**数据结构第五次上机实验报告**

学号：161730126 姓名：李双玖

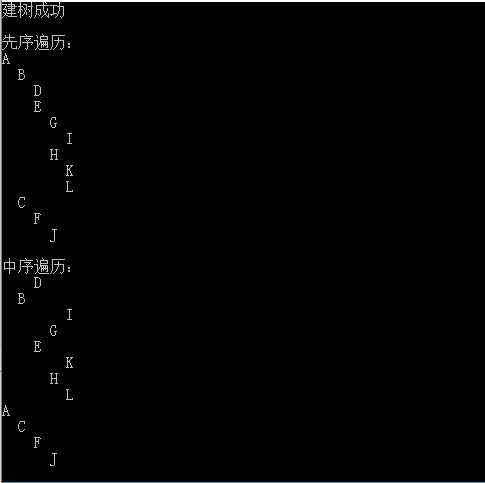
1. 调试成功程序及说明

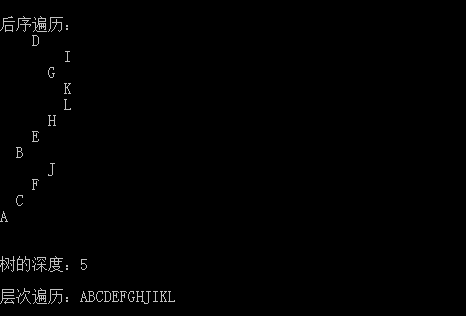
**1、**

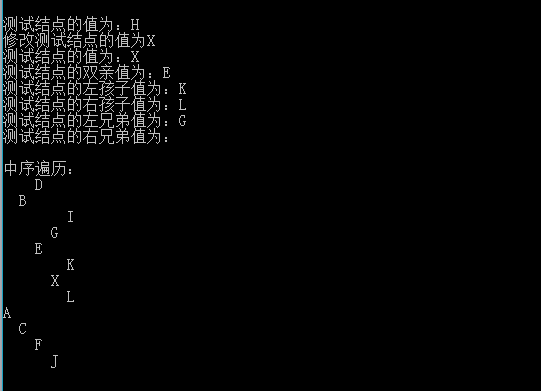
**题目：**编程实现书P121 ADT BinaryTree 基本操作20个，用二叉链表结构实现。

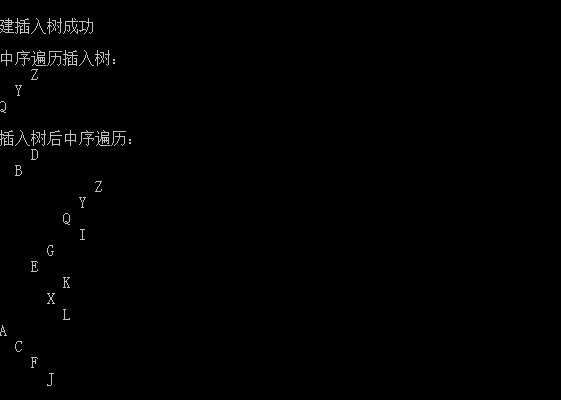
**算法思想**：20个函数中，建树函数采用先序+中序建树法；层次遍历采用队列；插入和删除采用栈（用递归也是一样的）；其它函数都采用递归法。

运行结果：











**结果分析：**成功。

**源程序：**

1. /\*
2. 编程实现书P121  ADT BinaryTree 基本操作20个，用二叉链表结构实现；
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<string>
13. #include<queue>
14. #include<stack>
15. **using** **namespace** std;
17. **typedef** **struct** BTNode {
18. **char** data;
19. BTNode \*lchild, \*rchild;
20. }BTNode, \*BiTree;

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

52. // 销毁树
53. **void** DestoryBiTree(BiTree &bTree) {
54. **if** (!bTree) {
55. cout<<"该树为空！";
56. **return**;
57. }
59. cout<<"开始销毁树：\n";
60. // 销毁树，只能将树结点指针传进去，因为free要传入的是指针
61. stack<BiTree> s;
62. s.push(bTree);
63. bTree = NULL;
64. BiTree temp;
66. **while** (!s.empty()) {
67. temp = s.top();
68. s.pop();
69. cout<<"销毁："<<temp->data<<endl;
70. **if** (temp->rchild) s.push(temp->rchild);
71. **if** (temp->lchild) s.push(temp->lchild);
72. free(temp);
73. }
74. cout<<"销毁树结束\n\n";
75. }

78. // 判断树是否为空
79. **bool** BiTreeEmpty(BiTree bTree) {
80. **if** (bTree) **return** **false**;
81. **else** **return** **true**;
82. }

85. // 返回树的深度，递归
86. **int** BiTreeDepth(BiTree bTree) {
87. **if** (bTree) {
88. **int** lchildDepth = 0;
89. **if** (bTree->lchild) lchildDepth = BiTreeDepth(bTree->lchild);
90. **int** rchildDepth = 0;
91. **if** (bTree->rchild) rchildDepth = BiTreeDepth(bTree->rchild);
92. **return** lchildDepth > rchildDepth ? lchildDepth+1 : rchildDepth+1;
93. }
94. **return** 0;
95. }

98. // 返回树的根
99. **char** Root(BiTree bTree) {
100. **return** bTree->data;
101. }

104. // 返回结点 bTNode 的值
105. **char** Value(BiTree bTree, **const** BTNode\* bTNode) {
106. stack<BiTree> s;
107. s.push(bTree);
108. BiTree temp;
109. **while** (!s.empty()) {
110. temp = s.top();
111. s.pop();
112. **if** (temp == bTNode) {
113. **return** temp->data;
114. }
115. **if** (temp->lchild) s.push(temp->lchild);
116. **if** (temp->rchild) s.push(temp->rchild);
117. }
118. **return** '\0';
119. }

122. // 给结点 bTNode 赋值为value
123. **bool** Assign(BiTree bTree, BTNode \*bTNode, **char** value) {
124. stack<BiTree> s;
125. s.push(bTree);
126. BiTree temp;
127. **while** (!s.empty()) {
128. temp = s.top();
129. s.pop();
130. **if** (temp == bTNode) {
131. temp->data = value;
132. **return** **true**;
133. }
134. **if** (temp->lchild) s.push(temp->lchild);
135. **if** (temp->rchild) s.push(temp->rchild);
136. }
137. **return** **false**;
138. }

141. // 返回结点 bTNode 的双亲
142. BTNode\* Parent(BiTree bTree, BTNode \*bTNode) {
143. **if** (bTree == bTNode) **return** NULL;
144. stack<BiTree> s;
145. s.push(bTree);
146. BiTree temp;
147. **while** (!s.empty()) {
148. temp = s.top();
149. s.pop();
150. **if** (temp->lchild) {
151. **if** (temp->lchild == bTNode) **return** temp;
152. s.push(temp->lchild);
153. }
154. **if** (temp->rchild) {
155. **if** (temp->rchild == bTNode) **return** temp;
156. s.push(temp->rchild);
157. }
158. }
159. **return** NULL;
160. }

163. // 返回结点 bTNode 的左孩子
164. BTNode\* LeftChild(BiTree bTree, BTNode \*bTNode) {
165. stack<BiTree> s;
166. s.push(bTree);
167. BiTree temp;
168. **while** (!s.empty()) {
169. temp = s.top();
170. s.pop();
171. **if** (temp->lchild) {
172. **if** (temp == bTNode) **return** temp->lchild;
173. s.push(temp->lchild);
174. }
175. **if** (temp->rchild) {
176. s.push(temp->rchild);
177. }
178. }
179. **return** NULL;
180. }

183. // 返回结点 bTNode 的左孩子
184. BTNode\* RightChild(BiTree bTree, BTNode \*bTNode) {
185. stack<BiTree> s;
186. s.push(bTree);
187. BiTree temp;
188. **while** (!s.empty()) {
189. temp = s.top();
190. s.pop();
191. **if** (temp->lchild) {
192. s.push(temp->lchild);
193. }
194. **if** (temp->rchild) {
195. **if** (temp == bTNode) **return** temp->rchild;
196. s.push(temp->rchild);
197. }
198. }
199. **return** NULL;
200. }

203. // 返回结点 bTNode 的左兄弟
204. BTNode\* LeftSibling(BiTree bTree, BTNode \*bTNode) {
205. **if** (bTree == bTNode) **return** NULL;
206. stack<BiTree> s;
207. s.push(bTree);
208. BiTree temp;
209. **while** (!s.empty()) {
210. temp = s.top();
211. s.pop();
212. **if** (temp->lchild) {
213. s.push(temp->lchild);
214. }
215. **if** (temp->rchild) {
216. **if** (temp->rchild == bTNode && temp->lchild) **return** temp->lchild;
217. s.push(temp->rchild);
218. }
219. }
220. **return** NULL;
221. }

224. // 返回结点 bTNode 的右兄弟
225. BTNode\* RightSibling(BiTree bTree, BTNode \*bTNode) {
226. **if** (bTree == bTNode) **return** NULL;
227. stack<BiTree> s;
228. s.push(bTree);
229. BiTree temp;
230. **while** (!s.empty()) {
231. temp = s.top();
232. s.pop();
233. **if** (temp->lchild) {
234. **if** (temp->lchild == bTNode && temp->rchild) **return** temp->rchild;
235. s.push(temp->lchild);
236. }
237. **if** (temp->rchild) {
238. s.push(temp->rchild);
239. }
240. }
241. **return** NULL;
242. }

245. // 将树insertBTree插入到树BTree的bTNode结点的左或右(condition=0或1)子树中
246. **bool** InsertChild(BiTree bTree, **const** BTNode \*bTNode, **int** condition, BiTree insertBTree) {
247. stack<BiTree> s;
248. s.push(bTree);
249. BiTree temp, tempLChild = NULL, tempRChild = NULL;
250. **while** (!s.empty()) {
251. temp = s.top();
252. s.pop();
253. **if** (temp->lchild) {
254. tempLChild = temp->lchild;
255. s.push(temp->lchild);
256. }
257. **if** (temp->rchild) {
258. tempRChild = temp->rchild;
259. s.push(temp->rchild);
260. }
261. **if** (temp == bTNode) {
262. **if** (temp->lchild && temp->rchild) **return** **false**;
263. **if** (condition) {
264. temp->rchild = insertBTree;
265. insertBTree->rchild = tempRChild;
266. }
267. **else** {
268. temp->lchild = insertBTree;
269. insertBTree->rchild = tempLChild;
270. }
271. **return** **true**;
272. }
273. }
274. **return** **false**;
275. }

278. // 删除BTNode结点的左或右子树(condition=0或1)
279. **bool** DeleteChild(BiTree bTree, BTNode \*bTNode, **int** condition) {
280. stack<BiTree> s;
281. s.push(bTree);
282. BiTree temp, tempChild;
283. **while** (!s.empty()) {
284. temp = s.top();
285. s.pop();
286. **if** (temp == bTNode) {
287. **if** (condition) {
288. DestoryBiTree(temp->rchild);
289. }
290. **else** {
291. DestoryBiTree(temp->lchild);
292. }
293. **return** **true**;
294. }
295. **if** (temp->lchild) s.push(temp->lchild);
296. **if** (temp->rchild) s.push(temp->rchild);
297. }
298. **return** **false**;
299. }

302. // 先序遍历，递归
303. **void** PreOrderTraverse(BiTree bTree, **int** tab) {
304. **if** (bTree) {
305. **for** (**int** i=0; i<tab; i++) cout<<"  ";
306. cout<<bTree->data<<endl;
307. PreOrderTraverse(bTree->lchild, tab+1);
308. PreOrderTraverse(bTree->rchild, tab+1);
309. }
310. }

313. // 中序遍历，递归
314. **void** InOrderTraverse(BiTree bTree, **int** tab) {
315. **if** (bTree) {
316. InOrderTraverse(bTree->lchild, tab+1);
317. **for** (**int** i=0; i<tab; i++) cout<<"  ";
318. cout<<bTree->data<<endl;
319. InOrderTraverse(bTree->rchild, tab+1);
320. }
321. }

324. // 后序遍历，递归
325. **void** PostOrderTraverse(BiTree bTree, **int** tab) {
326. **if**(bTree) {
327. PostOrderTraverse(bTree->lchild, tab+1);
328. PostOrderTraverse(bTree->rchild, tab+1);
329. **for** (**int** i=0; i<tab; i++) cout<<"  ";
330. cout<<bTree->data<<endl;
331. }
332. }

335. // 层次遍历，非递归
336. **void** LevelOrderTraverse(BiTree bTree) {
337. **if** (!bTree) {
338. cout<<"该树为空！";
339. **return**;
340. }
342. cout<<"层次遍历：";
343. queue<BiTree> q;
344. q.push(bTree);
345. BiTree temp;
346. **while** (!q.empty()) {
347. temp = q.front();
348. cout<<temp->data;
349. **if** (temp->lchild) q.push(temp->lchild);
350. **if** (temp->rchild) q.push(temp->rchild);
351. q.pop();
352. }
353. cout<<endl;
354. }

357. **int** main() {
359. string preOrder = "ABDEGIHKLCFJ";
360. string inOrder = "DBIGEKHLACFJ";
362. BiTree bTree;
363. CreateBiTree(bTree, preOrder, inOrder);
364. cout<<"\n建树成功\n\n";
366. // 递归实现
367. cout<<"先序遍历：\n";
368. PreOrderTraverse(bTree, 0);
369. cout<<endl;
371. cout<<"中序遍历：\n";
372. InOrderTraverse(bTree, 0);
373. cout<<endl;
375. cout<<"后序遍历：\n";
376. PostOrderTraverse(bTree, 0);
377. cout<<endl<<endl;
379. // 返回树的深度
380. cout<<"树的深度："<<BiTreeDepth(bTree);
381. cout<<endl<<endl;
383. // 层次遍历
384. LevelOrderTraverse(bTree);
385. cout<<endl;
387. // 测试获取值
388. BTNode \*bTNode = bTree->lchild->rchild->rchild;
389. cout<<"测试结点的值为："<<Value(bTree,  bTNode)<<endl;
390. cout<<"修改测试结点的值为X\n";
391. Assign(bTree, bTNode, 'X');
392. cout<<"测试结点的值为："<<Value(bTree,  bTNode)<<endl;
393. cout<<"测试结点的双亲值为："<<Value(bTree, Parent(bTree,  bTNode))<<endl;
394. cout<<"测试结点的左孩子值为："<<Value(bTree, LeftChild(bTree,  bTNode))<<endl;
395. cout<<"测试结点的右孩子值为："<<Value(bTree, RightChild(bTree,  bTNode))<<endl;
396. cout<<"测试结点的左兄弟值为："<<Value(bTree, LeftSibling(bTree,  bTNode))<<endl;
397. cout<<"测试结点的右兄弟值为："<<Value(bTree, RightSibling(bTree,  bTNode))<<endl<<endl;
399. cout<<"中序遍历：\n";
400. InOrderTraverse(bTree, 0);
401. cout<<endl;
403. // 测试插入和删除
404. preOrder = "QYZ";
405. inOrder = "ZYQ";
406. BiTree insertBTree;
407. CreateBiTree(insertBTree, preOrder, inOrder);
408. cout<<"\n建插入树成功\n\n";
410. cout<<"中序遍历插入树：\n";
411. InOrderTraverse(insertBTree, 0);
412. cout<<endl;
414. bTNode = bTree->lchild->rchild->lchild;
415. InsertChild(bTree, bTNode, 0, insertBTree);
417. cout<<"插入树后中序遍历：\n";
418. InOrderTraverse(bTree, 0);
419. cout<<endl;
421. DeleteChild(bTree, bTNode, 0);
422. cout<<"删除树后中序遍历：\n";
423. InOrderTraverse(bTree, 0);
424. cout<<endl;
426. // 销毁树
427. DestoryBiTree(bTree);
428. }

**2、**

**题目：**实现二叉树的先序、中序、后序遍历，用递归和非递归方法；实现层次遍历。

**算法思想：**

1. 层次遍历和先中后序递归遍历与上一题一致。
2. 对于先中后序的非递归遍历，我采用的算法是：模仿递归时函数栈的实现过程。具体思想如下：

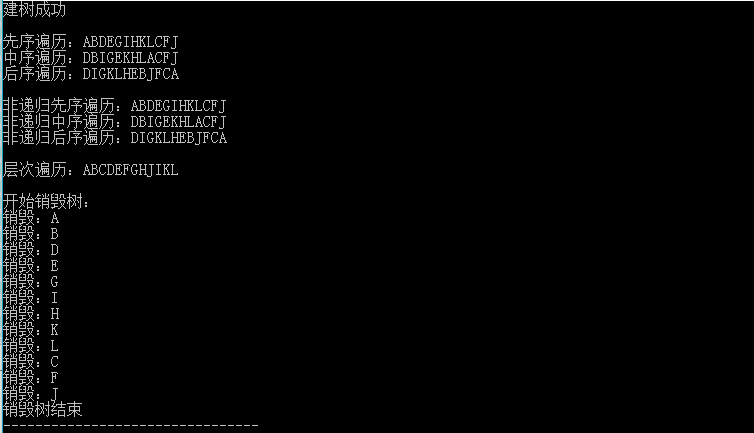
A.因为，递归时每个函数就是函数栈中一个单位。每次函数调用下一个函数时，会在函数中设置断点，将未执行的内容保存在它所在的栈单元中。当回到这个栈单元时，会继续其中该函数中未执行的内容。

B.以后序遍历为例，后序遍历的递归过程是左-右-根。那当函数在访问左子树的时候，会调用下一个函数，那右子树和根的信息将会保留在这个栈单元中，等栈顶回到这个单元时再使用。

C.那么，我们就可以模仿这个过程。定义一个单位为树节点值的栈，依照后序遍历的顺序，依次将根-右-左节点放入栈中。每次从栈中取出元素后，若左右指针都为空则输出数据，否则继续将该元素的根-右-左依次放入栈中。此处要注意的是，放入栈中的根应该讲左右指针都设为空，因为这个根节点只是用来输出数据的。

D.但是，这个算法是有缺陷的，因为不能将结构体指针作为栈的单位，不然放入根节点时每次都要申请新空间，但最后不能消除（又似乎可以）。

**运行结果：**



**结果分析：**成功。

**源程序：**

1. /\*
2. 实现二叉树的先序、中序、后序遍历，用递归和非递归方法；实现层次遍历。
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<string>
13. #include<queue>
14. #include<stack>
15. **using** **namespace** std;
17. **typedef** **struct** BTNode {
18. **char** data;
19. BTNode \*lchild, \*rchild;
20. }BTNode, \*BiTree;

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

54. // 先序遍历，递归
55. **void** preOrderTraverse(BiTree bTree) {
56. **if** (bTree) {
57. cout<<bTree->data;
58. preOrderTraverse(bTree->lchild);
59. preOrderTraverse(bTree->rchild);
60. }
61. }

64. // 先序遍历，非递归
65. **void** preOrderTraverseOther(BiTree bTree) {
66. **if** (!bTree) {
67. cout<<"该树为空！";
68. **return**;
69. }
71. cout<<"非递归先序遍历：";
72. stack<BTNode> s;
73. s.push(\*bTree);
74. BTNode temp;
75. **while** (!s.empty()) {
76. temp = s.top();
77. s.pop();
78. cout<<temp.data;
79. **if** (temp.rchild) s.push(\*temp.rchild);
80. **if** (temp.lchild) s.push(\*temp.lchild);
81. }
82. cout<<endl;
83. }

86. // 中序遍历，递归
87. **void** inOrderTraverse(BiTree bTree) {
88. **if** (bTree) {
89. inOrderTraverse(bTree->lchild);
90. cout<<bTree->data;
91. inOrderTraverse(bTree->rchild);
92. }
93. }

96. // 中序遍历，非递归
97. **void** inOrderTraverseOther(BiTree bTree) {
98. **if** (!bTree) {
99. cout<<"该树为空！";
100. **return**;
101. }
103. cout<<"非递归中序遍历：";
104. stack<BTNode> s;
105. s.push(\*bTree);
106. BTNode temp, dataBTree;
107. dataBTree.lchild = dataBTree.rchild = NULL;
108. **while** (!s.empty()) {
109. temp = s.top();
110. s.pop();
112. **if** (!temp.lchild && !temp.rchild) cout<<temp.data;
113. **else** {
114. dataBTree.data = temp.data;
116. **if** (temp.rchild) s.push(\*temp.rchild);
117. s.push(dataBTree);
118. **if** (temp.lchild) s.push(\*temp.lchild);
119. }
120. }
121. cout<<endl;
122. }

125. // 后序遍历，递归
126. **void** postOrderTraverse(BiTree bTree) {
127. **if**(bTree) {
128. postOrderTraverse(bTree->lchild);
129. postOrderTraverse(bTree->rchild);
130. cout<<bTree->data;
131. }
132. }

135. // 后序遍历，非递归
136. **void** postOrderTraverseOther(BiTree bTree) {
137. **if** (!bTree) {
138. cout<<"该树为空！";
139. **return**;
140. }
142. cout<<"非递归后序遍历：";
143. stack<BTNode> s;
144. s.push(\*bTree);
145. BTNode temp, dataBTree;
146. dataBTree.lchild = dataBTree.rchild = NULL;
147. **while** (!s.empty()) {
148. temp = s.top();
149. s.pop();
151. **if** (!temp.lchild && !temp.rchild) cout<<temp.data;
152. **else** {
153. dataBTree.data = temp.data;
155. s.push(dataBTree);
156. **if** (temp.rchild) s.push(\*temp.rchild);
157. **if** (temp.lchild) s.push(\*temp.lchild);
158. }
159. }
160. cout<<endl;
161. }

164. // 层次遍历
165. **void** levelOrderTraverse(BiTree bTree) {
166. **if** (!bTree) {
167. cout<<"该树为空！";
168. **return**;
169. }
171. cout<<"层次遍历：";
172. queue<BiTree> q;
173. q.push(bTree);
174. BiTree temp;
175. **while** (!q.empty()) {
176. temp = q.front();
177. cout<<temp->data;
178. **if** (temp->lchild) q.push(temp->lchild);
179. **if** (temp->rchild) q.push(temp->rchild);
180. q.pop();
181. }
182. cout<<endl;
183. }

186. // 销毁树
187. **void** DestoryBiTree(BiTree &bTree) {
188. **if** (!bTree) {
189. cout<<"该树为空！";
190. **return**;
191. }
193. cout<<"开始销毁树：\n";
194. // 销毁树，只能将树结点指针传进去，因为free要传入的是指针
195. stack<BiTree> s;
196. s.push(bTree);
197. bTree = NULL;
198. BiTree temp;
200. **while** (!s.empty()) {
201. temp = s.top();
202. s.pop();
203. cout<<"销毁："<<temp->data<<endl;
204. **if** (temp->rchild) s.push(temp->rchild);
205. **if** (temp->lchild) s.push(temp->lchild);
206. free(temp);
207. }
208. cout<<"销毁树结束";
209. }

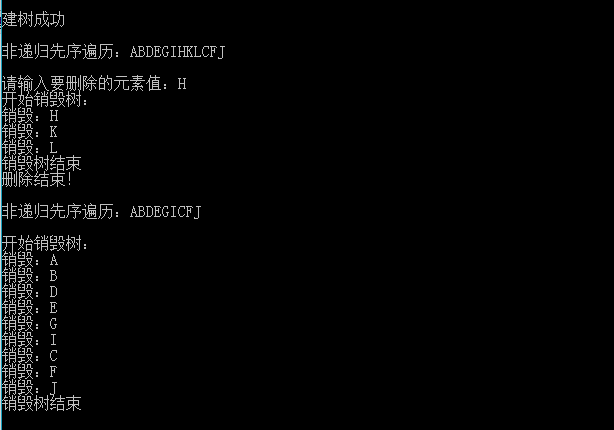
212. **int** main() {
214. string preOrder = "ABDEGIHKLCFJ";
215. string inOrder = "DBIGEKHLACFJ";
217. BiTree bTree;
218. CreateBiTree(bTree, preOrder, inOrder);
219. cout<<"\n建树成功\n\n";
221. // 递归实现
222. cout<<"先序遍历：";
223. preOrderTraverse(bTree);
224. cout<<endl;
226. cout<<"中序遍历：";
227. inOrderTraverse(bTree);
228. cout<<endl;
230. cout<<"后序遍历：";
231. postOrderTraverse(bTree);
232. cout<<endl<<endl;
234. // 非递归实现
235. preOrderTraverseOther(bTree);
236. inOrderTraverseOther(bTree);
237. postOrderTraverseOther(bTree);
238. cout<<endl;
240. // 层次遍历
241. levelOrderTraverse(bTree);
242. cout<<endl;
244. // 销毁树
245. DestoryBiTree(bTree);
246. }

**3、**

**题目：**编程实现，对二叉树中每个元素值为x的结点，删除以它为根的子树，并释放相应空间。

**算法思想：**非递归先序遍历，然后找到这个结点后，调用destory函数即可。

**运行结果：**



**结果分析：**成功。

**源程序：**

1. //编程实现，对二叉树中每个元素值为x的结点，删除以它为根的子树，并释放相应空间。
2. #include<iostream>
3. #include<stdlib.h>
4. #include<string>
5. #include<queue>
6. #include<stack>
7. **using** **namespace** std;
9. **typedef** **struct** BTNode {
10. **char** data;
11. BTNode \*lchild, \*rchild;
12. }BTNode, \*BiTree;

15. // 先序+中序 建树法
16. **void** CreateBiTree(BiTree &bTree, string preOrder, string inOrder) {
17. bTree = (BTNode\*)malloc(**sizeof**(BTNode));
18. bTree->data = preOrder[0];
20. **int** pos = inOrder.find(preOrder[0]);
21. string inOrderL = inOrder.substr(0, pos);
22. string inOrderR = inOrder.substr(pos + 1, inOrder.length() - 1);
23. string preOrderL = preOrder.substr(1, inOrderL.length());
24. string preOrderR = preOrder.substr(inOrderL.length() + 1, preOrder.length() - 1);
26. **int** isEmptyL = 0, isEmptyR = 0;     // 用来判断左右结点是否为空的标志位
27. **if** (inOrderL.empty()) {
28. bTree->lchild = NULL;
29. isEmptyL = 1;
30. }
31. **if** (inOrderR.empty()) {
32. bTree->rchild = NULL;
33. isEmptyR = 1;
34. }
35. **if** (!isEmptyL) {
36. cout<<preOrderL<<endl<<inOrderL<<endl;
37. CreateBiTree(bTree->lchild, preOrderL, inOrderL);
38. }
39. **if** (!isEmptyR) {
40. cout<<preOrderR<<endl<<inOrderR<<endl;
41. CreateBiTree(bTree->rchild, preOrderR, inOrderR);
42. }
43. }

46. // 先序遍历，非递归
47. **void** preOrderTraverseOther(BiTree bTree) {
48. **if** (!bTree) {
49. cout<<"该树为空！";
50. **return**;
51. }
53. cout<<"非递归先序遍历：";
54. stack<BTNode> s;
55. s.push(\*bTree);
56. BTNode temp;
57. **while** (!s.empty()) {
58. temp = s.top();
59. s.pop();
60. cout<<temp.data;
61. **if** (temp.rchild) s.push(\*temp.rchild);
62. **if** (temp.lchild) s.push(\*temp.lchild);
63. }
64. cout<<endl;
65. }

68. // 销毁树
69. **void** DestoryBiTree(BiTree &bTree) {
70. **if** (!bTree) {
71. cout<<"该树为空！";
72. **return**;
73. }
75. cout<<"开始销毁树：\n";
76. // 销毁树，只能将树结点指针传进去，因为free要传入的是指针
77. stack<BiTree> s;
78. s.push(bTree);
79. bTree = NULL;
80. BiTree temp;
82. **while** (!s.empty()) {
83. temp = s.top();
84. s.pop();
85. cout<<"销毁："<<temp->data<<endl;
86. **if** (temp->rchild) s.push(temp->rchild);
87. **if** (temp->lchild) s.push(temp->lchild);
88. free(temp);
89. }
90. cout<<"销毁树结束"<<endl;
91. }

94. **int** main() {
96. string preOrder = "ABDEGIHKLCFJ";
97. string inOrder = "DBIGEKHLACFJ";
99. BiTree bTree;
100. CreateBiTree(bTree, preOrder, inOrder);
101. cout<<"\n建树成功\n\n";
103. preOrderTraverseOther(bTree);
104. cout<<endl;
106. **char** ch;
107. cout<<"请输入要删除的元素值：";
108. cin>>ch;
110. stack<BiTree> s;
111. s.push(bTree);
112. BiTree tempTree;
113. **while** (!s.empty()) {
114. tempTree = s.top();
115. s.pop();
116. **if** (tempTree->rchild) {
117. **if** (tempTree->rchild->data == ch) {
118. DestoryBiTree(tempTree->rchild);
119. **break**;
120. }
121. **else** s.push(tempTree->rchild);
122. }
123. **if** (tempTree->lchild) {
124. **if** (tempTree->lchild->data == ch) {
125. DestoryBiTree(tempTree->lchild);
126. **break**;
127. }
128. **else** s.push(tempTree->lchild);
129. }
130. }
132. cout<<"删除结束！\n\n";
133. preOrderTraverseOther(bTree);
134. cout<<endl;
136. // 销毁树
137. DestoryBiTree(bTree);
138. }

**4、**

**题目：**问题描述

　　消除类游戏是深受大众欢迎的一种游戏，游戏在一个包含有n行m列的游戏棋盘上进行，棋盘的每一行每一列的方格上放着一个有颜色的棋子，当一行或一列上有连续三个或更多的相同颜色的棋子时，这些棋子都被消除。当有多处可以被消除时，这些地方的棋子将同时被消除。

　　现在给你一个n行m列的棋盘，棋盘中的每一个方格上有一个棋子，请给出经过一次消除后的棋盘。

　　请注意：一个棋子可能在某一行和某一列同时被消除。

输入格式

　　输入第一行包含两个整数n, m，用空格分隔，分别表示棋盘的行数和列数。

　　接下来n行，每行m个整数，用空格分隔，分别表示每一个方格中的棋子的颜色。颜色使用1至9编号。

输出格式

　　输出n行，每行m个整数，相邻的整数之间使用一个空格分隔，表示经过一次消除后的棋盘。如果一个方格中的棋子被消除，则对应的方格输出0，否则输出棋子的颜色编号。

样例输入

4 5

2 2 3 1 2

3 4 5 1 4

2 3 2 1 3

2 2 2 4 4

样例输出

2 2 3 0 2

3 4 5 0 4

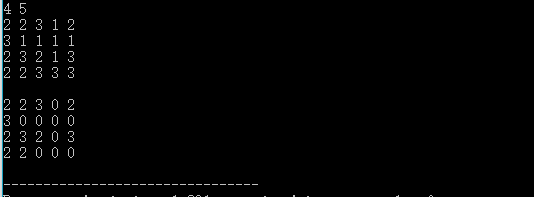
2 3 2 0 3

0 0 0 4 4

**算法思想：**

1. 设置一个点结构体，记录最初值和最后值。然后建一个基于这个点结构体的二维数组，将输入的值同时记录在最初值和最后值中。
2. 按行的顺序进行遍历，对第i行第j列的数，依次向后比较，若连续且相等数的个数大于等于3，那么就将这一系列位置数的最后值变为0。
3. 按列的顺序进行遍历，算法思想同上。

**运行结果：**



**结果分析：**成功。

**源程序：**

1. #include<stdio.h>
2. #include<stdlib.h>
4. **typedef** **struct** {
5. **int** pri;
6. **int** last;
7. }Elem;
9. **int** main() {
10. **int** n = 0, m = 0;
11. scanf("%d%d", &n, &m);
13. Elem W[n][m];
14. **for** (**int** i = 0; i < n; i++) {
15. **for** (**int** j = 0; j < m; j++) {
16. scanf("%d", &W[i][j].pri);
17. W[i][j].last = W[i][j].pri;
18. }
19. }
21. // 消每一行
22. **for** (**int** i = 0; i < n; i++) {
23. **for** (**int** j = 0; j < m-1;) {
24. **int** k = j+1;
25. **int** same = 1;   // 从当前开始与当前位相同且连续的数的个数
26. **while** ((k < m) && (W[i][j].pri == W[i][k].pri)) {
27. k++;
28. same++;
29. }
30. **if** (same >= 3) {
31. k = j;
32. **while** (k <= j+same-1) {
33. W[i][k].last = 0;
34. k++;
35. }
36. }
37. j += same;
38. }
39. }
41. // 消每一列，从列开始遍历
42. **for** (**int** col = 0; col < m; col++) {
43. **for** (**int** row = 0; row < n-1;) {
44. **int** k = row+1;
45. **int** same = 1;   // 从下一位开始与当前位相同且连续的数的个数
46. **while** ((k < n) && (W[row][col].pri == W[k][col].pri)) {
47. k++;
48. same++;
49. }
50. **if** (same >= 3) {
51. k = row;
52. **while** (k <= row+same-1) {
53. W[k][col].last = 0;
54. k++;
55. }
56. }
57. row += same;
58. //printf("%d  %d\n", col, row);
59. }
60. }
62. printf("\n");
63. **for** (**int** i = 0; i < n; i++) {
64. **for** (**int** j = 0; j < m; j++) {
65. printf("%d ", W[i][j].last);
66. }
67. printf("\n");
68. }
70. }

**5、**

**题目：**



**算法思想：**递归。

**运行结果：**



**结果分析：**成功。

**源程序：**

1. #include<iostream>
2. #include<stdlib.h>
3. #include<string>
4. #include<queue>
5. #include<stack>
6. **using** **namespace** std;
8. **typedef** **struct** BTNode {
9. **char** data;
10. BTNode \*lchild, \*rchild;
11. }BTNode, \*BiTree;

14. // 先序+中序 建树法
15. **void** CreateBiTree(BiTree &bTree, string preOrder, string inOrder) {
16. bTree = (BTNode\*)malloc(**sizeof**(BTNode));
17. bTree->data = preOrder[0];
19. **int** pos = inOrder.find(preOrder[0]);
20. string inOrderL = inOrder.substr(0, pos);
21. string inOrderR = inOrder.substr(pos + 1, inOrder.length() - 1);
22. string preOrderL = preOrder.substr(1, inOrderL.length());
23. string preOrderR = preOrder.substr(inOrderL.length() + 1, preOrder.length() - 1);
25. **int** isEmptyL = 0, isEmptyR = 0;     // 用来判断左右结点是否为空的标志位
26. **if** (inOrderL.empty()) {
27. bTree->lchild = NULL;
28. isEmptyL = 1;
29. }
30. **if** (inOrderR.empty()) {
31. bTree->rchild = NULL;
32. isEmptyR = 1;
33. }
34. **if** (!isEmptyL) {
35. cout<<preOrderL<<endl<<inOrderL<<endl;
36. CreateBiTree(bTree->lchild, preOrderL, inOrderL);
37. }
38. **if** (!isEmptyR) {
39. cout<<preOrderR<<endl<<inOrderR<<endl;
40. CreateBiTree(bTree->rchild, preOrderR, inOrderR);
41. }
42. }

45. // 先序遍历，非递归
46. **void** preOrderTraverseOther(BiTree bTree) {
47. **if** (!bTree) {
48. cout<<"该树为空！";
49. **return**;
50. }
52. cout<<"非递归先序遍历：";
53. stack<BTNode> s;
54. s.push(\*bTree);
55. BTNode temp;
56. **while** (!s.empty()) {
57. temp = s.top();
58. s.pop();
59. cout<<temp.data;
60. **if** (temp.rchild) s.push(\*temp.rchild);
61. **if** (temp.lchild) s.push(\*temp.lchild);
62. }
63. cout<<endl;
64. }

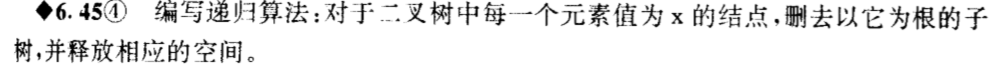
67. // 销毁树
68. **void** DestoryBiTree(BiTree &bTree) {
69. **if** (!bTree) {
70. cout<<"该树为空！";
71. **return**;
72. }
74. cout<<"开始销毁树：\n";
75. // 销毁树，只能将树结点指针传进去，因为free要传入的是指针
76. stack<BiTree> s;
77. s.push(bTree);
78. bTree = NULL;
79. BiTree temp;
81. **while** (!s.empty()) {
82. temp = s.top();
83. s.pop();
84. cout<<"销毁："<<temp->data<<endl;
85. **if** (temp->rchild) s.push(temp->rchild);
86. **if** (temp->lchild) s.push(temp->lchild);
87. free(temp);
88. }
89. cout<<"销毁树结束"<<endl;
90. }

93. // 计算树中叶子数
94. **int** leafSum(BiTree bTree) {
95. **int** sum = 0;
96. **if** (!bTree->lchild && !bTree->rchild) {
97. **return** 1;
98. }
99. **else** {
100. **if** (bTree->lchild) sum += leafSum(bTree->lchild);
101. **if** (bTree->rchild) sum += leafSum(bTree->rchild);
102. }
103. **return** sum;
104. }

107. **int** main() {
109. string preOrder = "ABDEGIHKLCFJ";
110. string inOrder = "DBIGEKHLACFJ";
112. BiTree bTree;
113. CreateBiTree(bTree, preOrder, inOrder);
114. cout<<"\n建树成功\n\n";
116. preOrderTraverseOther(bTree);
117. cout<<endl;
119. cout<<"叶子个数为："<<leafSum(bTree)<<endl<<endl;
121. // 销毁树
122. DestoryBiTree(bTree);
123. }

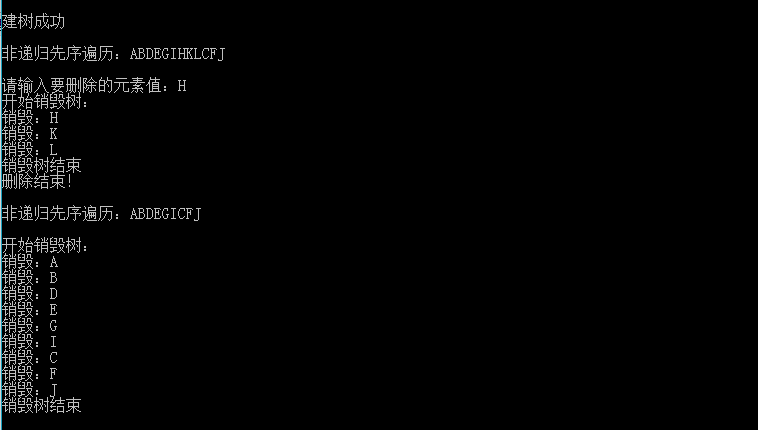
**6、**

**题目：**



**算法思想：**递归删树。

**运行结果：**



**结果分析：**成功。

**源程序：**

1. // 用递归算法编程实现，对二叉树中每个元素值为x的结点，删除以它为根的子树，并释放相应空间。
2. #include<iostream>
3. #include<stdlib.h>
4. #include<string>
5. #include<queue>
6. #include<stack>
7. **using** **namespace** std;
9. **typedef** **struct** BTNode {
10. **char** data;
11. BTNode \*lchild, \*rchild;
12. }BTNode, \*BiTree;

15. // 先序+中序 建树法
16. **void** CreateBiTree(BiTree &bTree, string preOrder, string inOrder) {
17. bTree = (BTNode\*)malloc(**sizeof**(BTNode));
18. bTree->data = preOrder[0];
20. **int** pos = inOrder.find(preOrder[0]);
21. string inOrderL = inOrder.substr(0, pos);
22. string inOrderR = inOrder.substr(pos + 1, inOrder.length() - 1);
23. string preOrderL = preOrder.substr(1, inOrderL.length());
24. string preOrderR = preOrder.substr(inOrderL.length() + 1, preOrder.length() - 1);
26. **int** isEmptyL = 0, isEmptyR = 0;     // 用来判断左右结点是否为空的标志位
27. **if** (inOrderL.empty()) {
28. bTree->lchild = NULL;
29. isEmptyL = 1;
30. }
31. **if** (inOrderR.empty()) {
32. bTree->rchild = NULL;
33. isEmptyR = 1;
34. }
35. **if** (!isEmptyL) {
36. cout<<preOrderL<<endl<<inOrderL<<endl;
37. CreateBiTree(bTree->lchild, preOrderL, inOrderL);
38. }
39. **if** (!isEmptyR) {
40. cout<<preOrderR<<endl<<inOrderR<<endl;
41. CreateBiTree(bTree->rchild, preOrderR, inOrderR);
42. }
43. }

46. // 先序遍历，非递归
47. **void** preOrderTraverseOther(BiTree bTree) {
48. **if** (!bTree) {
49. cout<<"该树为空！";
50. **return**;
51. }
53. cout<<"非递归先序遍历：";
54. stack<BTNode> s;
55. s.push(\*bTree);
56. BTNode temp;
57. **while** (!s.empty()) {
58. temp = s.top();
59. s.pop();
60. cout<<temp.data;
61. **if** (temp.rchild) s.push(\*temp.rchild);
62. **if** (temp.lchild) s.push(\*temp.lchild);
63. }
64. cout<<endl;
65. }

68. // 销毁树
69. **void** DestoryBiTree(BiTree &bTree) {
70. **if** (!bTree) {
71. cout<<"该树为空！";
72. **return**;
73. }
75. cout<<"开始销毁树：\n";
76. // 销毁树，只能将树结点指针传进去，因为free要传入的是指针
77. stack<BiTree> s;
78. s.push(bTree);
79. bTree = NULL;
80. BiTree temp;
82. **while** (!s.empty()) {
83. temp = s.top();
84. s.pop();
85. cout<<"销毁："<<temp->data<<endl;
86. **if** (temp->rchild) s.push(temp->rchild);
87. **if** (temp->lchild) s.push(temp->lchild);
88. free(temp);
89. }
90. cout<<"销毁树结束"<<endl;
91. }

94. // 删除以x为根的子树
95. **void** DeleteBiTree(BiTree &x, **char** ch) {
96. **if** (!x) **return**;
97. **if** (x->data == ch) DestoryBiTree(x);
98. **else** {
99. **if** (x->lchild) DeleteBiTree(x->lchild, ch);
100. **if** (x->rchild) DeleteBiTree(x->rchild, ch);
101. }
102. }

105. **int** main() {
107. string preOrder = "ABDEGIHKLCFJ";
108. string inOrder = "DBIGEKHLACFJ";
110. BiTree bTree;
111. CreateBiTree(bTree, preOrder, inOrder);
112. cout<<"\n建树成功\n\n";
114. preOrderTraverseOther(bTree);
115. cout<<endl;
117. **char** ch;
118. cout<<"请输入要删除的元素值：";
119. cin>>ch;
121. DeleteBiTree(bTree, ch);
123. cout<<"删除结束！\n\n";
124. preOrderTraverseOther(bTree);
125. cout<<endl;
127. // 销毁树
128. DestoryBiTree(bTree);
129. }