06.红黑树实现

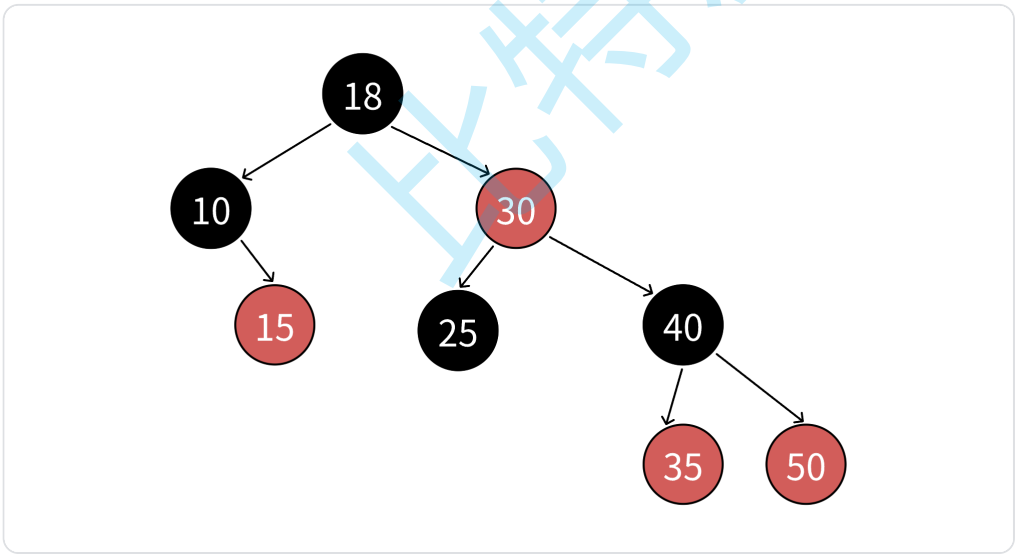
1. 红黑树的概念

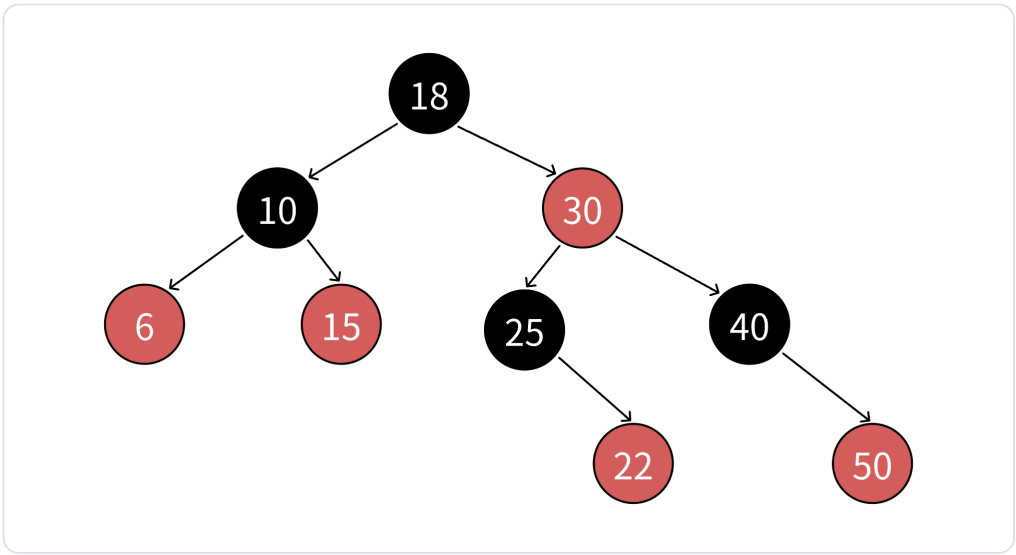
红黑树是一棵二叉搜索树，他的每个结点增加一个存储位来表示结点的颜色，可以是红色或者黑色。通过对任何一条从根到叶子的路径上各个结点的颜色进行约束，红黑树确保没有一条路径会比其他路径长出2倍，因而是接近平衡的。

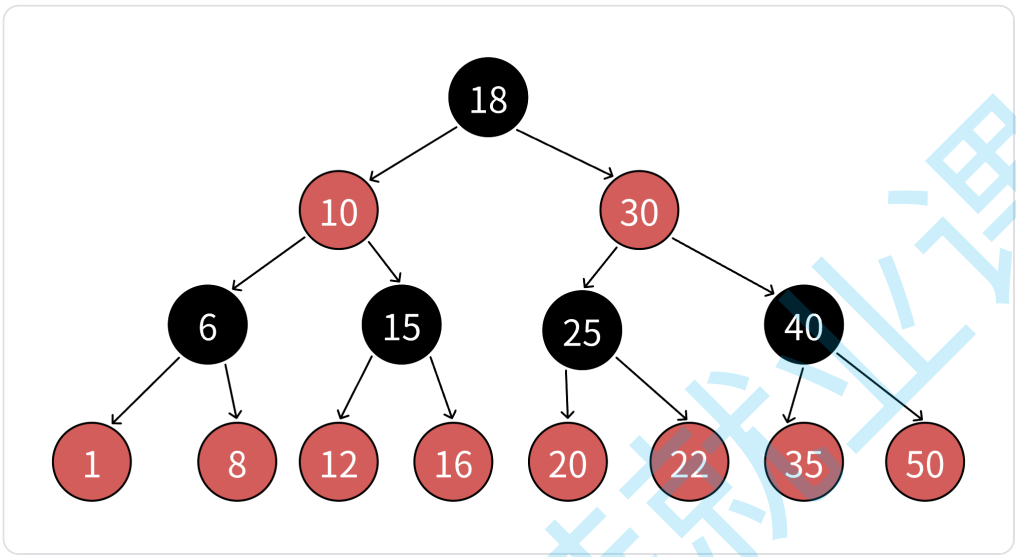
1.1 红黑树的规则：

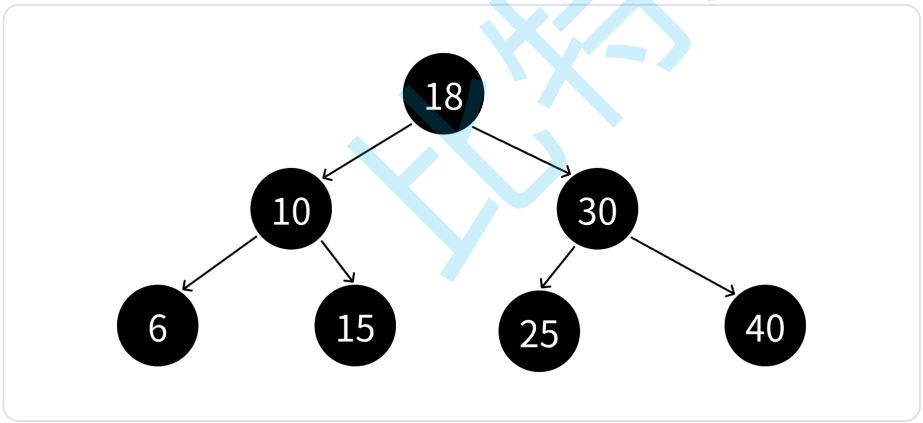
1. 每个结点不是红色就是黑色
2. 根结点是黑色的
3. 如果一个结点是红色的，则它的两个孩子结点必须是黑色的，也就是说任意一条路径不会有连续的红色结点。
4. 对于任意一个结点，从该结点到其所有NULL结点的简单路径上，均包含相同数量的黑色结点

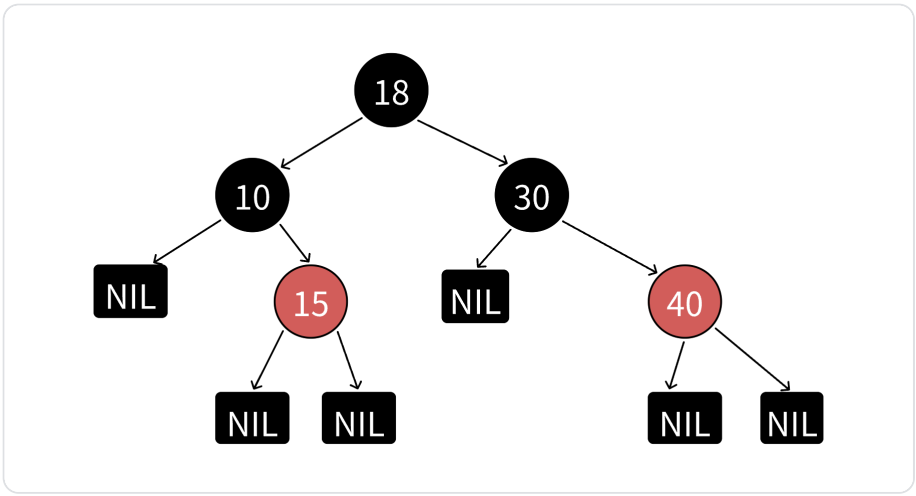
说明：《算法导论》等书籍上补充了一条每个叶子结点(NIL)都是黑色的规则。他这里所指的叶子结点不是传统的意义上的叶子结点，而是我们说的空结点，有些书籍上也把NIL叫做外部结点。NIL是为了方便准确的标识出所有路径，《算法导论》在后续讲解实现的细节中也忽略了NIL结点，所以我们知道一下这个概念即可。




18
10
30
6
15
25
40
22
50


18
10
30
6
15
25
40
1
8
12
16
20
22
35
50


18
10
30
6
15
25
40



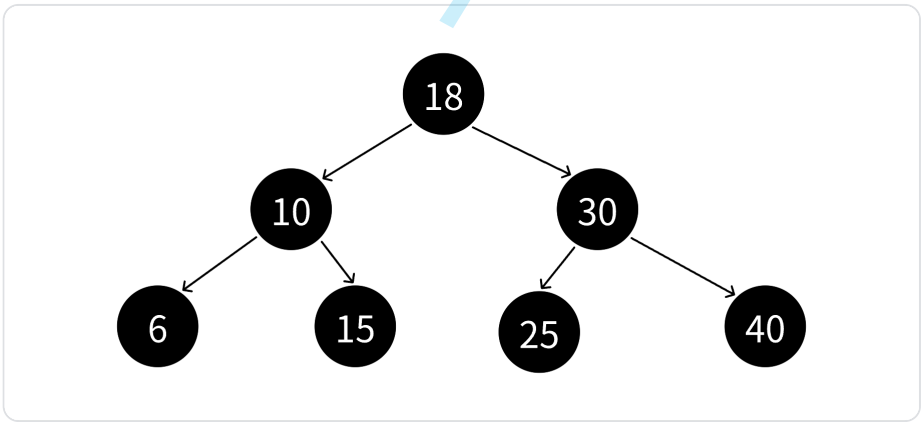
1.2 思考一下，红黑树如何确保最长路径不超过最短路径的2倍的?

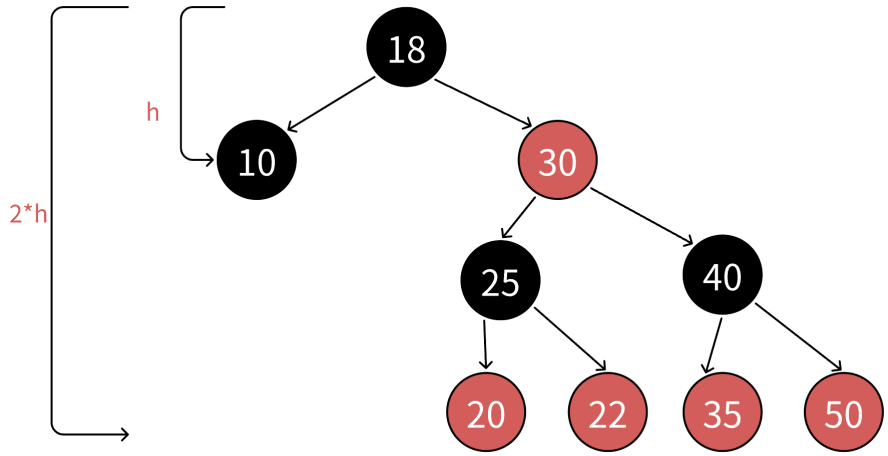
- 由规则4可知，从根到NULL结点的每条路径都有相同数量的黑色结点，所以极端场景下，最短路径就就是全是黑色结点的路径，假设最短路径长度为bh(black height)。
- 由规则2和规则3可知，任意一条路径不会有连续的红色结点，所以极端场景下，最长的路径就是一黑一红间隔组成，那么最长路径的长度为2*bh。
- 综合红黑树的4点规则而言，理论上的全黑最短路径和一黑一红的最长路径并不是在每棵红黑树都存在的。假设任意一条从根到NULL结点路径的长度为x，那么bh <= h <= 2*bh。

1.3 红黑树的效率：

假设N是红黑树树中结点数量，h最短路径的长度，那么 $2^h - 1 <= N < 2^{2*h} - 1$, 由此推出 $h \approx logN$ ，也就是意味着红黑树增删查改最坏也就是走最长路径 $2 * logN$ ，那么时间复杂度还是 $O(logN)$ 。

红黑树的表达相对AVL树要抽象一些，AVL树通过高度差直观的控制了平衡。红黑树通过4条规则的颜色约束，间接的实现了近似平衡，他们效率都是同一档次，但是相对而言，插入相同数量的结点，红黑树的旋转次数是更少的，因为他对平衡的控制没那么严格。





2. 红黑树的实现

2.1 红黑树的结构

```cpp
// 枚举值表示颜色
enum Colour
{
        RED,
        BLACK
};

// 这里我们默认按key/value结构实现
template<class K, class V>
struct RBTreeNode
{
        // 这里更新控制平衡也要加入parent指针
        pair<K, V> _kv;
        RBTreeNode<K, V>* _left;
        RBTreeNode<K, V>* _right;
        RBTreeNode<K, V>* _parent;
        Colour _col;

        RBTreeNode(const pair<K, V>& kv)
                :_kv(kv)
                , _left(nullptr)
                , _right(nullptr)
                , _parent(nullptr)
        {}
};

template<class K, class V>
class RBTree
```

```
{
        typedef RBTreeNode<K, V> Node;
public:
private:
        Node* _root = nullptr;
};
```

2.2 红黑树的插入

2.2.1 红黑树树插入一个值的大概过程

1. 插入一个值按二叉搜索树规则进行插入，插入后我们只需要观察是否符合红黑树的4条规则。
2. 如果是空树插入，新增结点是黑色结点。如果是非空树插入，新增结点必须红色结点，因为非空树插入，新增黑色结点就破坏了规则4，规则4是很难维护的。
3. 非空树插入后，新增结点必须红色结点，如果父亲结点是黑色的，则没有违反任何规则，插入结束
4. 非空树插入后，新增结点必须红色结点，如果父亲结点是红色的，则违反规则3。进一步分析，c是红色，p为红，g必为黑，这三个颜色都固定了，关键的变化看u的情况，需要根据u分为以下几种情况分别处理。

说明：下图中假设我们把新增结点标识为c (cur)，c的父亲标识为p(parent)，p的父亲标识为g(grandfather)，p的兄弟标识为u（uncle）。

2.2.2 情况1：变色

c为红，p为红，g为黑，u存在且为红，则将p和u变黑，g变红。在把g当做新的c，继续往上更新。

分析：因为p和u都是红色，g是黑色，把p和u变黑，左边子树路径各增加一个黑色结点，g再变红，相当于保持g所在子树的黑色结点的数量不变，同时解决了c和p连续红色结点的问题，需要继续往上更新是因为，g是红色，如果g的父亲还是红色，那么就还需要继续处理；如果g的父亲是黑色，则处理结束了；如果g就是整棵树的根，再把g变回黑色。

情况1只变色，不旋转。所以无论c是p的左还是右，p是g的左还是右，都是上面的变色处理方式。

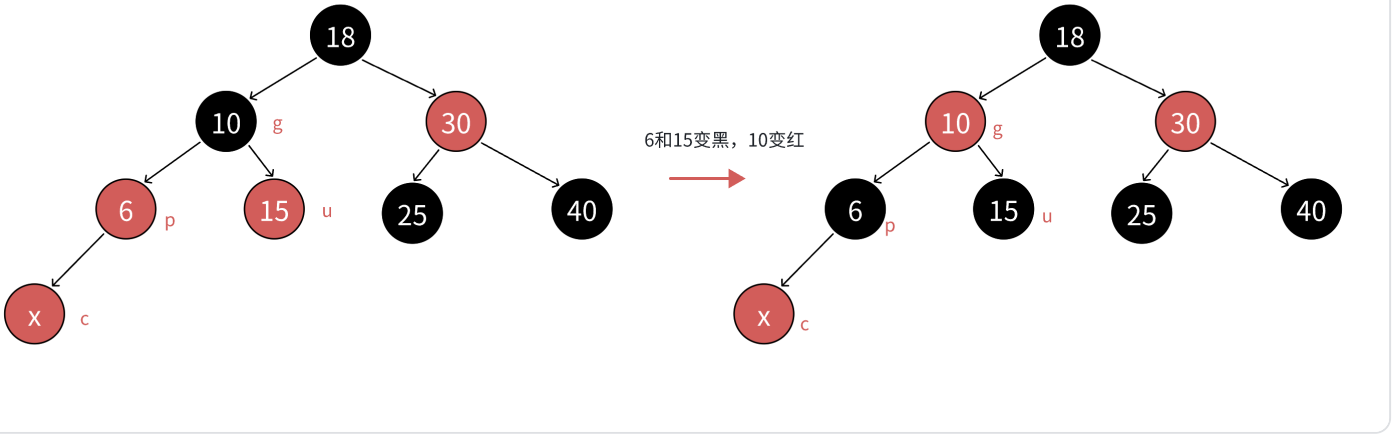


图0

- 跟AVL树类似，图0我们展示了一种具体情况，但是实际中需要这样处理的有很多种情况。
- 图1将以上类似的处理进行了抽象表达，d/e/f代表每条路径拥有hb个黑色结点的子树，a/b代表每条路径拥有hb-1个黑色结点的根为红的子树，hb>=0。
- 图2/图3/图4，分别展示了hb == 0/hb == 1/hb == 2的具体情况组合分析，当hb等于2时，这里组合情况上百亿种，这些样例是帮助我们理解，不论情况多少种，多么复杂，处理方式一样的，变色再继续往上处理即可，所以我们只需要看抽象图即可。

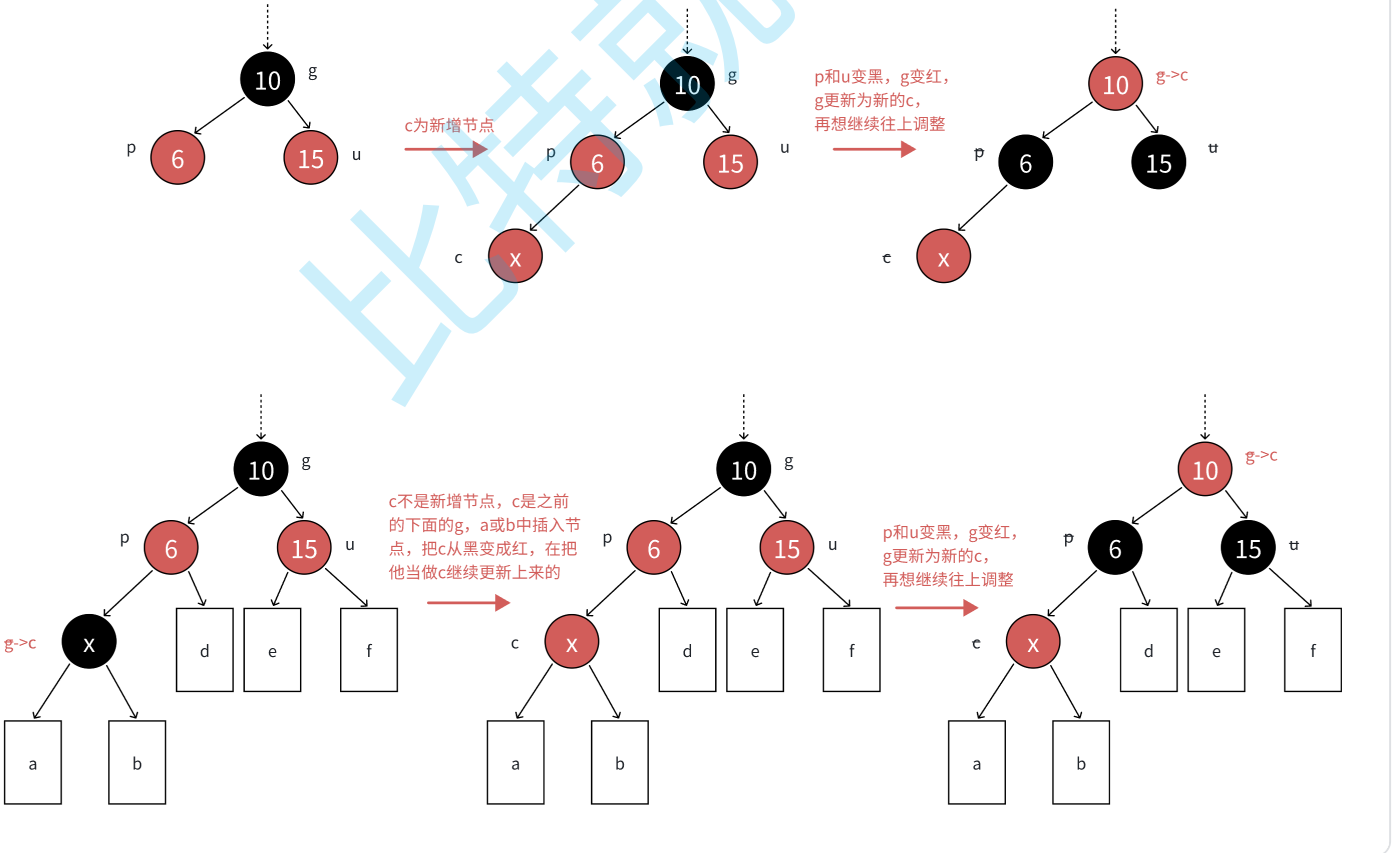


图1

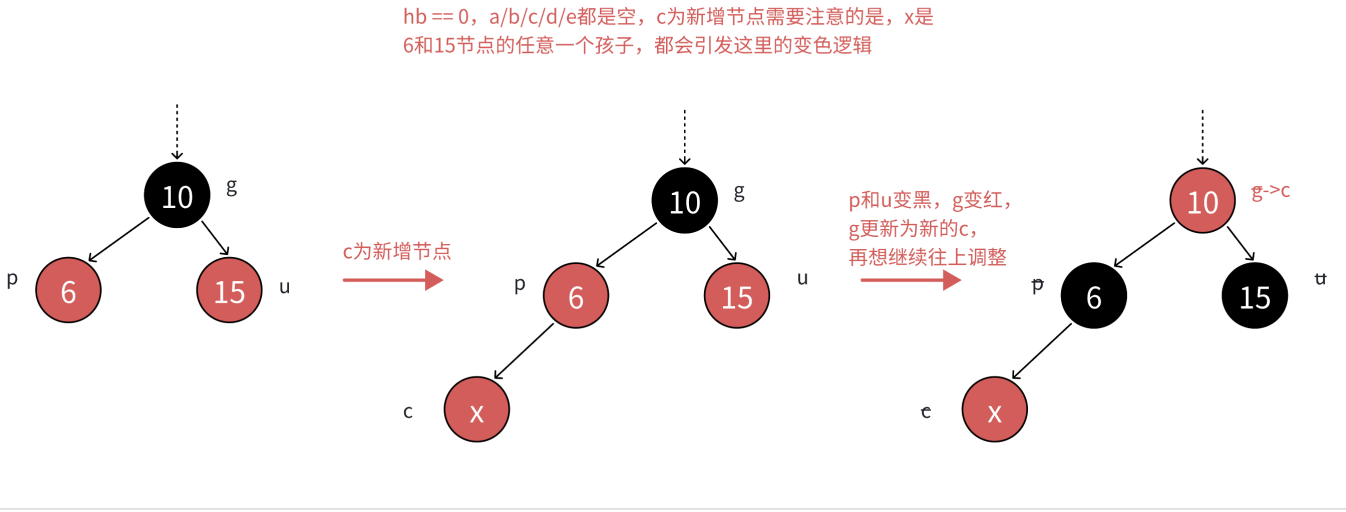

hb == 0，a/b/c/d/e都是空，c为新增节点需要注意的是，x是
6和15节点的任意一个孩子，都会引发这里的变色逻辑
10
g
p
6
15
u
c为新增节点
c
x
p和u变黑，g变红，
g更新为新的c，
再想继续往上调整
g->c

图2

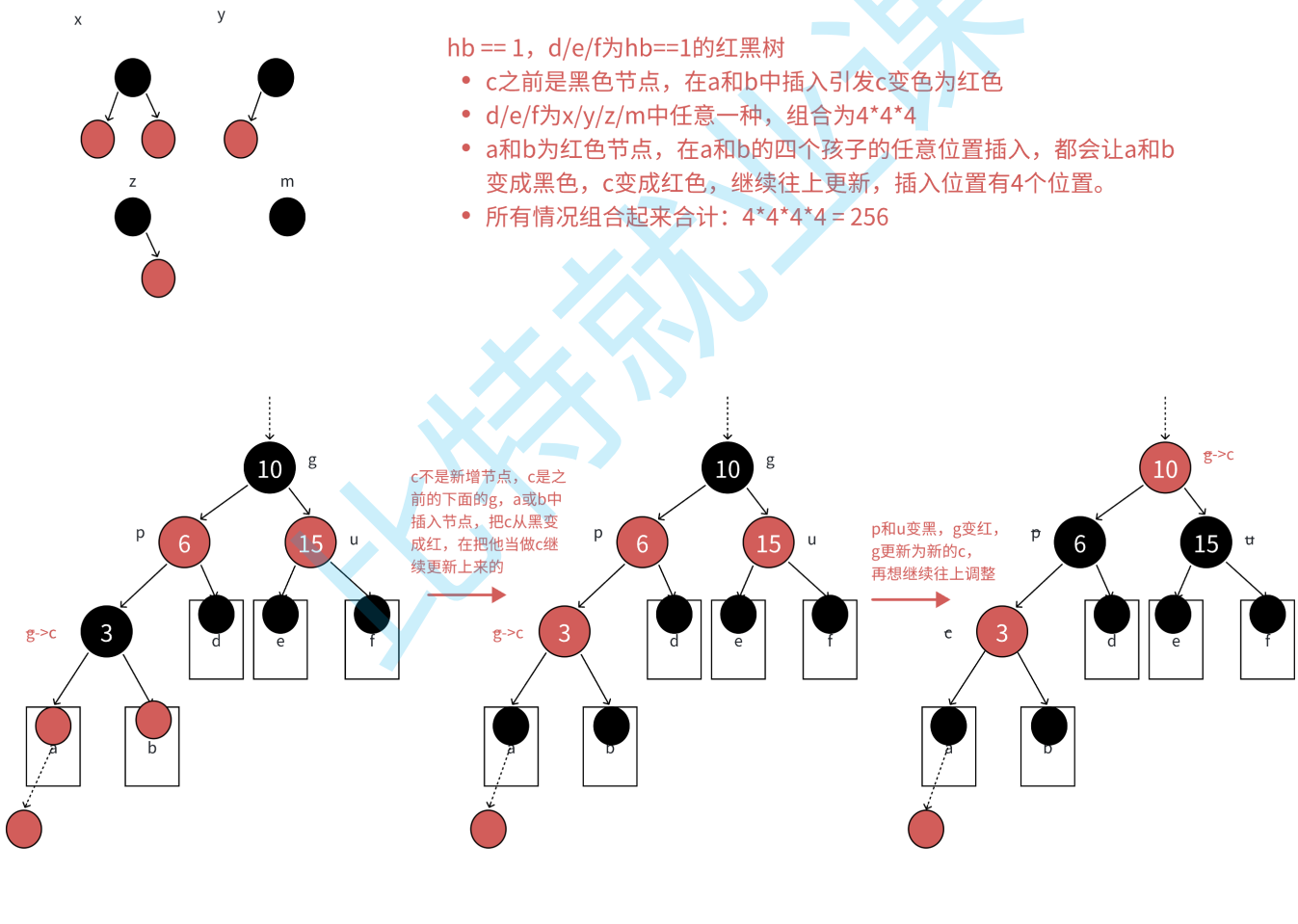

x
y
z
m
hb == 1，d/e/f为hb==1的红黑树
• c之前是黑色节点，在a和b中插入引发c变色为红色
• d/e/f为x/y/z/m中任意一种，组合为4*4*4
• a和b为红色节点，在a和b的四个孩子的任意位置插入，都会让a和b变成黑色，c变成红色，继续往上更新，插入位置有4个位置。
• 所有情况组合起来合计：4*4*4*4 = 256
10
g
p
6
15
u
g->c
3
d
e
f
a
b
c不是新增节点，c是之前的下面的g，a或b中插入节点，把c从黑变成红，在把他当做c继续更新上来的
p和u变黑，g变红，
g更新为新的c，
再想继续往上调整

图3

hb == 2，d/e/f为hb==2的红黑树，a和b是hb==1的根为红色的树

- d/e/f的组合为：(256+16)*(256+16)*(256+16) = 20123648
- a和b为根节点为红色节点的hb==1的树，这里可以看到a和b插入组合也不少
- a或者b插入至少要经历两次变色和向上处理才能得到这里的情况，这里的组合情况至少是百亿以上了。

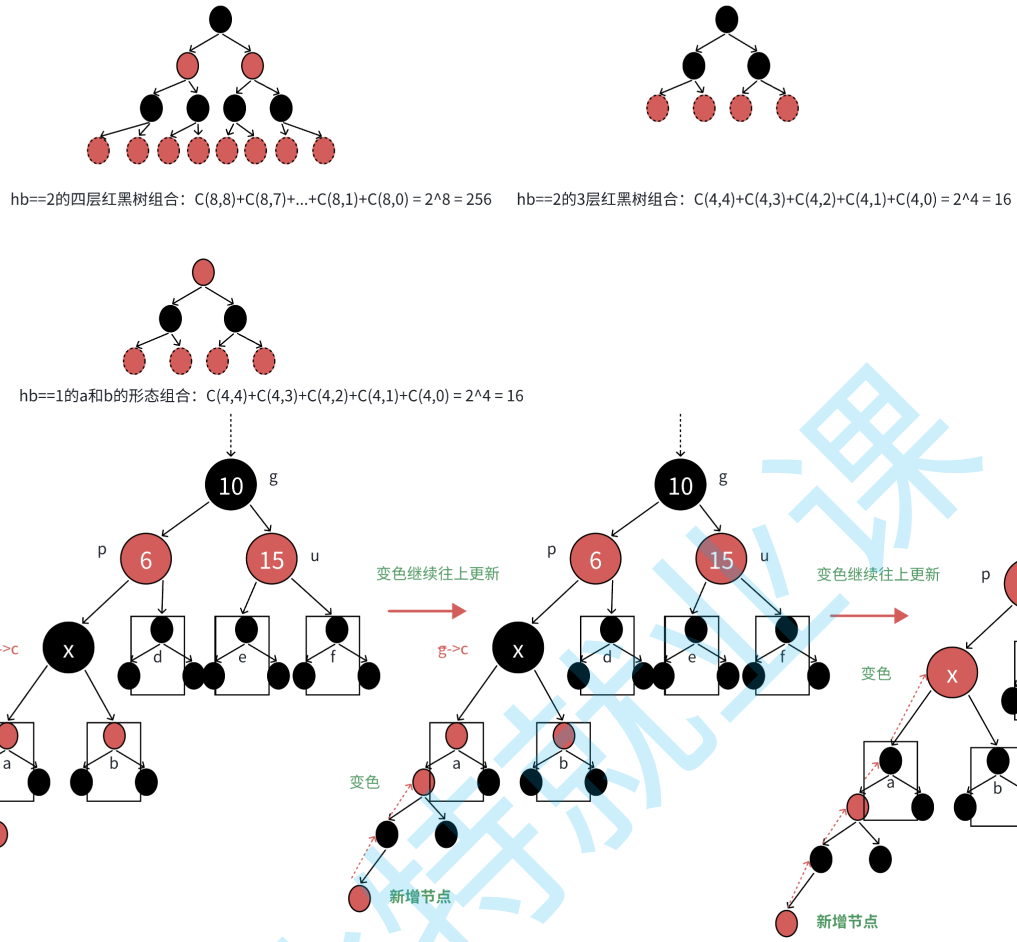


图4

2.2.3 情况2：单旋+变色

c为红，p为红，g为黑，u不存在或者u存在且为黑，u不存在，则c一定是新增结点，u存在且为黑，则c一定不是新增，c之前是黑色的，是在c的子树中插入，符合情况1，变色将c从黑色变成红色，更新上来的。

分析：p必须变黑，才能解决，连续红色结点的问题，u不存在或者是黑色的，这里单纯的变色无法解决问题，需要旋转+变色。

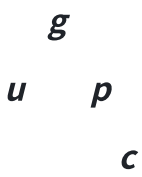
g

p ***u***

c

如果p是g的左，c是p的左，那么以g为旋转点进行右单旋，再把p变黑，g变红即可。p变成课这颗树新的根，这样子树黑色结点的数量不变，没有连续的红色结点了，且不需要往上更新，因为p的父亲是黑

色还是红色或者空都不违反规则。



如果p是g的右，c是p的右，那么以g为旋转点进行左单旋，再把p变黑，g变红即可。p变成课这颗树新的根，这样子树黑色结点的数量不变，没有连续的红色结点了，且不需要往上更新，因为p的父亲是黑色还是红色或者空都不违反规则。

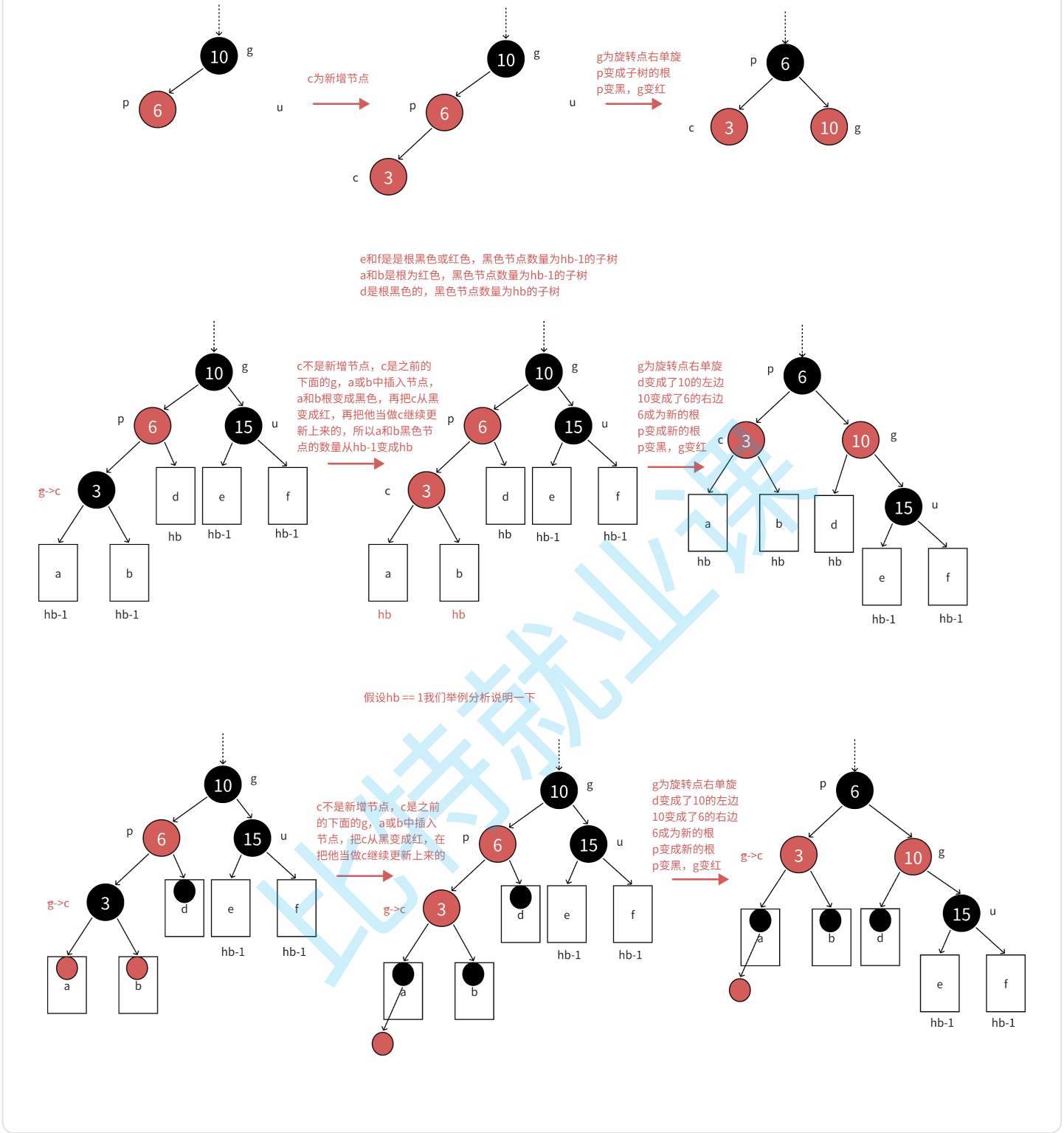


图1

2.2.4 情况2：双旋+变色

c为红，p为红，g为黑，u不存在或者u存在且为黑，u不存在，则c一定是新增结点，u存在且为黑，则c一定不是新增，c之前是黑色的，是在c的子树中插入，符合情况1，变色将c从黑色变成红色，更新上来的。

分析：p必须变黑，才能解决，连续红色结点的问题，u不存在或者是黑色的，这里单纯的变色无法解决问题，需要旋转+变色。

如果p是g的左，c是p的右，那么先以p为旋转点进行左单旋，再以g为旋转点进行右单旋，再把c变黑，g变红即可。c变成课这颗树新的根，这样子树黑色结点的数量不变，没有连续的红色结点了，且不需要往上更新，因为c的父亲是黑色还是红色或者空都不违反规则。

如果p是g的右，c是p的左，那么先以p为旋转点进行右单旋，再以g为旋转点进行左单旋，再把c变黑，g变红即可。c变成课这颗树新的根，这样子树黑色结点的数量不变，没有连续的红色结点了，且不需要往上更新，因为c的父亲是黑色还是红色或者空都不违反规则。

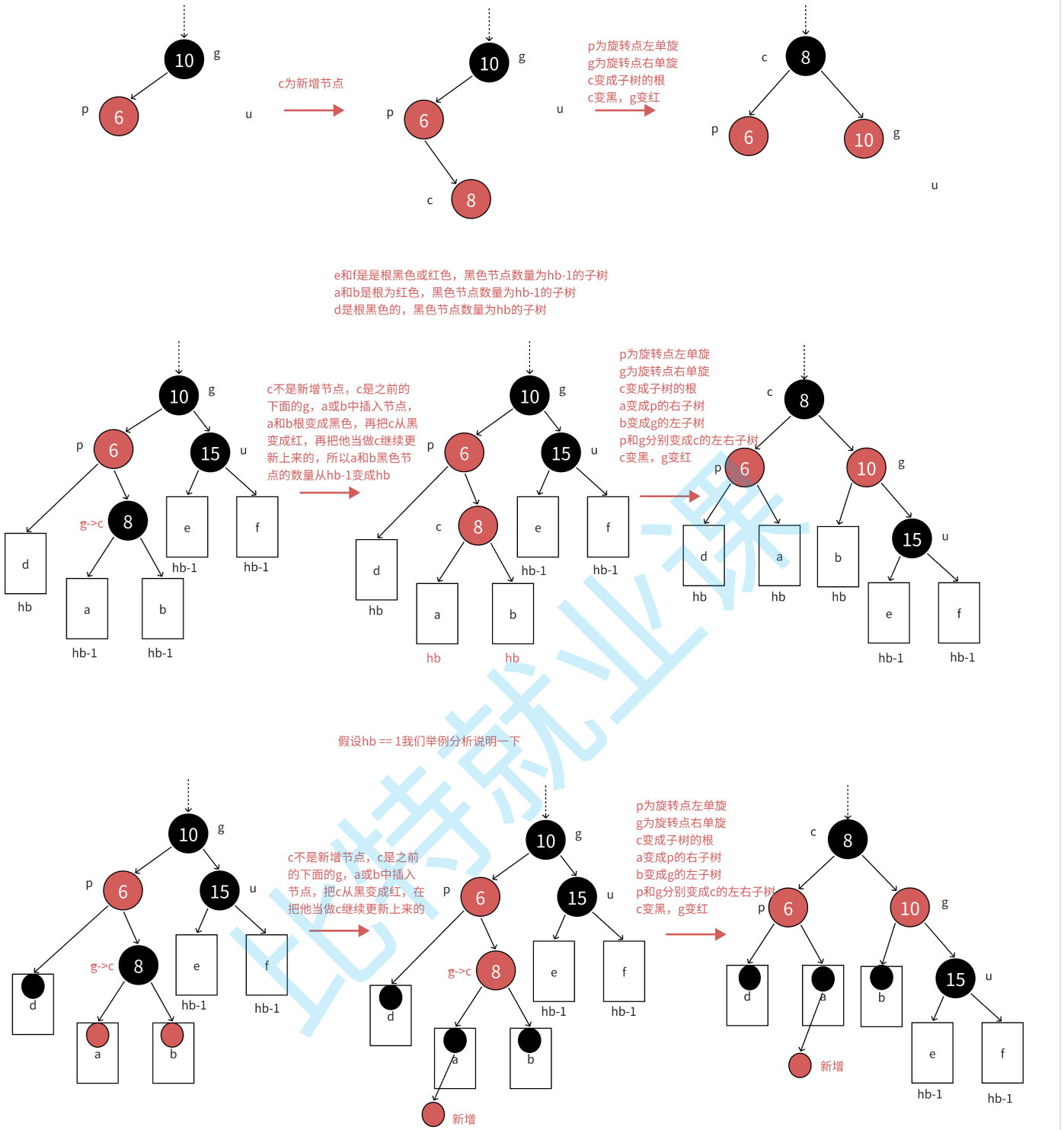


图1

2.3 红黑树的插入代码实现

```
// 旋转代码的实现跟AVL树是一样的，只是不需要更新平衡因子
bool Insert(const pair<K, V>& kv)
{
    if (_root == nullptr)
    {
```

```
6           _root = new Node(kv);
7           _root->_col = BLACK;
8           return true;
9       }
10
11      Node* parent = nullptr;
12      Node* cur = _root;
13      while (cur)
14      {
15          if (cur->_kv.first < kv.first)
16          {
17                  parent = cur;
18                  cur = cur->_right;
19          }
20          else if (cur->_kv.first > kv.first)
21          {
22                  parent = cur;
23                  cur = cur->_left;
24          }
25          else
26          {
27                  return false;
28          }
29      }
30
31      cur = new Node(kv);
32      // 新增结点。颜色红色给红色
33      cur->_col = RED;
34      if (parent->_kv.first < kv.first)
35      {
36              parent->_right = cur;
37      }
38      else
39      {
40              parent->_left = cur;
41      }
42      cur->_parent = parent;
43
44      while (parent && parent->_col == RED)
45      {
46          Node* grandfather = parent->_parent;
47          //    g
48          //  p   u
49          if (parent == grandfather->_left)
50          {
51              Node* uncle = grandfather->_right;
52              if (uncle && uncle->_col == RED)
```

```
53              {
54                  // u存在且为红 -》变色再继续往上处理
55                  parent->_col = uncle->_col = BLACK;
56                  grandfather->_col = RED;
57
58                  cur = grandfather;
59                  parent = cur->_parent;
60              }
61              else
62              {
63                  // u存在且为黑或不存在 -》旋转+变色
64                  if (cur == parent->_left)
65                  {
66                      //    g
67                      //  p   u
68                      //c
69                      //单旋
70                      RotateR(grandfather);
71                      parent->_col = BLACK;
72                      grandfather->_col = RED;
73                  }
74                  else
75                  {
76                      //    g
77                      //  p   u
78                      //    c
79                      //双旋
80                      RotateL(parent);
81                      RotateR(grandfather);
82
83                      cur->_col = BLACK;
84                      grandfather->_col = RED;
85                  }
86
87                  break;
88              }
89          }
90          else
91          {
92              //    g
93              //  u   p
94              Node* uncle = grandfather->_left;
95              // 叔叔存在且为红，-》变色即可
96              if (uncle && uncle->_col == RED)
97              {
98                  parent->_col = uncle->_col = BLACK;
99                  grandfather->_col = RED;
```

```
100
101                 // 继续往上处理
102                 cur = grandfather;
103                 parent = cur->_parent;
104             }
105             else // 叔叔不存在，或者存在且为黑
106             {
107                 // 情况二：叔叔不存在或者存在且为黑
108                 // 旋转+变色
109                 //      g
110                 //   u     p
111                 //             c
112                 if (cur == parent->_right)
113                 {
114                     RotateL(grandfather);
115                     parent->_col = BLACK;
116                     grandfather->_col = RED;
117                 }
118                 else
119                 {
120                     //      g
121                     //   u     p
122                     //      c
123                     RotateR(parent);
124                     RotateL(grandfather);
125                     cur->_col = BLACK;
126                     grandfather->_col = RED;
127                 }
128                 break;
129             }
130         }
131     }
132
133     _root->_col = BLACK;
134
135     return true;
136 }
```

2.4 红黑树的查找

按二叉搜索树逻辑实现即可，搜索效率为 $O(logN)$

```
1 Node* Find(const K& key)
2 {
3         Node* cur = _root;
```

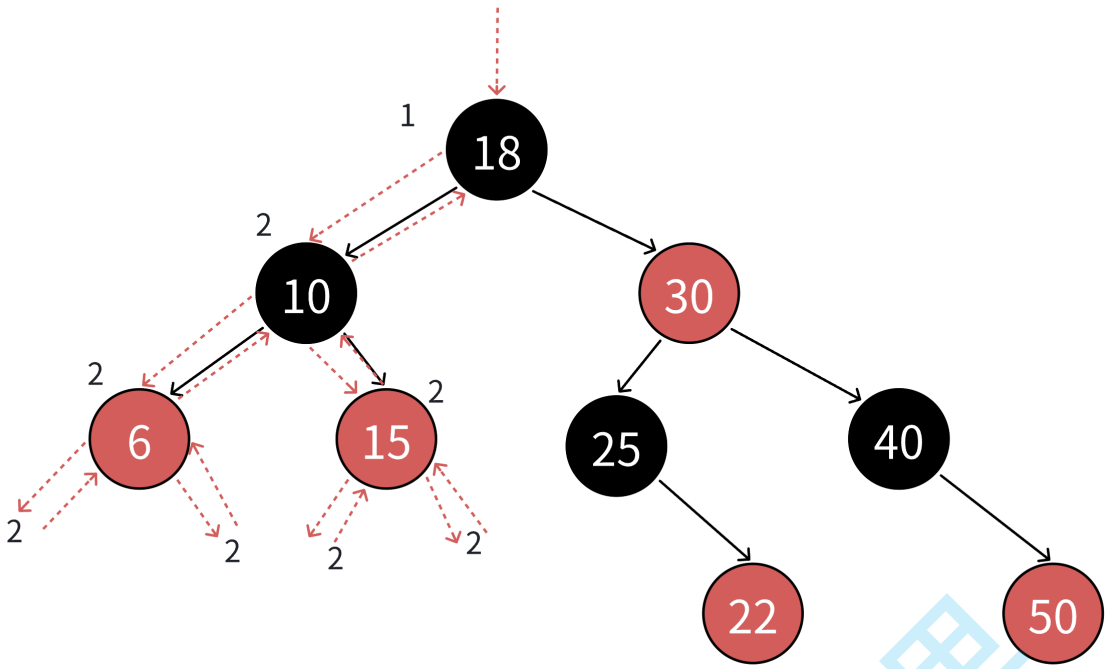
```
        while (cur)
        {
            if (cur->_kv.first < key)
            {
                    cur = cur->_right;
            }
            else if (cur->_kv.first > key)
            {
                    cur = cur->_left;
            }
            else
            {
                    return cur;
            }
        }

        return nullptr;
}
```

2.5 红黑树的验证

这里获取最长路径和最短路径，检查最长路径不超过最短路径的2倍是不可行的，因为就算满足这个条件，红黑树也可能颜色不满足规则，当前暂时没出问题，后续继续插入还是会出问题的。所以我们还是去检查4点规则，满足这4点规则，一定能保证最长路径不超过最短路径的2倍。

1. 规则1枚举颜色类型，天然实现保证了颜色不是黑色就是红色。
2. 规则2直接检查根即可
3. 规则3前序遍历检查，遇到红色结点查孩子不太方便，因为孩子有两个，且不一定存在，反过来检查父亲的颜色就方便多了。
4. 规则4前序遍历，遍历过程中用形参记录跟到当前结点的blackNum(黑色结点数量)，前序遍历遇到黑色结点就++blackNum，走到空就计算出了一条路径的黑色结点数量。再任意一条路径黑色结点数量作为参考值，依次比较即可。




```cpp
bool Check(Node* root, int blackNum, const int refNum)
{
    if (root == nullptr)
    {
            // 前序遍历走到空时，意味着一条路径走完了
            //cout << blackNum << endl;
            if (refNum != blackNum)
            {
                cout << "存在黑色结点的数量不相等的路径" << endl;
                return false;
            }

            return true;
    }

    // 检查孩子不太方便，因为孩子有两个，且不一定存在，反过来检查父亲就方便多了
    if (root->_col == RED && root->_parent->_col == RED)
    {
            cout << root->_kv.first << "存在连续的红色结点" << endl;
            return false;
    }

    if (root->_col == BLACK)
    {
            blackNum++;
    }

```

```
28      return Check(root->_left, blackNum, refNum)
29              && Check(root->_right, blackNum, refNum);
30  }
31
32  bool IsBalance()
33  {
34      if (_root == nullptr)
35              return true;
36
37      if (_root->_col == RED)
38              return false;
39
40      // 参考值
41      int refNum = 0;
42      Node* cur = _root;
43      while (cur)
44      {
45              if (cur->_col == BLACK)
46              {
47                      ++refNum;
48              }
49
50              cur = cur->_left;
51      }
52
53      return Check(_root, 0, refNum);
54  }
```

2.6 红黑树的删除

红黑树的删除本章节不做讲解，有兴趣的同学可参考：《算法导论》或者《STL源码剖析》中讲解。