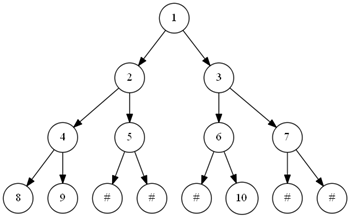
|  |
| --- |
| 哈尔滨工业大学(深圳) |
| **《数据结构》实验报告** |
|  |
| 实验三  树型结构及其应用  学 院: 计算机科学与技术学院   |  |  | | --- | --- | | 姓 名: | 梁鑫嵘 | | 学 号: | 200110619 | | 专 业: | 计算机系 | | 日 期: | 2021-04-16 | |

# 一、问题分析

### 题目1

按层次遍历建立二叉树，并输出该二叉树的前序遍历、 中序遍历和后序遍历的序列。



“按照层次遍历”，实际上是按照满二叉树的宽度优先遍历方式把输入序列和节点一一对应，即按照这个顺序每访问一个节点就填入一个输入的节点的值，并且把#转成-1。

### 题目2

给定一棵二叉树，**路径**定义为从树的根节点到叶子结点的任意路径，求取该二叉树的最大路径和，**路径和**定义为一条路径中各节点的权值之和。

由于路径的定义是从树的根节点到叶子结点的任意路径，所以可以使用先序遍历，当遍历到叶子节点的时候向上寻找，计算路径和，最后比较得出最大路径和。

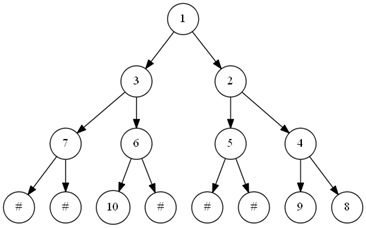
### 题目3

给定一棵二叉树，求取该二叉树的所有左子叶权重之和，左子叶被定义为二叉树叶子结点中属于左子树的节点。

同样可以使用先序遍历，当遇到左子叶的时候累计权重。

### 题目4

给定求取该树的镜像，即翻转该二叉树，如图1的结果如图2所示。并输出翻转后二叉树的中序遍历。



翻转二叉树可以使用先序遍历，把每次遍历到的节点左右互相交换，然后就可以继续访问左右节点。

# 二、详细设计

## 2.1 设计思想

#### 题目1

按层次遍历建立二叉树，并输出该二叉树的前序遍历、 中序遍历和后序遍历的序列。

“满二叉树的遍历” 方式，也就是第​层有​个节点。

所以可以对一棵空树通过宽度优先遍历的方式逐步建树，然后删除多余的节点来节约空间。细节上的实践方法，大致是设置遇到空节点的时候是否入队来控制是否遍历一棵满树。

三种遍历方法都很简单，详见后文的流程图。

#### 题目2

给定一棵二叉树，**路径**定义为从树的根节点到叶子结点的任意路径，求取该二叉树的最大路径和，**路径和**定义为一条路径中各节点的权值之和。

使用先序遍历，当遍历到叶子节点的时候向上寻找，计算路径和，最后比较得出最大路径和。

由于数据结构中没有实现指向父节点的指针，所以需要一个栈来记录访问的路径。

每向下访问一次，就有一个指针入栈；这次遍历的函数结束的时候，让这个指针出栈，就能完成记录路径的功能。

#### 题目3

给定一棵二叉树，求取该二叉树的所有左子叶权重之和，左子叶被定义为二叉树叶子结点中属于左子树的节点。

同样可以使用先序遍历，当遇到左子叶的时候累计权重。

如何判断是左子叶呢？如果已经遍历到左子叶的时候才判断是不是子叶，那么就已经丢失了“左右”的信息。所以应在遍历到的节点判断左边子树是不是叶节点，并且在每次遍历的时候跳过叶子节点判断。

#### 题目4

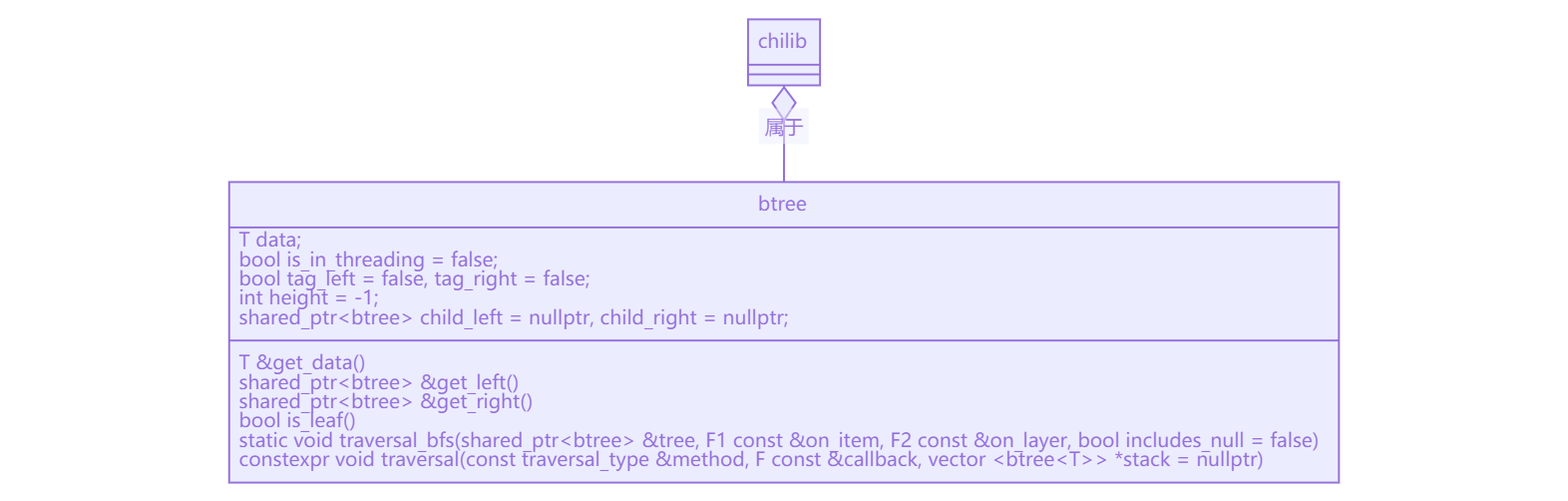
给定求取该树的镜像，即翻转该二叉树，如图1的结果如图2所示。并输出翻转后二叉树的中序遍历。

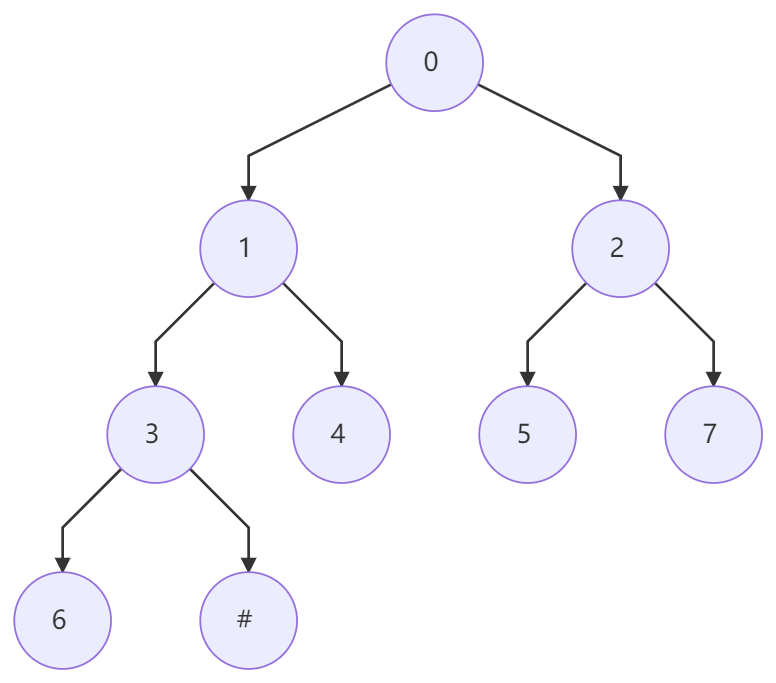
翻转二叉树可以使用先序遍历，把每次遍历到的节点左右互相交换，然后就可以继续访问左右节点。

在交换的时候，由于本工程底层使用了智能指针，如果使用一般的交换方式有出现环的可能，还要避免这样的可能性。

## 2.2 存储结构及操作

(1) 存储结构



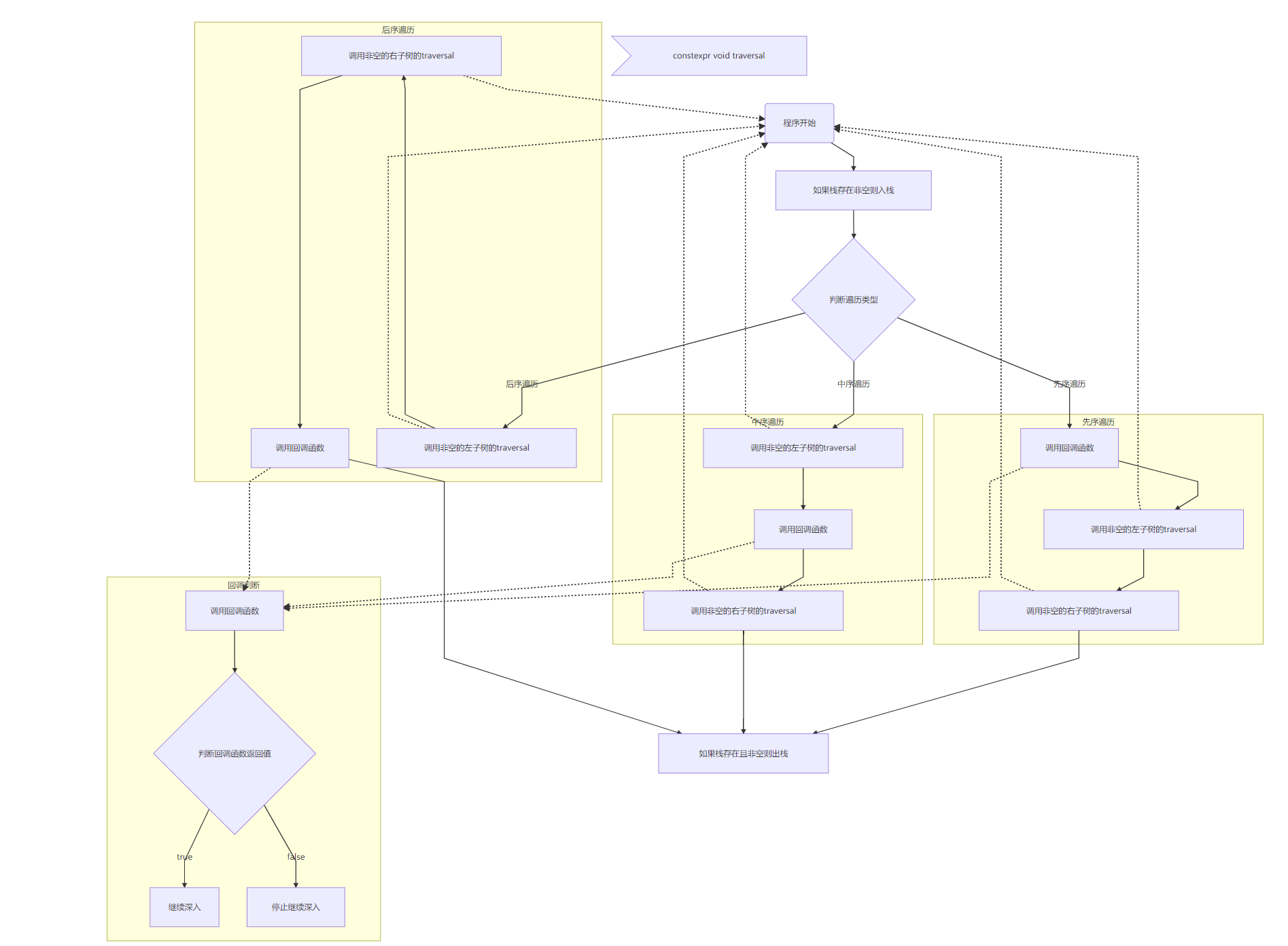


(2) 涉及的操作

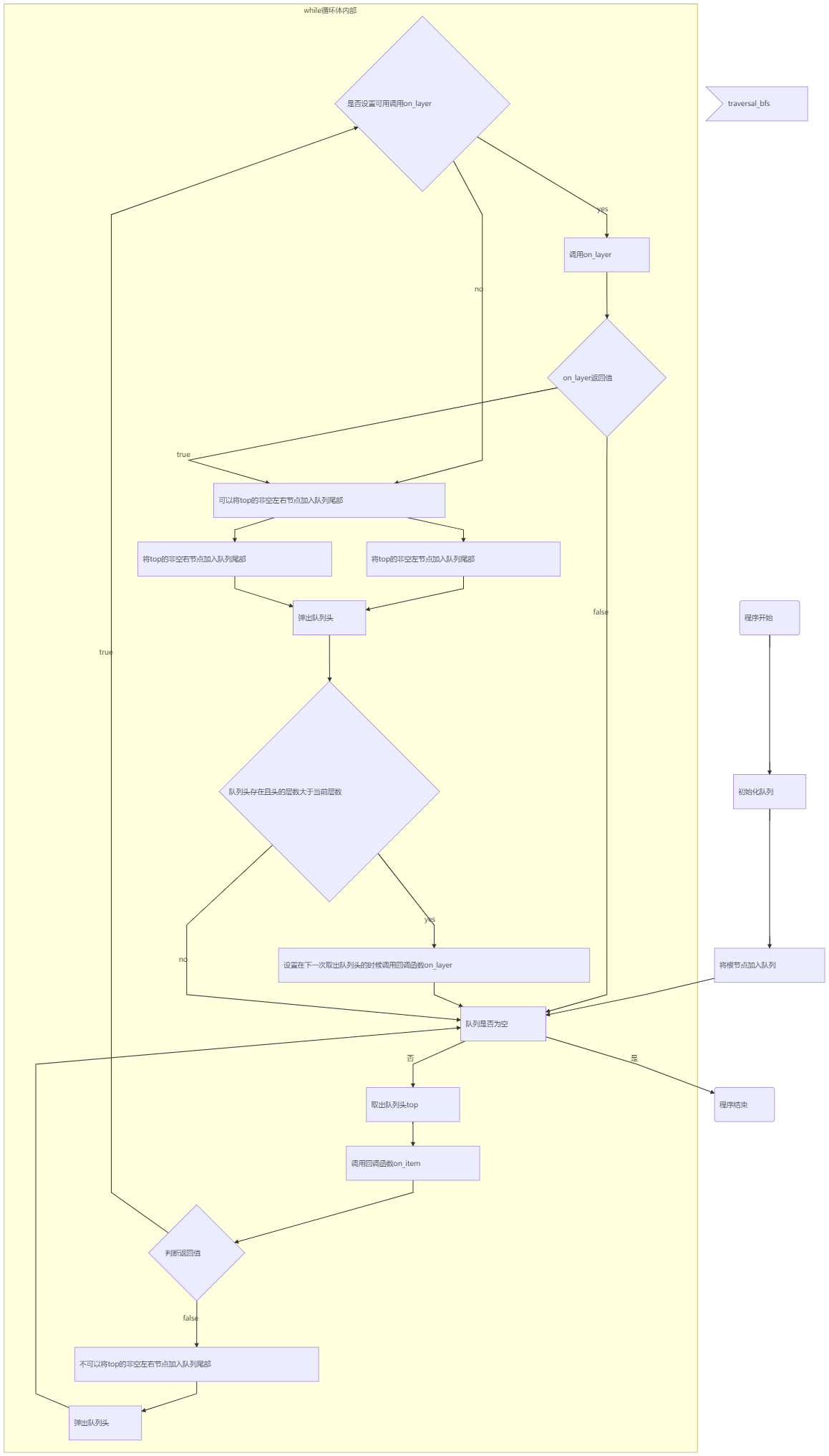
/\*!   
 \* 深度优先遍历（准备栈）   
 \* @tparam F bool(btree<T>&, vector<btree<T>>&)   
 \* @param callback 回调函数   
 \*/   
template<typename F>   
constexpr void traversal\_dfs(const traversal\_type method, F const &callback) {   
 vector <btree<T>> stack;   
 // 利用闭包特性传递栈信息   
 auto wrapper = [&stack, &callback](btree<T> &tr) -> bool {   
 return callback(tr, stack);   
 };   
 traversal(method, wrapper, &stack);   
}   
/\*!   
 \* 按照指定顺序遍历树   
 \* @tparam F   
 \* @param method [preorder, inorder, postoder]   
 \* @param callback bool(btree<T> &)   
 \* @return   
 \*/   
template<typename F>   
constexpr void traversal(const traversal\_type &method, F const &callback, vector <btree<T>> \*stack = nullptr) {   
 if (stack != nullptr) stack->emplace\_back(\*this);   
 if (method == preother) {   
 if (!callback(\*this)) return;   
 if (child\_left != nullptr) child\_left->traversal(method, callback);   
 if (child\_right != nullptr) child\_right->traversal(method, callback);   
 } else if (method == inorder) {   
 if (child\_left != nullptr) child\_left->traversal(method, callback);   
 if (!callback(\*this)) return;   
 if (child\_right != nullptr) child\_right->traversal(method, callback);   
 } else if (method == postoder) {   
 if (child\_left != nullptr) child\_left->traversal(method, callback);   
 if (child\_right != nullptr) child\_right->traversal(method, callback);   
 if (!callback(\*this)) return;   
 }   
 if (stack != nullptr) stack->pop\_back();   
}   
/\*!   
 \* 广度优先遍历   
 \* @tparam F1 bool(btree<T>&, bool\*)   
 \* @tparam F2 bool(btree<T>&, int&, bool\*)   
 \* @param tree 目标树   
 \* @param on\_item 每个元素的回调   
 \* @param on\_layer 每一层的回调   
 \* @param includes\_null=false 是否在元素回调中包含空节点   
 \*/   
template<typename F1, typename F2>   
static void   
traversal\_bfs(shared\_ptr<btree> &tree, F1 const &on\_item, F2 const &on\_layer, const bool includes\_null = false) {   
 if (tree == nullptr) return;   
 vector<shared\_ptr<btree> \*> q;   
 int height = 0;   
 tree->height = height;   
 shared\_ptr<btree> \*tree\_temp = &tree;   
 q.emplace\_back(tree\_temp);   
 bool new\_layer = false;   
 bool end\_all = false;   
 while (!q.empty()) {   
 shared\_ptr<btree> \*top = q.front();   
 if (!on\_item(\*top, &end\_all)) {   
 if (end\_all) return;   
 q.pop\_front();   
 continue;   
 }   
 if (end\_all) return;   
 if (new\_layer) {   
 if (!on\_layer(\*top, height + 1, &end\_all)) {   
 if (end\_all) return;   
 continue;   
 }   
 new\_layer = false;   
 if (end\_all) return;   
 }   
 if (includes\_null && \*top == nullptr) {   
 q.pop\_front();   
 continue;   
 }   
 shared\_ptr<btree> &left = (\*top)->get\_left(), &right = (\*top)->get\_right();   
 if ((\*top)->height > height) height = (\*top)->height;   
 auto push\_if = [&q, &includes\_null, &height](shared\_ptr<btree> &child) {   
 if (child != nullptr || includes\_null) {   
 shared\_ptr<btree> \*tmp = &child;   
 if (!includes\_null) (\*tmp)->height = height + 1;   
 q.emplace\_back(tmp);   
 }   
 };   
 push\_if(left);   
 push\_if(right);   
 q.pop\_front();   
 if (!q.empty()) {   
 // 即将到下一层   
 if (!includes\_null && (\*q[0])->height > height) {   
 new\_layer = true;   
 }   
 }   
 }   
}

## 2.3 程序整体流程

深度遍历操作（含栈）



广度优先遍历



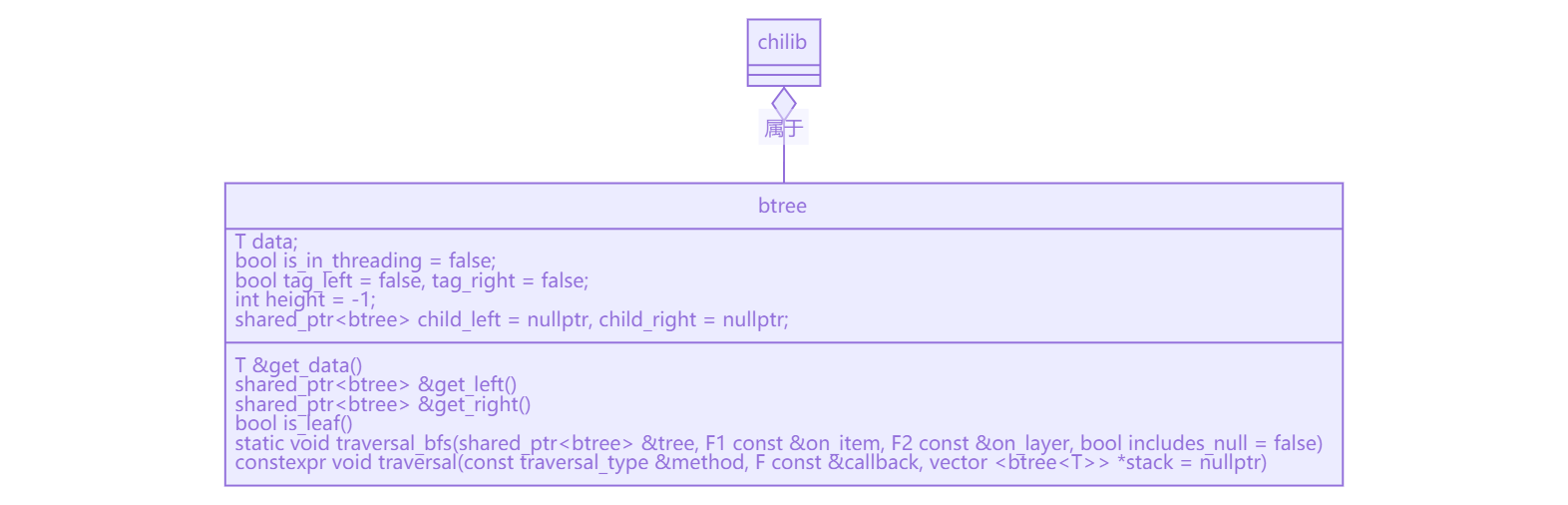
# 三、用户手册

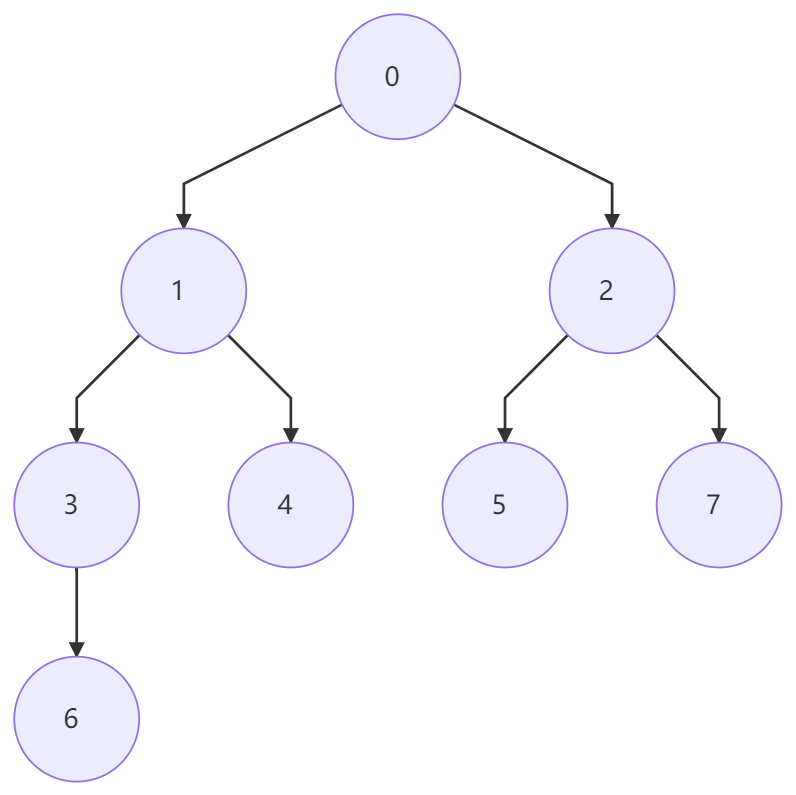
## 用户手册

chilib是本人在数据结构课中，尝试使用C++语言和自己有限的知识实现C++库中部分STL库的功能的一个库。以下是chilib::btree的使用说明。

### 二叉树结构

本程序的二叉树数据结构以及对应类方法表示如下：





### 获取二叉树信息

// 获取节点数据  
T &get\_data() { return data; }  
// 获取左右子树  
shared\_ptr<btree> &get\_left() { return child\_left; }  
shared\_ptr<btree> &get\_right() { return child\_right; }  
// 是否线索化  
bool &get\_is\_threading() { return is\_in\_threading; }  
// 线索化标志  
bool &get\_tag\_left() { return tag\_left; }  
bool &get\_tag\_right() { return tag\_right; }  
// 获取当前高度  
int &get\_height() { return height; }  
// 是否是叶子节点  
bool is\_leaf();

### 建立二叉树

// 用 std::make\_shared 建立节点  
root = std::make\_shared<btree<int>>(1);  
// 用遍历方法建树  
chilib::vector<int> line{1, 2, 3, 5, 6};  
auto p = line.begin();  
bt::traversal\_bfs(root, [&p, &line](std::shared\_ptr<bt> &tr, bool \*end\_all) -> bool {  
 // 在 includes\_null 情况下，需要设置 end\_all 结束遍历  
 if (tr != nullptr) return true;  
 tr = std::make\_shared<bt>(\*p);  
 ++p;  
 if (p == line.end()) {  
 \*end\_all = true;  
 return false;  
 }  
 return true;  
}, [](std::shared\_ptr<bt> &tr, int height, bool \*end\_all) -> bool {  
 // 这个函数是当遍历完一层之后触发的，height 表示下一层的高度  
 // printf("height now: %d\n", height);  
 return true;  
 // includes\_null 表示在遇到空节点的时候是否继续运行，设置为 true 则回调函数可以得到空节点的指针  
}, true);

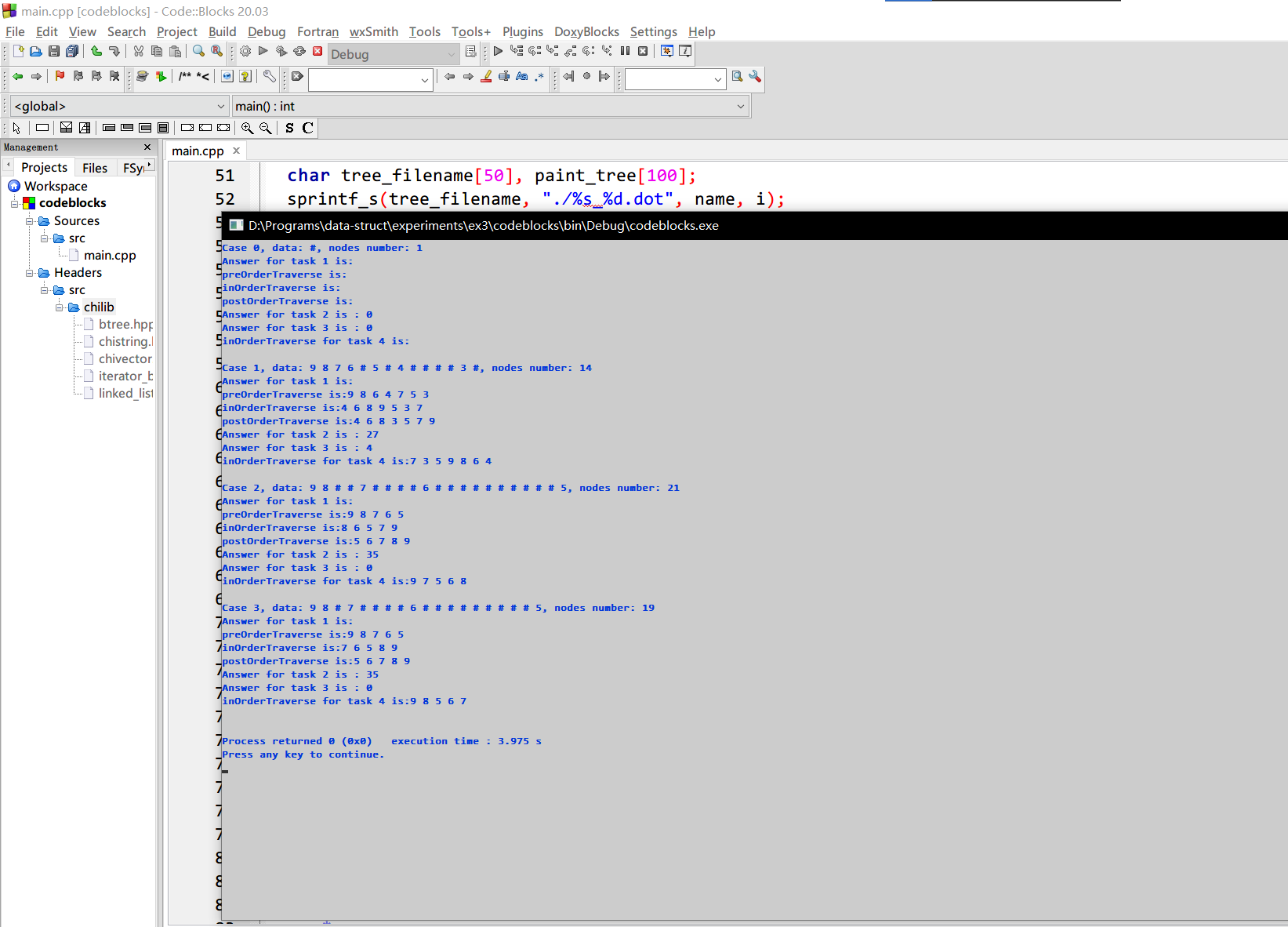
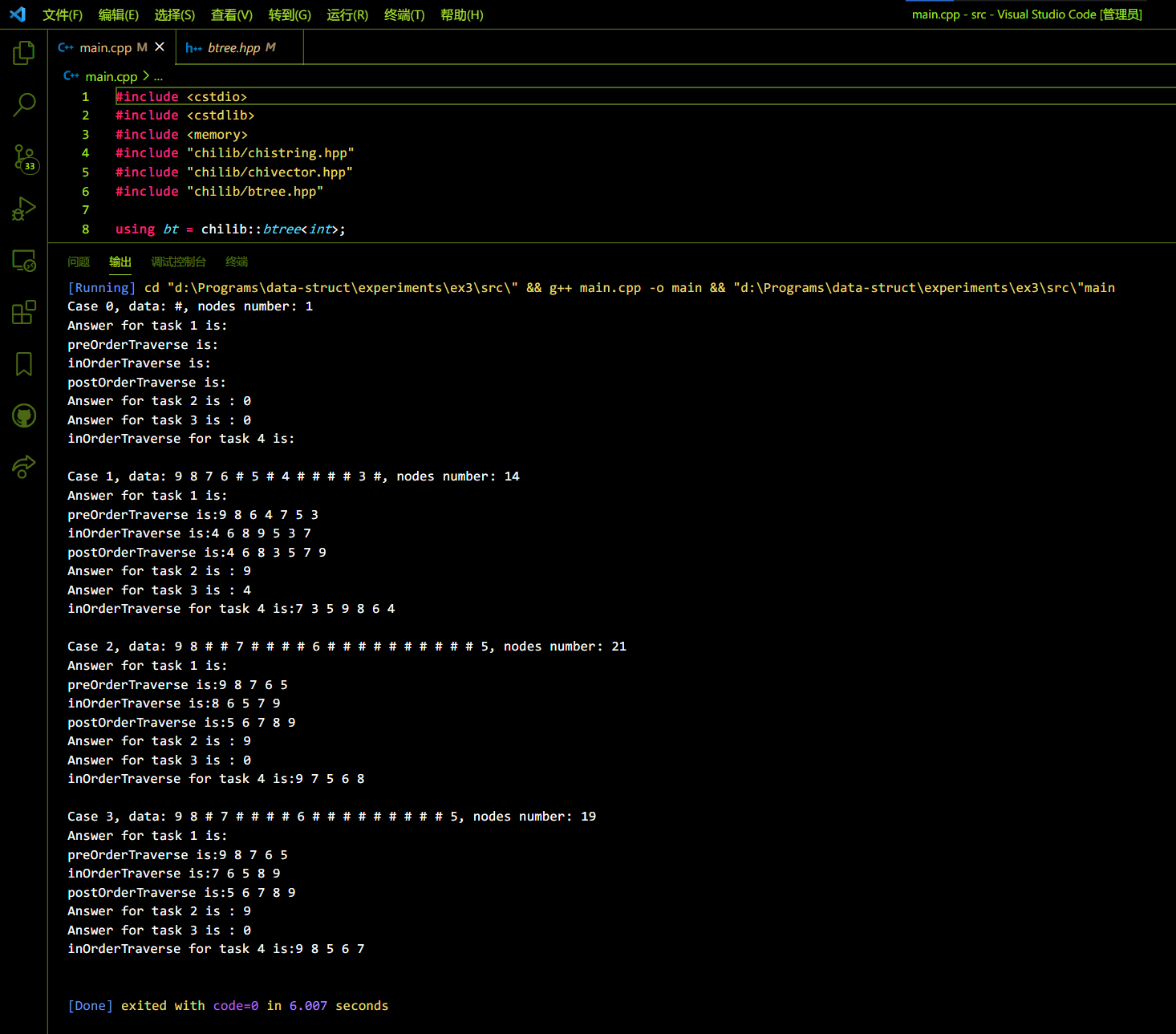
### 遍历二叉树

// 前序、中序、后序遍历  
auto print\_order = [](std::shared\_ptr<btree<int>> &tr) -> bool {  
 // tr 就是当前遍历到的节点  
 return true;  
}  
root->traversal\_preorder(print\_order);  
root->traversal\_inorder(print\_order);  
root->traversal\_postorder(print\_order);  
// 深度优先遍历，会额外传一个遍历经过的栈  
root->traversal\_dfs([](bt &tr, chilib::vector<bt> &stack) -> bool {  
 // 遇到叶子节点就向上计算经过的栈  
 if (tr.is\_leaf()) {  
 printf("visit stack: ");  
 for (auto &t : stack) printf("%d ", t.get\_data());  
 puts("");  
 }  
 return true;  
});  
// 宽度优先遍历  
btree<int>::traversal\_bfs(root,  
 [](std::shared\_ptr<bt> &tr, bool \*) -> bool {  
 // includes\_null 默认为否，此时返回 false 会终止 bfs 向下继续遍历  
 return true;  
 },  
 [](std::shared\_ptr<bt> &tr, int height, bool \*end\_all) -> bool { return true; }  
 );

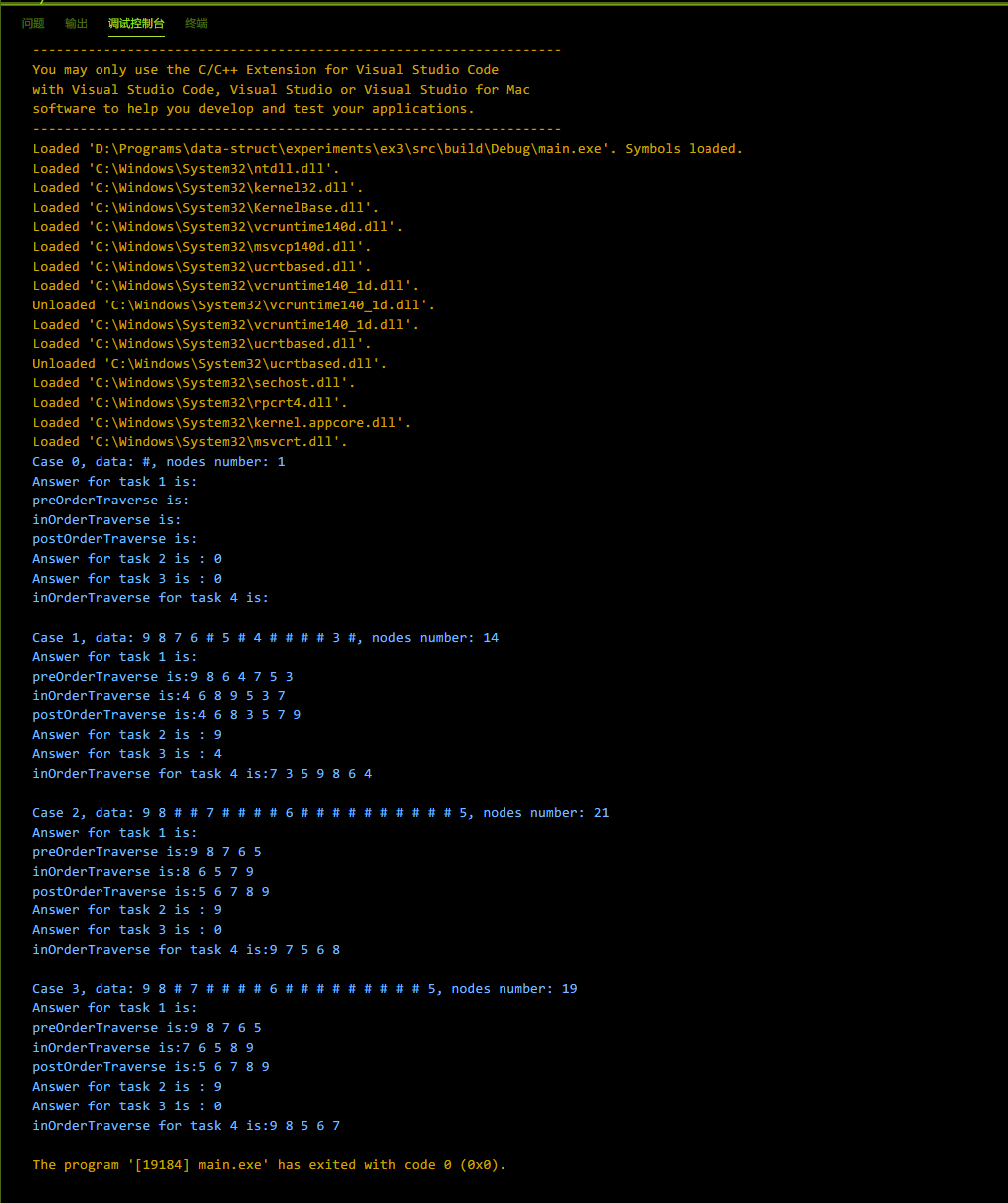
### 删除二叉树

root->traversal\_preorder([](btree<int> &tr) -> bool {  
 if (tr.get\_data() == 9) {  
 // 删除一些节点  
 tr.get\_left() = nullptr;  
 tr.get\_right() = nullptr;  
 }  
 return true;  
});  
// 删除整个二叉树  
root = nullptr;

# 四、结果

程序正确运行的结果截图。

使用MSVC19编译和运行：



# 五、总结

学习了数据结构中非常重要的树形结构的知识，熟练掌握了树形结构的建立和应用，并且在上一次实验的基础上编写出了可以迭代使用的代码，在软件开发等方面积累了经验。