144. 二叉树的前序遍历（中等）

1. 题目描述

   给定一个二叉树，返回它的 *前序* 遍历。

   示例:

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

   输出: [1,2,3]
   ```

   进阶: 递归算法很简单，你可以通过迭代算法完成吗?

2. 简单实现

   一直找左孩子并入栈，直到为NULL时进入栈顶元素右孩子，继续找右孩子的左孩子

   ```
   class Solution {
   public:
       vector<int> preorderTraversal(TreeNode* root) {
           if(!root) return vector<int>();
           stack<TreeNode*> s;
           vector<int> ans;
           while(root){
               s.push(root);
               ans.push_back(root->val);
               root = root->left;
           }
           while(!s.empty()){
               TreeNode* cur = s.top();
               s.pop();
               if(cur->right){
                   cur = cur->right;
                   while(cur){
                       s.push(cur);
                       ans.push_back(cur->val);
                       cur = cur->left;
                   }
               }
           }
           return ans;
       }
   };
   ```

94.中序遍历二叉树（中等）

1. 题目描述

   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?

2. 简单实现

   和前序遍历过程一样，只是在出栈时才将值push进ans

   ```
   class Solution {
   public:
       vector<int> inorderTraversal(TreeNode* root) {
           if(!root) return vector<int>();
           stack<TreeNode*> s;
           vector<int> ans;
           while(root){
               s.push(root);
               root = root->left;
           }
           while(!s.empty()){
               TreeNode* cur = s.top();
               s.pop();
               ans.push_back(cur->val);
               if(cur->right){
                   cur = cur->right;
                   while(cur){
                       s.push(cur);
                       cur = cur->left;
                   }
               }
           }
           return ans;
       }
   };
   ```

145.二叉树的后序遍历（中等）

1. 题目描述

   给定一个二叉树，返回它的 *后序* 遍历。

   示例:

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

输出: [3,2,1]
```

   进阶: 递归算法很简单，你可以通过迭代算法完成吗?

2. 简单实现

   在中序遍历的基础上，增设pre指针，指向前一个遍历的节点，用来判断右子树是否遍历过

```
class Solution {
public:
    vector<int> postorderTraversal(TreeNode* root) {
        if(!root) return vector<int>();
        stack<TreeNode*> s;
        vector<int> ans;
        while(root){
            s.push(root);
            root = root->left;
        }
        TreeNode* pre = NULL;
        while(!s.empty()){
            TreeNode* cur = s.top();
            if(!cur->right || cur->right == pre){//无需或右子树遍历完
                s.pop();
                ans.push_back(cur->val);
                pre = cur;
            }
            else{//进入右子树
                cur = cur->right;
                while(cur){
                    s.push(cur);
                    cur = cur->left;
                }
            }
        }
        return ans;
    }
};
```

103.二叉树的层次遍历（中等）

1. 题目描述

   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]
]
```

2. 简单实现

```cpp
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        if(!root) return vector<vector<int>>();
        vector<vector<int>> ans;
        queue<TreeNode*> q;
        q.push(root);
        while(!q.empty()){
            int size = q.size();
            vector<int> cur = vector<int>(size);
            for(int i = 0; i < size; i++){
                TreeNode* temp = q.front();
                q.pop();
                cur[i] = temp->val;
                if(temp->left) q.push(temp->left);
                if(temp->right) q.push(temp->right);
            }
            ans.push_back(cur);
        }
        return ans;
    }
};
```

运用递归解决树的问题

“自顶向下” 的解决方案

“自顶向下” 意味着在每个递归层级，我们将首先访问节点来计算一些值，并在递归调用函数时将这些值传递到子节点。 所以 “自顶向下” 的解决方案可以被认为是一种前序遍历。 具体来说，递归函数

```
top_down(root, params)
```

的原理是这样的：

```
1. return specific value for null node
2. update the answer if needed                      // anwer <-- params
3. left_ans = top_down(root.left, left_params)      // left_params <-- root.val, params
4. right_ans = top_down(root.right, right_params)   // right_params <-- root.val,
params
5. return the answer if needed                      // answer <-- left_ans, right_ans
```

例如，思考这样一个问题：给定一个二叉树，请寻找它的最大深度。

我们知道根节点的深度是 `1`。 对于每个节点，如果我们知道某节点的深度，那我们将知道它子节点的深度。 因此，在调用递归函数的时候，将节点的深度传递为一个参数，那么所有的节点都知道它们自身的深度。 而对于叶节点，我们可以通过更新深度从而获取最终答案。 这里是递归函数 `maximum_depth(root, depth)` 的伪代码：

```
1. return if root is null
2. if root is a leaf node:
3.      answer = max(answer, depth)         // update the answer if needed
4. maximum_depth(root.left, depth + 1)      // call the function recursively for left
child
5. maximum_depth(root.right, depth + 1)     // call the function recursively for right
child
```

“自底向上” 的解决方案

“自底向上” 是另一种递归方法。 在每个递归层次上，我们首先对所有子节点递归地调用函数，然后根据返回值和根节点本身的值得到答案。 这个过程可以看作是后序遍历的一种。 通常， “自底向上” 的递归函数

```
bottom_up(root)
```

为如下所示：

```
1. return specific value for null node
2. left_ans = bottom_up(root.left)          // call function recursively for left child
3. right_ans = bottom_up(root.right)        // call function recursively for right
child
4. return answers                           // answer <-- left_ans, right_ans, root.val
```

让我们继续讨论前面关于树的最大深度的问题，但是使用不同的思维方式：对于树的单个节点，以节点自身为根的子树的最大深度 `x` 是多少？

如果我们知道一个根节点，以其**左**子节点为根的最大深度为 `l` 和以其**右**子节点为根的最大深度为 `r`，我们是否可以回答前面的问题？ 当然可以，我们可以选择它们之间的最大值，再加上1来获得根节点所在的子树的最大深度。那就是 `x = max (l, r) + 1` 。

这意味着对于每一个节点来说，我们都可以在解决它子节点的问题之后得到答案。 因此，我们可以使用“自底向上”的方法。下面是递归函数 `maximum_depth(root)` 的伪代码：

```
1. return 0 if root is null                  // return 0 for null node
2. left_depth = maximum_depth(root.left)
3. right_depth = maximum_depth(root.right)
4. return max(left_depth, right_depth) + 1  // return depth of the subtree rooted at
root
```

总结

当遇到树问题时，请先思考一下两个问题：

1. 你能确定一些参数，从该节点自身解决出发寻找答案吗？
2. 你可以使用这些参数和节点本身的值来决定什么应该是传递给它子节点的参数吗？

如果答案都是肯定的，那么请尝试使用 “ `自顶向下` ” 的递归来解决此问题。

或者你可以这样思考：对于树中的任意一个节点，如果你知道它子节点的答案，你能计算出该节点的答案吗？ 如果答案是肯定的，那么 “ `自底向上` ” 的递归可能是一个不错的解决方法。

104.二叉树的最大深度（简单）

1. 题目描述

   给定一个二叉树，找出其最大深度。

   二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

   说明: 叶子节点是指没有子节点的节点。

   示例： 给定二叉树 `[3,9,20,null,null,15,7]`，

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

   返回它的最大深度 3 。

2. 简单实现

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if(!root)
            return 0;
        else
            return 1+max(maxDepth(root->left),maxDepth(root->right));
    }
};
```

101.对称二叉树（简单）

1. 题目描述

   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。

2. 简单实现

```
class Solution {
public:
    bool isequal(TreeNode* root1, TreeNode* root2){
        if(root1 == NULL && root2 == NULL)
            return true;
        else if((root1 == NULL && root2 != NULL) || (root1 != NULL && root2 ==
NULL))
            return false;

        if(root1->val != root2->val)
            return false;
        else
            return isequal(root1->left, root2->right) && isequal(root1->right,
root2->left);
```

```
    }
    bool isSymmetric(TreeNode* root) {
        if(!root) return true;
        return isequal(root->left, root->right);
    }

};
```

112. 路径总和（简单）

1. 题目描述

   给定一个二叉树和一个目标和，判断该树中是否存在根节点到叶子节点的路径，这条路径上所有节点值相加等于目标和。

   说明: 叶子节点是指没有子节点的节点。

   示例: 给定如下二叉树，以及目标和 `sum = 22`，

   ```
                 5
                / \
               4   8
              /   / \
             11  13  4
            /  \      \
           7    2      1
   ```

   返回 `true`, 因为存在目标和为 22 的根节点到叶子节点的路径 `5->4->11->2`。

2. 简单实现

```
class Solution {
public:
    bool hasPathSum(TreeNode* root, int sum) {
        if(!root) return false;
        if(!root->left && !root->right) return root->val == sum;
        return hasPathSum(root->left, sum-root->val) || hasPathSum(root->right,
sum-root->val);
    }
};
```

106.从中序与后序遍历序列构造二叉树（中等）

1. 题目描述

   根据一棵树的中序遍历与后序遍历构造二叉树。

   注意: 你可以假设树中没有重复的元素。

   例如，给出

```
中序遍历 inorder = [9,3,15,20,7]
后序遍历 postorder = [9,15,7,20,3]
```

返回如下的二叉树：

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

2. 简单实现

```
class Solution {
public:
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        int len = inorder.size();
        if(len == 0) return NULL;
        if(len == 1) return new TreeNode(inorder[0]);//只有一个节点，作为根节点返回
        TreeNode* root = new TreeNode(postorder[len-1]);//后序遍历的最后一个节点值为根节
点值
        int idx = 0;
        //找到根节点在中序遍历中的位置，其左方为左子树中序遍历结果，右方为右子树中序遍历结果
        while(inorder[idx] != postorder[len-1]) idx++;
        if(idx != 0){//有左子树
            vector<int> inorder_left = vector<int>(inorder.begin(),
                                                   inorder.begin()+idx);//左子树中序
遍历结果
            vector<int> postorder_left = vector<int>(postorder.begin(),
                                                     postorder.begin()+idx);//左子树
后序遍历结果
            root->left = buildTree(inorder_left, postorder_left);
        }
        if(idx != len-1){//有右子树
            vector<int> inorder_right = vector<int>(inorder.begin()+idx+1,
                                                    inorder.end());//右子树中序遍历结
果
            vector<int> postorder_right = vector<int>(postorder.begin()+idx,
                                                      postorder.end()-1);//右子树后
序遍历结果
            root->right = buildTree(inorder_right, postorder_right);
        }
        return root;
    }
};
```

3. 优化

- 可以将中序遍历各个值的索引做成哈希映射，使查找时间由O(N)变成O(1)，这样算法复杂度会从O(NlogN)变成O(logN)
- 不需要重新拷贝数子树组，直接传递其边界值即可

```
class Solution {
public:
    unordered_map<int, int> m;
    void init(vector<int>& v){
        for(int i = 0; i < v.size(); i++) m[v[i]] = i;
    }

    TreeNode* build(vector<int>& inorder, int l1, int r1, vector<int>&
postorder, int l2, int r2){
        if(l1 == r1) return new TreeNode(inorder[l1]);//只有一个节点，作为根节点返
回
        TreeNode* root = new TreeNode(postorder[r2]);//后序遍历的最后一个节点值为根
节点值
        //找到根节点在中序遍历中的位置，其左方为左子树中序遍历结果，右方为右子树中序遍历结果
        int idx = m[postorder[r2]];
        if(idx != l1)//有左子树
            root->left = build(inorder, l1, idx-1, postorder, l2, l2+idx-l1-
1);
        if(idx != r1)//有右子树
            root->right = build(inorder, idx+1, r1, postorder, r2-(r1-idx),
r2-1);
        return root;
    }

    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        int len = inorder.size();
        if(len == 0) return NULL;
        if(len == 1) return new TreeNode(inorder[0]);//只有一个节点，作为根节点返回
        init(inorder);
        return build(inorder, 0, len-1, postorder, 0, len-1);
    }
};
```

105.从前序与中序遍历序列构造二叉树（中等）

1. 题目描述

   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
```

2. 简单实现

和上题思路一样，只不过先序是先遍历了根节点

```
class Solution {
public:
    unordered_map<int, int> m;
    void init(vector<int>& v){
        for(int i = 0; i < v.size(); i++) m[v[i]] = i;
    }

    TreeNode* build(vector<int>& inorder, int l1, int r1, vector<int>& preorder,
int l2, int r2){
        if(l1 == r1) return new TreeNode(inorder[l1]);//只有一个节点，作为根节点返回
        TreeNode* root = new TreeNode(preorder[l2]);//前序遍历的第一个节点值为根节点值
        //找到根节点在中序遍历中的位置，其左方为左子树中序遍历结果，右方为右子树中序遍历结果
        int idx = m[preorder[l2]];
        if(idx != l1)//有左子树
            root->left = build(inorder, l1, idx-1, preorder, l2+1, l2+idx-l1);
        if(idx != r1)//有右子树
            root->right = build(inorder, idx+1, r1, preorder, r2-(r1-idx)+1, r2);
        return root;
    }
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        int len = inorder.size();
        if(len == 0) return NULL;
        if(len == 1) return new TreeNode(inorder[0]);//只有一个节点，作为根节点返回
        init(inorder);
        return build(inorder, 0, len-1, preorder, 0, len-1);
    }
};
```

116.填充每个节点的下一个右侧节点指针（中等）

1. 题目描述

给定一个**完美二叉树**，其所有叶子节点都在同一层，每个父节点都有两个子节点。二叉树定义如下：

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *next;
}
```

填充它的每个 next 指针，让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点，则将 next 指针设置为 `NULL`。初始状态下，所有 next 指针都被设置为 `NULL`。

示例：

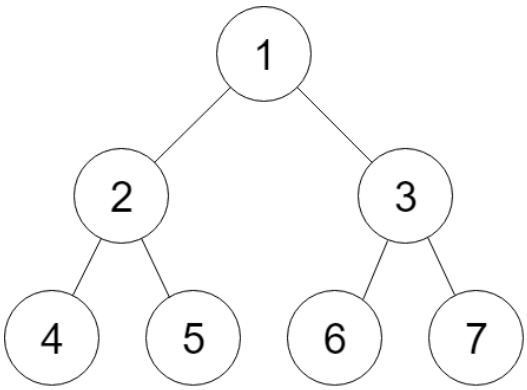


Figure A

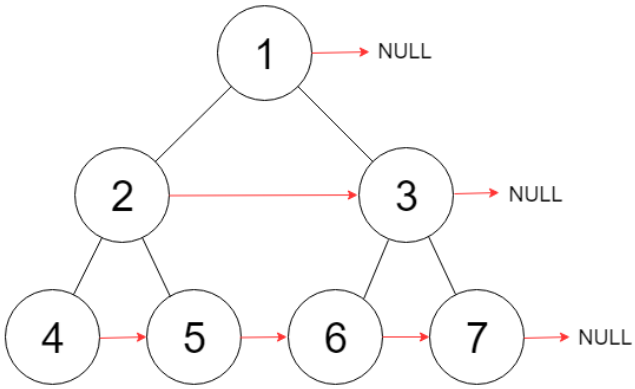


Figure B

```
输入：{"$id":"1","left":{"$id":"2","left":
{"$id":"3","left":null,"next":null,"right":null,"val":4},"next":null,"right":
{"$id":"4","left":null,"next":null,"right":null,"val":5},"val":2},"next":null,"right
":{"$id":"5","left":
{"$id":"6","left":null,"next":null,"right":null,"val":6},"next":null,"right":
{"$id":"7","left":null,"next":null,"right":null,"val":7},"val":3},"val":1}

输出：{"$id":"1","left":{"$id":"2","left":{"$id":"3","left":null,"next":
{"$id":"4","left":null,"next":{"$id":"5","left":null,"next":
{"$id":"6","left":null,"next":null,"right":null,"val":7},"right":null,"val":6},"righ
t":null,"val":5},"right":null,"val":4},"next":{"$id":"7","left":
{"$ref":"5"},"next":null,"right":{"$ref":"6"},"val":3},"right":
{"$ref":"4"},"val":2},"next":null,"right":{"$ref":"7"},"val":1}

解释：给定二叉树如图 A 所示，你的函数应该填充它的每个 next 指针，以指向其下一个右侧节点，如图 B 所示。
```

提示：

- 你只能使用常量级额外空间。
- 使用递归解题也符合要求，本题中递归程序占用的栈空间不算做额外的空间复杂度。

2. 简单实现

递归将左右子树分别连接好，再把左右子树之间连接好

```
/*
// Definition for a Node.
class Node {
public:
    int val;
    Node* left;
    Node* right;
    Node* next;
    Node() : val(0), left(NULL), right(NULL), next(NULL) {}
    Node(int _val) : val(_val), left(NULL), right(NULL), next(NULL) {}
    Node(int _val, Node* _left, Node* _right, Node* _next)
```

```
        : val(_val), left(_left), right(_right), next(_next) {}
};
*/
class Solution {
public:
    Node* connect(Node* root) {
        if (!root) return NULL;
        Node *i = root->left;
        Node *j = root->right;
        connect(i);
        connect(j);
        while(i){
            i->next = j;
            i=i->right;
            j=j->left;
        }
        return root;
    }
};
```

3. 其他方法

   层序遍历

```
class Solution {
public:
    Node* connect(Node* root) {
        if (!root) return NULL;
        queue<Node*> q;
        q.push(root);
        while(!q.empty()){
            int size = q.size();
            for(int i = 0; i < size; i++){
                Node* temp = q.front();
                q.pop();
                if(temp->left) q.push(temp->left);
                if(temp->right) q.push(temp->right);
                if(i != size-1)
                    temp->next = q.front();
            }
        }
        return root;
    }
};
```

4. 最优解法

   利用**完美二叉树**和next特性，可以逐层修改且不使用辅助空间队列

```
class Solution {
public:
    Node* connect(Node* root) {
        if (root == NULL)return NULL;
```

```
        if (root->left == NULL) return root;//无左子树也一定无右子树
        else  root->left->next = root->right;//第二层next构建好
        Node* cur = root->left->left;//指向待构建的那一层
        Node* pre = root->left;//指向cur上一层
        while (cur!=NULL) {
            while (pre!=NULL) {
                pre->left->next = pre->right;//左孩子的next是右孩子
                //处理右孩子的next
                if (pre->next != NULL)//pre右边还有兄弟
                    pre->right->next = pre->next->left;//右孩子的next为右兄弟的左孩子
                //else pre到达最右边，其右孩子的next为NULL，是初始值，无需再赋值
                pre = pre->next;//pre右移至下一个右兄弟
            }
            pre = cur;
            cur = cur->left;
        }
        return root;
    }
};
```

117. 填充每个节点的下一个右侧节点指针 II（中等）

1. 题目描述

   给定一个二叉树

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *next;
}
```

   填充它的每个 next 指针，让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点，则将 next 指针设置为 `NULL`。初始状态下，所有 next 指针都被设置为 `NULL`。

   进阶：

   - 你只能使用常量级额外空间。
   - 使用递归解题也符合要求，本题中递归程序占用的栈空间不算做额外的空间复杂度。

   示例：

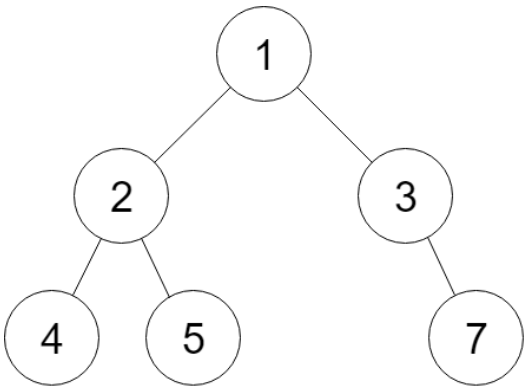


Figure A

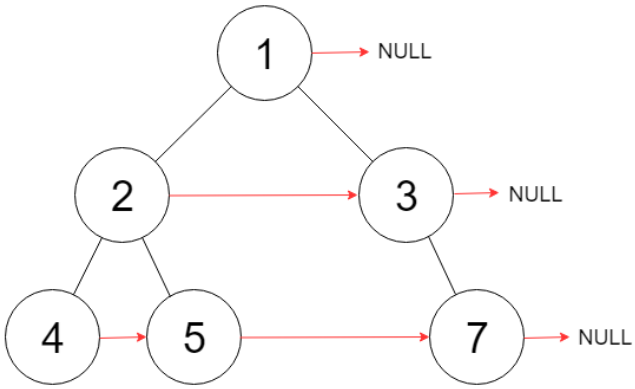


Figure B

```
输入：root = [1,2,3,4,5,null,7]
输出：[1,#,2,3,#,4,5,7,#]
解释：给定二叉树如图 A 所示，你的函数应该填充它的每个 next 指针，以指向其下一个右侧节点，如图 B 所示。
```

提示：

- 树中的节点数小于 `6000`
- `-100 <= node.val <= 100`

2. 简单实现

还可以使用层序遍历，但不能再使用上题中的其他方法

```
class Solution {
public:
    Node* connect(Node* root) {
        if (!root) return NULL;
        queue<Node*> q;
        q.push(root);
        while(!q.empty()){
            int size = q.size();
            for(int i = 0; i < size; i++){
                Node* temp = q.front();
                q.pop();
                if(temp->left) q.push(temp->left);
                if(temp->right) q.push(temp->right);
                if(i != size-1)
                    temp->next = q.front();
            }
        }
        return root;
    }
};
```

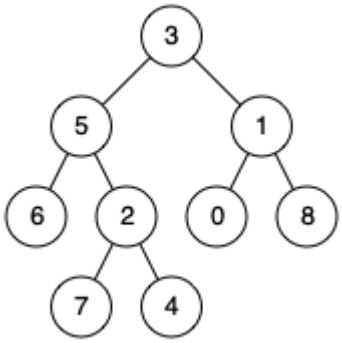
236.二叉树的最近公共祖先（中等）

1. 题目描述

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。
```

说明:

- 所有节点的值都是唯一的。
- p、q 为不同节点且均存在于给定的二叉树中。

2. 简单实现

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(!root || root == p || root == q) return root;
        TreeNode* l = lowestCommonAncestor(root->left, p, q);
        TreeNode* r = lowestCommonAncestor(root->right, p, q);
        if(l && r) return root;
        else if(l) return l;
        else return r;
    }
};
```

297.二叉树的序列化与反序列化（困难）

1. 题目描述

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：

```
    1
   / \
  2   3
     / \
    4   5
```

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。

2. 简单实现

序列化：带着null一起层序遍历

非序列化：不带Null层序构建

```
class Codec {
public:
    vector<string> split(string s){
        int l = 0, r = 0;
        vector<string> ans;
        while(r < s.size()){
            if(s[r] == ','){
                ans.push_back(s.substr(l, r-l));
                l = r+1;
                r++;
            }
            else r++;
        }
        ans.push_back(s.substr(l, r-l));
        return ans;
    }

    // Encodes a tree to a single string.
    string serialize(TreeNode* root) {
        if(!root) return "null";
        string ans = "";
        queue<TreeNode*> q;
        q.push(root);
        bool end = false;//标识是否为最后一层
        while(!end && !q.empty()){
            int size = q.size();
            end = true;
            for(int i = 0; i < size; i++){
                TreeNode* cur = q.front();
```

```
                q.pop();
                if(!cur)
                    ans += ",null";
                else{
                    ans += "," + to_string(cur->val);
                    q.push(cur->left);
                    q.push(cur->right);
                    if(cur->left || cur->right)//有非空子节点，不是最后一层
                        end = false;
                }
            }
        }
        return ans.substr(1,ans.size()-1);//记得去掉开头的','
    }

    // Decodes your encoded data to tree.
    TreeNode* deserialize(string data) {
        if(data == "null") return NULL;
        vector<string> nodes = split(data);
        TreeNode* root = new TreeNode(stoi(nodes[0]));
        queue<TreeNode*> q;
        q.push(root);
        int i = 1;
        while(i < nodes.size()){
            TreeNode* cur = q.front();
            q.pop();
            if(nodes[i] != "null"){
                cur->left = new TreeNode(stoi(nodes[i]));
                q.push(cur->left);
            }
            i++;
            if(nodes[i] != "null"){
                cur->right = new TreeNode(stoi(nodes[i]));
                q.push(cur->right);
            }
            i++;
        }
        return root;
    }
};
// Your Codec object will be instantiated and called as such:
// Codec codec;
// codec.deserialize(codec.serialize(root));
```