08-C3

高级搜索树

红黑树：插入

莫赤匪狐，莫黑匪乌；惠而好我，携手同车

有理走遍天下，无理寸步难行

邓俊辉

deng@tsinghua.edu.cn

双红

❖ 按BST规则插入关键码e //x = insert(e)必为叶节点

❖ 除非是首个节点（根），x的父亲p = x->parent必存在

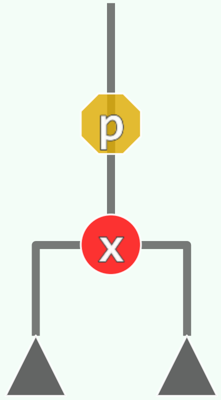
首先将x染红 //x->color = isRoot(x) ? B : R

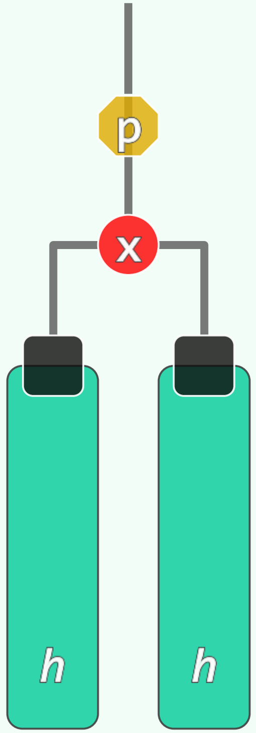
❖ 至此，条件1、2、4依然满足；

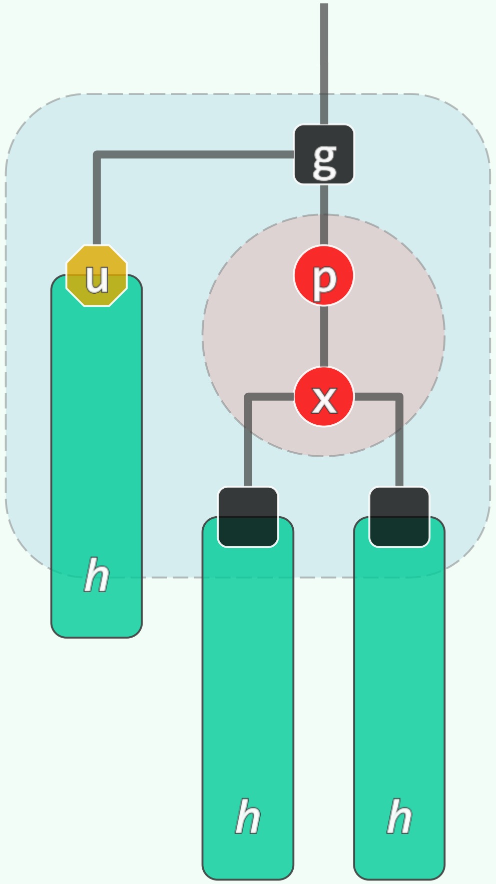
但3不见得，有可能...

❖ 双红/double-red

//p->color == x->color == R







❖ 考查：祖父g = p->parent //必存在，且必黑

叔父u = uncle(x) = sibling(p)

❖ 视u的颜色无非两种情况，分别加以处理...

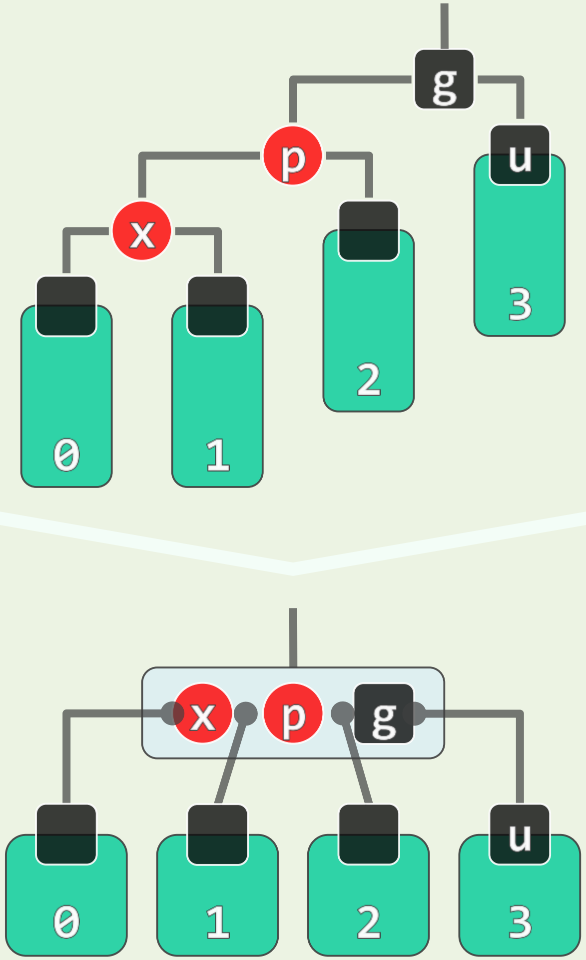
插入算法

```
template <typename T> BinNodePosi<T> RedBlack<T>::insert( const T & e ) {
   //确认目标节点不存在（留意对_hot的设置）
   BinNodePosi<T> & x = search( e ); if ( x ) return x;
   //创建红节点x，以_hot为父，黑高度 = 0
   x = new BinNode<T>( e, _hot, NULL, NULL, 0 ); _size++;
   //如有必要，需做双红修正，再返回插入的节点
   BinNodePosi<T> xOld = x; solveDoubleRed( x ); return xOld;
} //无论原树中是否存有e，返回时总有x->data == e
```

双红修正

```
template <typename T> void RedBlack<T>::solveDoubleRed( BinNodePosi<T> x ) {
   while ( 1 ) {
      if ( IsRoot( x ) ) { x->color = RB_BLACK; x->height++; return; } //调整至根
      BinNodePosi<T> p = x->parent; if ( IsBlack( p ) ) return; //x之父p为黑
      BinNodePosi<T> g = p->parent; //否则，x之祖父g必存在且黑
      BinNodePosi<T> u = uncle( x ); //以下，视x之叔父u的颜色分别处理
      if ( IsBlack( u ) ) { /* ... u为黑（或NULL） ... */ }
      else                { /* ... u为红 ... */ }
   } //while
} //solveDoubleRed
```

RR-1: u->color == B



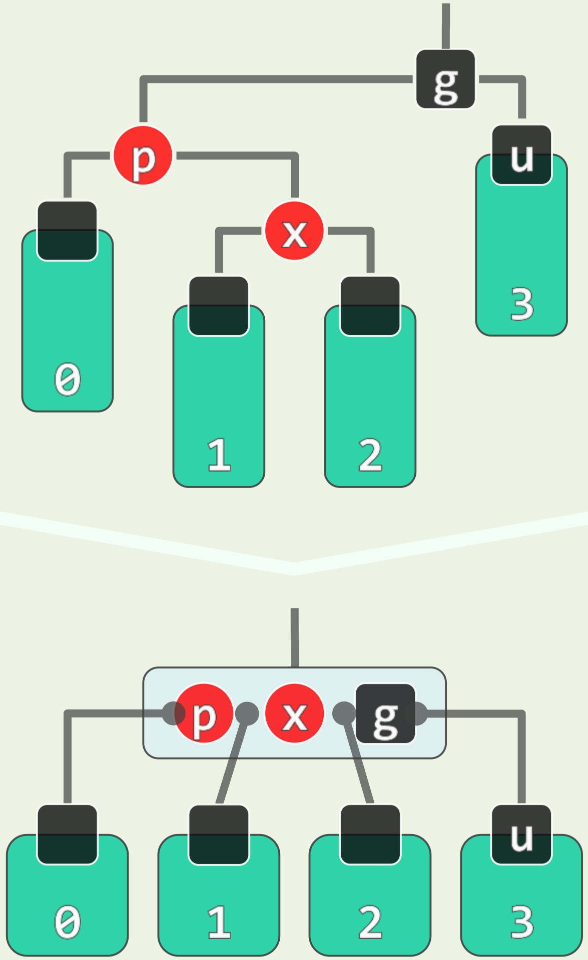
❖ 此时，x、p、g的四个孩子

(可能是外部节点)

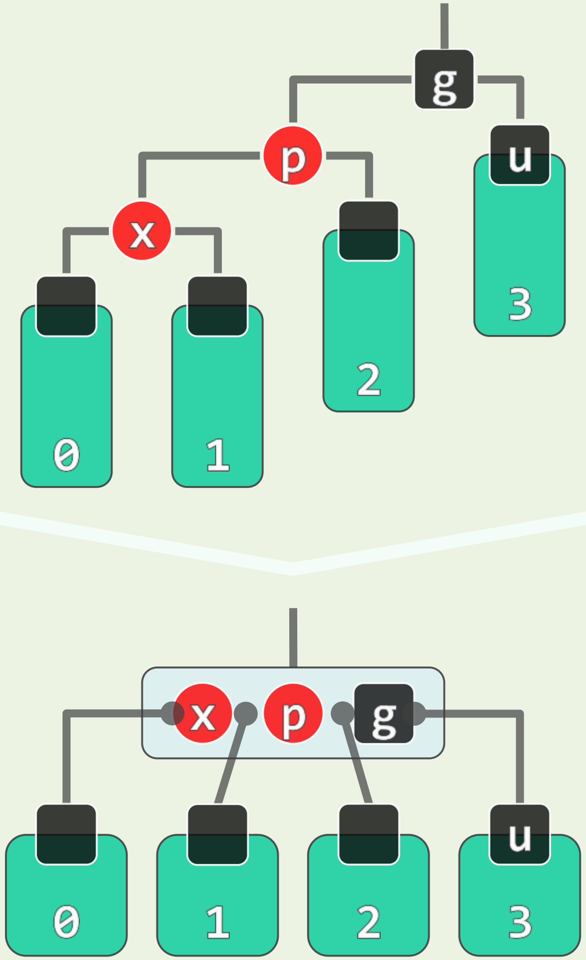
- 全为黑，且

- 黑高度相同

❖ 另两种对称情况，自行补充



RR-1: u->color == B



- ❖ 局部“3+4”重构

  b转黑，a或c转红

- ❖ 从B-树的角度，如何理解？

  所谓“非法”，无非是...

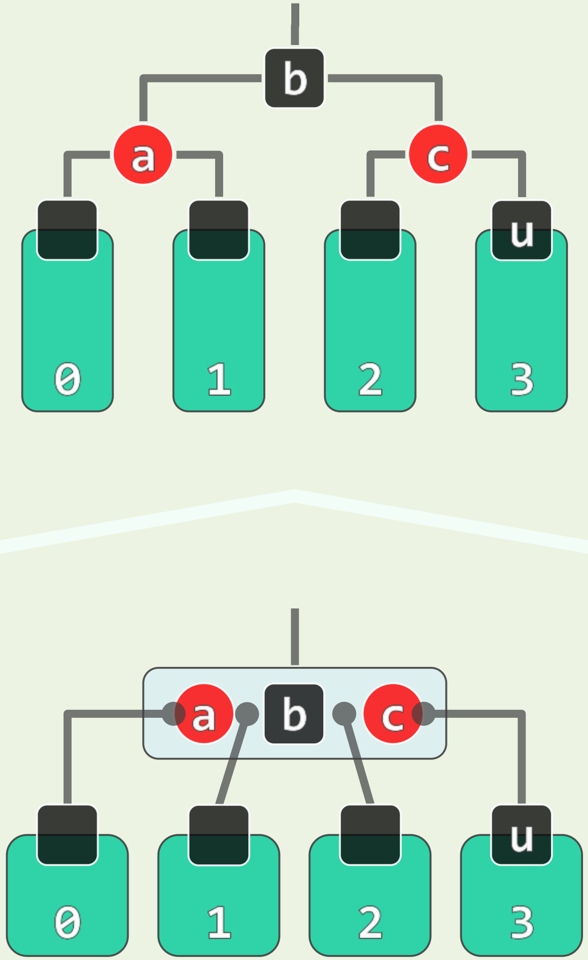
- ❖ 在某**三叉**节点中插入红关键码后

  原黑关键码不再居中（RRB或BRR）

- ❖ 调整的效果，无非是

  将三个关键码的颜色改为**RBR**

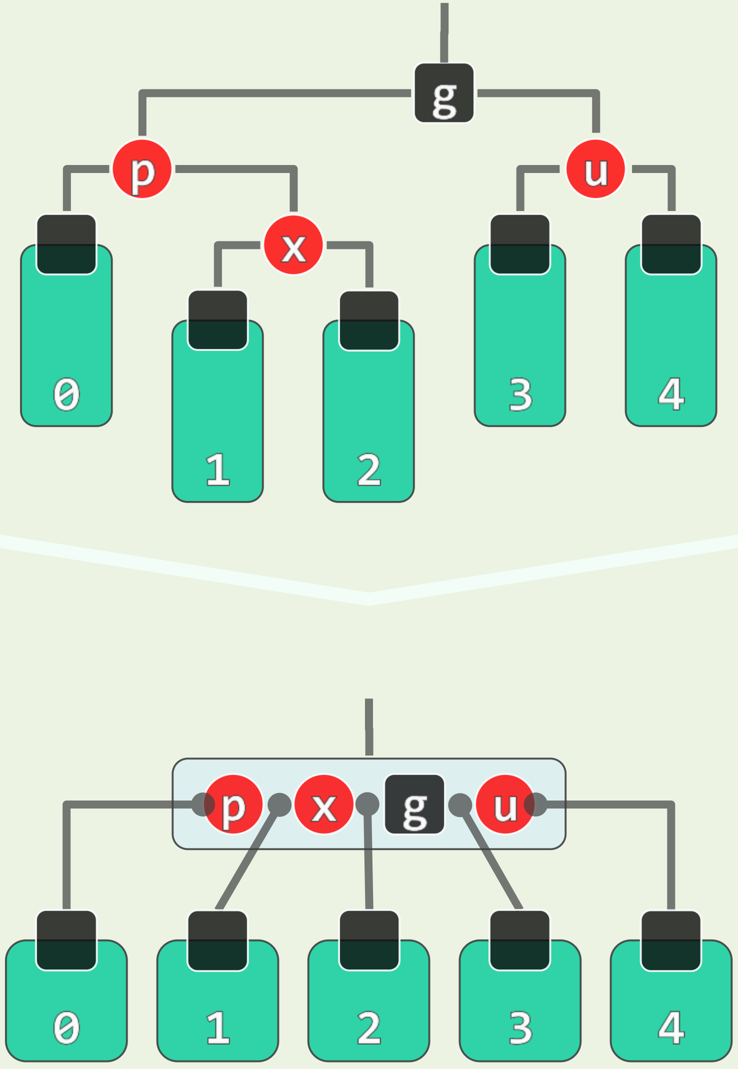
- ❖ 如此调整，**一蹴而就**



RR-1：实现

```
while ( 1 ) {
   /* ...... */
   if ( IsBlack( u ) ) { //u为黑或NULL
   // 若x与p同侧，则p由红转黑（x保持红）；否则，x由红转黑（p保持红）
      ( IsLChild( x ) == IsLChild( p ) ? p : x )->color = RB_BLACK;
   // g由黑转红并绕x旋转后，即完成修复
      g->color = RB_RED; rotateAt( x ); return;
   } else { /* ... u为红 ... */ }
} //while
```

RR-2: u->color == R (1/3)

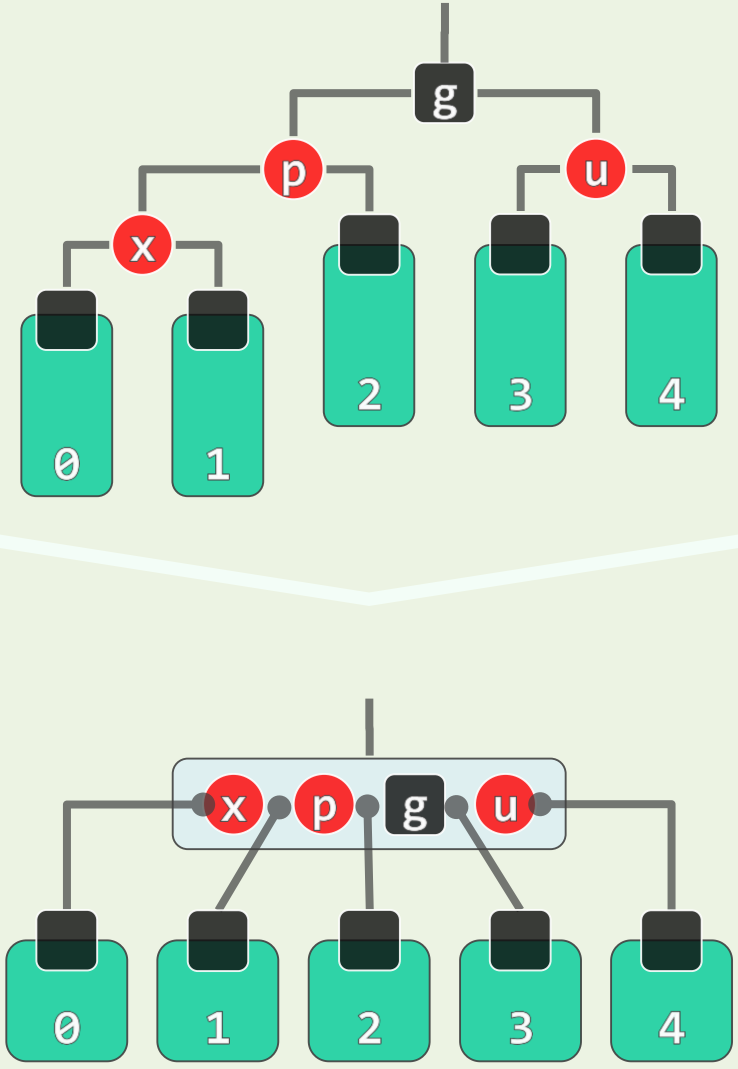


- 在B-树中，等效于

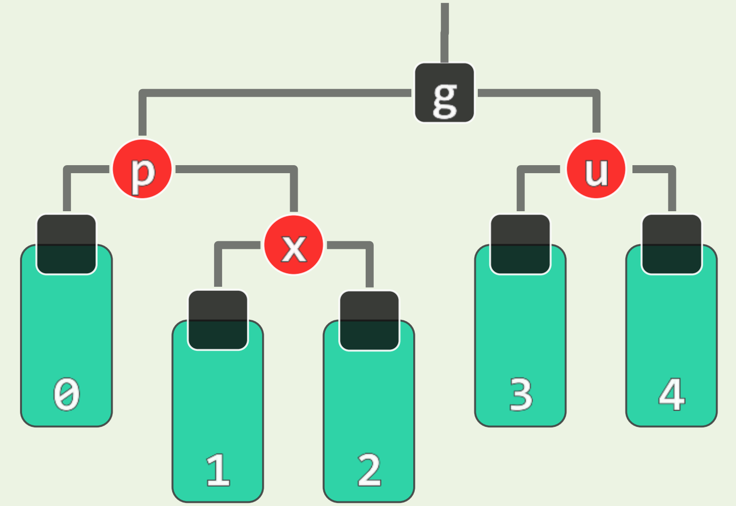
  超级节点发生**上溢**

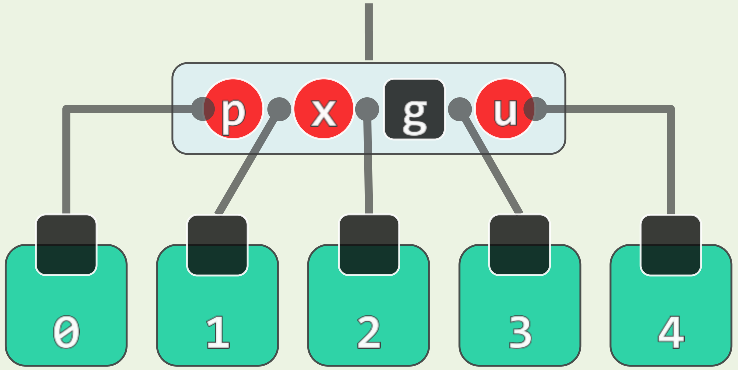
- 另两种对称情况

  请自行补充



RR-2: u->color == R (2/3)



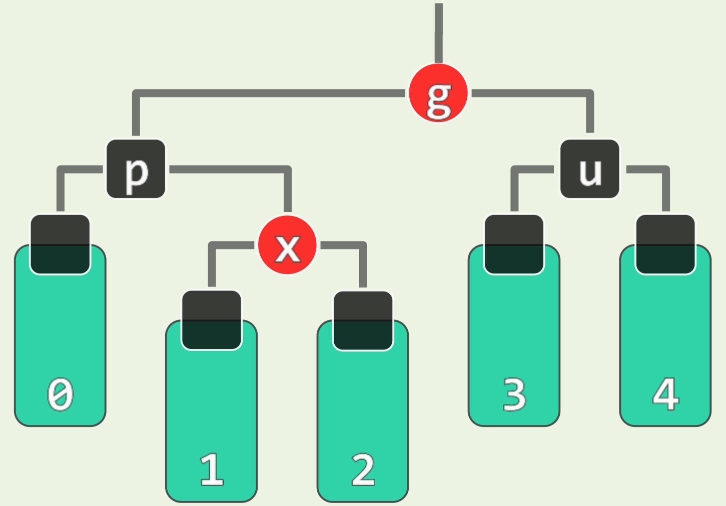


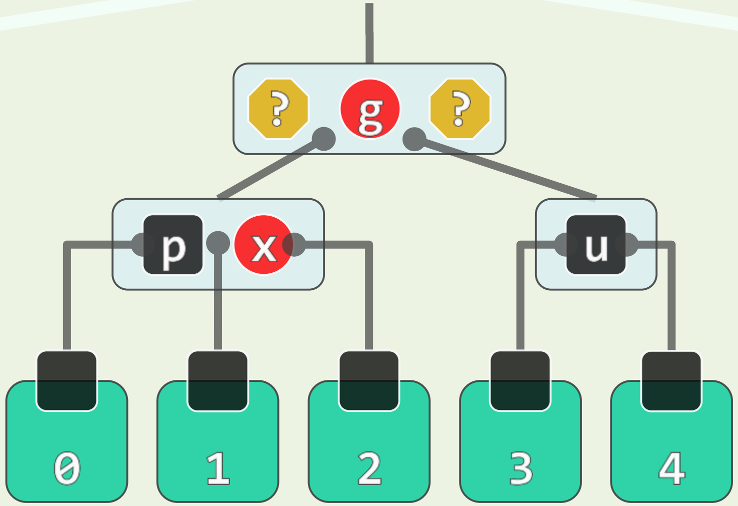
❖ p与u转黑，g转红

在B-树中，等效于...

❖ 节点**分裂**

关键码g上升一层





RR-2: u->color == R (3/3)

- 既是分裂，也应可能会继续向上传播

  ——g与parent(g)再次构成双红

- 果真如此，可等效地将g视作新插入节点

  无非以上两种情况，如法处置而已

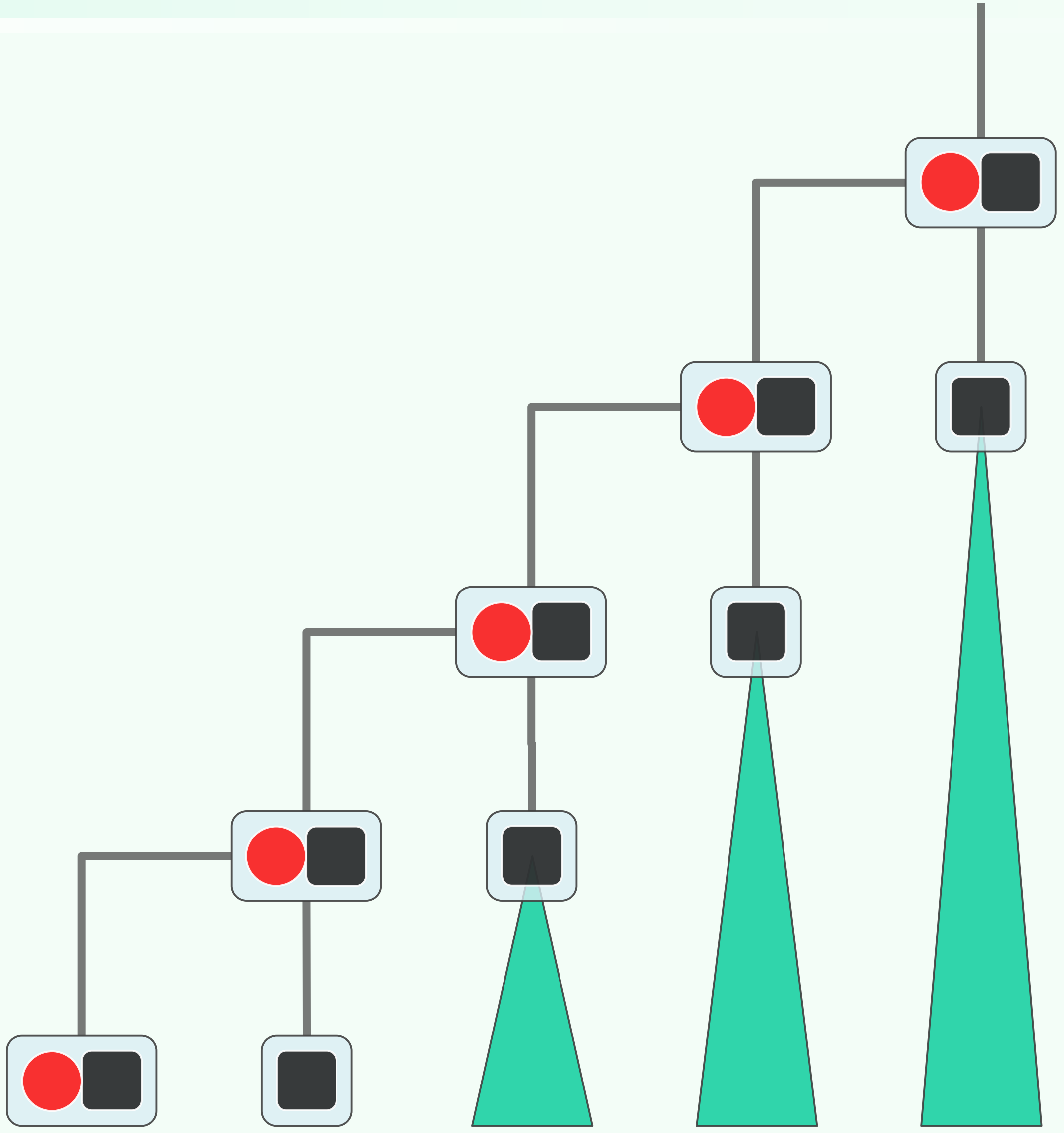
- 直到所有的条件均满足：

  不再双红，或抵达树根

- 若g果真到达树根，则

  强行将其转为黑色

  （整树黑高度加一）



RR-2：实现

```
while ( 1 ) {
   /* ...... */
   if ( IsBlack( u ) ) { /* ... u为黑（含NULL） ... */ }
   ----------------------------------------------------------------
   else { //u为红色
      p->color = RB_BLACK; p->height++; //p由红转黑，增高
      u->color = RB_BLACK; u->height++; //u由红转黑，增高
      g->color = RB_RED; //在B-树中g相当于上交给父节点的关键码，故暂标记为红
      x = g; //继续上溯
   }
} //while
```

复杂度

- 重构、染色均只需常数时间，故只需统计其总次数
- RedBlack::insert()仅需 $\mathcal{O}(\log n)$ 时间
- 其间至多做 $\mathcal{O}(\log n)$ 次重染色、$\mathcal{O}(1)$ 次旋转

	旋转	染色	此后
u为黑	1~2	2	调整随即完成
u为红	0	3	可能再次双红 但必上升两层