

二叉搜索树

概述：循关键码访问



There's nothing in your head the sorting hat can't see.
So try me on and I will tell you where you ought to be.

θ6 - A7

邓俊辉

deng@tsinghua.edu.cn

循关键码访问

- ❖ 很遗憾，向量、列表并不能兼顾静态查找与动态修改
- ❖ 能否综合二者的优点？
- ❖ 各数据项依所持**关键码**而彼此区分：call-by-KEY
- ❖ 当然，关键码之间必须同时支持**比较**（大小）与**比对**（相等）
- ❖ 数据集中的数据项，统一地表示和实现为词条（entry）形式



基本结构	查找	插入/删除
无序向量	$\Theta(n)$	$\Theta(n)$
有序向量	$\Theta(\log n)$	$\Theta(n)$
无序列表	$\Theta(n)$	$\Theta(1)$
有序列表	$\Theta(n)$	$\Theta(n)$

词条

```
template <typename K, typename V> struct Entry { //词条模板类
    K key; V value; //关键码、数值
    Entry( K k = K(), V v = V() ) : key(k), value(v) {};
    Entry( Entry<K, V> const & e ) : key(e.key), value(e.value) {};
    // 比较器、判等器 (从此，不必严格区分词条及其对应的关键码)
    bool operator< ( Entry<K, V> const & e ) { return key < e.key; } //小于
    bool operator> ( Entry<K, V> const & e ) { return key > e.key; } //大于
    bool operator==( Entry<K, V> const & e ) { return key == e.key; } //等于
    bool operator!=( Entry<K, V> const & e ) { return key != e.key; } //不等
};
```

二叉搜索树

概述：中序

06-A2

邓俊辉

deng@tsinghua.edu.cn

顺序性

❖ 任一节点均不小于/大于

其左/右后代

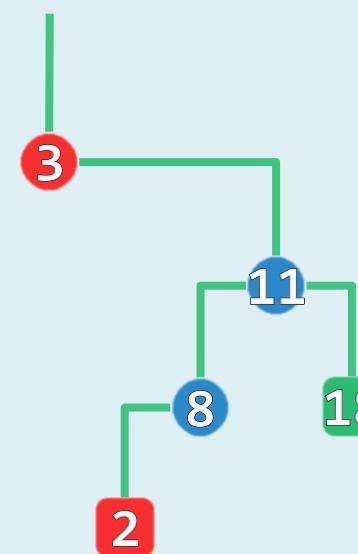
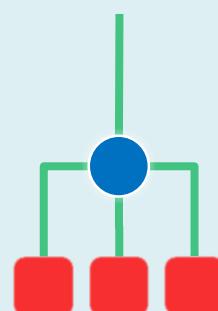
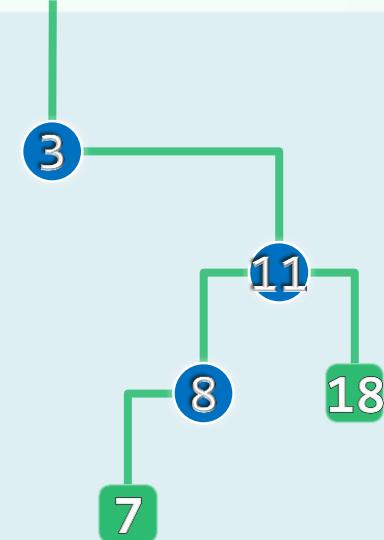
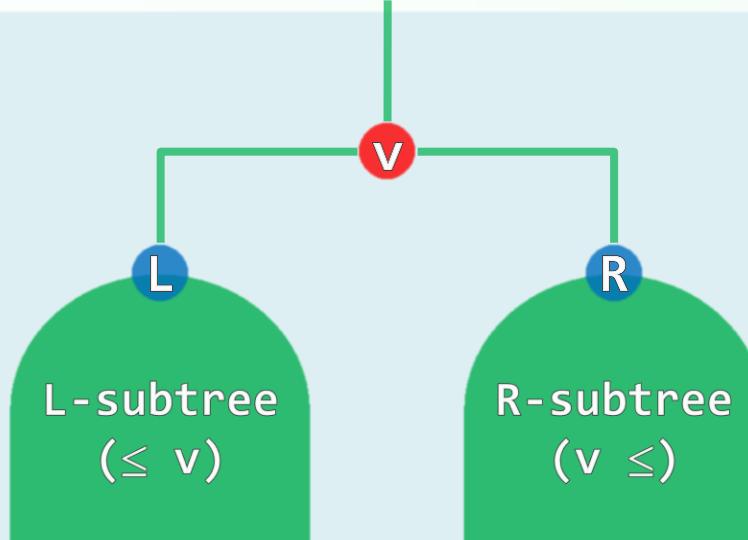
❖ 是否等效于...

❖ 任一节点均不小于/不大于

其左/右孩子

❖ 三位一体：

节点 ~ 词条 ~ 关键码



重复关键码

❖ 为简化起见，暂且

禁止词条的关键码重复

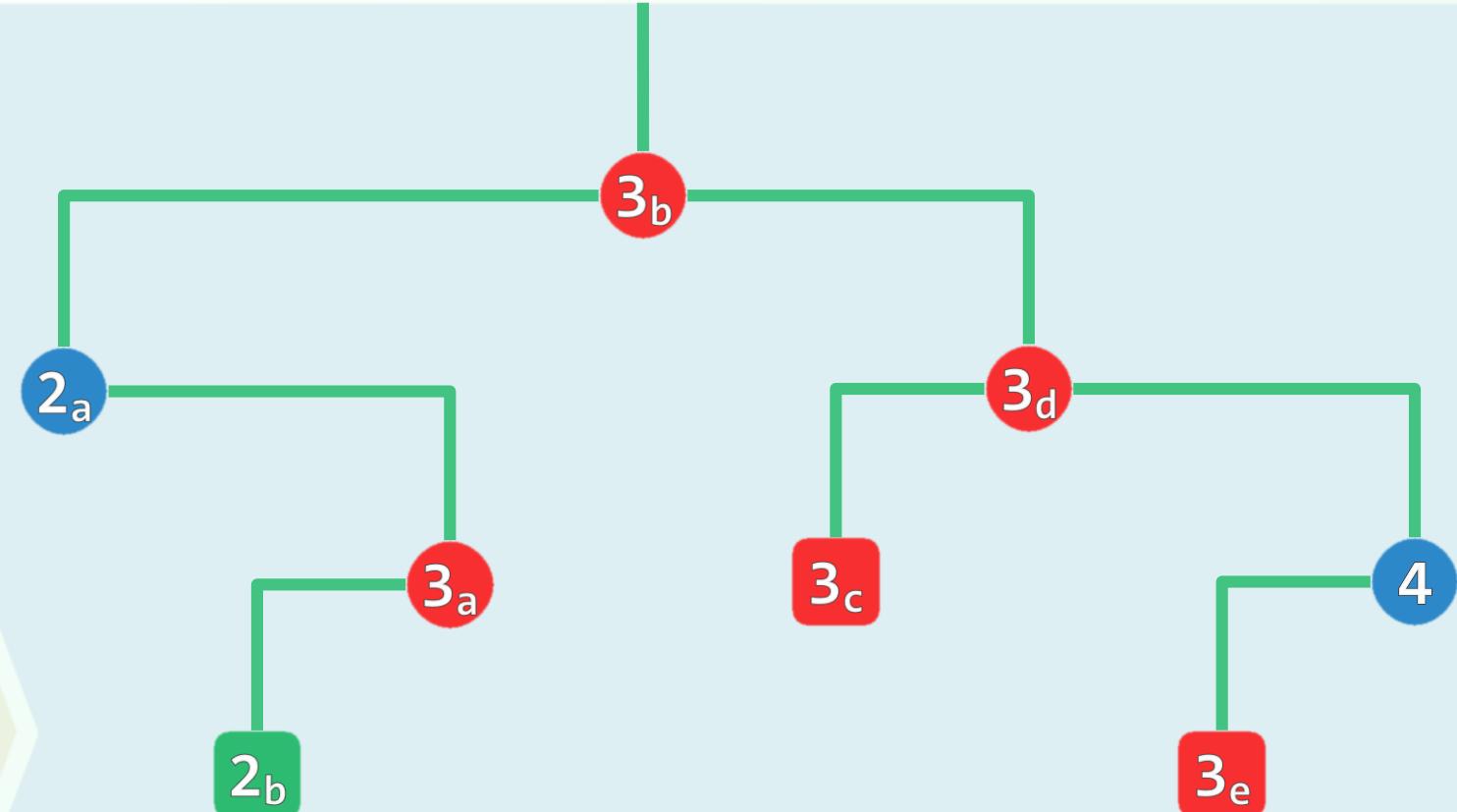
❖ 当然，这种简化

- 在应用中本非自然

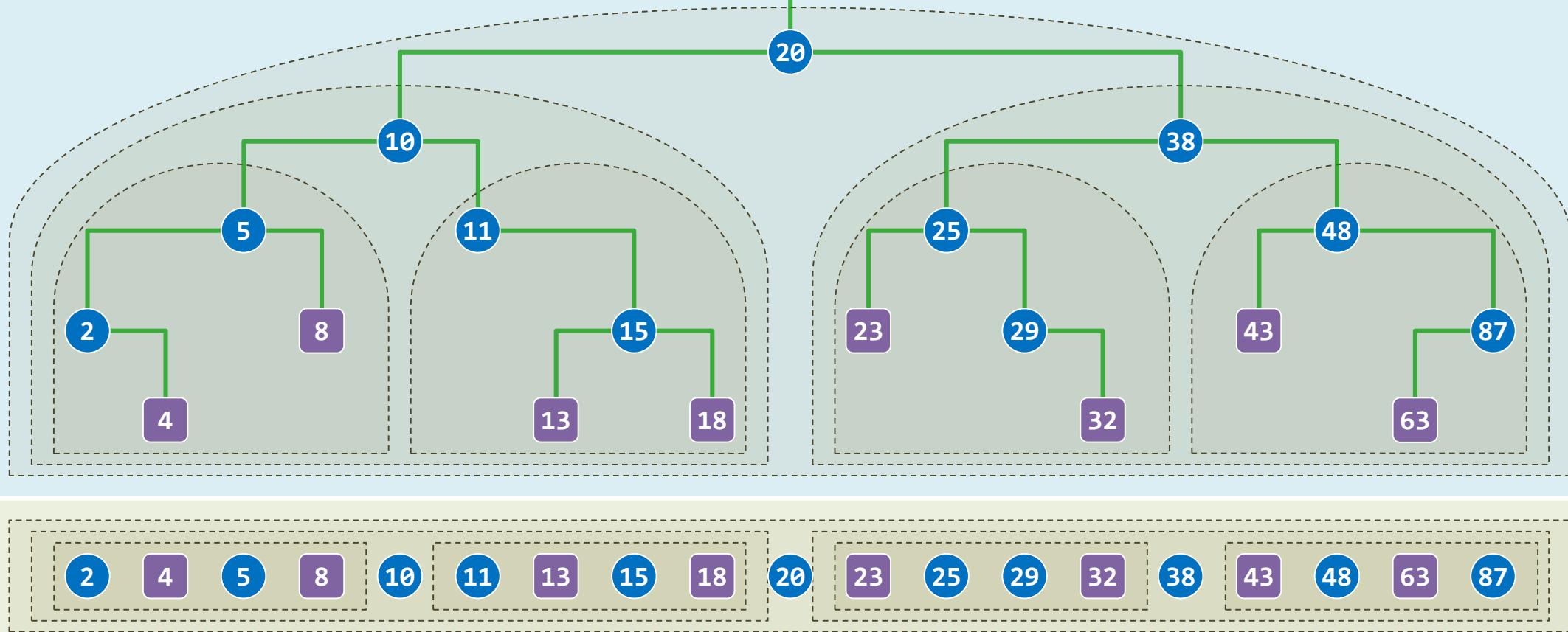
- 从算法看亦无必要

❖ 稍后将会看到，重复的关键码...

❖ 完全可以共存于同一棵BST中： `searchAll() + searchFirst()` //习题[7-10]、[7-16]、[8-3]



单调性



❖ 顺序性虽只是对局部特征的刻画

却可导出BST的整体特征

❖ 对树高做数学归纳，不难证明...

BST的中序遍历序列，必然单调非降

二叉搜索树

概述：接口

06-A3

邓俊辉

deng@tsinghua.edu.cn

对外接口

```
template <typename T> class BST : public BinTree<T> {
```

public: //以virtual修饰，以便派生类重写

```
    virtual BinNodePosi<T> & search( const T & ); //查找
```

```
    virtual BinNodePosi<T> insert( const T & ); //插入
```

```
    virtual bool remove( const T & ); //删除
```

protected:

```
/* ..... */
```

```
};
```

内部接口

```
template <typename T> class BST : public BinTree<T> { //由BinTree派生
public:
    /* ..... */

protected:
    BinNodePosi<T> _hot; //命中节点的父亲
    BinNodePosi<T> connect34( //3+4重构, 稍晚再详解
        BinNodePosi<T>, BinNodePosi<T>, BinNodePosi<T>,
        BinNodePosi<T>, BinNodePosi<T>, BinNodePosi<T>, BinNodePosi<T> );
    BinNodePosi<T> rotateAt( BinNodePosi<T> ); //旋转调整
};
```

二叉搜索树

算法及实现：查找

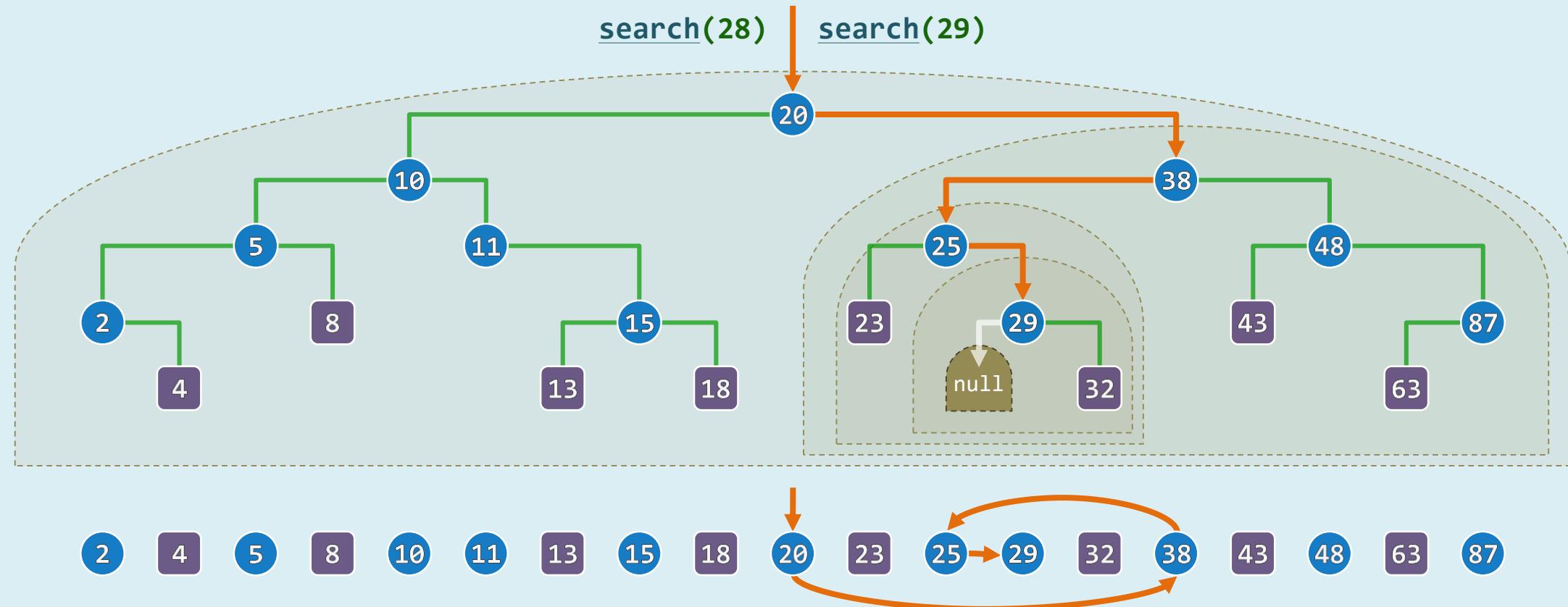
06-B1

邓俊辉

deng@tsinghua.edu.cn

为学日益，为道日损，损之又损，以至于无为，无为而无不为。

减而治之



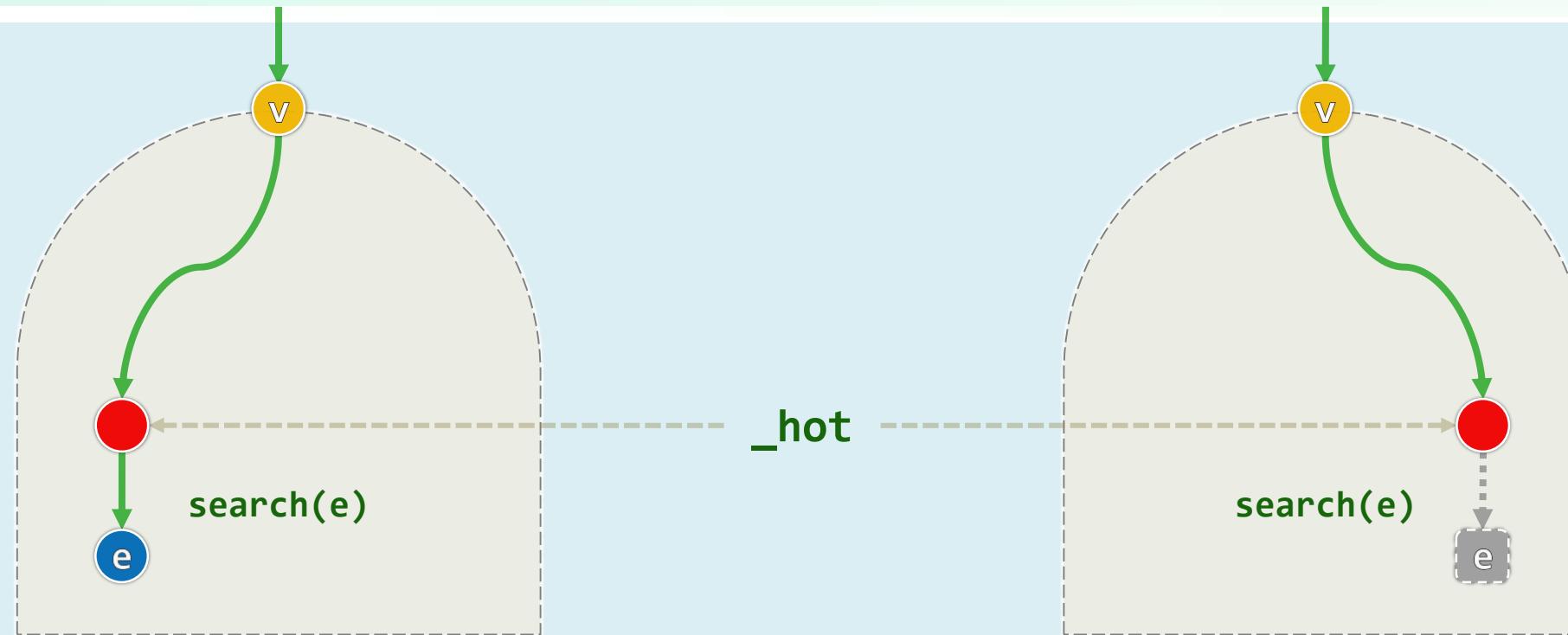
❖ 从根节点出发，逐步地缩小查找范围，直到
- 发现目标（成功），或抵达空树（失败）

❖ 对照中序遍历序列可见，整个过程可视作是在
仿效有序向量的二分查找

实现

```
❖ template <typename T> BinNodePosi<T> & BST<T>::search( const T & e ) {  
    if ( !_root || e == _root->data ) //空树, 或恰在树根命中  
    { _hot = NULL; return _root; }  
  
    for ( _hot = _root; ; ) { //否则, 自顶而下  
        BinNodePosi<T> & v = ( e < _hot->data ) ? _hot->lc : _hot->rc; //深入一层  
        if ( !v || e == v->data ) return v; _hot = v; //一旦命中或抵达叶子, 随即返回  
    } //返回目标节点位置的引用, 以便后续插入、删除操作  
} //无论命中或失败, _hot均指向v之父亲 (v是根时, hot为NULL)
```

返回的引用值



- ❖ 查找成功时，指向一个关键码为e且**真实存在的**节点
 - ❖ 失败时，指向最后一次试图转向的空节点**NULL**——随后可视需要进行**修改**
- 此时，不妨**假想地**将该空节点转换为一个数值为e的**哨兵节点**

二叉搜索树

算法及实现：插入

06-B2

邓俊辉

deng@tsinghua.edu.cn

算法

❖ 先借助search(e)

确定插入位置及方向

❖ 若e尚不存在，则再

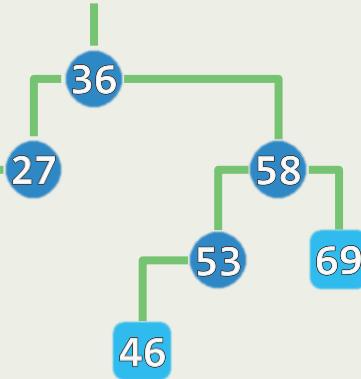
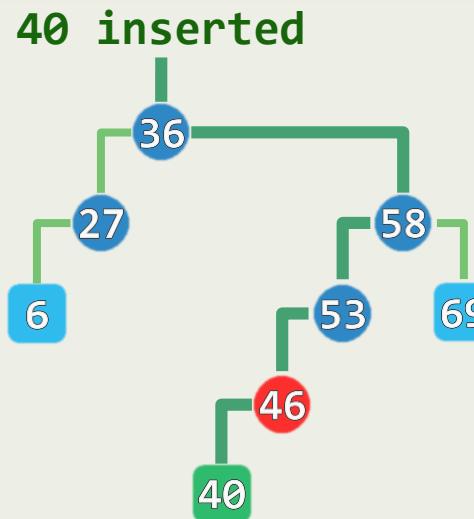
将新节点作为叶子插入

- _hot为新节点的**父亲**

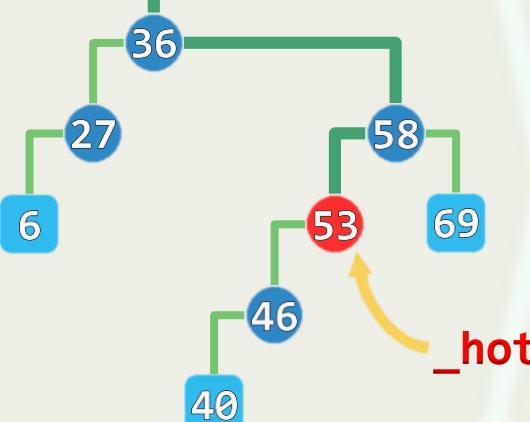
- $v = \text{search}(e)$

为_hot对新孩子的**引用**

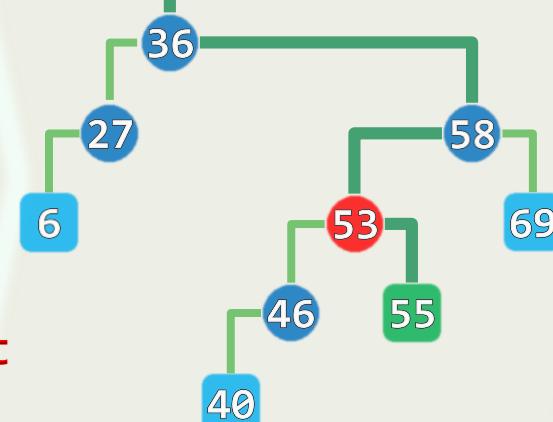
❖ 于是，只需令_hot通过v**指向新节点**



insert(55)



55 inserted



实现

```
template <typename T> BinNodePosi<T> BST<T>::insert( const T & e ) {  
    BinNodePosi<T> & x = search( e ); //查找目标 (留意_hot的设置)  
    if ( ! x ) { //既禁止雷同元素，故仅在查找失败时才实施插入操作  
        x = new BinNode<T>( e, _hot ); //在x处创建新节点，以_hot为父亲  
        _size++; updateHeightAbove( x ); //更新全树规模，更新x及其历代祖先的高度  
    }  
    return x; //无论e是否存在于原树中，至此总有x->data == e  
} //验证：对于首个节点插入之类的边界情况，均可正确处置
```

- ❖ 时间主要消耗于search(e)和updateHeightAbove(x)；均线性正比于x的深度，不超过树高

二叉搜索树

算法及实现：删除

06-B3

邓俊辉

deng@tsinghua.edu.cn

十步之泽，必有香草；十室之邑，必有忠士

君子无终食之间违仁，造次必于是，颠沛必于是

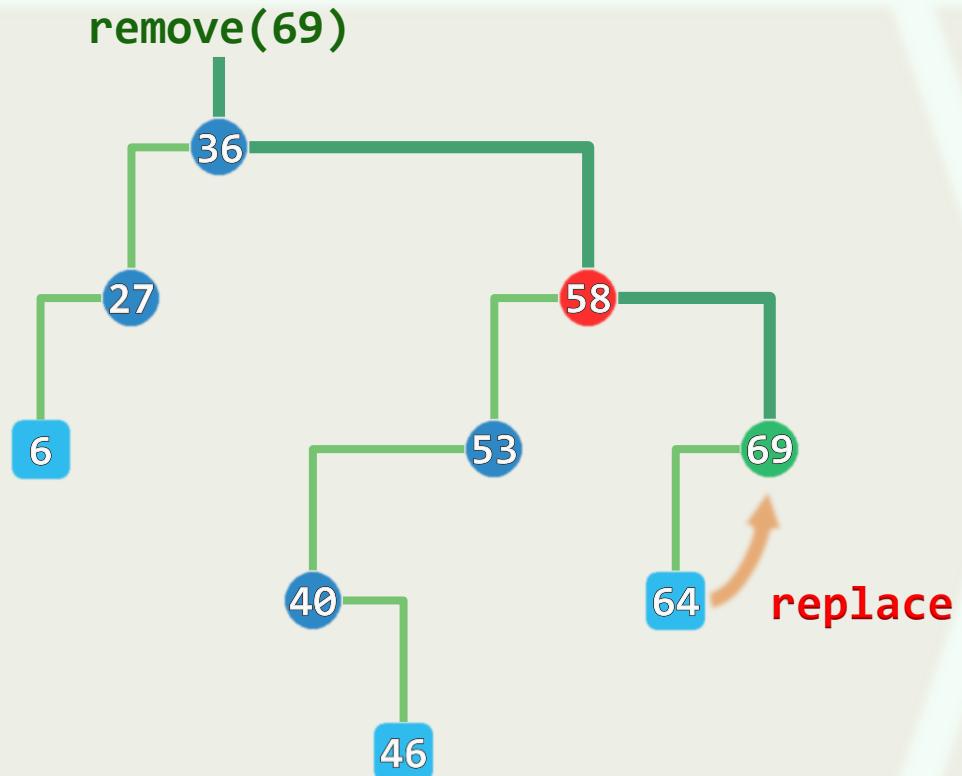
主算法

```
template <typename T> bool BST<T>::remove( const T & e ) {  
    BinNodePosi<T> & x = search( e ); //定位目标节点  
    if ( !x ) return false; //确认目标存在 (此时 _hot 为 x 的父亲)  
    removeAt( x, _hot ); _size--; //分两大类情况实施删除  
    _size--; updateHeightAbove( _hot ); //更新全树规模，更新 _hot 及其历代祖先的高度  
    return true;  
} //删除成功与否，由返回值指示
```

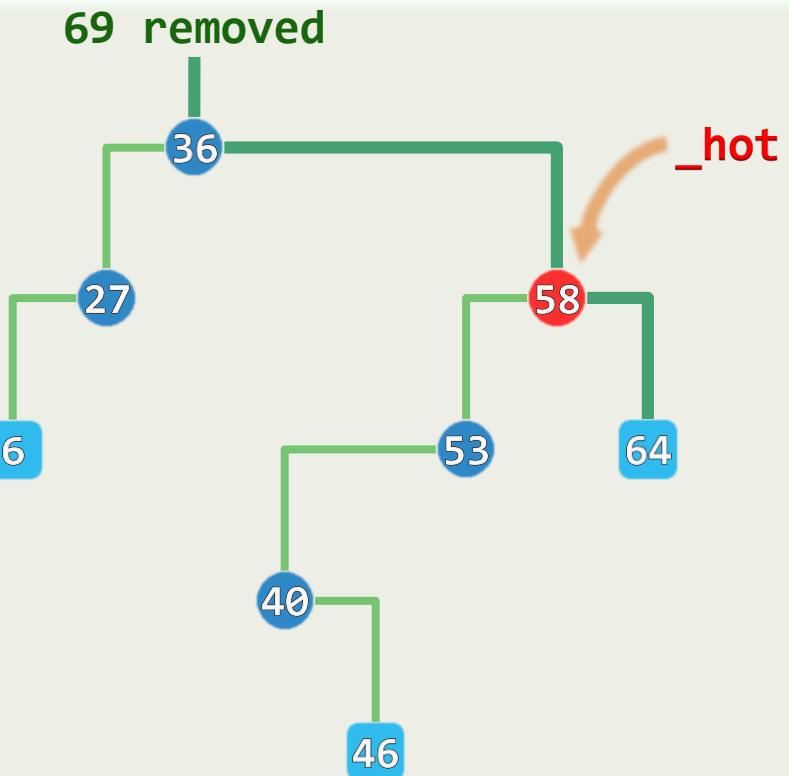
❖ 累计 $O(h)$ 时间： search()、updateHeightAbove()；还有 removeAt() 中可能调用的 succ()

单分支：实例

- ❖ 若 $*x$ (69) 的某一子树为空，则可将其替换为另一子树 (64) //可能亦为空



- ❖ 验证：如此操作之后，二叉搜索树的拓扑结构依然完整；顺序性依然满足



单分支：实现

```
template <typename T> static BinNodePosi<T>  
removeAt( BinNodePosi<T> & x, BinNodePosi<T> & hot ) {  
    BinNodePosi<T> w = x; //实际被摘除的节点，初值同x  
    BinNodePosi<T> succ = NULL; //实际被删除节点的接替者  
    if ( ! HasLChild( *x ) ) succ = x = x->rC; //左子树为空  
    else if ( ! HasRChild( *x ) ) succ = x = x->lC; //右子树为空  
    else { /* ...左、右子树并存的情况，略微复杂些... */ }  
    hot = w->parent; //记录实际被删除节点的父亲  
    if ( succ ) succ->parent = hot; //将被删除节点的接替者与hot相联  
    release( w->data ); release( w ); return succ; //释放被摘除节点，返回接替者  
} //此类情况仅需 $\Theta(1)$ 时间
```

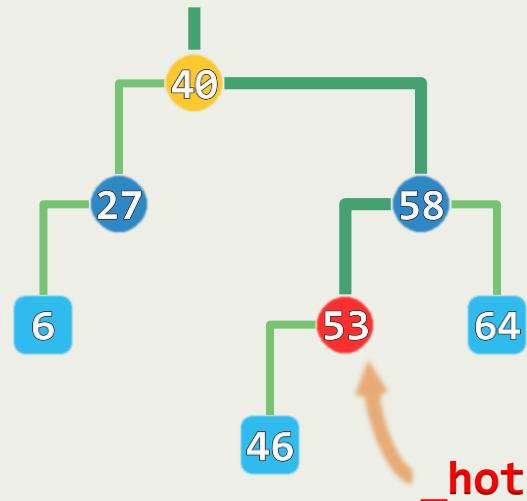
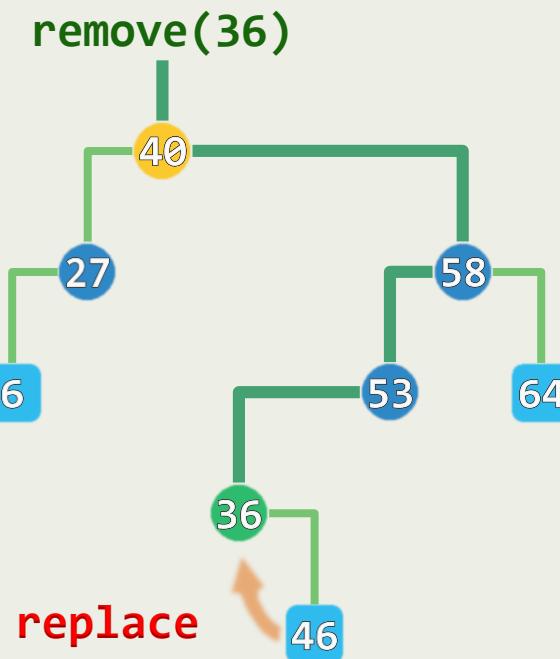
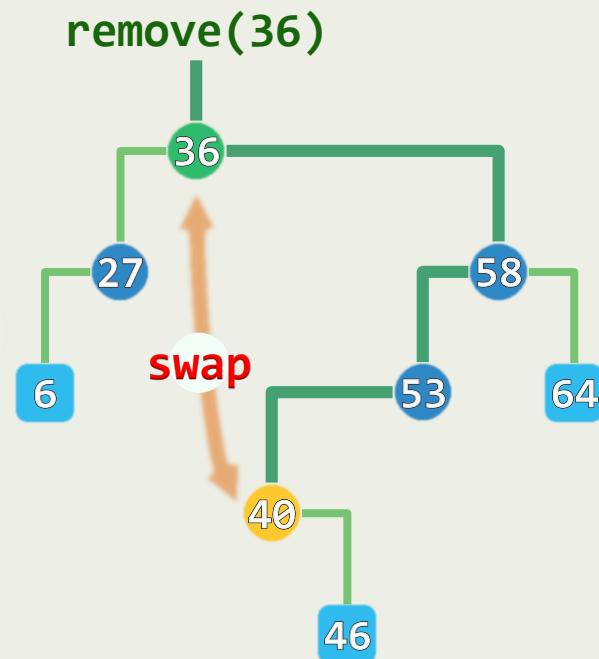
双分支：实例

❖ 若： $*x$ (36) 左、右孩子并存

则：调用`BinNode::succ()`找到 x 的直接后继
(必无左孩子)；交换 $*x$ (36) 与 $*w$ (40)

❖ 于是问题转化为删除 w ，可按前一情况处理

❖ 尽管顺序性在中途曾一度不合
但最终必将重新恢复



双分支：实现

```
template <typename T> static BinNodePosi<T>  
removeAt( BinNodePosi<T> & x, BinNodePosi<T> & hot ) {  
    /* ..... */  
    else { //若x的左、右子树并存，则  
        w = w->succ(); swap( x->data, w->data ); //令*x与其后继*w互换数据  
        BinNodePosi<T> u = w->parent; //原问题即转化为，摘除非二度的节点w  
        ( u == x ? u->rc : u->lc ) = succ = w->rc; //兼顾特殊情况：u可能就是x  
    }  
    /* ..... */  
} //时间主要消耗于succ()，正比于x的高度——更精确地，search()与succ()总共不过 $\mathcal{O}(h)$ 
```

二叉搜索树

平衡：期望树高

θ6 - C7

邓俊辉

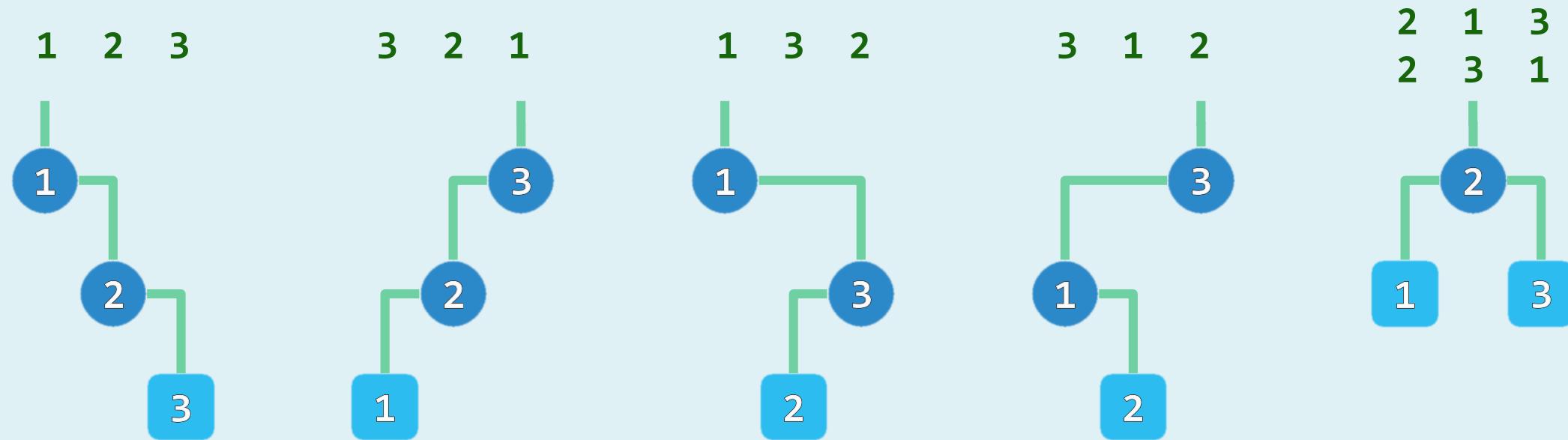
deng@tsinghua.edu.cn

上梁不正下梁歪

树高：最坏情况与平均情况

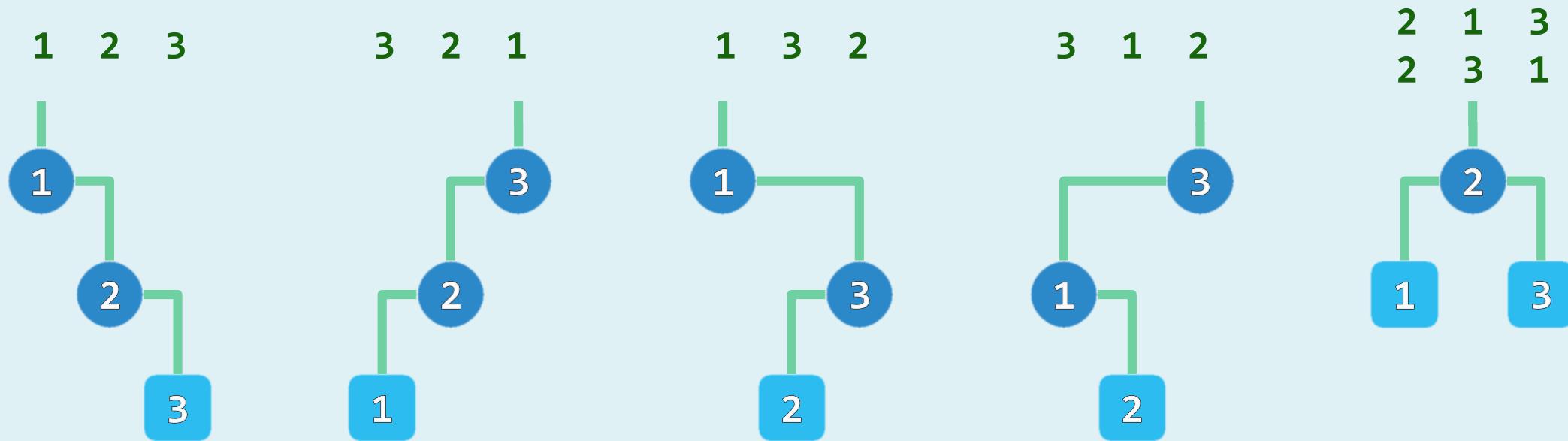
- ❖ 由以上的实现与分析，BST主要接口search()、insert()和remove()的运行时间在最坏情况下，均线性正比于其高度 $O(h)$
- ❖ 若不能有效地控制树高，就无法体现出BST相对于向量、列表等数据结构的明显优势
- ❖ 比如在最（较）坏情况下，二叉搜索树可能彻底地（接近地）退化为列表此时的性能不仅没有提高，而且因为结构更为复杂，反而会（在常系数意义上）下降
- ❖ 那么，出现此类最坏、较坏情况的概率有多大？或者，从平均复杂度的角度看，二叉搜索树的性能究竟如何？
- ❖ 以下按两种常用的随机统计口径，就此做一分析和对比

随机生成：n个词条 { $e_1, e_2, e_3, \dots e_n$ } 按随机排列 $\sigma = (e_{i_1}, e_{i_2}, e_{i_3}, \dots e_{i_n})$ 依次插入



- ❖ 若假设计各排列出现的概率均等 ($1/n!$)，则BST平均高度为 $\Theta(\log n)$
- ❖ 的确，多数实际应用中的BST总体上都是如此生成和演化的
即便计入remove()，也可通过随机使用succ()和pred()，避免逐渐倾斜的趋势
- ❖ 然而问题恰恰在于，所有排列出现的概率的确均等吗？

随机组成： n 个互异节点，在遵守顺序性的前提下，随机确定拓扑联接关系



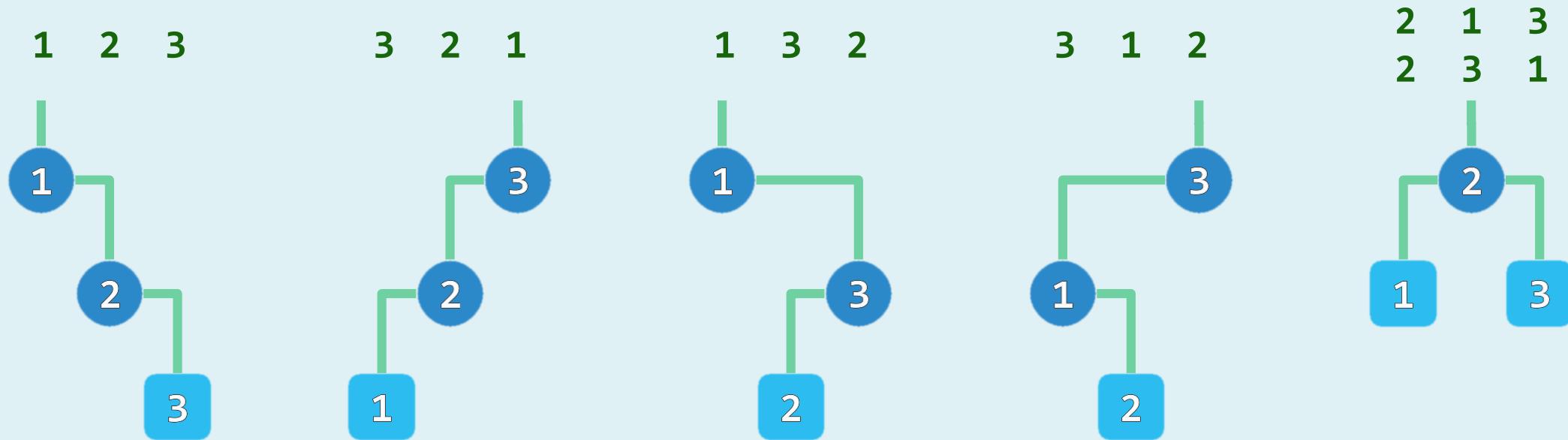
❖ 由 n 个互异节点随机**组成的**BST，若共计 $S(n)$ 棵，则有

$$S(n) = \sum_{k=1}^n S(k-1) \cdot S(n-k) = catalan(n) = (2n)!/(n+1)!/n!$$

❖ 假定所有BST**等概率**地出现，则其**平均高度**为 $\Theta(\sqrt{n})$

❖ 在Huffman编码之类的应用中，二叉树（尽管还不是BST）的确是逐渐**拼合**而成的

logn vs. sqrt(n)



- ❖ 两种口径所估计出的平均高度差异极大——谁更可信？谁更接近于真实情况？
- ❖ 后者未免太**悲观**，但前者则过于**乐观**：BST越低，**权重越大**；而最根本的原因在于…
- ❖ 理想随机在实际中**绝难出现**：局部性、关联性、(分段)单调性、(近似)周期性、…

较高甚至极高的BST频繁出现，不足为怪；平衡化处理很有必要！

二叉搜索树

平衡：理想与渐近

06 - C2

事当难处之时，只让退一步，便容易处矣

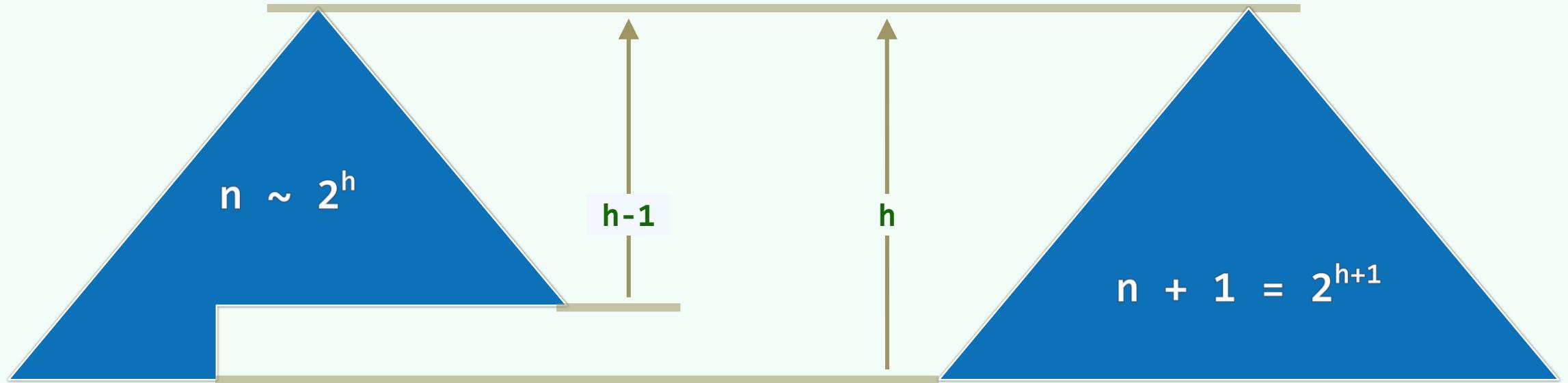
功到将成之候，若放松一着，便不能成矣

邓俊辉

deng@tsinghua.edu.cn

理想平衡

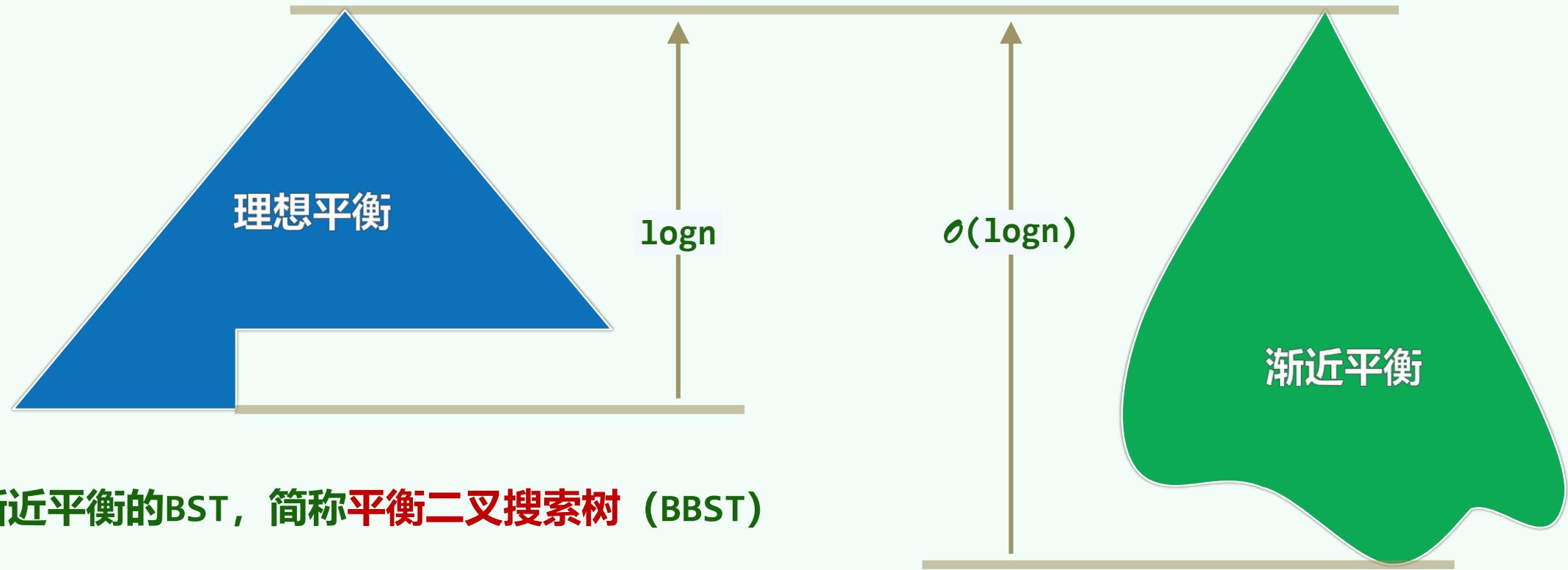
- ❖ 节点数目固定时，兄弟子树的高度越接近（平衡），全树也将倾向于更低
- ❖ 由 n 个节点组成的二叉树，高度不致低于 $\lfloor \log_2 n \rfloor$ ——达到这一下界时，称作**理想平衡**



- ❖ 大致相当于**完全树甚至满树**：叶节点只能出现于最底部的两层——条件过于苛刻

渐近平衡

- ❖ 理想平衡出现的概率极低、维护的成本过高，故须适当地放松标准
- ❖ 退一步海阔天空：高度渐近地不超过 $\mathcal{O}(\log n)$ ，即可接受



- ❖ 渐近平衡的BST，简称平衡二叉搜索树（BBST）

二叉搜索树

平衡：等价变换

06 - C3

邓俊辉

deng@tsinghua.edu.cn

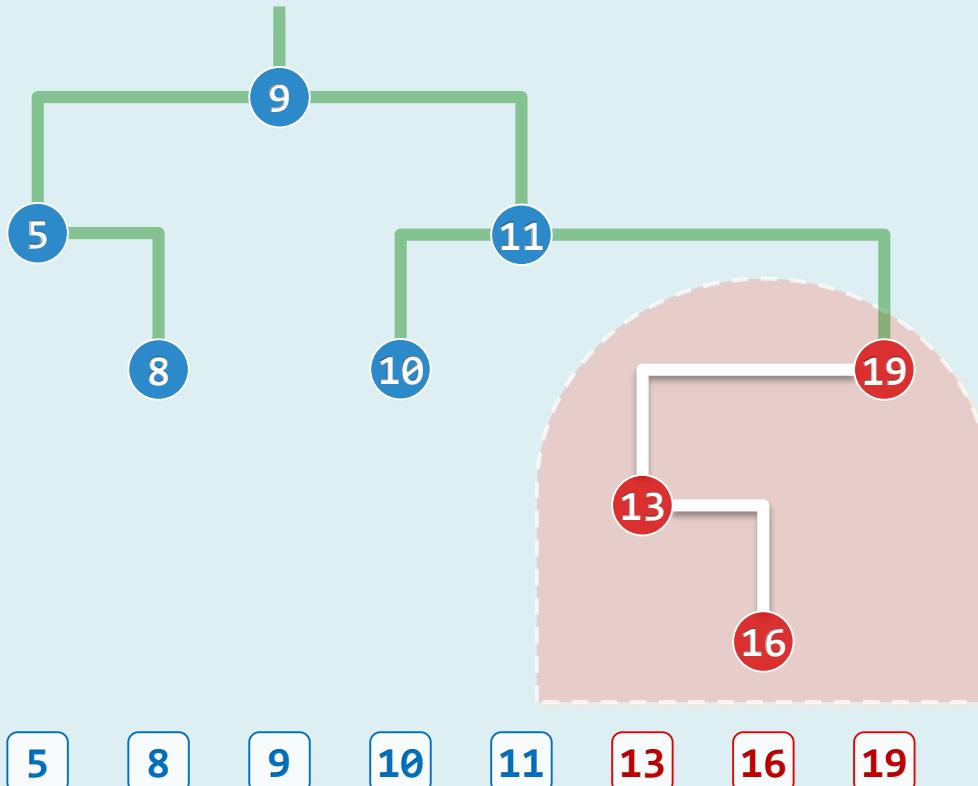
(清华) 校内无上下尊卑之分，当有长幼先后之序。

—— 周诒春

等价BST

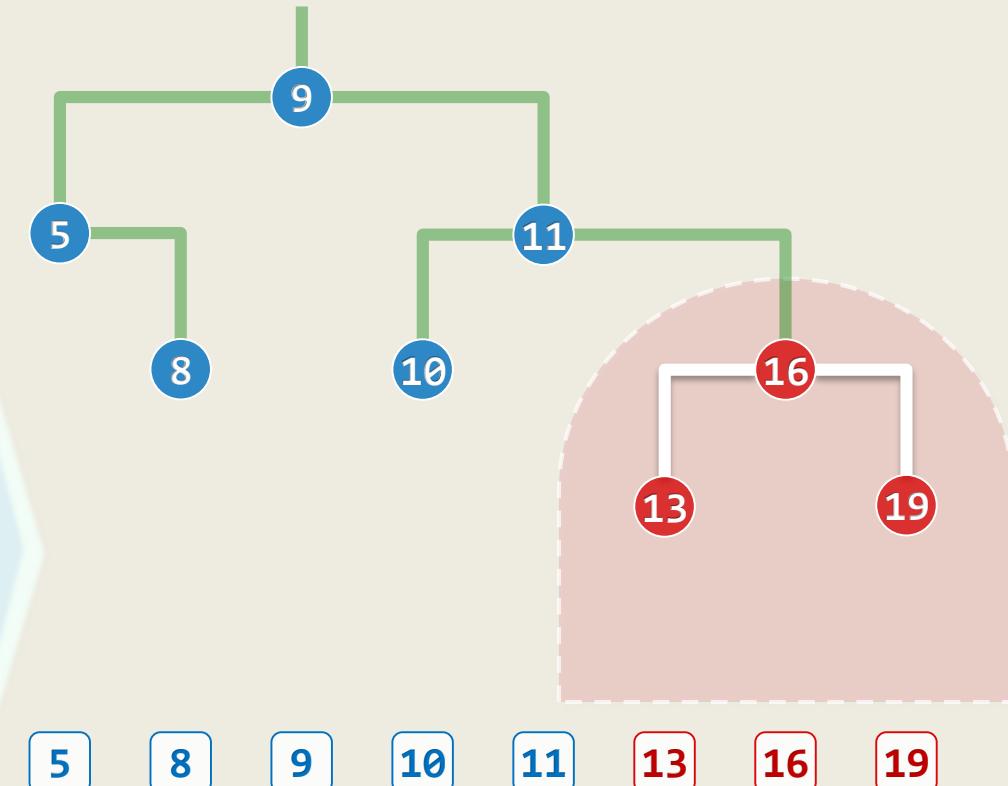
❖ 上下可变

联接关系不尽相同，承袭关系可能颠倒



❖ 左右不乱

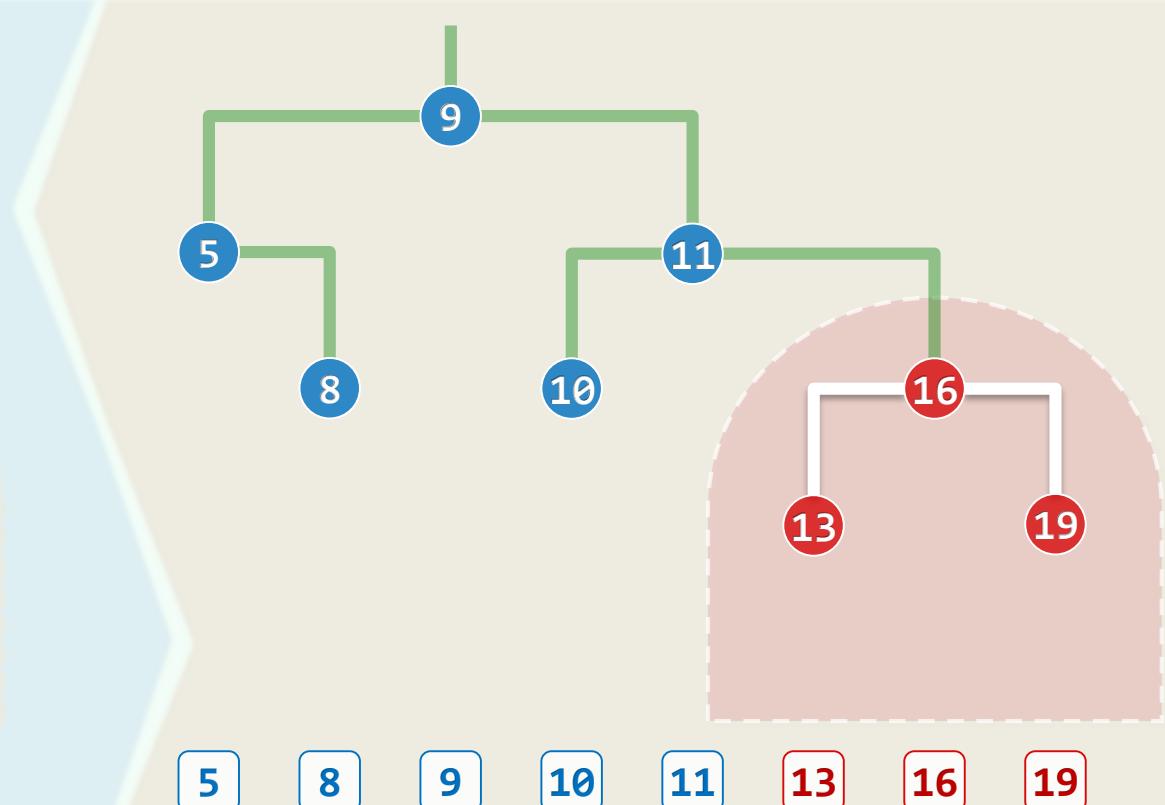
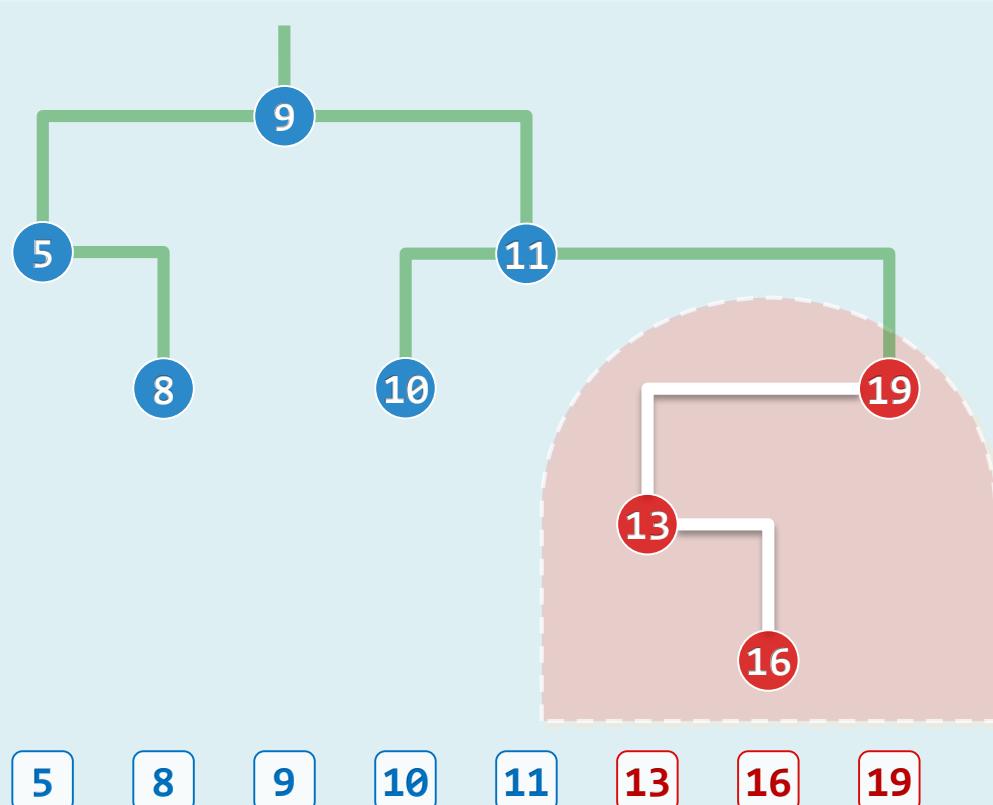
中序遍历序列完全一致，全局单调非降



限制条件 + 局部性

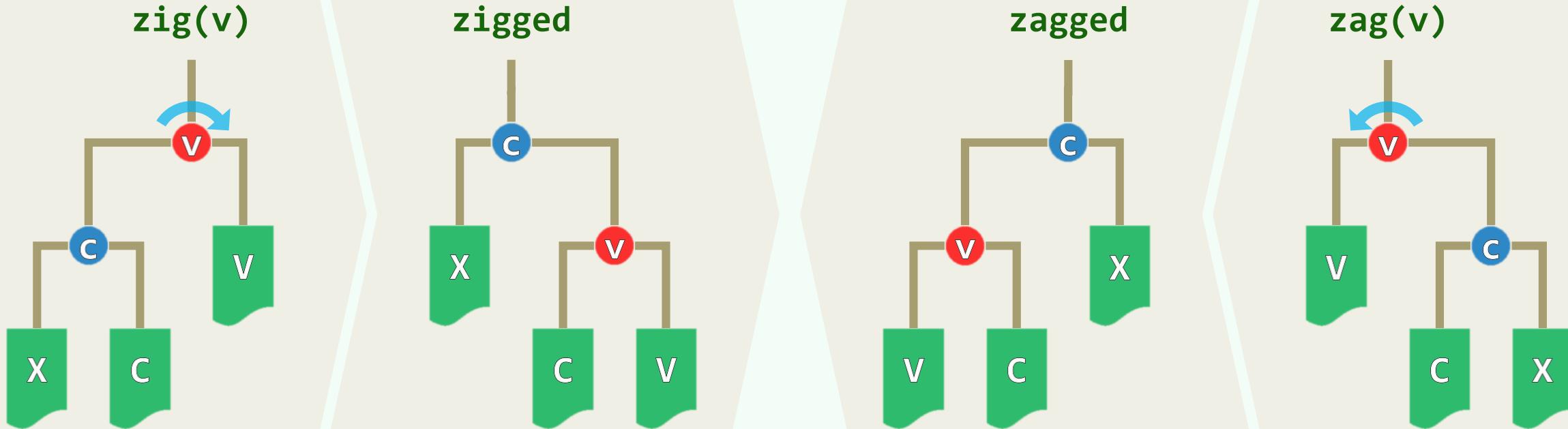
❖ 各种BBST都可视作BST的某一子集，相应地满足精心设计的**限制条件**

- 单次动态修改操作后，至多 $\mathcal{O}(\log n)$ 处局部不再满足限制条件（可能相继违反，未必同时）
- 可在 $\mathcal{O}(\log n)$ 时间内，使这些局部（以至全树）重新满足



等价变换 + 旋转调整：序齿不序爵

❖ 刚刚失衡的BST，必可迅速转换为一棵等价的BBST——为此，只需 $\mathcal{O}(\log n)$ 甚至 $\mathcal{O}(1)$ 次旋转



- ❖ zig和zag：仅涉及常数个节点，只需调整其间的联接关系；均属于局部的基本操作
- ❖ 调整之后：v/c深度加/减1，子（全）树高度的变化幅度，上下差异不超过1
- ❖ 实际上，经过不超过 $\mathcal{O}(n)$ 次旋转，等价的BST均可相互转化（习题解析[7-15]）

二叉搜索树

AVL树：渐近平衡

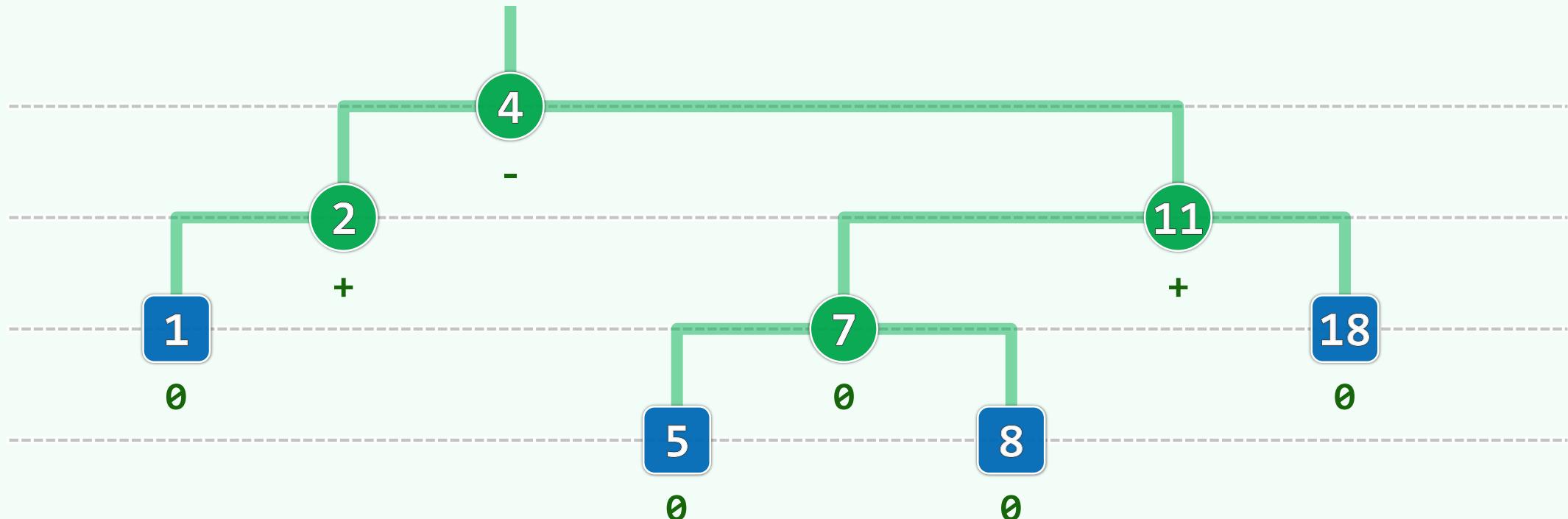
06 - D1

邓俊辉

deng@tsinghua.edu.cn

平衡因子

- ❖ Balance Factor: $balFac(v) = height(\text{lc}(v)) - height(\text{rc}(v))$
- ❖ G. Adelson-Velsky & E. Landis (1962): $\forall v \in \text{AVL}, |balFac(v)| \leq 1$



❖ AVL树未必理想平衡，但必然渐近平衡...

AVL = 漐近平衡

- ❖ 高度为 h 的AVL树，至少包含 $S(h) = fib(h + 3) - 1$ 个节点

为什么？

- ❖ 固定高度 h ，考查节点最少的AVL树...

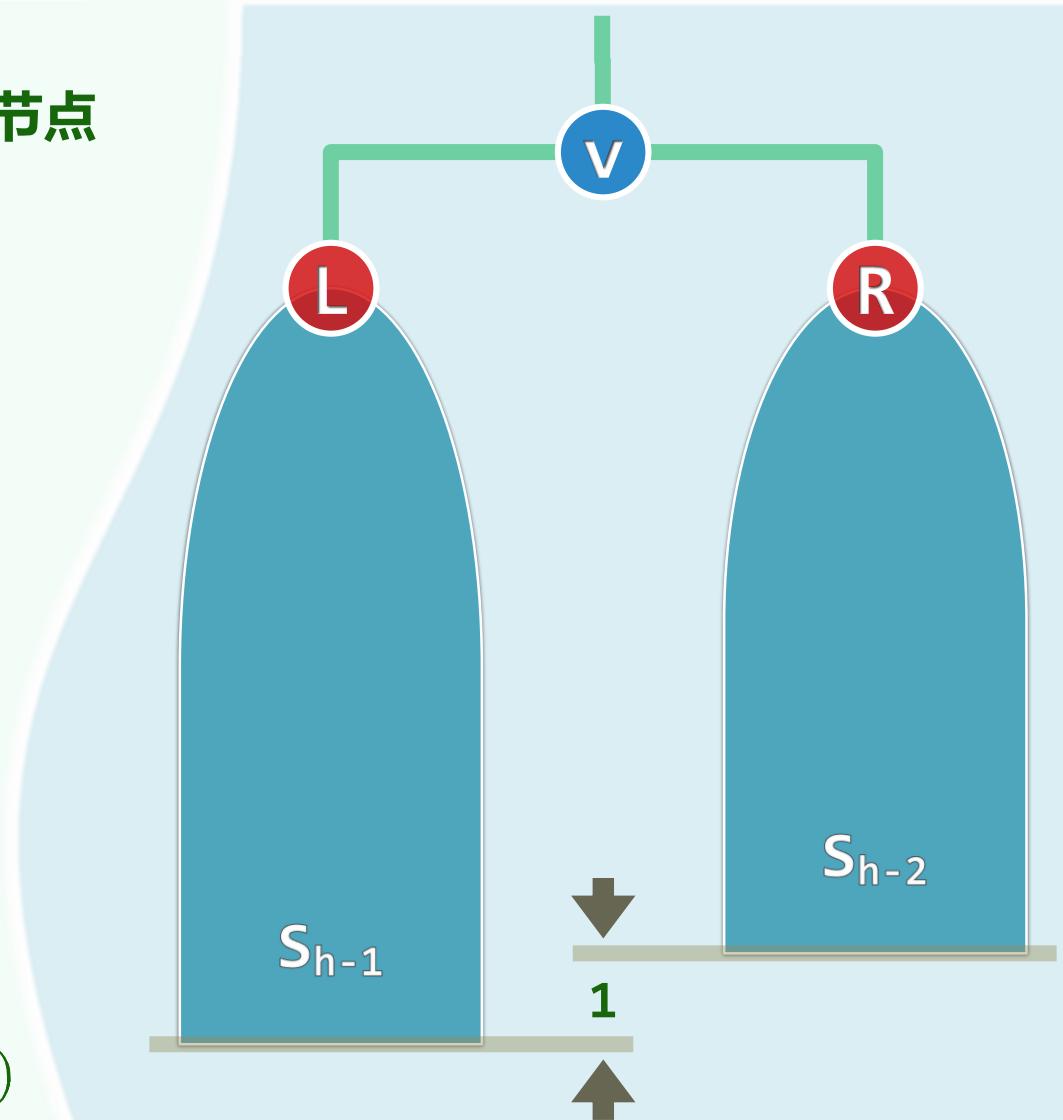
- ❖ 将其规模记作 $S(h)$

$$S(h) = 1 + S(h - 1) + S(h - 2)$$

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

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

- ❖ 反过来，由 n 个节点构成的AVL树，高度不超过 $\mathcal{O}(\log n)$



Fibonaccian Tree

❖ 高度为 h , 规模恰为 $S(h) = fib(h + 3) - 1$ 的AVL树

❖ 最“瘦”的、“临界”的AVL树

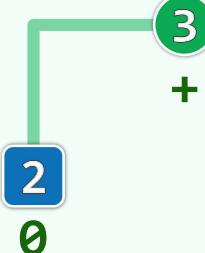
$$\begin{aligned} S(0) &= 1 \\ fib(3)-1 & \\ \end{aligned}$$



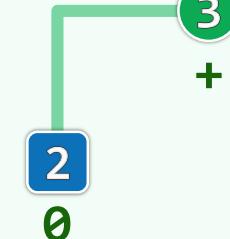
1, 2, 4, 7, 12, 20, 33, 54, 88, 143, ...

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

$$\begin{aligned} S(1) &= 2 \\ fib(4)-1 & \\ \end{aligned}$$



+



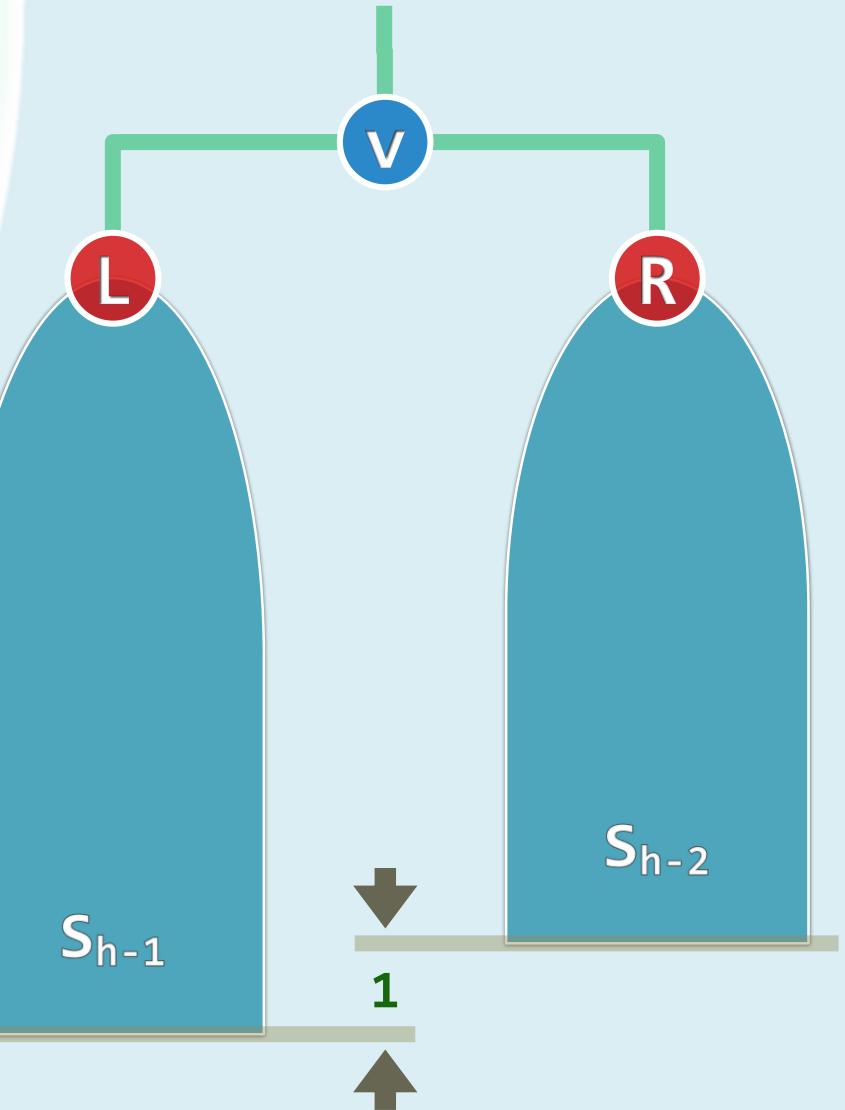
$$\begin{aligned} S(2) &= 4 \\ fib(5)-1 & \\ \end{aligned}$$



+

+

0



二叉搜索树

AVL树：重平衡

06-D2

邓俊辉

deng@tsinghua.edu.cn

接口

```
#define Balanced(x)  ( stature( (x).lc ) == stature( (x).rc ) ) //理想平衡

#define BalFac(x)  (stature( (x).lc ) - stature( (x).rc ) ) //平衡因子

#define AvlBalanced(x)  ( ( -2 < BalFac(x) ) && (BalFac(x) < 2 ) ) //AVL平衡条件

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）

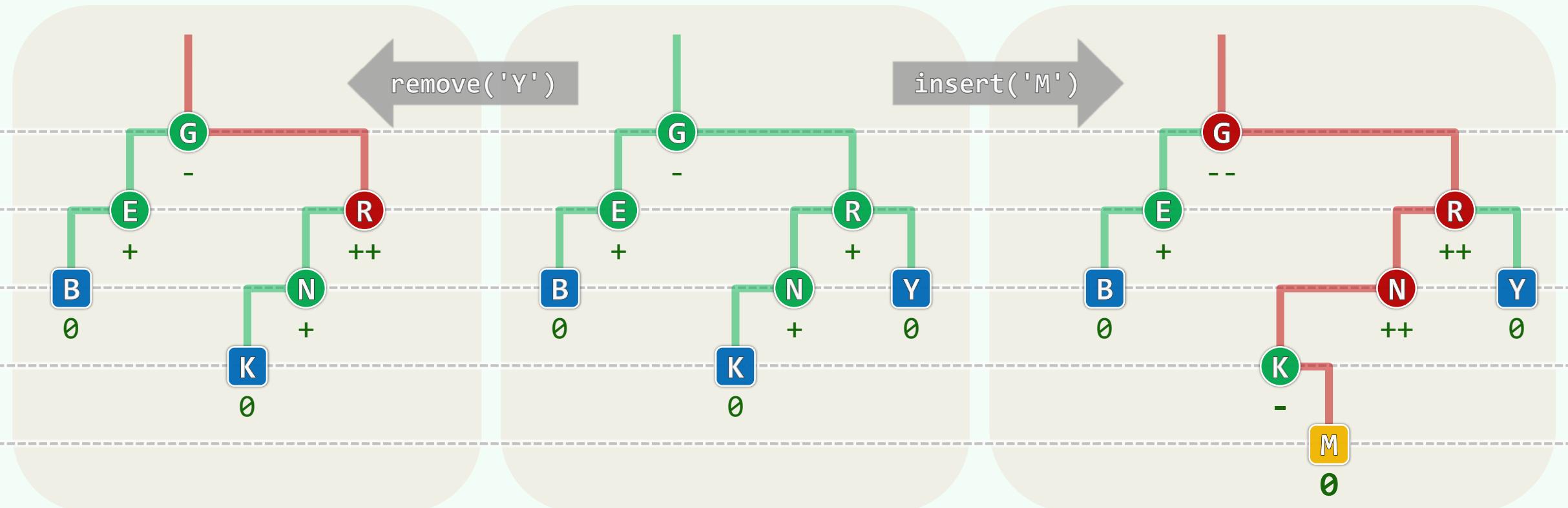
bool remove( const T & ); //删除（重写）

};

};
```

失衡

❖ 按BST规则动态操作之后，AVL平衡性可能破坏 //当然，只涉及到祖先

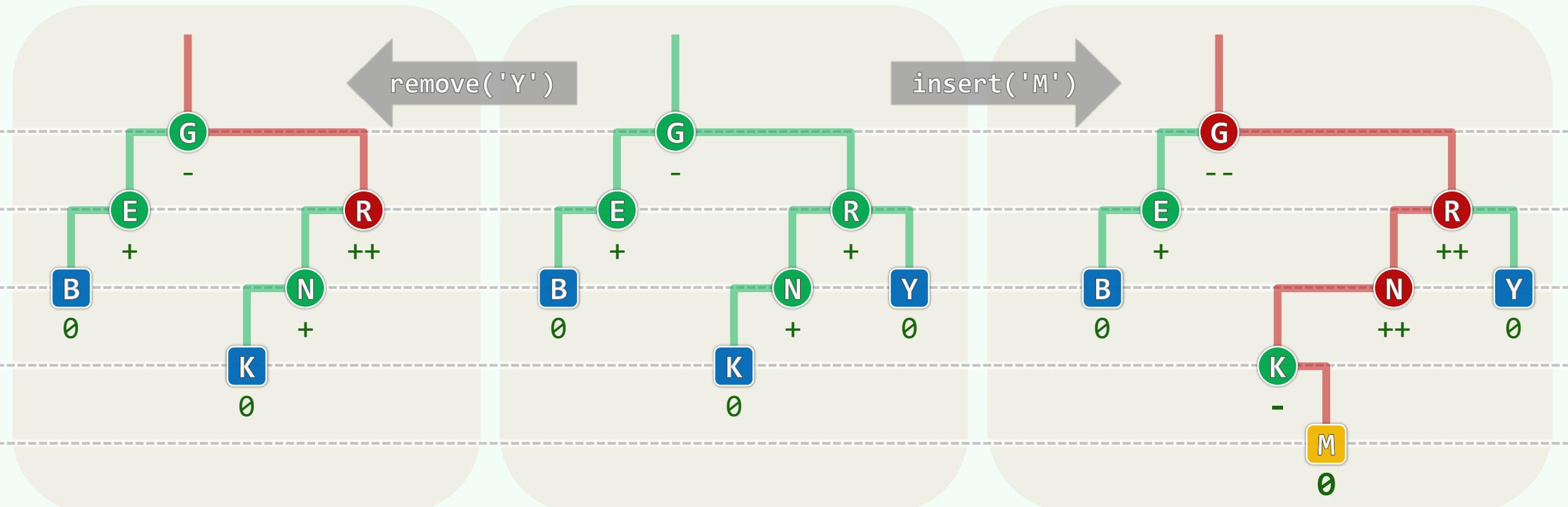


❖ 插入：从祖父开始，每个祖先都有可能失衡，且可能同时失衡！ //复杂？

❖ 删除：从父亲开始，每个祖先都有可能失衡，但至多一个！为什么？ //简单？

重平衡

❖ 如何恢复平衡? 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 $\mathcal{O}(1)$ 时间

快速性: 在每一深度只需检查并旋转至多一次 //共 $\mathcal{O}(\log n)$ 次

二叉搜索树

AVL树：插入

06 - D3

邓俊辉

deng@tsinghua.edu.cn

名不正则言不顺，言不顺则事不成

没有什么是一次旋转解决不了的

如果有，那就两次

单旋：黄色方块恰好存在其一

❖ 同时可有多个失衡节点

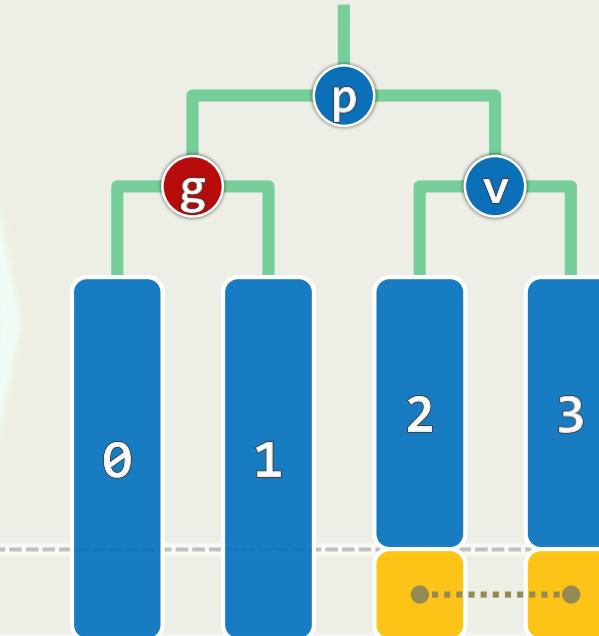
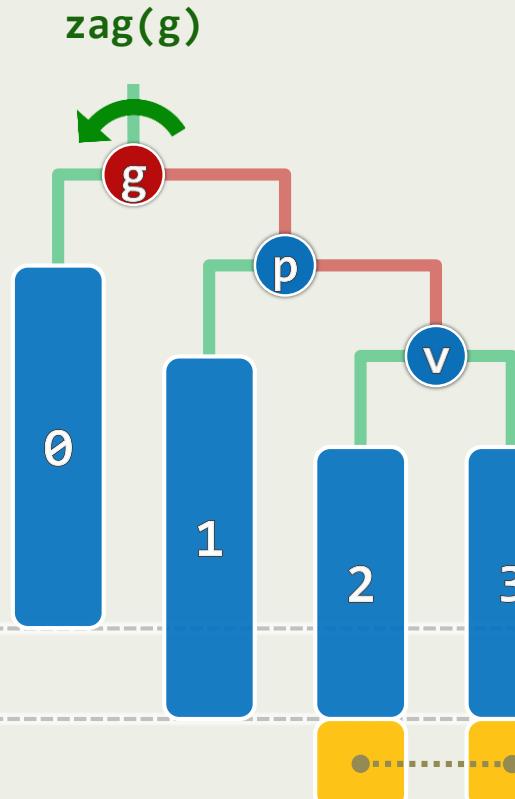
最低者g不低于x的祖父

❖ g经单旋调整后复衡

子树高度复原

❖ 更高祖先也必平衡

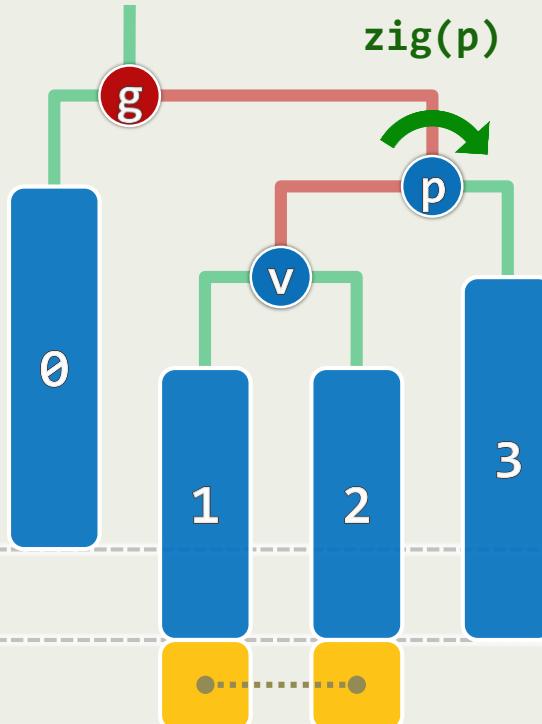
全树复衡



双旋

❖ 同时可有多个失衡节点

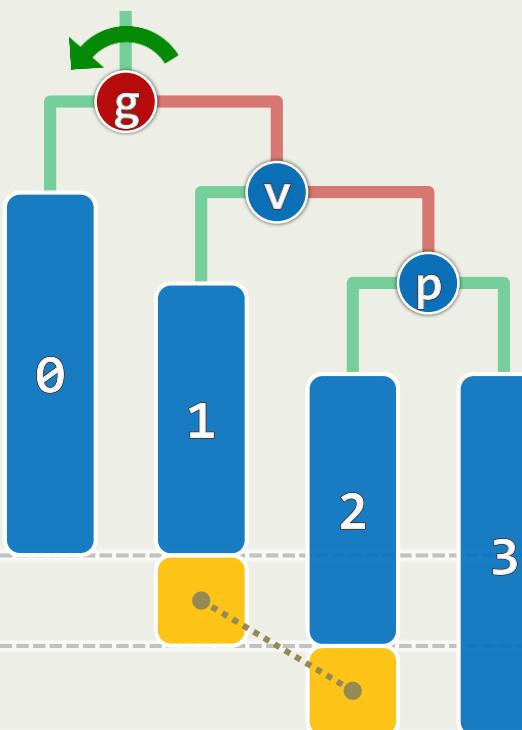
最低者g不低于x的祖父



❖ g经单旋调整后复衡

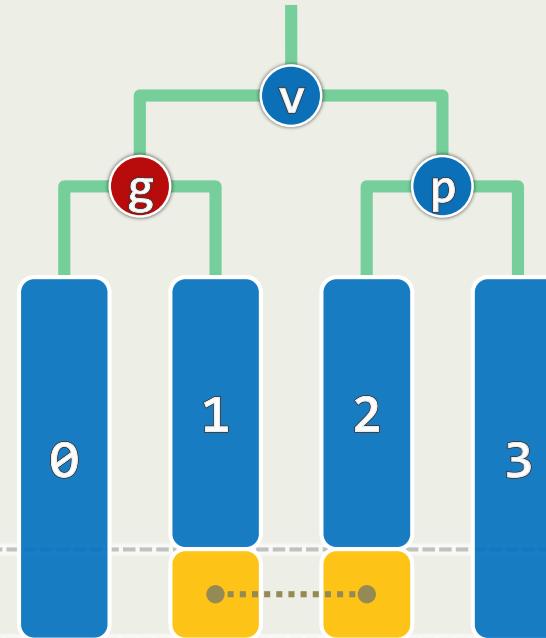
子树高度复原

zag(g)



❖ 更高祖先也必平衡

全树复衡



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  
    // 此时，若x的父亲_hot增高，则祖父有可能失衡  
    for ( BinNodePosi<T> g = _hot; g; g = g->parent ) //从_hot起，逐层检查各代祖先g  
        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; //局部子树复衡后，高度必然复原；其祖先亦必如此，故调整结束  
        } else //否则 (g仍平衡)  
            updateHeight( g ); //只需更新其高度 (注意：即便g未失衡，高度亦可能增加)  
    return xx; //返回新节点位置  
}
```

二叉搜索树

AVL树：删除

06 - D4

邓俊辉

deng@tsinghua.edu.cn

单旋：黄色方块至少存在其一；红色方块可有可无

❖ 同时至多一个失衡节点g,

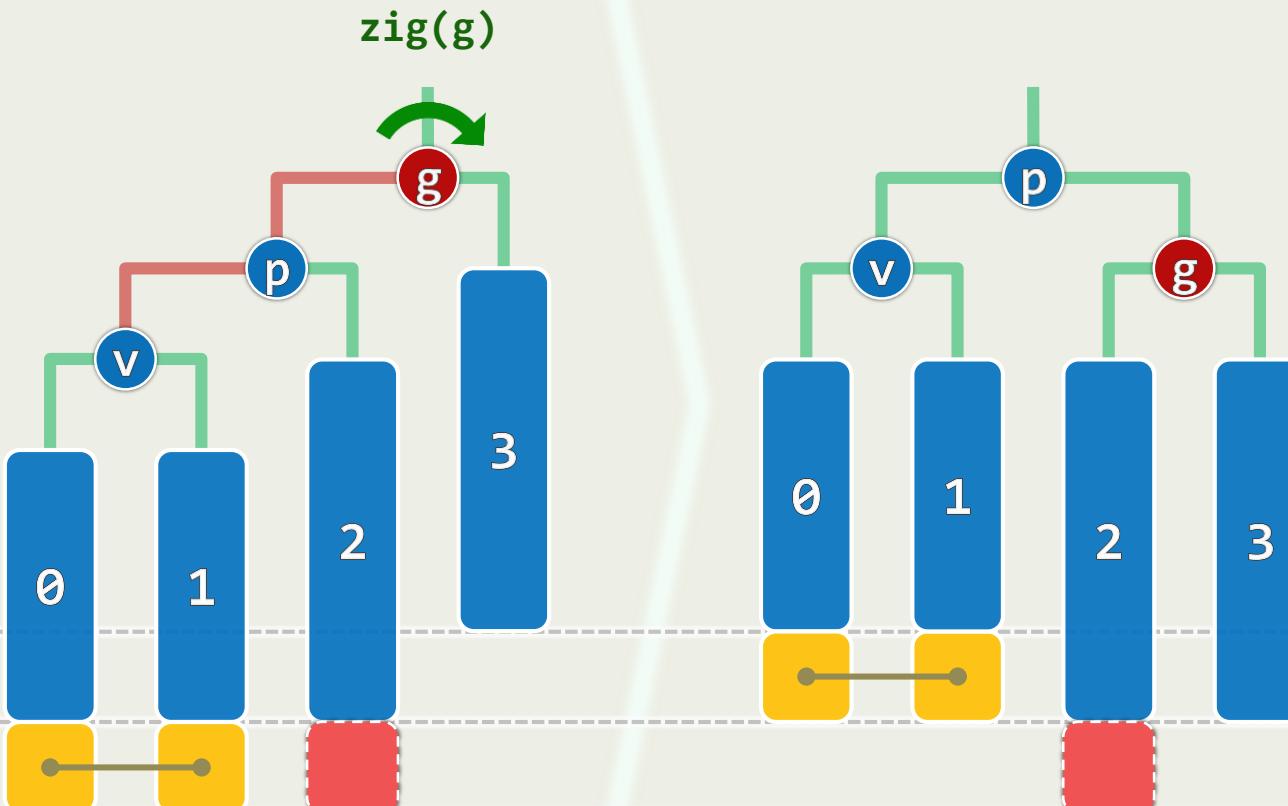
首个可能就是x的父亲_hot

❖ 复衡后子树高度未必复原

更高祖先仍可能随之失衡

❖ 失衡可能持续向上传播

最多需做 $\mathcal{O}(\log n)$ 次调整



双旋

❖ 同时至多一个失衡节点g,

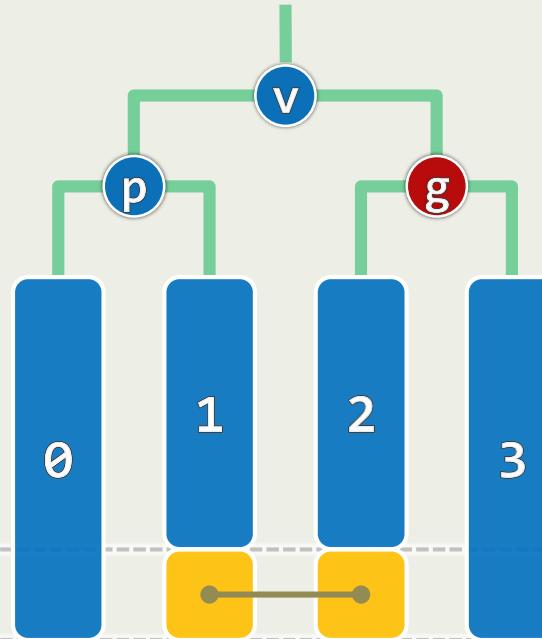
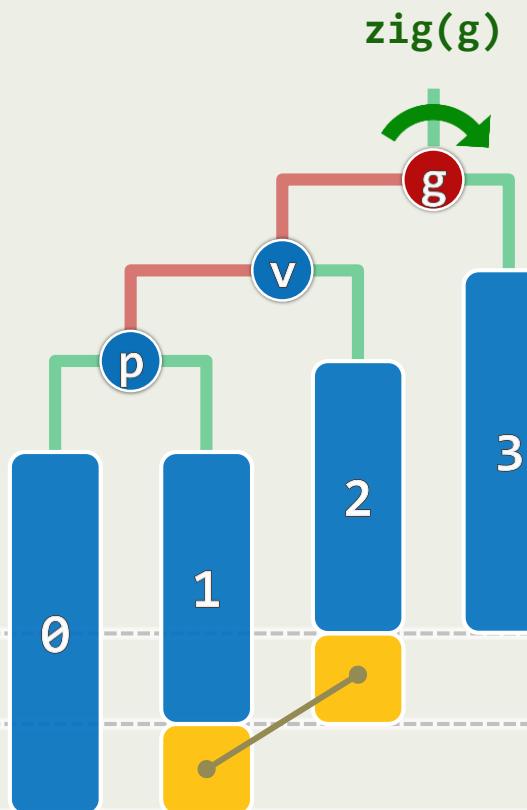
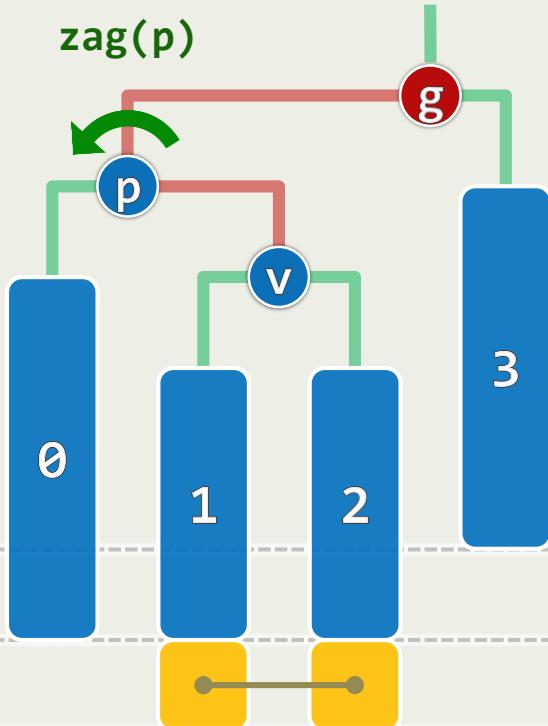
首个可能就是x的父亲_hot

❖ 复衡后子树高度不能复原

更高祖先仍可能随之失衡

❖ 失衡可能持续向上传播

最多需做 $\mathcal{O}(\log n)$ 次调整



实现

```
❖ 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 ); //更新高度（注意：即便g未曾失衡或已恢复平衡，高度均可能降低）  
    } //可能需做过 $\Omega(\log n)$ 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  
}
```

二叉搜索树

AVL树：(3+4)-重构

邓俊辉

deng@tsinghua.edu.cn

06 - D5

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： $T_0 < T_1 < T_2 < T_3$

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

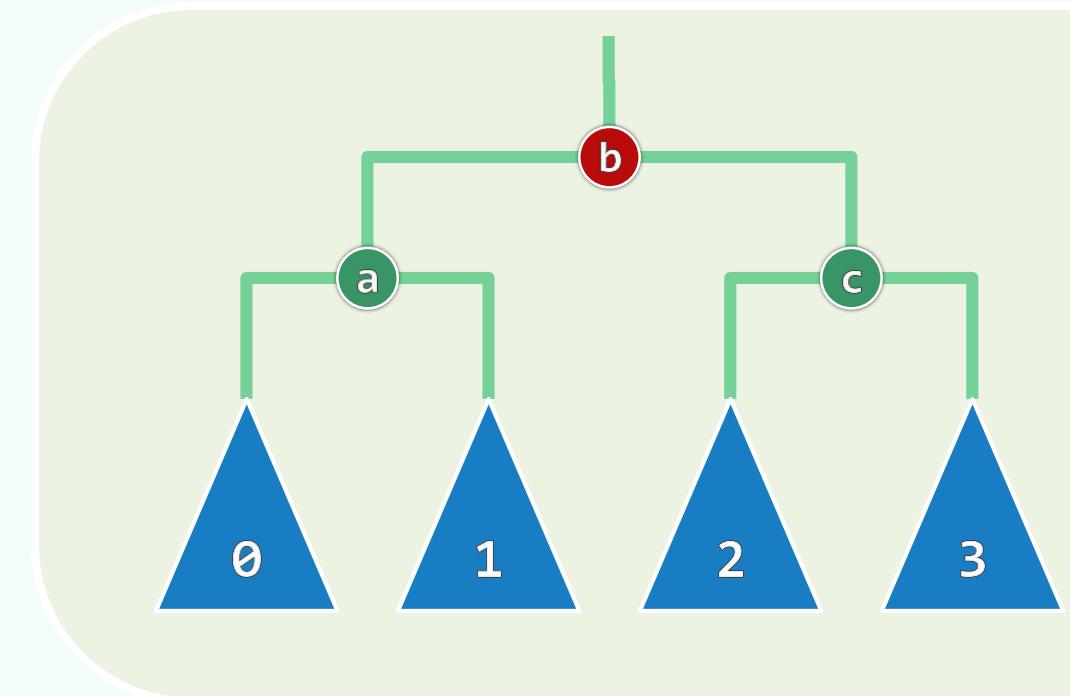
$$\text{lc}(b) = a$$

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

$$\text{lT}(a) = T_0 \quad \text{rT}(a) = T_1 \quad \text{lT}(c) = T_2 \quad \text{rT}(c) = T_3$$

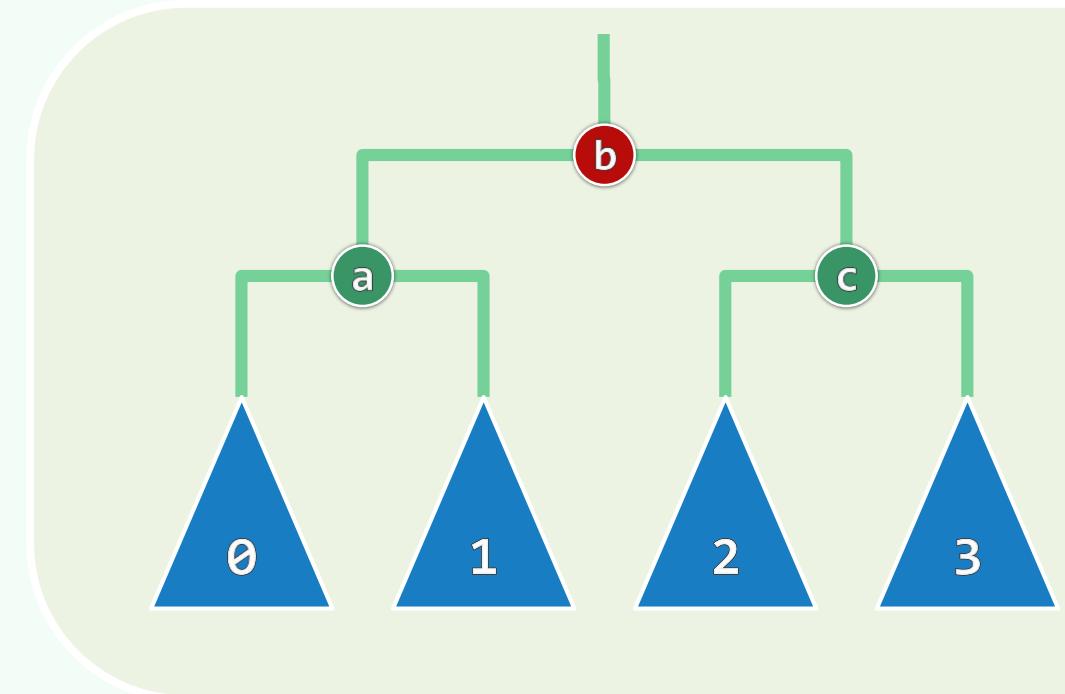
❖ 将原先以 g 为根的子树 S ，替换为一棵新子树 S'

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



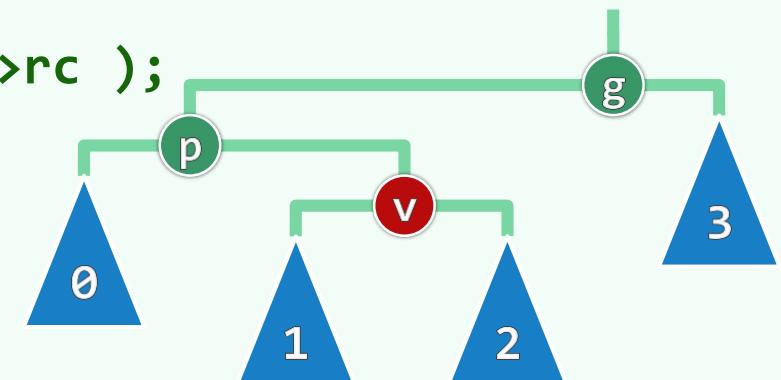
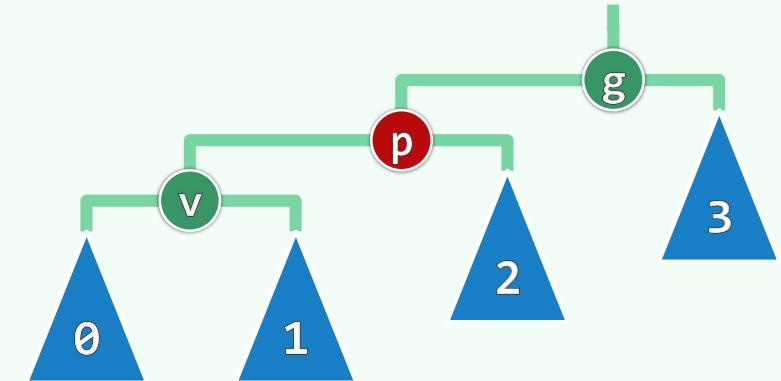
实现

```
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)

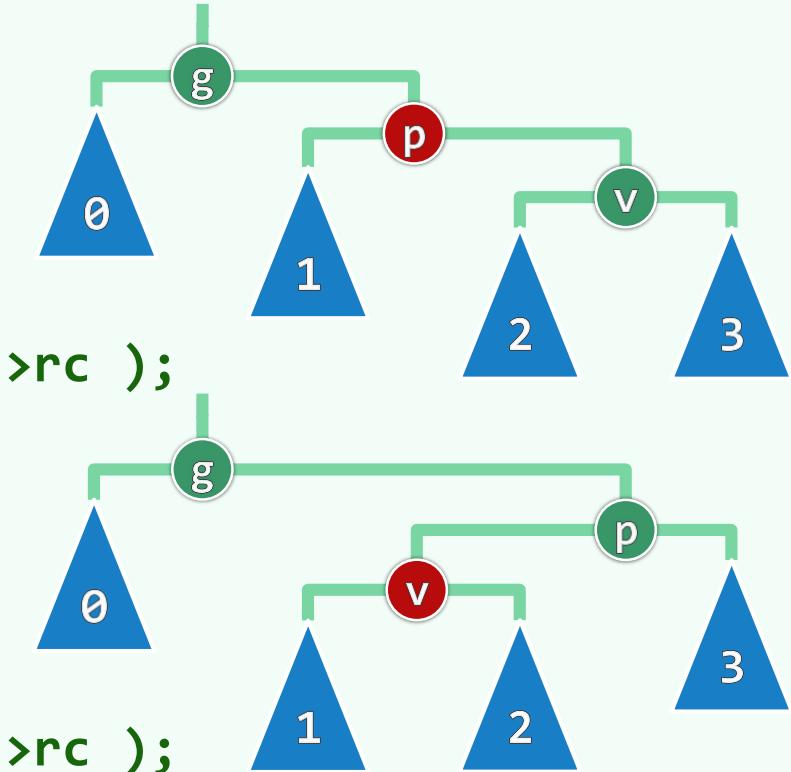
```
template<typename T> BinNodePosi<T> BST<T>::rotateAt( BinNodePosi<T> v ) {  
    BinNodePosi<T> p = v->parent, g = p->parent;  
    if ( IsLChild( * p ) ) //zig  
        if ( IsLChild( * v ) ) { //zig-zig  
            p->parent = g->parent;  
            return connect34( v, p, g,  
          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



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 $\Theta(\log n)$

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

❖ 有没有更好的结构呢？ //保持兴趣