

# 二叉搜索树的实现

## 0.1 CTree.h

```
1  #pragma once
2
3  class CTree
4  {
5  public:
6      struct NODE// 结点
7      {
8          NODE() :m_pFather(nullptr), m_pLeft(nullptr), m_pRight(nullptr), m_val{} {}
9          NODE(int val) :m_pFather(nullptr), m_pLeft(nullptr), m_pRight(nullptr),
10             m_val(val) {}
11             int m_val;// 值
12             NODE* m_pFather;// 父节点
13             NODE* m_pLeft;// 左孩子
14             NODE* m_pRight;// 右孩子
15         };
16         using PNODE = NODE*;
17     public:
18         CTree();
19         CTree(const CTree& obj) noexcept;
20         CTree(CTree&& obj) noexcept;
21         CTree& operator=(const CTree& obj) noexcept;
22         virtual ~CTree();
23
24         void Insert(int val);// 插入
25         void Delete(int val);// 删除
26         bool Find(int val);// 查找
27         void Modify(int nOld, int nNew);// 修改
28
29         size_t GetCount() const;// 元素个数
30         bool IsEmpty() const;// 是否为空
31
32         void Mid();// 中序遍历
33     private:
34         void MidNode(PNODE pNode);
35
36         void Init();
37         PNODE FindNode(int val);
38         void DelLeaf(PNODE pLeafToDel);// 删除叶子
39         void DelSingle(PNODE pSingleToDel);// 删除单分支
40         void DelDouble(PNODE pDoubleToDel);// 删除双分支
41
42         PNODE m_pRoot;// 根结点
43         size_t m_nCount;// 结点的个数
44     };
```

Fence 0-1

## 0.1 CTree.cpp

```
1  #include "CTree.h"
2  #include <new>
3  #include <iostream>
4
5  CTree::CTree()
```

```
6 {
7     Init();
8 }
9
10 CTree::CTree(const CTree& obj) noexcept
11 {
12     Init();
13     *this = obj;
14 }
15
16 CTree::CTree(CTree&& obj) noexcept
17 {
18     m_pRoot = obj.m_pRoot;
19     m_nCount = obj.m_nCount;
20     obj.m_pRoot = nullptr;
21     obj.m_nCount = 0;
22 }
23
24 CTree& CTree::operator=(const CTree& obj) noexcept
25 {
26     m_pRoot = obj.m_pRoot;
27     m_nCount = obj.m_nCount;
28
29     return *this;
30 }
31
32 CTree::~CTree()
33 {
34     if (m_pRoot != nullptr)
35     {
36         delete m_pRoot;
37         m_pRoot = nullptr;
38         m_nCount = 0;
39     }
40 }
41
42 void CTree::Insert(int val)
43 {
44     // 创建新结点
45     auto pNewNode = new(std::nothrow) NODE(val);
46
47     if (pNewNode == nullptr)
48     {
49         return;
50     }
51     // 判断是根结点否为空
52     if (m_pRoot == nullptr)
53     {
54         m_pRoot = pNewNode;
55         ++m_nCount;
56         return;
57     }
58     // 不为空,则查找插入位置
59     auto pNode = m_pRoot;
60     while (true)
61     {
62         // 比此节点的值小,则去左子树找位置
63         if (val < pNode->m_val)
```

```

64     {
65         // 如果此子结点没有左孩子,则新结点作为该结点的左孩子
66         if (pNode->m_pLeft == nullptr)
67         {
68             pNode->m_pLeft = pNewNode;
69             pNewNode->m_pFather = pNode;
70             ++m_nCount;
71             return;
72         }
73         pNode = pNode->m_pLeft;
74     }
75     else if (val > pNode->m_val) // 比此节点的值大,则去右子树找位置
76     {
77         // 如果此子结点没有右孩子,则新结点作为该结点的右孩子
78         if (pNode->m_pRight == nullptr)
79         {
80             pNode->m_pRight = pNewNode;
81             pNewNode->m_pFather = pNode;
82             ++m_nCount;
83             return;
84         }
85
86         pNode = pNode->m_pRight;
87     }
88     else
89     {
90         // 相等的情况暂时不考虑
91         delete pNewNode;
92     }
93 }
94 }
95
96 void CTree::Delete(int val)
97 {
98     // 找到待删除的结点
99     auto pNodeToDel = FindNode(val);
100     if (pNodeToDel == nullptr)
101     {
102         return;
103     }
104
105     // 删除
106     // 1.删除叶子结点
107     if (pNodeToDel->m_pLeft == nullptr && pNodeToDel->m_pRight == nullptr)
108     {
109         return DelLeaf(pNodeToDel);
110     }
111     // 2.删除单分支结点
112     if (pNodeToDel->m_pLeft == nullptr || pNodeToDel->m_pRight == nullptr)
113     {
114         return DelSingle(pNodeToDel);
115     }
116     // 3.删除双分支结点
117     return DelDouble(pNodeToDel);
118
119 }
120
121 bool CTree::Find(int val)

```

```

122 {
123     return FindNode(val) != nullptr;
124 }
125
126 void CTree::Modify(int nOld, int nNew)
127 {
128     if (Find(nOld))
129     {
130         Delete(nOld);
131         Insert(nNew);
132     }
133 }
134
135 size_t CTree::GetCount() const
136 {
137     return m_nCount;
138 }
139
140 bool CTree::IsEmpty() const
141 {
142     return m_nCount == 0; // m_pRoot == nullptr;
143 }
144
145 void CTree::Mid()
146 {
147     MidNode(m_pRoot);
148 }
149
150 void CTree::MidNode(PNODE pNode)
151 {
152     if (pNode == nullptr)
153     {
154         return;
155     }
156
157     // 先左孩子
158     MidNode(pNode->m_pLeft);
159     // 再自己
160     std::cout << pNode->m_val << " " << std::endl;
161     // 再右孩子
162     MidNode(pNode->m_pRight);
163 }
164
165 void CTree::Init()
166 {
167     m_pRoot = nullptr;
168     m_nCount = 0;
169 }
170
171 CTree::PNODE CTree::FindNode(int val)
172 {
173     auto pNode = m_pRoot;
174     while (pNode != nullptr)
175     {
176         if (val < pNode->m_val)
177         {
178             pNode = pNode->m_pLeft;
179         }

```

```

180         else if (val > pNode→m_val)
181         {
182             pNode = pNode→m_pRight;
183         }
184         else// 相等, 就是找到了
185         {
186             return pNode;
187         }
188     }
189
190     return nullptr;
191 }
192
193 void CTree::DelLeaf(PNODE pLeafToDel)
194 {
195     // 如果是根结点
196     if (pLeafToDel == m_pRoot)
197     {
198         delete m_pRoot;
199         m_pRoot = nullptr;
200         --m_nCount;
201         return;
202     }
203
204     // 判断该叶子节点是父亲的左孩子还是右孩子
205     auto pFather = pLeafToDel→m_pFather;
206     if (pFather→m_pLeft == pLeafToDel)// 左孩子
207     {
208         pFather→m_pLeft = nullptr;
209     }
210     else// 右孩子
211     {
212         pFather→m_pRight = nullptr;
213     }
214
215     // 删除
216     delete pLeafToDel;
217     --m_nCount;
218     pLeafToDel = nullptr;
219 }
220
221 void CTree::DelSingle(PNODE pSingleToDel)
222 {
223     // 获取子结点
224     auto pChild = pSingleToDel→m_pLeft;
225     if (pChild == nullptr)
226     {
227         pChild = pSingleToDel→m_pRight;
228     }
229
230     // 判断是否是根结点
231     if (pSingleToDel == m_pRoot)
232     {
233         // 子结点变成新的根结点
234         m_pRoot = pChild;
235         pChild→m_pFather = nullptr;
236         --m_nCount;
237

```

```

238         // 删除原来的根结点
239         delete pSingleToDel;
240         pSingleToDel = nullptr;
241         return;
242     }
243
244     // 判断待删除的单分支结点是父亲的左孩子还是右孩子
245     auto pFather = pSingleToDel->m_pFather;
246     if (pFather->m_pLeft == pSingleToDel)
247     {
248         // 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
249         pFather->m_pLeft = pChild;
250     }
251     else
252     {
253         // 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
254         pFather->m_pRight = pChild;
255     }
256     // 删除
257     pChild->m_pFather = pFather;
258     delete pSingleToDel;
259     --m_nCount;
260     return;
261 }
262
263 void CTree::DelDouble(PNODE pDoubleToDel)
264 {
265     // 查找左子树中的最大值,沿着左子树的右孩子一直找
266     auto pMaxInLeft = pDoubleToDel->m_pLeft;
267     while (pMaxInLeft->m_pRight != nullptr)
268     {
269         pMaxInLeft = pMaxInLeft->m_pRight;
270     }
271     // 提值到上面
272     pDoubleToDel->m_val = pMaxInLeft->m_val;
273
274     // 删除
275     // 1.删除叶子结点
276     if (pMaxInLeft->m_pLeft == nullptr && pMaxInLeft->m_pRight == nullptr)
277     {
278         return DelLeaf(pMaxInLeft);
279     }
280     // 2.删除单分支结点
281     if (pMaxInLeft->m_pLeft == nullptr || pMaxInLeft->m_pRight == nullptr)
282     {
283         return DelSingle(pMaxInLeft);
284     }
285 }
286

```

Fence 0-2

## 0.1 二叉搜索树.cpp

```

1 // 二叉搜索树.cpp
2
3 #include <iostream>
4 #include "CTree.h"
5 using namespace std;

```

```

6
7  int main()
8  {
9      CTree tr;
10     tr.Insert(12);
11     tr.Insert(5);
12     tr.Insert(18);
13     tr.Insert(3);
14     tr.Insert(8);
15     tr.Insert(15);
16     tr.Insert(25);
17     tr.Insert(7);
18     tr.Insert(11);
19     tr.Insert(20);
20
21     tr.Mid();
22
23     tr.Delete(8);
24     tr.Delete(12);
25
26     #if 0
27     tr.Delete(25);
28
29     CTree tr0;
30     tr0.Insert(8);
31     tr0.Insert(9);
32     tr0.Delete(8);
33
34     CTree tr1;
35     tr1.Insert(10);
36     tr1.Insert(9);
37     tr1.Delete(10);
38     #endif // 0
39
40     #if 0
41     tr.Delete(20);
42
43     CTree tr0;
44     tr0.Insert(20);
45     tr0.Delete(20);
46
47     bool bRes = tr.Find(7);
48     bRes = tr.Find(88);
49     bRes = tr.Find(25);
50     #endif // 0
51
52     return 0;
53 }

```

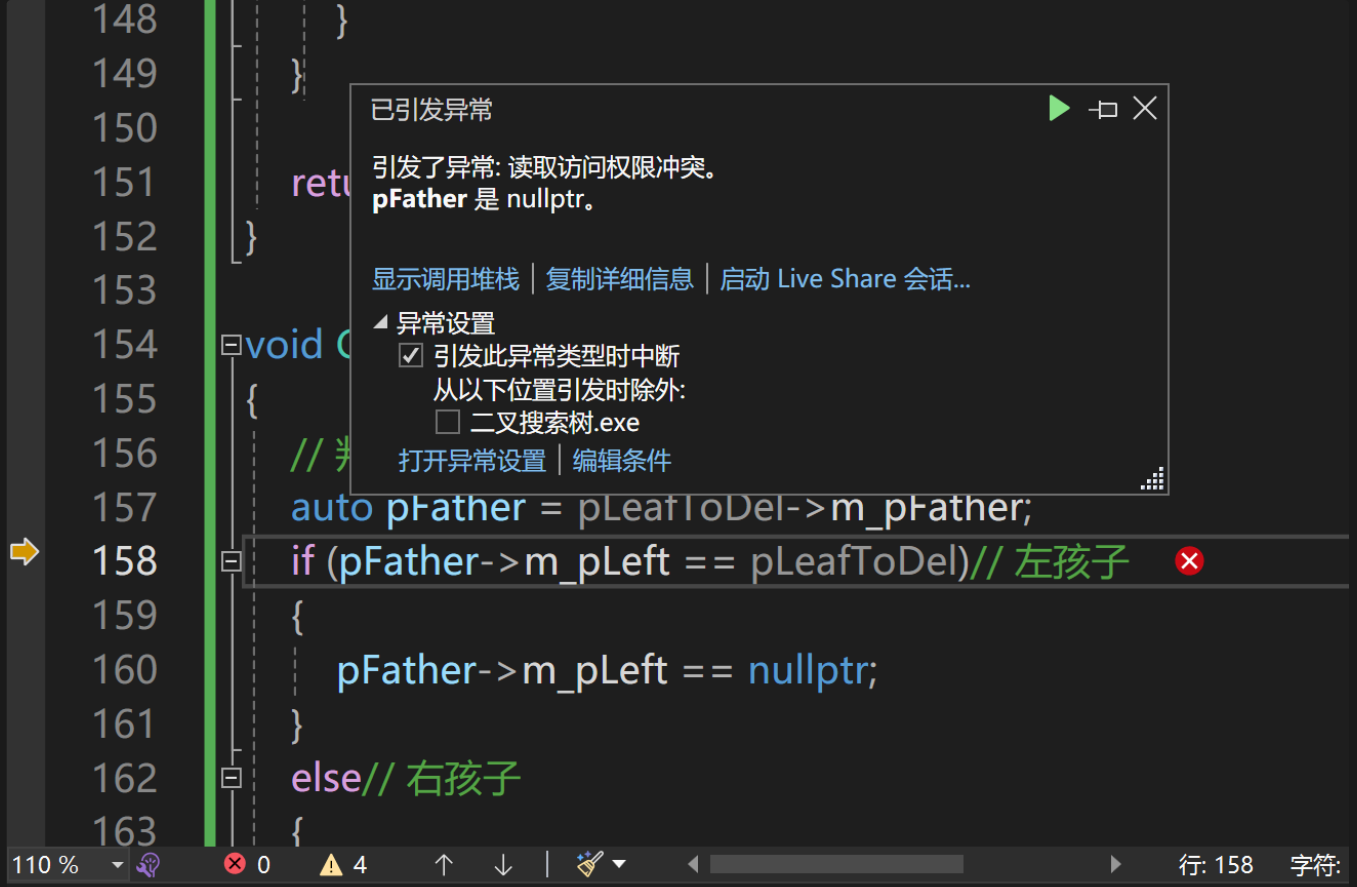


Figure 0-1

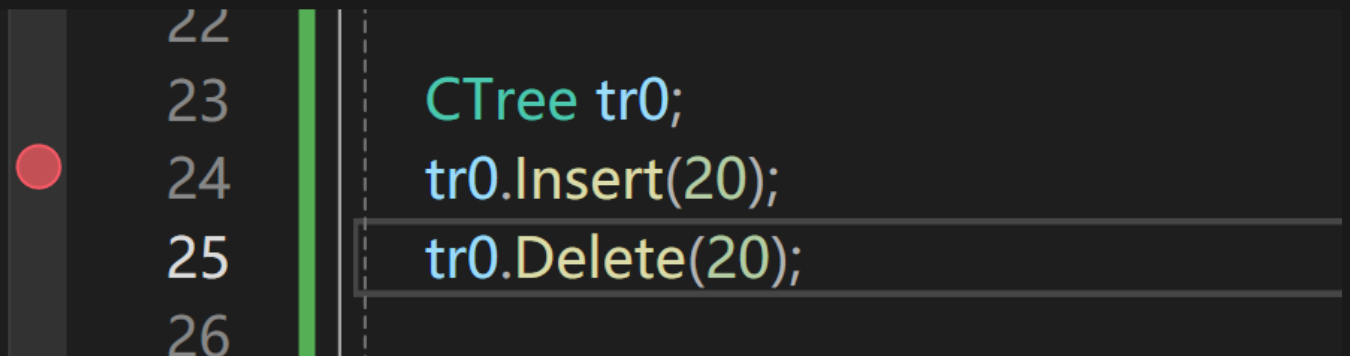


Figure 0-2

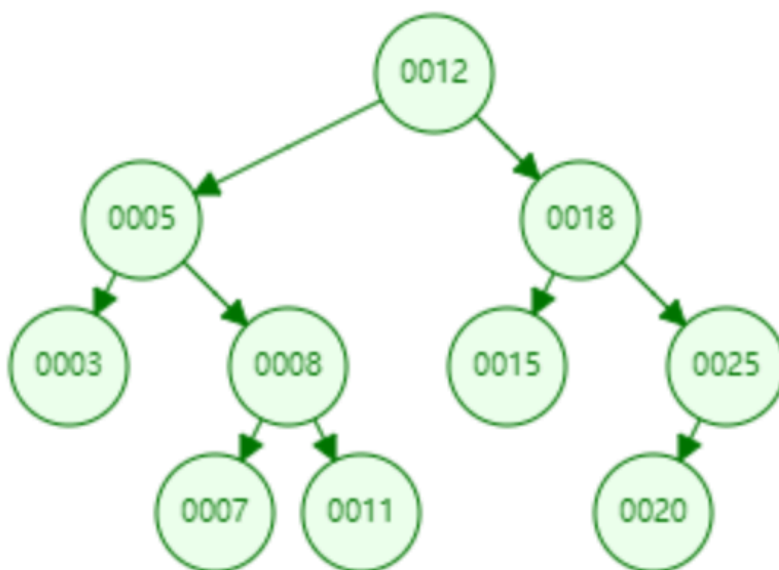


Figure 0-3



## 1. 二叉搜索树的各个操作的时间复杂度是 $\log n$

# 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

```
9 CTree tr;  
10 tr.Insert(12);  
11 tr.Insert(5);  
12 tr.Insert(18);  
13 tr.Insert(3);  
14 tr.Insert(8);  
15 tr.Insert(15);  
16 tr.Insert(25);  
17 tr.Insert(7);  
18 tr.Insert(11);  
19 tr.Insert(20);  
20  
21 tr.Mid();  
22  
23 tr.Delete(8); 已用时间 <= 360m  
24 tr.Delete(12);
```

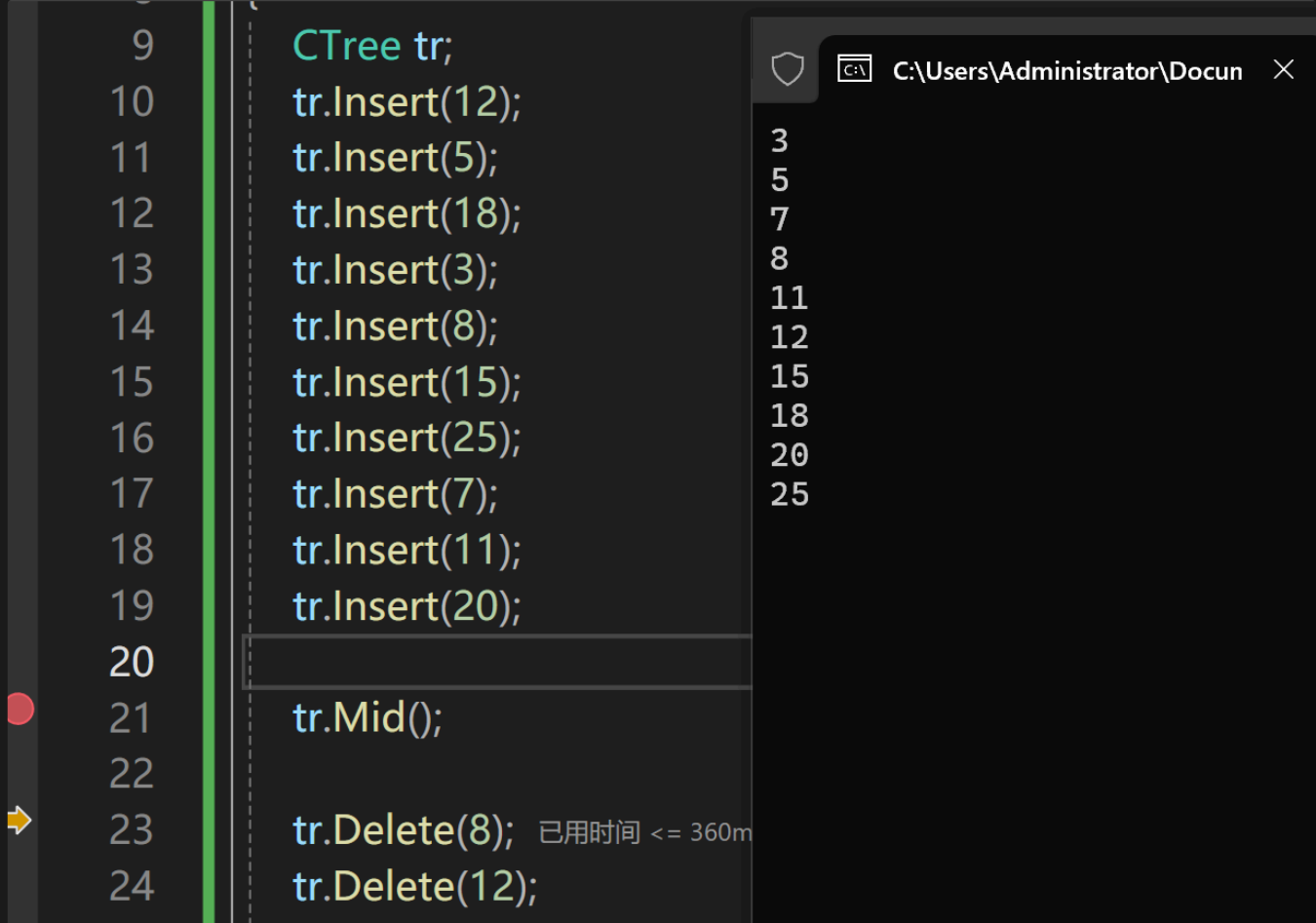


Figure 1-2