# AVLTree

制作者 Doxygen 1.14.0

# Chapter 1

# 命名空间索引

## 1.1 命名空间列表

这里列出了所有命名空间定义，附带简要说明:

# Chapter 2

# 类索引

## 2.1 类列表

这里列出了所有类、结构、联合以及接口定义等，并附带简要说明:

# Chapter 3

# 文件索引

## 3.1 文件列表

这里列出了所有文件，并附带简要说明:

# Chapter 4

# 命名空间文档

## 4.1 xjcad 命名空间参考

命名空间

- namespace lcd

## 4.2 xjcad::lcd 命名空间参考

类

- class AvlTree1

# Chapter 5

# 类说明

## 5.1 xjcad::lcd::AvlTree1< T > 模板类 参考

`#include <myavl.hpp>`

类

- class Iterator
- struct Node

  定义*AVL*树节点类型

**Public** 成员函数

- AvlTree1 ()
- AvlTree1 (const AvlTree1 &other)
- ~AvlTree1 ()
- AvlTree1 & operator= (const AvlTree1 &avl)
- void swap (AvlTree1 &first, AvlTree1 &second) noexcept
- T & operator[ ] (const int index) const
- T & n_inOrder (const int index, int &m) const
- bool operator== (const AvlTree1 &avl) const
- bool operator!= (const AvlTree1 &avl) const
- void insert (const T &val)
- void remove (const T &val)
- void pop_front ()
- void pop_back ()
- void erase (Iterator it)
- bool find (const T &val) const
- int size () const
- int high () const noexcept
- bool empty () const
- void levelNum () const
- bool isBalance () const
- T front () const
- T back () const
- void preOrder () const

- void inOrder () const
- void postOrder () const
- void levelOrder () const
- void clear ()
- void swap (AvlTree1 &other) noexcept
- T & inOrderIndex (const int index) const
- Iterator begin ()

  返回一个指向*AVL*树中序遍历中的第一个元素（也就是最小值）的迭代器
- Iterator end ()

  返回一个指向*AVL*树中序遍历中的最后一个元素（也就是最大值）后面位置的迭代器，所有这里*end()*迭代器下面的指针指向的是空
- Iterator last ()

**Private** 成员函数

- int height (Node *node)
- Node * rightRotate (Node *node)
- Node * leftRotate (Node *node)
- Node * leftBalance (Node *node)
- Node * rightBalance (Node *node)
- Node * insert (Node *node, Node *parent, int &size_, const T &val)
- Node * remove (Node *node, int &size_, const T &val)
- bool find (Node *node, const T &val) const
- int high (Node *node) const
- int levelNum (Node *node, int i) const
- int isBalance (Node *node, int l, bool &flag) const
- void preOrder (Node *node) const
- void inOrder (Node *node) const
- void postOrder (Node *node) const
- void levelOrder (Node *node, int i) const
- Node * copyTree (Node *node)
- void inOrderIndex (Node *node, const int index, int &m, T &result) const

**Private** 属性

- Node * root_

  指向根节点的指针
- int size_

  结点数目

### 5.1.1 详细描述

**template<class T>**
**class xjcad::lcd::AvlTree1< T >**

模板类Avl树实现

在文件 myavl.hpp 第 23 行定义.

## 5.1.2 构造及析构函数说明

### 5.1.2.1 AvlTree1() [1/2]

```
template<class T>
xjcad::lcd::AvlTree1< T >::AvlTree1 ()  [inline]
```

在文件 myavl.hpp 第 27 行定义.

### 5.1.2.2 AvlTree1() [2/2]

```
template<class T>
xjcad::lcd::AvlTree1< T >::AvlTree1 (
            const AvlTree1< T > & other)  [inline]
```

拷贝构造 (深拷贝,复制整颗树)

在文件 myavl.hpp 第 32 行定义.

### 5.1.2.3 ~AvlTree1()

```
template<class T>
xjcad::lcd::AvlTree1< T >::~AvlTree1 ()  [inline]
```

在文件 myavl.hpp 第 36 行定义.

## 5.1.3 成员函数说明

### 5.1.3.1 back()

```
template<class T>
T xjcad::lcd::AvlTree1< T >::back () const  [inline]
```

在文件 myavl.hpp 第 186 行定义.

### 5.1.3.2 begin()

```
template<class T>
Iterator xjcad::lcd::AvlTree1< T >::begin ()  [inline]
```

返回一个指向AVL树中序遍历中的第一个元素（也就是最小值）的迭代器

在文件 myavl.hpp 第 485 行定义.

#### 5.1.3.3 clear()

```
template<class T>
void xjcad::lcd::AvlTree1< T >::clear ()  [inline]
```

1.清空树

在文件 myavl.hpp 第 227 行定义.

#### 5.1.3.4 copyTree()

```
template<class T>
Node * xjcad::lcd::AvlTree1< T >::copyTree (
            Node * node)  [inline], [private]
```

递归复制子树(核心深拷贝函数)

在文件 myavl.hpp 第 868 行定义.

#### 5.1.3.5 empty()

```
template<class T>
bool xjcad::lcd::AvlTree1< T >::empty () const  [inline]
```

4.判空

在文件 myavl.hpp 第 158 行定义.

#### 5.1.3.6 end()

```
template<class T>
Iterator xjcad::lcd::AvlTree1< T >::end ()  [inline]
```

返回一个指向AVL树中序遍历中的最后一个元素（也就是最大值）后面位置的迭代器，所有这里end()迭代器下面的指针指向的是空

在文件 myavl.hpp 第 494 行定义.

#### 5.1.3.7 erase()

```
template<class T>
void xjcad::lcd::AvlTree1< T >::erase (
            Iterator it)  [inline]
```

删除迭代器指向元素（这个函数要么放到Iterator定义的下面，要么提前声明一下Iterator, 不然不知道↩ Iterator是啥） it._p是Iterator的私有成员变量，不能访问。两种解决方法 1.不访问私有成员变量_p，通过上面方式 2.在Iterator类里添加友元声明：friend void AvlTree1<T>::erase(Iterator it);

在文件 myavl.hpp 第 119 行定义.

#### 5.1.3.8 find() [1/2]

```
template<class T>
bool xjcad::lcd::AvlTree1< T >::find (
            const T & val) const  [inline]
```

1.AVL树的查询操作

在文件 myavl.hpp 第 139 行定义.

#### 5.1.3.9 find() [2/2]

```
template<class T>
bool xjcad::lcd::AvlTree1< T >::find (
            Node * node,
            const T & val) const  [inline], [private]
```

AVL树的查询操作递归实现（存在返回true，否则返回false）

在文件 myavl.hpp 第 747 行定义.

#### 5.1.3.10 front()

```
template<class T>
T xjcad::lcd::AvlTree1< T >::front () const  [inline]
```

7.返回第一个和最后一个元素

在文件 myavl.hpp 第 181 行定义.

#### 5.1.3.11 height()

```
template<class T>
int xjcad::lcd::AvlTree1< T >::height (
            Node * node)  [inline], [private]
```

返回节点的高度值

在文件 myavl.hpp 第 531 行定义.

#### 5.1.3.12 high() [1/2]

```
template<class T>
int xjcad::lcd::AvlTree1< T >::high () const  [inline], [noexcept]
```

3.AVL树层数查询

在文件 myavl.hpp 第 153 行定义.

#### 5.1.3.13 high() [2/2]

```
template<class T>
int xjcad::lcd::AvlTree1< T >::high (
            Node * node) const  [inline], [private]
```

AVL树层数查询递归实现

在文件 myavl.hpp 第 773 行定义.

#### 5.1.3.14 inOrder() [1/2]

```
template<class T>
void xjcad::lcd::AvlTree1< T >::inOrder () const  [inline]
```

中序遍历LVR

在文件 myavl.hpp 第 202 行定义.

#### 5.1.3.15 inOrder() [2/2]

```
template<class T>
void xjcad::lcd::AvlTree1< T >::inOrder (
            Node * node) const  [inline], [private]
```

中序遍历LVR

在文件 myavl.hpp 第 832 行定义.

#### 5.1.3.16 inOrderIndex() [1/2]

```
template<class T>
T & xjcad::lcd::AvlTree1< T >::inOrderIndex (
            const int index) const  [inline]
```

3.返回中序遍历结果中索引为i的元素

在文件 myavl.hpp 第 257 行定义.

#### 5.1.3.17 inOrderIndex() [2/2]

```
template<class T>
void xjcad::lcd::AvlTree1< T >::inOrderIndex (
            Node * node,
            const int index,
            int & m,
            T & result) const  [inline], [private]
```

重载[]运算符时使用，返回中序遍历结果中索引为i的元素（递归中序遍历）

在文件 myavl.hpp 第 885 行定义.

#### 5.1.3.18 insert() [1/2]

```
template<class T>
void xjcad::lcd::AvlTree1< T >::insert (
            const T & val)  [inline]
```

AVL树的插入操作

在文件 myavl.hpp 第 88 行定义.

#### 5.1.3.19 insert() [2/2]

```
template<class T>
Node * xjcad::lcd::AvlTree1< T >::insert (
            Node * node,
            Node * parent,
            int & size_,
            const T & val)  [inline], [private]
```

AVL树的插入操作递归实现

在文件 myavl.hpp 第 596 行定义.

#### 5.1.3.20 isBalance() [1/2]

```
template<class T>
bool xjcad::lcd::AvlTree1< T >::isBalance () const  [inline]
```

6.判断是否是AVL树

在文件 myavl.hpp 第 173 行定义.

#### 5.1.3.21 isBalance() [2/2]

```
template<class T>
int xjcad::lcd::AvlTree1< T >::isBalance (
            Node * node,
            int l,
            bool & flag) const  [inline], [private]
```

判断是否是AVL树 递归过程中记录节点的高度值 返回节点的高度值

在文件 myavl.hpp 第 797 行定义.

#### 5.1.3.22 last()

```
template<class T>
Iterator xjcad::lcd::AvlTree1< T >::last ()  [inline]
```

在文件 myavl.hpp 第 498 行定义.

#### 5.1.3.23 leftBalance()

```
template<class T>
Node * xjcad::lcd::AvlTree1< T >::leftBalance (
            Node * node)  [inline], [private]
```

左平衡操作（左孩子的右子树太高了） 左 - 右旋转，并把新的根节点返回

在文件 myavl.hpp 第 574 行定义.

#### 5.1.3.24 leftRotate()

```
template<class T>
Node * xjcad::lcd::AvlTree1< T >::leftRotate (
            Node * node)  [inline], [private]
```

左旋转操作 以节点node为轴做左旋转操作，并把新的根节点返回

在文件 myavl.hpp 第 555 行定义.

#### 5.1.3.25 levelNum() [1/2]

```
template<class T>
void xjcad::lcd::AvlTree1< T >::levelNum () const  [inline]
```

5.打印AVL树每一层的节点数

在文件 myavl.hpp 第 163 行定义.

#### 5.1.3.26 levelNum() [2/2]

```
template<class T>
int xjcad::lcd::AvlTree1< T >::levelNum (
            Node * node,
            int i) const  [inline], [private]
```

AVL树每一层的节点数递归实现（类似层数查询）

在文件 myavl.hpp 第 784 行定义.

#### 5.1.3.27 levelOrder() [1/2]

```
template<class T>
void xjcad::lcd::AvlTree1< T >::levelOrder () const  [inline]
```

层序遍历

在文件 myavl.hpp 第 216 行定义.

#### 5.1.3.28 levelOrder() [2/2]

```
template<class T>
void xjcad::lcd::AvlTree1< T >::levelOrder (
            Node * node,
            int i) const  [inline], [private]
```

层序遍历

在文件 myavl.hpp 第 852 行定义.

#### 5.1.3.29 n_inOrder()

```
template<class T>
T & xjcad::lcd::AvlTree1< T >::n_inOrder (
            const int index,
            int & m) const
```

非递归中序遍历，按照输入索引返回中序遍历序列对应元素

在文件 myavl.hpp 第 940 行定义.

#### 5.1.3.30 operator"!=()

```
template<class T>
bool xjcad::lcd::AvlTree1< T >::operator!= (
            const AvlTree1< T > & avl) const  [inline]
```

operator!= 等号运算符重载, 两颗树不一样返回true，否则false

在文件 myavl.hpp 第 71 行定义.

#### 5.1.3.31 operator=()

```
template<class T>
AvlTree1< T > & xjcad::lcd::AvlTree1< T >::operator= (
            const AvlTree1< T > & avl)
```

赋值操作，operator = 等号运算符重载，也可以防止浅拷贝问题

1.赋值操作，= 运算符重载，也可以防止浅拷贝问题（类外实现）

在文件 myavl.hpp 第 910 行定义.

#### 5.1.3.32 operator==()

```
template<class T>
bool xjcad::lcd::AvlTree1< T >::operator== (
            const AvlTree1< T > & avl) const  [inline]
```

operator== 等号运算符重载, 两颗树一样返回true，否则false

在文件 myavl.hpp 第 57 行定义.

#### 5.1.3.33 operator[]()

```
template<class T>
T & xjcad::lcd::AvlTree1< T >::operator[] (
            const int index) const
```

[] 运算符重载，通过下标的方式访问树的元素，默认按中序遍历访问 1.递归方式访问 T& operator[ ](const int index) { return inOrderIndex(index); } 2.非递归方式，利用栈

2.[]运算符重载，通过下标的方式访问树的元素，默认按中序遍历访问。 非递归方式，利用栈

参数

| | |
|---|---|
| *index* | 索引下标值 |

返回

中序遍历序列中索引对应元素

在文件 myavl.hpp 第 931 行定义.

#### 5.1.3.34 pop_back()

```
template<class T>
void xjcad::lcd::AvlTree1< T >::pop_back ()  [inline]
```

删除最后一个结点

在文件 myavl.hpp 第 106 行定义.

#### 5.1.3.35 pop_front()

```
template<class T>
void xjcad::lcd::AvlTree1< T >::pop_front ()  [inline]
```

删除第一个结点

在文件 myavl.hpp 第 98 行定义.

#### 5.1.3.36 postOrder() [1/2]

```
template<class T>
void xjcad::lcd::AvlTree1< T >::postOrder () const  [inline]
```

后序遍历LRV

在文件 myavl.hpp 第 209 行定义.

#### 5.1.3.37 postOrder() [2/2]

```
template<class T>
void xjcad::lcd::AvlTree1< T >::postOrder (
            Node * node) const  [inline], [private]
```

后序遍历LRV

在文件 myavl.hpp 第 842 行定义.

#### 5.1.3.38 preOrder() [1/2]

```
template<class T>
void xjcad::lcd::AvlTree1< T >::preOrder () const  [inline]
```

前序遍历VLR

在文件 myavl.hpp 第 195 行定义.

#### 5.1.3.39 preOrder() [2/2]

```
template<class T>
void xjcad::lcd::AvlTree1< T >::preOrder (
            Node * node) const  [inline], [private]
```

前序遍历VLR

在文件 myavl.hpp 第 822 行定义.

#### 5.1.3.40 remove() [1/2]

```
template<class T>
void xjcad::lcd::AvlTree1< T >::remove (
            const T & val)  [inline]
```

AVL树的删除操作

在文件 myavl.hpp 第 93 行定义.

#### 5.1.3.41 remove() [2/2]

```
template<class T>
Node * xjcad::lcd::AvlTree1< T >::remove (
            Node * node,
            int & size_,
            const T & val)  [inline], [private]
```

AVL树的删除操作递归实现 在根节点为node的树上，找到val节点删除它，并返回val节点的孩子节点的地址，以连到被删除节点的父节点上去

在文件 myavl.hpp 第 649 行定义.

#### 5.1.3.42 rightBalance()

```
template<class T>
Node * xjcad::lcd::AvlTree1< T >::rightBalance (
            Node * node)  [inline], [private]
```

右平衡操作（右孩子的左子树太高了） 右 - 左旋转操作，并把新的根节点返回

在文件 myavl.hpp 第 585 行定义.

#### 5.1.3.43 rightRotate()

```
template<class T>
Node * xjcad::lcd::AvlTree1< T >::rightRotate (
            Node * node)  [inline], [private]
```

右旋转操作 以节点node为轴做右旋转操作，并把新的根节点返回

在文件 myavl.hpp 第 536 行定义.

#### 5.1.3.44 size()

```
template<class T>
int xjcad::lcd::AvlTree1< T >::size () const  [inline]
```

2.AVL树节点总数查询

在文件 myavl.hpp 第 148 行定义.

#### 5.1.3.45 swap() [1/2]

```
template<class T>
void xjcad::lcd::AvlTree1< T >::swap (
            AvlTree1< T > & first,
            AvlTree1< T > & second)  [noexcept]
```

在文件 myavl.hpp 第 920 行定义.

#### 5.1.3.46 swap() [2/2]

```
template<class T>
void xjcad::lcd::AvlTree1< T >::swap (
            AvlTree1< T > & other)  [inline], [noexcept]
```

2.两颗avl树交换函数 eg: tree1.swap(tree2)

在文件 myavl.hpp 第 251 行定义.

### 5.1.4 类成员变量说明

#### 5.1.4.1 root_

```
template<class T>
Node* xjcad::lcd::AvlTree1< T >::root_  [private]
```

指向根节点的指针

在文件 myavl.hpp 第 527 行定义.

#### 5.1.4.2 size_

```
template<class T>
int xjcad::lcd::AvlTree1< T >::size_ [private]
```

结点数目

在文件 myavl.hpp 第 528 行定义.

该类的文档由以下文件生成:

- myavl.hpp

## 5.2 xjcad::lcd::AvlTree1< T >::Iterator类 参考

```
#include <myavl.hpp>
```

**Public** 成员函数

- Iterator (Node *p=nullptr, AvlTree1< T > *pavl=nullptr)
  迭代器构造函数
- bool operator!= (const Iterator &it) const
- bool operator== (const Iterator &it) const
- bool operator< (const Iterator &it) const
- bool operator<= (const Iterator &it) const
- bool operator> (const Iterator &it) const
- bool operator>= (const Iterator &it) const
- Iterator & operator++ ()
- Iterator & operator++ (int)
- Iterator & operator-- ()
- Iterator & operator-- (int)
- Iterator & operator+= (const int &n)
- Iterator operator+ (const int &n) const
- Iterator & operator-= (const int &n)
- Iterator operator- (const int &n) const
- T & operator* ()
  *n - it*没必要了
- Node * operator-> ()

静态 **Private** 成员函数

- static Node * nextInOrder (Node *node)
- static Node * findMin (Node *node)
- static Node * preInOrder (Node *node)
- static Node * findMax (Node *node)

**Private** 属性

- Node * _p
  维护一个指针，指向*AVL*树的当前节点
- AvlTree1< T > * _pavl
  让迭代器知道它是属于哪颗*AVL*树的.

友元

- Iterator operator+ (int n, const Iterator &it)
- void AvlTree1 (Iterator it)
  友元声明

### 5.2.1 详细描述

**template<class T>**
**class xjcad::lcd::AvlTree1< T >::Iterator**

Avl容器的迭代器实现

在文件 myavl.hpp 第 270 行定义.

### 5.2.2 构造及析构函数说明

#### 5.2.2.1 Iterator()

```
template<class T>
xjcad::lcd::AvlTree1< T >::Iterator::Iterator (
            Node * p = nullptr,
            AvlTree1< T > * pavl = nullptr)  [inline]
```

迭代器构造函数

在文件 myavl.hpp 第 273 行定义.

### 5.2.3 成员函数说明

#### 5.2.3.1 findMax()

```
template<class T>
Node * xjcad::lcd::AvlTree1< T >::Iterator::findMax (
            Node * node)  [inline], [static], [private]
```

在文件 myavl.hpp 第 471 行定义.

#### 5.2.3.2 findMin()

```
template<class T>
Node * xjcad::lcd::AvlTree1< T >::Iterator::findMin (
            Node * node)  [inline], [static], [private]
```

在文件 myavl.hpp 第 447 行定义.

#### 5.2.3.3 nextInOrder()

```
template<class T>
Node * xjcad::lcd::AvlTree1< T >::Iterator::nextInOrder (
            Node * node)  [inline], [static], [private]
```

查找中序后继

在文件 myavl.hpp 第 430 行定义.

#### 5.2.3.4 operator"!=()

```
template<class T>
bool xjcad::lcd::AvlTree1< T >::Iterator::operator!= (
            const Iterator & it) const  [inline]
```

不同容器的迭代器不能进行比较运算，没有意义

在文件 myavl.hpp 第 275 行定义.

#### 5.2.3.5 operator*()

```
template<class T>
T & xjcad::lcd::AvlTree1< T >::Iterator::operator* ()  [inline]
```

n - it没必要了

在文件 myavl.hpp 第 417 行定义.

#### 5.2.3.6 operator+()

```
template<class T>
Iterator xjcad::lcd::AvlTree1< T >::Iterator::operator+ (
            const int & n) const  [inline]
```

类成员运算符重载：it + n. 返回新迭代器，支持 (it+1)+3这种链式操作

在文件 myavl.hpp 第 378 行定义.

#### 5.2.3.7 operator++() [1/2]

```
template<class T>
Iterator & xjcad::lcd::AvlTree1< T >::Iterator::operator++ ()  [inline]
```

前置递增. 返回其中序遍历序列中该元素后继元素的地址 如果迭代器指向空，什么都不做，就不引发异常了

在文件 myavl.hpp 第 330 行定义.

#### 5.2.3.8 operator++() [2/2]

```
template<class T>
Iterator & xjcad::lcd::AvlTree1< T >::Iterator::operator++ (
            int )  [inline]
```

后置递增

在文件 myavl.hpp 第 338 行定义.

#### 5.2.3.9 operator+=()

```
template<class T>
Iterator & xjcad::lcd::AvlTree1< T >::Iterator::operator+= (
            const int & n)  [inline]
```

类成员运算符重载：it += n. 改变原迭代器本身，不返回新迭代器，故不支持(it+1)+3这种链式操作。默认n过大超界时指向最后一个元素

在文件 myavl.hpp 第 362 行定义.

#### 5.2.3.10 operator-()

```
template<class T>
Iterator xjcad::lcd::AvlTree1< T >::Iterator::operator- (
            const int & n) const  [inline]
```

类成员运算符重载：it - n（返回新迭代器，支持 (it-1)-3这种链式操作）

在文件 myavl.hpp 第 410 行定义.

#### 5.2.3.11 operator--() [1/2]

```
template<class T>
Iterator & xjcad::lcd::AvlTree1< T >::Iterator::operator-- ()  [inline]
```

前置递减

在文件 myavl.hpp 第 345 行定义.

#### 5.2.3.12 operator--() [2/2]

```
template<class T>
Iterator & xjcad::lcd::AvlTree1< T >::Iterator::operator-- (
            int )  [inline]
```

后置递减

在文件 myavl.hpp 第 353 行定义.

#### 5.2.3.13 operator-=()

```
template<class T>
Iterator & xjcad::lcd::AvlTree1< T >::Iterator::operator-= (
            const int & n)  [inline]
```

it -= n 默认n过大超界时指向第一个元素（最小值）

在文件 myavl.hpp 第 395 行定义.

#### 5.2.3.14 operator->()

```
template<class T>
Node * xjcad::lcd::AvlTree1< T >::Iterator::operator-> ()  [inline]
```

在文件 myavl.hpp 第 424 行定义.

#### 5.2.3.15 operator<()

```
template<class T>
bool xjcad::lcd::AvlTree1< T >::Iterator::operator< (
            const Iterator & it) const  [inline]
```

在文件 myavl.hpp 第 289 行定义.

#### 5.2.3.16 operator<=()

```
template<class T>
bool xjcad::lcd::AvlTree1< T >::Iterator::operator<= (
            const Iterator & it) const  [inline]
```

在文件 myavl.hpp 第 299 行定义.

#### 5.2.3.17 operator==()

```
template<class T>
bool xjcad::lcd::AvlTree1< T >::Iterator::operator== (
            const Iterator & it) const  [inline]
```

在文件 myavl.hpp 第 282 行定义.

#### 5.2.3.18 operator>()

```
template<class T>
bool xjcad::lcd::AvlTree1< T >::Iterator::operator> (
            const Iterator & it) const  [inline]
```

在文件 myavl.hpp 第 308 行定义.

#### 5.2.3.19 operator>=()

```
template<class T>
bool xjcad::lcd::AvlTree1< T >::Iterator::operator>= (
            const Iterator & it) const  [inline]
```

在文件 myavl.hpp 第 317 行定义.

#### 5.2.3.20 preInOrder()

```
template<class T>
Node * xjcad::lcd::AvlTree1< T >::Iterator::preInOrder (
            Node * node)  [inline], [static], [private]
```

查找中序前驱

在文件 myavl.hpp 第 454 行定义.

### 5.2.4 友元及相关符号说明

#### 5.2.4.1 AvlTree1

```
template<class T>
void AvlTree1 (
            Iterator it)  [friend]
```

友元声明

#### 5.2.4.2 operator+

```
template<class T>
Iterator operator+ (
            int n,
            const Iterator & it)  [friend]
```

在类内部定义友元 n + it.

参数

| | |
|---|---|
| *n* | 非负整数 |
| *it* | 迭代器 |

返回

一个新的迭代器

在文件 myavl.hpp 第 390 行定义.

### 5.2.5 类成员变量说明

#### 5.2.5.1 _p

```
template<class T>
Node* xjcad::lcd::AvlTree1< T >::Iterator::_p  [private]
```

维护一个指针，指向AVL树的当前节点

在文件 myavl.hpp 第 478 行定义.

#### 5.2.5.2 _pavl

```
template<class T>
AvlTree1<T>* xjcad::lcd::AvlTree1< T >::Iterator::_pavl  [private]
```

让迭代器知道它是属于哪颗AVL树的.

在文件 myavl.hpp 第 479 行定义.

该类的文档由以下文件生成:

- myavl.hpp

## 5.3 xjcad::lcd::AvlTree1< T >::Node结构体 参考

定义AVL树节点类型

**Public** 成员函数

- Node (T data=T())

**Public** 属性

- T data_
- Node * left_
- Node * right_
- Node * parent_
- int height_
  记录节点高度值

### 5.3.1 详细描述

**template<class T>**
**struct xjcad::lcd::AvlTree1< T >::Node**

定义AVL树节点类型

在文件 myavl.hpp 第 518 行定义.

## 5.3.2 构造及析构函数说明

### 5.3.2.1 Node()

```
template<class T>
xjcad::lcd::AvlTree1< T >::Node::Node (
            T data = T())  [inline]
```

在文件 myavl.hpp 第 520 行定义.

## 5.3.3 类成员变量说明

### 5.3.3.1 data_

```
template<class T>
T xjcad::lcd::AvlTree1< T >::Node::data_
```

在文件 myavl.hpp 第 521 行定义.

### 5.3.3.2 height_

```
template<class T>
int xjcad::lcd::AvlTree1< T >::Node::height_
```

记录节点高度值

在文件 myavl.hpp 第 525 行定义.

### 5.3.3.3 left_

```
template<class T>
Node* xjcad::lcd::AvlTree1< T >::Node::left_
```

在文件 myavl.hpp 第 522 行定义.

### 5.3.3.4 parent_

```
template<class T>
Node* xjcad::lcd::AvlTree1< T >::Node::parent_
```

在文件 myavl.hpp 第 524 行定义.

### 5.3.3.5 right_

```
template<class T>
Node* xjcad::lcd::AvlTree1< T >::Node::right_
```

在文件 myavl.hpp 第 523 行定义.

该结构体的文档由以下文件生成:

- myavl.hpp

# Chapter 6

# 文件说明

## 6.1 myavl.hpp 文件参考

实现avl树并兼容标准容器操作

```
#include <iostream>
#include <queue>
#include <stack>
#include <algorithm>
```

类

- class xjcad::lcd::AvlTree1< T >
- class xjcad::lcd::AvlTree1< T >::Iterator
- struct xjcad::lcd::AvlTree1< T >::Node

  定义*AVL*树节点类型

命名空间

- namespace xjcad
- namespace xjcad::lcd

### 6.1.1 详细描述

实现avl树并兼容标准容器操作

版权所有

Copyright © 2025, Wuxi Xinje Electric Co., Ltd.

版本

作者

lcd

日期

August 2025

在文件 myavl.hpp 中定义.

## 6.2 myavl.hpp

浏览该文件的文档.

```
00001 /*****************************************************************/
00009
00010 #pragma once
00011 #include<iostream>
00012 #include<queue>
00013 #include<stack>
00014 #include <algorithm>
00015
00016 namespace xjcad
00017 {
00018 namespace lcd
00019 {
00020 // AVL树 = BST树 + 节点平衡操作
00022 template<class T>
00023 class AvlTree1
00024 {
00025 public:
00026
00027     AvlTree1() {
00028         root_ = nullptr;
00029         size_ = 0;
00030     }
00032     AvlTree1(const AvlTree1& other) :
00033         root_(copyTree(other.root_)), size_(other.size_) {
00034     }  // 调用递归复制函数
00035
00036     ~AvlTree1() {
00037         clear();
00038     }
00039
00041     AvlTree1& operator=(const AvlTree1& avl);
00042     // 交换函数
00043     void swap(AvlTree1& first, AvlTree1& second) noexcept;
00044
00045
00053     T& operator[](const int index) const;
00054     T& n_inOrder(const int index, int& m) const;
00055
00057     bool operator== (const AvlTree1& avl) const
00058     {
00059         if (size_ != avl.size_) {
00060             return false;
00061         }
00062         for (int i = 0; i < size_; i++)
00063         {
00064             if ((*this)[i] != avl[i]) {// 使用 (*this)[i] 访问当前树的第 i 个元素
00065                 return false;
00066             }
00067         }
00068         return true;
00069     }
00071     bool operator!= (const AvlTree1& avl) const
00072     {
00073         if (size_ != avl.size_) {
00074             return true;
00075         }
00076         for (int i = 0; i < size_; i++)
00077         {
00078             if ((*this)[i] != avl[i]) { // 使用 (*this)[i] 访问当前树的第 i 个元素
00079                 return true;
00080             }
00081         }
00082         return false;
00083     }
00084
00085 public:
00086     //---------结点插入/删除-------------
00088     void insert(const T& val)
00089     {
00090         root_ = insert(root_, nullptr, size_, val);
00091     }
00093     void remove(const T& val)
00094     {
00095         root_ = remove(root_, size_, val); // 添加父节点参数
00096     }
00098     void pop_front()
00099     {
00100         if (root_) {
00101             T val = inOrderIndex(0);
00102             remove(val);
00103         }
00104     }
```

```
00106     void pop_back()
00107     {
00108         if (!empty()) {
00109             T val = inOrderIndex(size_ - 1);
00110             remove(val);
00111         }
00112     }
00113
00114     class Iterator;
00119     void erase(Iterator it) {
00120         if (it._p) {  //如果传进来的迭代器不是指向空的，即有效的迭代器
00121             //std::cout << "进入" << std::endl;
00122             remove(*it);
00123         }
00124     }
00125     //void erase(Iterator start, Iterator end)
00126     //{
00127     //  //如果start不指向空且start不等于end
00128     //  if (start._p && start != end) {
00129     //      for (Iterator it = start; it != end; ++it)
00130     //      {
00131     //          remove(*it);
00132     //      }
00133     //  }
00134     //}
00135
00136
00137     //--------------查询-----------------
00139     bool find(const T& val) const
00140     {
00141         return find(root_, val);
00142     }
00144     /*int size() const noexcept
00145     {
00146         return size(root_);
00147     }*/
00148     int size() const
00149     {
00150         return size_;
00151     }
00153     int high() const noexcept
00154     {
00155         return high(root_);
00156     }
00158     bool empty() const
00159     {
00160         return root_ == nullptr;
00161     }
00163     void levelNum() const
00164     {
00165         int h = high();
00166         for (int i = 0; i < h; i++)
00167         {
00168             std::cout << "第" << i + 1 << "层节点数目为：" << levelNum(root_, i) << "  ";
00169         }
00170         std::cout << std::endl;
00171     }
00173     bool isBalance() const
00174     {
00175         int l = 0; //递归函数的局部变量形参，用于记录每个节点的层数
00176         bool flag = true; //递归回溯时判断是否失衡的标志位
00177         isBalance(root_, l, flag);
00178         return flag;
00179     }
00181     T front() const {
00182         if (!empty()) {
00183             return inOrderIndex(0);
00184         }
00185     }
00186     T back() const {
00187         if (!empty()) {
00188             return inOrderIndex(size() - 1);
00189         }
00190     }
00191
00192
00193     //--------------遍历-----------------
00195     void preOrder() const
00196     {
00197         std::cout << "[前序]遍历：";
00198         preOrder(root_);
00199         std::cout << std::endl;
00200     }
00202     void inOrder() const
00203     {
00204         std::cout << "[中序]遍历：";
00205         inOrder(root_);
```

```
00206         std::cout << std::endl;
00207     }
00209     void postOrder() const
00210     {
00211         std::cout << "[后序]遍历: ";
00212         postOrder(root_);
00213         std::cout << std::endl;
00214     }
00216     void levelOrder() const
00217     {
00218         int high1 = high(); //树高
00219         std::cout << "[层序]遍历: ";
00220         for (int i = 0; i < high1; i++)
00221             levelOrder(root_, i); //每次只打印某一层
00222         std::cout << std::endl;
00223     }
00224
00225     //---------------其余------------------
00227     void clear()
00228     {
00229         if (root_ != nullptr) //树不为空再删（利用层序遍历思想删除BST/AVL树所有节点）
00230         {
00231             std::queue<Node*> q;
00232             q.push(root_);
00233             while (!q.empty())
00234             {
00235                 Node* cur = q.front(); //取出队列头元素
00236                 q.pop();
00237
00238                 if (cur->left_ != nullptr) {
00239                     q.push(cur->left_);
00240                 }
00241                 if (cur->right_ != nullptr) {
00242                     q.push(cur->right_);
00243                 }
00244                 delete cur;
00245             }
00246         }
00247         root_ = nullptr; // 置空根指针
00248         size_ = 0;
00249     }
00251     void swap(AvlTree1& other) noexcept {
00252         using std::swap;
00253         swap(root_, other.root_);
00254         swap(size_, other.size_);
00255     }
00257     T& inOrderIndex(const int index) const
00258     {
00259         if (index < 0 || index >= size()) {
00260             throw "out of range!";
00261         }
00262         int m = 0;
00263         T result;
00264         inOrderIndex(root_, index, m, result);
00265         return result;
00266     }
00267
00268     struct Node;
00270     class Iterator
00271     {
00272     public:
00273         Iterator(Node* p = nullptr, AvlTree1<T>* pavl = nullptr) :_p(p), _pavl(pavl) {}
00275         bool operator!=(const Iterator& it) const
00276         {
00277             //检查迭代器有效性：检查两迭代器对应的是不是同一个容器
00278             if (_pavl != it._pavl)
00279                 throw "Iterator incompatable!";
00280             return _p != it._p; //判断它们指向的节点是否一样
00281         }
00282         bool operator==(const Iterator& it) const
00283         {
00284             //检查迭代器有效性：检查两迭代器对应的是不是同一个容器
00285             if (_pavl != it._pavl)
00286                 throw "Iterator incompatable!";
00287             return _p == it._p;
00288         }
00289         bool operator<(const Iterator& it) const
00290         {
00291             //检查迭代器有效性：检查两迭代器对应的是不是同一个容器
00292             if (_pavl != it._pavl)
00293                 throw "Iterator incompatable!";
00294             if (!_p || !it._p)  //如果任一迭代器的底层指针为空，返回false
00295                 return false;
00296             else
00297                 return _p->data_ < it._p->data_;
00298         }
00299         bool operator<=(const Iterator& it) const
```

```
00300         {
00301             if (_pavl != it._pavl)
00302                 throw "Iterator incompatable!";
00303             if (!_p || !it._p)  //如果任一迭代器的底层指针为空，返回false
00304                 return false;
00305             else
00306                 return _p->data_ <= it._p->data_;
00307         }
00308         bool operator>(const Iterator& it) const
00309         {
00310             if (_pavl != it._pavl)
00311                 throw "Iterator incompatable!";
00312             if (!_p || !it._p)
00313                 return false;
00314             else
00315                 return _p->data_ > it._p->data_;
00316         }
00317         bool operator>=(const Iterator& it) const
00318         {
00319             if (_pavl != it._pavl)
00320                 throw "Iterator incompatable!";
00321             if (!_p || !it._p)
00322                 return false;
00323             else
00324                 return _p->data_ >= it._p->data_;
00325         }
00326 
00330         Iterator& operator++()
00331         {
00332             if (_p) {
00333                 _p = nextInOrder(_p);
00334             }
00335             return *this;
00336         }
00338         Iterator& operator++(int)
00339         {
00340             Iterator temp = *this;
00341             ++(*this);
00342             return temp;
00343         }
00345         Iterator& operator--()
00346         {
00347             if (_p) {
00348                 _p = preInOrder(_p);
00349             }
00350             return *this;
00351         }
00353         Iterator& operator--(int)
00354         {
00355             Iterator temp = *this;
00356             --(*this);
00357             return temp;
00358         }
00359 
00362         Iterator& operator+=(const int& n) {
00363             if (n <= 0 || !_p) { return *this; }//n非法或迭代器指向空
00364 
00365             Node* current = this->_p;
00366             for (int i = 0; i < n; i++) {
00367                 Node* next = nextInOrder(current);
00368                 if (next == nullptr) {
00369                     break;
00370                 }
00371                 current = next;
00372             }
00373             this->_p = current; // 避免提前修改 _p：使用 cur 临时变量进行移动，确保在移动过程中不破坏迭代器的当前状
    态
00374             return *this;
00375         }
00378         Iterator operator+(const int& n) const {
00379             Iterator temp = *this;
00380             temp += n;
00381             return temp;  //不能返回局部变量temp的引用，即Iterator&
00382         }
00390         friend Iterator operator+(int n, const Iterator& it) {
00391             return it + n;
00392         }
00393 
00395         Iterator& operator-=(const int& n) {
00396             if (n <= 0 || !_p) { return *this; }
00397 
00398             Node* current = this->_p;
00399             for (int i = 0; i < n; i++) {
00400                 Node* last = preInOrder(current);
00401                 if (last == nullptr) {
00402                     break;
00403                 }
```

```
00404                 current = last;
00405             }
00406             this->_p = current; // 避免提前修改 _p：使用 cur 临时变量进行移动，确保在移动过程中不破坏迭代器的当前状
      态
00407             return *this;
00408         }
00410         Iterator operator-(const int& n) const {
00411             Iterator temp = *this;
00412             temp -= n;
00413             return temp;  //不能返回局部变量temp的引用，即Iterator&
00414         }
00416 
00417         T& operator*() {
00418             //检查迭代器有效性：检查两迭代器对应的不是同一个容器
00419             if (!_p) {
00420                 throw "Iterator is nullptr,do not have the value!";
00421             }
00422             return _p->data_;  //解引用返回指针对应节点的数据域
00423         }
00424         Node* operator->() {    //it->_其实就是_p->_
00425             return _p;
00426         }
00427 
00428     private:
00430         static Node* nextInOrder(Node* node) {
00431             if (!node) { return nullptr; }
00432 
00433             // 情况1：存在右子树
00434             if (node->right_) {
00435                 return findMin(node->right_);
00436             }
00437 
00438             // 情况2：向上查找第一个左祖先
00439             Node* cur = node;
00440             Node* parent = node->parent_;
00441             while (parent != nullptr && cur == parent->right_) {
00442                 cur = parent;
00443                 parent = parent->parent_;
00444             }
00445             return parent;          //如果是右下角最大值的节点，它递增后会变为nullptr;
00446         }
00447         static Node* findMin(Node* node) {
00448             while (node && node->left_) {
00449                 node = node->left_;
00450             }
00451             return node;
00452         }
00454         static Node* preInOrder(Node* node) {
00455             if (!node) { return nullptr; }
00456 
00457             // 情况1：存在左子树
00458             if (node->left_) {
00459                 return findMax(node->left_);
00460             }
00461             // 情况2：向上查找第一个右祖先
00462             Node* cur = node;
00463             Node* parent = node->parent_;
00464             while (parent != nullptr && cur == parent->left_)
00465             {
00466                 cur = parent;
00467                 parent = parent->parent_;
00468             }
00469             return parent;            //如果是左下角最小值的节点，它递减后会变为nullptr;
00470         }
00471         static Node* findMax(Node* node) {
00472             while (node && node->right_) {
00473                 node = node->right_;
00474             }
00475             return node;
00476         }
00477 
00478         Node* _p;
00479         AvlTree1<T>* _pavl;
00480 
00481         friend void AvlTree1<T>::erase(Iterator it);
00482         //friend void AvlTree1<T>::erase(Iterator start, Iterator end);
00483     };
00484 
00485     Iterator begin()
00486     {
00487         Node* cur = root_;
00488         while (cur != nullptr && cur->left_ != nullptr)
00489         {
00490             cur = cur->left_;
00491         }
00492         return Iterator(cur, this); //this指向当前容器对象
00493     }
```

```
00494     Iterator end()
00495     {
00496         return Iterator(nullptr, this);
00497     }
00498     Iterator last() //  获取最后一个元素（中序最大）
00499     {
00500         Node* cur = root_;
00501         while (cur != nullptr && cur->right_ != nullptr)
00502         {
00503             cur = cur->right_;
00504         }
00505         return Iterator(cur, this);
00506     }
00507
00509     //void erase(Iterator it)
00510     //{
00511     //  T x = *it;
00512     //  remove(x);
00513     //}
00514
00515
00516 private:
00518     struct Node
00519     {
00520         Node(T data = T()) :data_(data), left_(nullptr), right_(nullptr), parent_(nullptr), height_(1)
    {}
00521         T data_;
00522         Node* left_;
00523         Node* right_;
00524         Node* parent_;
00525         int height_;
00526     };
00527     Node* root_;
00528     int size_;
00529
00531     int height(Node* node)
00532     {
00533         return node == nullptr ? 0 : node->height_;
00534     }
00536     Node* rightRotate(Node* node)
00537     {
00538         //旋转操作
00539         Node* child = node->left_;
00540         Node* grandchild = child->right_;
00541         node->left_ = grandchild;
00542         child->right_ = node;
00543         // 更新父指针
00544         child->parent_ = node->parent_;  // 新根继承原节点的父指针
00545         node->parent_ = child;           // node成为child的子节点
00546         if (grandchild) {
00547             grandchild->parent_ = node;  // 更新grandchild的父指针
00548         }
00549         //高度更新   node child
00550         node->height_ = std::max(height(node->left_), height(node->right_)) + 1; //当前节点高度值 = 左右孩
    子高度值较大者+1
00551         child->height_ = std::max(height(child->left_), height(child->right_)) + 1;
00552         return child; //返回旋转后的子树新的根节点地址，以回溯连接到老根节点的父节点上去
00553     }
00555     Node* leftRotate(Node* node)
00556     {
00557         //旋转操作
00558         Node* child = node->right_;
00559         Node* grandchild = child->left_;
00560         node->right_ = grandchild;
00561         child->left_ = node;
00562         // 更新父指针
00563         child->parent_ = node->parent_;
00564         node->parent_ = child;
00565         if (grandchild) {
00566             grandchild->parent_ = node;
00567         }
00568         //高度更新   node child
00569         node->height_ = std::max(height(node->left_), height(node->right_)) + 1;
00570         child->height_ = std::max(height(child->left_), height(child->right_)) + 1;
00571         return child;
00572     }
00574     Node* leftBalance(Node* node)
00575     {
00576         //先左旋
00577         node->left_ = leftRotate(node->left_);
00578         //if (node->left_) {
00579         //  node->left_->parent_ = node; // 更新新左孩子的父指针
00580         //}
00581         //再右旋
00582         return rightRotate(node); //最终新的根节点就是右旋函数返回的根节点
00583     }
00585     Node* rightBalance(Node* node)
```

```
00586     {
00587         //node->right_ = rightRotate(node->right_);
00588         //if (node->right_) {
00589         //  node->right_->parent_ = node; // 更新新右孩子的父指针
00590         //}
00591         return leftRotate(node);
00592     }
00593 
00594     //-------------结点插入/删除-----------------
00596     Node* insert(Node* node, Node* parent, int& size_, const T& val)
00597     {
00598         if (node == nullptr) //递归结束，找到插入位置
00599         {
00600             Node* newNode = new Node(val);
00601             newNode->parent_ = parent;
00602             size_++;
00603             return newNode;
00604         }
00605         if (node->data_ == val) //找到相同节点不用再往下递归了，直接向上回溯
00606         {
00607             return node;
00608         }
00609         else if (node->data_ > val)
00610         {
00611             node->left_ = insert(node->left_, node, size_, val); //回溯时更新一下node的左孩子!!!
00612             //添加1 在递归回溯时判断节点是否失衡(上面是往当前节点node的左子树插的) node的左子树太高，node失衡了
00613             if (height(node->left_) - height(node->right_) > 1)
00614             {
00615                 Node* child = node->left_;
00616                 if (height(child->left_) >= height(child->right_)) //1. LL情形 左孩子的左子树太高
00617                 {
00618                     return rightRotate(node); //父指针的更新在旋转操作里做了
00619                 }
00620                 else   //2. LR情形 左孩子的右子树太高
00621                 {
00622                     return leftBalance(node);
00623                 }
00624             }
00625         }
00626         else
00627         {
00628             node->right_ = insert(node->right_, node, size_, val);
00629 
00630             //添加2 在递归回溯时判断节点是否失衡
00631             if (height(node->right_) - height(node->left_) > 1)
00632             {
00633                 Node* child = node->right_;
00634                 if (height(child->right_) >= height(child->left_)) //1. RR情形 右孩子的右子树太高
00635                 {
00636                     return leftRotate(node);
00637                 }
00638                 else   //2. RL情形 左孩子的右子树太高
00639                 {
00640                     return rightBalance(node);
00641                 }
00642             }
00643         }
00644         //添加3 在递归回溯时检测更新节点高度
00645         node->height_ = std::max(height(node->left_), height(node->right_)) + 1;
00646         return node;
00647     }
00649     Node* remove(Node* node, int& size_, const T& val)
00650     {
00651         if (node == nullptr) { //没找到
00652             return nullptr;
00653         }
00654         if (node->data_ == val)
00655         {
00656             //情形3
00657             if (node->left_ != nullptr && node->right_ != nullptr)
00658             {
00659                 //*****为了尽量避免删除前驱或后继造成节点失衡，谁高删除谁
00660                 if (height(node->left_) >= height(node->right_)) //删前驱
00661                 {
00662                     Node* cur = node->left_;
00663                     while (cur->right_ != nullptr)
00664                         cur = cur->right_;
00665                     node->data_ = cur->data_; //2.覆盖数据并递归删除
00666                     node->left_ = remove(node->left_, size_, cur->data_); //3.现在要删的是前驱节点cur了,同时
    更新node的左孩子域
00667                 }
00668                 else  //删后继
00669                 {
00670                     Node* cur = node->right_;
00671                     while (cur->left_ != nullptr)
00672                         cur = cur->left_;
00673                     node->data_ = cur->data_;
```

```
00674                     node->right_ = remove(node->right_, size_, cur->data_);
00675                 }
00676             }
00677             else if (node->left_ == nullptr && node->right_ == nullptr) //情形1
00678             {
00679                 delete node;
00680                 size_--;
00681                 return nullptr;
00682             }
00683             else  //情形2
00684             {
00685                 if (node->left_ != nullptr)
00686                 {
00687                     Node* child = node->left_;
00688                     child->parent_ = node->parent_;
00689                     delete node;
00690                     size_--;
00691                     return child;
00692                 }
00693                 else
00694                 {
00695                     Node* child = node->right_;
00696                     child->parent_ = node->parent_;
00697                     delete node;
00698                     size_--;
00699                     return child;
00700                 }
00701             }
00702         }
00703         else if (node->data_ > val)
00704         {
00705             node->left_ = remove(node->left_, size_, val);
00706 
00707             //*****左子树删除节点，可能造成右子树太高
00708             if (height(node->right_) - height(node->left_) > 1)
00709             {
00710                 if (height(node->right_->right_) >= height(node->right_->left_))
00711                 {
00712                     //右孩子的右子树太高，RR  做左旋操作
00713                     return leftRotate(node);
00714                 }
00715                 else //RL
00716                 {
00717                     return rightBalance(node);
00718                 }
00719             }
00720         }
00721         else
00722         {
00723             node->right_ = remove(node->right_, size_, val);
00724 
00725             //*****右子树删除节点，可能造成左子树太高
00726             if (height(node->left_) - height(node->right_) > 1)
00727             {
00728                 if (height(node->left_->left_) >= height(node->left_->right_))
00729                 {
00730                     //左孩子的左子树太高，LL  做右旋操作
00731                     return rightRotate(node);
00732                 }
00733                 else //RL
00734                 {
00735                     return leftBalance(node);
00736                 }
00737             }
00738         }
00739         //*****更新节点高度
00740         node->height_ = std::max(height(node->left_), height(node->right_)) + 1;
00741 
00742         return node; //把该句放上面两个判断里会出错。把当前节点返回父节点，更新父节点相应的地址域
00743     }
00744 
00745     //-----------------查询----------------------
00747     bool find(Node* node, const T& val) const
00748     {
00749         if (node == nullptr) { //递归退出条件，此时没有查到
00750             return false;
00751         }
00752         if (node->data_ == val) {
00753             return true;
00754         }
00755         else if (node->data_ > val) {
00756             find(node->left_, val);
00757         }
00758         else {
00759             find(node->right_, val);
00760         }
00761     }
```

```
00762     //AVL树节点总数查询递归实现
00763     /*int size(Node* node) const
00764     {
00765         if (node == nullptr){
00766             return 0;
00767         }
00768         int left_num = size(node->left_);
00769         int right_num = size(node->right_);
00770         return left_num + right_num + 1;
00771     }*/
00773     int high(Node* node) const
00774     {
00775         if (node == nullptr) {
00776             return 0;
00777         }
00778         int l_high = high(node->left_);
00779         int r_high = high(node->right_);
00780         return std::max(l_high, r_high) + 1;
00781         //return l_high > r_high ? l_high + 1 : r_high + 1;
00782     }
00784     int levelNum(Node* node, int i) const
00785     {
00786         if (node == nullptr) {
00787             return 0;
00788         }
00789         if (i == 0) {
00790             return 1;
00791         }
00792         int left = levelNum(node->left_, i - 1);
00793         int right = levelNum(node->right_, i - 1);
00794         return left + right;
00795     }
00797     int isBalance(Node* node, int l, bool& flag) const
00798     {
00799         if (node == nullptr)
00800         {
00801             return l;  //把当前节点高度返回回去
00802         }
00803
00804         int left = isBalance(node->left_, l + 1, flag);  //L
00805         if (!flag) { //已经失衡了，不用再往下继续判断了
00806             return left;
00807         }
00808         int right = isBalance(node->right_, l + 1, flag); //R
00809         if (!flag) {
00810             return right;
00811         }
00812
00813         if (abs(left - right) > 1) //节点失衡
00814         {
00815             flag = false;
00816         }
00817         return std::max(left, right); //给父节点返回高度（左右子树高度较大者）
00818     }
00819
00820     //-------------------遍历---------------------
00822     void preOrder(Node* node) const
00823     {
00824         if (node != nullptr)
00825         {
00826             std::cout << node->data_ << " "; //V
00827             preOrder(node->left_); //L
00828             preOrder(node->right_); //R
00829         }
00830     }
00832     void inOrder(Node* node) const
00833     {
00834         if (node != nullptr)
00835         {
00836             inOrder(node->left_); //L
00837             std::cout << node->data_ << " ";
00838             inOrder(node->right_); //R
00839         }
00840     }
00842     void postOrder(Node* node) const
00843     {
00844         if (node != nullptr)
00845         {
00846             postOrder(node->left_); //L
00847             postOrder(node->right_); //R
00848             std::cout << node->data_ << " ";
00849         }
00850     }
00852     void levelOrder(Node* node, int i) const
00853     {
00854         if (node == nullptr) {
00855             return;
```

```
00856         }
00857         if (i == 0)
00858         {
00859             std::cout << node->data_ << " ";
00860             return;
00861         }
00862         levelOrder(node->left_, i - 1);
00863         levelOrder(node->right_, i - 1);
00864     }
00865
00866     //-----------------辅助函数-------------------
00868     Node* copyTree(Node* node) {
00869         if (node == nullptr) {
00870             return nullptr;
00871         }
00872
00873         // 创建新节点（复制数据）
00874         Node* newNode = new Node(node->data_);
00875         newNode->height_ = node->height_;  // 复制高度值
00876         newNode->parent_ = node->parent_;
00877
00878         // 递归复制左右子树
00879         newNode->left_ = copyTree(node->left_);
00880         newNode->right_ = copyTree(node->right_);
00881
00882         return newNode;
00883     }
00885     void inOrderIndex(Node* node, const int index, int& m, T& result) const
00886     {
00887         if (node != nullptr)
00888         {
00889             inOrderIndex(node->left_, index, m, result); //L
00890             /*std::cout << node->data_ << " ";*/
00891             //if (m == index)
00892             //{
00893             //  std::cout << "m: " << m << std::endl;
00894             //  std::cout << node->data_ << std::endl;
00895             //}     //V
00896             if (m == index)
00897             {
00898                 result = node->data_;
00899             }
00900             m++;
00901             inOrderIndex(node->right_, index, m, result); //R
00902         }
00903     }
00904
00905 };
00906
00907 //--------------部分成员函数类外实现-------------- -
00909 template<class T>
00910 AvlTree1<T>& AvlTree1<T>::operator=(const AvlTree1<T>& avl){
00911 if (this != &avl) //防止自赋值误操作(避免"自杀式"资源释放)，如tree = tree;
00912 {
00913     AvlTree1 temp(avl);
00914     swap(*this, temp);
00915 } // 临时对象销毁（自动清理原资源）
00916 return *this;
00917 }
00918 //交换函数
00919 template<class T>
00920 void AvlTree1<T>::swap(AvlTree1<T>& first, AvlTree1<T>& second) noexcept {
00921     using std::swap;
00922     swap(first.root_, second.root_);
00923     swap(first.size_, second.size_);
00924 }
00930 template<class T>
00931 T& AvlTree1<T>::operator[](const int index) const {
00932     if (index < 0 || index >= size()) {
00933         throw "out of range!";
00934     }
00935     int m = 0; //计数器标志位
00936     return n_inOrder(index, m);
00937 }
00939 template<class T>
00940 T& AvlTree1<T>::n_inOrder(const int index, int& m) const {
00941     std::stack<Node*> s;
00942     //先把根节点开始的左孩子依次入栈
00943     Node* cur = root_;
00944     while (cur != nullptr)
00945     {
00946         s.push(cur);
00947         cur = cur->left_;
00948     }
00949
00950     while (!s.empty())
00951     {
```

```
00952         Node* top = s.top(); //获取栈顶元素
00953         s.pop();
00954         if (m == index) {
00955             return top->data_;
00956         }
00957         m++;
00958         //std::cout << top->data_ << " ";
00959 
00960         cur = top->right_; //右孩子
00961         while (cur != nullptr)
00962         {
00963             s.push(cur);
00964             cur = cur->left_; //右孩子要是左边还有节点，一直入
00965         }
00966     }
00967 }
00968 
00969 }
00970 }
```

## 6.3 test.cpp 文件参考

测试avl树

```
#include "myavl.hpp"
```

函数

- void show01 (xjcad::lcd::AvlTree1< int > &avl)
  遍历树等信息查询
- void show02 (xjcad::lcd::AvlTree1< int > &avl, xjcad::lcd::AvlTree1< int >::Iterator &it)
  迭代器双向遍历
- void test01 (xjcad::lcd::AvlTree1< int > &avl)
  测试插入、删除
- void test02 (xjcad::lcd::AvlTree1< int > &avl)
  测试拷贝与赋值
- void test03 (xjcad::lcd::AvlTree1< int > &avl, xjcad::lcd::AvlTree1< int >::Iterator &it)
  迭代器测试
- int main ()

### 6.3.1 详细描述

测试avl树

作者

lcd

日期

August 2025

在文件 test.cpp 中定义.

### 6.3.2 函数说明

#### 6.3.2.1 main()

```
int main ()
```

在文件 test.cpp 第 216 行定义.

#### 6.3.2.2 show01()

```
void show01 (
            xjcad::lcd::AvlTree1< int > & avl)
```

遍历树等信息查询

在文件 test.cpp 第 12 行定义.

#### 6.3.2.3 show02()

```
void show02 (
            xjcad::lcd::AvlTree1< int > & avl,
            xjcad::lcd::AvlTree1< int >::Iterator & it)
```

迭代器双向遍历

在文件 test.cpp 第 28 行定义.

#### 6.3.2.4 test01()

```
void test01 (
            xjcad::lcd::AvlTree1< int > & avl)
```

测试插入、删除

在文件 test.cpp 第 65 行定义.

#### 6.3.2.5 test02()

```
void test02 (
            xjcad::lcd::AvlTree1< int > & avl)
```

测试拷贝与赋值

在文件 test.cpp 第 90 行定义.

#### 6.3.2.6 test03()

```
void test03 (
            xjcad::lcd::AvlTree1< int > & avl,
            xjcad::lcd::AvlTree1< int >::Iterator & it)
```

迭代器测试

在文件 test.cpp 第 136 行定义.

## 6.4 test.cpp

浏览该文件的文档.

```
00001 /*****************************************************************/
00008
00009 #include "myavl.hpp"
00010
00012 void show01(xjcad::lcd::AvlTree1<int>& avl) {
00013     //遍历
00014     avl.preOrder();
00015     avl.inOrder();
00016     avl.postOrder();
00017     avl.levelOrder();
00018     avl.levelNum();
00019     std::cout << "树的结点总数为：" << avl.size() << std::endl;
00020     std::cout << "树的层数为：" << avl.high() << std::endl;
00021     std::cout << "avl树为空吗？ 标志：" << avl.empty() << std::endl;
00022     std::cout << "树的第一个元素为：" << avl.front() << std::endl;
00023     std::cout << "树的最后一个元素为：" << avl.back() << std::endl;
00024     std::cout << "-----------------------------------------------" << std::endl;
00025 }
00026
00028 void show02(xjcad::lcd::AvlTree1<int>& avl, xjcad::lcd::AvlTree1<int>::Iterator& it) {
00029
00030     std::cout << "前序遍历1：";
00031     for (; it != avl.end(); ++it)
00032     {
00033         std::cout << *it << " ";
00034     }
00035     std::cout << std::endl;
00036
00037     //C++11的foreach遍历
00038     std::cout << "for each前向遍历：";
00039     for (int x : avl)
00040     {
00041         std::cout << x << " ";
00042     }
00043     std::cout << std::endl;
00044
00045     //后序遍历输出
00046     std::cout << "后序遍历1：";
00047     xjcad::lcd::AvlTree1<int>::Iterator it1 = avl.last();
00048     for (; it1 != xjcad::lcd::AvlTree1<int>::Iterator(nullptr, &avl); it1--)
00049     {
00050         std::cout << *it1 << " ";
00051     }
00052     std::cout << std::endl;
00053
00054     std::cout << "后序遍历2：";
00055     xjcad::lcd::AvlTree1<int>::Iterator it2 = avl.last();
00056     for (; it2 >= avl.begin(); it2--) //it2--到最后肯定使得it2的成员变量_p指针变为nullptr的
00057     {
00058         std::cout << *it2 << " ";
00059     }
00060     std::cout << std::endl;
00061     std::cout << "------------------------" << std::endl;
00062 }
00063
00065 void test01(xjcad::lcd::AvlTree1<int>& avl) {
00066
00067     show01(avl);
00068
00069     //删除测试
00070     std::cout <<"树中是否存在结点9："<< avl.find(9) << std::endl;
00071     std::cout << "---删除9：" << std::endl;
00072     avl.remove(9);
```

```
00073     std::cout << "树中是否存在结点9: " << avl.find(9) << std::endl;
00074
00075     //std::cout << "---删除10: " << std::endl;
00076     //avl.remove(10);
00077
00078     //std::cout << "---删除6: " << std::endl;
00079     //avl.remove(6);
00081     //std::cout << "---删除1,2,3: " << std::endl;
00082     //avl.remove(1);
00083     //avl.remove(2);
00084     //avl.remove(3);
00085
00086     show01(avl);
00087 }
00088
00090 void test02(xjcad::lcd::AvlTree1<int>& avl) {
00091     // 深拷贝
00092     std::cout << "---深拷贝验证: " << std::endl;
00093     xjcad::lcd::AvlTree1<int> avl2(avl);
00094     std::cout << "拷贝构造后的树avl2为: ";
00095     avl2.inOrder();
00096     std::cout << "树avl2的节点总数为: " << avl2.size() << std::endl;
00097     std::cout << "树avl2的层数为: " << avl2.high() << std::endl;
00098     std::cout << "avl2 == avl? 标志: " << (avl2 == avl) << std::endl;
00099
00100     std::cout << "---深拷贝独立性验证: " << std::endl;
00101     avl.insert(42);  // 修改原树
00102     std::cout << "找到在原树avl中新插入的数了吗: " << avl.find(42) << std::endl;;
00103     std::cout << "拷贝树avl2找到在原树avl中新插入的数了吗: " << avl2.find(42) << std::endl;
00104     avl.remove(42);
00105     avl2.remove(5);
00106     std::cout << "原树avl中找到在拷贝树avl2中新删除的数了吗: " << avl.find(5) << std::endl;;
00107     std::cout << "找到在拷贝树avl2中新删除的数了吗: " << avl2.find(5) << std::endl;
00108     std::cout << "-----------------------------------------------" << std::endl;
00109
00110     //深拷贝赋值
00111     std::cout << "---赋值操作验证: " << std::endl;
00112     xjcad::lcd::AvlTree1<int> avl3;
00113     avl3 = avl;
00114     std::cout << "拷贝构造后的树avl3为: ";
00115     avl3.inOrder();
00116     std::cout << "树avl3的节点总数为: " << avl3.size() << std::endl;
00117     std::cout << "树avl3的层数为: " << avl3.high() << std::endl;
00118     std::cout << "avl3 == avl? 标志: " << (avl3 == avl) << std::endl;
00119
00120     std::cout << "---赋值独立性验证: " << std::endl;
00121     avl.insert(42);  // 修改原树
00122     std::cout << "找到在原树avl中新插入的数了吗: " << avl.find(42) << std::endl;;
00123     std::cout << "拷贝树avl3找到在原树avl中新插入的数了吗: " << avl3.find(42) << std::endl;  // 副本不受影响
00124     avl.remove(42);
00125
00126     avl2.insert(99);
00127     avl2.remove(5);
00128     std::cout << "原树avl中找到在拷贝树avl3中新删除的数了吗: " << avl.find(5) << std::endl;;
00129     std::cout << "原树avl中找到在拷贝树avl3中新插入的数了吗: " << avl.find(99) << std::endl;;
00130     std::cout << "找到在拷贝树avl2中新删除的数了吗: " << avl2.find(5) << std::endl;
00131     std::cout << "找到在拷贝树avl2中新插入的数了吗: " << avl2.find(99) << std::endl;
00132     std::cout << "-----------------------------------------------" << std::endl;
00133 }
00134
00136 void test03(xjcad::lcd::AvlTree1<int>& avl, xjcad::lcd::AvlTree1<int>::Iterator& it) {
00137
00138     //++、-- 测试（改变原迭代器）
00139     it = avl.begin();
00140     std::cout << "++、-- 测试（改变原迭代器）: " << std::endl;
00141     std::cout << "it: " << *it << std::endl;
00142     std::cout << "++it: " << *(++it) << std::endl;
00143     std::cout << "it++: " << *(it++) << std::endl;
00144     std::cout << "现在it为: " << *it << std::endl;
00145     std::cout << "--it: " << *(--it) << std::endl;
00146     std::cout << "it--: " << *(it--) << std::endl;
00147     std::cout << "现在it为: " << *it << std::endl;
00148     std::cout << "--------------------" << std::endl;
00149
00150     //+=、-= 测试（改变原迭代器）
00151     it = avl.begin();
00152     std::cout << "+=、-= 测试（改变原迭代器）: " << std::endl;
00153     std::cout << "it: " << *it << std::endl;
00154     it += 3;
00155     std::cout << "it += 3后it为: " << *it << std::endl;
00156     it -= 2;
00157     std::cout << "it -= 2后it为: " << *it << std::endl;
00158     it += 20;
00159     std::cout << "it += 20后it为(边界限定): " << *it << std::endl;
00160     it -= 20;
00161     std::cout << "it -= 20后it为(边界限定): " << *it << std::endl;
00162     std::cout << "--------------------" << std::endl;
```

```
00163
00164     //+、- 测试（返回新迭代器）
00165     it = avl.begin();
00166     std::cout << "+、- 测试（返回新迭代器）: " << std::endl;
00167
00168     std::cout << "it: " << *it << std::endl;
00169     auto it1 = it + 2;
00170     std::cout << "it1 = it + 2, it1为: " << *it1 << std::endl;
00171     std::cout << "it: " << *it << std::endl;
00172
00173     auto it2 = (it + 2) + 1;
00174     std::cout << "it2 = (it + 2) + 1, it2为: " << *it2 << std::endl;
00175
00176     auto it3 = it2 - 1;
00177     std::cout << "it3 = it2 - 1, it3为: " << *it3 << std::endl;
00178
00179     auto it4 = (it + 7) - 1;
00180     std::cout << "it4 =  (it + 7) - 1, it4为: " << *it4 << std::endl;
00181     std::cout << "it: " << *it << std::endl;
00182
00183     auto it5 = 2 + it;
00184     std::cout << "it5 = 2 + it, it5为: " << *it5 << std::endl;
00185
00186     auto it6 = (3 + it) - 1;
00187     std::cout << "it6 = (3 + it) - 1, it6为: " << *it6 << std::endl;
00188
00189     auto it7 = it + 20;
00190     std::cout << "it7 = it + 20, it7为(边界限定): " << *it7 << std::endl;
00191
00192     auto it8 = it6 - 20;
00193     std::cout << "it8 = it6 - 20, it8为(边界限定): " << *it8 << std::endl;
00194
00195     std::cout << "--------------------" << std::endl;
00196
00197     //删除erase测试
00198     std::cout << "删除迭代器指向位置测试: " << std::endl;
00199     std::cout << "it: " << *it << std::endl;
00200     avl.erase(it);
00201     it = avl.begin(); //删除会释放结点导致迭代器失效
00202     show02(avl, it);  //遍历里面有++it，最终it会变为end()，即为空
00203
00204     auto itx = avl.end();
00205     if (it == itx) {
00206         std::cout << "迭代器it此时指向end()" << std::endl;
00207     }
00208
00209     it = avl.begin(); //重置it
00210     avl.erase(it + 100);
00211     it = avl.begin(); //删除会释放结点导致迭代器失效
00212     show02(avl, it); //打印avl树遍历信息
00213 }
00214
00215
00216 int main()
00217 {
00218     xjcad::lcd::AvlTree1<int> avl = {};
00219
00220     for (int i = 1; i < 11; i++)
00221     {
00222         avl.insert(i);
00223     }
00224
00225     test01(avl);
00226
00227     //xjcad::lcd::AvlTree1<double> avl2;
00228     //std::cout << "树avl2为空吗?  标志: " << avl2.empty() << std::endl;
00229     //std::cout << "avl2是平衡树吗: " << avl2.isBalance() << std::endl;
00230
00231     //test02(avl);
00232
00233
00234     //索引访问元素
00235     //std::cout << "---索引访问元素: " << std::endl;
00236     //std::cout << avl.inOrderIndex(5) << std::endl; //调用函数访问中序遍历结果中索引为5的元素
00237     //std::cout << avl[5] << std::endl; //[]运算符重载访问元素
00239
00240     std::cout << "------------------------------------------------" << std::endl;
00241
00242
00243
00244     xjcad::lcd::AvlTree1<int>::Iterator it = avl.begin();
00245
00246     show02(avl, it); //打印avl树遍历信息
00247
00248     //test03(avl, it);
00249
00250
```

```
00251     return 0;
00252 }
```

# Index