

个人资料



苏叔叔

发私信



访问：125200次
积分：2639
等级：
排名：第10260名

原创：107篇 转载：0篇
译文：0篇 评论：106条

等你连接

我的新浪微博
<http://weibo.com/zhangxiagDavid>
我的邮箱
zhangxiangdavid@126.com

为何没力气去捉紧这一点火花

天高海深

有什么可拥有

文章搜索

我的新浪微博

二叉树前序、中序、后序遍历非递归写法的透彻解析

标签： 二叉树 遍历 非递归 算法 栈

2014-07-06 22:14 1972人阅读 评论

分类： 数据结构与算法 (49)

版权声明：本文为博主原创文章，转载，请注明出处。若是商业用途，请事先联系作者。

[目录\(?\)](#) [\[+\]](#)

前言

在前两篇文章[二叉树](#)和[二叉搜索树](#)中已经涉及到了二叉树的三种遍历。递归写法，只要理解思想，递归写法却很不容易。这里特地总结下，透彻解析它们的非递归写法。其中，中序遍历的非递归写法最困难。我们的讨论基础是这样的：

```
[cpp]
01. //Binary Tree Node
02. typedef struct node
03. {
04.     int data;
05.     struct node* lchild; //左孩子
06.     struct node* rchild; //右孩子
07. }BTNode;
```

首先，有一点是明确的：非递归写法一定会用到栈，这个应该不用太多的解释。我们先看中序遍历：中序遍历

分析

中序遍历的递归定义：先左子树，后根节点，再右子树。如何写非递归代码呢？一句话：让代码跟递归思维是什么？思维就是中序遍历的路径。假设，你面前有一棵二叉树，现要求你写出它的中序遍历序列。你理解透彻的话，你肯定先找到左子树的最下边的节点。那么下面的代码就是理所当然的：

中序代码段(i)

```
[cpp]
01. BTreeNode* p = root; //p指向树根
02. stack<BTreeNode*> s; //STL中的栈
03. //一直遍历到左子树最下边，边遍历边保存根节点到栈中
04. while (p)
05. {
06.     s.push(p);
07.     p = p->lchild;
08. }
```

保存一路走过的根节点的理由是：中序遍历的需要，遍历完左子树后，需要借助根节点进入右子树。当p为空，此时无非两种情况：

微博

Mr_苏叔叔

广东 深圳

加关注

鹅狼难瘦

10月16日 17:14

转发 | 评论

TA 的粉丝 (61)

全部»

蔡蔡蔡蔡

水惟怡

勇敢的夏

谁的季子

博客专栏

C++拾遗

文章：23篇

阅读：20976

C指针

文章：9篇

阅读：13629

经典算法揭秘

文章：24篇

阅读：28708

数据结构与算法

文章：49篇

阅读：61710

文章分类

数据结构与算法 (50)

娓娓道来c指针 (9)

挑战面试编程 (14)

C++拾遗 (23)

设计模式 (1)

linux系统编程 (10)

LeetCode (1)

阅读排行

数据结构与算法目录 (3789)

LeetCode-Linked List (2713)

娓娓道来c指针 (0)c语言的梦... (2424)

数据结构：最小生成树--Prim... (2209)

数据结构：图的遍历--深度优... (2033)

二叉树前序、中序、后序遍历... (1965)

图论基础 (1913)

挑战面试编程：大整数的加、... (1880)

哈夫曼树 (1875)

图 a. 最近保存的根节点是叶子节点 图 b. 最近保存的根节点无左孩子

说明：

1. 上图中只给出了必要的节点和边，其它的边和节点与讨论无关，不必画出。
2. 你可能认为图a中最近保存节点算不得是根节点。如果你看过[树、二叉树基础](#)，使用扩充二叉树解释。总之，不用纠结这个没有意义问题。
3. 整个二叉树只有一个根节点的情况可以划到图a。

仔细想想，二叉树的左子树，最下边是不是上图两种情况？不管怎样，此时都要出栈，并访问该节点。根据我们的思维，代码应该是这样：

```
[cpp] C P
01. p = s.top();
02. s.pop();
03. cout << p->data;
```

我们的思维接着走，两图情形不同得区别对待：

1. 图a中访问的是一个左孩子，按中序遍历顺序，接下来应访问它的根节点。也就是图a中的另一个节点被保存在栈中。我们只需这样的代码和上一步一样的代码：

```
[cpp] C P
01. p = s.top();
02. s.pop();
03. cout << p->data;
```

左孩子和根都访问完了，接着就是右孩子了，对吧。接下来只需一句代码：p=p->rchild;在右子树中继续遍历。代码段(i)、代码段(ii)……直到栈空且p空。

2. 再看图b，由于没有左孩子，根节点就是中序序列中第一个，然后直接进入右子树：p=p->rchild。新一轮的代码段(i)、代码段(ii)……直到栈空且p空。

思维到这里，似乎很不清晰，真的要区分吗？根据图a接下来的代码段(ii)这样的：

```
[cpp] C P
01. p = s.top();
02. s.pop();
03. cout << p->data;
04. p = s.top();
05. s.pop();
06. cout << p->data;
07. p = p->rchild;
```

根据图b，代码段(ii)又是这样的：

```
[cpp] C P
01. p = s.top();
02. s.pop();
03. cout << p->data;
04. p = p->rchild;
```

评论排行

二分查找的改进--差值查找	(8)
娓娓道来c指针 (1)指针就是地址	(7)
哈夫曼树	(7)
C++拾遗--name_cast 显式类...	(7)
数据结构：图的遍历--深度优...	(6)
图论基础	(6)
链表常见操作：逆置(反转)	(6)
约瑟夫问题的数组解法	(4)
挑战面试编程：字符串替换	(4)
挑战面试编程：计算整数二进...	(4)

最新评论

哈夫曼树 Ouyang_Lianjun：苏前辈，你真会开玩笑
C++拾遗--new delete 重载 苏叔叔：@xufeng0991:多谢指正，已修改。
C++拾遗--new delete 重载 J寒...风：void *mem = malloc(size); if (mem) //内存分配...
链表常见操作：逆置(反转) hMotion：明天考c#，就是这个链表的逆置不会，还好看到了你的分享。真的很感谢。
C++拾遗--lambda表达式原理 mymodian：对比理解上来看，基本没问题，我在VS2013上测试，通过引用捕获外部对象的实现方式不是采用引用，而是...
C++拾遗--lambda表达式原理 sdghcj：那请问一下，lambda被编译成一个functor对象，为什么可以当作函数指针传递给其它有相同签名回...
哈夫曼树 楊羲Sunshine：c和c++很像，但后者要难一点。
娓娓道来c指针 (1)指针就是地址 qq_33800727：不错很好。。。
约瑟夫问题的数组解法 苏叔叔：@qq_33836426:欢迎交流^^^
约瑟夫问题的数组解法 qq_33836426：大神！！！！

文章存档

2016年01月	(1)
2015年12月	(1)
2015年07月	(9)
2015年06月	(3)
2015年03月	(7)
展开	

我们可小结下：遍历过程是个循环，并且按代码段(i)、代码段(ii)构成一次循环体，循环直到栈空不同的处理方法很让人抓狂，可统一处理吗？真的是可以的！回顾扩充二叉树，是不是每个节点都可呢？那么，代码只需统一写成图b的这种形式。也就是说代码段(ii)统一是这样的：

中序代码段(ii)

```
[cpp]
01. p = s.top();
02. s.pop();
03. cout << p->data;
04. p = p->rchild;
```

口说无凭，得经的过理论检验。

图a的代码段(ii)也可写成图b的理由是：由于是叶子节点，p==p->rchild;之后p肯定为空。为空，代码段(i)吗？显然不需。(因为不满足循环条件)那就直接进入代码段(ii)。看！最后还是一样的吧。看到这里，要仔细想想哦！相信你一定会明白的。

这时写出遍历循环体就不难了：

```
[cpp]
01. BTreeNode* p = root;
02. stack<BTreeNode*> s;
03. while (!s.empty() || p)
04. {
05.     //代码段(i)一直遍历到左子树最下边，边遍历边保存根节点到栈中
06.     while (p)
07.     {
08.         s.push(p);
09.         p = p->lchild;
10.     }
11.     //代码段(ii)当p为空时，说明已经到达左子树最下边，这时需要出栈了
12.     if (!s.empty())
13.     {
14.         p = s.top();
15.         s.pop();
16.         cout << setw(4) << p->data;
17.         //进入右子树，开始新一轮左子树遍历(这是递归的自我实现)
18.         p = p->rchild;
19.     }
20. }
```

仔细想想，上述代码是不是根据我们的思维走向而写出来的呢？再加上边界条件的检测，中序遍历非递归代码是这样的：

中序遍历代码一

```
[cpp]
01. //中序遍历
02. void InOrderWithoutRecursion1(BTreeNode* root)
03. {
04.     //空树
05.     if (root == NULL)
06.         return;
07.     //树非空
08.     BTreeNode* p = root;
09.     stack<BTreeNode*> s;
10.     while (!s.empty() || p)
11.     {
12.         //一直遍历到左子树最下边，边遍历边保存根节点到栈中
13.         while (p)
14.         {
15.             s.push(p);
16.             p = p->lchild;
17.         }
18.         //当p为空时，说明已经到达左子树最下边，这时需要出栈了
19.         if (!s.empty())
20.         {
21.             p = s.top();
```

```

22.         s.pop();
23.         cout << setw(4) << p->data;
24.         //进入右子树，开始新一轮左子树遍历(这是递归的自我实现)
25.         p = p->rchild;
26.     }
27. }
28. }

```

恭喜你，你已经完成了中序遍历非递归形式的代码了。回顾一下难吗？

接下来的这份代码，本质上是一样的，相信不用我解释，你也能看懂的。

中序遍历代码二

```

[cpp]
01. //中序遍历
02. void InOrderWithoutRecursion2(BTNode* root)
03. {
04.     //空树
05.     if (root == NULL)
06.         return;
07.     //树非空
08.     BTNode* p = root;
09.     stack<BTNode*> s;
10.     while (!s.empty() || p)
11.     {
12.         if (p)
13.         {
14.             s.push(p);
15.             p = p->lchild;
16.         }
17.         else
18.         {
19.             p = s.top();
20.             s.pop();
21.             cout << setw(4) << p->data;
22.             p = p->rchild;
23.         }
24.     }
25. }

```

前序遍历

分析

前序遍历的递归定义：先根节点，后左子树，再右子树。有了中序遍历的基础，不用我再像中序遍历首先，我们遍历左子树，边遍历边打印，并把根节点存入栈中，以后需借助这些节点进入右子树开屏得重复一句：所有的节点都可看做是根节点。根据思维走向，写出代码段(i)：

前序代码段(i)

```

[cpp]
01. //边遍历边打印，并存入栈中，以后需要借助这些根节点(不要怀疑这种说法哦)进入右子树
02. while (p)
03. {
04.     cout << setw(4) << p->data;
05.     s.push(p);
06.     p = p->lchild;
07. }

```

接下来就是：出栈，根据栈顶节点进入右子树。

前序代码段(ii)

```

[cpp]
01. //当p为空时，说明根和左子树都遍历完了，该进入右子树了
02. if (!s.empty())
03. {
04.     p = s.top();
05.     s.pop();
06.     p = p->rchild;
07. }

```

同样地，代码段(i) (ii) 构成了一次完整的循环体。至此，不难写出完整的前序遍历的非递归写法。

前序遍历代码一

```
[cpp]
01. void PreOrderWithoutRecursion1(BTNode* root)
02. {
03.     if (root == NULL)
04.         return;
05.     BTNode* p = root;
06.     stack<BTNode*> s;
07.     while (!s.empty() || p)
08.     {
09.         //边遍历边打印，并存入栈中，以后需要借助这些根节点(不要怀疑这种说法哦)进入右子树
10.         while (p)
11.         {
12.             cout << setw(4) << p->data;
13.             s.push(p);
14.             p = p->lchild;
15.         }
16.         //当p为空时，说明根和左子树都遍历完了，该进入右子树了
17.         if (!s.empty())
18.         {
19.             p = s.top();
20.             s.pop();
21.             p = p->rchild;
22.         }
23.     }
24.     cout << endl;
25. }
```

下面给出，本质是一样的另一段代码：

前序遍历代码二

```
[cpp]
01. //前序遍历
02. void PreOrderWithoutRecursion2(BTNode* root)
03. {
04.     if (root == NULL)
05.         return;
06.     BTNode* p = root;
07.     stack<BTNode*> s;
08.     while (!s.empty() || p)
09.     {
10.         if (p)
11.         {
12.             cout << setw(4) << p->data;
13.             s.push(p);
14.             p = p->lchild;
15.         }
16.         else
17.         {
18.             p = s.top();
19.             s.pop();
20.             p = p->rchild;
21.         }
22.     }
23.     cout << endl;
24. }
```

在二叉树中使用的是这样的写法，略有差别，本质上也是一样的：

前序遍历代码三

```
[cpp]
01. void PreOrderWithoutRecursion3(BTNode* root)
02. {
03.     if (root == NULL)
04.         return;
05.     stack<BTNode*> s;
06.     BTNode* p = root;
07.     s.push(root);
08.     while (!s.empty()) //循环结束条件与前两种不一样
09.     {
10.         //这句表明p在循环中总是非空的
```

```

11.         cout << setw(4) << p->data;
12.         /*
13.         栈的特点: 先进后出
14.         先被访问的根节点的右子树后被访问
15.         */
16.         if (p->rchild)
17.             s.push(p->rchild);
18.         if (p->lchild)
19.             p = p->lchild;
20.         else
21.             { //左子树访问完了, 访问右子树
22.                 p = s.top();
23.                 s.pop();
24.             }
25.     }
26.     cout << endl;
27. }

```

最后进入最难的后序遍历:

后序遍历

分析

后序遍历递归定义: 先左子树, 后右子树, 再根节点。后序遍历的难点在于: 需要判断上次访问的节点是左子树还是右子树。若是位于左子树, 则需跳过根节点, 先进入右子树, 再回头访问根节点, 若是位于右子树, 则直接访问根节点。直接看代码, 代码中有详细的注释。

后序遍历代码一

```

[cpp]
01. //后序遍历
02. void PostOrderWithoutRecursion(BTNode* root)
03. {
04.     if (root == NULL)
05.         return;
06.     stack<BTNode*> s;
07.     //pCur:当前访问节点, pLastVisit:上次访问节点
08.     BTNode* pCur, *pLastVisit;
09.     //pCur = root;
10.     pCur = root;
11.     pLastVisit = NULL;
12.     //先把pCur移动到左子树最下边
13.     while (pCur)
14.     {
15.         s.push(pCur);
16.         pCur = pCur->lchild;
17.     }
18.     while (!s.empty())
19.     {
20.         //走到这里, pCur都是空, 并已经遍历到左子树底端(看成扩充二叉树, 则空, 亦是某棵树的左孩子)
21.         pCur = s.top();
22.         s.pop();
23.         //一个根节点被访问的前提是: 无右子树或右子树已被访问过
24.         if (pCur->rchild == NULL || pCur->rchild == pLastVisit)
25.         {
26.             cout << setw(4) << pCur->data;
27.             //修改最近被访问的节点
28.             pLastVisit = pCur;
29.         }
30.         /*这里的else语句可换成带条件的else if:
31.         else if (pCur->lchild == pLastVisit) //若左子树刚被访问过, 则需先进入右子树(根节点需再次入栈)
32.         因为: 上面的条件没通过就一定是下面的条件满足。仔细想想!
33.         */
34.         else
35.         {
36.             //根节点再次入栈
37.             s.push(pCur);
38.             //进入右子树, 且可肯定右子树一定不为空
39.             pCur = pCur->rchild;
40.             while (pCur)
41.             {
42.                 s.push(pCur);
43.                 pCur = pCur->lchild;
44.             }
45.         }
46.     }
47.     cout << endl;
48. }

```

下面给出另一种思路下的代码。它的想法是：给每个节点附加一个标记(left, right)。如果该节点首次被访问，则置标记为left；若右子树被访问过，则置标记为right。显然，只有当节点的标记位是right时，才需要访问其右子树。否则，必须先访问其左子树。详细细节看代码中的注释。

后序遍历代码二

```
[cpp]
01. //定义枚举类型: Tag
02. enum Tag{left,right};
03. //自定义新的类型, 把二叉树节点和标记封装在一起
04. typedef struct
05. {
06.     BTreeNode* node;
07.     Tag tag;
08. }TagNode;
09. //后序遍历
10. void PostOrderWithoutRecursion2(BTreeNode* root)
11. {
12.     if (root == NULL)
13.         return;
14.     stack<TagNode> s;
15.     TagNode tagnode;
16.     BTreeNode* p = root;
17.     while (!s.empty() || p)
18.     {
19.         while (p)
20.         {
21.             tagnode.node = p;
22.             //该节点的左子树被访问过
23.             tagnode.tag = Tag::left;
24.             s.push(tagnode);
25.             p = p->lchild;
26.         }
27.         tagnode = s.top();
28.         s.pop();
29.         //左子树被访问过, 则还需进入右子树
30.         if (tagnode.tag == Tag::left)
31.         {
32.             //置换标记
33.             tagnode.tag = Tag::right;
34.             //再次入栈
35.             s.push(tagnode);
36.             p = tagnode.node;
37.             //进入右子树
38.             p = p->rchild;
39.         }
40.         else//右子树已被访问过, 则可访问当前节点
41.         {
42.             cout << setw(4) << (tagnode.node->data);
43.             //置空, 再次出栈(这一步是理解的难点)
44.             p = NULL;
45.         }
46.     }
47.     cout << endl;
48. }
```

总结

思维和代码之间总是有巨大的鸿沟。通常是思维正确，清楚，但却不易写出正确的代码。要想越过鸿沟，多借鉴，别无它法。

以下几点是理解上述代码的关键：

1. 所有的节点都可看做是父节点(叶子节点可看做是两个孩子为空的父节点)。
2. 把同一算法的代码对比着看。在差异中往往可看到算法的本质。
3. 根据自己的理解，尝试修改代码。写出自己理解下的代码。写成了，那就是真的掌握了。

转载请注明出处，本文地址：<http://blog.csdn.net/zhangxiangdavid/article/details/3711535>

专栏目录：

- [数据结构与算法目录](#)