**数据结构第六次上机实验报告**

学号：161730126 姓名：李双玖

1. 调试成功程序及说明

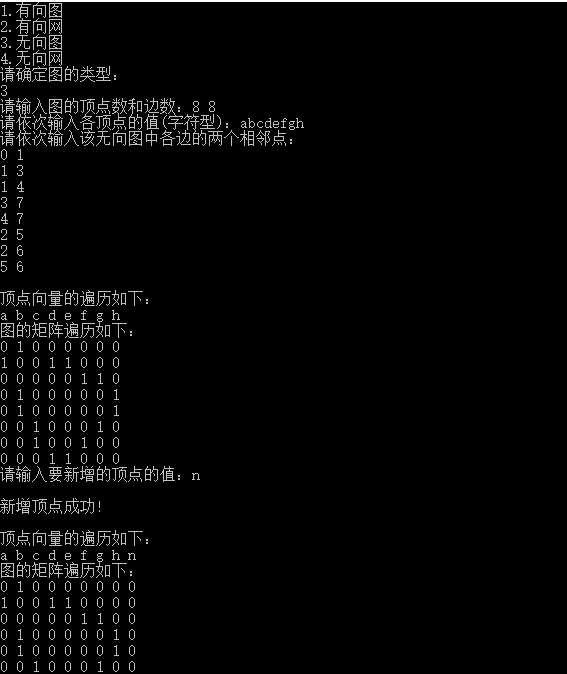
**1、**

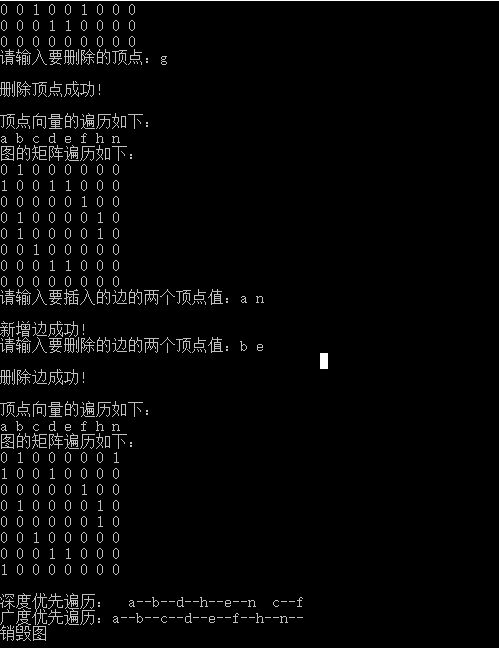
**题目：**编程实现书P156 ADT Graph 基本操作13个，用邻接矩阵存储结构实现；

**算法思想：**1）深度优遍历。数据结构为栈，同时定义一个长度与顶点数目相同的bool类型数组来记录每个点是否被走过。每次判断栈顶元素的邻接点是否走过，未走过则入栈，都走过则该栈顶元素出栈。2）广度优先遍历，数据结构为队列，同样思路，只是换成队列。

另外，不管是深度还是广度，都要从每个没走过的点都出发一遍，防止不连通的情况。

**运行结果：**





**结果分析：**成功。

**源程序：**

1. /\*
2. 编程实现书P156  ADT Graph 基本操作13个，用邻接矩阵存储结构实现；
3. 1.有向图
4. 2.有向网
5. 3.无向图
6. 4.无向网
7. 请确定图的类型：
8. 3
9. 请输入图的顶点数和边数：5 7
10. 请依次输入各顶点的值(字符型)：a b c d e
11. 请依次输入该无向图中各边的两个相邻点：
12. 0 1
13. 0 3
14. 0 4
15. 1 2
16. 1 3
17. 2 4
18. 3 4
20. 请输入图的顶点数和边数：8 9
21. 0 1
22. 0 2
23. 1 3
24. 1 4
25. 3 7
26. 4 7
27. 2 5
28. 2 6
29. 5 6
31. 不连通图
32. 请输入图的顶点数和边数：8 8
33. 0 1
34. 1 3
35. 1 4
36. 3 7
37. 4 7
38. 2 5
39. 2 6
40. 5 6
41. \*/
42. #include<iostream>
43. #include<stdlib.h>
44. #include<stack>
45. #include<queue>
46. **using** **namespace** std;

49. // 图的类型{有向图，有向网(带权图)，无向图， 无向网}
50. **typedef** **enum** {DG, DN, UDG, UDN} GraphKind;

53. // 边(不使用)
54. **typedef** **struct** ArcCell {
55. **int** adj;    // 对于无权图，用1或0表示相邻与否；对于带权图，则为权值。
56. **int** \*info;  // 该边对应的信息
57. }ArcCell, AdjMatrix[20][20];

60. // 图
61. **typedef** **struct** {
62. **char** \*vexs;         // 顶点值的向量
63. **int** vexNum, arcNum;         // 图的顶点数和边数
64. **int** \*\*arcs; // 边的矩阵
65. GraphKind kind;             // 图的类型
66. }MGraph;

69. // 构建有向图
70. **void** CreateDG(MGraph &G) {
71. cout<<"请依次输入该有向图中各边的始点、终点：\n";
73. **int** x, y;
74. **for** (**int** i=0; i<G.arcNum; i++) {
75. cin>>x>>y;
76. **if** (x == y) {
77. cout<<"不能出现环！\n";
78. i--;
79. **continue**;
80. }
81. **if** (x == G.vexNum) {
82. cout<<"不存在该点！\n";
83. i--;
84. **continue**;
85. }
86. **if** (y == G.vexNum) {
87. cout<<"不存在该点！\n";
88. i--;
89. **continue**;
90. }
91. G.arcs[x][y] = 1;
92. }
93. }

96. // 构建有向网
97. **void** CreateDN(MGraph &G) {
98. cout<<"请依次输入该有向网中各边的始点、终点和权值：\n";
100. **int** x, y, weight;
101. **for** (**int** i=0; i<G.arcNum; i++) {
102. cin>>x>>y>>weight;
103. **if** (x == y) {
104. cout<<"不能出现环！\n";
105. i--;
106. **continue**;
107. }
108. **if** (x == G.vexNum) {
109. cout<<"不存在该点！\n";
110. i--;
111. **continue**;
112. }
113. **if** (y == G.vexNum) {
114. cout<<"不存在该点！\n";
115. i--;
116. **continue**;
117. }
118. **if** (weight == 0) {
119. cout<<"权重不能为0！\n";
120. i--;
121. **continue**;
122. }
123. G.arcs[x][y] = weight;
124. }
125. }

128. // 构建无向图
129. **void** CreateUDG(MGraph &G) {
130. cout<<"请依次输入该无向图中各边的两个相邻点：\n";
132. **int** x, y;
133. **for** (**int** i=0; i<G.arcNum; i++) {
134. cin>>x>>y;
135. **if** (x == y) {
136. cout<<"不能出现环！\n";
137. i--;
138. **continue**;
139. }
140. **if** (x == G.vexNum) {
141. cout<<"不存在该点！\n";
142. i--;
143. **continue**;
144. }
145. **if** (y == G.vexNum) {
146. cout<<"不存在该点！\n";
147. i--;
148. **continue**;
149. }
150. G.arcs[x][y] = 1;
151. G.arcs[y][x] = 1;
152. }
153. }

156. // 构建有向网
157. **void** CreateUDN(MGraph &G) {
158. cout<<"请依次输入该无向网中各边的两个相邻点和权值：\n";
160. **int** x, y, weight;
161. **for** (**int** i=0; i<G.arcNum; i++) {
162. cin>>x>>y>>weight;
163. **if** (x == y) {
164. cout<<"不能出现环！\n";
165. i--;
166. **continue**;
167. }
168. **if** (x == G.vexNum) {
169. cout<<"不存在该点！\n";
170. i--;
171. **continue**;
172. }
173. **if** (y == G.vexNum) {
174. cout<<"不存在该点！\n";
175. i--;
176. **continue**;
177. }
178. **if** (weight == 0) {
179. cout<<"权重不能为0！\n";
180. i--;
181. **continue**;
182. }
183. G.arcs[x][y] = weight;
184. G.arcs[y][x] = weight;
185. }
186. }

189. // 构建图G
190. **void** CreateGraph(MGraph &G) {
191. **int** choose;
192. cout<<"1.有向图\n2.有向网\n3.无向图\n4.无向网\n请确定图的类型：\n";
193. cin>>choose;
195. cout<<"请输入图的顶点数和边数：";
196. cin>>G.vexNum>>G.arcNum;
198. // 动态初始化图，将每一条边的值设为0
199. G.arcs = (**int**\*\*)malloc(**sizeof**(**int**\*) \* G.vexNum);
200. **for** (**int** i=0; i<G.vexNum; i++) {
201. G.arcs[i] = (**int**\*)malloc(**sizeof**(**int**) \* G.vexNum);
202. **for** (**int** j=0; j<G.vexNum; j++) {
203. G.arcs[i][j] = 0;       // 用0表示这两条边不相邻
204. }
205. }
207. // 初始化顶点值
208. G.vexs = (**char**\*)malloc(**sizeof**(**char**) \* G.vexNum);
209. cout<<"请依次输入各顶点的值(字符型)：";
210. **for** (**int** i=0; i<G.vexNum; i++) {
211. cin>>G.vexs[i];
212. }
214. **switch**(choose) {
215. **case** 1: {
216. G.kind = DG;
217. CreateDG(G);
218. **break**;
219. }
220. **case** 2: {
221. G.kind = DN;
222. CreateDN(G);
223. **break**;
224. }
225. **case** 3: {
226. G.kind = UDG;
227. CreateUDG(G);
228. **break**;
229. }
230. **case** 4: {
231. G.kind = UDN;
232. CreateUDN(G);
233. **break**;
234. }
235. }
236. }

239. // 销毁图
240. **void** DestroyGraph(MGraph &G) {
241. cout<<"\n销毁图\n";
242. free(G.vexs);
243. **for** (**int** i=0; i<G.vexNum; i++) {
244. free(G.arcs[i]);
245. }
246. free(G.arcs);
247. G.vexNum = G.arcNum = 0;
248. }

251. // 返回顶点位置
252. **int** LocateVex(MGraph &G, **char** value) {
253. **for** (**int** i=0; i<G.vexNum; i++) {
254. **if** (G.vexs[i] == value) {
255. **return** i;
256. }
257. }
258. **return** -1;
259. }

262. // 返回顶点的值
263. **char** GetVex(MGraph &G, **int** v) {
264. **return** G.vexs[v];
265. }

268. // 设置顶点的值
269. **void** PutVex(MGraph &G, **int** v, **char** value) {
270. G.vexs[v] = value;
271. }

274. // 返回v的第一个邻接顶点
275. **int** FisrtAdjVex(MGraph G, **int** v) {
276. **for** (**int** j=0; j<G.vexNum; j++) {
277. **if** (!G.arcs[v][j]) **return** j;
278. }
279. **return** -1;
280. }

283. // 返回v相对领接顶点w外的下一个顶点
284. **int** NextAdjVex(MGraph G, **int** v, **int** w) {
285. // 从第v行第w列位置开始向右找下一个顶点
286. **for** (**int** j=w; j<G.vexNum; j++) {
287. **if** (!G.arcs[v][j]) **return** j;
288. }
289. **return** -1;
290. }

293. // 新增顶点
294. **void** InsertVex(MGraph &G, **char** value) {
296. G.vexNum++;
298. // 重新申请点向量空间，并加入新顶点值
299. G.vexs = (**char**\*)realloc(G.vexs, **sizeof**(**char**) \* G.vexNum);
300. G.vexs[G.vexNum-1] = value;
302. // 重新申请图空间，将新边的值设为0
303. G.arcs = (**int**\*\*)realloc(G.arcs, **sizeof**(**int**\*) \* G.vexNum);
304. //cout<<"ok\n";
305. **for** (**int** i=0; i<G.vexNum-1; i++) {
306. //cout<<i<<endl;
307. G.arcs[i] = (**int**\*)realloc(G.arcs[i], **sizeof**(**int**) \* G.vexNum);
308. //cout<<"ok   "<<G.vexNum-1<<"\n";
309. G.arcs[i][G.vexNum-1] = 0;
310. //cout<<"ok\n";
311. }
312. //cout<<"ok\n";
313. G.arcs[G.vexNum-1] = (**int**\*)malloc(**sizeof**(**int**) \* G.vexNum);
314. **for** (**int** j=0; j<G.vexNum; j++) {
315. //cout<<"  "<<j<<endl;
316. G.arcs[G.vexNum-1][j] = 0;
317. }
319. cout<<"\n新增顶点成功!\n";
320. }

323. // 删除顶点
324. **void** DeleteVex(MGraph &G, **char** value) {
326. **int** v = LocateVex(G, value);
327. **if** (v == -1) {
328. cout<<"不存在";
329. **return**;
330. }
331. // 在顶点数组中删去该值，并在边图中删去该行
332. **for** (**int** i=v+1; i<G.vexNum; i++) {
333. G.vexs[i-1] = G.vexs[i];
334. G.arcs[i-1] = G.arcs[i];
335. }
337. // 在边图中删去该列
338. **for** (**int** i=0; i<G.vexNum-1; i++) {
339. **for** (**int** j=v+1; j<G.vexNum; j++) {
340. G.arcs[i][j-1] = G.arcs[i][j];
341. }
342. }
344. G.vexNum--;
346. // 重新申请顶点向量空间，并加入新顶点值
347. G.vexs = (**char**\*)realloc(G.vexs, **sizeof**(**char**) \* G.vexNum);
349. // 重新申请图空间，将新边的值设为0
350. G.arcs = (**int**\*\*)realloc(G.arcs, **sizeof**(**int**\*) \* G.vexNum);
351. **for** (**int** i=0; i<G.vexNum; i++) {
352. G.arcs[i] = (**int**\*)realloc(G.arcs[i], **sizeof**(**int**) \* G.vexNum);
353. }
355. cout<<"\n删除顶点成功!\n";
356. }

359. // 新增边
360. **void** InsertArc(MGraph &G, **char** valueV, **char** valueW) {
361. **int** v = LocateVex(G, valueV), w = LocateVex(G, valueW);
362. **if** (G.kind == DG) {
363. G.arcs[v][w] = 1;
364. }
365. **else** **if** (G.kind == DN) {
366. cout<<"请输入新增边的权重：";
367. **int** weight = 0;
368. cin>>weight;
369. G.arcs[v][w] = weight;
370. }
371. **else** **if** (G.kind == UDG) {
372. G.arcs[v][w] = 1;
373. G.arcs[w][v] = 1;
374. }
375. **else** **if** (G.kind == UDN) {
376. cout<<"请输入新增边的权重：";
377. **int** weight = 0;
378. cin>>weight;
379. G.arcs[v][w] = weight;
380. G.arcs[w][v] = weight;
381. }
382. cout<<"\n新增边成功!\n";
383. }

386. // 删除边
387. **void** DeleteArc(MGraph &G, **char** valueV, **char** valueW) {
388. **int** v = LocateVex(G, valueV), w = LocateVex(G, valueW);
389. **if** (G.kind == DG || G.kind == DN) {
390. G.arcs[v][w] = 0;
391. }
392. **else** **if** (G.kind == UDG || G.kind == UDN) {
393. G.arcs[v][w] = 0;
394. G.arcs[w][v] = 0;
395. }
396. cout<<"\n删除边成功!\n";
397. }

400. // 深度优先遍历
401. **void** DFSTraverse(MGraph G) {
402. **bool** \*vexsCheck = (**bool**\*)malloc(**sizeof**(**bool**) \* G.vexNum);
403. **for** (**int** i=0; i<G.vexNum; i++) vexsCheck[i] = **false**;
405. cout<<"\n深度优先遍历：";
406. stack<**int**> s;
407. **for** (**int** i=0; i<G.vexNum; i++) {
408. **if** (!vexsCheck[i]) {
409. cout<<"  "<<GetVex(G, i);
410. s.push(i);
411. vexsCheck[i] = **true**;
412. **int** top;
413. **while** (!s.empty()) {
414. top = s.top();
415. // 如果该点未走过则输出
416. **if** (!vexsCheck[top]) cout<<"--"<<GetVex(G, top);
417. vexsCheck[top] = **true**;
418. **int** j;
419. // 遍历该结点所有邻接节点
420. **for** (j=0; j<G.vexNum; j++) {
421. **if** (j != top && G.arcs[top][j] && !vexsCheck[j]) {
422. s.push(j);
423. **break**;
424. }
425. }
426. // 如果该点所有邻接点都走过，则弹出该点
427. **if** (j == G.vexNum) s.pop();
428. }
429. }
430. }
431. free(vexsCheck);
432. }

435. // 广度优先遍历
436. **void** BFSTraverse(MGraph G) {
437. **bool** \*vexsCheck = (**bool**\*)malloc(**sizeof**(**bool**) \* G.vexNum);
438. **for** (**int** i=0; i<G.vexNum; i++) vexsCheck[i] = **false**;
440. cout<<"\n广度优先遍历：";
441. queue<**int**> q;
442. // for循环找所有未遍历的点，是为了有多个连通分支的情况
443. **for** (**int** i=0; i<G.vexNum; i++) {
444. **if** (!vexsCheck[i]) {
445. q.push(i);
446. vexsCheck[i] = **true**;
447. **int** front;
448. **while** (!q.empty()) {
449. front = q.front();
450. cout<<GetVex(G, front)<<"--";
451. q.pop();
452. **for** (**int** j=0; j<G.vexNum; j++) {
453. **if** (front != j && !vexsCheck[j]) {
454. q.push(j);
455. vexsCheck[j] = **true**;
456. }
457. }
458. }
459. }
460. }
461. }

464. // 普通遍历
465. **void** Traverse(MGraph G) {
466. cout<<"\n顶点向量的遍历如下：\n";
467. **for** (**int** i=0; i<G.vexNum; i++) {
468. cout<<G.vexs[i]<<" ";
469. }
470. cout<<"\n图的矩阵遍历如下：\n";
471. **for** (**int** i=0; i<G.vexNum; i++) {
472. **for** (**int** j=0; j<G.vexNum; j++) {
473. cout<<G.arcs[i][j]<<" ";
474. }
475. cout<<endl;
476. }
477. }

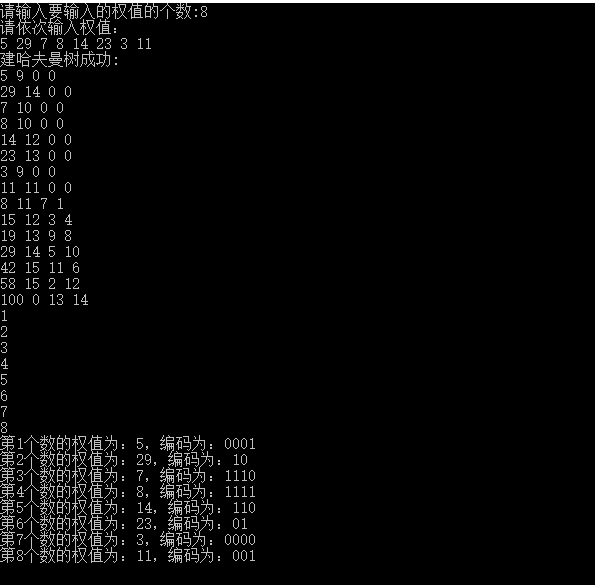
480. **int** main() {
481. MGraph G;
482. CreateGraph(G);
483. Traverse(G);
485. cout<<"请输入要新增的顶点的值：";
486. **char** ch;
487. cin>>ch;
488. InsertVex(G, ch);
489. Traverse(G);
490. cout<<"请输入要删除的顶点：";
491. cin>>ch;
492. DeleteVex(G, ch);
493. Traverse(G);
495. cout<<"请输入要插入的边的两个顶点值：";
496. **char** chV, chW;
497. cin>>chV>>chW;
498. InsertArc(G, chV, chW);
499. cout<<"请输入要删除的边的两个顶点值：";
500. cin>>chV>>chW;
501. DeleteArc(G, chV, chW);
502. Traverse(G);
504. DFSTraverse(G);
505. BFSTraverse(G);
506. DestroyGraph(G);
507. }

**2、**

**题目：**输入N个权值（1-100正整数），建立哈夫曼树。

**算法思想：**数据结构为线性表，每次将最小的两个提出来，权相加作为新的结点，并确立父子关系。时间复杂度为O(2\*n-1)。

**运行结果：**



**结果分析：**成功。

**源程序：**

1. /\*
2. 输入N个权值（1-100正整数），建立哈夫曼树。
3. 8
4. 5 29 7 8 14 23 3 11
5. \*/
6. #include<iostream>
7. #include<stdlib.h>
8. #include<vector>
9. #include<stack>
10. **using** **namespace** std;
12. **typedef** **struct** {
13. **int** weight;
14. **int** parent;
15. **int** lchild, rchild;
16. }HuffmanTree;
18. // 建哈夫曼树表
19. **void** CreateHuffmanTree(HuffmanTree \* &hTree, **int** n, **int** m) {
20. // 经过n-1次找两个最小值
21. **for** (**int** i=1; i<=n-1; i++) {
22. **int** fMin = INT\_MAX, sMin = INT\_MAX-1;   // 初始化第一小和第二小的权值
23. **int** fMinP = 0, sMinP = 0;               // 权值第一小和第二小的位置
24. **int** j = 1;
25. // 遍历所有已经有权值的结点
26. **while** (j<m && hTree[j].weight) {
27. // 如果该结点没有双亲，且权值小于第二小的权值
28. **if** ((!hTree[j].parent) && (hTree[j].weight < sMin)) {
29. // 如果该权值小于第一个小的权值
30. **if** (hTree[j].weight < fMin) {
31. sMin = fMin;
32. sMinP = fMinP;
33. fMin = hTree[j].weight;
34. fMinP = j;
35. }
36. // 如果在第一和第二小权值中间
37. **else** {
38. sMin = hTree[j].weight;
39. sMinP = j;
40. }
41. }
42. j++;
43. }
44. // j最后停在没有权值的新结点的那一行，将上述两个结点变为新结点的孩子
45. **if** (j < m) {
47. hTree[fMinP].parent = j;
48. hTree[sMinP].parent = j;
50. hTree[j].lchild = fMinP;
51. hTree[j].rchild = sMinP;
52. hTree[j].weight = fMin + sMin;
53. }
54. }
56. cout<<"建哈夫曼树成功:\n";
57. **for** (**int** i=1; i<m; i++) {
58. cout<<hTree[i].weight<<" "<<hTree[i].parent<<" "<<hTree[i].lchild<<" "<<hTree[i].rchild<<endl;
59. }
60. }

63. // 显示哈夫曼编码
64. **void** ShowHuffmanCoding(HuffmanTree \*hTree, **int** n, **int** m) {
65. vector< stack<**char**> > hCode;
66. stack<**char**> temp;
67. // 从结点开始找双亲
68. **for** (**int** i=1; i<=n; i++) {
69. cout<<i<<endl;
70. **int** j = i;
71. **int** parent = hTree[i].parent;   // 找到双亲
72. // 当双亲不为空，即没有到头
73. **while** (parent) {
74. // 如果该结点是双亲的左孩子
75. **if** (hTree[parent].lchild == j) temp.push('0');
76. // 如果是右孩子
77. **else** temp.push('1');
78. j = parent;
79. parent = hTree[j].parent;
80. }
81. hCode.push\_back(temp);
82. **while** (!temp.empty()) temp.pop();
83. }
85. **for** (**int** i=0; i<n; i++) {
86. cout<<"第"<<i+1<<"个数的权值为："<<hTree[i+1].weight<<"，编码为：";
87. **while** (!hCode[i].empty()) {
88. cout<<hCode[i].top();
89. hCode[i].pop();
90. }
91. cout<<endl;
92. }
93. }

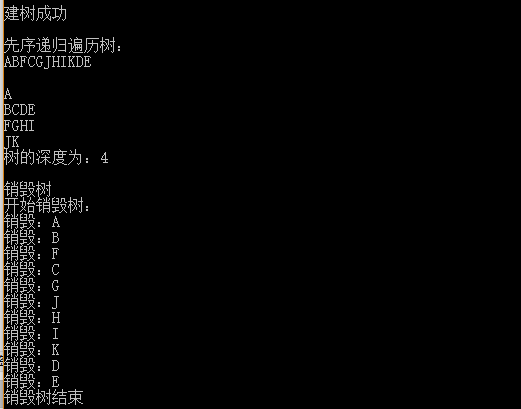
96. **int** main() {
97. printf("请输入要输入的权值的个数:");
98. **int** n = 0;
99. scanf("%d", &n);
101. // 初始化哈夫曼树线性表
102. **int** m = 2 \* n;
103. HuffmanTree \*hTree = (HuffmanTree\*)malloc(**sizeof**(HuffmanTree) \* m);
104. **for** (**int** i=1; i<m; i++) {
105. hTree[i].weight = hTree[i].parent = hTree[i].lchild = hTree[i].rchild = 0;
106. }
108. printf("请依次输入权值：\n");
109. **for** (**int** i=1; i<=n; i++) {
110. scanf("%d", &hTree[i].weight);
111. }
113. CreateHuffmanTree(hTree, n, m);
114. ShowHuffmanCoding(hTree, n, m);
116. free(hTree);
117. }

**3、**

**题目：**编程实现，对一棵以孩子-兄弟链表表示的树，求树的高度。

**算法思想：**1）创建、遍历、销毁和二叉树一样，只是把孩子看成左孩子，兄弟看成右孩子。2）求树的高度。数据结构为队列，并定义两个队列。每次从一个不为空的队列中获取队头结点，并让该结点pop出队列。接着，将该结点的孩子push进另一个队列中，将兄弟push进当前队列中，若为孩子或兄弟为空则不push。当当前队列为空时，切换到下一个队列。一直循环重复上述过程，直到两个队列都为空。每次当当前队列为空后，树的高度就加一(初始为0)。最终得到树的高度。算法时间复杂度为O(n)。

**运行结果：**



**结果分析：**成功。

**源程序：**

1. /\*
2. 编程实现，对一棵以孩子-兄弟链表表示的树，求树的高度。
3. \*/
4. #include<iostream>
5. #include<stdlib.h>
6. #include<stack>
7. #include<queue>
9. **using** **namespace** std;
11. **typedef** **struct** CSTNode {
12. **char** data;
13. CSTNode \*firstChild;
14. CSTNode \*nextSibling;
15. }CSTNode, \*CSTree;

18. // 先序+中序 建树法
19. **void** CreateCSTree(CSTree &csTree, string preOrder, string inOrder) {
20. csTree = (CSTNode\*)malloc(**sizeof**(CSTNode));
21. csTree->data = preOrder[0];
23. **int** pos = inOrder.find(preOrder[0]);
24. string inOrderL = inOrder.substr(0, pos);
25. string inOrderR = inOrder.substr(pos + 1, inOrder.length() - 1);
26. string preOrderL = preOrder.substr(1, inOrderL.length());
27. string preOrderR = preOrder.substr(inOrderL.length() + 1, preOrder.length() - 1);
29. **int** isEmptyL = 0, isEmptyR = 0;     // 用来判断左右结点是否为空的标志位
30. **if** (inOrderL.empty()) {
31. csTree->firstChild = NULL;
32. isEmptyL = 1;
33. }
34. **if** (inOrderR.empty()) {
35. csTree->nextSibling = NULL;
36. isEmptyR = 1;
37. }
38. **if** (!isEmptyL) {
39. cout<<preOrderL<<endl<<inOrderL<<endl;
40. CreateCSTree(csTree->firstChild, preOrderL, inOrderL);
41. }
42. **if** (!isEmptyR) {
43. cout<<preOrderR<<endl<<inOrderR<<endl;
44. CreateCSTree(csTree->nextSibling, preOrderR, inOrderR);
45. }
46. }

49. // 先序遍历，递归
50. **void** PreOrderTraverse(CSTree csTree) {
51. **if** (csTree) {
52. cout<<csTree->data;
53. PreOrderTraverse(csTree->firstChild);
54. PreOrderTraverse(csTree->nextSibling);
55. }
56. }

59. // 销毁树
60. **void** DestroyCSTree(CSTree &csTree) {
61. **if** (!csTree) {
62. cout<<"该树为空！";
63. **return**;
64. }
66. cout<<"开始销毁树：\n";
67. // 销毁树，只能将树结点指针传进去，因为free要传入的是指针
68. stack<CSTree> s;
69. s.push(csTree);
70. csTree = NULL;
71. CSTree temp;
73. **while** (!s.empty()) {
74. temp = s.top();
75. s.pop();
76. cout<<"销毁："<<temp->data<<endl;
77. **if** (temp->nextSibling) s.push(temp->nextSibling);
78. **if** (temp->firstChild) s.push(temp->firstChild);
79. free(temp);
80. }
81. cout<<"销毁树结束";
82. }

85. // 求树的深度
86. **int** GetTreeDepth(CSTree csTree) {
87. **if** (!csTree) {
88. cout<<"该树为空！";
89. **return** 0;
90. }
92. queue<CSTree> qA, qB;
93. qA.push(csTree);
95. **int** depth = 0;
96. **while** (!qA.empty() || !qB.empty()) {
97. CSTree temp;
98. // 如果队列A不为空
99. **if** (!qA.empty()) {
100. **while** (!qA.empty()) {
101. temp = qA.front();
102. qA.pop();
103. cout<<temp->data;
104. // 如果孩子不为空，则把孩子丢进另一个队列
105. **if** (temp->firstChild) qB.push(temp->firstChild);
106. // 如果兄弟不为空，则把兄弟继续丢进该队列中
107. **if** (temp->nextSibling) qA.push(temp->nextSibling);
108. }
109. }
110. // 反之，如果队列B不为空
111. **else** {
112. **while** (!qB.empty()) {
113. temp = qB.front();
114. qB.pop();
115. cout<<temp->data;
116. // 同上
117. **if** (temp->firstChild) qA.push(temp->firstChild);
118. **if** (temp->nextSibling) qB.push(temp->nextSibling);
119. }
120. }
121. depth++;
122. cout<<endl;
123. }
124. **return** depth;
125. }

128. **int** main() {
129. string preOrder = "ABFCGJHIKDE";
130. string inOrder = "FBJGHKICDEA";
132. CSTree csTree;
133. CreateCSTree(csTree, preOrder, inOrder);
134. cout<<"\n建树成功\n\n";
136. cout<<"先序递归遍历树：\n";
137. PreOrderTraverse(csTree);
138. cout<<endl<<endl;
140. **int** depth = GetTreeDepth(csTree);
141. cout<<"树的深度为："<<depth<<endl<<endl;
143. cout<<"销毁树\n";
144. DestroyCSTree(csTree);
145. }

**4、**

**题目：** CSP题目 碰撞的小球

问题描述

　　数轴上有一条长度为L（L为偶数)的线段，左端点在原点，右端点在坐标L处。有n个不计体积的小球在线段上，开始时所有的小球都处在偶数坐标上，速度方向向右，速度大小为1单位长度每秒。

　　当小球到达线段的端点（左端点或右端点）的时候，会立即向相反的方向移动，速度大小仍然为原来大小。

　　当两个小球撞到一起的时候，两个小球会分别向与自己原来移动的方向相反的方向，以原来的速度大小继续移动。

　　现在，告诉你线段的长度L，小球数量n，以及n个小球的初始位置，请你计算t秒之后，各个小球的位置。

提示

　　因为所有小球的初始位置都为偶数，而且线段的长度为偶数，可以证明，不会有三个小球同时相撞，小球到达线段端点以及小球之间的碰撞时刻均为整数。

　　同时也可以证明两个小球发生碰撞的位置一定是整数（但不一定是偶数）。

输入格式

　　输入的第一行包含三个整数n, L, t，用空格分隔，分别表示小球的个数、线段长度和你需要计算t秒之后小球的位置。

　　第二行包含n个整数a1, a2, …, an，用空格分隔，表示初始时刻n个小球的位置。

输出格式

　　输出一行包含n个整数，用空格分隔，第i个整数代表初始时刻位于ai的小球，在t秒之后的位置。

样例输入

3 10 5

4 6 8

样例输出

7 9 9

**算法思想：**1）定义一个小球结构体，存储小球位置和方向两个值。定义一个一维数组，存储小球初始序号值(如0,1,2...)“指针”。2）对一维“指针”数组中的“指针”，根据“指针”指向的小球的位置值，从小到大进行选择排序。得到“指针数组”中的“指针”依次指向的小球的位置值是有序的。3）对时间从1到t进行遍历，每次将小球的位置值 + 小球的方向值(1或-1)。然后，通过对于位置值是有序的“指针”数组，进行判断是否跟前一个小球相碰，是否跟两边的壁相碰。4）最后遍历小球结构体数组，依次输出。

**运行结果：**





**结果分析：**成功。

**源程序：**

1. #include<iostream>
2. #include<stdlib.h>
4. **using** **namespace** std;
6. **typedef** **struct** {
7. **int** pos;    // 小球的位置
8. **int** dir;    // 小球的方向
9. }Pellet;
11. **int** main() {
12. **int** n, l, t;
13. scanf("%d%d%d", &n, &l, &t);
15. Pellet \*pellet = (Pellet\*)malloc(**sizeof**(Pellet) \* n);
16. **int** \*p = (**int**\*)malloc(**sizeof**(**int**) \* n);
17. **for** (**int** i=0; i<n; i++) {
18. scanf("%d", &pellet[i].pos);
19. p[i] = i;           // “指针”指向小球本来位置
20. pellet[i].dir = 1;  // 小球初始向右
21. }
23. // 将p这个“指针”数组，根据“指针”指向的值排序
24. **for** (**int** i=0; i<n-1; i++) {
25. **int** min = pellet[p[i]].pos;
26. **int** temp;
27. **for** (**int** j=i+1; j<n; j++) {
28. **if** (pellet[p[j]].pos < min) {
29. min = pellet[p[j]].pos;
30. // 将“指针”交换
31. temp = p[j];
32. p[j] = p[i];
33. p[i] = temp;
34. }
35. }
36. }
38. // 用排序好的“指针”数组进行上述过程
39. // 每过一秒
40. **for** (**int** i=0; i<t; i++) {
41. // 遍历“指针”数组
42. **for** (**int** j=0; j<n; j++) {
43. pellet[p[j]].pos += pellet[p[j]].dir;   // 根据每个球的方向（1或-1）进行移位
44. // 如果小球和左边一个小球位置相同，则将两个小球都改变方向
45. **if** ((j != 0) && (pellet[p[j-1]].pos == pellet[p[j]].pos)) {
46. pellet[p[j]].dir = 0 - pellet[p[j]].dir;
47. pellet[p[j-1]].dir = 0 - pellet[p[j-1]].dir;
48. }
49. // 如果小球碰壁了，反向
50. **else** **if** ((pellet[p[j]].pos == l) || (pellet[p[j]].pos == 0)) {
51. pellet[p[j]].dir = 0 - pellet[p[j]].dir;
52. }
53. }
54. }
56. **for** (**int** i=0; i<n; i++) {
57. printf("%d ", pellet[i].pos);
58. }
60. free(pellet);
61. free(p);
62. }

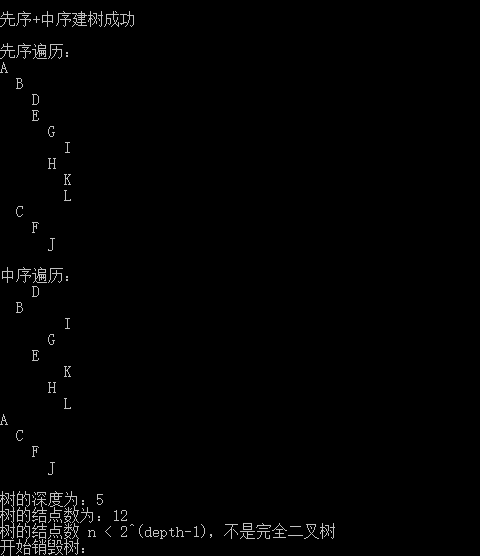
**5、**

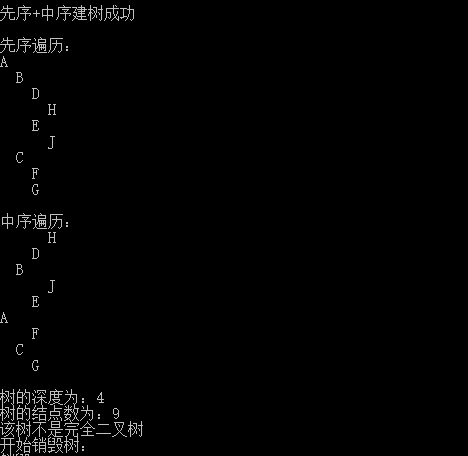
**题目：**



**算法思想：**1）进行数学判断，如果树的结点数 n < 2^(depth-1)，则不是完全二叉树。2）如果数学条件满足，则层次遍历二叉树。每遍历完一行，若结点数 n < 2^d - 1 (n为遍历过的结点数，d为遍历过的深度)则不是完全二叉树。3）若进行到最后一行，那就判断如果该结点的左孩子为空、右孩子不为空，或者左孩子不为空、但左结点的右孩子为空，则不是完全二叉树。

**运行结果：**







**结果分析：**成功。

**源程序：**

1. /\*
2. 编写算法判定二叉树是否为完全二叉树
3. 图：
4. A
5. B  C
6. D  E   F
7. G  H    J
8. I  K L
9. \*/
10. #include<iostream>
11. #include<stdlib.h>
12. #include<math.h>
13. #include<string>
14. #include<queue>
15. #include<stack>
17. **using** **namespace** std;

20. **typedef** **struct** BTNode {
21. **char** data;
22. BTNode \*lchild, \*rchild;
23. }BTNode, \*BiTree;

26. // 先序+中序 建树法
27. **void** CreateBiTree(BiTree &bTree, string preOrder, string inOrder) {
28. bTree = (BTNode\*)malloc(**sizeof**(BTNode));
29. bTree->data = preOrder[0];
31. **int** pos = inOrder.find(preOrder[0]);
32. string inOrderL = inOrder.substr(0, pos);
33. string inOrderR = inOrder.substr(pos + 1, inOrder.length() - 1);
34. string preOrderL = preOrder.substr(1, inOrderL.length());
35. string preOrderR = preOrder.substr(inOrderL.length() + 1, preOrder.length() - 1);
37. **int** isEmptyL = 0, isEmptyR = 0;     // 用来判断左右结点是否为空的标志位
38. **if** (inOrderL.empty()) {
39. bTree->lchild = NULL;
40. isEmptyL = 1;
41. }
42. **if** (inOrderR.empty()) {
43. bTree->rchild = NULL;
44. isEmptyR = 1;
45. }
46. **if** (!isEmptyL) {
47. // cout<<preOrderL<<endl<<inOrderL<<endl;
48. CreateBiTree(bTree->lchild, preOrderL, inOrderL);
49. }
50. **if** (!isEmptyR) {
51. // cout<<preOrderR<<endl<<inOrderR<<endl;
52. CreateBiTree(bTree->rchild, preOrderR, inOrderR);
53. }
54. }

57. // 先序遍历，递归
58. **void** PreOrderTraverse(BiTree bTree, **int** tab) {
59. **if** (bTree) {
60. **for** (**int** i=0; i<tab; i++) cout<<"  ";
61. cout<<bTree->data<<endl;
62. PreOrderTraverse(bTree->lchild, tab+1);
63. PreOrderTraverse(bTree->rchild, tab+1);
64. }
65. }

68. // 中序遍历，递归
69. **void** InOrderTraverse(BiTree bTree, **int** tab) {
70. **if** (bTree) {
71. InOrderTraverse(bTree->lchild, tab+1);
72. **for** (**int** i=0; i<tab; i++) cout<<"  ";
73. cout<<bTree->data<<endl;
74. InOrderTraverse(bTree->rchild, tab+1);
75. }
76. }

79. // 返回树的深度，递归
80. **int** BiTreeDepth(BiTree bTree) {
81. **if** (bTree) {
82. **int** lchildDepth = 0;
83. **if** (bTree->lchild) lchildDepth = BiTreeDepth(bTree->lchild);
84. **int** rchildDepth = 0;
85. **if** (bTree->rchild) rchildDepth = BiTreeDepth(bTree->rchild);
86. **return** lchildDepth > rchildDepth ? lchildDepth+1 : rchildDepth+1;
87. }
88. **return** 0;
89. }

92. // 销毁树
93. **void** DestoryBiTree(BiTree &bTree) {
94. **if** (!bTree) {
95. cout<<"该树为空！";
96. **return**;
97. }
99. cout<<"开始销毁树：\n";
100. // 销毁树，只能将树结点指针传进去，因为free要传入的是指针
101. stack<BiTree> s;
102. s.push(bTree);
103. bTree = NULL;
104. BiTree temp;
106. **while** (!s.empty()) {
107. temp = s.top();
108. s.pop();
109. cout<<"销毁："<<temp->data<<endl;
110. **if** (temp->rchild) s.push(temp->rchild);
111. **if** (temp->lchild) s.push(temp->lchild);
112. free(temp);
113. }
114. cout<<"销毁树结束\n";
115. }

118. // 判断是否为完全二叉树
119. **void** IsCompleteBiTree(BiTree bTree, string preOrder) {
120. **int** depth = BiTreeDepth(bTree);
121. **int** count = preOrder.length();
122. cout<<"树的深度为："<<depth<<endl;
123. cout<<"树的结点数为："<<count<<endl;
125. // 数学判断
126. **if** (count < pow((**double**)2, (**double**)depth-1)) {
127. cout<<"树的结点数 n < 2^(depth-1)，不是完全二叉树\n";
128. **return**;
129. }
130. **else** {
131. queue<BiTree> q;
132. q.push(bTree);
134. // 定义界符
135. BiTree limit = (BTNode\*)malloc(**sizeof**(BTNode) \* 1);
136. limit->data = ' ';
137. limit->lchild = limit->rchild = NULL;
138. q.push(limit);
140. **int** d = 0, node = 0;  // 树的深度和结点数
141. BiTree top, next;
142. **while** (1) {
143. top = q.front();
144. q.pop();
145. node++;
147. // 如果q.pop()弹出一个元素后队列为空，说明弹出的是最后一行的最后一个界符
148. **if** (q.empty()) **break**;
149. **else** next = q.front();
151. // 如果得到的是界符
152. **if** (top->data == ' ') {
153. q.push(top);// 将界符放回队列中
154. d++;        // 深度加一
155. node--;     // 结点减一，因为界符不是结点
156. // 如果在d行结束时（d不可能到最后一行），不满足结点数  n = 2^d - 1 ，则不是完全二叉树
157. **if** (pow((**double**)2, (**double**)d) - 1 != node) {
158. cout<<"由第"<<d<<"行可以看出，该树不是二叉树"<<endl;
159. **return**;
160. }
161. }
162. // 用来判断最后一行：如果 左孩子为空但右孩子不空 或 右孩子空但下一个结点左孩子不空
163. **else** **if** ((top->lchild == NULL && top->rchild != NULL)
164. || (top->rchild == NULL && next->lchild != NULL)) {
165. cout<<"该树不是完全二叉树"<<endl;
166. **return**;
167. }
168. **else** {
169. **if** (top->lchild) q.push(top->lchild);
170. **if** (top->rchild) q.push(top->rchild);
171. }
172. }
173. }
174. cout<<"该树是完全二叉树\n";
175. }

178. **int** main() {
180. // 第一颗树
181. string preOrder = "ABDEGIHKLCFJ";
182. string inOrder = "DBIGEKHLACFJ";
184. BiTree bTree;
185. CreateBiTree(bTree, preOrder, inOrder);
186. cout<<"\n先序+中序建树成功\n\n";
188. // 递归实现
189. cout<<"先序遍历：\n";
190. PreOrderTraverse(bTree, 0);
191. cout<<endl;
193. cout<<"中序遍历：\n";
194. InOrderTraverse(bTree, 0);
195. cout<<endl;
197. // 检测是否为二叉树
198. IsCompleteBiTree(bTree, preOrder);
200. DestoryBiTree(bTree);

203. // 第二颗树
204. preOrder = "ABDHEJCFG";
205. inOrder = "HDBJEAFCG";
207. bTree;
208. CreateBiTree(bTree, preOrder, inOrder);
209. cout<<"\n先序+中序建树成功\n\n";
211. // 递归实现
212. cout<<"先序遍历：\n";
213. PreOrderTraverse(bTree, 0);
214. cout<<endl;
216. cout<<"中序遍历：\n";
217. InOrderTraverse(bTree, 0);
218. cout<<endl;
220. // 检测是否为二叉树
221. IsCompleteBiTree(bTree, preOrder);
223. DestoryBiTree(bTree);
225. // 第三颗树
226. preOrder = "ABDHIEJCF";
227. inOrder = "HDIBJEAFC";
229. bTree;
230. CreateBiTree(bTree, preOrder, inOrder);
231. cout<<"\n先序+中序建树成功\n\n";
233. // 递归实现
234. cout<<"先序遍历：\n";
235. PreOrderTraverse(bTree, 0);
236. cout<<endl;
238. cout<<"中序遍历：\n";
239. InOrderTraverse(bTree, 0);
240. cout<<endl;
242. // 检测是否为二叉树
243. IsCompleteBiTree(bTree, preOrder);
245. DestoryBiTree(bTree);
246. }

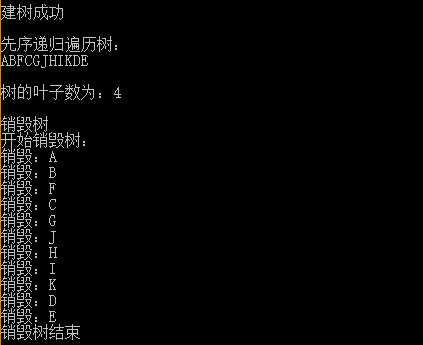
**6、**

**题目：**



**算法思想：**直接遍历树，双孩子为空，则叶子数加一。

**运行结果：**



**结果分析：**成功。

**源程序：**

1. /\*
2. 计算孩子兄弟树的叶子数
3. \*/
4. /\*
5. 编程实现，对一棵以孩子-兄弟链表表示的树，求树的高度。
6. \*/
7. #include<iostream>
8. #include<stdlib.h>
9. #include<stack>
10. #include<queue>
12. **using** **namespace** std;
14. **typedef** **struct** CSTNode {
15. **char** data;
16. CSTNode \*firstChild;
17. CSTNode \*nextSibling;
18. }CSTNode, \*CSTree;

21. // 先序+中序 建树法
22. **void** CreateCSTree(CSTree &csTree, string preOrder, string inOrder) {
23. csTree = (CSTNode\*)malloc(**sizeof**(CSTNode));
24. csTree->data = preOrder[0];
26. **int** pos = inOrder.find(preOrder[0]);
27. string inOrderL = inOrder.substr(0, pos);
28. string inOrderR = inOrder.substr(pos + 1, inOrder.length() - 1);
29. string preOrderL = preOrder.substr(1, inOrderL.length());
30. string preOrderR = preOrder.substr(inOrderL.length() + 1, preOrder.length() - 1);
32. **int** isEmptyL = 0, isEmptyR = 0;     // 用来判断左右结点是否为空的标志位
33. **if** (inOrderL.empty()) {
34. csTree->firstChild = NULL;
35. isEmptyL = 1;
36. }
37. **if** (inOrderR.empty()) {
38. csTree->nextSibling = NULL;
39. isEmptyR = 1;
40. }
41. **if** (!isEmptyL) {
42. cout<<preOrderL<<endl<<inOrderL<<endl;
43. CreateCSTree(csTree->firstChild, preOrderL, inOrderL);
44. }
45. **if** (!isEmptyR) {
46. cout<<preOrderR<<endl<<inOrderR<<endl;
47. CreateCSTree(csTree->nextSibling, preOrderR, inOrderR);
48. }
49. }

52. // 先序遍历，递归
53. **void** PreOrderTraverse(CSTree csTree) {
54. **if** (csTree) {
55. cout<<csTree->data;
56. PreOrderTraverse(csTree->firstChild);
57. PreOrderTraverse(csTree->nextSibling);
58. }
59. }

62. // 销毁树
63. **void** DestroyCSTree(CSTree &csTree) {
64. **if** (!csTree) {
65. cout<<"该树为空！";
66. **return**;
67. }
69. cout<<"开始销毁树：\n";
70. // 销毁树，只能将树结点指针传进去，因为free要传入的是指针
71. stack<CSTree> s;
72. s.push(csTree);
73. csTree = NULL;
74. CSTree temp;
76. **while** (!s.empty()) {
77. temp = s.top();
78. s.pop();
79. cout<<"销毁："<<temp->data<<endl;
80. **if** (temp->nextSibling) s.push(temp->nextSibling);
81. **if** (temp->firstChild) s.push(temp->firstChild);
82. free(temp);
83. }
84. cout<<"销毁树结束";
85. }

88. // 求树的叶子数
89. **int** GetTreeLeaves(CSTree csTree) {
90. **if** (!csTree) {
91. cout<<"该树为空！";
92. **return** 0;
93. }
95. //cout<<"ok\n";
96. stack<CSTree> s;
97. s.push(csTree);
98. CSTree top;
99. **int** leaves = 0;
100. **while** (!s.empty()) {
101. //cout<<leaves<<endl;
102. top = s.top();
103. s.pop();
104. **if** (!top->firstChild && !top->nextSibling) leaves++;
105. **else** {
106. **if** (top->firstChild) s.push(top->firstChild);
107. **if** (top->nextSibling) s.push(top->nextSibling);
108. }
109. }
111. **return** leaves;
112. }

115. **int** main() {
116. string preOrder = "ABFCGJHIKDE";
117. string inOrder = "FBJGHKICDEA";
119. CSTree csTree;
120. CreateCSTree(csTree, preOrder, inOrder);
121. cout<<"\n建树成功\n\n";
123. cout<<"先序递归遍历树：\n";
124. PreOrderTraverse(csTree);
125. cout<<endl<<endl;
127. **int** leaves = GetTreeLeaves(csTree);
128. cout<<"树的叶子数为："<<leaves<<endl<<endl;
130. cout<<"销毁树\n";
131. DestroyCSTree(csTree);
132. }

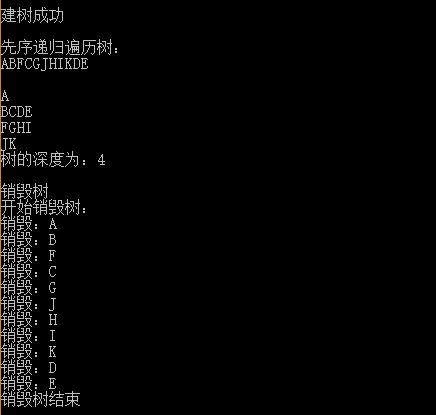
**7、**

**题目：**



**算法思想：**同第三题。

**运行结果：**



**结果分析：**成功。

**源程序：**

1. /\*
2. 编程实现，对一棵以孩子-兄弟链表表示的树，求树的深度。
3. \*/
4. #include<iostream>
5. #include<stdlib.h>
6. #include<stack>
7. #include<queue>
9. **using** **namespace** std;
11. **typedef** **struct** CSTNode {
12. **char** data;
13. CSTNode \*firstChild;
14. CSTNode \*nextSibling;
15. }CSTNode, \*CSTree;

18. // 先序+中序 建树法
19. **void** CreateCSTree(CSTree &csTree, string preOrder, string inOrder) {
20. csTree = (CSTNode\*)malloc(**sizeof**(CSTNode));
21. csTree->data = preOrder[0];
23. **int** pos = inOrder.find(preOrder[0]);
24. string inOrderL = inOrder.substr(0, pos);
25. string inOrderR = inOrder.substr(pos + 1, inOrder.length() - 1);
26. string preOrderL = preOrder.substr(1, inOrderL.length());
27. string preOrderR = preOrder.substr(inOrderL.length() + 1, preOrder.length() - 1);
29. **int** isEmptyL = 0, isEmptyR = 0;     // 用来判断左右结点是否为空的标志位
30. **if** (inOrderL.empty()) {
31. csTree->firstChild = NULL;
32. isEmptyL = 1;
33. }
34. **if** (inOrderR.empty()) {
35. csTree->nextSibling = NULL;
36. isEmptyR = 1;
37. }
38. **if** (!isEmptyL) {
39. cout<<preOrderL<<endl<<inOrderL<<endl;
40. CreateCSTree(csTree->firstChild, preOrderL, inOrderL);
41. }
42. **if** (!isEmptyR) {
43. cout<<preOrderR<<endl<<inOrderR<<endl;
44. CreateCSTree(csTree->nextSibling, preOrderR, inOrderR);
45. }
46. }

49. // 先序遍历，递归
50. **void** PreOrderTraverse(CSTree csTree) {
51. **if** (csTree) {
52. cout<<csTree->data;
53. PreOrderTraverse(csTree->firstChild);
54. PreOrderTraverse(csTree->nextSibling);
55. }
56. }

59. // 销毁树
60. **void** DestroyCSTree(CSTree &csTree) {
61. **if** (!csTree) {
62. cout<<"该树为空！";
63. **return**;
64. }
66. cout<<"开始销毁树：\n";
67. // 销毁树，只能将树结点指针传进去，因为free要传入的是指针
68. stack<CSTree> s;
69. s.push(csTree);
70. csTree = NULL;
71. CSTree temp;
73. **while** (!s.empty()) {
74. temp = s.top();
75. s.pop();
76. cout<<"销毁："<<temp->data<<endl;
77. **if** (temp->nextSibling) s.push(temp->nextSibling);
78. **if** (temp->firstChild) s.push(temp->firstChild);
79. free(temp);
80. }
81. cout<<"销毁树结束";
82. }

85. // 求树的深度
86. **int** GetTreeDepth(CSTree csTree) {
87. **if** (!csTree) {
88. cout<<"该树为空！";
89. **return** 0;
90. }
92. queue<CSTree> qA, qB;
93. qA.push(csTree);
95. **int** depth = 0;
96. **while** (!qA.empty() || !qB.empty()) {
97. CSTree temp;
98. // 如果队列A不为空
99. **if** (!qA.empty()) {
100. **while** (!qA.empty()) {
101. temp = qA.front();
102. qA.pop();
103. cout<<temp->data;
104. // 如果孩子不为空，则把孩子丢进另一个队列
105. **if** (temp->firstChild) qB.push(temp->firstChild);
106. // 如果兄弟不为空，则把兄弟继续丢进该队列中
107. **if** (temp->nextSibling) qA.push(temp->nextSibling);
108. }
109. }
110. // 反之，如果队列B不为空
111. **else** {
112. **while** (!qB.empty()) {
113. temp = qB.front();
114. qB.pop();
115. cout<<temp->data;
116. // 同上
117. **if** (temp->firstChild) qA.push(temp->firstChild);
118. **if** (temp->nextSibling) qB.push(temp->nextSibling);
119. }
120. }
121. depth++;
122. cout<<endl;
123. }
124. **return** depth;
125. }

128. **int** main() {
129. string preOrder = "ABFCGJHIKDE";
130. string inOrder = "FBJGHKICDEA";
132. CSTree csTree;
133. CreateCSTree(csTree, preOrder, inOrder);
134. cout<<"\n建树成功\n\n";
136. cout<<"先序递归遍历树：\n";
137. PreOrderTraverse(csTree);
138. cout<<endl<<endl;
140. **int** depth = GetTreeDepth(csTree);
141. cout<<"树的深度为："<<depth<<endl<<endl;
143. cout<<"销毁树\n";
144. DestroyCSTree(csTree);
145. }