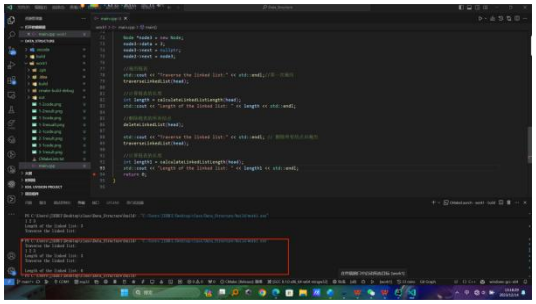
1. 单链表的建立:

RESULT:

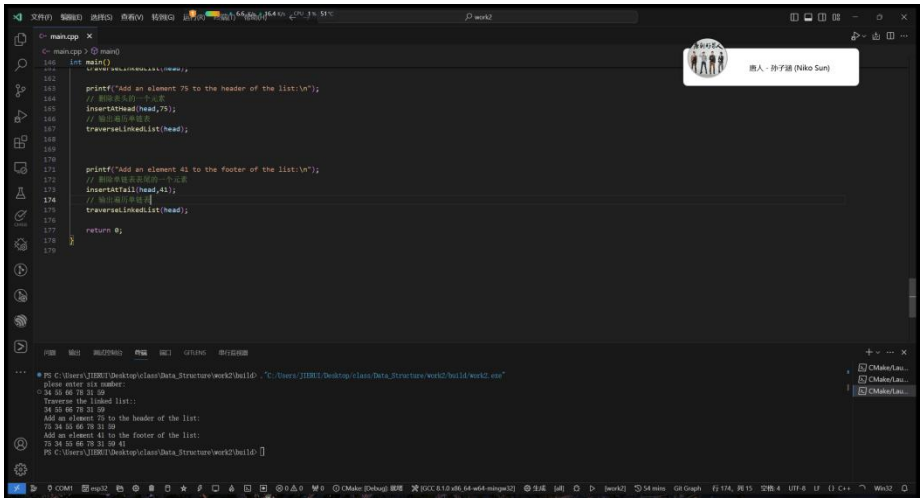


CODE:

```cpp
#include <iostream>
// 定义链表的结点结构体
struct Node
{
    int data;
    Node *next;
};
/**
 * @brief 遍历链表
 *
 * @param head 链表头结点
 */
void traverseLinkedList(Node *head)
{
    Node *current = head;
    while (current != nullptr) // 当前结点非空
    {
        std::cout << current->data << " ";
        current = current->next;
    }
    std::cout << std::endl;
}
/**
 * @brief 计算链表的长度
 *
 * @param head 链表头结点
 * @return int 链表长度
 */
int calculateLinkedListLength(Node *head)
{
    int length = 0;
    Node *current = head;
    while (current != nullptr)
    {
        length++;                // 长度自增
        current = current->next; // 指针指向下一个结点
    }
    return length;
}
/**
 * @brief 删除链表的所有结点
 *
 * @param head 链表头结点
 */
void deleteLinkedList(Node *&head)
{
    Node *current = head;
    while (current != nullptr)
    {
        Node *temp = current;
        current = current->next; // 指针指向下一个结点
        delete temp;
    }
    head = nullptr;
}

int main()
{
    Node *head = nullptr;

    // 建立一个单链表
    Node *node1 = new Node;
    node1->data = 1;
    node1->next = nullptr;
    head = node1;

    Node *node2 = new Node;
    node2->data = 2;
    node2->next = nullptr;
    node1->next = node2;

    Node *node3 = new Node;
    node3->data = 3;
    node3->next = nullptr;
    node2->next = node3;

    // 遍历链表
    std::cout << "Traverse the linked list:" << std::endl; // 第一次遍历
    traverseLinkedList(head);

    // 计算链表的长度
    int length = calculateLinkedListLength(head);
    std::cout << "Length of the linked list: " << length << std::endl;

    // 删除链表的所有结点
    deleteLinkedList(head);

    std::cout << "Traverse the linked list:" << std::endl; // 删除所有结点后遍历
    traverseLinkedList(head);

    // 计算链表的长度
    int length1 = calculateLinkedListLength(head);
    std::cout << "Length of the linked list: " << length1 << std::endl;
    return 0;
}
```

2. 单链表的插入:

RESULT:

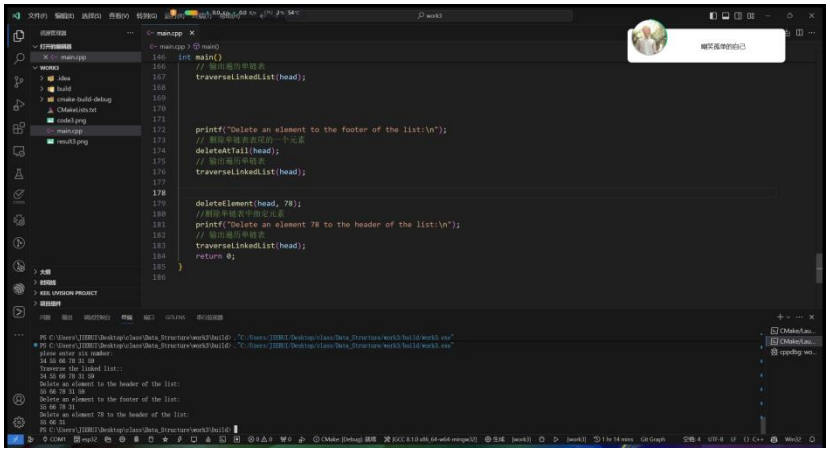


CODE:

```cpp
#include <iostream>
// 单链表的节点结构体,即一个节点包含数据和指向下一个节点的指针
struct Node
{
    int data;
    Node *next;
};
/**
 * @brief 遍历单链表
 *
 * @param head 单链表的头指针
 */
void traverseLinkedList(Node *head)
{
    Node *current = head;
    while (current != nullptr)
    {
        std::cout << current->data << " "; // 输出当前节点的数据,并换行
        current = current->next;           // 将current指向下一个节点
    }
    std::cout << std::endl; // 使用std::endl换行
}
/**
 * @brief 向单链表的表头添加一个元素
 *
 * @param head 单链表的头指针
 * @param value 要添加的元素
 */
void insertAtHead(Node *&head, int value)
{
    Node *newNode = new Node();
    newNode->data = value; // 将新节点的数据域赋值为value
    newNode->next = head;  // 将新节点的next指针指向原来的头节点
    head = newNode;
}
/**
 * @brief 向单链表的表尾添加一个元素
 *
 * @param head 单链表的头指针
 * @param value 要添加的元素
 */
void insertAtTail(Node *&head, int value)
{
    Node *newNode = new Node();
    newNode->data = value;  // 将新节点的数据域赋值为value
    newNode->next = nullptr; // 将新节点的next指针置空

    if (head == nullptr) // 如果为空链表,直接将新节点作为头节点
    {
        head = newNode;
    }
    else
    {
        Node *current = head;
        while (current->next != nullptr)
        {
            current = current->next;
        }
        current->next = newNode; // 将新节点添加到最后一个节点的后面
    }
}

/**
 * @brief 删除单链表的表头元素
 *
 * @param head 单链表的头指针
 */
void deleteAtHead(Node *&head)
{
    if (head == nullptr) // 如果为空链表,直接返回
    {
        return;
    }

    Node *temp = head; // 保存要删除的节点的地址
    head = head->next; // 头指针指向下一个节点
    delete temp;       // release原来的头节点
}

/**
 * @brief 删除单链表的表尾元素
 *
 * @param head 单链表的头指针
 */
void deleteAtTail(Node *&head)
{
    if (head == nullptr) // 如果为空链表,直接返回
    {
        return;
    }

    if (head->next == nullptr) // 如果只有一个节点,直接删除
    {
        delete head;
        head = nullptr;
        return;
    }

    Node *current = head;
    while (current->next->next != nullptr) // 找到倒数第二个节点
    {
        current = current->next;
    }
    delete current->next;    // 删除最后一个节点
    current->next = nullptr; // 将倒数第二个节点的next指针置空
}

/**
 * @brief 删除单链表中指定元素
 *
 * @param head 单链表的头指针
 * @param value 要删除的元素
 */
void deleteElement(Node *&head, int value)
{
    if (head == nullptr) // 如果为空链表,直接返回
    {
        return;
    }

    // 如果要删除的元素在头节点
    if (head->data == value)
    {
        deleteAtHead(head);//直接删除头结点并返回
        return;
    }

    Node *current = head;//临时结点,用于保存当前结点的地址
    Node *previous = nullptr;//临时结点,用于保存当前结点的前驱结点地址

    // 遍历链表查找要删除的元素
    while (current != nullptr && current->data != value)// 当链表不为空且当前结点的数据域不等于value,继续向后遍历
    {
        previous = current;
        current = current->next;
    }

    // 如果找到要删除的元素
    if (current != nullptr)
    {
        previous->next = current->next;//将当前结点的前驱结点的next指针指向当前结点的后继结点
        delete current;//释放当前结点,从链表删除元素
    }
}

int main()
{
    Node *head = nullptr; // 单链表的头指针,先初始化为nullptr

    // 从键盘上读取六个数据,并插入保存到单链表中
    printf("please enter six number:\n");
    for (int i = 0; i < 6; i++)
    {
        int num;
        std::cin >> num;
        insertAtTail(head, num); // 在单链表的表尾添加元素
    }

    // 输出遍历单链表
    printf("Traverse the linked list:\n");
    traverseLinkedList(head);

    printf("Add an element 75 to the header of the list:\n");
    // 删除表头的一个元素
    insertAtHead(head,75);
    // 输出遍历单链表
    traverseLinkedList(head);



    printf("Add an element 41 to the footer of the list:\n");
    // 删除单链表表尾的一个元素
    insertAtTail(head,41);
    // 输出遍历单链表
    traverseLinkedList(head);

    return 0;
}
```

3. 单链表的删除:

RESULT:

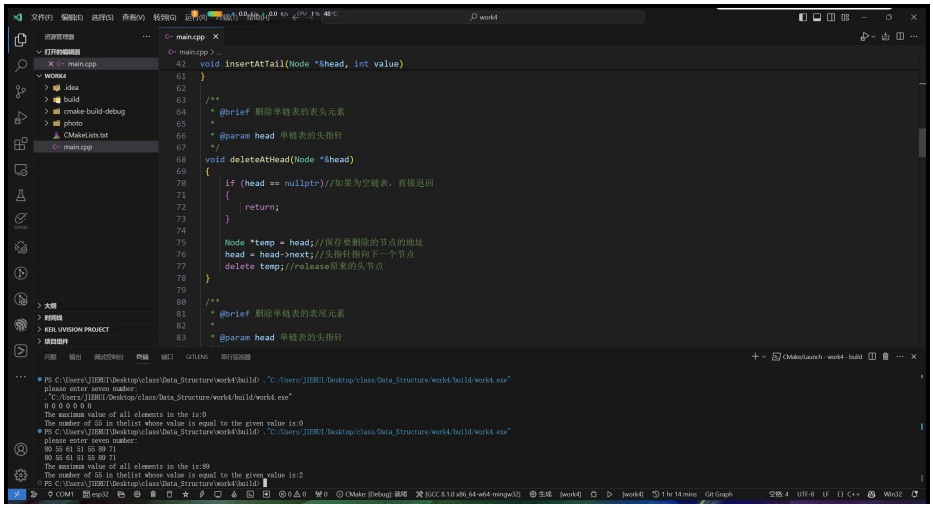


CODE:

```cpp
#include <iostream>
// 单链表的节点结构体,即一个节点包含数据和指向下一个节点的指针
struct Node
{
    int data;
    Node *next;
};
/**
 * @brief 遍历单链表
 *
 * @param head 单链表的头指针
 */
void traverseLinkedList(Node *head)
{
    Node *current = head;
    while (current != nullptr)
    {
        std::cout << current->data << " "; // 输出当前节点的数据,并换
行      current = current->next;           // 将current指向下一个节点
    }
    std::cout << std::endl; // 也用std::endl换行
}
/**
 * @brief 向单链表的表头添加一个元素
 *
 * @param head 单链表的头指针
 * @param value 要添加的元素
 */
void insertAtHead(Node *&head, int value)
{
    Node *newNode = new Node();
    newNode->data = value; // 将新节点的数据域赋值为value
    newNode->next = head;  // 将新节点的next指针指向原来的头节点
    head = newNode;
}
/**
 * @brief 向单链表的表尾添加一个元素
 *
 * @param head 单链表的头指针
 * @param value 要添加的元素
 */
void insertAtTail(Node *&head, int value)
{
    Node *newNode = new Node();
    newNode->data = value;   // 将新节点的数据域赋值为value
    newNode->next = nullptr; // 将新节点的next指针置空

    if (head == nullptr) // 如果为空链表，直接将新节点作为头节点
    {
        head = newNode;
    }
    else
    {
        Node *current = head;
        while (current->next != nullptr)
        {
            current = current->next;
        }
        current->next = newNode; // 将新节点添加到最后一个节点的后面
    }
}

/**
 * @brief 删除单链表的表头元素
 *
 * @param head 单链表的头指针
 */
void deleteAtHead(Node *&head)
{
    if (head == nullptr) // 如果为空链表，直接返回
    {
        return;
    }

    Node *temp = head; // 保存要删除的节点的地址
    head = head->next; // 头指针指向下一个节点
    delete temp;       // release原来的头节点
}

/**
 * @brief 删除单链表的表尾元素
 *
 * @param head 单链表的头指针
 */
void deleteAtTail(Node *&head)
{
    if (head == nullptr) // 如果为空链表，直接返回
    {
        return;
    }

    if (head->next == nullptr) // 如果只有一个节点，直接删除
    {
        delete head;
        head = nullptr;
        return;
    }

    Node *current = head;
    while (current->next->next != nullptr) // 找到倒数第二个节点
    {
        current = current->next;
    }
    delete current->next;    // 删除最后一个节点
    current->next = nullptr; // 将倒数第二个节点的next指针置空
}

/**
 * @brief 删除单链表中指定元素
 *
 * @param head 单链表的头指针
 * @param value 要删除的元素
 */
void deleteElement(Node *&head, int value)
{
    if (head == nullptr) // 如果为空链表，直接返回
    {
        return;
    }

    // 如果要删除的元素是头节点
    if (head->data == value)
    {
        deleteAtHead(head);
        return;
    }

    Node *current = head;
    Node *previous = nullptr;

    // 遍历链表查找要删除的元素
    while (current != nullptr && current->data != value)
    {
        previous = current;
        current = current->next;
    }

    // 如果找到要删除的元素
    if (current != nullptr)
    {
        previous->next = current->next;
        delete current;
    }
}

int main()
{
    Node *head = nullptr; // 单链表的头指针,先初始化为nullptr

    // 从键盘上读取六个数数，并依次保存到单链表中
    printf("plese enter six number:\n");
    for (int i = 0; i < 6; i++)
    {
        int num;
        std::cin >> num;
        insertAtTail(head, num); // 在单链表的表尾添加元素
    }

    // 输出遍历单链表
    printf("Traverse the linked list::\n");
    traverseLinkedList(head);

    printf("Delete an element to the header of the list:\n");
    // 删除表头的一个元素
    deleteAtHead(head);
    // 输出遍历单链表
    traverseLinkedList(head);



    printf("Delete an element to the footer of the list:\n");
    // 删除单链表表尾的一个元素
    deleteAtTail(head);
    // 输出遍历单链表
    traverseLinkedList(head);


    deleteElement(head, 78);
    //删除单链表中指定元素
    printf("Delete an element 78 to the header of the list:\n");
    // 输出遍历单链表
    traverseLinkedList(head);
    return 0;
}
```

4. 单链表的查找:

RESULT:

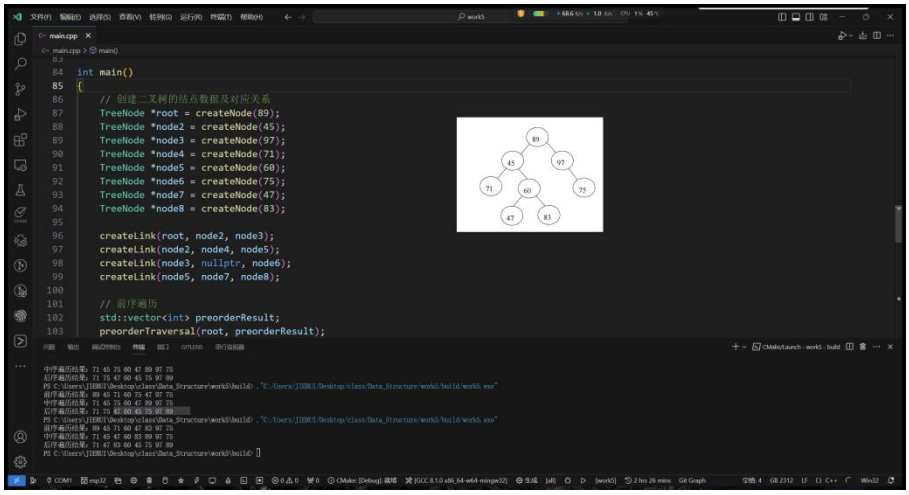


CODE:

```cpp
#include <iostream>
// 单链表的节点结构体,每一个节点包含数据和指向下一个节点的指针
struct Node
{
    int data;
    Node *next;
};
/**
 * @brief 遍历单链表
 *
 * @param head 单链表的头指针
 */
void traverseLinkedList(Node *head)
{
    Node *current = head;
    while (current != nullptr)
    {
        std::cout << current->data << " ";//输出当前节点的数据,并换行
        current = current->next;//将current指向下一个节点
    }
    std::cout << std::endl;//调用std::endl换行
}
/**
 * @brief 向单链表的表头添加一个元素
 *
 * @param head 单链表的头指针
 * @param value 要添加的元素
 */
void insertAtHead(Node *&head, int value)
{
    Node *newNode = new Node();
    newNode->data = value;//将新节点的数据域赋值为value
    newNode->next = head;//将新节点的next指针指向原来的头节点
    head = newNode;
}
/**
 * @brief 向单链表的表尾添加一个元素
 *
 * @param head 单链表的头指针
 * @param value 要添加的元素
 */
void insertAtTail(Node *&head, int value)
{
    Node *newNode = new Node();
    newNode->data = value;//将新节点的数据域赋值为value
    newNode->next = nullptr;//将新节点的next指针置空

    if (head == nullptr)//如果为空链表,直接将新节点作为头节点
    {
        head = newNode;
    }
    else
    {
        Node *current = head;
        while (current->next != nullptr)
        {
            current = current->next;
        }
        current->next = newNode;//将新节点添加到最后一个节点的后面
    }
}

/**
 * @brief 删除单链表的表头元素
 *
 * @param head 单链表的头指针
 */
void deleteAtHead(Node *&head)
{
    if (head == nullptr)//如果为空链表,直接返回
    {
        return;
    }

    Node *temp = head;//保存要删除的节点的地址
    head = head->next;//头指针指向下一个节点
    delete temp;//release原来的头节点
}

/**
 * @brief 删除单链表的表尾元素
 *
 * @param head 单链表的头指针
 */
void deleteAtTail(Node *&head)
{
    if (head == nullptr)//如果为空链表,直接返回
    {
        return;
    }

    if (head->next == nullptr)//如果只有一个节点,直接删除
    {
        delete head;
        head = nullptr;
        return;
    }

    Node *current = head;
    while (current->next->next != nullptr)//找到倒数第二个节点
    {
        current = current->next;
    }
    delete current->next;//删除最后一个节点
    current->next = nullptr;//将倒数第二个节点的next指针置空
}

/**
 * @brief 删除单链表中指定元素
 *
 * @param head 单链表的头指针
 * @param value 要删除的元素
 */
void deleteElement(Node *&head, int value)
{
    if (head == nullptr) // 如果为空链表,直接返回
    {
        return;
    }

    // 如果要删除的元素是头节点
    if (head->data == value)
    {
        deleteAtHead(head);
        return;
    }

    Node *current = head;
    Node *previous = nullptr;

    // 遍历链表查找要删除的元素
    while (current != nullptr && current->data != value)
    {
        previous = current;
        current = current->next;
    }

    // 如果找到要删除的元素
    if (current != nullptr)
    {
        previous->next = current->next;
        delete current;
    }
}

/**
 * @brief 从单链表中查找出所有元素的最大值
 *
 * @param head 单链表的头指针
 * @return int 单链表中所有元素的最大值
 */
int findMaxValue(Node *head)
{
    if (head == nullptr) // 如果为空链表,显示出错信息并终止运行
    {
        printf("Error: 链表为空\n");
        exit(1);
    }

    int maxValue = head->data;  // 假设第一个元素为最大值
    Node *current = head->next; // 指针指向第二个元素开始遍历

    while (current != nullptr) // 开始遍历链表
    {
        if (current->data > maxValue)
        {
            maxValue = current->data;
        }
        current = current->next;
    }
    return maxValue;
}


/**
 * @brief 统计单链表中数等于给定值的结点个数
 *
 * @param head 单链表的头指针
 * @param value 给定数
 * @return int 数等于给定数的结点个数
 */
int countNodesWithValue(Node *head, int value)
{
    int count = 0;
    Node *current = head;

    while (current != nullptr)
    {
        if (current->data == value)
        {
            count++;
        }
        current = current->next;
    }
    return count;
}


int main()
{
    Node *head = nullptr; // 单链表的头指针,初始化为nullptr

    // 从键盘上读取六个整数,并按顺序存放到单链表中
    printf("please enter seven number:\n");
    for (int i = 0; i < 7; i++)
    {
        int num;
        std::cin >> num;
        insertAtTail(head, num);//在单链表的表尾添加元素
    }
    // 输出整个单链表
    traverseLinkedList(head);

    //查找单链表中所有元素的最大值
    printf("The maximum value of all elements in the is:%d\n", findMaxValue(head));
    printf("The number of 55 in thelist whose value is equal to the given value is:%d\n", countNodesWithValue(head, 55));
    return 0;
}
```

5. 树的操作与处理:

RESULT:



CODE:

```cpp
#include <iostream>
#include <vector>
// 树的结构体
struct TreeNode
{
    int data;        // 数据
    TreeNode *left;  // 左子节点
    TreeNode *right; // 右子节点
};

/**
 * @brief 创建一个二叉树节点
 * @param data 节点数据
 * @return 新创建的节点指针
 */
TreeNode *createNode(int data)
{
    TreeNode *newNode = new TreeNode();
    if (newNode)
    {
        newNode->data = data;
        newNode->left = newNode->right = nullptr;
    }
    return newNode;
}

/**
 * @brief 创建节点之间的链接
 * @param parent 父节点
 * @param leftChild 左子节点
 * @param rightChild 右子节点
 */
void createLink(TreeNode *parent, TreeNode *leftChild, TreeNode *rightChild)
{
    parent->left = leftChild;
    parent->right = rightChild;
}

/**
 * @brief 前序遍历二叉树
 * @param root 根节点
 * @param result 存储遍历结果的向量
 */
void preorderTraversal(TreeNode *root, std::vector<int> &result)
{
    if (root)
    {
        result.push_back(root->data);
        preorderTraversal(root->left, result);
        preorderTraversal(root->right, result);
    }
}

/**
 * @brief 中序遍历二叉树
 * @param root 根节点
 * @param result 存储遍历结果的向量
 */
void inorderTraversal(TreeNode *root, std::vector<int> &result)
{
    if (root)
    {
        inorderTraversal(root->left, result);
        result.push_back(root->data);
        inorderTraversal(root->right, result);
    }
}

/**
 * @brief 后序遍历二叉树
 * @param root 根节点
 * @param result 存储遍历结果的向量
 */
void postorderTraversal(TreeNode *root, std::vector<int> &result)
{
    if (root)
    {
        postorderTraversal(root->left, result);
        postorderTraversal(root->right, result);
        result.push_back(root->data);
    }
}

int main()
{
    // 创建二叉树的结点数据及对应关系
    TreeNode *root = createNode(89);
    TreeNode *node2 = createNode(45);
    TreeNode *node3 = createNode(97);
    TreeNode *node4 = createNode(71);
    TreeNode *node5 = createNode(60);
    TreeNode *node6 = createNode(75);
    TreeNode *node7 = createNode(47);
    TreeNode *node8 = createNode(83);

    createLink(root, node2, node3);
    createLink(node2, node4, node5);
    createLink(node3, nullptr, node6);
    createLink(node5, node7, node8);

    // 前序遍历
    std::vector<int> preorderResult;
    preorderTraversal(root, preorderResult);
    std::cout << "前序遍历结果：";
    for (int num : preorderResult)
    {
        std::cout << num << " ";
    }
    std::cout << std::endl;



    // 中序遍历
    std::vector<int> inorderResult;
    inorderTraversal(root, inorderResult);
    std::cout << "中序遍历结果：";
    for (int num : inorderResult)
    {
        std::cout << num << " ";
    }
    std::cout << std::endl;


    // 后序遍历
    std::vector<int> postorderResult;
    postorderTraversal(root, postorderResult);
    std::cout << "后序遍历结果：";
    for (int num : postorderResult)
    {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```