本次报告的目的是探究在二叉查找树中递归遍历和非递归遍历的时间效率问题，代码实现如下：

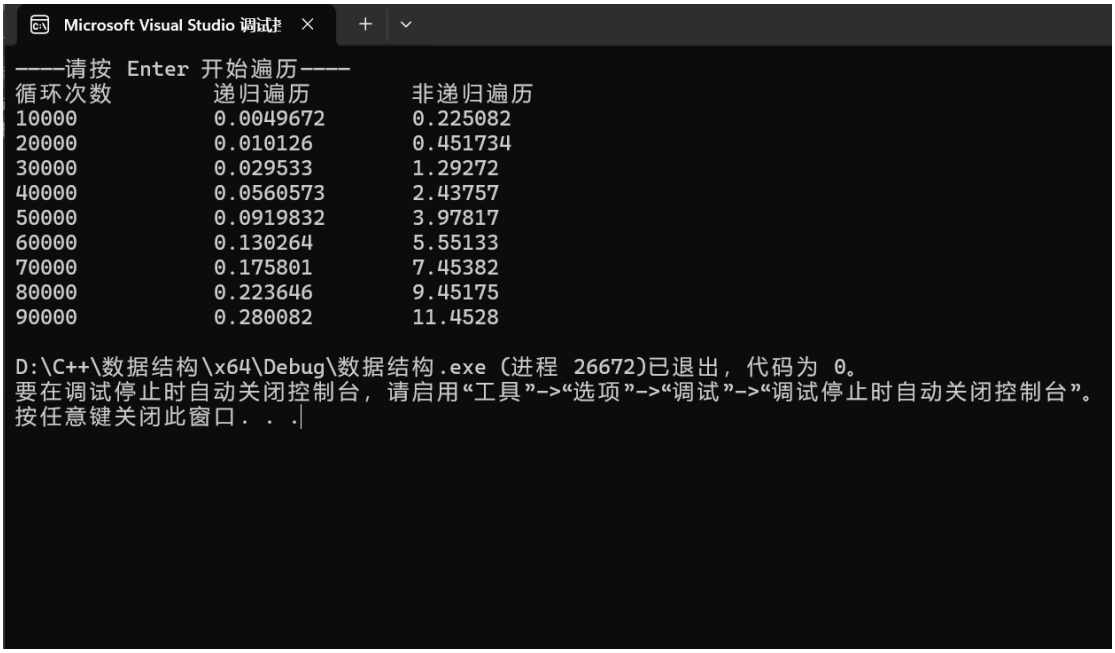
```
#include <iostream>
#include <stack>
#include <omp.h>
#include <random>
using namespace std;


// 定义 BST 的节点
class TreeNode {

private:
	int val;
	TreeNode* left;
	TreeNode* right;

public:
	TreeNode(int x) : val(x), left(nullptr), right(nullptr) {
	}
	TreeNode() {
	}
	friend class BST;
};

// BST 类
class BST {
private:
	TreeNode* root;

public:
	BST() : root(nullptr) {}

	// 插入新值
	void insert(int val) {
		root = insertRec(root, val);
	}

	// 递归插入
	TreeNode* insertRec(TreeNode* root, int val) {
		if (root == nullptr) {
			root = new TreeNode(val);
			return root;
		}

		if (val < root->val) {
			root->left = insertRec(root->left, val);
		}
		else if (val > root->val) {
			root->right = insertRec(root->right, val);
		}

		// 如果值相等，这里简单处理为不插入，可修改
		return root;
	}
```

```
// 查找值
bool search(int val) {
	return searchRec(root, val);
}

// 递归查找
bool searchRec(TreeNode* root, int val) {
	if (root == nullptr) {
		return false;
	}

	if (root->val == val) {
		return true;
	}

	if (val < root->val) {
		return searchRec(root->left, val);
	}
	else {
		return searchRec(root->right, val);
	}
}


// 递归中序遍历
void inorderTraversalRec(TreeNode* root) {
	if (root != nullptr) {
		inorderTraversalRec(root->left);
		//cout << root->val << " ";
		inorderTraversalRec(root->right);
	}
}
// 中序遍历 BST
void inorderTraversal() {
	inorderTraversalRec(root);
	//cout << endl;
}


// 前序遍历 BST
void preorderTraversal() {
	preorderTraversalRec(root);
	cout << endl;
}
// 递归前序遍历
void preorderTraversalRec(TreeNode* root) {
	if (root != nullptr) {
		cout << root->val << " "; // 访问根节点
		preorderTraversalRec(root->left); // 递归遍历左子树
		preorderTraversalRec(root->right); // 递归遍历右子树
	}
}


// 后序遍历 BST
void postorderTraversal() {
	postorderTraversalRec(root);
	cout << endl;
```

```
    }
    // 递归后序遍历
    void postorderTraversalRec(TreeNode* root) {
        if (root != nullptr) {
            postorderTraversalRec(root->left); // 递归遍历左子树
            postorderTraversalRec(root->right); // 递归遍历右子树
            cout << root->val << " "; // 访问根节点
        }
    }

    //非递归中序遍历
    void inorderTraversal_alter() {
        inorderTraversal_alterRec(root);
    }
    void inorderTraversal_alterRec(TreeNode* root) {
        stack<TreeNode*> stk;
        TreeNode* curr = root;
        while (curr != nullptr || !stk.empty()) {
            while (curr != nullptr) {
                stk.push(curr);
                curr = curr->left;
            }
            curr = stk.top();
            stk.pop();
            //cout << curr->val << " ";
            curr = curr->right;
        }
        //cout << endl;
    }

};

int main() {
    BST bst;
    cout << "————请按 Enter 开始遍历————";
    cin.get();
    cout << "循环次数\t" << "递归遍历\t" << "非递归遍历" << endl;
    for (int n = 10000; n < 100000; n += 10000) {
        srand(time(NULL));
        for (int i = 0; i < 100; i++) {
            bst.insert(rand() % 1000);
        }

        double start = 0, end = 0;

        //递归中序遍历所用时间
        start = omp_get_wtime();
        for (int i = 0; i < n; i++) {
            bst.inorderTraversal();
        }
        end = omp_get_wtime();
        cout << n << "\t\t" << end - start;

        //中序遍历所用时间
        start = omp_get_wtime();
        for (int i = 0; i < n; i++) {
            bst.inorderTraversal_alter();
        }
```

```
        end = omp_get_wtime();
        cout << "\t" << end - start << endl;
    }

    return 0;
}
```

代码运行过结果如下图：



由此可见递归的效率高于非递归栈遍历并约为它所耗时间的 1/40。

同时，递归函数具有以下的优点：

代码简洁性：递归遍历的代码往往更加简洁、易于理解。递归方法通过函数调用自身来解决问题，可以自然地反映出树结构的层次关系，使得代码结构更加清晰。

逻辑直观性：对于树结构的遍历，递归方法更贴近人的直觉思维。人们习惯于将大问题分解为小问题，递归正是利用了这种分解的思想，使得问题的求解过程更加直观。

但为什么递归要比非递归要快呢？以下是综合了网络资料和 AI 解答的答案。

1. 调用栈的优化

内置栈的高效性：递归遍历在内部使用调用栈来保存函数调用过程中的状态（如局部变量和返回地址）。大多数现代编程语言和运行时环境都对调用栈进行了优化，以提供快速的函数调用和返回操作。这种优化可能包括使用快速的内存分配和回收机制、减少缓存未命中以及利用现代 CPU 的硬件特性等。

减少内存访问：递归遍历通过内置的调用栈自动管理数据，可能减少了程序员手动管理数据结构（如栈或队列）时所需的额外内存访问和管理开销。

2. 编译器优化

尾递归优化：虽然并非所有编程语言都支持尾递归优化，但那些支持的语言可以通过将尾递归调用转换为循环来消除递归调用带来的开销。这可以显著提高递归遍历的性能，尤其是在处理深度较大的树时。

内联展开：编译器可能会将小型的递归函数内联展开，以减少函数调用的开销。这种优化在递归深度较浅时特别有效。

3. 特定场景下的优势

小数据集：对于较小的数据集或树结构，递归遍历可能由于其简洁性和内置栈的高效性而表现出更快的性能。

缓存友好性：在某些情况下，递归遍历可能由于其对内存访问模式的特定方式而更加缓存友好，从而减少了缓存未命中的次数。