LeetCode 98

98. 验证二叉搜索树

难度 中等 446

给定一个二叉树，判断其是否是一个有效的二叉搜索树。

假设一个二叉搜索树具有如下特征：

- 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:
    2
   / \
  1   3
输出: true
```

示例 2:

```
输入:
    5
   / \
  1   4
     / \
    3   6
输出: false
解释: 输入为: [5,1,4,null,null,3,6]。
     根节点的值为 5 ，但是其右子节点值为 4 。
```

```cpp
/*
自底向上
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) :
 *          val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        if (!root) return true;
        int maxv, minv;
```

```
        return dfs(root, maxv, minv);
    }

    bool dfs(TreeNode* root, int &maxv, int &minv)
    {
        maxv = minv = root->val;
        if (root->left)
        {
            int nowMaxv, nowMinv;
            if (!dfs(root->left, nowMaxv, nowMinv))
                return false;
            if (nowMaxv >= root->val)
                return false;
            maxv = max(maxv, nowMaxv);
            minv = min(minv, nowMinv);
        }
        if (root->right)
        {
            int nowMaxv, nowMinv;
            if (!dfs(root->right, nowMaxv, nowMinv))
                return false;
            if (nowMinv <= root->val)
                return false;
            maxv = max(maxv, nowMaxv);
            minv = min(minv, nowMinv);
        }
        return true;
    }
};
**********************************************************************************
********
/*
自顶向下
左右子树满足二叉搜索树的定义，一直缩小区间，不出现矛盾就正确
*/
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        return dfs(root,INT_MIN,INT_MAX);
    }
    bool dfs(TreeNode *root,long long minv,long long maxv)
    {
        if(!root) return true;
        if(root->val < minv || root->val > maxv) return false;

        return dfs(root->left,minv,root->val - 1ll) && dfs(root->right,root->val
+ 1ll,maxv);//左子树的区间范围[minv,root->val - 1],右子树区间范围[root->val + 1,maxv]
    }
};
```

LeetCode 94. 二叉树的中序遍历

94. 二叉树的中序遍历

难度 中等 419

给定一个二叉树，返回它的*中序*遍历。

示例:

```
输入: [1,null,2,3]
   1
    \
     2
    /
   3

输出: [1,3,2]
```

进阶: 递归算法很简单，你可以通过迭代算法完成吗?

(栈模拟递归) $O(n)$

如果用递归方法做，我们在处理每个节点时，要按照 左子树 => 根节点 => 右子树的顺序进行遍历，如下所示:

```
void dfs(TreeNode *u)
{
    if (!u) return;
    dfs(u->left);
    cout << u->val << ' ';
    dfs(u->right);
}
```

可以用栈来模拟这个过程。栈中每个元素存储两个值：TreeNode节点和一个整型的标记。

- 标记 = 0，表示还没遍历该节点的左子树；
- 标记 = 1，表示已经遍历完左子树，但还没遍历右子树；
- 标记 = 2，表示已经遍历完右子树；

然后我们可以根据标记的值，来分别处理各种情况：

- 标记 = 0，则将该节点的标记改成1，然后将其左儿子压入栈中；
- 标记 = 1，则说明左子树已经遍历完，将根节点的值插入答案序列中，然后将该节点的标记改成2，并将右儿子压入栈中；
- 标记 = 2，则说明以该节点为根的子树已经遍历完，直接从栈中弹出即可；

时间复杂度分析：树中每个节点仅会遍历一遍，且进栈出栈一次，所以时间复杂度是 $O(n)$。

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
```

```
 */
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> res;
        stack<pair<TreeNode*, int>>sta;
        sta.push(make_pair(root, 0));
        while (!sta.empty())
        {
            if (sta.top().first == NULL)
            {
                sta.pop();
                continue;
            }
            int t = sta.top().second;
            if (t == 0)
            {
                sta.top().second = 1;
                sta.push(make_pair(sta.top().first->left, 0));
            }
            else if (t == 1)
            {
                res.push_back(sta.top().first->val);
                sta.top().second = 2;
                sta.push(make_pair(sta.top().first->right, 0));
            }
            else sta.pop();
        }
        return res;
    }
};
*********************************************************************************
/*
用栈
1.将整棵树的最左边一条链压入栈中
2.每次取出栈顶元素，如果它有右子树，则将右子树压入栈中
 /**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> res;
        stack<TreeNode*> stk;

        auto p = root;
        while(p || stk.size())
        {
            //把整个左边的一条链压入栈中
            while(p)
            {
                stk.push(p);
```

```
                p = p->left;
            }

            p = stk.top();
            stk.pop();

            res.push_back(p->val);
            p = p->right;
        }

        return res;
    }
};
```

LeetCode 101.对称二叉树

101. 对称二叉树

难度 简单 648

给定一个二叉树，检查它是否是镜像对称的。

例如，二叉树 `[1,2,2,3,4,4,3]` 是对称的。

```
    1
   / \
  2   2
 / \ / \
3  4 4  3
```

但是下面这个 `[1,2,2,null,3,null,3]` 则不是镜像对称的:

```
    1
   / \
  2   2
   \   \
   3    3
```

说明:

如果你可以运用递归和迭代两种方法解决这个问题，会很加分。

(递归) $O(n)$

递归判断两个子树是否互为镜像。

两个子树互为镜像当且仅当：

1. 两个子树的根节点值相等；
2. 第一棵子树的左子树和第二棵子树的右子树互为镜像，且第一棵子树的右子树和第二棵子树的左子树互为镜像；

时间复杂度分析：从上到下每个节点仅被遍历一遍，所以时间复杂度是 $O(n)$。

```
/*
1.两个根节点的值要相等
```

```
2.左边的左子树和右边的右子树对称
3.左边的右子树和右边的左子树对称
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        return !root || dfs(root->left, root->right);
    }

    bool dfs(TreeNode*p, TreeNode*q)
    {
        if (!p || !q) return !p && !q;//同时空才对称
        return p->val == q->val && dfs(p->left, q->right) && dfs(p->right, q-
>left);
    }
};
```

(迭代) $O(n)$

用栈模拟递归，对根节点的左子树，我们用中序遍历；对根节点的右子树，我们用反中序遍历。
则两个子树互为镜像，当且仅当同时遍历两课子树时，对应节点的值相等。

时间复杂度分析：树中每个节点仅被遍历一遍，所以时间复杂度是 $O(n)$。

```
/*
按照左中右遍历左子树
按照右中左遍历右子树
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        if(!root) return true;
        stack<TreeNode*> left,right;
        auto l = root->left;
        auto r = root->right;
        while(l || r || left.size() || right.size())
        {
            while(l && r)
            {
                left.push(l);
                right.push(r);
```

```
                l = l->left;
                r = r->right;
            }

            if(l || r) return false;
            l = left.top(),left.pop();
            r = right.top(),right.pop();

            if(l->val != r->val) return false;
            l = l->right,r = r->left;
        }
        return true;
    }
};
```

LeetCode 105. 从前序与中序遍历序列构造二叉树 !!!

105. 从前序与中序遍历序列构造二叉树

难度 中等 364

根据一棵树的前序遍历与中序遍历构造二叉树。

注意:
你可以假设树中没有重复的元素。

例如，给出

```
前序遍历 preorder = [3,9,20,15,7]
中序遍历 inorder = [9,3,15,20,7]
```

返回如下的二叉树：

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

(递归) $O(n)$

递归建立整棵二叉树：先递归创建左右子树，然后创建根节点，并让指针指向两棵子树。

具体步骤如下：

1. 先利用前序遍历找根节点：前序遍历的第一个数，就是根节点的值；
2. 在中序遍历中找到根节点的位置 k，则 k 左边是左子树的中序遍历，右边是右子树的中序遍历；
3. 假设左子树的中序遍历的长度是 l，则在前序遍历中，根节点后面的 l 个数，是左子树的前序遍历，剩下的数是右子树的前序遍历；
4. 有了左右子树的前序遍历和中序遍历，我们可以先递归创建出左右子树，然后再创建根节点；

时间复杂度分析：我们在初始化时，用哈希表(`unordered_map<int,int>`)记录每个值在中序遍历中的位置，这样我们在递归到每个节点时，在中序遍历中查找根节点位置的操作，只需要 $O(1)$ 的时间。此时，创建每个节点需要的时间是 $O(1)$，所以总时间复杂度是 $O(n)$。

```
/**
```

```
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:

    unordered_map<int,int> pos;

    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        int n = preorder.size();
        for (int i = 0; i < n; i ++ )
            pos[inorder[i]] = i;
        return dfs(preorder, inorder, 0, n - 1, 0, n - 1);//递归处理
    }

    TreeNode* dfs(vector<int>&pre, vector<int>&in, int pl, int pr, int il, int
ir)
    {
        if (pl > pr) return NULL;
        int k = pos[pre[pl]] - il;
        TreeNode* root = new TreeNode(pre[pl]);
        root->left = dfs(pre, in, pl + 1, pl + k, il, il + k - 1);
        root->right = dfs(pre, in, pl + k + 1, pr, il + k + 1, ir);
        return root;
    }
};
```

LeetCode 102. 二叉树的层次遍历

102. 二叉树的层次遍历

难度 中等 402

给定一个二叉树，返回其按层次遍历的节点值。 （即逐层地，从左到右访问所有节点）。

例如:
给定二叉树: `[3,9,20,null,null,15,7]`,

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

返回其层次遍历结果：

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

(BFS) $O(n)$

宽度优先遍历，一层一层来做。即：

1. 将根节点插入队列中；
2. 创建一个新队列，用来按顺序保存下一层的所有子节点；
3. 对于当前队列中的所有节点，按顺序依次将儿子加入新队列，并将当前节点的值记录在答案中；
4. 重复步骤2-3，直到队列为空为止。

时间复杂度分析：树中每个节点仅会进队出队一次，所以时间复杂度是 $O(n)$。

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    vector<int> get_val(vector<TreeNode*> level)
    {
        vector<int> res;
        for (auto &u : level)
            res.push_back(u->val);
        return res;
    }

    vector<vector<int>> levelOrder(TreeNode* root) {
```

```
        vector<vector<int>>res;
        if (!root) return res;
        vector<TreeNode*>level;
        level.push_back(root);
        res.push_back(get_val(level));
        while (true)
        {
            vector<TreeNode*> newLevel;
            for (auto &u : level)
            {
                if (u->left) newLevel.push_back(u->left);
                if (u->right) newLevel.push_back(u->right);
            }
            if (newLevel.size())
            {
                res.push_back(get_val(newLevel));
                level = newLevel;
            }
            else break;
        }
        return res;
    }
};
*****************************************************************************
 /**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> res;

        if(!root) return res;

        queue<TreeNode*> q;
        q.push(root);

        while(q.size())
        {
            int len = q.size();//当前层的节点个数
            vector<int> level;

            for(int i = 0;i < len;i ++)//扩展当前层
            {
                auto t = q.front();
                q.pop();
                level.push_back(t->val);
                if(t->left) q.push(t->left);
                if(t->right) q.push(t->right);
            }

            res.push_back(level);
```

```
        }
        return res;
    }
};
```

LeetCode 236.二叉树的最近公共祖先

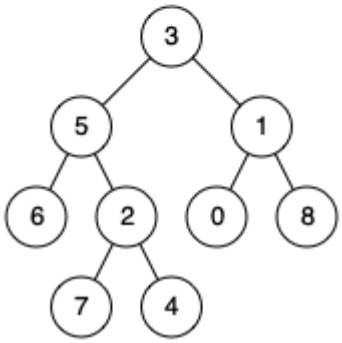
236. 二叉树的最近公共祖先

难度 中等 406

给定一个二叉树, 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，给定如下二叉树: root = [3,5,1,6,2,0,8,null,null,7,4]



示例 1:

输入： root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
输出： 3
解释： 节点 5 和节点 1 的最近公共祖先是节点 3。

示例 2:

输入： root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
输出： 5
解释： 节点 5 和节点 4 的最近公共祖先是节点 5。因为根据定义最近公共祖先节点可以为节点本身。

```
/*
返回值的情况
1.如果以root为根的子树子树中包含p和q，则返回他们的最近公共祖先
2.如果只包含p，则返回p
3.如果只包含q，则返回q
4.如果都不包含，则返回NULL
```

```
若左边为空时
1.若右边都不包含，则right = NULL 最终需要返回NULL；
2.若右边只包含p或q，则right = p或q，最终需要返回p或q；
3.若右边同时包含p和q，则right是最近公共祖先，最终返回最近公共祖先

右边也同理

左右都不空(左右各一个)那就是根节点
*/
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(!root || root == p || root == q) return root;

        auto left = lowestCommonAncestor(root->left,p,q);
        auto right = lowestCommonAncestor(root->right,p,q);

        if(!left) return right;
        if(!right) return left;
        return root;
    }
};
```

LeetCode 543. 二叉树的直径

543. 二叉树的直径

难度 简单 278

给定一棵二叉树，你需要计算它的直径长度。一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过根结点。

示例：
给定二叉树

返回 **3**, 它的长度是路径 [4,2,1,3] 或者 [5,2,1,3]。

注意：两结点之间的路径长度是以它们之间边的数目表示。

(递归遍历) $O(n)$

1. 递归函数的返回值定义为从当前结点到叶子结点的最大长度，当前结点为空返回 -1。
2. 递归时，分别得到左右子树递归的返回值，则可以更新答案 ans = max(ans, d1 + d2 + 2)；然后返回 max(d1, d2) + 1。

时间复杂度

- 每个结点最多仅被遍历一次，故时间复杂度为 $O(n)$。

```
/*枚举所有最高点
最高的点左右也是最高点

递归求左右子树最大深度的长度之和
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    int dfs(TreeNode *r, int &ans) {
        if (r == NULL)
            return -1;
        int d1 = dfs(r -> left, ans);
        int d2 = dfs(r -> right, ans);
        ans = max(ans, d1 + d2 + 2);
        return max(d1, d2) + 1;
    }
    int diameterOfBinaryTree(TreeNode* root) {
        int ans = 0;
        dfs(root, ans);
        return ans;
    }
};
*********************************************************************************
*****
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    int ans = 0;
    int diameterOfBinaryTree(TreeNode* root) {
        dfs(root);

        return ans;
    }
```

```
    int dfs(TreeNode* root)
    {
        if(!root) return 0;
        auto left = dfs(root->left);
        auto right = dfs(root->right);

        ans = max(ans,left + right);
        return max(left + 1,right + 1);
    }
};
```

LeetCode 124. 二叉树中的最大路径和

124. 二叉树中的最大路径和

难度 困难 341

给定一个**非空**二叉树，返回其最大路径和。

本题中，路径被定义为一条从树中任意节点出发，达到任意节点的序列。该路径**至少包含一个**节点，且不一定经过根节点。

示例 1:

输入： [1,2,3]

```
      1
     / \
    2   3
```

输出： 6

示例 2:

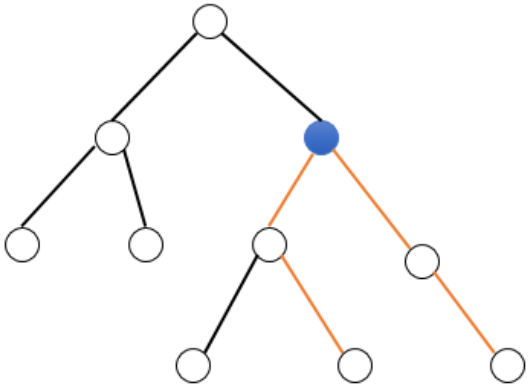
输入： [-10,9,20,null,null,15,7]

```
   -10
   / \
  9  20
    /  \
   15   7
```

输出： 42

(递归，树的遍历) $O(n^2)$

树中每条路径，都存在一个离根节点最近的点，我们把它记为割点，用割点可以将整条路径分为两部分：从该节点向左子树延伸的路径，和从该节点向右子树延伸的部分，而且两部分都是自上而下延伸的。如下图所示，蓝色的节点为割点：



我们可以递归遍历整棵树，递归时维护从每个节点开始往下延伸的最大路径和。
对于每个点，递归计算完左右子树后，我们将左右子树维护的两条最大路径，和该点拼接起来，就可以得到以这个点为割点的最大路径。
然后维护从这个点往下延伸的最大路径：从左右子树的路径中选择权值大的一条延伸即可。

```
/*
枚举最高点
左右两边的最大权重
向左走，向右走，不走，三种情况，取最大值
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    int ans = INT_MIN;
    int maxPathSum(TreeNode* root) {
        dfs(root);
        return ans;
    }
    //从root向下走的最大值
    int dfs(TreeNode *root)
    {
        if(!root) return 0;

        auto left = dfs(root->left);
        auto right = dfs(root->right);
        ans = max(ans,left + root->val + right);
        return max(0,root->val + max(left,right));//向左走，向右走，不走，三种情况，取最大值
    }
};
```

LeetCode 173. 二叉搜索树迭代器

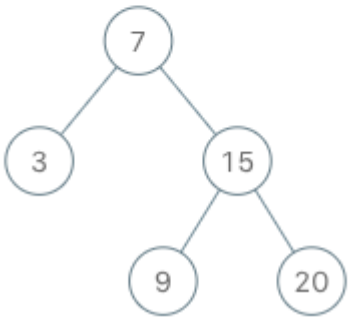
173. 二叉搜索树迭代器

难度 中等 139

实现一个二叉搜索树迭代器。你将使用二叉搜索树的根节点初始化迭代器。

调用 `next()` 将返回二叉搜索树中的下一个最小的数。

示例：




```
BSTIterator iterator = new BSTIterator(root);
iterator.next();    // 返回 3
iterator.next();    // 返回 7
iterator.hasNext(); // 返回 true
iterator.next();    // 返回 9
iterator.hasNext(); // 返回 true
iterator.next();    // 返回 15
iterator.hasNext(); // 返回 true
iterator.next();    // 返回 20
iterator.hasNext(); // 返回 false
```

提示：

- `next()` 和 `hasNext()` 操作的时间复杂度是 O(1)，并使用 O(*h*) 内存，其中 *h* 是树的高度。
- 你可以假设 `next()` 调用总是有效的，也就是说，当调用 `next()` 时，BST 中至少存在一个下一个最小的数。

(栈)

1. 用栈来模拟BST的中序遍历过程，当前结点进栈，代表它的左子树正在被访问。栈顶结点代表当前访问到的结点。
2. 求后继时，只需要弹出栈顶结点，取出它的值。然后将它的右儿子以及右儿子的左儿子等一系列结点进栈，这一步代表找右子树中的最左子结点，并记录路径上的所有结点。
3. 判断是否还存在后继只需要判断栈是否为空即可，因为栈顶结点是下一次即将被访问到的结点。

```
/**
 * Definition for binary tree
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
```

```
 * };
 */
class BSTIterator {
public:
    stack<TreeNode*> st;
    BSTIterator(TreeNode *root) {
        TreeNode *p = root;
        while (p) {
            st.push(p);
            p = p -> left;
        }
    }

    /** @return whether we have a next smallest number */
    bool hasNext() {
        return !st.empty();
    }

    /** @return the next smallest number */
    int next() {
        TreeNode *cur = st.top();
        st.pop();
        int v = cur -> val;
        cur = cur -> right;
        while (cur) {
            st.push(cur);
            cur = cur -> left;
        }
        return v;
    }
};

/**
 * Your BSTIterator will be called like this:
 * BSTIterator i = BSTIterator(root);
 * while (i.hasNext()) cout << i.next();
 */
********************************************************************************
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class BSTIterator {
public:
    stack<TreeNode*> stk;
    BSTIterator(TreeNode* root) {
        while(root)
        {
            stk.push(root);
            root = root->left;
        }
    }
```

```
    /** @return the next smallest number */
    int next() {
        auto p = stk.top();
        stk.pop();
        int res = p->val;
        p = p->right;
        while(p)          //p不为空就把p最左边的链加入栈中
        {
            stk.push(p);
            p = p->left;
        }

        return res;
    }

    /** @return whether we have a next smallest number */
    bool hasNext() {
        return !stk.empty();
    }
};

/**
 * Your BSTIterator object will be instantiated and called as such:
 * BSTIterator* obj = new BSTIterator(root);
 * int param_1 = obj->next();
 * bool param_2 = obj->hasNext();
 */
```

LeetCode 297. 二叉树的序列化与反序列化

297. 二叉树的序列化与反序列化

难度 困难 145

序列化是将一个数据结构或者对象转换为连续的比特位的操作，进而可以将转换后的数据存储在一个文件或者内存中，同时也可以通过网络传输到另一个计算机环境，采取相反方式重构得到原数据。

请设计一个算法来实现二叉树的序列化与反序列化。这里不限定你的序列 / 反序列化算法执行逻辑，你只需要保证一个二叉树可以被序列化为一个字符串并且将这个字符串反序列化为原始的树结构。

示例:

你可以将以下二叉树：

序列化为 "[1,2,3,null,null,4,5]"

提示: 这与 LeetCode 目前使用的方式一致，详情请参阅 LeetCode 序列化二叉树的格式。你并非必须采取这种方式，你也可以采用其他的方法解决这个问题。

说明: 不要使用类的成员 / 全局 / 静态变量来存储状态，你的序列化和反序列化算法应该是无状态的。

(先序遍历序列化) $O(n)$

1. 我们按照先序遍历，即可完整唯一的序列化一棵二叉树。但空结点需要在序列化中有所表示。
2. 例如样例中的二叉树可以表示为 `"1,2,n,n,3,4,n,n,5,n,n,"` ，其中 `n` 可以去掉，进行简化。
3. 通过DFS即可序列化该二叉树；反序列化时，按照 `,` 作为分隔，构造当前结点后分别通过递归构造左右字数即可。

时间复杂度

- 每个结点仅遍历两次，故时间复杂度为$O(n)$。

```
/*
序列化
前序遍历，用逗号隔开，空的为#号

反序列化
先左子树，再右子树

/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
```

```
class Codec {
public:

    void solve(TreeNode* root, string& s) {
        if (root == NULL) {
            s += ",";
            return;
        }
        s += to_string(root -> val) + ",";
        solve(root -> left, s);
        solve(root -> right, s);
    }
    // Encodes a tree to a single string.
    string serialize(TreeNode* root) {
        string s = "";
        solve(root, s);
        return s;
    }

    TreeNode* decode(int &cur, const string& data) {
        if (data[cur] == ',') {
            cur++;
            return NULL;
        }
        int nxt = data.find(',', cur);
        int val = stoi(data.substr(cur, nxt - cur));
        TreeNode *r = new TreeNode(val);
        cur = nxt + 1;
        r -> left = decode(cur, data);
        r -> right = decode(cur, data);
        return r;
    }
    // Decodes your encoded data to tree.
    TreeNode* deserialize(string data) {
        int cur = 0;
        return decode(cur, data);
    }
};

// Your Codec object will be instantiated and called as such:
// Codec codec;
// codec.deserialize(codec.serialize(root));
********************************************************************************
***
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Codec {
public:

    // Encodes a tree to a single string.
    string serialize(TreeNode* root) {
```

```
        string res;
        dfs1(root,res);//不能直接用返回值
        return res;
    }
    void dfs1(TreeNode *root,string &res)
    {
        if(!root)
        {
            res += "#,";
            return;
        }

        res += to_string(root->val) + ',';
        dfs1(root->left,res);
        dfs1(root->right,res);
    }
    // Decodes your encoded data to tree.
    TreeNode* deserialize(string data) {
        int u = 0;
        return dfs2(data,u);

    }
    TreeNode* dfs2(string &data,int &u)
    {
        if(data[u] == '#')
        {
            u += 2;
            return NULL;
        }

        int t = 0;
        bool is_minus = false;
        if(data[u] == '-')
        {
            is_minus = true;
            u ++;
        }
        while(data[u] != ',')
        {
            t = t * 10 + data[u] - '0';
            u ++;
        }
        u ++;
        if(is_minus) t = -t;

        auto root = new TreeNode(t);
        root->left = dfs2(data,u);
        root->right = dfs2(data,u);

        return root;

    }
};

// Your Codec object will be instantiated and called as such:
// Codec codec;
// codec.deserialize(codec.serialize(root));
```