二叉搜索树的实现

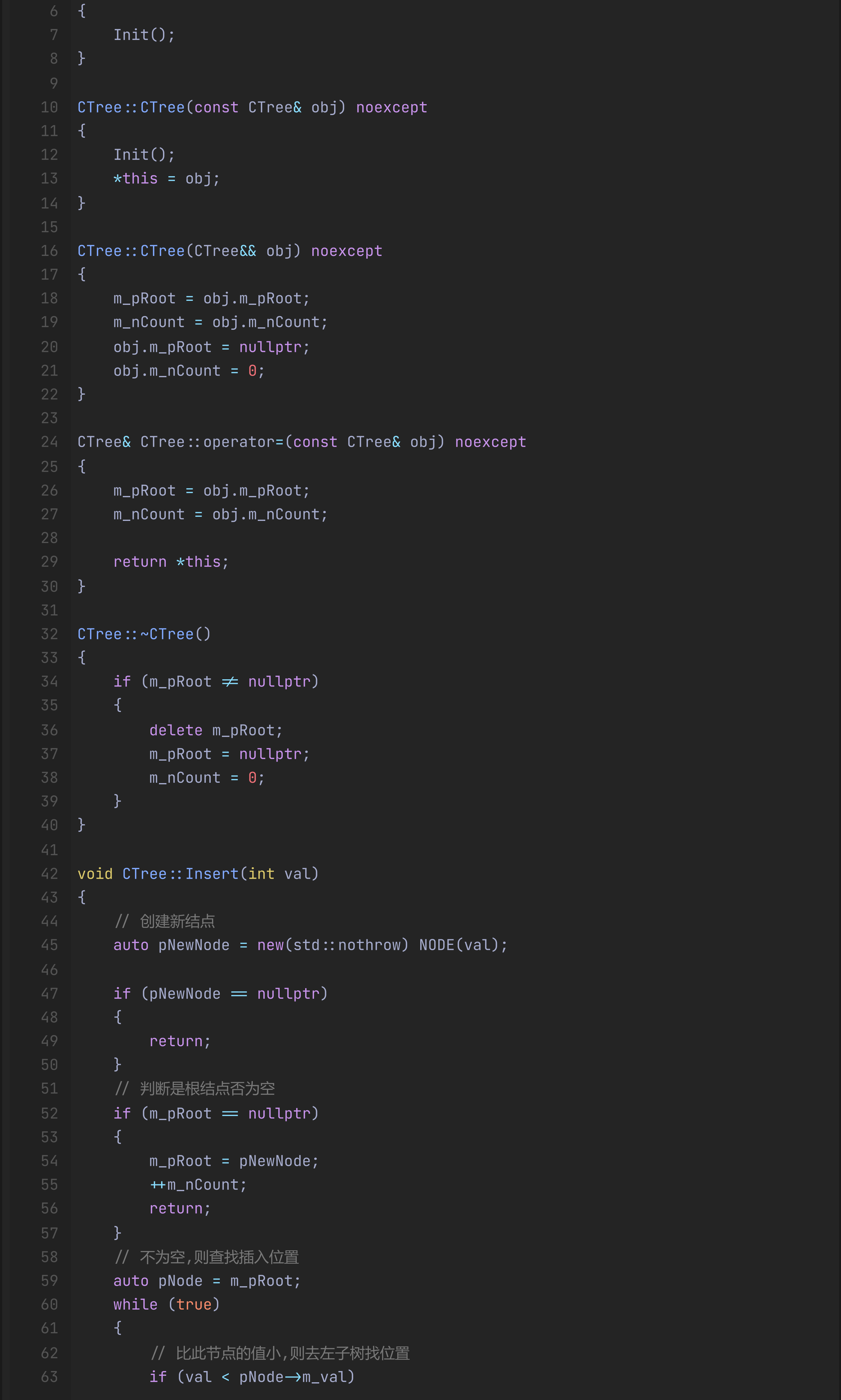
0.1 CTree.h

```cpp
#pragma once

class CTree
{
public:
    struct NODE// 结点
    {
        NODE() :m_pFather(nullptr), m_pLeft(nullptr), m_pRight(nullptr), m_val{} {}
        NODE(int val) :m_pFather(nullptr), m_pLeft(nullptr), m_pRight(nullptr), m_val(val) {}
        int m_val;// 值
        NODE* m_pFather;// 父节点
        NODE* m_pLeft;// 左孩子
        NODE* m_pRight;// 右孩子
    };
    using PNODE = NODE*;
public:
    CTree();
    CTree(const CTree& obj) noexcept;
    CTree(CTree&& obj) noexcept;
    CTree& operator=(const CTree& obj) noexcept;
    virtual ~CTree();

    void Insert(int val);// 插入
    void Delete(int val);// 删除
    bool Find(int val);// 查找
    void Modify(int nOld, int nNew);// 修改

    size_t GetCount() const;// 元素个数
    bool IsEmpty() const;// 是否为空

    void Mid();// 中序遍历
private:
    void MidNode(PNODE pNode);

    void Init();
    PNODE FindNode(int val);
    void DelLeaf(PNODE pLeafToDel);// 删除叶子
    void DelSingle(PNODE pSingleToDel);// 删除单分支
    void DelDouble(PNODE pDoubleToDel);// 删除双分支

    PNODE m_pRoot;// 根结点
    size_t m_nCount;// 结点的个数
};
```

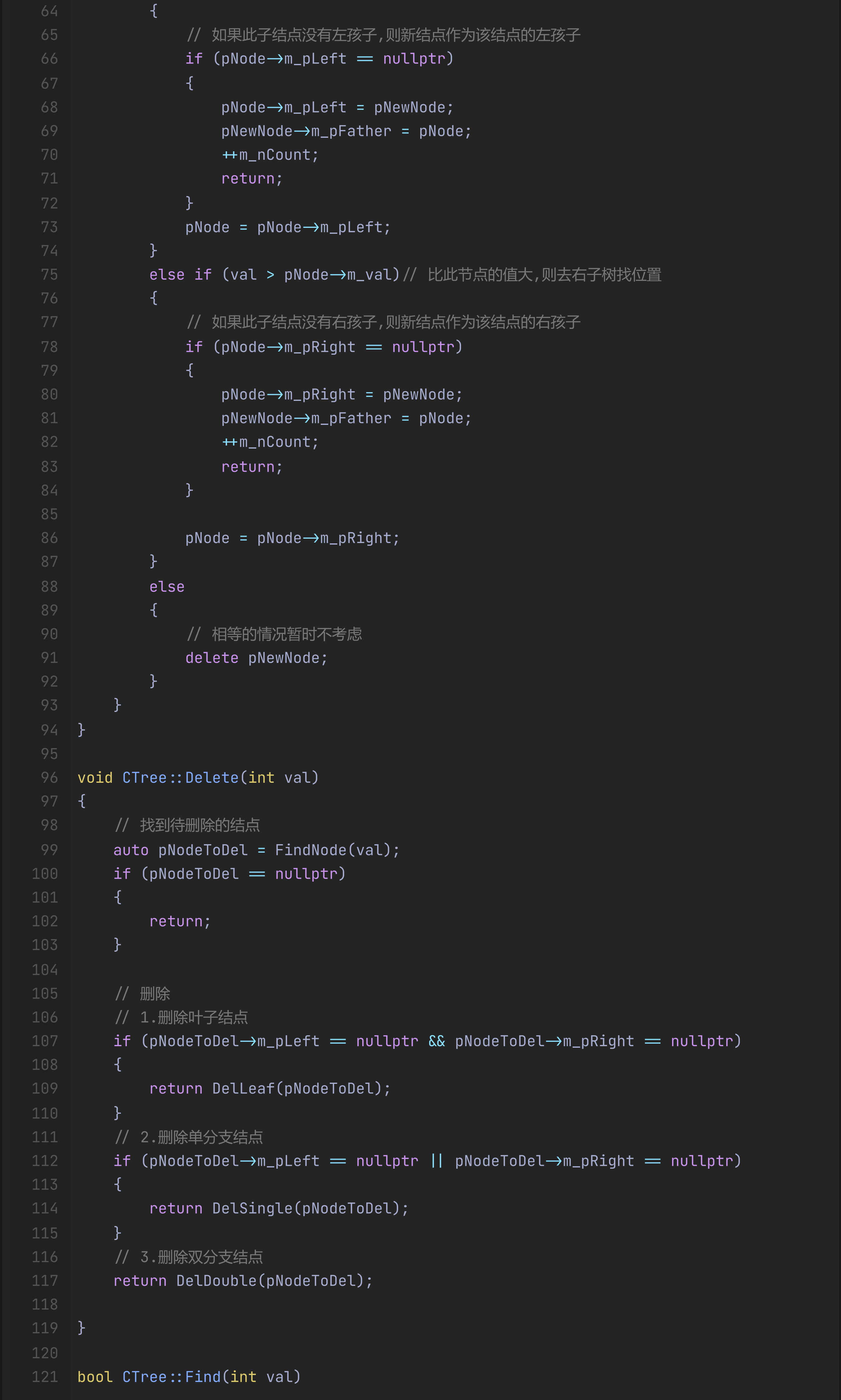
Fence 0-1

0.1 CTree.cpp

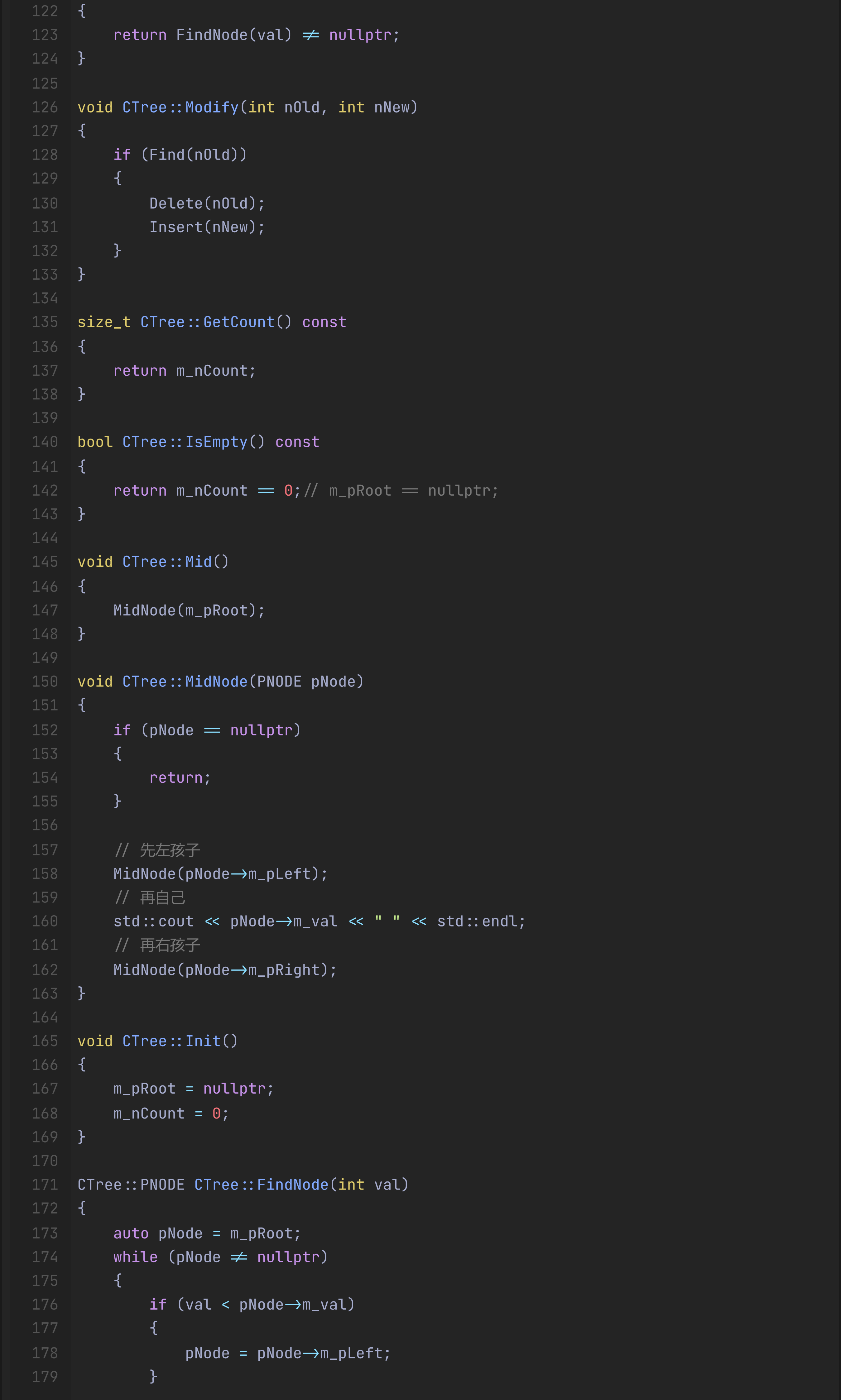
```cpp
#include "CTree.h"
#include <new>
#include <iostream>

CTree::CTree()
```



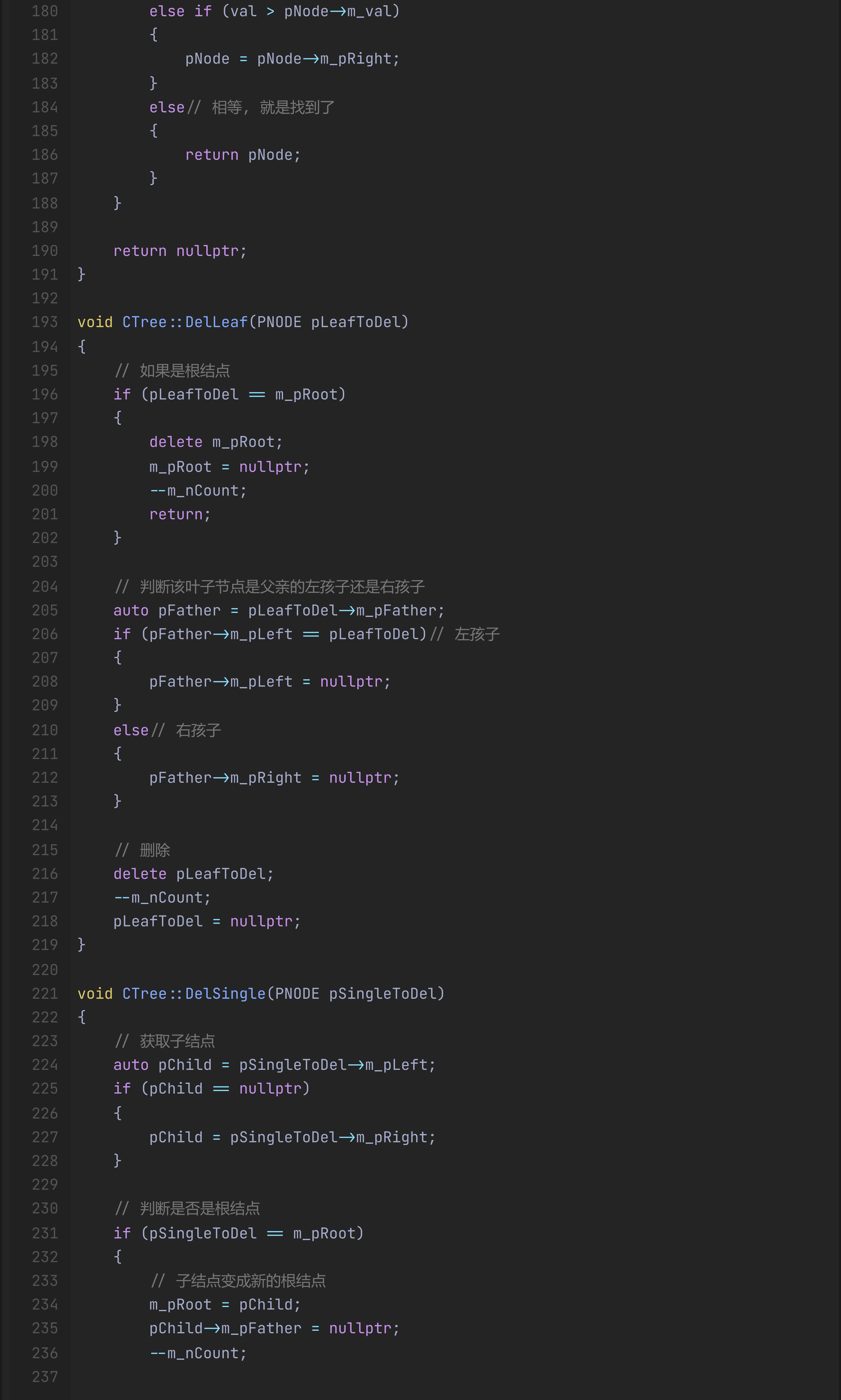
```cpp
{
    Init();
}

CTree::CTree(const CTree& obj) noexcept
{
    Init();
    *this = obj;
}

CTree::CTree(CTree&& obj) noexcept
{
    m_pRoot = obj.m_pRoot;
    m_nCount = obj.m_nCount;
    obj.m_pRoot = nullptr;
    obj.m_nCount = 0;
}

CTree& CTree::operator=(const CTree& obj) noexcept
{
    m_pRoot = obj.m_pRoot;
    m_nCount = obj.m_nCount;

    return *this;
}

CTree::~CTree()
{
    if (m_pRoot != nullptr)
    {
        delete m_pRoot;
        m_pRoot = nullptr;
        m_nCount = 0;
    }
}

void CTree::Insert(int val)
{
    // 创建新结点
    auto pNewNode = new(std::nothrow) NODE(val);

    if (pNewNode == nullptr)
    {
        return;
    }
    // 判断是根结点否为空
    if (m_pRoot == nullptr)
    {
        m_pRoot = pNewNode;
        ++m_nCount;
        return;
    }
    // 不为空,则查找插入位置
    auto pNode = m_pRoot;
    while (true)
    {
        // 比此节点的值小,则去左子树找位置
        if (val < pNode->m_val)
```



```cpp
        {
            // 如果此子结点没有左孩子,则新结点作为该结点的左孩子
            if (pNode->m_pLeft == nullptr)
            {
                pNode->m_pLeft = pNewNode;
                pNewNode->m_pFather = pNode;
                ++m_nCount;
                return;
            }
            pNode = pNode->m_pLeft;
        }
        else if (val > pNode->m_val)// 比此节点的值大,则去右子树找位置
        {
            // 如果此子结点没有右孩子,则新结点作为该结点的右孩子
            if (pNode->m_pRight == nullptr)
            {
                pNode->m_pRight = pNewNode;
                pNewNode->m_pFather = pNode;
                ++m_nCount;
                return;
            }

            pNode = pNode->m_pRight;
        }
        else
        {
            // 相等的情况暂时不考虑
            delete pNewNode;
        }
    }
}

void CTree::Delete(int val)
{
    // 找到待删除的结点
    auto pNodeToDel = FindNode(val);
    if (pNodeToDel == nullptr)
    {
        return;
    }

    // 删除
    // 1.删除叶子结点
    if (pNodeToDel->m_pLeft == nullptr && pNodeToDel->m_pRight == nullptr)
    {
        return DelLeaf(pNodeToDel);
    }
    // 2.删除单分支结点
    if (pNodeToDel->m_pLeft == nullptr || pNodeToDel->m_pRight == nullptr)
    {
        return DelSingle(pNodeToDel);
    }
    // 3.删除双分支结点
    return DelDouble(pNodeToDel);

}

bool CTree::Find(int val)
```



```cpp
{
    return FindNode(val) != nullptr;
}

void CTree::Modify(int nOld, int nNew)
{
    if (Find(nOld))
    {
        Delete(nOld);
        Insert(nNew);
    }
}

size_t CTree::GetCount() const
{
    return m_nCount;
}

bool CTree::IsEmpty() const
{
    return m_nCount == 0;// m_pRoot == nullptr;
}

void CTree::Mid()
{
    MidNode(m_pRoot);
}

void CTree::MidNode(PNODE pNode)
{
    if (pNode == nullptr)
    {
        return;
    }

    // 先左孩子
    MidNode(pNode->m_pLeft);
    // 再自己
    std::cout << pNode->m_val << " " << std::endl;
    // 再右孩子
    MidNode(pNode->m_pRight);
}

void CTree::Init()
{
    m_pRoot = nullptr;
    m_nCount = 0;
}

CTree::PNODE CTree::FindNode(int val)
{
    auto pNode = m_pRoot;
    while (pNode != nullptr)
    {
        if (val < pNode->m_val)
        {
            pNode = pNode->m_pLeft;
        }
```



```cpp
        else if (val > pNode->m_val)
        {
            pNode = pNode->m_pRight;
        }
        else// 相等，就是找到了
        {
            return pNode;
        }
    }

    return nullptr;
}

void CTree::DelLeaf(PNODE pLeafToDel)
{
    // 如果是根结点
    if (pLeafToDel == m_pRoot)
    {
        delete m_pRoot;
        m_pRoot = nullptr;
        --m_nCount;
        return;
    }

    // 判断该叶子节点是父亲的左孩子还是右孩子
    auto pFather = pLeafToDel->m_pFather;
    if (pFather->m_pLeft == pLeafToDel)// 左孩子
    {
        pFather->m_pLeft = nullptr;
    }
    else// 右孩子
    {
        pFather->m_pRight = nullptr;
    }

    // 删除
    delete pLeafToDel;
    --m_nCount;
    pLeafToDel = nullptr;
}

void CTree::DelSingle(PNODE pSingleToDel)
{
    // 获取子结点
    auto pChild = pSingleToDel->m_pLeft;
    if (pChild == nullptr)
    {
        pChild = pSingleToDel->m_pRight;
    }

    // 判断是否是根结点
    if (pSingleToDel == m_pRoot)
    {
        // 子结点变成新的根结点
        m_pRoot = pChild;
        pChild->m_pFather = nullptr;
        --m_nCount;

        // 删除原来的根结点
        delete pSingleToDel;
        pSingleToDel = nullptr;
        return;
    }

    // 判断待删除的单分支结点是父亲的左孩子还是右孩子
    auto pFather = pSingleToDel->m_pFather;
    if (pFather->m_pLeft == pSingleToDel)
    {
        // 删除的分支结点是父亲的左孩子,子节点提升为老父亲的左孩子
        pFather->m_pLeft = pChild;
    }
    else
    {
        // 删除的分支结点是父亲的右孩子,子节点提升为老父亲的右孩子
        pFather->m_pRight = pChild;
    }
    // 删除
    pChild->m_pFather = pFather;
    delete pSingleToDel;
    --m_nCount;
    return;
}

void CTree::DelDouble(PNODE pDoubleToDel)
{
    // 查找左子树中的最大值,沿着左子树的右孩子一直找
    auto pMaxInLeft = pDoubleToDel->m_pLeft;
    while (pMaxInLeft->m_pRight != nullptr)
    {
        pMaxInLeft = pMaxInLeft->m_pRight;
    }
    // 提值到上面
    pDoubleToDel->m_val = pMaxInLeft->m_val;

    // 删除
// 1.删除叶子结点
    if (pMaxInLeft->m_pLeft == nullptr && pMaxInLeft->m_pRight == nullptr)
    {
        return DelLeaf(pMaxInLeft);
    }
    // 2.删除单分支结点
    if (pMaxInLeft->m_pLeft == nullptr || pMaxInLeft->m_pRight == nullptr)
    {
        return DelSingle(pMaxInLeft);
    }
}

```

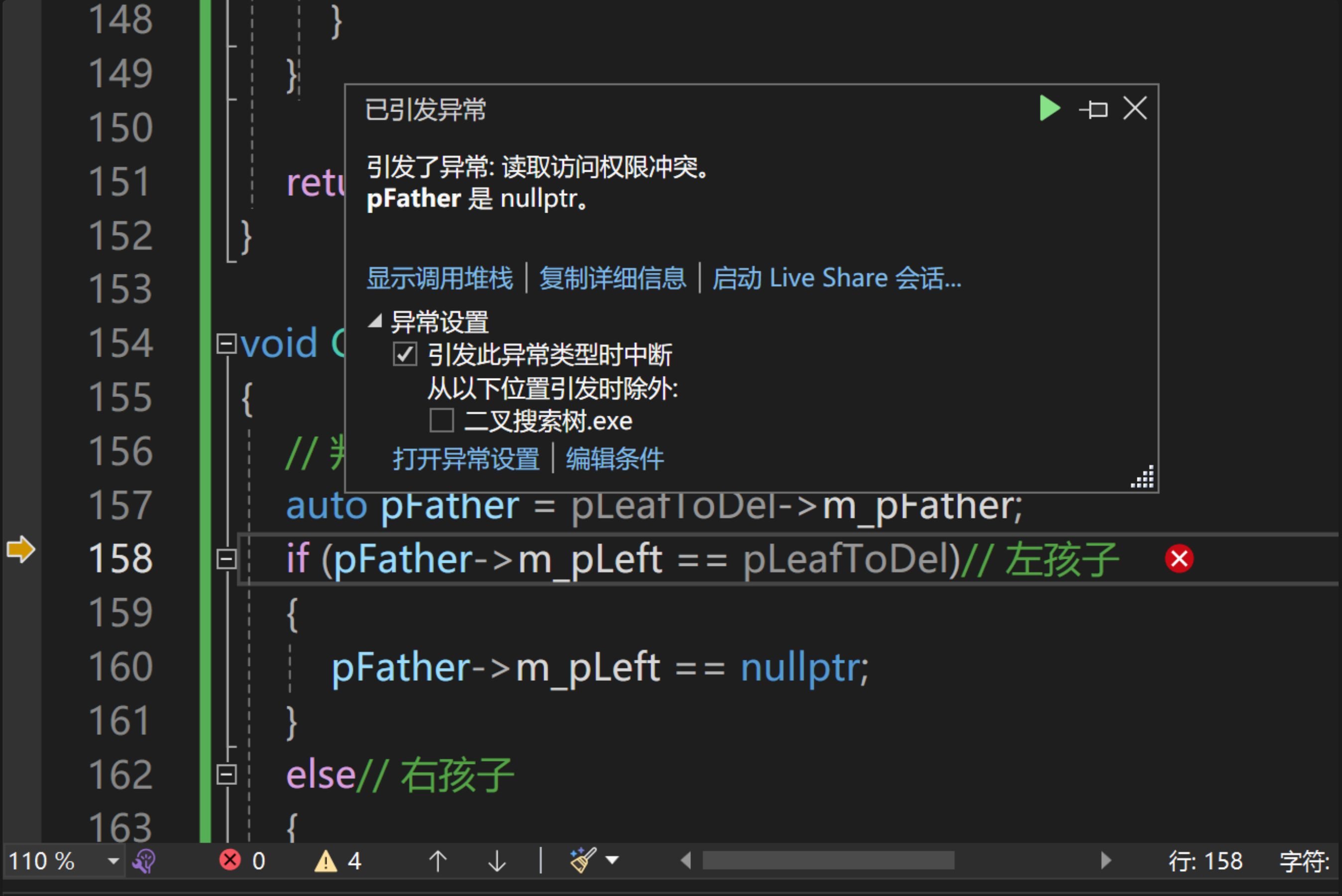
Fence 0-2

0.1 二叉搜索树.cpp

```
// 二叉搜索树.cpp

#include <iostream>
#include "CTree.h"
using namespace std;
```

```cpp

int main()
{
    CTree tr;
    tr.Insert(12);
    tr.Insert(5);
    tr.Insert(18);
    tr.Insert(3);
    tr.Insert(8);
    tr.Insert(15);
    tr.Insert(25);
    tr.Insert(7);
    tr.Insert(11);
    tr.Insert(20);

    tr.Mid();

    tr.Delete(8);
    tr.Delete(12);

#if 0
    tr.Delete(25);

    CTree tr0;
    tr0.Insert(8);
    tr0.Insert(9);
    tr0.Delete(8);

    CTree tr1;
    tr1.Insert(10);
    tr1.Insert(9);
    tr1.Delete(10);
#endif // 0

#if 0
    tr.Delete(20);

    CTree tr0;
    tr0.Insert(20);
    tr0.Delete(20);

    bool bRes = tr.Find(7);
    bRes = tr.Find(88);
    bRes = tr.Find(25);
#endif // 0

    return 0;
}
```

Fence 0-3



```
            }
        }

        retu
    }

void C
{
    // 判
    auto pFather = pLeafToDel->m_pFather;
    if (pFather->m_pLeft == pLeafToDel)// 左孩子
    {
        pFather->m_pLeft == nullptr;
    }
    else// 右孩子
    {
```

已引发异常

引发了异常: 读取访问权限冲突。
pFather 是 nullptr。

显示调用堆栈 | 复制详细信息 | 启动 Live Share 会话...

◢ 异常设置

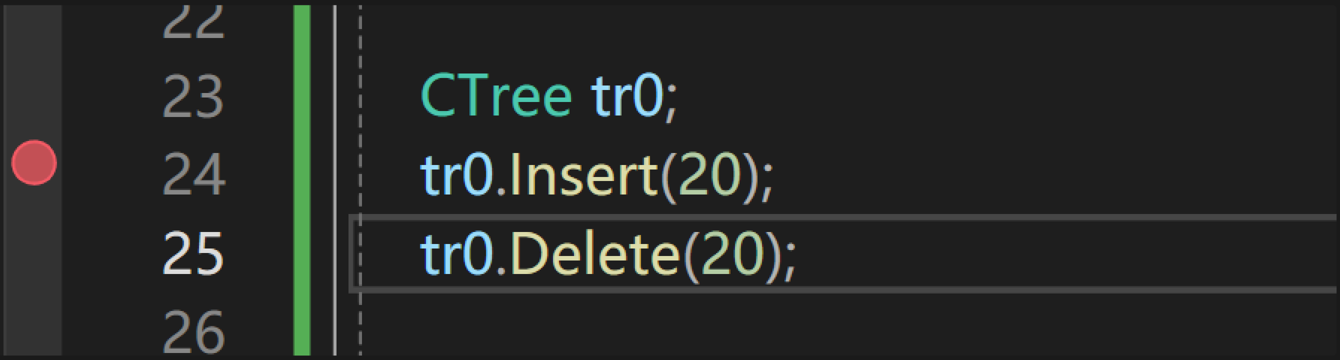
☑ 引发此异常类型时中断

从以下位置引发时除外:

□ 二叉搜索树.exe

打开异常设置 | 编辑条件

Figure 0-1



```
CTree tr0;
tr0.Insert(20);
tr0.Delete(20);
```

Figure 0-2

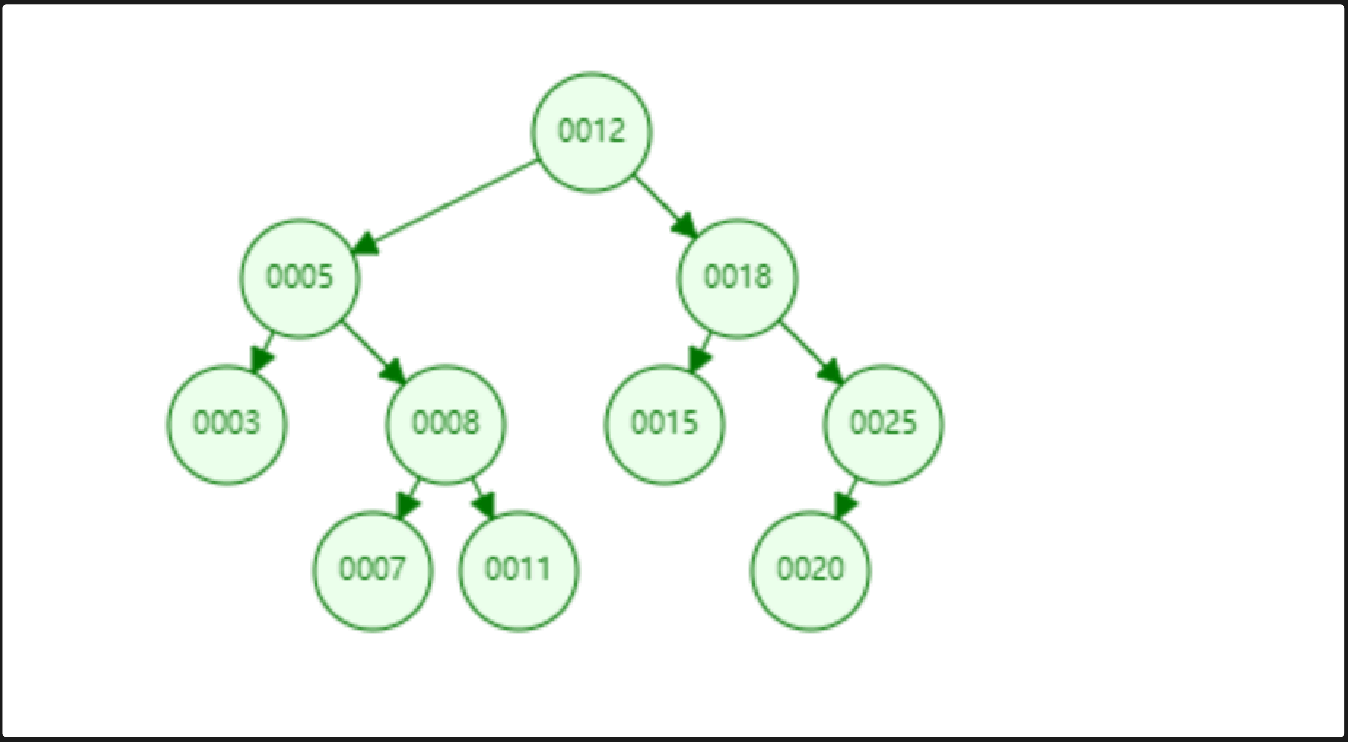


Figure 0-3

1. **二叉搜索树的各个操作的时间复杂度是logn**

二叉搜索树的遍历

中序:先左孩子,再自己,再右孩子

前序:先自己,再左孩子,再右孩子

后序:先左孩子,再右孩子,再自己

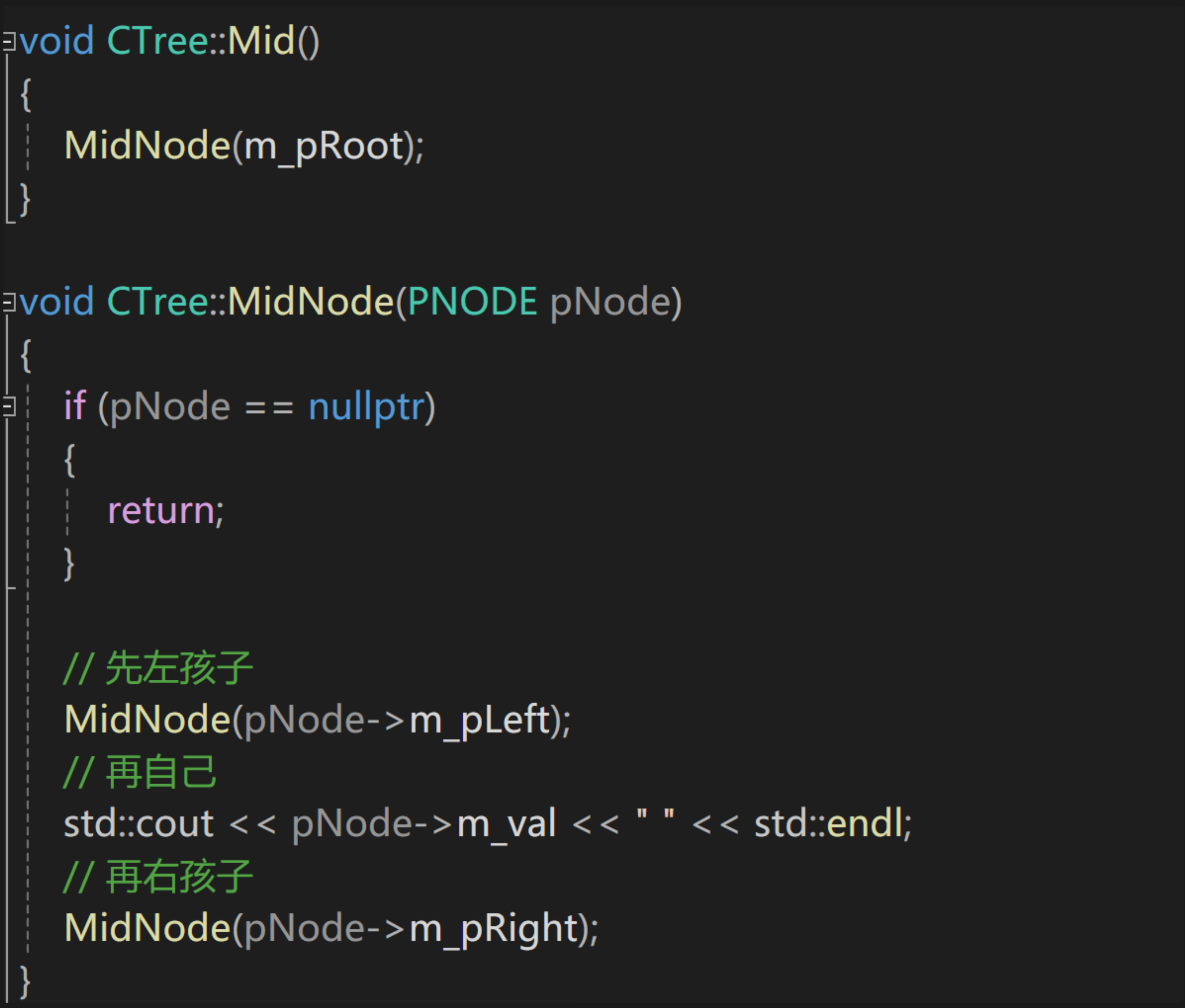
层序:一层层的

逆中序:先右孩子,再自己,再左孩子

逆前序:先自己,再右孩子,再左孩子

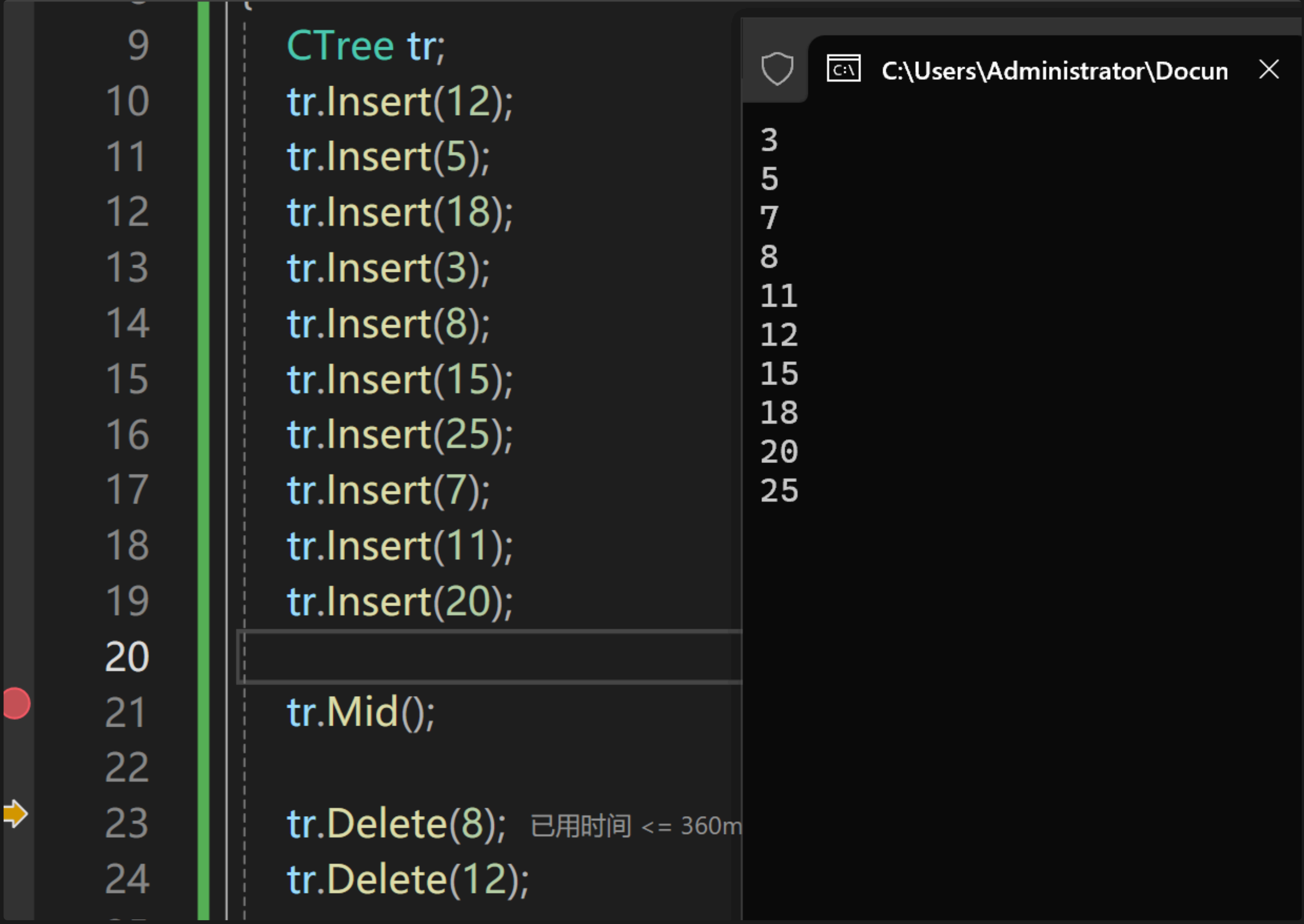
逆后序:先右孩子,再左孩子,再自己

1. 中序



```
void CTree::Mid()
{
    MidNode(m_pRoot);
}

void CTree::MidNode(PNODE pNode)
{
    if (pNode == nullptr)
    {
        return;
    }

    // 先左孩子
    MidNode(pNode->m_pLeft);
    // 再自己
    std::cout << pNode->m_val << " " << std::endl;
    // 再右孩子
    MidNode(pNode->m_pRight);
}
```

Figure 1-1



```
    CTree tr;
    tr.Insert(12);
    tr.Insert(5);
    tr.Insert(18);
    tr.Insert(3);
    tr.Insert(8);
    tr.Insert(15);
    tr.Insert(25);
    tr.Insert(7);
    tr.Insert(11);
    tr.Insert(20);

    tr.Mid();

    tr.Delete(8);  已用时间 <= 360m
    tr.Delete(12);
```

```
3
5
7
8
11
12
15
18
20
25
```

Figure 1-2