

06-D5

二叉搜索树

AVL树: (3+4)-重构

邓俊辉

deng@tsinghua.edu.cn

大道至简

# 返璞归真

❖ 设 $g$ 为**最低**的失衡节点，沿**最长**分支考察祖孙三代： $g \sim p \sim v$

按**中序**遍历次序，**重命名**为： $a < b < c$

❖ 它们总共拥有四棵子树（或为空）

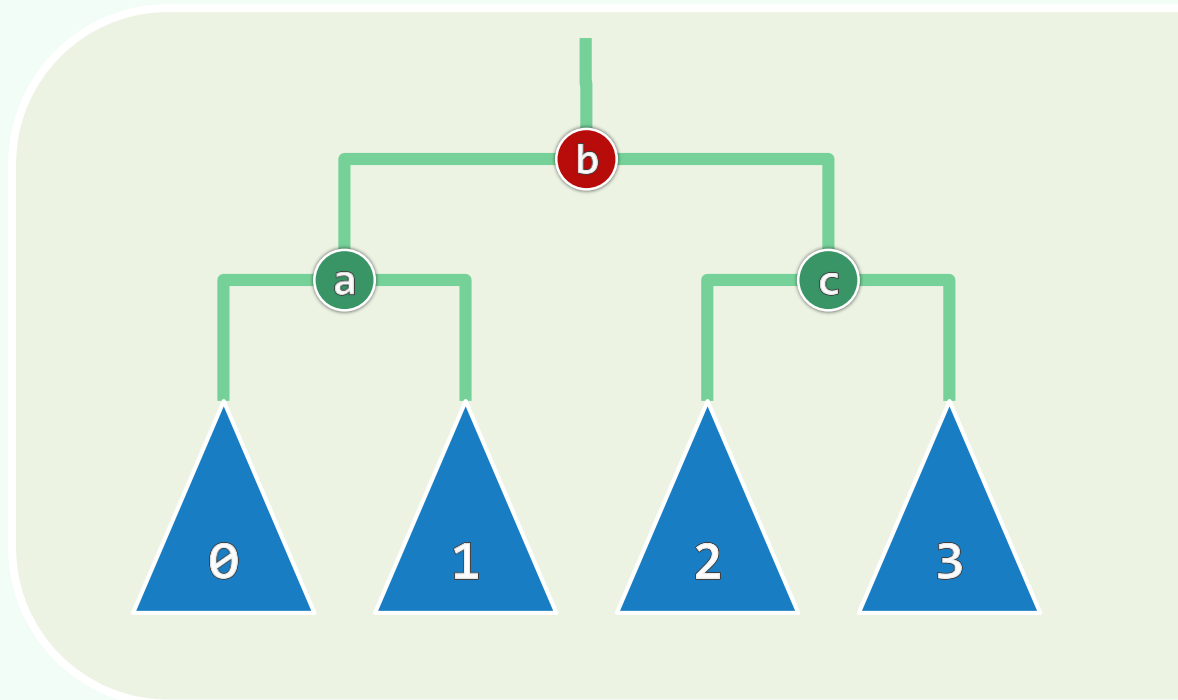
按**中序**遍历次序，**重命名**为： $T_0 < T_1 < T_2 < T_3$

$$\text{root}(S') = b$$

$$\text{lc}(b) = a$$

$$\text{rc}(b) = c$$

$$\text{lt}(a) = T_0 \quad \text{rt}(a) = T_1 \quad \text{lt}(c) = T_2 \quad \text{rt}(c) = T_3$$

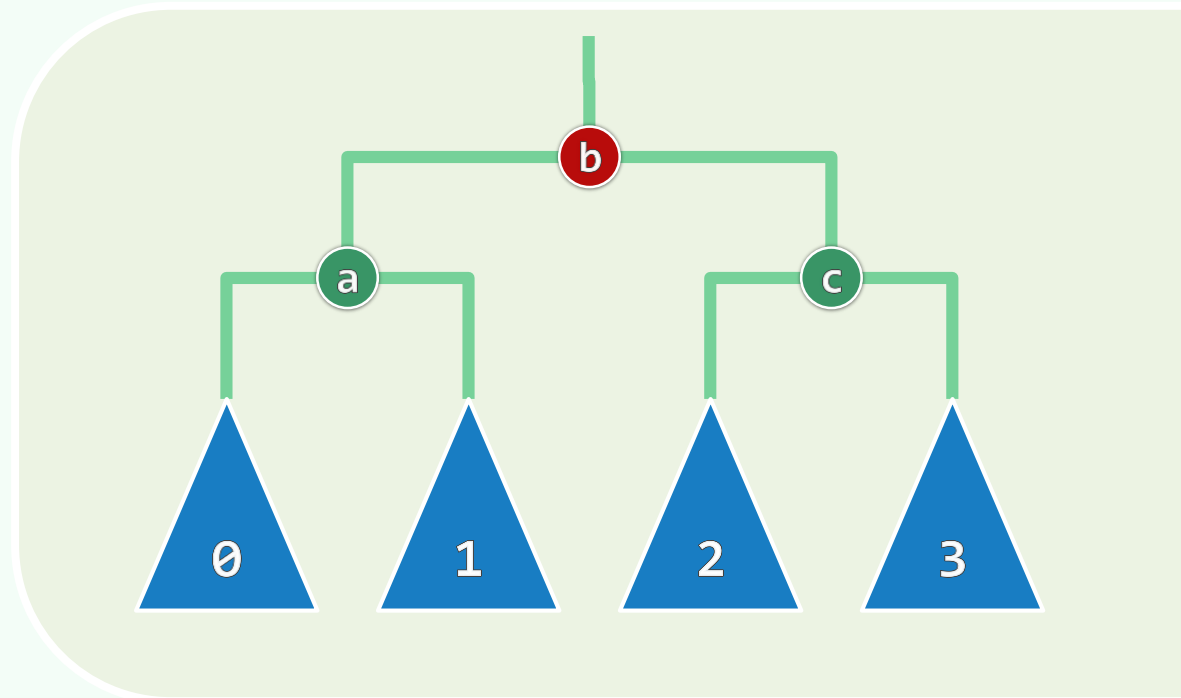


❖ 将原先以 $g$ 为根的子树 $s$ ，替换为一棵新子树 $s'$

❖ 等价变换，保持中序遍历次序： $T_0 < a < T_1 < b < T_2 < c < T_3$

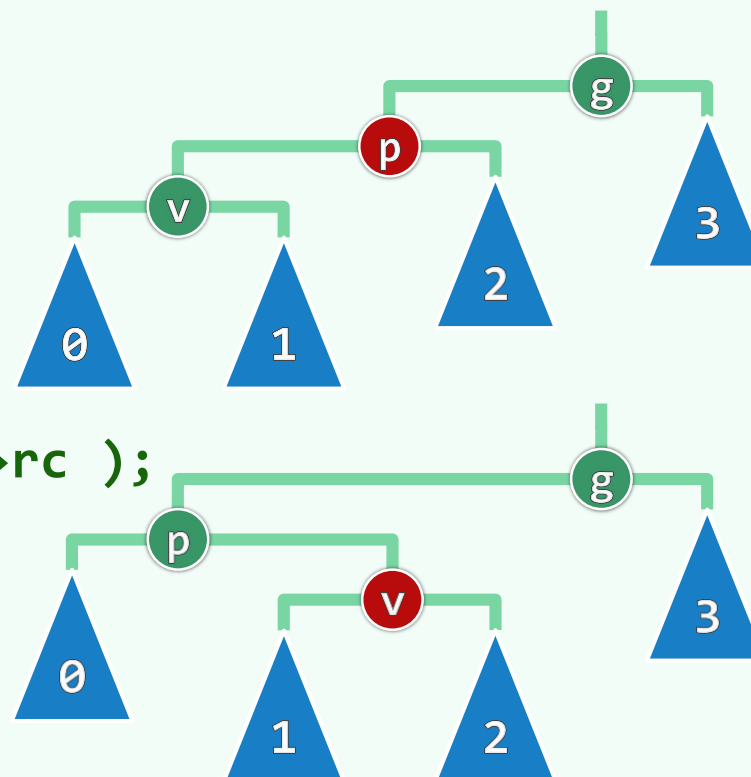
# 实现

```
template <typename T> BinNodePosi<T> BST<T>::connect34(  
    BinNodePosi<T> a, BinNodePosi<T> b, BinNodePosi<T> c,  
    BinNodePosi<T> T0, BinNodePosi<T> T1,  
    BinNodePosi<T> T2, BinNodePosi<T> T3)  
{  
    a->lc = T0; if (T0) T0->parent = a;  
    a->rc = T1; if (T1) T1->parent = a;  
    c->lc = T2; if (T2) T2->parent = c;  
    c->rc = T3; if (T3) T3->parent = c;  
    b->lc = a; a->parent = b; b->rc = c; c->parent = b;  
    updateHeight(a); updateHeight(c); updateHeight(b); return b;  
}
```



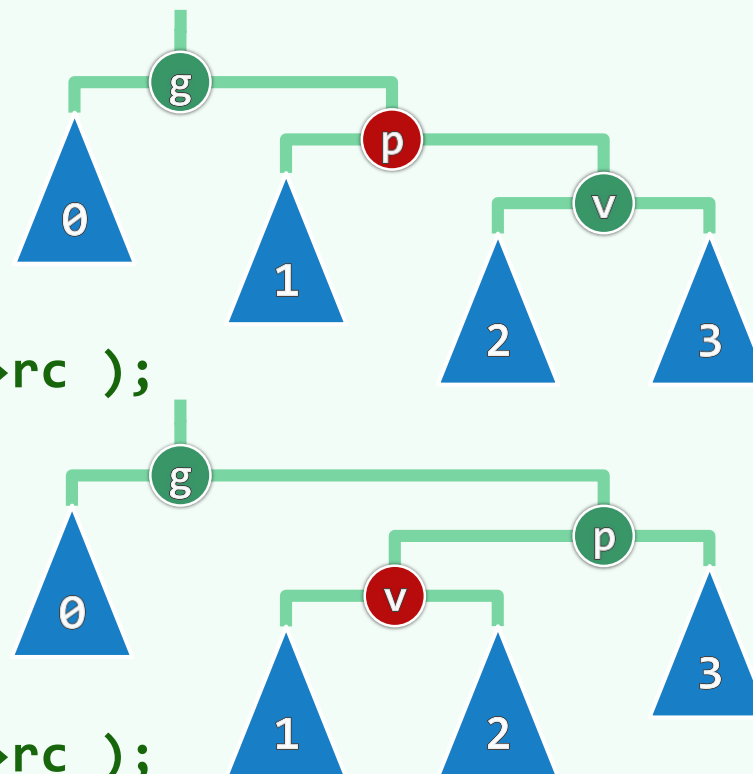
## 统一调整 (1/2)

```
template<typename T> BinNodePosi<T> BST<T>::rotateAt( BinNodePosi<T> v ) {  
    BinNodePosi<T> p = v->parent, g = p->parent;  
    if ( IsLChild( * p ) ) //zig  
        if ( IsLChild( * v ) ) { //zig-zig  
            p->parent = g->parent;  
            return connect34( v, p, g,  
                               v->lc, v->rc, p->rc, g->rc );  
        } else { //zig-zag  
            v->parent = g->parent;  
            return connect34( p, v, g,  
                               p->lc, v->lc, v->rc, g->rc );  
        }  
}
```



## 统一调整 (2/2)

```
else //zag
    if ( IsRChild( * v ) ) { //zag-zag
        p->parent = g->parent;
        return connect34( g, p, v,
                           g->lc, p->lc, v->lc, v->rc );
    } else { //zag-zig
        v->parent = g->parent;
        return connect34( g, v, p,
                           g->lc, v->lc, v->rc, p->rc );
    }
}
```



## AVL：综合评价

❖ 优点 无论查找、插入或删除，最坏情况下的复杂度均为 $O(\log n)$

$O(n)$ 的存储空间

❖ 缺点 借助高度或平衡因子，为此需改造元素结构，或额外封装

实测复杂度与理论值尚有差距

- 插入/删除后的旋转，成本不菲
- 删除操作后，最多需旋转 $O(\log n)$ 次（Knuth：平均仅0.21次）
- 若需频繁进行插入/删除操作，未免得得不偿失

单次动态调整后，全树拓扑结构的变化量可能高达 $O(\log n)$

❖ 有没有更好的结构呢？ //保持兴趣