树型DP（动态规划）解法套路

求解一个二叉树问题时（并不适用于所有二叉树问题，需判断），先列出：

1. 满足问题条件时，左子树需要满足的条件，右子树需要满足的条件；
2. 归纳左、右子树**满足条件时所需的信息**；
3. 将**左树要求和右树要求的信息求全集**，作为递归的返回；

以搜索二叉树为例

判断二叉树是否为搜索树需满足以下条件：

1. 左子树为搜索树，且左子树的最大值 < 当前节点值；
2. 右子树为搜索树，且右子树的最小值 > 当前节点值；

根据以上条件可知递归节点时需返回：

1. 当前树是否为搜索树（`isBST`）；
2. 当前树的最大值（max）；
3. 当前树的最小值（min）；

如何判断一颗二叉树是否是搜索二叉树?

98. 验证二叉搜索树

中等 2.1K

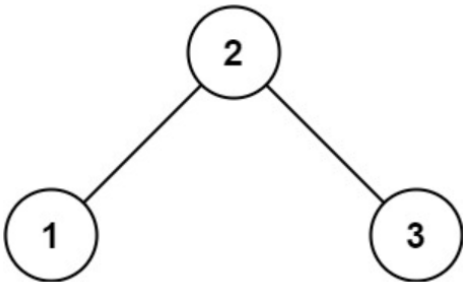
相关企业

给你一个二叉树的根节点 `root` ，判断其是否是一个有效的二叉搜索树。

有效 二叉搜索树定义如下：

- 节点的左子树只包含 **小于** 当前节点的数。
- 节点的右子树只包含 **大于** 当前节点的数。
- 所有左子树和右子树自身必须也是二叉搜索树。

示例 1：




```
输入：root = [2,1,3]
输出：true
```

解1：

直接使用中序递归-> 左中右，如果是搜索二叉树，那么最后中序递归出来的顺序一定是**升序**的

如果某个点不是升序的，那就不是搜索二叉树

```
var isValidBST = function(root) {
    if (root === null) {
        return true;
    }
    let preValue = -Infinity;
    let stack = [];
    while (stack.length !== 0 || root !== null) {
        if (root !== null) {
            stack.push(root);
            root = root.left;
        } else {
            root = stack.pop();
            if (preValue >= root.val) {
                return false;
            } else {
                preValue = root.val
            }
            root = root.right;
        }
    }
    return true;
};
```

解2（递归）：

判断二叉树是否为搜索树需满足以下条件：

1. 左子树为搜索树，且左子树的最大值 < 当前节点值；
2. 右子树为搜索树，且右子树的最小值 > 当前节点值；

根据以上条件可知递归节点时需返回：

1. 当前树是否为搜索树（`isBST`）；
2. 当前树的最大值（max）；
3. 当前树的最小值（min）；

```
function process(root) {
    if (root === null) {
        return null;
    }

    let leftData = process(root.left);
    let rightData = process(root.right);
    let isBST = true;
    let max = root.val;
    let min = root.val;
    if (
        (leftData !== null && (!leftData.isBST || leftData.max >= max)) ||
        (rightData !== null && (!rightData.isBST || rightData.min <= min))
        ) {
        isBST = false
    }

    if (leftData !== null) {
        max = max > leftData.max ? max : leftData.max;
        min = min < leftData.min ? min : leftData.min;
    }
```

```
    if (rightData !== null) {
        max = max > rightData.max ? max : rightData.max;
        min = min < rightData.min ? min : rightData.min;
    }

    return {
        isBST,
        max,
        min
    }
}
var isValidBST = function(root) {
    return process(root).isBST
};
```

如何判断一颗二叉树是完全二叉树?

958. 二叉树的完全性检验

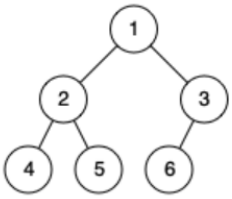
中等 258

相关企业

给定一个二叉树的 `root` ，确定它是否是一个 *完全二叉树*。

在一个 **完全二叉树** 中，除了最后一个关卡外，所有关卡都是完全被填满的，并且最后一个关卡中的所有节点都是尽可能靠左的。它可以包含 `1` 到 2^h 节点之间的最后一级 `h` 。

示例 1：




```
输入：root = [1,2,3,4,5,6]
输出：true
解释：最后一层前的每一层都是满的（即，结点值为 {1} 和 {2,3} 的两层），且最后一层中的所有结点（{4,5,6}）都尽可能地向左。
```

思路：

1. 宽度优先遍历整棵树
2. 任一节点，**有右子树，无左子树**，返回false
3. 在(2) 不成立时，如果遇到了第一个左右两个孩子不双全，**后续节点必须是叶子节点**，否则返回false

解：

```
var isCompleteTree = function(root) {
    // 宽度优先遍历
    let queen = [];
    let curr = null;
```

```
    // 记录后续节点是否需要为叶子节点
    let isLeaf = false
    queen.push(root)
    while (queen.length !== 0) {
       curr =  queen.shift();
       if (curr.left === null && curr.right !== null) {
           // 有右子树 无左子树 返回false
           return false
       }
       if (isLeaf && (curr.left !== null || curr.right !== null)) {
           // 必须为叶子节点
           return false;
       }
       if (curr.left === null || curr.right === null) {
           // 遇到第一个节点左右子节点不全时，isLeaf转为true
           isLeaf = true;
       }
       if (curr.left !== null) {
           queen.push(curr.left);
       }
       if (curr.right !== null) {
           queen.push(curr.right);
       }
    }
    return true;
};
```

如何判断一颗二叉树是否是满二叉树?

满二叉树（除了叶子节点外，所有节点都有两个子节点）的节点数N和最大深度L满足，N=2^L-1

```
function isFBTProcess(root) {
    if (root === null) {
        return {
            N: 0,
            L: 0
        }
    }
    let leftData = isFBTProcess(root.left);
    let rightData = isFBTProcess(root.right);

    return {
        N: leftData.N + rightData.N + 1,
        L: Math.max(leftData.L, rightData.L) +  1
    }
}
function isFBT(root) {
    // 满二叉树需的节点数和深度需满足 N = 2 ^ L - 1
    // 递归 左子树的深度和节点数 右子树的深度和节点数
    if (root === null) {
        return true
    }
    let {N, L} = isFBTProcess(root)
    return N === 2 ** L - 1
}
```

如何判断一颗二叉树是否是平衡二叉树?

剑指 Offer 55 - II. 平衡二叉树

简单 365

相关企业

输入一棵二叉树的根节点，判断该树是不是平衡二叉树。如果某二叉树中任意节点的左右子树的深度相差不超过1，那么它就是一棵平衡二叉树。

示例 1:

给定二叉树 `[3,9,20,null,null,15,7]`

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

返回 `true` 。

判断是否为平衡二叉树：

1. 左子树是否是平衡二叉树
2. 右子树是否是平衡二叉树
3. |左高 - 右高| 小于等于1

左子树返回的信息：是否是平衡二叉树、高度；

右子树返回的信息：是否是平衡二叉树、高度

```
var isBalanced = function(root) {
    // 左子树为平衡二叉树 右子树为平衡二叉树 左右子树深度差不超过1
    // 递归信息 是否为平衡二叉树 深度
    return isBalancedProcess(root).isBalancedTree
};
function isBalancedProcess(root) {
    if (root === null) {
        return {
            isBalancedTree: true,
            L: 0
        }
    }

    let leftData = isBalancedProcess(root.left);
    let rightData = isBalancedProcess(root.right);
    let isBalancedTree = leftData.isBalancedTree && rightData.isBalancedTree &&
Math.abs(leftData.L - rightData.L) < 2
    let L = Math.max(leftData.L, rightData.L) + 1
    return {
        isBalancedTree,
        L
    }
```

```
}
```

最低公共祖先节点

剑指 Offer 68 - I. 二叉搜索树的最近公共祖先

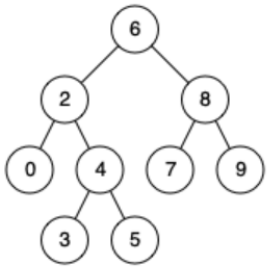
简单 316

相关企业

给定一个二叉搜索树, 找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（**一个节点也可以是它自己的祖先**）。”

例如，给定如下二叉搜索树: root = [6,2,8,0,4,7,9,null,null,3,5]



示例 1:

输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
输出: 6
解释: 节点 `2` 和节点 `8` 的最近公共祖先是 `6`。

思路1:

- 使用`hashMap`存储所有节点的父节点（root的父节点为root）
- 通过`hasMap`遍历p的所有祖先，并加入`hashset`
- 遍历q的祖先，并判断是否再`hashset`中，如果`hashset`中存在返回当前节点

```
var lowestCommonAncestor = function(root, p, q) {
    let hashMap = new Map();
    let queen = [];
    let curr = null;
    queen.push(root);
    hashMap.set(root, root);

    while (queen.length !== 0) {
        curr = queen.shift();

        if (curr.left) {
            queen.push(curr.left);
            hashMap.set(curr.left, curr);
        }
        if (curr.right) {
            queen.push(curr.right);
            hashMap.set(curr.right, curr);
```

```
        }
    }

    let hashSet = new Set();
    curr = p;
    while (curr !== hashMap.get(curr)) {
        hashSet.add(curr);
        curr = hashMap.get(curr)
    }

    curr = q;
    while (curr !== hashMap.get(curr)) {
        if (hashSet.has(curr)) {
            return curr;
        }
        curr = hashMap.get(curr);
    }
    return root
};
```

思路2:

1. p 是 q的 `LCA` (最低公共祖先), 或者 q是 p的 `LCA`

> 那么我们直接把那个更上面的那个返回出来，另外一个没有遇到 (根本不去), 代表最后只有一个然后另外一个是 null
>
> 那么整体返回的就是那一个，也就是对的

2. p和 q不互为 `LCA`, 要往上才能找到

> 那么这两个都会被找到然后往上传，一直传到他们的 `LCA` 就会直接返回那个 `LCA`

```
var lowestCommonAncestor = function(root, p, q) {
    if (root === null || root === p || root === q) {
        // p 是 q的 `LCA` (最低公共祖先), 或者 q是 p的 `LCA`
        return root;
    }

    let left = lowestCommonAncestor(root.left, p, q);
    let right = lowestCommonAncestor(root.right, p, q);

    if (left !== null && right !== null) {
        // p和 q不互为 `LCA`, 要往上才能找到
        return root;
    }
    return left !== null ? left : right;
};
```

二叉树中的后继节点

【题目】 现在有一种新的二叉树节点类型如下:

```
public class Node {
    public int value;
    public Node left;
    public Node right;
    public Node parent;
    public Node(int val) {
        value = val;
    }
}
```

该结构比普通二叉树节点结构多了一个指向父节点的parent指针。

假设有一棵Node类型的节点组成的二叉树，树中每个节点的parent指针都正确地指向自己的父节点，头节 点的parent指向null。

只给一个在二叉树中的某个节点node，请实现返回node的后继节点的函数。

在二叉树的中序遍历的序列中， node的下一个节点叫作node的后继节点。

思路:

给定节点 X 找后继节点，有两种情况:

- X 有右子树时，那么右子树的**最左**的 leaf 节点就是 X 的后继节点 (因为中序遍历时，遍历到 X 节点后下一步要遍历其右子树，而在遍历X节点的右子树时需先遍历右子树的左子树，这样X节点的**后继为右子树的最左leaf节点**)
- X 没有右子树，那么判断X节点是否是父节点的左子树节点
  - 如果是，那么这个父亲就是X的后继节点
  - 如果不是，那就向上遍历父节点，判断所遍历的节点是否是其父节点的左子树节点
    - 如果是此节点为X的后继节点
    - 如果直到root还未找到，那就代表 X节点 是整颗树最右的叶节点，中序排序最后一个节点，没有后继节点，返回 null

```
function getSuccessorNode(head) {
    if (head === null) {
        return null;
    }
    // 判断head是否有右节点
    if (head.right !== null) {
        // 返回右子树中最深的左节点
        return getLeftNode(head.right)
    }
    let parent = head.parent
    // 无右节点，则需要向上找到第一个为父亲节点左节点的节点
    while (parent !== null && head !== parent.left) {
        head = parent
        parent = head.parent;
    }
    return parent
}

function getLeftNode(head) {
```

```
    if (head === null) {
        return null;
    }
    if (head.left !== null) {
        return getLeftNode(head.left);
    } else {
        return head;
    }
}
```

二叉树的序列化和反序列化

```
//先序方式
public static String seriaByPre(Node node){
    if(head == null){
        return "#_";
    }
    String res = head.value + "_";
    res += serialByPre(head.left);
    res += serialByPre(head.right);
    return res;
}

//反序列化
public static Node reconByPreString(String preStr){
    String[] values = preStr.split("_");
    Queue<String> queue = new LinkedList<String>();
    for(int i = 0; i != values.length; i++){
        queue.add(values[i]);
    }
    return reconPreOrder(queue);
}

public static Node reconPreOrder(Queue<String> queue){
        String value = queue.poll();
        if(value.equals("#"){
        return null;
}
        Node head = new Node(Integer.valueOf(value));
        head.left = reconPreOrder(queue);
        head.right = reconPreOrder(queue);
        return head;
}
```

折纸问题

请把一段纸条竖着放在桌子上，然后从纸条的下边向上方对折1次，压出折痕后 展开。

此时折痕是凹下去的，即折痕突起的方向指向纸条的背面。

如果从纸条的下边向上方连续对折2次，压出折痕后展开，此时有三条折痕，从 上到下依次是下折痕、下折痕和上折痕。

给定一个输入参数N，代表纸条都从下边向上方连续对折N次。

请从上到下打印所有折痕的方向。

例如:N=1时，打印: down N=2时，打印: down down up

思路:

整个就是二叉树

- 左子树都是凹折痕
- 右子树都是凸折痕
- 然后整个的头节点就是凹折痕

```
//递归过程
//i是节点的层数，N一共的层数，down == true凹 down ==false 凸
public static void printProcess(int i, int N, boolean down){
    if(i > N){
        return ;
    }
    printProcess(i+1,N,true);
    System.out.println(down ? "凹" :"凸");
    printProcess(i+1,N,false);
}
```