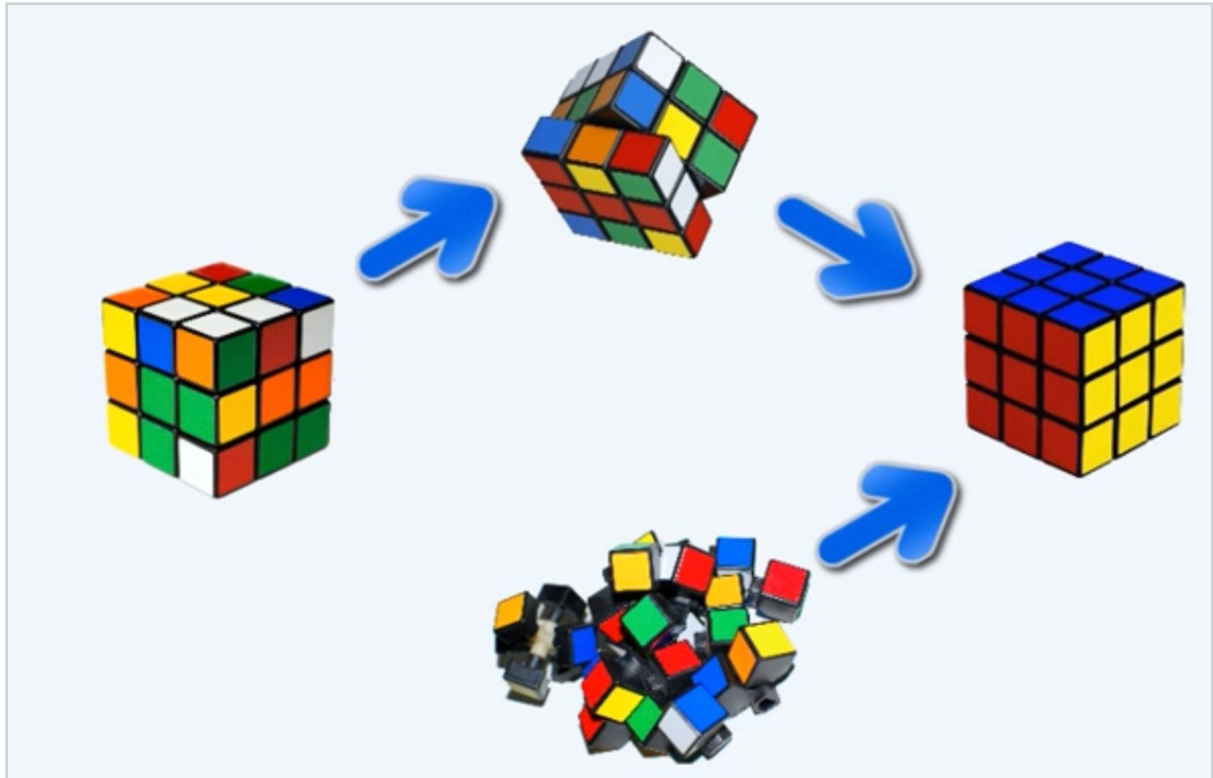


(3+4)重构

#数据结构邓神

算法

根据我们之前在插入中提出的方法，因为节点左右相对次序不变而且，永远都是把中间的向上提升，我们不妨把所有联系都切断，然后把所有节点摆好位置重新组合



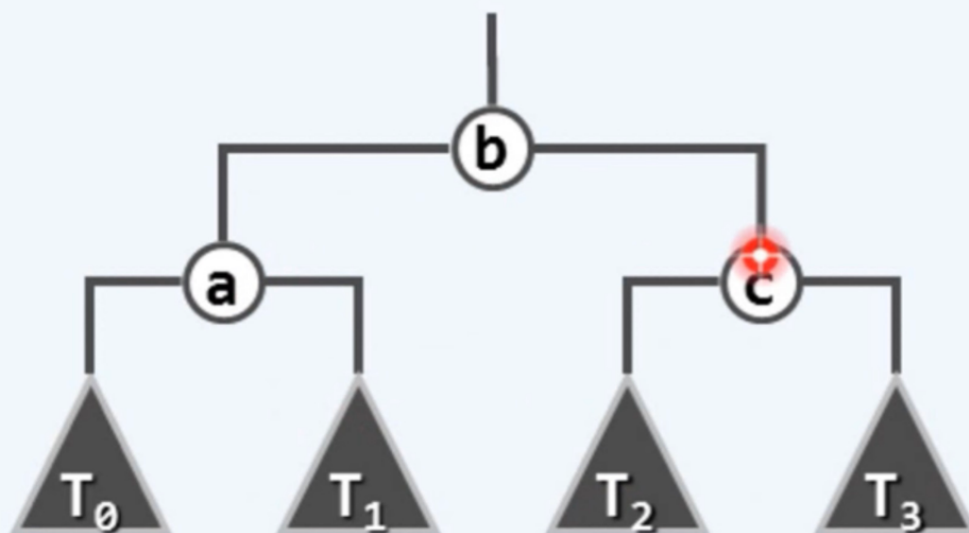
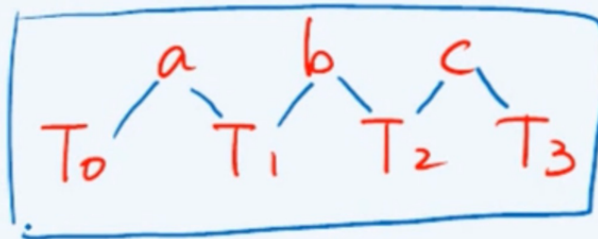
3+4重构：算法

❖ 设 $g(x)$ 为最低的失衡节点，考察祖孙三代： $g \sim p \sim v$

按中序遍历次序，将其重命名为： $a < b < c$

❖ 它们总共拥有互不相交的四棵（可能为空的）子树

按中序遍历次序，将其重命名为： $T_0 < T_1 < T_2 < T_3$



实现

3+4重构：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->lChild = T0; if (T0) T0->parent = a;
    a->rChild = T1; if (T1) T1->parent = a; updateHeight(a);
    c->lChild = T2; if (T2) T2->parent = c;
    c->rChild = T3; if (T3) T3->parent = c; updateHeight(c);
    b->lChild = a; a->parent = b;
    b->rChild = c; c->parent = b; updateHeight(b);
    return b; //该子树新的根节点
}
```

Data Structures & Algorithms (Fall 2013) - Tsinghua University

```
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->lChild = T0;
```

```

    if (T0){
        T0->parent = a;
    }
    a->rChild = T1;
    if (T1){
        T1->parent = a;
    }
    updateHeight(a);
    c->parent = T2;
    if (T2){
        T2->parent = c
    }
    c->rChild = T3;
    if (T3){
        T3->parent = c;
    }
    updateHeight(c);
    b->lChild = a;
    a->parent = b;
    b->rChild = c;
    c->parent = b;
    updateHeight(b);
    return b;
}

```

rotateAt

统一调整：实现

```
❖ template<typename T> BinNodePosi(T) BST<T>::rotateAt( BinNodePosi(T) v ) {
```

```
    BinNodePosi(T) p = v->parent, g = p->parent; //父亲、祖父
```

```
    if ( IsLChild( *p ) ) //zig
```

```
        if ( IsLChild( *v ) ) { //zig-zig
```

```
            p->parent = g->parent; //向上联接
```

```
            return connect34( v, p, g,
```

```
                v->lChild, v->rChild, p->rChild, g->rChild );
```

```
        } else { //zig-zag
```

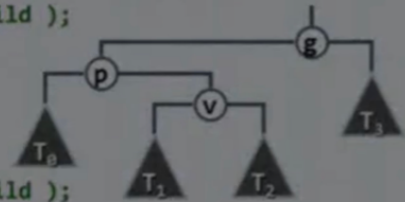
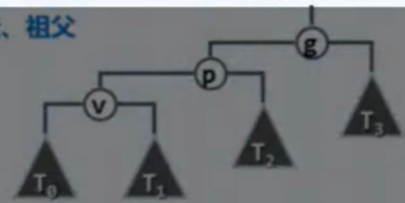
```
            v->parent = g->parent; //向上联接
```

```
            return connect34( p, v, g,
```

```
                p->lChild, v->lChild, v->rChild, g->rChild );
```

```
        }
```

```
    else { /*.. zag-zig & zag-zag ..*/ }
```



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

综合评价

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 $\Omega(\log n)$ 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 $\Omega(\log n)$