# 实验4 二叉树的遍历实验指导

## 1.题目

二叉树的先序遍历、中序遍历、后序遍历的递归、非递归实现，分层遍历。

二叉树以lson-rson链接方式存储，以菜单方式设计并完成功能任务：建立并存储树，输出先序、中序、后序遍历结果，分层遍历二叉树，其中对于中序、后序的遍历操作要求采用递归和非递归两种方式。

## 2.二叉树遍历的测试用例

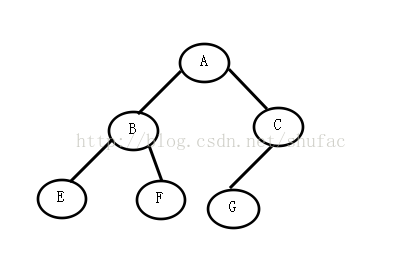


图2-1二叉树遍历用例图

如上图所示二叉树

先序遍历：A,B,E,F,C,G  
中序遍历：E,B,F,A,G,C  
后序遍历：E,F,B,G,C,A

层次遍历结果：A,B,C,E,F,G

树的高度:3

## 3.二叉树的存储与遍历算法

### 3.1 二叉树的链式存储结构

1. *//二叉树结点*
2. typedef struct BiTNode{
3. *//数据*
4. char data;
5. *//左右孩子指针*
6. struct BiTNode \*lchild,\*rchild;
7. }BiTNode,\*BiTree;

### 3.2 二叉树遍历的递归算法

#### 3.2.1二叉树的创建

**问题：二叉树的先序、中序、后序任一种遍历方式，能唯一表示二叉树图结构吗？**

创建二叉树时，输入的二叉树如果按先序次序输入，建立二叉树时就要以先序遍历建立，空树或叶子结点需要明确表示，空树或叶子结点以#代替，以图2-1为例，应该输入的字符串顺序为：ABE##F##CG###（最后一个#是结束符），“#”表示空树，如下图所示：

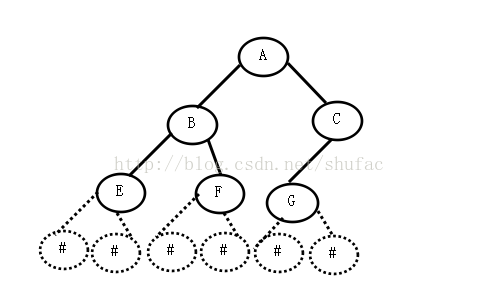


图2-1二叉树遍历用例图

通过读入一个字符串，建立二叉树的算法如下：

*//按先序序列创建二叉树*

1. int CreateBiTree(BiTree &T){
2. char data;
3. *//按先序次序输入二叉树中结点的值（一个字符），‘#’表示空树*
4. scanf("%c",&data);
5. if(data == '#'){
6. T = NULL;
7. }
8. else{
9. T = (BiTree)malloc(sizeof(BiTNode));
10. *//生成根结点*
11. T->data = data;
12. *//构造左子树*
13. CreateBiTree(T->lchild);
14. *//构造右子树*
15. CreateBiTree(T->rchild);
16. }
17. return 0;
18. }

### 3.3 二叉树的递归遍历

先序递归遍历输出二叉树的字符序列，以 '#' 结束。先访问根节点，然后访问左节点，再访问右节点。

1. *//输出一个结点的数据*
2. void Visit(BiTree T){
3. if(T->data != '#'){
4. printf("%c ",T->data);
5. }
6. }
7. *//先序遍历*
8. void PreOrder(BiTree T){
9. if(T != NULL){
10. *//访问根节点*
11. Visit(T);
12. *//访问左子结点*
13. PreOrder(T->lchild);
14. *//访问右子结点*
15. PreOrder(T->rchild);
16. }
17. }

中序递归遍历，先访问左节点，然后访问根节点，再访问右节点。

1. void InOrder(BiTree T){
2. if(T != NULL){
3. *//访问左子结点*
4. InOrder(T->lchild);
5. *//访问根节点*
6. Visit(T);
7. *//访问右子结点*
8. InOrder(T->rchild);
9. }
10. }

后序递归遍历：先访问左节点，然后访问右节点，再访问根节点。

1. void PostOrder(BiTree T){
2. if(T != NULL){
3. *//访问左子结点*
4. PostOrder(T->lchild);
5. *//访问右子结点*
6. PostOrder(T->rchild);
7. *//访问根节点*
8. Visit(T);
9. }
10. }

### 3.4二叉树的非递归遍历

**二叉树用非递归表示时，算法思路是什么？**

### 3.4.1先序遍历

【思路】：访问T->data后，将T入栈，遍历左子树；遍历完左子树返回时，栈顶元素应为T，出栈，再先序遍历T的右子树。其实每次都是走树的左分支(left)，直到左子树为空，然后开始从递归的最深处返回，再开始恢复递归现场，访问右子树。

1. */\* 先序遍历(非递归)*
2. *思路：访问T->data后，将T入栈，遍历左子树；遍历完左子树返回时，栈顶元素应为T，出栈，再先序遍历T的右子树。*
3. *\*/*
4. void PreOrder2(BiTree T){
5. stack<BiTree> stack;
6. *//p是遍历指针*
7. BiTree p = T;
8. *//栈不空或者p不空时循环*
9. while(p || !stack.empty()){
10. if(p != NULL){
11. *//存入栈中*
12. stack.push(p);
13. *//访问根节点*
14. printf("%c ",p->data);
15. *//遍历左子树*
16. p = p->lchild;
17. }
18. else{
19. *//退栈*
20. p = stack.top();
21. stack.pop();
22. *//访问右子树*
23. p = p->rchild;
24. }
25. }*//while*
26. }

### 3.4.2中序遍历

【思路】：T是要遍历树的根指针，中序遍历要求在遍历完左子树后，访问根，再遍历右子树。  
         先将T入栈，遍历左子树；遍历完左子树返回时，栈顶元素应为T，出栈，访问T->data，再中序遍历T的右子树。

1. void InOrder2(BiTree T){
2. stack<BiTree> stack;
3. *//p是遍历指针*
4. BiTree p = T;
5. *//栈不空或者p不空时循环*
6. while(p || !stack.empty()){
7. if(p != NULL){
8. *//存入栈中*
9. stack.push(p);
10. *//遍历左子树*
11. p = p->lchild;
12. }
13. else{
14. *//退栈，访问根节点*
15. p = stack.top();
16. printf("%c ",p->data);
17. stack.pop();
18. *//访问右子树*
19. p = p->rchild;
20. }
21. }*//while*
22. }

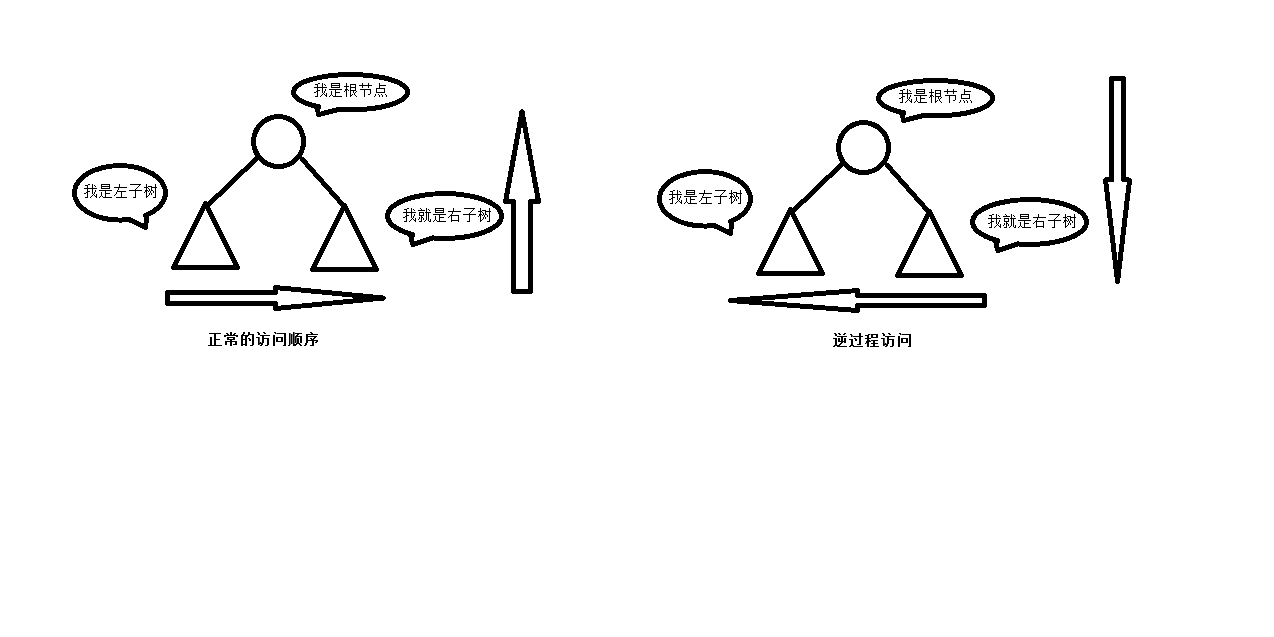
### 3.4.3后序遍历

【思路】：T是要遍历树的根指针，后序遍历要求在遍历完左右子树后，再访问根。需要判断根结点的左右子树是否均遍历过。

实现非递归方式的后序遍历，有两种方式。

**第一种:**

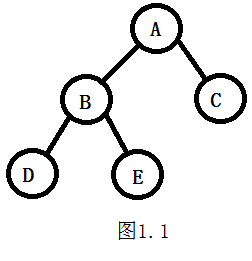
  使用两个栈来进行实现。注意到，后序遍历可以看做以下遍历过程的逆过程：先遍历某个节点，然后遍历其右孩子节点，再遍历其左孩子节点，该过程的逆过程，即为后序遍历的遍历过程。如图:

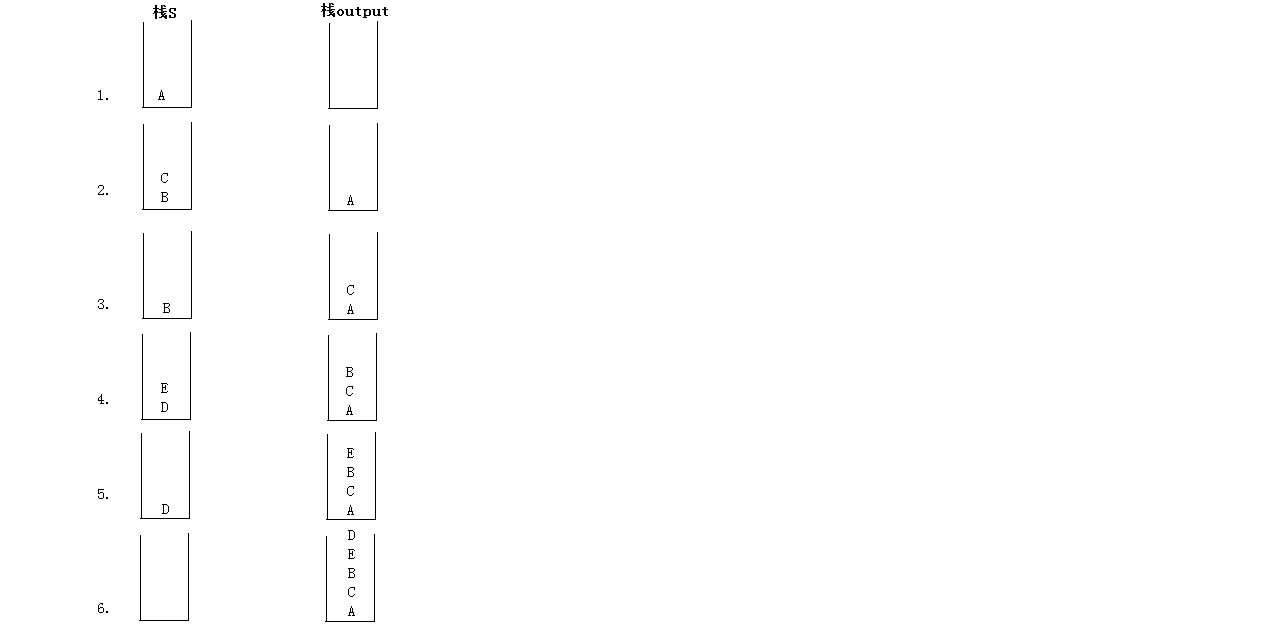


为此，我们可以按照如下的算法得到二叉树的后序遍历结果:

1. 初始化两个栈，一个用于保存中间遍历过程称为栈s，一个用于保存最终的结果称为栈output
2. push根节点到第一个栈s中
3. 从第一个栈s中pop出一节点，并将其push到第二个栈output中
4. 将第一个栈s中pop出的节点的孩子节点，按左孩子、右孩子的顺序push到第一个栈s中
5. 重复步骤3和4直到栈s为空
6. 栈s为空时，所有节点都已push到栈output中，且按后序遍历顺序存放，依次将栈output的节点pop出并进行访问，即为二叉树后序遍历的结果

以图1.1为例，其过程如下:





**第二种:**

 使用一个栈来进行实现，在搜索遍历的过程中，从二叉树的根节点出发，沿着该节点的左子树向下搜索，在搜索的过程中每遇到一个节点判断该节点是否是第一次经过，若是，则不立即访问，而是将该节点入栈保存，遍历该节点的左子树。当左子树遍历完毕后再返回该节点，这时还不能立即访问该节点，而是应当继续进入该节点的右子树进行遍历，当左右子树均遍历完毕后，才能从栈顶弹出该节点并访问它。由于在决定栈顶节点是否能访问时，需要知道该节点的右子树是否已经被遍历完毕。因此，为解决这个问题，在算法中还应当引入一个布尔型的访问标志变量flag和一个节点指针p。其中flag用来标志当前栈顶节点是否被访问过，当值为true的时候，表示栈顶节点已被访问过，当值为false的时候，表示当前栈顶节点未被访问过，指针p指向当前遍历过程中最后一个访问的节点。若当前栈顶节点的右孩子节点是空，或者就是p指向的节点，则表明当前节点的右子树已遍历完毕，此时就可以访问当前栈顶节点。其操作的实现过程描述如下:

1. 创建一个栈对象，根节点进栈，p赋初始化值为null
2. 若栈非空，则栈顶节点的非空左孩子相继进栈
3. 若栈非空，查看栈顶节点，若栈顶节点的右孩子为空，或者与p相等，则将栈顶节点弹出栈并访问它，同时使p指向该节点，并置flag为true，否则，将栈顶节点的右孩子压入栈，并置flag的值为false
4. 若flag值为true，则重复执行步骤3。否则，重复执行步骤2和3，直到栈为空为止。
5. *//后序遍历(非递归)*
6. typedef struct BiTNodePost{
7. BiTree biTree;
8. char tag;
9. }BiTNodePost,\*BiTreePost;
10. void PostOrder2(BiTree T){
11. stack<BiTreePost> stack;
12. *//p是遍历指针*
13. BiTree p = T;
14. BiTreePost BT;
15. *//栈不空或者p不空时循环*
16. while(p != NULL || !stack.empty()){
17. *//遍历左子树*
18. while(p != NULL){
19. BT = (BiTreePost)malloc(sizeof(BiTNodePost));
20. BT->biTree = p;
21. *//访问过左子树*
22. BT->tag = 'L';
23. stack.push(BT);
24. p = p->lchild;
25. }
26. *//左右子树访问完毕访问根节点*
27. while(!stack.empty() && (stack.top())->tag == 'R'){
28. BT = stack.top();
29. *//退栈*
30. stack.pop();
31. //BT->biTree;
32. printf("%c ",BT->biTree->data);
33. }
34. *//遍历右子树*
35. if(!stack.empty()){
36. BT = stack.top();
37. *//访问过右子树*
38. BT->tag = 'R';
39. p = BT->biTree;
40. p = p->rchild;
41. }
42. }*//while*
43. }

### 3.4.4层次遍历

【思路】：按从顶向下，从左至右的顺序来逐层访问每个节点，层次遍历的过程中需要用队列。

1. *//层次遍历*
2. void LevelOrder(BiTree T){
3. BiTree p = T;
4. *//队列*
5. queue<BiTree> queue;
6. *//根节点入队*
7. queue.push(p);
8. *//队列不空循环*
9. while(!queue.empty()){
10. *//对头元素出队*
11. p = queue.front();
12. *//访问p指向的结点*
13. printf("%c ",p->data);
14. *//退出队列*
15. queue.pop();
16. *//左子树不空，将左子树入队*
17. if(p->lchild != NULL){
18. queue.push(p->lchild);
19. }
20. *//右子树不空，将右子树入队*
21. if(p->rchild != NULL){
22. queue.push(p->rchild);
23. }
24. }
25. }

## 4.程序

1. *// BinaryTree.cpp : 定义控制台应用程序的入口点*
2. #include<iostream>
3. #include<stack>
4. #include<queue>
5. using namespace std;
7. *//二叉树结点*
8. typedef struct BiTNode{
9. *//数据*
10. char data;
11. *//左右孩子指针*
12. struct BiTNode \*lchild,\*rchild;
13. }BiTNode,\*BiTree;

16. *//按先序序列创建二叉树*
17. int CreateBiTree(BiTree &T){
18. char data;
19. *//按先序次序输入二叉树中结点的值（一个字符），‘#’表示空树*
20. scanf("%c",&data);
21. if(data == '#'){
22. T = NULL;
23. }
24. else{
25. T = (BiTree)malloc(sizeof(BiTNode));
26. *//生成根结点*
27. T->data = data;
28. *//构造左子树*
29. CreateBiTree(T->lchild);
30. *//构造右子树*
31. CreateBiTree(T->rchild);
32. }
33. return 0;
34. }
35. *//输出*
36. void Visit(BiTree T){
37. if(T->data != '#'){
38. printf("%c ",T->data);
39. }
40. }
41. *//先序遍历*
42. void PreOrder(BiTree T){
43. if(T != NULL){
44. *//访问根节点*
45. Visit(T);
46. *//访问左子结点*
47. PreOrder(T->lchild);
48. *//访问右子结点*
49. PreOrder(T->rchild);
50. }
51. }
52. *//中序遍历*
53. void InOrder(BiTree T){
54. if(T != NULL){
55. *//访问左子结点*
56. InOrder(T->lchild);
57. *//访问根节点*
58. Visit(T);
59. *//访问右子结点*
60. InOrder(T->rchild);
61. }
62. }
63. *//后序遍历*
64. void PostOrder(BiTree T){
65. if(T != NULL){
66. *//访问左子结点*
67. PostOrder(T->lchild);
68. *//访问右子结点*
69. PostOrder(T->rchild);
70. *//访问根节点*
71. Visit(T);
72. }
73. }
74. */\* 先序遍历(非递归)*
75. *思路：访问T->data后，将T入栈，遍历左子树；遍历完左子树返回时，栈顶元素应为T，出栈，再先序遍历T的右子树。*
76. *\*/*
77. void PreOrder2(BiTree T){
78. stack<BiTree> stack;
79. *//p是遍历指针*
80. BiTree p = T;
81. *//栈不空或者p不空时循环*
82. while(p || !stack.empty()){
83. if(p != NULL){
84. *//存入栈中*
85. stack.push(p);
86. *//访问根节点*
87. printf("%c ",p->data);
88. *//遍历左子树*
89. p = p->lchild;
90. }
91. else{
92. *//退栈*
93. p = stack.top();
94. stack.pop();
95. *//访问右子树*
96. p = p->rchild;
97. }
98. }*//while*
99. }
100. */\* 中序遍历(非递归)*
101. *思路：T是要遍历树的根指针，中序遍历要求在遍历完左子树后，访问根，再遍历右子树。*
102. *先将T入栈，遍历左子树；遍历完左子树返回时，栈顶元素应为T，出栈，访问T->data，再中序遍历T的右子树。*
103. *\*/*
104. void InOrder2(BiTree T){
105. stack<BiTree> stack;
106. *//p是遍历指针*
107. BiTree p = T;
108. *//栈不空或者p不空时循环*
109. while(p || !stack.empty()){
110. if(p != NULL){
111. *//存入栈中*
112. stack.push(p);
113. *//遍历左子树*
114. p = p->lchild;
115. }
116. else{
117. *//退栈，访问根节点*
118. p = stack.top();
119. printf("%c ",p->data);
120. stack.pop();
121. *//访问右子树*
122. p = p->rchild;
123. }
124. }*//while*
125. }

128. *//后序遍历(非递归)*
129. typedef struct BiTNodePost{
130. BiTree biTree;
131. char tag;
132. }BiTNodePost,\*BiTreePost;

135. void PostOrder2(BiTree T){
136. stack<BiTreePost> stack;
137. *//p是遍历指针*
138. BiTree p = T;
139. BiTreePost BT;
140. *//栈不空或者p不空时循环*
141. while(p != NULL || !stack.empty()){
142. *//遍历左子树*
143. while(p != NULL){
144. BT = (BiTreePost)malloc(sizeof(BiTNodePost));
145. BT->biTree = p;
146. *//访问过左子树*
147. BT->tag = 'L';
148. stack.push(BT);
149. p = p->lchild;
150. }
151. *//左右子树访问完毕访问根节点*
152. while(!stack.empty() && (stack.top())->tag == 'R'){
153. BT = stack.top();
154. *//退栈*
155. stack.pop();
156. BT->biTree;
157. printf("%c ",BT->biTree->data);
158. }
159. *//遍历右子树*
160. if(!stack.empty()){
161. BT = stack.top();
162. *//访问过右子树*
163. BT->tag = 'R';
164. p = BT->biTree;
165. p = p->rchild;
166. }
167. }*//while*
168. }
169. *//层次遍历*
170. void LevelOrder(BiTree T){
171. BiTree p = T;
172. *//队列*
173. queue<BiTree> queue;
174. *//根节点入队*
175. queue.push(p);
176. *//队列不空循环*
177. while(!queue.empty()){
178. *//对头元素出队*
179. p = queue.front();
180. *//访问p指向的结点*
181. printf("%c ",p->data);
182. *//退出队列*
183. queue.pop();
184. *//左子树不空，将左子树入队*
185. if(p->lchild != NULL){
186. queue.push(p->lchild);
187. }
188. *//右子树不空，将右子树入队*
189. if(p->rchild != NULL){
190. queue.push(p->rchild);
191. }
192. }
193. }
194. int main()
195. {
196. BiTree T;
197. CreateBiTree(T);
198. printf("先序遍历：\n");
199. PreOrder(T);
200. printf("\n");
201. printf("先序遍历(非递归)：\n");
202. PreOrder2(T);
203. printf("\n");
204. printf("中序遍历：\n");
205. InOrder(T);
206. printf("\n");
207. printf("中序遍历(非递归)：\n");
208. InOrder2(T);
209. printf("\n");
210. printf("后序遍历：\n");
211. PostOrder(T);
212. printf("\n");
213. printf("后序遍历(非递归)：\n");
214. PostOrder2(T);
215. printf("\n");
216. printf("层次遍历：\n");
217. LevelOrder(T);
218. printf("\n");
219. system("pause");
220. return 0;
221. }

以上实例在VS2008上编译通过。

## 本文链接：<https://blog.csdn.net/shufac/article/details/26979517>

参考：http://blog.csdn.net/sjf0115/article/details/8645991