

【第二十四周】八月月度测试题

1、[700. 二叉搜索树中的搜索](#)

方法一：迭代

因为二叉搜索树的特殊性，也就是节点的有序性，可以不使用辅助栈或者队列就可以写出迭代法。

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @param {number} val
 * @return {TreeNode}
 */
var searchBST = function (root, val) {
    while (root !== null) {
        if (root.val > val)
            root = root.left;
        else if (root.val < val)
            root = root.right;
        else
            return root;
    }
    return root;
};
```

方法二：递归

- 1、二叉搜索树的特点：根节点的所有左节点比根节点小，根节点的所有右节点比根节点大
- 2、根据二叉搜索树的特点：递归比较根节点与左右节点的大小，可以找到该值的节点

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @param {number} val
```

```

    * @return {TreeNode}
    */
    var searchBST = function (root, val) {
        if (!root || root.val === val) {
            return root;
        }
        if (root.val > val)
            return searchBST(root.left, val);
        if (root.val < val)
            return searchBST(root.right, val);
        return null;
    };

```

2、965. 单值二叉树

- 1、把根节点的值当做一个基准，如果和根节点的值不同，那么就不是单值二叉树，否则是单值二叉树。
- 2、遍历一次二叉树，每次和根节点的值比较即可。
- 3、创建一个递归函数，判断每个数的节点是否与给定值相等。

```

/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {boolean}
 */
const isUnivalTree = root => {
    // 定义一个递归函数，判断当前节点是否和给定值相等
    const check = (node, val) => {
        // 递归出口：节点空，返回true
        if (!node) return true;
        // 不相等返回false
        if (node.val !== val) return false;
        // 返回上一级的内容：左子树和右子树是否都相等
        return check(node.left, val) && check(node.right, val);
    };
    // 将根节点放入函数，根节点的值作为给定值
    return check(root, root.val);
};

```

3、1110. 删点成林

1、递归遍历树的所有节点，删除某个节点的时候，将其左、右节点放入到结果数组中即可，但是需要注意，若其左右节点也在to_delete中那么不能将其记录到result中，最后判断一下根节点

2、技巧：将to_delete转换为一个Set可以优化速度

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @param {number[]} to_delete
 * @return {TreeNode[]}
 */
var delNodes = function(root, to_delete) {
    let result = [];
    let needDelete = new Set(to_delete);
    if(!needDelete.has(root.val)){
        result.push(root);
    }

    tools(root,needDelete,{left:root});
    return result;

    function tools(node,needDelete,p){
        if(!node)return ;
        if(needDelete.has(node.val)){
            node.left && !needDelete.has(node.left.val) &&
            result.push(node.left)
            node.right && !needDelete.has(node.right.val) &&
            result.push(node.right);
        }
        node.left && tools(node.left,needDelete,node);
        node.right && tools(node.right,needDelete,node);
        if(needDelete.has(node.val)){
            node===p.left?p.left=null:p.right=null;
        }
    }
};
```

4、1022. 从根到叶的二进制数之和

下午 4:45 8月28日周六 C语言笔记 97%

2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0
 128 64 32 16 8 4 2 1
 $4+0+0=4$ 1 0 0
 $4+1=5$ 1 0 1
 $4+2=6$ 1 1 0
 $4+3=7$ 1 1 1
 $4+5+6+7=22$

42 / 42

- 1、自顶向下求出每一条路当前对应的数字，保存在入参中。
- 2、在叶子节点处将值累加起来即可。
- 3、需要注意的是，要在叶子节点就处理，而不是在 null 的时候处理，不然会重复计算。

```
var sumRootToLeaf = function(root) {
  // if(!root) return 0 //题目已知节点是 1-1000
  let ret = 0
  const dfs = (root, sum) => {
    const temp = (sum<<1) + root.val
    if(!root.left && !root.right){
      ret +=temp
      return
    }
    if(root.left) dfs(root.left, temp)
    if(root.right) dfs(root.right, temp)
  }

  dfs(root, 0)
  return ret
};
```

5、876. 链表的中间结点

用两个指针 slow 与 fast 一起遍历链表。slow 一次走一步，fast 一次走两步。那么当 fast 到达链表的末尾时，slow 必然位于中间。

```
var middleNode = function(head) {
    slow = fast = head;
    while (fast && fast.next) {
        slow = slow.next;
        fast = fast.next.next;
    }
    return slow;
};
```

6、59. 螺旋矩阵 II

- 1、构建 $n * n$ 的矩阵，确定矩阵的四个边界，它是初始遍历的边界。
- 2、模拟顺时针画矩阵的过程：（1）填充上行从左到右（2）填充右列从上到下（3）填充下行从右到左（4）填充左列从下到上。
- 3、每遍历一个格子，填上对应的 count，count自增。

```
/**
 * @param {number} n
 * @return {number[][]}
 */
var generateMatrix = function(n) {
    // new Array(n).fill(new Array(n))
    // 使用fill --> 填充的是同一个数组地址
    const res = Array.from({length: n}).map(() => new Array(n));
    let loop = n >> 1, i = 0, //循环次数
        count = 1,
        startX = startY = 0; // 起始位置
    while(++i <= loop) {
        // 定义行列
        let row = startX, column = startY;
        // [ startY, n - i) 填充左到右
        while(column < n - i){
            res[row][column++] = count++;
        }
        // [ startX, n - i) 填充上到下
        while(row < n - i) {
            res[row++][column] = count++;
        }
        // [n - i, startY) 填充右到左
        while(column > startY) {
            res[row][column--] = count++;
        }
        // [n - i, startX) 填充下到上
        while(row > startY) {
            res[row--][column] = count++;
        }
        startX = ++startY;
    }
    if(n & 1) {
```

```
    res[startX][startY] = count;
  }
  return res;
};
```

7、1480. 一维数组的动态和

- 1、从i = 1开始，循环遍历数组。
- 2、遍历时直接更新当前的数。
- 3、nums[i] = nums[i] + nums[i - 1]。
- 4、最后返回原数组nums。

```
const runningSum = nums => {
  const len = nums.length;
  for (let i = 1; i < len; i++) {
    nums[i] += nums[i - 1];
  }
  return nums;
};
```

8、151. 翻转字符串里的单词

- 1、使用 split 将字符串按空格分割成字符串数组。
- 2、使用 reverse 将字符串数组进行反转。
- 3、使用 join 方法将字符串数组拼成一个字符串。

```
var reverseWords = function(s) {
  return s.trim().split(/\s+/).reverse().join(' ');
};
```

9、1367. 二叉树中的列表

- 1.先序遍历二叉树，寻找 root.val == head.val 的二叉树节点，与链表开头不一样的直接略过
- 2.每次找到这种节点后，递归的判断该子树能否和链表匹配上，代码中的judge()函数

```
var isSubPath = function(head, root) {
  // 在一颗树上面找一条空链表肯定能找到
  if(head == null) return true;
  if(root == null) return false;
```

```

// 从root开始捋着比较，是否能找到连续的符合题意的链表
if(root.val == head.val && judge(root,head))return true;
//否则就递归地比较 用树中的每一个节点依次比较链表中的头节点
return isSubPath(head,root.left) || isSubPath(head,root.right);
};
var judge = function(root,head){
  if(head == null) return true;
  if(root == null) return false;
  if(root.val != head.val) return false;
  // 这里证明root节点的值，等于head节点的值
  // 捋着向下比较左子树，向下比较右子树
  // 在左右子树中 找到任意一条路径 能够匹配到 链表剩余部分的节点 证明能够匹配成功
  return judge(root.left,head.next) || judge(root.right,head.next);
}

```

10、669. 修剪二叉搜索树

方法一：迭代

在剪枝的时候需要注意三步：1、将root移动到 [L,R]范围内，注意是左闭右闭区间。2、剪枝左子树。3、剪枝右子树。

```

var trimBST = function(root, low, high) {
  if(root === null) {
    return null;
  }
  while(root !==null &&(root.val<low||root.val>high)) {
    if(root.val<low) {//如果该节点值小于最小值，则该节点更换为该节点的右节点值，继续遍历
      root = root.right;
    }else {//如果该节点的值大于最大值，则该节点更换为该节点的左节点值，继续遍历
      root = root.left;
    }
  }
  let cur = root;
  while(cur!==null) {
    while(cur.left&&cur.left.val<low) {
      cur.left = cur.left.right;
    }
    cur = cur.left;
  }
  cur = root;
  //判断右子树大于high的情况
  while(cur!==null) {
    while(cur.right&&cur.right.val>high) {
      cur.right = cur.right.left;
    }
    cur = cur.right;
  }
  return root;
};

```

方法二：递归

- 1、二叉搜索树的特点是，节点值大于它的左节点并且小于它的右节点。
- 2、当节点值 $< L$ ，把它的左子树抛弃掉，继续修剪它的右子树。
- 3、当节点值 $> R$ ，把它的右子树抛弃掉，继续修剪它的左子树。
- 4、否则当前节点值满足 $L < \text{node.val} < R$ ，那么它的左右子树都有可能仍然有符合条件的节点值，所以要继续修剪左、右子树。

```
var trimBST = function (root, low, high) {  
    if(root === null) {  
        return null;  
    }  
    if(root.val < low) {  
        let right = trimBST(root.right, low, high);  
        return right;  
    }  
    if(root.val > high) {  
        let left = trimBST(root.left, low, high);  
        return left;  
    }  
    root.left = trimBST(root.left, low, high);  
    root.right = trimBST(root.right, low, high);  
    return root;  
}
```

