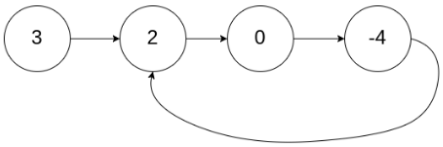
链表中环的入口节点

给定一个链表，返回链表开始入环的第一个节点。 从链表的头节点开始沿着 `next` 指针进入环的第一个节点为环的入口节点。如果链表无环，则返回 `null`。

为了表示给定链表中的环，我们使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。如果 `pos` 是 `-1`，则在该链表中没有环。**注意，`pos` 仅仅是用于标识环的情况，并不会作为参数传递到函数中。**

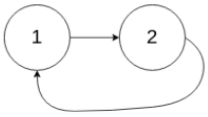
说明：不允许修改给定的链表。

示例 1：




```
输入：head = [3,2,0,-4], pos = 1
输出：返回索引为 1 的链表节点
解释：链表中有一个环，其尾部连接到第二个节点。
```

示例 2：




```
输入：head = [1,2], pos = 0
输出：返回索引为 0 的链表节点
解释：链表中有一个环，其尾部连接到第一个节点。
```

示例 3：


```
输入：head = [1], pos = -1
输出：返回 null
解释：链表中没有环。
```

解1：哈希表

利用Hash表存储节点，每次遍历前在Hash表中查找是否存在当前节点，如果存在即代表存在环，且当前节点为环的入口节点，直到遍历到null代表无环。

时间复杂度：O（N）

空间复杂度：O（N）

```
function detectCycle(head: ListNode | null): ListNode | null {
    if (head === null || head.next === null) {
        return null;
    }
    let map = new Set();
    while (head !== null) {
        if (map.has(head)) {
```

```
                return head;
            }
            map.add(head);
            head = head.next;
        }
        return null;
    };
```

解2：快慢指针

我们使用两个指针，$fast$ 与 $slow$。它们起始都位于链表的头部。随后，$slow$ 指针每次向后移动一个位置，而 $fast$ 指针向后移动两个位置。如果链表中存在环，则 $fast$ 指针最终将再次与 $slow$ 指针在环中相遇。

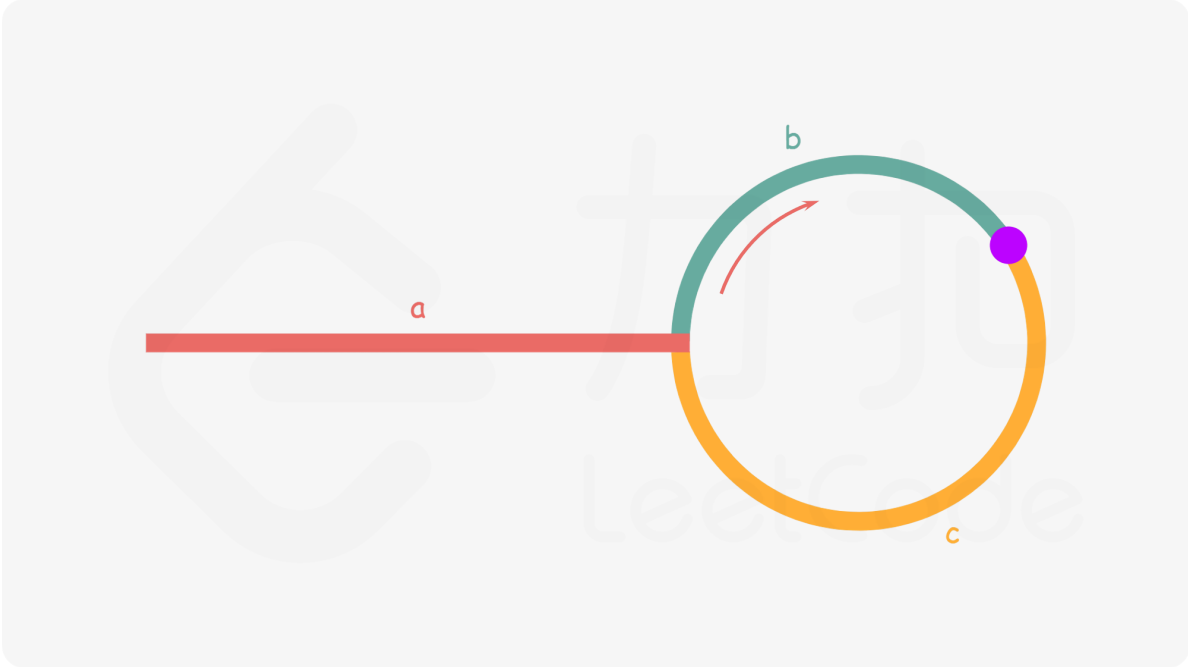
如下图所示，设链表中环外部分的长度为 a。$slow$ 指针进入环后，又走了 b 的距离与 $fast$ 相遇。此时，$fast$ 指针已经走完了环的 n 圈，因此它走过的总距离为 $a+n(b+c)+b=a+(n+1)b+nc$。

根据题意，任意时刻，$fast$ 指针走过的距离都为 $slow$ 指针的 2 倍。因此，我们有

$$a+(n+1)b+nc=2(a+b) \implies a=c+(n-1)(b+c)$$

有了 $a=c+(n-1)(b+c)$ 的等量关系，我们会发现：从相遇点到入环点的距离加上 $n-1$ 圈的环长，恰好等于从链表头部到入环点的距离。

因此，当发现 $slow$ 与 $fast$ 相遇时，我们再额外使用一个指针 ptr。起始，它指向链表头部；随后，它和 $slow$ 每次向后移动一个位置。最终，它们会在入环点相遇。



时间复杂度：O（N）

空间复杂度：O（1）

```
function detectCycle(head: ListNode | null): ListNode | null {
    if (head === null || head.next === null || head.next.next === null) {
        return null;
    }
    let slow: ListNode | null = head.next;
    let quick: ListNode | null = head.next.next;
    while (quick !== null) {
```

```
        // 如果快慢指针相遇则代表有环
        if (quick === slow) {
            break;
        }
        slow = slow.next;
        quick = quick.next === null ? null : quick.next.next;
    }
    // 如果快指针走到null代表没有环
    if (quick === null) {
        return null;
    }
    // 让快指针重新执行头节点，快慢指针同时移动一步，当快慢指针再次相遇时即为环的入口节点
    quick = head;
    while (quick !== slow) {
        quick = quick.next;
        slow = slow.next;
    }
    return slow;
};
```

两个单链表相交的一系列问题

【题目】给定两个可能有环也可能无环的单链表，头节点`head1`和`head2`。请实 现一个函数，如果两个链表相交，请返回相交的 第一个节点。如果不相交，返 回null

【要求】如果两个链表长度之和为N，时间复杂度请达到O(N)，额外空间复杂度 请达到O(1)。

解1：哈希表

第一条链表节点都入 set，第二条链表查，查到重复即为相交节点

```
// 哈希表
function getIntersectNode(head1, head2) {
    let hashMap = new Set();
    while (head1 !== null) {
        if (hashMap.has(head1)) {
            break;
        }
        hashMap.add(head1);
        head1 = head1.next;
    }
    while (head2 !== null) {
        if (hashMap.has(head2)) {
            return head2;
        }
        head2 = head2.next;
    }
    return null;
}
```

解2:

第一步：判断有无环，有环返回第一个入环节点。

第二步：讨论，获知了两个链表是否有环，分情况讨论。

- 都无环，可能相交也可能不相交，讨论两个链表的尾节点的地址是否一致，不是，则不相交，一致，则让长链表先走两个链表长度的插值步之后，再一起走，会在第一个相交节点相遇；
- 一个有环，一个无环，不相交，返回null；
- 都有环，分为三种情况，**不相交**，**相交且入环节点一样**，**相交但入环节点不一样。**
    1. 首先判断入环节点一样时，则链表1和链表2 的入环节点的内存结点相等，此时，我们可以把入环节点看作是终止节点，则将问题转换成了都无环时候的情况；
    2. 入环节点不一样时，给一个指针指向链表1的入环节点，继续往下走，在遇到自己之前，遇到了2的入环节点，则返回1或2的入环节点，如果没遇到，则说明不相交，返回null。

```
// 判断链表是否有环，并返回入环节点
function detectCycle(head) {
    if (
        head === null ||
        head.next === null ||
        head.next.next === null
    ) {
        return null;
    }
    let slow = head.next;
    let quick = head.next.next;
    while (quick !== null) {
        // 如果快慢指针相遇则代表有环
        if (quick === slow) {
            break;
        }
        slow = slow.next;
        quick = quick.next === null ? null : quick.next.next;
    }
    // 如果快指针走到null代表没有环
    if (quick === null) {
        return null;
    }
    // 让快指针重新执行头节点，快慢指针同时移动一步，当快慢指针再次相遇时即为环的入口节点
    quick = head;
    while (quick !== slow) {
        quick = quick.next;
        slow = slow.next;
    }
    return slow;
}
// 省空间的解法
function getIntersectNode2(head1, head2) {
    let loop1 = detectCycle(head1);
    let loop2 = detectCycle(head2);
    let node1 = head1;
    let node2 = head2;
    if (loop1 === null && loop2 === null) {
        // 链表1和2都无环,如果最后一个节点是同一个则有交点
        let n = 0;
        while (node1.next !== null) {
            n++;
            node1 = node1.next;
        }
        while (node2.next !== null) {
```

```
            n--;
            node2 = node2.next;
        }
        if (node1 !== node2) {
            return null;
        }
        // 将链表长的赋值为node1，短的为node2
        node1 = n > 0 ? head1 : head2;
        node2 = n > 0 ? head2 : head1;
        n = Math.abs(n);
        while (n > 0) {
            // 将node1和node2对齐
            node1 = node1.next;
            n--;
        }
        while (node1 !== node2) {
            node1 = node1.next;
            node2 = node2.next;
        }
        return node1;
    } else if (loop1 !== null && loop2 !== null) {
        if (loop1 === loop2) {
            // 两个链表的入环节点是同一个
            // 从入环节点以上截取，看成两个无环链表求交点
            let n = 0;
            while (node1 !== loop1) {
                n++;
                node1 = node1.next;
            }
            while (node2 !== loop2) {
                n--;
                node2 = node2.next;
            }
            // 将链表长的赋值为node1，短的为node2
            node1 = n > 0 ? head1 : head2;
            node2 = n > 0 ? head2 : head1;
            n = Math.abs(n);
            while (n > 0) {
                // 将node1和node2对齐
                node1 = node1.next;
                n--;
            }
            while (node1 !== node2) {
                node1 = node1.next;
                node2 = node2.next;
            }
            return node1;
        } else {
            node1 = loop1;
            while (node1 === loop2) {
                node1 = node1.next;
                if (node1 === loop1) {
                    return null;
                }
            }
            return loop2;
        }
    } else {
```

```
        return null;
    }
}
```

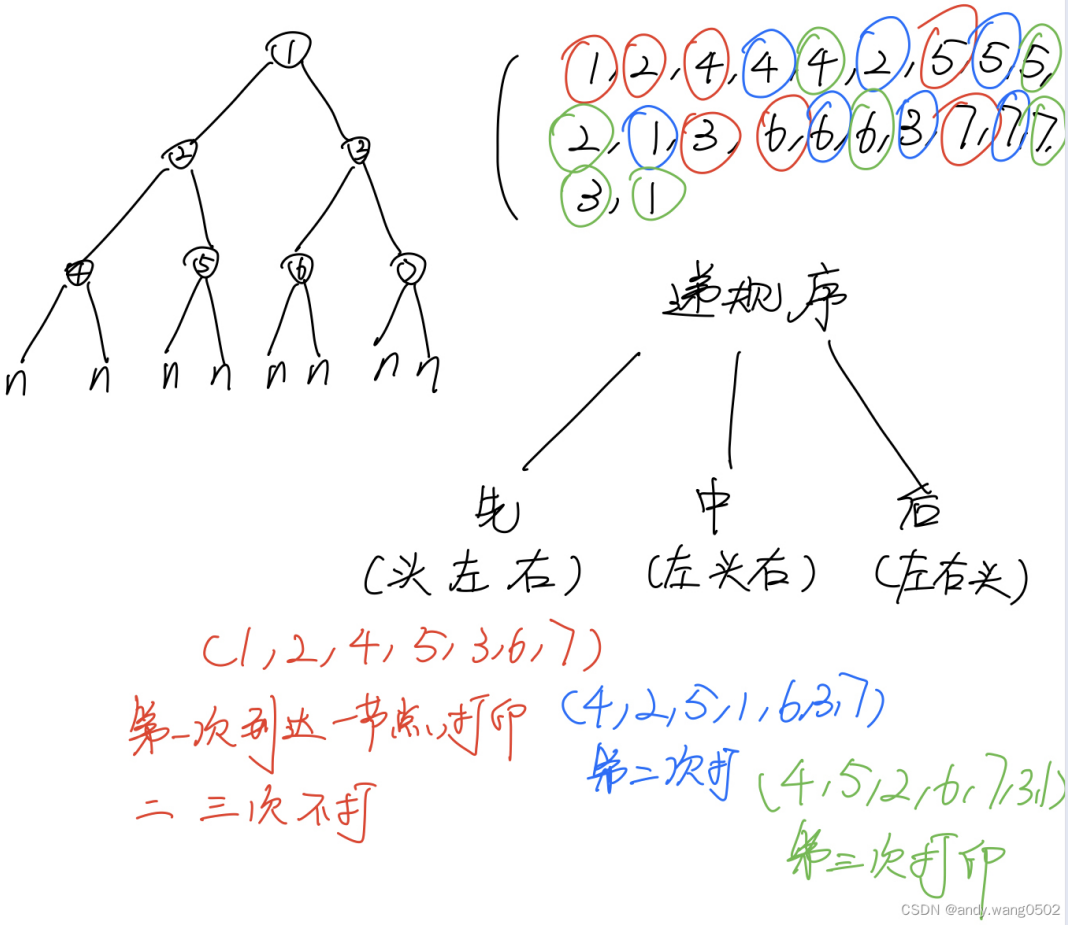
二叉树节点结构

```
 function TreeNode(val, left, right) {
     this.val = (val===undefined ? 0 : val)
     this.left = (left===undefined ? null : left)
     this.right = (right===undefined ? null : right)
}
```

二叉树

一个结点，有自己的值类型和指向孩子的两条指针

递归遍历

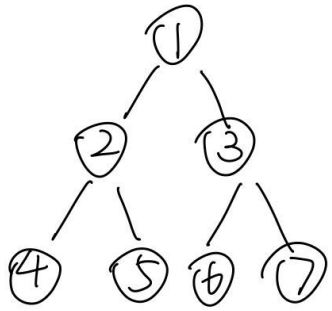



```
// 递归遍历二叉树
function f(head){
    //1
    if(head == null){
        return;
    }
    //1 第一次到达节点
    f(head.left);
    //2 第二次返回节点
    f(head.right);
    //3 第三次返回节点
}
```

前序遍历（递归）：在第一次到达节点时打印或操作

```
function process(root, arr) {
    if(root === null) {
        return null
    }
    arr.push(root.val) // 第一次到达节点时加入
    process(root.left, arr);
    process(root.right, arr);
}
var preorderTraversal = function(root) {
    let arr = []
    process(root, arr)
    return arr
};
```

前序遍历：非递归



先序 (1,2,4,5,3,6,7)

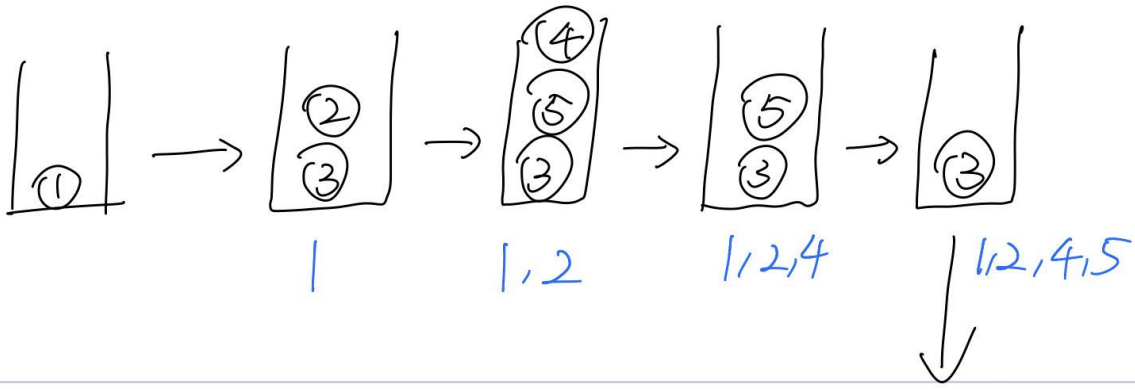
用一个栈

1) 每次从栈中弹出一个节点 Cur

2) 打印 Cur

3) 先把 Cur 右孩子压到栈里，再把 Cur 左孩子压到栈里

4) 重复




```
var preorderTraversal = function(root) {
    if (root === null) return []
    // 不使用递归遍历
    // 使用栈
    let stack = [];
    let arr = [];
    let curr = null;
    stack.push(root);
    while (stack.length !== 0) {
        curr = stack.pop();
        arr.push(curr.val);
```

```
        if (curr.right) {
            stack.push(curr.right);
        }
        if (curr.left) {
            stack.push(curr.left);
        }
    }
    return arr;
};
```

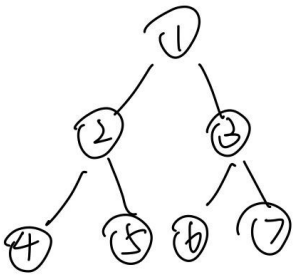
中序遍历：在第二次返回节点时打印或操作

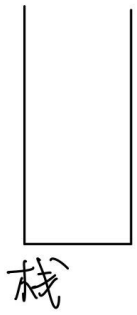
```
 function process(root, arr) {
    if(root === null) {
        return null
    }
    process(root.left, arr);
    arr.push(root.val) // 第二次返回节点时加入
    process(root.right, arr);
}
var inorderTraversal = function(root) {
    let arr = [];
    process(root,arr);
    return arr;
};
```

中序遍历（非递归）

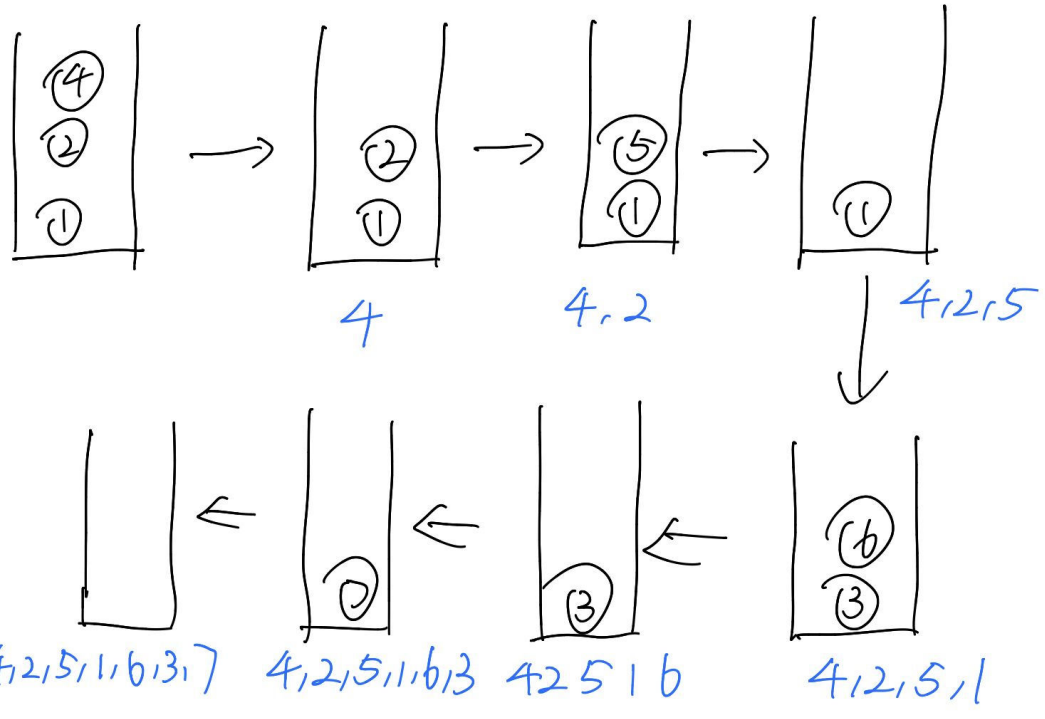
note：中序（左头右）

中序（左右头）

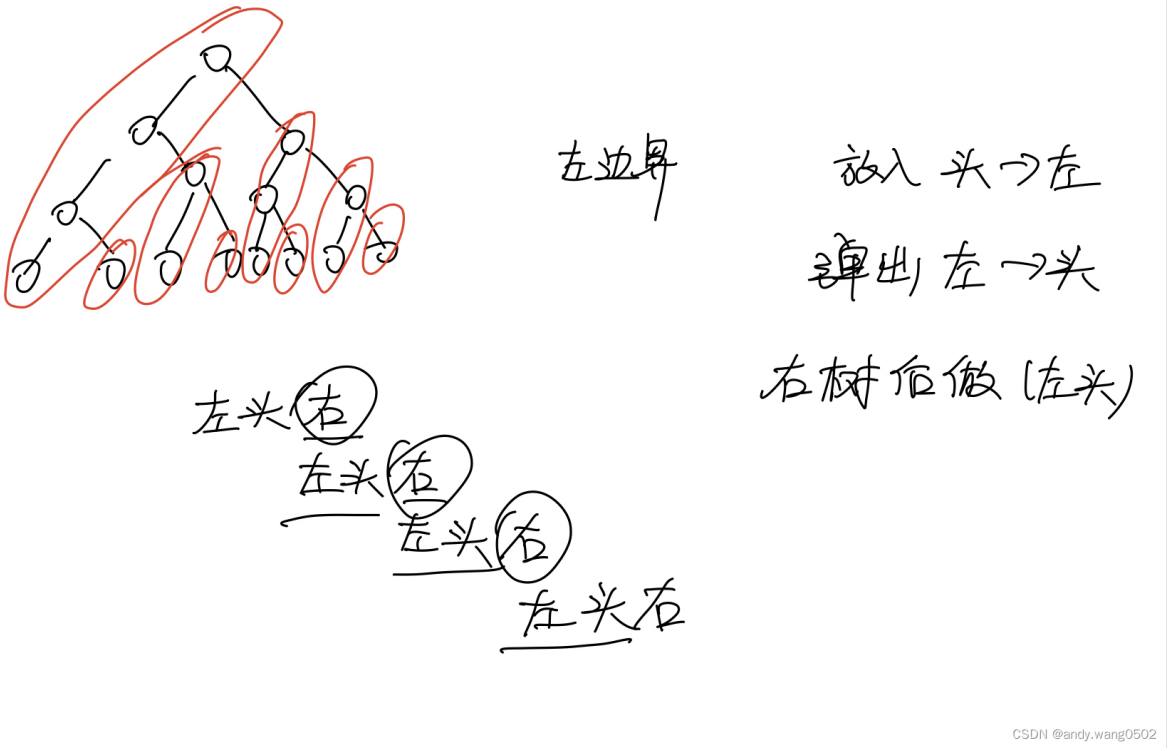




每棵子树，整棵树左边界进栈，依次弹出结点的过程中，打印，对弹出节点的右树重复



所有的树都可以被左边界分解掉



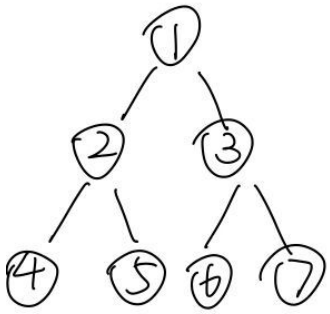

```
var inorderTraversal = function(root) {
    if (root === null) return []
    // 非递归，用栈
    let stack = [];
    let arr = [];
    let curr = root;
    while (stack.length !== 0 || curr !== null) {
        if (curr !== null) {
            stack.push(curr);
            curr = curr.left
        } else {
            curr = stack.pop();
            arr.push(curr.val);
            curr = curr.right;
        }
    }

    return arr;
};
```

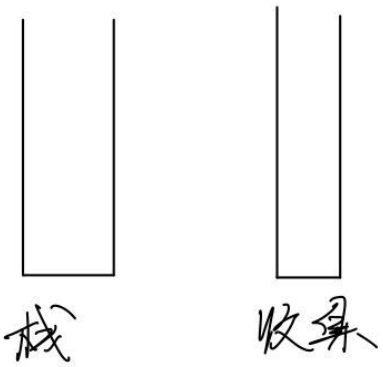
后序遍历：在第三次返回节点时打印或操作

```
function process(root, arr) {
    if(root === null) {
        return null
    }
    process(root.left, arr);
    process(root.right, arr);
    arr.push(root.val) // 第三次返回节点时加入
}
var postorderTraversal = function(root) {
    let arr = [];
    process(root, arr);
    return arr;
};
```

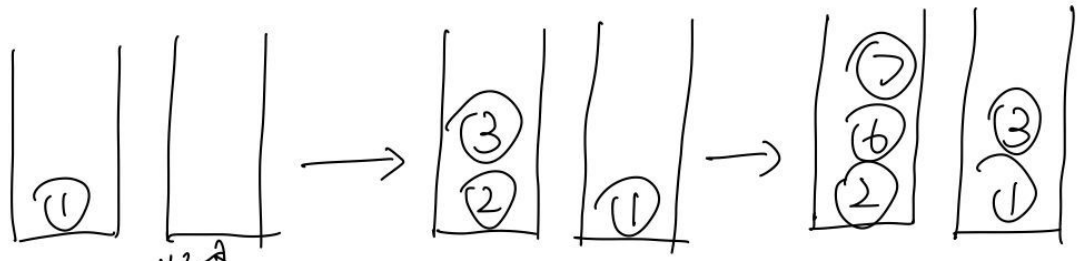
后序遍历（非递归）

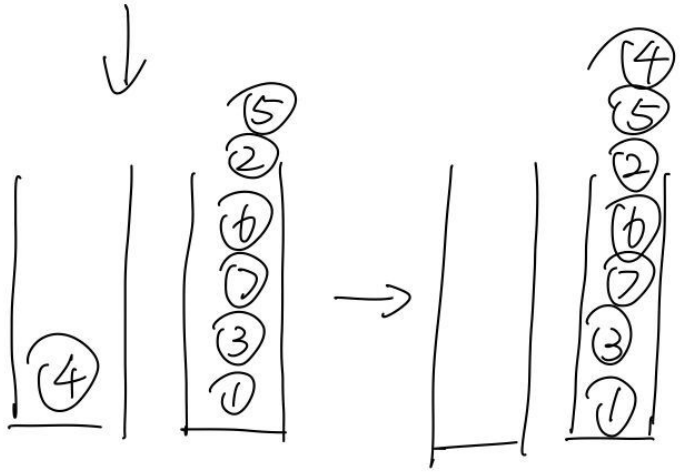
后序

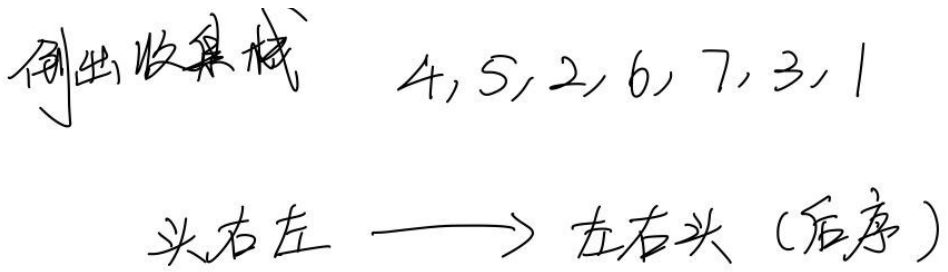




1) 弹 Cur
2) Cur放入收集栈
3) 先左后右
4) 重复








```
var postorderTraversal = function(root) {
    // 非递归
    if (root === null ) {
        return []
    }
    let arr = [];
    let stack = [];
    let collect = []; // 收集栈
    let curr = null;
    stack.push(root);
    while (stack.length !== 0) {
        curr = stack.pop();
        collect.push(curr); // 将弹出节点压入收集栈
        if (curr.left) { // 先添加左节点
            stack.push(curr.left);
        }
        if (curr.right) {
            stack.push(curr.right);
        }
    }
    while (collect.length !== 0) {
        arr.push(collect.pop().val)
    }
    return arr;
};
```

设计题：如何完成二叉树的宽度优先遍历（求一棵二叉树的宽度）

剑指 Offer 32 - I. 从上到下打印二叉树

中等 283

相关企业

从上到下打印出二叉树的每个节点，同一层的节点按照从左到右的顺序打印。

例如:
给定二叉树: `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

返回:

```
[3,9,20,15,7]
```

利用队列，弹出队头节点，将左节点放入再放右节点。直到队列为空。

```
var levelOrder = function(root) {
    if (root === null) return []
    let queen = [];
    let curr = null;
    let arr = [];
    queen.push(root);

    while (queen.length !== 0) {
        curr = queen.shift();
        if (curr.left) {
            queen.push(curr.left);
        }
        if (curr.right) {
            queen.push(curr.right);
        }
        arr.push(curr.val);
    }
    return arr;
};
```

二叉树最大宽度

空节点不计入宽度时:

准备一个`levelMap`记录点在第几层，设置三变量，当前在哪一层`curLevel`，当前层发现几个节点`curLevelNodes`，所有层中哪一层发现的节点最多的max

```
public static int void w(Node node){
     if(head == null){
        return;
}
```

```
    Queue<Node> queue = new LinkedList<>();
    queue.add(head);
    HashMap<Node, Integer> levelMap = new HashMap<>();//记录每个节点对应的层数
    levelMap.put(head,1);//放入第一个节点
    int curLevel = 1;//当前节点所在的层数
    int curLevelNodes = 0;//当前层发现几个节点数
    int max = Integer.Min_VALUE;//哪一层发现的最多的节点数
    while(!queue.isEmpty()){
        Node cur = queue.poll();
        int curNodeLevel = levelMap.get(cur);//节点的层数
        if(curNodeLebel == curLevel){//节点是否是当前统计的层
            curLevelNodes++;
}else{
        max = Math.max(max,curLevelNodes);
        curLevel++;
        curLevelNodes=1;
}
        //先放左再放右边，记录每个点所在的层数
        if(cur.left != null){
            levelMap.put(cur.left, curNodeLevel+1);
            queue.add(cur.left);
}
        if(cur.right != null){
            levelMap.put(cur.right, curNodeLevel+1);
            queue.add(cur.right);
}

}
}
```

空节点计入宽度：

662. 二叉树最大宽度

中等 578

相关企业

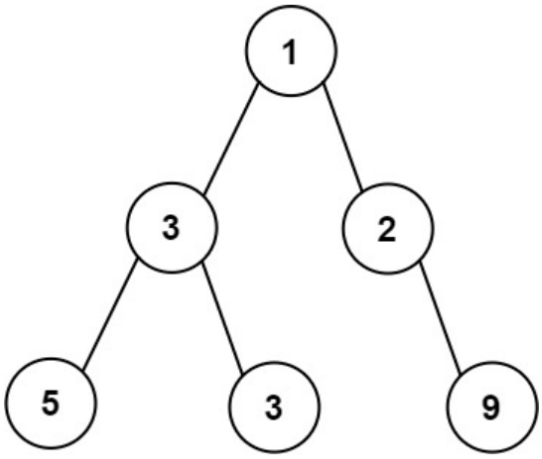
给你一棵二叉树的根节点 `root` ，返回树的 **最大宽度** 。

树的 **最大宽度** 是所有层中最大的 **宽度** 。

每一层的 **宽度** 被定义为该层最左和最右的非空节点（即，两个端点）之间的长度。将这个二叉树视作与满二叉树结构相同，两端点间会出现一些延伸到这一层的 `null` 节点，这些 `null` 节点也计入长度。

题目数据保证答案将会在 **32 位** 带符号整数范围内。

示例 1：



输入： root = [1,3,2,5,3,null,9]
输出： 4
解释： 最大宽度出现在树的第 3 层，宽度为 4 (5,3,null,9) 。

```
var widthOfBinaryTree = function(root) {
    // 广度优先遍历
    // 记录当前层和当前层的节点数
    // 记录当前层最后一个节点（上一层最后一个节点的右节点）
    if (root === null) return 0;
    let curr = null;
    let queen = [];
    let arr = [[]];
    queen.push(root);

    // 判断是否是当前层
    // 根据hashmap中的currNodeLevel和currLevel对比
    let hashMap = new Map();
    hashMap.set(root, 1)
    let currLevel = 1;
    let currNodeLevel = 1;
    // 取模防止下标溢出
    let mod = 10000000007;
    // 返回的最大值
    let max = 1;
    while (queen.length !== 0) {
        curr = queen.shift();
        currNodeLevel = hashMap.get(curr);
        console.log(curr.val)
        if (currNodeLevel !== currLevel) {
```

```
            currLevel += 1;
            arr[currLevel - 1] = []
        }
        arr[currLevel-1].push(curr.val)
        if (curr.left) {
            // 下标
            curr.left.val = (2 * curr.val - 1) % mod
            hashMap.set(curr.left, currLevel + 1)
            queen.push(curr.left);
        }
        if (curr.right) {
            curr.right.val = (2 * curr.val) % mod
            hashMap.set(curr.right, currLevel + 1)
            queen.push(curr.right);
        }
    }
    for (let i = 0; i < arr.length; i++) {
        max = max > (arr[i][arr[i].length - 1] - arr[i][0]) + 1 ? max : (arr[i]
[arr[i].length - 1] - arr[i][0]) + 1
    }
    return max;
};
```

递归——深度优先遍历

```
var widthOfBinaryTree = function(root) {
    // 深度优先遍历
    let arr = [];
    let mod = 10000000007;
    function getDeepNode(root, deep, val) {
        if (root === null) return null;
        if (!arr[deep]) {
            arr[deep] = []
        }
        arr[deep].push(val)
        getDeepNode(root.left, deep + 1, (2*val -1) % mod);
        getDeepNode(root.right, deep + 1, (2*val) % mod);
    }
    getDeepNode(root, 0, 1)
    let max = 1;
    for (let i = 0; i < arr.length; i++) {
        max = max > (arr[i][arr[i].length - 1] - arr[i][0] + 1) ? max : (arr[i]
[arr[i].length - 1] - arr[i][0] + 1)
    }
    return max;
};
```