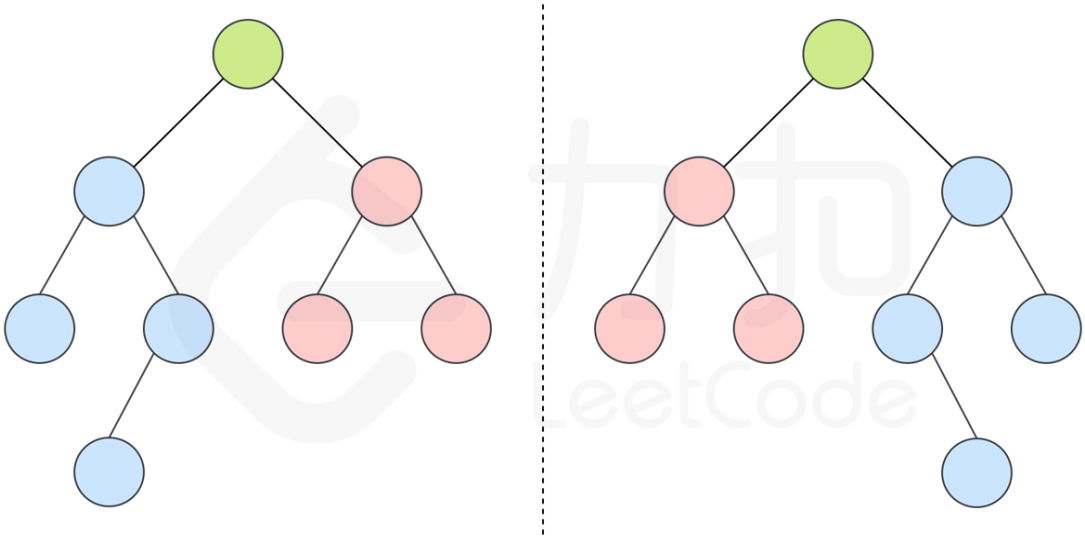
对称二叉树

递归

如果同时满足下面的条件，两个树互为镜像：

- 它们的两个根结点具有相同的值
- 每棵树的右子树都与另一棵树的左子树镜像对称



我们可以实现这样一个递归函数，通过**同步移动**两个指针的方法来遍历这棵树，p 指针和 q 指针一开始都指向这棵树的根，随后 p 右移时，q 左移，p 左移时，q 右移。每次检查当前 p 和 q 节点的值是否相等，如果相等再判断左右子树是否对称。

```
class Solution {
public:
    bool check(TreeNode *p, TreeNode *q) {
        if (!p && !q) return true;
        if (!p || !q) return false;
        return p->val == q->val && check(p->left, q->right) && check(p->right,
q->left);
    }

    bool isSymmetric(TreeNode* root) {
        return check(root, root);
    }
};
```

迭代

方法一中我们用递归的方法实现了对称性的判断，那么如何用迭代的方法实现呢？首先我们引入一个**队列**，这是把递归程序改写成迭代程序的常用方法。初始化时我们把根节点入队两次。每次提取两个结点并比较它们的值（队列中每两个连续的结点应该是相等的，而且它们的子树互为镜像），然后将两个结点的左右子结点按相反的顺序插入队列中。当队列为空时，或者我们检测到树不对称（即从队列中取出两个不相等的连续结点）时，该算法结束。

```
class Solution {
public:
    bool check(TreeNode *u, TreeNode *v) {
        queue <TreeNode*> q;
        q.push(u); q.push(v);
        while (!q.empty()) {
            u = q.front(); q.pop();
            v = q.front(); q.pop();
            if (!u && !v) continue;
            if ((!u || !v) || (u->val != v->val)) return false;

            q.push(u->left);
            q.push(v->right);

            q.push(u->right);
            q.push(v->left);
        }
        return true;
    }

    bool isSymmetric(TreeNode* root) {
        return check(root, root);
    }
};
```

二叉树的层序遍历

广度优先搜索

我们可以想到最朴素的方法是用一个二元组 $(node, level)$ 来表示状态，它表示某个节点和它所在的层数，每个新进队列的节点的 `level` 值都是父亲节点的 `level` 值加一。最后根据每个点的 `level` 对点进行分类，分类的时候我们可以利用哈希表，维护一个以 `level` 为键，对应节点值组成的数组为值，广度优先搜索结束以后按键 `level` 从小到大取出所有值，组成答案返回即可。

我们可以用一种巧妙的方法修改广度优先搜索：

- 首先根元素入队
- 当队列不为空的时候
    - 求当前队列的长度 s_i
    - 依次从队列中取 s_i 个元素进行拓展，然后进入下一次迭代

它和普通广度优先搜索的区别在于，普通广度优先搜索每次只取一个元素拓展，而这里每次取 s_i 个元素。在上述过程中的第 i 次迭代就得到了二叉树的第 i 层的 s_i 个元素。

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector <vector <int>> ret;
        if (!root) {
            return ret;
        }

        queue <TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
            int currentLevelSize = q.size();
            ret.push_back(vector <int> ());
            for (int i = 1; i <= currentLevelSize; ++i) {
                auto node = q.front(); q.pop();
                ret.back().push_back(node->val);
                if (node->left) q.push(node->left);
                if (node->right) q.push(node->right);
            }
        }

        return ret;
    }
};
```

二叉树的锯齿形层序遍历

为了满足题目要求的返回值为「先从左往右，再从右往左」交替输出的锯齿形，我们可以利用**双端队列**的数据结构来维护当前层节点值输出的顺序。

双端队列是一个可以在队列任意一端插入元素的队列。在广度优先搜索遍历当前层节点拓展下一层节点的时候我们仍然从左往右按顺序拓展，但是对当前层节点的存储我们维护一个变量 $isOrderLeft$ 记录是从左至右还是从右至左的：

- 如果从左至右，我们每次将被遍历到的元素插入至双端队列的末尾。
- 如果从右至左，我们每次将被遍历到的元素插入至双端队列的头部。

```
class Solution {
```

```
public:
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        vector<vector<int>> ans;
        if (!root) {
            return ans;
        }

        queue<TreeNode*> nodeQueue;
        nodeQueue.push(root);
        bool isOrderLeft = true;

        while (!nodeQueue.empty()) {
            deque<int> levelList;
            int size = nodeQueue.size();
            for (int i = 0; i < size; ++i) {
                auto node = nodeQueue.front();
                nodeQueue.pop();
                if (isOrderLeft) {
                    levelList.push_back(node->val);
                } else {
                    levelList.push_front(node->val);
                }
                if (node->left) {
                    nodeQueue.push(node->left);
                }
                if (node->right) {
                    nodeQueue.push(node->right);
                }
            }
            ans.emplace_back(vector<int>{levelList.begin(), levelList.end()});
            isOrderLeft = !isOrderLeft;
        }

        return ans;
    }
};
```

二叉树最大深度

深度优先搜索

如果我们知道了左子树和右子树的最大深度 l 和 r，那么该二叉树的最大深度即为：

$$max(l, r) + 1$$

而左子树和右子树的最大深度又可以以同样的方式进行计算。因此我们可以用「深度优先搜索」的方法来计算二叉树的最大深度。具体而言，在计算当前二叉树的最大深度时，可以先递归计算出其左子树和右子树的最大深度，然后在 $O(1)$ 时间内计算出当前二叉树的最大深度。递归在访问到空节点时退出。

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

广度优先搜索

每次拓展下一层的时候，我们需要将队列里的所有节点都拿出来进行拓展，这样能保证每次拓展完的时候队列里存放的是当前层的所有节点，即我们是一层一层地进行拓展，最后我们用一个变量 ans 来维护拓展的次数，该二叉树的最大深度即为 ans。

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if (root == nullptr) return 0;
        queue<TreeNode*> Q;
        Q.push(root);
        int ans = 0;
        while (!Q.empty()) {
            int sz = Q.size();
            while (sz > 0) {
                TreeNode* node = Q.front();Q.pop();
                if (node->left) Q.push(node->left);
                if (node->right) Q.push(node->right);
                sz -= 1;
            }
            ans += 1;
        }
        return ans;
    }
};
```

从前序与中序遍历序列构造二叉树

递归

对于任意一颗树而言，前序遍历的形式总是：

`[ 根节点, [左子树的前序遍历结果], [右子树的前序遍历结果] ]`

即根节点总是前序遍历中的第一个节点。而中序遍历的形式总是：

`[ [左子树的中序遍历结果], 根节点, [右子树的中序遍历结果] ]`

只要我们在中序遍历中定位到根节点，那么我们就可以分别知道左子树和右子树中的节点数目。由于同一颗子树的前序遍历和中序遍历的长度显然是相同的，因此我们就可以对应到前序遍历的结果中，对上述形式中的所有**左右括号**进行定位。

细节：在中序遍历中对根节点进行定位时，一种简单的方法是直接扫描整个中序遍历的结果并找出根节点，但这样做的时间复杂度较高。我们可以考虑使用哈希表来帮助我们快速地定位根节点。对于哈希映射中的每个键值对，键表示一个元素（节点的值），值表示其在中序遍历中的出现位置。在构造二叉树的过程之前，我们可以对中序遍历的列表进行一遍扫描，就可以构造出这个哈希映射。在此后构造二叉树的过程中，我们就只需要 $O(1)$ 的时间对根节点进行定位了。

```
class Solution {
private:
    unordered_map<int, int> index;

public:
    TreeNode* myBuildTree(const vector<int>& preorder, const vector<int>&
inorder,
```

```
                            int preorder_left, int preorder_right, int
inorder_left, int inorder_right) {

        if (preorder_left > preorder_right) {
            return nullptr;
        }

        // 前序遍历中的第一个节点就是根节点
        int preorder_root = preorder_left;
        // 在中序遍历中定位根节点
        int inorder_root = index[preorder[preorder_root]];

        // 先把根节点建立出来
        TreeNode* root = new TreeNode(preorder[preorder_root]);
        // 得到左子树中的节点数目
        int size_left_subtree = inorder_root - inorder_left;
        // 递归地构造左子树，并连接到根节点
        // 先序遍历中「从 左边界+1 开始的 size_left_subtree」个元素就对应了中序遍历中「从
左边界 开始到 根节点定位-1」的元素
        root->left = myBuildTree(preorder, inorder, preorder_left + 1,
preorder_left + size_left_subtree,
                                 inorder_left, inorder_root - 1);
        // 递归地构造右子树，并连接到根节点
        // 先序遍历中「从 左边界+1+左子树节点数目 开始到 右边界」的元素就对应了中序遍历中「从
根节点定位+1 到 右边界」的元素
        root->right = myBuildTree(preorder, inorder, preorder_left +
size_left_subtree + 1, preorder_right,
                                  inorder_root + 1, inorder_right);
        return root;
    }

    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        int n = preorder.size();
        // 构造哈希映射，帮助我们快速定位根节点
        for (int i = 0; i < n; ++i) {
            index[inorder[i]] = i;
        }
        return myBuildTree(preorder, inorder, 0, n - 1, 0, n - 1);
    }
};
```

迭代

迭代法是一种非常巧妙的实现方法。

对于前序遍历中的任意两个连续节点 u 和 v，根据前序遍历的流程，我们可以知道 u 和 v 只有两种可能的关系：

- v 是 u 的左儿子。这是因为在遍历到 u 之后，下一个遍历的节点就是 u 的左儿子，即 v；
- u 没有左儿子，并且 v 是 u 的某个祖先节点（或者 u 本身）的右儿子。如果 u 没有左儿子，那么下一个遍历的节点就是 u 的右儿子。如果 u 没有右儿子，我们就会向上回溯，直到遇到第一个有右儿子（且 u 不在它的右儿子的子树中）的节点 u_a，那么 v 就是 u_a 的右儿子。

以树

```
        3
       / \
      9  20
     /  /  \
    8  15   7
   / \
  5  10
 /
4
```

为例。它的前序遍历和中序遍历分别为：

```
preorder = [3, 9, 8, 5, 4, 10, 20, 15, 7]
inorder = [4, 5, 8, 10, 9, 3, 15, 20, 7]
```

我们用一个栈 `stack` 来维护**当前节点的所有还没有考虑过右儿子的祖先节点**，栈顶就是当前节点。也就是说，只有在栈中的节点才可能连接一个新的右儿子。同时，我们用一个指针 `index` 指向中序遍历的某个位置，初始值为 0。`index` 对应的节点是**当前节点不断往左走达到的最终节点**，这也是符合中序遍历的，它的作用在下面的过程中会有所体现。

首先我们将根节点 3 入栈，再初始化 `index` 所指向的节点为 4，随后对于前序遍历中的每个节点，我们依次判断它是栈顶节点的左儿子，还是栈中某个节点的右儿子。

- 我们遍历 9。9 一定是栈顶节点 3 的左儿子。我们使用反证法，假设 9 是 3 的右儿子，那么 3 没有左儿子，`index` 应该恰好指向 3，但实际上为 4，因此产生了矛盾。所以我们将 9 作为 3 的左儿子，并将 9 入栈。
  - `stack = [3, 9]`
  - `index -> inorder[0] = 4`
- 我们遍历 8，5 和 4。同理可得它们都是上一个节点（栈顶节点）的左儿子，所以它们会依次入栈。
  - `stack = [3, 9, 8, 5, 4]`
  - `index -> inorder[0] = 4`
- 我们遍历 10，这时情况就不一样了。我们发现 `index` 恰好指向当前的栈顶节点 4，也就是说 4 没有左儿子，那么 10 必须为栈中某个节点的右儿子。那么如何找到这个节点呢？栈中的节点的顺序和它们在前序遍历中出现的顺序是一致的，而且每一个节点的右儿子都还没有被遍历过，那么**这些节点的顺序和它们在中序遍历中出现的顺序一定是相反的。**

  > 这是因为栈中的任意两个相邻的节点，前者都是后者的某个祖先。并且我们知道，栈中的任意一个节点的右儿子还没有被遍历过，说明后者一定是前者左儿子的子树中的节点，那么后者就先于前者出现在中序遍历中。

  因此我们可以把 `index` 不断向右移动，并与栈顶节点进行比较。如果 `index` 对应的元素恰好等于栈顶节点，那么说明我们在中序遍历中找到了栈顶节点，所以将 `index` 增加 1 并弹出栈顶节点，直到 `index` 对应的元素不等于栈顶节点。按照这样的过程，我们弹出的最后一个节点 x 就是 10 的双亲节点，**这是因为 10 出现在了 x 与 x 在栈中的下一个节点的中序遍历之间**，因此 10 就是 x 的右儿子。

  回到我们的例子，我们会依次从栈顶弹出 4，5 和 8，并且将 `index` 向右移动了三次。我们将 10 作为最后弹出的节点 8 的右儿子，并将 10 入栈。
  - `stack=[3, 8, 10]`
  - `index -> inorder[3] = 10`
- 我们遍历 20。同理，`index` 恰好指向当前栈顶节点 10，那么我们会依次从栈顶弹出 10，9 和 3，并且将 `index` 向右移动了三次。我们将 20 作为最后弹出的节点 3 的右儿子，并将 20 入栈。

- `stack=[20]`
- `index -> inorder[6] = 15`

- 我们遍历 15，将 15 作为栈顶节点 20 的左儿子，并将 15 入栈。
  - `stack=[15]`
  - `index -> inorder[6] = 15`
- 我们遍历 7。`index` 恰好指向当前栈顶节点 15，那么我们会依次从栈顶弹出 15 和 20，并且将 `index` 向右移动了两次。我们将 7 作为最后弹出的节点 20 的右儿子，并将 7 入栈。
  - `stack=[7]`
  - `index -> inorder[8] = 7`

算法步骤：

1. 用一个栈和一个指针辅助进行二叉树的构造。初始时栈中存放了根节点（前序遍历的第一个节点），指针指向中序遍历的第一个节点；
2. 依次枚举前序遍历中除了第一个节点以外的每个节点。如果 `index` 恰好指向栈顶节点，那么我们不断地弹出栈顶节点并向右移动 `index`，并将当前节点作为最后一个弹出的节点的右儿子；如果 `index` 和栈顶节点不同，我们将当前节点作为栈顶节点的左儿子；
3. 无论是哪一种情况，我们最后都将当前的节点入栈。

```
class Solution {
public:
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        if (!preorder.size()) {
            return nullptr;
        }
        TreeNode* root = new TreeNode(preorder[0]);
        stack<TreeNode*> stk;
        stk.push(root);
        int inorderIndex = 0;
        for (int i = 1; i < preorder.size(); ++i) {
            int preorderVal = preorder[i];
            TreeNode* node = stk.top();
            if (node->val != inorder[inorderIndex]) {
                node->left = new TreeNode(preorderVal);
                stk.push(node->left);
            }
            else {
                while (!stk.empty() && stk.top()->val == inorder[inorderIndex])
{
                    node = stk.top();
                    stk.pop();
                    ++inorderIndex;
                }
                node->right = new TreeNode(preorderVal);
                stk.push(node->right);
            }
        }
        return root;
    }
};
```

从中序和后序遍历序列构造二叉树

递归

给定一棵二叉树，对其进行中序遍历与后序遍历的顺序为：

- 中序遍历的顺序是每次遍历左孩子，再遍历根节点，最后遍历右孩子。
- 后序遍历的顺序是每次遍历左孩子，再遍历右孩子，最后遍历根节点。

写成递归代码形式为：

```
// 中序遍历
void inorder(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    inorder(root->left);
    ans.push_back(root->val);
    inorder(root->right);
}
// 后序遍历
void postorder(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    postorder(root->left);
    postorder(root->right);
    ans.push_back(root->val);
}
```

我们可以发现**后序遍历的数组最后一个元素代表的即为根节点。**知道这个性质后，我们可以利用已知的根节点信息在中序遍历的数组中找到根节点所在的下标，然后根据其将中序遍历的数组分成左右两部分，左边部分即左子树，右边部分为右子树，针对每个部分可以用同样的方法继续递归下去构造。

算法步骤：

- 为了高效查找根节点元素在中序遍历数组中的下标，我们选择创建**哈希表**来存储中序序列，即建立一个（元素，下标）键值对的哈希表。
- 定义递归函数 `helper(in_left, in_right)` 表示当前递归到中序序列中当前子树的左右边界，递归入口为 `helper(0, n - 1)`：
    - 如果 `in_left > in_right`，说明子树为空，返回空节点。
    - 选择后序遍历的最后一个节点作为根节点。
    - 利用哈希表 $O(1)$ 查询当根节点在中序遍历中下标为 `index`。从 `in_left` 到 `index - 1` 属于左子树，从 `index + 1` 到 `in_right` 属于右子树。
    - 根据后序遍历逻辑，递归创建右子树 `helper(index + 1, in_right)` 和左子树 `helper(in_left, index - 1)`。注意这里有需要**先创建右子树，再创建左子树的依赖关系。**可以理解为在后序遍历的数组中整个数组是先存储左子树的节点，再存储右子树的节点，最后存储根节点，如果按每次选择「后序遍历的最后一个节点」为根节点，则先被构造出来的应该为右子树。
    - 返回根节点 `root`。

```
class Solution {
    int post_idx;
    unordered_map<int, int> idx_map;
public:
    TreeNode* helper(int in_left, int in_right, vector<int>& inorder,
vector<int>& postorder){
```

```
        // 如果这里没有节点构造二叉树了，就结束
        if (in_left > in_right) {
            return nullptr;
        }

        // 选择 post_idx 位置的元素作为当前子树根节点
        int root_val = postorder[post_idx];
        TreeNode* root = new TreeNode(root_val);

        // 根据 root 所在位置分成左右两棵子树
        int index = idx_map[root_val];

        // 下标减一
        post_idx--;
        // 构造右子树
        root->right = helper(index + 1, in_right, inorder, postorder);
        // 构造左子树
        root->left = helper(in_left, index - 1, inorder, postorder);
        return root;
    }
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        // 从后序遍历的最后一个元素开始
        post_idx = (int)postorder.size() - 1;

        // 建立（元素，下标）键值对的哈希表
        int idx = 0;
        for (auto& val : inorder) {
            idx_map[val] = idx++;
        }
        return helper(0, (int)inorder.size() - 1, inorder, postorder);
    }
};
```

迭代

迭代法是一种非常巧妙的实现方法。迭代法的实现基于以下两点发现：

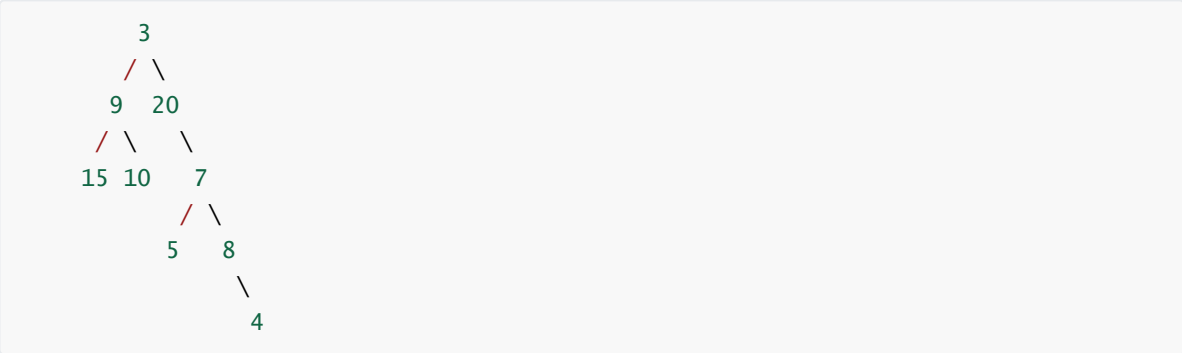
- 如果将中序遍历反序，则得到反向的中序遍历，即每次遍历右孩子，再遍历根节点，最后遍历左孩子。
- 如果将后序遍历反序，则得到反向的前序遍历，即每次遍历根节点，再遍历右孩子，最后遍历左孩子。

「反向」的意思是交换遍历左孩子和右孩子的顺序，即反向的遍历中，右孩子在左孩子之前被遍历。

对于后序遍历中的任意两个连续节点 u 和 v（在后序遍历中，u 在 v 的前面），根据后序遍历的流程，我们可以知道 u 和 v 只有两种可能的关系：

- u 是 v 的右儿子。这是因为在遍历到 u 之后，下一个遍历的节点就是 u 的双亲节点，即 v；
- v 没有右儿子，并且 u 是 v 的某个祖先节点（或者 v 本身）的左儿子。如果 v 没有右儿子，那么上一个遍历的节点就是 v 的左儿子。如果 v 没有左儿子，则从 v 开始向上遍历 v 的祖先节点，直到遇到一个有左儿子（且 v 不在它的左儿子的子树中）的节点 v_a，那么 u 就是 v_a 的左儿子。

以树



为例，它的中序遍历和后序遍历分别为:

```
inorder = [15, 9, 10, 3, 20, 5, 7, 8, 4]
postorder = [15, 10, 9, 5, 4, 8, 7, 20, 3]
```

我们用一个栈 `stack` 来维护**当前节点的所有还没有考虑过左儿子的祖先节点**，栈顶就是当前节点。也就是说，只有在栈中的节点才可能连接一个新的左儿子。同时，我们用一个指针 `index` 指向中序遍历的某个位置，初始值为 `n - 1`，其中 `n` 为数组的长度。`index` 对应的节点是**当前节点不断往右走达到的最终节点**，这也是符合反向中序遍历的，它的作用在下面的过程中会有所体现。

首先我们将根节点 `3` 入栈，再初始化 `index` 所指向的节点为 `4`，随后对于后序遍历中的每个节点，我们依次判断它是栈顶节点的右儿子，还是栈中某个节点的左儿子。

- 我们遍历 `20`。`20` 一定是栈顶节点 `3` 的右儿子。我们使用反证法，假设 `20` 是 `3` 的左儿子，因为 `20` 和 `3` 中间不存在其他的节点，那么 `3` 没有右儿子，`index` 应该恰好指向 `3`，但实际上为 `4`，因此产生了矛盾。所以我们将 `20` 作为 `3` 的右儿子，并将 `20` 入栈。
  - `stack = [3, 20]`
  - `index -> inorder[8] = 4`
- 我们遍历 `7`，`8` 和 `4`。同理可得它们都是上一个节点（栈顶节点）的右儿子，所以它们会依次入栈。
  - `stack = [3, 20, 7, 8, 4]`
  - `index -> inorder[8] = 4`
- 我们遍历 `5`，这时情况就不一样了。我们发现 `index` 恰好指向当前的栈顶节点 `4`，也就是说 `4` 没有右儿子，那么 `5` 必须为栈中某个节点的左儿子。那么如何找到这个节点呢？栈中的节点的顺序和它们在反向前序遍历中出现的顺序是一致的，而且每一个节点的左儿子都还没有被遍历过，那么**这些节点的顺序和它们在反向中序遍历中出现的顺序一定是相反的。**

  > 这是因为栈中的任意两个相邻的节点，前者都是后者的某个祖先。并且我们知道，栈中的任意一个节点的左儿子还没有被遍历过，说明后者一定是前者右儿子的子树中的节点，那么后者就先于前者出现在反向中序遍历中。

  因此我们可以把 `index` 不断向左移动，并与栈顶节点进行比较。如果 `index` 对应的元素恰好等于栈顶节点，那么说明我们在反向中序遍历中找到了栈顶节点，所以将 `index` 减少 1 并弹出栈顶节点，直到 `index` 对应的元素不等于栈顶节点。按照这样的过程，我们弹出的最后一个节点 `x` 就是 `5` 的双亲节点，这是因为 `5` 出现在了 `x` 与 `x` 在栈中的下一个节点的反向中序遍历之间，因此 `5` 就是 `x` 的左儿子。

  回到我们的例子，我们会依次从栈顶弹出 `4`，`8` 和 `7`，并且将 `index` 向左移动了三次。我们将 `5` 作为最后弹出的节点 `7` 的左儿子，并将 `5` 入栈。

  - `stack = [3, 20, 5]`
  - `index -> inorder[5] = 5`

- 我们遍历 `9`。同理，`index` 恰好指向当前栈顶节点 `5`，那么我们会依次从栈顶弹出 `5`，`20` 和 `3`，并且将 `index` 向左移动了三次。我们将 `9` 作为最后弹出的节点 `3` 的左儿子，并将 `9` 入栈。
  - `stack = [9]`
  - `index -> inorder[2] = 10`
- 我们遍历 `10`，将 `10` 作为栈顶节点 `9` 的右儿子，并将 `10` 入栈。
  - `stack = [9, 10]`
  - `index -> inorder[2] = 10`
- 我们遍历 `15`。`index` 恰好指向当前栈顶节点 `10`，那么我们会依次从栈顶弹出 `10` 和 `9`，并且将 `index` 向左移动了两次。我们将 `15` 作为最后弹出的节点 `9` 的左儿子，并将 `15` 入栈。
  - `stack = [15]`
  - `index -> inorder[0] = 15`

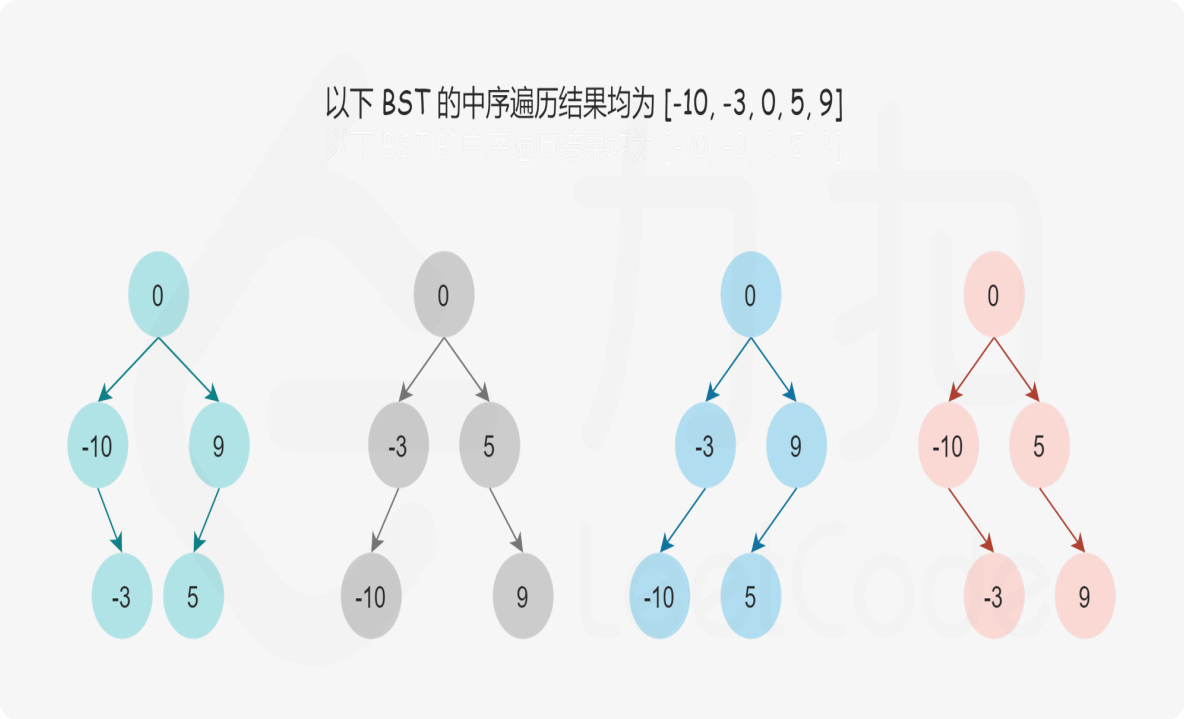
算法步骤：

- 我们用一个栈和一个指针辅助进行二叉树的构造。初始时栈中存放了根节点（后序遍历的最后一个节点），指针指向中序遍历的最后一个节点；
- 依次枚举后序遍历中除了第一个节点以外的每个节点。如果 `index` 恰好指向栈顶节点，那么我们不断地弹出栈顶节点并向左移动 `index`，并将当前节点作为最后一个弹出的节点的左儿子；如果 `index` 和栈顶节点不同，我们将当前节点作为栈顶节点的右儿子；
- 无论是哪一种情况，我们最后都将当前的节点入栈。

```
class Solution {
public:
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        if (postorder.size() == 0) {
            return nullptr;
        }
        auto root = new TreeNode(postorder[postorder.size() - 1]);
        auto s = stack<TreeNode*>();
        s.push(root);
        int inorderIndex = inorder.size() - 1;
        for (int i = int(postorder.size()) - 2; i >= 0; i--) {
            int postorderVal = postorder[i];
            auto node = s.top();
            if (node->val != inorder[inorderIndex]) {
                node->right = new TreeNode(postorderVal);
                s.push(node->right);
            } else {
                while (!s.empty() && s.top()->val == inorder[inorderIndex]) {
                    node = s.top();
                    s.pop();
                    inorderIndex--;
                }
                node->left = new TreeNode(postorderVal);
                s.push(node->left);
            }
        }
        return root;
    }
};
```

将有序数组转换为二叉搜索树

直观地看，我们可以选择中间数字作为二叉搜索树的根节点，这样分给左右子树的数字个数相同或只相差 11，可以使得树保持平衡。如果数组长度是奇数，则根节点的选择是唯一的，如果数组长度是偶数，则可以选择中间位置左边的数字作为根节点或者选择中间位置右边的数字作为根节点，选择不同的数字作为根节点则创建的平衡二叉搜索树也是不同的。



确定平衡二叉搜索树的根节点之后，其余的数字分别位于平衡二叉搜索树的左子树和右子树中，左子树和右子树分别也是平衡二叉搜索树，因此可以通过**递归的方式**创建平衡二叉搜索树。

在给定中序遍历序列数组的情况下，每一个子树中的数字在数组中一定是连续的，因此可以通过数组下标范围确定子树包含的数字，下标范围记为 $[left, right]$。对于整个中序遍历序列，下标范围从 $left = 0$ 到 $right = nums.\text{length} - 1$。当 $left > right$ 时，平衡二叉搜索树为空。

```
class Solution {
public:
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        return helper(nums, 0, nums.size() - 1);
    }

    TreeNode* helper(vector<int>& nums, int left, int right) {
        if (left > right) {
            return nullptr;
        }

        // 总是选择中间位置左边的数字作为根节点
        int mid = (left + right) / 2;

        TreeNode* root = new TreeNode(nums[mid]);
        root->left = helper(nums, left, mid - 1);
        root->right = helper(nums, mid + 1, right);
        return root;
    }
};
```

```
class Solution {
public:
    TreeNode* sortedArrayToBST(vector<int>& nums) {
```

```
        return helper(nums, 0, nums.size() - 1);
    }

    TreeNode* helper(vector<int>& nums, int left, int right) {
        if (left > right) {
            return nullptr;
        }

        // 总是选择中间位置右边的数字作为根节点
        int mid = (left + right + 1) / 2;

        TreeNode* root = new TreeNode(nums[mid]);
        root->left = helper(nums, left, mid - 1);
        root->right = helper(nums, mid + 1, right);
        return root;
    }
};
```

```
class Solution {
public:
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        return helper(nums, 0, nums.size() - 1);
    }

    TreeNode* helper(vector<int>& nums, int left, int right) {
        if (left > right) {
            return nullptr;
        }

        // 选择任意一个中间位置数字作为根节点
        int mid = (left + right + rand() % 2) / 2;

        TreeNode* root = new TreeNode(nums[mid]);
        root->left = helper(nums, left, mid - 1);
        root->right = helper(nums, mid + 1, right);
        return root;
    }
};
```

将有序链表转换为二叉搜索树

分治

设当前链表的左端点为 $left$，右端点 $right$，包含关系为**左闭右开**，即 $left$ 包含在链表中而 $right$ 不包含在链表中。我们希望快速地找出链表的中位数节点 mid。

> 为什么要设定「左闭右开」的关系？由于题目中给定的链表为单向链表，访问后继元素十分容易，但无法直接访问前驱元素。因此在找出链表的中位数节点 mid 之后，如果设定「左闭右开」的关系，我们就可以直接用 $(left, mid)$ 以及 $(mid.next, right)$ 来表示左右子树对应的列表了。并且，初始的列表也可以用 $(head, null)$ 方便地进行表示，其中 $null$ 表示空节点。

用**快慢指针法**找到中位数节点后，我们将其作为当前根节点的元素，并递归地构造其左侧部分的链表对应的左子树，以及右侧部分的链表对应的右子树。

```
class Solution {
public:
```

```
    ListNode* getMedian(ListNode* left, ListNode* right) {
        ListNode* fast = left;
        ListNode* slow = left;
        while (fast != right && fast->next != right) {
            fast = fast->next;
            fast = fast->next;
            slow = slow->next;
        }
        return slow;
    }

    TreeNode* buildTree(ListNode* left, ListNode* right) {
        if (left == right) {
            return nullptr;
        }
        ListNode* mid = getMedian(left, right);
        TreeNode* root = new TreeNode(mid->val);
        root->left = buildTree(left, mid);
        root->right = buildTree(mid->next, right);
        return root;
    }

    TreeNode* sortedListToBST(ListNode* head) {
        return buildTree(head, nullptr);
    }
};
```

- 时间复杂度：$O(n \log n)$，其中 n 为链表长度
- 空间复杂度： $O(\log n)$，平衡二叉树的高度为 $O(\log n)$。

分治+中序遍历优化

方法一的时间复杂度的瓶颈在于寻找中位数节点。由于**构造出的二叉搜索树的中序遍历结果就是链表本身**，因此我们可以将分治和中序遍历结合起来，减少时间复杂度。

具体地，设当前链表的左端点编号为 $left$，右端点编号为 $right$，包含关系为「双闭」，即 $left$ 和 $right$ 均包含在链表中。链表节点的编号为 $[0, n)$。中序遍历的顺序是**左子树 - 根节点 - 右子树**，那么在分治的过程中，我们不用急着找出链表的中位数节点，而是使用一个**占位节点**，等到中序遍历到该节点时，再填充它的值。

可以通过计算编号范围来进行中序遍历：

- 中位数节点对应的编号为 $mid = (left + right + 1)/2$;

  > 编号为 $(left + right)/2$ 的节点同样也可以是中位数节点。

- 左右子树对应的编号范围分别为 $[left, mid - 1]$ 和 $[mid + 1, right]$。

如果 $left > right$，那么遍历到的位置对应着一个空节点，否则对应着二叉搜索树中的一个节点。

这样一来，我们其实已经知道了这棵二叉搜索树的结构，并且题目给定了它的中序遍历结果，那么我们只要对其进行中序遍历，就可以还原出整棵二叉搜索树了。

```
class Solution {
public:
    int getLength(ListNode* head) {
        int ret = 0;
        for (; head != nullptr; ++ret, head = head->next);
```

```
        return ret;
    }

    TreeNode* buildTree(ListNode*& head, int left, int right) {
        if (left > right) {
            return nullptr;
        }
        int mid = (left + right + 1) / 2;
        TreeNode* root = new TreeNode();
        root->left = buildTree(head, left, mid - 1);
        root->val = head->val;
        head = head->next;
        root->right = buildTree(head, mid + 1, right);
        return root;
    }

    TreeNode* sortedListToBST(ListNode* head) {
        int length = getLength(head);
        return buildTree(head, 0, length - 1);
    }
};
```

- 时间复杂度：$O(n^2)$。最坏的情况，二叉树形成链式结构，高度为 $O(n)$，此时总时间复杂度为 $O(n^2)$。

平衡二叉树

自顶向下的递归

定义函数 $height$，用于计算二叉树中的任意一个节点 p 的高度：

$$height(p) = \begin{cases} 0 & p \text{ 是空节点} \\ max(height(p.left),\ height(p.right)) + 1 & p \text{ 是非空节点} \end{cases}$$

有了计算节点高度的函数，即可判断二叉树是否平衡。具体做法类似于二叉树的前序遍历，即对于当前遍历到的节点，首先计算左右子树的高度，如果左右子树的高度差不超过 1，再分别递归地遍历左右子节点，并判断左子树和右子树是否平衡。这是一个自顶向下的递归的过程。

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

    bool isBalanced(TreeNode* root) {
        if (root == NULL) {
            return true;
        } else {
            return abs(height(root->left) - height(root->right)) <= 1 &&
isBalanced(root->left) && isBalanced(root->right);
        }
    }
};
```

- 时间复杂度：$O(n^2)$

自底向上的递归

方法一由于是自顶向下递归，因此对于同一个节点，函数 `height` 会 **被重复调用**，导致时间复杂度较高。如果使用自底向上的做法，则对于每个节点，函数 `height` 只会被调用一次。

自底向上递归的做法类似于后序遍历，对于当前遍历到的节点，先递归地判断其左右子树是否平衡，再判断以当前节点为根的子树是否平衡。如果一棵子树是平衡的，则返回其高度（高度一定是非负整数），否则返回 -1。如果存在一棵子树不平衡，则整个二叉树一定不平衡。

```
class Solution {
public:
    int height(TreeNode* root) {
        if (root == NULL) {
            return 0;
        }
        int leftHeight = height(root->left);
        int rightHeight = height(root->right);
        if (leftHeight == -1 || rightHeight == -1 || abs(leftHeight -
rightHeight) > 1) {
            return -1;
        } else {
            return max(leftHeight, rightHeight) + 1;
        }
    }

    bool isBalanced(TreeNode* root) {
        return height(root) >= 0;
    }
};
```

- 时间复杂度：$O(n)$，其中 n 是二叉树中的节点个数。使用自底向上的递归，每个节点的计算高度和判断是否平衡都只需要处理一次，最坏情况下需要遍历二叉树中的所有节点，因此时间复杂度是 $O(n)$。

二叉树的最小深度

深度优先搜索

对于每一个非叶子节点，我们只需要分别计算其左右子树的最小叶子节点深度。这样就将一个大问题转化为了小问题，可以递归地解决该问题。

```
class Solution {
public:
    int minDepth(TreeNode *root) {
        if (root == nullptr) {
            return 0;
        }

        if (root->left == nullptr && root->right == nullptr) {
            return 1;
        }

        int min_depth = INT_MAX;
```

```
        if (root->left != nullptr) {
            min_depth = min(minDepth(root->left), min_depth);
        }
        if (root->right != nullptr) {
            min_depth = min(minDepth(root->right), min_depth);
        }

        return min_depth + 1;
    }
};
```

广度优先搜索

当我们找到一个叶子节点时，直接返回这个叶子节点的深度。广度优先搜索的性质保证了最先搜索到的叶子节点的深度一定最小。

```
class Solution {
public:
    int minDepth(TreeNode *root) {
        if (root == nullptr) {
            return 0;
        }

        queue<pair<TreeNode *, int> > que;
        que.emplace(root, 1);
        while (!que.empty()) {
            TreeNode *node = que.front().first;
            int depth = que.front().second;
            que.pop();
            if (node->left == nullptr && node->right == nullptr) {
                return depth;
            }
            if (node->left != nullptr) {
                que.emplace(node->left, depth + 1);
            }
            if (node->right != nullptr) {
                que.emplace(node->right, depth + 1);
            }
        }

        return 0;
    }
};
```

路径总和

题目

给你二叉树的根节点 `root` 和一个表示目标和的整数 `targetSum` ，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 `targetSum` 。

广度优先搜索

首先我们可以想到使用广度优先搜索的方式，记录从根节点到当前节点的路径和，以防止重复计算。

这样我们**使用两个队列**，分别存储将要遍历的节点，以及根节点到这些节点的路径和即可。

```
class Solution {
public:
    bool hasPathSum(TreeNode *root, int sum) {
        if (root == nullptr) {
            return false;
        }
        queue<TreeNode *> que_node;
        queue<int> que_val;
        que_node.push(root);
        que_val.push(root->val);
        while (!que_node.empty()) {
            TreeNode *now = que_node.front();
            int temp = que_val.front();
            que_node.pop();
            que_val.pop();
            if (now->left == nullptr && now->right == nullptr) {
                if (temp == sum) {
                    return true;
                }
                continue;
            }
            if (now->left != nullptr) {
                que_node.push(now->left);
                que_val.push(now->left->val + temp);
            }
            if (now->right != nullptr) {
                que_node.push(now->right);
                que_val.push(now->right->val + temp);
            }
        }
        return false;
    }
};
```

递归

假定从根节点到当前节点的值之和为 val，我们可以将这个大问题转化为一个小问题：是否存在从当前节点的子节点到叶子的路径，满足其路径和为 $sum - val$。

不难发现这满足递归的性质，若当前节点就是叶子节点，那么我们直接判断 sum 是否等于 val 即可（因为路径和已经确定，就是当前节点的值，我们只需要判断该路径和是否满足条件）。若当前节点不是叶子节点，我们只需要递归地询问它的子节点是否能满足条件即可。

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

路径总和2

深度优先搜索

枚举每一条从根节点到叶子节点的路径。当我们遍历到叶子节点，且此时路径和恰为目标和时，我们就找到了一条满足条件的路径。

```
class Solution {
public:
    vector<vector<int>> ret;
    vector<int> path;

    void dfs(TreeNode* root, int sum) {
        if (root == nullptr) {
            return;
        }
        path.emplace_back(root->val);
        sum -= root->val;
        if (root->left == nullptr && root->right == nullptr && sum == 0) {
            ret.emplace_back(path);
        }
        dfs(root->left, sum);
        dfs(root->right, sum);
        path.pop_back();
    }

    vector<vector<int>> pathSum(TreeNode* root, int sum) {
        dfs(root, sum);
        return ret;
    }
};
```

广度优先搜索

也可以采用广度优先搜索的方式，遍历这棵树。当我们遍历到叶子节点，且此时路径和恰为目标和时，我们就找到了一条满足条件的路径。

为了节省空间，我们使用哈希表记录树中的每一个节点的父节点。每次找到一个满足条件的节点，我们就从该节点出发不断向父节点迭代，即可还原出从根节点到当前节点的路径。

```
class Solution {
```

```
public:
    vector<vector<int>> ret;
    unordered_map<TreeNode*, TreeNode*> parent;

    void getPath(TreeNode* node) {
        vector<int> tmp;
        while (node != nullptr) {
            tmp.emplace_back(node->val);
            node = parent[node];
        }
        reverse(tmp.begin(), tmp.end());
        ret.emplace_back(tmp);
    }

    vector<vector<int>> pathSum(TreeNode* root, int sum) {
        if (root == nullptr) {
            return ret;
        }

        queue<TreeNode*> que_node;
        queue<int> que_sum;
        que_node.emplace(root);
        que_sum.emplace(0);

        while (!que_node.empty()) {
            TreeNode* node = que_node.front();
            que_node.pop();
            int rec = que_sum.front() + node->val;
            que_sum.pop();

            if (node->left == nullptr && node->right == nullptr) {
                if (rec == sum) {
                    getPath(node);
                }
            } else {
                if (node->left != nullptr) {
                    parent[node->left] = node;
                    que_node.emplace(node->left);
                    que_sum.emplace(rec);
                }
                if (node->right != nullptr) {
                    parent[node->right] = node;
                    que_node.emplace(node->right);
                    que_sum.emplace(rec);
                }
            }
        }

        return ret;
    }
};
```

二叉树展开为链表

题目

给你二叉树的根结点 `root`，请你将它展开为一个单链表：

- 展开后的单链表应该同样使用 `TreeNode`，其中 `right` 子指针指向链表中下一个结点，而左子指针始终为 `null` 。
- 展开后的单链表应该与二叉树先序遍历顺序相同。

前序遍历

将二叉树展开为单链表之后，单链表中的节点顺序即为二叉树的前序遍历访问各节点的顺序。因此，可以对二叉树进行前序遍历，获得各节点被访问到的顺序。由于将二叉树展开为链表之后会破坏二叉树的结构，因此在前序遍历结束之后更新每个节点的左右子节点的信息，将二叉树展开为单链表。

前序遍历可以通过递归或者迭代的方式实现。以下代码通过递归实现前序遍历：

```
class Solution {
public:
    void flatten(TreeNode* root) {
        vector<TreeNode*> l;
        preorderTraversal(root, l);
        int n = l.size();
        for (int i = 1; i < n; i++) {
            TreeNode *prev = l.at(i - 1), *curr = l.at(i);
            prev->left = nullptr;
            prev->right = curr;
        }
    }

    void preorderTraversal(TreeNode* root, vector<TreeNode*> &l) {
        if (root != NULL) {
            l.push_back(root);
            preorderTraversal(root->left, l);
            preorderTraversal(root->right, l);
        }
    }
};
```

以下代码通过迭代实现前序遍历：

```
class Solution {
public:
    void flatten(TreeNode* root) {
        auto v = vector<TreeNode*>();
        auto stk = stack<TreeNode*>();
        TreeNode *node = root;
        while (node != nullptr || !stk.empty()) {
            while (node != nullptr) {
                v.push_back(node);
                stk.push(node);
                node = node->left;
            }
            node = stk.top(); stk.pop();
            node = node->right;
        }
        int size = v.size();
```

```
        for (int i = 1; i < size; i++) {
            auto prev = v.at(i - 1), curr = v.at(i);
            prev->left = nullptr;
            prev->right = curr;
        }
    }
};
```

前序遍历和展开同步进行

使用方法一的前序遍历，由于将节点展开之后会破坏二叉树的结构而丢失子节点的信息，之所以会在破坏二叉树的结构之后丢失子节点的信息，是因为在对左子树进行遍历时，没有存储右子节点的信息，在遍历完左子树之后才获得右子节点的信息。只要对前序遍历进行修改，在遍历左子树之前就获得左右子节点的信息，并存入栈内，子节点的信息就不会丢失，就可以将前序遍历和展开为单链表同时进行。

修改后的前序遍历的具体做法是，每次从栈内弹出一个节点作为当前访问的节点，获得该节点的子节点，如果子节点不为空，则**依次将右子节点和左子节点压入栈内（注意入栈顺序）**。

展开为单链表的做法是，维护上一个访问的节点 `prev`，每次访问一个节点时，令当前访问的节点为 `curr`，将 `prev` 的左子节点设为 `null` 以及将 `prev` 的右子节点设为 `curr`，然后将 `curr` 赋值给 `prev`，进入下一个节点的访问，直到遍历结束。需要注意的是，初始时 `prev` 为 `null`，只有在 `prev` 不为 `null` 时才能对 `prev` 的左右子节点进行更新。

```
class Solution {
public:
    void flatten(TreeNode* root) {
        if (root == nullptr) {
            return;
        }
        auto stk = stack<TreeNode*>();
        stk.push(root);
        TreeNode *prev = nullptr;
        while (!stk.empty()) {
            TreeNode *curr = stk.top(); stk.pop();
            if (prev != nullptr) {
                prev->left = nullptr;
                prev->right = curr;
            }
            TreeNode *left = curr->left, *right = curr->right;
            if (right != nullptr) {
                stk.push(right);
            }
            if (left != nullptr) {
                stk.push(left);
            }
            prev = curr;
        }
    }
};
```

寻找前驱节点

前两种方法都借助前序遍历，前序遍历过程中需要使用栈存储节点。有没有空间复杂度是 $O(1)$ 的做法呢?

注意到前序遍历访问各节点的顺序是根节点、左子树、右子树。如果一个节点的左子节点为空，则该节点不需要进行展开操作。如果一个节点的左子节点不为空，则**该节点的左子树中的最后一个节点被访问之后，该节点的右子节点被访问**。该节点的左子树中最后一个被访问的节点是左子树中的最右边的节点，也是该节点的前驱节点。因此，问题转化成寻找当前节点的前驱节点。

具体做法是，对于当前节点，如果其左子节点不为空，则在其左子树中找到最右边的节点，作为前驱节点，将当前节点的右子节点赋给前驱节点的右子节点，然后将当前节点的左子节点赋给当前节点的右子节点，并将当前节点的左子节点设为空。对当前节点处理结束后，继续处理链表中的下一个节点，直到所有节点都处理结束。

```
class Solution {
public:
    void flatten(TreeNode* root) {
        TreeNode *curr = root;
        while (curr != nullptr) {
            if (curr->left != nullptr) {
                auto next = curr->left;
                auto predecessor = next;
                while (predecessor->right != nullptr) {
                    predecessor = predecessor->right;
                }
                predecessor->right = curr->right;
                curr->left = nullptr;
                curr->right = next;
            }
            curr = curr->right;
        }
    }
};
```

不同子序列

假设字符串 s 和 t 的长度分别为 m 和 n。如果 t 是 s 的子序列，则 s 的长度一定大于或等于 t 的长度，即只有当 $m \geq n$ 时，t 才可能是 s 的子序列。如果 $m < n$，则 t 一定不是 s 的子序列，因此直接返回 0。

创建二维数组 dp，其中 $dp[i][j]$ 表示在 $s[i:]$ 的子序列中 $t[j:]$ 出现的个数。

> 上述表述中，$s[i:]$ 表示 s 从下标 i 到末尾的子字符串，$t[j:]$ 表示 t 从下标 j 到末尾的子字符串

考虑边界情况：

- 当 $j = n$ 时，$t[j:]$ 为空字符串，由于空字符串是任何字符串的子序列，因此对任意 $0 \leq i \leq m$，有 $dp[i][n] = 1$；
- 当 $i = m$ 且 $j < n$ 时，$s[i:]$ 为空字符串，$t[j:]$ 为非空字符串，由于非空字符串不是空字符串的子序列，因此对任意 $0 \leq j < n$，有 $dp[m][j] = 0$。

当 $i < m$ 且 $j < n$ 时，考虑 $dp[i][j]$ 的计算：

- 当 $s[i] = t[j]$ 时，$dp[i][j]$ 由两部分组成：
    - 如果 $s[i]$ 和 $t[j]$ 匹配，则考虑 $t[j+1:]$ 作为 $s[i+1:]$ 的子序列，子序列数为 $dp[i+1][j+1]$；

- 如果 $s[i]$ 不和 $t[j]$ 匹配，则考虑 $t[j:]$ 作为 $s[i+1:]$ 的子序列，子序列数为 $dp[i+1][j]$。

因此，当 $s[i] = t[j]$ 时，有 $dp[i][j] = dp[i+1][j+1] + dp[i+1][j]$[j]。

- 当$s[i] \neq t[j]$ 时，$s[i]$ 不能和 $t[j]$ 匹配，因此只考虑 $t[j:]$ 作为 $s[i+1:]$ 的子序列，子序列数为 $dp[i+1][j]$。

  因此，当 $s[i] \neq t[j]$ 时，有 $dp[i][j] = dp[i+1][j]$[j]。

由此，可以得到状态转移方程：

$$dp[i][j] = \begin{cases} dp[i+1][j+1] + dp[i+1][j], & s[i] == t[j] \\ dp[i+1][j] & s[i] \neq [j] \end{cases}$$

最终计算得到 $dp[0][0]$ 即为在 s 的子序列中 t 出现的个数。

```
class Solution {
public:
    int numDistinct(string s, string t) {
        int m = s.length(), n = t.length();
        if (m < n) {
            return 0;
        }
        vector<vector<long>> dp(m + 1, vector<long>(n + 1));
        for (int i = 0; i <= m; i++) {
            dp[i][n] = 1;
        }
        for (int i = m - 1; i >= 0; i--) {
            char sChar = s.at(i);
            for (int j = n - 1; j >= 0; j--) {
                char tChar = t.at(j);
                if (sChar == tChar) {
                    dp[i][j] = dp[i + 1][j + 1] + dp[i + 1][j];
                } else {
                    dp[i][j] = dp[i + 1][j];
                }
            }
        }
        return dp[0][0];
    }
};
```

注意

上述代码无法通过测试，中间结果数可能超过了 $long\ long$ 的表示范围，因此，需要添加预处理代码：

```
unordered_map<char, int> um;
for (auto &c : s)
    ++um[c];

for (auto &c : t) {
    if (um.find(c) == um.end() || um[c] == 0)
        return 0;
    else
        --um[c];
}
```

填充每个节点的下一个右侧节点指针

层次遍历

使用已建立的 $next$ 指针

一棵树中，存在两种类型的 $next$ 指针：

1. 第一种情况是连接同一个父节点的两个子节点。它们可以通过同一个节点直接访问到，因此执行下面操作即可完成连接。

   ```
   node.left.next = node.right;
   ```

2. 第二种情况在不同父亲的子节点之间建立连接，这种情况不能直接连接。

   > 第 N 层节点之间建立 next 指针后，再建立第 $N+1$ 层节点的 next 指针。可以通过 next 指针访问同一层的所有节点，因此可以使用第 N 层的 next 指针，为第 $N+1$ 层节点建立 next 指针。

```
class Solution {
    public Node connect(Node root) {
        if (root == null) {
            return root;
        }

        // 从根节点开始
        Node leftmost = root;

        while (leftmost.left != null) {

            // 遍历这一层节点组织成的链表，为下一层的节点更新 next 指针
            Node head = leftmost;

            while (head != null) {

                // CONNECTION 1
                head.left.next = head.right;

                // CONNECTION 2
                if (head.next != null) {
                    head.right.next = head.next.left;
                }

                // 指针向后移动
                head = head.next;
            }

            // 去下一层的最左的节点
            leftmost = leftmost.left;
        }

        return root;
    }
}
```

填充每个节点的下一个右侧节点指针2

层次遍历

使用已建立的 $next$ 指针

```
class Solution {
public:
    void handle(Node* &last, Node* &p, Node* &nextStart) {
        if (last) {
            last->next = p;
        }
        if (!nextStart) {
            nextStart = p;
        }
        last = p;
    }

    Node* connect(Node* root) {
        if (!root) {
            return nullptr;
        }
        Node *start = root;
        while (start) {
            Node *last = nullptr, *nextStart = nullptr;
            for (Node *p = start; p != nullptr; p = p->next) {
                if (p->left) {
                    handle(last, p->left, nextStart);
                }
                if (p->right) {
                    handle(last, p->right, nextStart);
                }
            }
            start = nextStart;
        }
        return root;
    }
};
```

三角形的最小路径和

动态规划

用 $f[i][j]$ 表示从三角形顶部走到位置 (i, j) 的最小路径和。这里的位置 (i, j) 指的是三角形中第 i 行第 j 列（均从 0 开始编号）的位置。

由于每一步只能移动到下一行「相邻的节点」上，因此要想走到位置 (i, j)，上一步就只能在位置 $(i-1, j-1)$ 或者位置 $(i-1, j)$。我们在这两个位置中选择一个路径和较小的来进行转移，状态转移方程为：

$$f[i][j] = min(f[i-1][j-1], f[i-1][j]) + c[i][j]$$

其中 $c[i][j]$ 表示位置 (i, j) 对应的元素值。

注意第 i 行有 $i+1$ 个元素，它们对应的 j 的范围为 $[0, i]$。当 $j = 0$ 或 $j = i$ 时，上述状态转移方程中有一些项是没有意义的。例如当 $j = 0$ 时，$f[i-1][j-1]$ 没有意义，因此状态转移方程为：

$$f[i][0] = f[i-1][0] + c[i][0]$$

即当我们在第 i 行的最左侧时，我们只能从第 $i-1$ 行的最左侧移动过来。当 $j=i$ 时，$f[i-1][j]$ 没有意义，因此状态转移方程为：

$$f[i][i]=f[i-1][i-1]+c[i][i]$$

即当我们在第 i 行的最右侧时，我们只能从第 $i-1$ 行的最右侧移动过来。

最终的答案即为 $f[n-1][0]$到 $f[n-1][n-1]$ 中的最小值，其中 n 是三角形的行数。

```
class Solution {
public:
    int minimumTotal(vector<vector<int>>& triangle) {
        int n = triangle.size();
        vector<vector<int>> f(n, vector<int>(n));
        f[0][0] = triangle[0][0];
        for (int i = 1; i < n; ++i) {
            f[i][0] = f[i - 1][0] + triangle[i][0];
            for (int j = 1; j < i; ++j) {
                f[i][j] = min(f[i - 1][j - 1], f[i - 1][j]) + triangle[i][j];
            }
            f[i][i] = f[i - 1][i - 1] + triangle[i][i];
        }
        return *min_element(f[n - 1].begin(), f[n - 1].end());
    }
};
```

动态规划+空间优化

方法一中的状态转移方程为：

$$f[i][j]=\begin{cases}f[i][0]=f[i-1][0]+c[i][0], & j==0\\ f[i][i]=f[i-1][i-1]+c[i][i], & j==i\\ min(f[i-1][j-1],f[i-1][j])+c[i][j] & otherwise\end{cases}$$

可以发现，$f[i][j]$ 只与 $f[i-1][..]$ 有关，而与 $f[i-2][..]$ 及之前的状态无关，因此我们不必存储这些无关的状态。具体地，我们使用两个长度为 n 的一维数组进行转移，将 i 根据奇偶性映射到其中一个一维数组，那么 $i-1$ 就映射到了另一个一维数组。这样我们使用这两个一维数组，交替地进行状态转移。

```
class Solution {
public:
    int minimumTotal(vector<vector<int>>& triangle) {
        int n = triangle.size();
        vector<vector<int>> f(2, vector<int>(n));
        f[0][0] = triangle[0][0];
        for (int i = 1; i < n; ++i) {
            int curr = i % 2;
            int prev = 1 - curr;
            f[curr][0] = f[prev][0] + triangle[i][0];
            for (int j = 1; j < i; ++j) {
                f[curr][j] = min(f[prev][j - 1], f[prev][j]) + triangle[i][j];
            }
            f[curr][i] = f[prev][i - 1] + triangle[i][i];
        }
        return *min_element(f[(n - 1) % 2].begin(), f[(n - 1) % 2].end());
    }
};
```

上述方法的空间复杂度为 $O(n)$，使用了 $2n$ 的空间存储状态。我们还可以继续进行优化：

从 i 到 0 递减地枚举 j，这样我们只需要一个长度为 n 的一维数组 f，就可以完成状态转移。

> 当我们在计算位置 (i, j) 时，$f[j+1]$ 到 $f[i]$ 已经是第 i 行的值，而 $f[0]$ 到 $f[j]$仍然是第 $i-1$ 行的值。此时我们直接通过
>
> $$f[i] = min(f[j-1], f[j]) + c[i][j]$$
>
> 进行转移，恰好就是在 $(i-1, j-1)$ 和 $(i-1, j)$ 中进行选择。

```
class Solution {
public:
    int minimumTotal(vector<vector<int>>& triangle) {
        int n = triangle.size();
        vector<int> f(n);
        f[0] = triangle[0][0];
        for (int i = 1; i < n; ++i) {
            f[i] = f[i - 1] + triangle[i][i];
            for (int j = i - 1; j > 0; --j) {
                f[j] = min(f[j - 1], f[j]) + triangle[i][j];
            }
            f[0] += triangle[i][0];
        }
        return *min_element(f.begin(), f.end());
    }
};
```

买卖股票的最佳时机1

题目

给定一个数组 $prices$，它的第 i 个元素 $prices[i]$ 表示一支给定股票第 i 天的价格。

你只能选择 **某一天** 买入这只股票，并选择在 **未来的某一个不同的日子** 卖出该股票。设计一个算法来计算你所能获取的最大利润。

返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 0 。

一次遍历

遍历价格数组一遍，记录历史最低点，然后在每一天考虑这么一个问题：如果我是在历史最低点买进的，那么我今天卖出能赚多少钱？当考虑完所有天数之时，我们就得到了最好的答案。

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int inf = 1e9;
        int minprice = inf, maxprofit = 0;
        for (int price: prices) {
            maxprofit = max(maxprofit, price - minprice);
            minprice = min(price, minprice);
        }
        return maxprofit;
    }
};
```

买卖股票的最佳时机2

题目

给定一个数组 $prices$，其中 $prices[i]$ 是一支给定股票第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你可以 **尽可能地完成更多的交易（多次买卖一支股票）**。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

动态规划

考虑到「不能同时参与多笔交易」，因此每天交易结束后只可能存在手里有一支股票或者没有股票的状态。

定义状态 $dp[i][0]$ 表示第 i 天交易完后手里没有股票的最大利润，$dp[i][1]$ 表示第 i 天交易完后手里持有一支股票的最大利润（i 从 0 开始）。

考虑 $dp[i][0]$ 的转移方程，如果这一天交易完后手里没有股票，那么可能的转移状态为前一天已经没有股票，即 $dp[i-1][0]$，或者前一天结束的时候手里持有一支股票，即 $dp[i-1][1]$，这时候我们要将其卖出，并获得 $prices[i]$ 的收益。因此为了收益最大化，我们列出如下的转移方程：

$$dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])$$

再来考虑 $dp[i][1]$，按照同样的方式考虑转移状态，那么可能的转移状态为前一天已经持有一支股票，即 $dp[i-1][1]$，或者前一天结束时还没有股票，即 $dp[i-1][0]$，这时候我们要将其买入，并减少 $prices[i]$ 的收益。可以列出如下的转移方程：

$$dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i])$$

对于初始状态，根据状态定义我们可以知道第 0 天交易结束的时候 $dp[0][0] = 0$，$dp[0][1] = -prices[0]$。

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int n = prices.size();
        int dp[n][2];
        dp[0][0] = 0, dp[0][1] = -prices[0];
        for (int i = 1; i < n; ++i) {
            dp[i][0] = max(dp[i - 1][0], dp[i - 1][1] + prices[i]);
            dp[i][1] = max(dp[i - 1][1], dp[i - 1][0] - prices[i]);
        }
        return dp[n - 1][0];
    }
};
```

空间优化

注意到上面的状态转移方程中，每一天的状态只与前一天的状态有关，而与更早的状态都无关，因此我们不必存储这些无关的状态，只需要将 $dp[i-1][0]$ 和 $dp[i-1][1]$ 存放在两个变量中，通过它们计算出 $dp[i][0]$ 和 $dp[i][1]$ 并存回对应的变量，以便于第 $i+1$ 天的状态转移即可。

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int n = prices.size();
```

```
        int dp0 = 0, dp1 = -prices[0];
        for (int i = 1; i < n; ++i) {
            int newDp0 = max(dp0, dp1 + prices[i]);
            int newDp1 = max(dp1, dp0 - prices[i]);
            dp0 = newDp0;
            dp1 = newDp1;
        }
        return dp0;
    }
};
```

贪心

由于股票的购买没有限制，因此整个问题等价于寻找 x 个不相交的区间 $(l_i, r_i]$ 使得如下的等式最大化：

$$\sum_{i=1}^{x} a[r_i] - a[l_i]$$

其中 l_i 表示在第 l_i 天买入，r_i 表示在第 r_i 天卖出。

同时我们注意到对于 $(l_i, r_i]$ 这一个区间贡献的价值 $a[r_i] - a[l_i]$，其实等价于 $(l_i, l_i + 1], (l_i + 1, l_i + 2], \ldots, (r_i - 1, r_i]$ 这若干个区间长度为 1 的区间的价值和，即：

$$a[r_i] - a[l_i] = (a[r_i] - a[r_{i-1}]) + (a[r_{i-1}] - a[r_{i-2}]) + \cdots + (a[l_{i+1}] - a[l_i])$$

因此问题可以简化为找 x 个长度为 11 的区间 $(l_i, l_i + 1]$ 使得 $\sum_{i=1}^{x} a[l_i + 1] - a[l_i]$ 价值最大化。

贪心的角度考虑我们每次选择贡献大于 0 的区间即能使得答案最大化，因此最后答案为：

$$ans = \sum_{i=1}^{x} max\{0, a[i] - a[i-1]\}$$

其中 n 为数组的长度。

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int ans = 0;
        int n = prices.size();
        for (int i = 1; i < n; ++i) {
            ans += max(0, prices[i] - prices[i - 1]);
        }
        return ans;
    }
};
```

买卖股票的最佳时机3

题目

给定一个数组，它的第 i 个元素是一支给定的股票在第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你 **最多可以完成 两笔 交易。**

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

动态规划

由于我们最多可以完成两笔交易，因此在任意一天结束之后，我们会处于以下五个状态中的一种：

- 未进行过任何操作；
- 只进行过一次买操作；
- 进行了一次买操作和一次卖操作，即完成了一笔交易；
- 在完成了一笔交易的前提下，进行了第二次买操作；
- 完成了全部两笔交易。

由于第一个状态的利润显然为 0，因此我们可以不用将其记录。对于剩下的四个状态，我们分别将它们的最大利润记为 buy_1，$sell_1$，buy_2 以及 $sell_2$。

我们可以根据前两题写出状态转移方程：

$$\begin{cases} buy_1 = max\{buy_1, -prices[i]\} \\ sell_1 = max\{sell_1, buy_1 + prices[i]\} \\ buy_2 = max\{buy_2, sell_1 - prices[i]\} \\ sell_2 = max\{sell_2, buy_2 + prices[i]\} \end{cases}$$

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int n = prices.size();
        int buy1 = -prices[0], sell1 = 0;
        int buy2 = -prices[0], sell2 = 0;
        for (int i = 1; i < n; ++i) {
            buy1 = max(buy1, -prices[i]);
            sell1 = max(sell1, buy1 + prices[i]);
            buy2 = max(buy2, sell1 - prices[i]);
            sell2 = max(sell2, buy2 + prices[i]);
        }
        return sell2;
    }
};
```

最长连续序列

题目

给定一个未排序的整数数组 $nums$，找出数字连续的最长序列（不要求序列元素在原数组中连续）的长度。

请你设计并实现时间复杂度为 $O(n)$ 的算法解决此问题。

哈希

我们考虑枚举数组中的每个数 x，考虑以其为起点，不断尝试匹配 $x+1, x+2, \cdots$ 是否存在，假设最长匹配到了 $x+y$，那么以 x 为起点的最长连续序列即为 $x, x+1, x+2, \cdots, x+y$，其长度为 $y+1$，我们不断枚举并更新答案即可。

对于匹配的过程，暴力的方法是 $O(n)$ 遍历数组去看是否存在这个数，但其实更高效的方法是用一个哈希表存储数组中的数，这样查看一个数是否存在即能优化至 $O(1)$ 的时间复杂度。

仅仅是这样我们的算法时间复杂度最坏情况下还是会达到 $O(n^2)$（即外层需要枚举 $O(n)$ 个数，内层需要暴力匹配 $O(n)$ 次），无法满足题目的要求。但仔细分析这个过程，我们会发现其中执行了很多不必要的枚举，如果已知有一个 $x, x+1, x+2, \cdots, x+y$ 的连续序列，而我们却重新从 $x+1$，$x+2$ 或者是 $x+y$ 处开始尝试匹配，那么得到的结果肯定不会优于枚举 x 为起点的答案，因此我们在外层循环的时候碰到这种情况跳过即可。

那么怎么判断是否跳过呢？由于我们要枚举的数 x 一定是在数组中不存在前驱数 $x-1$ 的，不然按照上面的分析我们会从 $x-1$ 开始尝试匹配，因此我们每次在哈希表中检查是否存在 $x-1$ 即能判断是否需要跳过了。

增加了判断跳过的逻辑之后，时间复杂度是多少呢？外层循环需要 $O(n)$ 的时间复杂度，只有当一个数是连续序列的第一个数的情况下才会进入内层循环，然后在内层循环中匹配连续序列中的数，因此数组中的每个数只会进入内层循环一次。根据上述分析可知，总时间复杂度为 $O(n)$，符合题目要求。

```
class Solution {
public:
    int longestConsecutive(vector<int>& nums) {
        unordered_set<int> num_set;
        for (const int& num : nums) {
            num_set.insert(num);
        }

        int longestStreak = 0;

        for (const int& num : num_set) {
            if (!num_set.count(num - 1)) {
                int currentNum = num;
                int currentStreak = 1;

                while (num_set.count(currentNum + 1)) {
                    currentNum += 1;
                    currentStreak += 1;
                }

                longestStreak = max(longestStreak, currentStreak);
            }
        }

        return longestStreak;
    }
};
```

并查集

1. 初始化的时候先把数组里每个元素初始化为他的下一个数；
2. 并的时候找它能到达的最远的数字就可以了

```
class Solution {
public:
    unordered_map<int,int> a;
    int find(int x) {
        return a.count(x)? a[x]=find(a[x]) : x;
    }
    int longestConsecutive(vector<int> &nums) {
        for(auto i:nums)
            a[i] = i+1;
```

```
        int ans=0;
        for(auto i:nums){
            int y = find(i+1);
            ans = max(ans,y-i);
        }
        return ans;
    }
};
```

分割回文串1

回溯+动态规划预处理

当我们在判断 $s[i..j]$ 是否为回文串时，常规的方法是使用双指针分别指向 i 和 j，每次判断两个指针指向的字符是否相同，直到两个指针相遇。然而这种方法会产生重复计算

因此，我们可以将字符串 s 的每个子串 $s[i..j]$ 是否为回文串预处理出来，使用动态规划即可。

预处理完成之后，我们只需要 $O(1)$ 的时间就可以判断任意 $s[i..j]$ 是否为回文串了。

```
class Solution {
private:
    vector<vector<int>> f;
    vector<vector<string>> ret;
    vector<string> ans;
    int n;

public:
    void dfs(const string& s, int i) {
        if (i == n) {
            ret.push_back(ans);
            return;
        }
        for (int j = i; j < n; ++j) {
            if (f[i][j]) {
                ans.push_back(s.substr(i, j - i + 1));
                dfs(s, j + 1);
                ans.pop_back();
            }
        }
    }

    vector<vector<string>> partition(string s) {
        n = s.size();
        f.assign(n, vector<int>(n, true));

        for (int i = n - 1; i >= 0; --i) {
            for (int j = i + 1; j < n; ++j) {
                f[i][j] = (s[i] == s[j]) && f[i + 1][j - 1];
            }
        }

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

回溯+记忆化搜索

方法一中的动态规划预处理计算出了任意的 $s[i..j]$ 是否为回文串，我们也可以将这一步改为记忆化搜索。

```
class Solution {
private:
    vector<vector<int>> f;
    vector<vector<string>> ret;
    vector<string> ans;
    int n;

public:
    void dfs(const string& s, int i) {
        if (i == n) {
            ret.push_back(ans);
            return;
        }
        for (int j = i; j < n; ++j) {
            if (isPalindrome(s, i, j) == 1) {
                ans.push_back(s.substr(i, j - i + 1));
                dfs(s, j + 1);
                ans.pop_back();
            }
        }
    }

    // 记忆化搜索中，f[i][j] = 0 表示未搜索，1 表示是回文串，-1 表示不是回文串
    int isPalindrome(const string& s, int i, int j) {
        if (f[i][j]) {
            return f[i][j];
        }
        if (i >= j) {
            return f[i][j] = 1;
        }
        return f[i][j] = (s[i] == s[j] ? isPalindrome(s, i + 1, j - 1) : -1);
    }

    vector<vector<string>> partition(string s) {
        n = s.size();
        f.assign(n, vector<int>(n));

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

分割回文串2

动态规划

设 $f[i]$ 表示字符串的前缀 $s[0..i]$ 的最少分割次数。要想得出 $f[i]$ 的值，我们可以考虑枚举 $s[0..i]$ 分割出的最后一个回文串，这样我们就可以写出状态转移方程：

$$f[i] = \min_{0 \le j < i} \{f[j]\} + 1$$

其中 $s[j+1,i]$ 是一个回文串。即我们枚举最后一个回文串的起始位置 $j+1$，保证 $s[j+1..i]$ 是一个回文串，那么 $f[i]$ 就可以从 $f[j]$ 转移而来，附加 1 次额外的分割次数。

我们可以使用与上一题中相同的预处理方法，将字符串 s 的每个子串是否为回文串预先计算出来。

这样一来，我们只需要 $O(1)$ 的时间就可以判断任意 $s[i..j]$ 是否为回文串了。通过动态规划计算出所有的 f 值之后，最终的答案即为 $f[n-1]$，其中 n 是字符串 s 的长度。

```
class Solution {
public:
    int minCut(string s) {
        int n = s.size();
        vector<vector<int>> g(n, vector<int>(n, true));

        for (int i = n - 1; i >= 0; --i) {
            for (int j = i + 1; j < n; ++j) {
                g[i][j] = (s[i] == s[j]) && g[i + 1][j - 1];
            }
        }

        vector<int> f(n, INT_MAX);
        for (int i = 0; i < n; ++i) {
            if (g[0][i]) {
                f[i] = 0;
            }
            else {
                for (int j = 0; j < i; ++j) {
                    if (g[j + 1][i]) {
                        f[i] = min(f[i], f[j] + 1);
                    }
                }
            }
        }

        return f[n - 1];
    }
};
```

后序遍历

```
class Solution {
public:
    vector<int> postorderTraversal(TreeNode *root) {
        vector<int> res;
        if (root == nullptr) {
            return res;
        }

        stack<TreeNode *> stk;
        TreeNode *prev = nullptr;
        while (root != nullptr || !stk.empty()) {
            while (root != nullptr) {
                stk.emplace(root);
                root = root->left;
            }
            root = stk.top();
            stk.pop();
```

```
            if (root->right == nullptr || root->right == prev) {
                res.emplace_back(root->val);
                prev = root;
                root = nullptr;
            } else {
                stk.emplace(root);
                root = root->right;
            }
        }
        return res;
    }
};
```