

08-C4

高级搜索树

红黑树：删除

他仿佛这一刻才第一次看见这些颜色，并为它们取下崭新又美妙的名字。
在这里，没有人会在冬天时哀悼已逝去的夏天或春天。

邓俊辉

deng@tsinghua.edu.cn

等效删除

❖ 首先按照BST常规算法，执行

```
r = removeAt( x, _hot )
```

//实际被摘除的可能是x的前驱或后继w

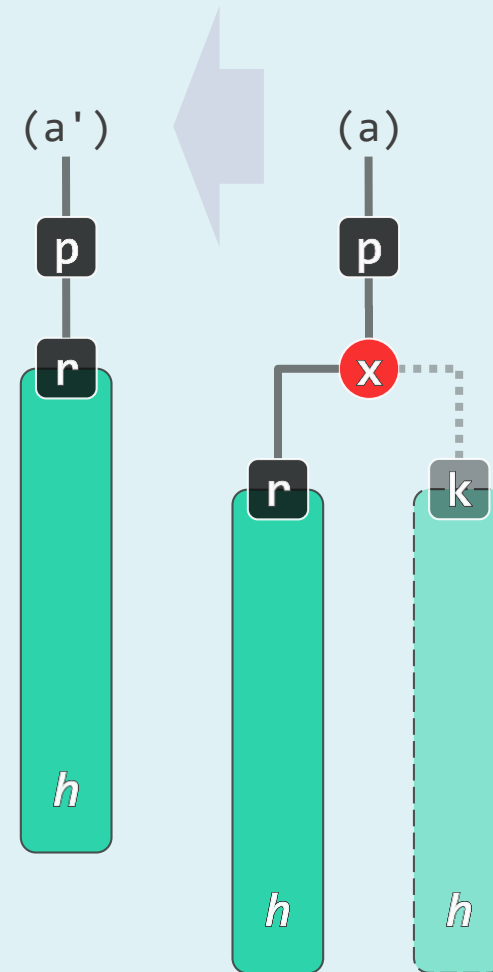
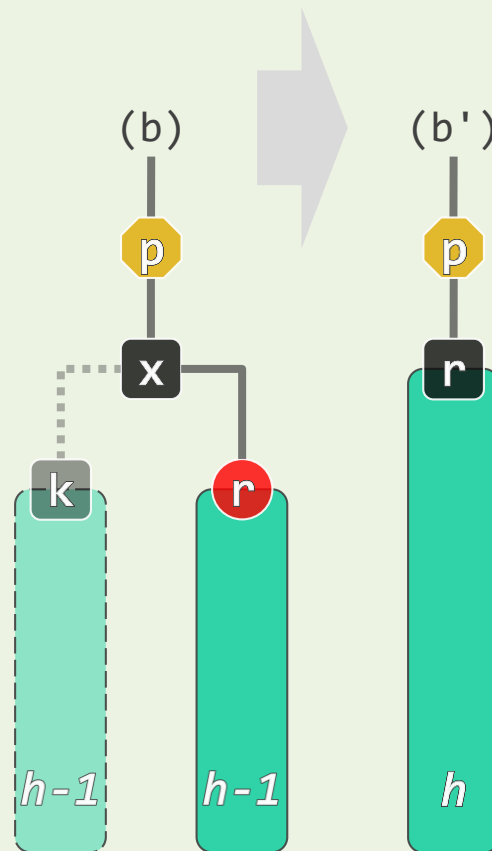
//简捷起见，以下不妨统称作x

❖ x由孩子r接替，此时另一孩子k必为NULL

❖ 但在随后的调整过程中，x可能逐层上升

❖ 故需**假想地、统一地、等效地**理解为：

- k为一棵黑高度与r相等的子树，且
- 随x一并摘除（尽管实际上从未存在过）



其一为红

❖ 完成`removeAt()`之后

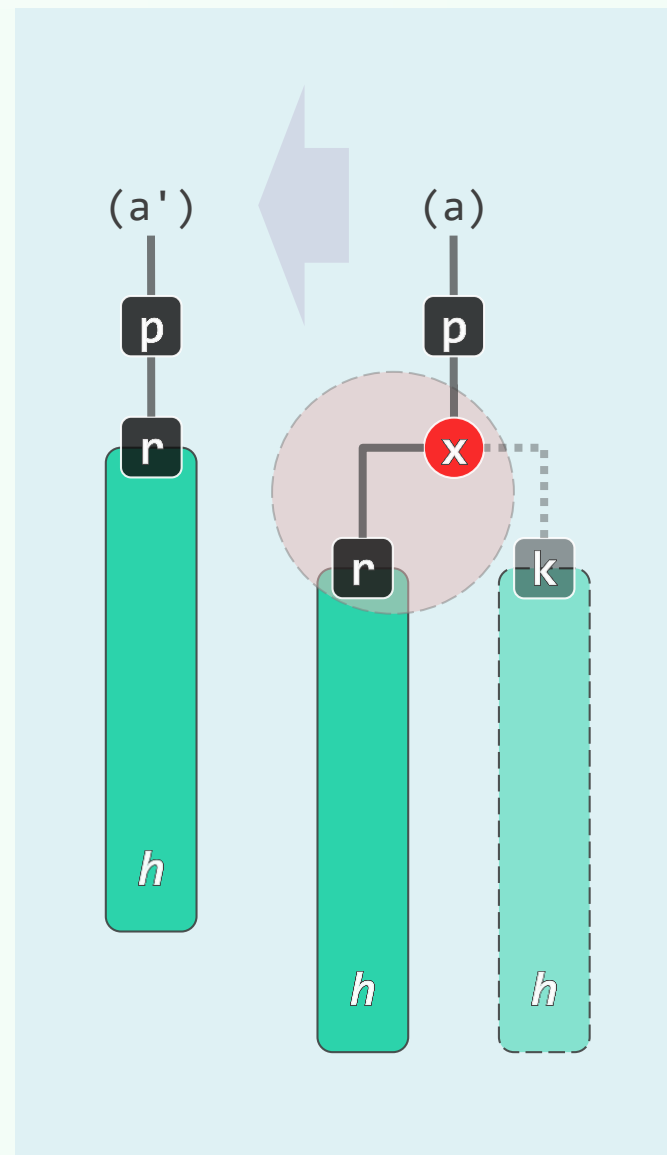
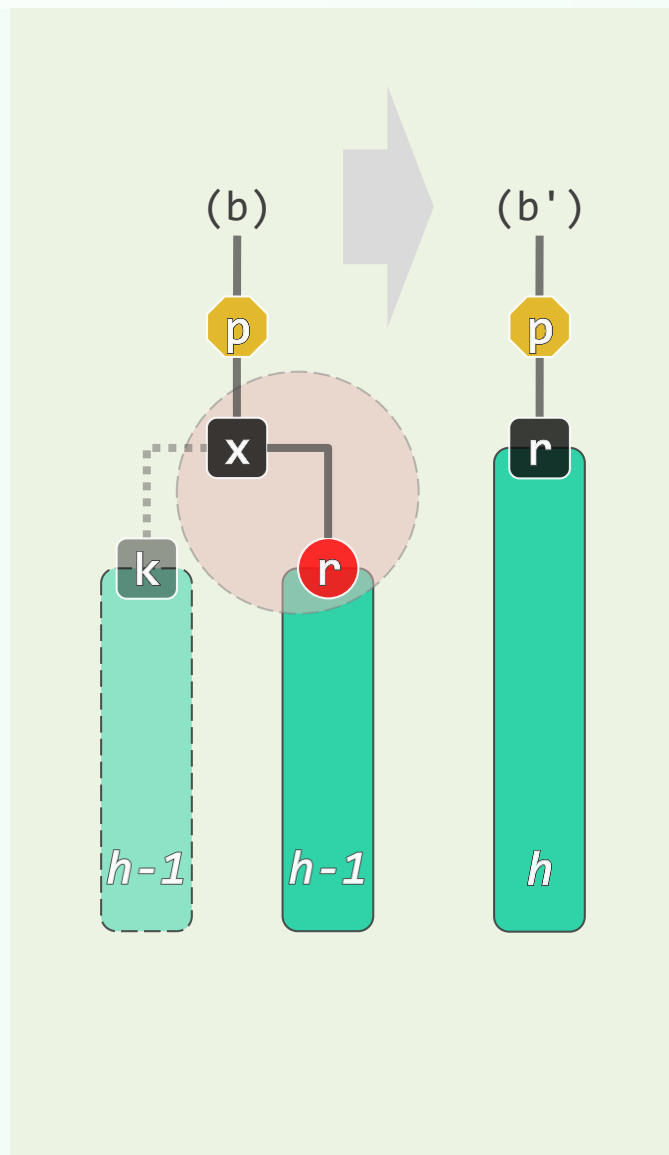
- 条件1、2依然满足
- 但条件3、4却不见得

❖ 在原树中，考查x与r

- 若x为红，则条件3、4自然满足
- 若r为红，则令其与x交换颜色

❖ 总之，无论x或r为红，则3、4均不难满足

——删除遂告完成！



双黑

❖ 若x与r均黑 (double black) , 则不然...

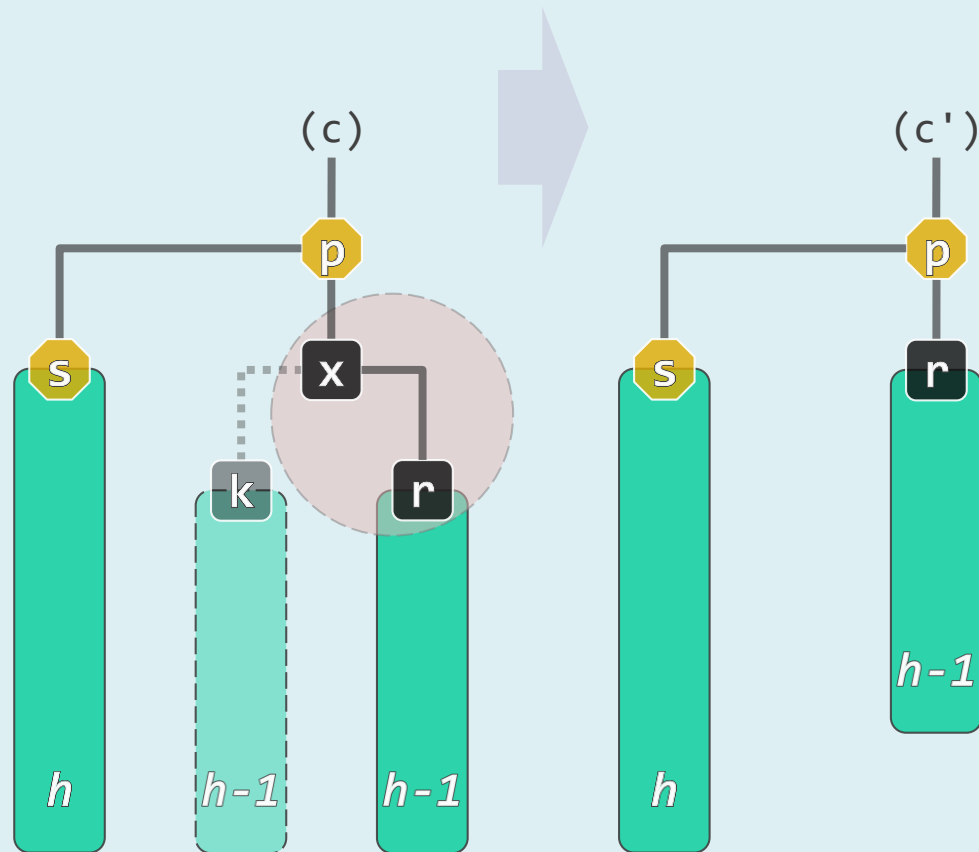
❖ 摘除x并代之以r后, 全树**黑深度**不再统一
(稍后可见, 等效于B-树中x所属节点**下溢**)

❖ 在新树中, 考查r的父亲、兄弟

- $p = r \rightarrow \text{parent}$ //亦是原x的父亲

- $s = \text{sibling}(r)$

❖ 以下分四种情况处理...



实现 (1/2)

```
template <typename T> bool RedBlack<T>::remove( const T & e ) {  
    BinNodePosi<T> & x = search( e ); if ( !x ) return false; //查找定位  
    BinNodePosi<T> r = removeAt( x, _hot ); //删除_hot的某孩子, r指向其接替者  
    if ( ! ( -- _size ) ) return true; //若删除后为空树, 可直接返回  
    if ( ! _hot ) { //若被删除的是根, 则  
        _root->color = RB_BLACK; //将其置黑, 并  
        updateHeight( _root ); //更新 (全树) 黑高度  
        return true;  
    } //至此, 原x (现r) 必非根
```

实现 (2/2)

// 若父亲（及祖先）依然平衡，则无需调整

```
if ( BlackHeightUpdated( * _hot ) ) return true;
```

// 至此，必失衡

// 若替代节点r为红，则只需简单地翻转其颜色

```
if ( IsRed( r ) ) { r->color = RB_BLACK; r->height++; return true; }
```

// 至此，r以及被其替代的x均为黑色

```
solveDoubleBlack( r ); //双黑调整（入口处必有 r == NULL）
```

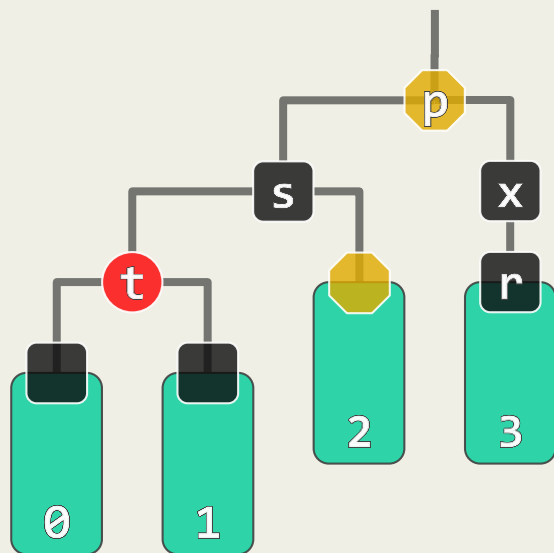
```
return true;
```

```
}
```

双黑修正

```
template <typename T> void RedBlack<T>::solveDoubleBlack( BinNodePosi<T> r ) {  
    BinNodePosi<T> p = r ? r->parent : _hot; if ( !p ) return; //r的父亲  
    BinNodePosi<T> s = (r == p->lc) ? p->rc : p->lc; //r的兄弟  
    if ( IsBlack( s ) ) { //兄弟s为黑  
        BinNodePosi<T> t = NULL; //s的红孩子（若左、右孩子皆红，左者优先；皆黑时为NULL）  
        if ( IsRed ( s->rc ) ) t = s->rc;  
        if ( IsRed ( s->lc ) ) t = s->lc;  
        if ( t ) { /* ... 黑s有红孩子: BB-1 ... */ }  
        else { /* ... 黑s无红孩子: BB-2R或BB-2B ... */ }  
    } else { /* ... 兄弟s为红: BB-3 ... */ }  
}
```

BB-1: s为黑, 且至少有一个红孩子t



❖ “3+4” 重构:

$t \sim a$

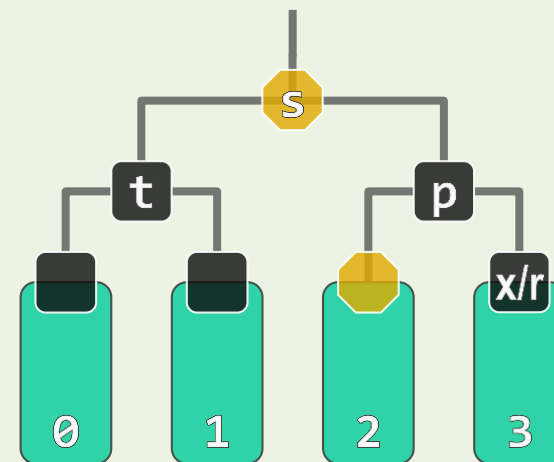
$s \sim b$

$p \sim c$

❖ r保持黑

a、c染黑

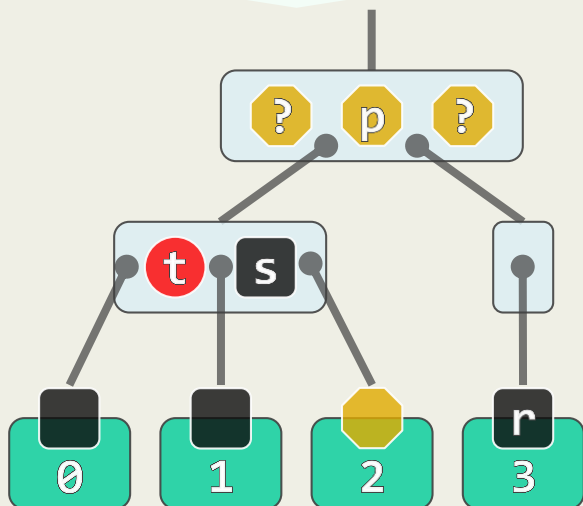
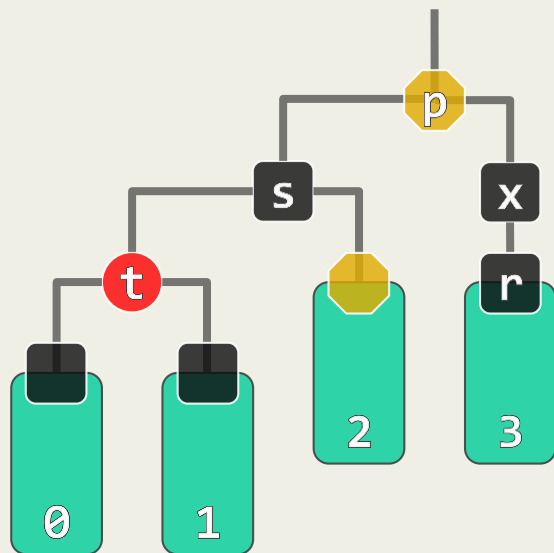
b继承p的原色



❖ 如此, 红黑树性质在全局得以恢复——删除完成! //zig-zag等类似

❖ 在对应的B-树中, 以上操作等效于...

BB-1: s 为黑, 且至少有一个红孩子 t



❖ 通过关键码的**旋转**

消除超级节点的下溢

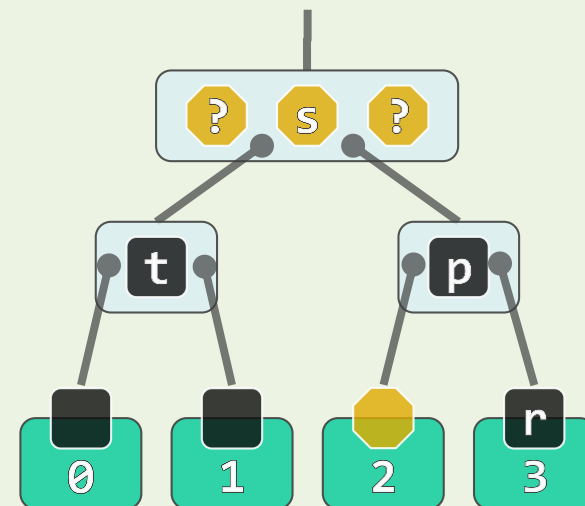
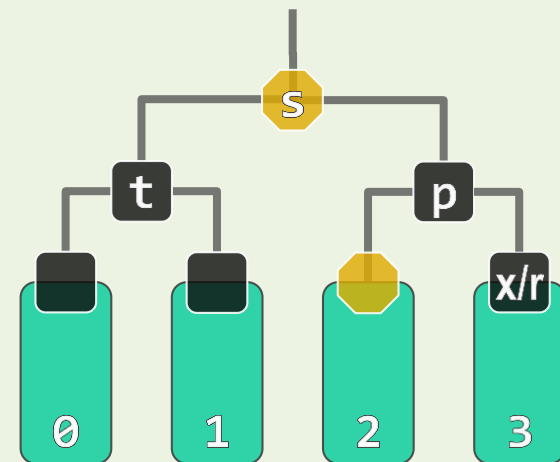
❖ 在对应的B-树中

- p 若为红

问号之一为黑关键码

- p 若为黑

必自成一个超级节点



BB-1: 实现

```
if ( IsBlack( s ) ) { //兄弟s为黑

    /* ..... */

    if ( t ) { //黑s有红孩子: BB-1

        RBColor oldColor = p->color; //备份p颜色, 并对t、父亲、祖父

        BinNodePosi<T> b = FromParentTo( *p ) = rotateAt( t ); //旋转

        if ( HasLChild( *b ) ) { b->lc->color = RB_BLACK; updateHeight( b->lc ); }

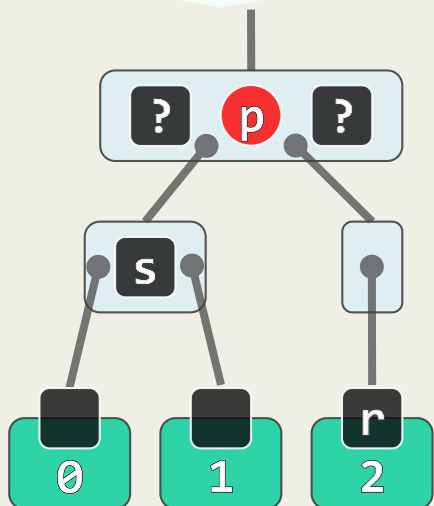
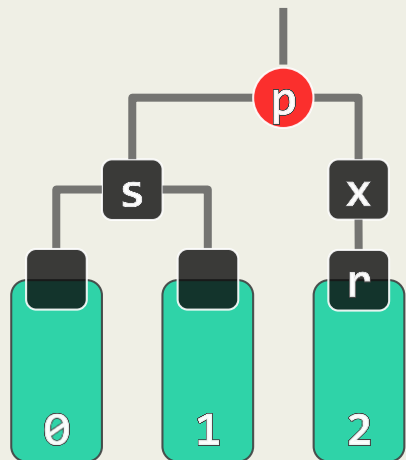
        if ( HasRChild( *b ) ) { b->rc->color = RB_BLACK; updateHeight( b->rc ); }

        b->color = oldColor; updateHeight( b ); //新根继承原根的颜色

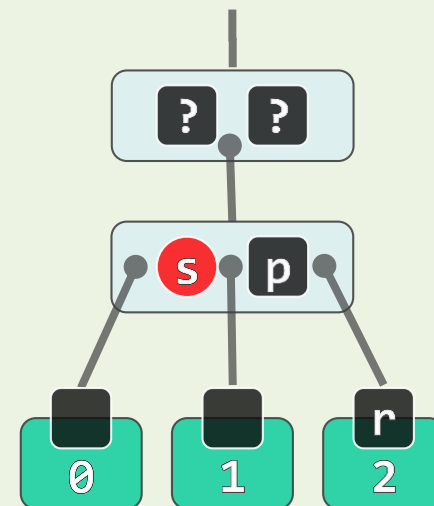
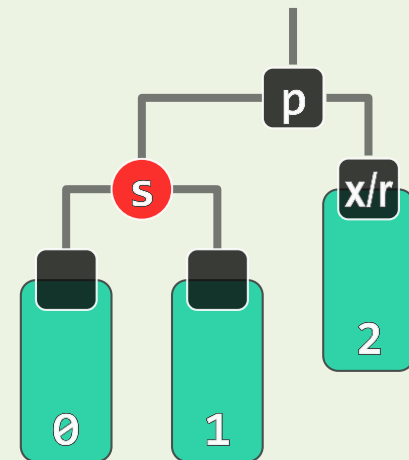
    } else { /* ... 黑s无红孩子: BB-2R或BB-2B ... */ }

} else { /* ... 兄弟s为红: BB-3 ... */ }
```

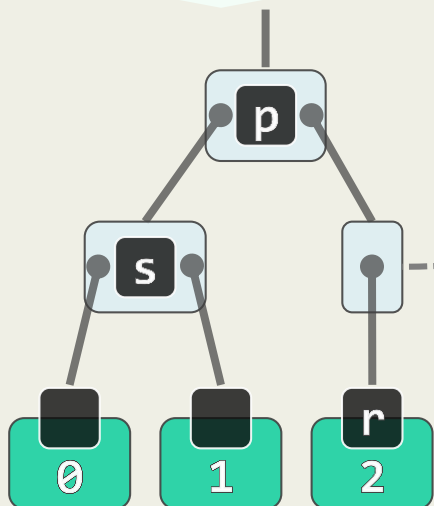
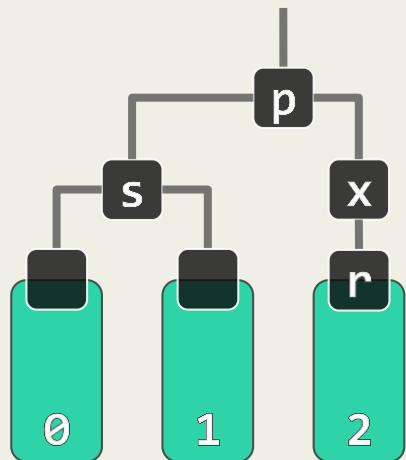
BB-2R: s为黑, 且两个孩子均为黑; p为红



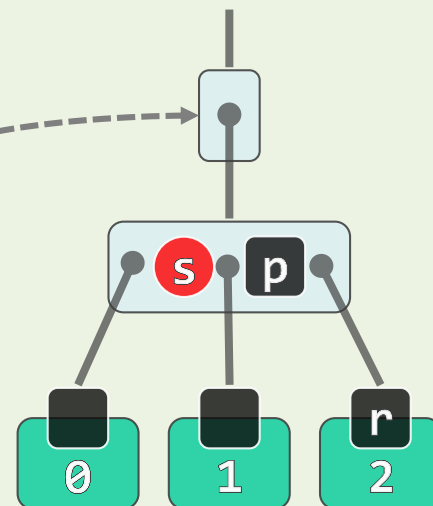
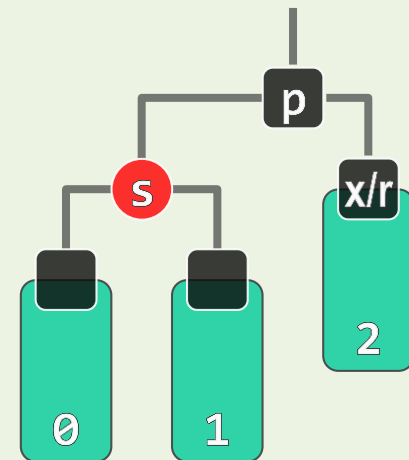
- ❖ r 保持黑; s 转红; p 转黑
- ❖ 在对应的B-树中, 等效于
下溢节点与兄弟合并
- ❖ 红黑树性质在**全局**得以恢复
- ❖ 失去关键码 p 后, 上层节点
会否**继**而下溢? 不会!
- ❖ 合并之前, 在 p 之左或右侧
还应有一个黑关键码



BB-2B: s为黑, 且两个孩子均为黑; p为黑



- ❖ s转红; r与p保持黑
- ❖ 红黑树性质在局部得以恢复
- ❖ 在对应的B-树中, 等效于
下溢节点与兄弟合并
- ❖ 合并前, p和s均属于单关键码节点
- 孩子的下溢修复后, 父节点继而下溢
- ❖ 好在可继续分情况处理
高度递增, 至多 $O(\log n)$ 层 (步)

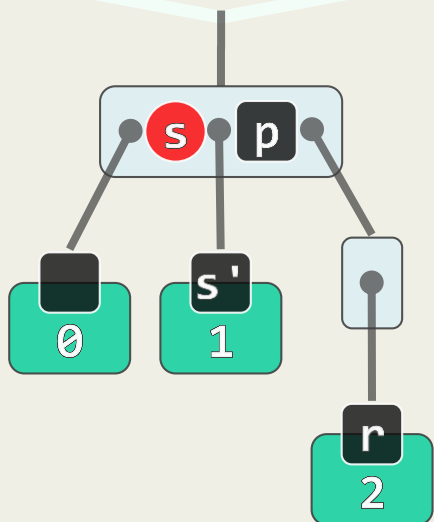
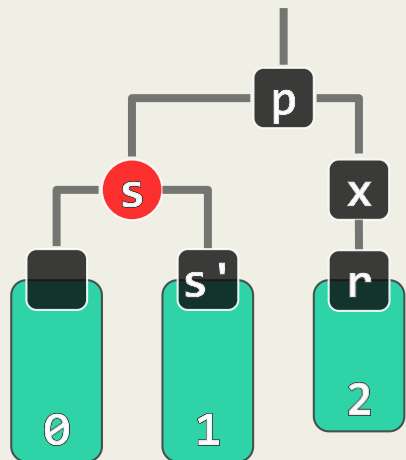


BB-(2R+2B): 实现

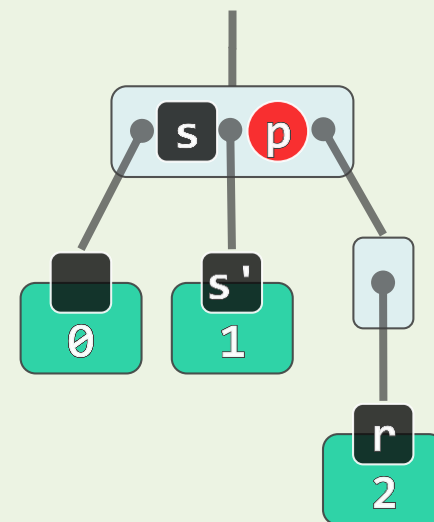
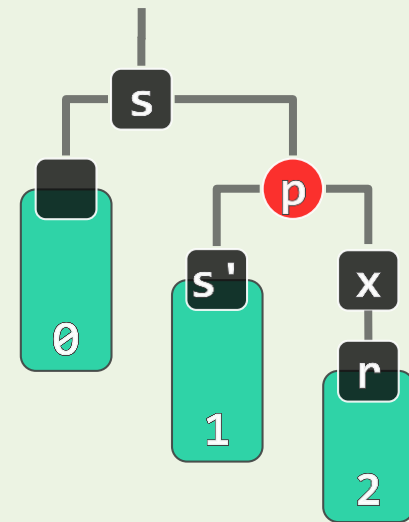
```
if ( IsBlack( s ) ) { //兄弟s为黑
    /* ..... */
    if ( t ) { /* ... 黑s有红孩子: BB-1 ... */ }
    else { /* 黑s无红孩子 */
        s->color = RB_RED; s->height--; //s转红
        if ( IsRed( p ) ) //BB-2R: p转黑, 但黑高度不变
            { p->color = RB_BLACK; }
        else //BB-2B: p保持黑, 但黑高度下降; 递归修正
            { p->height--; solveDoubleBlack( p ); }
    }
}

} else { /* ... 兄弟s为红: BB-3 ... */ }
```

BB-3: s为红 (其孩子均为黑)



- ❖ 绕p单旋; s红转黑, p黑转红
- ❖ 黑高度依然异常, 但...
- ❖ r有了新的黑兄弟s'
故转化为前述情况, 而且...
- ❖ 既然p已转红, 接下来
 - 绝不会是BB-2B
 - 而只能是BB-2R或BB-1
- ❖ 于是, 再经一轮调整
红黑树性质必然全局恢复



BB-3: 实现

```
if ( IsBlack( s ) ) { //兄弟s为黑

    if ( t ) { /* ... 黑s有红孩子: BB-1 ... */ }

    else { /* ... 黑s无红孩子: BB-2R或BB-2B ... */ }

} else { //兄弟s为红: BB-3

    s->color = RB_BLACK; p->color = RB_RED; //s转黑, p转红

    BinNodePosi<T> t = IsLChild( *s ) ? s->lc : s->rc; //取t与其父s同侧

    _hot = p; FromParentTo( *p ) = rotateAt( t ); //对t及其父亲、祖父做平衡调整

    solveDoubleBlack( r ); //继续修正r——此时p已转红, 故后续只能是BB-1或BB-2R

}
```

复杂度

❖ `RedBlack<T>::remove()`

仅需 $O(\log n)$ 时间

- $O(\log n)$ 次重染色
- $O(1)$ 次旋转

| | 旋转 | 染色 | 此后 |
|----------------|-----|----|---------------|
| (1) 黑s有红子t | 1~2 | 3 | 调整随即完成 |
| (2R) 黑s无红子, p红 | 0 | 2 | 调整随即完成 |
| (2B) 黑s无红子, p黑 | 0 | 1 | 必再次双黑, 但将上升一层 |
| (3) 红s | 1 | 2 | 转为(1)或(2R) |

