7.二叉搜索树

(d) AVL树

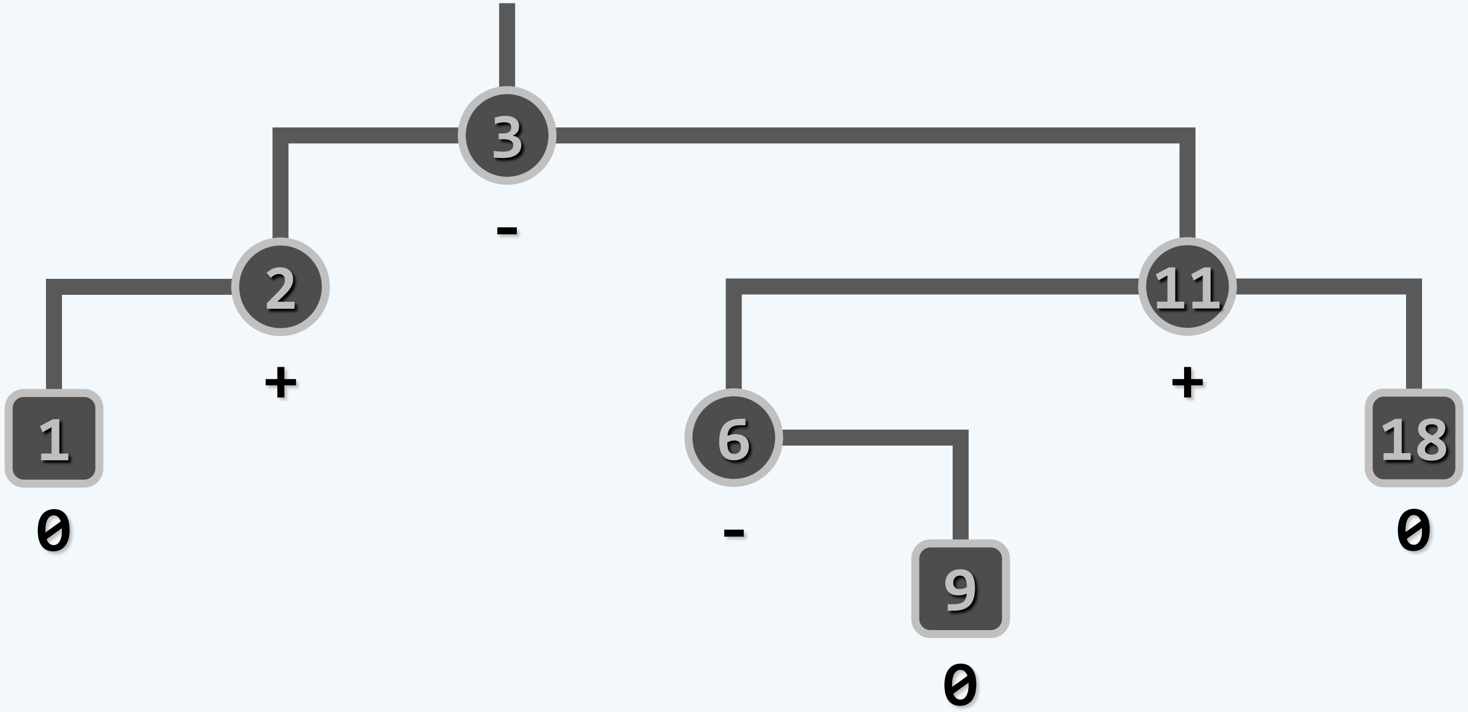
邓俊辉

deng@tsinghua.edu.cn

平衡因子

❖ balFac(v) = $\boxed{\text{height(lc(v)) - height(rc(v))}}$

❖ G. Adelson-Velsky & E. Landis (1962) : $\boxed{\forall\ v,\ |\ \text{balFac(v)}\ |\ \leq 1}$



❖ AVL树未必 理想平衡 ，必然 适度平衡 ...

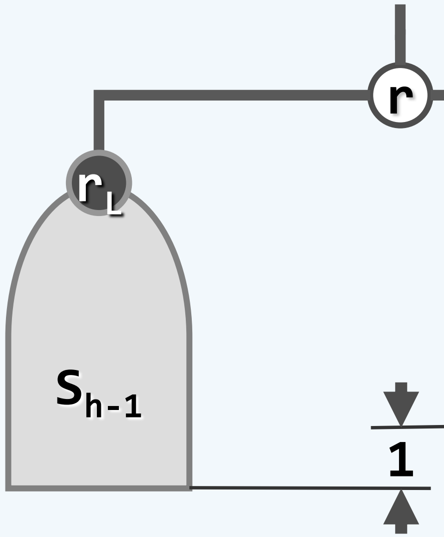
AVL = 适度平衡

❖ 高度为h的AVL树，$\boxed{\text{至少}}$包含$\boxed{S(h) = fib(h + 3) - 1}$个节点

❖ $S(h) = 1 + S(h - 1) + S(h - 2)$

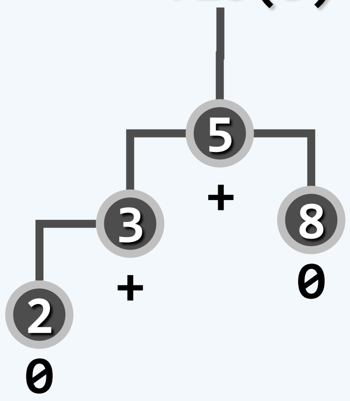
❖ $S(h) + 1 = [S(h - 1) + 1] + [S(h - 2) + 1]$

❖ $fib(h + 3) = fib(h + 2) + fib(h + 1)$

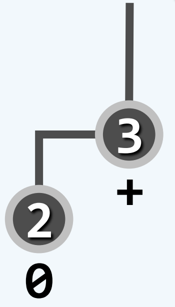


❖ 反过来，由n个节点构成的AVL树，高度$\boxed{\text{至多}}$为$O(\log n)$

$S(2) + 1 = 5$
$= fib(5)$



$S(1) + 1 = 3$
$= fib(4)$



$S(0) + 1 = 2$
$= fib(3)$

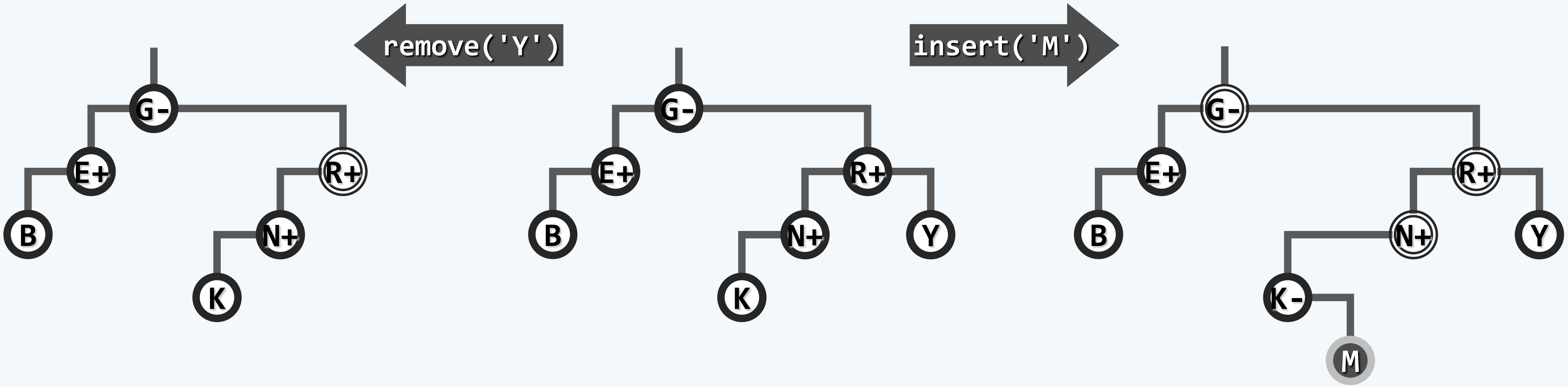
AVL：接口

❖
```
#define Balanced(x) \ //理想平衡
    ( stature( (x).lChild ) == stature( (x).rChild ) )
#define BalFac(x) \ //平衡因子
    ( stature( (x).lChild ) - stature( (x).rChild ) )
#define AvlBalanced(x) \ //AVL平衡条件
    ( ( -2 < BalFac(x) ) && ( BalFac(x) < 2 ) )
```

❖
```
template <typename T> class AVL : public BST<T> { //由BST派生
public: // BST::search()等接口，可直接沿用
    BinNodePosi(T) insert( const T & ); //插入重写
    bool remove( const T & ); //删除重写
};
```

失衡与重平衡

❖ 按BST规则插入或删除节点之后，AVL平衡性可能破坏——如何恢复？



❖ 蛮力不足取，须借助等价变换

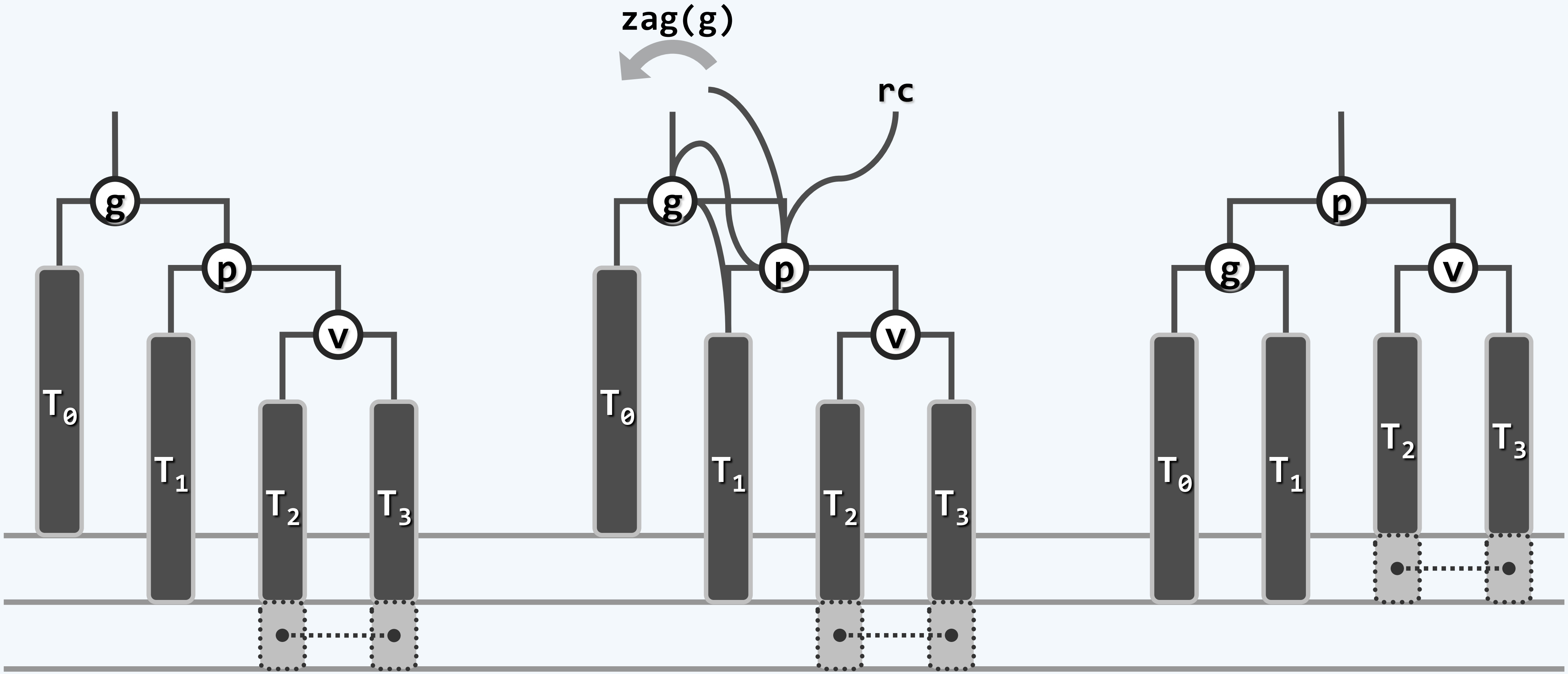
局部性：所有的旋转都在局部进行 //每次只需$O(1)$时间

快速性：在每一深度只需检查并旋转至多一次 //共$O(\log n)$次

插入：单旋

❖ 同时可有 多个失衡 节点，最低者g不低于 x祖父

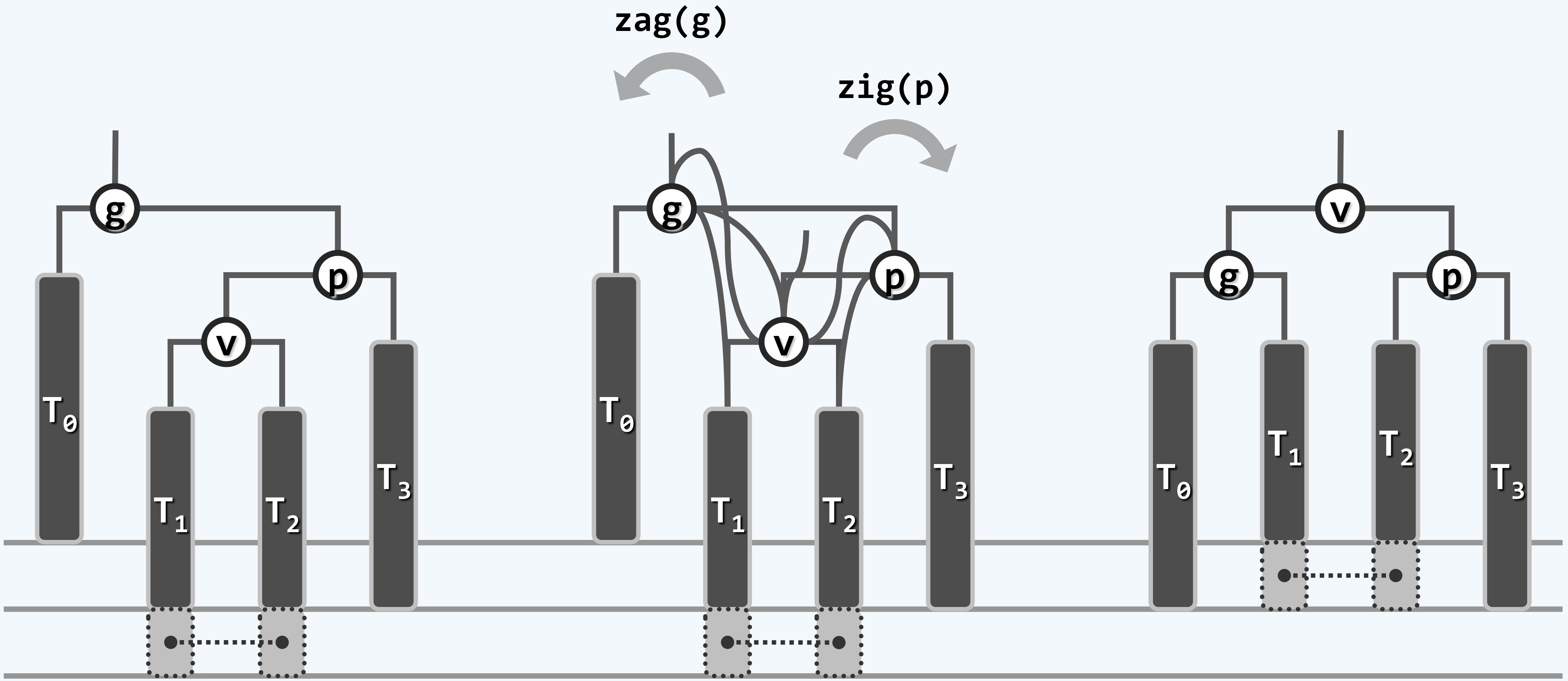
❖ g经单旋调整后复衡，子树 高度复原 ；更高祖先也必平衡， 全树复衡



插入：双旋

❖ 同时可有 多个失衡 节点，最低者g不低于 x祖父

❖ g经双旋调整后复衡，子树 高度复原 ；更高祖先也必平衡， 全树复衡

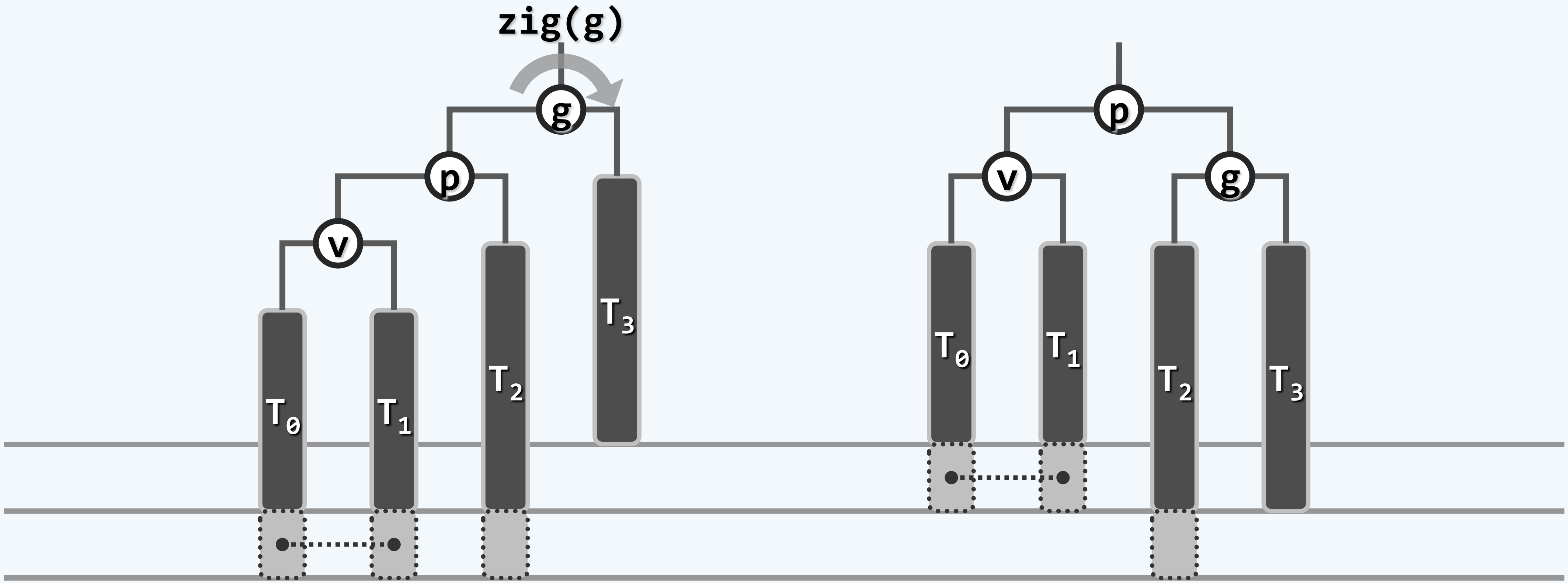


插入：实现

```
❖ template <typename T> BinNodePosi(T) AVL<T>::insert( const T & e ) {
     BinNodePosi(T) & x = search( e ); if ( x ) return x; //若目标尚不存在
     BinNodePosi(T) xx = x = new BinNode<T>( e, _hot ); _size++; //则创建新节点
  // 以下，从x的父亲_hot出发逐层向上，依次检查各代祖先g
     for ( BinNodePosi(T) g = _hot; g; g = g->parent )
        if ( ! AvlBalanced( *g ) ) { //一旦发现g失衡，则通过调整恢复平衡
           FromParentTo( *g ) = rotateAt( tallerChild( tallerChild( g ) ) );
           break; //g复衡后，局部子树高度必然复原；其祖先亦必如此，故调整结束
        } else //否则（在依然平衡的祖先处），只需简单地
           updateHeight( g ); //更新其高度（平衡性虽不变，高度却可能改变）
     return xx; //返回新节点：至多只需一次调整
  }
```

删除：单旋

- ❖ 同时至多一个失衡节点g，首个可能就是x的父亲_hot
- ❖ g经单旋调整后复衡，子树高度未必复原；更高祖先仍可能失衡
- ❖ 因有失衡传播现象，可能需做$\mathcal{O}(\log n)$次调整

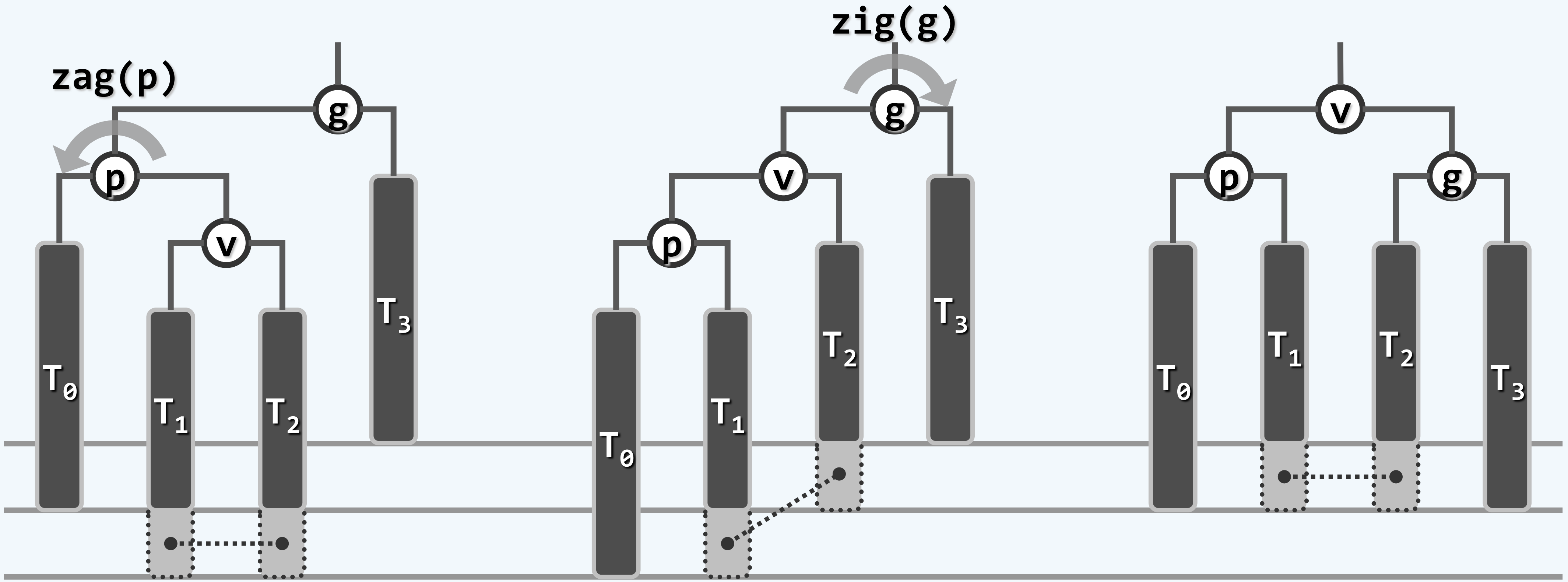


删除：双旋

❖ 同时至多一个失衡节点g，首个可能就是x的父亲_hot

❖ g经单旋调整后复衡，子树高度未必复原；更高祖先仍可能失衡

❖ 因有失衡传播现象，可能需做$O(\log n)$次调整



删除：实现

- template <typename T> bool AVL<T>::remove(const T & e) {

```
template <typename T> bool AVL<T>::remove( const T & e ) {
   BinNodePosi(T) & x = search( e ); if ( !x ) return false; //若目标的确存在
   removeAt( x, _hot ); _size--; //则在按BST规则删除之后，_hot及其祖先均有可能失衡
// 以下，从_hot出发逐层向上，依次检查各代祖先g
   for ( BinNodePosi(T) g = _hot; g; g = g->parent ) {
      if ( ! AvlBalanced( *g ) ) //一旦发现g失衡，则通过调整恢复平衡
         g = FromParentTo( *g ) = rotateAt( tallerChild( tallerChild( g ) ) );
      updateHeight( g ); //并更新其高度
   } //可能需做过Ω(logn)次调整；无论是否做过调整，全树高度均可能下降
   return true; //删除成功
}
```

3+4重构：算法

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

  按中序遍历次序，将其重命名为：$\boxed{a < b < c}$

- 它们总共拥有互不相交的四棵（可能为空的）子树

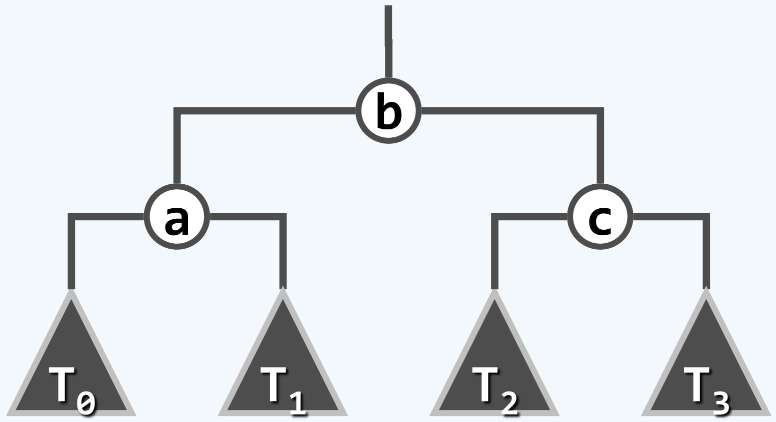
  按中序遍历次序，将其重命名为：$\boxed{T_0 < T_1 < T_2 < T_3}$

- 将原先以g为根的子树S，替换为一棵新子树S'

root(S’) = b

lc(b) = a　　rc(b) = c

lT(a) = T0　　rT(a) = T1　　lT(c) = T2　　rT(c) = T3



- 等价变换，保持中序遍历次序：$\boxed{T_0 < a < T_1 < b < T_2 < c < T_3}$

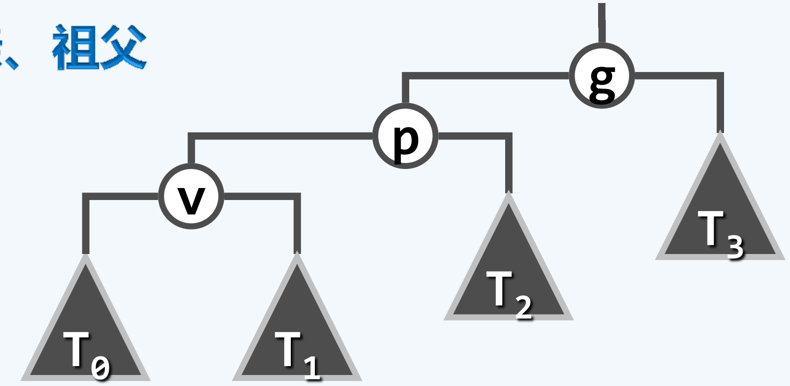
3+4重构：实现

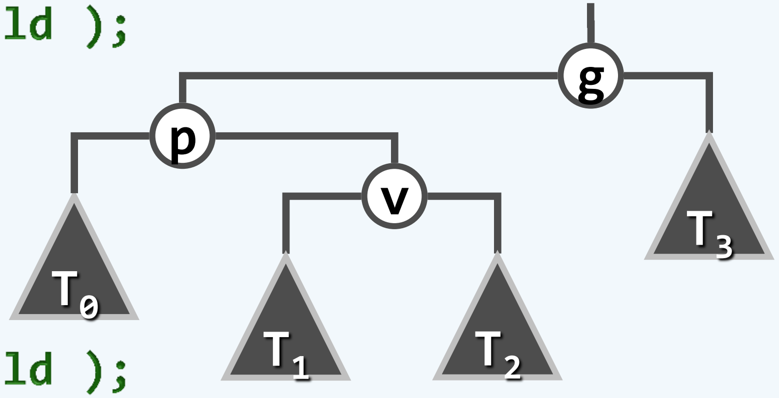
❖ template <typename T> BinNodePosi(T) BST<T>::connect34(

```
    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
}
```

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 ..*/ }
}
```





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 $\boxed{O(\log n)}$

  $O(n)$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 $\boxed{\Omega(\log n)}$ 次（Knuth：平均仅 0.21 次）

  若需频繁进行插入/删除操作，未免得不偿失

  单次动态调整后，全树 拓扑 结构的变化量可能高达 $\boxed{\Omega(\log n)}$

- 有没有更好的结构呢？　//保持兴趣