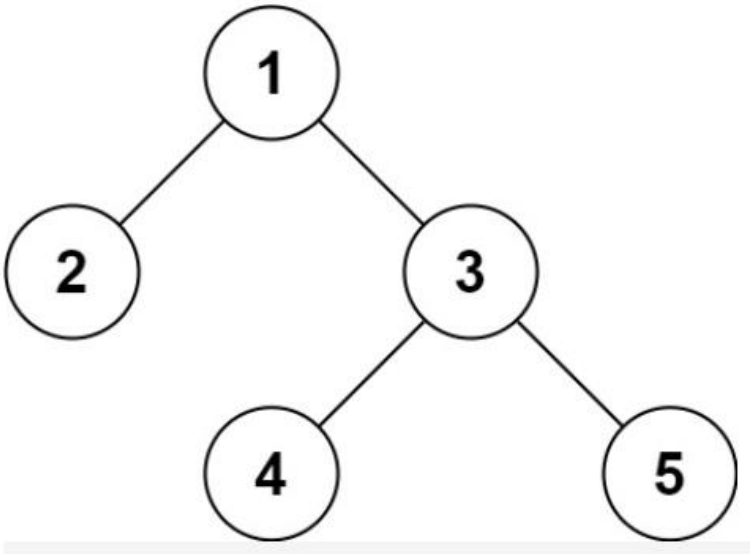
序列化二叉树

请实现两个函数，分别用来序列化和反序列化二叉树。

你需要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。

提示：输入输出格式与 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。

示例：



输入：`root = [1,2,3,null,null,4,5]`

输出：`[1,2,3,null,null,4,5]`

1. **特例处理：** 若 `root` 为空，则直接返回空列表 `"[]"` ；
2. **初始化：** 队列 `queue` （包含根节点 `root` ）；序列化列表 `res` ；
3. **层序遍历：** 当 `queue` 为空时跳出；
    1. 节点出队，记为 `node` ；
    2. 若 `node` 不为空：① 打印字符串 `node.val` ，② 将左、右子节点加入 `queue` ；
    3. 否则（若 `node` 为空）：打印字符串 `"null"` ；
4. **返回值：** 拼接列表，用 `','` 隔开，首尾添加中括号；

1. **特例处理：** 若 `data` 为空，直接返回 `null` ；
2. **初始化：** 序列化列表 `vals` （先去掉首尾中括号，再用逗号隔开），指针 `i = 1` ，根节点 `root` （值为 `vals[0]` ），队列 `queue` （包含 `root` ）；
3. **按层构建：** 当 `queue` 为空时跳出；
    1. 节点出队，记为 `node` ；
    2. 构建 `node` 的左子节点：`node.left` 的值为 `vals[i]` ，并将 `node.left` 入队；
    3. 执行 `i += 1` ；
    4. 构建 `node` 的右子节点：`node.left` 的值为 `vals[i]` ，并将 `node.left` 入队；
    5. 执行 `i += 1` ；
4. **返回值：** 返回根节点 `root` 即可；

```
/**
 * 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) {
        if(root==NULL)//特判
        return "";
        queue<TreeNode*> Q;
        string ans;
        Q.push(root);
        while(!Q.empty())
        {
            for(int i=Q.size();i>0;--i)
            {
                TreeNode* front=Q.front();
                Q.pop();
                if(front)
                {
                    ans+=to_string(front->val);
                    ans.push_back(',');
```

```
                    Q.push(front->left);//不用做判断，不管左子树或右子树是否为空都加入队列
                    Q.push(front->right);
                }
                else
                {
                    ans+="null";
                    ans.push_back(',');
                }
            }
        }
        ans.pop_back();
        return ans;
    }

    // Decodes your encoded data to tree.
    TreeNode* deserialize(string data) {
        if(data.empty())//特判
        return NULL;
        vector<string> v;
        queue<TreeNode*> Q;
        int pre=0;
        string temp;
        for(int i=0;i<=data.size();++i)
        {
            if(i==data.size()||data[i]==',')
            {
                temp=data.substr(pre,i-pre);//将每一个节点分割出来加入到v中
                pre=i+1;
            }
            if(!temp.empty())
            v.emplace_back(temp);
            temp.clear();
        }
        TreeNode* root=new TreeNode(stoi(v[0]));
        Q.push(root);
        int i=1;//i的作用类似于一个全局变量，在整个循环的过程中，i的位置始终是队列顶点的左孩子或右孩子
        while(!Q.empty())
        {
            TreeNode* front=Q.front();
            Q.pop();
            if(v[i]=="null")
            front->left=NULL;
```

```
            else
            {
                front->left=new TreeNode(stoi(v[i]));
                Q.push(front->left);
            }
            ++i;
            if(v[i]=="null")
            front->right=NULL;
            else
            {
                front->right=new TreeNode(stoi(v[i]));
                Q.push(front->right);
            }
            ++i;
        }
        return root;
    }
};

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