**二叉树的三种遍历方式（递归和非递归的）**

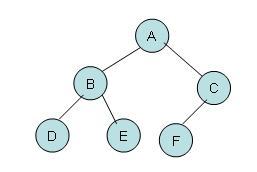
二叉树有前、中、后三种遍历方式，因为树的本身就是用递归定义的，因此采用递归的方法实现三种遍历，不仅代码简洁且容易理解，但其开销也比较大，而若采用非递归方法实现三种遍历，则要用栈来模拟实现（递归也是用栈实现的）

一：三种遍历方式的递归实现

1. 先序遍历——按照“根节点-左孩子-右孩子”的顺序进行访问。
2. **void** pre\_traverse(BTree pTree)
3. {
4. **if**(pTree)
5. {
6. printf("%c ",pTree->data);
7. **if**(pTree->pLchild)
8. pre\_traverse(pTree->pLchild);
9. **if**(pTree->pRchild)
10. pre\_traverse(pTree->pRchild);
11. }
12. }
13. 中序遍历——按照“左孩子-根节点-右孩子”的顺序进行访问。
14. **void** in\_traverse(BTree pTree)
15. {
16. **if**(pTree)
17. {
18. **if**(pTree->pLchild)
19. in\_traverse(pTree->pLchild);
20. printf("%c ",pTree->data);
21. **if**(pTree->pRchild)
22. in\_traverse(pTree->pRchild);
23. }
24. }
25. 后序遍历——按照“左孩子-右孩子-根节点”的顺序进行访问。
26. **void** beh\_traverse(BTree pTree)
27. {
28. **if**(pTree)
29. {
30. **if**(pTree->pLchild)
31. beh\_traverse(pTree->pLchild);
32. **if**(pTree->pRchild)
33. beh\_traverse(pTree->pRchild);
34. printf("%c ",pTree->data);
35. }

二、三种遍历方式的非递归实现

    为了便于理解，这里以下图的二叉树为例，分析二叉树的三种遍历方式的实现过程。



1. 前序遍历的非递归实现

根据前序遍历的顺序，优先访问根结点，然后在访问左子树和右子树。所以，对于任意结点node，第一部分即直接访问之，之后在判断左子树是否为空，不为空时即重复上面的步骤，直到其为空。若为空，则需要访问右子树。注意，在访问过左孩子之后，需要反过来访问其右孩子，所以，需要栈这种数据结构的支持。对于任意一个结点node，具体步骤如下：

a)访问之，并把结点node入栈，当前结点置为左孩子；

b)判断结点node是否为空，若为空，则取出栈顶结点并出栈，将右孩子置为当前结点；否则重复a)步直到当前结点为空或者栈为空（可以发现栈中的结点就是为了访问右孩子才存储的）

1. **public** **void** preOrderTraverse2(TreeNode root) {
2. LinkedList<TreeNode> stack = **new** LinkedList<>();
3. TreeNode pNode = root;
4. **while** (pNode != **null** || !stack.isEmpty()) {
5. **if** (pNode != **null**) {
6. System.out.print(pNode.val+"  ");
7. stack.push(pNode);
8. pNode = pNode.left;
9. } **else** { //pNode == null && !stack.isEmpty()
10. TreeNode node = stack.pop();
11. pNode = node.right;
12. }
13. }
14. }

3、Morris方法

前序遍历和中序遍历的Morris方法基本一样，不同之处在访问的顺序。代码如下：

**[cpp]** [view plain](http://blog.csdn.net/u012877472/article/details/49431103" \o "view plain) [copy](http://blog.csdn.net/u012877472/article/details/49431103" \o "copy)

1. //Morris
2. **class** Solution3 {
3. **public**:
4. vector<**int**> preorderTraversal(TreeNode\* root) {
5. vector<**int**> ret;
6. **if**(root==NULL) **return** ret;
7. TreeNode \*curr=root;
8. TreeNode \*pre;
9. **while**(curr)
10. {
11. **if**(curr->left==NULL)
12. {
13. ret.push\_back(curr->val);
14. curr=curr->right;
15. }
16. **else**
17. {
18. pre=curr->left;
19. **while**(pre->right&&pre->right!=curr)
20. pre=pre->right;
21. **if**(pre->right==NULL)
22. {
23. ret.push\_back(curr->val);
24. pre->right=curr;
25. curr=curr->left;
26. }
27. **else**
28. {
29. pre->right=NULL;
30. curr=curr->right;
31. }
32. }
33. }
34. **return** ret;
35. }
36. };

2、中序遍历的非递归实现

有了上面前序的解释，中序也就比较简单了，相同的道理。只不过访问的顺序移到出栈时。代码如下：**void** in\_traverse(BTree pTree)

1. **public** **void** inOrderTraverse2(TreeNode root) {
2. LinkedList<TreeNode> stack = **new** LinkedList<>();
3. TreeNode pNode = root;
4. **while** (pNode != **null** || !stack.isEmpty()) {
5. **if** (pNode != **null**) {
6. stack.push(pNode);
7. pNode = pNode.left;
8. } **else** { //pNode == null && !stack.isEmpty()
9. TreeNode node = stack.pop();
10. System.out.print(node.val+"  ");
11. pNode = node.right;
12. }
13. }
14. }

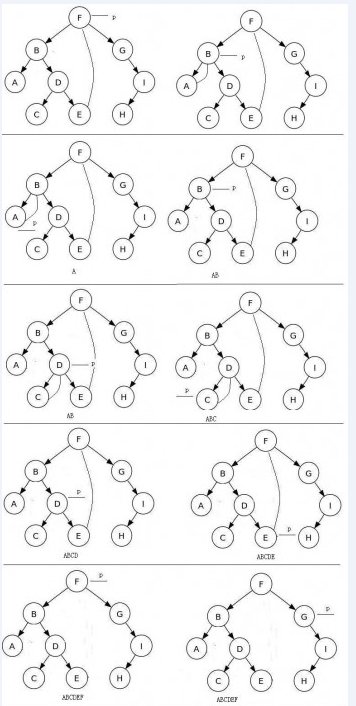
3、Morris法

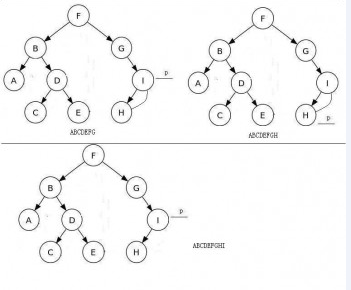
|  |
| --- |
| 大家都很熟悉用递归和堆栈来实现二叉树的遍历，比如，前序遍历，中序遍历，后序遍历。但Morris 遍历，使用无堆栈，O(1) 空间进行二叉树遍历。它的原理很简单，利用所有叶子结点的右指针，指向其后继结点，组成一个环，在第二次遍历到这个结点时，由于其左子树已经遍历完了，则访问该结点。  ****算法伪码：****  MorrisInOrder()：  while 没有结束     如果当前节点没有左后代       访问该节点       转向右节点     否则       找到左后代的最右节点，且使最右节点的右指针指向当前节点       转向左后代节点 |

这就是Morris方法，时间复杂度为O(n)，空间复杂度是O(1)。代码如下：

**[cpp]** [view plain](http://blog.csdn.net/u012877472/article/details/49401751" \o "view plain) [copy](http://blog.csdn.net/u012877472/article/details/49401751" \o "copy)

1. //Morris traversal,without a stack
2. **class** Solution3 {
3. **public**:
4. vector<**int**> inorderTraversal(TreeNode\* root) {
5. vector<**int**> ret;
6. **if**(root==NULL)**return** ret;
7. TreeNode \*curr=root;
8. TreeNode \*pre;
9. **while**(curr)
10. {
11. **if**(curr->left==NULL)
12. {
13. ret.push\_back(curr->val);
14. curr=curr->right;
15. }
16. **else**
17. {
18. pre=curr->left;
19. **while**(pre->right&&pre->right!=curr)  //找到左后代的最右节点
20. pre=pre->right;
21. **if**(pre->right==NULL)  //pre->right为假时
22. {
23. pre->right=curr;  //最右节点的右指针指向当前节点
24. curr=curr->left;  //转向左后代节点
25. }
26. **else**  //pre->right=curr时
27. {
28. ret.push\_back(curr->val);
29. pre->right=NULL;
30. curr=curr->right;
31. }
32. }
33. }
34. **return** ret;
35. }
36. };





**3、后序遍历的非递归实现**

**根据后序遍历的顺序，先访问左子树，再访问右子树，后访问根节点，而对于每个子树来说，又按照同样的访问顺序进行遍历，上图的后序遍历顺序为：DEBFCA。后序遍历的非递归的实现相对来说要难一些，要保证根节点在左子树和右子树被访问后才能访问，思路如下：**

对于任一节点P，

1）先将节点P入栈；

2）若P不存在左孩子和右孩子，或者P存在左孩子或右孩子，但左右孩子已经被输出，则可以直接输出节点P，并将其出栈，将出栈节点P标记为上一个输出的节点，再将此时的栈顶结点设为当前节点；

3）若不满足2）中的条件，则将P的右孩子和左孩子依次入栈，当前节点重新置为栈顶结点，之后重复操作2）；

4）直到栈空，遍历结束。

**下面以上图为例详细分析其后序遍历的非递归实现过程：**

首先，设置两个指针：Cur指针指向当前访问的节点，它一直指向栈顶节点，每次出栈一个节点后，将其重新置为栈顶结点，Pre节点指向上一个访问的节点；

Cur首先指向根节点A，Pre先设为NULL，由于A存在左孩子和右孩子，根据操作3），先将右孩子C入栈，再将左孩子B入栈，Cur改为指向栈顶结点B；

由于B的也有左孩子和右孩子，根据操作3），将E、D依次入栈，Cur改为指向栈顶结点D；

由于D没有左孩子，也没有右孩子，根据操作2），直接输出D，并将其出栈，将Pre指向D，Cur指向栈顶结点E，此时输出序列为：D；

由于E也没有左右孩子，根据操作2），输出E，并将其出栈，将Pre指向E，Cur指向栈顶结点B，此时输出序列为：DE；

由于B的左右孩子已经被输出，即满足条件Pre==Cur->lchild或Pre==Cur->rchild，根据操作2），输出B，并将其出栈，将Pre指向B，Cur指向栈顶结点C，此时输出序列为：DEB；

由于C有左孩子，根据操作3），将其入栈，Cur指向栈顶节点F；

由于F没有左右孩子，根据操作2），输出F，并将其出栈，将Pre指向F，Cur指向栈顶结点C，此时输出序列为：DEBF；

由于C的左孩子已经被输出，即满足Pre==Cur->lchild，根据操作2），输出C，并将其出栈，将Pre指向C，Cur指向栈顶结点A，此时输出序列为：DEBFC；

由于A的左右孩子已经被输出，根据操作2），输出A，并将其出栈，此时输出序列为：DEBFCA；

此时栈空，遍历结束。

1. **void** beh\_traverse(BTree pTree)
2. {
3. PSTACK stack = create\_stack();  //创建一个空栈
4. BTree node\_pop;          //用来保存出栈的节点
5. BTree pCur;              //定义指针，指向当前节点
6. BTree pPre = NULL;       //定义指针，指向上一个访问的节点
8. //先将树的根节点入栈
9. push\_stack(stack,pTree);
10. //直到栈空时，结束循环
11. **while**(!is\_empty(stack))
12. {
13. pCur = getTop(stack);   //当前节点置为栈顶节点
14. **if**((pCur->pLchild==NULL && pCur->pRchild==NULL) ||
15. (pPre!=NULL && (pCur->pLchild==pPre || pCur->pRchild==pPre)))
16. {
17. //如果当前节点没有左右孩子，或者有左孩子或有孩子，但已经被访问输出，
18. //则直接输出该节点，将其出栈，将其设为上一个访问的节点
19. printf("%c ", pCur->data);
20. pop\_stack(stack,&node\_pop);
21. pPre = pCur;
22. }
23. **else**
24. {
25. //如果不满足上面两种情况,则将其右孩子左孩子依次入栈
26. **if**(pCur->pRchild != NULL)
27. push\_stack(stack,pCur->pRchild);
28. **if**(pCur->pLchild != NULL)
29. push\_stack(stack,pCur->pLchild);
30. }
31. }
32. }

后序遍历的Morris方法

首先建立一个临时节点dump，令其左儿子是root。并且还需要一个子过程，就是倒序输出某两个节点之间路径上的各个节点。  
步骤：  
当前节点设置为临时节点dump。  
（1）如果当前节点的左儿子为空，则将其右儿子作为当前节点；  
（2）如果当前节点的左儿子非空，在当前节点的左子树中找到当前节点在中序遍历下的前驱节点；  
   a) 如果前驱节点的右孩子为空，将它的右儿子设置为当前节点。当前节点更新为当前节点的左儿子；  
   b) 如果前驱节点的右儿子为当前节点，将它的右孩子重新设为空。倒序输出从当前节点的左儿子到该前驱节点这条路径上的所有节点。当前节点更新为当前节点的右儿子；  
（3）重复以上（1）（2）直到当前节点为空。

1. **class** Solution4 {
2. **public**:
3. vector<**int**> postorderTraversal(TreeNode\* root) {
4. vector<**int**> ret;
5. TreeNode \*dump=**new** TreeNode(0);
6. dump->left=root;
7. TreeNode \*curr=dump;
8. TreeNode \*pre;
9. **while**(curr)
10. {
11. **if**(curr->left==NULL)
12. {
13. curr=curr->right;
14. }
15. **else**
16. {
17. pre=curr->left;
18. **while**(pre->right&&pre->right!=curr)
19. pre=pre->right;
20. **if**(pre->right==NULL)
21. {
22. pre->right=curr;
23. curr=curr->left;
24. }
25. **else**
26. {
27. reverseAddNodes(curr->left,pre,ret);
28. pre->right=NULL;
29. curr=curr->right;
30. }
31. }
32. }
33. **return** ret;
34. }
35. **private**:
36. **void** reverseAddNodes(TreeNode \*begin,TreeNode \*end,vector<**int**>& ret)
37. {
38. reverseNodes(begin,end);
39. TreeNode \*curr=end;
40. **while**(**true**)
41. {
42. ret.push\_back(curr->val);
43. **if**(curr==begin)**break**;
44. curr=curr->right;
45. }
46. reverseNodes(end,begin);
47. }
48. **void** reverseNodes(TreeNode \*begin,TreeNode \*end)
49. {
50. TreeNode \*pre=begin;
51. TreeNode \*curr=pre->right;
52. TreeNode \*post;
53. **while**(pre!=end)
54. {
55. post=curr->right;
56. curr->right=pre;
57. pre=curr;
58. curr=post;
59. }
60. }
61. };

四、层次遍历

层次遍历的代码比较简单，只需要一个队列即可，先在队列中加入根结点。之后对于任意一个结点来说，在其出队列的时候，访问之。同时如果左孩子和右孩子有不为空的，入队列。代码如下：

1. **public** **void** levelTraverse(TreeNode root) {
2. **if** (root == **null**) {
3. **return**;
4. }
5. LinkedList<TreeNode> queue = **new** LinkedList<>();
6. queue.offer(root);
7. **while** (!queue.isEmpty()) {
8. TreeNode node = queue.poll();
9. System.out.print(node.val+"  ");
10. **if** (node.left != **null**) {
11. queue.offer(node.left);
12. }
13. **if** (node.right != **null**) {
14. queue.offer(node.right);
15. }
16. }
17. }

深度优先遍历

其实深度遍历就是上面的前序、中序和后序。但是为了保证与广度优先遍历相照应，也写在这。代码也比较好理解，其实就是前序遍历，代码如下

1. **public** **void** depthOrderTraverse(TreeNode root) {
2. **if** (root == **null**) {
3. **return**;
4. }
5. LinkedList<TreeNode> stack = **new** LinkedList<>();
6. stack.push(root);
7. **while** (!stack.isEmpty()) {
8. TreeNode node = stack.pop();
9. System.out.print(node.val+"  ");
10. **if** (node.right != **null**) {
11. stack.push(node.right);
12. }
13. **if** (node.left != **null**) {
14. stack.push(node.left);
15. }
16. }
17. }