# 项目选题

使用二叉链表存储森林，并完成增删查功能，以及对二叉链表的先序、中序以及后序遍历。具体步骤为先创建若干个树组成森林，通过森林转叉树的算法将森林以二叉树的形式存储。并以森林构成的二叉树完成各种二叉树的功能。

# 项目设计

整个项目可以划分成三个部分，分别是树的创建、树转二叉树，二叉树的合并。其中使用树结构的数组表示森林，将其中的没一棵树转换成二叉树后，在将所有的二叉树相连，最终实现森林转二叉树的效果。

# 项目实现

1. 编译环境：Visual Studio Code
2. 编码语言：C++
3. 编写目标：完成三部分功能需求的代码
4. 测试方法：分别编写三部分代码，各自测试通过后，在合并测试。各测试结点多数使用白盒测试，跟踪每一步执行的效果，检查是否正确。最后对整体使用黑盒测试检查是否存在错误。
5. 编写注释
6. 书写报告

# 项目代码

1. #include <iostream>
2. #include <iomanip>
3. #include <cmath>
4. #include <cstring>
5. **using** **namespace** std;
7. #define MAX\_TREE\_SIZE 20 // 一棵树最多可以拥有的结点数
8. **int** nodeNumber = 0;
9. /\* 树的结点 \*/
10. **typedef** **struct** PTNode{
11. **char** data;
12. **int** parent;
13. **int** firstchild;
14. }TreeNode;
15. /\* 树 \*/
16. **typedef** **struct**{
17. TreeNode nodes[MAX\_TREE\_SIZE];
18. **int** count = 0;
19. }Tree;
20. /\* 二叉树 \*/
21. **typedef** **struct** BinaryTree {
22. **char** data;
23. BinaryTree \*Lchild, \*Rchild;
24. }BinaryTree;
26. /\* 树的相关函数 \*/
27. TreeNode\* CreateTreeNode(**char** nodeData, **int** parentData);
28. Tree\* CreatTree(**int** number,**char** dataInfo[], **int** parentInfo[]);
29. **void** SortTreeNode(**int** number, **char**\* dataInfo, **int**\* parentInfo);
30. **int** getTreeDeep(Tree\* tree);
31. **int** numlen(**int** num);
32. **void** PrintLayout(**int** data, **int** length);
33. **void** PrintLayout(**char** data, **int** length);
34. **void** ShowTree(Tree\* tree);
35. /\* 树转二叉树的相关函数 \*/
36. **void** TreeToBTree(Tree\* tree);
37. BinaryTree\* ForestToBinaryTree(Tree\* tree[], **int** count);
38. /\* 二叉树的相关函数 \*/
39. BinaryTree\* CreateTreeNode(**char** data);
40. BinaryTree\* CreateTree(Tree\* tree, **int** i);
41. **int** GetNodeNumber(BinaryTree\* root);
42. **void** ShowTreeMLR(BinaryTree\* root);
43. **void** ShowTreeLMR(BinaryTree\* root);
44. **void** ShowTreeLRM(BinaryTree\* root);
46. **int** main() {
47. Tree\* tree[4];
48. **int** TREECOUNT = 3;
49. **int** size;
50. // 数据，父节点
51. **char** dataInfo1[] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};
52. **int** parentInfo1[] = {-1, 0, 0, 1, 2, 2, 3, 3, 3, 4};
53. size = **sizeof**(dataInfo1) / **sizeof**(dataInfo1[0]);
54. tree[0] = CreatTree(size, dataInfo1, parentInfo1);
55. //ShowTree(tree[0]);
57. **char** dataInfo2[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'};
58. **int** parentInfo2[] = {-1, 0, 0, 0, 2, 2, 3, 3, 3, 4};
59. size = **sizeof**(dataInfo2) / **sizeof**(dataInfo2[0]);
60. tree[1] = CreatTree(size, dataInfo2, parentInfo2);
61. //ShowTree(tree[1]);
63. **char** dataInfo3[] = {'a', 'b'};
64. **int** parentInfo3[] = {-1, 0};
65. size = **sizeof**(dataInfo3) / **sizeof**(dataInfo3[0]);
66. tree[2] = CreatTree(size, dataInfo3, parentInfo3);
67. //ShowTree(tree[2]);
69. BinaryTree\* root = ForestToBinaryTree(tree, TREECOUNT);
70. cout << GetNodeNumber(root);
71. cout << endl;
72. ShowTreeMLR(root);
73. cout << endl;
74. ShowTreeLMR(root);
75. **return** 0;
76. }
78. /\*\*
79. \* @brief 初始化树的一个结点
80. \*
81. \* @param nodeData :结点数据
82. \* @param parentData :父节点数据
83. \* @return TreeNode\* :树结构
84. \*/
85. TreeNode\* CreateTreeNode(**char** nodeData, **int** parentData) {
86. TreeNode\* newNode = **new** TreeNode;
87. newNode->data = nodeData;
88. newNode->parent = parentData;
89. newNode->firstchild = -1;
90. **return** newNode;
91. }
93. /\*\*
94. \* @brief 创建树（双亲表示法）
95. \*
96. \* @param number :结点个数
97. \* @param dataInfo :结点数据数组
98. \* @param parentInfo :父结点数组
99. \* @return Tree\* :树结构
100. \*/
101. Tree\* CreatTree(**int** number,**char** dataInfo[], **int** parentInfo[]) {
102. **int** farther = -1;
103. Tree\* tree = **new** Tree;
104. // 先将结点排序
105. SortTreeNode(number, dataInfo, parentInfo);
106. // 遍历所有结点数组信息
107. **for** (**int** i = 0; i < number; i++) {
108. // 创建一个树的结点，将输入录入
109. tree->nodes[i] = \*CreateTreeNode(dataInfo[i], parentInfo[i]);
110. // 将第一次出现的子结点记为父节点的第一孩子结点
111. **if** (farther != parentInfo[i]) {
112. farther = parentInfo[i];
113. tree->nodes[farther].firstchild = tree->count;
114. }
115. tree->count++;
116. }
117. **return** tree;
118. }
120. /\*\*
121. \* @brief 计算数的长度（包括负号）
122. \*
123. \* @param num :需要计算的数
124. \* @return int :数的长度
125. \*/
126. **int** numlen(**int** num) {
127. **int** len = 0;
128. // 如果数为0则直接返回1
129. **if** (num == 0) **return** 1;
130. // 如果数为负数，则长度加1且对数取绝对值
131. **if** (num < 0) {
132. len++;
133. num = abs(num);
134. }
135. // 计算出数的长度
136. **for**(; num > 0; ++len)
137. num /= 10;
138. **return** len;
139. }
141. /\*\*
142. \* @brief 输出模板
143. \*
144. \* @param data :要输出的数据
145. \* @param length :包括数在内一共站几格
146. \*/
147. **void** PrintLayout(**int** data, **int** length) {
148. **int** i;
149. cout<< "| " << data;
150. **for** (i = 0; i < length - numlen(data); i++) {
151. cout<< ' ';
152. }
153. }
154. **void** PrintLayout(**char** data, **int** length) {
155. **int** i;
156. cout<< "| " << data;
157. **for** (i = 0; i < length - strlen(&data); i++) {
158. cout<< ' ';
159. }
160. }
162. /\*\*
163. \* @brief 显示树的整个内容包含下标，数据，父节点位置
164. \*
165. \* @param tree :树结构
166. \*/
167. **void** ShowTree(Tree\* tree) {
168. **int** i, j;
169. cout << "树的内容为:" << endl;
170. cout << "==============================" << endl;
171. cout << "|index| data|parent|fistchild|" << endl;
172. **for** (i = 0; i < tree->count; i++) {
173. PrintLayout(i, 4);
174. PrintLayout(tree->nodes[i].data, 4);
175. PrintLayout(tree->nodes[i].parent, 5);
176. PrintLayout(tree->nodes[i].firstchild, 8);
177. cout << "|" << endl;
178. }
179. cout << "==============================" << endl;
180. }
182. /\*\*
183. \* @brief 将数据信息和父结点信息以父节点升顺序排列
184. \*
185. \* @param number :结点个数
186. \* @param dataInfo :结点数据数组
187. \* @param parentInfo :父结点数组
188. \*/
189. **void** SortTreeNode(**int** number, **char**\* dataInfo, **int**\* parentInfo) {
190. **int** i, j;
191. **int** intTemp;
192. **char** charTemp;
193. // 冒泡排序
194. **for** (i = 0; i < number; i++) {
195. **for**(j = i; j < number - i - 1; j++){
196. **if** (parentInfo[j] > parentInfo[j+1]) {
197. charTemp = dataInfo[j + 1];
198. dataInfo[j + 1] = dataInfo[j];
199. dataInfo[j] = charTemp;
201. intTemp = parentInfo[j + 1];
202. parentInfo[j + 1] = parentInfo[j];
203. parentInfo[j] = intTemp;
204. }
205. }
206. }
207. }
209. /\*\*
210. \* @brief 获得数的深度
211. \*
212. \* @param tree :树结构
213. \* @return int :深度
214. \*/
215. **int** getTreeDeep(Tree\* tree) {
216. **int** i;
217. **int** deep = 1;
218. **int** tempDeep = 1;
219. // 遍历整个树结构
220. **for** (i = 0; i < tree->count; i++) {
221. // 找到所有叶子结点
222. **if** (tree->nodes[i].firstchild == -1) {
223. // 将叶子结点的父节点提出
224. **int** perantIndex = tree->nodes[i].parent;
225. // 沿着叶子结点一直到根结点，并计算这个叶子结点的深度
226. **while** (perantIndex != -1) {
227. perantIndex = tree->nodes[perantIndex].parent;
228. tempDeep++;
229. }
230. }
231. // 如果新计算比之前计算的要高则从新赋值
232. **if** (tempDeep > deep) {
233. deep = tempDeep;
234. }
235. tempDeep = 1;
236. }
237. **return** deep;
238. }
240. /\*\*
241. \* @brief 树转二叉树（双亲表示法）
242. \*
243. \* @param tree :树结构
244. \*/
245. **void** TreeToBTree(Tree\* tree) {
246. **int** i;
247. **int** index = 0;
248. **bool** change = **false**;
249. **int** nodeCount;
250. **for** (i = 1; i < tree->count; i++) {
251. **if** (index == tree->nodes[i].parent) {
252. // 更改第二次及以后出现相同的父结点的结点信息（即除了第一孩子结点）
253. **if** (change) {
254. // 将该点的父结点位置指向上一个结点，即左边的兄弟结点
255. tree->nodes[i].parent = i-1;
256. // 如果这个点是叶子结点，就将上一个结点的第一孩子结点指向这里
257. **if** (tree->nodes[i-1].firstchild == -1) {
258. tree->nodes[i-1].firstchild = i;
259. }
260. } **else** {
261. change = **true**;
262. }
263. } **else** {
264. index = tree->nodes[i].parent;
265. }
266. }
267. }
269. /\*\*
270. \* @brief 创建一个二叉树结点
271. \*
272. \* @param data :数据
273. \* @return BinaryTree\* :二叉树结点
274. \*/
275. BinaryTree\* CreateTreeNode(**char** data) {
276. BinaryTree\* tree = **new** BinaryTree;
277. tree->data = data;
278. tree->Lchild = NULL;
279. tree->Rchild = NULL;
280. **return** tree;
281. }
283. /\*\*
284. \* @brief 创建树
285. \*
286. \* @param data :数据数组
287. \* @param length :数据数组的长度
288. \* @param i :当前节点值的下标
289. \* @return BinaryTree\* :树的根节点
290. \*/
291. BinaryTree\* CreateTree(Tree\* tree, **int** i = 0) {
292. // 不能是叶子结点
293. **if** (i != -1) {
294. // 初始化该结点
295. BinaryTree\* root = CreateTreeNode(tree->nodes[i].data);
296. // 遍历数结构
297. **for** (**int** j = i; j < tree->count; j++) {
298. // 找到是该结点的孩子结点
299. **if** (tree->nodes[j].parent == i) {
300. //cout << root->data;
301. **if** (tree->nodes[tree->nodes[j].parent].firstchild == j) {
302. // 如果这个节点是左孩子结点
303. **if** (root->Lchild == NULL) {
304. //cout << "L" << tree->nodes[j].data << endl;
305. root->Lchild = CreateTree(tree, j);
306. }
307. } **else** {
308. // 如果这个节点是右孩子结点
309. **if** (root->Rchild == NULL) {
310. //cout << "R" << tree->nodes[j].data << endl;
311. root->Rchild = CreateTree(tree, j);
312. }
313. }
314. }
315. }
316. **return** root;
317. } **else** {
318. **return** NULL;
319. }
320. }
322. /\*\*
323. \* @brief 树转二叉树
324. \*
325. \* @param tree :树结构
326. \*/
327. BinaryTree\* ForestToBinaryTree(Tree\* tree[], **int** count) {
328. BinaryTree\* Btree[count];
329. **for** (**int** i = 0; i < count; i++) {
330. //ShowTree(tree[i]);
331. TreeToBTree(tree[i]);
332. //ShowTree(tree[i]);
333. Btree[i] = CreateTree(tree[i]);
334. }
335. **for** (**int** i = 0; i < count-1; i++) {
336. Btree[i]->Rchild = Btree[i+1];
337. }
338. **return** Btree[0];
339. }
341. /\*\*
342. \* @brief 获得树的结点数
343. \*
344. \* @param root :树的根结点
345. \* @return int :结点数
346. \*/
347. **int** GetNodeNumber(BinaryTree\* root) {
348. **if** (root != NULL) {
349. nodeNumber++;
350. GetNodeNumber(root->Lchild);
351. GetNodeNumber(root->Rchild);
352. }
353. **return** nodeNumber;
354. }
356. /\*\*
357. \* @brief 先序遍历
358. \*
359. \* @param root :根节点
360. \*/
361. **void** ShowTreeMLR(BinaryTree\* root) {
362. **if** (root != NULL) {
363. cout << root->data;
364. ShowTreeMLR(root->Lchild);
365. ShowTreeMLR(root->Rchild);
366. }
367. }
369. /\*\*
370. \* @brief 中序遍历
371. \*
372. \* @param root :根节点
373. \*/
374. **void** ShowTreeLMR(BinaryTree\* root) {
375. **if** (root != NULL) {
376. ShowTreeLMR(root->Lchild);
377. cout << root->data;
378. ShowTreeLMR(root->Rchild);
379. }
380. }
382. /\*\*
383. \* @brief 后序遍历
384. \*
385. \* @param root :根节点
386. \*/
387. **void** ShowTreeLRM(BinaryTree\* root) {
388. **if** (root != NULL) {
389. ShowTreeLRM(root->Lchild);
390. ShowTreeLRM(root->Rchild);
391. cout << root->data;
392. }
393. }

# 项目调试

在编码的过程中难免会出现结果不一致的情况，这时需要对函数进行调试，通过调试模式将程序每一次运行的结构返回出来，并与理论分析的过程向比较，分析具体是什么问题，如条件判断的边界问题，循环中断判断等等。

# 感受

通过这次的项目，感受到了理论分析与项目结果之间还存在着实际编码的巨大问题，例如对于我们来说给一串数字排序很多时候并没有严谨的流程，而是随着心感性的完成了排序，但对于计算机来说，需要一套标准的可重复使用严谨算法支持才能完成。但事实上，计算机的任务远不止对数排序这么简单，这个项目中对于将一棵树转换为二叉树我们只需要将图画出来更改连线就可以完成。但对于计算机来说如何用双亲表示法的树结构转换成链式的二叉树结构，是整个项目花费时间最长的，也是最核心的。其中也出现过偶然情况，例如对于一个数组可以正常转换但换一个数组就有问题的巧合性的解决问题，这时就不得不重新分析并重构整个代码。