更简单的非递归遍历二叉树的方法

2014-04-13 13:18 7961

解决二叉树的很多问题的方案都是基于对二叉树的遍历。遍历二叉树的前序，中序，后序三大方法算是计算机科班学生必写代码了。其递归遍历是人人都能信手拈来，可是在手生时写出非递归遍历恐非易事。正因为并非易事，所以网上出现无数的介绍二叉树非递归遍历方法的文章。可是大家需要的真是那些非递归遍历代码和讲述吗？代码早在学数据结构时就看懂了，理解了，可为什么我们一而再再而三地忘记非递归遍历方法，却始终记住了递归遍历方法？

三种递归遍历对遍历的描述，思路非常简洁，最重要的是三种方法完全统一，大大减轻了我们理解的负担。而我们常接触到那三种非递归遍历方法，除了都使用栈，具体实现各有差异，导致了理解的模糊。本文给出了一种统一的三大非递归遍历的实现思想。

三种递归遍历

```
//前序遍历
void preorder(TreeNode *root, vector<int> &path)
{
    if(root != NULL)
    {
```

```
        path.push_back(root->val);
        preorder(root->left, path);
        preorder(root->right, path);
    }
}

//中序遍历
void inorder(TreeNode *root, vector<int> &path)
{
    if(root != NULL)
    {
        inorder(root->left, path);
        path.push_back(root->val);
        inorder(root->right, path);
    }
}

//后续遍历
void postorder(TreeNode *root, vector<int> &path)
{
    if(root != NULL)
    {
        postorder(root->left, path);
        postorder(root->right, path);
        path.push_back(root->val);
    }
}
```

由上可见，递归的算法实现思路和代码风格非常统一，关于“递归”的理解可见我的《人脑理解递归》。

教科书上的非递归遍历

```
//非递归前序遍历
void preorderTraversal(TreeNode *root, vector<int> &path)
{
    stack<TreeNode *> s;
    TreeNode *p = root;
    while(p != NULL || !s.empty())
```

```
        {
            while(p != NULL)
            {
                path.push_back(p->val);
                s.push(p);
                p = p->left;
            }
            if(!s.empty())
            {
                p = s.top();
                s.pop();
                p = p->right;
            }
        }
    }

    //非递归中序遍历
    void inorderTraversal(TreeNode *root, vector<int> &path)
    {
        stack<TreeNode *> s;
        TreeNode *p = root;
        while(p != NULL || !s.empty())
        {
            while(p != NULL)
            {
                s.push(p);
                p = p->left;
            }
            if(!s.empty())
            {
                p = s.top();
                path.push_back(p->val);
                s.pop();
                p = p->right;
            }
        }
    }
```

```
//非递归后序遍历-迭代
void postorderTraversal(TreeNode *root, vector<int> &path)
{
    stack<TempNode *> s;
    TreeNode *p = root;
    TempNode *temp;
    while(p != NULL || !s.empty())
    {
        while(p != NULL) //沿左子树一直往下搜索，直至出现没有左子树的结点
        {
            TreeNode *tempNode = new TreeNode;
            tempNode->btnode = p;
            tempNode->isFirst = true;
            s.push(tempNode);
            p = p->left;
        }
        if(!s.empty())
        {
            temp = s.top();
            s.pop();
            if(temp->isFirst == true)   //表示是第一次出现在栈顶
            {
                temp->isFirst = false;
                s.push(temp);
                p = temp->btnode->right;
            }
            else  //第二次出现在栈顶
            {
                path.push_back(temp->btnode->val);
                p = NULL;
            }
        }
    }
}
```

看了上面教科书的三种非递归遍历方法，不难发现，后序遍历的实现的复杂程度明显高于前序遍历和中序遍历，前序遍历和中序遍历看似实现风格一样，但是实际上前者是在指针迭代时访问结点值，后者是在栈顶访问

结点值，实现思路也是有本质区别的。而这三种方法最大的缺点就是都使用嵌套循环，大大增加了理解的复杂度。

更简单的非递归遍历二叉树的方法

这里我给出统一的实现思路和代码风格的方法，完成对二叉树的三种非递归遍历。

```
//更简单的非递归前序遍历
void preorderTraversalNew(TreeNode *root, vector<int> &path)
{
    stack< pair<TreeNode *, bool> > s;
    s.push(make_pair(root, false));
    bool visited;
    while(!s.empty())
    {
        root = s.top().first;
        visited = s.top().second;
        s.pop();
        if(root == NULL)
            continue;
        if(visited)
        {
            path.push_back(root->val);
        }
        else
        {
            s.push(make_pair(root->right, false));
            s.push(make_pair(root->left, false));
            s.push(make_pair(root, true));
        }
    }
}

//更简单的非递归中序遍历
void inorderTraversalNew(TreeNode *root, vector<int> &path)
{
```

```
    stack< pair<TreeNode *, bool> > s;
    s.push(make_pair(root, false));
    bool visited;
    while(!s.empty())
    {
        root = s.top().first;
        visited = s.top().second;
        s.pop();
        if(root == NULL)
            continue;
        if(visited)
        {
            path.push_back(root->val);
        }
        else
        {
            s.push(make_pair(root->right, false));
            s.push(make_pair(root, true));
            s.push(make_pair(root->left, false));
        }
    }
}

//更简单的非递归后序遍历
void postorderTraversalNew(TreeNode *root, vector<int> &path)
{
    stack< pair<TreeNode *, bool> > s;
    s.push(make_pair(root, false));
    bool visited;
    while(!s.empty())
    {
        root = s.top().first;
        visited = s.top().second;
        s.pop();
        if(root == NULL)
            continue;
        if(visited)
        {
            path.push_back(root->val);
```

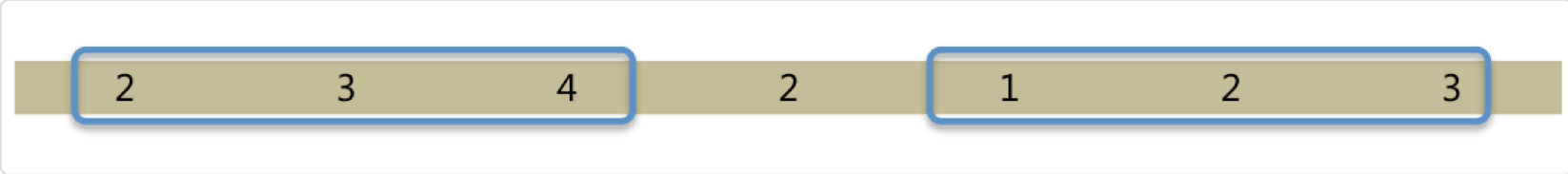
```
            }
            else
            {
                s.push(make_pair(root, true));
                s.push(make_pair(root->right, false));
                s.push(make_pair(root->left, false));
            }
        }
    }
```

以上三种遍历实现代码行数一模一样，如同递归遍历一样，只有三行核心代码的先后顺序有区别。为什么能产生这样的效果？下面我将会介绍。

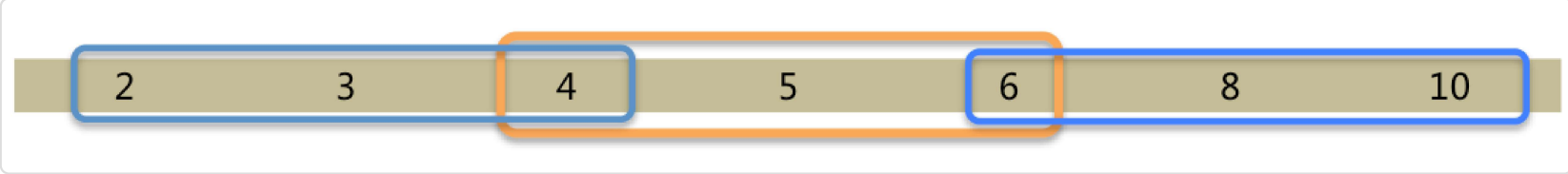
有重合元素的局部有序一定能导致整体有序

这就是我得以统一三种更简单的非递归遍历方法的基本思想：**有重合元素的局部有序一定能导致整体有序**。

如下这段序列，局部2 3 4和局部1 2 3都是有序的，但是不能由此保证整体有序。



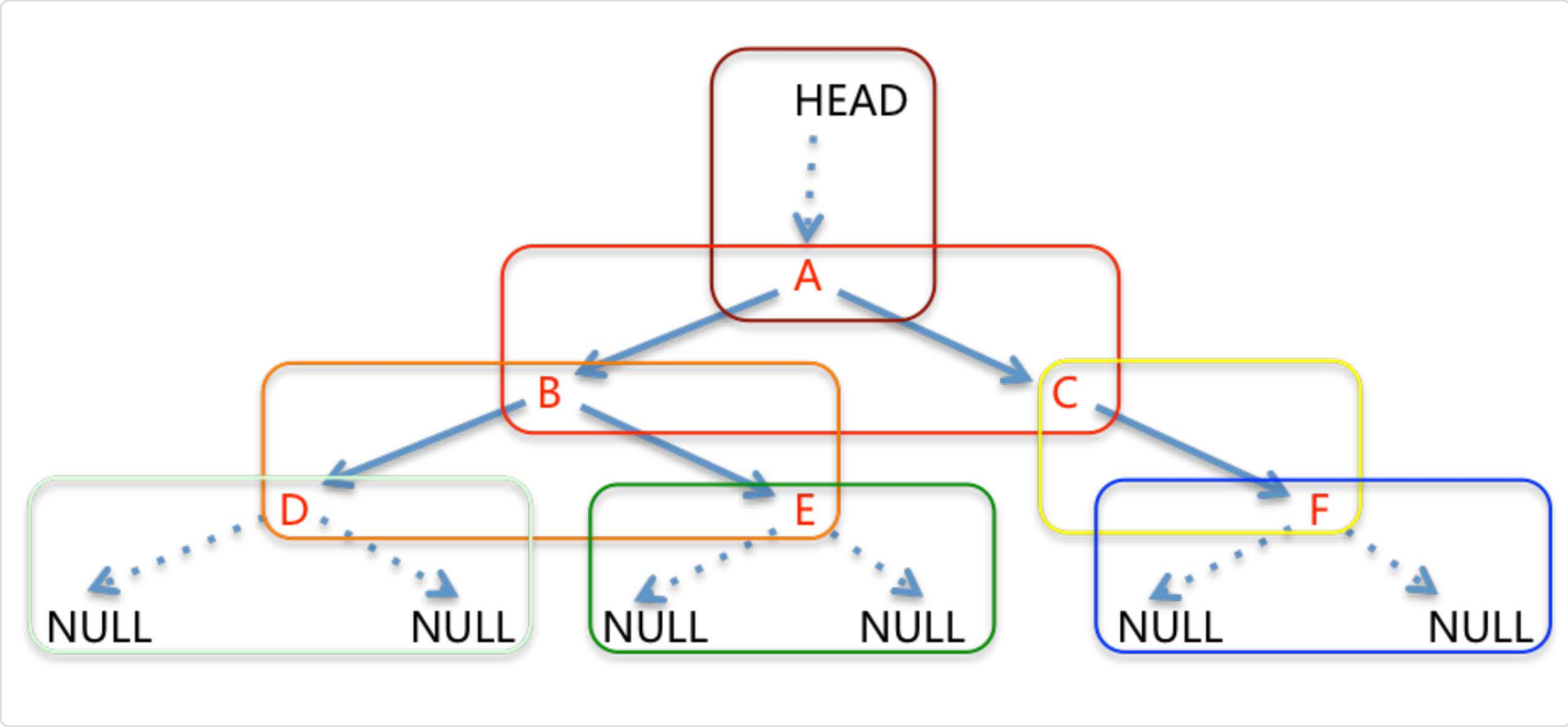
而下面这段序列，局部2 3 4,4 5 6,6 8 10都是有序的，而且相邻局部都有一个重合元素，所以保证了序列整体也是有序的。



应用于二叉树

基于这种思想，我就构思三种非递归遍历的统一思想：不管是前序，中序，后序，只要我能保证对每个结点而言，该结点，其左子结点，其右子结点都满足以前序/中序/后序的访问顺序,整个二叉树的这种三结点局部有序一定能保证整体以前序/中序/后序访问，因为相邻的局部必有重合的结点，即一个局部的“根”结点是另外一个局部的“子”结点。

如下图，对二叉树而言，将每个框内结点集都看做一个局部，那么局部有A,A B C,B D E,D,E,C F,F,并且可以发现每个结点元素都是相邻的两个局部的重合结点。发觉这个是非常关键的，因为知道了重合结点，就可以对一个局部排好序后，通过取出一个重合结点过渡到与之相邻的局部进行新的局部排序。我们可以用栈来保证局部的顺序（排在顺序前面的后入栈，排在后面的先入栈，保证这个局部元素出栈的顺序一定正确），然后通过栈顶元素(重合元素)过渡到对新局部的排序，对新局部的排序会导致该重合结点再次入栈，所以当栈顶出现已完成过渡使命的结点时，就可以彻底出栈输出了（而这个输出可以保证该结点在它过渡的那个局部一定就是排在最前面的），而新栈顶元素将会继续完成新局部的过渡。当所有结点都完成了过渡使命时，就全部出栈了，这时我敢保证所有局部元素都是有序出栈，而相邻局部必有重合元素则保证了整体的输出一定是有序的。这种思想的好处是将算法与顺序分离，定义何种顺序并不影响算法，算法只做这么一件事：**将栈顶元素取出，使以此元素为“根”结点的局部有序入栈，但若此前已通过该结点将其局部入栈，则直接出栈输出即可**。



从实现的程序中可以看到：三种非递归遍历唯一不同的就是局部入栈的三行代码的先后顺序。所以不管是根->左->右,左->根->右,左->右->根,甚至是根->右->左,右->根->左,右->左->根定义的新顺序，算法实现都无变化，除了改变局部入栈顺序。

值得一提的是，对于前序遍历，大家可能发现取出一个栈顶元素，使其局部前序入栈后，栈顶元素依然是此元素，接着就要出栈输出了，所以使其随局部入栈是没有必要的，其代码就可以简化为下面的形式。

```
void preorderTraversalNew(TreeNode *root, vector<int> &path)
{
    stack<TreeNode *> s;
    s.push(root);
    while(!s.empty())
    {
```

```
            root = s.top();
            s.pop();
            if(root == NULL)
            {
                continue;
            }
            else
            {
                path.push_back(root->val);
                s.push(root->right);
                s.push(root->left);
            }
        }
    }
```

这就是我要介绍的一种更简单的非递归遍历二叉树的方法。

如果觉得我的文章对您有用，可以随意打赏，您的支持将鼓励我持续创作。

qiaokan 2014-06-23 11:06:32

您这个虽然容易理解，但是缺点是等于多用了O(n)的空间

logicmd 2014-11-30 16:19:13

漂亮的写法，可以预料的是99%的面试官都没见过这种写法，然后以开了额外空间为由，要求你写出教科书写法。

zisong 2014-11-30 16:28:23

@logicmd 估计真是这样。我想出了这样的方法之后，即便过了很久让写，基本停顿10s，就可以一气呵成了，已经有过两次实践了。

logicmd 2014-11-30 16:31:49

@zisong 面试就是这样，对空间时间限制要求可能还稍显合理；可是对面试者写法风格也都有要求，却又说不出要求的所以然来，你跟他喷，他还觉得你合作精神差，难以共事。所以这事也是随缘了。

jojo 2014-12-31 22:17:00

非常感谢，对遍历有了更深的理解

huangqiang 2015-01-29 10:23:13

学弟来膜拜:)

huangqiang 2015-01-29 10:27:25

支付宝链接不可用，支持不了:(

zdczdcc 2015-08-25 14:14:07

栈内那个bool浪费哦

YuanX 2015-10-18 01:41:43

非递归后序遍历-迭代中有两个小错误：

1.TreeNode *tempNode = new TreeNode;

此处应为 TempNode *tempNode = new TempNode;

2.else //第二次出现在栈顶

{

path.push_back(temp->btnode->val);

p = NULL;

}

此处应在else内 delete temp

Tkkk 2017-01-22 21:41:11

@YuanX有道理。

新的结构里面加了Flag， 同时新建的对象没有释放。

称呼: 邮箱: 网站: 发表评论