# 二叉树 review

2017-04-13 | 数据结构和算法 | | | 95.5k | 23 | 21

始于生活，高于生活。

二叉树的结构：

```
class Node{
    int value;
    Node left;
    Node right;
    Node(int data){
        this.value = data;
    }
}
```

or

```
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None
```

不管是递归方法还是非递归方法，遍历整棵树的时间复杂度都是N(N),N为二叉树的节点数，额外空间复杂度为O(L),L为二叉树的层数。

## 递归方式实现先序遍历

```
public void preOrderRecur(Node head){
    if (head == null){
        return;
    }
    Systerm.out.print(head.value + " ");
    preOrderRecur(head.left);
    preOrderRecur(head.right);
}
```

### 非递归方式实现先序遍历

具体过程：

1.首先申请一个新的栈，记为stack.

2.然后将头节点head压入stack中.

3.每次从stack中弹出栈顶节点，记为cur,然后打印cur节点的值。如果cur右孩子不为空的话，将cur的右孩子先压入stack中。最后如果cur的左孩子不为空的话，将cur的左孩子压入stack中。

4.不断重复步骤3，直到stack为空，全部过程结束。

### 递归方式实现中序遍历

```
public void preOrderRecur(Node head){
    if (head == null){
        return;
    }
    Systerm.out.print(head.value + " ");
    preOrderRecur(head.left);
    preOrderRecur(head.right);
}
```

### 非递归方式实现中序遍历

具体过程：

1.首先申请一个新的栈，记为stack,申请一个变量cur,初始时令cur等于头节点.

2.先把cur节点压入stack中，对一cur节点为头的整颗子树来说，依次把整棵树的左边压入栈中，即不断令cur = cur.left，然后重复步骤2.

3.不断重复步骤2，直到发现cur为空，此时从stack中弹出一个节点，记为node.打印node的值，并让cur = node.right,然后重复步骤2.

4.当stack为空并且cur为空时，整个过程结束.

### 递归方式实现后序遍历

```
public void posOrderRecur(Node head){
    if (head == null){
        return;
    }
    posOrderRecur(head.left);
    posOrderRecur(head.right);
    Systerm.out.print(head.value + " ");
}
```

### 非递归方式实现后序遍历

方法一：使用两个栈来实现

具体过程如下：

1.申请一个栈，记为s1,然后将头节点压入s1中.

2.从s1中弹出的节点记为cur,然后先把cur的左孩子压入s1中，然后将cur1的右孩子压入s1中。

3.在整个过程中，每一个从s1中弹出的节点都放进第二个栈s2中.

4.不断重复步骤2和步骤3，直到s1为空，过程停止.

5.从s2中依次弹出节点并打印，打印的顺序就是后序遍历的顺序。

方法二：使用一个栈来实现

具体过程如下：

1.申请一个栈，记为stack，将头节点压入stack,同时设置两个变量h 和 c。在整个流程中，h 代表最近一次弹出并打印的节点，c 代表当前stack 的栈顶节点，初始时令 h 为头节点，c 为null.

2.每次令c 等于当前stack的栈顶节点，但是不从stack中弹出节点，此时分以下三种情况。

- 1.如果 c 的左孩子不为空，并且h 不等于 c 的左孩子，也不等于 c 的右孩子，则把c 的左孩子压入stack中。
- 2.如果情况1不成立，并且c 的右孩子不为空，并且h不等于 c 的右孩子，则把c 的右孩子压入stack中。
- 3.如果情况1 和情况2都不成立，那么从stack中弹出c 并打印，然后令h等于c.

4.一直重复步骤2，直到stack为空，过程停止。

### python 递归实现二叉树三种遍历

```python
# -*- coding:utf-8 -*-

class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None
class TreeToSequence:
    def convert(self, root):
        res = [[],[],[]]
        self.preorder(root, res[0])
        self.inorder(root, res[1])
        self.postorder(root, res[2])
        return res

    def preorder(self,root,res):
        if not root:
             return None
        res.append(root.val)
        self.preorder(root.left, res)
        self.preorder(root.right, res)
        return res

    def inorder(self,root,res):
        if not root:
             return None
        self.inorder(root.left, res)
        res.append(root.val)
        self.inorder(root.right, res)
        return res

    def postorder(self,root,res):
        if not root:
             return None
        self.postorder(root.left, res)
        self.postorder(root.right, res)
        res.append(root.val)
        return res
```

## python非递归实现二叉树三种遍历

```python
# -*- coding:utf-8 -*-

class TreeNode:
    def __init__(self, x):
        self.val = x
```

```
        self.left = None
        self.right = None
class TreeToSequence:
    def convert(self, root):
         res = [[],[],[]]
         self.preOrder(root, res[0])
         self.inOrder(root, res[1])
         self.postOrder(root, res[2])
         return res

    def preOrder(self, root, res):
         if not root:
             return
         stack = []
         stack.append(root)
         while stack:
             cur = stack.pop()
             res.append(cur.val)
             if cur.right:
                 stack.append(cur.right)
             if cur.left:
                 stack.append(cur.left)



    def inOrder(self, root, res):
         if not root:
             return
         stack = [root]
         cur  = root.left
         while stack or cur:
             while cur:
                 stack.append(cur)
                 cur = cur.left
             cur = stack.pop()
             res.append(cur.val)
             cur = cur.right

    def postOrder(self, root, res):
         if not root:
             return
         stack1 = []
         stack2 = []
         stack1.append(root)
         while stack1:
             cur = stack1.pop()
             stack2.append(cur.val)
             if cur.left:
                 stack1.append(cur.left)
```

```
            if cur.right:
                stack1.append(cur.right)
        while stack2:
            res.append(stack2.pop())
```

## 二叉树的打印

有一棵二叉树，请设计一个算法，按照层次打印这棵二叉树。

给定二叉树的根结点root，请返回打印结果，结果按照每一层一个数组进行储存，所有数组的顺序按照层数从上往下，且每一层的数组内元素按照从左往右排列。保证结点数小于等于500。

```
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
class TreePrinter:
    def printTree(self, root):
        # 设定需要返回打印的树为res
        res = []
        # 序列为queue
        queue = []
        # 如果根为空的情况，返回空序列
        if root == None:
            return res
        # 设定last，nlast都为根节点
        last = nlast = root
        # 将根节点添加到队列
        queue.append(root)
        temp = []
        while len(queue):
            # 从队列中pop出一个node
            node = queue.pop(0)
            # 将node的节点值赋给temp
            temp.append(node.val)
            # 如果node左节点不为空，则将其添加到队列中，并将nlast设置为此左节点
            if node.left != None:
                queue.append(node.left)
                nlast = node.left
            if node.right != None:
                queue.append(node.right)
                nlast = node.right
            # 如果当前节点等于last
            if node == last:
```

```
                res.append(temp[:])
                temp = []
                last = nlast
        return res
```

## 二叉树的序列化

首先我们介绍二叉树先序序列化的方式，假设序列化的结果字符串为str，初始时str等于空字符串。先序遍历二叉树，如果遇到空节点，就在str的末尾加上“#!” ，“#”表示这个节点为空，节点值不存在，当然你也可以用其他的特殊字符， “!”表示一个值的结束。如果遇到不为空的节点，假设节点值为3，就在str的末尾加上“3!”。现在请你实现树的先序序列化。

给定树的根结点root，请返回二叉树序列化后的字符串。

```
# -*- coding:utf-8 -*-

# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
class TreeToString:
    def __init__(self):
        self.serial = ''
    def toString(self, root):
        if root is None:
            self.serial += '#!'
        else:
            self.serial += str(root.val)
            self.serial += '!'
            self.toString(root.left)
            self.toString(root.right)
        return self.serial
```

## 平衡二叉树（AVL）

1.空树是平衡二叉树

2.如果一颗树不为空，并且其中所有的子树都满足各自的左子树与右子树的高度差都不超过91.

## 平衡二叉树判断

有一棵二叉树，请设计一个算法判断这棵二叉树是否为平衡二叉树。给定二叉树的根结点root，请返回一个bool值，代表这棵树是否为平衡二叉树。

```
# -*- coding:utf-8 -*-

class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None
class CheckBalance:

        if self.getHeight(root) >= 0:
            return True
        else:
            return False
    def getHeight(self, root):
        #后序遍历得到左右子树的深度
        if root == None:
            return True
        left = self.getHeight(root.left)
        right = self.getHeight(root.right)

        if(abs(left-right)>1):
            return -1

        return max(left, right)+1
```

## 搜索二叉树

每一颗二叉树的头节点的值都比各自左子树上的所有节点值要大，也都比各自右子数上的所有节点值要小。

搜索二叉树按照中序遍历得到的序列，一定是从小到大的排列的。逆命题也成立。

## 满二叉树

满二叉树是除了最后一层的节点无任何子节点外，剩下每一层上的节点都有两个子节点。

## 完全二叉树

完全二叉树是指除了最后一层外，其他每一层的节点数都是满的，最后一层如果也满了，是一颗满二叉树，也是完全二叉树。最后一层如果不满，缺少的节点也全部的集中在右边，那也是一颗完全二叉树。

## 完全二叉树判断

1.采用按层遍历二叉树的方式，从每层的左边向右边依次遍历所有的节点。

2.如果当前节点有右孩子，但没有左孩子，直接返回false.

3.如果当前节点并不是左右孩子全有，那之后的节点必须都为叶节点，否则返回false.

4.遍历过程中如果不返回false,遍历结束后返回true即可。

```
# -*- coding:utf-8 -*-

# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
class CheckCompletion:
    def chk(self, root):
        # write code here
        queue = []
        node = root
        queue.append(node)
        flag = 0
        while len(queue):
            temp = queue.pop(0)
            if flag:
                if temp.left != None or temp.right != None:
                    return False
            if temp.left == None and temp.right != None:
                return False
            if temp.right == None:
                flag = 1
            if temp.left != None:
                queue.append(temp.left)
            if temp.right != None:
                queue.append(temp.right)
        return True
```

扫一扫，关注我的微信公众号！

# 二叉树、binary tree

链表review

linux下c/c++ 编译器:gcc/g++ vs clang/clang++