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

邓俊辉

deng@tsinghua.edu.cn

算法

❖ 按BST规则插入关键码e //x = insert(e)必为叶节点

❖ 除非系首个节点（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...

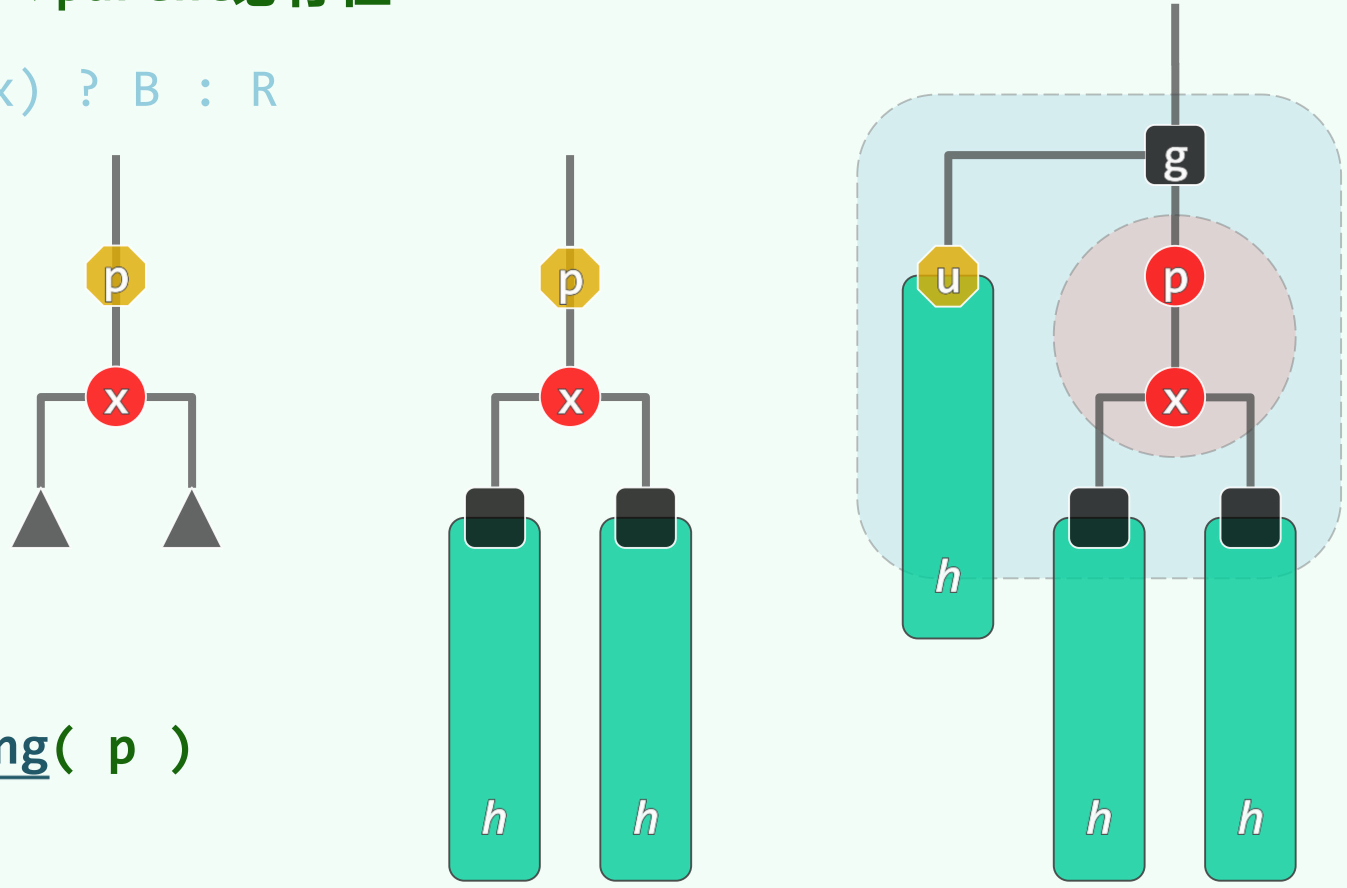
❖ 双红（double-red）

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

❖ 考查：祖父g = p->parent

叔父u = uncle(x) = sibling(p)

❖ 以下，视u的颜色，分两种情况处理...



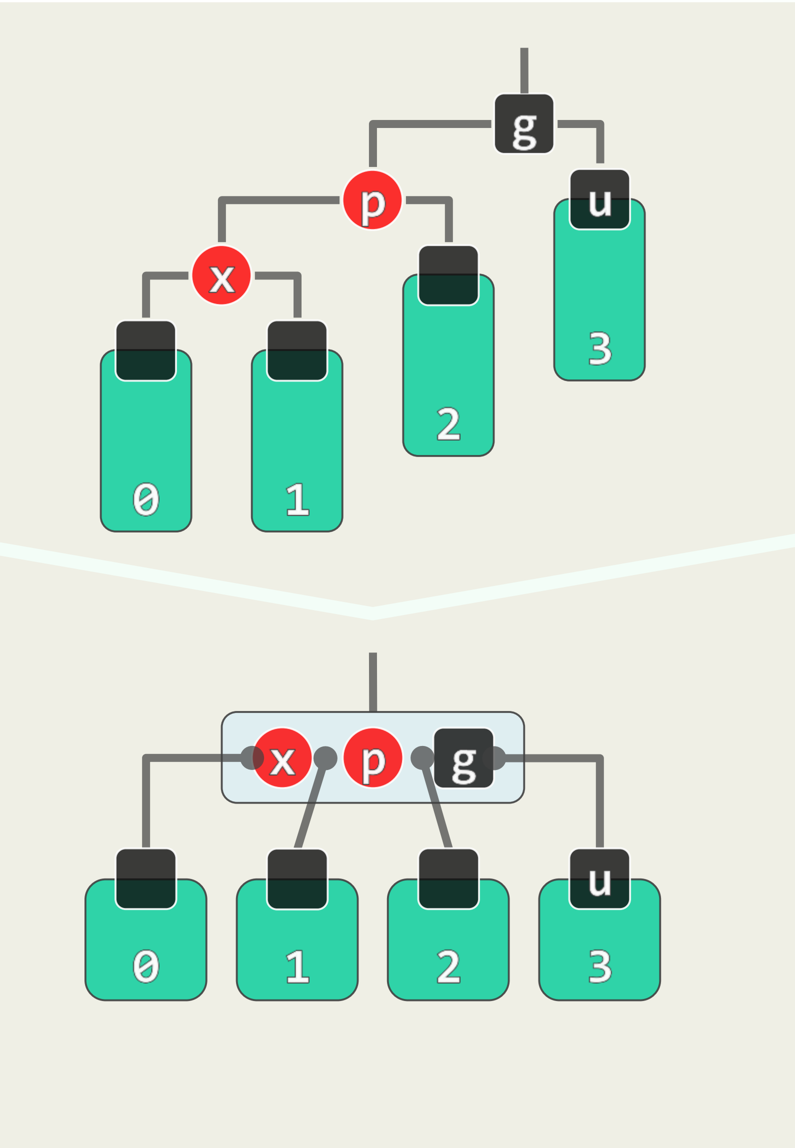
实现

```
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 ) {
   if ( IsRoot( *x ) ) { //若已（递归）转至树根，则将其转黑，整树黑高度也随之递增
      { _root->color = RB_BLACK; _root->height++; return; } //否则...
   BinNodePosi<T> p = x->parent; //考查x的父亲p（必存在）
   if ( IsBlack( p ) ) return; //若p为黑，则可终止调整；否则
   BinNodePosi<T> g = p->parent; //x祖父g必存在，且必黑
   BinNodePosi<T> u = uncle( x ); //以下视叔父u的颜色分别处理
   if ( IsBlack( u ) ) { /* ... u为黑（或NULL） ... */ }
   else                { /* ... u为红 ... */ }
}
```

RR-1: u->color == B

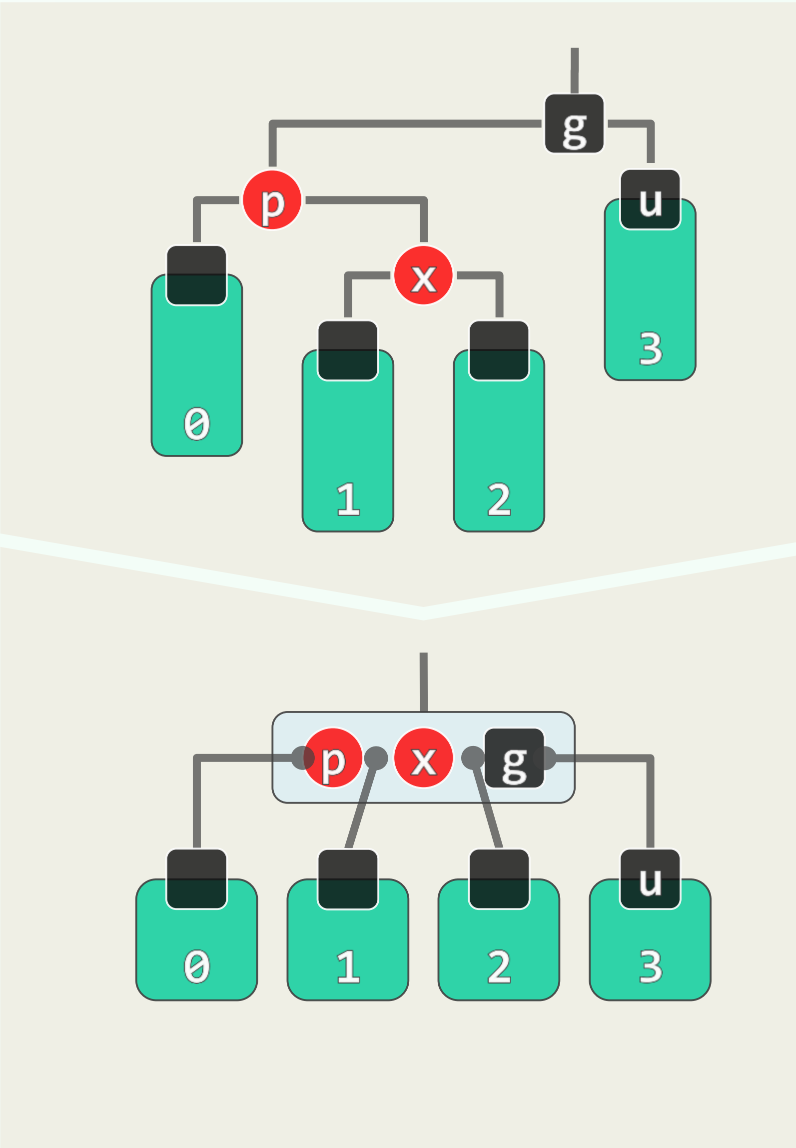


- ❖ 此时，x、p、g的四个孩子

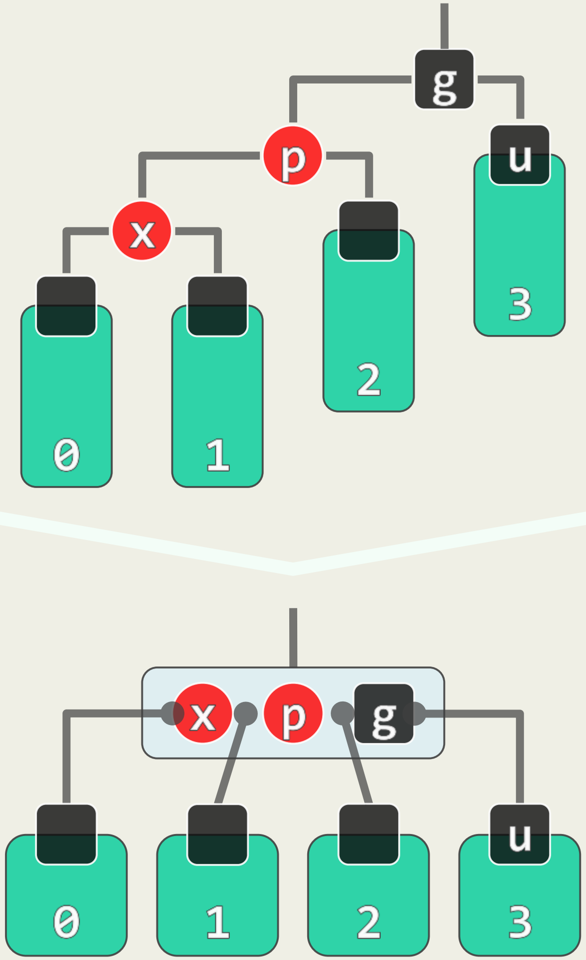
  (可能是外部节点)

  - 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...

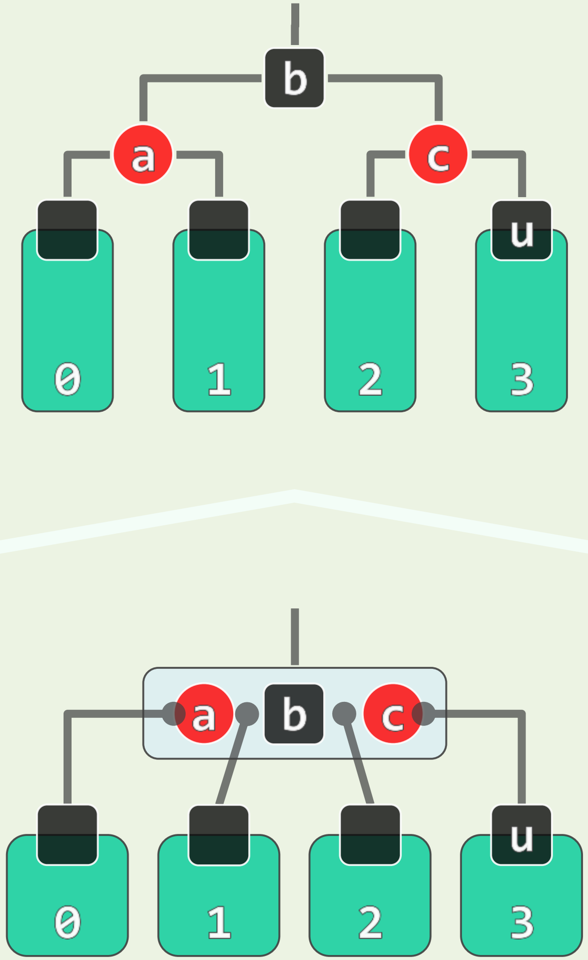
- 在某三叉节点中插入红关键码后

  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

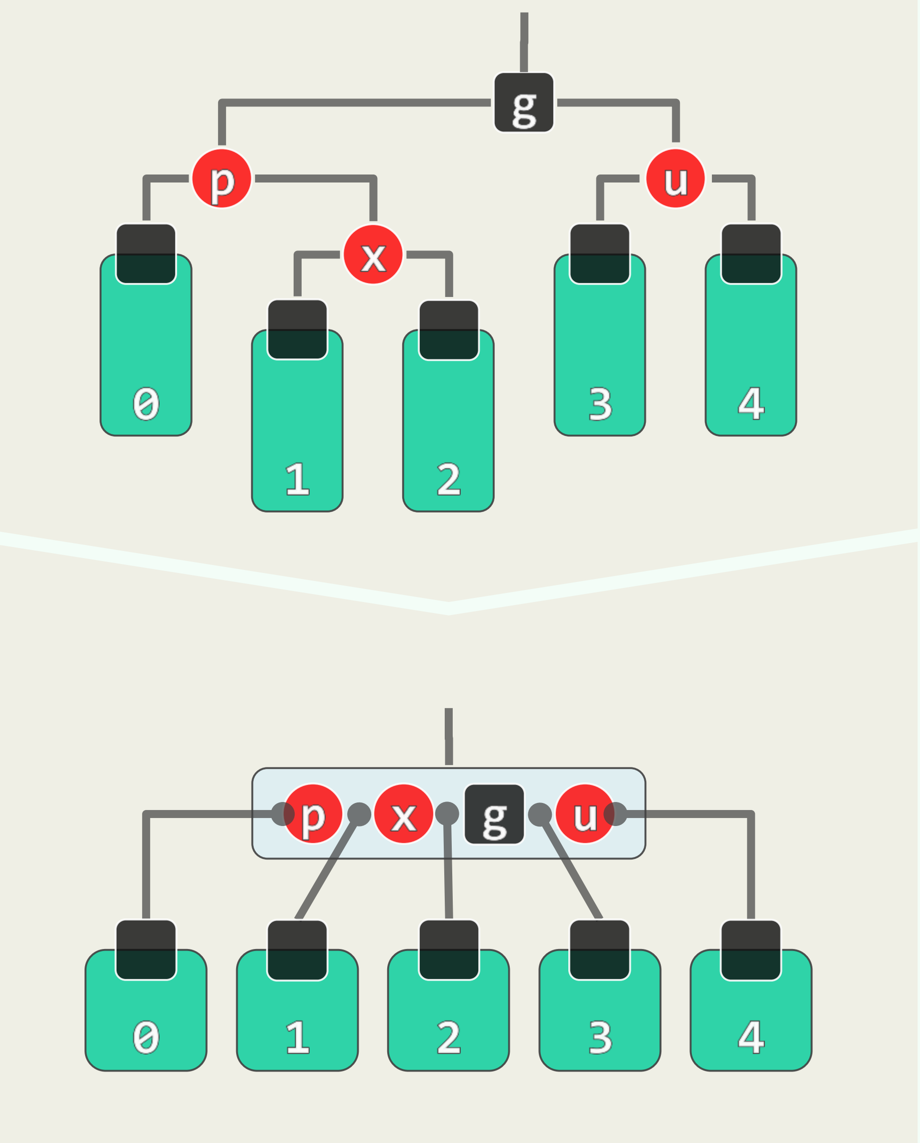
- 如此调整，一蹴而就



RR-1：实现

```
template <typename T> void RedBlack<T>::solveDoubleRed( BinNodePosi<T> x ) {
   /* ...... */
   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
      if ( IsLChild( *x ) == IsLChild( *p ) ) p->color = RB_BLACK;
      else                                    x->color = RB_BLACK;
      g->color = RB_RED; //g必定由黑转红
      BinNodePosi<T> gg = g->parent; //great-grand parent
      BinNodePosi<T> r = FromParentTo( *g ) = rotateAt( x );
      r->parent = gg; //调整之后的新子树，需与原曾祖父联接
   } else { /* ... u为红 ... */ }
}
```

RR-2: u->color == R

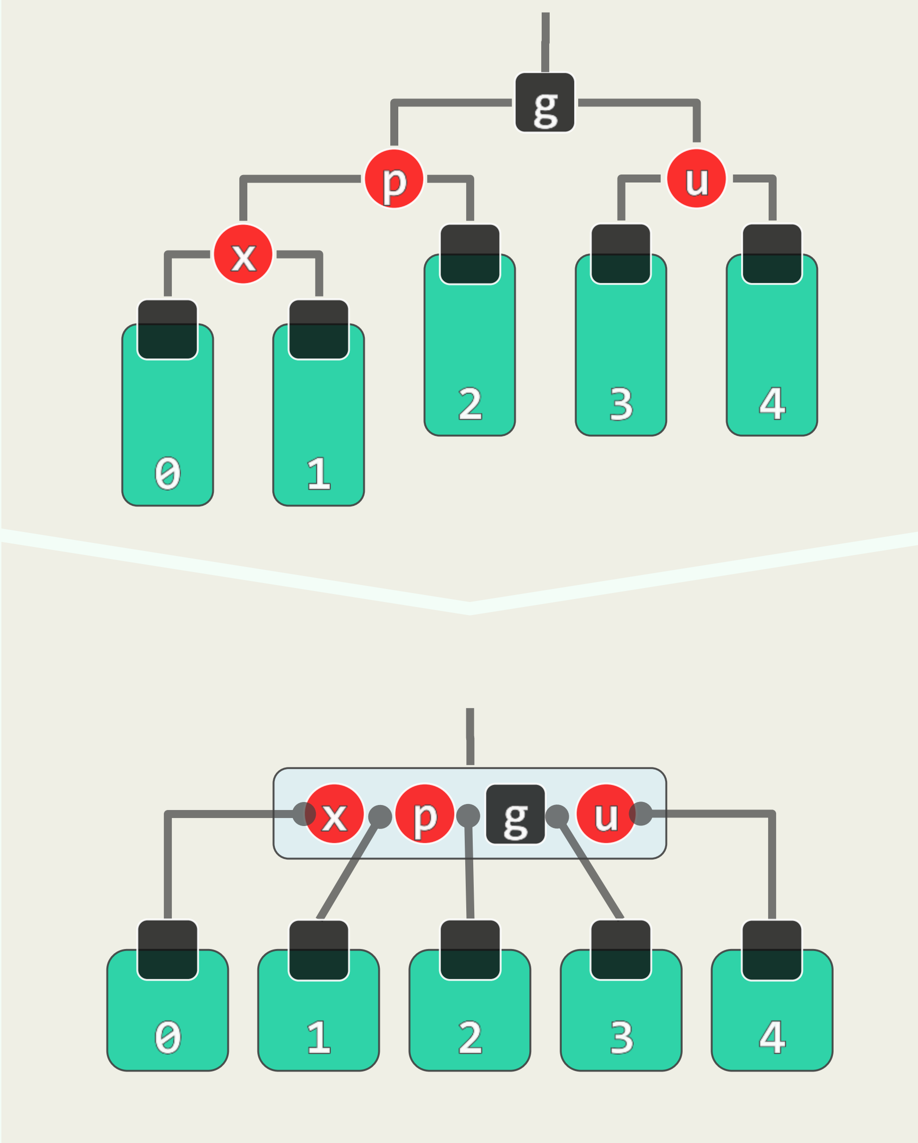


❖ 在B-树中，等效于

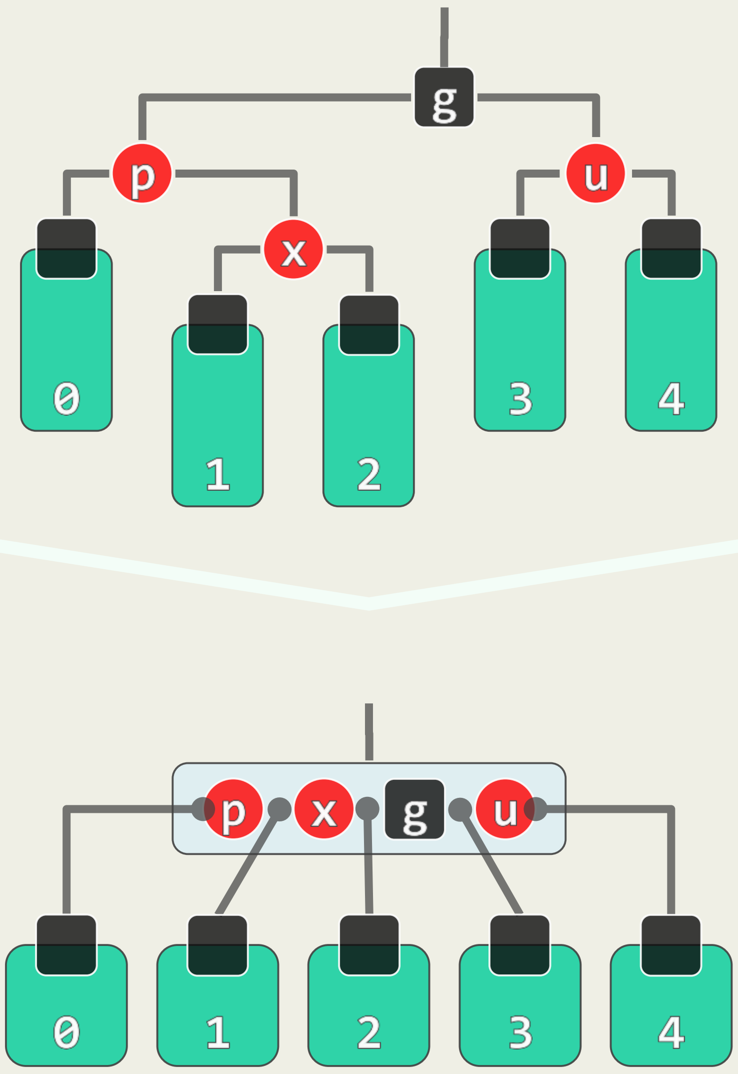
超级节点发生上溢

❖ 另两种对称情况

请自行补充



RR-2: u->color == R

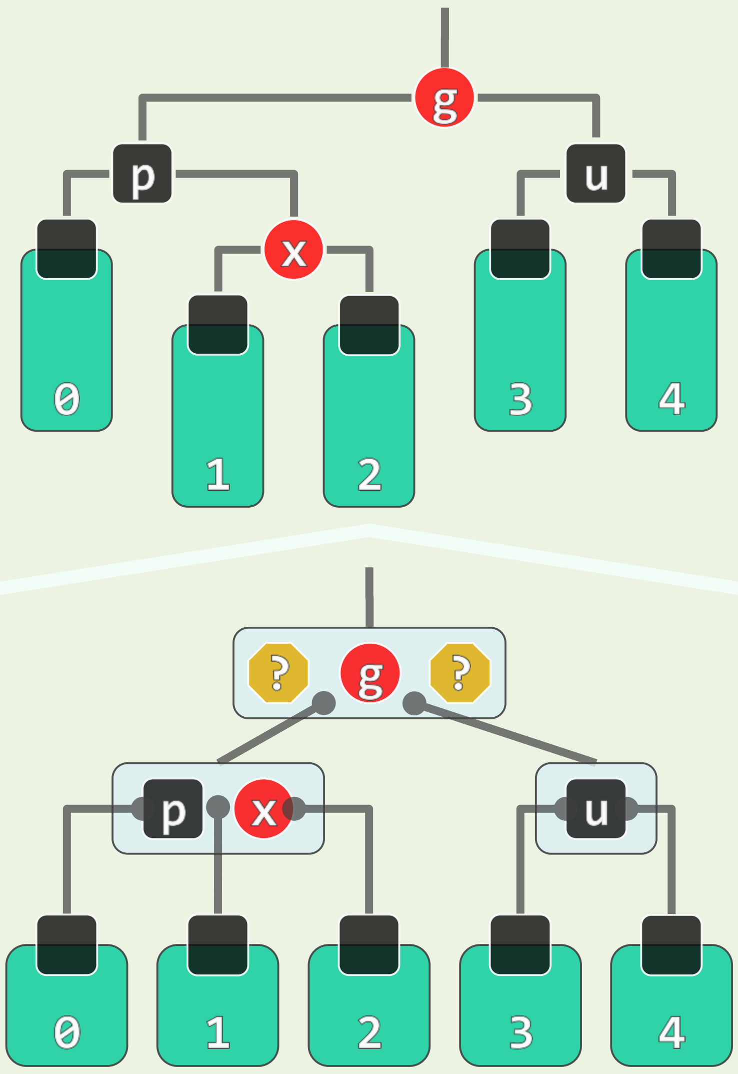


❖ p与u转黑，g转红

在B-树中，等效于...

❖ 节点**分裂**

关键码g上升一层



RR-2: u->color == R

❖ 既然是分裂，也应有**可能**继续向上传递——亦即，g与parent(g)**再次构成双红**

❖ 果真如此，可：等效地将g视作新插入的节点

区分以上两种情况，如法处置

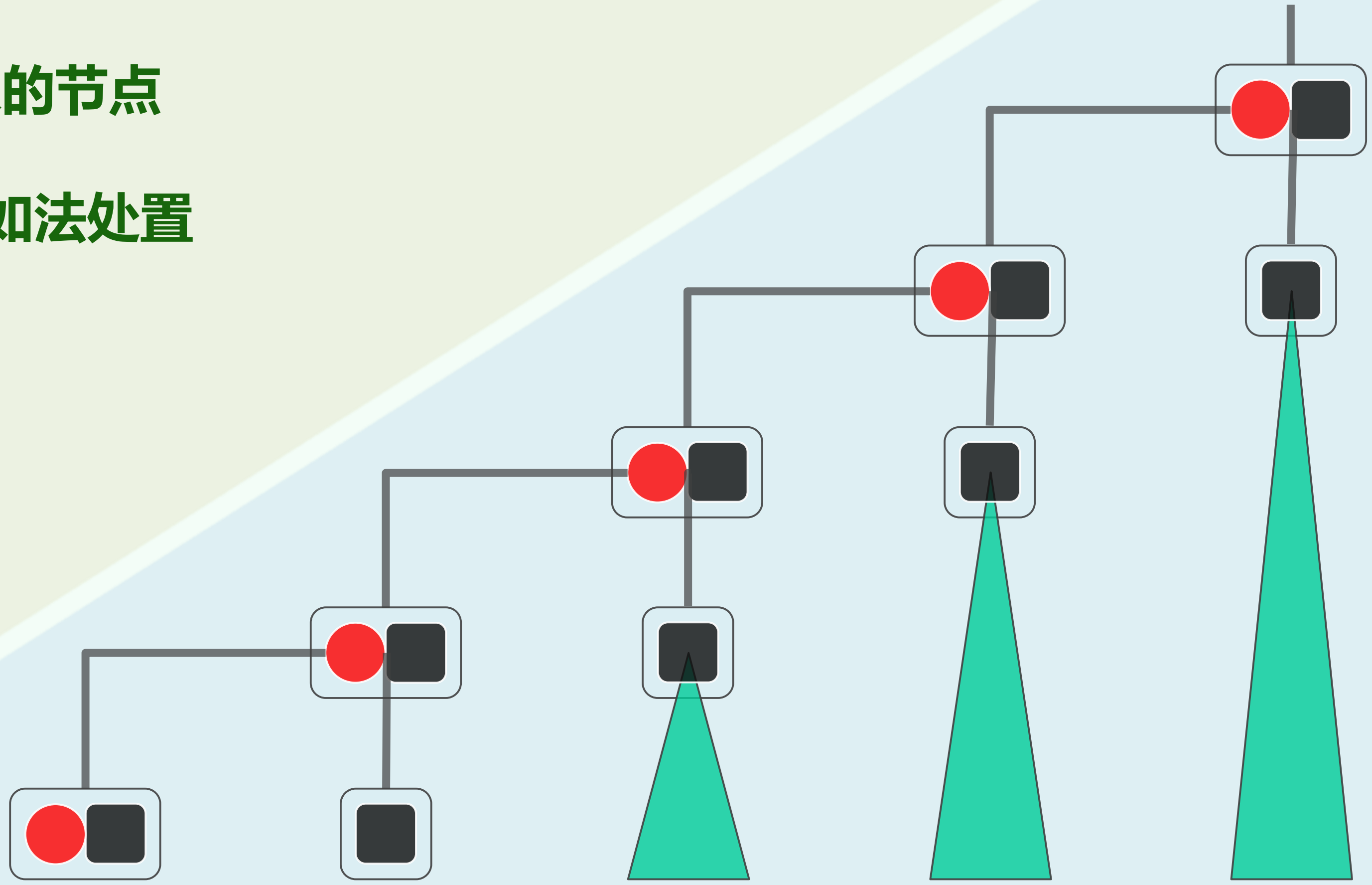
❖ 直到所有条件满足（即不再双红）

或者抵达树根

❖ g若果真到达树根，则

强行将其转为黑色

（整树黑高度加一）



RR-2：实现

```
template <typename T> void RedBlack<T>::solveDoubleRed( BinNodePosi<T> x ) {
   /* ...... */
   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
      solveDoubleRed( g ); //继续调整：若已至树根，接下来的递归会将g转黑（尾递归）
   }
}
```

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