

高级搜索树

伸展树：逐层伸展

08-A7

我要一步一步往上爬
在最高点乘着叶片往前飞

邓俊辉

deng@tsinghua.edu.cn

局部性/Locality: 刚被访问过的数据, 极有可能很快地再次被访问

❖ 这一现象在信息处理过程中屡见不鲜...

[Pfaff-2004] Performance Analysis Of BSTs In System Software

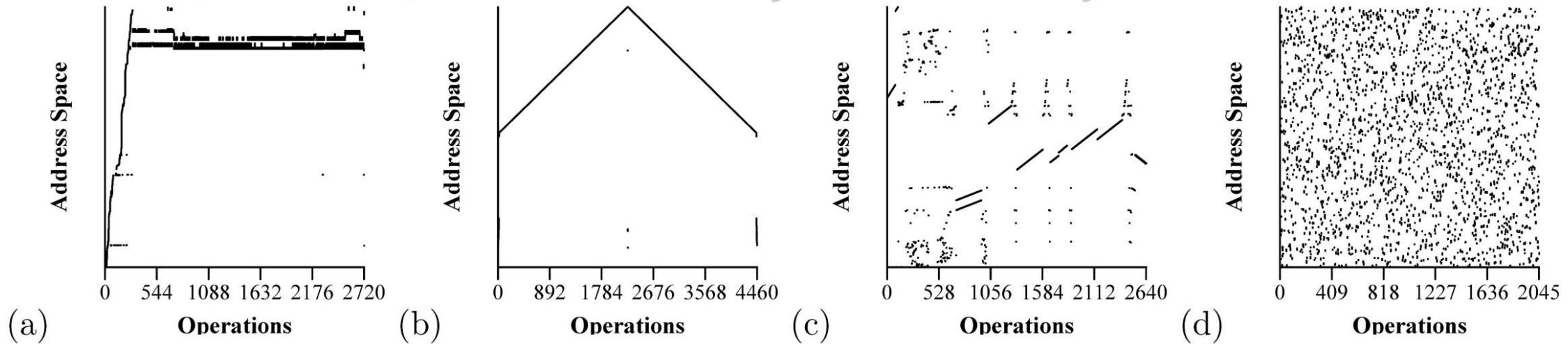
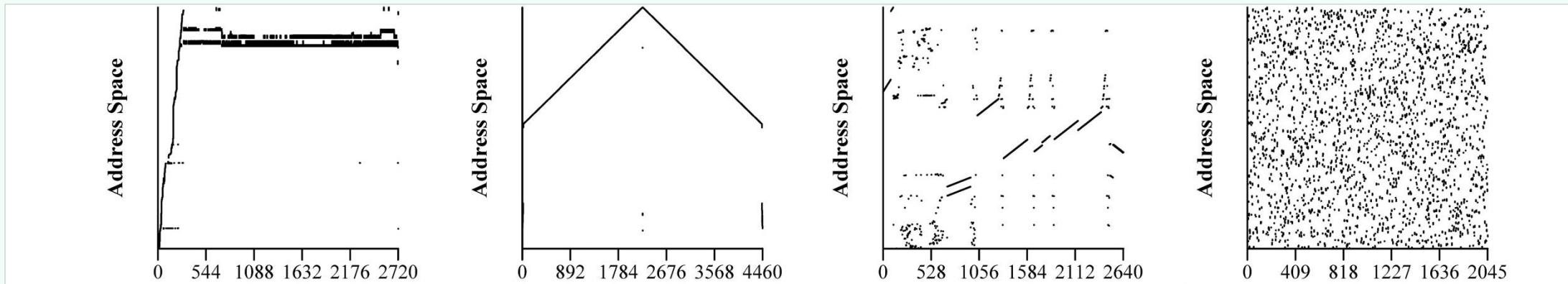


Figure 2: Call sequences in (a) Mozilla 1.0, (b) VMware GSX Server 2.0.1, (c) squid running under User-Mode Linux 2.4.18.48, and (d) random test sets. Part (b) omits one `mmap-munmap` pair for memory region 0x20000000 to 0x30000000 and (c) omits address space gaps; the others are complete.

BST的局部性

- ❖ **时间：**刚被访问过的**节点**，极有可能很快地**再次被访问**
- ❖ **空间：**下一将要访问的**节点**，极有可能就在刚被访问过**节点的附近**
- ❖ 对AVL**连续的m次查找** ($m \gg n$)，共需 $\mathcal{O}(m \cdot \log n)$ **时间**——能否利用**局部性加速**？



- ❖ **自适应链表：**节点一旦被访问，随即移动到**最前端**
- ❖ **模仿：**bst的节点一旦被访问，随即调整到**树根**
- ❖ **难点：**如何实现这种**调整**？**调整过程自身的复杂度**如何控制？

逐层伸展

❖ 节点 v 一旦被访问

随即被**推送至根**

❖ 与其说“推”，不如说“爬”

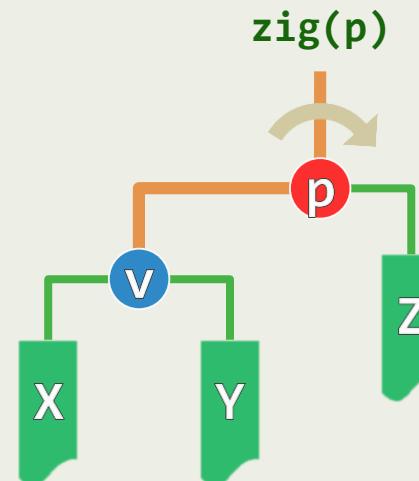
一步一步地往上爬

❖ 自下而上，逐层**旋转**

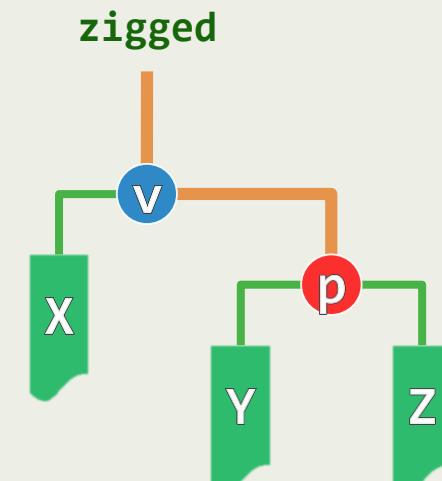
- `zig(v->parent)`

- `zag(v->parent)`

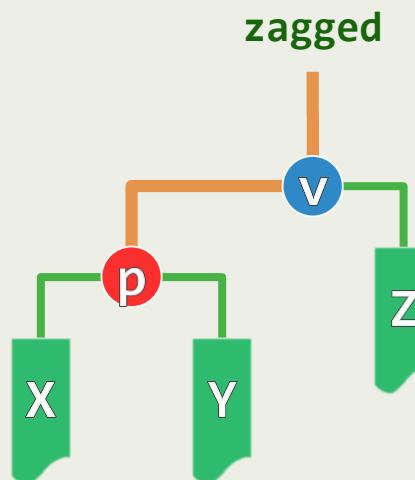
`zig(p)`



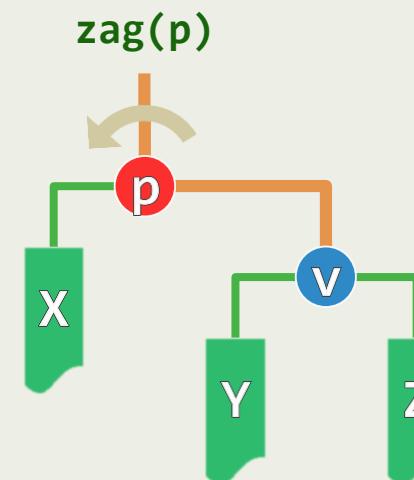
`zipped`



`zagged`



`zag(p)`



实例

❖ 伸展过程的效率

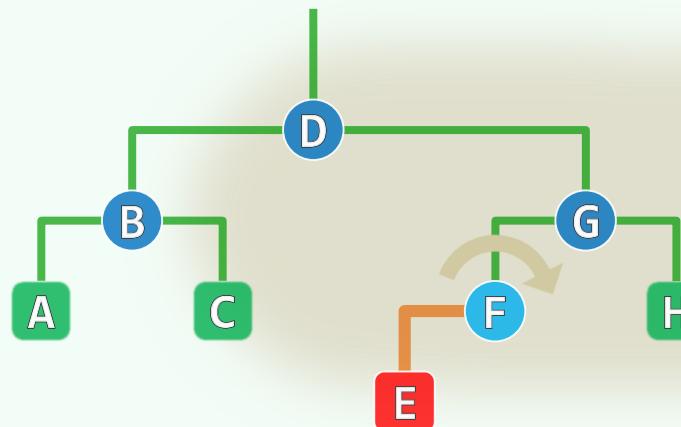
是否足够地高?

❖ 这取决于

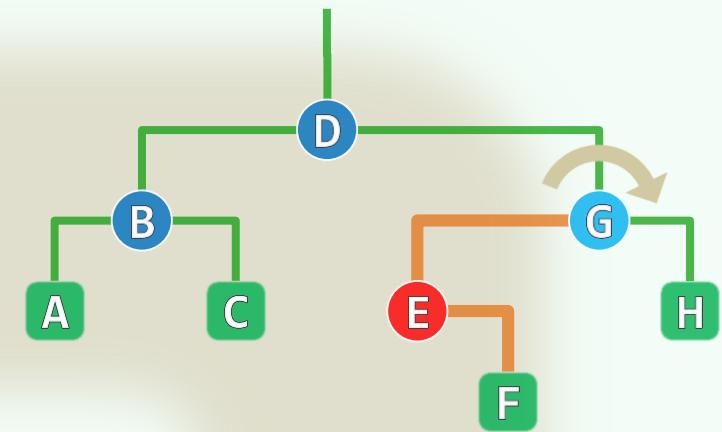
- 树的初始形态和

- 节点的访问次序

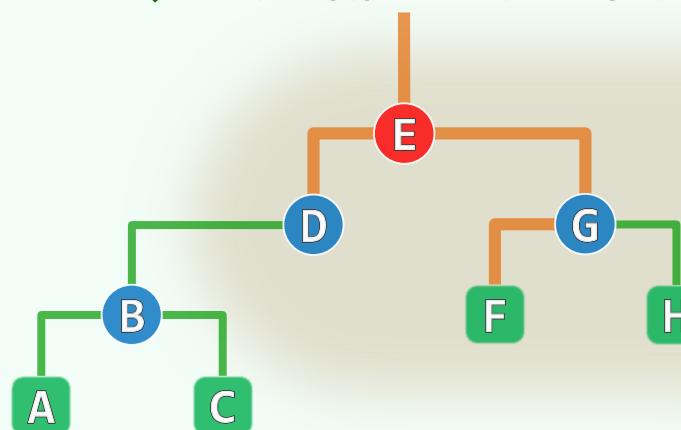
a) 访问E之后, 做zig(F)



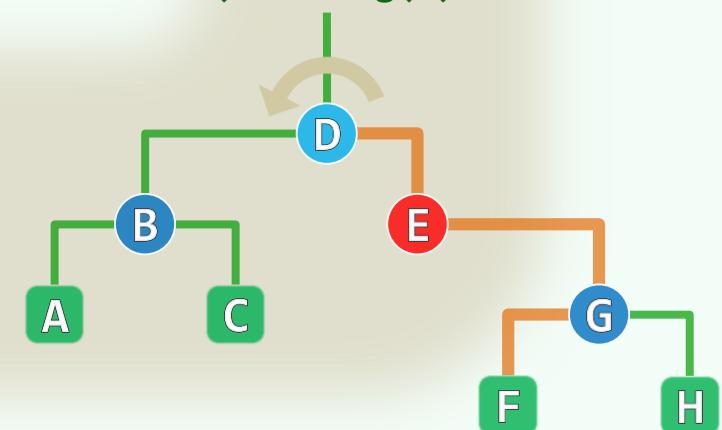
b) 继而zig(G)



d) 经3次旋转, E最终调整至树根



c) 继而zag(D)



最坏情况

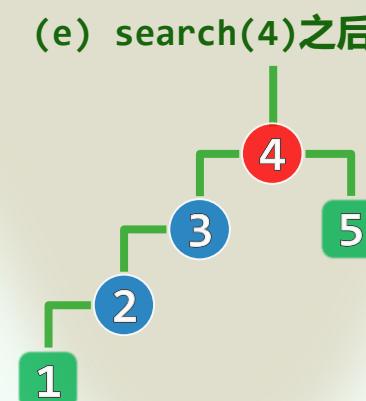
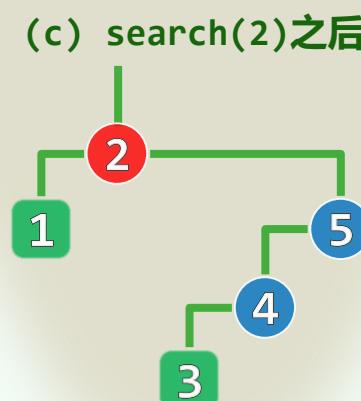
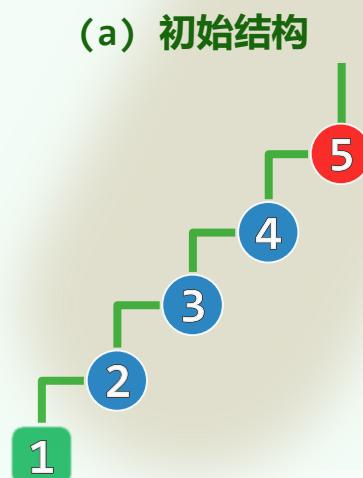
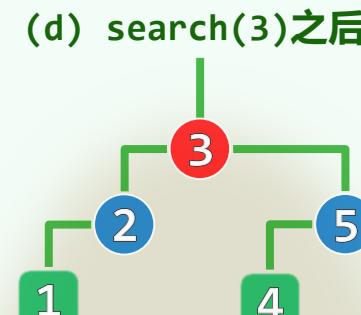
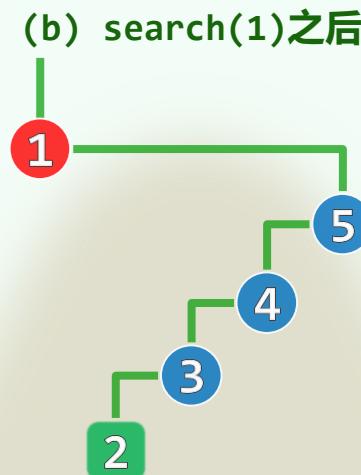
❖ 旋转次数

呈周期性的算术级数

❖ 每一周期累计 $\Omega(n^2)$

分摊 $\Omega(n)$

❖ 怎么破?



高级搜索树

伸展树：双层伸展

08-A2

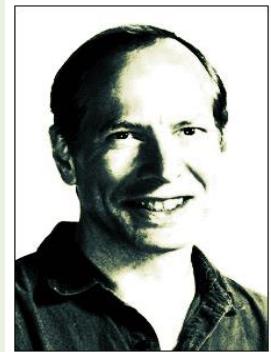
邓俊辉

deng@tsinghua.edu.cn

贾政道：“不用全打开，怕叠起来倒费事。” 詹光便与冯紫英一层一层折好收拾。

双层伸展

❖ Self-Adjusting Binary Trees

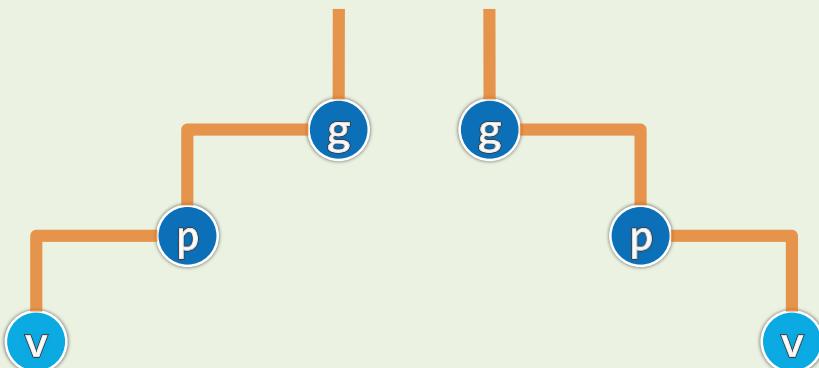


D. D. Sleator

R. E. Tarjan

J. ACM, 32:652-686, 1985

❖ 构思的精髓：向上追溯两层，而非一层



❖ 反复考察祖孙三代：

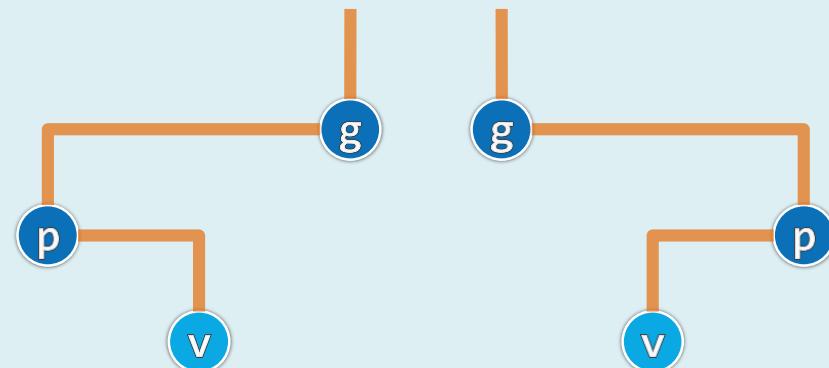
$g = \text{parent}(p)$, $p = \text{parent}(v)$, v

❖ 根据它们的相对位置，经两次旋转

使 v 上升两层，成为（子）树根

❖ 如此，性能的确会有改善？

❖ 具体地，应该如何旋转？



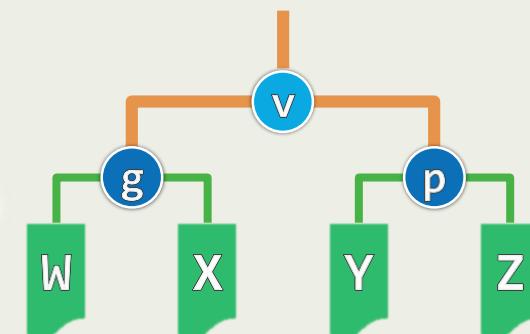
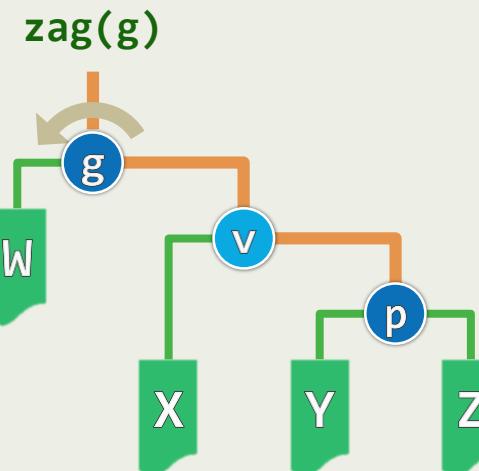
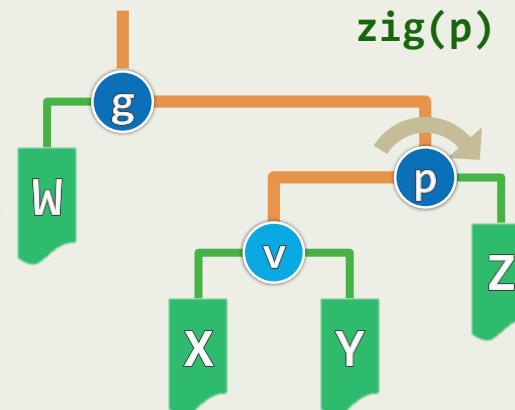
zig-zag / zag-zig

❖ 此时的v按中序遍历次序居中

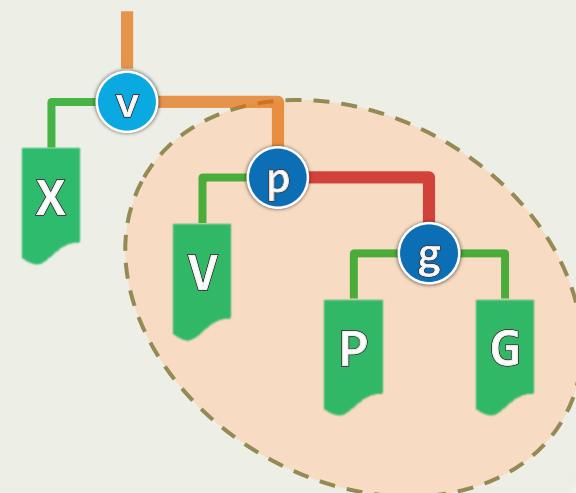
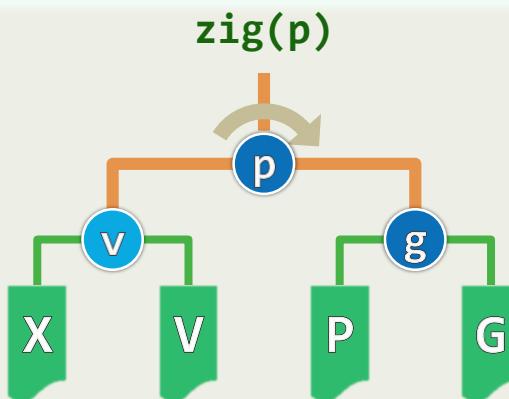
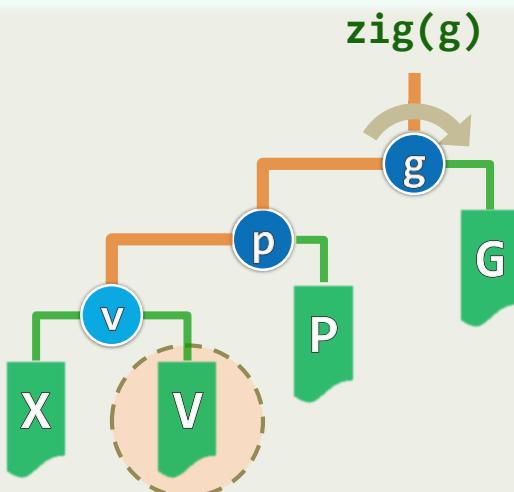
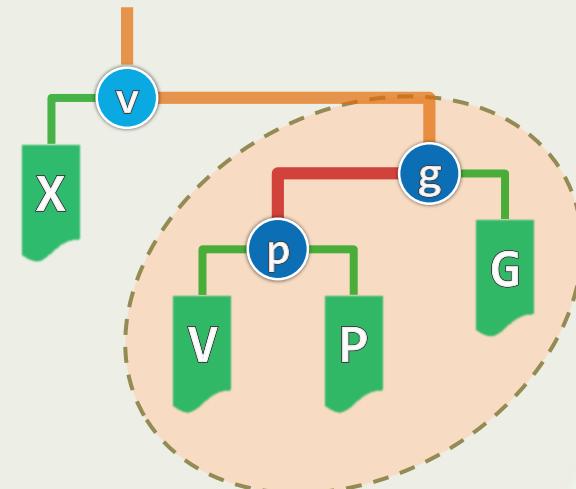
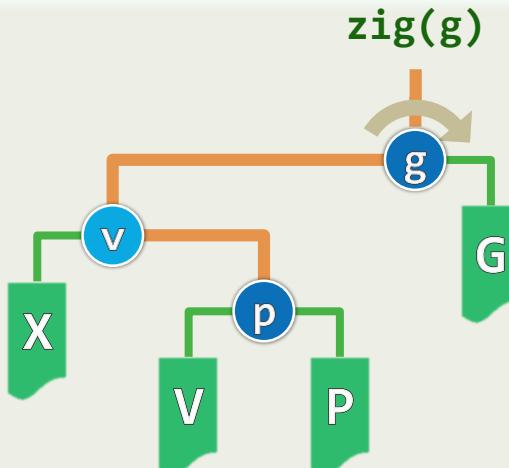
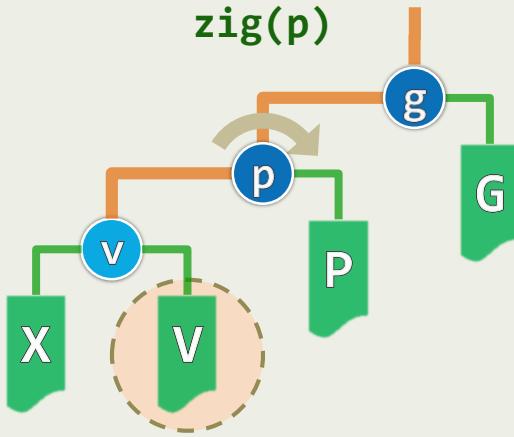
❖ 如此调整的效果，与逐层调整别无二致！

❖ 故若欲使之成为根，最终无非一种姿势

❖ 难道，就这样平淡无奇？



zig-zig / zag-zag



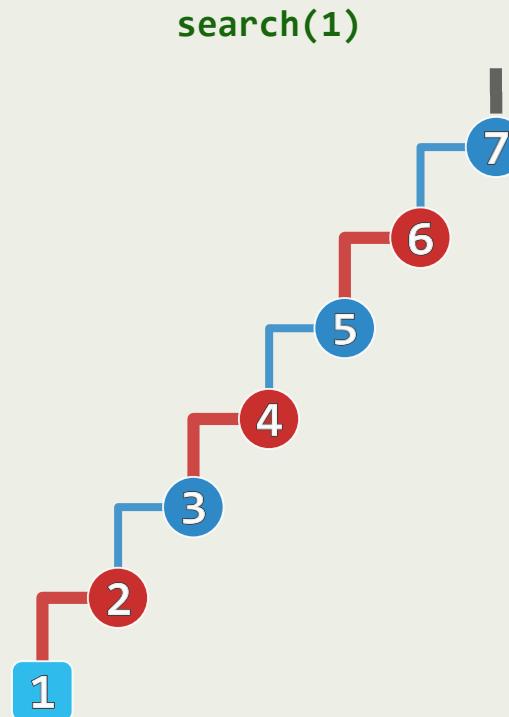
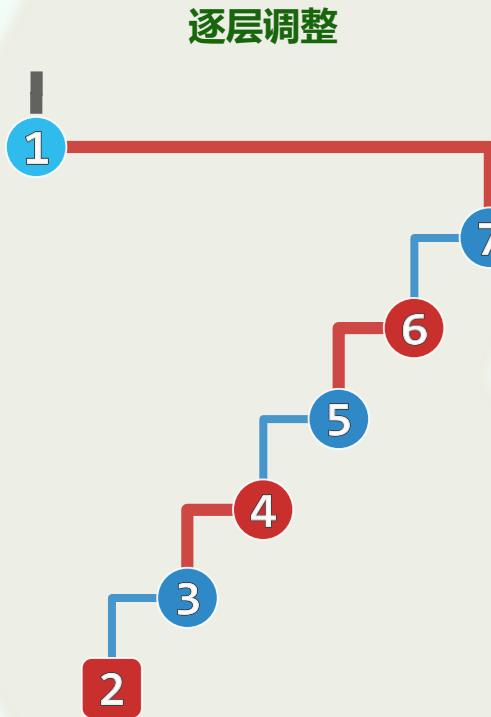
zig-zig / zag-zag

❖ 节点访问之后，对应路径的长度随即折半

//含羞草般的折叠效果

❖ 最坏情况不致持续发生！

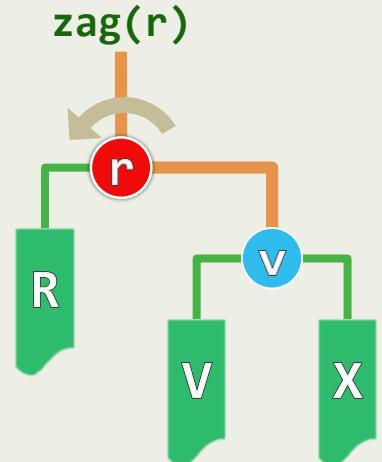
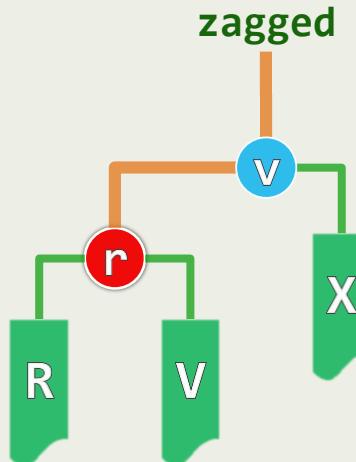
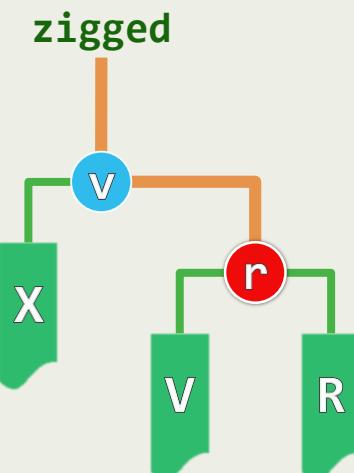
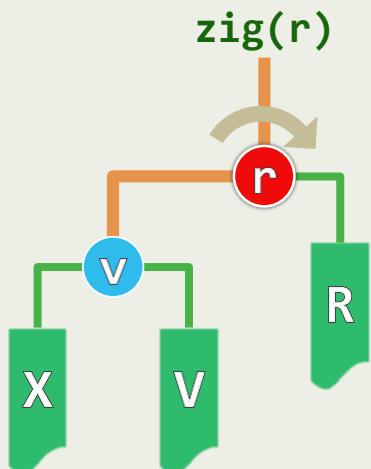
习题[8-2]：伸展操作分摊仅需 $\mathcal{O}(\log n)$ 时间



zig / zag

❖ 要是v只有父亲，没有祖父呢？

❖ 此时必有 $v.parent() == T.root()$



❖ 只需做单次旋转：zig(r)或zag(r)

❖ 好在，这种情况至多（在最后）出现一次

高级搜索树

伸展树：算法实现

08-A3

邓俊辉

deng@tsinghua.edu.cn

到了所在，住了脚，便把这驴似纸一般折叠起来，其厚也只比张纸，放在巾箱里面。

接口

```
template <typename T> class Splay : public BST<T> { //由BST派生
```

```
protected:
```

```
BinNodePosi<T> splay( BinNodePosi<T> v ); //将v伸展至根
```

```
public: //伸展树的查找也会引起整树的结构调整，故search()也需重写
```

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

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

```
bool remove( const T & e ); //删除（重写）
```

```
};
```

伸展算法：总体

```
template <typename T> BinNodePosi<T> Splay<T>::splay( BinNodePosi<T> v ) {  
  
    if ( ! v ) return NULL; BinNodePosi<T> p; BinNodePosi<T> g; //父亲、祖父  
  
    while ( (p = v->parent) && (g = p->parent) ) {  
  
        /* 自下而上，反复地双层伸展 */  
  
    }  
  
    if ( p = v->parent ) { /* 若p果真是根，只需再额外单旋一次 */ }  
  
    v->parent = NULL; return v; //伸展完成，v抵达树根  
  
}
```

伸展算法：双层伸展

```
while ( (p = v->parent) && (g = p->parent) ) { //自下而上，反复双层伸展  
    BinNodePosi<T> gg = g->parent; //每轮之后，v都将以原曾祖父为父  
    if ( IsLChild( * v ) )  
        if ( IsLChild( * p ) ) { /* zig-zig */ } else { /* zig-zag */ }  
    else  
        if ( IsRChild( * p ) ) { /* zag-zag */ } else { /* zag-zig */ }  
    if ( !gg ) v->parent = NULL; //无曾祖父gg的v即为树根；否则，gg此后应以为  
    else ( g == gg->lC ) ? attachAsLC(v, gg) : attachAsRC(gg, v); //左或右孩子  
    updateHeight( g ); updateHeight( p ); updateHeight( v );  
}
```

伸展算法：举例 (zig-zig)

```
if ( IsLChild( * v ) )

    if ( IsLChild( * p ) ) { //zIg-zIg

        attachAsLC( p->rc, g ); //Y

        attachAsLC( v->rc, p ); //X

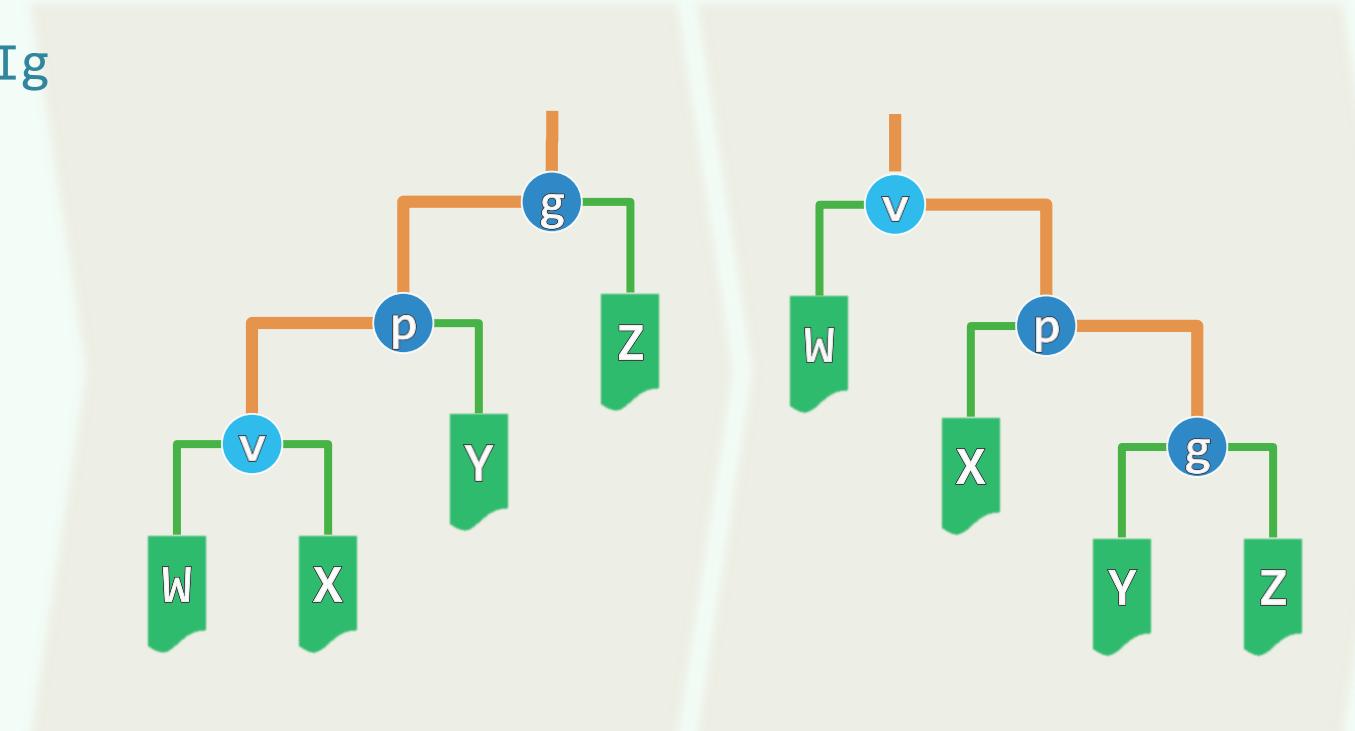
        attachAsRC( p, g );

        attachAsRC( v, p );

    } else { /* zIg-zAg */ }

else

    if ( IsRChild( * p ) ) { /* zAg-zAg */ }     else { /* zAg-zIg */ }
```



查找算法

- ❖ `template <typename T> BinNodePosi<T> & Splay<T>::search(const T & e) {`
 `// 调用标准BST的内部接口定位目标节点`

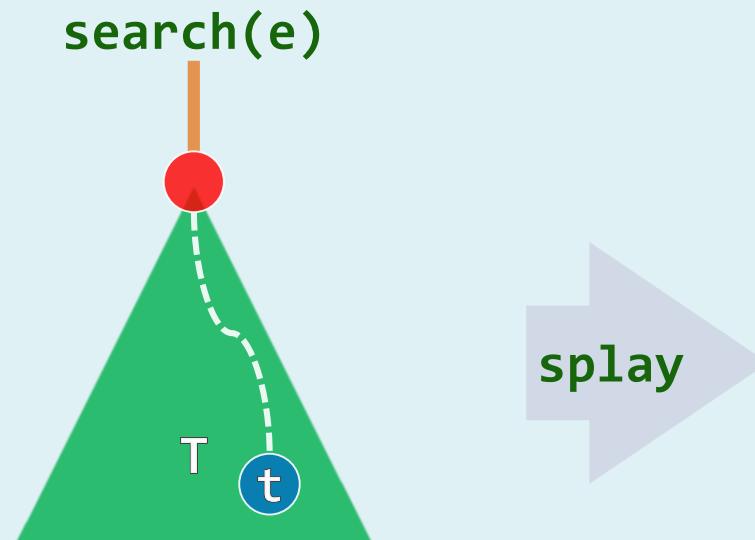
 `BinNodePosi<T> p = BST<T>::search(e);`
 `// 无论成功与否，最后被访问的节点都将伸展至根`

 `_root = splay(p ? p : _hot); //成功、失败`
 `// 总是返回根节点`

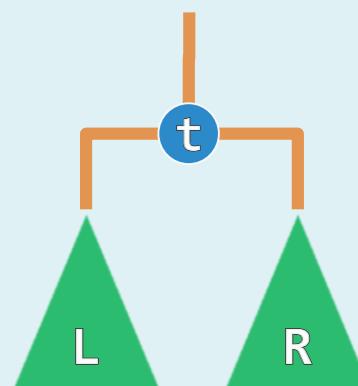
 `return _root;`
}
- ❖ 伸展树的查找，与常规BST::search()不同：很可能改变树的拓扑结构，不再属于静态操作

插入算法

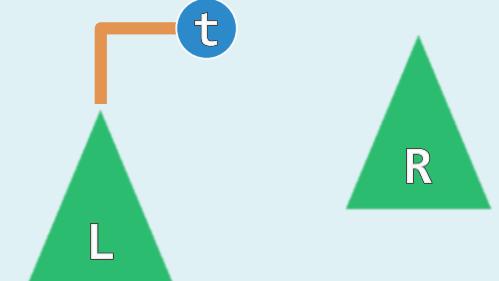
- ❖ 直观方法：先调用标准的`BST::search()`，再将新节点伸展至根
- ❖ `Splay::search()`已集成`splay()`，查找失败之后，`_hot`即是根
- ❖ 既如此，何不随即就在树根附近接入新节点？



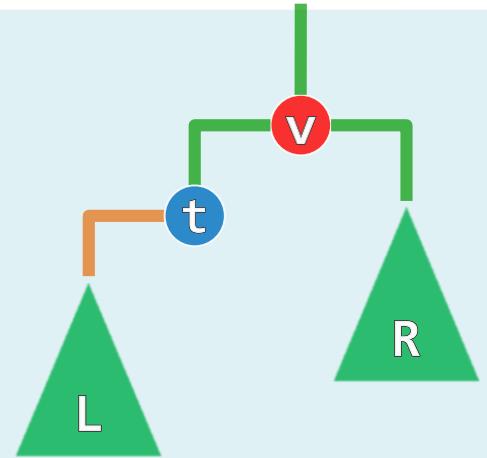
`splay`



`split`



`join`

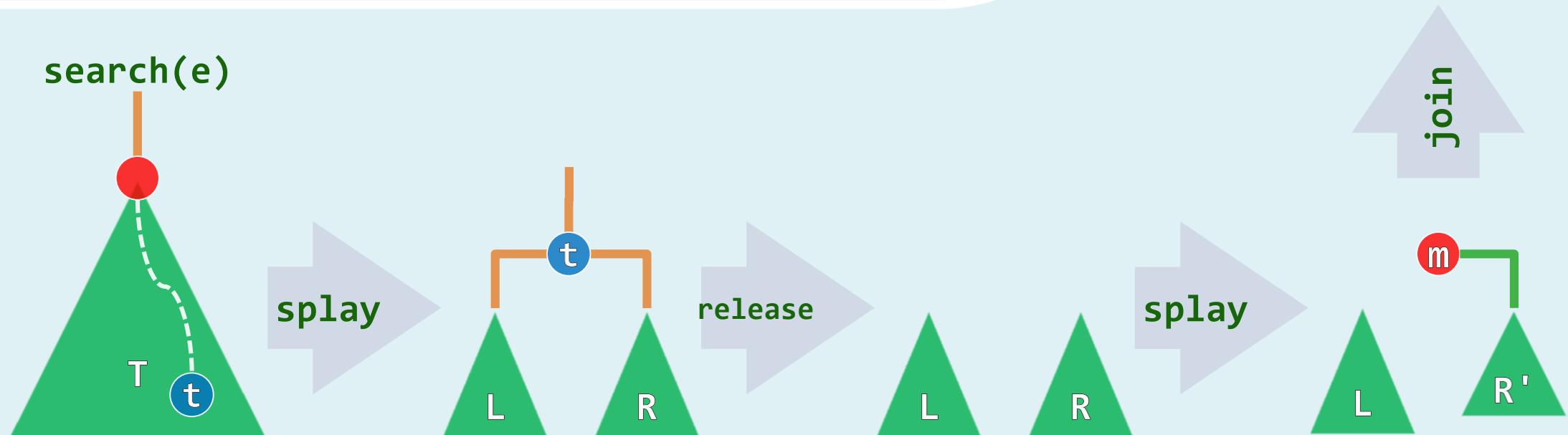
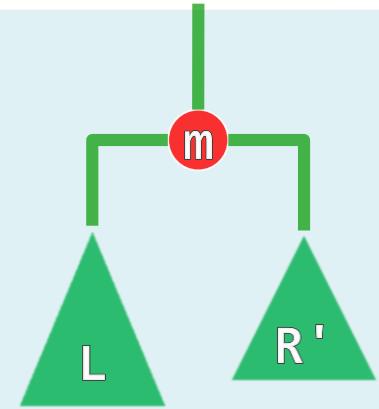


插入算法实现

```
template <typename T> BinNodePosi<T> Splay<T>::insert( const T & e ) {  
    if ( !_root ) { _size = 1; return _root = new BinNode<T>( e ); } //原树为空  
    BinNodePosi<T> t = search( e ); if ( e == t->data ) return t; //t若存在, 伸展至根  
    if ( t->data < e ) { //在右侧嫁接 (rc或为空, lc == t必非空)  
        t->parent = _root = new BinNode<T>( e, NULL, t, t->rc );  
        if ( t->rc ) { t->rc->parent = _root; t->rc = NULL; }  
    } else { //e < t->data, 在左侧嫁接 (lc或为空, rc == t必非空)  
        t->parent = _root = new BinNode<T>( e, NULL, t->lc, t );  
        if ( t->lc ) { t->lc->parent = _root; t->lc = NULL; }  
    }  
    _size++; updateHeightAbove( t ); return _root; //更新规模及t与_root的高度, 插入成功  
} //无论如何, 返回时总有_root->data == e
```

删除算法

- ❖ 直观方法：调用BST标准的删除算法，再将`_hot`伸展至根
- ❖ 注意到，Splay::search()成功之后，目标节点即是树根
- ❖ 既如此，何不随即就在树根附近完成目标节点的摘除...



删除算法实现

```
template <typename T> bool Splay<T>::remove( const T & e ) {  
    if ( !_root || ( e != search( e )->data ) ) return false; //若目标存在, 则伸展至根  
    BinNodePosi<T> L = _root->lc, R = _root->rc; release(_root); //记下子树后, 释放之  
    if ( !R ) { //若R空  
        if ( L ) L->parent = NULL; _root = L; //则L即是余树  
    } else { //否则  
        _root = R; R->parent = NULL; search( e ); //在R中再找e: 注定失败, 但最小节点必  
        if ( L ) L->parent = _root; _root->lc = L; //伸展至根, 故可令其以L作为左子树  
    }  
    _size--; if ( _root ) updateHeight( _root ); //更新记录  
    return true; //删除成功  
}
```

综合评价

❖ 无需记录高度或平衡因子；编程实现简单——优于AVL树

分摊复杂度 $\mathcal{O}(\log n)$ ——与AVL树相当

❖ 局部性强、缓存命中率极高时（即 $k \ll n \ll m$ ）

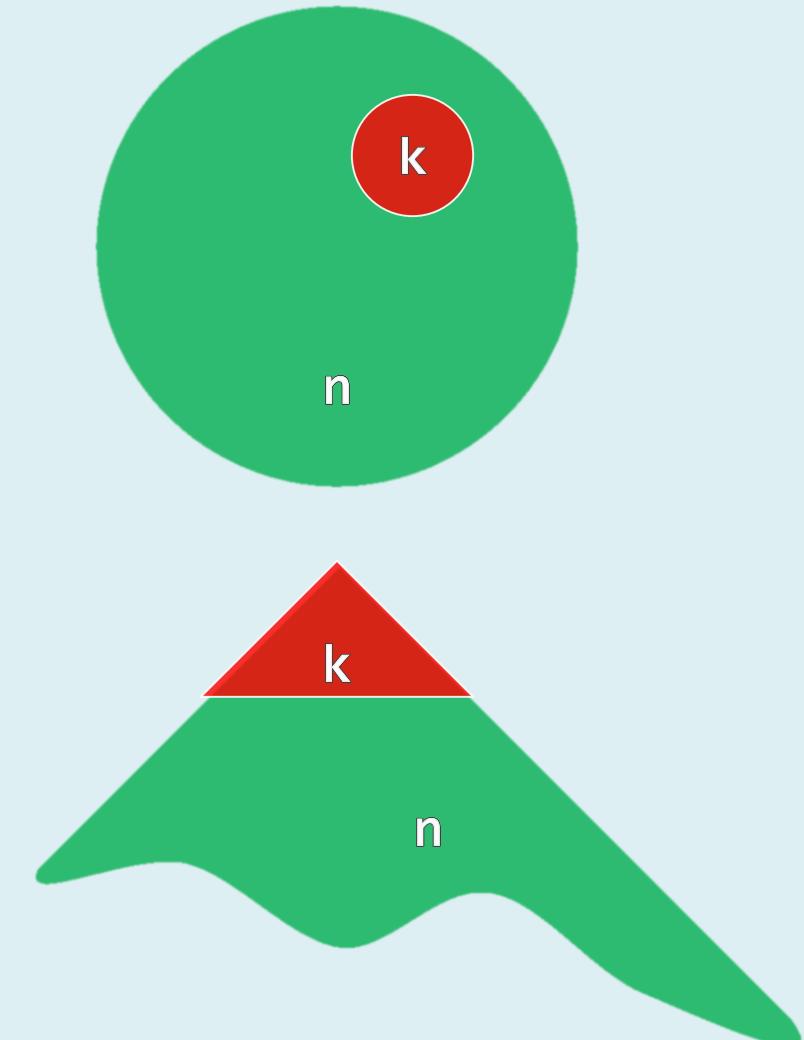
- 效率甚至可以更高——自适应的 $\mathcal{O}(\log k)$

- 任何连续的 m 次查找，仅需 $\mathcal{O}(m \log k + n \log n)$ 时间

❖ 若反复地顺序访问任一子集，分摊成本仅为常数

❖ 不能杜绝单次最坏情况，不适用于对效率敏感的场合

❖ 复杂度的分析稍嫌复杂——好在有初等的证明...



高级搜索树

伸展树：分摊分析

08-A4

转圆石于千仞之山者，势也

所谓物价，其实就是我称之为生命的那部分，必须在交换时支付：要么立即支付，要么以后支付

邓俊辉

deng@tsinghua.edu.cn

S的势能

❖ (任何时刻的) 任何一棵伸展树 S , 都可以假想地被认为具有势能:

$$\Phi(S) = \log \left(\prod_{v \in S} \text{size}(v) \right) = \sum_{v \in S} \log (\text{size}(v)) = \sum_{v \in S} \text{rank}(v) = \sum_{v \in S} \log V$$

❖ 直觉: 越平衡/倾侧的树, 势能越小/大

- 单链: $\Phi(S) = \log n! = \mathcal{O}(n \log n)$

$$\begin{aligned} - \text{满树: } \Phi(S) &= \log \prod_{d=0}^h (2^{h-d+1} - 1)^{2^d} \leq \log \prod_{d=0}^h (2^{h-d+1})^{2^d} \\ &= \log \prod_{d=0}^h 2^{(h-d+1) \cdot 2^d} = \sum_{d=0}^h (h - d + 1) \cdot 2^d = (h + 1) \cdot \sum_{d=0}^h 2^d - \sum_{d=0}^h d \cdot 2^d \\ &= (h + 1) \cdot (2^{h+1} - 1) - [(h - 1) \cdot 2^{h+1} + 2] = 2^{h+2} - h - 3 = \mathcal{O}(n) \end{aligned}$$

T的上界

❖ 考查对伸展树 S 的 $m \gg n$ 次连续访问 (不妨仅考查 $\text{search}(e)$)

❖ 若记: $A^{(k)} = T^{(k)} + \Delta\Phi^{(k)}$, $k = 0, 1, 2, \dots, m$

则有: $A - \mathcal{O}(n \log n) \leq T = A - \Delta\Phi \leq A + \mathcal{O}(n \log n)$

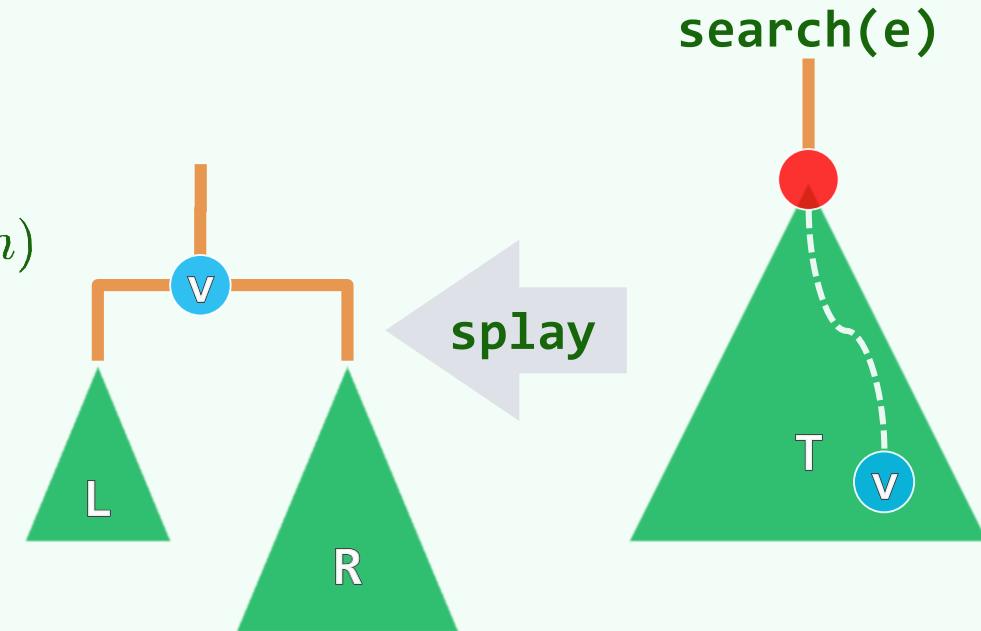
❖ 故若能证明: $A = \mathcal{O}(m \log n)$

则必有: $T = \mathcal{O}(m \log n)$

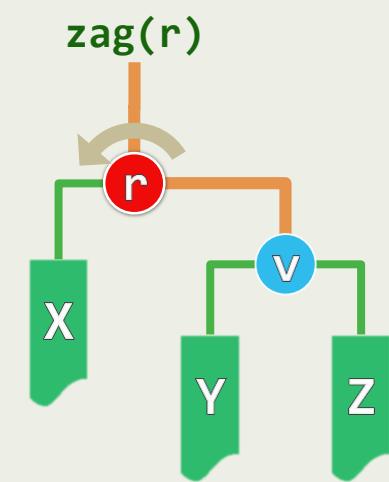
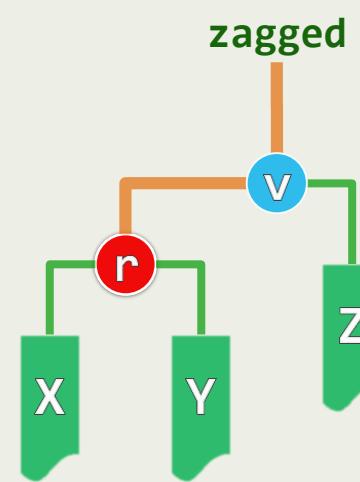
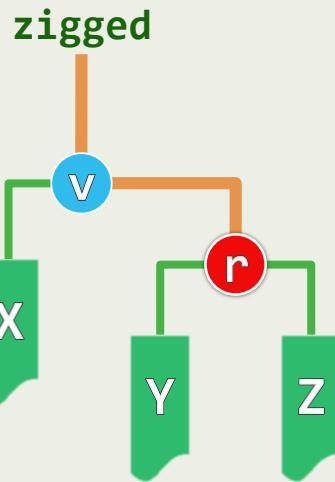
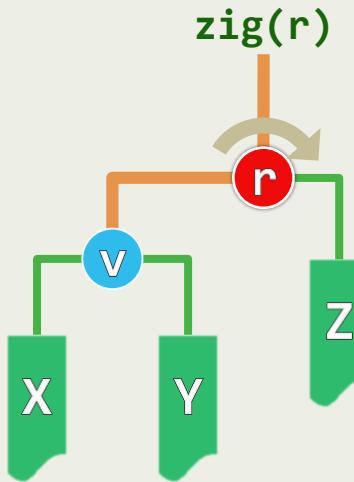
❖ 好消息是, 尽管 $T^{(k)}$ 的变化幅度可能很大, 我们却能证明:

$A^{(k)}$ 都不致超过节点 v 的势能变化量, 即: $\mathcal{O}(\text{rank}^{(k)}(v) - \text{rank}^{(k-1)}(v)) = \mathcal{O}(\log n)$

❖ 事实上, $A^{(k)}$ 不过是 v 的若干次连续伸展操作 (时间成本) 的累积, 这些操作无非三种情况...



Zig / Zag

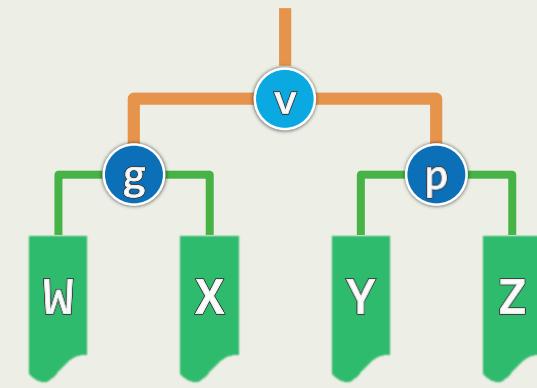
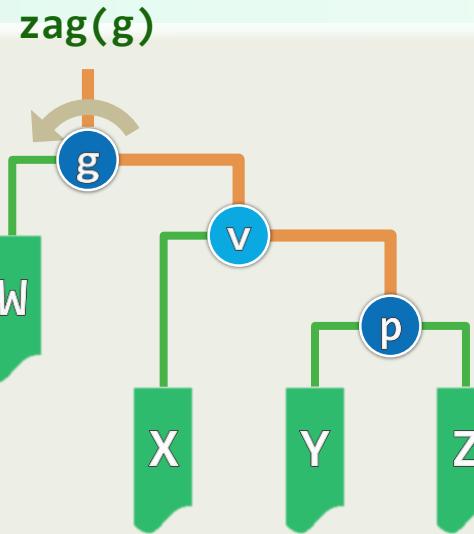
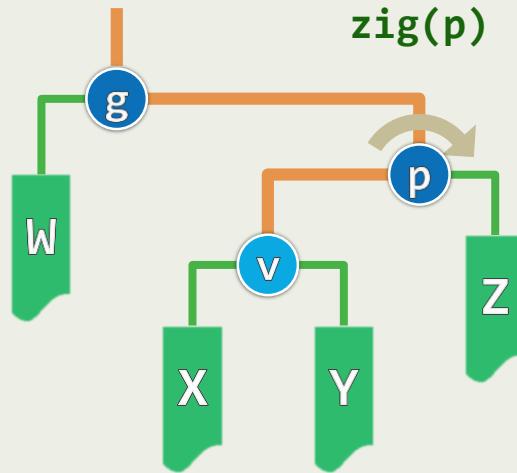


$$A_i^{(k)} = T_i^{(k)} + \Delta\Phi(S_i^{(k)}) = 1 + \Delta rank_i(v) + \Delta rank_i(r)$$

$$= 1 + [rank_i(v) - rank_{i-1}(v)] + \underline{[rank_i(r) - rank_{i-1}(r)]}$$

$$< 1 + [rank_i(v) - rank_{i-1}(v)]$$

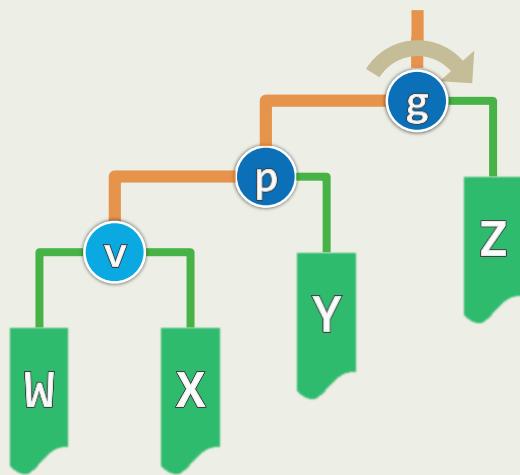
zig-zag / zag-zig



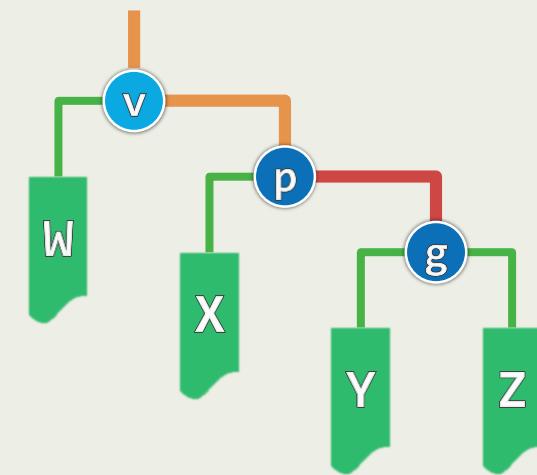
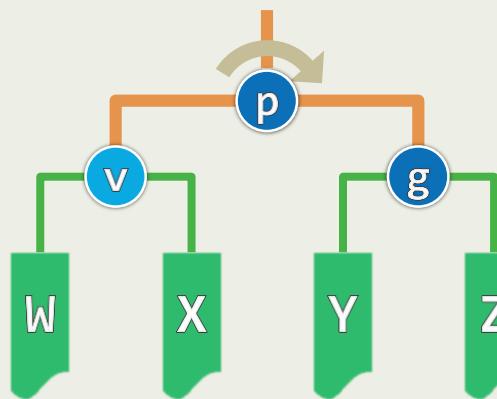
$$\begin{aligned}
 A_i^{(k)} &= T_i^{(k)} + \Delta\Phi(S_i^{(k)}) = 2 + \Delta\text{rank}_i(g) + \Delta\text{rank}_i(p) + \Delta\text{rank}_i(v) \\
 &= 2 + [\text{rank}_i(g) - \underline{\text{rank}_{i-1}(g)}] + [\text{rank}_i(p) - \underline{\text{rank}_{i-1}(p)}] + [\underline{\text{rank}_i(v)} - \underline{\text{rank}_{i-1}(v)}] \\
 &< 2 + \underline{\text{rank}_i(g) + \text{rank}_i(p)} - 2 \cdot \text{rank}_{i-1}(v) \quad (\because \text{rank}_{i-1}(p) > \text{rank}_{i-1}(v)) \\
 &< 2 + \underline{2 \cdot \text{rank}_i(v) - 2} - 2 \cdot \text{rank}_{i-1}(v) \quad (\because \frac{\log G_i + \log P_i}{2} \leq \log \frac{G_i + P_i}{2} < \log \frac{V_i}{2}) \\
 &= 2 \cdot (\text{rank}_i(v) - \text{rank}_{i-1}(v))
 \end{aligned}$$

zig-zig / zag-zag

$\text{zig}(g)$



$\text{zig}(p)$



$$\begin{aligned}
 A_i^{(k)} &= T_i^{(k)} + \Delta\Phi(S_i^{(k)}) = 2 + \Delta\text{rank}_i(g) + \Delta\text{rank}_i(p) + \Delta\text{rank}_i(v) \\
 &= 2 + [\text{rank}_i(g) - \cancel{\text{rank}_{i-1}(g)}] + [\text{rank}_i(p) - \cancel{\text{rank}_{i-1}(p)}] + [\cancel{\text{rank}_i(v)} - \cancel{\text{rank}_{i-1}(v)}] \\
 &< 2 + \text{rank}_i(g) + \cancel{\text{rank}_i(p)} - 2 \cdot \text{rank}_{i-1}(v) \quad (\because \text{rank}_{i-1}(p) > \text{rank}_{i-1}(v)) \\
 &< 2 + \text{rank}_i(g) + \cancel{\text{rank}_i(v)} - 2 \cdot \text{rank}_{i-1}(v) \quad (\because \text{rank}_i(p) < \text{rank}_i(v)) \\
 &< 3 \cdot (\text{rank}_i(v) - \text{rank}_{i-1}(v)) \quad (\because \frac{\log G_i + \log V_{i-1}}{2} \leq \log \frac{G_i + V_{i-1}}{2} < \log \frac{V_i}{2})
 \end{aligned}$$



08 - B7

高级搜索树

B-树：大数据

640K ought to be enough for anybody.

- B. Gates, 1981

白嘉轩一听就不由得火了：“又是个百日忌讳！”仙草却说：“百日又不是百年。你权当百日后才娶我。你就忍一忍，一百天很快就过去了……”

邓俊辉
deng@tsinghua.edu.cn

现实A：存储器容量的增长速度 << 应用问题规模的增长速度

1 Kilobyte = $2^{10} = 10^3$

1 Megabyte = $2^{20} = 10^6$

1 Gigabyte = $2^{30} = 10^9$

1 Terabyte = $2^{40} = 10^{12}$

1 Petabyte = $2^{50} = 10^{15}$

1 Exabyte = $2^{60} = 10^{18}$

1 Zettabyte = $2^{70} = 10^{21}$

1 Yottabyte = $2^{80} = 10^{24}$

1 Nonabyte = $2^{90} = 10^{27}$

1 Doggabyte = $2^{100} = 10^{30}$

❖ 存储器？ - RAM: 不就是...无限可数个...寄存器吗！

- Turing: 不就是...无限长的...纸带吗！

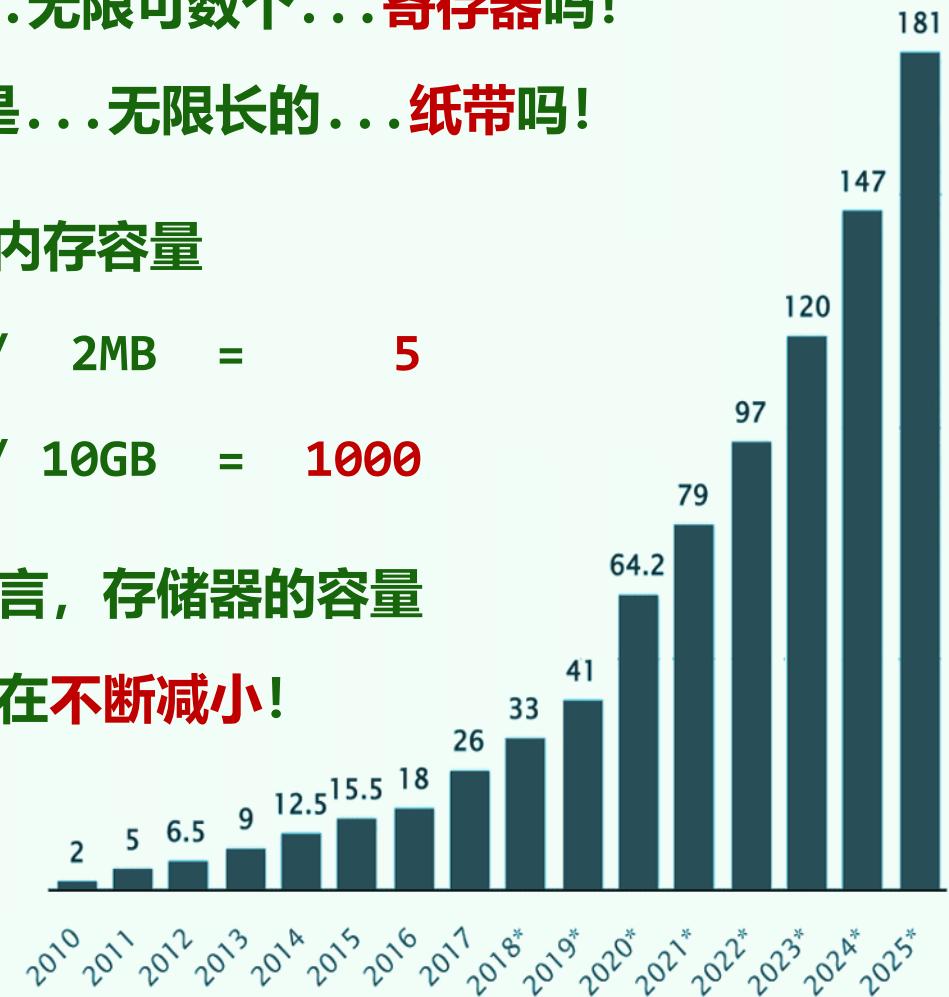
❖ 典型的数据库规模/内存容量

1990: 10MB / 2MB = 5

2020: 10TB / 10GB = 1000

❖ 相对而言，存储器的容量

实际上在不断减小！



现实B：在特定工艺及成本下，存储器都是容量与速度的折中产物

1 Kilobyte = $2^{10} = 10^3$

1 Megabyte = $2^{20} = 10^6$

1 Gigabyte = $2^{30} = 10^9$

1 Terabyte = $2^{40} = 10^{12}$

1 Petabyte = $2^{50} = 10^{15}$

1 Exabyte = $2^{60} = 10^{18}$

1 Zettabyte = $2^{70} = 10^{21}$

1 Yottabyte = $2^{80} = 10^{24}$

1 Nonabyte = $2^{90} = 10^{27}$

1 Doggabyte = $2^{100} = 10^{30}$

❖ 为何...不把存储器做得...更大些？非不为，实不能！

❖ 存储器越大、越快，成本也越高

❖ 存储器容量越大/小，访问速度越慢/快

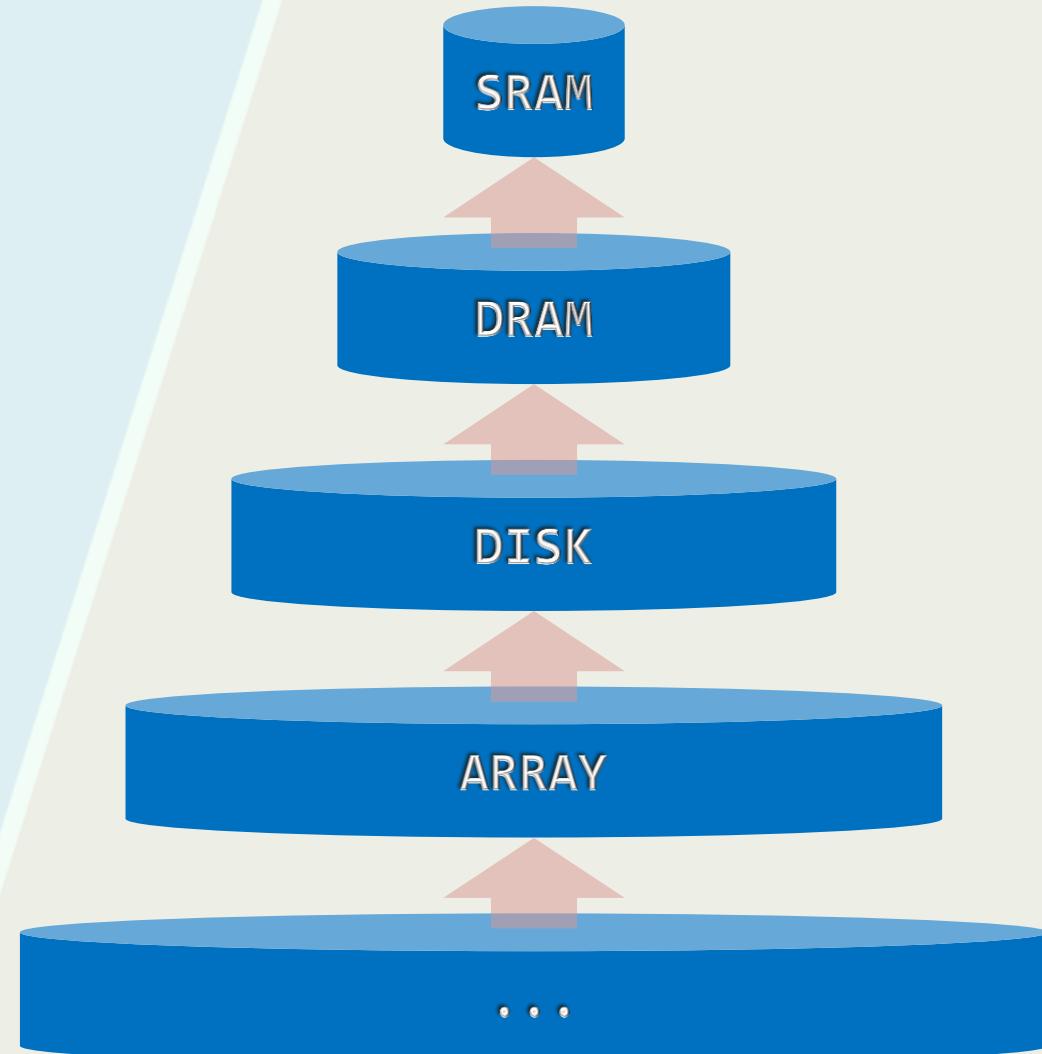


现实C：实用的存储系统，由不同类型的存储器级联而成，以综合其各自的优势

- ❖ 不同类型的存储器，容量、访问速度差异悬殊

	#cycles	sec
CPU Register :	0	ns
SRAM/cache :	4~75	ns
DRAM/main memory :	10^2	ns
DISK :	10^7	ms

- ❖ 若一次内存访问需要一秒，则一次磁盘访问就需一天
为避免一次磁盘访问，我们宁愿访问内存1000次
- ❖ 在分级的存储系统中，各类存储器有其各自的角色



分级存储：利用数据访问的局部性

❖ 机制与策略

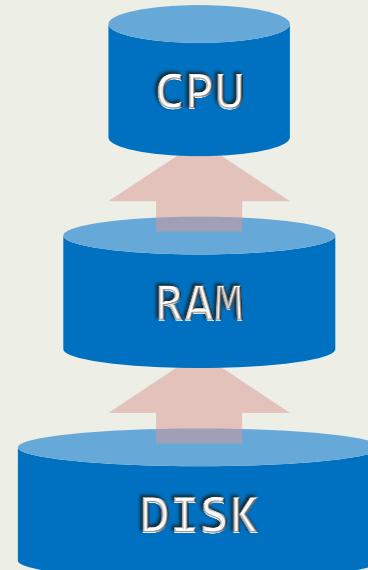
- 常用的数据，复制到更高层、更小的存储器中
- 找不到，才向更低层、更大的存储器索取

❖ 算法的实际运行时间，主要取决于

相邻存储级别之间

数据传输（I/O）的

速度与次数



T S I N G H U A

A G H I N S T U

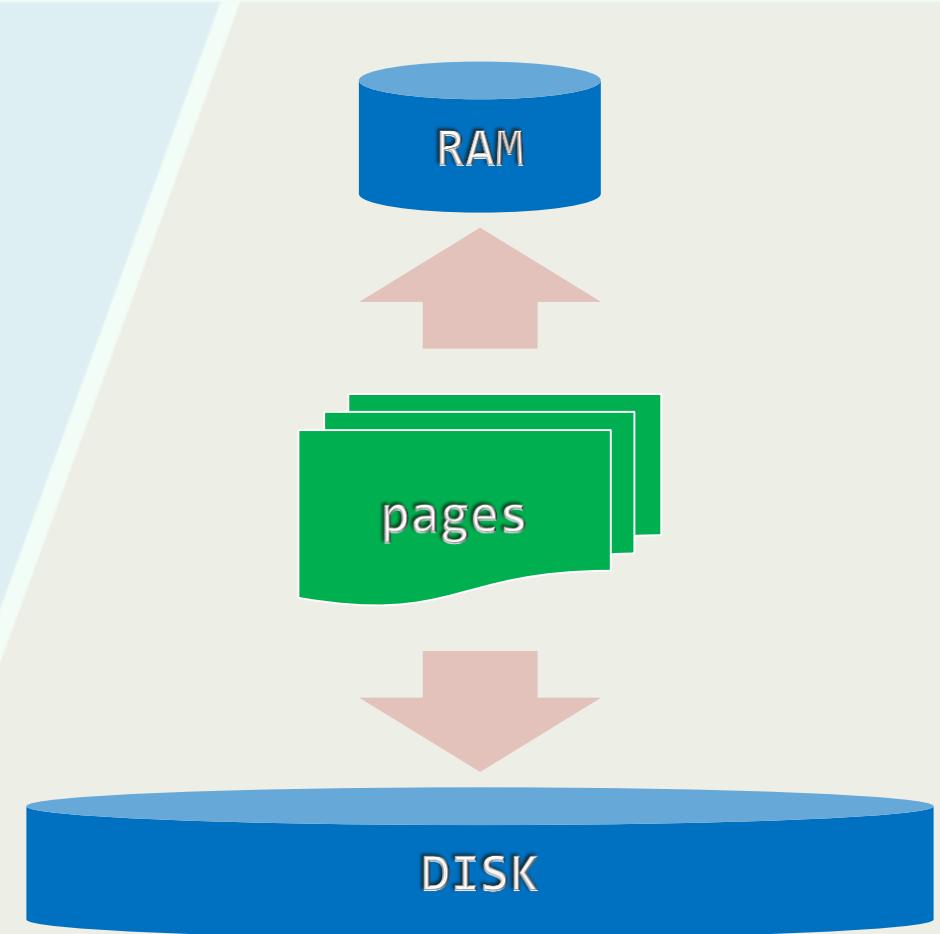


分级存储：批量访问：在外存读写1B，与读写1KB几乎一样快

❖ 以页（page）或块（block）为单位，借助缓冲区，可大大缩短单位字节的平均访问时间

❖ #include <stdio.h>

```
#define BUFSIZ 512 //缓冲区默认容量  
  
int setvbuf( //定制缓冲区  
    FILE* fp, //流  
    char* buf, //缓冲区  
    int _Mode, //_IOFBF | _IOLBF | _IONBF  
    size_t size); //缓冲区容量  
  
int fflush( FILE* fp ); //强制清空缓冲区
```



高级搜索树

B-树：缓存

To my mind the most interesting thing in art is the personality of artists;
and if that is singular, I am willing to excuse a thousand faults.

He has given signs of himself which are visible to those who seek
him, and not to those who do not seek him.

邓俊辉

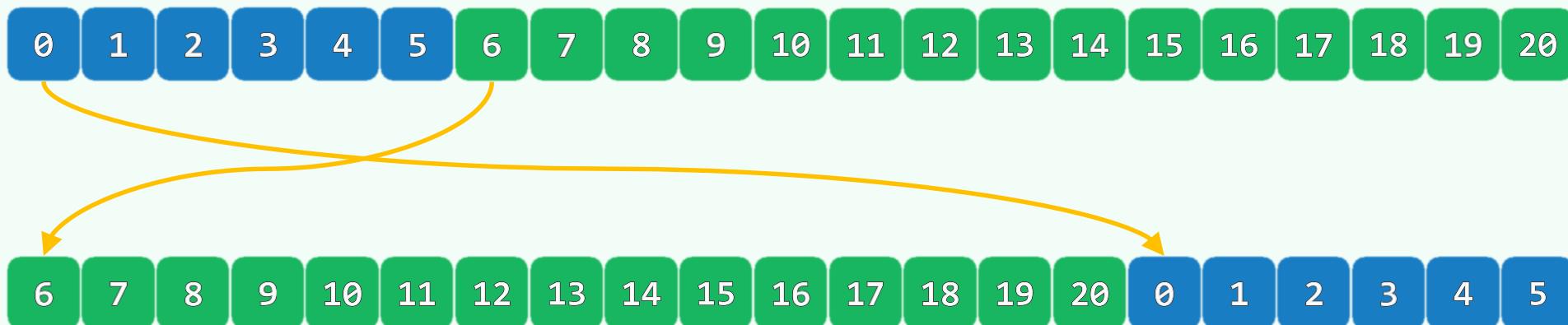
deng@tsinghua.edu.cn

就地循环位移

❖ 仅用 $O(1)$ 辅助空间，将数组A[0, n)中的元素向左循环移动k个单元

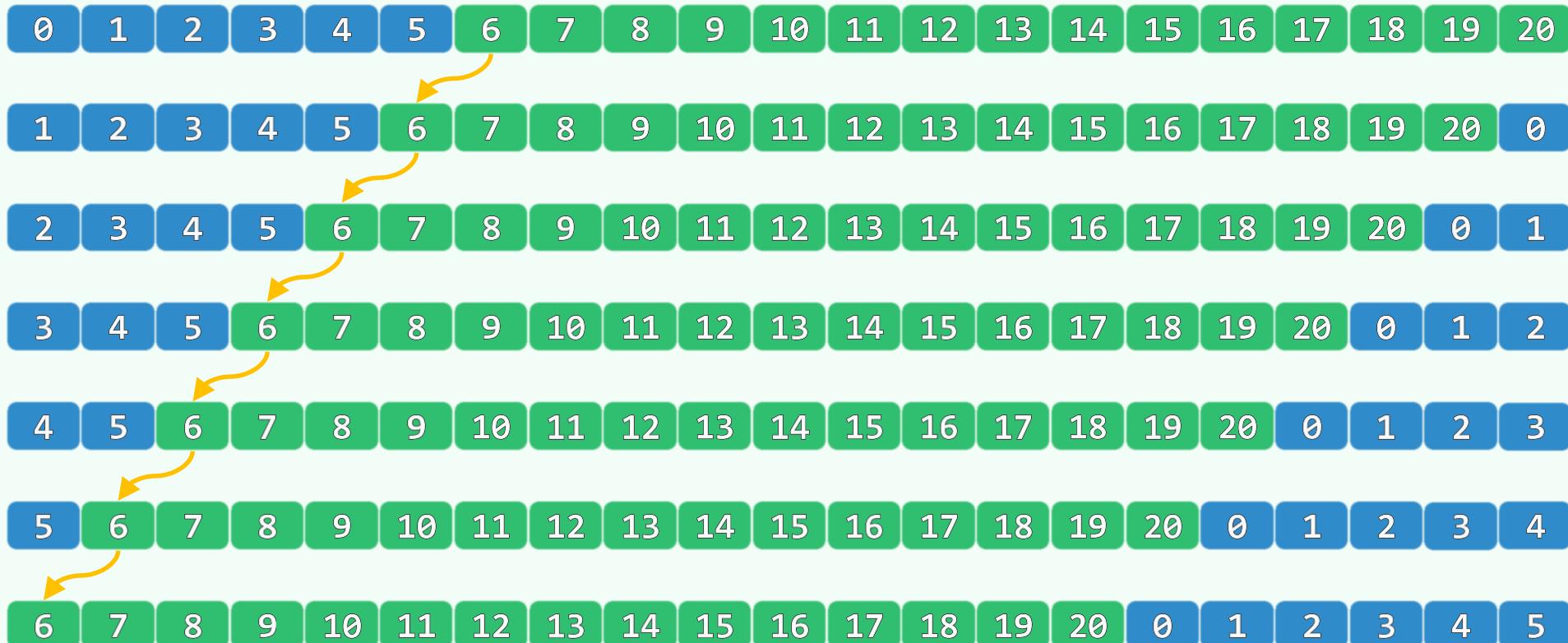
```
void shift( int * A, int n, int k );
```

❖ 比如: shift(A, 21, 6);



蛮力版

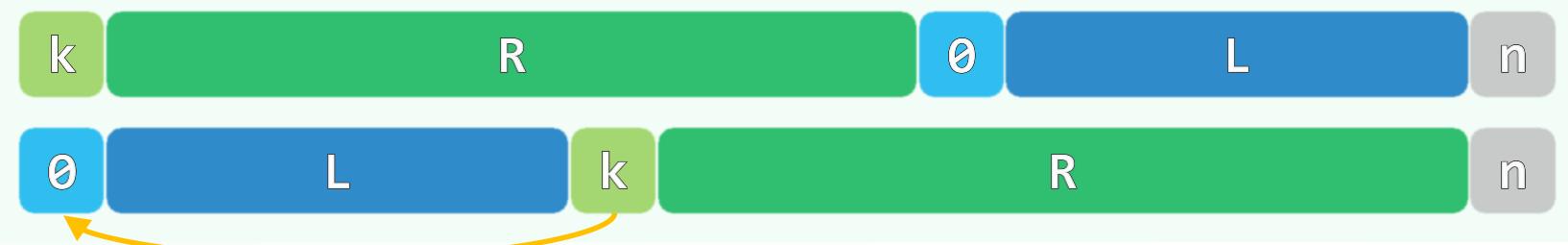
```
void shift0( int * A, int n, int k ) //反复以1为间距循环左移  
{ while ( k-- ) shift( A, n, 0, 1 ); } //共迭代k次, O(n*k)
```



迭代版: Stride- k Reference Pattern (1/2)

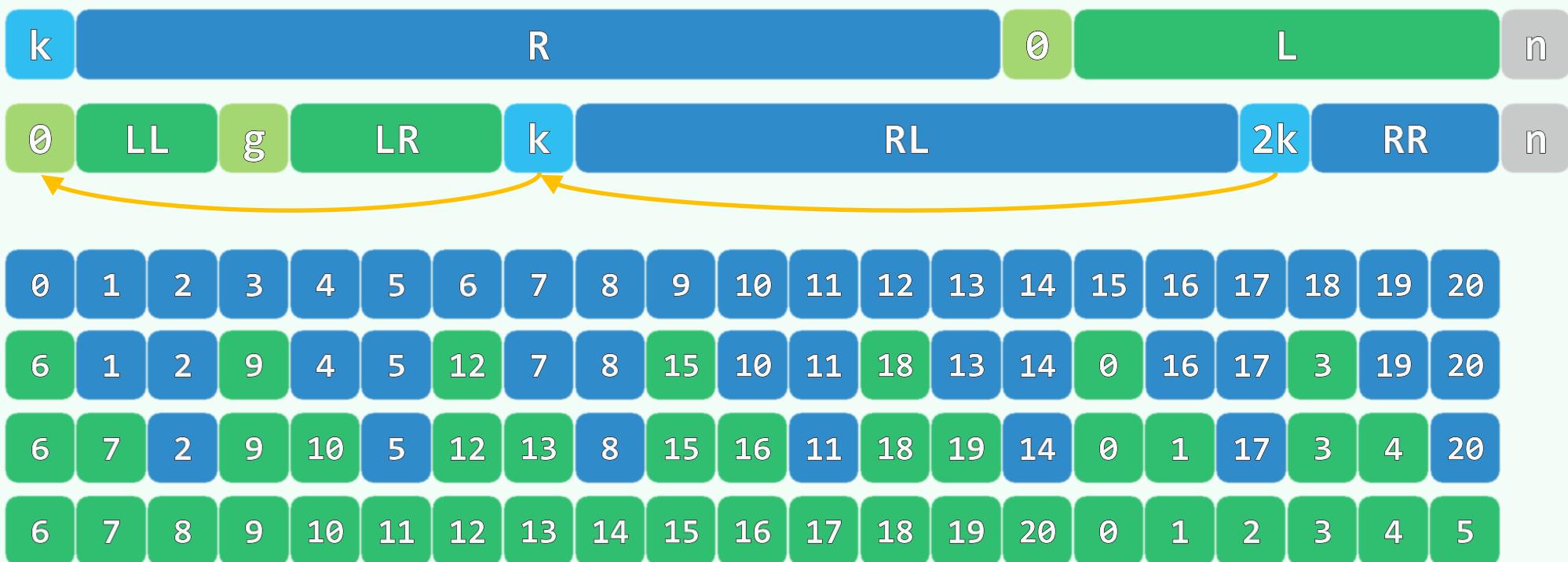
```
int shift( int * A, int n, int s, int k ) { // O( n / GCD(n, k) )  
    int b = A[s]; int i = s, j = (s + k) % n; int mov = 0; //mov记录移动次数  
    while ( s != j ) //从A[s]出发, 以k为间隔, 依次左移k位  
    { A[i] = A[j]; i = j; j = (j + k) % n; mov++; }  
    A[i] = b; return mov + 1; //最后, 起始元素转入对应位置  
} // [0, n)由关于k的g = GCD(n, k)个同余类组成, shift(s, k)能够且只能够使其中之一就位
```

其它的同余类呢...



迭代版: Stride-k Reference Pattern (2/2)

```
void shift1(int* A, int n, int k) { //经多轮迭代, 实现数组循环左移k位, 累计 $\theta(n+g)$   
    for (int s = 0, mov = 0; mov < n; s++) // $\theta(g) = \theta(\text{GCD}(n, k))$   
        mov += shift(A, n, s, k);  
}
```



倒置版: Stride-1 Reference Pattern

```
void shift2( int * A, int n, int k ) {
```

```
    reverse( A, k ); // $\theta(3k/2)$ 
```

```
    reverse( A + k, n - k ); // $\theta(3(n-k)/2)$ 
```

```
    reverse( A, n ); // $\theta(3n/2)$ 
```

```
} // $\theta(3n)$ 
```



高级搜索树

B-树：结构

e8-B3

邓俊辉

deng@tsinghua.edu.cn

妻子好合，如鼓瑟琴；兄弟既翕，和乐且湛

等价变换

❖ 平衡的多路搜索树

R. Bayer & E. McCreight

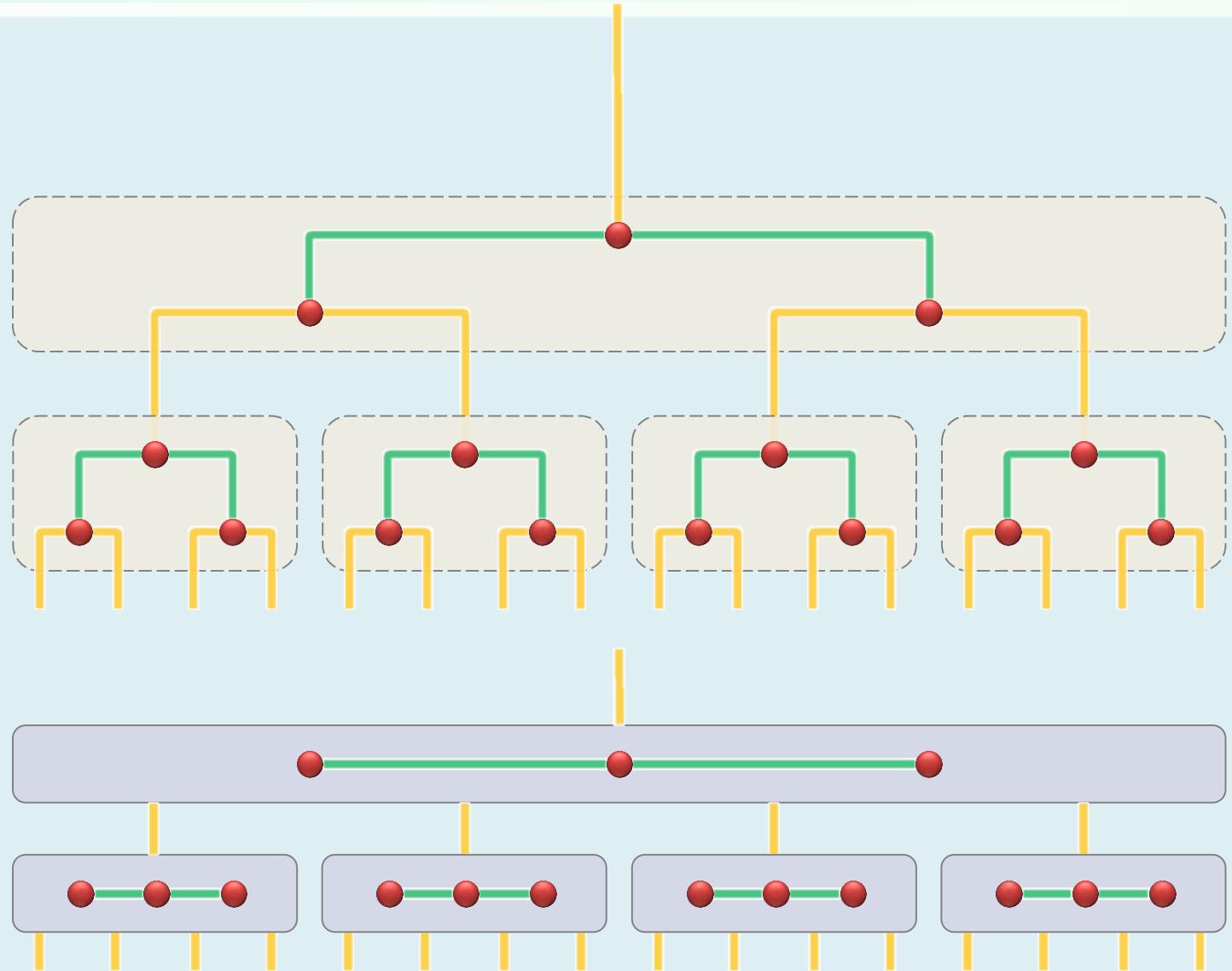
1970

❖ 每d代合并为超级节点

- $m = 2^d$ 路
- $m-1$ 个关键码

❖ 逻辑上与BBST完全等价

既如此，B-树之意义何在？



I/O优化：多级存储系统中使用B-树，可针对外部查找，大大减少I/O次数

❖ 难道，AVL还不够？比如，若有 $n = 1G$ 个记录...

- 每次查找需要 $\log_2 10^9 \approx 30$ 次I/O操作
- 每次只读出单个关键码，得不偿失

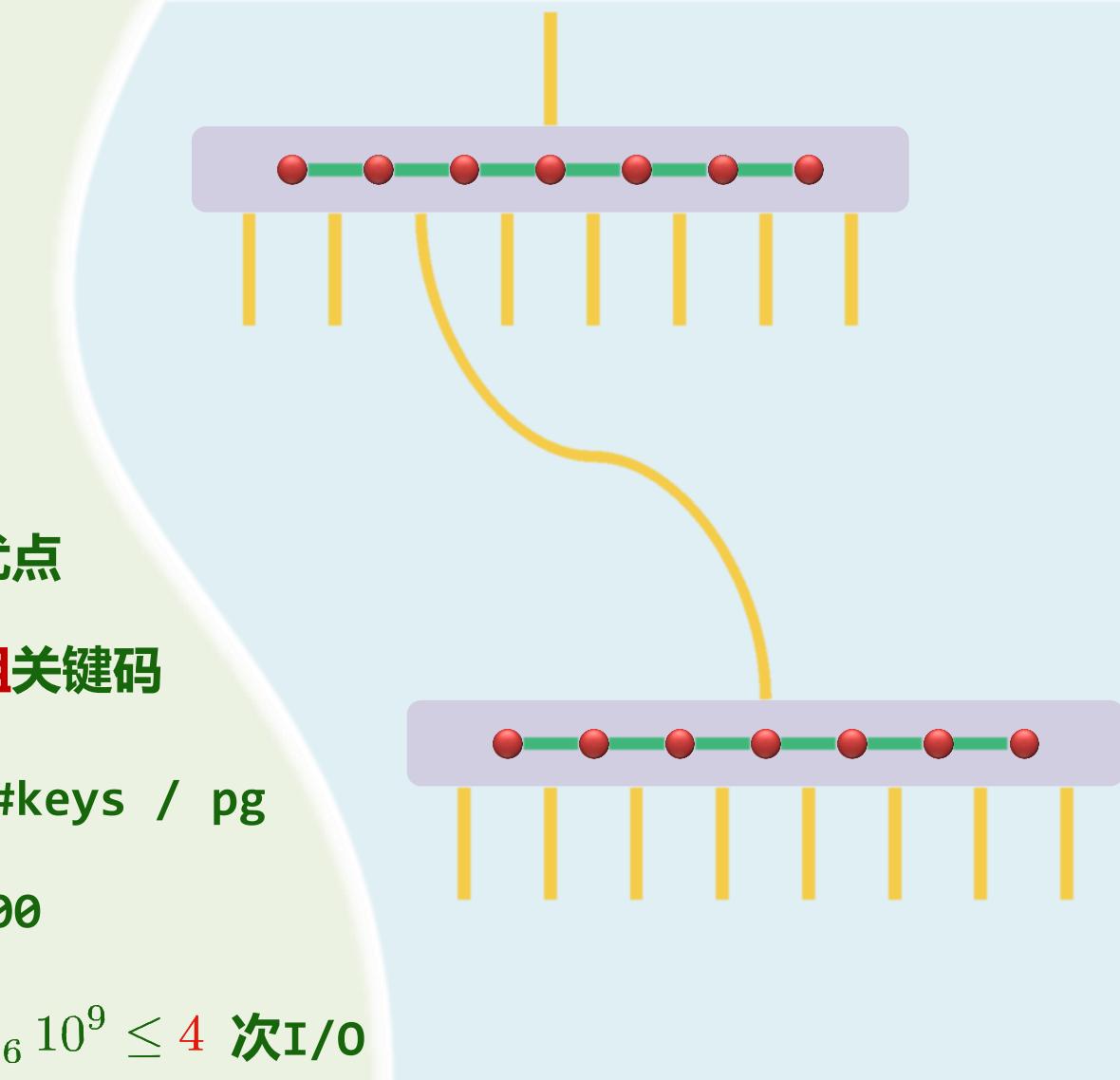
❖ B-树又能如何？

- 充分利用外存的**批量访问**，将此特点转化为优点
- 每下降一层，都以**超级节点**为单位，读入**一组关键码**

❖ 具体多大一组？视磁盘的**数据块大小**而定， $m = \#keys / pg$

- 比如，目前多数数据库系统采用 $m = 200\sim300$

❖ 回到上例，若取 $m = 256$ ，则每次查找只需 $\log_{256} 10^9 \leq 4$ 次I/O



外部节点 + 叶子

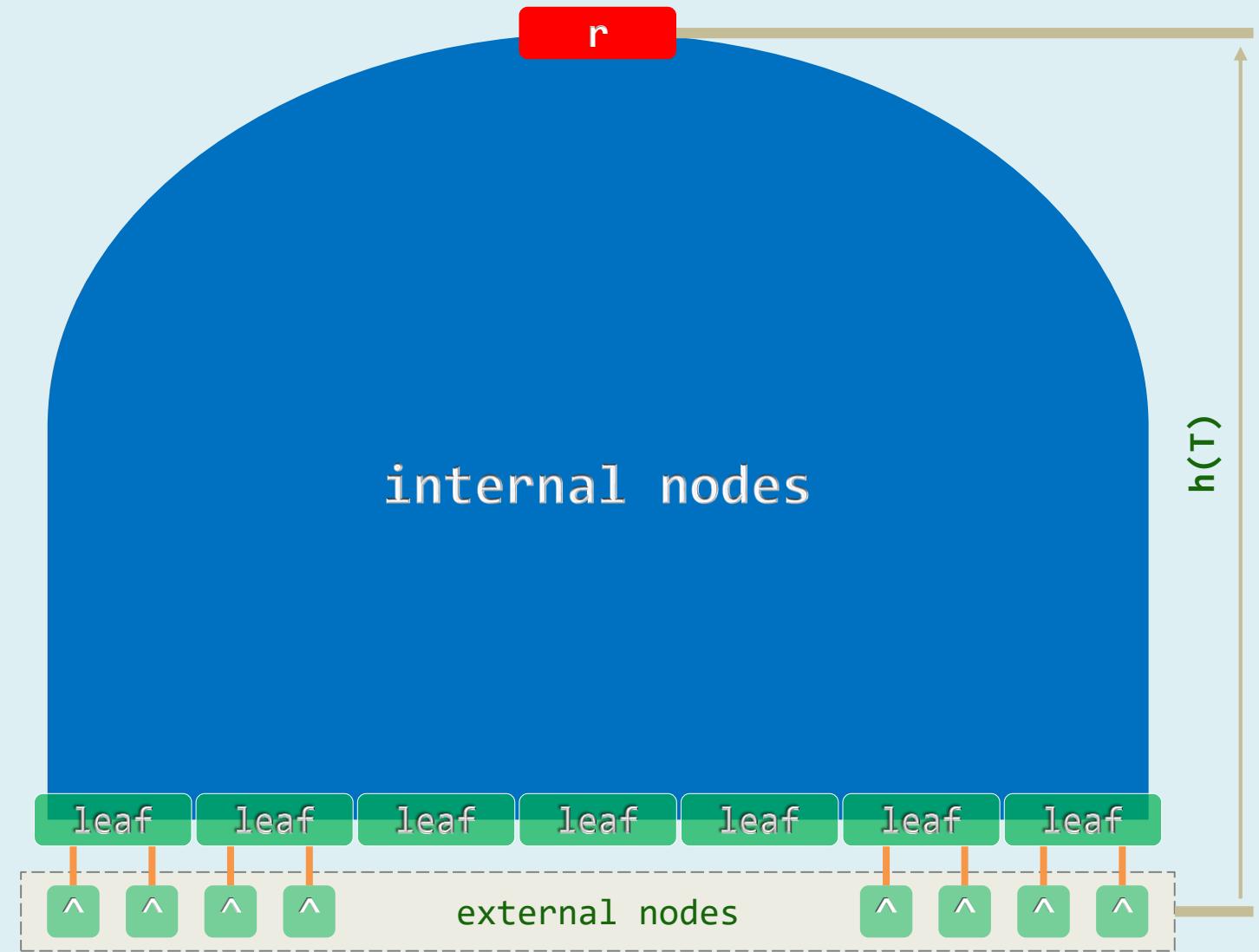
❖ 所谓m阶B-树，即

m路完全平衡搜索树 ($m \geq 3$)

❖ 外部节点的深度统一相等

约定以此深度作为树高h

❖ 叶节点的深度统一相等 ($h-1$)



内部节点

❖ 各含 $n \leq m-1$ 个关键码:

$$K_1 < K_2 < K_3 < \dots < K_n$$

❖ 各有 $n+1 \leq m$ 个分支:

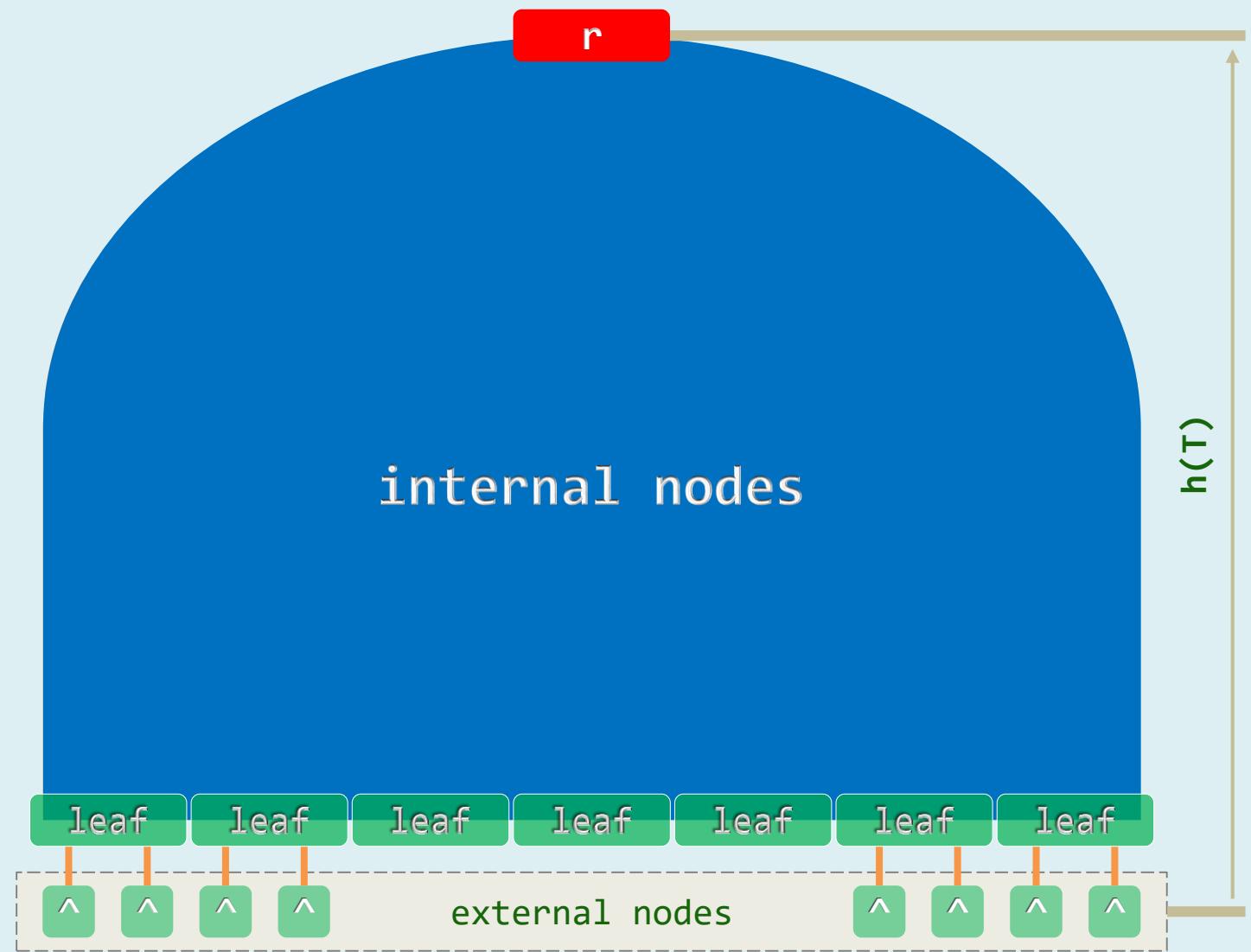
$$A_0, A_1, A_2, A_3, \dots, A_n$$

❖ 反过来, 分支数也不能太少

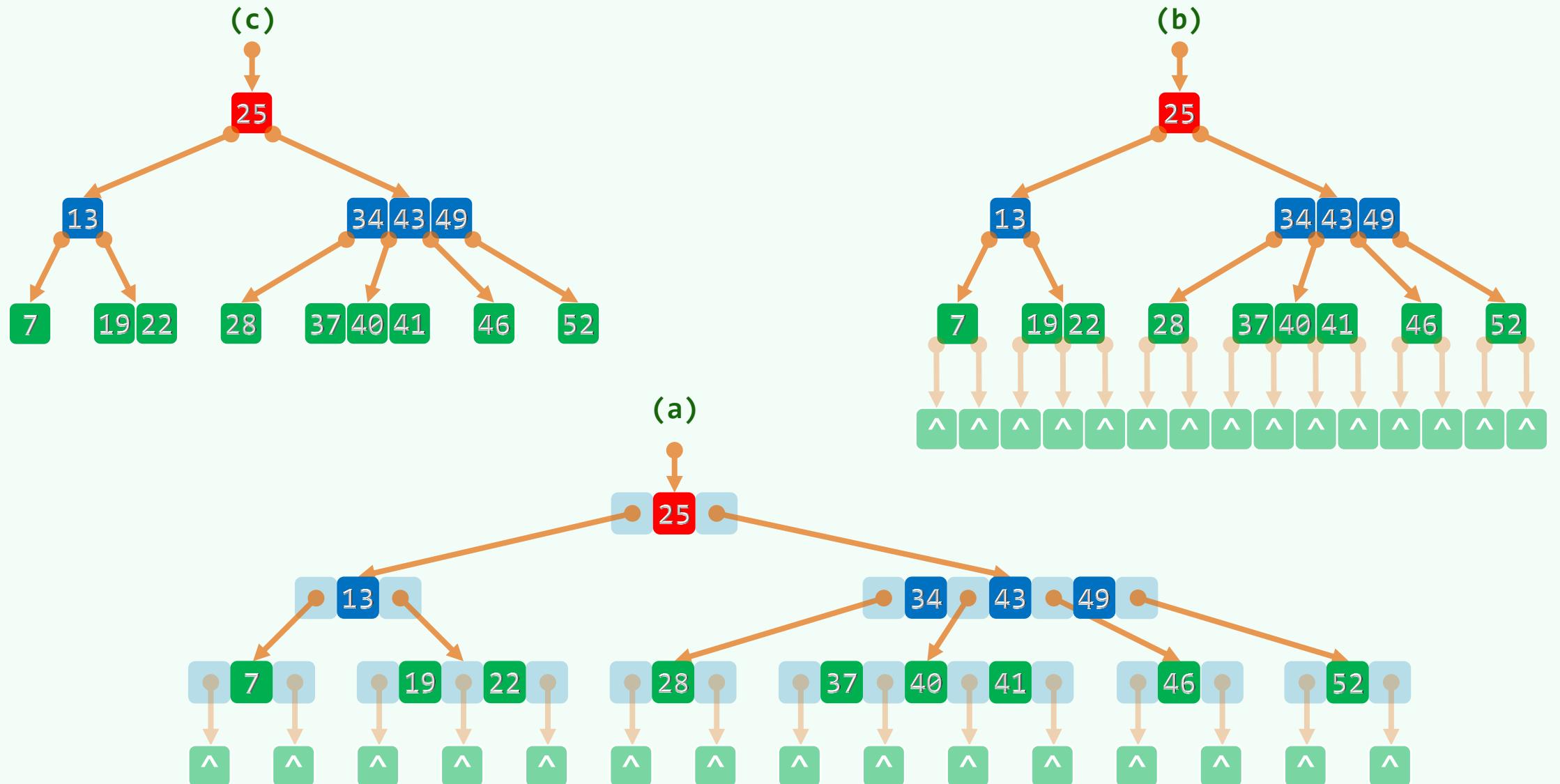
- 树根: $2 \leq n+1$
- 其余: $\lceil m/2 \rceil \leq n+1$

❖ 故亦称作 $(\lceil m/2 \rceil, m)$ -树

- $(3, 5)$ -树
- $(9, 18)$ -树
- ...



紧凑表示



实例

❖ $m = 3$: 2-3-树, $(2,3)$ -树

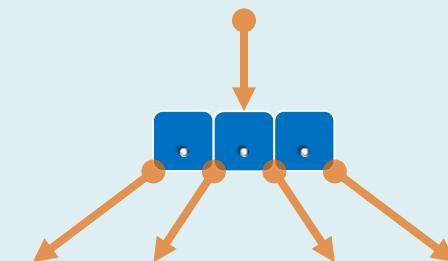
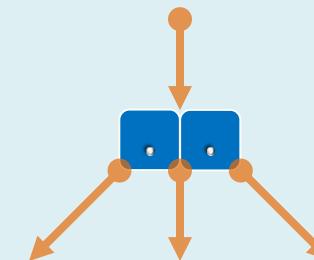
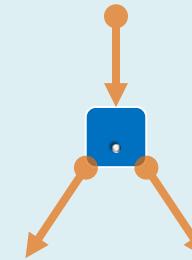
最简单的B-树 //J. Hopcroft, 1970

- 各(内部)节点的分支数, 可能是2或3
- 各节点所含key的数目, 可能是1或2

❖ $m = 4$: 2-3-4-树, $(2,4)$ -树

- 各节点的分支数, 可能是2、3或4
- 各节点所含key的数目, 可能是1、2或3

❖ 留意把玩4阶B-树, 稍后对于理解红黑树大有裨益



BTNode

```
template <typename T> struct BTNode { //B-树节点
```

```
    BTNodePosi<T> parent; //父
```

```
    Vector<T> key; //关键码 (总比孩子少一个)
```

```
    Vector< BTNodePosi<T> > child; //孩子
```

```
BTNode() { parent = NULL; child.insert( NULL ); }
```

```
BTNode( T e, BTNodePosi<T> lc = NULL, BTNodePosi<T> rc = NULL ) {
```

```
    parent = NULL; //作为根节点
```

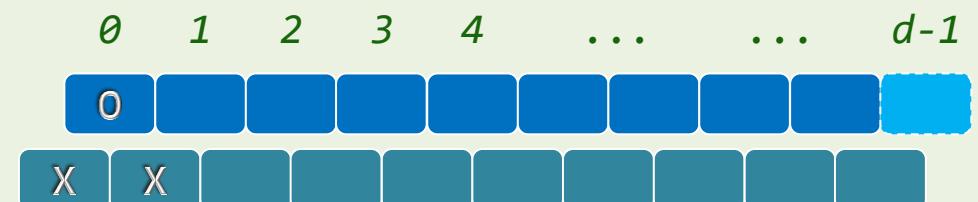
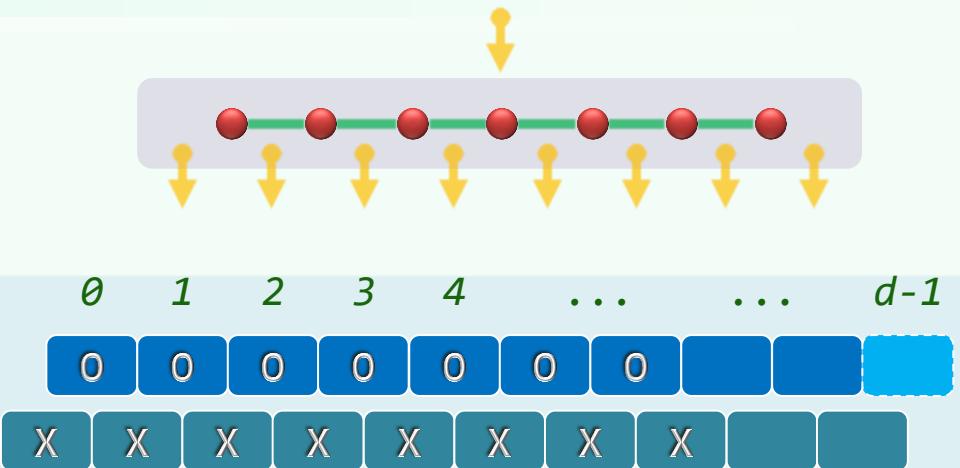
```
    key.insert( e ); //仅一个关键码, 以及
```

```
    child.insert( lc ); if ( lc ) lc->parent = this; //左孩子
```

```
    child.insert( rc ); if ( rc ) rc->parent = this; //右孩子
```

```
}
```

```
};
```



BTree

```
template <typename T> using BTNodePosi = BTNode<T>*; //B-树节点位置

template <typename T> class BTree { //B-树

protected:

    Rank _size, _m; //关键码总数、阶次

    BTNodePosi<T> _root, _hot; //根、search()最后访问的非空节点

    void solveOverflow( BTNodePosi<T> ); //因插入而上溢后的分裂处理

    void solveUnderflow( BTNodePosi<T> ); //因删除而下溢后的合并处理

public:

    BTNodePosi<T> search( const T & e ); //查找

    bool insert( const T & e ); //插入

    bool remove( const T & e ); //删除

};
```

高级搜索树

B-树：查找

高至天低至深海

每寸搜索着这天下

寻觅着那个"它"

...按照模型的运算量，用现有的最高计算能力模拟百分之一秒的聚变过程，就需大约二十年时间。而研究过程中的模拟需要反复进行，这使得模型的实际应用成为不可能。

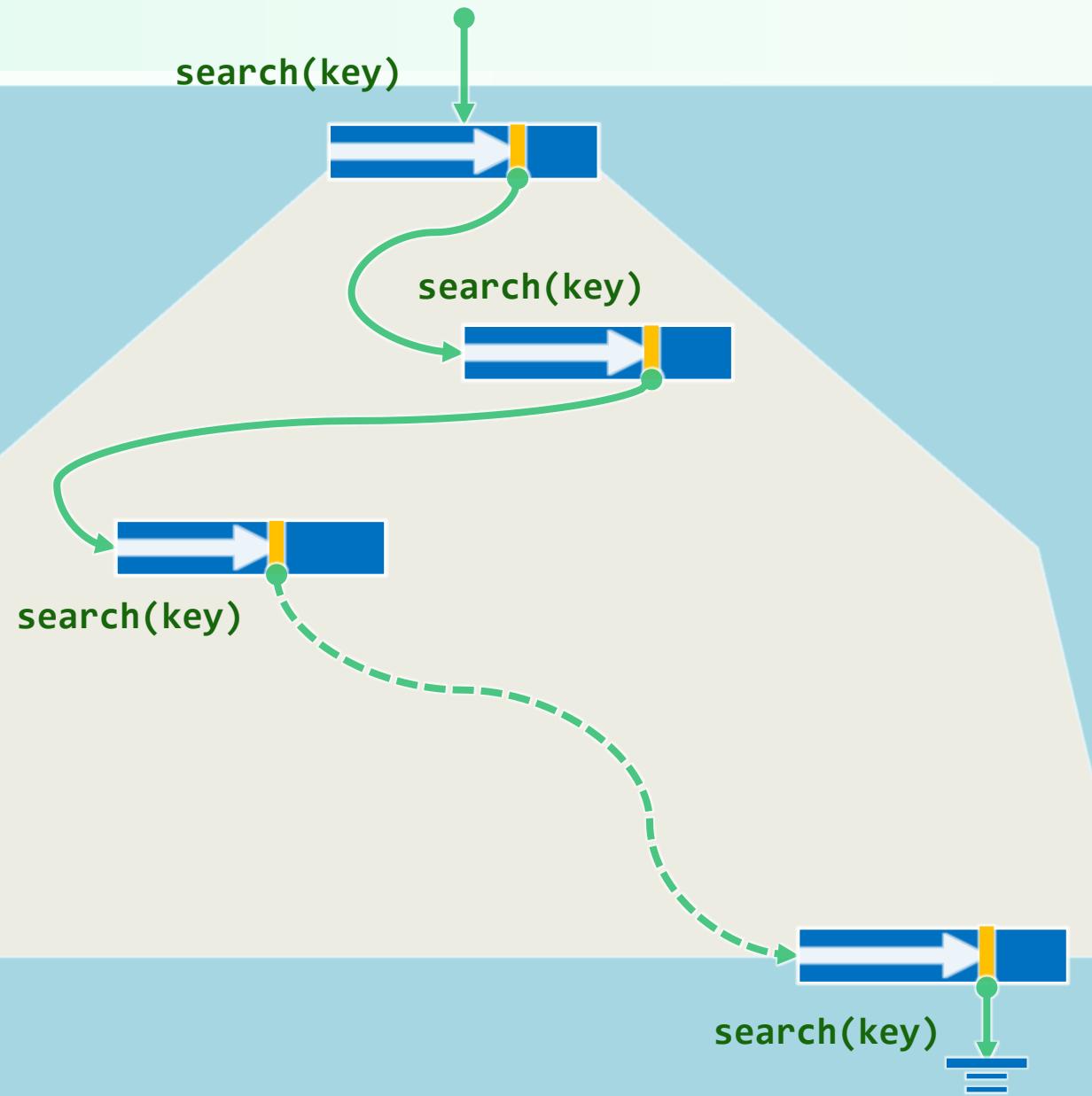
08-B4

邓俊辉

deng@tsinghua.edu.cn

算法

从 (常驻RAM的) 根节点开始
只要当前节点不是外部节点
在当前节点中顺序查找 //RAM内部
若找到目标关键码，则
 返回查找成功
否则 //止于某一向下的引用
 沿引用找到孩子节点
 将其读入内存 //I/O耗时
返回查找失败



实例

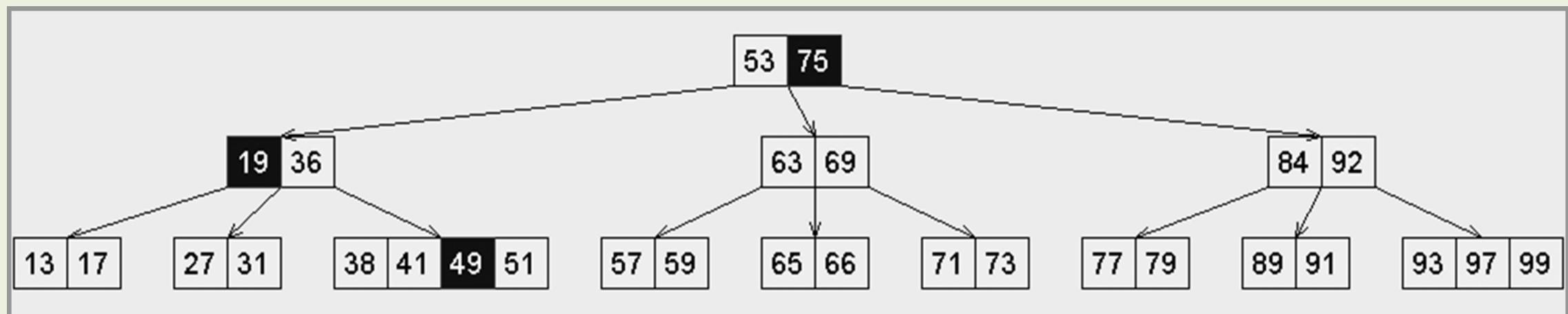
❖ (3,5)-树:

53 97 36 89 41 75 19 84 77 79 51 57 99 91

92 93 17 73 13 66 59 49 63 65 71 69 27 31 38

成功查找: 75, 19, 49

失败查找: 5, 45

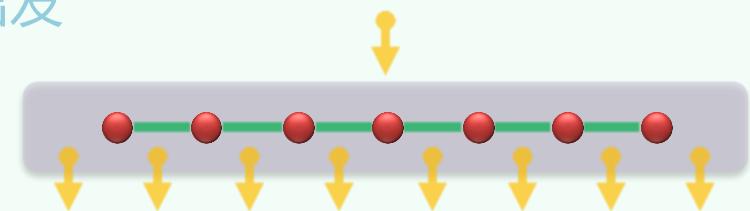


实现

```
❖ template <typename T> BTNodePosi<T> BTree<T>::search( const T & e ) {
```

```
    BTNodePosi<T> v = _root; _hot = NULL; //从根节点出发
```

```
    while ( v ) { //逐层深入地
```



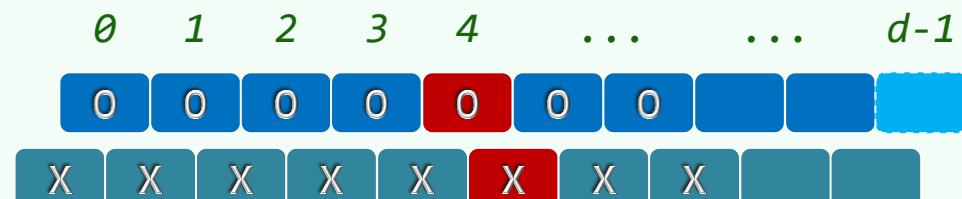
```
        Rank r = v->key.search( e ); //在当前节点对应的向量中顺序查找
```

```
        if ( 0 <= r && e == v->key[r] ) return v; //若成功，则返回；否则...
```

```
        _hot = v; v = v->child[ r + 1 ]; //沿引用转至对应的下层子树，并载入其根(I/O)
```

```
    } //若因!v而退出，则意味着抵达外部节点
```

```
return NULL; //失败
```



性能

❖ 约定：根节点常驻RAM

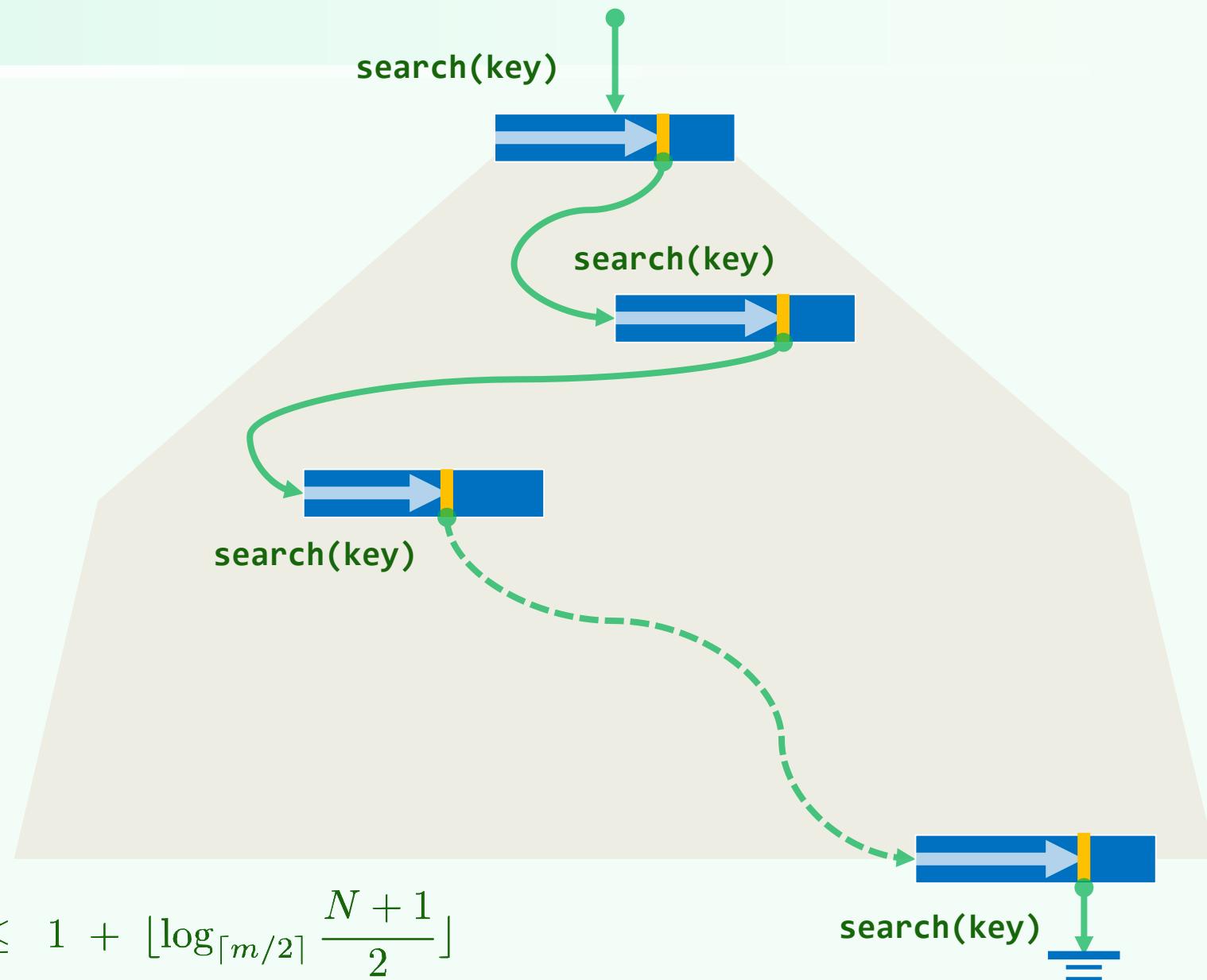
❖ 忽略内存中的查找

运行时间主要取决于I/O次数

❖ 在每一深度至多一次I/O

❖ 故运行时间 = $\mathcal{O}(\log n)$

❖ 可以证明： $\log_m (N + 1) \leq h \leq 1 + \lfloor \log_{\lceil m/2 \rceil} \frac{N + 1}{2} \rfloor$



最大树高

含N个关键码的m阶B-树，可能有多“高”？

为此，内部节点应尽可能地“瘦”

$$n_k \geq 2 \times \lceil m/2 \rceil^{k-1}, \quad \forall k > 0$$

考查外部节点所在的那层：

$$N + 1 = n_h \geq 2 \times \lceil m/2 \rceil^{h-1}$$

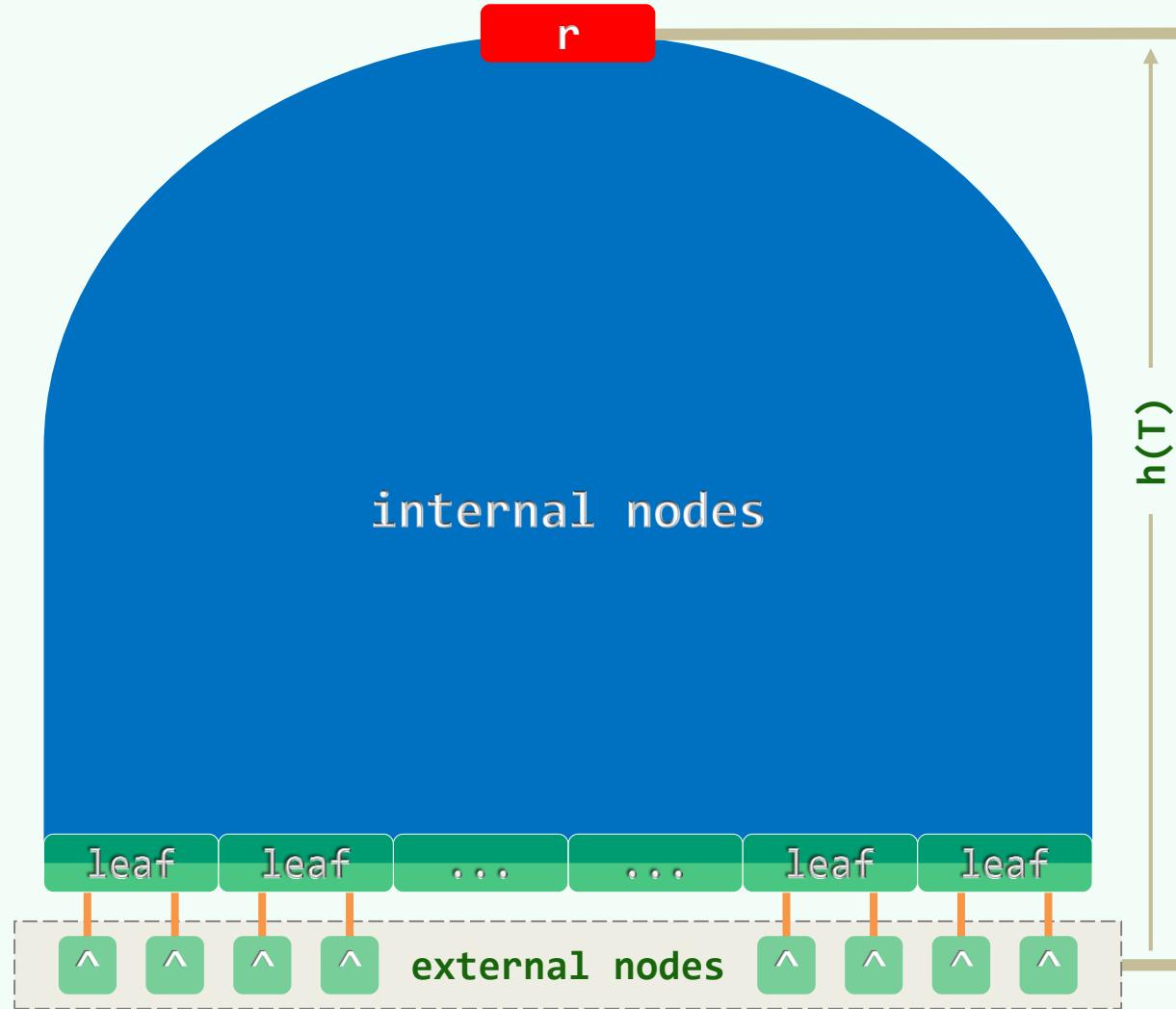
$$h \leq 1 + \lfloor \log_{\lceil \frac{m}{2} \rceil} \frac{N+1}{2} \rfloor = \mathcal{O}(\log_m N)$$

相对于BBST：

$$\log_{\lceil \frac{m}{2} \rceil} (N/2) / \log_2 N = 1/(\log_2 m - 1)$$

若取m = 256，树高约降低至1/7

...用4年上完大学，还是28年？



最小树高

含N个关键码的m阶B-树，可能有多“矮”？

为此，内部节点应尽可能“胖”

$$n_k \leq m^k, \quad \forall k \geq 0$$

依然，考查外部节点所在的那层

$$N + 1 = n_h \leq m^h$$

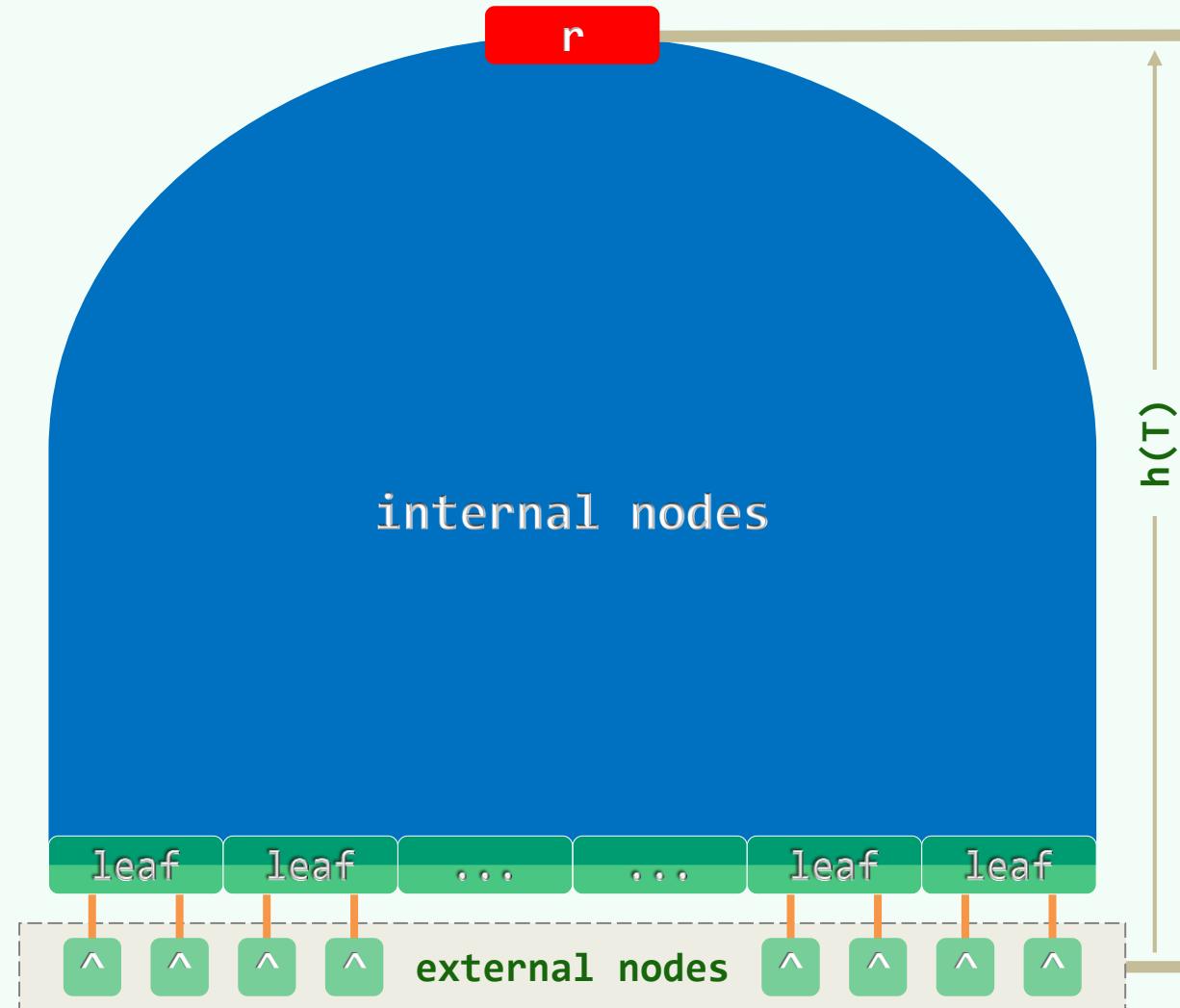
$$h \geq \lceil \log_m (N + 1) \rceil = \Omega(\log_m N)$$

相对于BBST：

$$(\log_m N - 1) / \log_2 N$$

$$= \log_m 2 - \log_N 2 \approx 1 / \log_2 m$$

若取m = 256，树高约降低至1/8



高级搜索树

B-树：插入

08-B5

邓俊辉

deng@tsinghua.edu.cn

说再见，在这梦幻国度，最后的一瞥

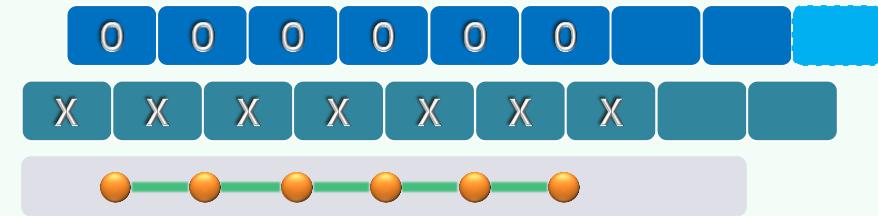
清醒让我，分裂再分裂

算法

```
template <typename T> bool BTTree<T>::insert( const T & e ) {
```

```
    BTNodePosi<T> v = search( e );
```

```
    if ( v ) return false; //确认e不存在
```



```
    Rank r = _hot->key.search( e ); //在节点_hot中确定插入位置
```

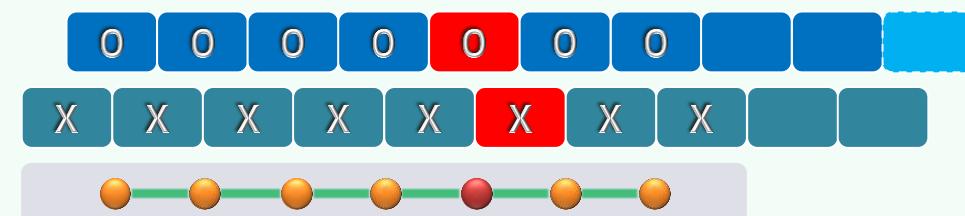
```
    _hot->key.insert( r+1, e ); //将新关键码插至对应的位置
```

```
    _hot->child.insert( r+2, NULL ); _size++; //创建一个空子树指针
```

```
    solveOverflow( _hot ); //若上溢，则分裂
```

```
    return true; //插入成功
```

```
}
```



分裂

❖ 设上溢节点中的关键码依次为：

$$\{ k_0, k_1, \dots, k_{m-1} \}$$

❖ 取中位数 $s = \lfloor m/2 \rfloor$, 以关键码 k_s 为界划分为

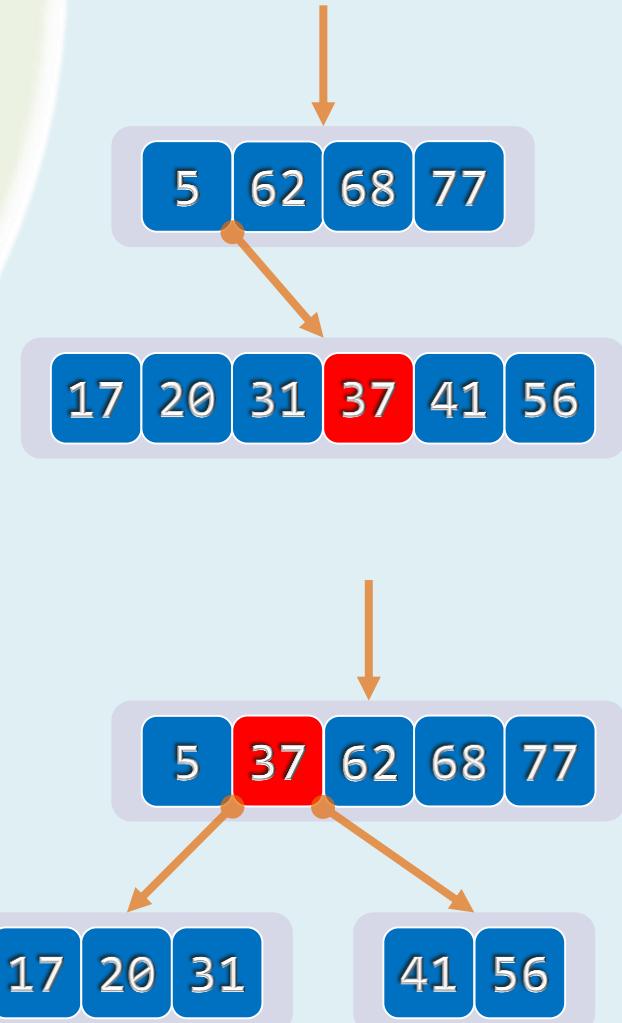
$$\{ k_0, \dots, k_{s-1} \} \quad \{ k_s \} \quad \{ k_{s+1}, \dots, k_{m-1} \}$$

❖ 关键码 k_s 上升一层，并分裂 (split)

以所得的两个节点作为左、右孩子

❖ 不难验证，如此分裂后

左、右孩子所含关键码数目，依然符合m阶B-树的条件



再分裂

❖ 若上溢节点的父亲本已饱和，则在接纳被提升的关键码之后，也将上溢

此时，大可套用前法，继续分裂

❖ 上溢可能持续发生，并逐层向上传播

纵然最坏情况，亦不过到根 //若果真抵达树根…

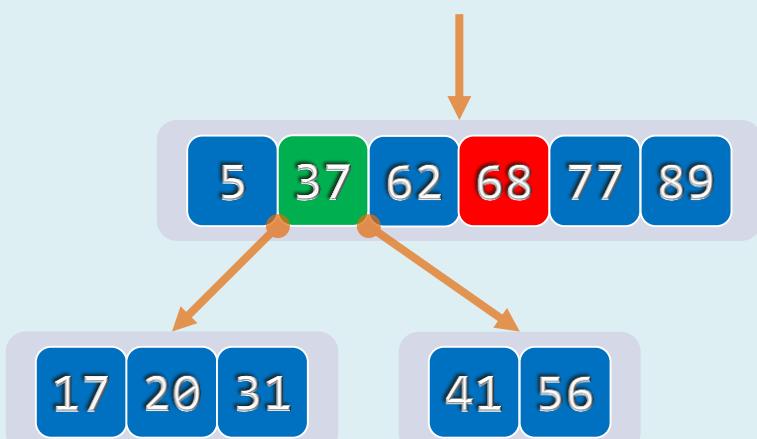
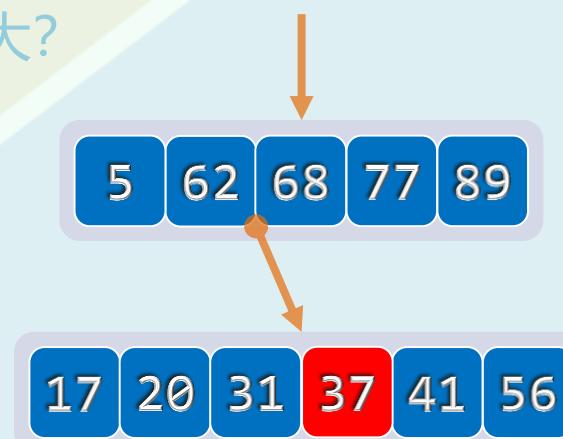
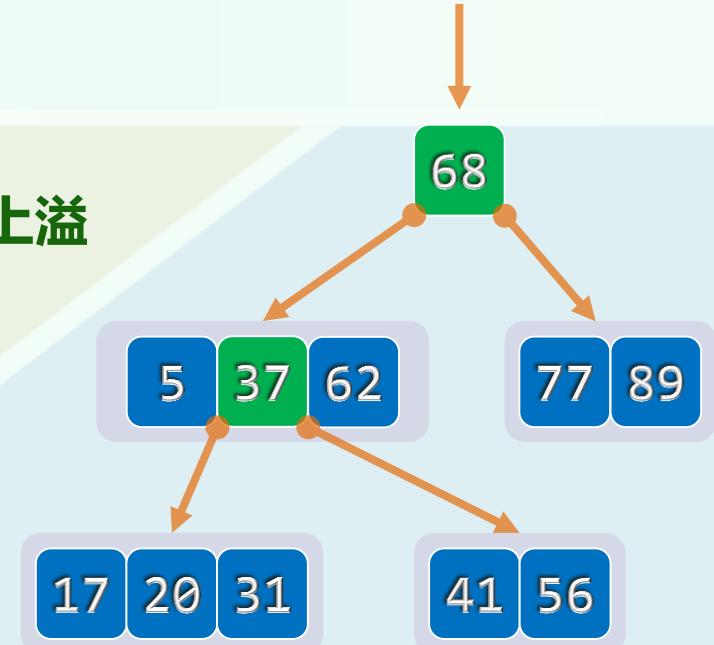
❖ 可令被提升的关键码自成节点，作为新的树根

这是B-树增高的唯一可能 //概率多大？

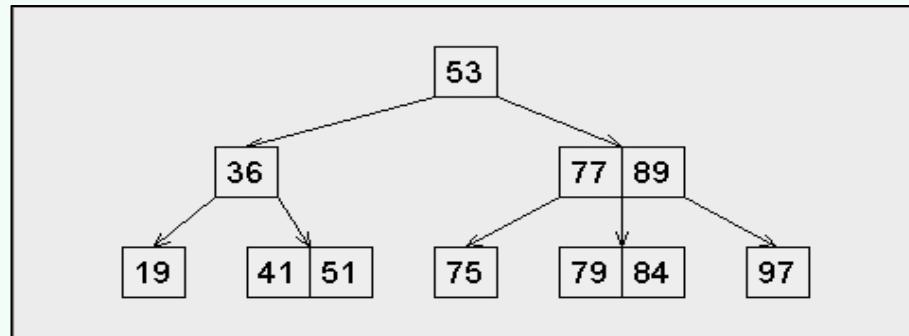
❖ 注意：新生的树根仅有两个分支

❖ 总体执行时间正比于

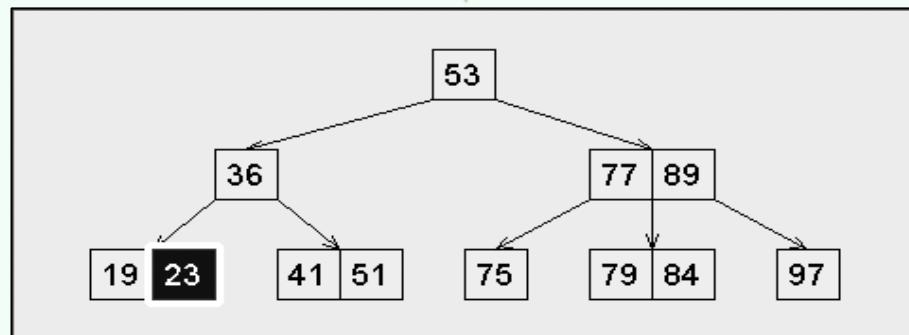
分裂次数， $\theta(h)$



实例：(2,3)-树

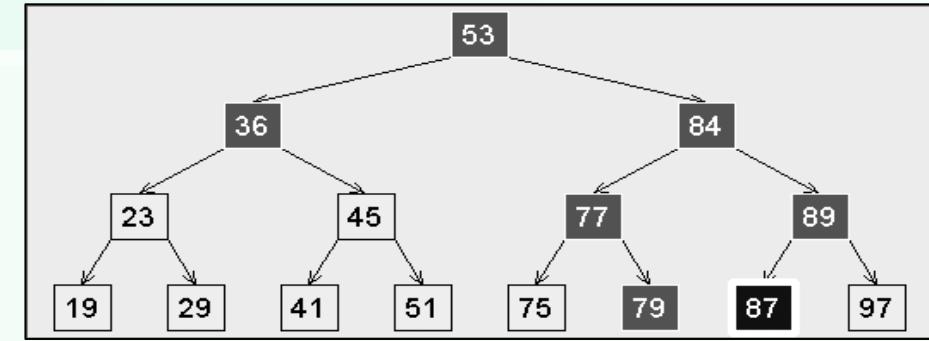


insert(23) //无需分裂

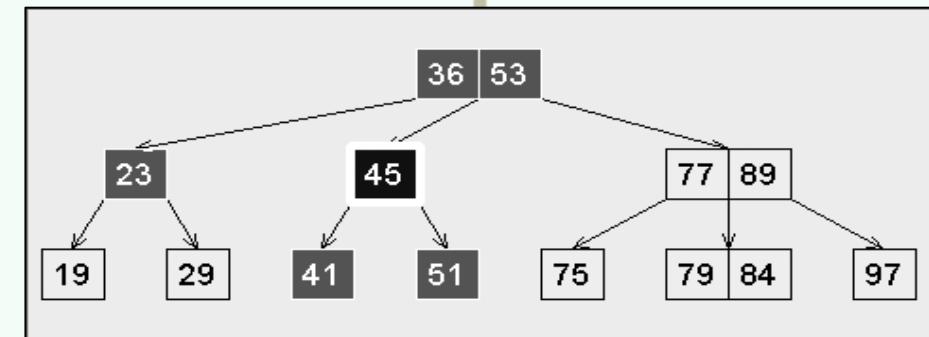


insert(29) //分裂一次

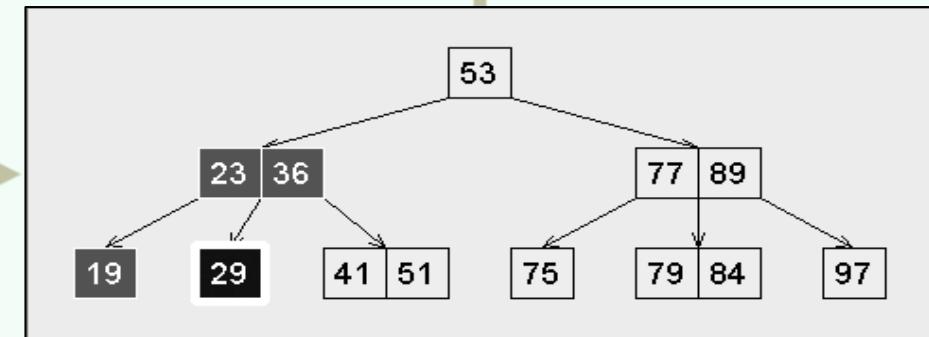
❖ 53 97 36 89 41 75 19 84 77 79 51



insert(87) //分裂到根

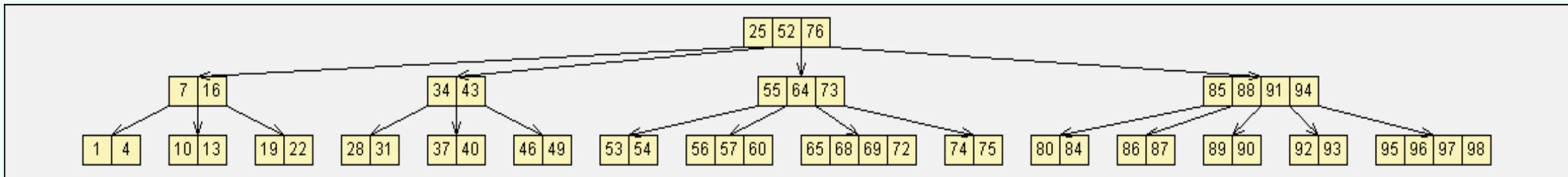


insert(45) //分裂两次

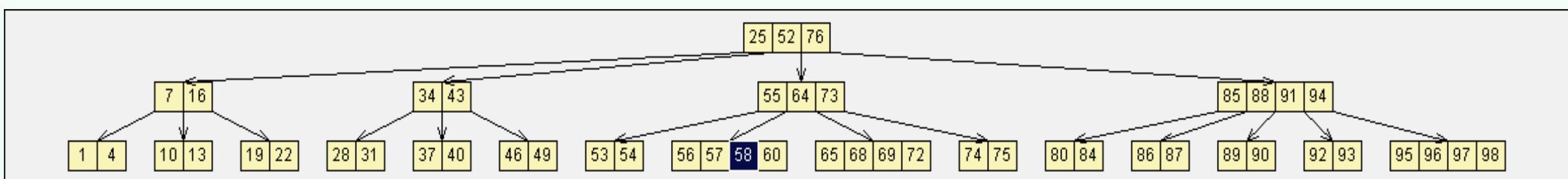


实例：(3,5)-树

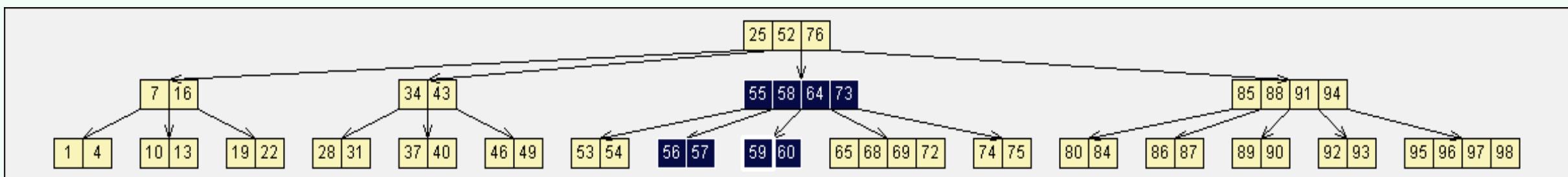
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 56 60 64 68 72 76 80 84 53 54 55 85 86 87 88 89 90 91 92 93 94 95 96 97 98 73 74 75 57 65 69



insert(58) //无需分裂

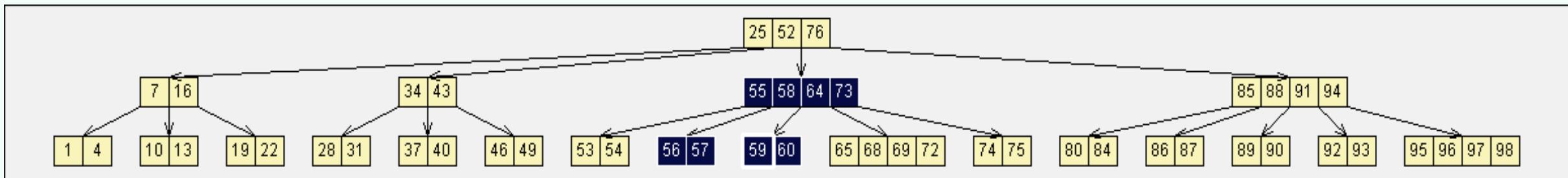


insert(59) //分裂 1 次

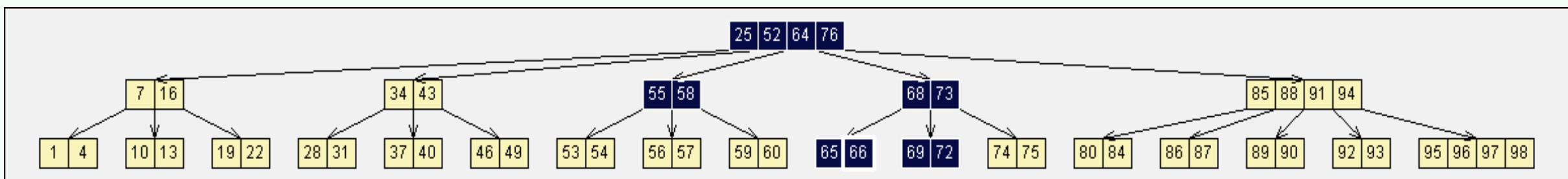


实例：(3,5)-树

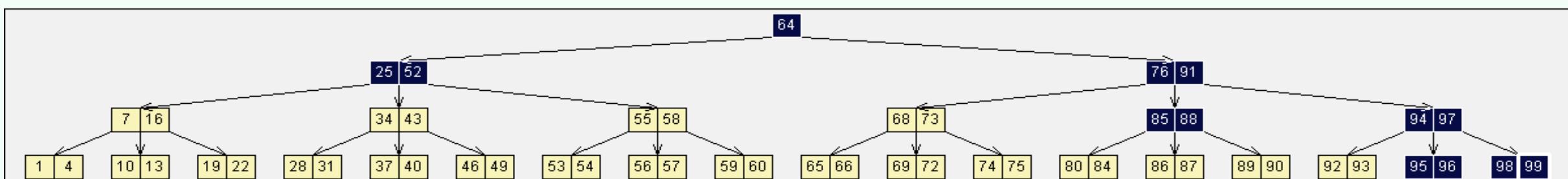
insert(59) //分裂 1 次



insert(66) //分裂 2 次



insert(99) //分裂到根



上溢修复 (1/2)

```
template <typename T> void BTree<T>::solveOverflow( BTNodePosi<T> v ) {
    while ( _m <= v->key.size() ) { //除非当前节点不再上溢
        Rank s = _m / 2; //轴点 (此时 _m = key.size() = child.size() - 1)
        BTNodePosi<T> u = new BTNode<T>(); //注意：新节点已有一个空孩子
        for ( Rank j = 0; j < _m - s - 1; j++ ) { //分裂出右侧节点u (效率低可改进)
            u->child.insert( j, v->child.remove( s + 1 ) ); //v右侧 _m-s-1 个孩子
            u->key.insert( j, v->key.remove( s + 1 ) ); //v右侧 _m-s-1 个关键码
        }
        u->child[ _m - s - 1 ] = v->child.remove( s + 1 ); //移动v最靠右的孩子
        /* ... TBC ... */
    }
}
```

上溢修复 (2/2)

```
if ( u->child[ 0 ] ) //若u的孩子们非空，则统一令其以u为父节点
    for ( Rank j = 0; j < _m - s; j++ ) u->child[ j ]->parent = u;
BTNodePosi<T> p = v->parent; //v当前的父节点p
if ( ! p ) //若p为空，则创建之（全树长高一层，新根节点恰好两度）
{ _root = p = new BTNode<T>(); p->child[0] = v; v->parent = p; }
Rank r = 1 + p->key.search( v->key[0] ); //p中指向u的指针的秩
p->key.insert( r, v->key.remove( s ) ); //轴点关键码上升
p->child.insert( r + 1, u ); u->parent = p; //新节点u与父节点p互联
v = p; //上升一层，如有必要则继续分裂——至多 $\log n$ 层
} //while
} //solveOverflow
```

高级搜索树

B-树：删除

08-B6

邓俊辉

deng@tsinghua.edu.cn

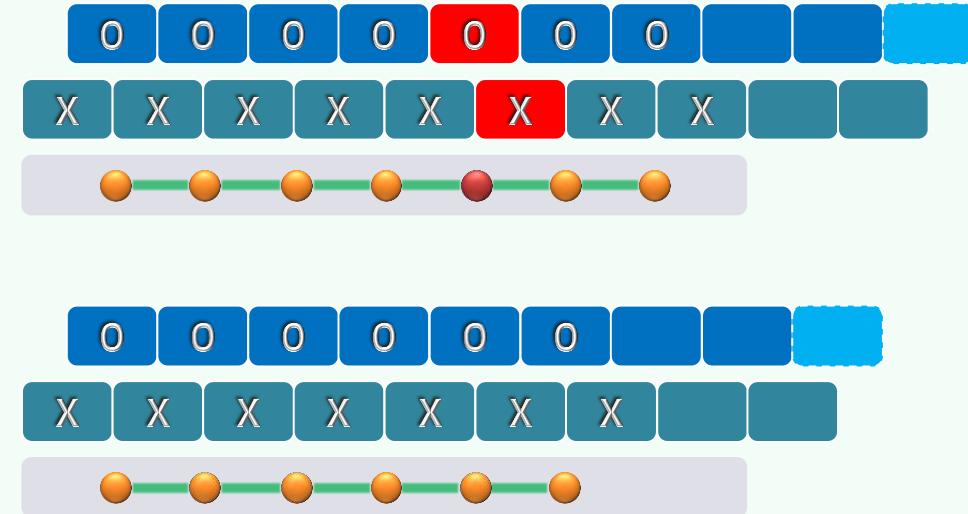
射影，变了形，反而结晶

或动了情，也要合并，或归了零

也不愿不生不死不悔的倒影

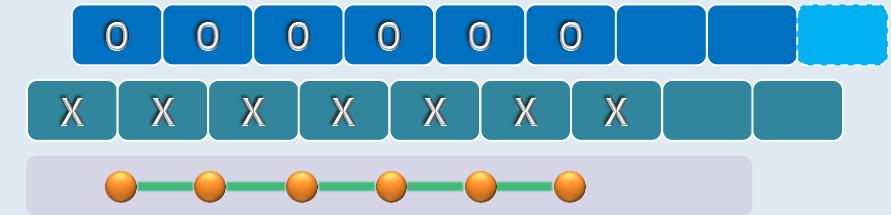
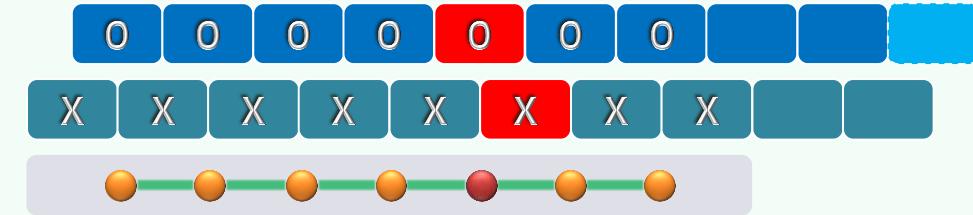
算法：确保目标在叶子中

```
template <typename T>
bool BTTree<T>::remove( const T & e ) {
    BTNodePosi<T> v = search( e );
    if ( ! v ) return false; //确认e存在
    Rank r = v->key.search(e); //e在v中的秩
    if ( v->child[0] ) { /* 若v非叶子，则可经过腾挪，确保... */ }
    //assert: 至此，v必位于最底层，且其中第r个关键码就是待删除者
    v->key.remove( r ); v->child.remove( r + 1 ); _size--;
    solveUnderflow( v ); return true; //如有必要，需做旋转或合并
}
```



算法：腾挪 = 与后继交换

```
template <typename T>  
bool BTTree<T>::remove( const T & e ) {  
    /* ..... */  
  
    if ( v->child[0] ) { //若v非叶子，则  
        BTNodePosi<T> u = v->child[r + 1]; //在右子树中  
  
        while ( u->child[0] ) u = u->child[0]; //一直向左，即可找到e的后继（必在底层）  
  
        v->key[r] = u->key[0]; v = u; r = 0; //交换  
    }  
  
    //assert: 至此，v必位于最底层，且其中第r个关键码就是待删除者  
  
    /* ..... */
```



旋转

❖ 非根节点 V 下溢时，必恰有 $\lceil m/2 \rceil - 2$ 个关键码和 $\lceil m/2 \rceil - 1$ 个分支

❖ 视其左、右兄弟 L 、 R 的规模，可分三种情况加以处理

1) 若 L 存在，且至少包含 $\lceil m/2 \rceil$ 个关键码

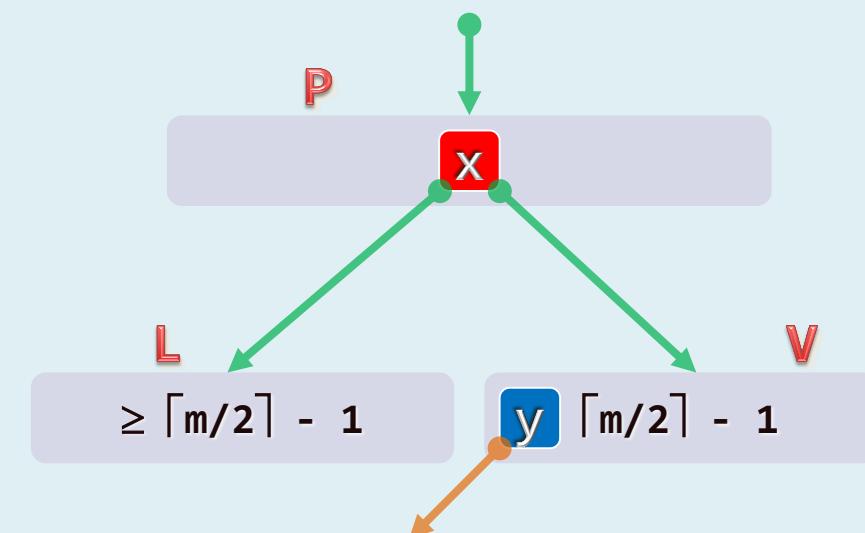
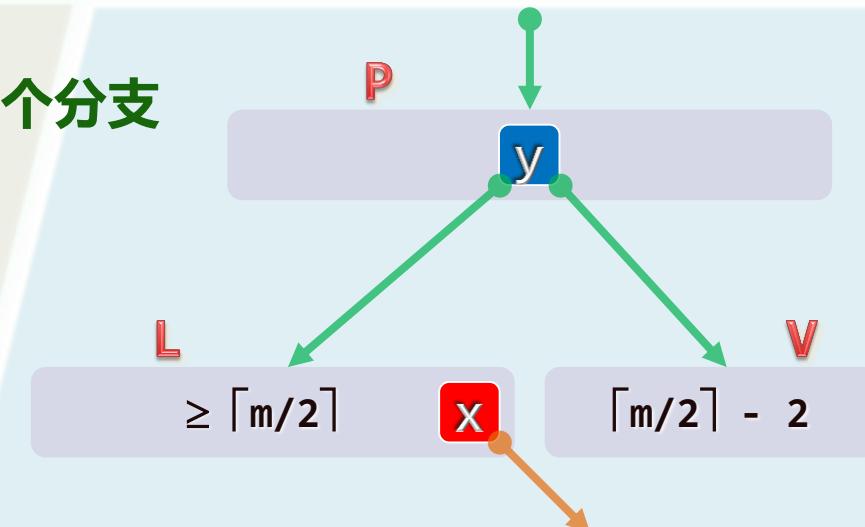
- 将 P 中的分界关键码 y 移至 V 中（作为最小关键码）
- 将 L 中的最大关键码 x 移至 P 中（取代原关键码 y ）

❖ 如此旋转之后，局部乃至全树都重新满足B-树条件

下溢修复完毕

2) 若 R 存在，且至少包含 $\lceil m/2 \rceil$ 个关键码

- 也可旋转，完全对称



合并

3) \boxed{L} 和 \boxed{R} 或不存在，或均不足 $\lceil m/2 \rceil$ 个关键码——即便如此

- \boxed{L} 和 \boxed{R} 仍必有其一（不妨以 \boxed{L} 为例），且
- 恰含 $\lceil m/2 \rceil - 1$ 个关键码

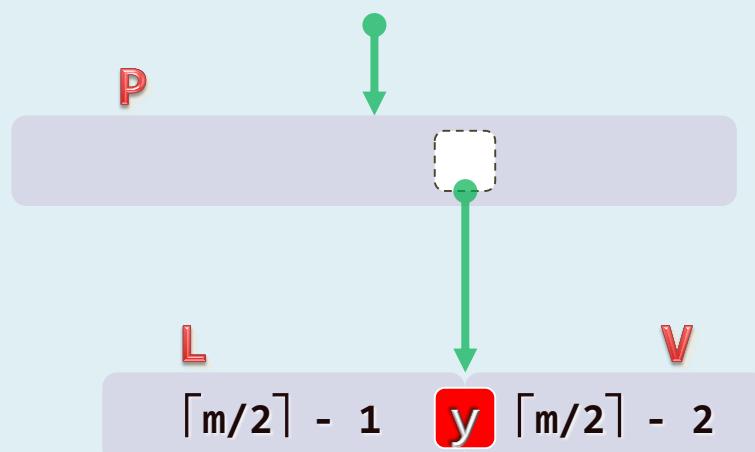
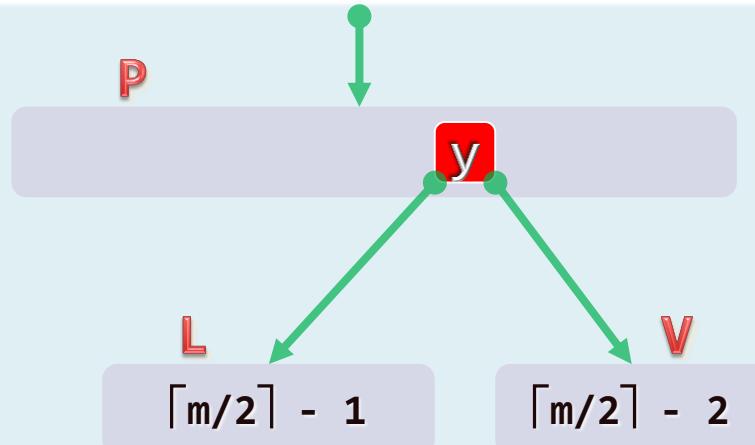
❖ 从 \boxed{P} 中抽出介于 \boxed{L} 和 \boxed{V} 之间的分界关键码 \boxed{y}

- 通过 \boxed{y} 做粘接，将 \boxed{L} 和 \boxed{V} 合成一个节点
- 同时合并此前 \boxed{y} 的孩子引用

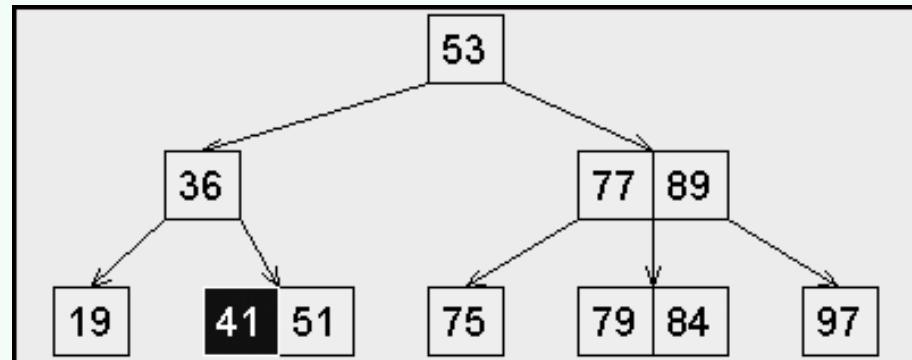
❖ 此处下溢得以修复，但可能继而导致 \boxed{P} 下溢

若果真如此，大可套用前法，继续旋转或合并

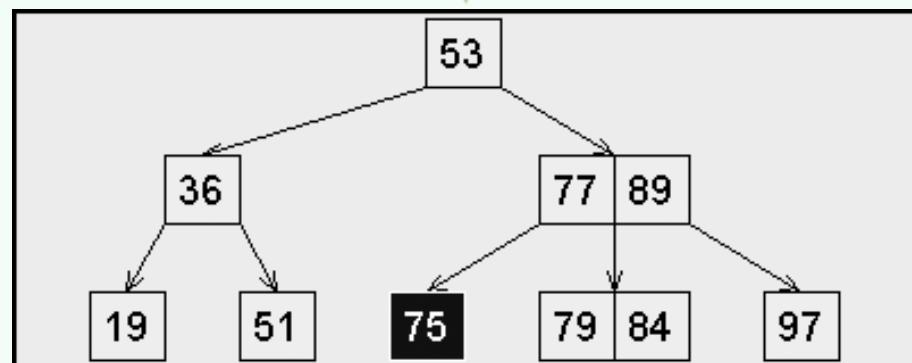
❖ 下溢可能持续发生并向上传播；但至多不过 $\Theta(h)$ 层



实例：(2,3)-树：底层节点

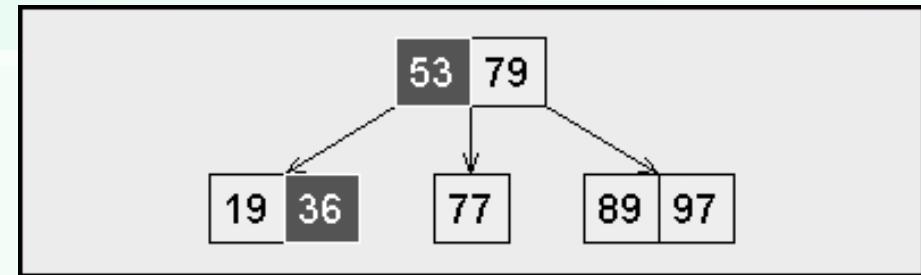


remove(41) //直接删除

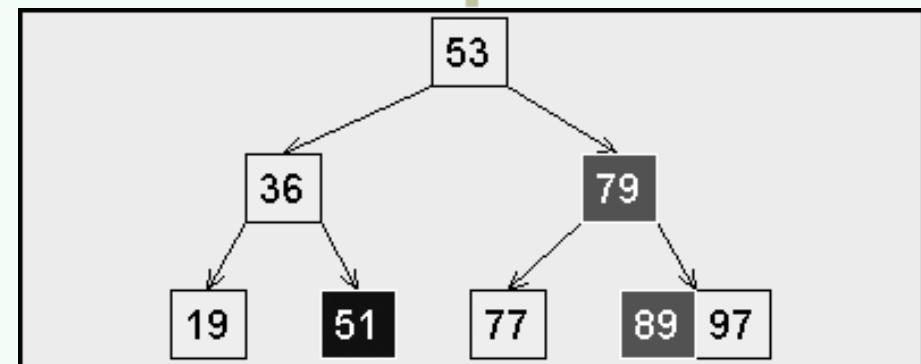


remove(75) //旋转

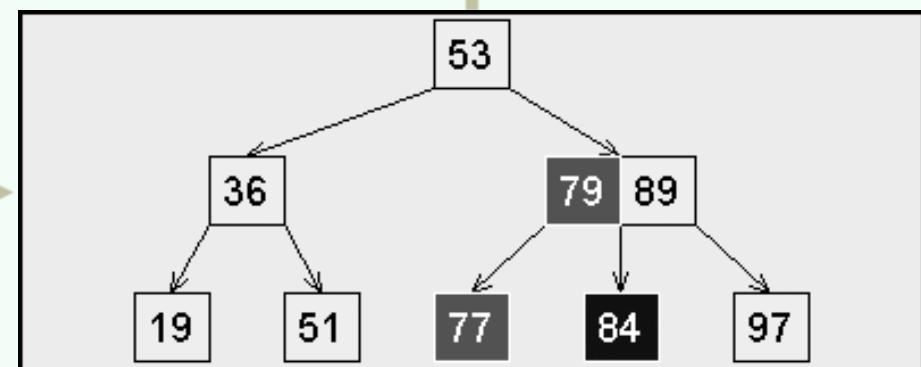
❖ 53 97 36 89 41 75 19 84 77 79 51



remove(51) //多次合并

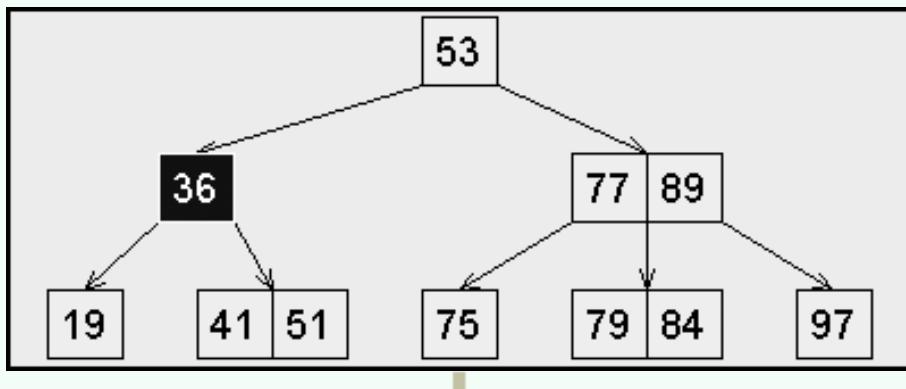


remove(84) //单次合并

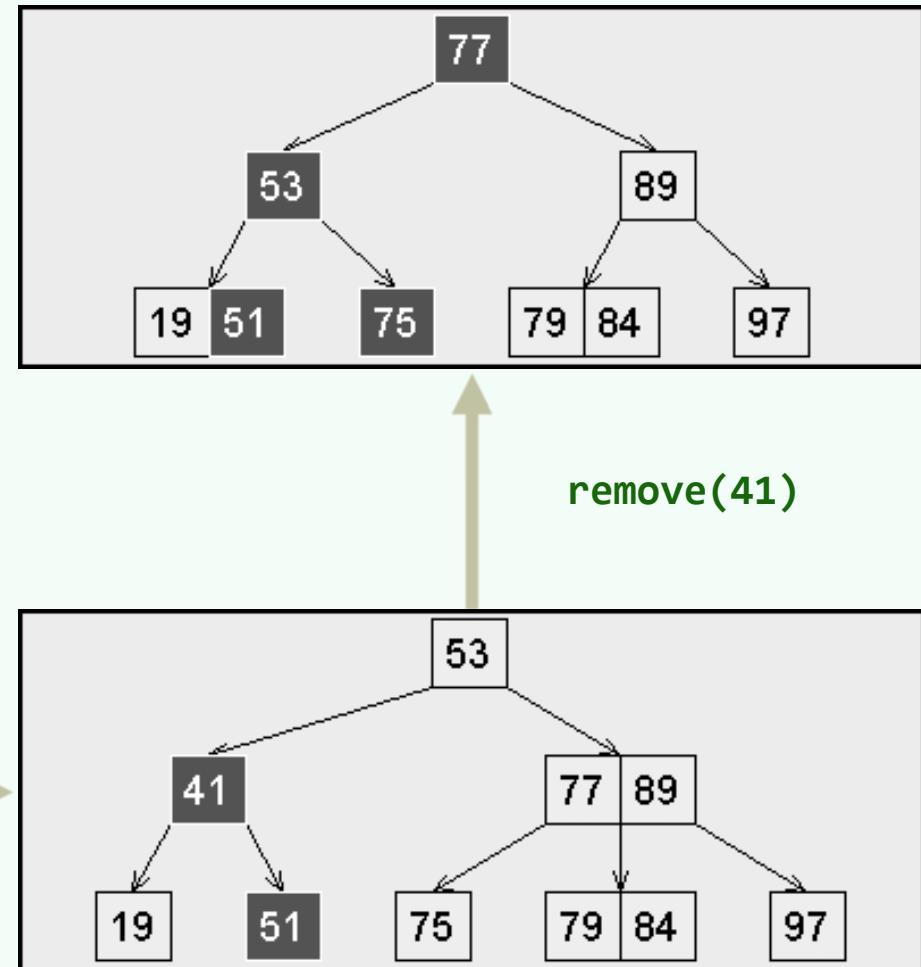


实例：(2,3)-树：非底层节点

❖ 53 97 36 89 41 75 19 84 77 79 51



remove(36)



下溢修复

```
template <typename T> void BTTree<T>::solveUnderflow( BTNodePosi<T> v ) {  
    while ( (_m + 1) / 2 > v->child.size() ) { //除非当前节点没有下溢  
        BTNodePosi<T> p = v->parent; if ( !p ) { /* 已到根节点 */ }  
        Rank r = 0; while ( p->child[r] != v ) r++; //确定v是p的第r个孩子  
        if ( 0 < r ) { /* 情况 #1: 若v的左兄弟存在, 且... */ }  
        if ( p->child.size() - 1 > r ) { /* 情况 #2: 若v的右兄弟存在, 且... */ }  
        if ( 0 < r ) { /* 与左兄弟合并 */ } else { /* 与右兄弟合并 */ } //情况 #3  
        v = p; //上升一层, 如有必要则继续旋转或合并——至多O(logn)层  
    } //while  
} //solveUnderflow
```

下溢修复：情况#1：旋转（向左兄弟借关键码）

```
if (0 < r) { //若v不是p的第一个孩子，则
    BTNodePosi<T> ls = p->child[r - 1]; //左兄弟必存在
    if ((_m + 1) / 2 < ls->child.size()) { //若该兄弟足够“胖”，则
        v->key.insert(0, p->key[r-1]); //p借出一个关键码给v（作为最小关键码）
        p->key[r - 1] = ls->key.remove(ls->key.size() - 1); //ls的最大key转入p
        v->child.insert(0, ls->child.remove(ls->child.size() - 1));
        //同时ls的最右侧孩子过继给v（作为v的最左侧孩子）
        if (v->child[0]) v->child[0]->parent = v;
    }
    return; //至此，通过右旋已完成当前层（以及所有层）的下溢处理
}
} //情况#2完全对称
```

下溢修复：情况#3：合并 (1/2)

```
if (0 < r) { //与左兄弟合并

BTNodePosi<T> ls = p->child[r-1]; //左兄弟必存在

ls->key.insert( ls->key.size(), p->key.remove(r - 1) );
p->child.remove( r ); //p的第r - 1个关键码转入ls, v不再是p的第r个孩子

ls->child.insert( ls->child.size(), v->child.remove( 0 ) );
if ( ls->child[ ls->child.size() - 1 ] ) //v的最左侧孩子过继给ls做最右侧孩子

ls->child[ ls->child.size() - 1 ]->parent = ls;

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

下溢修复：情况#3：合并 (2/2)

```
while ( !v->key.empty() ) { //v剩余的关键码和孩子，依次转入ls
    ls->key.insert( ls->key.size(), v->key.remove(0) );
    ls->child.insert( ls->child.size(), v->child.remove(0) );
    if ( ls->child[ ls->child.size() - 1 ] )
        ls->child[ ls->child.size() - 1 ]->parent = ls;
} //while
release(v); //释放v
} else
{ /* 与右兄弟合并，完全对称 */ }
```

高级搜索树

红黑树：动机

所有的过去
都留下了痕迹
哪怕是
一次最微妙的心动
一声最轻渺的叹息

e8 - c1

因为过去要进入未来，所以有了故事；因为在深夜里，你会想不起你是怎么从原来走到的现在，所以有了故事；当记忆被抹去，当你除了故事就再无任何可以去记忆、可以被记住的东西的时候，因为要有永恒，所以有了故事。

邓俊辉

deng@tsinghua.edu.cn

并发性: Concurrent Access To A Database

❖ 修改之前先加锁 (lock) ; 完成后解锁 (unlock)

访问延迟主要取决于 “lock/unlock” 周期

❖ 对于BST而言, 每次修改过程中, 唯

结构有变 (reconstruction) 处才需加锁

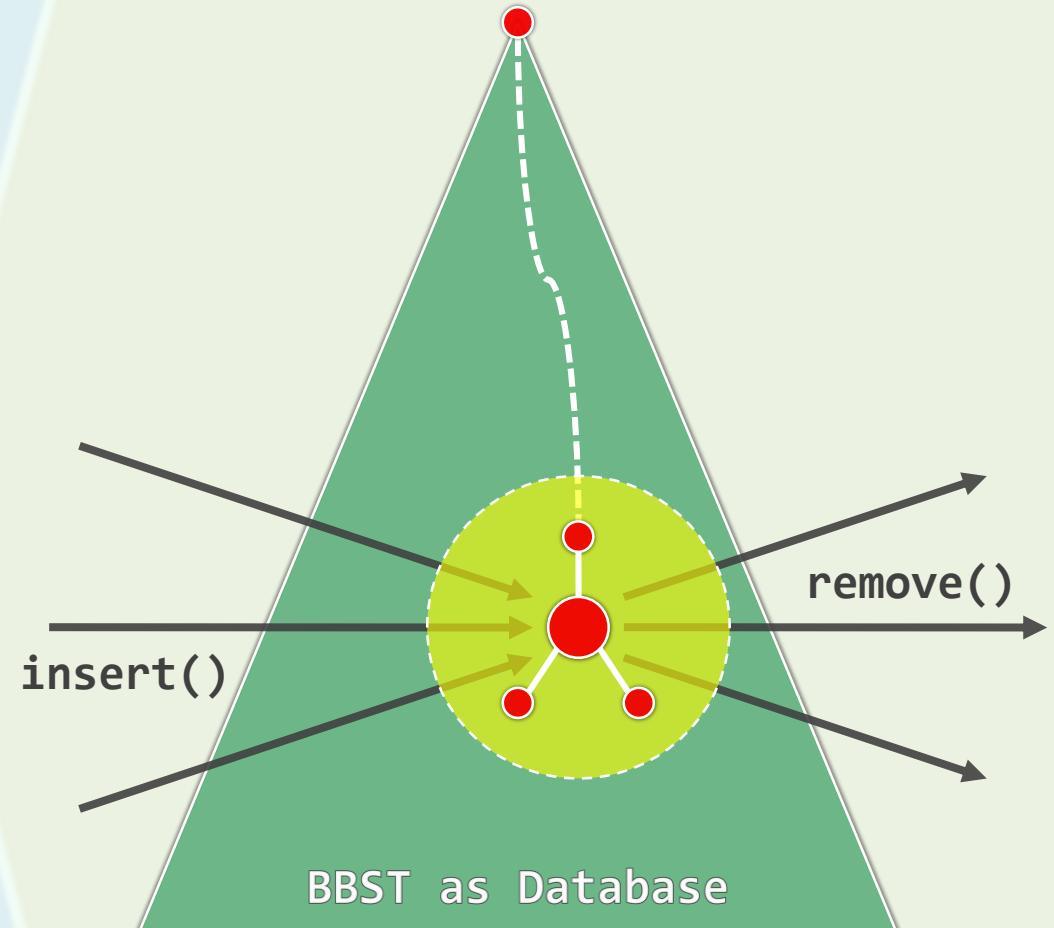
访问延迟主要取决于这类局部之数量...

❖ Splay: 结构变化剧烈, 最差可达 $\mathcal{O}(n)$

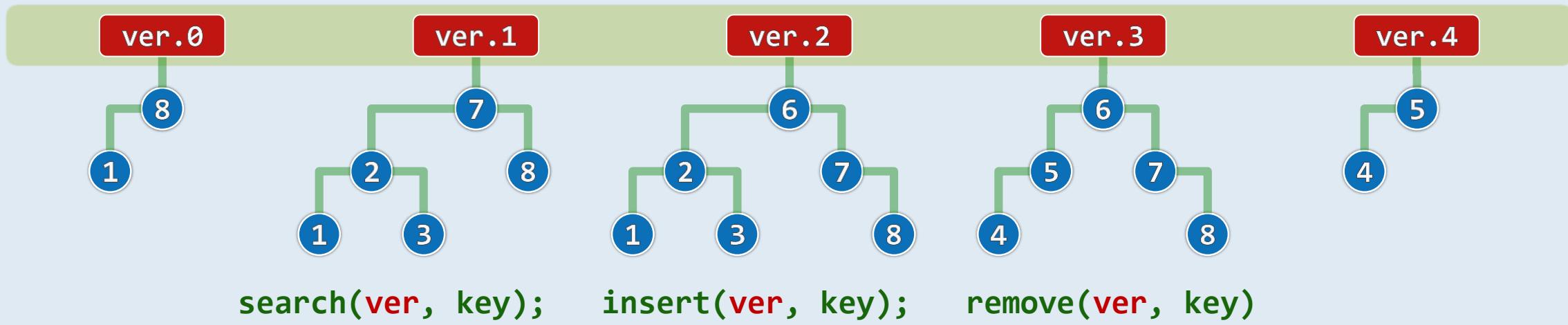
❖ AVL: $\text{remove}()$ 时 $\mathcal{O}(\log n)$ —— 尽管

$\text{insert}()$ 时可保证 $\mathcal{O}(1)$

❖ Red-Black: 无论 $\text{insert}/\text{remove}$, 均不超过 $\mathcal{O}(1)$



❖ 蛮力实现: 每个版本独立保存; 各版本自成一个搜索结构



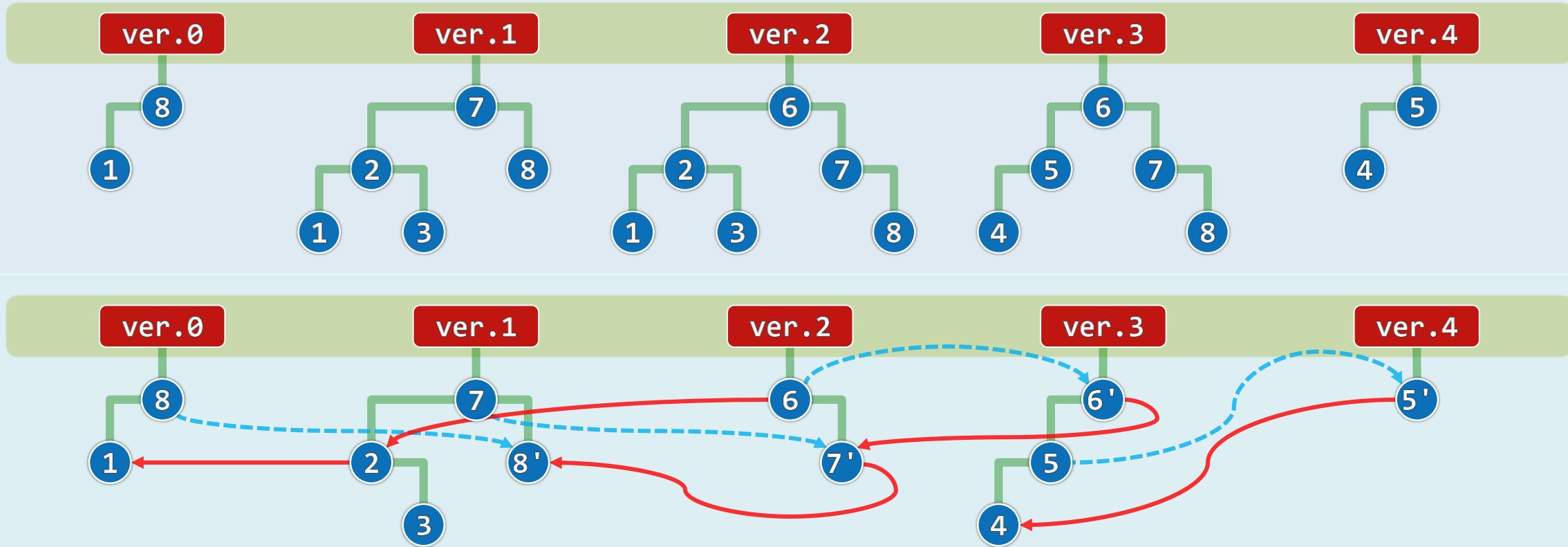
❖ 单次操作 $\mathcal{O}(\log h + \log n)$, 累计 $\mathcal{O}(n \cdot h)$ 时间/空间 // $h = |\text{history}|$

❖ 挑战: 可否将复杂度控制在 $\mathcal{O}(n + h \cdot \log n)$ 以内?

❖ 可以! 为此需利用相邻版本之间的**相关性**...

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

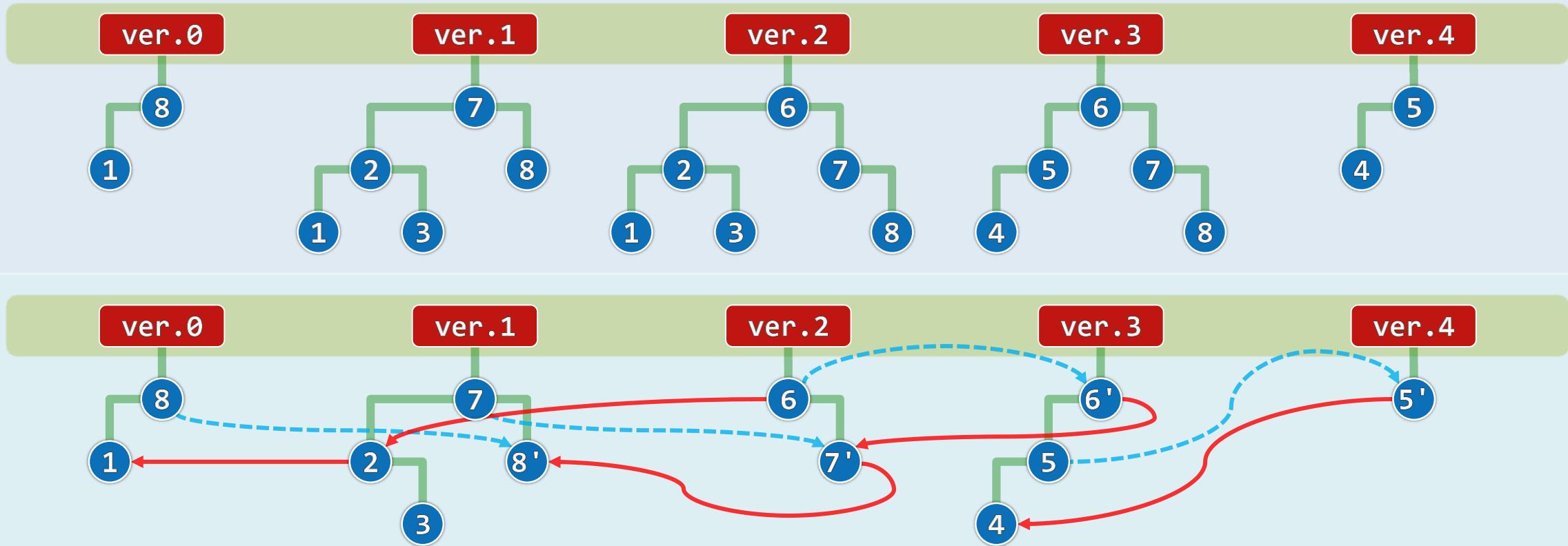
❖ **Partial Persistence**: 仅支持对历史版本的读取 // 监控录像、飞行器黑盒子、代码版本管理...



❖ 这类情况下，还可进一步提高至总体 $\mathcal{O}(n + h)$ 、单版本 $\mathcal{O}(1)$...

$\mathcal{O}(1)$ 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 $\mathcal{O}(1)$



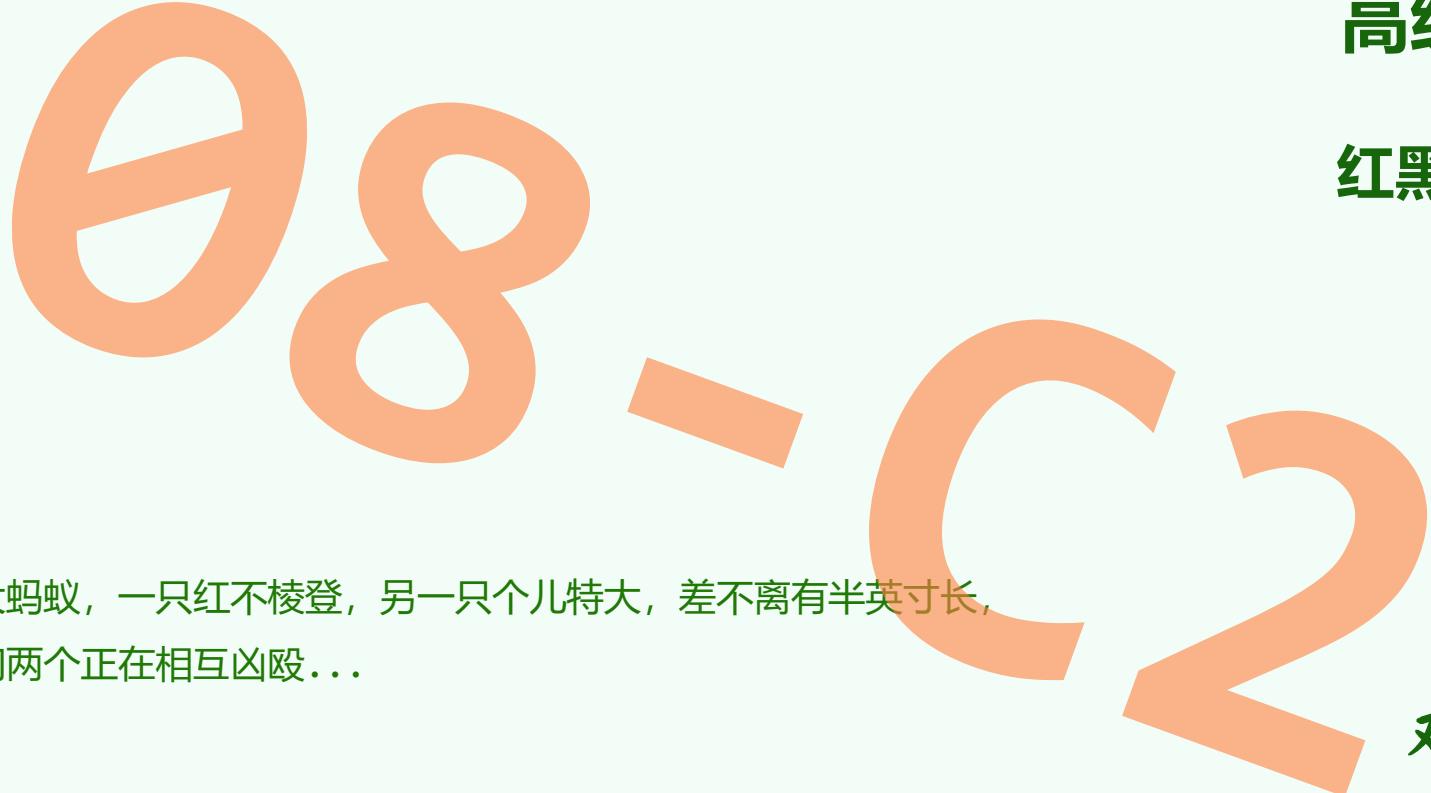
❖ 很遗憾，AVL、Splay等BBST均不具备这一性质；须另辟蹊径...

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 );  
  
    }  
  
}
```

高级搜索树

红黑树：结构



这时，我看见两只大蚂蚁，一只红不棱登，另一只个儿特大，差不离有半英寸长，
是黑不溜秋的，它们两个正在相互凶殴...

玉帝即传旨宣托塔李天王，教：“把照妖镜来照这厮谁真谁假，教他假灭真存。”

邓俊辉

deng@tsinghua.edu.cn

红与黑

❖ 1972, R. Bayer

Symmetric Binary B-Tree

❖ 1978, L. Guibas & R. Sedgewick

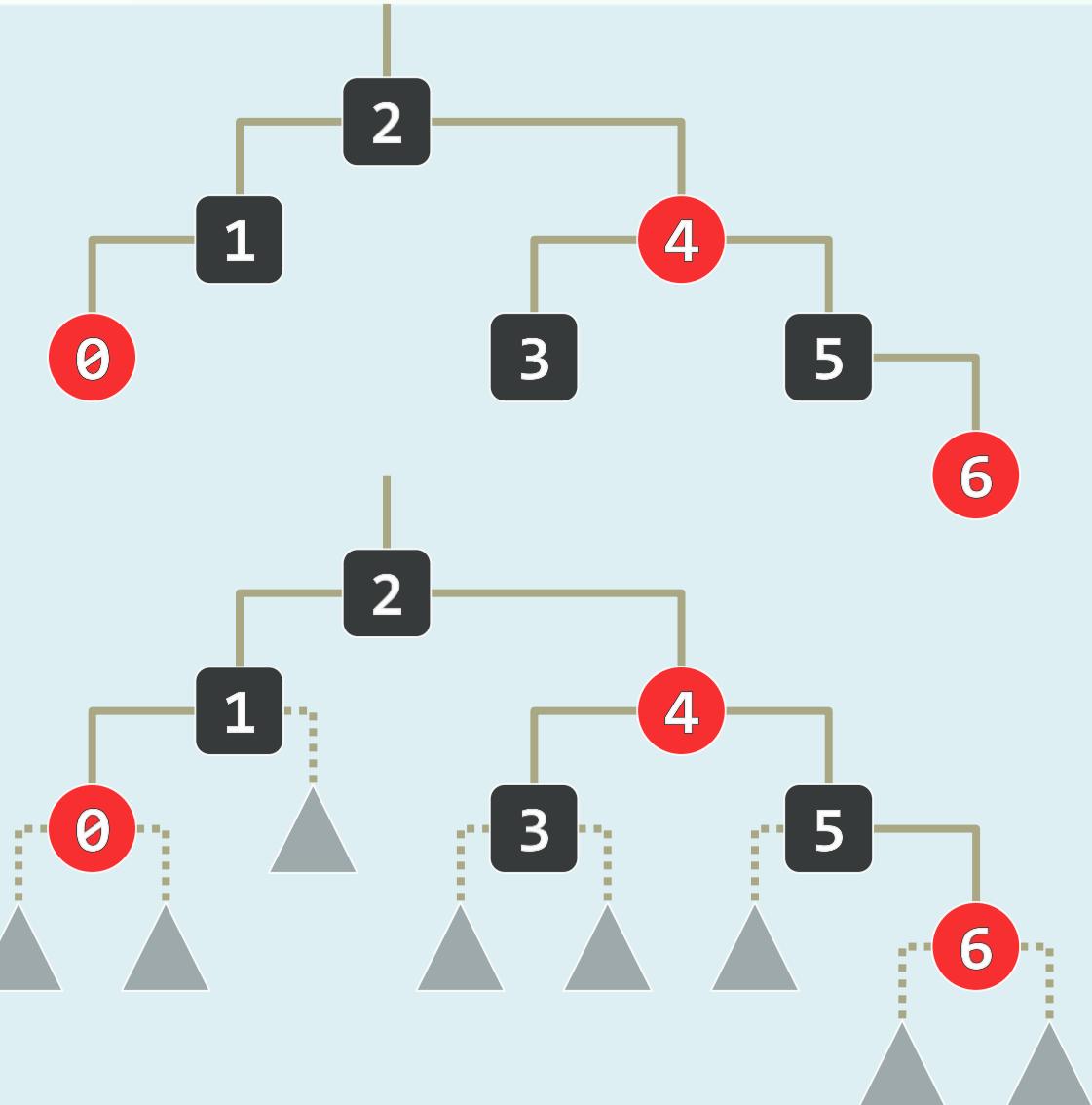
Red-Black Tree

❖ 1982, H. Olivie

Half-Balanced Binary Search Tree

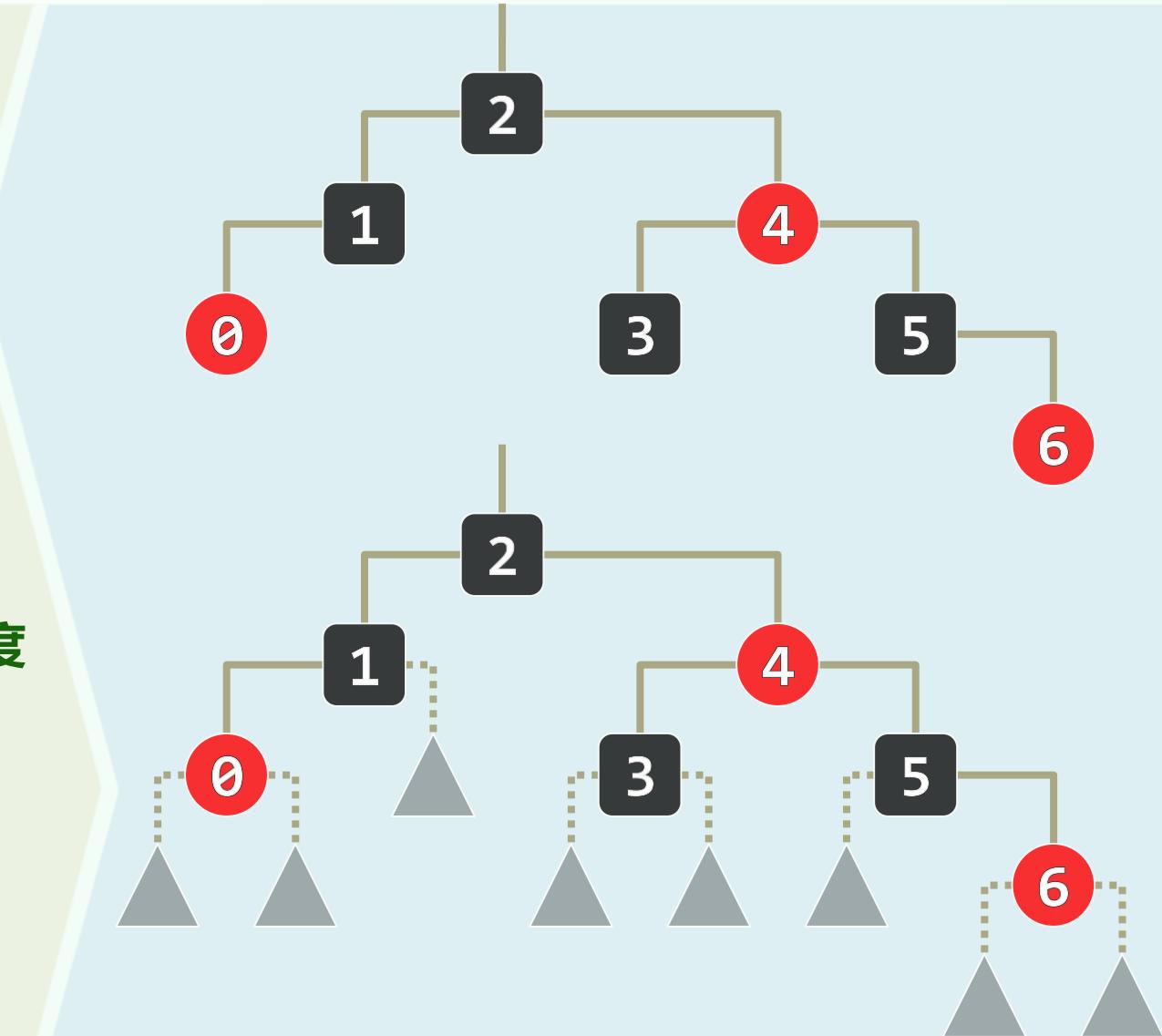
❖ 由红、黑两类节点组成的BST

统一增设外部节点NULL，使之成为真二叉树

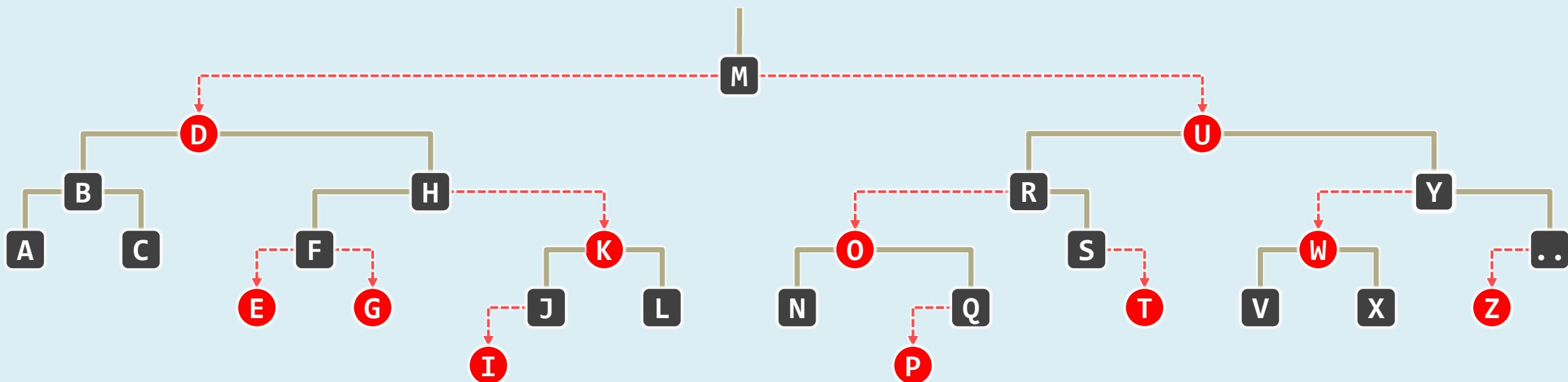
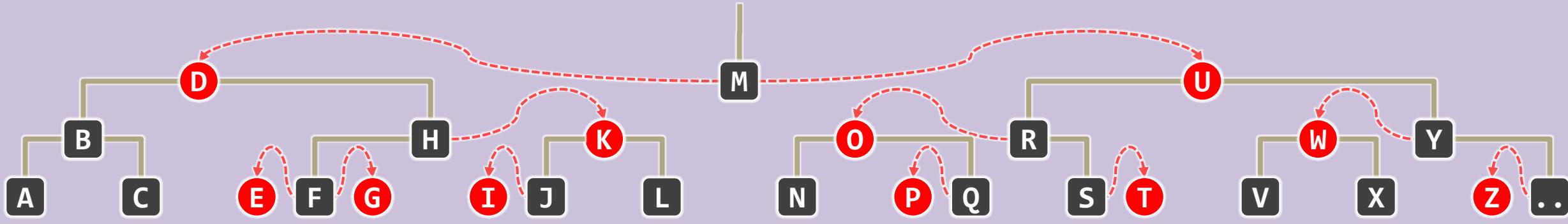


规则

- 1) 树根: 必为黑色
 - 2) 外部节点: 均为黑色
 - 3) 红节点: 只能有黑孩子 (及黑父亲)
 - 4) 外部节点: 黑深度 (黑的真祖先数目) 相等
 - 亦即根 (全树) 的黑高度
 - 子树的黑高度, 即后代NULL的相对黑深度
- ❖ 节点的颜色, 只能显式地记录?
 - ❖ 以上定义颇为费解, 有直观解释吗?
 - 如此定义的BST, 也是BBST?

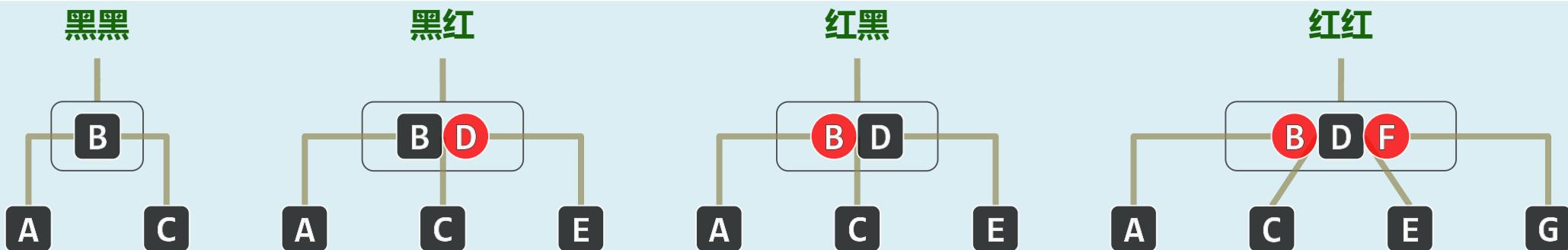
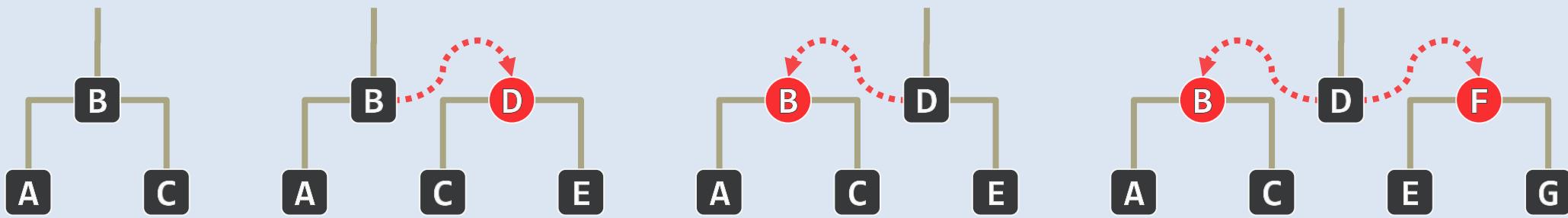


将红节点提升至与其（黑）父亲等高，红边折叠起来...



红黑树 = (2,4)-树

- 将红节点提升至与其（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，再合并为B-树的超级节点... //元音 + 辅音 = 音节
- 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？



红黑树 \in BBST

- ❖ 包含 n 个内部节点的红黑树 T , 高度 $h = \mathcal{O}(\log n)$ //既然B-树是平衡的, 由等价性红黑树也应是

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

- ❖ 若 T 高度为 h , 红/黑高度为 R/H , 则

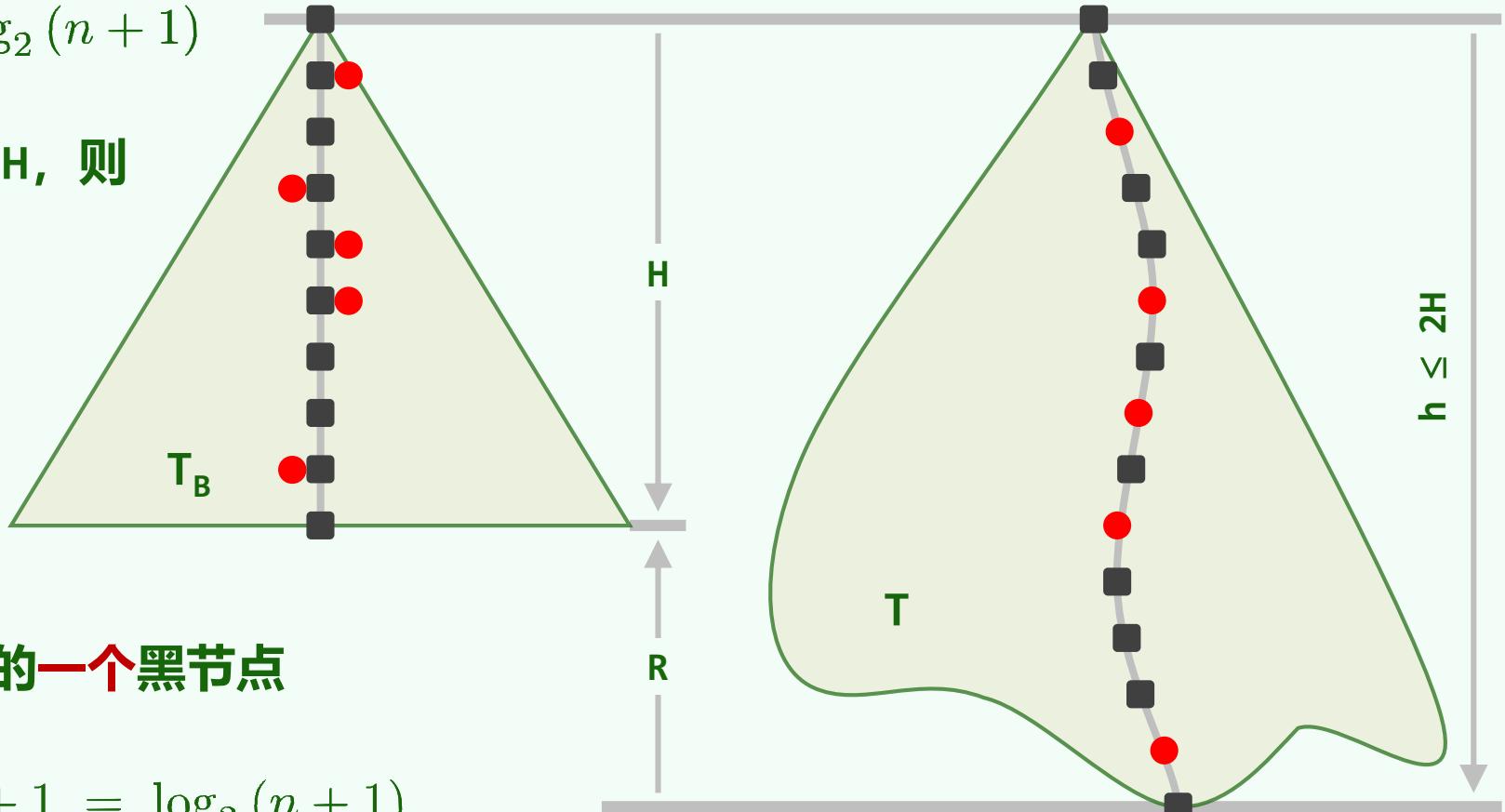
$$H \leq h \leq R + H \leq 2 \cdot H$$

- ❖ 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, 都恰好包含 T 的一个黑节点

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



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
                Rank updateHeight( BinNodePosi<T> x ); //更新节点x的高度 (重写)
};

#define stature( p ) ( ( p ) ? ( p )->height : 0 ) //外部节点黑高度为0, 以上递推

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

高级搜索树

红黑树：插入

08 - C3

邓俊辉

deng@tsinghua.edu.cn

莫赤匪狐，莫黑匪乌；惠而好我，携手同车

算法

❖ 按BST规则插入关键码e //`x = insert(e)`必为叶节点

❖ 除非系首个节点（根），`x`的父亲`p = x->parent`必存在

首先将`x`染红 //`x->color = isRoot(x) ? B : R`

❖ 至此，条件1、2、4依然满足；

但3不见得，有可能...

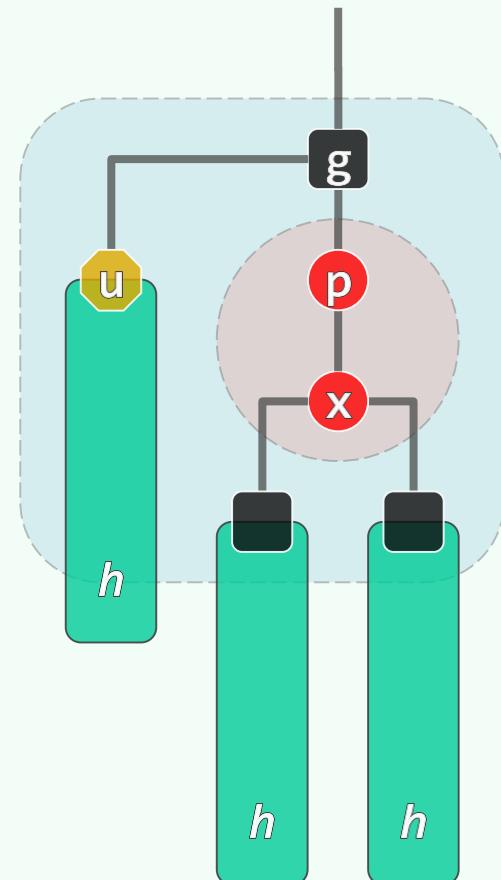
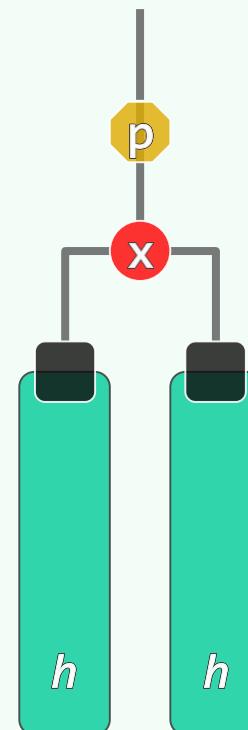
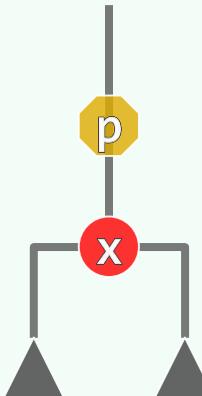
❖ 双红 (double-red)

//`p->color == x->color == R`

❖ 考查：祖父`g = p->parent`

叔父`u = uncle(x) = sibling(p)`

❖ 以下，视`u`的颜色，分两种情况处理...



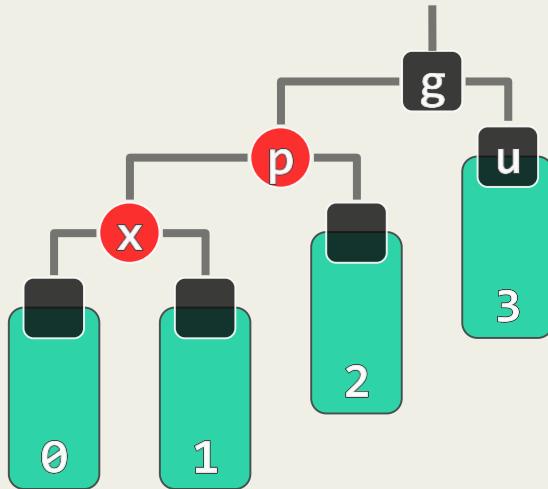
实现

```
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 = 0  
  
    x = new BinNode<T>( e, _hot, NULL, NULL, 0 ); _size++;  
  
    // 如有必要, 需做双红修正, 再返回插入的节点  
  
    BinNodePosi<T> xOld = x; solveDoubleRed( x ); return xOld;  
  
} //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 ... */ }  
}
```

RR-1: $u \rightarrow \text{color} == \text{B}$

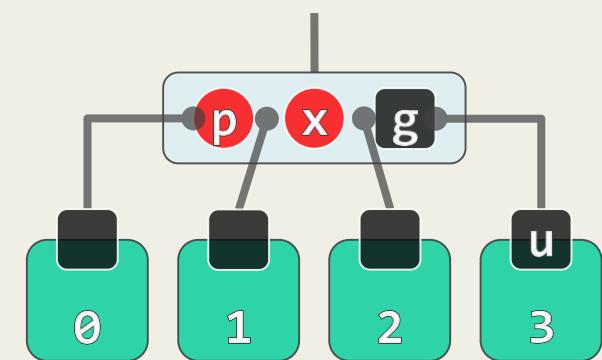
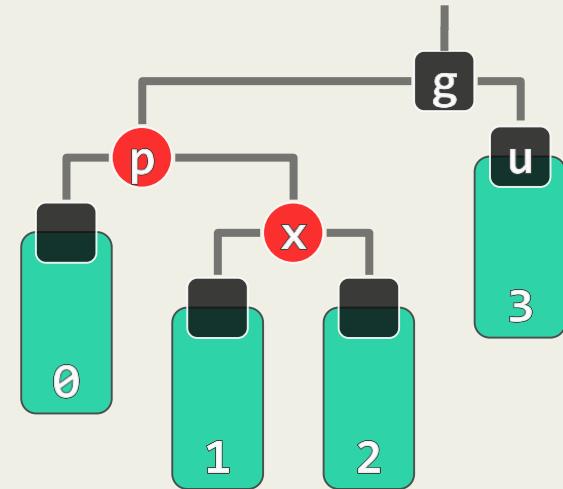
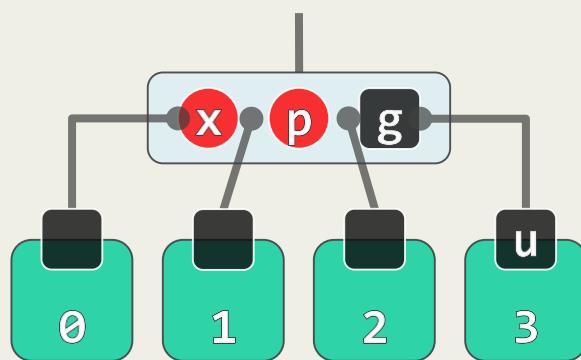


❖ 此时, x 、 p 、 g 的四个孩子

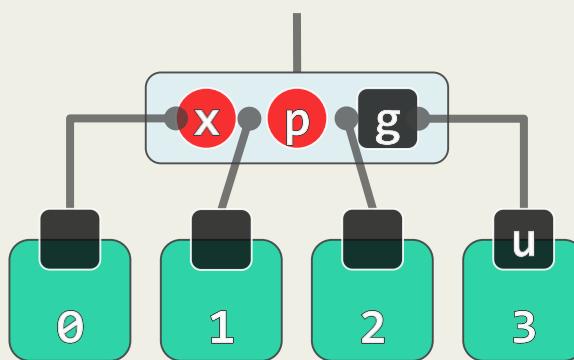
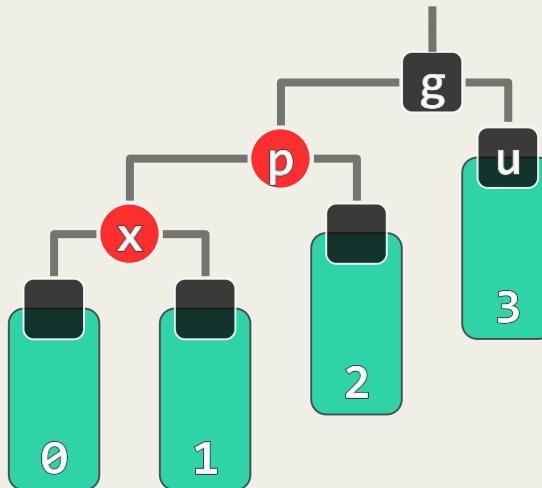
(可能是外部节点)

- 全为黑, 且
- 黑高度相同

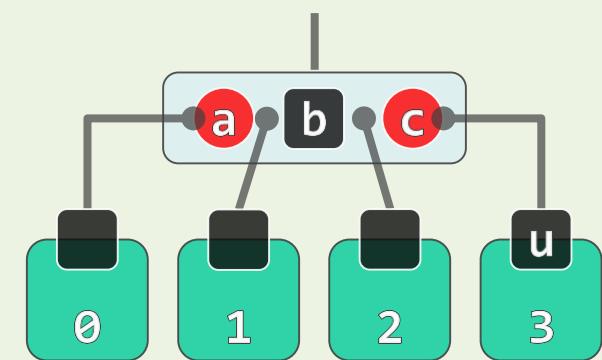
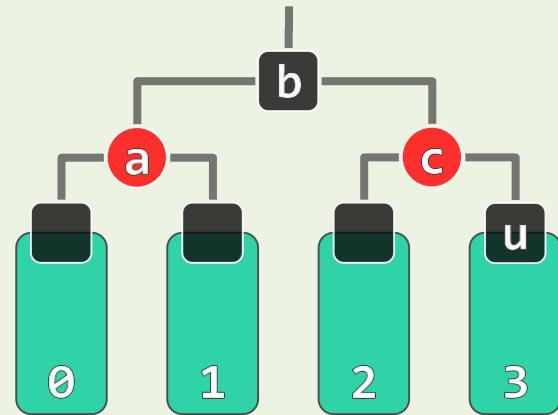
❖ 另两种对称情况, 自行补充



RR-1: $u \rightarrow \text{color} == \text{B}$



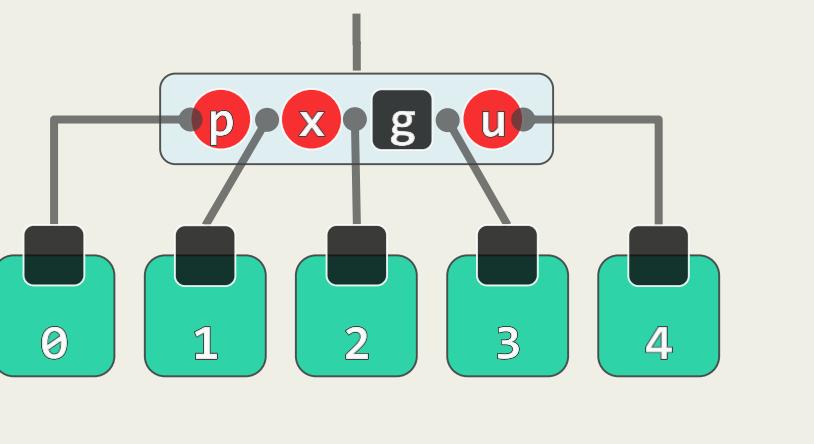
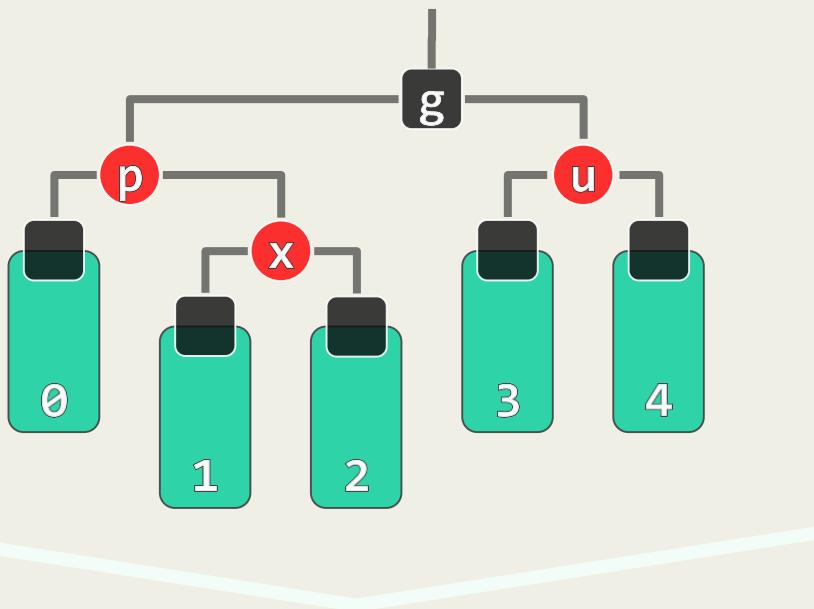
- ❖ 局部“3+4”重构
b转黑，a或c转红
- ❖ 从B-树的角度，如何理解？
所谓“非法”，无非是…
- ❖ 在某三叉节点中插入红关键码后
原黑关键码不再居中（RRB或BRR）
- ❖ 调整的效果，无非是
将三个关键码的颜色改为RBR
- ❖ 如此调整，一蹴而就



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

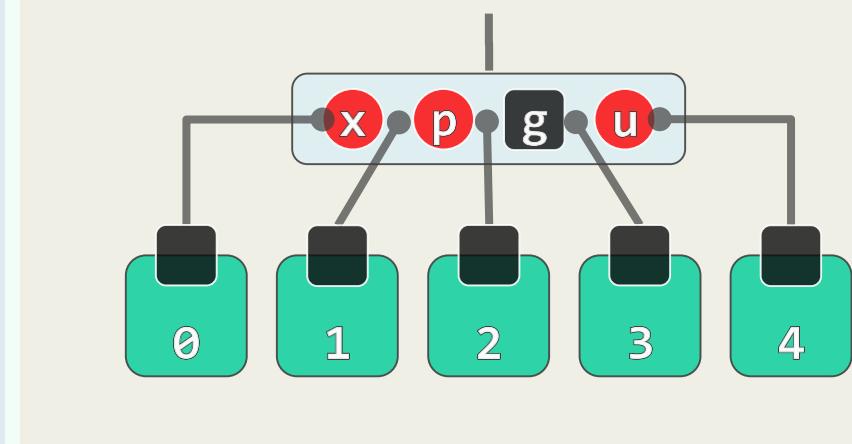
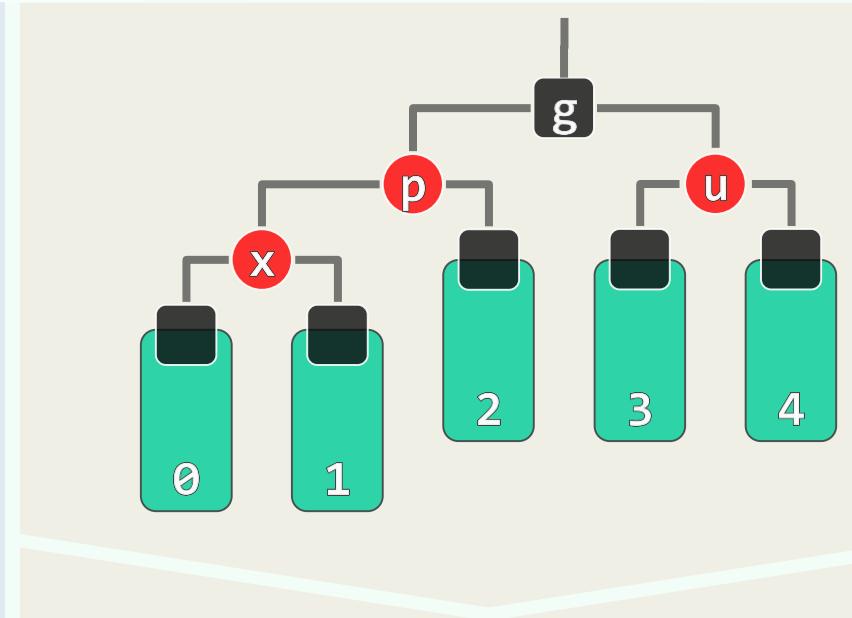


❖ 在B-树中，等效于

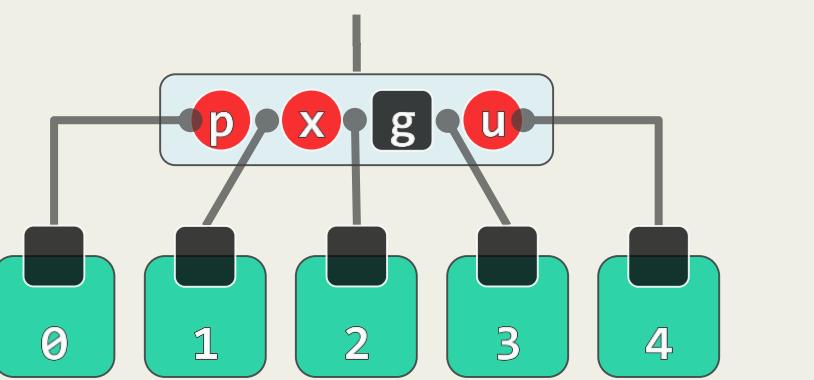
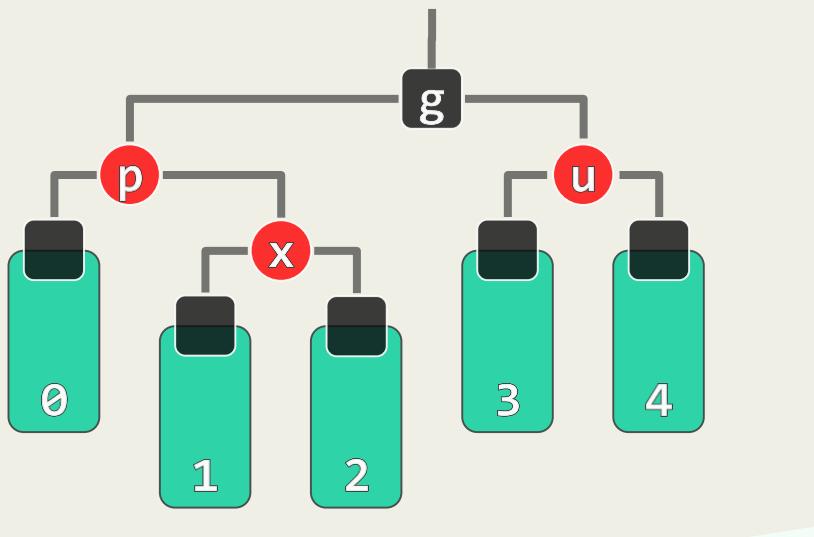
超级节点发生上溢

❖ 另两种对称情况

请自行补充



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

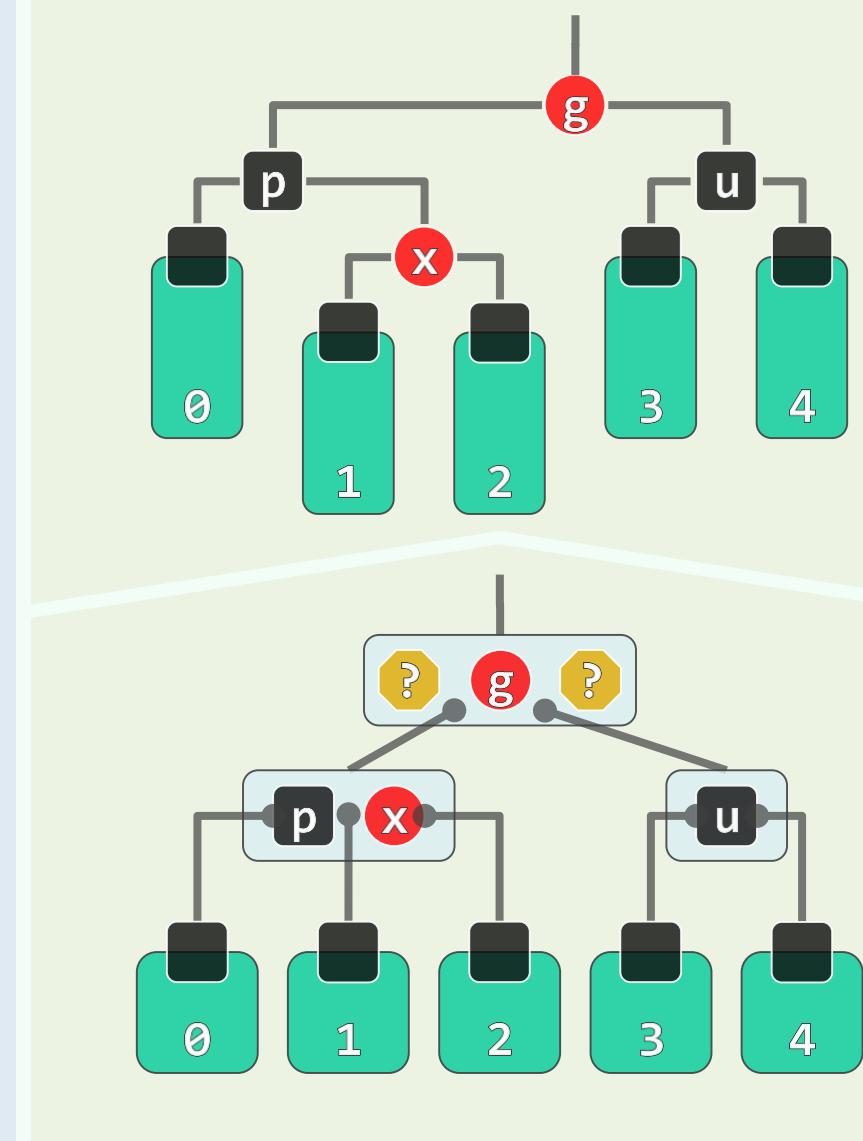


❖ p与u转黑，g转红

在B-树中，等效于...

❖ 节点分裂

关键码g上升一层



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

❖ 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

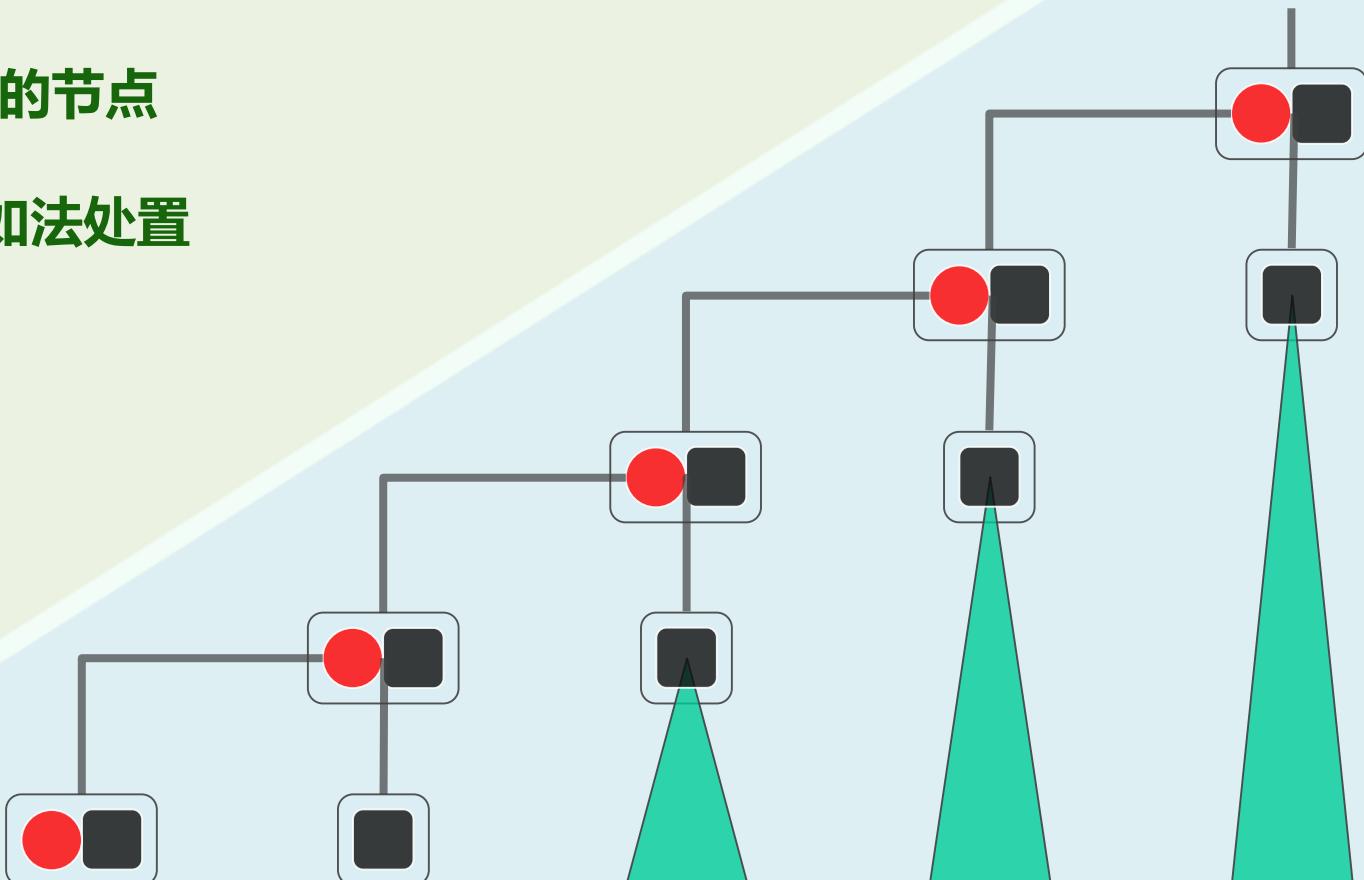
❖ 直到所有条件满足（即不再双红）

或者抵达树根

❖ g 若果真到达树根，则

强行将其转为黑色

（整树黑高度加一）



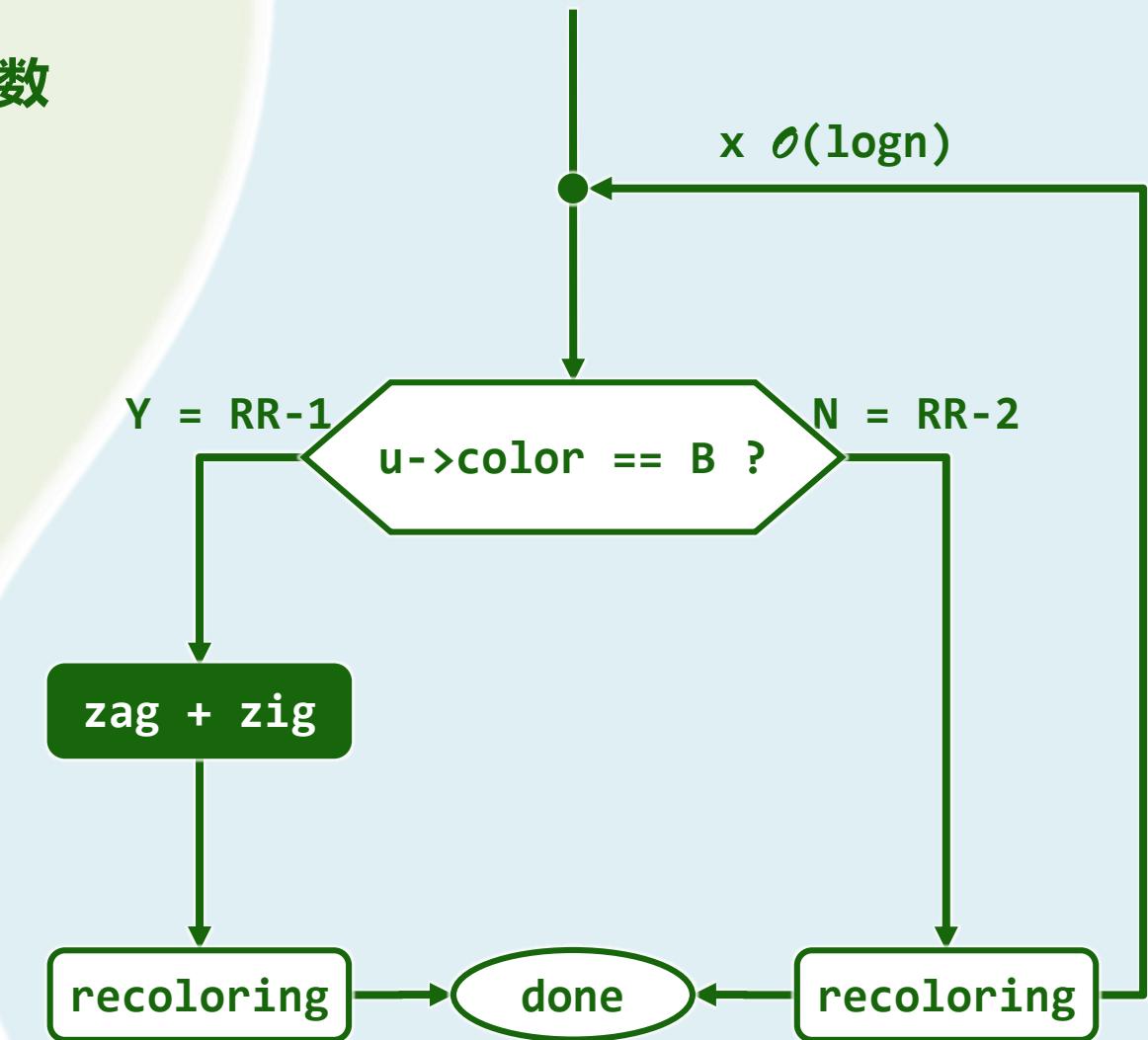
RR-2: 实现

```
template <typename T> void RedBlack<T>::solveDoubleRed( BinNodePosi<T> x ) {  
    /* ..... */  
  
    if ( IsBlack( u ) ) { /* ... u为黑 (含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  
  
        g->color = RB_RED; //在B-树中g相当于上交给父节点的关键码, 故暂标记为红  
        solveDoubleRed( g ); //继续调整: 若已至树根, 接下来的递归会将g转黑 (尾递归)  
    }  
}
```

复杂度

- ❖ 重构、染色均只需常数时间，故只需统计其总次数
- ❖ RedBlack::insert()仅需 $\mathcal{O}(\log n)$ 时间
- ❖ 其间至多做 $\mathcal{O}(\log n)$ 次重染色、 $\mathcal{O}(1)$ 次旋转

	旋转	染色	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



高级搜索树

红黑树：删除

08 - C4

他仿佛这一刻才第一次看见这些颜色，并为它们取下崭新又美妙的名字。

在这里，没有人会在冬天时哀悼已逝去的夏天或春天。

邓俊辉

deng@tsinghua.edu.cn

等效删除

❖ 首先按照BST常规算法，执行

```
r = removeAt( x, _hot )
```

//实际被摘除的可能是x的前驱或后继w

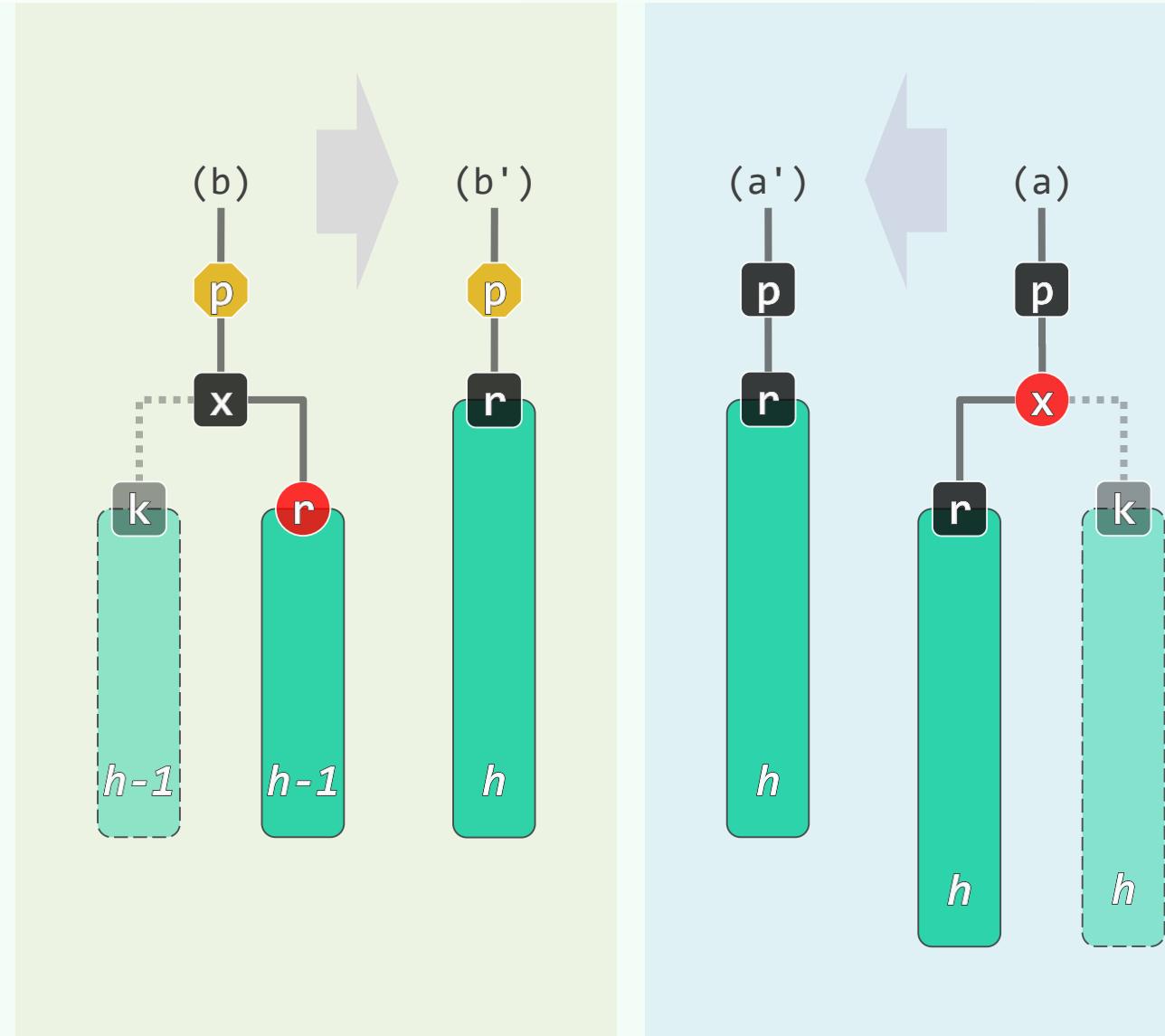
//简捷起见，以下不妨统称作x

❖ x由孩子r接替，此时另一孩子k必为NULL

❖ 但在随后的调整过程中，x可能逐层上升

❖ 故需假想地、统一地、等效地理解为：

- k为一棵黑高度与r相等的子树，且
- 随x一并摘除（尽管实际上从未存在过）



删除 (1/2)

```
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
```

删除 (2/2)

```
// 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;  
}
```

其一为红

❖ 完成removeAt()之后

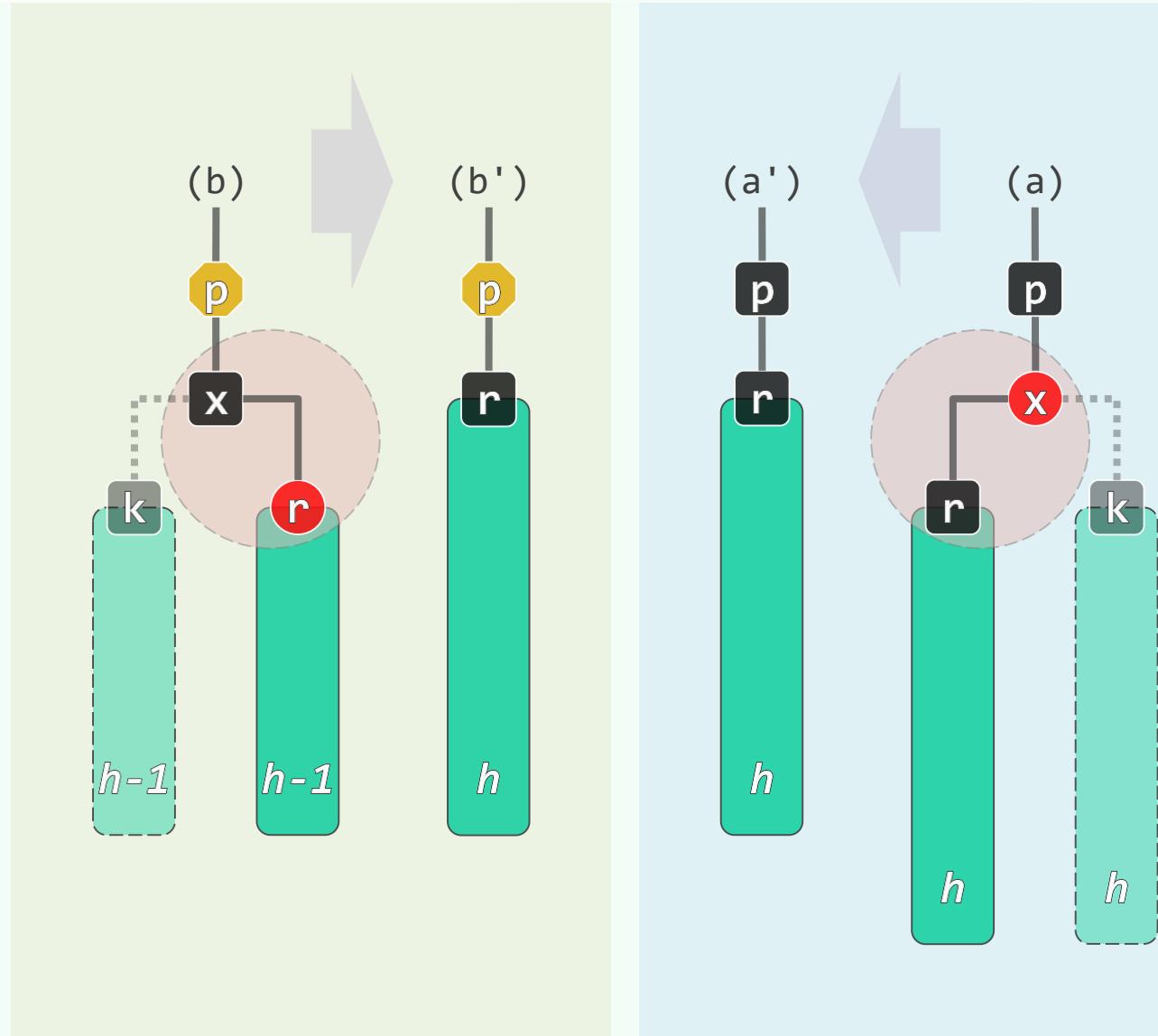
- 条件1、2依然满足
- 但条件3、4却不见得

❖ 在原树中，考查x与r

- 若x为红，则条件3、4自然满足
- 若r为红，则令其与x交换颜色

❖ 总之，无论x或r为红，则3、4均不难满足

——删除遂告完成！



双黑

❖ 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

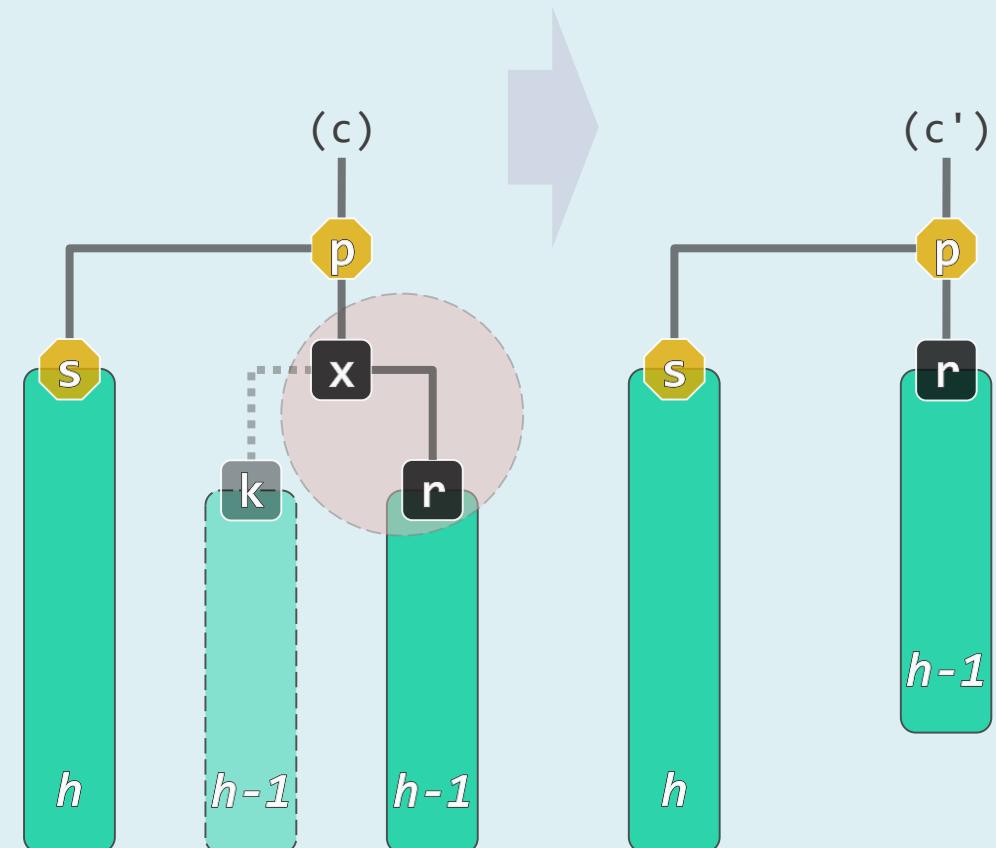
(稍后可见, 等效于B-树中x所属节点下溢)

❖ 在新树中, 考查r的父亲、兄弟

- `p = r->parent //亦是原x的父亲`

- `s = sibling(r)`

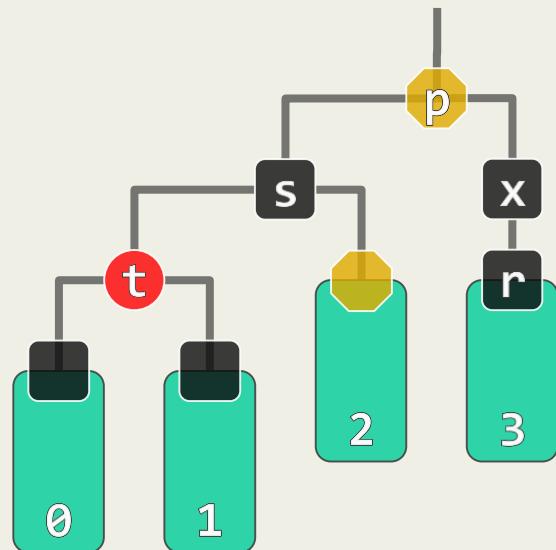
❖ 以下分四种情况处理...



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; //s的红孩子 (若左、右孩子皆红，左者优先；皆黑时为NULL)
        if ( IsRed( s->rC ) ) t = s->rC;
        if ( IsRed( s->lC ) ) t = s->lC;
        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 ~ a

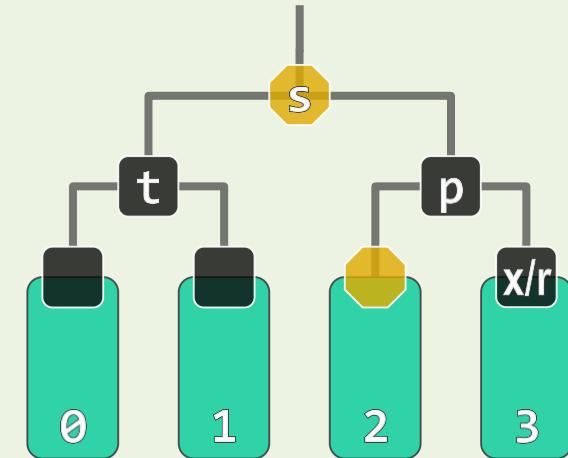
s ~ b

p ~ c

❖ r保持黑

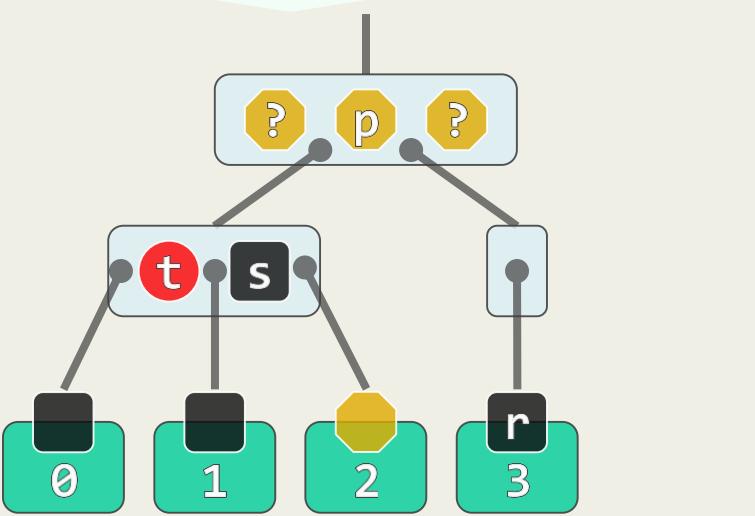
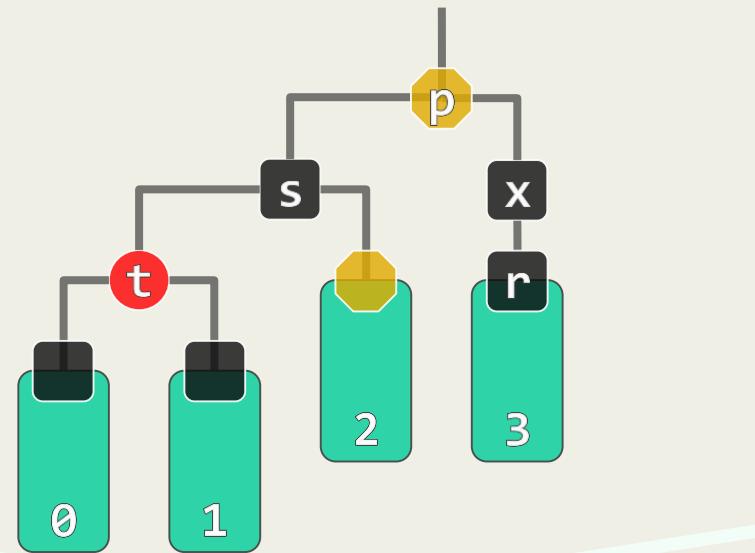
a、c染黑

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

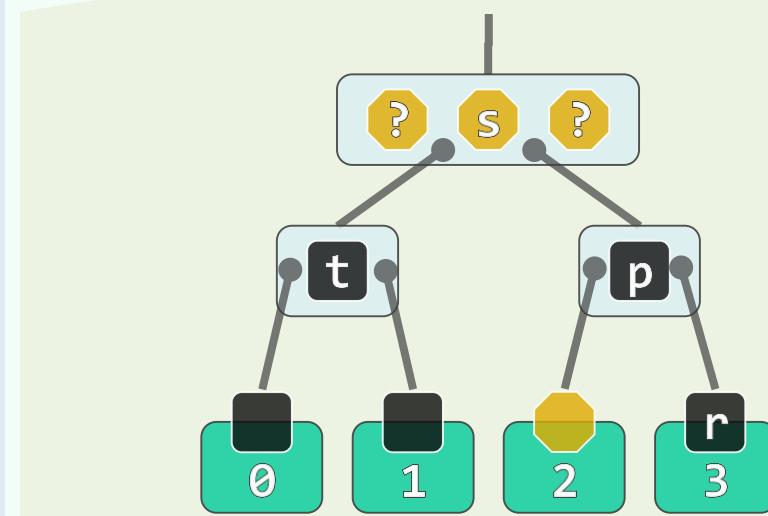
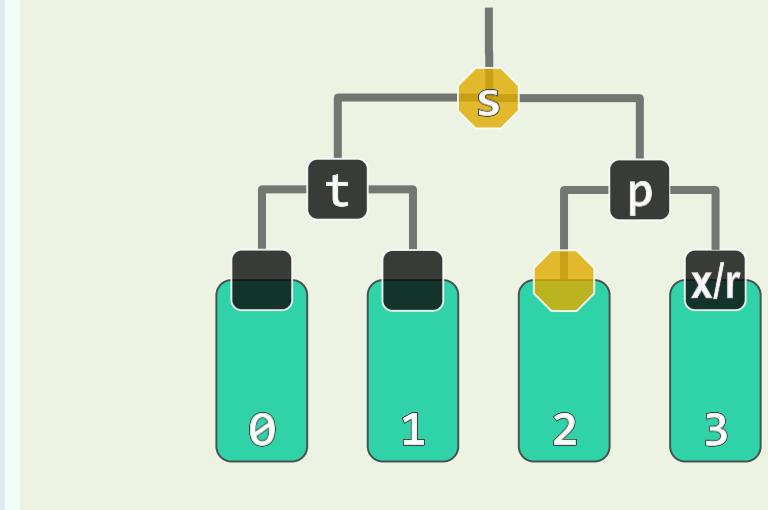
❖ 在对应的B-树中

- p 若为红

问号之一为黑关键码

- p 若为黑

必自成一个超级节点

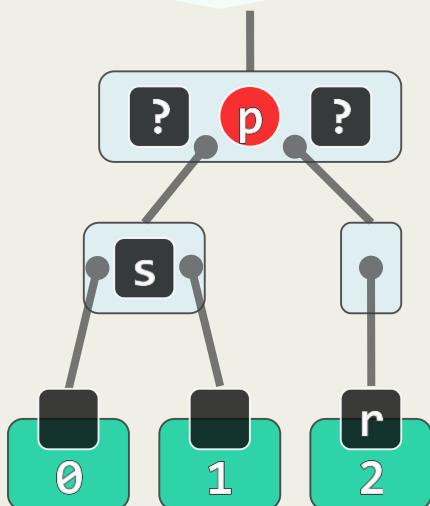
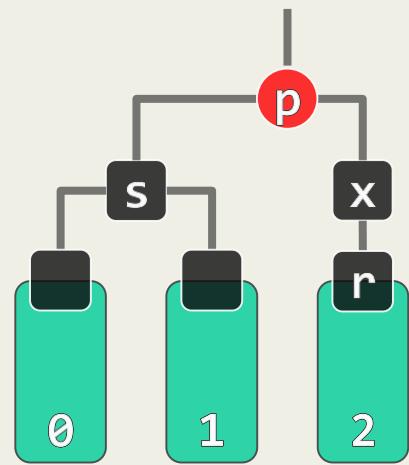


BB-1: 实现

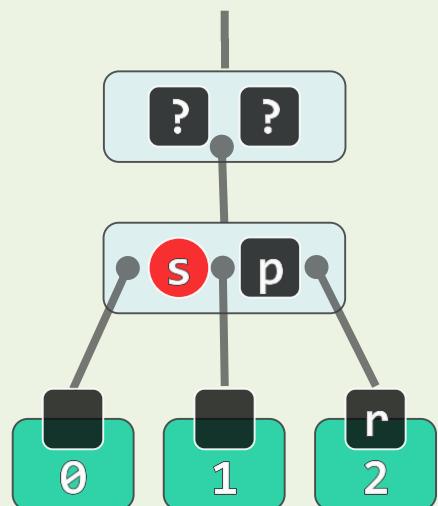
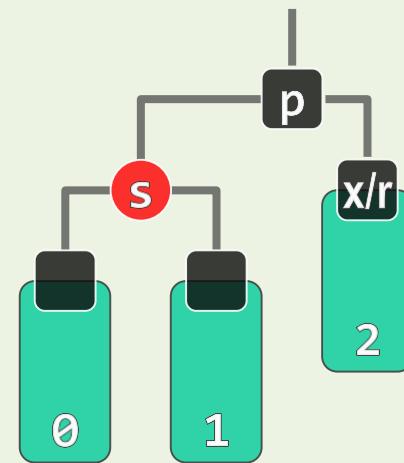
```
if ( IsBlack( s ) ) { //兄弟s为黑
    /* ..... */

    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->lC->color = RB_BLACK; updateHeight( b->lC ); }
        if ( HasRChild( *b ) ) { b->rC->color = RB_BLACK; 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 ... */ }
```

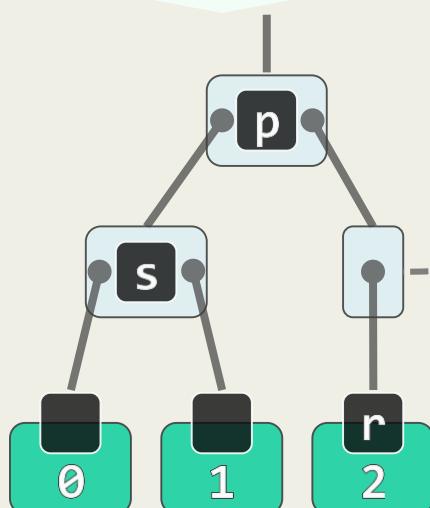
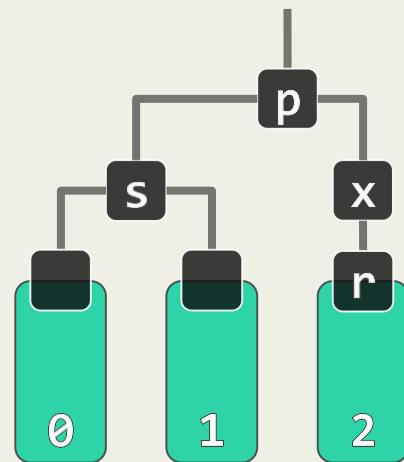
BB-2R: s为黑，且两个孩子均为黑；p为红



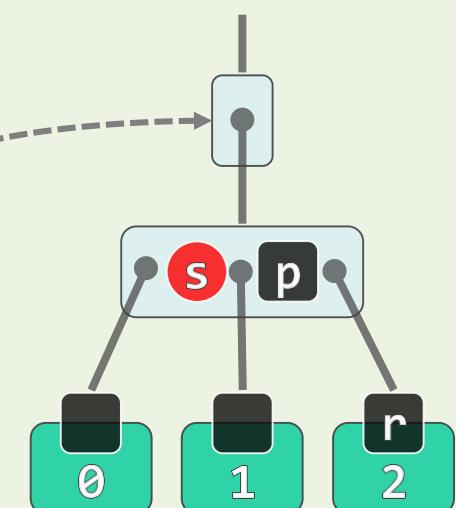
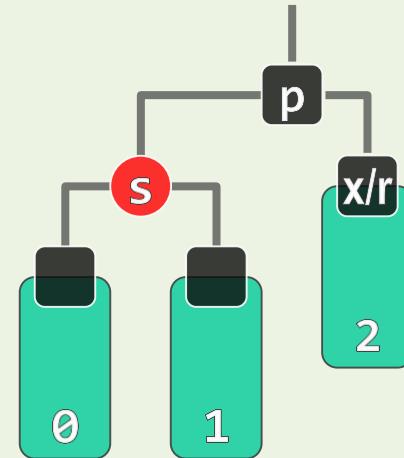
- ❖ r保持黑；s转红；p转黑
- ❖ 在对应的B-树中，等效于
下溢节点与兄弟合并
- ❖ 红黑树性质在全局得以恢复
- ❖ 失去关键码p后，上层节点
会否继而下溢？不会！
- ❖ 合并之前，在p之左或右侧
还应有一个黑关键码



BB-2B: s为黑，且两个孩子均为黑；p为黑



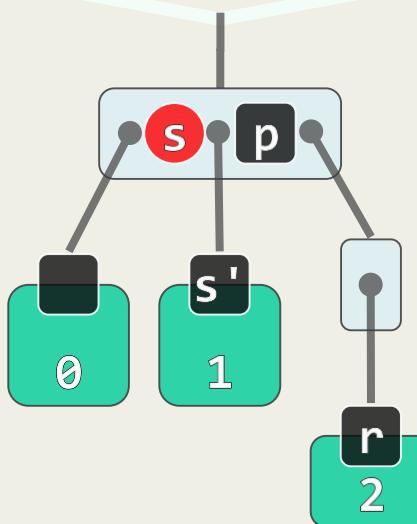
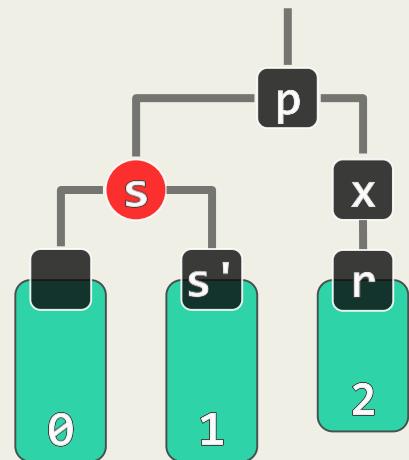
- ❖ s转红；r与p保持黑
- ❖ 红黑树性质在局部得以恢复
- ❖ 在对应的B-树中，等效于
下溢节点与兄弟合并
- ❖ 合并前，p和s均属于单关键码节点
孩子的下溢修复后，父节点继而下溢
- ❖ 好在可继续分情况处理
高度递增，至多 $\mathcal{O}(\log n)$ 层 (步)



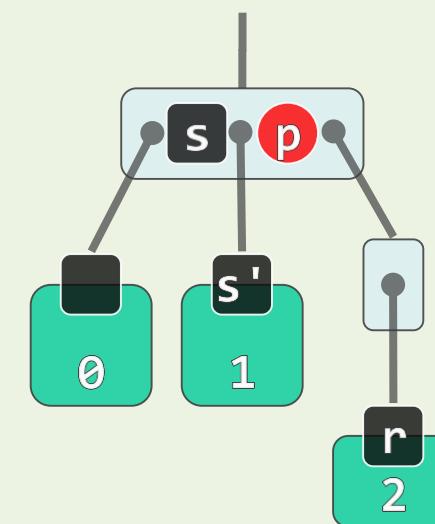
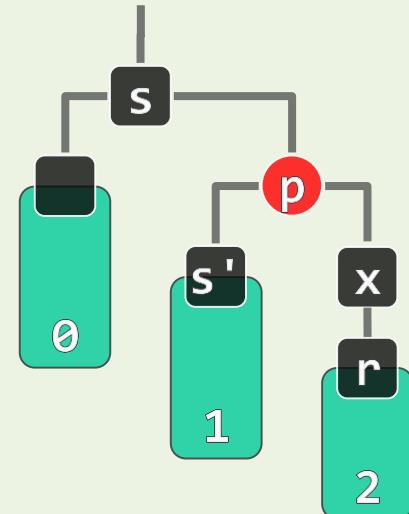
BB-(2R+2B): 实现

```
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)



- ❖ 绕p单旋; s红转黑, p黑转红
- ❖ 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BB-2B
 - 而只能是BB-2R或BB-1
- ❖ 于是, 再经一轮调整
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

❖ RedBlack<T>::remove()

仅需 $\mathcal{O}(\log n)$ 时间

- $\mathcal{O}(\log n)$ 次重染色
- $\mathcal{O}(1)$ 次旋转

	旋转	染色	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	必再次双黑, 但将上升一层
(3) 红s	1	2	转为(1)或(2R)

