# 题目

实现二叉搜索树（BST）的中序遍历迭代器 BSTIterator 类：

BSTIterator(TreeNode root) 初始化 BSTIterator 类的实例。二叉搜索树的根节点 root 作为构造函数的参数传入。内部指针使用一个不存在于树中且小于树中任意值的数值来初始化。

boolean hasNext() 如果当前指针在中序遍历序列中，存在右侧数值，返回 true ，否则返回 false 。

int next() 将指针在中序遍历序列中向右移动，然后返回移动后指针所指数值。

boolean hasPrev() 如果当前指针在中序遍历序列中，存在左侧数值，返回 true ，否则返回 false 。

int prev() 将指针在中序遍历序列中向左移动，然后返回移动后指针所指数值。

注意，虽然我们使用树中不存在的最小值来初始化内部指针，第一次调用 next() 需要返回二叉搜索树中最小的元素。

你可以假设 next() 和 prev() 的调用总是有效的。即，当 next()/prev() 被调用的时候，在中序遍历序列中一定存在下一个/上一个元素。

进阶：你可以不提前遍历树中的值来解决问题吗？

示例 1:

形状

描述已自动生成

输入

["BSTIterator", "next", "next", "prev", "next", "hasNext", "next", "next", "next", "hasNext", "hasPrev", "prev", "prev"]

[[[7, 3, 15, null, null, 9, 20]], [null], [null], [null], [null], [null], [null], [null], [null], [null], [null], [null], [null]]

输出

[null, 3, 7, 3, 7, true, 9, 15, 20, false, true, 15, 9]

解释

// 划线的元素表示指针当前的位置。

BSTIterator bSTIterator = new BSTIterator([7, 3, 15, null, null, 9, 20]); // 当前状态为 <u> </u> [3, 7, 9, 15, 20]

bSTIterator.next(); // 状态变为 [<u>3</u>, 7, 9, 15, 20], 返回 3

bSTIterator.next(); // 状态变为 [3, <u>7</u>, 9, 15, 20], 返回 7

bSTIterator.prev(); // 状态变为 [<u>3</u>, 7, 9, 15, 20], 返回 3

bSTIterator.next(); // 状态变为 [3, <u>7</u>, 9, 15, 20], 返回 7

bSTIterator.hasNext(); // 返回 true

bSTIterator.next(); // 状态变为 [3, 7, <u>9</u>, 15, 20], 返回 9

bSTIterator.next(); // 状态变为 [3, 7, 9, <u>15</u>, 20], 返回 15

bSTIterator.next(); // 状态变为 [3, 7, 9, 15, <u>20</u>], 返回 20

bSTIterator.hasNext(); // 返回 false

bSTIterator.hasPrev(); // 返回 true

bSTIterator.prev(); // 状态变为 [3, 7, 9, <u>15</u>, 20], 返回 15

bSTIterator.prev(); // 状态变为 [3, 7, <u>9</u>, 15, 20], 返回 9

提示:

树中节点个数的范围是 [1, 105] 。

0 <= Node.val <= 106

最多调用 105 次 hasNext、 next、 hasPrev 和 prev 。

# 分析

要实现BST中序遍历的双向迭代器（支持`next()`/`hasNext()`和`prev()`/`hasPrev()`），核心是利用BST中序遍历“左→根→右”的有序性，通过\*\*栈维护遍历状态\*\*和\*\*数组存储已访问节点值\*\*，兼顾高效查询和双向移动能力。

思路：

BST的中序遍历结果是严格递增的，这是实现迭代器的关键。为了支持双向移动，我们需要：

1、记录已访问节点\*\*：用数组`visited`存储已通过`next()`返回的节点值（即中序序列的前缀），数组索引对应节点在中序序列中的位置。

2、维护待访问节点\*\*：用栈`stack`模拟中序遍历的迭代过程，存储尚未访问的节点（优先左子树入栈，确保弹出顺序为中序）。

3、跟踪当前指针位置\*\*：用`currPos`表示当前指针在`visited`数组中的位置（初始为-1，对应“小于树中所有值的虚拟初始指针”）：

- `next()`：若栈非空，继续中序遍历并将节点值加入`visited`，`currPos`右移并返回新值；

- `prev()`：`currPos`左移，返回`visited`中对应位置的值；

- `hasNext()`：栈非空（仍有未访问节点）或`currPos`未到`visited`末尾（已访问节点还有右侧元素）；

- `hasPrev()`：`currPos` > 0（当前指针左侧有已访问节点）。

代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode() : val(0), left(nullptr), right(nullptr) {}

\* TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

\* TreeNode(int x, TreeNode \*left, TreeNode \*right) : val(x), left(left), right(right) {}

\* };

\*/

class BSTIterator {

private:

stack<TreeNode\*> stk; // 存储待访问的节点（中序遍历栈）

vector<int> visited; // 存储已通过next()返回的节点值（中序序列前缀）

int currPos; // 当前指针在visited中的位置（初始为-1，虚拟初始状态）

// 辅助函数：将当前节点的所有左子树入栈（中序遍历的左子树优先处理）

void pushLeft(TreeNode\* node) {

while (node != nullptr) {

stk.push(node);

node = node->left;

}

}

public:

// 构造函数：初始化栈（压入根节点的所有左子树）和指针位置

BSTIterator(TreeNode\* root) {

pushLeft(root);

currPos = -1;

}

// 判断当前指针右侧是否有元素（未访问节点或已访问节点的右侧）

bool hasNext() {

// 栈非空：仍有未访问的节点；currPos < visited.size()-1：已访问节点还有右侧元素

return !stk.empty() || currPos < (int)visited.size() - 1;

}

// 指针右移，返回新位置的元素

int next() {

// 若栈非空，继续中序遍历，将下一个节点值加入visited

if (!stk.empty()) {

TreeNode\* top = stk.top();

stk.pop();

// 中序遍历：左→根→右，弹出根节点后，压入其右子树的所有左节点

pushLeft(top->right);

// 将当前根节点值加入已访问列表

visited.push\_back(top->val);

}

// 指针右移，返回对应值

currPos++;

return visited[currPos];

}

// 判断当前指针左侧是否有元素（已访问节点的左侧）

bool hasPrev() {

// currPos > 0：指针左侧存在已访问节点（visited[0..currPos-1]）

return currPos > 0;

}

// 指针左移，返回新位置的元素

int prev() {

// 指针左移，返回对应值（题目保证调用有效，无需判断hasPrev()）

currPos--;

return visited[currPos];

}

};

/\*\*

\* Your BSTIterator object will be instantiated and called as such:

\* BSTIterator\* obj = new BSTIterator(root);

\* bool param\_1 = obj->hasNext();

\* int param\_2 = obj->next();

\* bool param\_3 = obj->hasPrev();

\* int param\_4 = obj->prev();

\*/

代码解释：

1、构造函数与pushLeft辅助函数：

- pushLeft：将传入节点的所有左子树节点入栈，确保栈顶始终是“当前待访问的最小节点”（中序遍历的下一个节点）；

- 构造函数调用pushLeft(root)，初始化遍历栈，currPos设为-1（虚拟初始指针，小于所有节点值）。

2、hasNext()：

- 若栈非空，说明还有未访问的节点（中序序列的后缀）；

- 若currPos未到visited末尾，说明已访问节点中还有右侧元素（如visited = [3,7]，currPos=0时右侧有7）。

3、next()：

- 栈非空时，弹出栈顶节点（当前最小节点），压入其右子树的所有左节点（为下一次`next()`做准备），并将弹出节点值加入visited；

- currPos右移，返回visited[currPos]（新指针位置的元素）。

4、hasPrev()：

- 若currPos > 0，说明指针左侧有已访问节点（如currPos=1时，左侧有visited[0]）。

5、prev()：

- currPos左移，返回visited[currPos]（题目保证调用时hasPrev()为真，无需额外判断）。

复杂度分析

- 时间复杂度：next()和prev()均为均摊O(1)。每个节点入栈和出栈各1次，visited数组的访问和修改为O(1)，总操作次数与节点数成正比，均摊到每次调用为O(1)。

- 空间复杂度：O(h)，其中h为BST的高度。栈的最大存储量为树的高度（平衡树h=logn，最坏情况h=n），visited数组的存储量取决于next()的调用次数（最多为n，但题目未要求提前存储所有节点，符合“不提前遍历树”的进阶要求）。

进阶适配（不提前遍历树）

本方案未在初始化时遍历整个树，而是通过栈动态维护待访问节点，仅在next()调用时才访问节点并加入visited，完全满足“不提前遍历树”的进阶要求，同时支持高效的双向移动。