【第二十二课】手撕红黑树(上) - 插入调整

剑指 Offer II 053. 二叉搜索树中的中序后继

结构化思维：就是在中序遍历的过程中，去查找答案，因为中序遍历可以从小到大去遍历所有的值

```
  /**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @param {TreeNode} p
 * @return {TreeNode}
 */
let pre,ans;//pre 中序遍历的前一个节点，ans 查找节点就是特定节点在中序后继中的后继节点
var inorder = function(root, p) {
    if(root == null) return false;
    if(inorder(root.left,p)) return true;
    //如果当前中序遍历的前一个节点等于当前节点，那么当前节点就是要找的后继节点
    if(pre == p){
        ans = root;
        return true;
    }
    pre = root;
    if(inorder(root.right,p)) return true;
    return false;
};
var inorderSuccessor = function(root, p) {
    let pre =  ans = null;//先赋值为空地址
    inorder(root,p);
    return ans;
};
```

117. 填充每个节点的下一个右侧节点指针 II

1.题意要求把二叉树的每一层节点连成一条链表。

2.我们这里利用常量的空间：拿着本层去连接下一层，下一层连接好了，去连接下下一层，这种做法完全不需要再开辟一个队列

封装一个lay_connect方法：传入本层的起始节点，链接下一层的节点，返回下一层的第一个节点地址，其实就是返回下一层的链表的头节点地址，一直到本层的下一层没有节点停止。

```
/**
 * // Definition for a Node.
 * function Node(val, left, right, next) {
 *    this.val = val === undefined ? null : val;
 *    this.left = left === undefined ? null : left;
 *    this.right = right === undefined ? null : right;
 *    this.next = next === undefined ? null : next;
 * };
 */

/**
 * @param {Node} root
 * @return {Node}
 */
var connect = function(root) {
    let p = root;
    while(p = lay_connect(p));
    return root;
};

var lay_connect = function(head) {
    let p = head,pre = null,new_head = null; //pre 下一层连接节点的下一个节点 ，
new_head 下一层链表的起始位置
    // 记录本层链表
    while(p){
        if(p.left){//左子树不为空，pre前面右节点开始记录
            if(pre) pre.next = p.left;
            pre = p.left;
        }
        if(new_head == null) new_head = pre;
        if(p.right){
            if(pre) pre.next = p.right;
            pre = p.right;
        }
        if(new_head == null) new_head = pre;
        p = p.next;
    }
    return new_head;
};
```

78. 子集

1.二进制的子集枚举，其实枚举的数字0-7，每个数字代表着1种选择方法

2.子集枚举法：用二进制数字1 和 0 代表着当前元素是否选择成当前的子集元素

```
var subsets = function(nums) {
    const n = nums.length;
     const ret = [];
    // i 代表着一种选取元素的方法
    for (let i = 0; i < (1 << n); ++i) {
        const arr = [];
        // 枚举I这个数字的n位
        for (let j = 0; j < n; ++j) {
            // 如果I的数字第J位等于1，证明第J个数字是我们选择的
```

```
            if (i & (1 << j)) {
                arr.push(nums[j]);
            }
        }
        ret.push(arr);
    }
    return ret;
};
```

47. 全排列 II

1.这样的题之前做过类似的，返回数组的全排列，在C++里面有现成的方法：next_permutation

2.在JS里面，首先我们对数组进行排序，让重复的数字相邻。

3.然后开始递归（要回溯），如果当前数字跟前一个相同，则跳过。

```
/**
 * @param {number[]} nums
 * @return {number[][]}
 */
var permuteUnique = function(nums) {
    let res = [];
    let len = nums.length
    nums.sort((a,b)=>{ //排序
        return a-b
    })
    unique([],0)
    return res
    function unique(arr) {
        if(arr.length == len) res.push([...arr])
        for(let i=0;i<nums.length;i++){
            if(nums[i] == nums[i-1]) continue // 跳过，避免重复结果
            arr.push(nums[i])
            nums.splice(i,1)
            unique(arr)
            nums.splice(i,0,arr.pop()) // 回溯
        }
    }

};
```

41. 缺失的第一个正数

1.把元素1存到下标0的位置，把元素2存在下标1的位置，就是把元素x存到下标x-1位置

2.然后扫描这个数组，看看元素1是不是在下标为0的位置，元素2是不是在下标为1的位置，找到第一个违反规则的位置，找到第一个没有存放正确数字的位置

3.这个就是缺失的第一个正数

```
var firstMissingPositive = function(nums) {
    for(let i = 0; i < nums.length; i++){
        while(nums[i] != i + 1){
            if(nums[i] <= 0 || nums[i] > nums.length) break;
            let ind = nums[i] - 1;
            if(nums[i] == nums[ind]) break;
            [nums[i],nums[ind]] = [nums[ind],nums[i]];
        }
    }
    // 到这，上面的代码就已经把所有的数字都放到正确的数字
    let ind = 0;
    while(ind < nums.length && nums[ind] == ind + 1) ++ind;//ind存放的是正确的数字，
ind会继续往后指
    return ind + 1;
};
```