

Red-Black Tree

8. 高级搜索树

(xa1) 红黑树：动机

As she looks at the blood on the snow, she
says to herself, "Oh, how I wish that I had
a daughter that had skin WHITE as snow, lips
RED as blood, and hair BLACK as ebony".

邓俊辉

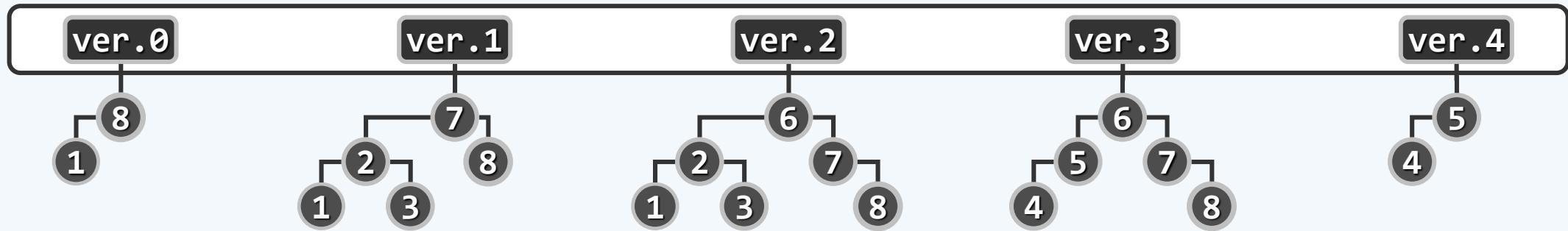
deng@tsinghua.edu.cn

一致性结构

- ❖ **Persistent structure**: 支持对**历史**版本的访问 //ephemeral

T.search(**ver**, key); T.insert(**ver**, key); T.remove(**ver**, key)

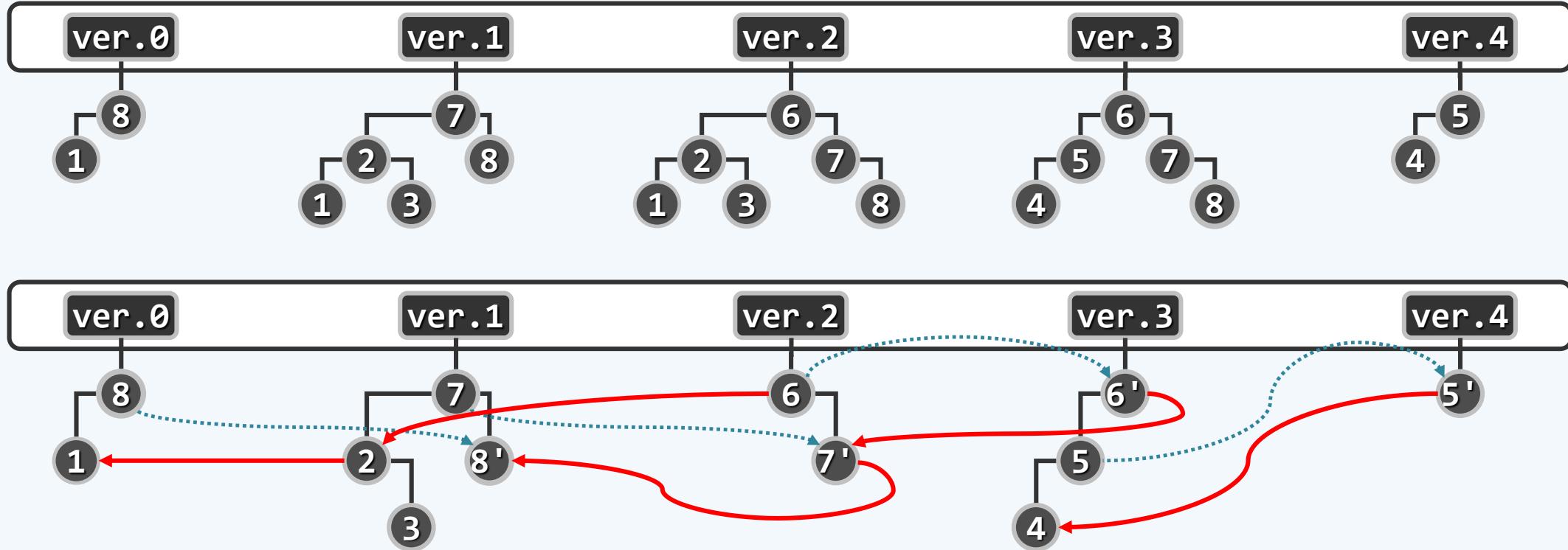
- ❖ 蛮力实现：每个版本独立保存；各版本入口自成一个搜索结构



- ❖ 单次操作 $O(\log h + \log n)$, 累计 $O(h*n)$ 时间/空间 //h = |history|
- ❖ 挑战：可否将复杂度控制在 $O(n + h*\log n)$ 内？
- ❖ 可以！为此需利用相邻版本之间的**关联性**...

$\mathcal{O}(1)$ 重构

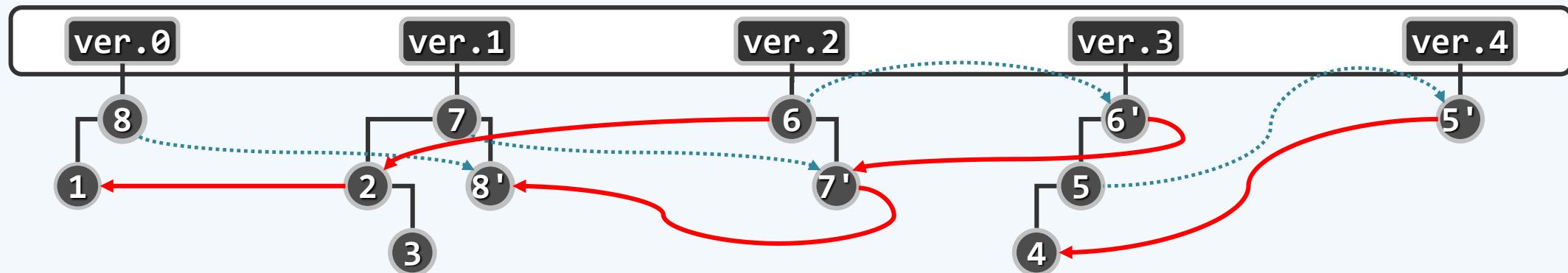
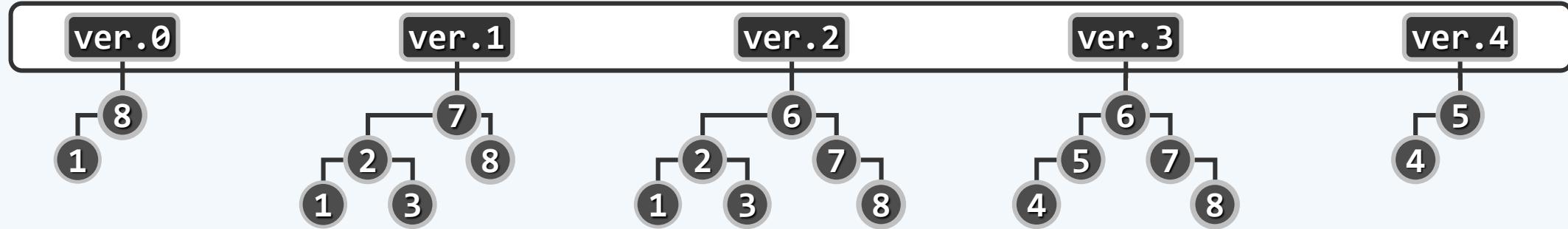
❖ 大量共享，少量更新：每个版本的新增复杂度，仅为 $\mathcal{O}(\log n)$



❖ 能否进一步提高，比如总体 $\mathcal{O}(n + h)$ 、单版本 $\mathcal{O}(1)$? 可以!

$\mathcal{O}(1)$ 重构

❖ 为此，就树形结构的**拓扑**而言，相邻版本之间的差异不能超过 $\mathcal{O}(1)$



❖ 很遗憾，AVL、Splay等BBST均不具备这一性质；须另辟蹊径...

java.util.TreeMap

```
import java.util.*;  
  
public class TestTreeMap {  
    public static void main( String[] args ) {  
        TreeMap scarborough = new TreeMap();  
  
        scarborough.put( "P", "parsley" );  
  
        scarborough.put( "S", "sage" );  
  
        scarborough.put( "R", "rosemary" );  
  
        scarborough.put( "T", "thyme" );  
  
        System.out.println( scarborough );  
    }  
}
```

8. 高级搜索树

(xa2) 红黑树：结构

崖前土黑没芝兰

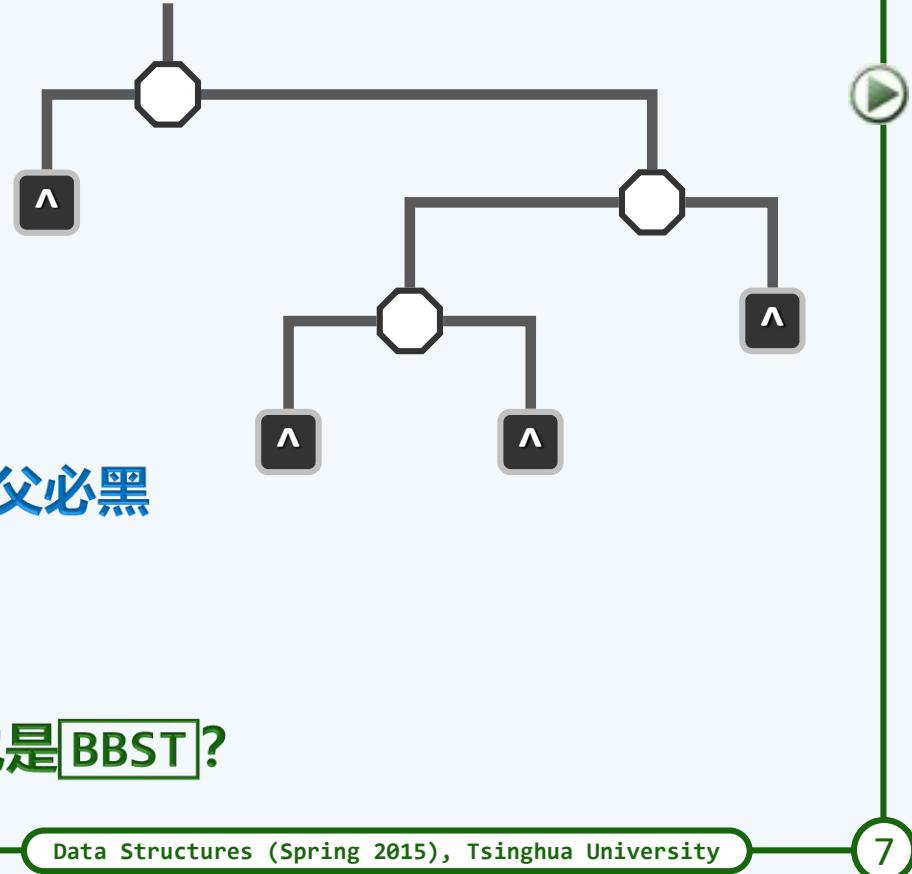
路畔泥红藤薜攀

邓俊辉

deng@tsinghua.edu.cn

红与黑

- ❖ 1972, R. Bayer, “symmetric binary B-tree”
- 1978, L. Guibas & R. Sedgewick, “red-black tree”
- 1982, H. Olivie, "half-balanced binary search tree"
- ❖ 由红、黑两类节点组成的BST //亦可给边染色
(统一增设外部节点NULL, 使之成为真二叉树)
 - (1) 树根: 必为黑色
 - (2) 外部节点: 均为黑色
 - (3) 其余节点: 若为红, 则只能有黑孩子 //红之子、之父必黑
 - (4) 外部节点到根: 途中黑节点数目相等 //黑深度
- ❖ 以上定义颇为费解, 有直观解释吗? 如此定义的BST, 也是BBST?



Acta Informatica 1, 290—306 (1972)
© by Springer-Verlag 1972

Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms*

R. Bayer

Received January 24, 1972

Summary. A class of binary trees is described for maintaining ordered sets of data. Random insertions, deletions, and retrievals of keys can be done in time proportional to $\log N$ where N is the cardinality of the data-set. Symmetric B-Trees are a modification of B-trees described previously by Bayer and McCreight. This class of trees properly contains the balanced trees.

Example

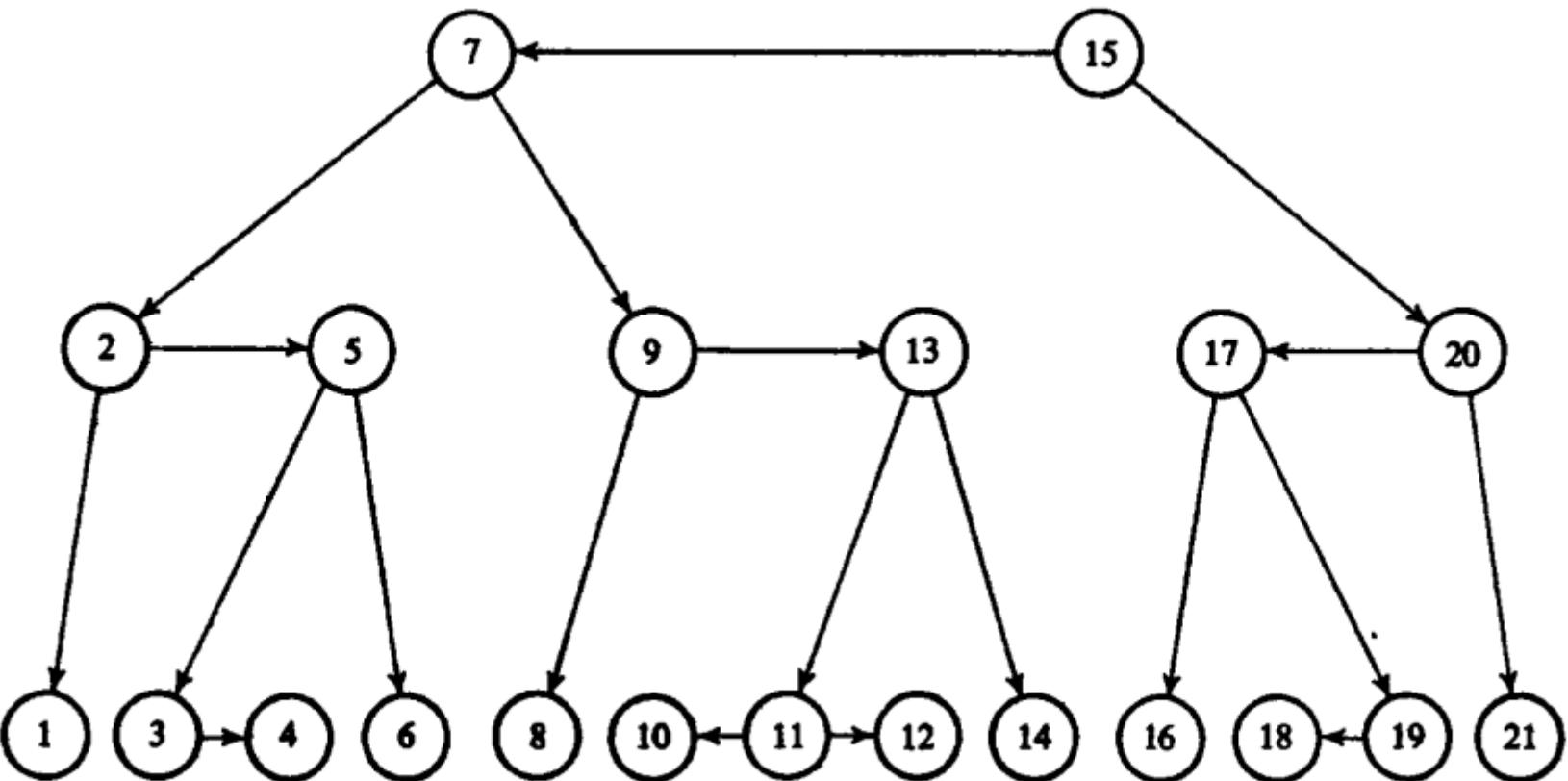
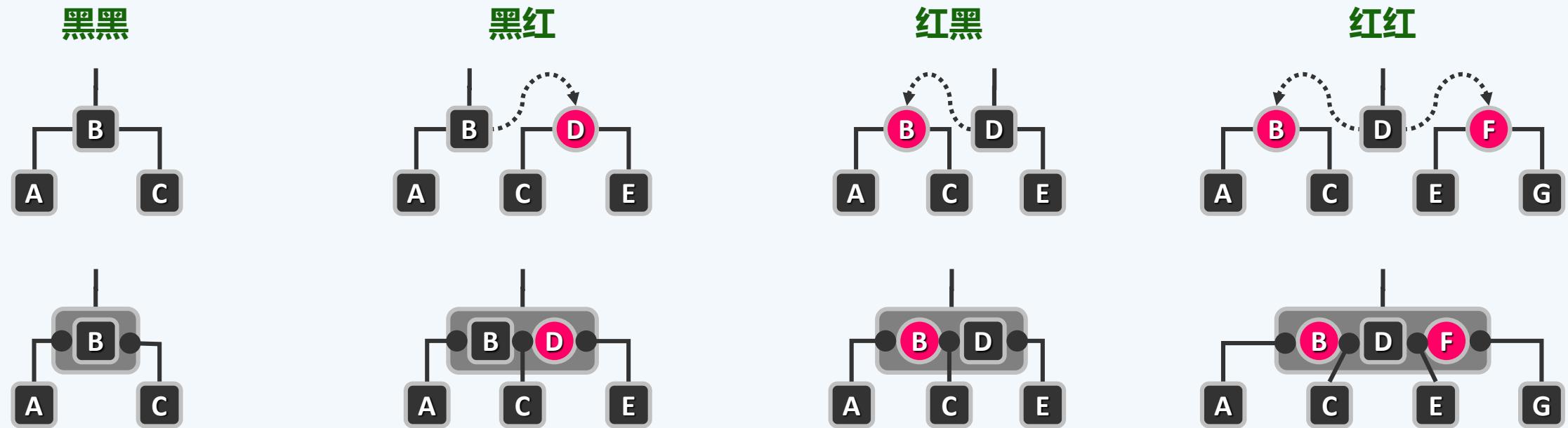


Fig. 1. Example of a symmetric binary B-tree

(2, 4)树 == 红黑树

- 提升各红节点，使之与其（黑）父亲等高——于是每棵红黑树，都对应于一棵(2, 4)-树
- 将黑节点与其红孩子视作（关键码并合并为）超级节点...
- 无非四种组合，分别对应于4阶B-树的一类内部节点 //反过来呢？



❖ 由等价性，既然B-树是平衡的，红黑树自然也应是

//更严谨地...

❖ 定理：包含n个内部节点的红黑树T，高度 $h = \Theta(\log n)$

// $n + 1$ 个外部节点

$$\log_2(n + 1) \leq h \leq 2 * \log_2(n + 1)$$

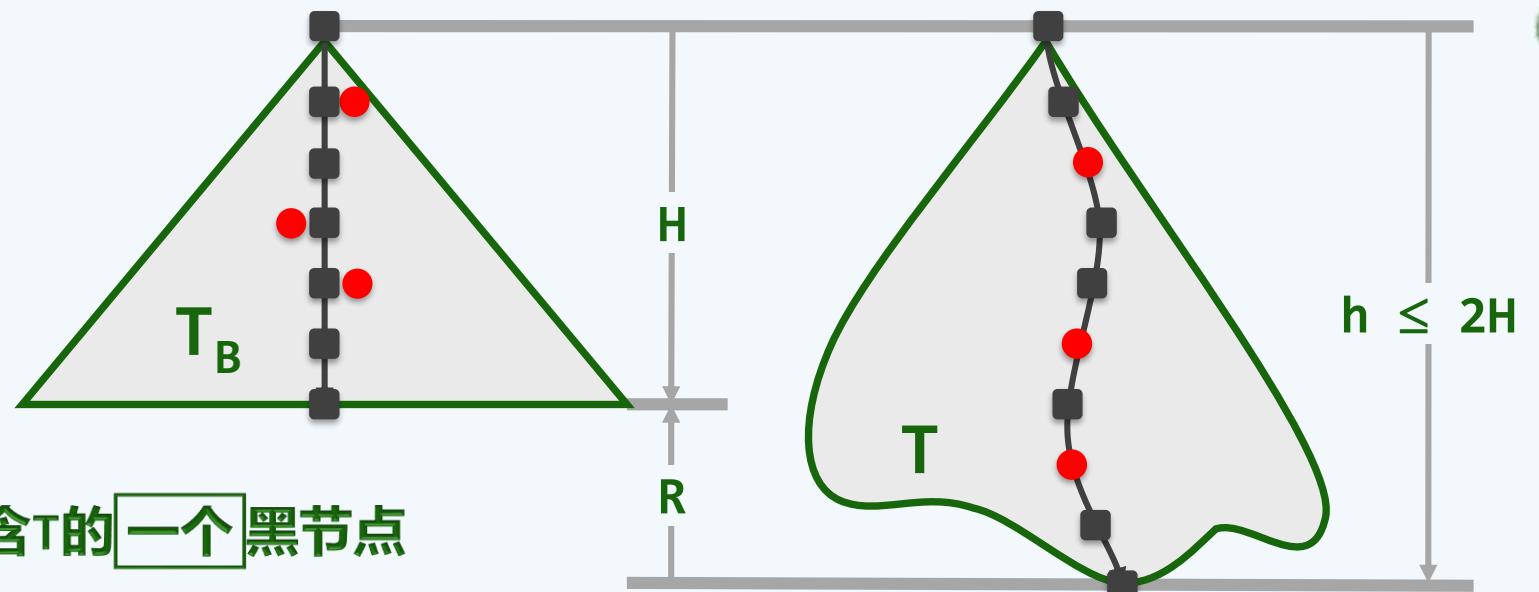
❖ 若：T高度为h，黑高度为H

$$\text{则: } h = R + H \leq 2H$$

❖ 若T所对应的B-树为T_B

则H即是T_B的高度

❖ T_B的每个节点，包含且仅包含T的一个黑节点



❖ 于是， $H \leq \log_{[4/2]} \frac{n+1}{2} + 1 \leq \log_2(n + 1)$

RedBlack

```
❖ template <typename T> class RedBlack : public BST<T> { //红黑树
public:    //BST::search()等其余接口可直接沿用
            BinNodePosi(T) insert( const T & e ); //插入 (重写)
            bool remove( const T & e ); //删除 (重写)
protected: void solveDoubleRed( BinNodePosi(T) x ); //双红修正
            void solveDoubleBlack( BinNodePosi(T) x ); //双黑修正
            int updateHeight( BinNodePosi(T) x ); //更新节点x的高度
};

❖ template <typename T> int RedBlack<T>::updateHeight( BinNodePosi(T) x ) {
    x->height = max( stature( x->lC ), stature( x->rC ) );
    if ( IsBlack( x ) ) x->height++; return x->height; //只计黑节点
}
```

8. 高级搜索树

(xa3) 红黑树：插入

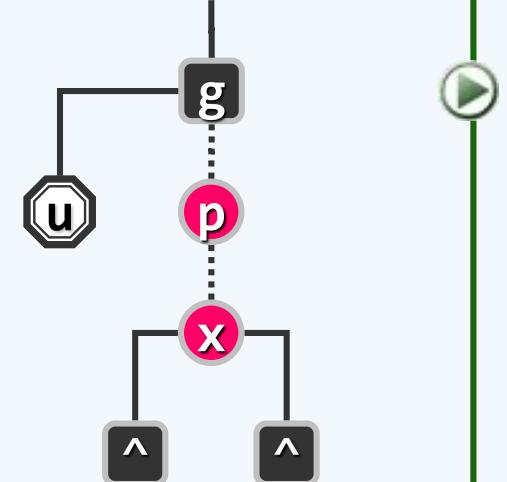
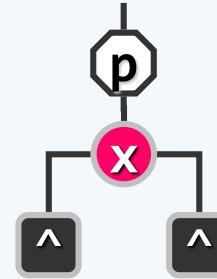
莫赤匪狐，莫黑匪乌
惠而好我，携手同车

邓俊辉

deng@tsinghua.edu.cn

算法

- ❖ 现拟插入关键码 e //不妨设 T 中本不含 e
- ❖ 按 BST 的常规算法，插入之 // $x = \text{insert}(e)$ 必为末端节点
不妨设 x 的父亲 $p = x \rightarrow \text{parent}$ 存在 // 否则，即平凡的首次插入
- ❖ 将 x 染红 (除非它是根) // $x \rightarrow \text{color} = \text{isRoot}(x) ? B : R$
条件 1 + 2 + 4 依然满足；但 3 不见得，有可能...
- ❖ 双红 double-red // $p \rightarrow \text{color} == x \rightarrow \text{color} == R$
- ❖ 考查： x 的祖父 $g = p \rightarrow \text{parent}$ // $g != \text{null} \&& g \rightarrow \text{color} == B$
 p 的兄弟 $u = p == g \rightarrow \text{lc} ? g \rightarrow \text{rc} : g \rightarrow \text{lc}$ // 即 x 的叔父
- ❖ 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 -1  
    x = new BinNode<T>( e, _hot, NULL, NULL, -1 ); _size++;  
    // 如有必要, 需做双红修正  
    solveDoubleRed( x );  
    // 返回插入的节点  
    return x ? x : _hot->parent;  
} // 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 ... */ }  
}
```

❖ 此时：

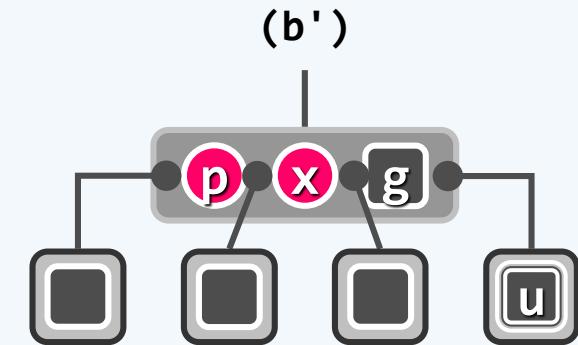
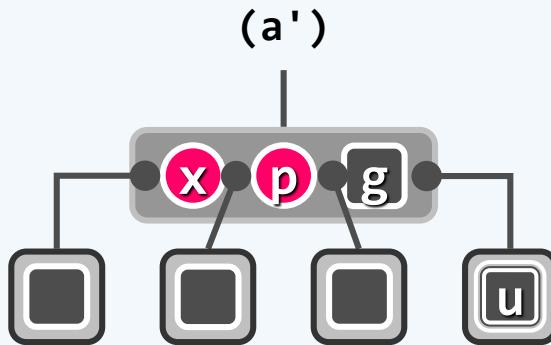
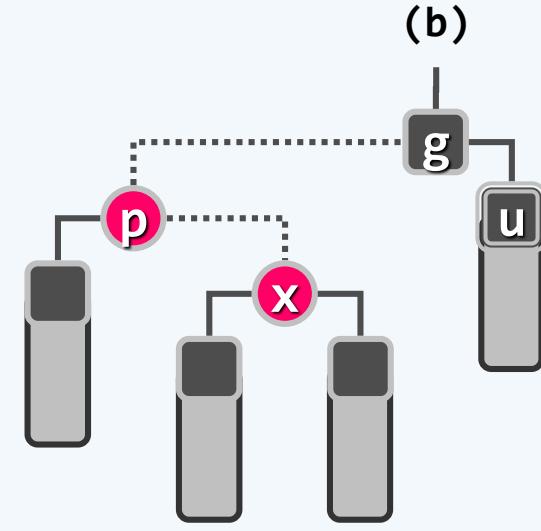
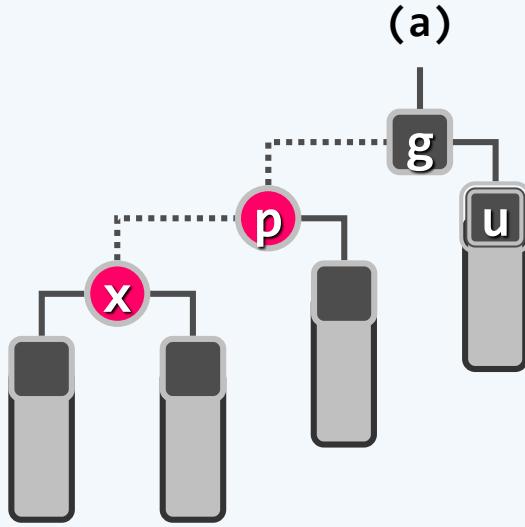
x 、 p 、 g 的四个孩子

(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: $u \rightarrow \text{color} == \text{B}$

- 参照AVL树算法，做局部3+4重构

将节点x、p、g及其四棵子树，按中序组合为：

$T_0 < a < T_1 < b < T_2 < c < T_3$

- 染色：b转黑，a或c转红

❖ 从B-树的角度，如何理解这一情况？

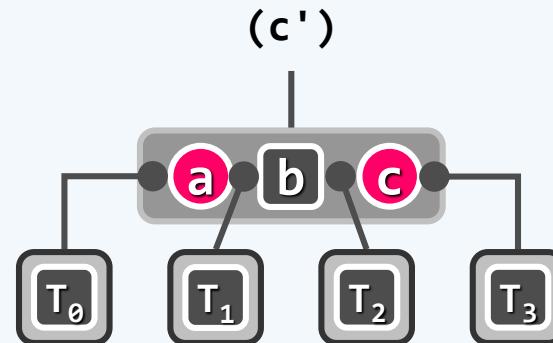
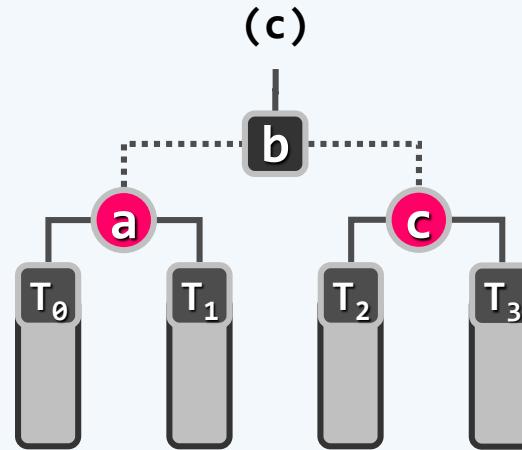
- 调整前之所以非法，是因为

在某个三叉节点中插入红关键码，使得

原黑关键码不再居中 // RRB或BRR，出现相邻的红关键码

- 调整之后的效果相当于 // B-树的拓扑结构不变，但

在新的四叉节点中，三个关键码的颜色改为RBR



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: $u \rightarrow \text{color} == R$

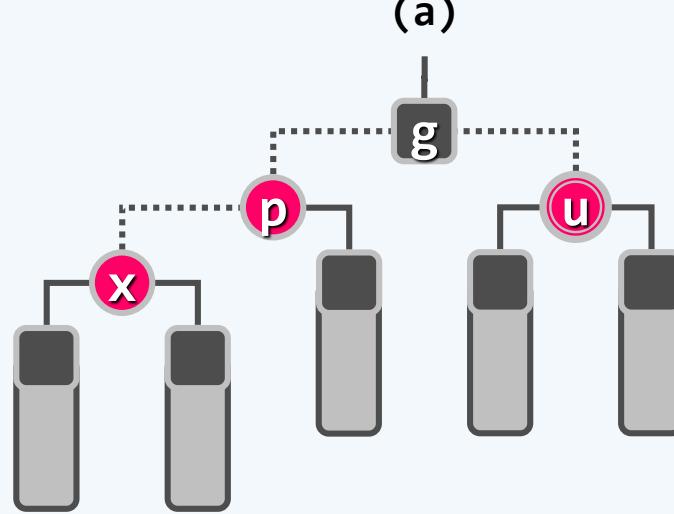
❖ 在B-树中，等效于

超级节点发生上溢

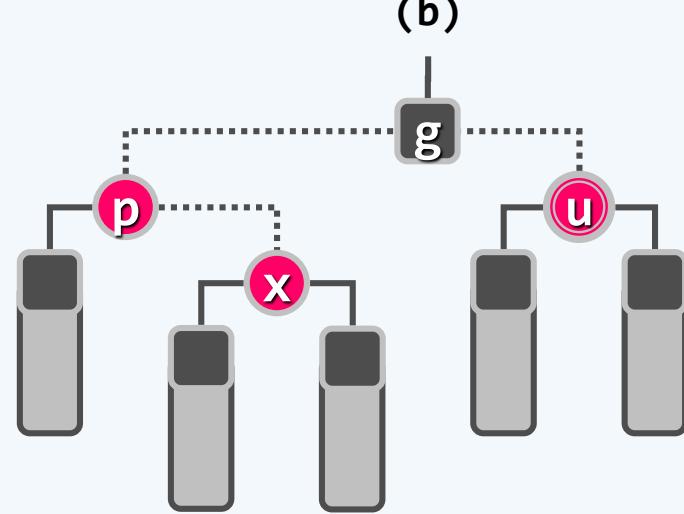
//另两种对称情况

//请自行补充

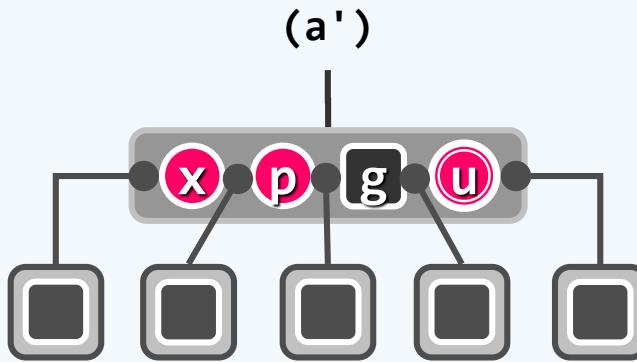
(a)



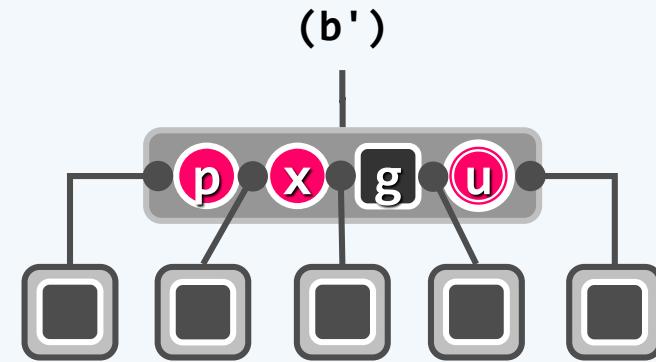
(b)



(a')



(b')



RR-2: $u \rightarrow \text{color} == R$

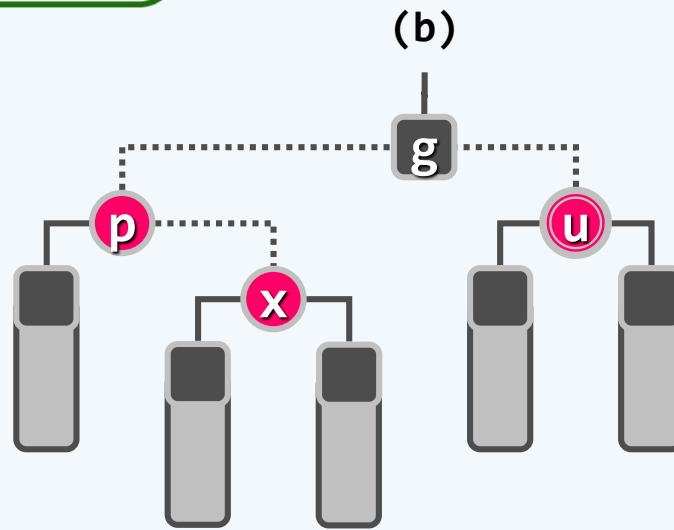
❖ p与u转黑，g转红

❖ 在B-树中，等效于

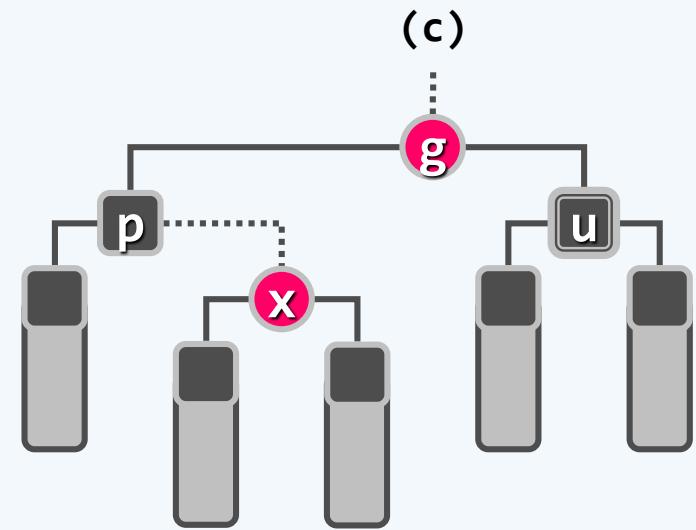
节点分裂

关键码g上升一层

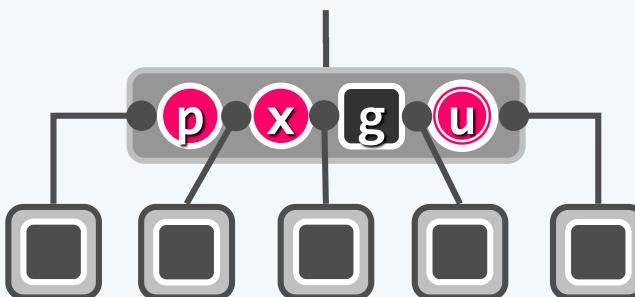
(b)



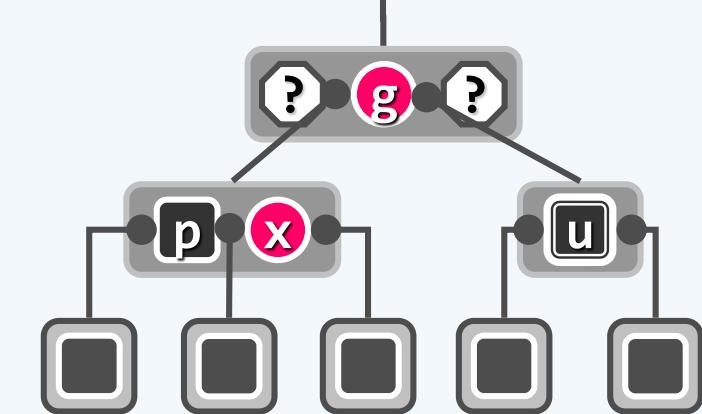
(c)



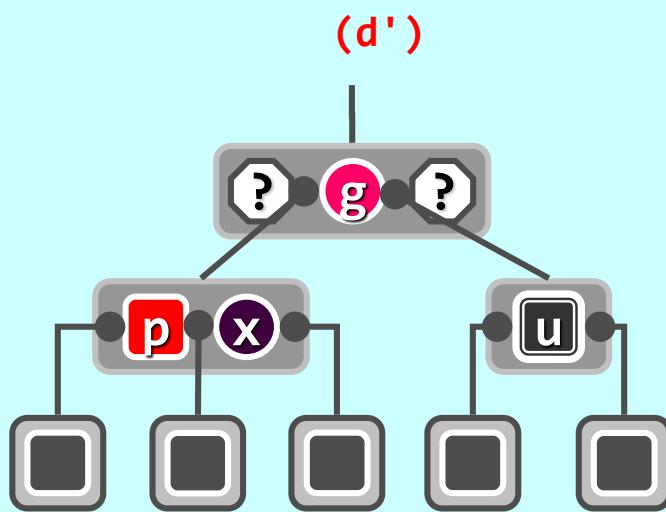
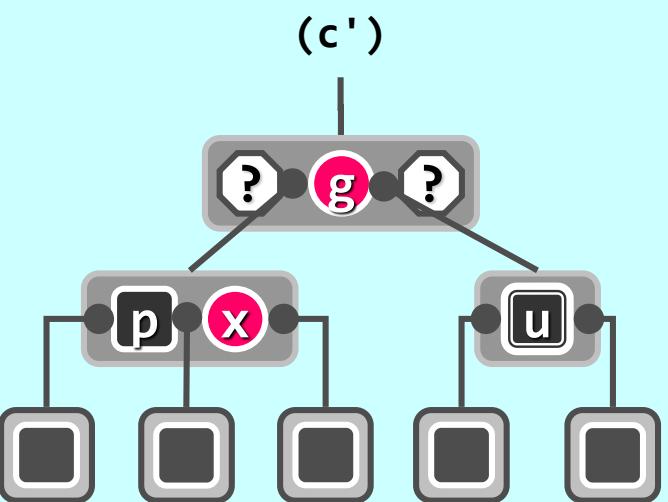
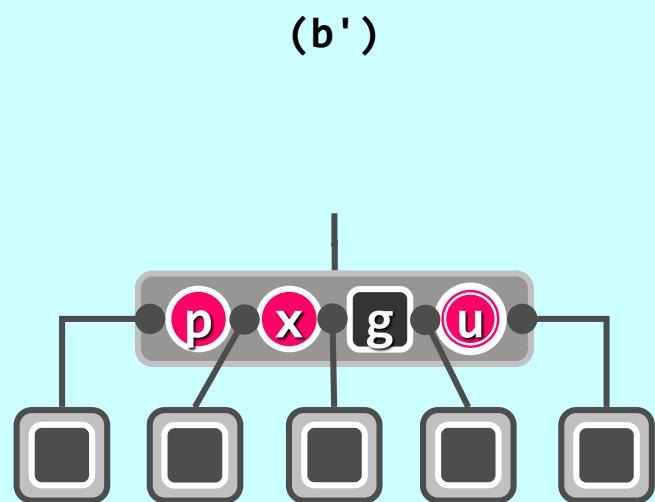
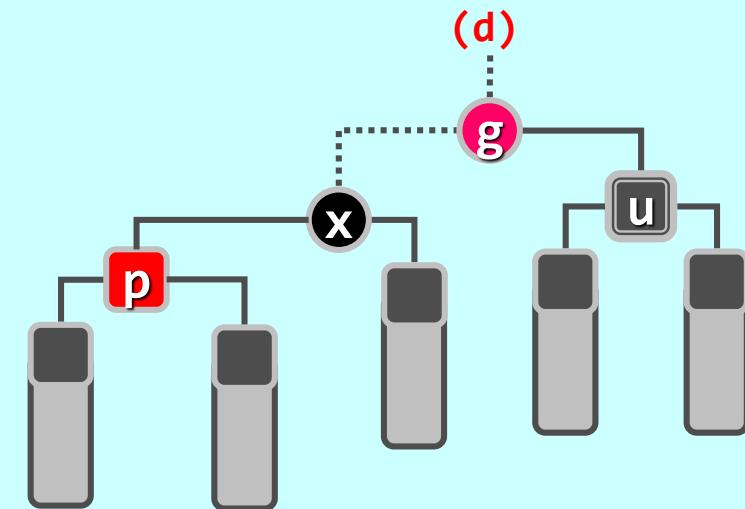
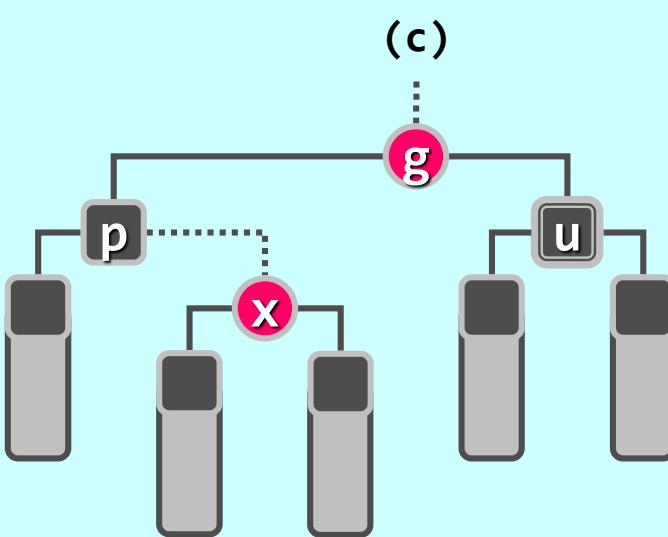
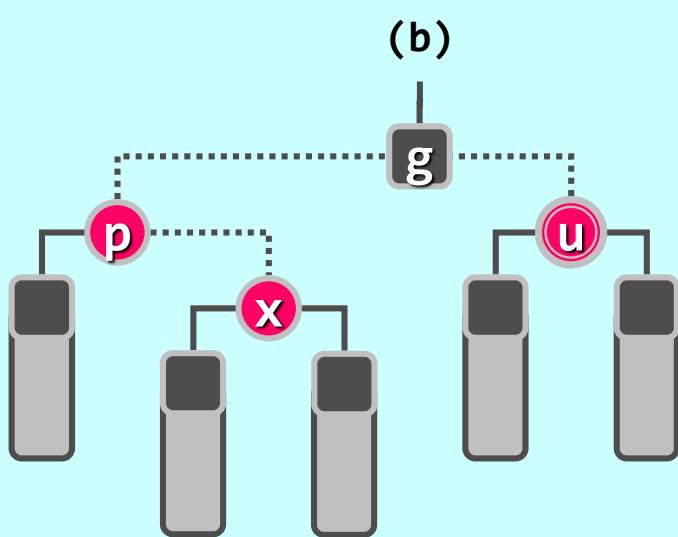
(b')



(c')



RR-2: $u \rightarrow \text{color} == R$ 另外一种情况?



❖ 既然是分裂，也应有可能继续向上传递

亦即， g 与 $\text{parent}(g)$ 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

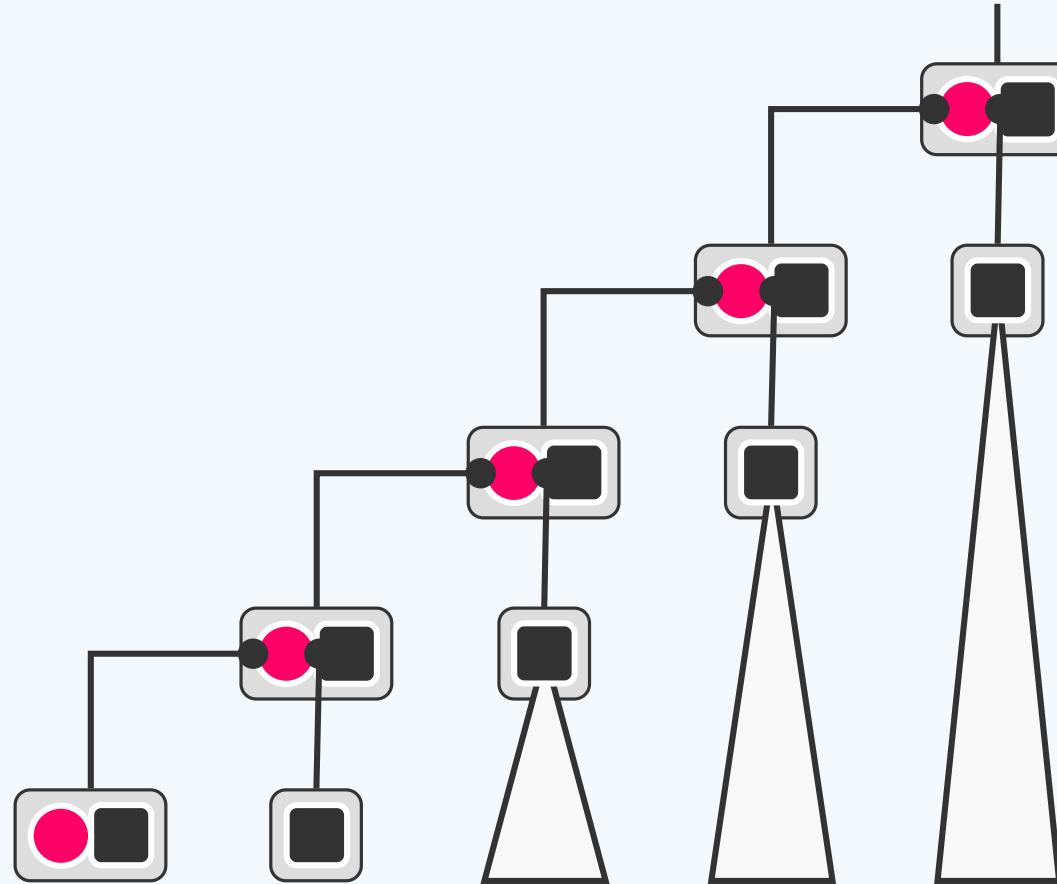
❖ 直到所有条件满足（即不再双红）

或者抵达树根

❖ g 若果真到达树根，则

1. 强行将 g 转为黑色

2. 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 (或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  
        if ( !IsRoot( *g ) ) g->color = RB_RED; //g若非根则转红  
        solveDoubleRed( g ); //继续调整g (类似于尾递归, 可优化)  
    }  
}
```

双红修正：复杂度

❖ 重构、染色均属常数时间的局部操作

故只需统计其总次数

❖ 红黑树的每一次插入操作

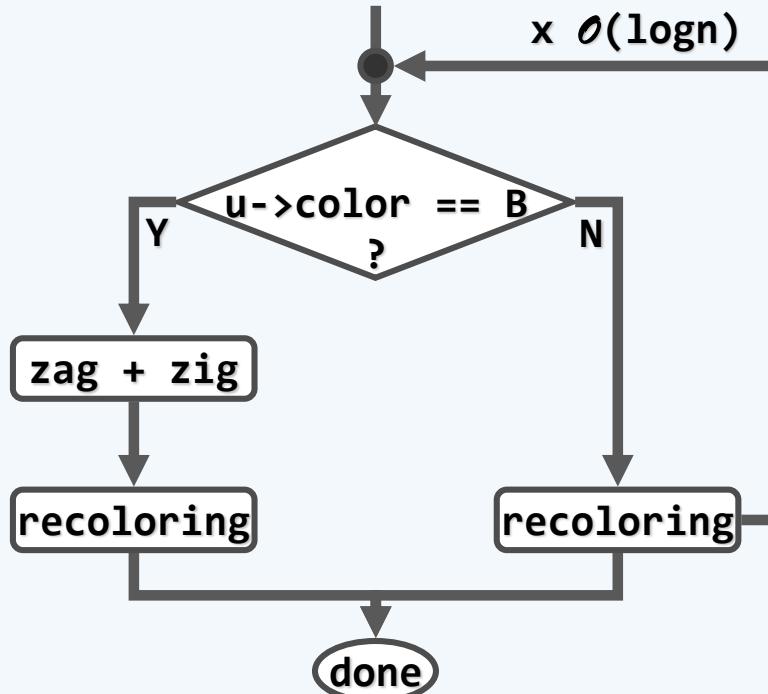
都可在 $\mathcal{O}(\log n)$ 时间内完成

❖ 其中至多做：

1. $\mathcal{O}(\log n)$ 次 节点染色

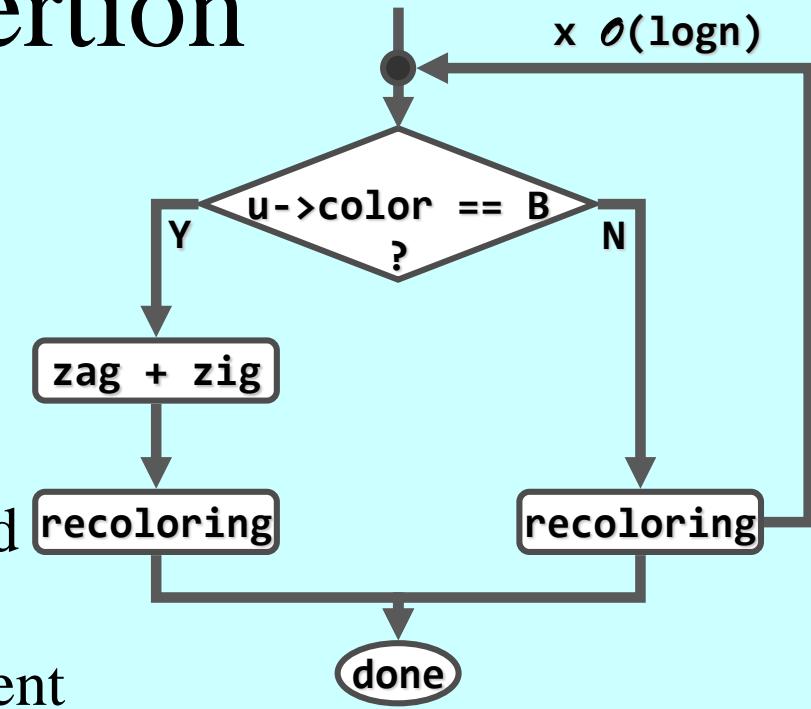
2. 一次“3+4”重构

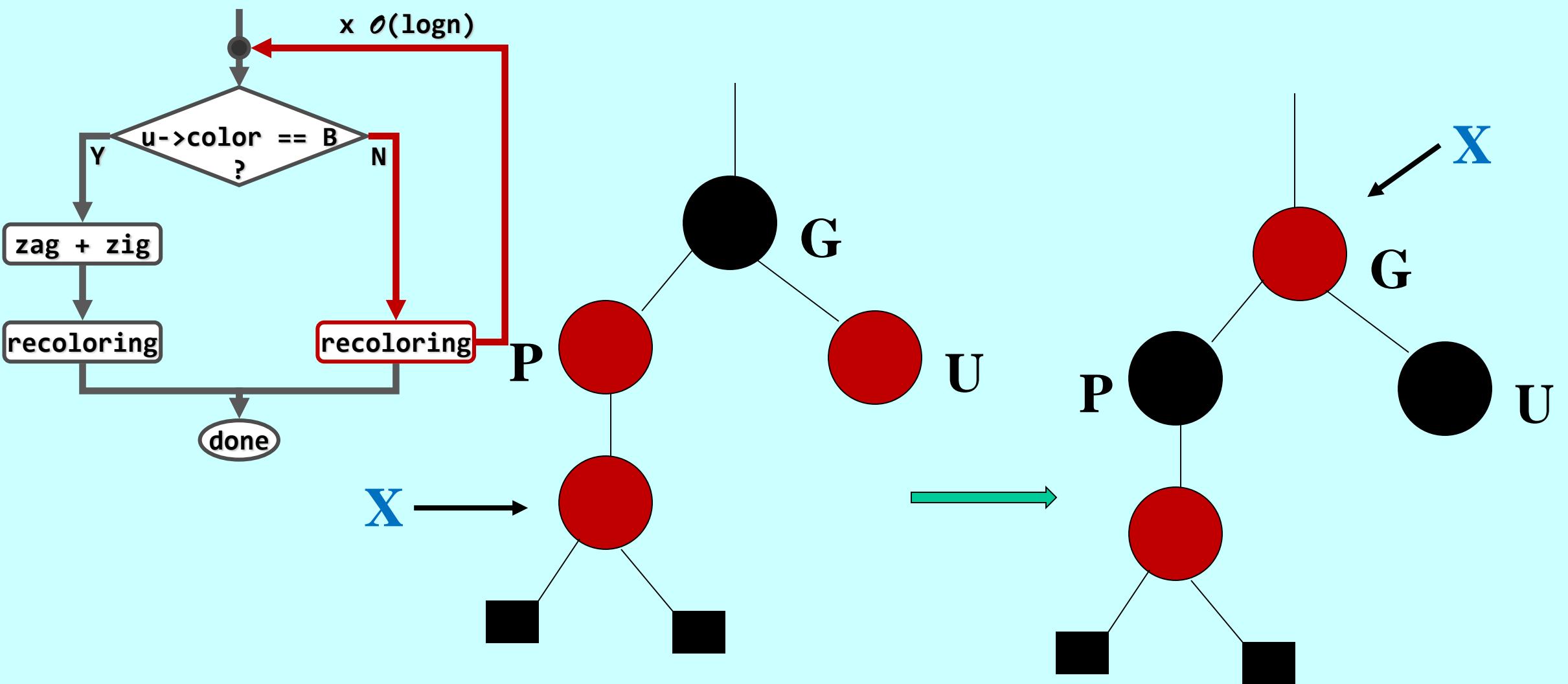
情况	旋转次数	染色次数	此后
u为黑	1~2	2	调整随即完成
u为红	0	3	可能再次双红 但必上升两层



Review Bottom Up Insertion

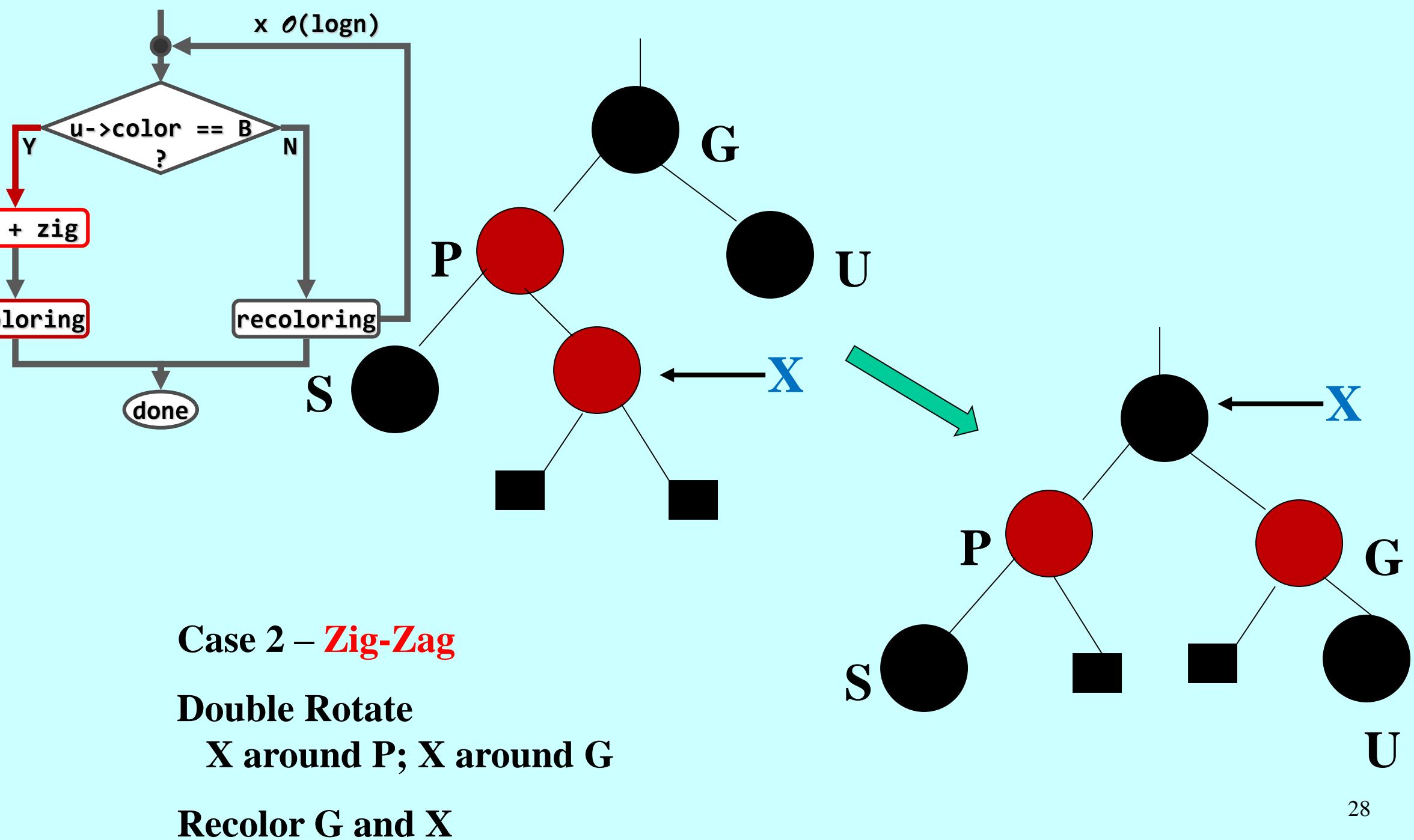
- Insert node; Color it **Red**; X is pointer to it
- Cases
 - 0: X is the root -- color it Black
 - 1: Both parent and **uncle** are Red -- color parent and uncle Black, color grandparent Red. Point X to **grandparent** and check new situation.
 - 2 (**zig-zag**): Parent is Red, but uncle is Black. X and its parent are opposite type children -- color grandparent Red, color X Black, rotate left(right) on parent, rotate right(left) on grandparent
 - 3 (**zig-zig**): Parent is Red, but uncle is Black. X and its parent are both left (right) children -- color parent Black, color grandparent Red, rotate right(left) on grandparent

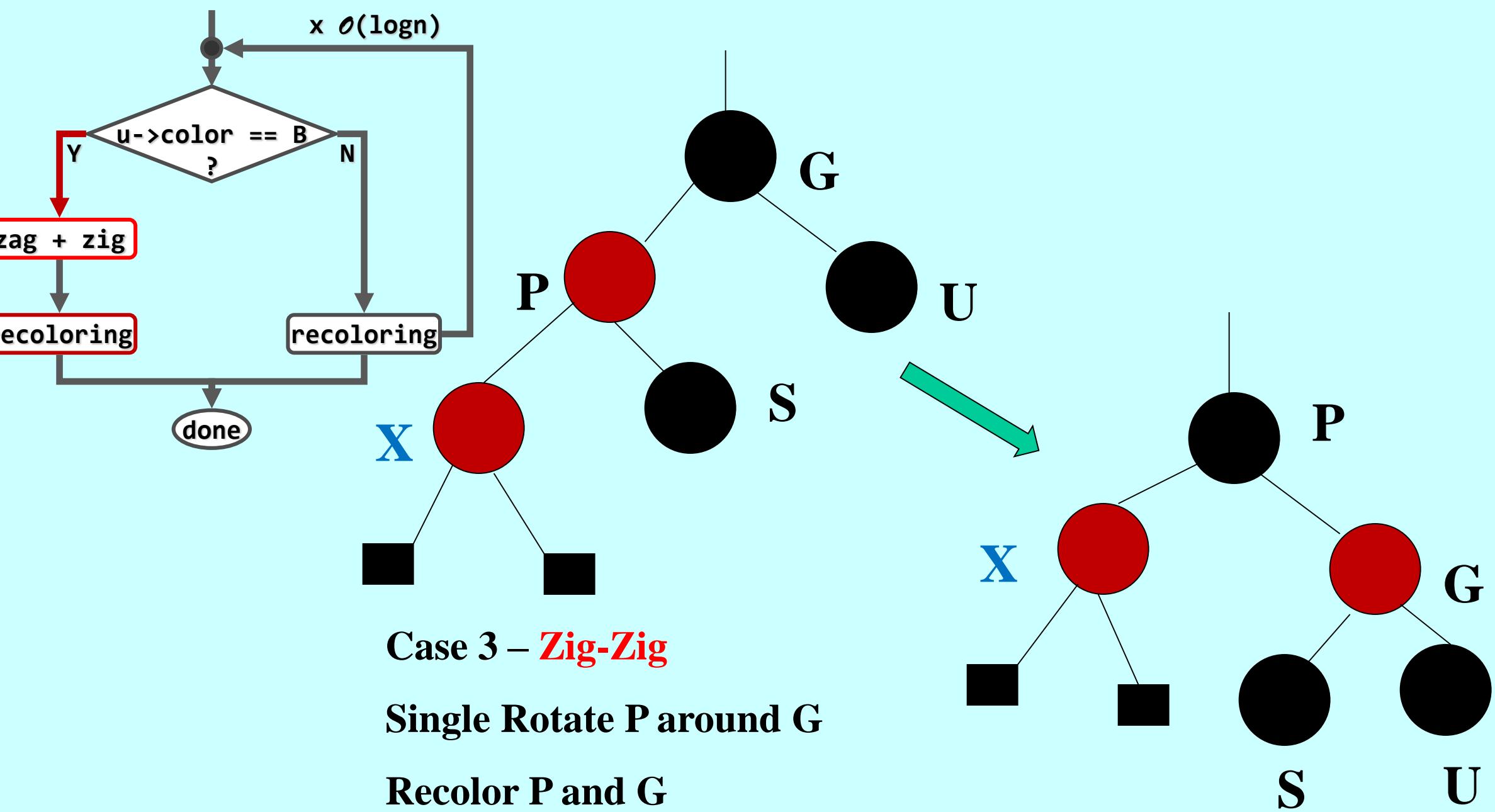




Case 1 – U is Red

Just Recolor and move up

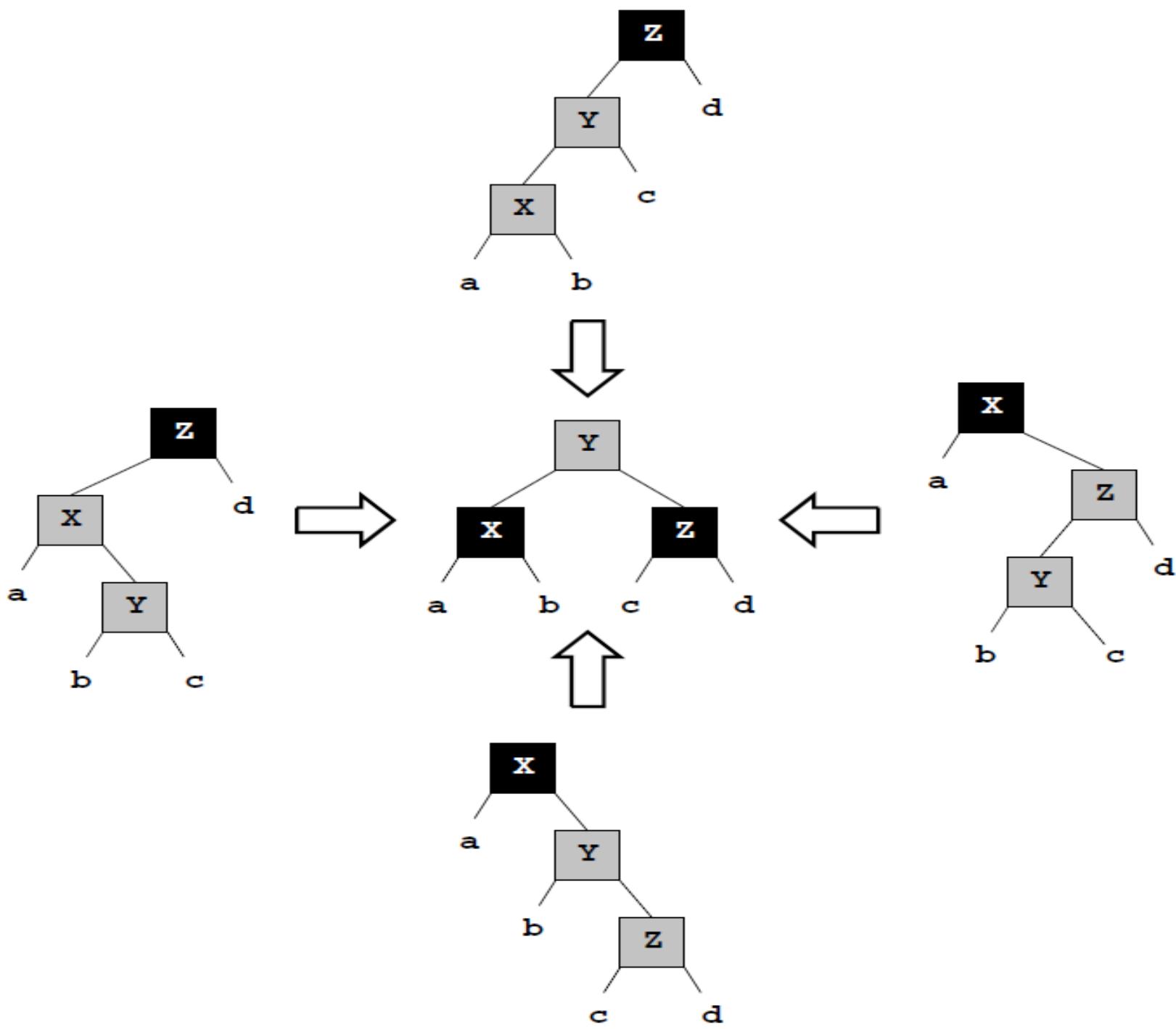




Case 3 – **Zig-Zig**

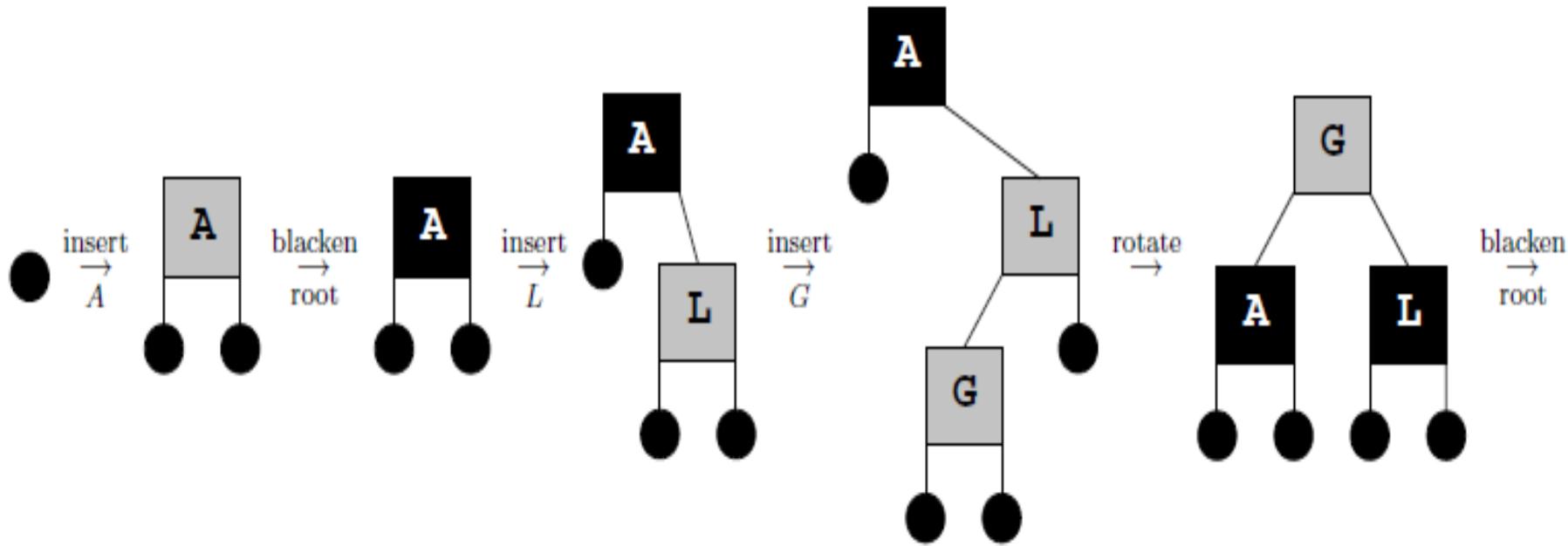
Single Rotate P around G

Recolor P and G

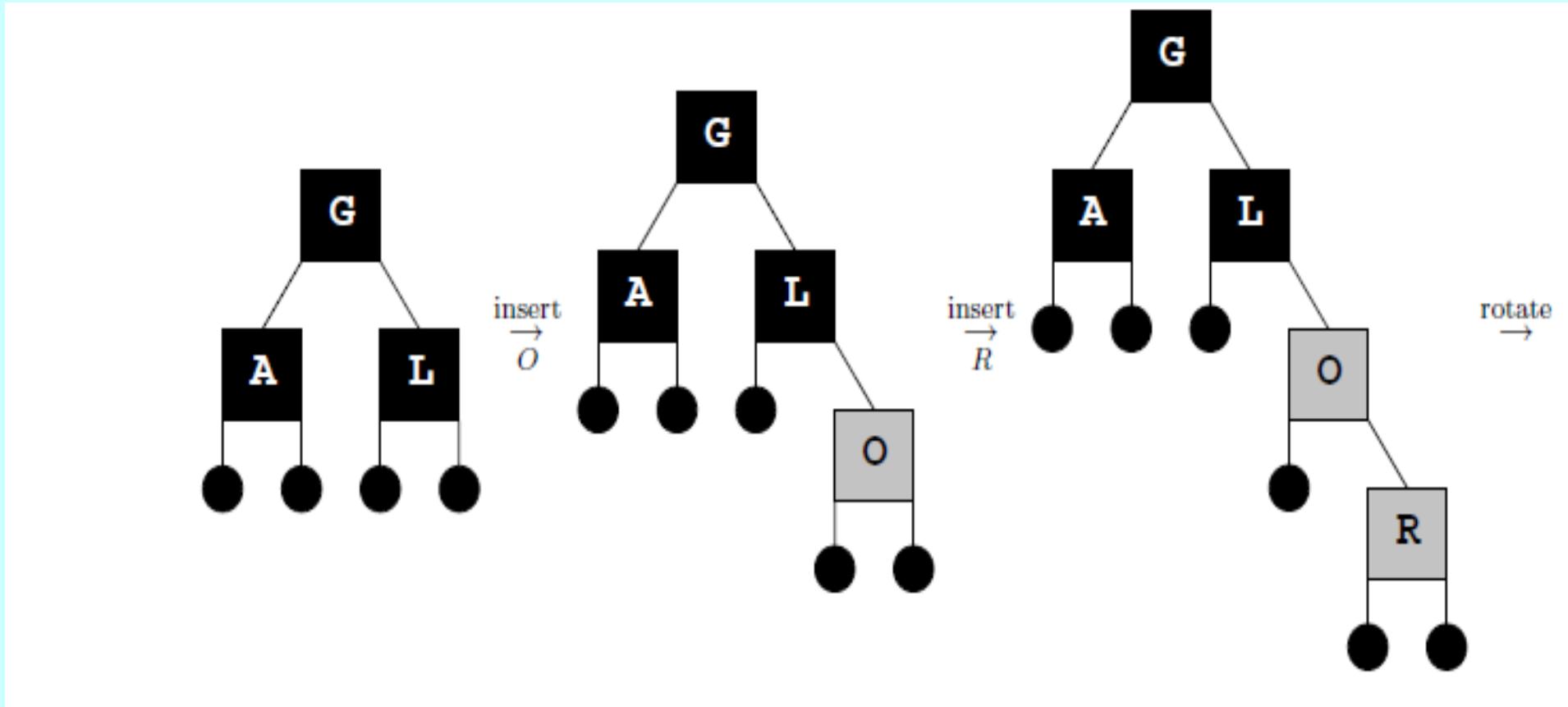


Bottom Up Insertion Example

Insert the letters A L G O R I T H M in order into a red-black tree.

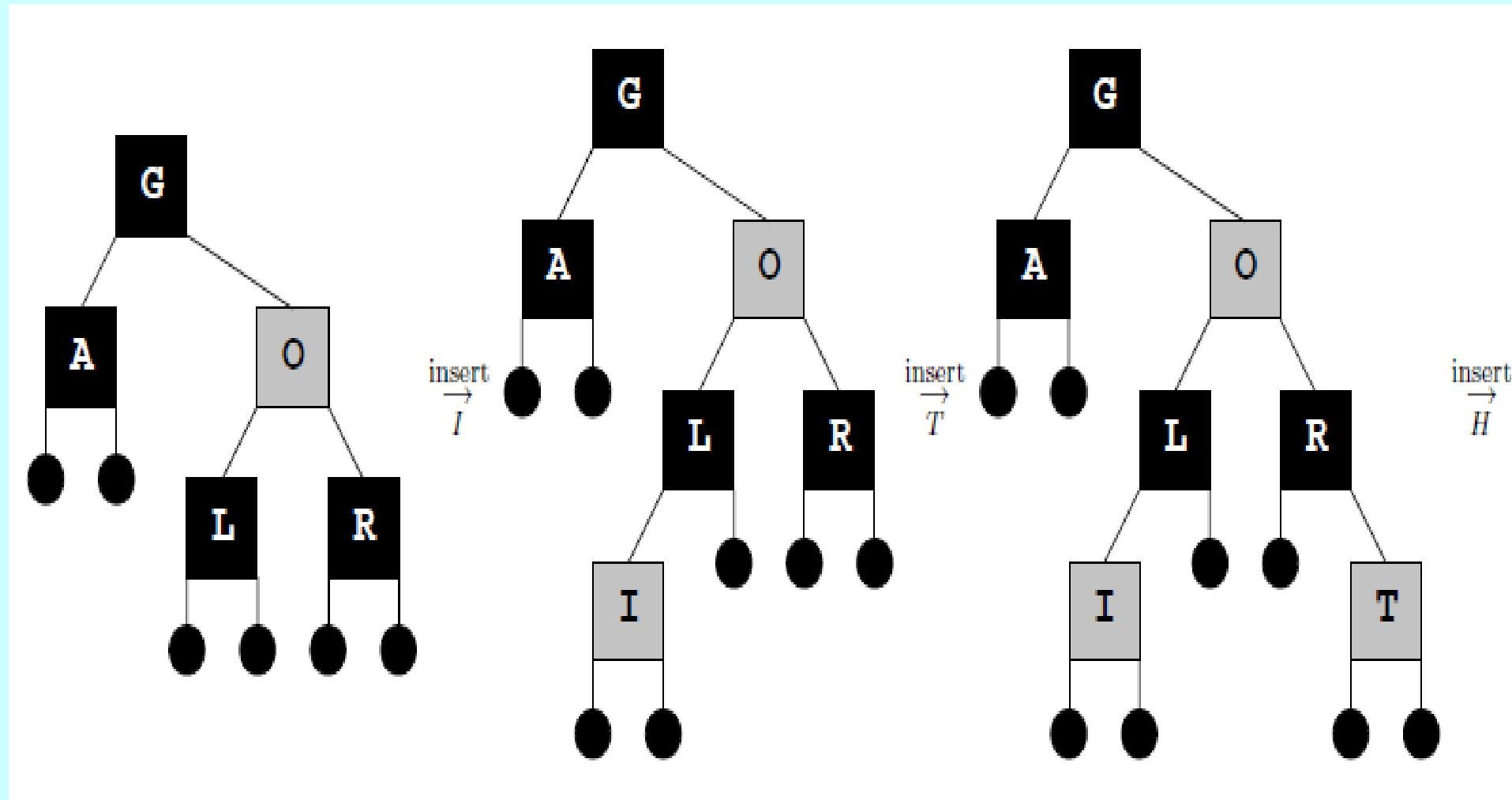


Bottom Up Insertion Example



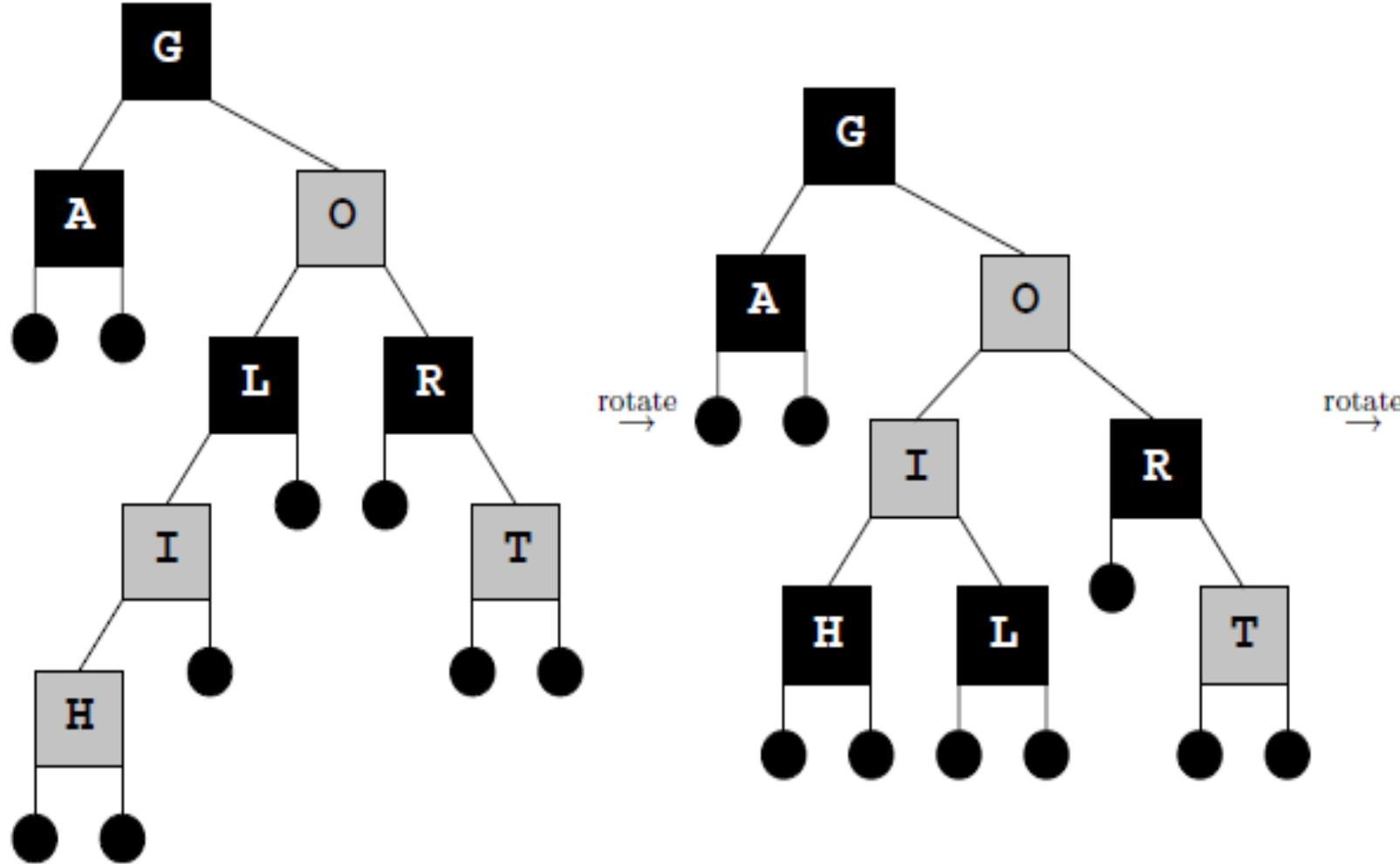
Example: Insert A L G O R I T H M in order into a red-black tree

Bottom Up Insertion Example



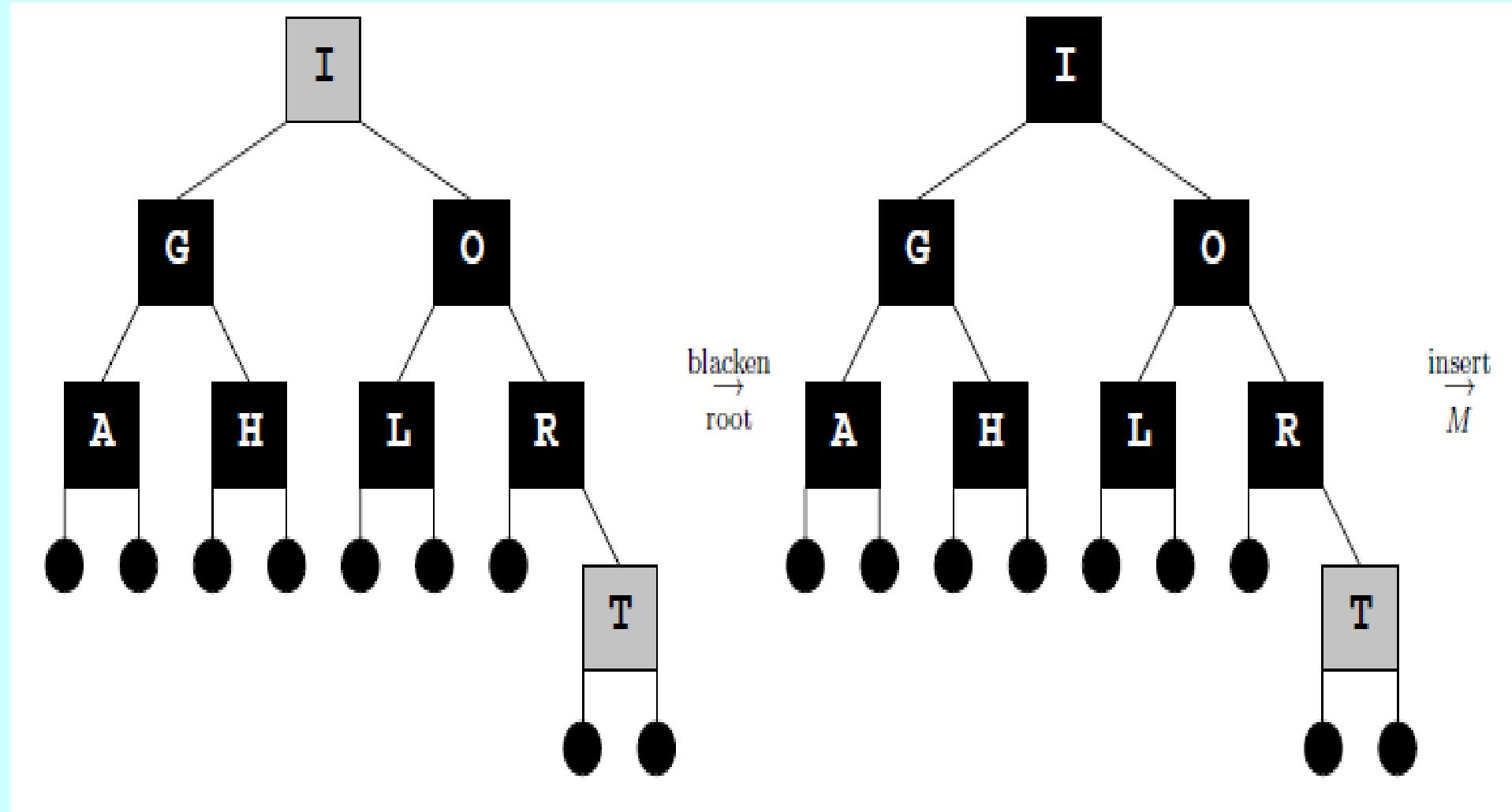
Example: Insert A L G O R I T H M in order into a red-black tree

Bottom Up Insertion Example



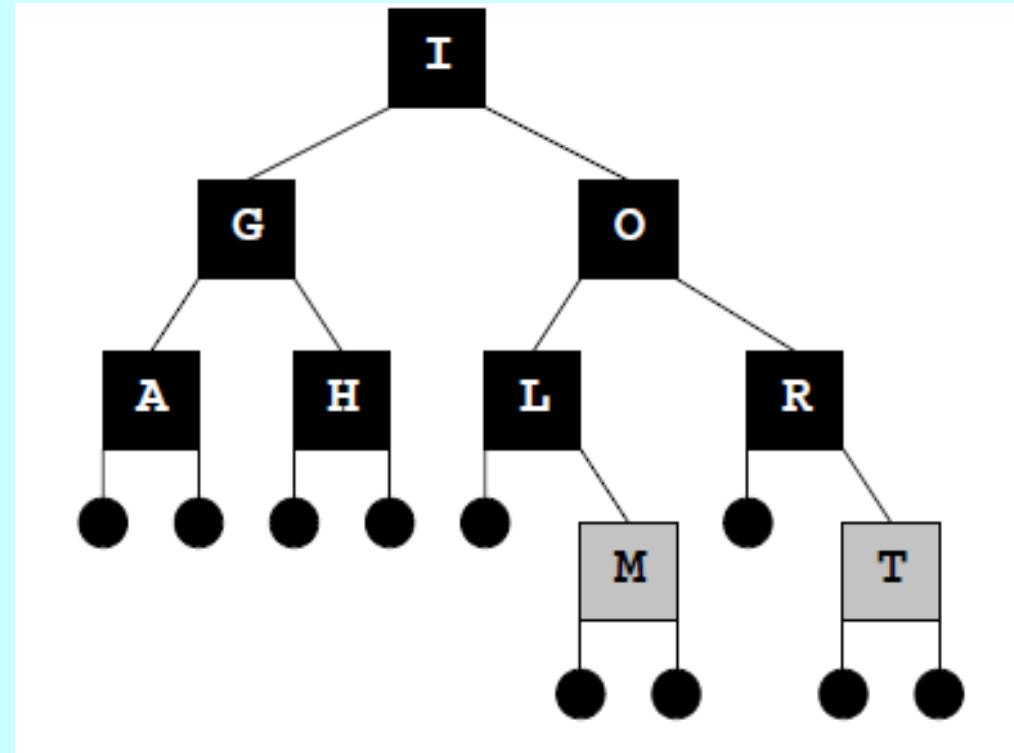
Example: Insert A L G O R I T H M in order into a red-black tree

Bottom Up Insertion Example



Example: Insert A L G O R I T H M in order into a red-black tree

Bottom Up Insertion Example



Example: Insert A L G O R I T H M in order into a red-black tree

Review of Bottom-Up Insertion

- In Bottom-Up insertion, “ordinary” BST insertion was used, followed by correction of the tree on the way back up to the root
- This is most easily done recursively
 - Insert winds up the recursion on the way down the tree to the insertion point
 - Fixing the tree occurs as the recursion unwinds

8. 高级搜索树

(xa4) 红黑树：删除

邓俊辉

变白以为黑兮，倒上以为下

deng@tsinghua.edu.cn

算法

❖ 首先按照BST常规算法，执行：

`r = removeAt(x, _hot)`

❖ `x`由孩子`r`接替

//另一孩子记作`w`（即黑的NULL）

❖ 条件①和②依然满足

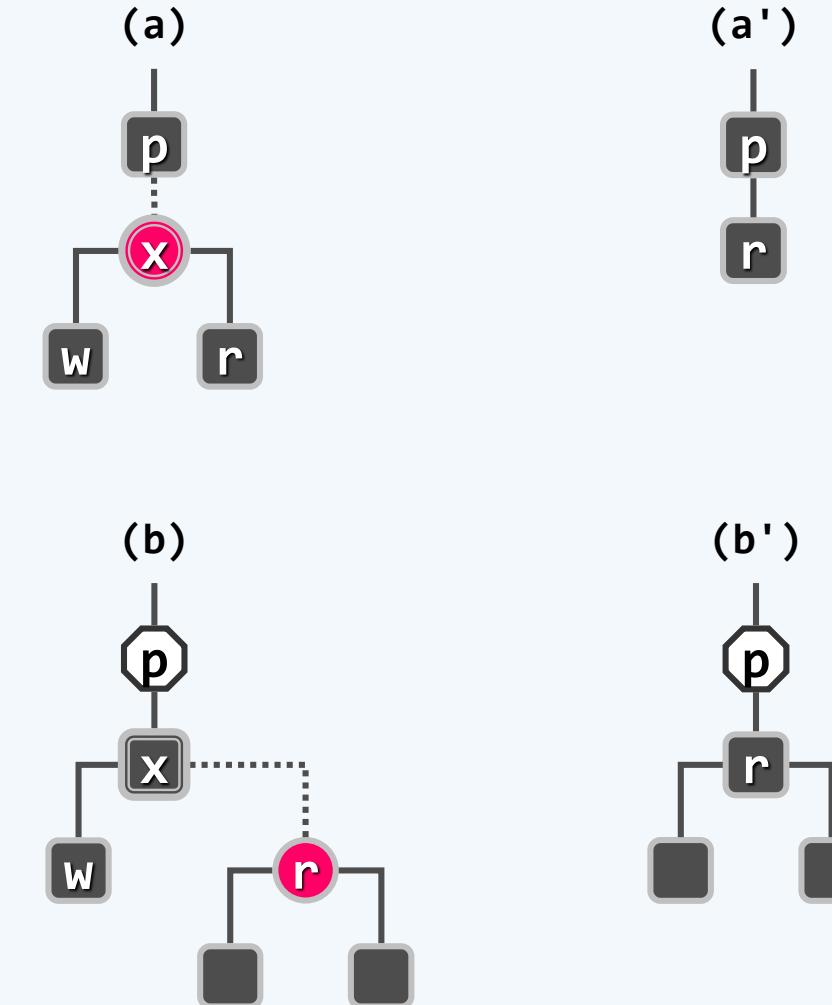
但③和④不见得

//在原树中，考查`x`与`r`...

❖ 若二者之一为红

则③和④不难满足

//删除遂告完成！



算法

❖ 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

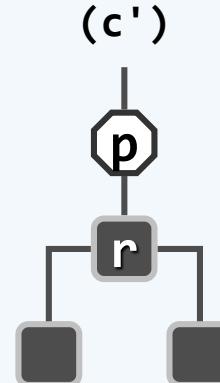
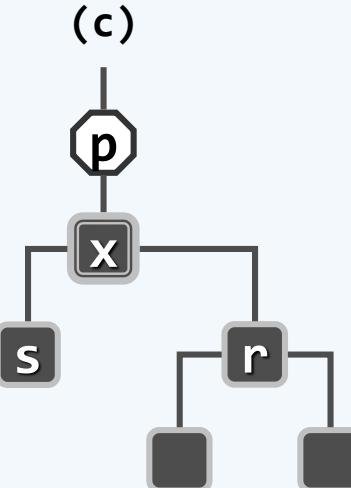
原B-树中 x 所属节点 下溢

❖ 在新树中，考查

r 的父亲 $p = r \rightarrow \text{parent}$ // 亦即原树中 x 的父亲

r 的兄弟 $s = [r == p \rightarrow \text{lc}] ? p \rightarrow \text{rc} : p \rightarrow \text{lc}$

❖ 以下分四种情况处理...



实现

```
❖ 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

❖ // 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;
```

```
}
```

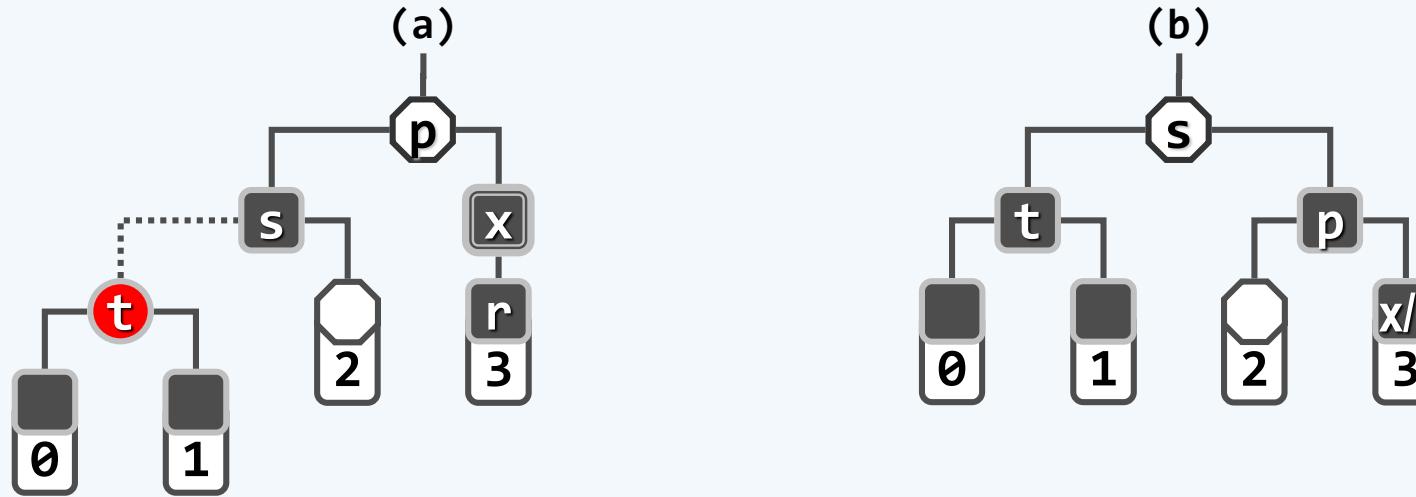
双黑修正

```
❖ 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; //以下将t取作s的红孩子  
        if ( HasLChild( *s ) && IsRed( s->lc ) ) t = s->lc;  
        else if ( HasRChild( *s ) && IsRed( s->rc ) ) t = s->rc;  
        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**、**s**、**p**重命名为**a**、**b**、**c**

r保持黑; **a**和**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**

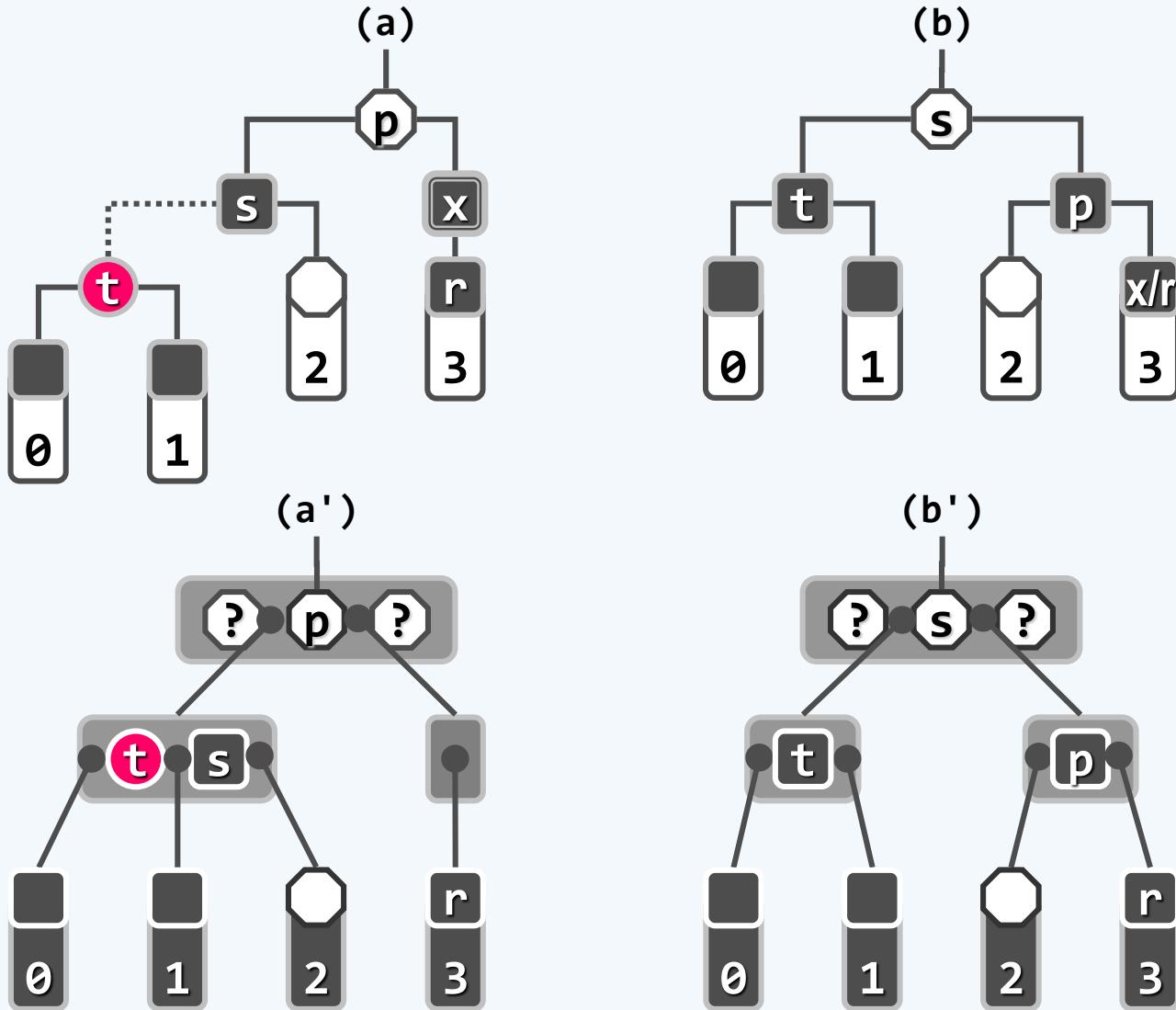
❖ 通过关键码的**旋转**

消除超级节点的**下溢**

❖ **问号节点**

可同时存在

颜色不定

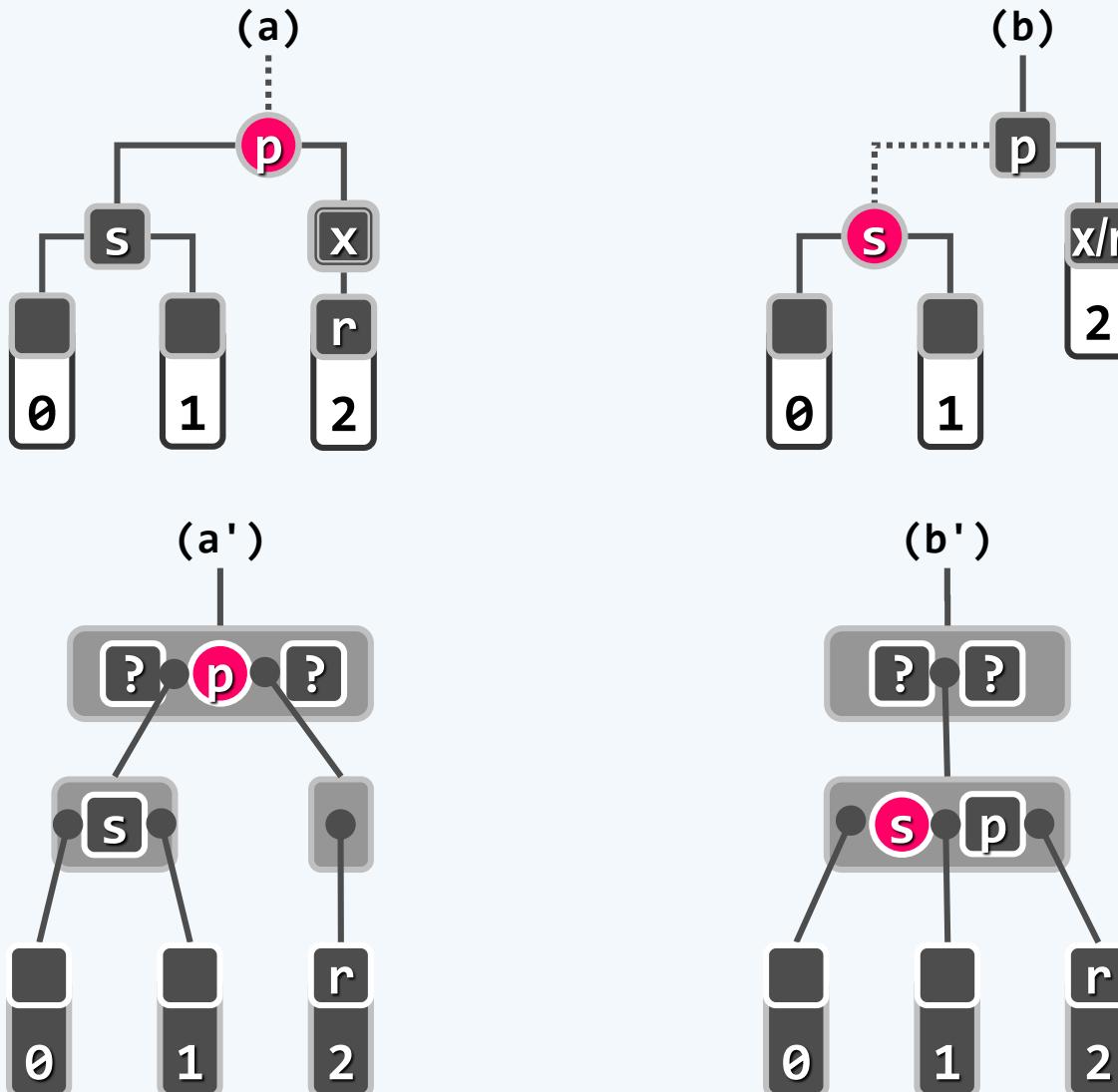


BB-1: 实现

```
❖ 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->lChild->color = RB_BLACK; //新子树之左子染黑  
        if ( HasRChild( *b ) ) b->rChild->color = RB_BLACK; //新子树之右子染黑  
        updateHeight( b->lC ); 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 ... */ }
```

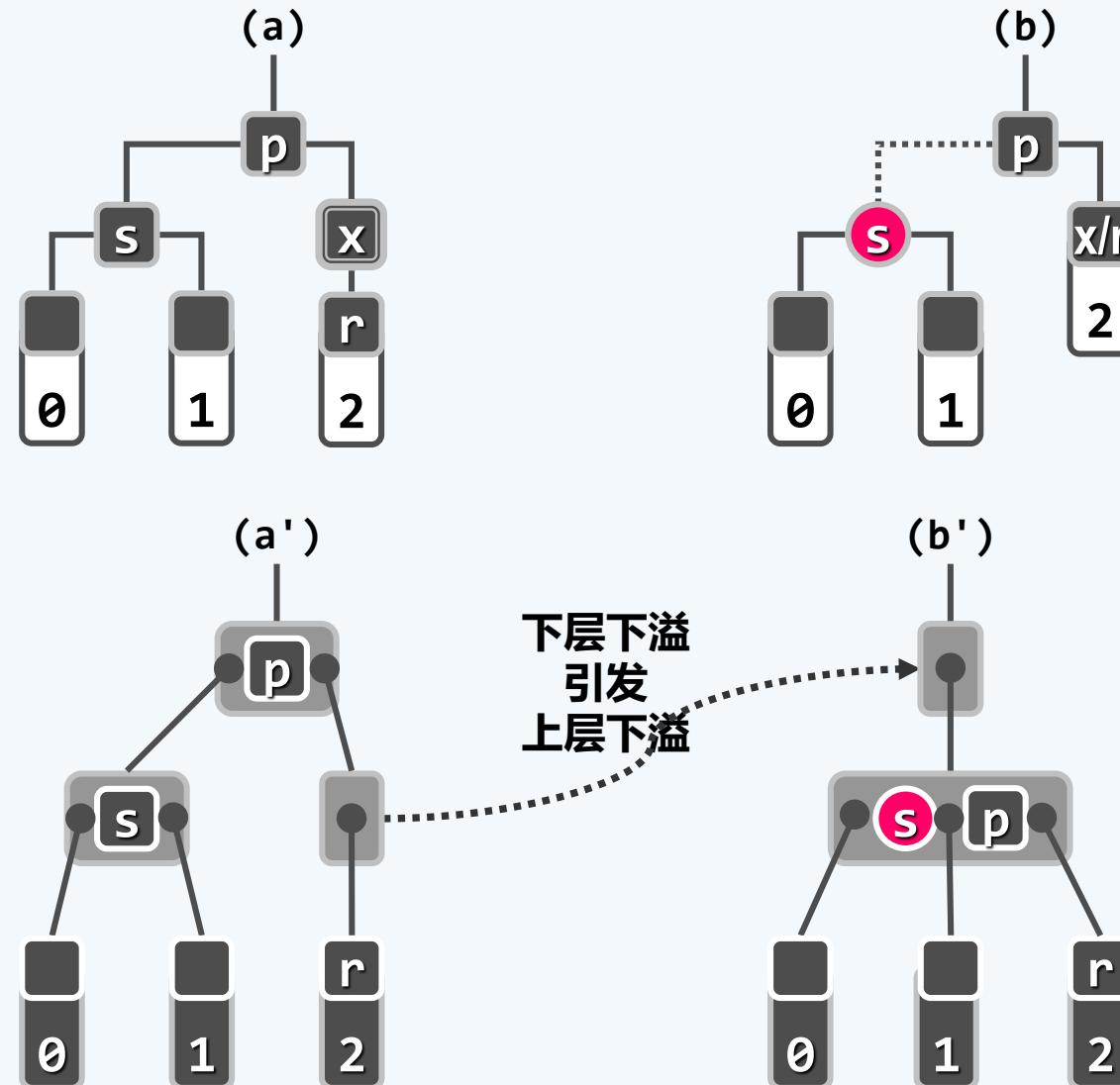
BB-2R: **s**为黑，且两个孩子均为黑；**p**为红

- ❖ 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（问号）关键码必为黑色
有且仅有**一个**



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
- ❖ 合并之前，**p**和**s**均对应于**单关键码**节点
- ❖ 失去关键码**p**后上层节点必然继而下溢
- ❖ 好在可继续分情况处理高度递增，至多 $\mathcal{O}(\log n)$ 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**)

❖ $\text{zag}(p)$ 或 $\text{zig}(p)$; 红**s**转黑, 黑**p**转红

❖ 黑高度依然异常, 但...

❖ **r**有了一个新的**黑**兄弟 **s'**

故转化为前述情况, 而且...

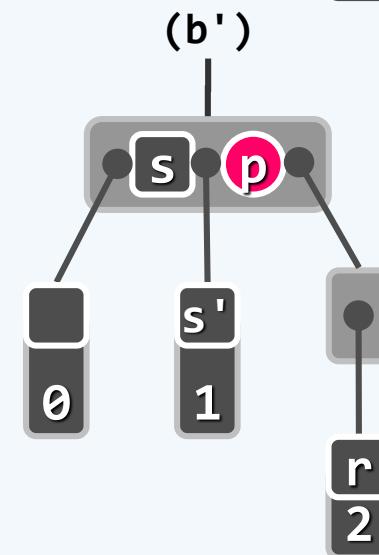
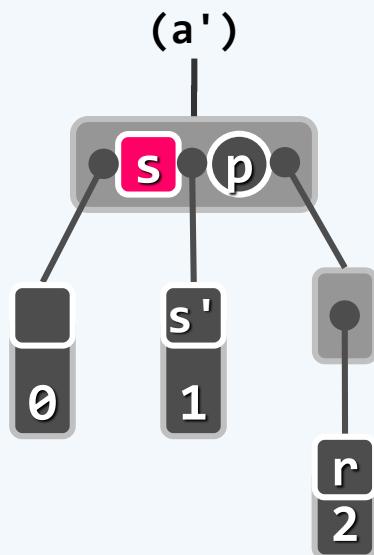
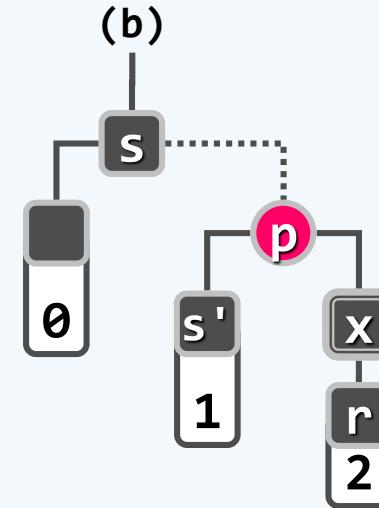
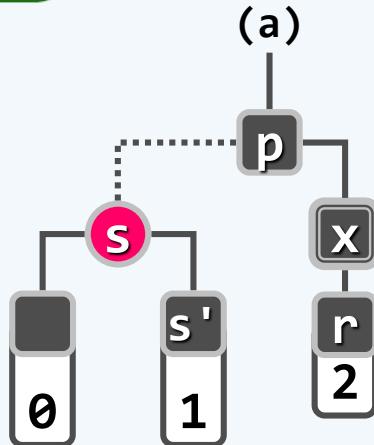
❖ 既然**p**已转**红**, 接下来

绝不会是情况**BB-2B**

而只能是**BB-1**或**BB-2R**

❖ 于是, 再经**一轮**调整之后

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

- ❖ 红黑树的每一删除操作都可在 $\mathcal{O}(\log n)$ 时间内完成

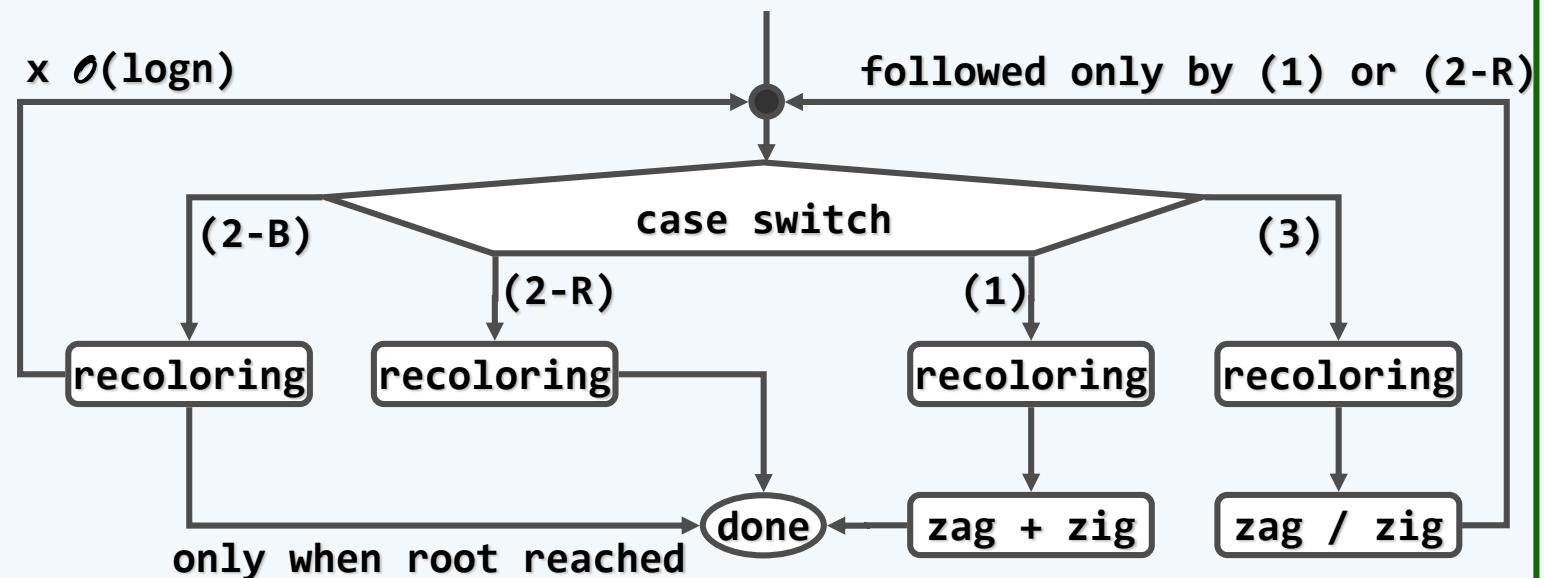
- ❖ 其中，至多做

1. $\mathcal{O}(\log n)$ 次重染色

2. 一次“3+4”重构

3. 一次单旋

情况	旋转次数	染色次数	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	必然再次双黑 但将上升一层
(3) 红s	1	2	转为(1)或(2R)



思考题

- Bottom-Up is recursive
 - BST deletion going down the tree (winding up the recursion)
 - Fixing the RB properties coming back up the tree (unwinding the recursion)
- Top-Down is **iterative**
 - Restructure the tree on the way down so we don't have to go back up

类似于Spray树，是否有Top-down的删除算法呢？

Reference

- Rudolf Bayer (1972). "Symmetric binary B-Trees: Data structure and maintenance algorithms". *Acta Informatica*. 1 (4): 290 - 306
- Data Structures and Algorithm Analysis in C++ Section 12.2
- http://users.cis.fiu.edu/~weiss/dsaa_c++3/code/RedBlackTree.h

Next

- kd- 树
- 数据结构(C++语言版)第三版 Chapter 8.4