方法一：递归

二叉树的前序遍历顺序是：根节点、左子树、右子树，每个子树的遍历顺序同样满足前序遍历顺序。

二叉树的中序遍历顺序是：左子树、根节点、右子树，每个子树的遍历顺序同样满足中序遍历顺序。

前序遍历的第一个节点是根节点，只要找到根节点在中序遍历中的位置，在根节点之前被访问的节点都位于左子树，在根节点之后被访问的节点都位于右子树，由此可知左子树和右子树分别有多少个节点。

由于树中的节点数量与遍历方式无关，通过中序遍历得知左子树和右子树的节点数量之后，可以根据节点数量得到前序遍历中的左子树和右子树的分界，因此可以进一步得到左子树和右子树各自的前序遍历和中序遍历，可以通过递归的方式，重建左子树和右子树，然后重建整个二叉树。

使用一个 Map 存储中序遍历的每个元素及其对应的下标，目的是为了快速获得一个元素在中序遍历中的位置。调用递归方法，对于前序遍历和中序遍历，下标范围都是从 0 到 n-1，其中 n 是二叉树节点个数。

递归方法的基准情形有两个：判断前序遍历的下标范围的开始和结束，若开始大于结束，则当前的二叉树中没有节点，返回空值 null。若开始等于结束，则当前的二叉树中恰好有一个节点，根据节点值创建该节点作为根节点并返回。

若开始小于结束，则当前的二叉树中有多个节点。在中序遍历中得到根节点的位置，从而得到左子树和右子树各自的下标范围和节点数量，知道节点数量后，在前序遍历中即可得到左子树和右子树各自的下标范围，然后递归重建左子树和右子树，并将左右子树的根节点分别作为当前根节点的左右子节点。

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode buildTree(int[] preorder, int[] inorder) {

if (preorder == null || preorder.length == 0) {

return null;

}

Map<Integer, Integer> indexMap = new HashMap<Integer, Integer>();

int length = preorder.length;

for (int i = 0; i < length; i++) {

indexMap.put(inorder[i], i);

}

TreeNode root = buildTree(preorder, 0, length - 1, inorder, 0, length - 1, indexMap);

return root;

}

public TreeNode buildTree(int[] preorder, int preorderStart, int preorderEnd, int[] inorder, int inorderStart, int inorderEnd, Map<Integer, Integer> indexMap) {

if (preorderStart > preorderEnd) {

return null;

}

int rootVal = preorder[preorderStart];

TreeNode root = new TreeNode(rootVal);

if (preorderStart == preorderEnd) {

return root;

} else {

int rootIndex = indexMap.get(rootVal);

int leftNodes = rootIndex - inorderStart, rightNodes = inorderEnd - rootIndex;

TreeNode leftSubtree = buildTree(preorder, preorderStart + 1, preorderStart + leftNodes, inorder, inorderStart, rootIndex - 1, indexMap);

TreeNode rightSubtree = buildTree(preorder, preorderEnd - rightNodes + 1, preorderEnd, inorder, rootIndex + 1, inorderEnd, indexMap);

root.left = leftSubtree;

root.right = rightSubtree;

return root;

}

}

}

复杂度分析

时间复杂度：O(n)。对于每个节点都有创建过程以及根据左右子树重建过程。

空间复杂度：O(n)。存储整棵树的开销。

### 方法二：迭代

例如要重建的是如下二叉树。

3

/ \

9 20

/ / \

8 15 7

/ \

5 10

/

4

其前序遍历和中序遍历如下。

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

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

前序遍历的第一个元素 3 是根节点，第二个元素 9 可能位于左子树或者右子树，需要通过中序遍历判断。

中序遍历的第一个元素是 4 ，不是根节点 3，说明 9 位于左子树，因为根节点不是中序遍历中的第一个节点。同理，前序遍历的后几个元素 8、5、4 也都位于左子树，且每个节点都是其上一个节点的左子节点。

前序遍历到元素 4，和中序遍历的第一个元素相等，说明前序遍历的下一个元素 10 位于右子树。那么 10 位于哪个元素的右子树？从前序遍历看，10 可能位于 4、5、8、9、3 这些元素中任何一个元素的右子树。从中序遍历看，10 在 8 的后面，因此 10 位于 8 的右子树。把前序遍历的顺序反转，则在 10 之前的元素是 4、5、8、9、3，其中 8 是最后一次相等的节点，因此前序遍历的下一个元素位于中序遍历中最后一次相等的节点的右子树。

同理可知，20 位于 3 的右子树，15 和 7 分别是 20 的左右子节点。

根据上述例子和分析，可以使用栈保存遍历过的节点。初始时令中序遍历的指针指向第一个元素，遍历前序遍历的数组，如果前序遍历的元素不等于中序遍历的指针指向的元素，则前序遍历的元素为上一个节点的左子节点。如果前序遍历的元素等于中序遍历的指针指向的元素，则正向遍历中序遍历的元素同时反向遍历前序遍历的元素，找到最后一次相等的元素，将前序遍历的下一个节点作为最后一次相等的元素的右子节点。其中，反向遍历前序遍历的元素可通过栈的弹出元素实现。

使用前序遍历的第一个元素创建根节点。

创建一个栈，将根节点压入栈内。

初始化中序遍历下标为 0。

遍历前序遍历的每个元素，判断其上一个元素（即栈顶元素）是否等于中序遍历下标指向的元素。

若上一个元素不等于中序遍历下标指向的元素，则将当前元素作为其上一个元素的左子节点，并将当前元素压入栈内。

若上一个元素等于中序遍历下标指向的元素，则从栈内弹出一个元素，同时令中序遍历下标指向下一个元素，之后继续判断栈顶元素是否等于中序遍历下标指向的元素，若相等则重复该操作，直至栈为空或者元素不相等。然后令当前元素为最后一个想等元素的右节点。

遍历结束，返回根节点。

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public TreeNode buildTree(int[] preorder, int[] inorder) {

if (preorder == null || preorder.length == 0) {

return null;

}

TreeNode root = new TreeNode(preorder[0]);

int length = preorder.length;

Stack<TreeNode> stack = new Stack<TreeNode>();

stack.push(root);

int inorderIndex = 0;

for (int i = 1; i < length; i++) {

int preorderVal = preorder[i];

TreeNode node = stack.peek();

if (node.val != inorder[inorderIndex]) {

node.left = new TreeNode(preorderVal);

stack.push(node.left);

} else {

while (!stack.isEmpty() && stack.peek().val == inorder[inorderIndex]) {

node = stack.pop();

inorderIndex++;

}

node.right = new TreeNode(preorderVal);

stack.push(node.right);

}

}

return root;

}

}

* 时间复杂度：O(n)。前序遍历和后序遍历都被遍历。
* 空间复杂度：O(n)。额外使用栈存储已经遍历过的节点。