# 题目

将一个 二叉搜索树 就地转化为一个 已排序的双向循环链表 。

对于双向循环列表，你可以将左右孩子指针作为双向循环链表的前驱和后继指针，第一个节点的前驱是最后一个节点，最后一个节点的后继是第一个节点。

特别地，我们希望可以 原地 完成转换操作。当转化完成以后，树中节点的左指针需要指向前驱，树中节点的右指针需要指向后继。还需要返回链表中最小元素的指针。

示例 1：

输入：root = [4,2,5,1,3]

图片包含 图示

描述已自动生成

输出：[1,2,3,4,5]

解释：下图显示了转化后的二叉搜索树，实线表示后继关系，虚线表示前驱关系。

图示

描述已自动生成

示例 2：

输入：root = [2,1,3]

输出：[1,2,3]

提示：

树中节点的数量在范围 [0, 2000] 中

-1000 <= Node.val <= 1000

树中的所有值都是 独一无二 的

# 分析

要解决将二叉搜索树（BST）就地转化为已排序的双向循环链表的问题，我们可以利用BST的特性：中序遍历BST会得到一个递增的序列。通过中序遍历，我们可以将节点按顺序连接成双向链表，最后再处理循环关系。

思路分析

1、中序遍历的特性：BST的中序遍历（左→根→右）结果是一个递增的序列，这与目标双向链表的排序要求一致。

2、双向链表的构建：在中序遍历过程中，通过记录前一个访问的节点（prev），将当前节点（curr）与prev连接：

- prev->right = curr（prev的后继是curr）

- curr->left = prev（curr的前驱是prev）

3、循环链表的处理：遍历结束后，需要将链表的头节点（最小元素）和尾节点（最大元素）连接起来，形成循环：

- 头节点的左指针指向尾节点

- 尾节点的右指针指向头节点

代码：

/\*

// Definition for a Node.

class Node {

public:

int val;

Node\* left;

Node\* right;

Node() {}

Node(int \_val) {

val = \_val;

left = NULL;

right = NULL;

}

Node(int \_val, Node\* \_left, Node\* \_right) {

val = \_val;

left = \_left;

right = \_right;

}

};

\*/

class Solution {

private:

Node\* head; // 双向链表的头节点（最小元素）

Node\* prev; // 中序遍历中当前节点的前一个节点

// 中序遍历函数：递归处理左子树→连接当前节点→递归处理右子树

void inorder(Node\* curr) {

if (curr == nullptr) return;

// 递归处理左子树

inorder(curr->left);

// 连接当前节点与前一个节点

if (prev == nullptr) {

// 第一个节点（最小元素），记录为头节点

head = curr;

} else {

// 前一个节点的右指针指向当前节点

prev->right = curr;

// 当前节点的左指针指向前一个节点

curr->left = prev;

}

// 更新prev为当前节点，供下一个节点连接

prev = curr;

// 递归处理右子树

inorder(curr->right);

}

public:

Node\* treeToDoublyList(Node\* root) {

if (root == nullptr) return nullptr;

head = nullptr;

prev = nullptr;

// 中序遍历构建双向链表

inorder(root);

// 处理循环：头节点的左指针指向尾节点（prev），尾节点的右指针指向头节点

head->left = prev;

prev->right = head;

return head;

}

};

代码解释：

1、中序遍历构建双向链表：

- 递归遍历左子树，直到找到最左节点（BST中的最小元素），该节点将作为双向链表的头节点（head）。

- 遍历过程中，prev始终记录上一个访问的节点。对于每个当前节点curr：

若prev为nullptr，说明curr是第一个节点，将其设为head。

否则，通过prev->right = curr和curr->left = prev建立双向连接。

每次处理完当前节点后，更新prev为curr，继续遍历右子树。

2、形成循环链表：

- 中序遍历结束后，prev指向链表的最后一个节点（BST中的最大元素）。

- 将head->left指向prev，prev->right指向head，使链表首尾相连，形成循环。

复杂度分析

时间复杂度：O(n)，其中n是BST的节点数。中序遍历会访问每个节点恰好一次，连接操作是常数时间。

空间复杂度：O(h)，其中h是BST的高度。递归调用栈的深度取决于树的高度，最坏情况下（树为链状）h = n，平均情况下（平衡树）h = log n。

该方法充分利用BST中序遍历的有序性，在遍历过程中就地构建双向链表，最后通过简单的首尾连接形成循环，实现了高效的转换。