **for** (**int** j = 0; j < N; j++)
12. printf("%d ", arr[j]);
13. **return** 0;
14. }
16. **void** quicksort(**int**\* arr, **int** low, **int** high) {
17. **if** (low >= high)**return**;      //一趟快排结束的标志
18. **int** pivot = arr[low], i = low, j = high;    //设定基准为第一个元素
19. **while** (i<j)
20. {
21. **while** (arr[j]>=pivot && i<j) j--; //在右子序列找到比基准值小的元素
22. arr[i] = arr[j];
23. **while** (arr[i]<=pivot && i<j) i++; //在左子序列找到比基准值大的元素
24. arr[j] = arr[i];
25. }
26. arr[i] = pivot;             //将基准值赋给i元素
27. quicksort(arr, low, i);     //递归快排左序列
28. quicksort(arr, i+1, high);  //递归快排右序列
29. **return**;
30. }

## 7.4 测试用例与结果分析

|  |  |
| --- | --- |
| 用例1 | 待排序列为25,30,0,15,288,1209,12,-6,88,-279 |
| 预期结 果 | -279 -6 0 12 15 25 30 88 288 1209 |

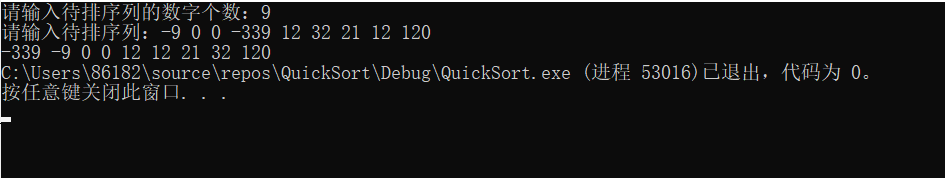
运行结果如图7-1



**图7-1 用例一的测试结果**

|  |  |
| --- | --- |
| 用例2 | 待排序列为-9,0,0,-339,12,32,21,12,120 |
| 预期结 果 | -339 -9 0 0 12 12 21 32 120 |

运行结果如图7-2



**图7-2 用例二的测试结果**

# 8最小生成树

## 8.1 问题描述和分析

现在假设有一个很实际的问题：我们要在n个城市中建立一个通信网络，则连通这n个城市需要布置n-1一条通信线路，这个时候我们需要考虑如何在成本最低的情况下建立这个通信网？

于是我们就可以引入连通图来解决我们遇到的问题，n个城市就是图上的n个顶点，然后，边表示两个城市的通信线路，每条边上的权重就是我们搭建这条线路所需要的成本，所以现在我们有n个顶点的连通网可以建立不同的生成树，每一颗生成树都可以作为一个通信网，当我们构造这个连通网所花的成本最小时，搭建该连通网的生成树，就称为最小生成树。

构造最小生成树有很多算法，但是他们都是利用了最小生成树的同一种性质：MST性质（假设N=(V,{E})是一个连通网，U是顶点集V的一个非空子集，如果（u，v）是一条具有最小权值的边，其中u属于U，v属于V-U，则必定存在一颗包含边（u，v）的最小生成树），下面就介绍两种使用MST性质生成最小生成树的算法：普里姆算法和克鲁斯卡尔算法。

## 8.2 设计与实现

本次采用克鲁斯卡尔算法来构造最小生成树。假设连通网N=(V,{E})，则令最小生成树的初始状态为只有n个顶点而无边的非连通图T=(V,{}) ，图中每个顶点自成一个连通分量。在E中选择代价最小的边，若该边依附的顶点落在T 中不同的连通分量上，则将此边加入到T中，否则舍去此边而选择下一条代价最小的边。依次类推，直至T中所有顶点都在同一连通分量上为止。

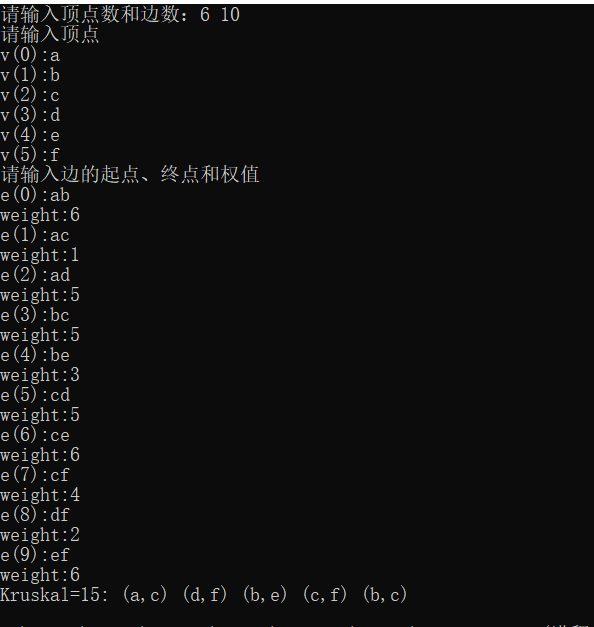
## 8.3 代码清单

1. #include <stdio.h>
2. #include <stdlib.h>
4. #define MAX   100
5. #define INF   (((unsigned int)(-1)) >> 1)
7. //邻接矩阵定义
8. **typedef** **struct** graph {
9. **char** vexs[MAX];         //顶点集合
10. **int** vexnum, edgnum;     //定点数和边数
11. **int** matrix[MAX][MAX];   //邻接矩阵
12. }Graph, \* PGraph;
14. //边定义
15. **typedef** **struct** EdgeData {
16. **char** start, end;        //边的起点和终点
17. **int** weight;             //边的权重
18. }Edata;
20. //返回ch在matrix矩阵中的位置
21. **static** **int** get\_position(Graph G, **char** ch) {
22. **int** i;
23. **for** (i = 0; i < G.vexnum; i++)
24. **if** (G.vexs[i] == ch)
25. **return** i;
26. **return** -1;
27. }
29. //创建图
30. Graph\* CreatGraph() {
31. **char** c1, c2;
32. **int** vex, edg;
33. **int** i, j, weight, p1, p2;
34. Graph\* graph;
35. graph = (Graph\*)malloc(**sizeof**(Graph));
36. //输入顶点数和边数
37. printf("请输入顶点数和边数：");
38. scanf("%d%d", &vex, &edg);
39. getchar();
40. **if** (vex < 1 || edg < 1 || (edg > (vex \* (vex - 1)))) {
41. printf("输入不符合要求！\n");
42. **return** NULL;
43. }
44. graph->vexnum = vex;
45. graph->edgnum = edg;
46. //输入顶点
47. printf("请输入顶点\n");
48. **for** (i = 0; i < graph->vexnum; i++) {
49. printf("v(%d):", i);
50. scanf("%c", &graph->vexs[i]);
51. getchar();
52. }
53. //初始化边的权值
54. **for** (i = 0; i < graph->vexnum; i++) {
55. **for** (j = 0; j < graph->vexnum; j++) {
56. **if** (i == j)
57. graph->matrix[i][j] = 0;
58. **else**
59. graph->matrix[i][j] = INF;
60. }
61. }
62. //输入边的起点、终点和权值
63. printf("请输入边的起点、终点和权值\n");
64. **for** (i = 0; i < graph->edgnum; i++) {
65. printf("e(%d):", i);
66. scanf("%c%c", &c1, &c2);
67. getchar();
68. printf("weight:");
69. scanf("%d", &weight);
70. getchar();
71. p1 = get\_position(\*graph, c1);
72. p2 = get\_position(\*graph, c2);
73. **if** (p1 == -1 || p2 == -1) {
74. printf("边不符合要求！\n");
75. free(graph);
76. **return** NULL;
77. }
78. graph->matrix[p1][p2] = weight;
79. graph->matrix[p2][p1] = weight;
80. }
81. **return** graph;
82. }
84. //返回所有的边
85. Edata\* get\_edges(Graph graph) {
86. **int** i, j, index=0;
87. Edata\* edges = (Edata\*)malloc(graph.edgnum \* **sizeof**(Edata));
88. **for** (i = 0; i < graph.vexnum; i++) {
89. **for** (j = i + 1; j < graph.vexnum; j++) {
90. **if** (graph.matrix[i][j] != INF) {
91. edges[index].start = graph.vexs[i];
92. edges[index].end = graph.vexs[j];
93. edges[index].weight = graph.matrix[i][j];
94. index++;
95. }
96. }
97. }
98. **return** edges;
99. }
101. //按边的权值对边进行排序
102. **void** sort\_edges(Edata\* edges, **int** n) {
103. **for** (**int** i = 0; i < n - 1; i++) {
104. **for** (**int** j = 0; j < n - 1 - i; j++) {
105. **if** (edges[j].weight > edges[j + 1].weight) {
106. Edata temp = edges[j];
107. edges[j] = edges[j + 1];
108. edges[j + 1] = temp;
109. }
111. }
112. }
113. }
115. //获取边在已有最小生成树中的起点
116. **int** get\_end(**int** vends[MAX], **int** i)
117. {
118. **while** (vends[i]!=-1)
119. i = vends[i];
120. **return** i;
121. }
123. //kruskal
124. **void** kruskal(Graph graph) {
125. **int** i, m, n, p1, p2;
126. **int** tot = 0;;
127. **int** index = 0;          // rets数组的索引
128. **int** vends[MAX];     // 用于保存"已有最小生成树"中每个顶点在该最小树中的起点。
129. Edata rets[MAX];        // 结果数组，保存kruskal最小生成树的边
130. Edata\* edges;           // 图对应的所有边
131. **for** (**int** k = 0; k < MAX; k++)
132. vends[k] = -1;
133. edges = get\_edges(graph);
134. sort\_edges(edges, graph.edgnum);
136. **for** (i = 0; i < graph.edgnum; i++) {
137. p1 = get\_position(graph, edges[i].start);
138. p2 = get\_position(graph, edges[i].end);
139. m = get\_end(vends, p1);
140. n = get\_end(vends, p2);
141. **if** (m != n) {
142. vends[n] = m;
143. rets[index++] = edges[i];
144. }
145. }
146. //free(edges);
147. //打印"kruskal最小生成树"的信息
148. **for** (i = 0; i < index; i++)
149. tot += rets[i].weight;
150. printf("Kruskal=%d: ", tot);
151. **for** (i = 0; i < index; i++)
152. printf("(%c,%c) ", rets[i].start, rets[i].end);
153. printf("\n");
154. }
156. **int** main(**void**) {
157. Graph\* graph = CreatGraph();
158. kruskal(\*graph);
159. }

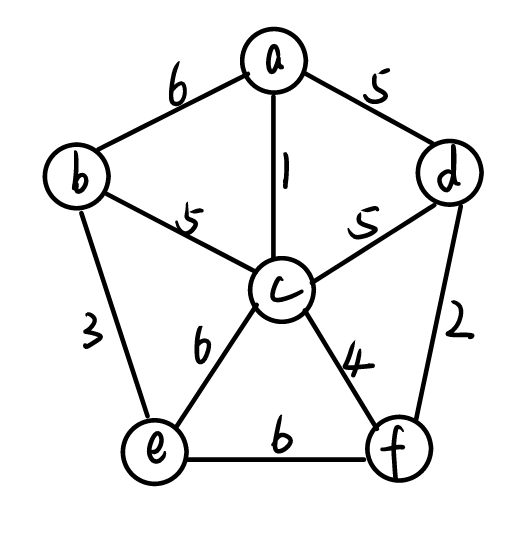
## 8.4 测试用例与结果分析

|  |  |
| --- | --- |
| 用例1 | 依次输入6,10,a,b,c,d,e,f,ab,6,ac,1,ad,5,bc,5,be,3,cd,5,ce,6,cf,4,df,2,ef,6 |
| 预期结 果 | Kruskal=15: (a,c) (d,f) (b,e) (c,f) (b,c) |

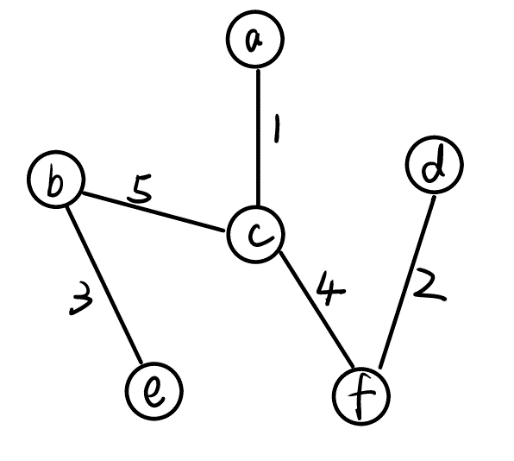
运行结果如图8-1，建立的初始图如图8-2，构造的最小生成树如图8-3



**图8-1 用例一的测试结果**



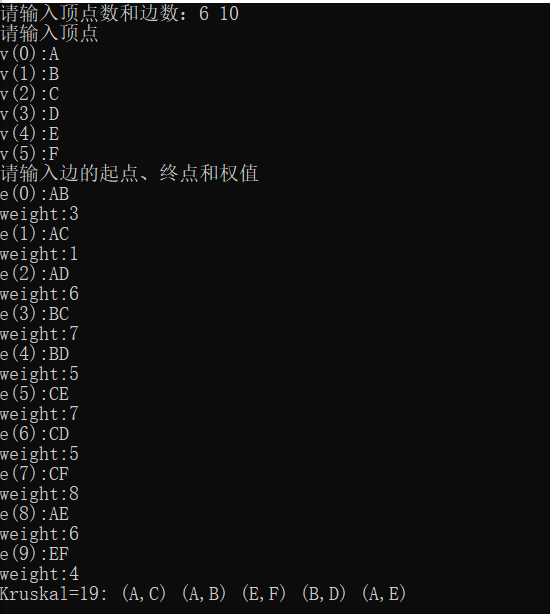
**图8-2 用例一建立的初始图**



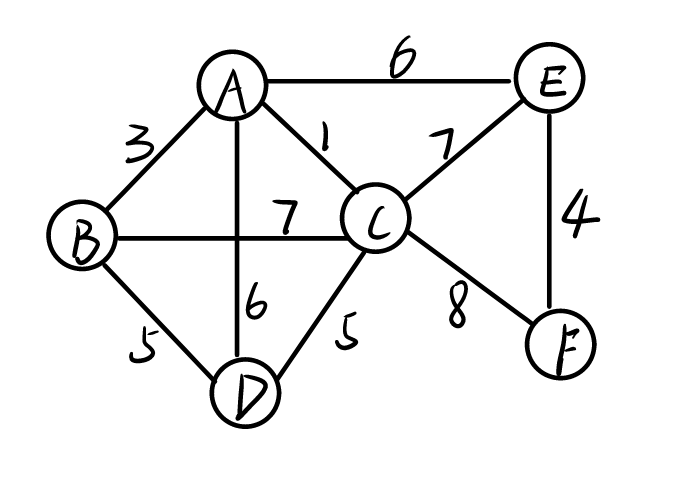
**图8-3 用例一构造的最小生成树**

|  |  |
| --- | --- |
| 用例2 | 依次输入6,10,A,B,C,D,E,F,AB,3,AC,1,AD,6,BC,7,BD,5,CE,7,CD,5,CF,8,AE,6,EF,4 |
| 预期结 果 | Kruskal=19: (A,C) (A,B) (E,F) (B,D) (A,E) |

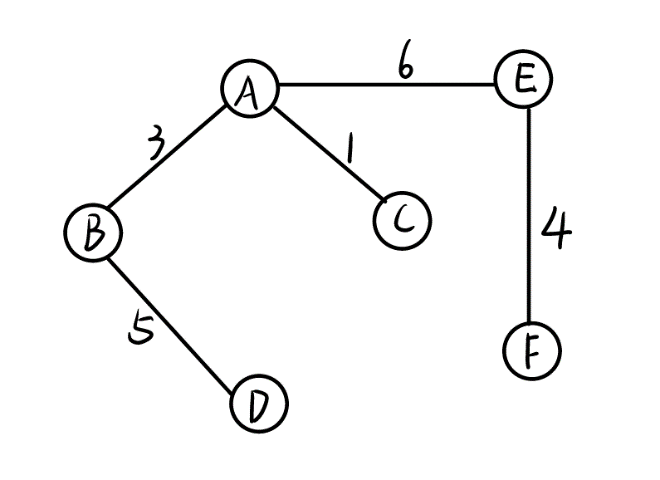
运行结果如图8-4，建立的初始图如图8-5，构造的最小生成树如图8-6



**图8-4 用例二的测试结果**



**图8-5 用例二建立的初始图**



**图8-6 用例二构造的最小生成树**