Lesson06---map和set

[本节目标]

- **1. 关联式容器**
- **2. 键值对**
- **3. 树形结构的关联式容器**
- **4. 底层结构**

1. 关联式容器

> 在初阶阶段，我们已经接触过STL中的部分容器，比如：vector、list、deque、forward_list(C++11)等，这些容器统称为序列式容器，因为其底层为线性序列的数据结构，里面存储的是元素本身。那什么是关联式容器？它与序列式容器有什么区别？
>
> **关联式容器**也是用来存储数据的，与序列式容器不同的是，**其里面存储的是<key, value>结构的键值对，在数据检索时比序列式容器效率更高。**

2. 键值对

> **用来表示具有一一对应关系的一种结构，该结构中一般只包含两个成员变量key和value，key代表键值，value表示与key对应的信息。**比如：现在要建立一个英汉互译的字典，那该字典中必然有英文单词与其对应的中文含义，而且，英文单词与其中文含义是一一对应的关系，即通过该应该单词，在词典中就可以找到与其对应的中文含义。
>
> SGI-STL中关于键值对的定义：

```
template <class T1, class T2>
struct pair
{
typedef T1 first_type;
typedef T2 second_type;

T1 first;
T2 second;
pair(): first(T1()), second(T2())
{}

pair(const T1& a, const T2& b): first(a), second(b)
{}
};
```

3. 树形结构的关联式容器

> 根据应用场景的不桶，STL总共实现了两种不同结构的管理式容器：树型结构与哈希结构。**树型结构的关联式容器主要有四种：map、set、multimap、multiset**。这四种容器的共同点是：使用平衡搜索树(即红黑树)作为其底层结果，容器中的元素是一个有序的序列。下面一依次介绍每一个容器。

3.1 set

3.1.1 set的介绍

> set文档介绍
>
> 翻译：
>
> 1. set是按照一定次序存储元素的容器
> 2. 在set中，元素的value也标识它(value就是key，类型为T)，并且每个value必须是唯一的。set中的元素不能在容器中修改(元素总是const)，但是可以从容器中插入或删除它们。
> 3. 在内部，set中的元素总是按照其内部比较对象(类型比较)所指示的特定严格弱排序准则进行排序。
> 4. set容器通过key访问单个元素的速度通常比unordered_set容器慢，但它们允许根据顺序对子集进行直接迭代。
> 5. set在底层是用二叉搜索树(红黑树)实现的。
>
> 注意：
>
> 1. 与map/multimap不同，map/multimap中存储的是真正的键值对<key, value>，set中只放value，但在底层实际存放的是由<value, value>构成的键值对。
> 2. set中插入元素时，只需要插入value即可，不需要构造键值对。
> 3. set中的元素不可以重复(因此可以使用set进行去重)。
> 4. 使用set的迭代器遍历set中的元素，可以得到有序序列
> 5. set中的元素默认按照小于来比较
> 6. set中查找某个元素，时间复杂度为：$log_2 n$
> 7. set中的元素不允许修改(为什么?)
> 8. set中的底层使用二叉搜索树(红黑树)来实现。

3.1.2 set的使用

1. **set的模板参数列表**

   class template
   std::**set**

   ```
   template < class T,                        // set::key_type/value_type
              class Compare = less<T>,        // set::key_compare/value_compare
              class Alloc = allocator<T>      // set::allocator_type
              > class set;
   ```

   T: set中存放元素的类型，实际在底层存储<value, value>的键值对。

   Compare：set中元素默认按照小于来比较

   Alloc：set中元素空间的管理方式，使用STL提供的空间配置器管理

2. **set的构造**

函数声明	功能介绍
set (const Compare& comp = Compare(), const Allocator& = Allocator());	构造空的set
set (InputIterator first, InputIterator last, const Compare& comp = Compare(), const Allocator& = Allocator());	用[first, last)区间中的元素构造set
set (const set<Key,Compare,Allocator>& x);	set的拷贝构造

3. **set的迭代器**

函数声明	功能介绍

函数声明	功能介绍
iterator begin()	返回set中起始位置元素的迭代器
iterator end()	返回set中最后一个元素后面的迭代器
const_iterator cbegin() const	返回set中起始位置元素的const迭代器
const_iterator cend() const	返回set中最后一个元素后面的const迭代器
reverse_iterator rbegin()	返回set第一个元素的反向迭代器，即end
reverse_iterator rend()	返回set最后一个元素下一个位置的反向迭代器，即rbegin
const_reverse_iterator crbegin() const	返回set第一个元素的反向const迭代器，即cend
const_reverse_iterator crend() const	返回set最后一个元素下一个位置的反向const迭代器，即crbegin

4. set的容量

函数声明	功能介绍
bool empty ( ) const	检测set是否为空，空返回true，否则返回true
size_type size() const	返回set中有效元素的个数

5. set修改操作

函数声明	功能介绍

函数声明	功能介绍
pair<iterator,bool> insert (const value_type& x)	在set中插入元素x，实际插入的是<x, x>构成的键值对，如果插入成功，返回<该元素在set中的位置，true>,如果插入失败，说明x在set中已经存在，返回<x在set中的位置，false>
void erase (iterator position)	删除set中position位置上的元素
size_type erase (const key_type& x)	删除set中值为x的元素，返回删除的元素的个数
void erase (iterator first, iterator last)	删除set中[first, last)区间中的元素
void swap (set<Key,Compare,Allocator>& st);	交换set中的元素
void clear ( )	将set中的元素清空
iterator find (const key_type& x) const	返回set中值为x的元素的位置
size_type count (const key_type& x) const	返回set中值为x的元素的个数

6. **set的使用举例**

```
#include <set>
void TestSet()
{
    // 用数组array中的元素构造set
    int array[] = { 1, 3, 5, 7, 9, 2, 4, 6, 8, 0, 1, 3, 5, 7, 9, 2, 4, 6, 8, 0 };
    set<int> s(array, array+sizeof(array)/sizeof(array));
    cout << s.size() << endl;

    // 正向打印set中的元素，从打印结果中可以看出：set可去重
    for (auto& e : s)
        cout << e << " ";
    cout << endl;

    // 使用迭代器逆向打印set中的元素
    for (auto it = s.rbegin(); it != s.rend(); ++it)
        cout << *it << " ";
    cout << endl;

    // set中值为3的元素出现了几次
    cout << s.count(3) << endl;
}
```

3.2 map

3.2.1 map的介绍

> [map的文档简介]()

翻译：

1. map是关联容器，它按照特定的次序(按照key来比较)存储由键值key和值value组合而成的元素。
2. 在map中，键值key通常用于排序和惟一地标识元素，而值value中存储与此键值key关联的内容。键值key和值value的类型可能不同，并且在map的内部，key与value通过成员类型value_type绑定在一起，为其取别名称为pair:

typedef pair<const key, T> value_type;

3. 在内部，map中的元素总是按照键值key进行比较排序的。
4. map中通过键值访问单个元素的速度通常比unordered_map容器慢，但map允许根据顺序对元素进行直接迭代(即对map中的元素进行迭代时，可以得到一个有序的序列)。
5. map支持下标访问符，即在[]中放入key，就可以找到与key对应的value。
6. map通常被实现为二叉搜索树(更准确的说：平衡二叉搜索树(红黑树))。

3.2.2 map的使用

1. **map的模板参数说明**

class template
std::map

```
template < class Key,                                     // map::key_type
           class T,                                       // map::mapped_type
           class Compare = less<Key>,                     // map::key_compare
           class Alloc = allocator<pair<const Key,T> >    // map::allocator_type
           > class map;
```

key: 键值对中key的类型

T： 键值对中value的类型

Compare: 比较器的类型，map中的元素是按照key来比较的，缺省情况下按照小于来比较，一般情况下(内置类型元素)该参数不需要传递，如果无法比较时(自定义类型)，需要用户自己显式传递比较规则(一般情况下按照函数指针或者仿函数来传递)

Alloc：通过空间配置器来申请底层空间，不需要用户传递，除非用户不想使用标准库提供的空间配置器

注意：在使用map时，需要包含头文件。

2. **map的构造**

函数声明	功能介绍
map()	构造一个空的map

3. **map的迭代器**

函数声明	功能介绍
begin()和end()	begin:首元素的位置，end最后一个元素的下一个位置
cbegin()和cend()	与begin和end意义相同，但cbegin和cend所指向的元素不能修改
rbegin()和rend()	反向迭代器，rbegin在end位置，rend在begin位置，其++和--操作与begin和end操作移动相反
crbegin()和crend()	与rbegin和rend位置相同，操作相同，但crbegin和crend所指向的元素不能修改

4. **map的容量与元素访问**

函数声明	功能简介
bool empty ( ) const	检测map中的元素是否为空，是返回true，否则返回false
size_type size() const	返回map中有效元素的个数
mapped_type& operator[] (const key_type& k)	返回去key对应的value

问题：当key不在map中时，通过operator获取对应value时会发生什么问题？

```
mapped_type& operator[] (const key_type& k);
```

Access element

If *k* matches the key of an element in the container, the function returns a reference to its mapped value.

If *k* does not match the key of any element in the container, the function inserts a new element with that key and returns a reference to its mapped value. Notice that this always increases the container size by one, even if no mapped value is assigned to the element (the element is constructed using its default constructor).

A similar member function, map::at, has the same behavior when an element with the key exists, but throws an exception when it does not.

A call to this function is equivalent to:

```
(*((this->insert(make_pair(k,mapped_type()))).first)).second
```

注意：在元素访问时，有一个与operator[]类似的操作at()(该函数不常用)函数，都是通过key找到与key对应的value然后返回其引用，不同的是：**当key不存在时，operator[]用默认value与key构造键值对然后插入，返回该默认value，at()函数直接抛异常。**

5. **map中元素的修改**

函数声明	功能简介

函数声明	功能简介
pair<iterator,bool> insert (const value_type& x)	在map中插入键值对x，注意x是一个键值对，返回值也是键值对：iterator代表新插入元素的位置，bool代表释放插入成功
void erase (iterator position)	删除position位置上的元素
size_type erase (const key_type& x)	删除键值为x的元素
void erase (iterator first, iterator last)	删除[first, last)区间中的元素
void swap (map<Key,T,Compare,Allocator>& mp)	交换两个map中的元素
void clear ( )	将map中的元素清空
iterator find (const key_type& x)	在map中插入key为x的元素，找到返回该元素的位置的迭代器，否则返回end
const_iterator find (const key_type& x) const	在map中插入key为x的元素，找到返回该元素的位置的const迭代器，否则返回cend
size_type count (const key_type& x) const	返回key为x的键值在map中的个数，注意map中key是唯一的，因此该函数的返回值要么为0，要么为1，因此也可以用该函数来检测一个key是否在map中

```
#include <string>
#include <map>
void TestMap()
{
    map<string, string> m;

    // 向map中插入元素的方式：
    // 将键值对<"peach","桃子">插入map中，用pair直接来构造键值对
    m.insert(pair<string, string>("peach", "桃子"));

    // 将键值对<"peach","桃子">插入map中，用make_pair函数来构造键值对
    m.insert(make_pair("banan", "香蕉"));

    // 借用operator[]向map中插入元素
    /*
    operator[]的原理是：
      用<key, T()>构造一个键值对，然后调用insert()函数将该键值对插入到map中
      如果key已经存在，插入失败，insert函数返回该key所在位置的迭代器
      如果key不存在，插入成功，insert函数返回新插入元素所在位置的迭代器
      operator[]函数最后将insert返回值键值对中的value返回
    */
    // 将<"apple", "">插入map中，插入成功，返回value的引用，将“苹果”赋值给该引
用结果，
    m["apple"] = "苹果";

    // key不存在时抛异常
```

```
    //m.at("waterme") = "水蜜桃";
    cout << m.size() << endl;

    // 用迭代器去遍历map中的元素，可以得到一个按照key排序的序列
    for (auto& e : m)
        cout << e.first << "--->" << e.second << endl;
    cout << endl;

    // map中的键值对key一定是唯一的，如果key存在将插入失败
    auto ret = m.insert(make_pair("peach", "桃色"));
    if (ret.second)
        cout << "<peach, 桃色>不在map中, 已经插入" << endl;
    else
        cout << "键值为peach的元素已经存在：" << ret.first->first << "--->" << ret.first->second <<" 插入失败"<< endl;

    // 删除key为"apple"的元素
    m.erase("apple");

    if (1 == m.count("apple"))
        cout << "apple还在" << endl;
    else
        cout << "apple被吃了" << endl;
}
```

【总结】

1. map中的的元素是键值对
2. map中的key是唯一的，并且不能修改
3. 默认按照小于的方式对key进行比较
4. map中的元素如果用迭代器去遍历，可以得到一个有序的序列
5. map的底层为平衡搜索树(红黑树)，查找效率比较高$O(log_2 N)$
6. 支持[]操作符，operator[]中实际进行插入查找。

3.3 multiset

3.3.1 multiset的介绍

[multiset文档介绍]

[翻译]:

1. multiset是按照特定顺序存储元素的容器，其中元素是可以重复的。
2. 在multiset中，元素的value也会识别它(因为multiset中本身存储的就是<value, value>组成的键值对，因此value本身就是key，key就是value，类型为T). multiset元素的值不能在容器中进行修改(因为元素总是const的)，但可以从容器中插入或删除。
3. 在内部，multiset中的元素总是按照其内部比较规则(类型比较)所指示的特定严格弱排序准则进行排序。
4. multiset容器通过key访问单个元素的速度通常比unordered_multiset容器慢，但当使用迭代器遍历时会得到一个有序序列。
5. multiset底层结构为二叉搜索树(红黑树)。

注意：

1. multiset中再底层中存储的是<value, value>的键值对
2. mtltiset的插入接口中只需要插入即可
3. 与set的区别是，multiset中的元素可以重复，set是中value是唯一的
4. 使用迭代器对multiset中的元素进行遍历，可以得到有序的序列

5. multiset中的元素不能修改
6. 在multiset中找某个元素，时间复杂度为$O(log_2 N)$
7. multiset的作用：可以对元素进行排序

3.3.2 multiset的使用

此处只简单演示set与multiset的不同，其他接口接口与set相同，同学们可参考set。

```
#include <set>
void TestSet()
{
  int array[] = { 2, 1, 3, 9, 6, 0, 5, 8, 4, 7 };

     // 注意：multiset在底层实际存储的是<int, int>的键值对
     multiset<int> s(array, array + sizeof(array)/sizeof(array[0]));
     for (auto& e : s)
         cout << e << " ";
     cout << endl;
     return 0;
}
```

3.4 multimap

3.4.1 multimap的介绍

[multimap文档介绍]

翻译：

1. Multimaps是关联式容器，它按照特定的顺序，存储由key和value映射成的键值对<key, value>，其中多个键值对之间的key是可以重复的。
2. 在multimap中，通常按照key排序和惟一地标识元素，而映射的value存储与key关联的内容。key和value的类型可能不同，通过multimap内部的成员类型value_type组合在一起，value_type是组合key和value的键值对:

   `typedef pair<const Key, T> value_type;`

3. 在内部，multimap中的元素总是通过其内部比较对象，按照指定的特定严格弱排序标准对key进行排序的。
4. multimap通过key访问单个元素的速度通常比unordered_multimap容器慢，但是使用迭代器直接遍历multimap中的元素可以得到关于key有序的序列。
5. multimap在底层用二叉搜索树(红黑树)来实现。

注意：multimap和map的唯一不同就是：map中的key是唯一的，而multimap中key是可以重复的。

3.4.2 multimap的使用

multimap中的接口可以参考map，功能都是类似的。

注意：

1. multimap中的key是可以重复的。
2. multimap中的元素默认将key按照小于来比较
3. multimap中没有重载operator[]操作(同学们可思考下为什么?)。
4. 使用时与map包含的头文件相同：

3.5 在OJ中的使用

2. [前K个高频单词]

```
class Solution {
public:

    class Compare
    {
    public:
        // 在set中进行排序时的比较规则
        bool operator()(const pair<string, int>& left, const pair<string, int>& right)
        {
            return left.second > right.second;
        }
    };

    vector<string> topKFrequent(vector<string>& words, int k)
    {
        // 用<单词，单词出现次数>构建键值对，然后将vector中的单词放进去，统计每个单词出现的次数
        map<string, int> m;
        for (size_t i = 0; i < words.size(); ++i)
            ++(m[words[i]]);

        // 将单词按照其出现次数进行排序，出现相同次数的单词集中在一块
        multiset<pair<string, int>, Compare> ms(m.begin(), m.end());

        // 将相同次数的单词放在set中，然后再放到vector中
        set<string> s;
        size_t count = 0;   // 统计相同次数单词的个数
        size_t leftCount = k;

        vector<string> ret;
        for (auto& e: ms)
        {
            if (!s.empty())
            {
                // 相同次数的单词已经全部放到set中
                if (count != e.second)
                {
                    if (s.size() < leftCount)
                    {
                        ret.insert(ret.end(), s.begin(), s.end());
                        leftCount -= s.size();
                        s.clear();
                    }
                    else
                    {
                        break;
                    }
                }
            }

            count = e.second;
            s.insert(e.first);
        }
```

```
        for (auto& e : s)
        {
            if (0 == leftCount)
                break;

            ret.push_back(e);
            leftCount--;
        }
        return ret;
    }
};
```

3. [两个数组的交集I]()

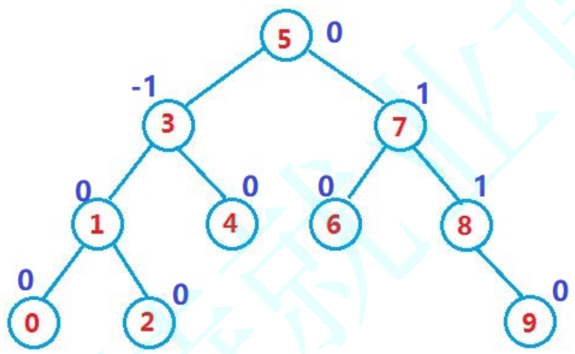
```
class Solution {
public:
    vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
        // 先去重
        set<int> s1;
        for(auto e : nums1)
        {
            s1.insert(e);
        }

        set<int> s2;
        for(auto e : nums2)
        {
            s2.insert(e);
        }

        // set排过序，依次比较，小的一定不是交集，相等的是交集
        auto it1 = s1.begin();
        auto it2 = s2.begin();
        vector<int> ret;
        while(it1 != s1.end() && it2 != s2.end())
        {
            if(*it1 < *it2)
            {
                it1++;
            }
            else if(*it2 < *it1)
            {
                it2++;
            }
            else
            {
                ret.push_back(*it1);
                it1++;
                it2++;
            }
        }

        return ret;
    }
};
```

4. 底层结构

> 前面对map/multimap/set/multiset进行了简单的介绍，在其文档介绍中发现，这几个容器有个共同点是：**其底层都是按照二叉搜索树来实现的**，但是二叉搜索树有其自身的缺陷，假如往树中插入的元素有序或者接近有序，二叉搜索树就会退化成单支树，时间复杂度会退化成O(N)，因此map、set等关联式容器的底层结构是对二叉树进行了平衡处理，即采用平衡树来实现。

4.1 AVL 树

4.1.1 AVL树的概念

> 二叉搜索树虽可以缩短查找的效率，但**如果数据有序或接近有序二叉搜索树将退化为单支树，查找元素相当于在顺序表中搜索元素，效率低下。**因此，两位俄罗斯的数学家G.M.Adelson-Velskii和E.M.Landis在1962年
> 发明了一种解决上述问题的方法：**当向二叉搜索树中插入新结点后，如果能保证每个结点的左右子树高度之差的绝对值不超过1(需要对树中的结点进行调整)**，即可降低树的高度，从而减少平均搜索长度。
>
> 一棵AVL树或者是空树，或者是具有以下性质的二叉搜索树：
>
> - **它的左右子树都是AVL树**
> - **左右子树高度之差(简称平衡因子)的绝对值不超过1(-1/0/1)**
>
> 
>
>
> **如果一棵二叉搜索树是高度平衡的，它就是AVL树。如果它有n个结点，其高度可保持在$O(log_2 n)$，搜索时间复杂度O($log_2 n$)。**

4.1.2 AVL树节点的定义

> AVL树节点的定义：
>
> ```
> template<class T>
> struct AVLTreeNode
> {
>     AVLTreeNode(const T& data)
>         : _pLeft(nullptr), _pRight(nullptr), _pParent(nullptr)
>         , _data(data), _bf(0)
>     {}
>
>     AVLTreeNode<T>* _pLeft;   // 该节点的左孩子
>     AVLTreeNode<T>* _pRight;  // 该节点的右孩子
>     AVLTreeNode<T>* _pParent; // 该节点的双亲
>     T _data;
>     int _bf;                  // 该节点的平衡因子
> };
> ```

4.1.3 AVL树的插入

AVL树就是在二叉搜索树的基础上引入了平衡因子，因此AVL树也可以看成是二叉搜索树。那么AVL树的插入过程可以分为两步：

1. **按照二叉搜索树的方式插入新节点**
2. **调整节点的平衡因子**

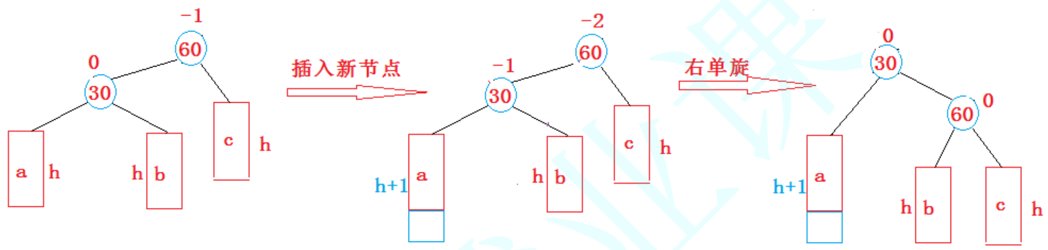
```
bool Insert(const T& data)
{
    // 1. 先按照二叉搜索树的规则将节点插入到AVL树中
    // ...

    // 2. 新节点插入后，AVL树的平衡性可能会遭到破坏，此时就需要更新平衡因子，并检测是否
破坏了AVL树
    //    的平衡性

    /*
     pCur插入后，pParent的平衡因子一定需要调整，在插入之前，pParent
     的平衡因子分为三种情况：-1，0, 1, 分以下两种情况：
      1. 如果pCur插入到pParent的左侧，只需给pParent的平衡因子-1即可
      2. 如果pCur插入到pParent的右侧，只需给pParent的平衡因子+1即可

     此时：pParent的平衡因子可能有三种情况：0，正负1， 正负2
      1. 如果pParent的平衡因子为0，说明插入之前pParent的平衡因子为正负1，插入后被调整
成0，此时满足
         AVL树的性质，插入成功
      2. 如果pParent的平衡因子为正负1，说明插入前pParent的平衡因子一定为0，插入后被更
新成正负1，此
         时以pParent为根的树的高度增加，需要继续向上更新
      3. 如果pParent的平衡因子为正负2，则pParent的平衡因子违反平衡树的性质，需要对其进
行旋转处理
    */

    while (pParent)
    {
        // 更新双亲的平衡因子
        if (pCur == pParent->_pLeft)
            pParent->_bf--;
        else
            pParent->_bf++;

        // 更新后检测双亲的平衡因子
        if (0 == pParent->_bf)
        {
            break;
        }
        else if (1 == pParent->_bf || -1 == pParent->_bf)
        {
            // 插入前双亲的平衡因子是0，插入后双亲的平衡因为为1 或者 -1 ，说明以双亲
为根的二叉树
            // 的高度增加了一层，因此需要继续向上调整
            pCur = pParent;
            pParent = pCur->_pParent;
        }
        else
        {
            // 双亲的平衡因子为正负2，违反了AVL树的平衡性，需要对以pParent
            // 为根的树进行旋转处理
```

```
                if(2 == pParent->_bf)
                {
                    // ...
                }
                else
                {
                    // ...
                }
            }
        }

        return true;
    }
```

4.1.4 AVL树的旋转

如果在一棵原本是平衡的AVL树中插入一个新节点，可能造成不平衡，此时必须调整树的结构，使之平衡化。根据节点插入位置的不同，AVL树的旋转分为四种：

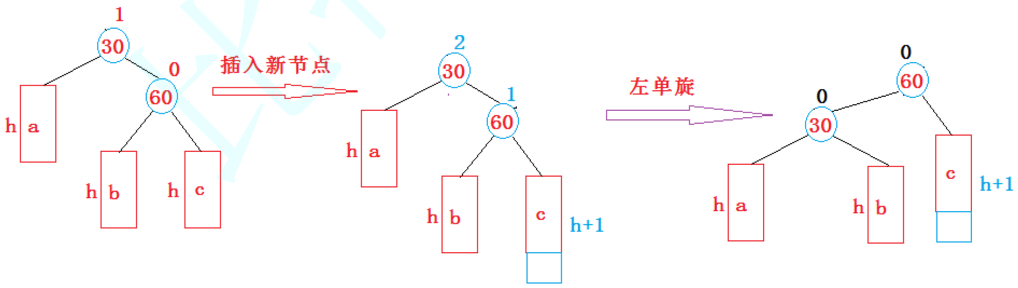
1. **新节点插入较高左子树的左侧---左左：右单旋**




```
/*
  上图在插入前，AVL树是平衡的，新节点插入到30的左子树(注意：此处不是左孩子)中，30左子树增加
  了一层，导致以60为根的二叉树不平衡，要让60平衡，只能将60左子树的高度减少一层，右子树增加一层，
  即将左子树往上提，这样60转下来，因为60比30大，只能将其放在30的右子树，而如果30有右子树，右子树根的值一定大于30，小于60，只能将其放在60的左子树，旋转完成后，更新节点的平衡因子即可。在旋转过程中，有以下几种情况需要考虑：
  1. 30节点的右孩子可能存在，也可能不存在
  2. 60可能是根节点，也可能是子树
     如果是根节点，旋转完成后，要更新根节点
     如果是子树，可能是某个节点的左子树，也可能是右子树

 同学们再此处可举一些详细的例子进行画图，考虑各种情况，加深旋转的理解
*/
void _RotateR(PNode pParent)
{
    // pSubL: pParent的左孩子
    // pSubLR: pParent左孩子的右孩子，注意：该
    PNode pSubL = pParent->_pLeft;
    PNode pSubLR = pSubL->_pRight;

    // 旋转完成之后，30的右孩子作为双亲的左孩子
    pParent->_pLeft = pSubLR;
    // 如果30的左孩子的右孩子存在，更新亲双亲
    if(pSubLR)
        pSubLR->_pParent = pParent;

    // 60 作为 30的右孩子
```

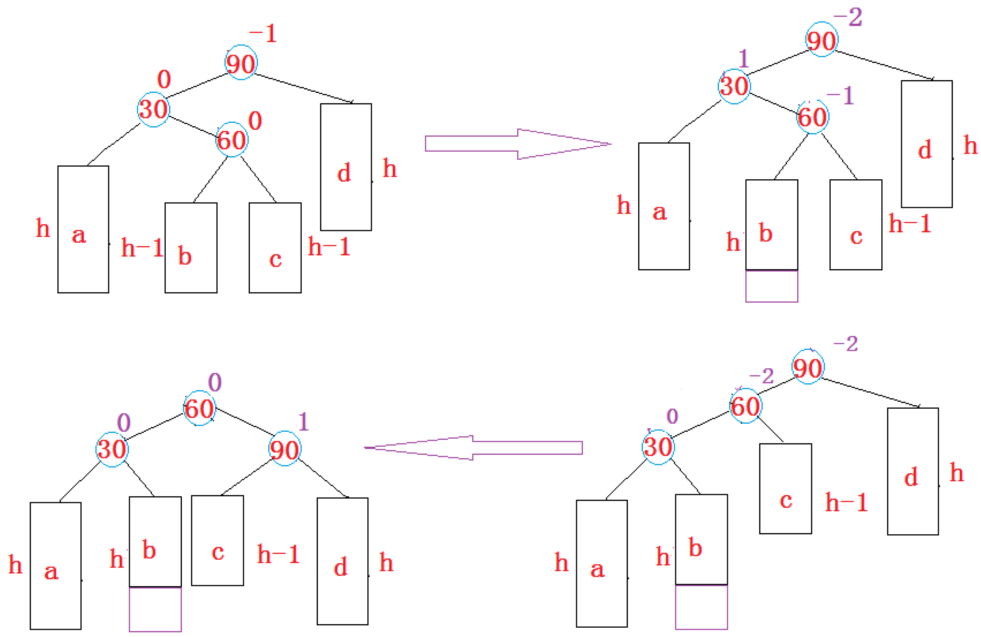
```
    pSubL->_pRight = pParent;

    // 因为60可能是棵子树，因此在更新其双亲前必须先保存60的双亲
    PNode pPParent = pParent->_pParent;

    // 更新60的双亲
    pParent->_pParent = pSubL;

    // 更新30的双亲
    pSubL->_pParent = pPParent;

    // 如果60是根节点，根新指向根节点的指针
    if(NULL == pPParent)
    {
        _pRoot = pSubL;
        pSubL->_pParent = NULL;
    }
    else
    {
         // 如果60是子树，可能是其双亲的左子树，也可能是右子树
        if(pPParent->_pLeft == pParent)
            pPParent->_pLeft = pSubL;
        else
            pPParent->_pRight = pSubL;
    }

    // 根据调整后的结构更新部分节点的平衡因子
    pParent->_bf = pSubL->_bf = 0;
}
```

2. **新节点插入较高右子树的右侧---右右：左单旋**



实现及情况考虑可参考右单旋。

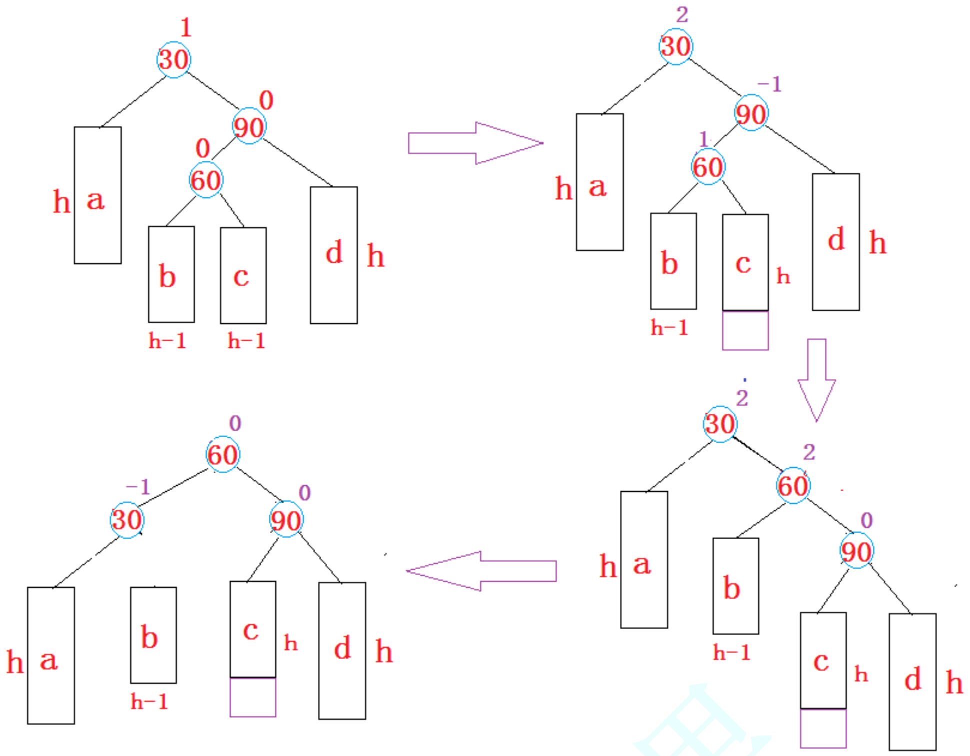
3. **新节点插入较高左子树的右侧---左右：先左单旋再右单旋**



将双旋变成单旋后再旋转，即：**先对30进行左单旋，然后再对90进行右单旋**，旋转完成后再考虑平衡因子的更新。

```
// 旋转之前，60的平衡因子可能是-1/0/1，旋转完成之后，根据情况对其他节点的平衡因子进行调整
void _RotateLR(PNode pParent)
{
    PNode pSubL = pParent->_pLeft;
    PNode pSubLR = pSubL->_pRight;

    // 旋转之前，保存pSubLR的平衡因子，旋转完成之后，需要根据该平衡因子来调整其他节点的平衡因子
    int bf = pSubLR->_bf;

    // 先对30进行左单旋
    _RotateL(pParent->_pLeft);

    // 再对90进行右单旋
    _RotateR(pParent);

    if(1 == bf)
        pSubL->_bf = -1;
    else if(-1 == bf)
        pParent->_bf = 1;
}
```

4. **新节点插入较高右子树的左侧---右左：先右单旋再左单旋**



参考右左双旋。

总结：

假如以pParent为根的子树不平衡，即pParent的平衡因子为2或者-2，分以下情况考虑

1. pParent的平衡因子为2，说明pParent的右子树高，设pParent的右子树的根为pSubR
   - 当pSubR的平衡因子为1时，执行左单旋
   - 当pSubR的平衡因子为-1时，执行右左双旋
2. pParent的平衡因子为-2，说明pParent的左子树高，设pParent的左子树的根为pSubL
   - 当pSubL的平衡因子为-1是，执行右单旋
   - 当pSubL的平衡因子为1时，执行左右双旋

旋转完成后，原pParent为根的子树个高度降低，已经平衡，不需要再向上更新。

4.1.5 AVL树的验证

AVL树是在二叉搜索树的基础上加入了平衡性的限制，因此要验证AVL树，可以分两步：

1. **验证其为二叉搜索树**

   如果中序遍历可得到一个有序的序列，就说明为二叉搜索树

2. **验证其为平衡树**
   - 每个节点子树高度差的绝对值不超过1(注意节点中如果没有平衡因子)
   - 节点的平衡因子是否计算正确

```
int _Height(PNode pRoot);
bool _IsBalanceTree(PNode pRoot)
{
    // 空树也是AVL树
    if (nullptr == pRoot) return true;

    // 计算pRoot节点的平衡因子：即pRoot左右子树的高度差
    int leftHeight = _Height(pRoot->_pLeft);
    int rightHeight = _Height(pRoot->_pRight);
    int diff = rightHeight - leftHeight;
```

```
        // 如果计算出的平衡因子与pRoot的平衡因子不相等，或者
        // pRoot平衡因子的绝对值超过1，则一定不是AVL树
        if (diff != pRoot->_bf || (diff > 1 || diff < -1))
            return false;

        // pRoot的左和右如果都是AVL树，则该树一定是AVL树
        return _IsBalanceTree(pRoot->_pLeft) && _IsBalanceTree(pRoot-
>_pRight);
    }
```

3. **验证用例**

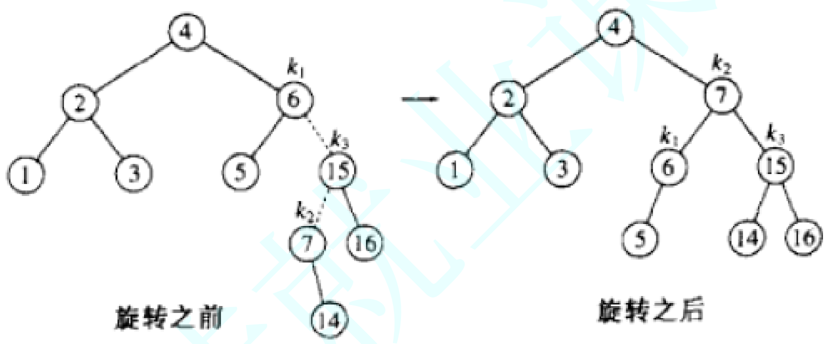
   请同学们结合上述代码按照以下的数据次序，自己动手画AVL树的创建过程，验证代码是否有漏洞。

   - 常规场景1

     {16, 3, 7, 11, 9, 26, 18, 14, 15}

   - 特殊场景2

     {4, 2, 6, 1, 3, 5, 15, 7, 16, 14}



4.1.6 AVL树的删除(了解)

> 因为AVL树也是二叉搜索树，可按照二叉搜索树的方式将节点删除，然后再更新平衡因子，只不错与删除不同的时，删除节点后的平衡因子更新，最差情况下一直要调整到根节点的位置。
>
> 具体实现学生们可参考《算法导论》或《数据结构-用面向对象方法与C++描述》殷人昆版。

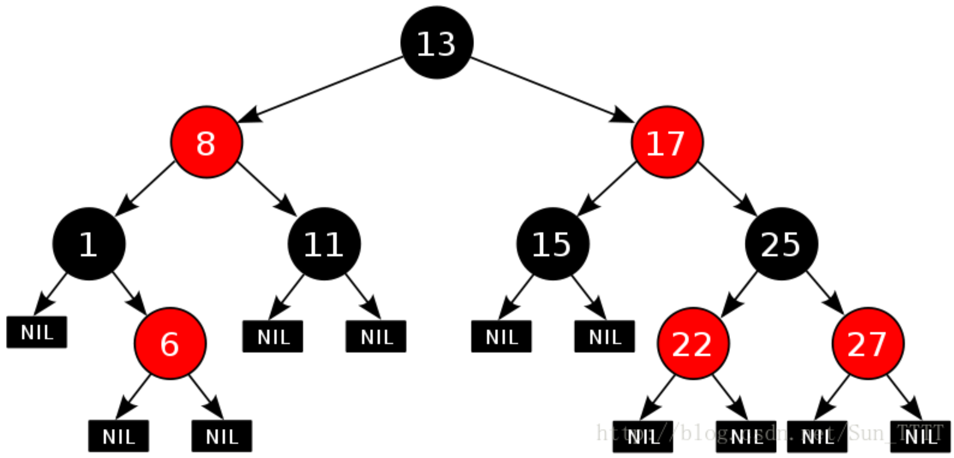
4.1.7 AVL树的性能

> AVL树是一棵绝对平衡的二叉搜索树，其要求每个节点的左右子树高度差的绝对值都不超过1，这样可以保证查询时高效的时间复杂度，即$log_2 (N)$。但是如果要对AVL树做一些结构修改的操作，性能非常低下，比如：插入时要维护其绝对平衡，旋转的次数比较多，更差的是在删除时，有可能一直要让旋转持续到根的位置。因此：如果需要一种查询高效且有序的数据结构，而且数据的个数为静态的(即不会改变)，可以考虑AVL树，但一个结构经常修改，就不太适合。

4.2 红黑树

4.2.1 红黑树的概念

> **红黑树**，是一种**二叉搜索树**，但**在每个结点上增加一个存储位表示结点的颜色，可以是Red或Black**。 通过对**任何一条从根到叶子的路径上各个结点着色方式的限制，红黑树确保没有一条路径会比其他路径长出俩倍**，因而是**接近平衡**的。



4.2.2 红黑树的性质

> 1. **每个结点不是红色就是黑色**
> 2. **根节点是黑色的**
> 3. **如果一个节点是红色的，则它的两个孩子结点是黑色的**
> 4. **对于每个结点，从该结点到其所有后代叶结点的简单路径上，均 包含相同数目的黑色结点**
> 5. **每个叶子结点都是黑色的(此处的叶子结点指的是空结点)**
>
> **思考：为什么满足上面的性质，红黑树就能保证：其最长路径中节点个数不会超过最短路径节点个数的两倍？**

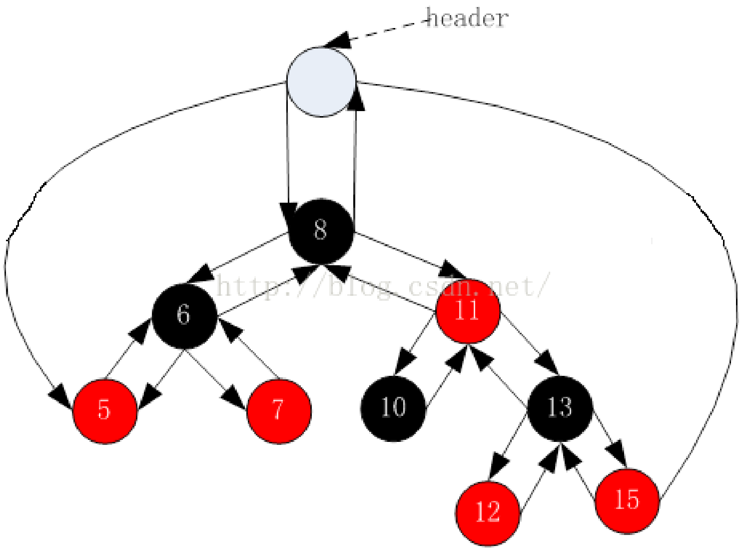
4.2.3 红黑树节点的定义

```
// 节点的颜色
enum Color{RED, BLACK};

// 红黑树节点的定义
template<class ValueType>
struct RBTreeNode
{
     RBTreeNode(const ValueType& data = ValueType(), Color color = RED)
          : _pLeft(nullptr), _pRight(nullptr), _pParent(nullptr)
          , _data(data), _color(color)
     {}

     RBTreeNode<ValueType>* _pLeft;   // 节点的左孩子
     RBTreeNode<ValueType>* _pRight;  // 节点的右孩子
     RBTreeNode<ValueType>* _pParent; // 节点的双亲(红黑树需要旋转，为了实现简单给出该字段)
     ValueType _data;            // 节点的值域
     Color _color;               // 节点的颜色
};
```

> **思考：在节点的定义中，为什么要将节点的默认颜色给成红色的？**

4.2.4 红黑树结构

> 为了后续实现关联式容器简单，红黑树的实现中增加一个头结点，因为跟节点必须为黑色，为了与根节点进行区分，将头结点给成黑色，并且让头结点的 pParent 域指向红黑树的根节点，pLeft域指向红黑树中最小的节点，_pRight域指向红黑树中最大的节点，如下：



4.2.5 红黑树的插入操作

红黑树是在二叉搜索树的基础上加上其平衡限制条件，因此红黑树的插入可分为两步：

1. **按照二叉搜索的树规则插入新节点**

```
template<class ValueType>
class RBTree
{
    //……
    bool Insert(const ValueType& data)
    {
        PNode& pRoot = GetRoot();
        if (nullptr == pRoot)
        {
            pRoot = new Node(data, BLACK);
            // 根的双亲为头节点
            pRoot->_pParent = _pHead;
            _pHead->_pParent = pRoot;
        }
        else
        {
            // 1. 按照二叉搜索的树方式插入新节点
             // 2. 检测新节点插入后，红黑树的性质是否造到破坏，
            //     若满足直接退出，否则对红黑树进行旋转着色处理
        }

        // 根节点的颜色可能被修改，将其改回黑色
        pRoot->_color = BLACK;
        _pHead->_pLeft = LeftMost();
        _pHead->_pRight = RightMost();
        return true;
    }
private:
    PNode& GetRoot(){ return _pHead->_pParent;}
    // 获取红黑树中最小节点，即最左侧节点
    PNode LeftMost();
    // 获取红黑树中最大节点，即最右侧节点
    PNode RightMost();
private:
    PNode _pHead;
```

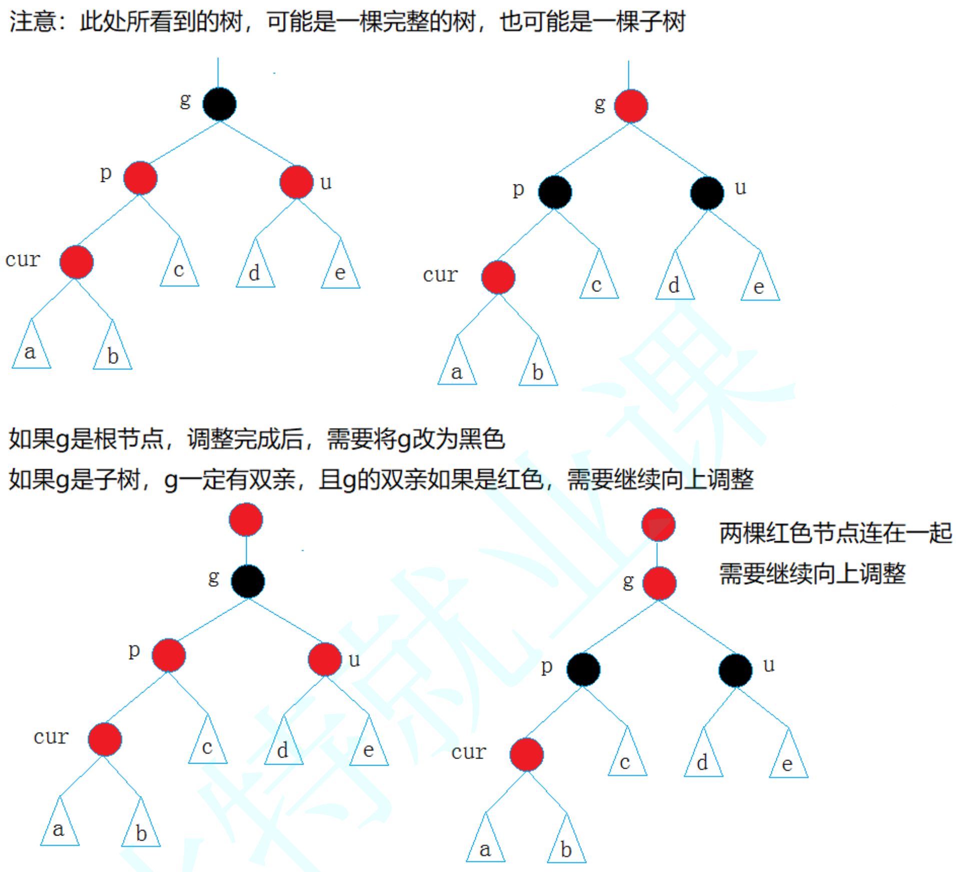
```
};
```

2. **检测新节点插入后，红黑树的性质是否造到破坏**

   因为**新节点的默认颜色是红色**，因此：如果**其双亲节点的颜色是黑色，没有违反红黑树任何性质**，则不需要调整；但**当新插入节点的双亲节点颜色为红色时，就违反了性质三不能有连在一起的红色节点**，此时需要对红黑树分情况来讨论：

   约定:cur为当前节点，p为父节点，g为祖父节点，u为叔叔节点

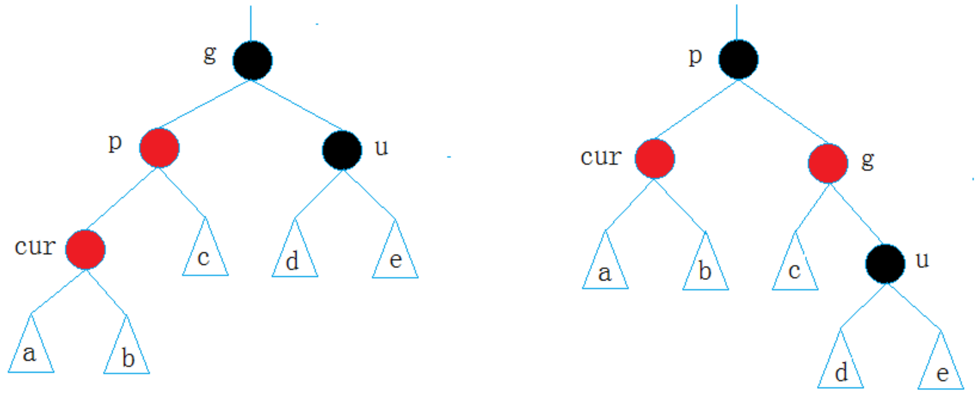
   - 情况一: **cur为红，p为红，g为黑，u存在且为红**

     cur和p均为红，违反了性质三，此处能否将p直接改为黑?

     解决方式：将p,u改为黑，g改为红，然后把g当成cur，继续向上调整。

   - 情况二: **cur为红，p为红，g为黑，u不存在/u存在且为黑**



说明：u的情况有两种

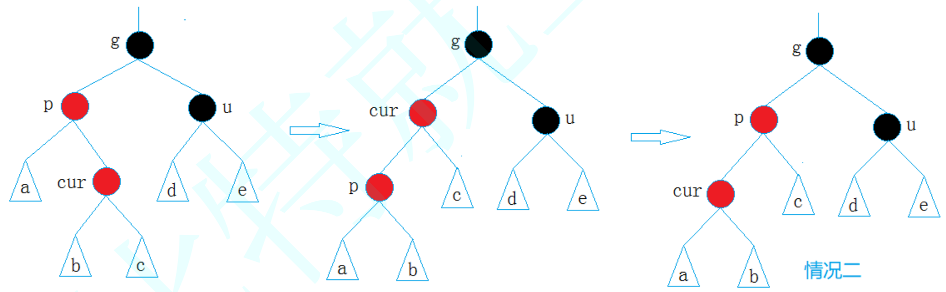
1. 如果u节点不存在，则cur一定是新插入节点，因为如果cur不是新插入节点，则cur和p一定有一个节点的颜色是黑色，就不满足性质4：每条路径黑色节点个数相同。
2. 如果u节点存在，则其一定是黑色的，那么cur节点原来的颜色一定是黑色的，现在看到其是红色的原因是因为cur的子树在调整的过程中将cur节点的颜色由黑色改成红色。

p为g的左孩子，cur为p的左孩子，则进行右单旋转；相反，

p为g的右孩子，cur为p的右孩子，则进行左单旋转

p、g变色--p变黑，g变红

- 情况三：**cur为红，p为红，g为黑，u不存在/u存在且为黑**



p为g的左孩子，cur为p的右孩子，则针对p做左单旋转；相反，

p为g的右孩子，cur为p的左孩子，则针对p做右单旋转

则转换成了情况2

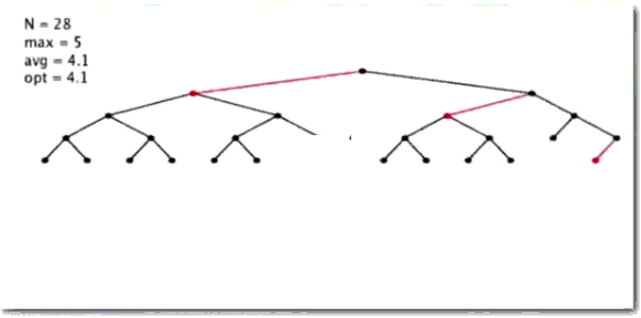
针对每种情况进行相应的处理即可。

```
bool Insert(const ValueType& data)
{
    // ...
    // 新节点插入后，如果其双亲节点的颜色为空色，则违反性质3：不能有连在一起的红色结
点
    while(pParent && RED == pParent->_color)
    {
        // 注意：grandFather一定存在
```

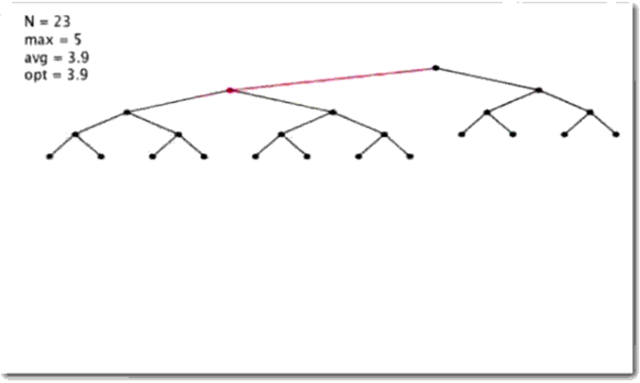
```
            // 因为pParent存在，且不是黑色节点，则pParent一定不是根，则其一定有双亲
            PNode grandFather = pParent->_pParent;
            // 先讨论左侧情况
            if(pParent == grandFather->_pLeft)
            {
                PNode unclue = grandFather->_pRight;
                // 情况三：叔叔节点存在，且为红
                if(unclue && RED == unclue->_color)
                {
                    pParent->_color = BLACK;
                    unclue->_color = BLACK;
                    grandFather->_color = RED;
                    pCur = grandFather;
                    pParent = pCur->_pParent;
                }
                else
                {
                    // 情况五：叔叔节点不存在，或者叔叔节点存在且为黑
                    if(pCur == pParent->_pRight)
                    {
                        _RotateLeft(pParent);
                        swap(pParent, pCur);
                    }

                     // 情况五最后转化成情况四
                    grandFather->_color = RED;
                    pParent->_color = BLACK;
                    _RotateRight(grandFather);
                }
            }
            else
            {
                // 右侧请学生们自己动手完成
            }
        }
        // ...
}
```

动态效果演示：

- **以升序(降序)插入构建红黑树**

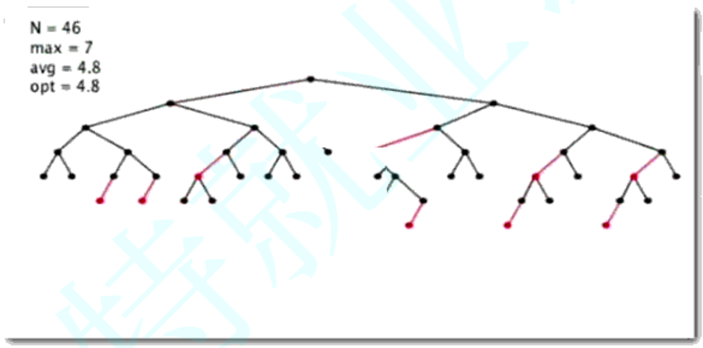


255 insertions in ascending order



255 insertions in descending order

- **随机插入构建红黑树**



255 random insertions

4.2.6 红黑树的验证

红黑树的检测分为两步:

1. 检测其是否满足二叉搜索树(中序遍历是否为有序序列)
2. 检测其是否满足红黑树的性质

```
bool IsValidRBTree()
{
	PNode pRoot = GetRoot();
	// 空树也是红黑树
	if (nullptr == pRoot)
		return true;

	// 检测根节点是否满足情况
	if (BLACK != pRoot->_color)
	{
		cout << "违反红黑树性质二：根节点必须为黑色" << endl;
		return false;
	}

	// 获取任意一条路径中黑色节点的个数
	size_t blackCount = 0;
```

```
    PNode pCur = pRoot;
    while (pCur)
    {
        if (BLACK == pCur->_color)
            blackCount++;

        pCur = pCur->_pLeft;
    }

    // 检测是否满足红黑树的性质，k用来记录路径中黑色节点的个数
    size_t k = 0;
    return _IsValidRBTree(pRoot, k, blackCount);
}

bool _IsValidRBTree(PNode pRoot, size_t k, const size_t blackCount)
{
    //走到null之后，判断k和black是否相等
    if (nullptr == pRoot)
    {
        if (k != blackCount)
        {
            cout << "违反性质四：每条路径中黑色节点的个数必须相同" << endl;
            return false;
        }
        return true;
    }

    // 统计黑色节点的个数
    if (BLACK == pRoot->_color)
        k++;

    // 检测当前节点与其双亲是否都为红色
    PNode pParent = pRoot->_pParent;
    if (pParent && RED == pParent->_color && RED == pRoot->_color)
    {
        cout << "违反性质三：没有连在一起的红色节点" << endl;
        return false;
    }

    return _IsValidRBTree(pRoot->_pLeft, k, blackCount) &&
           _IsValidRBTree(pRoot->_pRight, k, blackCount);
}
```

4.2.7 红黑树的删除

> 红黑树的删除本节不做讲解，有兴趣的同学可参考：《算法导论》或者《STL源码剖析》
>
> http://www.cnblogs.com/fornever/archive/2011/12/02/2270692.html

4.2.8 红黑树与AVL树的比较

> **红黑树和AVL树都是高效的平衡二叉树，增删改查的时间复杂度都是O($log_2 N$)，红黑树不追求绝对平衡，其只需保证最长路径不超过最短路径的2倍，相对而言，降低了插入和旋转的次数，所以在经常进行增删的结构中性能比AVL树更优，而且红黑树实现比较简单，所以实际运用中红黑树更多。**

4.2.9 红黑树的应用

> 1. **C++ STL库 -- map/set、mutil_map/mutil_set**

2. Java 库
3. linux内核
4. 其他一些库

http://www.cnblogs.com/yangecnu/p/Introduce-Red-Black-Tree.html

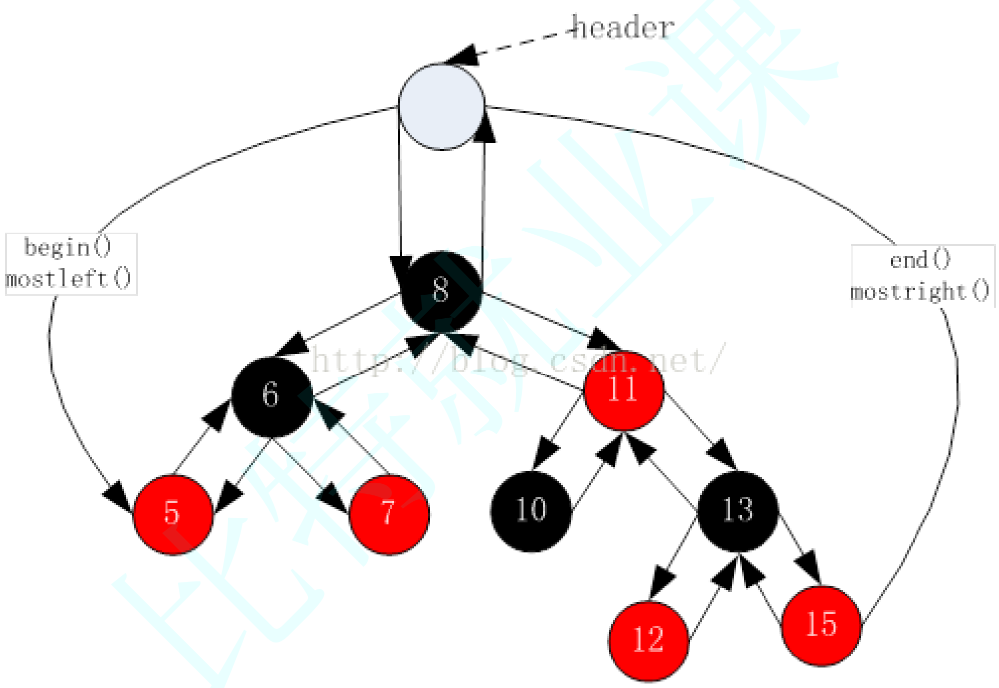
4.3 红黑树模拟实现STL中的map与set

4.3.1 红黑树的迭代器

迭代器的好处是可以方便遍历，是数据结构的底层实现与用户透明。如果想要给红黑树增加迭代器，需要考虑以前问题：

- **begin()与end()**

  STL明确规定，begin()与end()代表的是一段前闭后开的区间，而对红黑树进行中序遍历后，可以得到一个有序的序列，因此：**begin()可以放在红黑树中最小节点(即最左侧节点)的位置**，**end()放在最大节点(最右侧节点)的下一个位置**，关键是最大节点的下一个位置在哪块？能否给成nullptr呢？答案是行不通的，因为**对end()位置的迭代器进行--操作，必须要能找最后一个元素**，此处就不行，因此最好的方式是**将end()放在头结点的位置**：



- **operator++()与operator--()**

```
// 找迭代器的下一个节点，下一个节点肯定比其大
void Increasement()
{
    //分两种情况讨论:_pNode的右子树存在和不存在
    // 右子树存在
    if(_pNode->_pRight)
    {
         // 右子树中最小的节点，即右子树中最左侧节点
        _pNode = _pNode->_pRight;
        while(_pNode->_pLeft)
            _pNode = _pNode->_pLeft;
    }
    else
    {
         // 右子树不存在，向上查找，直到_pNode != pParent->right
        PNode pParent = _pNode->_pParent;
```

```
            while(pParent->_pRight == _pNode)
            {
                _pNode = pParent;
                pParent = _pNode->_pParent;
            }

            // 特殊情况：根节点没有右子树
            if(_pNode->_pRight != pParent)
                _pNode = pParent;
        }
    }

    // 获取迭代器指向节点的前一个节点
    void Decreasement()
    {
         //分三种情况讨论：_pNode 在head的位置，_pNode 左子树存在，_pNode 左子树不存在
         // 1. _pNode 在head的位置，--应该将_pNode放在红黑树中最大节点的位置
        if(_pNode->_pParent->_pParent == _pNode && _pNode->_color == RED)
            _pNode = _pNode->_pRight;
        else if(_pNode->_pLeft)
        {
            // 2. _pNode的左子树存在,在左子树中找最大的节点，即左子树中最右侧节点
            _pNode = _pNode->_pLeft;
            while(_pNode->_pRight)
                _pNode = _pNode->_pRight;
        }
        else
        {
             // _pNode的左子树不存在，只能向上找
            PNode pParent = _pNode->_pParent;
            while(_pNode == pParent->_pLeft)
            {
                _pNode = pParent;
                pParent = _pNode->_pParent;
            }
            _pNode = pParent;
        }
    }
```

4.3.2 改造红黑树

```
// 因为关联式容器中存储的是<key, value>的键值对，因此
// k为key的类型，
// ValueType: 如果是map，则为pair<K, V>; 如果是set，则为k
// KeyOfValue: 通过value来获取key的一个仿函数类
template<class K, class ValueType, class KeyOfValue>
class RBTree
{
     typedef RBTreeNode<ValueType> Node;
     typedef Node* PNode;
public:
     typedef RBTreeIterator<ValueType, ValueType*, ValueType&> Iterator;
public:
   RBTree();
     ~RBTree()
     ///////////////////////////////////////////////////////////////////
```

```
    // Iterator
    Iterator Begin(){ return Iterator(_pHead->_pLeft);}
    Iterator End(){ return Iterator(_pHead);}

    //////////////////////////////////////////////////////////////////
    // Modify
    pair<Iterator, bool> Insert(const ValueType& data)
    {
     // 插入节点并进行调整
     // 参考上文...
        return make_pair(Iterator(pNewNode), true);
    }

    // 将红黑树中的节点清空
    void Clear();
    Iterator Find(const K& key);

    //////////////////////////////////////////////////////////////////
    // capacity
    size_t Size()const;
    bool Empty()const;
  // ……
private:
    PNode _pHead;
    size_t _size;  // 红黑树中有效节点的个数
};
```

4.3.3 map的模拟实现

> map的底层结构就是红黑树，因此在map中直接封装一棵红黑树，然后将其接口包装下即可

```
namespace bite
{
    template<class K, class V>
    class map
    {
        typedef pair<K, V> ValueType;
     // 作用：将value中的key提取出来
        struct KeyOfValue
        {
            const K& operator()(const ValueType& v)
            { return v.first;}
        };

        typedef RBTree<K, ValueType, KeyOfValue> RBTree;
    public:
        typedef typename RBTree::Iterator iterator;
    public:
        map(){}
        /////////////////////////////////////////////
        // Iterator
        iterator begin(){ return _t.Begin();}
        iterator end(){ return _t.End();}
        /////////////////////////////////////////////////
        // Capacity
        size_t size()const{ return _t.Size();}
        bool empty()const{ return _t.Empty();}
```

```
        ////////////////////////////////////////////////
        // Acess
        V& operator[](const K& key)
        { return (*(_t.Insert(ValueType(key, V()))).first).second;}
        const V& operator[](const K& key)const;
        ////////////////////////////////////////////////////
        // modify
        pair<iterator, bool> insert(const ValueType& data) { return
_t.Insert(data);}
        void clear(){ _t.Clear();}
        iterator find(const K& key){ return _t.Find(key);}
    private:
        RBTree _t;
    };
}
```

4.3.4 set的模拟实现

set的底层为红黑树，因此只需在set内部封装一棵红黑树，即可将该容器实现出来(具体实现可参考map)。

```
namespace bit
{
  template<class K>
    class set
    {
        typedef K ValueType;
        // 作用是：将value中的key提取出来
        struct KeyOfValue
        {
            const K& operator()(const ValueType& key)
            { return key;}
        };

       // 红黑树类型重命名
        typedef RBTree<K, ValueType, KeyOfValue> RBTree;
    public:
        typedef typename RBTree::Iterator iterator;
    public:
        Set(){}
        ////////////////////////////////////////////
        // Iterator
        iterator Begin();
        iterator End();
        ////////////////////////////////////////////////
        // Capacity
        size_t size()const;
        bool empty()const;
        ////////////////////////////////////////////////////
        // modify
        pair<iterator, bool> insert(const ValueType& data)
        {
            return _t.Insert(data);
        }
        void clear();
        iterator find(const K& key);
```

```
    private:
        RBTree _t;
    };
}
```