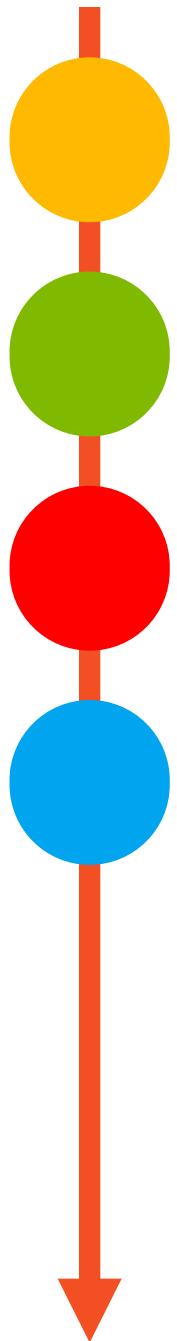


2018 TJMSC Tech. Courses

C++ STL

Zhen Luo
Tongji Microsoft Student Club



介绍

序列式容器

关联容器

常用算法

Most materials are referenced from CS451, David Kauchak

序列式容器

- Vector 容器
- Deque 双向队列
- List 链表

Vector

vector是STL中最常见的容器，它是一种**顺序容器**，支持随机访问。vector是一块连续分配的内存，从数据安排的角度来讲，和数组极其相似，不同的地方就是：数组是静态分配空间，一旦分配了空间的大小，就不可再改变了；而vector是**动态分配空间**，随着元素的不断插入，它会按照自身的一套机制不断扩充自身的容量。

vector的扩充机制：按照容器现在容量的一倍进行增长。vector容器分配的是一块连续的内存空间，每次容器的增长，并不是在原有连续的内存空间后再进行简单的叠加，而是**重新申请一块更大的新内存**，并把现有容器中的元素逐个复制过去，然后销毁旧的内存。这时原有指向旧内存空间的迭代器已经失效，所以当操作容器时，迭代器要及时更新。

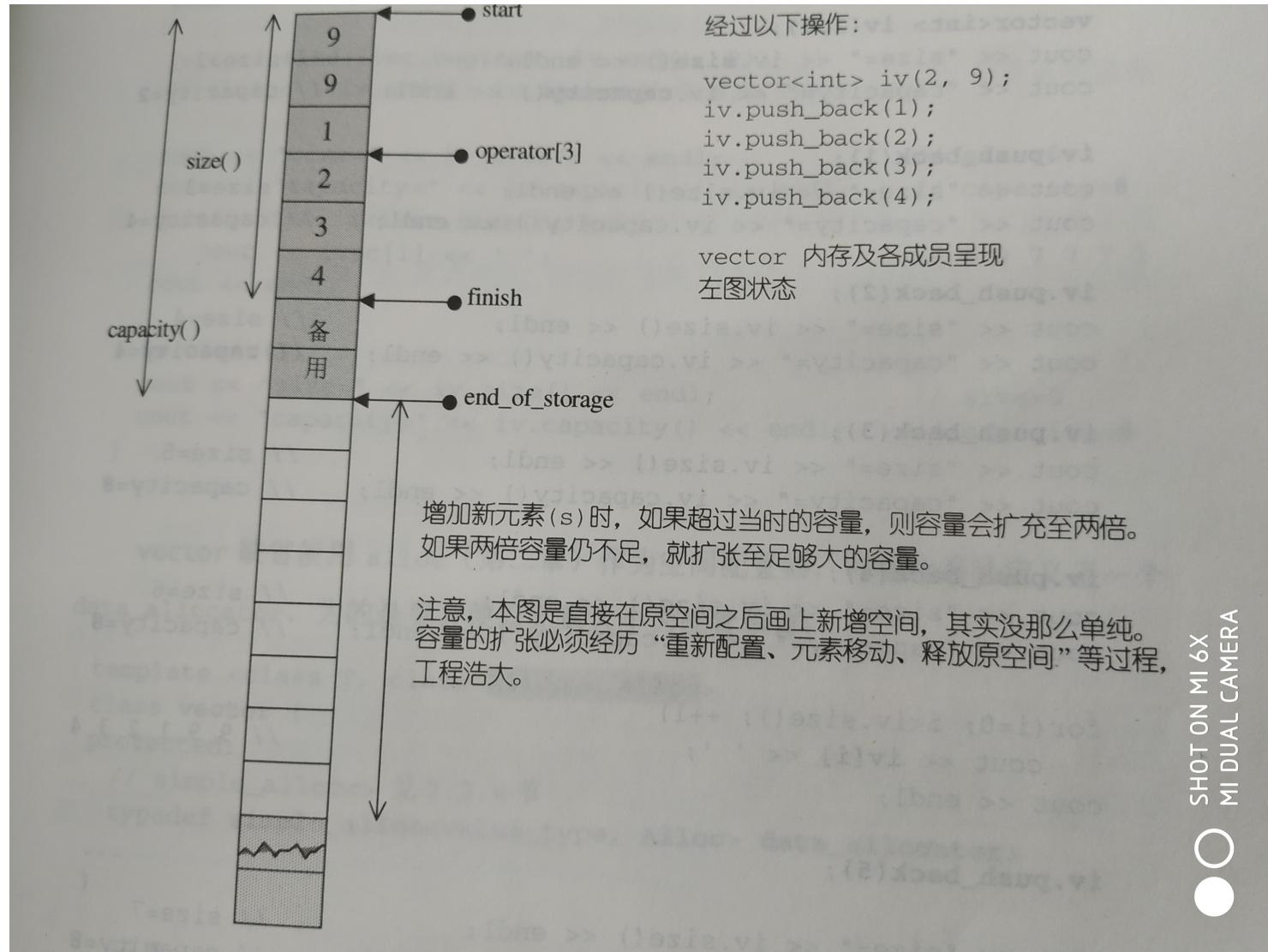
Vector

vector采用3个迭代器start、 finish、 end_of_storage来指向分配来的线性空间的不同范围，下面是声明3个迭代器变量的源代码。

```
template<class _Ty, class _A= allocator< _Ty> >
class vector{
    ...
protected:
    iterator start, finish, end_of_storage;
};
```

start指向使用空间的头部，finish指向使用数据空间大小（size）的尾部，end_of_storage指向使用空间容量（capacity）的尾部。

Vector



Vector

```
template <class T, class Alloc>
void vector<T, Alloc>::insert_aux(iterator position, const T& x) {
    if (finish != end_of_storage) {      //还有备用空间
        //在备用空间起始处构造一个元素，并以vector最后一个元素值为其初值
        construct(finish, *(finish - 1));
        //调整尾指针
        ++finish;
        T x_copy = x;
        //将 (position, finish-2) 中的数据向后移一格，即复值到 (position-1, finish-1) 中
        copy_backward(position, finish - 2, finish - 1);
        //将x的值插入指定position
        *position = x_copy;
    }
    else {      //已无备用空间
        //如果原大小为0，则配置1个元素大小
        //若不为0，配置两倍大小
        const size_type old_size = size();
        const size_type len = old_size != 0 ? 2 * old_size : 1;

        iterator new_start = data_allocator.allocate(len); //配置空间
        iterator new_finish = new_start;      //空间没有数据，另首位指针都指向空间开始位置
        try {
            //将 (position, new_start) 中的数据向后移一格，即复值到 (position-1, new_start) 中
            copy_backward(position, new_start, new_start + len - 1);
            //将x的值插入指定position
            *position = x;
            //将新大小设置为len
            size_type new_size = len;
            if (old_size != 0)
                new_size += old_size;
            resize(new_size);
        }
        catch (bad_alloc& e) {
            //清理新分配的空间
            data_allocator.deallocate(new_start, len);
            throw;
        }
    }
}
```

Vector

```
try {
    //复制原数据并插入新数据
    new_finish = uninitialized_copy(start, position, new_start);
    construct(new_finish, x);
    ++new_finish;
    new_finish = uninitialized_copy(position, finish, new_finish);
}
catch(...){
    //若出现异常，则析构数据并释放空间
    destroy(new_start, new_finish);
    data_allocator::deallocate(new_start, len);
    throw;
}

//析构并释放原vector
destroy(begin(), end());
deallocate();

//调整迭代器，指向新vector
start = new_start;
finish = new_finish;
end_of_storage = new_start + len;
};
```

Vector

1. 连续储存结构，支持高效的尾部插入，删除操作，但是在其他位置性能不高。
2. 可以动态扩展内存，任何扩展内存的操作都会引起迭代器失效
3. 支持高效的随机访问

List

非连续存储结构，具有双链表结构，每个元素维护一对前向和后向指针，因此支持前向/后向遍历。支持高效的随机插入/删除操作，但随机访问效率低下，且由于需要额外维护指针，内存开销也比较大。每一个结点都包括一个信息快Info、一个前驱指针Pre、一个后驱指针Post。

List

由于List是一个双向链表，所以内部只保留一个节点的指针即可遍历整个链表。

```
template<class T, class Alloc = alloc >
class list{
    ...
protected:
    link_type node;
};
```

link_type是一个指向节点的指针类型，node指向的是list的尾后节点，即end()；

List

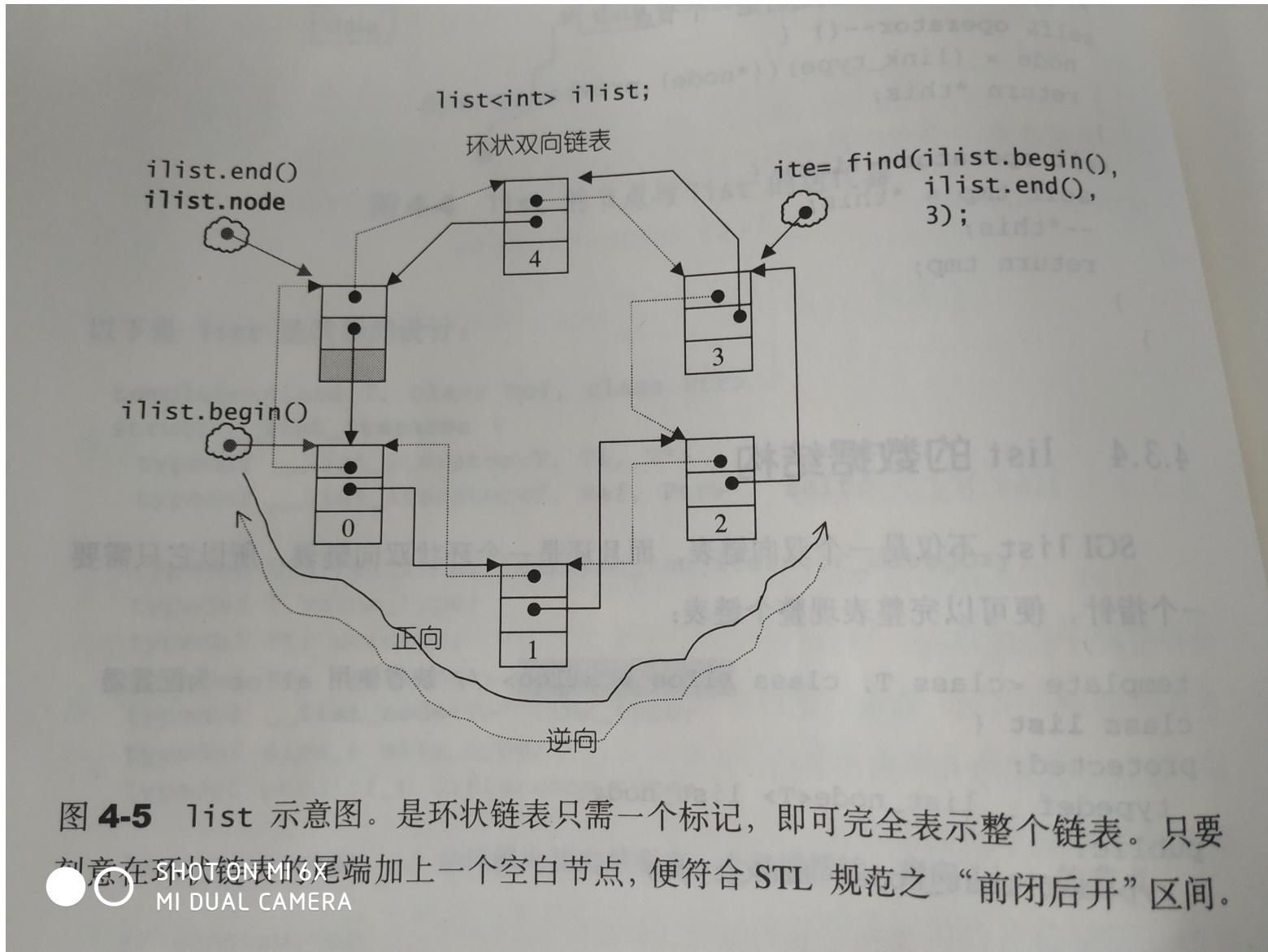


图 4-5 list 示意图。是环状链表只需一个标记，即可完全表示整个链表。只要刻意在环状链表的尾端加上一个空白节点，便符合 STL 规范之“前闭后开”区间。



SHOT ON MI 6X
MI DUAL CAMERA

List常用的方法

splice:

//将x接合与position所指的位置之前，x必须不同于*this

```
splice(iterator position, list& x)
```

//将i所指元素接合于position所指的位置之前。Position和i可指向同一个list

```
splice(iterator position, list&, iterator i)
```

//将[first,last)内的所有元素接合于position所指位置之前

//position不能位于[first,last)

```
void splice(iterator position, list&, iterator first, iterator last)
```

List常用的方法

splice:

//将x接合与position所指的位置之前，x必须不同于*this

```
splice(iterator position, list& x)
```

//将i所指元素接合于position所指的位置之前。Position和i可指向同一个list

```
splice(iterator position, list&, iterator i)
```

//将[first,last)内的所有元素接合于position所指位置之前

//position不能位于[first,last)

```
void splice(iterator position, list&, iterator first, iterator last)
```

List常用的方法

merge:

//将x合并到*this上。两个list的内容都必须先经过递增排序

```
merge(list<T,Alloc>& x)
```

reverse:

//将*this的内容逆向重置

```
reverse()
```

sort:

//将*this的内容递增排序

```
sort()
```

List

1. 非连续储存结构，支持高效的插入、删除操作。
2. 扩展内存方便，迭代器不会失效
3. 不支持随机访问

deque

分段连续结构，`deque`提供了两级数组结构，第一级完全类似于`array`，是一段定量的连续空间，代表实际容器；另一级维护容器的首位地址。这样，`deque`除了具有`vector`的所有功能外，还支持高效的首/尾端插入/删除操作。

`Deque`由一段一段定量连续空间构成，一旦有必要在`deque`的前端或尾端增加新空间，便配置一段定量连续空间，串联在整个`deque`的头端或尾端，`deque`的最大任务，便是在这些分段的定量连续空上，维护其整体连续的假象，并提供随机存取接口，避开了像`vector`那样扩充空间的开销，代价则是复杂的迭代器结构。

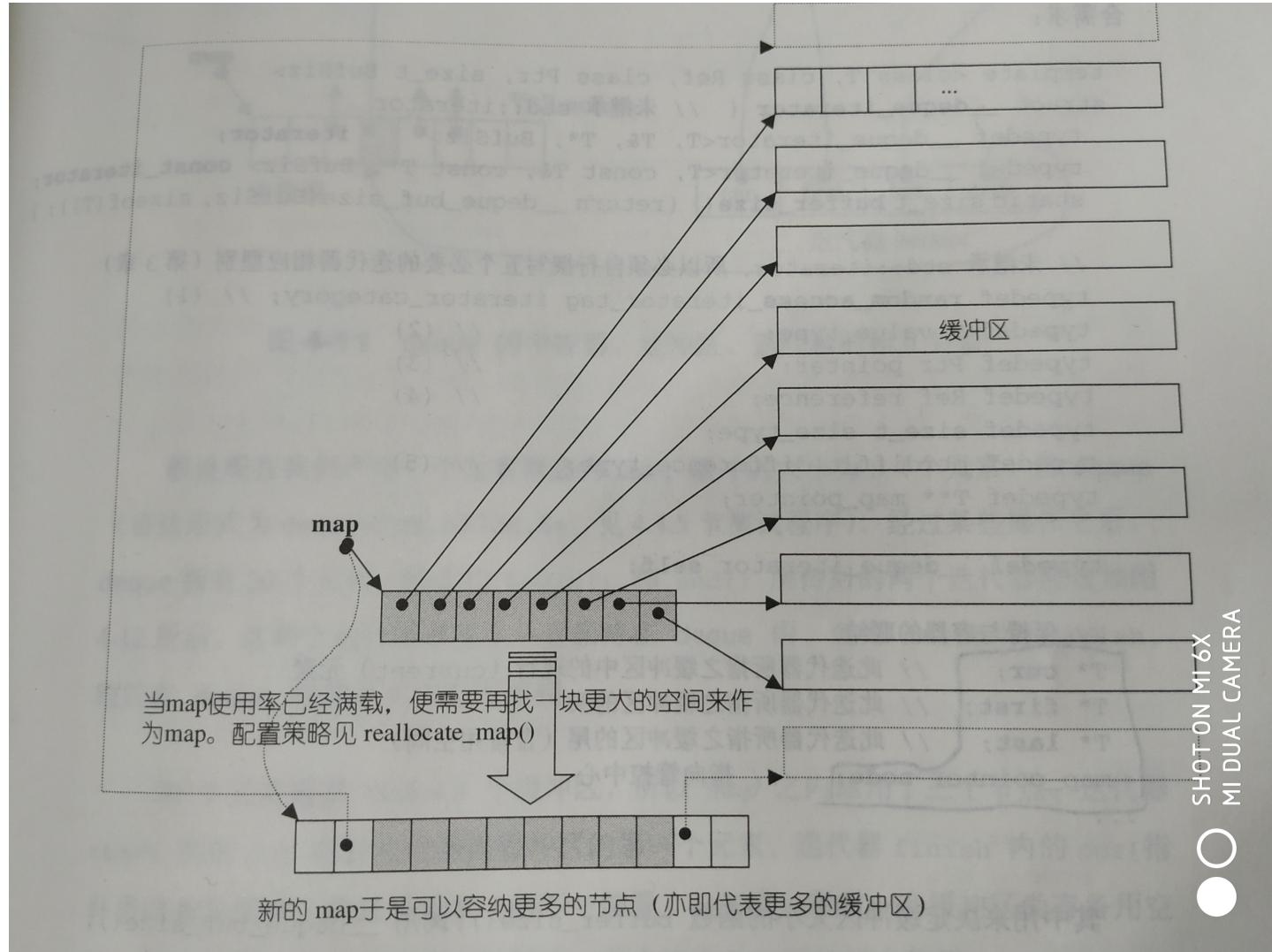
deque

Deque的数据结构：

```
template<class T, class Ref, class Rtr, size_t BufSize>
struct deque {
    ...
    iterator start;           //所有节点的首节点
    iterator finish;          //所有节点的尾后节点

    map_pointer map;          //指向中控器，其每个元素都是指针
    //指向缓冲区的首节点
    size_type map_size;       //中控器所含节点个数
};
```

deque

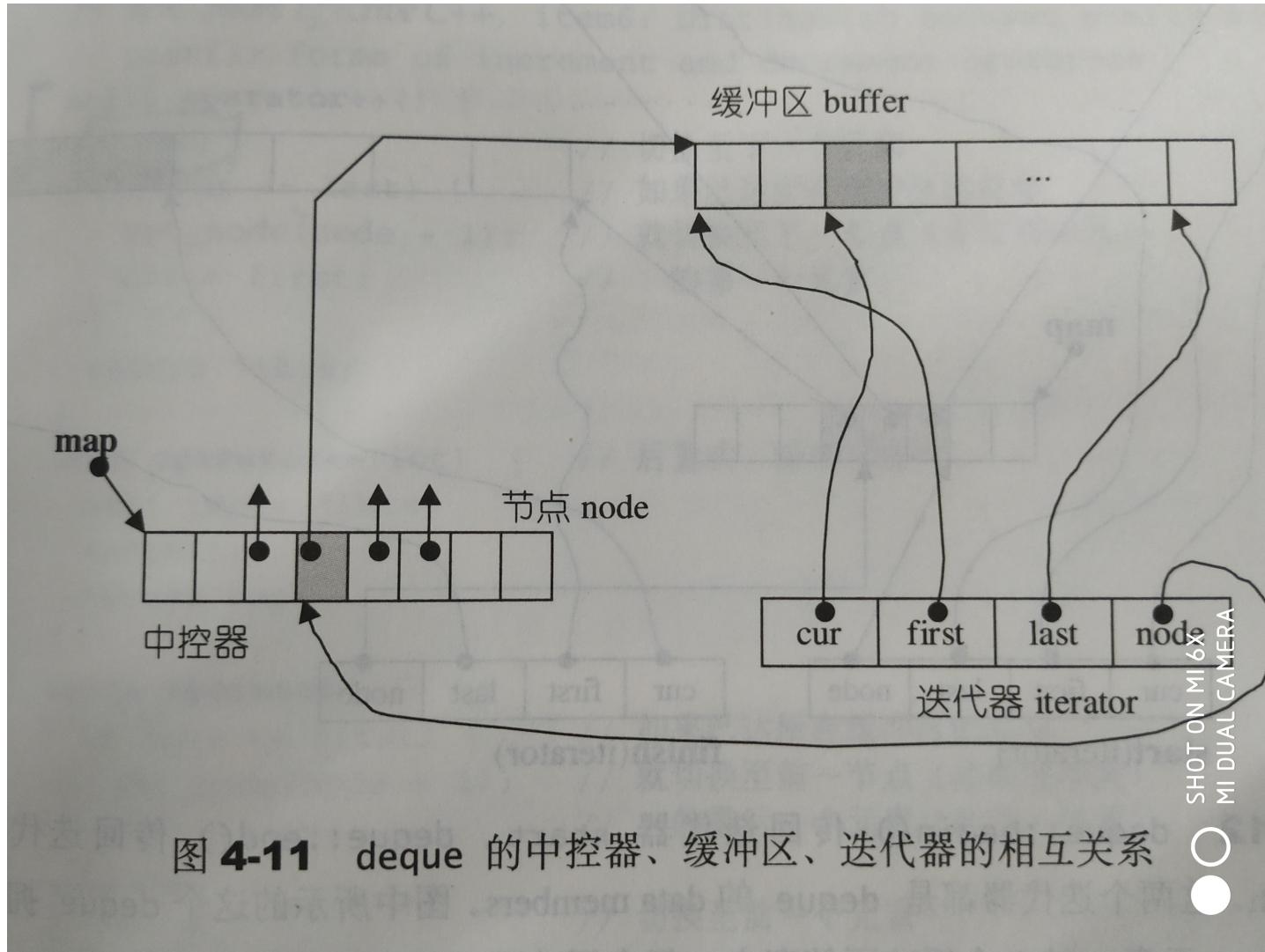


deque

Deque的迭代器：

```
template<class T, class Ref, class Rtr, size_t BufSize>
struct __deque_iterator{
    ...
    T* cur;          //此迭代器所指之缓冲区的当前元素
    T* first;        //此迭代器所指之缓冲区的首元素
    T* last;         //此迭代器所指之缓冲区的尾后元素
    map_pointer node; //指向缓冲区的中控器，即map
};
```

deque



deque

综合了list与vector的优点，并保留了一些无法避免的缺点：

1. 片段连续储存结构，插入、删除操作性能适中
2. 扩展内存性能等同于list，迭代器不会失效
3. 支持随机存取
4. 中控器、迭代器的维护需要额外开销

三者的取舍

1. 若需要随机访问操作，则选择vector；
2. 若已经知道需要存储元素的数目，则选择vector；
3. 若需要随机插入/删除（不仅仅在两端），则选择list
4. 只有需要在首端进行插入/删除操作的时候，还要兼顾随机访问效率，才选择deque，否则都选择vector。
5. 若既需要随机插入/删除，又需要随机访问，则需要在vector与list间做个折中-deque。
6. 当要存储的是大型负责类对象时，list要优于vector；当然这时候也可以用vector来存储指向对象的指针，同样会取得较高的效率，但是指针的维护非常容易出错，因此不推荐使用。

关联式容器

- RB-tree 红黑树
- map
- hash_table 哈希表

红黑树

树：

树是一种[抽象数据类型](#)（ADT）或是实现这种抽象数据类型的[数据结构](#)，用来模拟[具有树状结构](#)性质的数据集合。它是由 n ($n > 0$) 个有限节点组成一个具有层次关系的[集合](#)。把它叫做“树”是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。它具有以下的特点：

每个节点都只有有限个子节点或无子节点；

没有父节点的节点称为根节点；

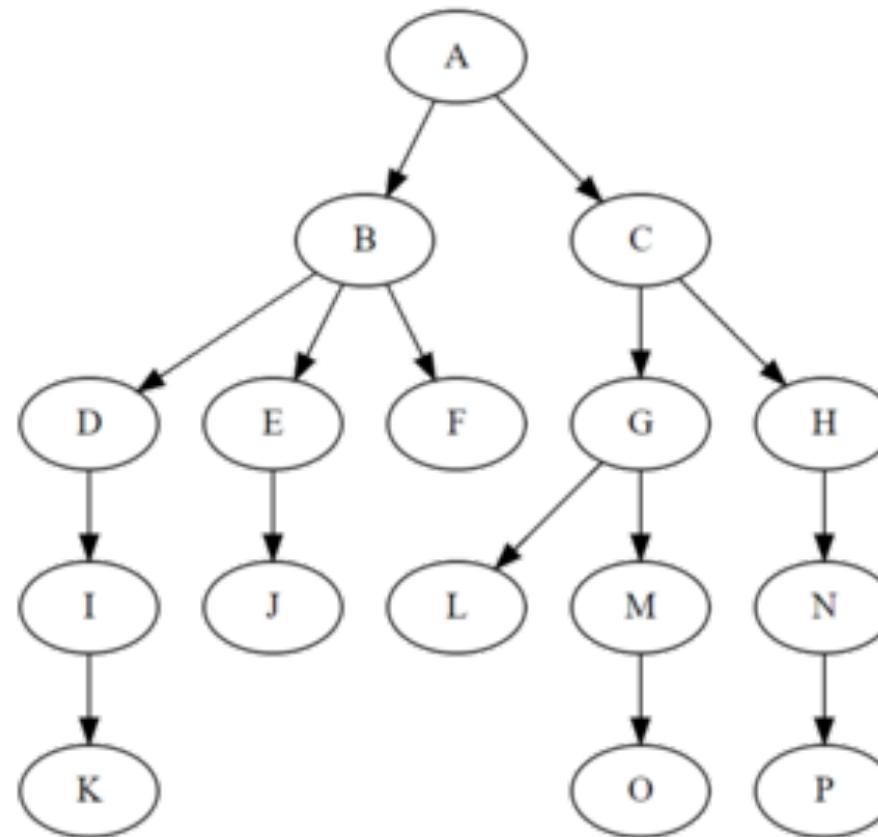
每一个非根节点有且只有一个父节点；

除了根节点外，每个子节点可以分为多个不相交的子树；

树里面没有环路(cycle)

红黑树

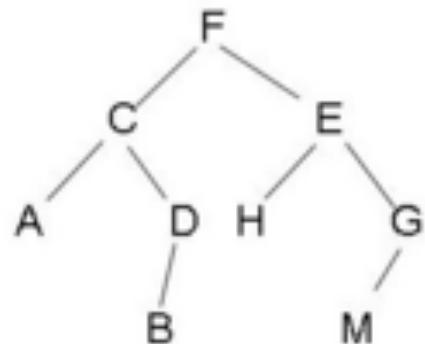
树：



红黑树

二叉树：

一个节点最多有两个子节点的树就叫二叉树



红黑树

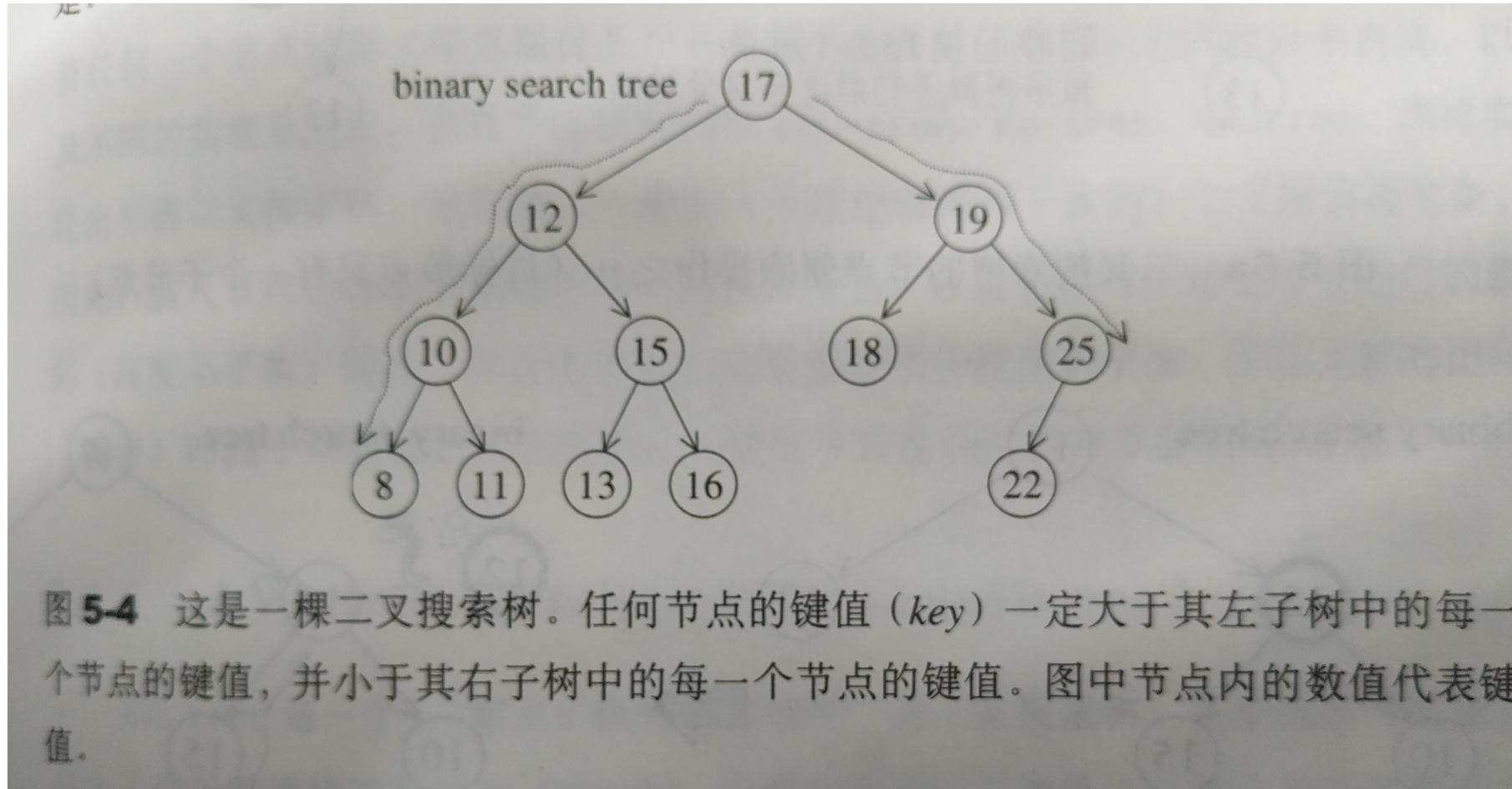
二叉搜索树：

对于list、vector等容器来说，容器内部的元素都是无序的，查找某个元素需要遍历整个数组。而在二叉搜索树中，节点放置有一定规律：**左子节点必小于父节点，右子节点必大于父节点。**

如果二叉查找树的所有非叶子结点的左右子树的结点数目均保持差不多（平衡），那么二叉查找树的搜索性能逼近二分查找，即可提供对数时间的元素插入合访问。

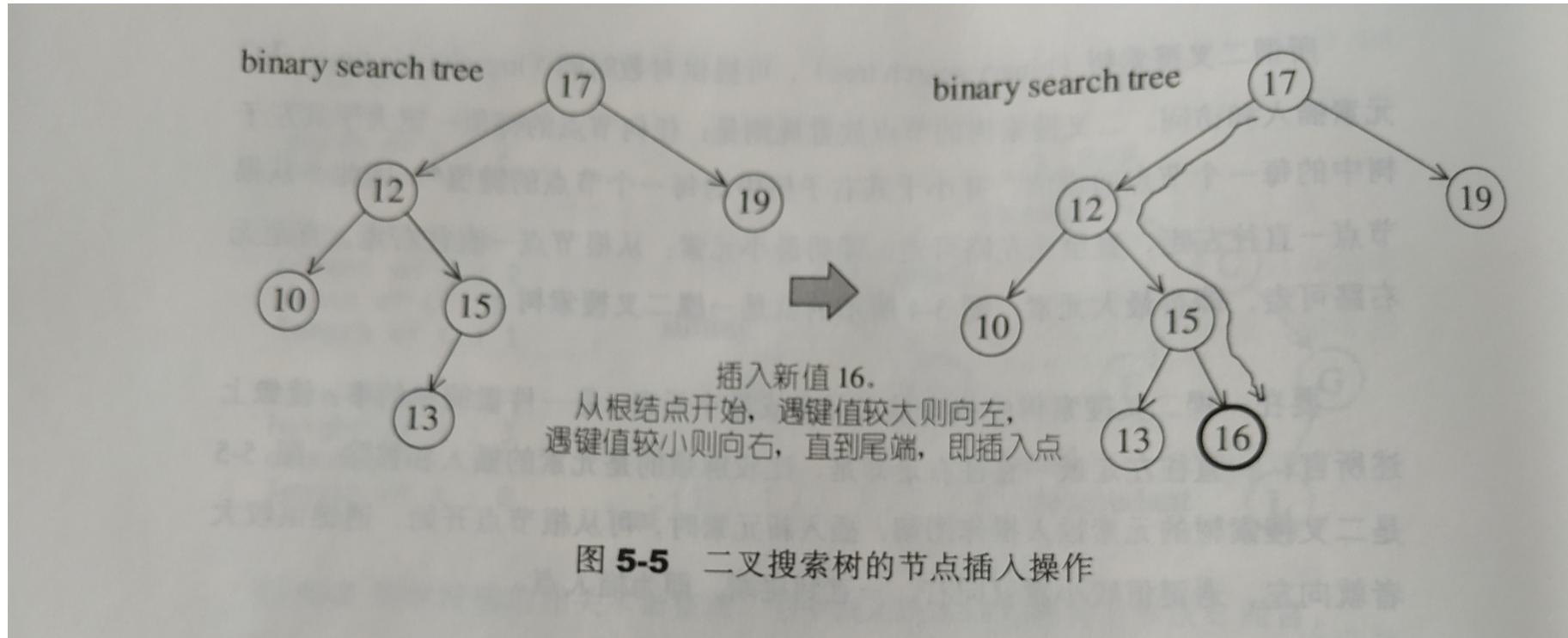
红黑树

查找：



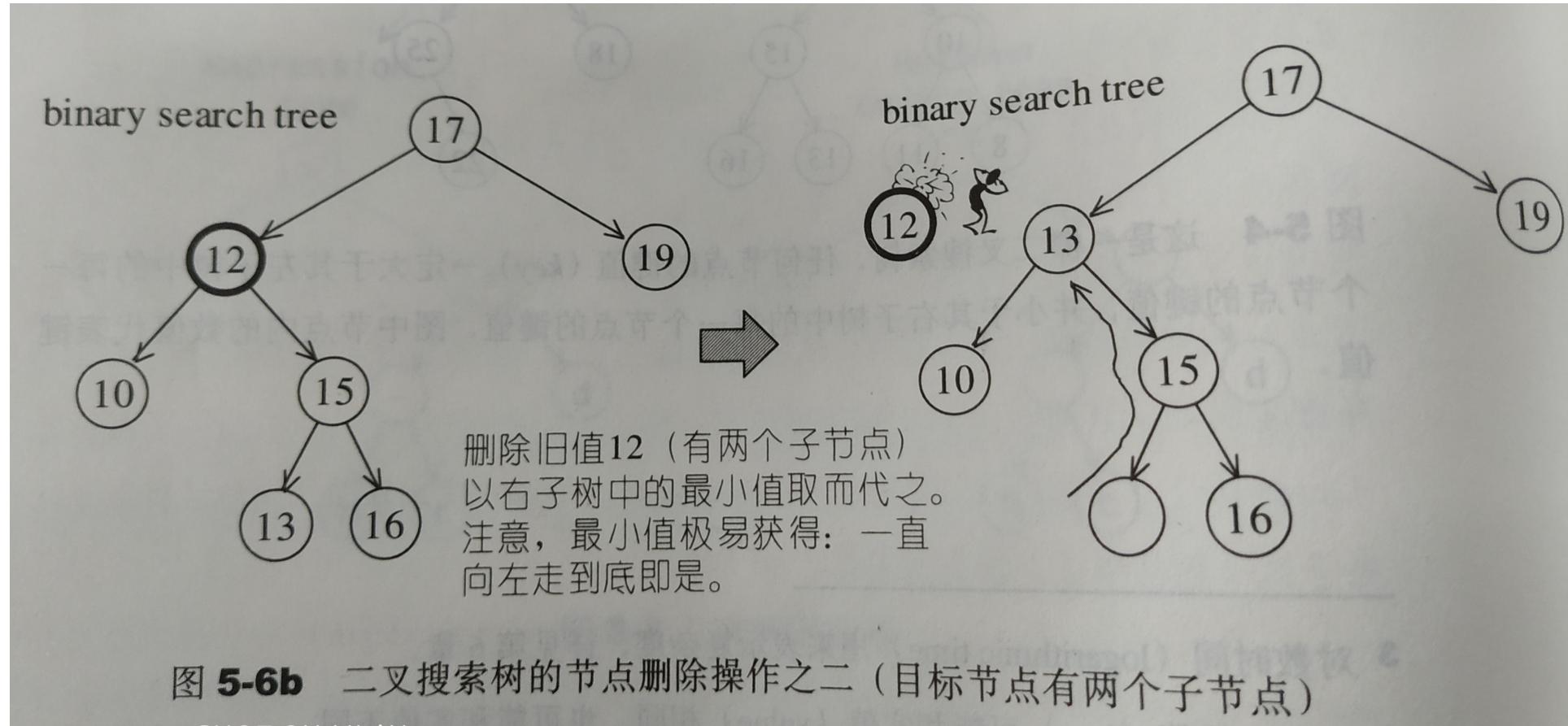
红黑树

插入：



红黑树

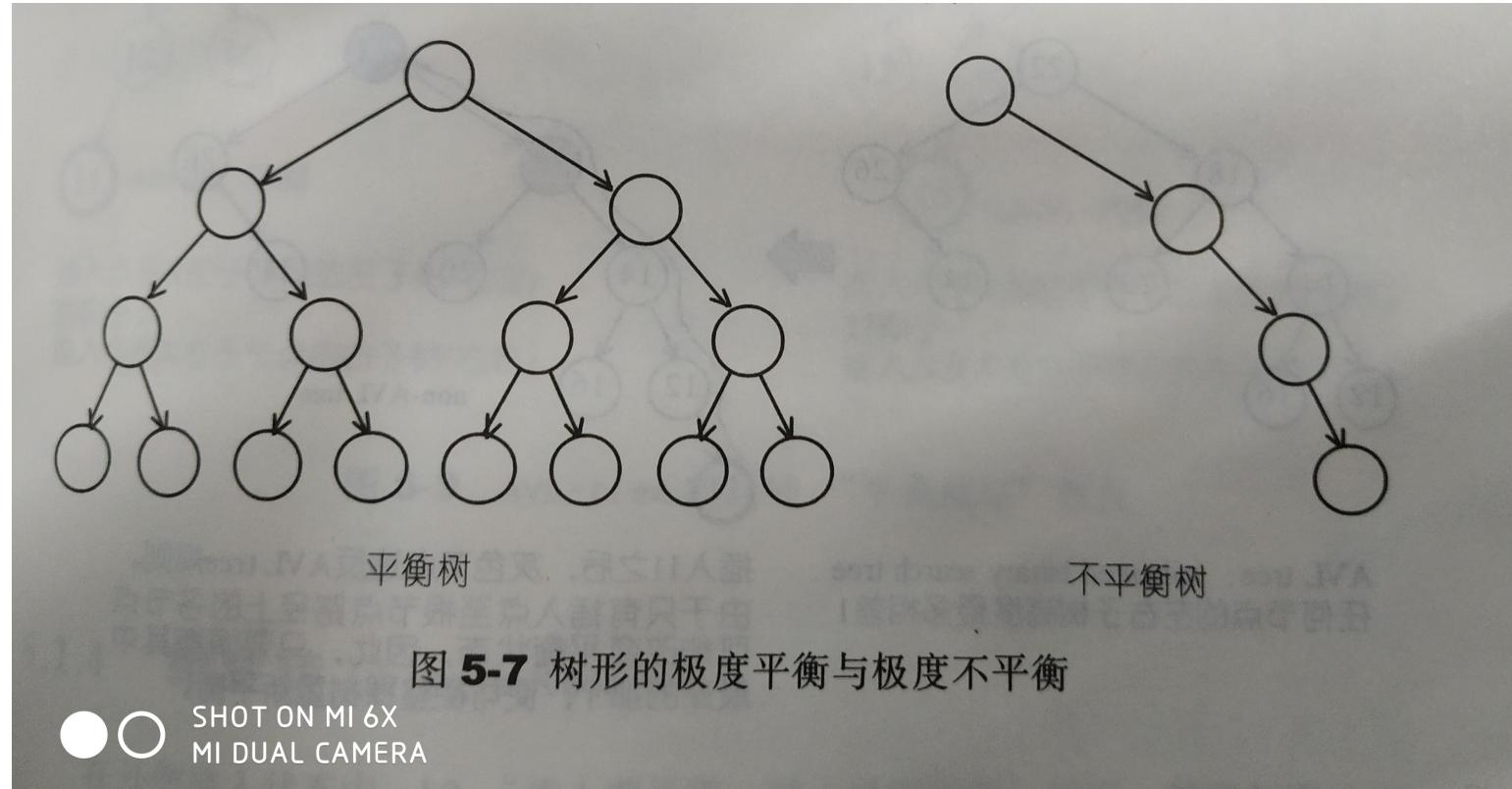
删除：



红黑树

平衡二叉搜索树：

可能会因为输入值不够随机，或某些插入删除操作，二叉搜索树可能会失去平衡，造成搜寻效率低落的情况



红黑树

为了避免出现不平衡树而影响查询效率，就需要一定条件约束树的结构。而RB-tree就是一种比较常见的平衡二叉树，它的约束条件如下：

为了方便，称一个节点到null所经过的节点和边的路径为路径

1. 每个节点不是红色就是黑色
2. 根节点为黑色
3. 如果节点为红，其子节点比为黑色
4. 任一节点到null的任何路径所含黑节点数相同

由第点可知，不存在两个红色节点连在一起的情况，多以，一条路径中红色节点数不超过整条路径的节点数的一半

红黑树

红黑树的平衡性证明：

设树高度为 h ，节点总数为 n ，根节点 x 的任意路径上的黑节点数为 $bh(x)$ ，对于一棵红黑树来说，只有两种情况：

1. 节点全是黑色

各个路径节点数相同，此时的红黑树是完全平衡树， h 、 n 、 $bh(x)$ 的关系为： $n \geq 2^{bh(x)} - 1$ ， $h = bh(x)$

2. 节点不全是黑色

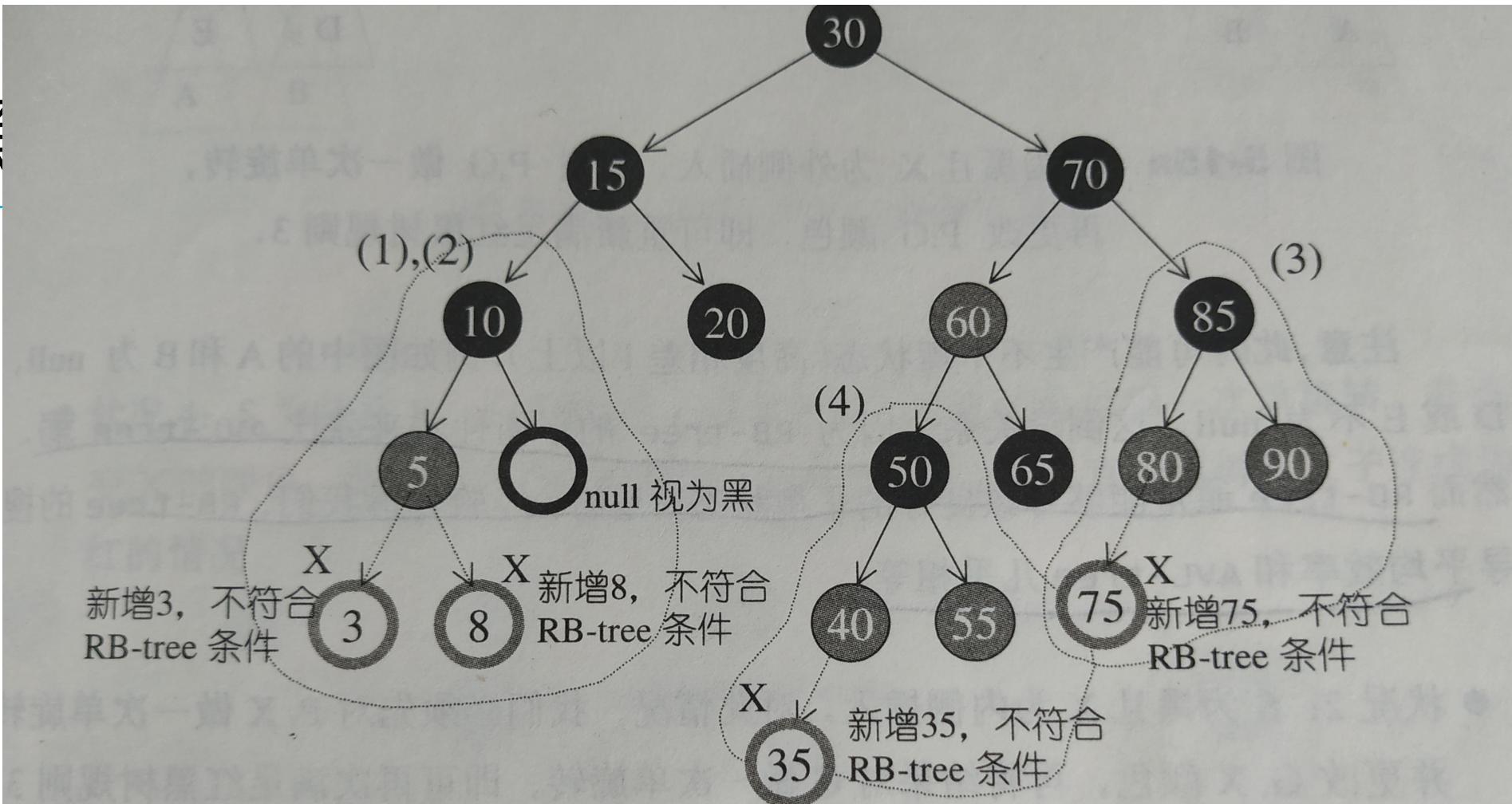
可以把这种情况视作第一种情况下的红黑树被添加了一些红色节点或者黑色节点被替换成红色节点。这种情况下， $n \geq 2^{bh(x)} - 1$ 即 $bh(x) \leq \log(n+1)$ 仍成立，而 $bh(x) \geq h/2$ ，所以 $h \leq 2\log(n+1)$

红黑树

红黑树满足规则的方法：

根据规则4，新增节点必为红，根据规则3，新增节点的父节点必须为黑。那么当新节点根据二叉树节点分布规则插入时，如果不能满足上诉条件时，就需要对节点颜色和数型进行调整。

红黑



5-14 为 RB-tree 插入四个新节点：3, 8, 35, 75。新增节点必为红色，暂以空心框表示。不论插入 3, 8, 35, 75 之中的哪一个节点，都会破坏 RB-tree 的规则，到我们必须旋转树形并调整节点的颜色。

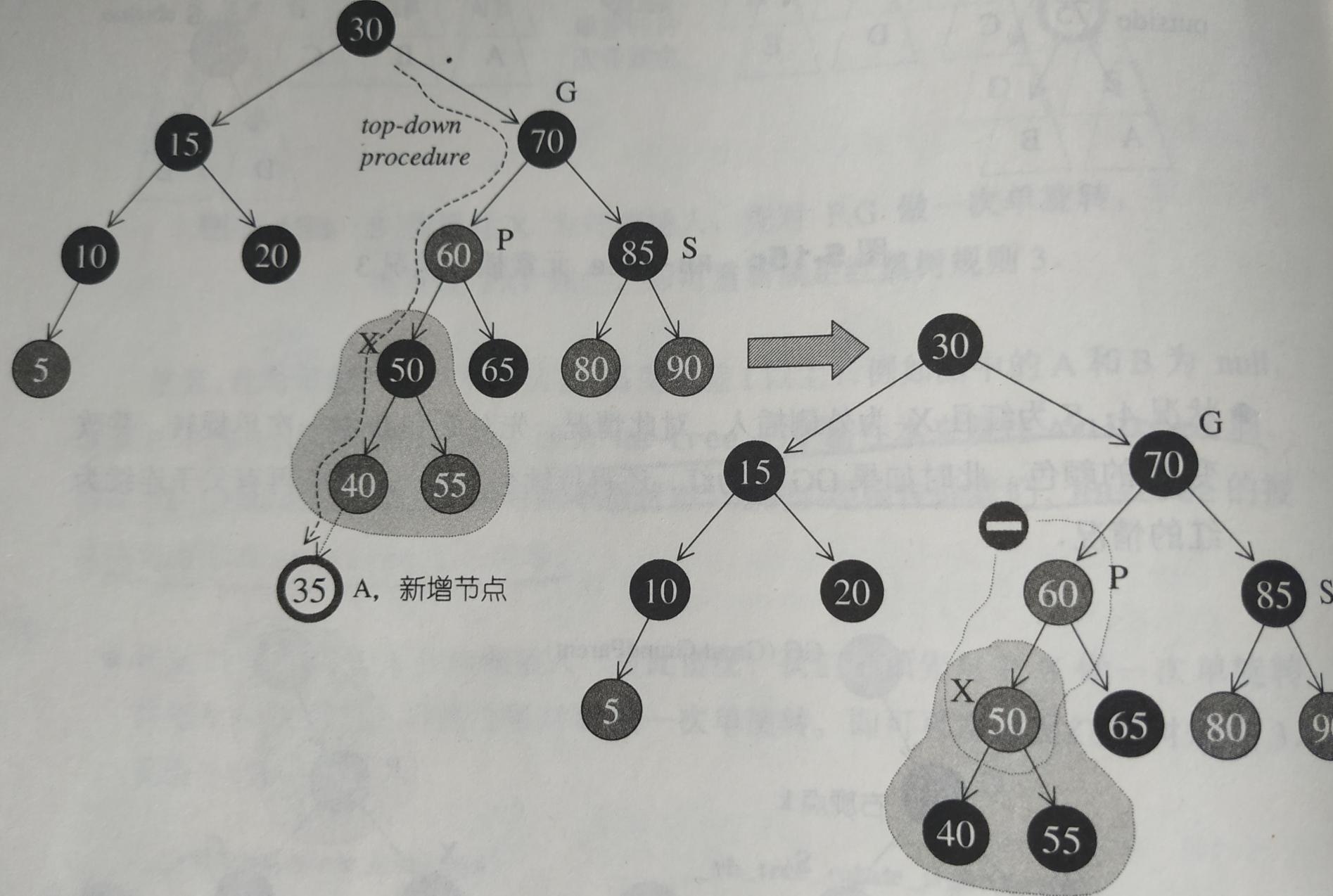


SHOT ON MI 6X
MI DUAL CAMERA

红黑树

红黑树满足规则的方法：

假设新增节点为A，那么就沿着A的路径，只要看到有某节点x的两个子节点皆为红色，就把x改为红色，并把两个子节点改为黑色。并假设x的父节点为P，P的兄弟节点为S，P的父节点为G。按照该方法改变节点颜色，并不会改变x节点任何路径中的黑节点个数，所以对整棵红黑树，任何节点的各路径中黑色节点数量依然相等。



SHOT ON MI 6X
MI DUAL CAMERA

图 5-15e

沿着 X 的路径，由上而下修正节点颜色。

红黑树

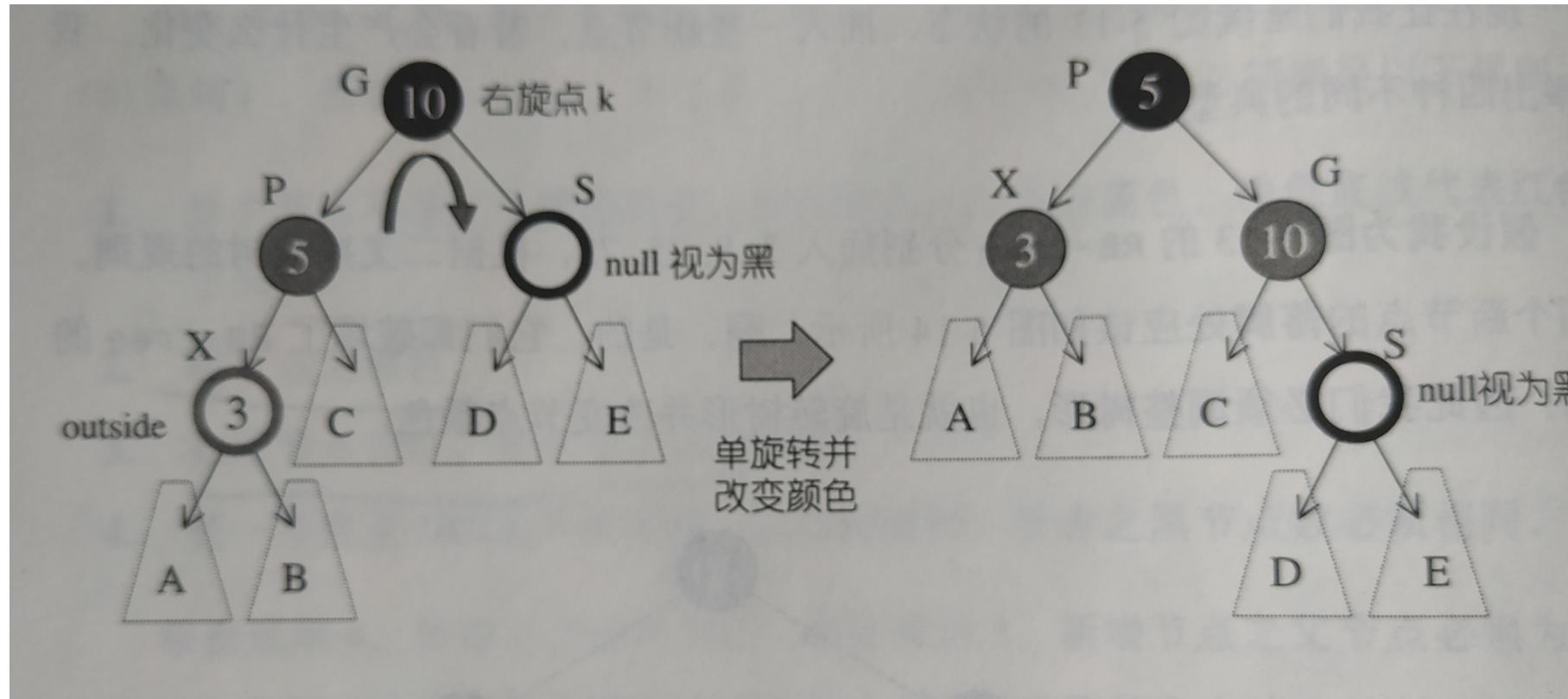
红黑树满足规则的方法：

但是该方法依然会出现矛盾，若此时x的父节点P亦为红色，就需要对树型进行调整。注意在这种情况下，因为P是红色节点，所以P的父节点G必为黑色，又因为该方法由上至下寻找两个子节点均为红色的节点x，所以G的两个子节点x、s不都为红色，所以s为黑色。

红黑树

红黑树出现矛盾的两种情况与其解决方法：

情况1.



红黑树

红黑树出现矛盾的两种情况与其解决方法：

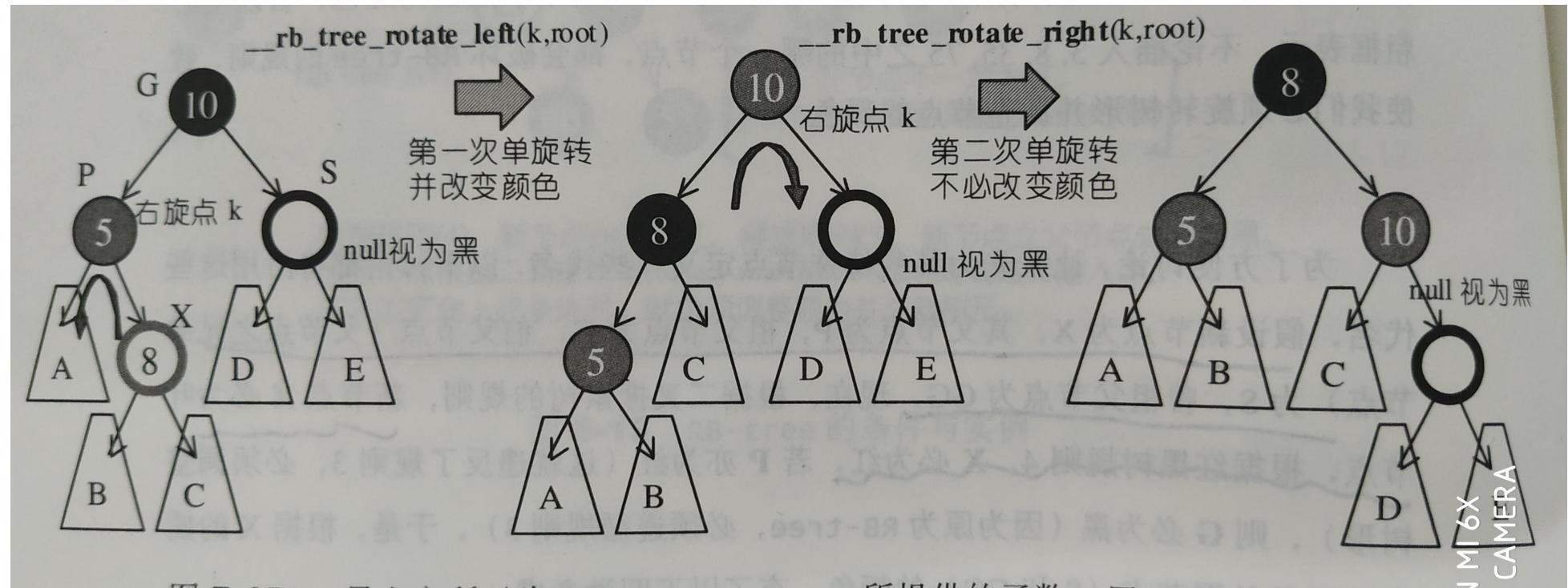
将P作为新的父节点，x与G作为P的子节点，P的原右节点作为G的左子节点，将新的父节点（即P）改为黑色，两个子节点（即x、G）改为红色。

假设以n节点为起始节点的任何路径中的黑色节点个数为bh(n).....
所以，旋转后仍然满足红黑树规则。

红黑树

红黑树出现矛盾的两种情况与其解决方法：

情况2.



红黑树总结

1. 内部数据已经排序，支持对数时间的查找、插入、删除操作。
2. 用四个规则保证了红黑树的平衡性，即保证 $h \leq 2\log(n+1)$
3. 用自上而下的方法，改变节点颜色并调整树型，来保证红黑树满足规则

map

map的特性是，所有元素都会根据元素的键值自动排序。map的所有元素都是pair，同时拥有实值（value）和键值（key）。Pair的第一元素被视为键值，第二元素被视为实值。Map不允许两个元素拥有相同的键值。

由于红黑树是一种平衡二叉树，自动排序效果很不错，所以map以红黑树为底层机制。

又因为红黑树以数据的键值来构造，所以map不允许修改键值，那样会关系到元素的排列规则。

unordered_map

与map有一个相对应的数据结构叫unordered_map，正如该数据结构的名称，
unordered_map不像map一样会根据键值排序，unordered_map的内部数据是无序的，
但是，unordered_map支持常数时间的查找，插入，删除。而做到这一点的就是
unordered_map的底层容器：hash_table。

Hash_table

Hash_table被视为一种字典结构，这种结构的用意就在于提供常数时间的基本操作。

Hash_table与数组非常相似，只不过数组的键值是一定范围内整数，而hash_table的键值可以是任何数据类型。

实际上，hash_table的实值数据就是用vector储存的，特别的是需要特定的函数（hash()函数）将特殊数据类型的key转化为一定范围内的整数，之后对vector的大小进行取模运算，运算结果就是实值数据在vector中的储存位置。

在STL中，只提供了整数、字符和字符串的hash函数，而其他数据类型，比如浮点数、自己定义的类，就需要自己定义hash函数。

Hash_table

可能出现的问题：

hash_table在插入数据时，可能有不同元素被映射到相同的位置。这就是所谓的碰撞问题。

解决碰撞的方法：

- 1.线性探测、二次线性探测
- 2.开链法

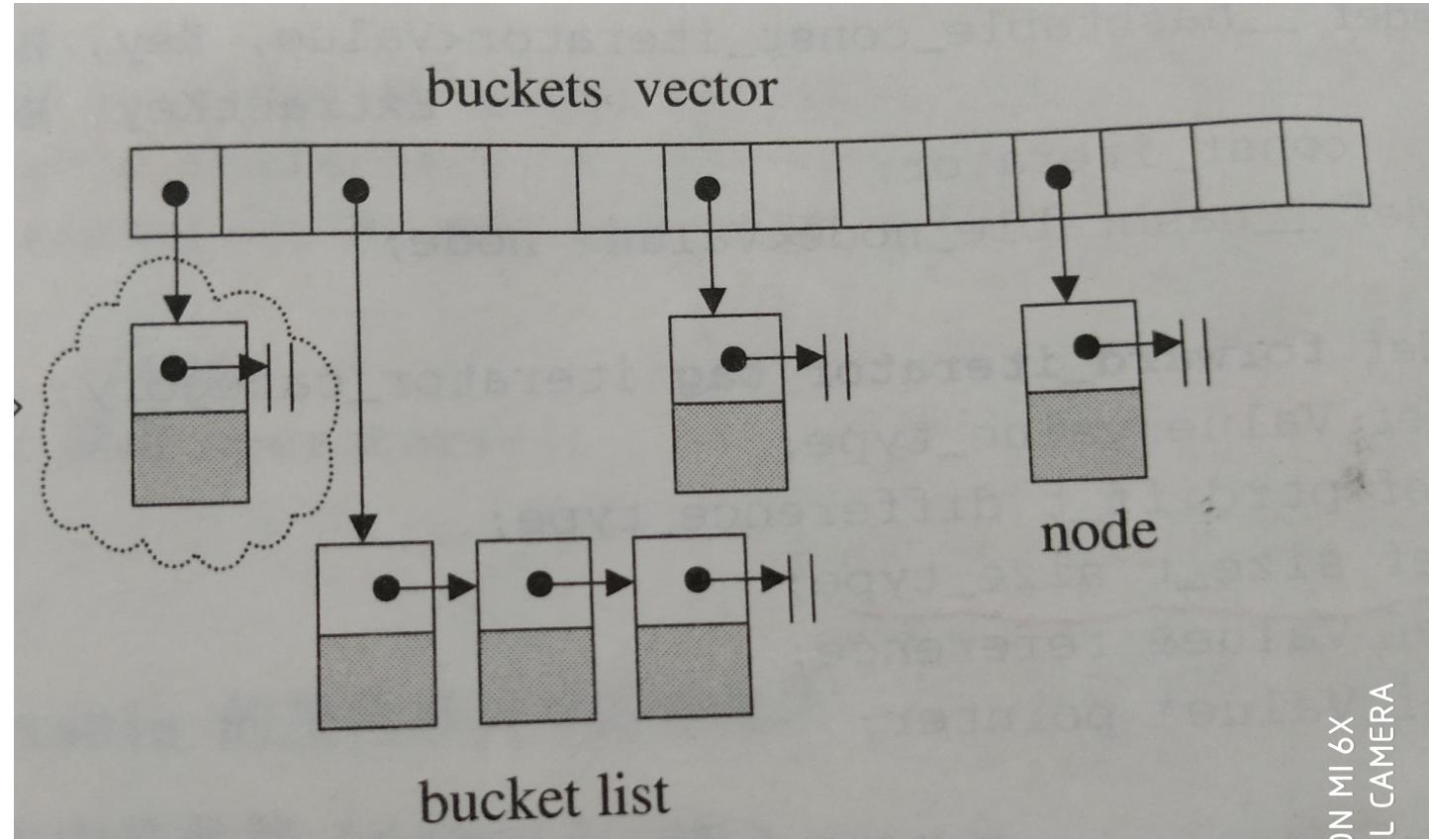
Hash_table

可能出现的问题：

hash_table在插入数据时，可能有不同元素被映射到相同的位置。这就是所谓的碰撞问题。

开链法

Hash_table的数据容器（ bucket vector ）中的元素为一个数据链表（叫做桶子），可以储存多个数据；这样以来，即使映射到相同的位置，也可以方便的储存。



开链法

在输入数据不够随机时，所有数据可能会全部挤在一个桶子里，这样极大影响了各项操作的性能。

因此hash_table在构建时，往往使用质数作为自己bucket vector的大小，这样在取模运算时就可以避免数据之间存在的关系。

并且，在插入数据时，hash_table会对比元素个数和bucket vector的大小，如果前者大于后者，就会扩充hash_table大小。

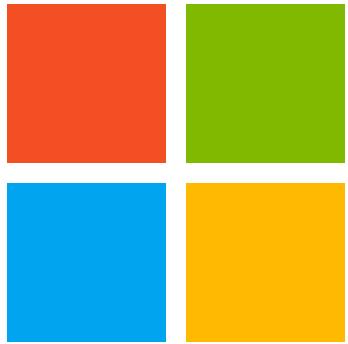
二者的差同

异：

1. Map使用红黑树作为底层容器，unordered_map使用hash_table作为底层容器。
2. Map的数据会自动排序，基本操作需要对数时间（可以用map的迭代器遍历，得到有序的数据）；unordered_map中数据是无序的，但是基本操作只要常数时间

同：

不能有键值相同的数据，都是字典型数据结构，不允许修改键值



Microsoft



加入TJMSC QQ群



关注TJMSC微信公众号