

复杂度

❖ 时间

$$\mathcal{O}(n^2) + \mathcal{O}(n^3) = \mathcal{O}(n^3)$$

与n次调用Dijkstra相同

❖ 空间

存储一张 $n \times n$ 的表格， $\mathcal{O}(n^2)$

每个单元为两个整数

❖ 对于稀疏图和稠密图，你会分别选择哪种算法？

❖ 根据`midv[][]`矩阵，如何重构出u和v之间的最短路径？

为此，你需要多长时间？

7. 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.



邓俊辉

- Harry Potter and the Sorcerer's Stone

deng@tsinghua.edu.cn

查找

- ❖ 按照事先约定的规则，从数据集合中找出符合**特定条件**的对象
- ❖ 对于算法的构建而言，属于最为基本而重要的静态操作
- ❖ 很遗憾，基本的数据结构并不能**高效**地**兼顾**静态查找与动态修改

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	$\Theta(n)$	$\Theta(n)$

- ❖ 那么，能否**综合**现有方法的优点？如何做到？

循关键码访问

❖ 数据项之间，依照各自的**关键码**彼此区分

call-by-key

❖ 为此，关键码之间必须支持

- 大小**比较**与
- 相等**比对**

❖ 数据集合中的数据项

统一地表示和实现为词条**entry**形式



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) {}; //默认构造函数  
  
    Entry( Entry<K, V> const & e ) : key(e.key), value(e.value) {}; //克隆  
  
    // 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  
};
```

7. 二叉搜索树

概述

中序

邓俊辉

deng@tsinghua.edu.cn

顺序性

❖ 任一节点均不小于/不大于

其左/右后代

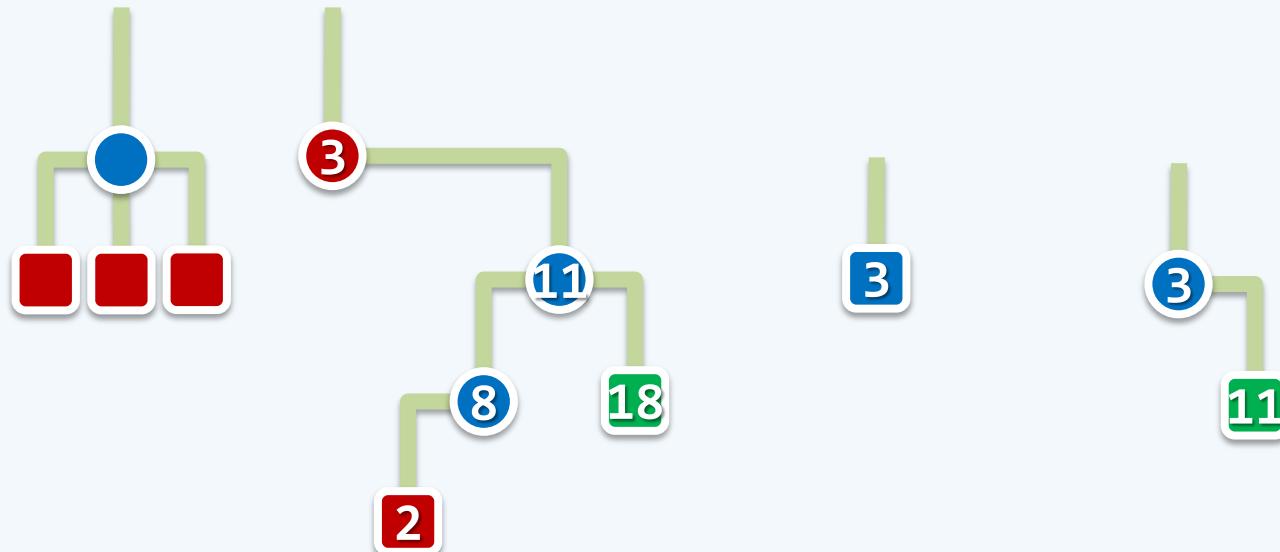
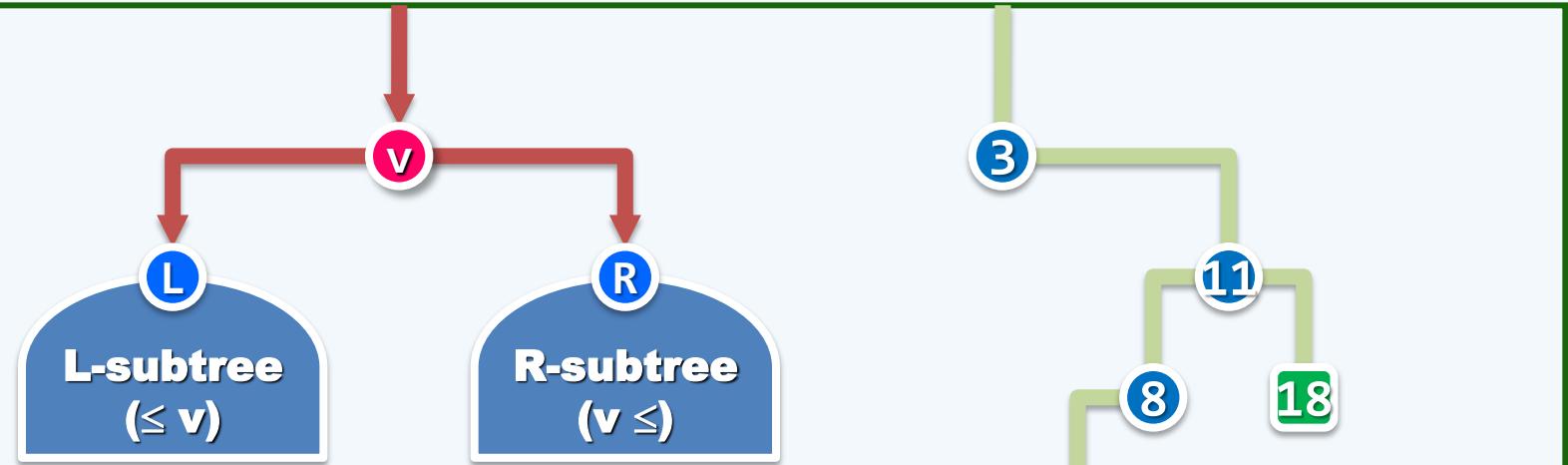
❖ 是否等效于...

❖ 任一节点均不小于/不大于

其左/右孩子

❖ Binary Search Tree

- 节点
- 词条
- 关键码



互异假定

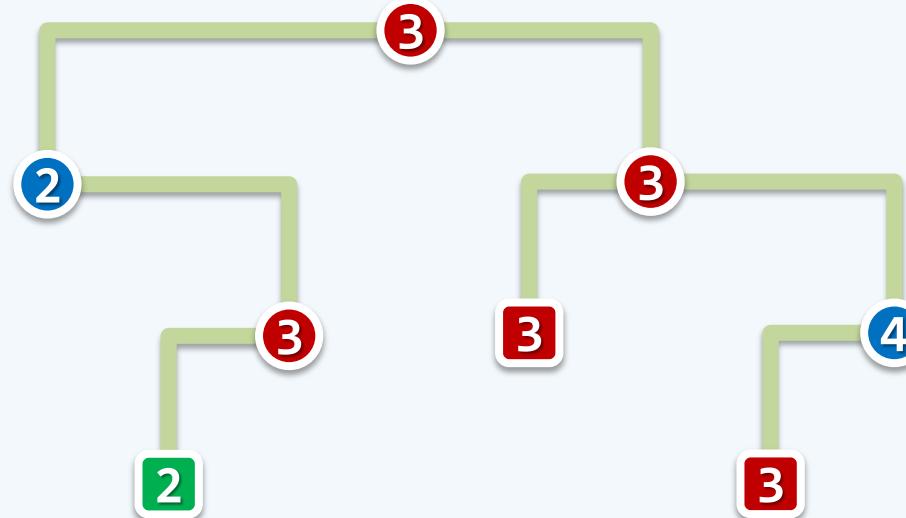
❖ 为简化起见，暂且禁止重复词条

❖ 这种简化

- 应用中不自然
- 算法上无必要

❖ 习题解析

- [7-10]
- [7-13]
- [8-3]



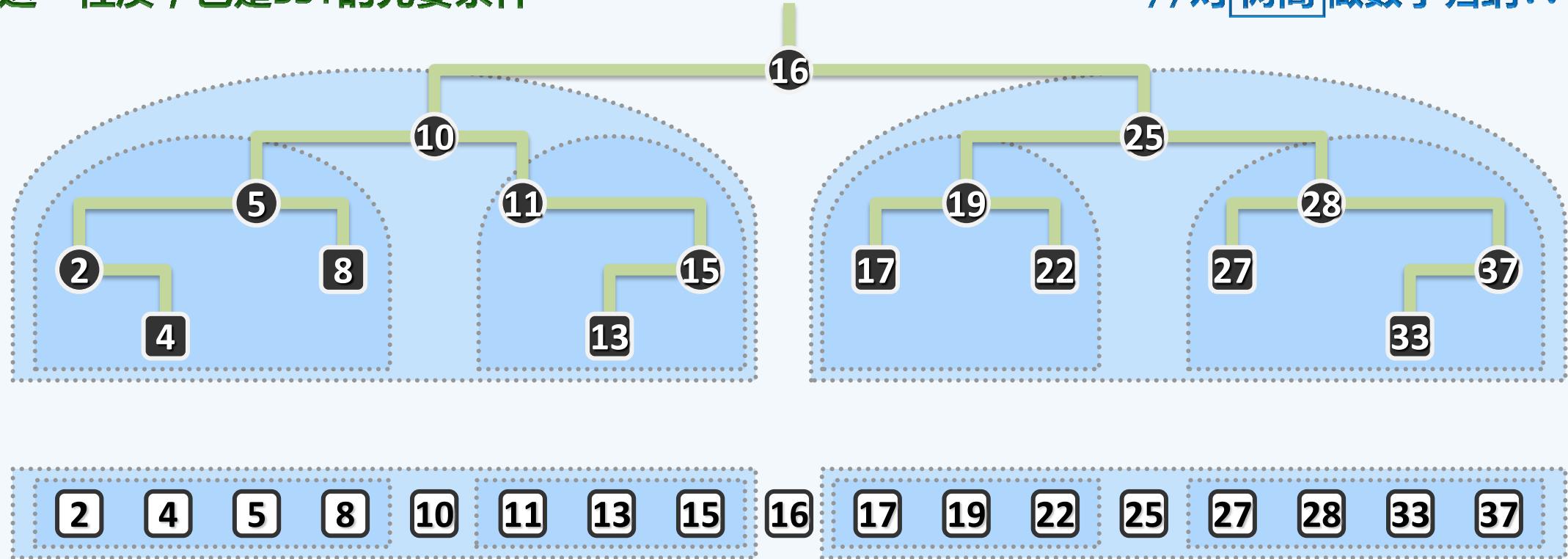
单调性

◆ 顺序性虽然只是对**局部**特征的刻画，但由此却可导出某种**全局**特征…

◆ 单调性：BST的**中序**遍历序列，必然**单调**非降

◆ 这一性质，也是BST的充要条件

//对**树高**做数学归纳…



7. 二叉搜索树

概述

接口

邓俊辉

deng@tsinghua.edu.cn

对外接口

```
❖ template <typename T> class BST : public BinTree<T> { //由BinTree派生  
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  
        BinNodePosi(T), BinNodePosi(T), BinNodePosi(T),  
        BinNodePosi(T), BinNodePosi(T), BinNodePosi(T), BinNodePosi(T));  
    BinNodePosi(T) rotateAt( BinNodePosi(T) ); //旋转调整  
};
```

7. 二叉搜索树

算法及实现

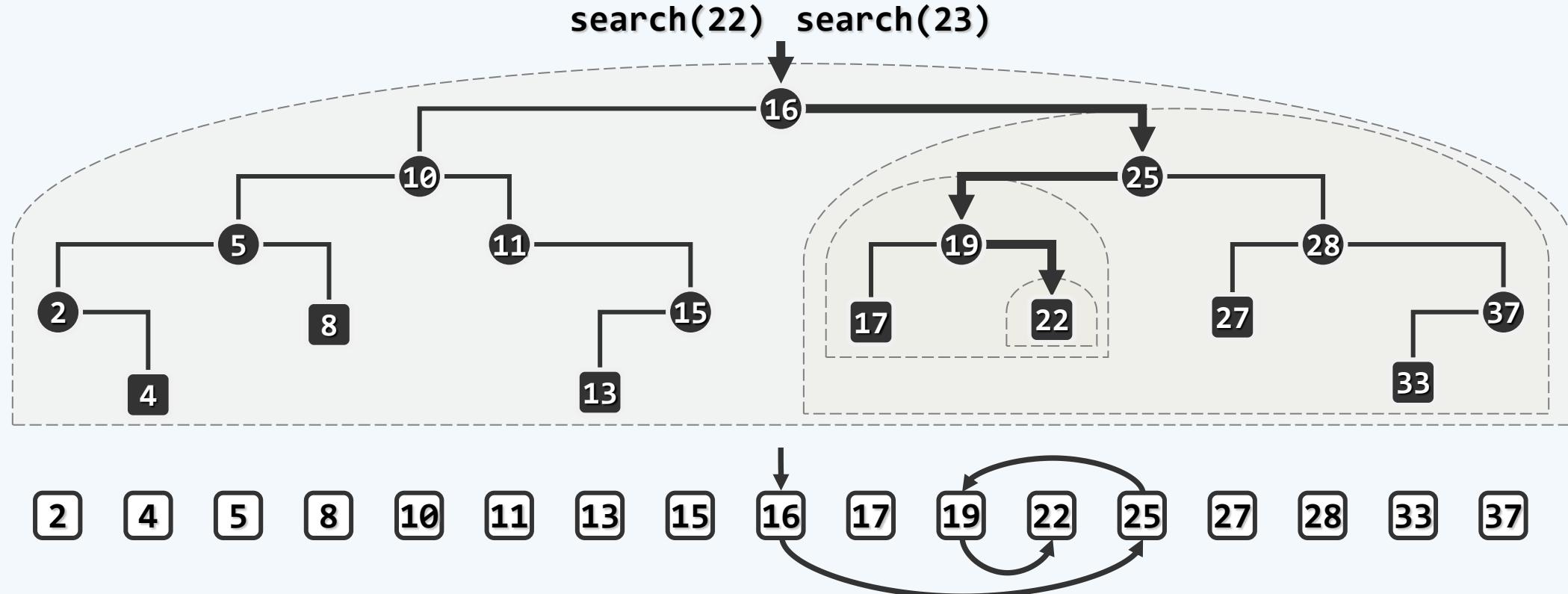
查找

邓俊辉

deng@tsinghua.edu.cn

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），或查找范围缩小至空树（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)
{ return searchIn( _root, e, _hot = NULL ); } //从根节点启动查找

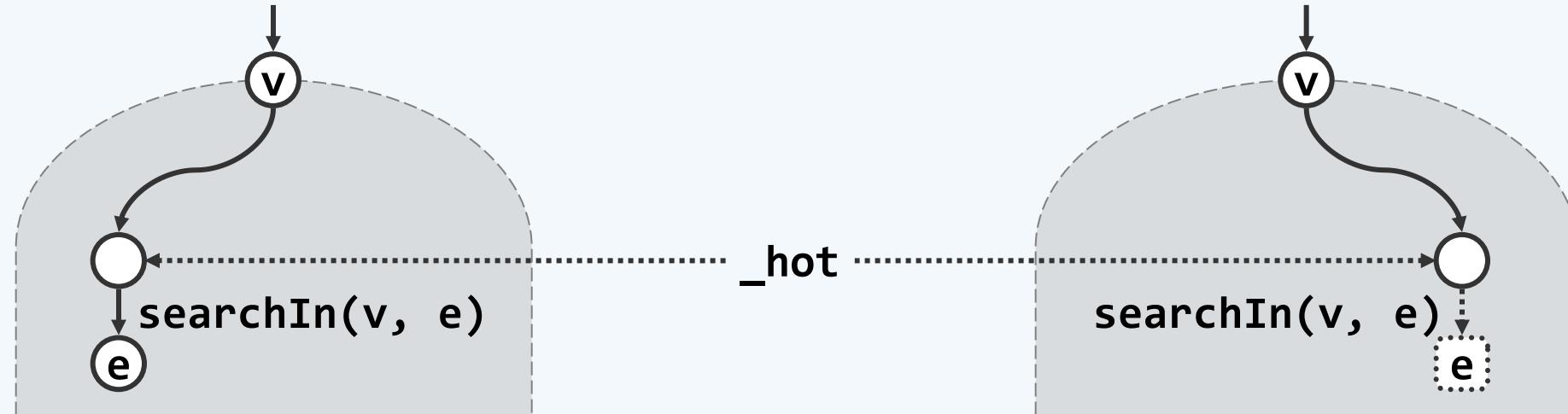
❖ static BinNodePosi(T) & searchIn( //典型的尾递归，可改为迭代版
    BinNodePosi(T) & v, const T & e, BinNodePosi(T) & hot) {
    // 当前(子)树根、目标关键码、记忆热点
    if ( !v || ( e == v->data ) ) return v; //足以确定失败、成功，或者
    hot = v; //先记下当前(非空)节点，然后再...
    return searchIn( ( e < v->data ? v->lcv : v->rc ), e, hot );
} //运行时间正比于返回节点v的深度，不超过树高 $\Theta(h)$ 

```

接口语义

❖ 返回的引用值：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**



❖ 失败时，不妨假想地将此空节点，转换为一个数值为e的**哨兵**节点

如此，依然满足BST的充要条件；而且更重要地...

❖ 无论成功与否：返回值总是**等效地**指向**命中**节点，而**_hot**总是指向**命中**节点的**父亲**

7. 二叉搜索树

算法及实现

插入

邓俊辉

deng@tsinghua.edu.cn

算法

❖ 先借助 `search(e)` 确定插入位置及方向

再将新节点作为 **叶子** 插入

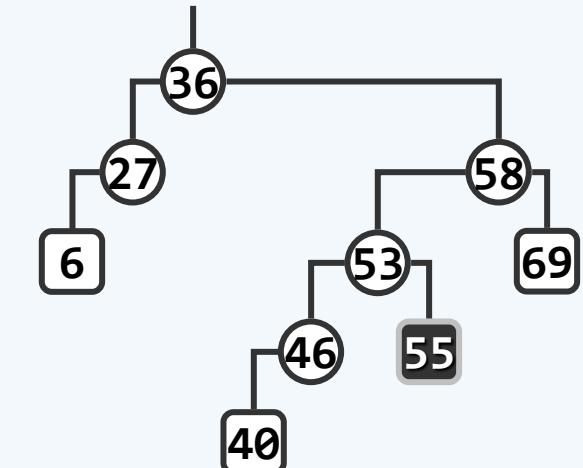
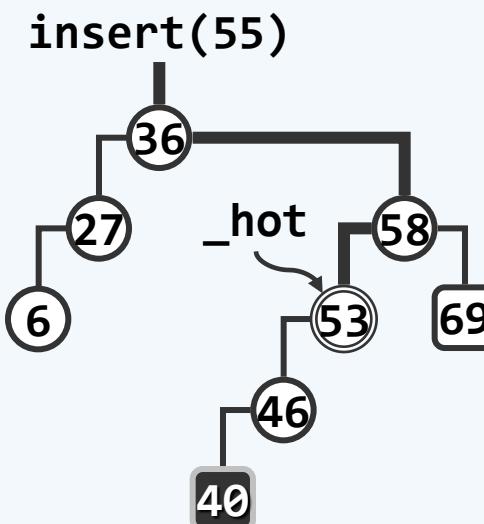
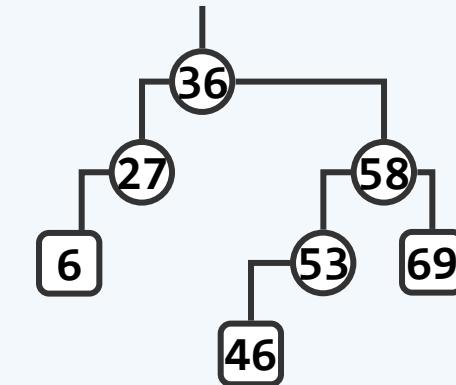
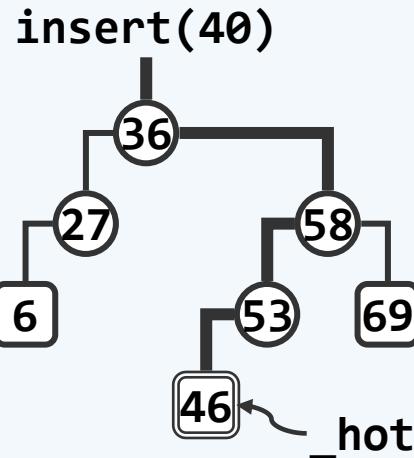
❖ 若 `e` 尚不存在，则

`_hot` 为新节点的 **父亲**

`v = 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



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)
均线性正比于返回节点x的深度，不超过树高 $O(h)$

7. 二叉搜索树

算法及实现

删除

邓俊辉

deng@tsinghua.edu.cn

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 ); //分两大类情况实施删除，更新全树规模
    _size--; //更新全树规模
    updateHeightAbove( _hot ); //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

```

❖ 时间主要消耗于 search()、updateHeightAbove()
 还有 removeAt() 中可能调用的 succ()
 累计 $\mathcal{O}(h)$

单分支

❖ 若 $*x(69)$ 的某一子树为空

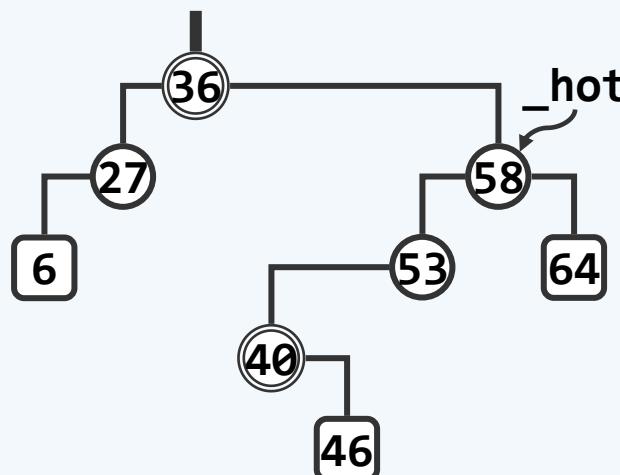
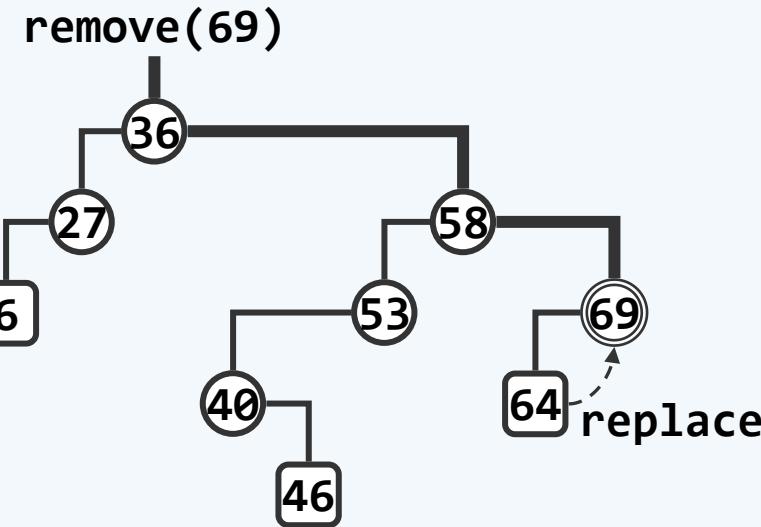
则可将其替换为另一子树(64) //可能亦为空

❖ 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

```

❖ 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->rChild; //左子树为空
    else if ( !HasRChild( *x ) ) succ = x = x->lChild; //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 ); //释放被摘除节点
    return succ; //返回接替者
} //此类情况仅需 $O(1)$ 时间

```

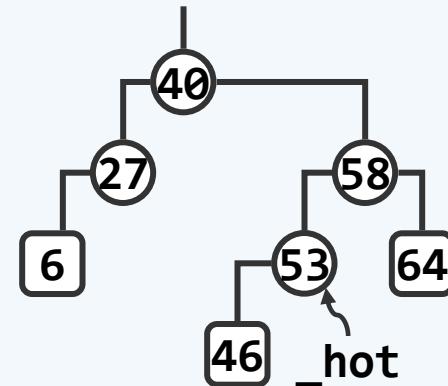
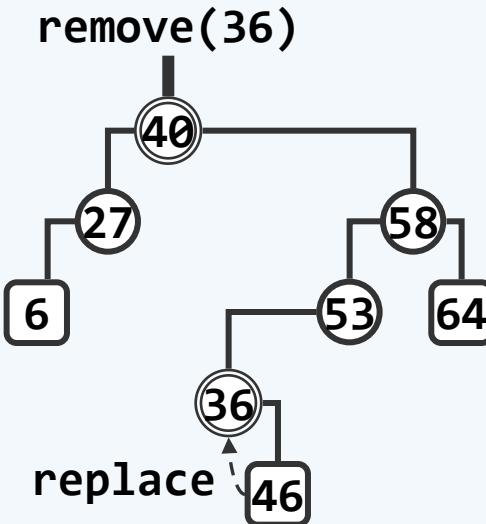
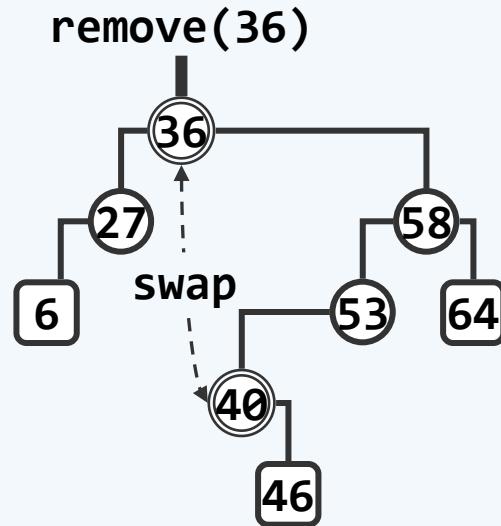
双分支

❖ 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

❖ 尽管顺序性曾在中途一度不合，但最终必将重新恢复



双分支

```

❖ 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 $O(h)$

7. 二叉搜索树

平衡

期望树高

邓俊辉

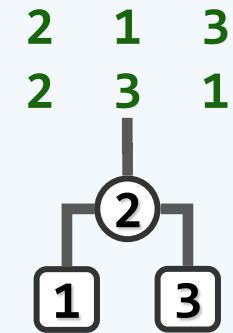
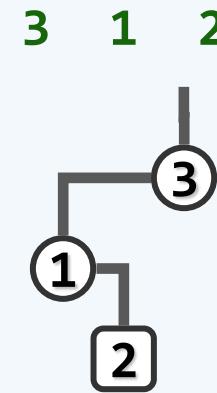
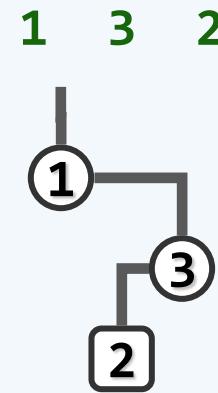
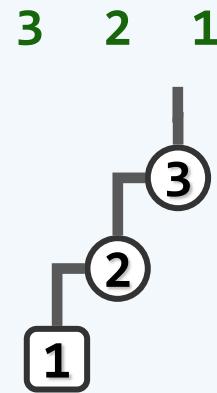
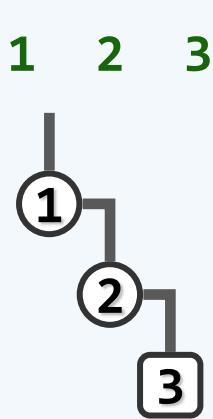
deng@tsinghua.edu.cn

树高

- ❖ 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 $\mathcal{O}(h)$
- ❖ 因此，若不能有效地控制树高，则就实际的性能而言较之此前的向量和列表等数据结构，BST将无法体现出明显优势
- ❖ 比如在最坏情况下，二叉搜索树可能彻底地退化为列表此时的查找效率甚至会降至 $\mathcal{O}(n)$ ，线性正比于树（列表）的规模
- ❖ 那么，出现此类最坏或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呢？
- ❖ 以下按两种常用的随机统计口径，就BST的平均性能做一分析和对比

随机生成

- 考查 n 个互异词条 $\{e_1, e_2, \dots, e_n\}$ ，对任一排列 $\sigma = (e_{i1}, e_{i2}, \dots, e_{in}) \dots$
- 从空树开始，反复调用 `insert()` 接口将各词条 **依次插入**，得到 $T(\sigma)$
- 与 σ 相对应的 $T(\sigma)$ ，称由 σ **随机生成** (randomly generated)

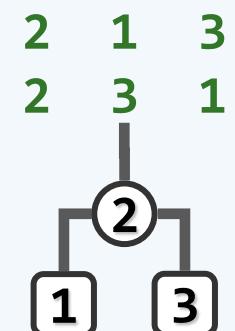
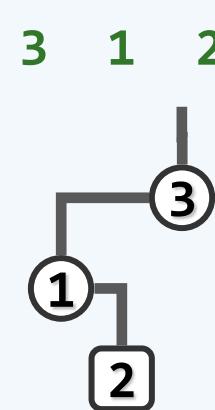
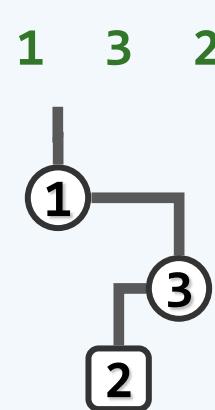
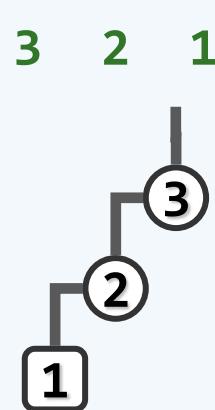
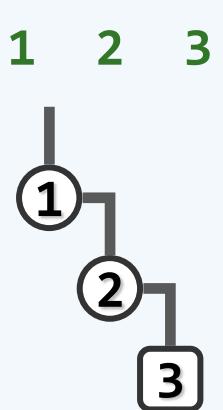


若假定任一排列 σ 作为输入的概率均等 $1/n!$

则由 n 个互异词条随机生成的二叉搜索树，平均高度为 $\Theta(\log n)$

随机组成

- ◆ n 个互异节点，在遵守顺序性的前提下，可随机确定拓扑联接关系
- ◆ 如此所得的BST，称由这组节点 **随机组成** (randomly composed)



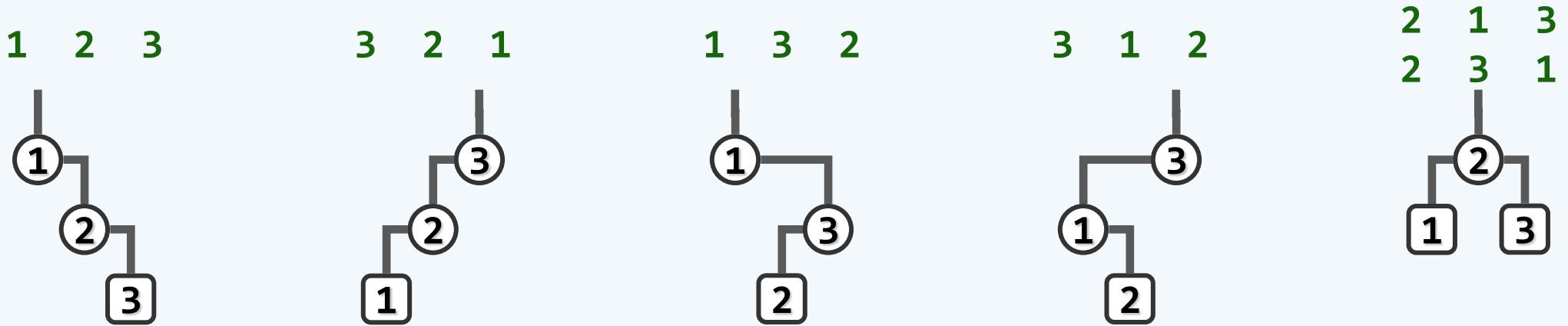
- ◆ 由 n 个互异节点随机组成的BST，若共计 $T(n)$ 棵，则有

$$T(n) = \sum_{k=1}^n SP(k-1) \cdot SP(n-k) = \text{Catalan}(n) = (2n)! / (n+1)! / n!$$

- ◆ 假定所有BST等概率出现，则其平均高度为 $\Theta(\sqrt{n})$

$\Theta(\log n)$ vs. $\Theta(\sqrt{n})$

❖ 按两种口径所估计的平均性能，差异极大——谁更可信？谁更接近于真实情况？



- ❖ 前一口径中，越低的BST被**重复统计**更多次——故嫌过于**乐观**
- ❖ 若删除算法固定使用succ()，则每棵BST都有越来越**左倾**的趋势
- ❖ 理想随机在实际中并不常见，关键码往往按**单调**甚至**线性**的次序出现
极高的的BST频繁出现，不足为怪

7. 二叉搜索树

平衡

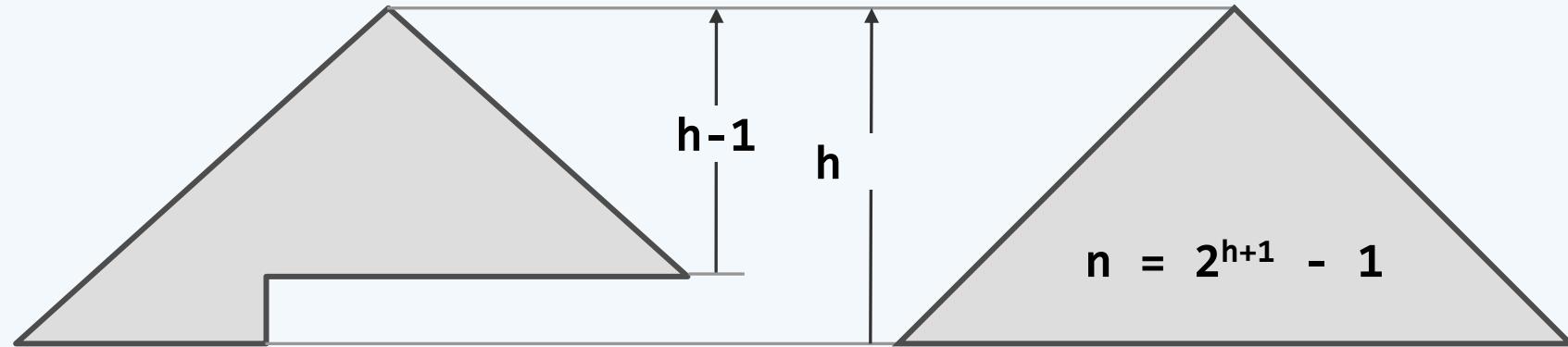
理想与适度

邓俊辉

deng@tsinghua.edu.cn

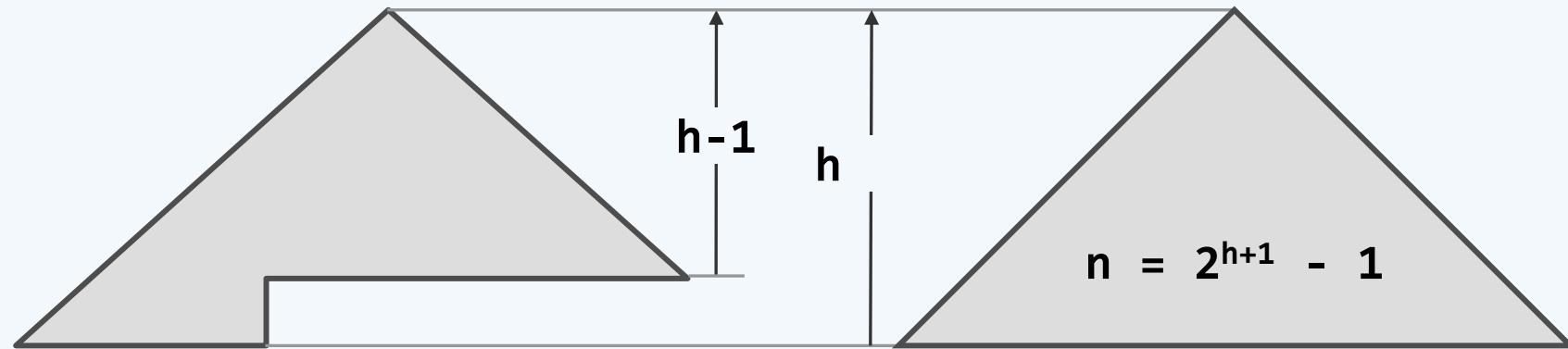
理想平衡

- ❖ 节点数目固定时，兄弟子树高度越接近（平衡），全树也将倾向于更低
- ❖ 由 n 个节点组成的二叉树，高度不低于 $\lfloor \log_2 n \rfloor$ ——恰为 $\lfloor \log_2 n \rfloor$ 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



适度平衡

- ❖ 理想平衡出现概率极低、维护成本过高，故须适当地放松标准
- ❖ 退一步海阔天空：高度渐进地不超过 $\mathcal{O}(\log n)$ ，即可称作适度平衡
- ❖ 适度平衡的BST，称作平衡二叉搜索树（BBST）



7. 二叉搜索树

平衡

等价变换

邓俊辉

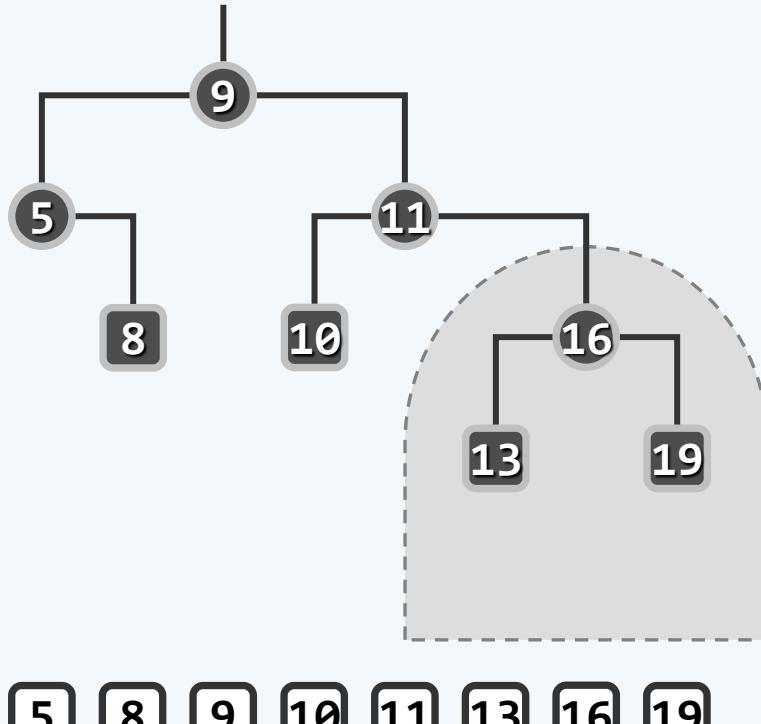
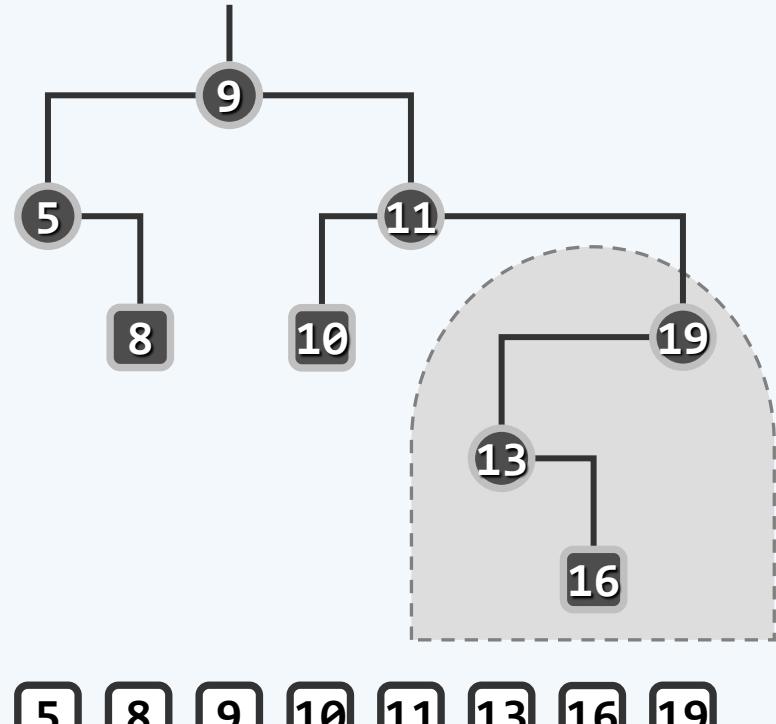
言与行其可迹兮，情与貌其不变

deng@tsinghua.edu.cn

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

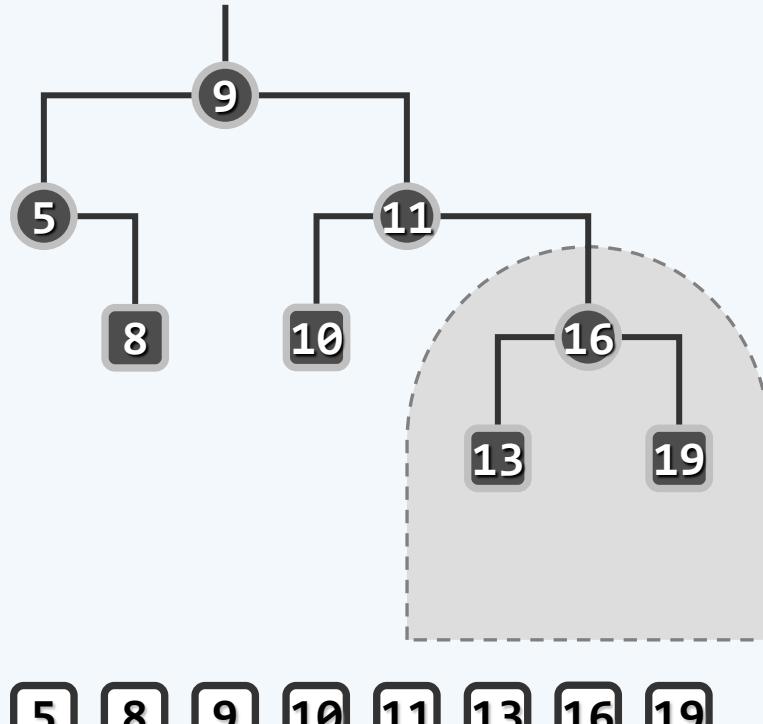
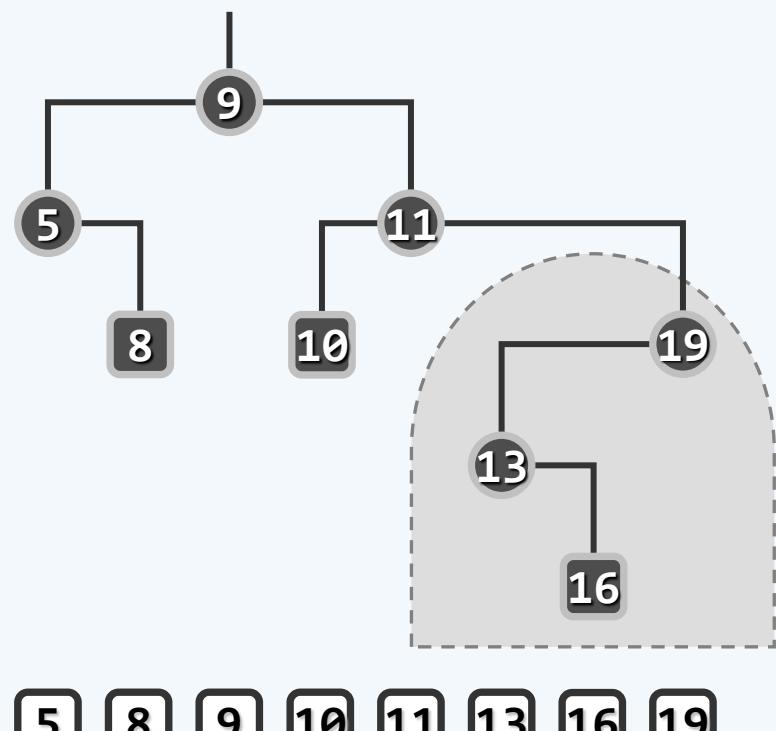


限制条件 + 局部性

❖ 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**

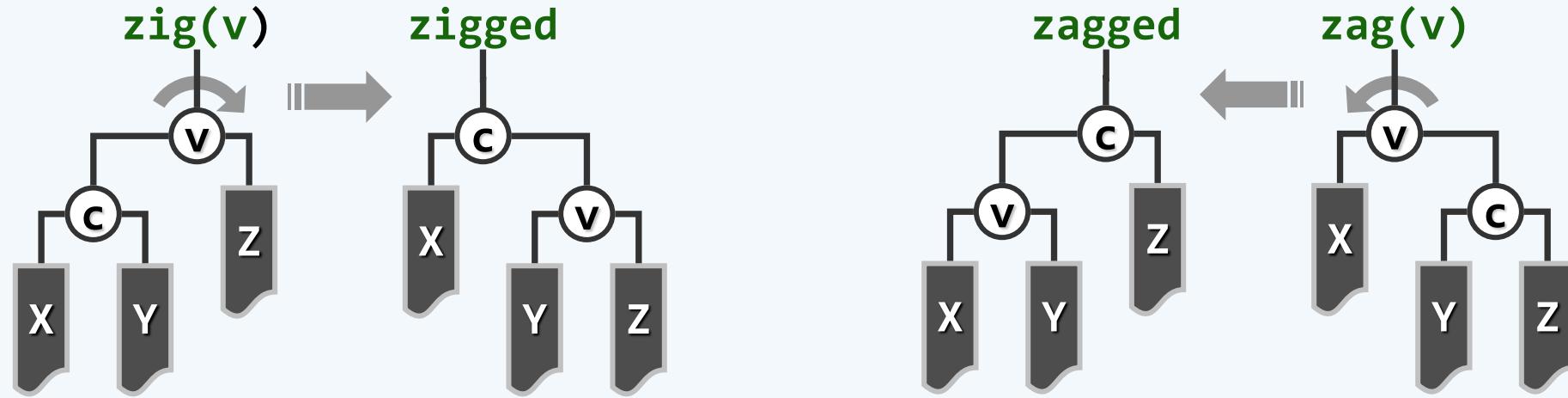
1) 单次动态修改操作后，至多 $O(\log n)$ 处局部不再满足限制条件

2) 可在 $O(\log n)$ 时间内，使这些局部（以至全树）重新满足



等价变换 + 旋转调整

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



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

7. 二叉搜索树

AVL树

适度平衡

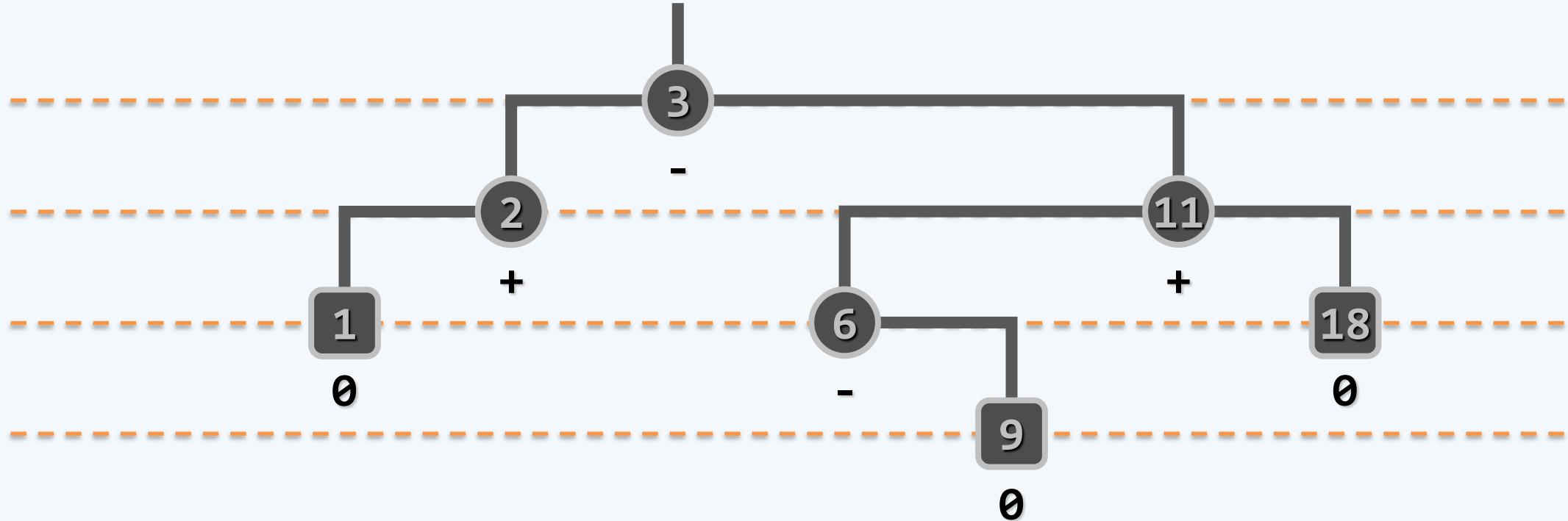
邓俊辉

deng@tsinghua.edu.cn

平衡因子

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

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



❖ AVL树未必**理想平衡**，必然**适度平衡**...

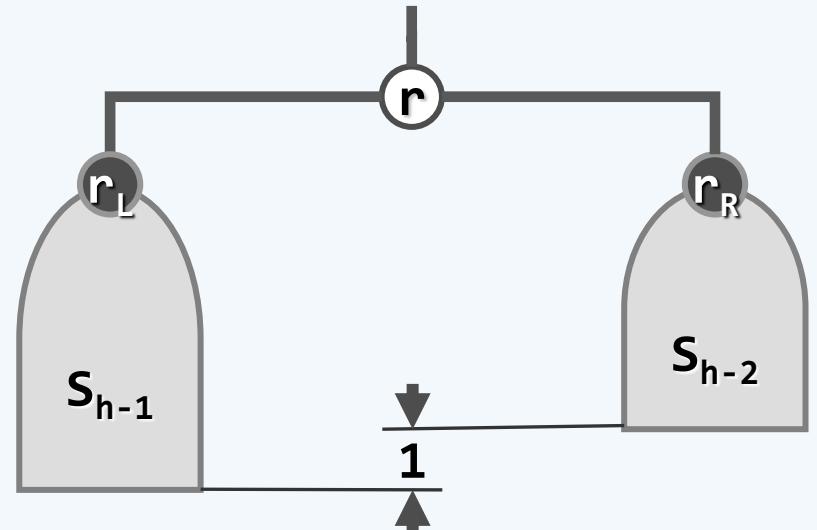
AVL = 适度平衡

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

❖ $S(h) = 1 + S(h - 1) + S(h - 2)$

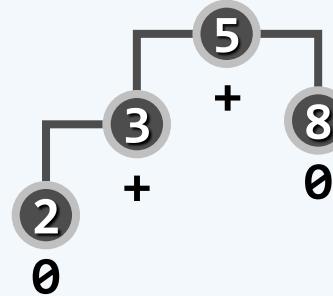
❖ $S(h) + 1 = [S(h - 1) + 1] + [S(h - 2) + 1]$

❖ $\text{fib}(h + 3) = \text{fib}(h + 2) + \text{fib}(h + 1)$

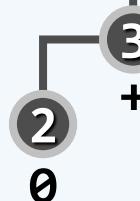


❖ 反过来，由 n 个节点构成的AVL树，高度至多为 $\mathcal{O}(\log n)$

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



$$\begin{aligned} S(1) + 1 &= 3 \\ &= \text{fib}(4) \end{aligned}$$



$$\begin{aligned} S(0) + 1 &= 2 \\ &= \text{fib}(3) \end{aligned}$$



7. 二叉搜索树

AVL树

重平衡

邓俊辉

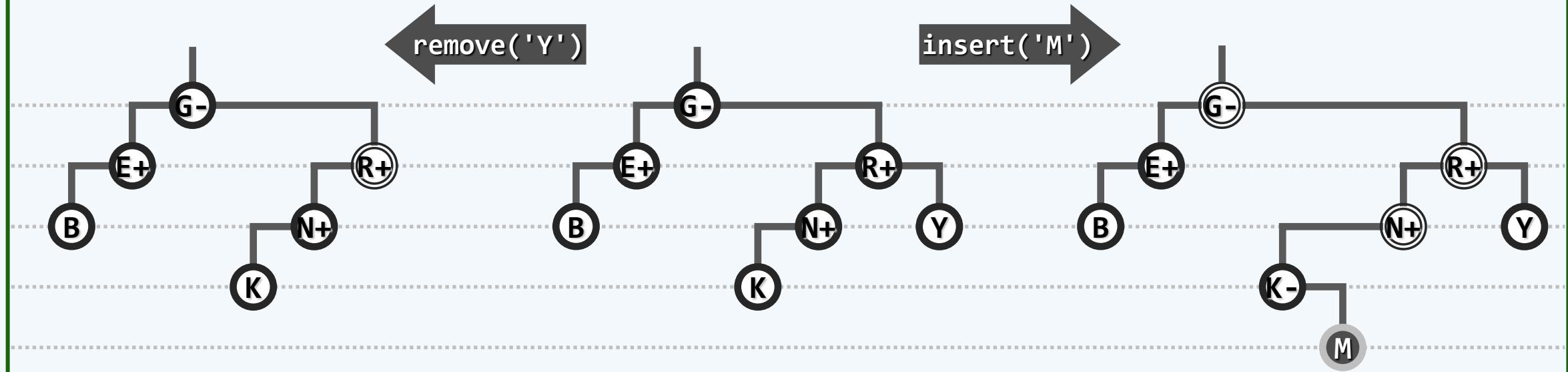
deng@tsinghua.edu.cn

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  
};
```

失衡与重平衡

- 按BST规则插入或删除节点之后，AVL平衡性可能破坏——如何恢复？



- 蛮力不足取，须借助等价变换

局部性：所有的旋转都在局部进行 // 每次只需 $O(1)$ 时间

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

7. 二叉搜索树

AVL树

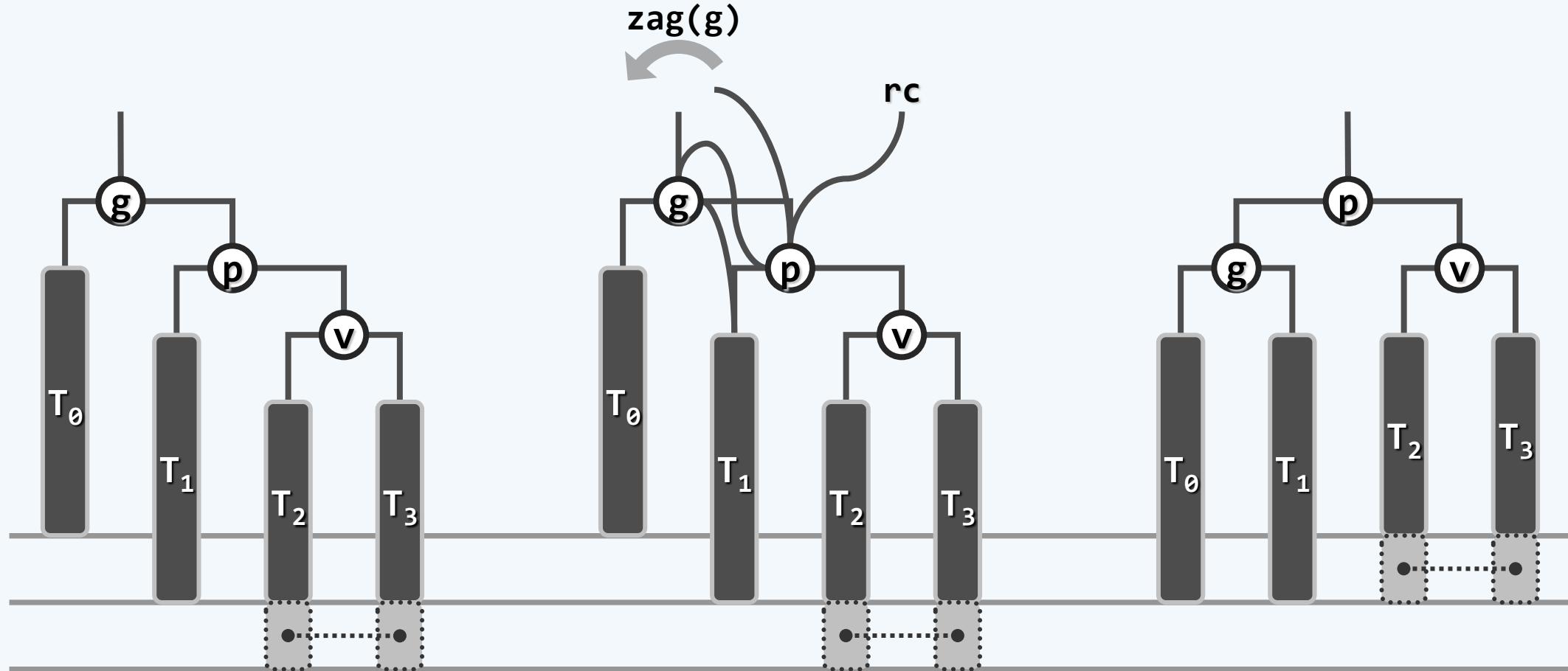
插入

邓俊辉

deng@tsinghua.edu.cn

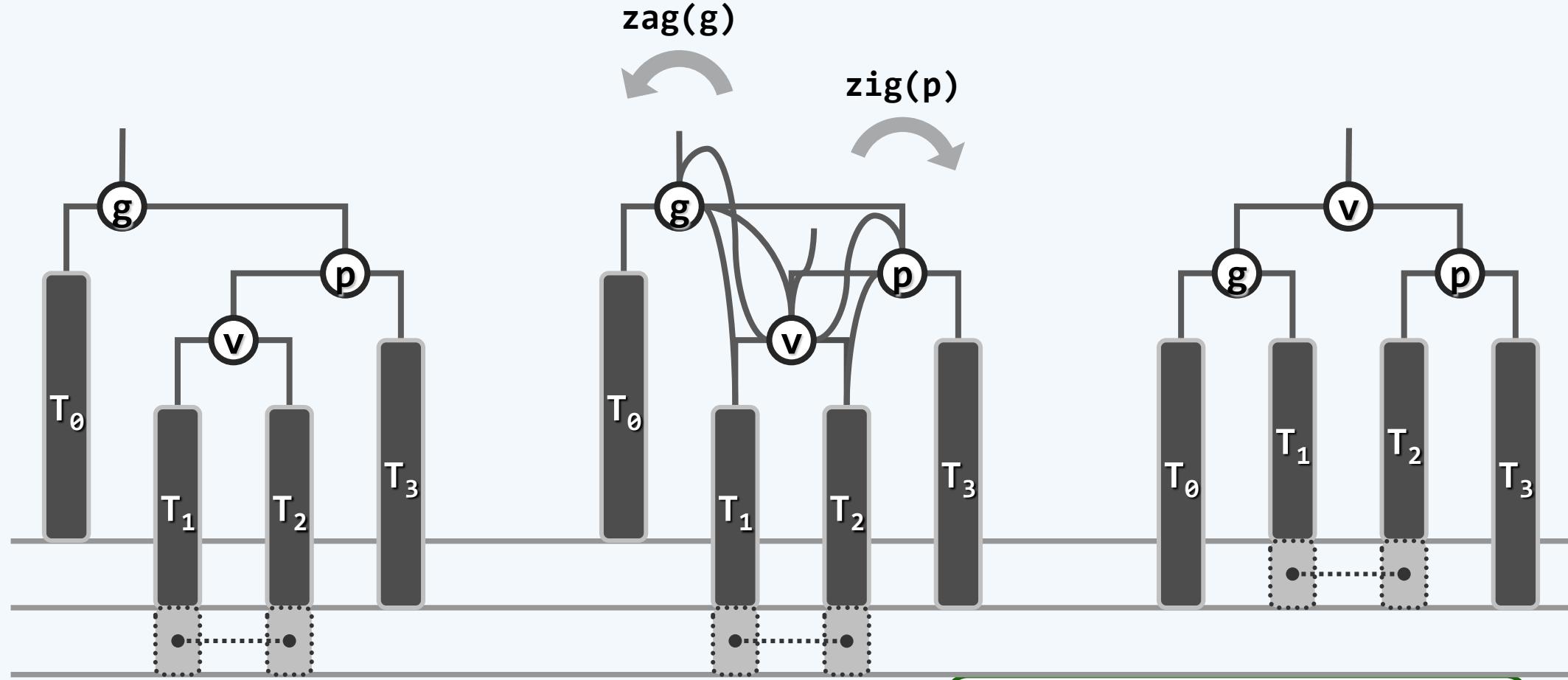
单旋

- 同时可有多个失衡节点，最低者g不低于x祖父
- g经单旋调整后复衡，子树高度复原；更高祖先也必平衡，全树复衡



双旋

- 同时可有多个失衡节点，最低者g不低于x祖父
- g经双旋调整后复衡，子树高度复原；更高祖先也必平衡，全树复衡



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。故以下从_hot起，向上逐层检查各代祖先
    for ( BinNodePosi(T) g = _hot; g; g = g->parent )
        if ( ! AvlBalanced( *g ) ) { //一旦发现g失衡，则通过调整恢复平衡
            FromParentTo( *g ) = rotateAt( tallerChild( tallerChild( g ) ) );
            break; //g复衡后，局部子树高度必然复原；其祖先亦必如此，故调整结束
        } else //否则（在依然平衡的祖先处），只需简单地
            updateHeight( g ); //更新其高度（平衡性虽不变，高度却可能改变）
    return xx; //返回新节点：至多只需一次调整
}

```

7. 二叉搜索树

AVL树

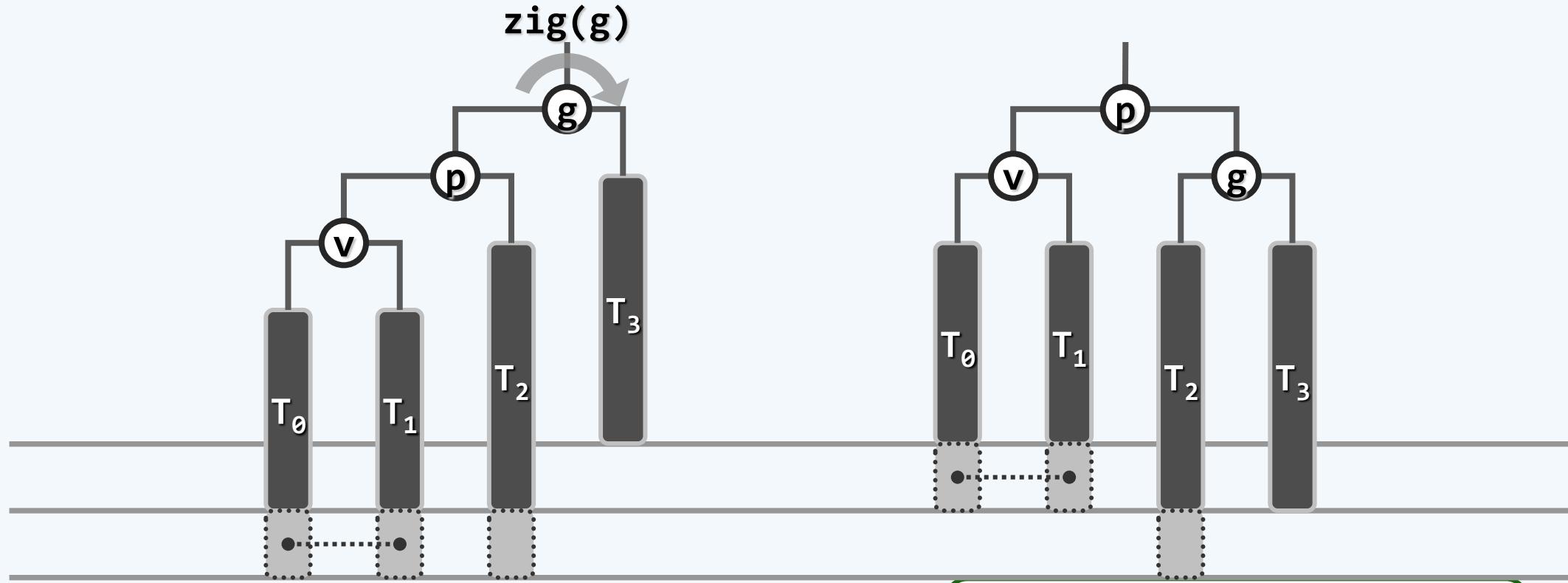
删除

邓俊辉

deng@tsinghua.edu.cn

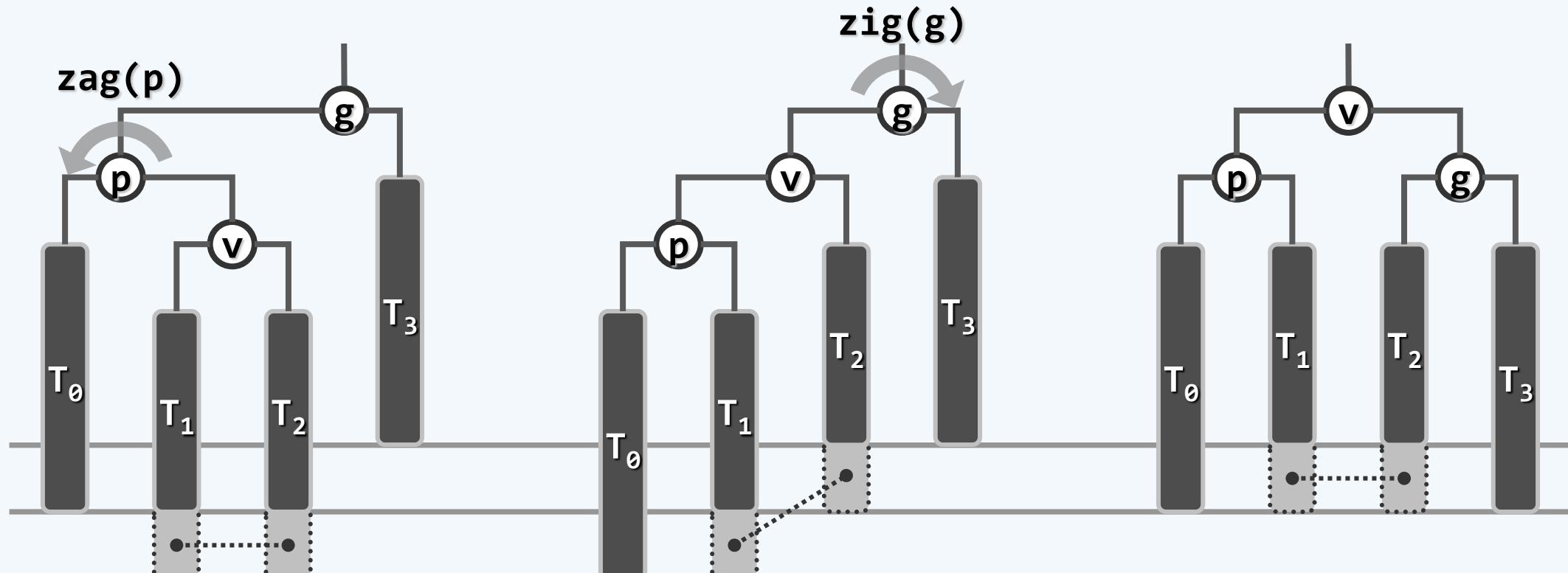
单旋

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



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 ); //并更新其高度
    } //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
}

```

7. 二叉搜索树

AVL树

(3+4)-重构

邓俊辉

deng@tsinghua.edu.cn

算法

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

按中序遍历次序，将其重命名为： $a < b < c$

它们总共拥有互不相交的四棵（可能为空的）子树

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

将原先以 g 为根的子树 s ，替换为一棵新子树 s'

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

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

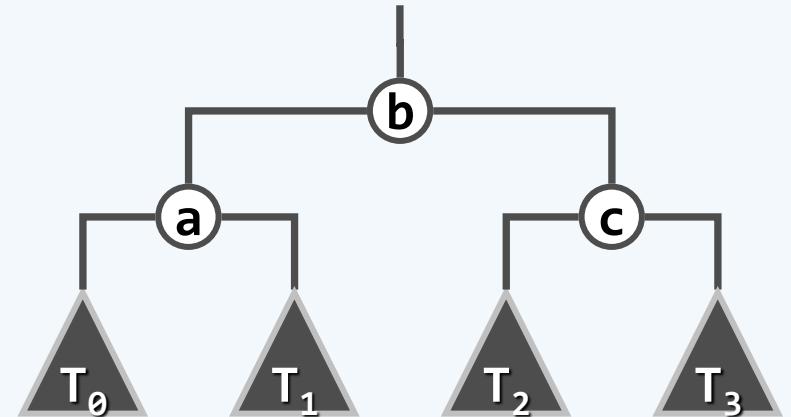
$$\text{rc}(b) = c$$

$$\text{lT}(a) = T_0$$

$$\text{rT}(a) = T_1$$

$$\text{lT}(c) = T_2$$

$$\text{rT}(c) = T_3$$



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

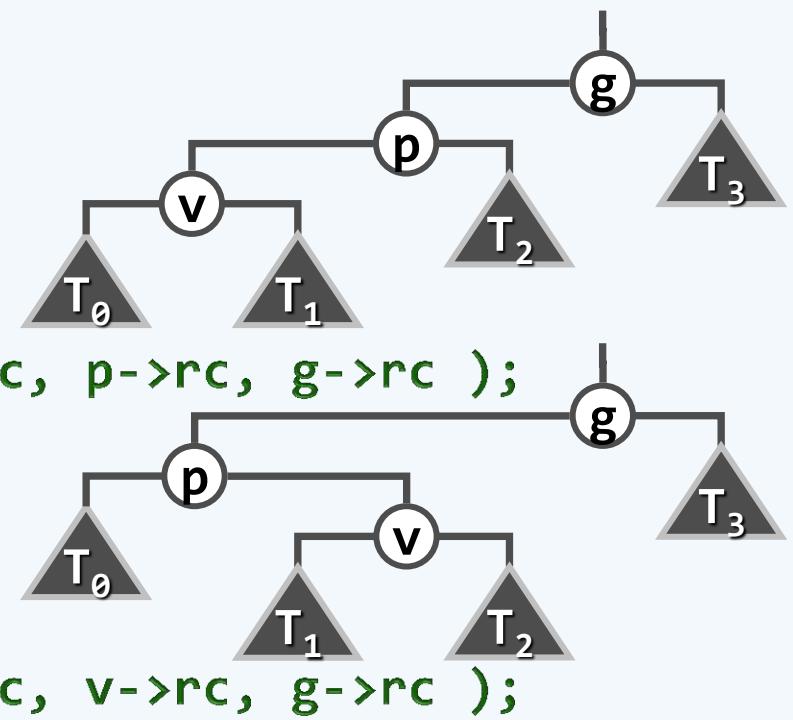
```
❖ template <typename T> BinNodePosi(T) BST<T>::connect34(  
    BinNodePosi(T) a, BinNodePosi(T) b, BinNodePosi(T) c,  
    BinNodePosi(T) T0, BinNodePosi(T) T1, BinNodePosi(T) T2, BinNodePosi(T) T3)  
{  
    a->lC = T0; if (T0) T0->parent = a;  
    a->rC = T1; if (T1) T1->parent = a; updateHeight(a);  
    c->lC = T2; if (T2) T2->parent = c;  
    c->rC = T3; if (T3) T3->parent = c; updateHeight(c);  
    b->lC = a; a->parent = b;  
    b->rC = c; c->parent = b; updateHeight(b);  
    return b; //该子树新的根节点  
}
```

统一调整

```

❖ template<typename T> BinNodePosi(T) BST<T>::rotateAt( BinNodePosi(T) v ) {
    BinNodePosi(T) p = v->parent, g = p->parent; //父亲、祖父
    if ( IsLChild( * p ) ) //zig
        if ( IsLChild( * v ) ) { //zig-zig
            p->parent = g->parent; //向上联接
            return connect34( v, p, g, v->lc, v->rc, p->rc, g->rc );
        } else { //zig-zag
            v->parent = g->parent; //向上联接
            return connect34( p, v, g, p->lc, v->lc, v->rc, g->rc );
        }
    else { /*.. zig-zig & zig-zag ..*/ }
}

```



综合评价

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

$O(n)$ 的存储空间

❖ 缺点 借助高度或平衡因子，为此需改造元素结构，或额外封装

实测复杂度与理论值尚有差距

插入/删除后的旋转，成本不菲

删除操作后，最多需旋转 $\Omega(\log n)$ 次（Knuth：平均仅0.21次）

若需频繁进行插入/删除操作，未免得不偿失

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

8. 高级搜索树

伸展树

逐层伸展

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 //BST就是这样的一个例子
- ❖ BST : 刚刚被访问过的节点，极有可能很快地再次被访问
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
- ❖ 连续的 m 次查找 ($m \gg n = |BST|$)，采用AVL共需 $O(m\log n)$ 时间
- ❖ 利用局部性，能否更快？ //仿效自适应链表
- ❖ 策略： 节点一旦被访问，随即调整至树根 //如此，下次访问即可...
- ❖ 问题： 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，随即转移至树根

❖ 一步一步往上爬

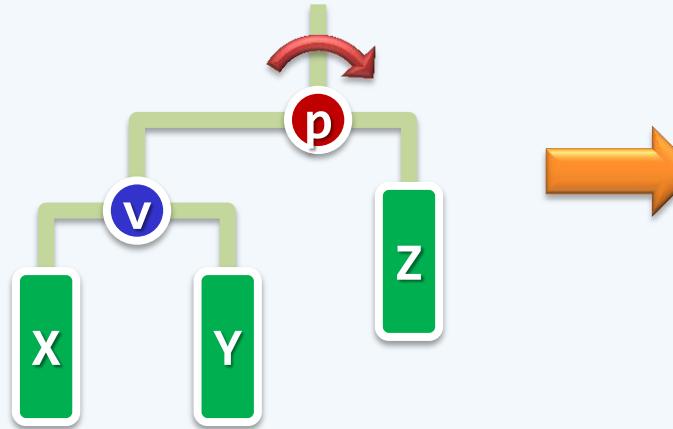
- 自下而上，逐层单旋

- zig(v->parent)

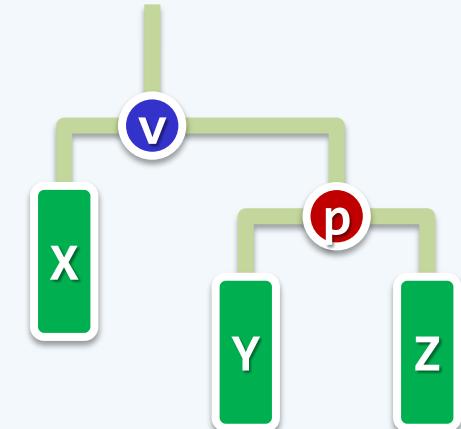
- zag(v->parent)

- 直到v最终被推送至根

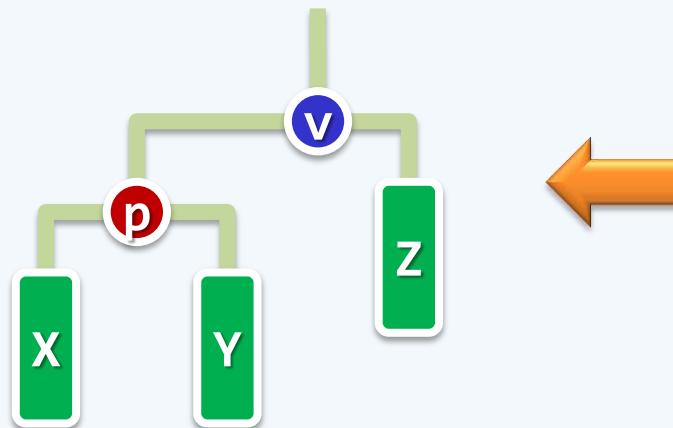
zig(p)



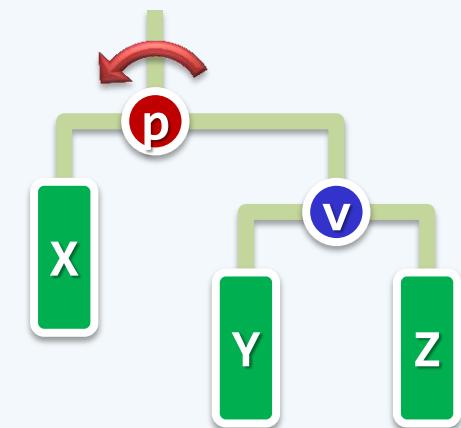
zigged



zagged



zag(p)



实例

❖ 伸展过程的效率

是否足够地高？

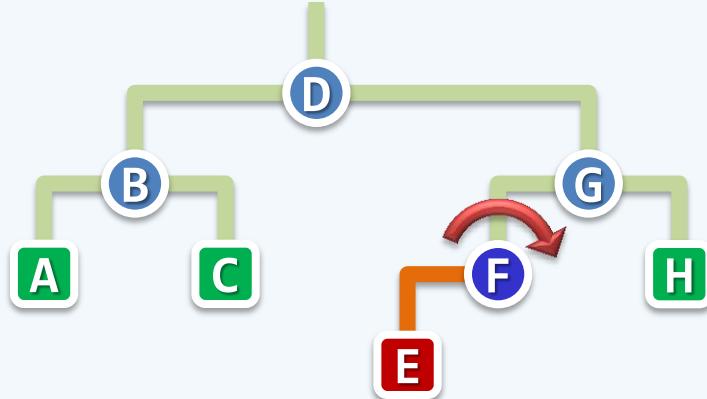
❖ 就逐层伸展的

策略而言

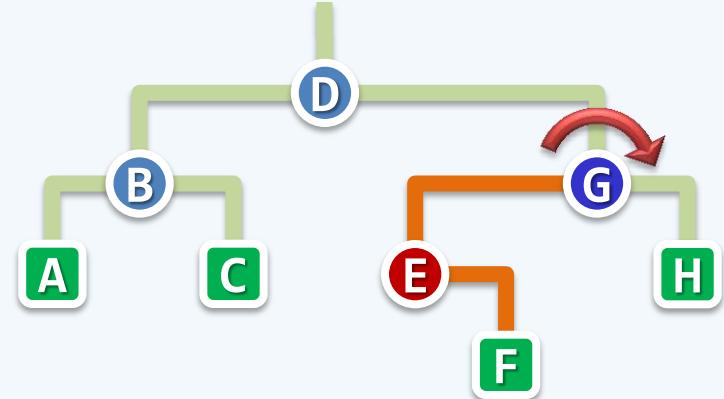
这取决于

- 树的初始形态和
 - 节点的访问次序

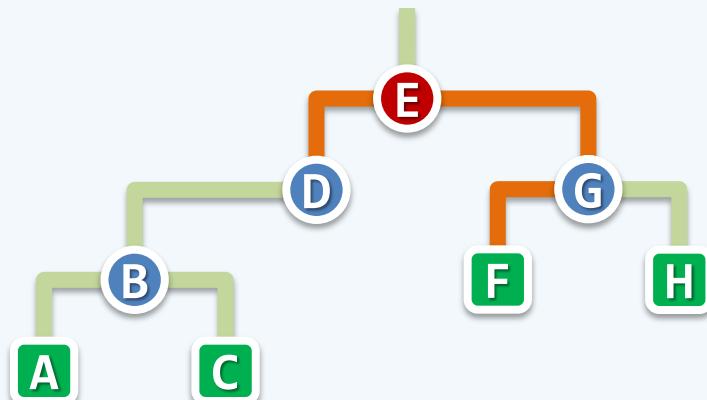
a) 访问E之后，做zig(F)



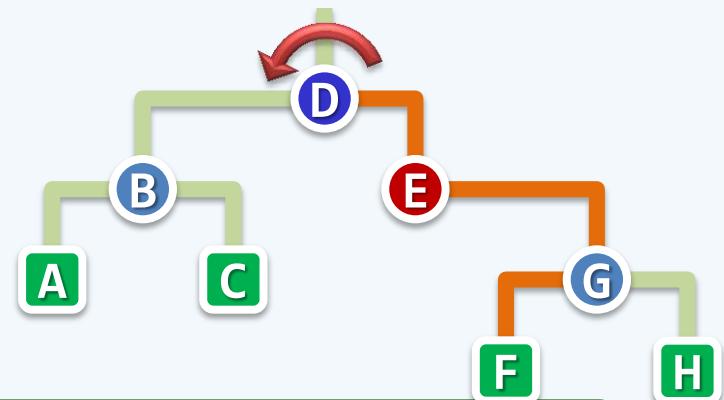
b) 继而zig(G)



d) 经3次旋转，E最终调整至树根

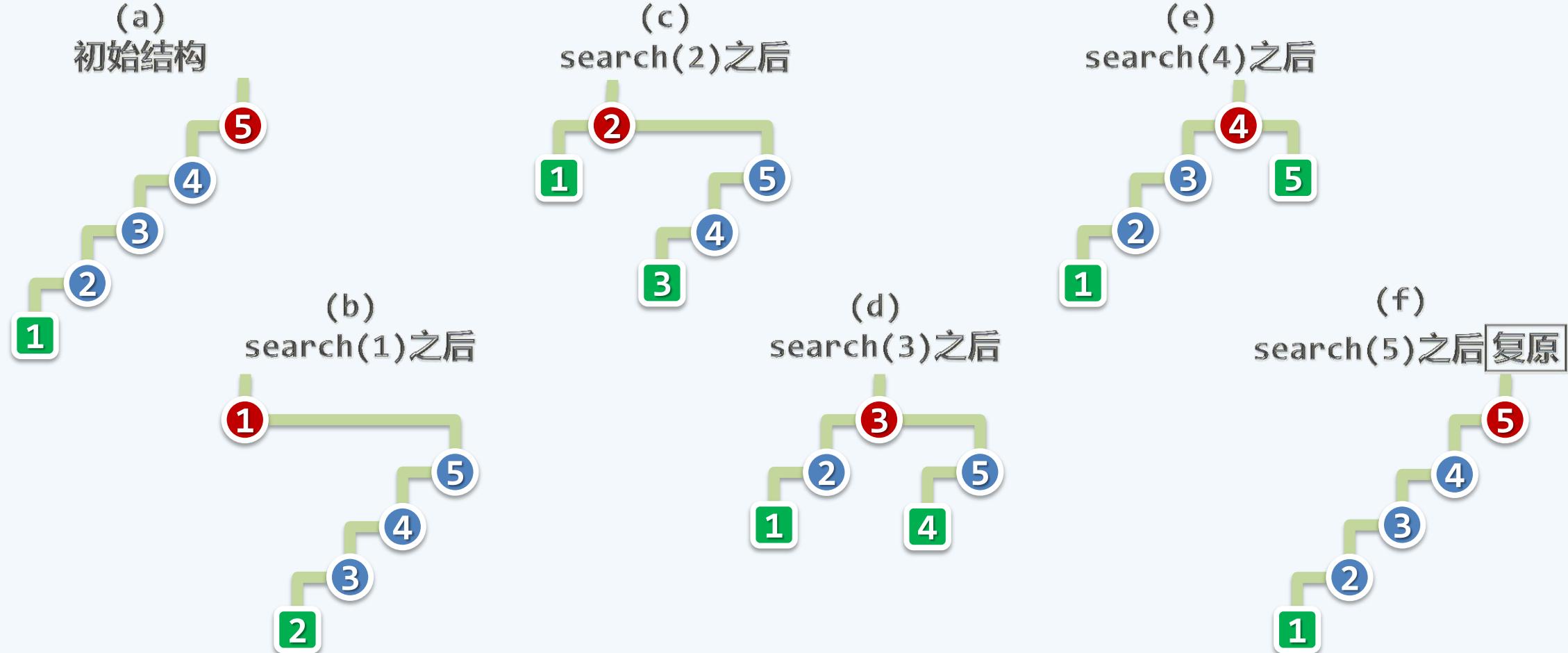


c) 继而zag(D)



最坏情况

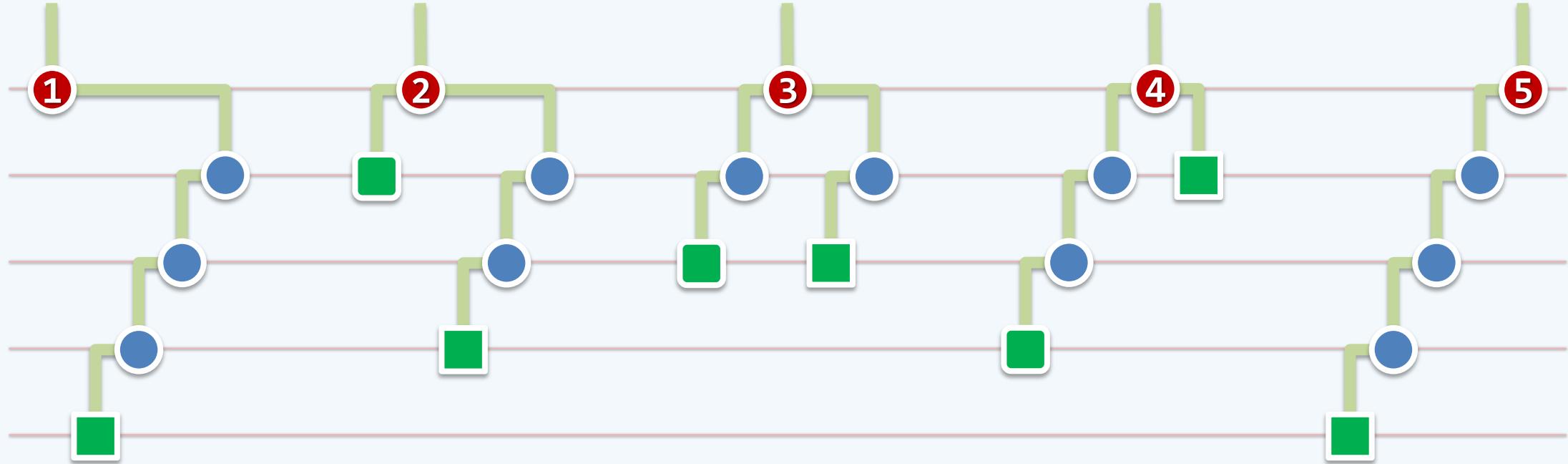
◆ 旋转次数呈周期性的算术级数演变：每一周期累计 $\Omega(n^2)$ ，分摊 $\Omega(n)$ ！



问题出在...

❖ 全树拓扑始终呈单链条结构，等价于一维列表

被访问节点的深度，呈**周期性的算术级数**演变：{ n-1, n-2, n-3, ..., 3, 2, 1 }



❖ 问题的症结既已确定，便可针对性地改进...

8. 高级搜索树

伸展树

双层伸展

贾政道：“不用全打开，怕叠起来倒费事。”

邓俊辉

詹光便与冯紫英一层一层折好收拾。

deng@tsinghua.edu.cn

双层伸展

❖ D. D. Sleator & R. E. Tarjan

Self-Adjusting Binary Trees

J. ACM, 32:652-686, 1985



❖ 构思的精髓：向上追溯**两层**，而非**一层**

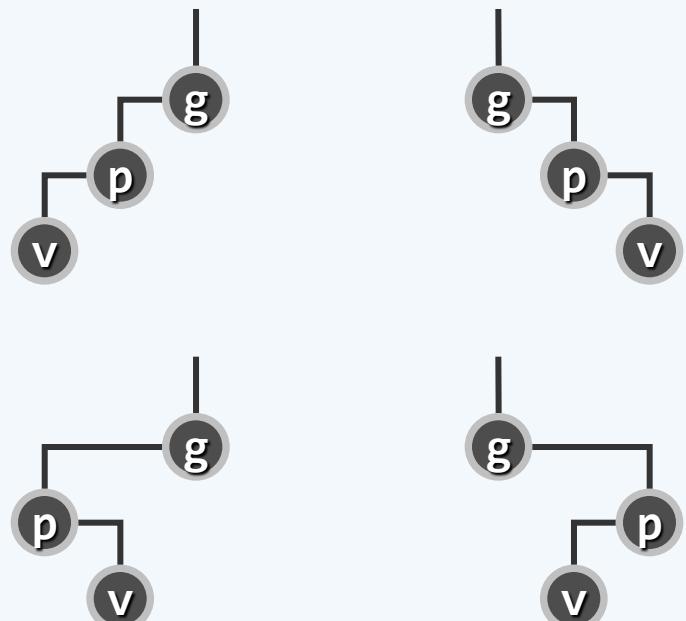
❖ 反复考察**祖孙三代**： $g = \text{parent}(p)$, $p = \text{parent}(v)$, v

❖ 根据它们的相对位置，经**两次旋转**使得

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？

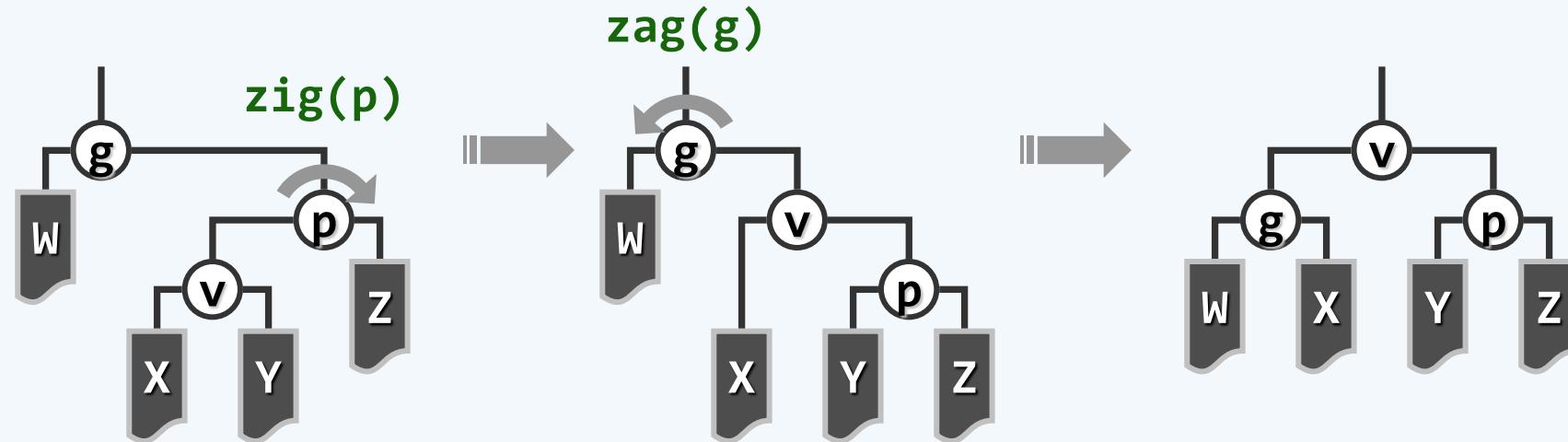


zig-zag / zag-zig

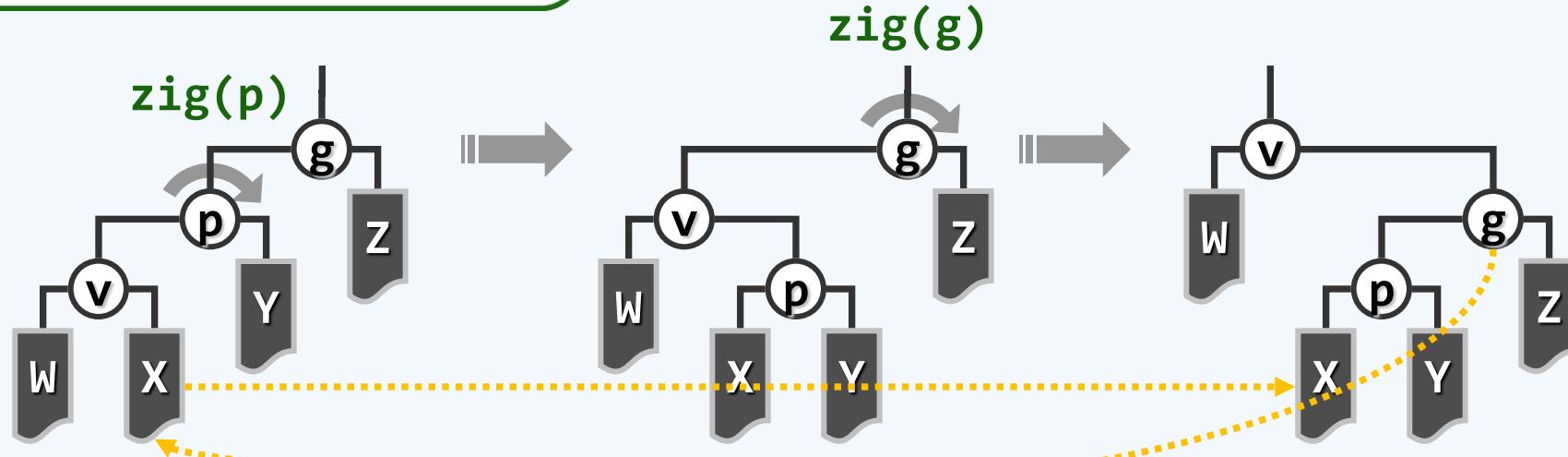
❖ 与 AVL 树双旋完全等效！

❖ 与逐层伸展别无二致！

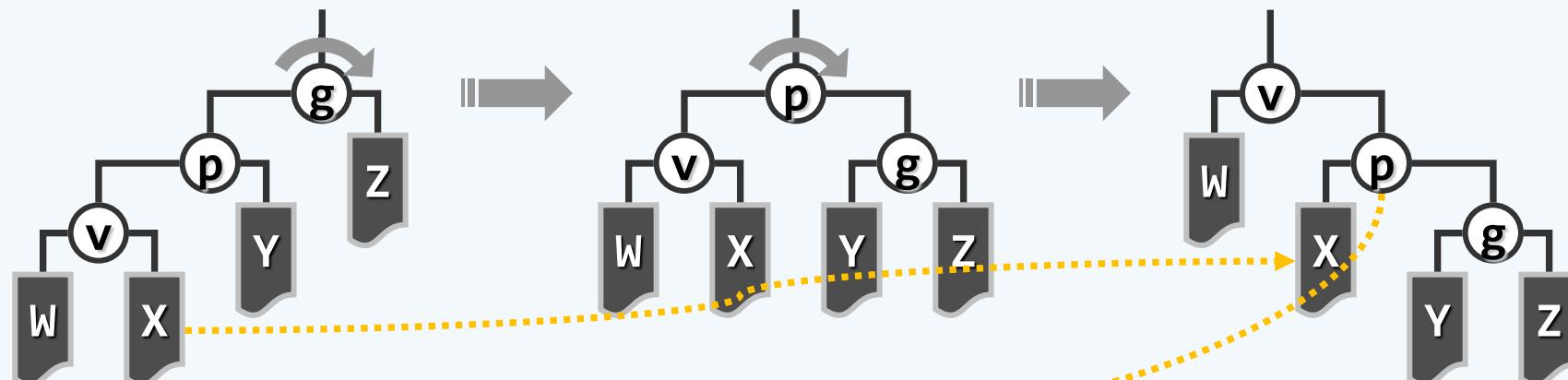
❖ 难道，就这样平淡无奇？



zig-zig / zag-zag



◆ 颠倒次序之后，**局部**的细微差异，将彻底地改变**整体**…

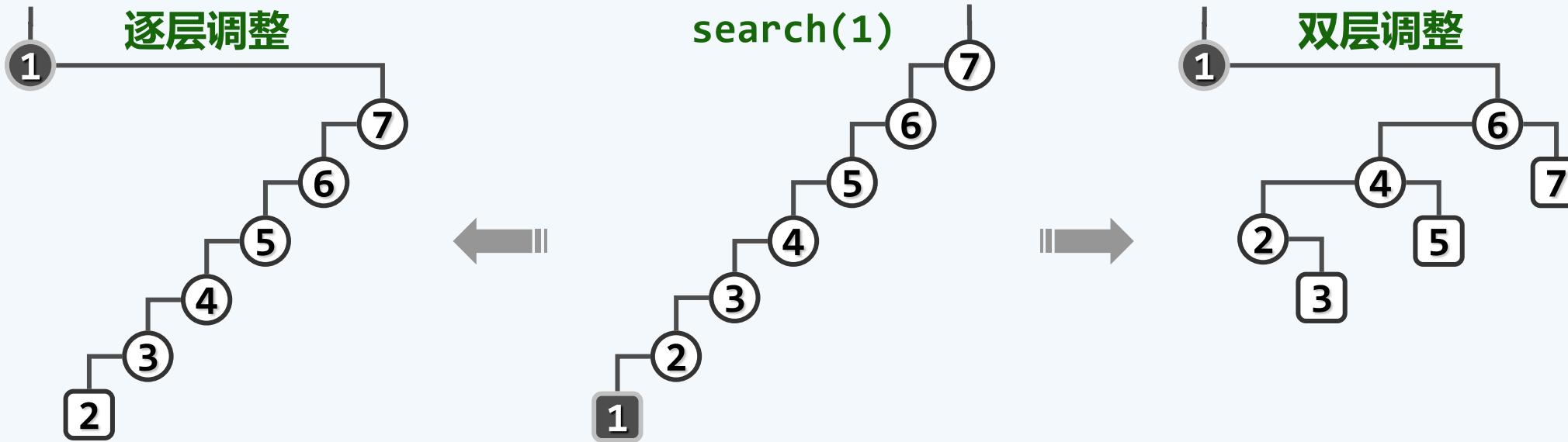


zig-zig / zag-zag

- ◆ 折叠效果：一旦访问坏节点，**对应路径**的长度将随即**减半** //含羞草
- ◆ 最坏情况不致持续发生！

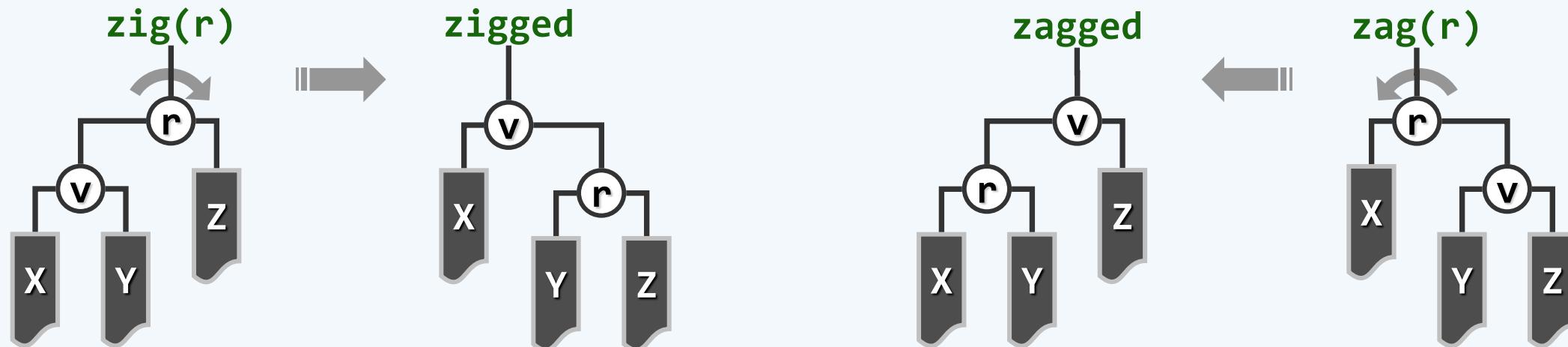
单趟伸展操作，分摊 $\mathcal{O}(\log n)$ 时间！

//严格证明，详见习题[8-2]



zig / zag

- ❖ 要是 v 只有父亲，没有祖父呢？
- ❖ 此时必有 $\text{parent}(v) == \text{root}(T)$ ，且
每轮调整中，这种情况至多（在最后）出现一次
- ❖ 视具体形态，做单次旋转： $\text{zig}(r)$ 或 $\text{zag}(r)$



8. 高级搜索树

伸展树

算法实现

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。

邓俊辉

deng@tsinghua.edu.cn

伸展树接口

❖ `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

```

❖ 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; //若无曾祖父gg，则v现即为树根；否则，gg此后应以v为左或右
        else ( g == gg->lc ) ? attachAsLChild(gg, v) : attachAsRChild(gg, v); //孩子
        updateHeight( g ); updateHeight( p ); updateHeight( v );
    } //双层伸展结束时，必有g == NULL，但p可能非空
    if ( p = v->parent ) { /* 若p果真是根，只需在额外单旋（至多一次） */ }
    v->parent = NULL; return v; //伸展完成，v抵达树根
}

```

伸展算法

```

❖ if ( IsLChild( * v ) )

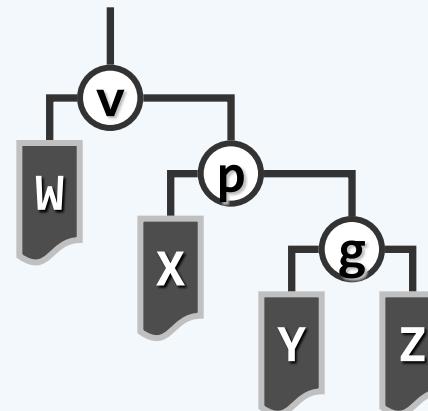
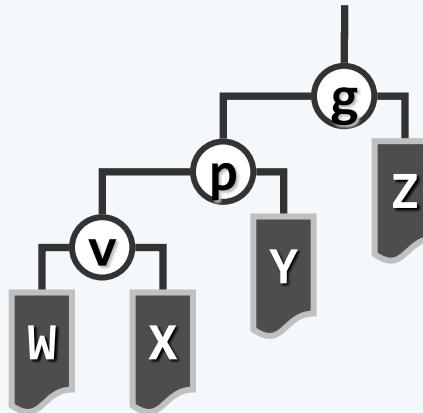
    if ( IsLChild( * p ) ) { //zIg-zIg
        attachAsLChild( g, p->rc );
        attachAsLChild( p, v->rc );
        attachAsRChild( p, g );
        attachAsRChild( v, p );
    } else { /* zIg-zAg */ }

else

    if ( IsRChild( * p ) ) { /* zAg-zAg */ }

    else { /* zAg-zIg */ }

```



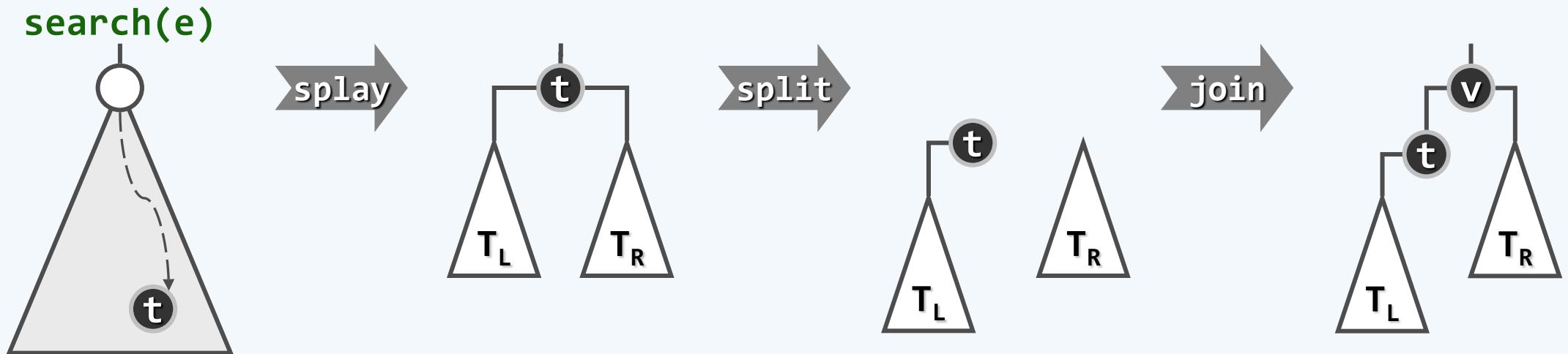
查找算法

- ❖

```
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 = searchIn( _root, e, _hot = NULL );  
  
    // 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;  
}
```
- ❖ 伸展树的查找操作，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

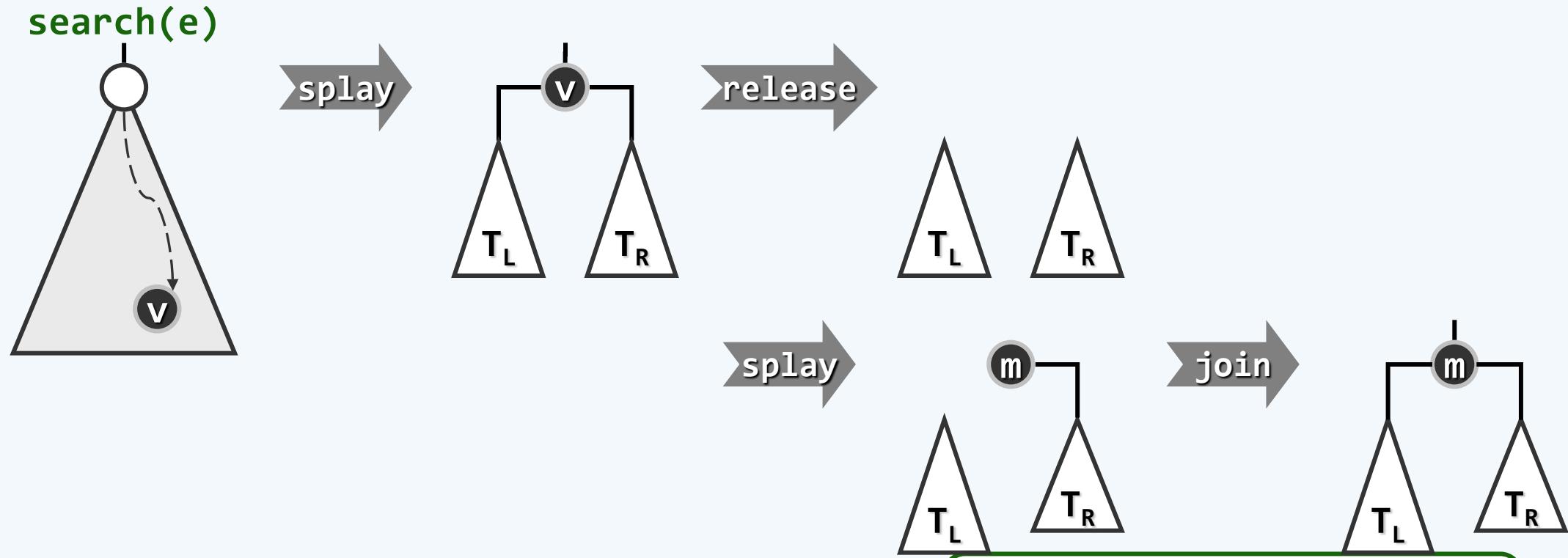
插入算法

- 直观方法：调用BST标准的插入算法，再将新节点伸展至根
其中，首先需调用`BST::search()`
- 重写后的`Splay::search()`已集成了`splay()`操作
查找（失败）之后，`_hot`即是根节点
- 既如此，何不随即就在树根附近完成新节点的接入…



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
- 同样地，`Splay::search()`查找（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...



综合评价

❖ 无需记录节点高度或平衡因子；编程实现简单易行——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 $k \ll n \ll m$ ）

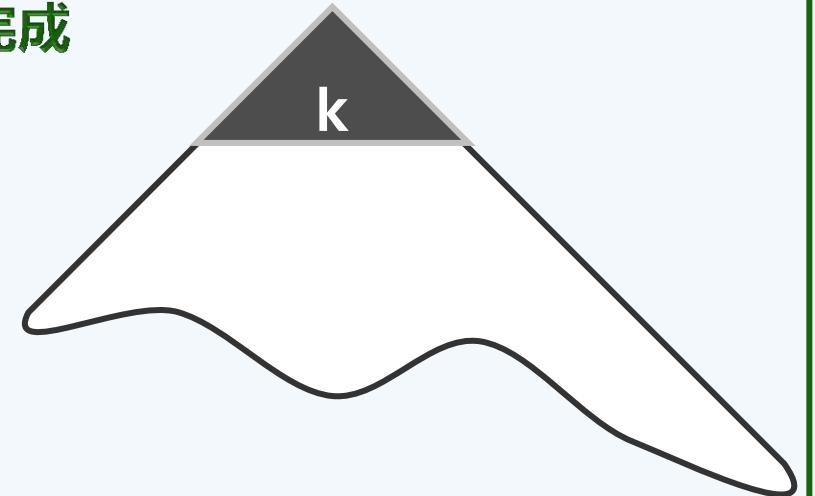
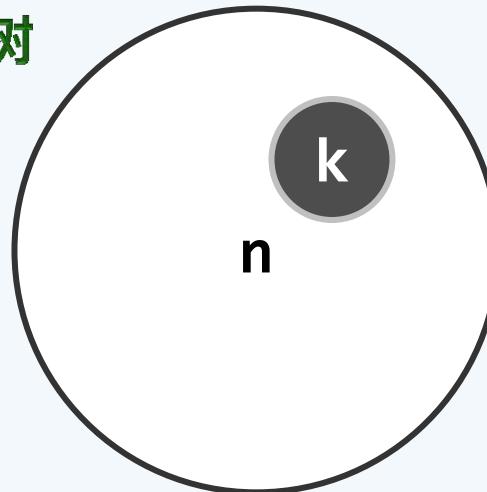
效率甚至可以更高——自适应的 $\mathcal{O}(\log k)$

任何连续的 m 次查找，都可在 $\mathcal{O}(m \log k + n \log n)$ 时间内完成

❖ 仍不能杜绝单次最坏情况的出现

不适用于对效率敏感的场合

❖ 复杂度的分析稍嫌复杂——好在有初等的方法



8. 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。你权当百日後才娶我。你就忍一忍，一百天很快就过去了.....”

邓俊辉

deng@tsinghua.edu.cn

越来越小的内存

❖ RAM : 存储器？不就是无限可数个寄存器吗？

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

Turing : 存储器？不就是无限长的纸带吗？

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

❖ 但事实上

系统存储容量的增长速度

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

<< 应用问题规模的增长速度

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

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

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

2010

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

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

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

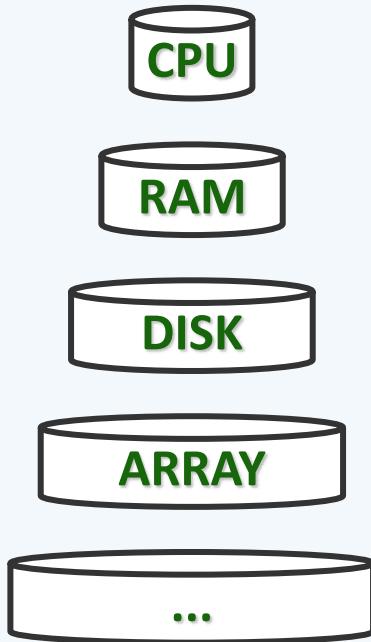
$$1 \text{ Doggabyte} = 2^{100} = 10^{30}$$

越来越小的内存

- ❖ 典型的
数据库规模 / 内存容量 1980 : 10MB / 1MB = 10
- ❖ 今天典型的数据集
须以TB为单位度量 2000 : 1TB / 1GB = 1000
- ❖ 亦即，相对而言...内存容量是在...不断减小！
- ❖ 为什么不把内存做得更大？
- ❖ 物理上，存储器的容量越大/小，访问速度就越慢/快

高速缓存

- ❖ 事实1：不同容量的存储器，访问速度差异悬殊
- ❖ 以磁盘与内存为例： ms / ns > 10^5
- ❖ 若一次内存访问需要一秒，则一次外存访问就相当于一天
- ❖ 为避免1次外存访问，我们宁愿访问内存10次、100次，甚至...
- ❖ 多数存储系统，都是分级组织的——Caching
 最常用的数据尽可能放在更高层、更小的存储器中
 实在找不到，才向更低层、更大的存储器索取
- ❖ 算法的I/O复杂度 ∝ 数据在不同存储级别之间的传输次数
 算法的实际运行时间，往往主要取决于此



高速缓存

- ❖ 事实2：从磁盘中读写1B，与读写1KB几乎一样快
- ❖ 批量式访问：以页（page）或块（block）为单位，使用缓冲区 //<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
  
```

- ❖ 效果：单位字节的平均访问时间大大缩短

8. 高级搜索树

B-树
结构

妻子好合，如鼓瑟琴

兄弟既翕，和乐且湛

邓俊辉

deng@tsinghua.edu.cn

等价变换

❖ 1970, R. Bayer & E. McCreight

❖ 平衡的多路 (multi-way) 搜索树

❖ 经适当合并，得 **超级节点**

每 **2代** 合并 : **4路**

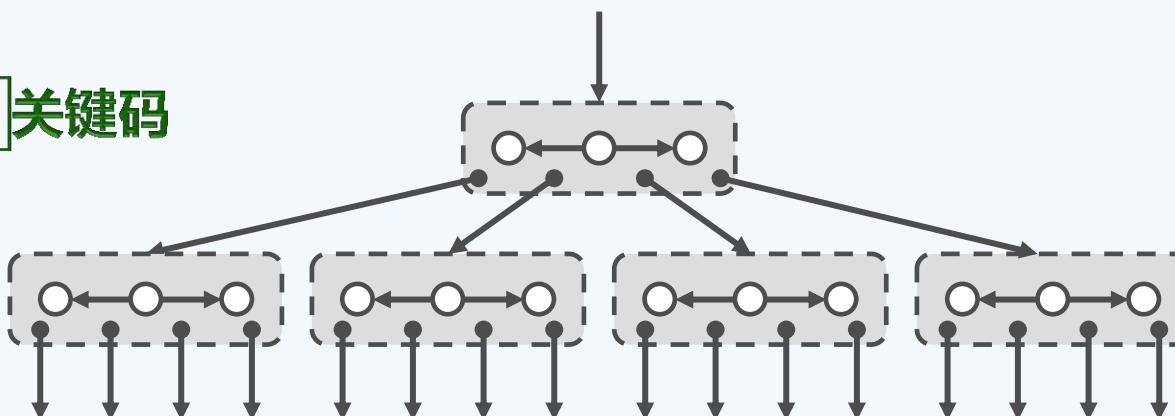
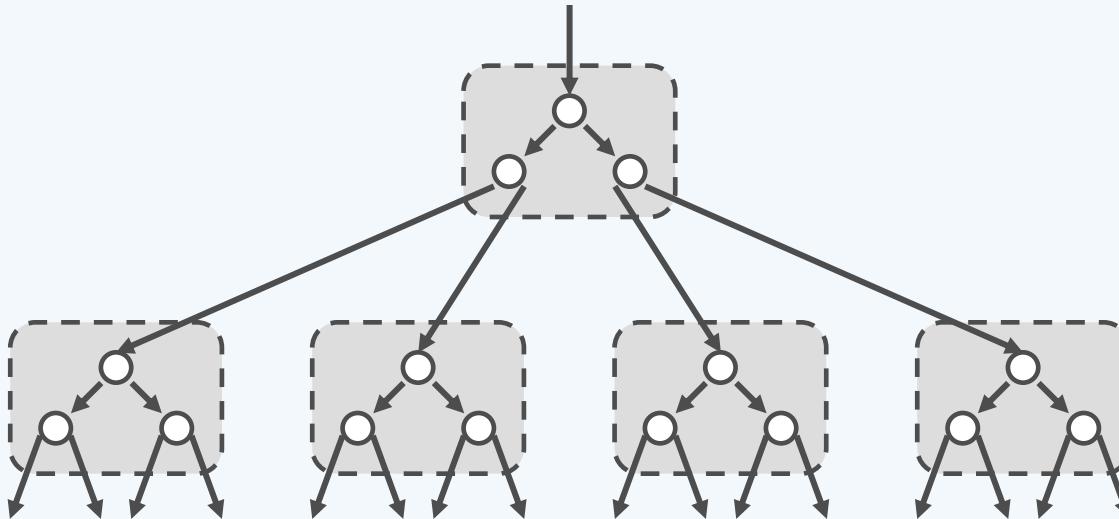
每 **3代** 合并 : **8路**

...

每 **d代** 合并 : **$m = 2^d$ 路** , **$m - 1$ 个关键码**

❖ 逻辑上与BBST **完全等价**

——既然如此，为何还要引入B-树？



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 $\log(2, 10^9) = 30$ 次I/O操作，每次只读出一个关键码，得不偿失

❖ B-树又能如何？

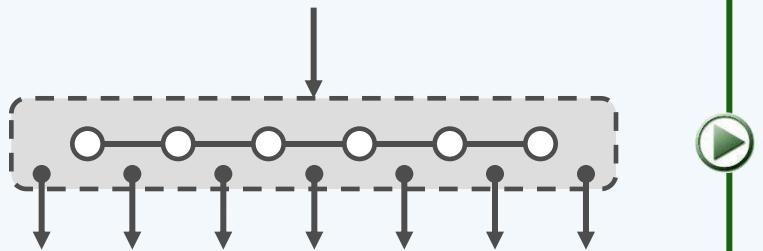
充分利用外存对批量访问的高效支持，将此特点转化为优点

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 $m = 200 \sim 300$

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



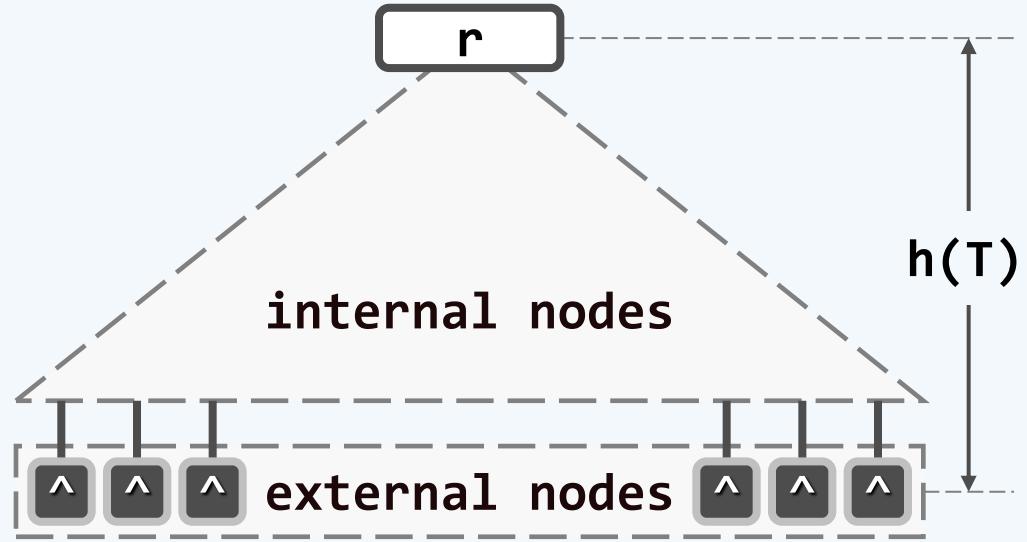
外部节点 + 叶子

❖ 所谓**m阶B-树**，即m路平衡搜索树（ $m \geq 2$ ）

❖ **外部节点**的深度统一相等

所有**叶节点**的深度统一相等

❖ 树高**h** = 外部节点的深度



内部节点

❖ 内部节点各有

不超过 $m - 1$ 个关键码 :

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

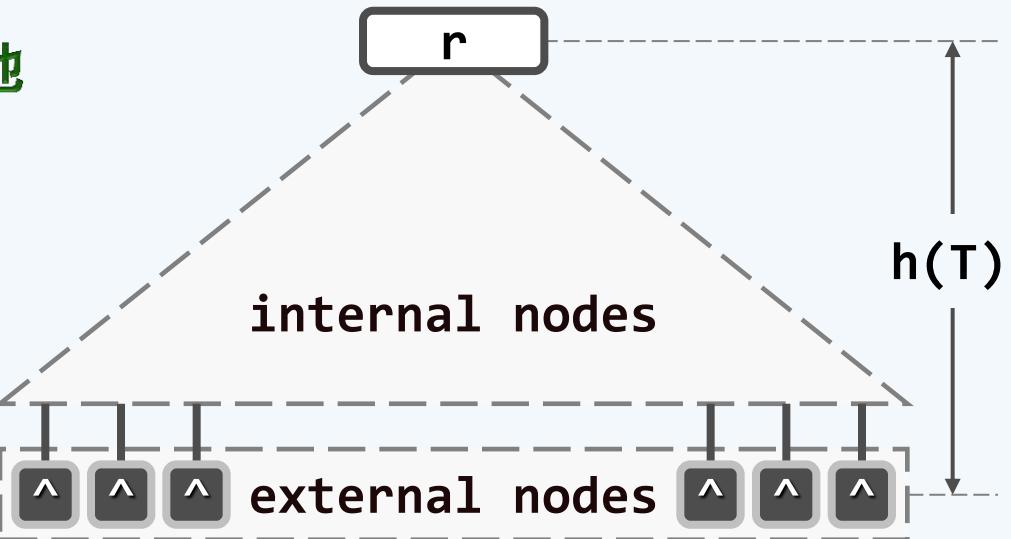
不超过 m 个分支 :

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

❖ 内部节点 的 分支数 $n + 1$ 也不能太少 , 具体地

树根 : $2 \leq n + 1$

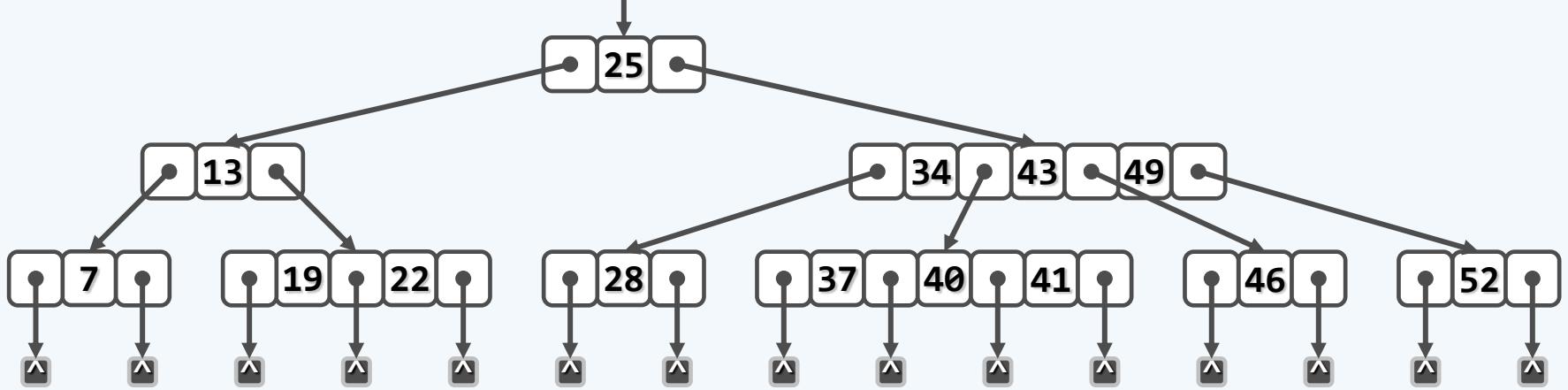
其余 : $\lceil m/2 \rceil \leq n + 1$



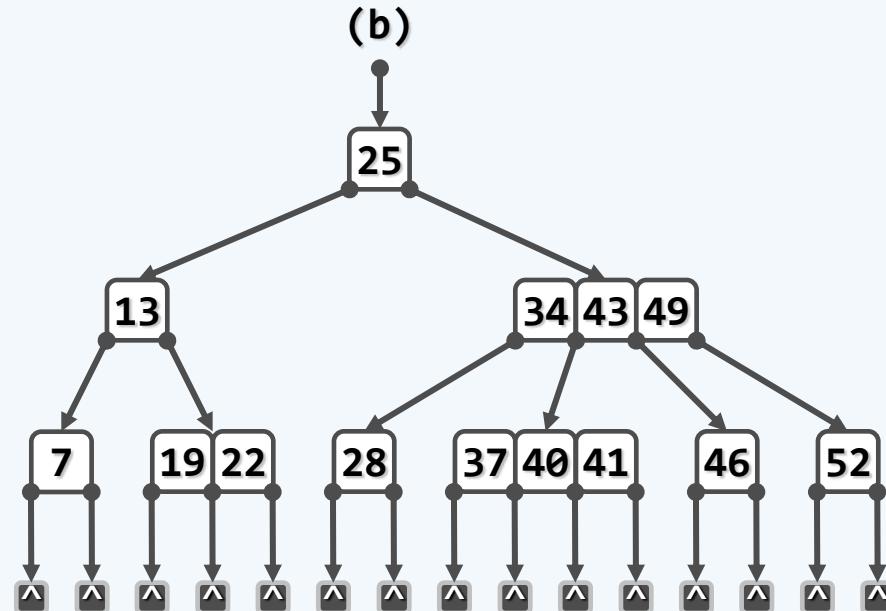
❖ 故亦称作 $(\lceil m/2 \rceil, m)$ -树

紧凑表示

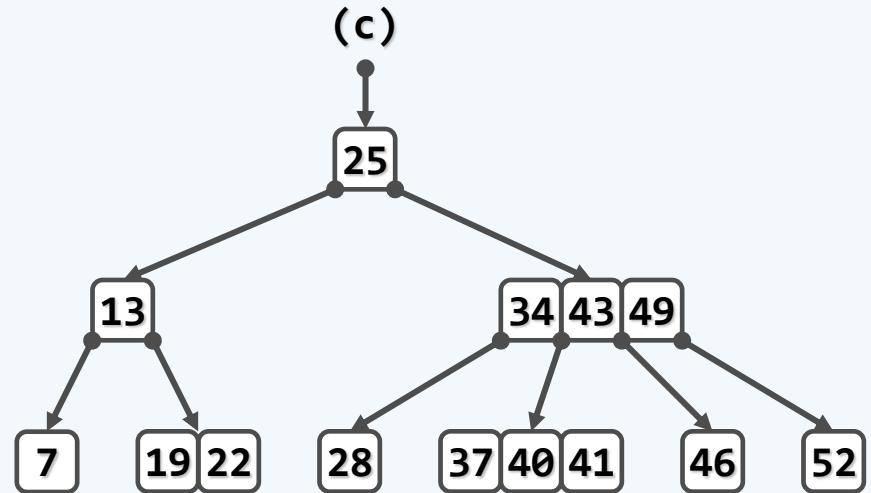
(a)



(b)



(c)



实例

❖ demo/b_tree/

❖ **m = 3**

▶ 2-3-树，(2,3)-树，最简单的B-树 //John 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

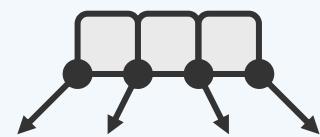


❖ **m = 4**

▶ 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**，对稍后理解**红黑树**大有裨益

BTNode

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

 BTNodePosi(T) parent; //父

vector<T> key; //数值向量

vector< BTNodePosi(T) > child; //孩子向量 (其长度总比key多一)

BTNode() { parent = NULL; child.insert(0, NULL); }

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

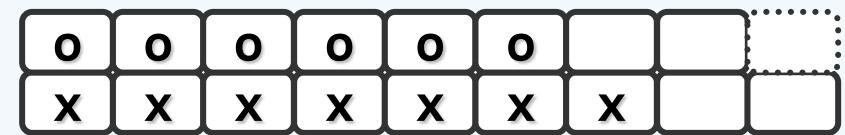
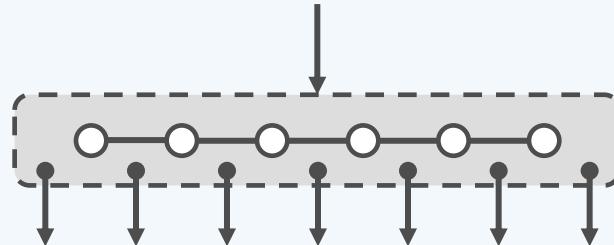
 parent = NULL; //作为根节点，而且初始时

 key.insert(0, e); //仅一个关键码，以及

 child.insert(0, lc); child.insert(1, rc); //两个孩子

 if (lc) lc->parent = this; if (rc) rc->parent = this;

}



BTree

```
❖ #define BTNodePosi(T) BTNode<T>* //B-树节点位置  
❖ template <typename T> class BTree { //B-树  
protected:  
    int _size; int _order; BTNodePosi(T) _root; //关键码总数、阶次、根  
    BTNodePosi(T) _hot; //search()最后访问的非空节点位置  
    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  
};
```

8. 高级搜索树

高至天低至深海

每寸搜索着这天下

寻觅着那个"它"

B-树

查找

...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。

邓俊辉

deng@tsinghua.edu.cn

算法

❖ 将根节点作为当前节点 //常驻RAM

只要当前节点非外部节点

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 //止于某一对下层引用

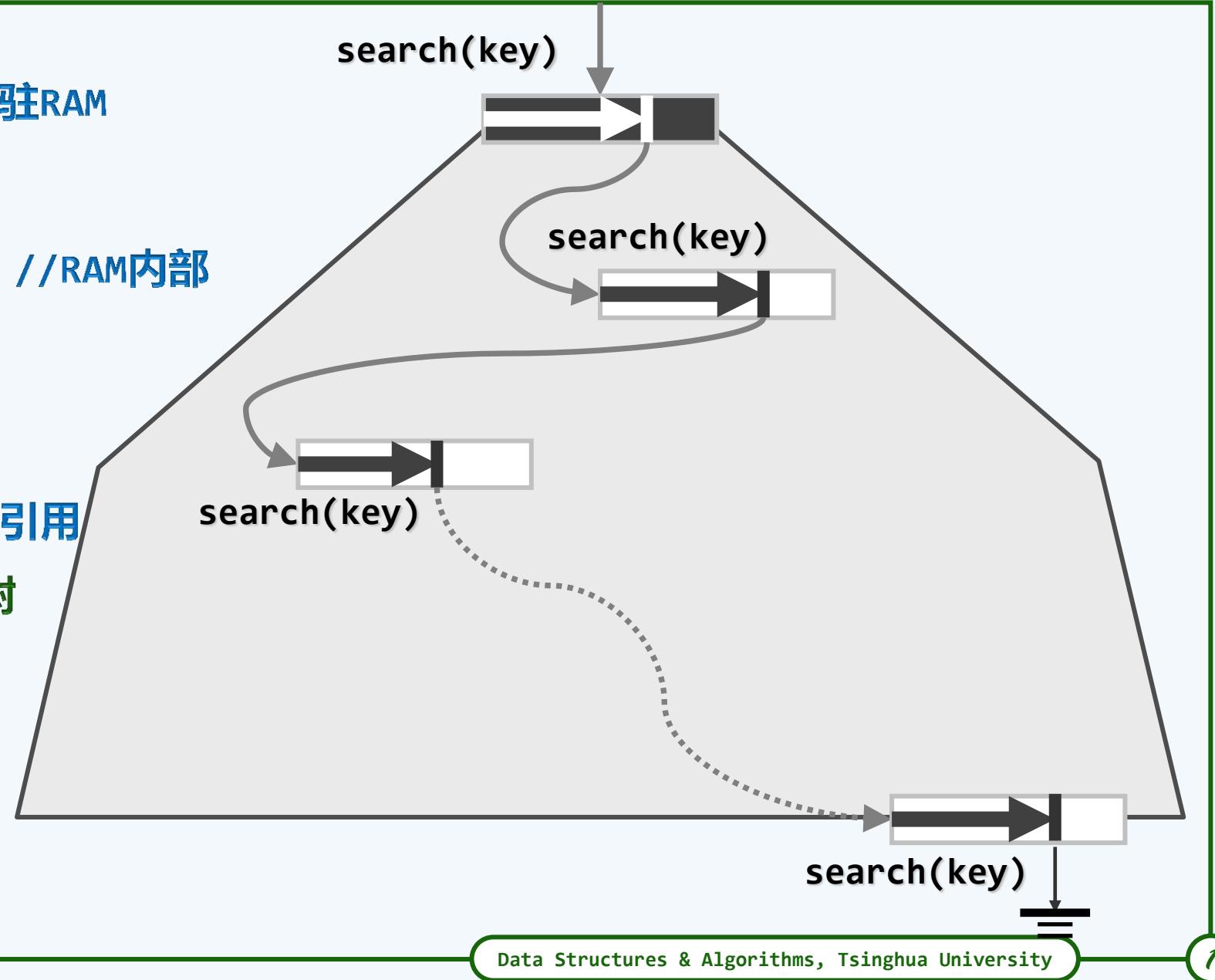
沿引用，转至对应子树

将其根节点读入内存

//I/O，最为耗时

更新当前节点

返回查找失败

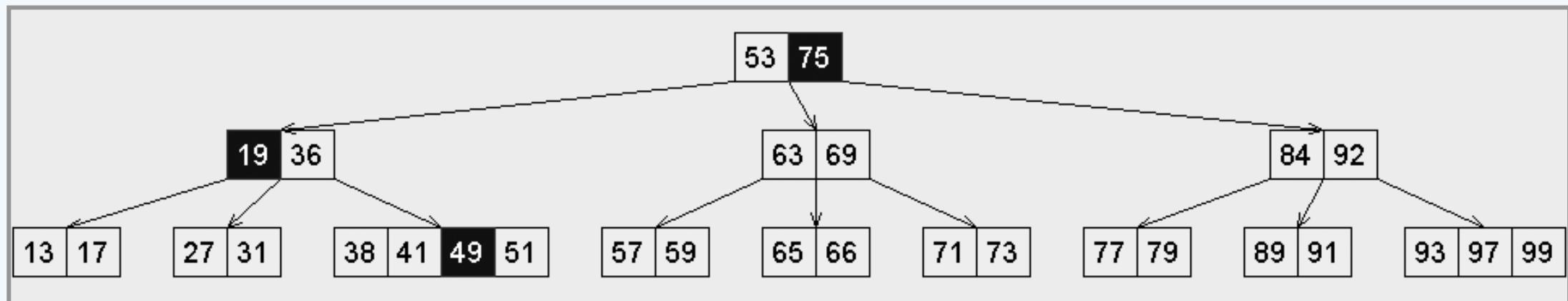


实例

❖ (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



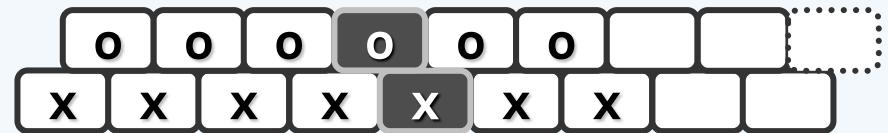
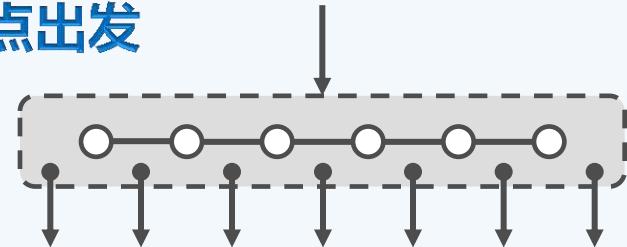
实现

```

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

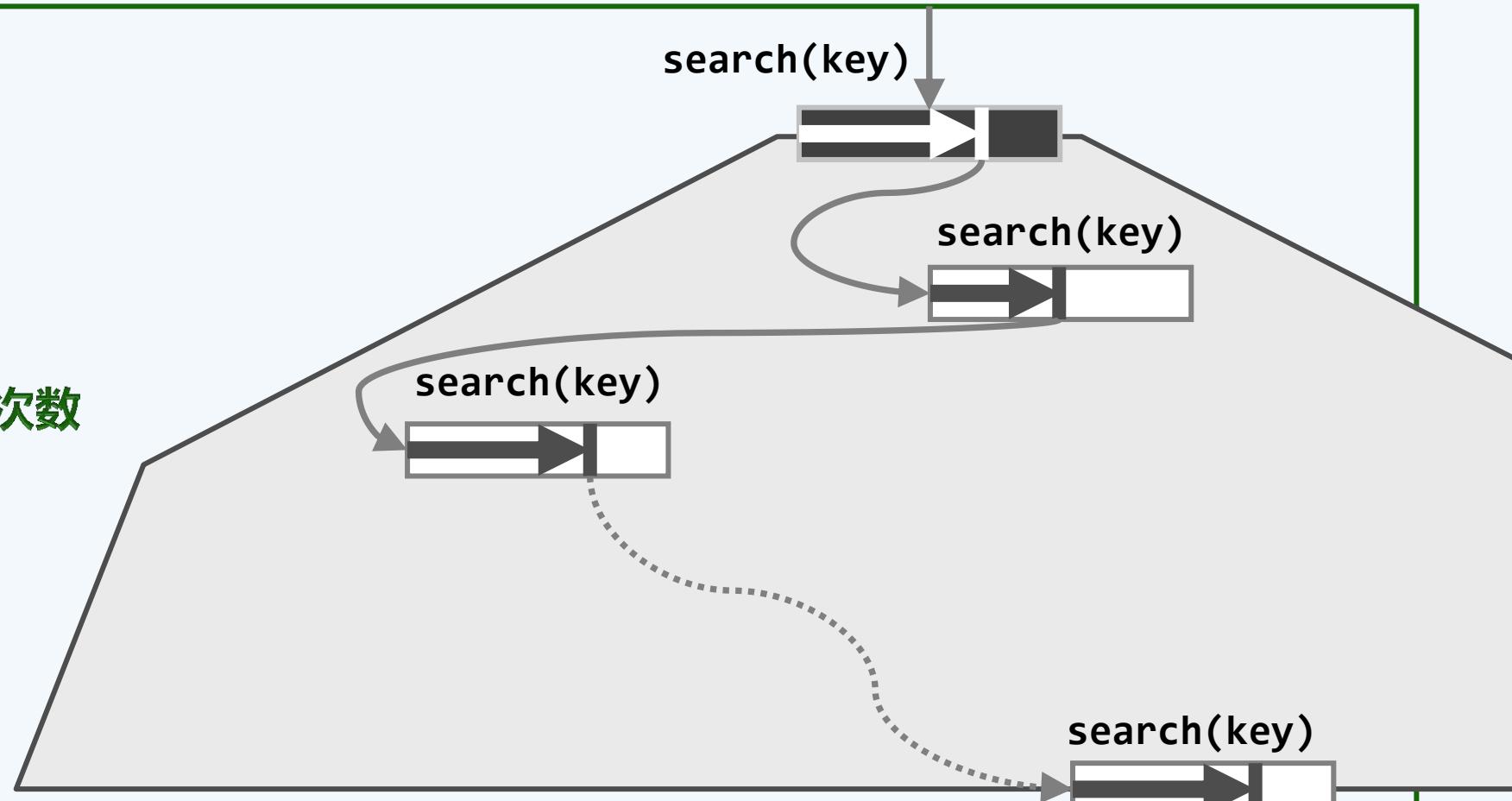
    BTNodePosi(T) v = _root; _hot = NULL; //从根节点出发
    while ( v ) { //逐层查找
        Rank r = v->key.search( e ); //在当前节点对应的向量中顺序查找
        if ( 0 <= r && e == v->key[ r ] ) return v; //若成功，则返回；否则...
        _hot = v; v = v->child[ r + 1 ]; //沿引用转至对应的下层子树，并载入其根I/O
    } //若因!v而退出，则意味着抵达外部节点
    return NULL; //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
 - = $\Theta(\text{终止节点的深度})$
 - = $\Theta(h)$
- ❖ 可以证明： $\log_m(N + 1) \leq h \leq 1 + \log_{\lceil m/2 \rceil} \lfloor (N + 1)/2 \rfloor$



最大树高

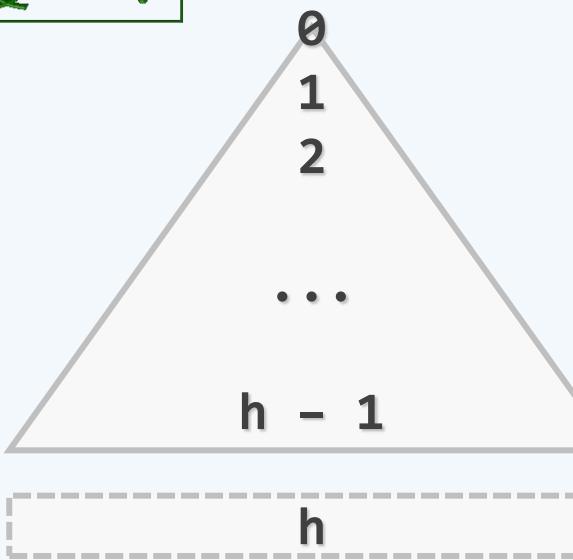
含 N 个关键码的 m 阶 B- 树，最大高度 = ?

为此，内部节点应尽可能 “瘦”

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

考查外部节点所在层

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



n_0	≥ 1
n_1	≥ 2
n_2	$\geq 2 \times \lceil m/2 \rceil$
n_3	$\geq 2 \times \lceil m/2 \rceil^2$
\dots	\dots
n_{h-1}	$\geq 2 \times \lceil m/2 \rceil^{h-2}$
n_h	$\geq 2 \times \lceil m/2 \rceil^{h-1}$

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

相对于 BST： $\log_{\lceil m/2 \rceil}(N/2) / \log_2 N = 1/(\log_2 m - 1)$

若取 $m = 256$ ，树高 (I/O 次数) 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， 最小高度 = ?

为此，内部节点应尽可能 “胖”

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

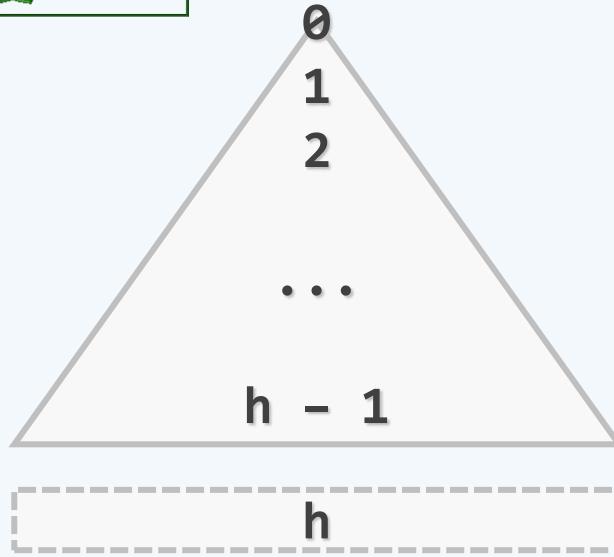
考查外部节点所在层：

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

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

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（I/O次数）约降低至 $1/8$



$$\begin{array}{lcl} n_0 & \leq & 1 \\ n_1 & \leq & m \\ n_2 & \leq & m^2 \\ n_3 & \leq & m^3 \\ \dots & \leq & \dots \\ n_{h-1} & \leq & m^{h-1} \\ n_h & \leq & m^h \end{array}$$

意义与价值

- ❖ BBST，究竟可能多高？
- ❖ 考查高度为 h 的BBST（如AVL）可包含节点的数目 f

$h = 33$ 时， $f = 2^{33} = 8.6 * 10^9$ //全球人口

$h = 77$ 时， $f = 2^{77} = 1.5 * 10^{23}$ //全球人口的体细胞总数

$h = 133$ 时， $f = 2^{133} = 1.1 * 10^{40}$ //国际象棋可能的局面总数

$h = 260$ 时， $f = 2^{260} = 1.8 * 10^{78}$ //目前可观测宇宙中基本粒子总数

- ❖ 由此可见，B-树的价值，的确更多地体现在实用方面

通过选取适当的节点规模（ m ），**弥合**存储层级之间巨大的**速度差异**

- ❖ 另外，在算法方面，B-树也有其独特的价值与地位...

8. 高级搜索树

B-树

插入

说再见，在这梦幻国度，最后的一瞥

邓俊辉

清醒让我，分裂再分裂

deng@tsinghua.edu.cn

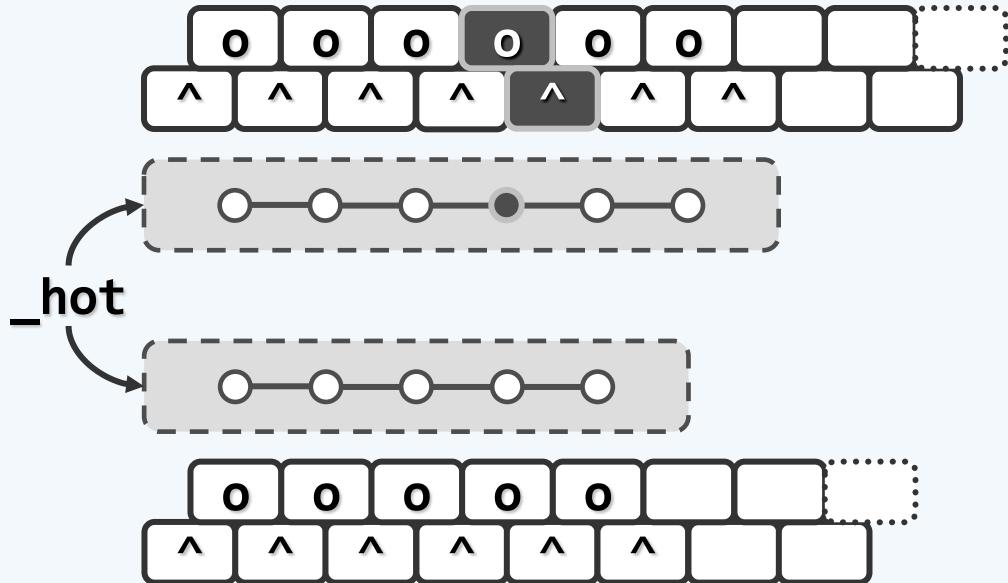
算法

```

❖ 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 ); // 创建一个空子树指针
    _size++; solveOverflow( _hot ); // 如发生上溢，需做分裂
    return true; // 插入成功
}

```



分裂

设上溢节点中的关键码依次为 k_0, \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} , k_s , k_{s+1}, \dots, k_{m-1}

关键码 k_s 上升一层，并

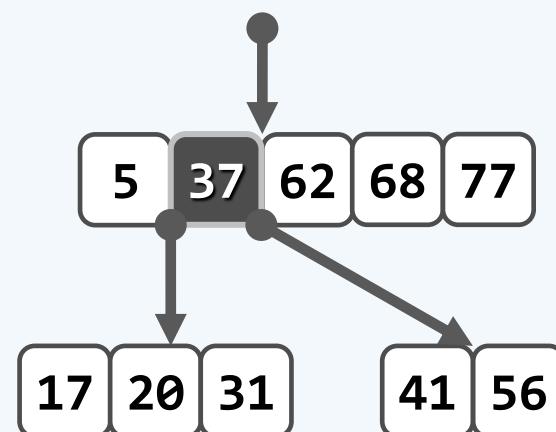
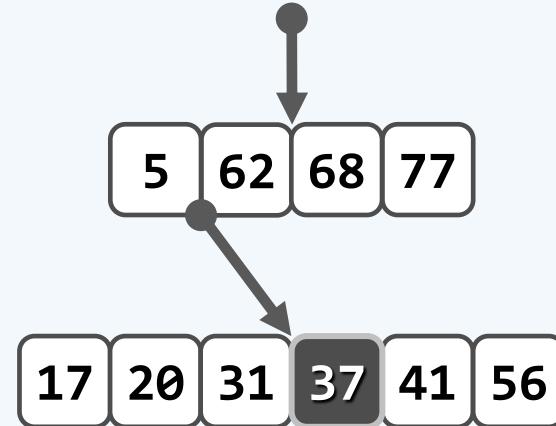
分裂 **split** : 以所得的两个节点作为左、右孩子

确认：如此分裂后

左、右孩子所含关键码数依然符合 m 阶 B-树的条件

$$s = \lfloor m/2 \rfloor \geq \lceil m/2 \rceil - 1$$

$$m - s - 1 = m - \lfloor m/2 \rfloor - 1 = \lceil m/2 \rceil - 1$$



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...

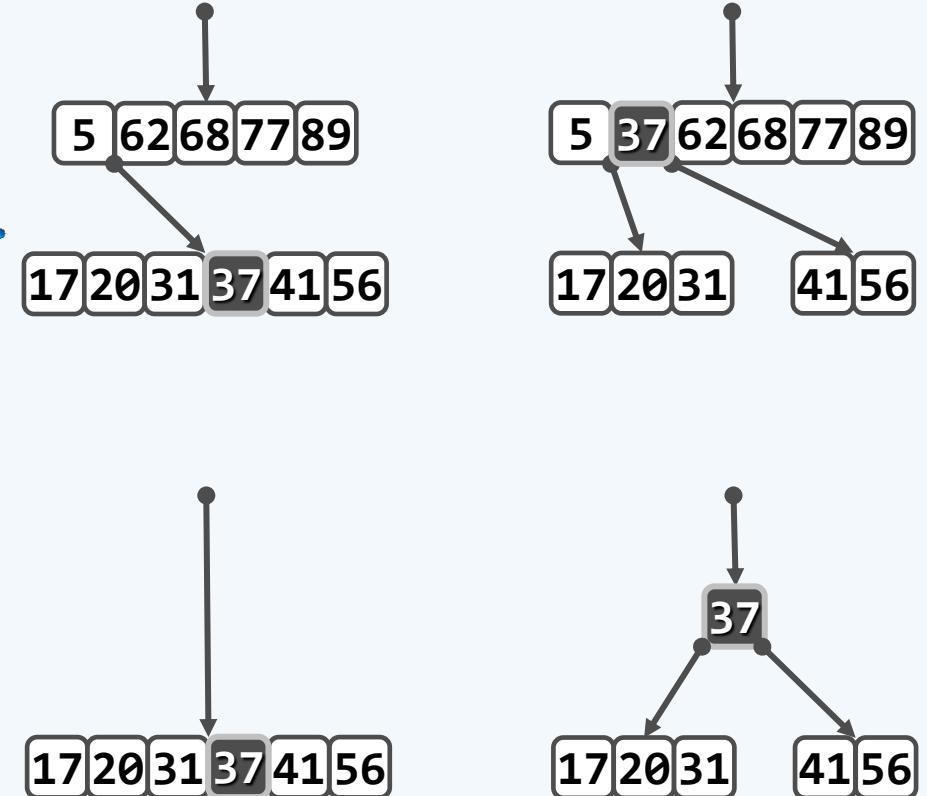
- 可令被提升的关键码自成节点，作为新的树根

这也是B-树增高的唯一可能 //概率多大？

- 注意：新的树根仅有两个分支

//可见树根的确不能固守下限 $m/2$

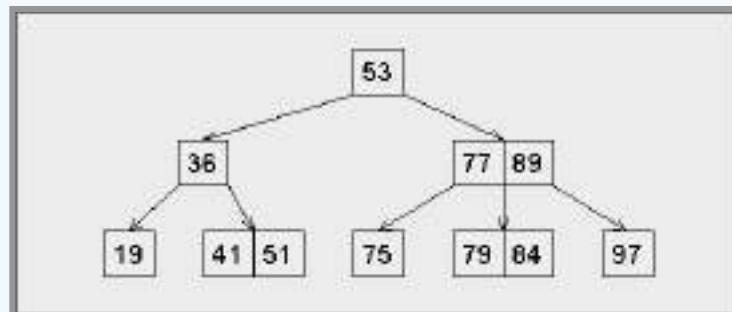
- 总体执行时间线性正比于分裂次数，不超过 $O(h)$



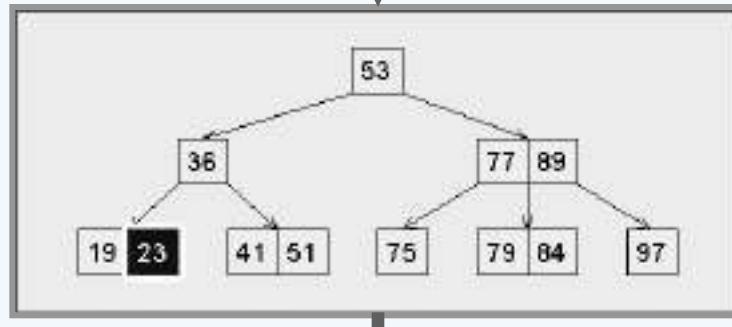
实例

❖ 2-3-树：

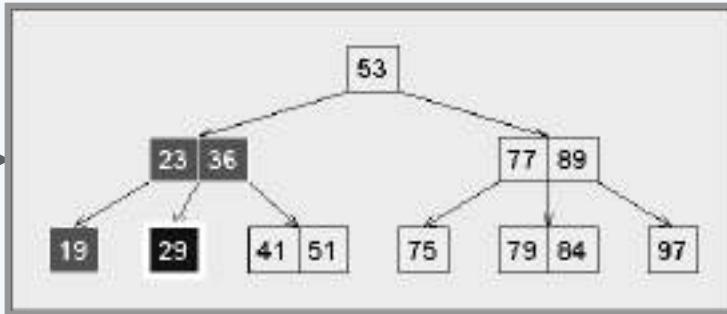
53 97 36 89 41 75 19 84 77 79 51



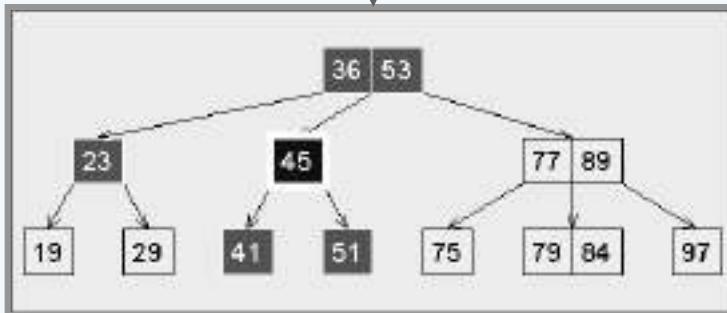
insert(23) //无需分裂



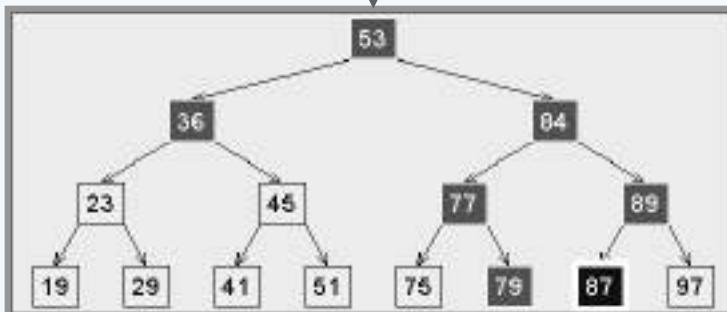
insert(29) //分裂一次



insert(45) //分裂两次



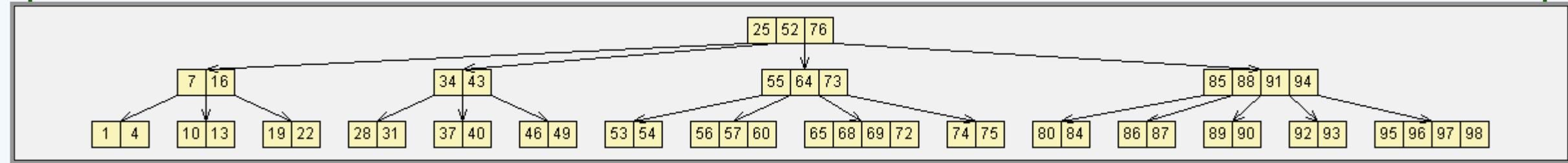
insert(87) //分裂到根



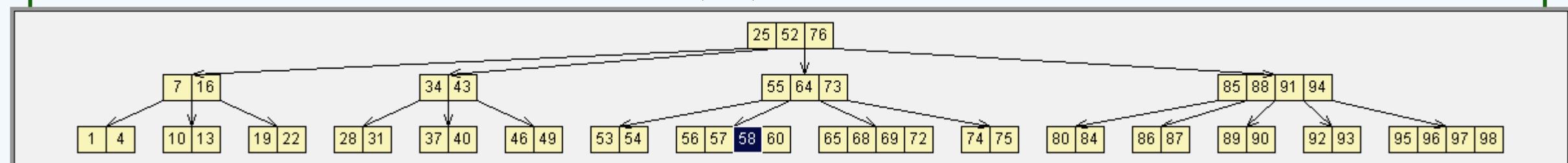
实例

3-5-树

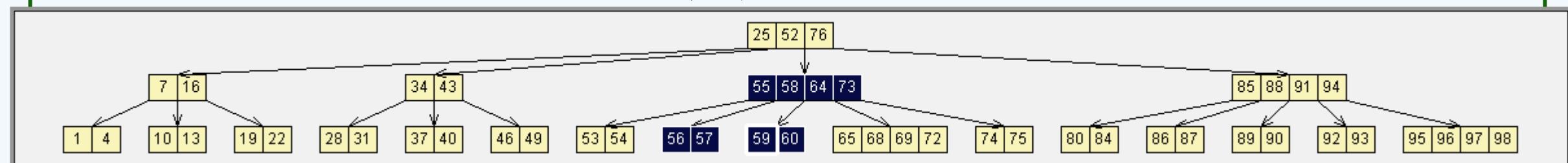
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 56 60 64 68 72 76 80 84 53 54 55 85 86 87 88 89 90 91 92 93 94 95 96 97 98 73 74 75 57 65 69



`insert(58) //无需分裂`

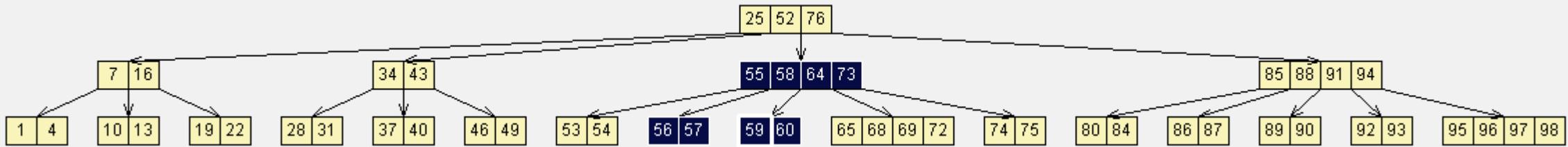


`insert(59) //分裂 1 次`

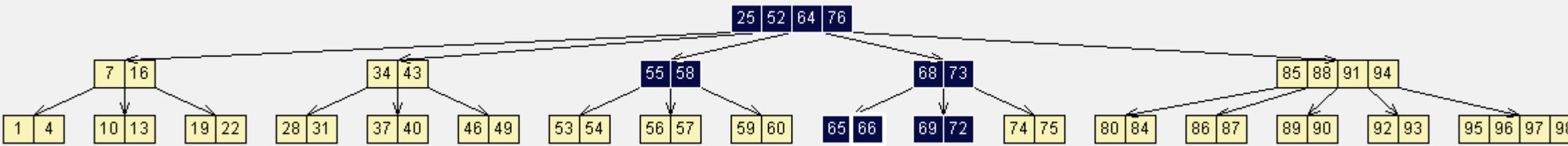


实例

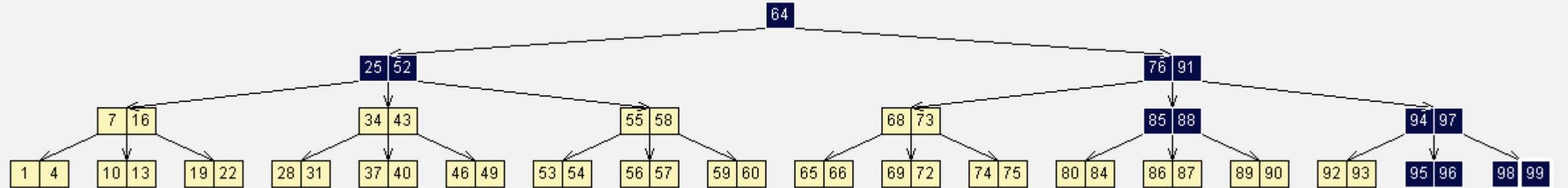
insert(59) //分裂 1 次



insert(66) //分裂 2 次



insert(99) //分裂到根



上溢修复

```

❖ template <typename T> void BTTree<T>::solveOverflow( BTNodePosi(T) v ) {
    if ( _order >= v->child.size() ) return; //递归基：不再上溢

    Rank s = _order / 2; //轴点（此时 _order = 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 < _order - [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 _order-s-1 个孩子

        u->key.insert( j, v->key.remove([s + 1]) ); //v右侧 _order-s-1 个关键码
    }

    u->child[_order - [s] - 1] = v->child.remove( [s + 1] ); //移动v最靠右的孩子
}
/* TBC */

```

上溢修复

```

❖ 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 < _order - 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

solveOverflow( p ); //上升一层，如有必要则继续分裂——至多递归O(logn)层
}

```

8. 高级搜索树

B-树

删除

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

邓俊辉

deng@tsinghua.edu.cn

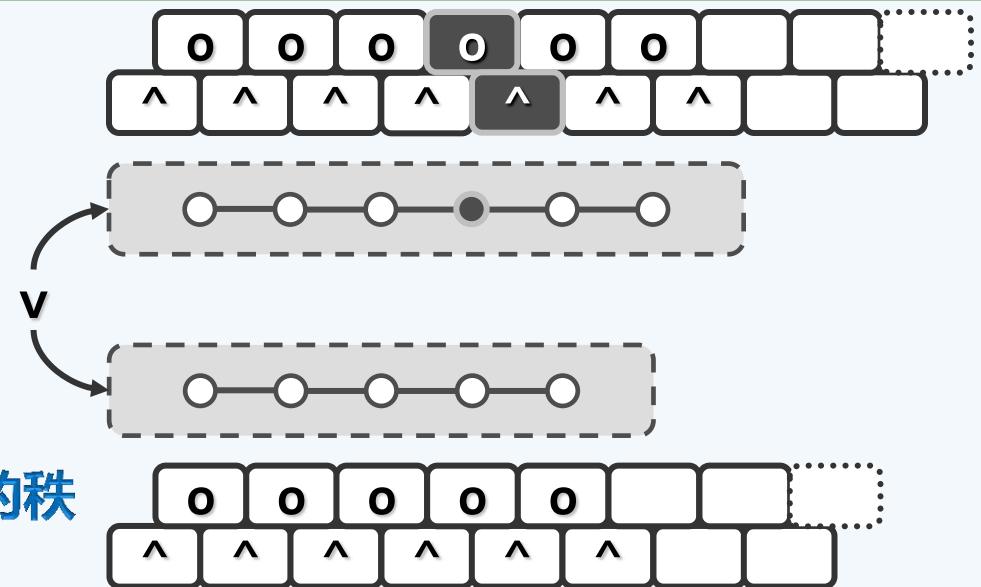
算法

❖ 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); //确定e在v中的秩
    if ( v->child[0] ) { //若v非叶子，则
        BTNodePosi(T) u = v->child[r + 1]; //在右子树中一直向左，即可
        while ( u->child[0] ) u = u->child[0]; //找到e的后继（必属于某叶节点）
        v->key[r] = u->key[0]; v = u; r = 0; //并与之交换位置
    } //至此，v必然位于最底层，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
}

```



旋转

◆ 节点 V 下溢时，必恰好包含： $\lceil m/2 \rceil - 2$ 个关键码 + $\lceil m/2 \rceil - 1$ 个分支

◆ 视其左、右兄弟 L 、 R 所含关键码的数目，可分三种情况处理

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

将关键码 y 从 P 移至 V 中（作为最小关键码）

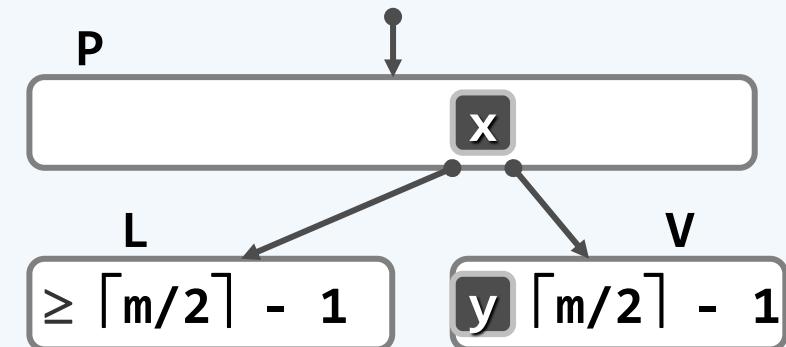
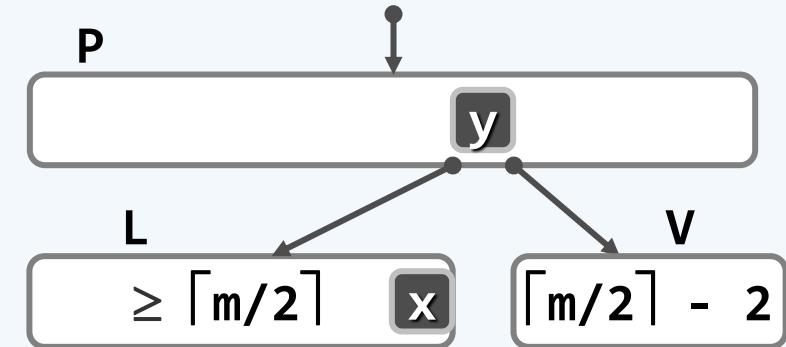
将关键码 x 从 L 移至 P 中（取代其中原关键码 y ）

◆ 如此旋转之后，局部乃至整树都重新满足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

完全对称



合并

3) $\lfloor L \rfloor$ 和 $\lfloor R \rfloor$ 或者不存在，或者所含的关键码均不足 $\lceil m/2 \rceil$ 个

注意， $\lfloor L \rfloor$ 和 $\lfloor R \rfloor$ 仍必有其一，且恰含 $\lceil m/2 \rceil - 1$ 个关键码（不妨以 $\lfloor L \rfloor$ 为例）

❖ 从 P 中抽出介于 $\lfloor L \rfloor$ 和 $\lfloor V \rfloor$ 之间的关键码 y

通过 y 做粘接，以 $\lfloor L \rfloor$ 和 $\lfloor V \rfloor$ 合成一个节点

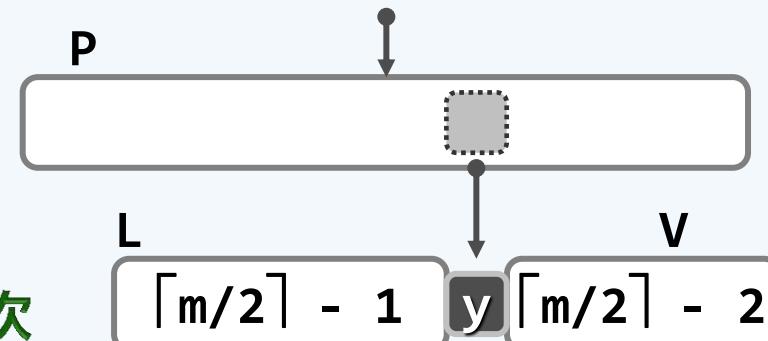
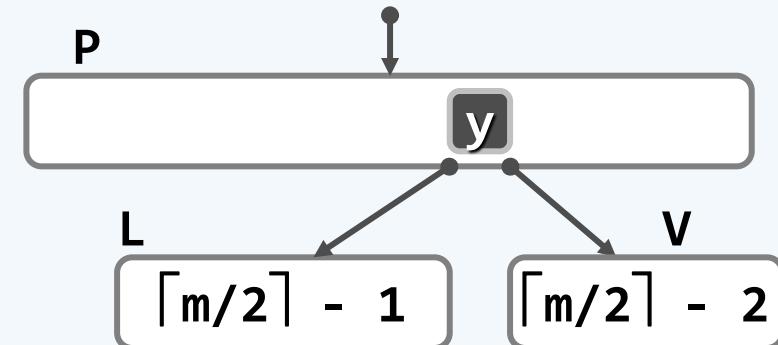
同时合并此前 y 的孩子引用

❖ 如此合并之后，原高度处的下溢得以修复

但可能导致更高处的 P 下溢

此时，大可套用前法，继续旋转或合并

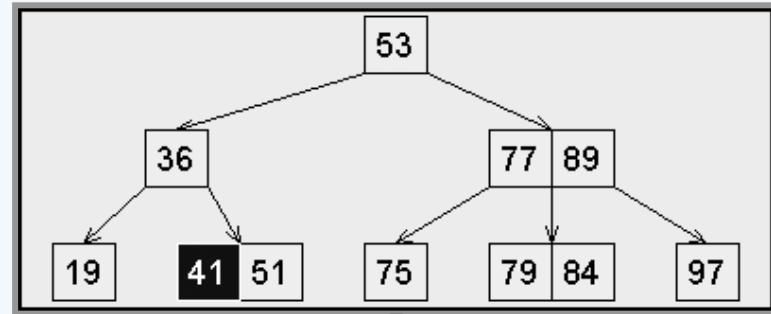
❖ 下溢可能持续发生，并逐层向上传播；但至多不过 $O(h)$ 次



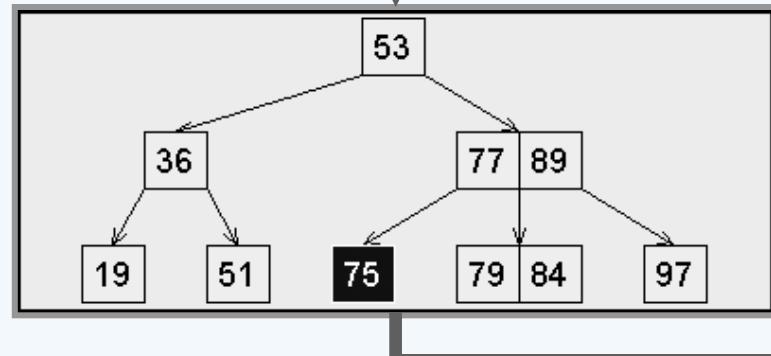
实例：底层节点

❖ 2-3-树

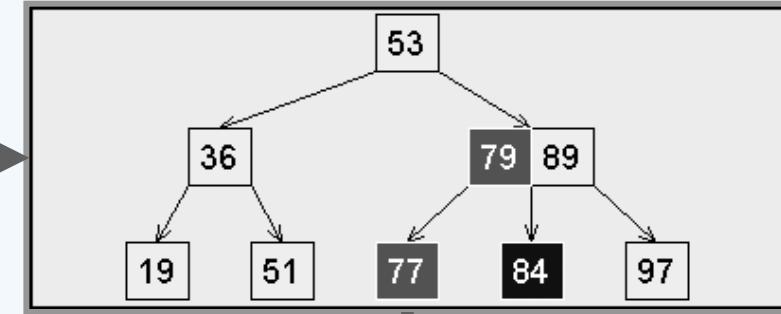
53 97 36 89 41 75 19 84 77 79 51



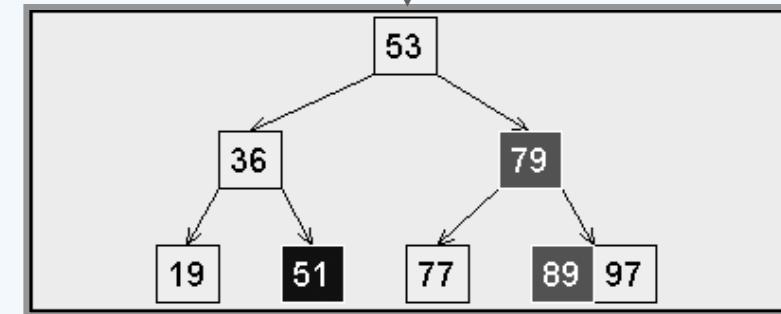
`remove(41)` //直接删除



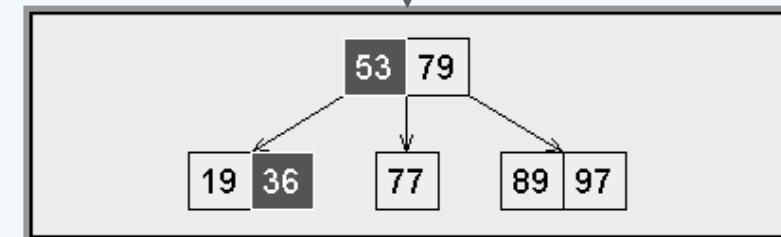
`remove(75)` //旋转



`remove(84)` //单次合并



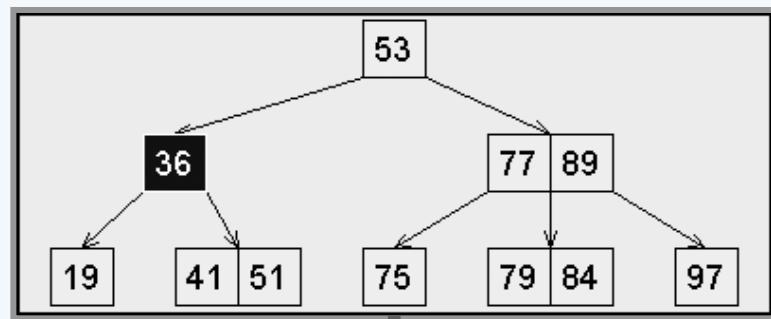
`remove(51)` //多次合并



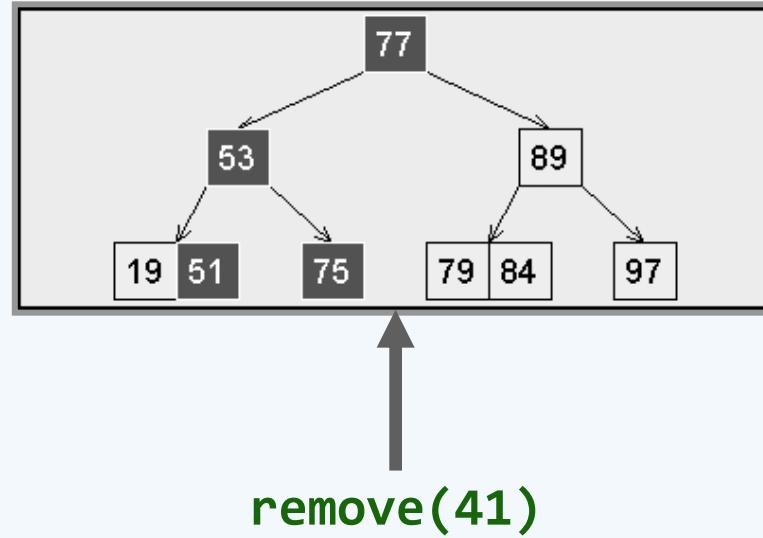
实例：非底层节点

❖ 2-3-树

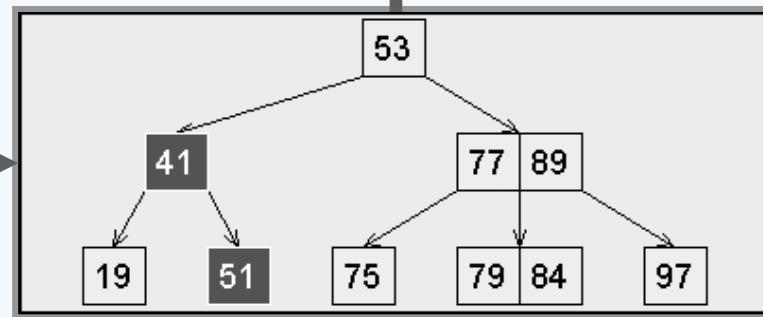
53 97 36 89 41 75 19 84 77 79 51



remove(36)



remove(41)



下溢修复

```

❖ template <typename T> void BTree<T>::solveUnderflow( BTNodePosi(T) v ) {
    if ( (_order + 1) / 2 <= v->child.size() ) return; //递归基：v并未下溢
    BTNodePosi(T) p = v->parent; if ( !p ) { /* 递归基：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
    solveUnderflow( p ); //上升一层，继续分裂——至多递归O(logn)层——典型尾递归
    return;
}

```

下溢修复：旋转

❖ 情况1：向左兄弟借关键码——情况2完全对称

❖ if ($\theta < r$) { //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 ((_order + 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的最大关键码转入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

}

下溢修复：合并

❖ 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 ... */

} else { /* 与右兄弟合并，完全对称 */ }

下溢修复：合并

```
❖ if (θ < r) { //与左兄弟合并  
    /* ..... */  
  
    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;  
    }  
  
    release(v); //释放v  
} else { /* 与右兄弟合并，完全对称 */ }
```

习题解析

❖ 举例说明，在最坏情况下，一次插入操作会引发 $\Omega(\log n)$ 次分裂

在连续的插入操作过程中，发生这种情况的概率有多大？

就连续意义而言，其间每次插入操作平均会引发多少次分裂？

❖ 就原理而言，与下溢修复一样，上溢修复即可做旋转，也可做分裂

试扩充 `BTree::solveOverflow()` 接口，加入这种策略

这一对称的策略，因何未被普遍采用？

习题解析

❖ B*-tree

从**独自分裂**到**联合分裂**

节点上溢后未必独自分裂，也可由**k**个饱和的兄弟**均摊**新关键码

得到**k + 1**个相邻节点，各含有至少 **$\lfloor (m - 1) * k / (k + 1) \rfloor$** 个关键码

如此，可将空间使用率从**50%**提高至 **$k / (k + 1)$**

8. 高级搜索树

红黑树

一致性

As she looks at the blood on the snow,
she says to herself, "Oh, how I wish
that I had a daughter that had skin
WHITE as snow, lips RED as blood, and
hair BLACK as ebony".

邓俊辉

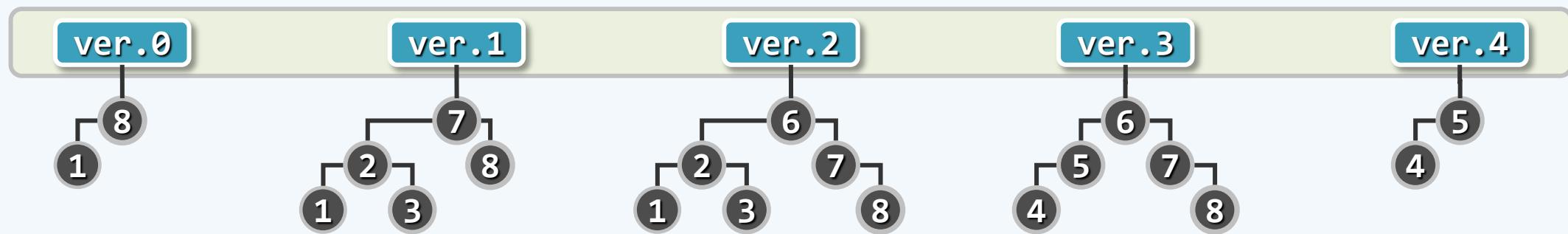
deng@tsinghua.edu.cn

一致性结构

❖ **Persistent structure** : 支持对**历史**版本的访问 //ephemeral

`T.search(ver, key); T.insert(ver, key); T.remove(ver, key)`

❖ 蛮力实现：每个版本独立保存；各版本入口自成一个搜索结构



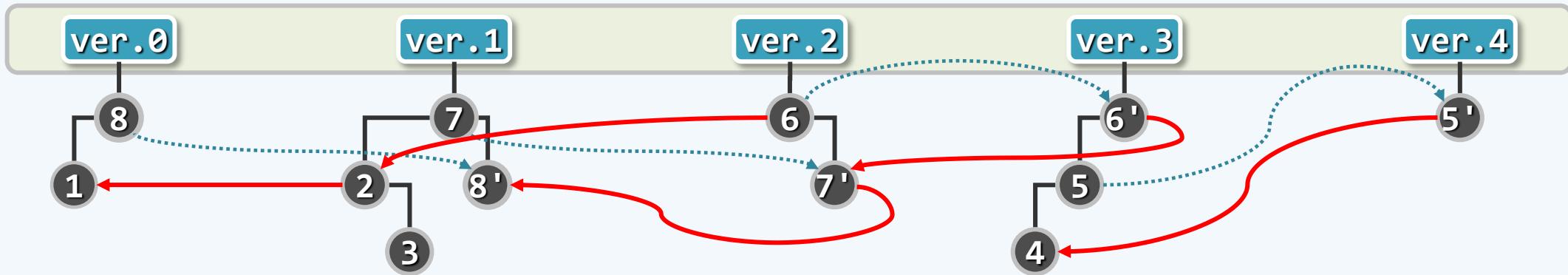
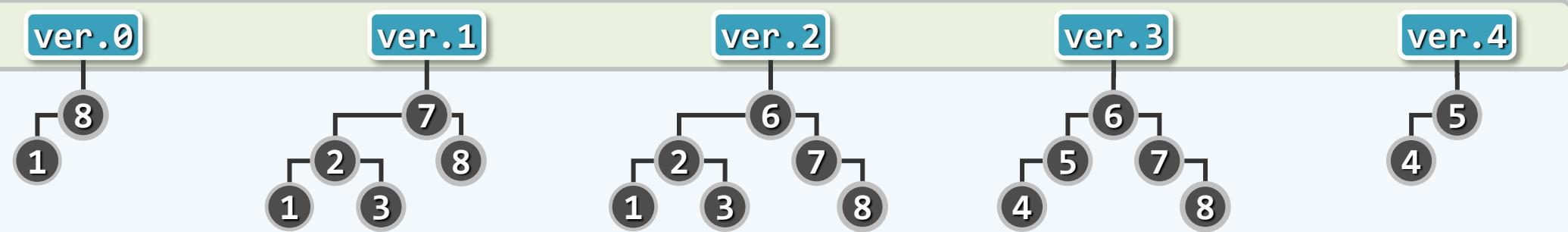
❖ 单次操作 $\mathcal{O}(\log h + \log n)$ ，累计 $\mathcal{O}(h*n)$ 时间/空间 //h = |history|

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

❖ 可以！为此需利用相邻版本之间的**关联性**...

$\mathcal{O}(1)$ 重构

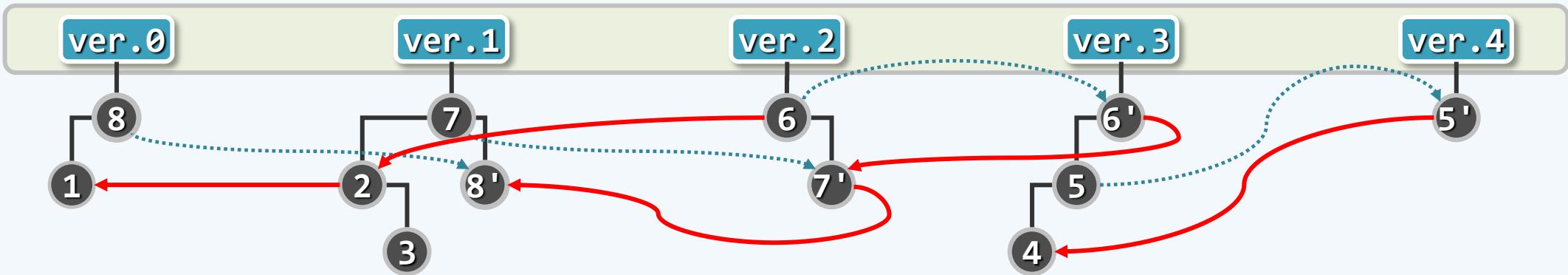
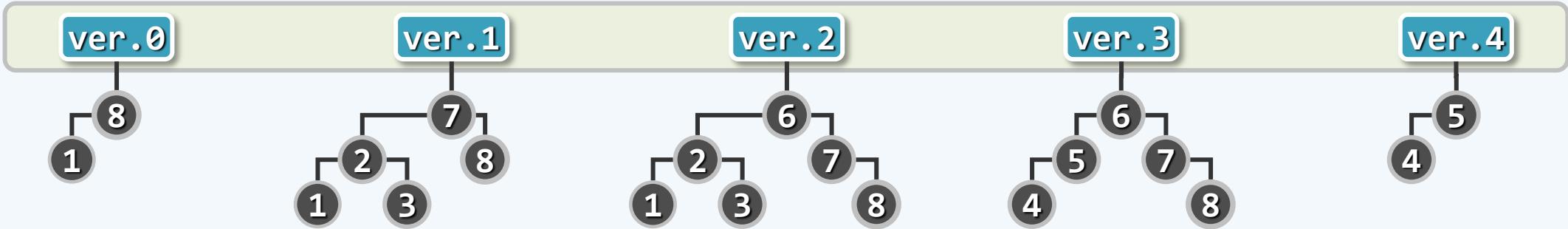
❖ 大量共享，少量更新：每个版本的新增复杂度，仅为 $\mathcal{O}(\log n)$



❖ 能否进一步提高，比如总体 $\mathcal{O}(n + h)$ 、单版本 $\mathcal{O}(1)$ ？可以！

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

8. 高级搜索树

红黑树

结构

崖前土黑没芝兰

邓俊辉

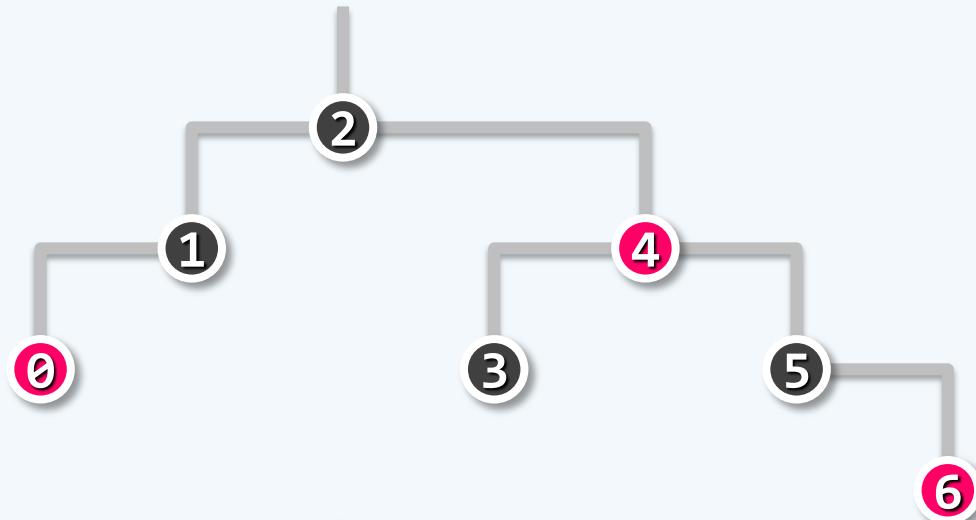
路畔泥红藤薜攀

deng@tsinghua.edu.cn

红与黑

❖ 1972, R. Bayer

symmetric binary B-tree



❖ 1978, L. Guibas & R. Sedgewick

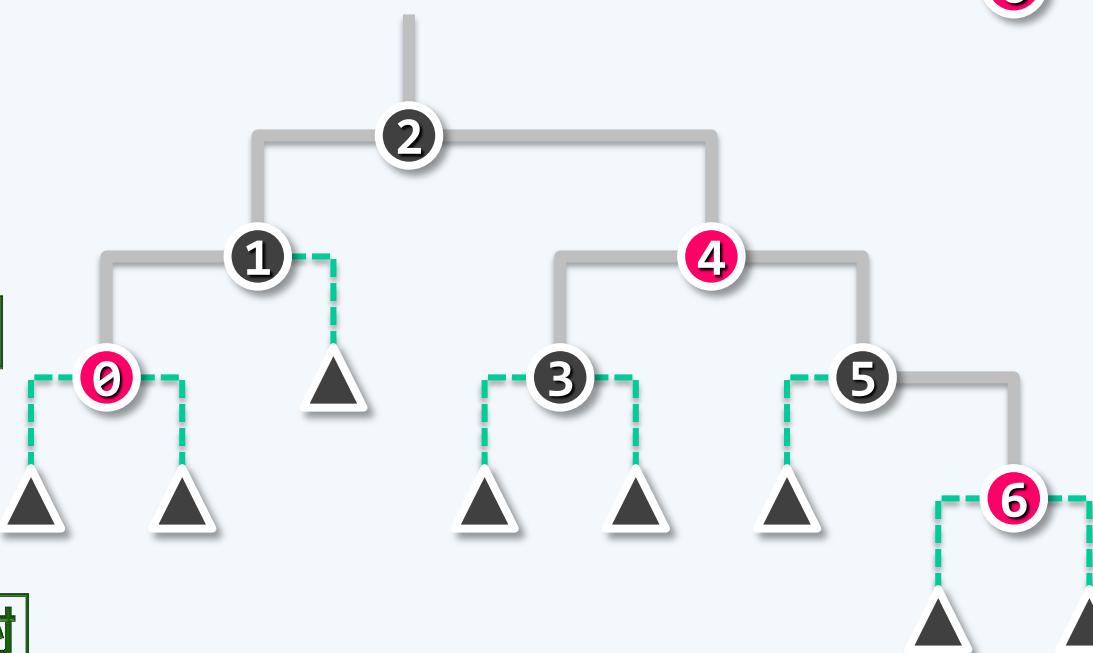
red-black tree

❖ 1982, H. Olivie

half-balanced binary search tree

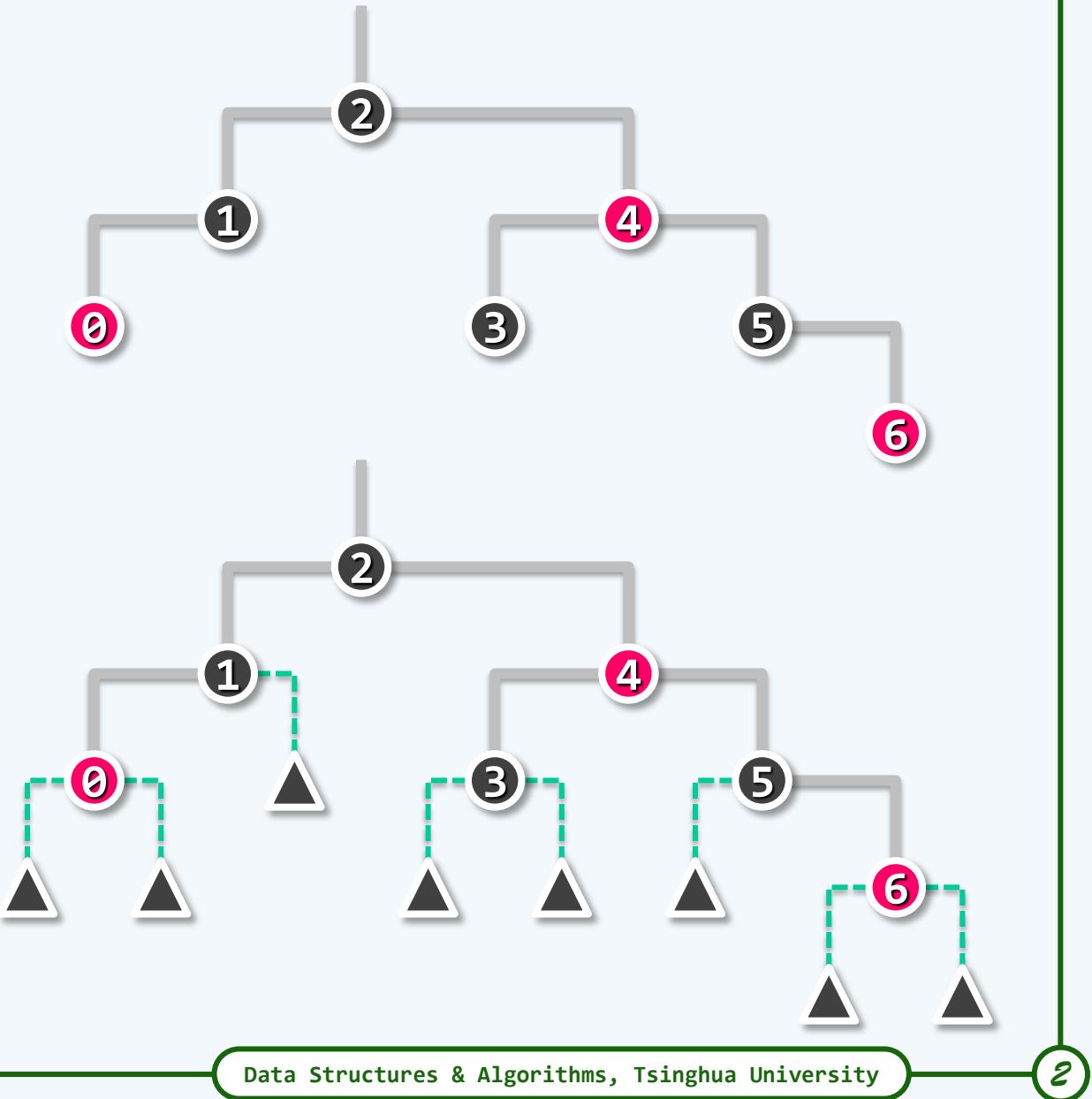
❖ 由红、黑两类节点组成的BST

统一增设外部节点NULL，使之成为真二叉树



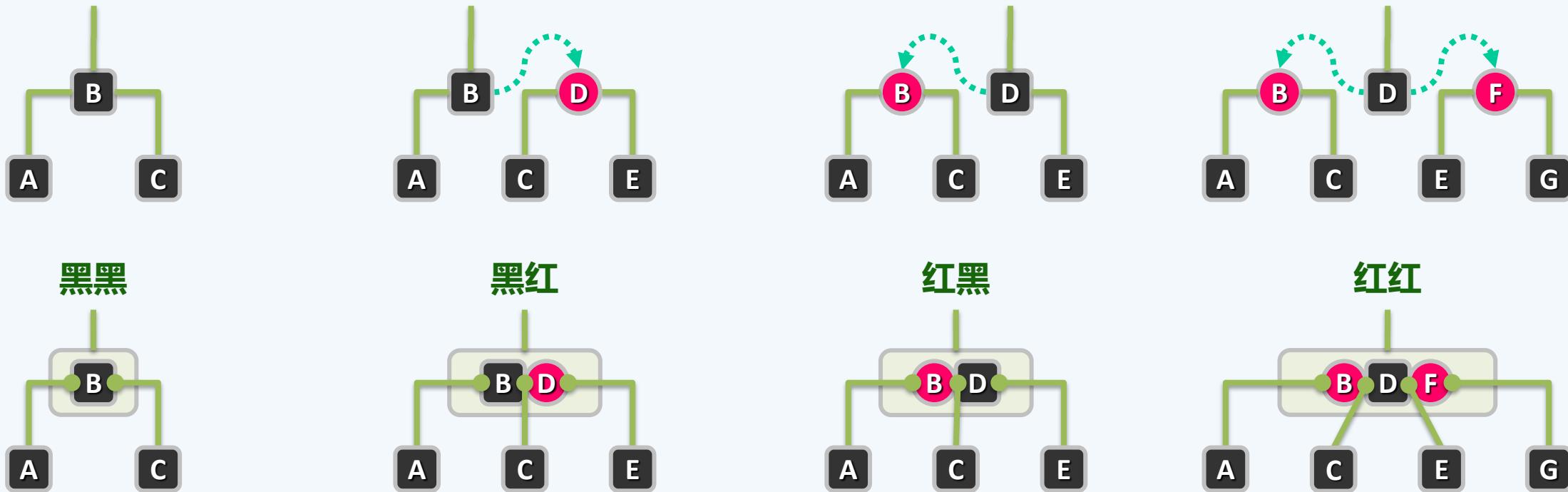
规则

- 1) 树根：必为黑色
 - 2) 外部节点：均为黑色
 - 3) 其余节点：若为红，则只能有黑孩子
//红之子、之父必黑
 - 4) 外部节点到根：途中黑节点数目相等
//黑深度
- ❖ 节点的颜色，只能显示地记录？
 - ❖ 以上定义颇为费解，有直观解释吗？
 - ❖ 如此定义的BST，也是BBST？



(2, 4)树 == 红黑树

- 提升各红节点，使之与其（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（关键码并合并为）超级节点...
- 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

- ❖ 由等价性，既然B-树是平衡的，红黑树自然也应是 //更严谨地...
- ❖ 定理：包含n个内部节点的红黑树T，高度 $h = \Theta(\log n)$ // $n + 1$ 个外部节点

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

❖ 若：T高度为h，红/黑高度为R/H

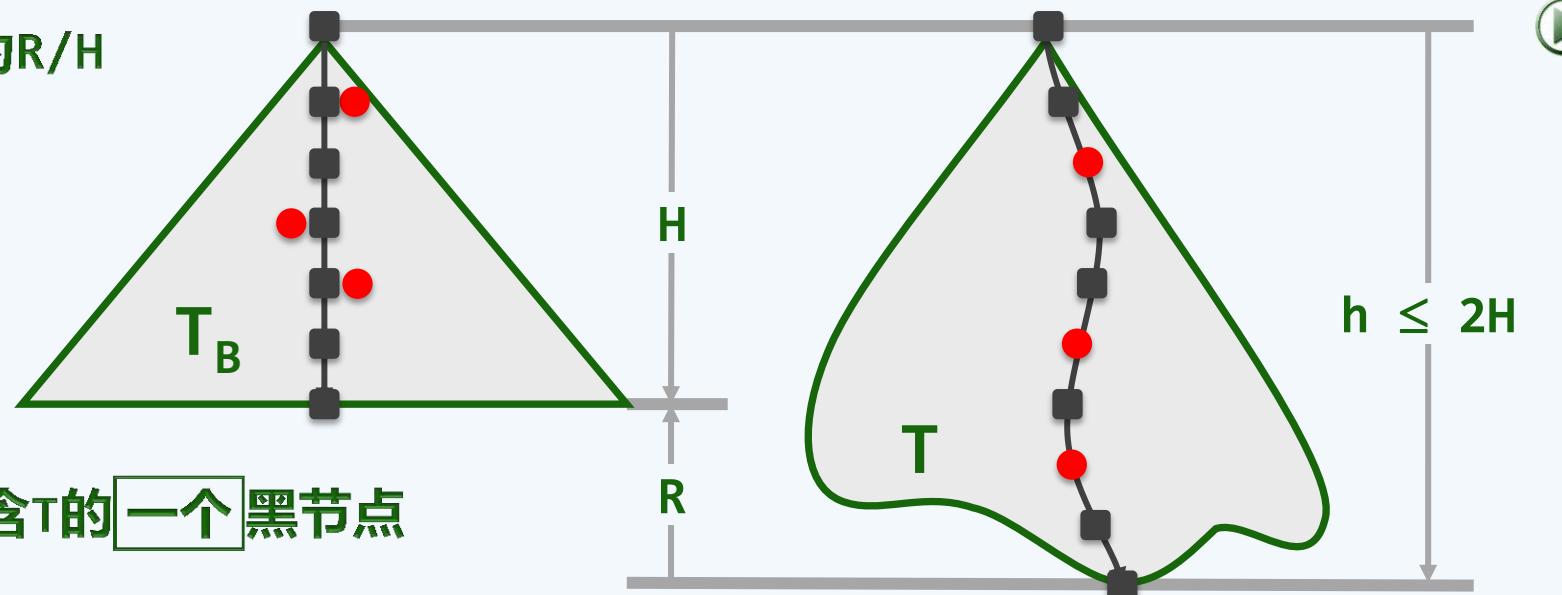
则： $h = R + H \leq 2H$

❖ 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，包含且仅包含T的一个黑节点

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



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
            int updateHeight( BinNodePosi(T) x ); //更新节点x的高度
};

❖ template <typename T> int RedBlack<T>::updateHeight( BinNodePosi(T) x ) {
    x->height = max( stature( x->lc ), stature( x->rc ) );
    if ( IsBlack( x ) ) x->height++; return x->height; //只计黑节点
}
```

8. 高级搜索树

红黑树

插入

莫赤匪狐，莫黑匪乌

惠而好我，携手同车

邓俊辉

deng@tsinghua.edu.cn

算法

❖ 按BST的常规算法，插入关键码 e // $x = \text{insert}(e)$ 必为末端节点

不妨设 x 的父亲 $p = x \rightarrow \text{parent}$ 存在 //否则，即平凡的首次插入

❖ 将 x 染红（除非它是根） // $x \rightarrow \text{color} = \text{isRoot}(x) ? \text{B} : \text{R}$

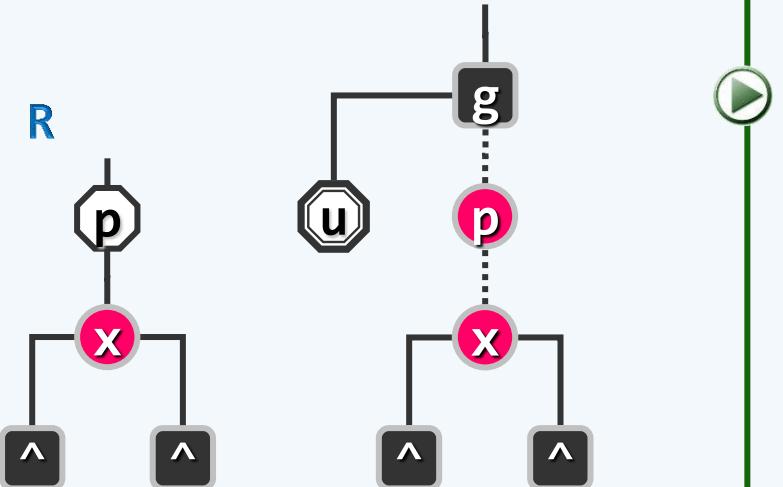
条件 1 + 2 + 4 依然满足；但 3 不见得，有可能...

❖ 双红 double-red // $p \rightarrow \text{color} == x \rightarrow \text{color} == \text{R}$

❖ 考查： x 的祖父 $g = p \rightarrow \text{parent}$ // $g != \text{null} \&& g \rightarrow \text{color} == \text{B}$

p 的兄弟 $u = p == g \rightarrow \text{lc} ? g \rightarrow \text{rc} : g \rightarrow \text{lc}$ //即 x 的叔父

❖ 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 -1  
  
    x = new BinNode<T>( e, _hot, NULL, NULL, -1 ); _size++;  
  
    // 如有必要，需做双红修正  
  
    solveDoubleRed( x );  
  
    // 返回插入的节点  
  
    return x ? x : _hot->parent;  
} // 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 ... */ }
}

```

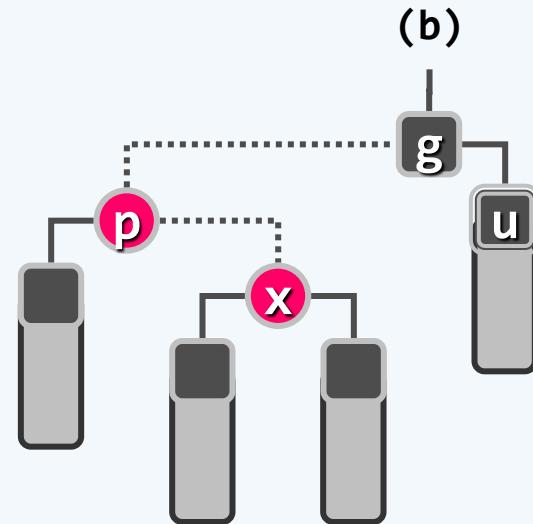
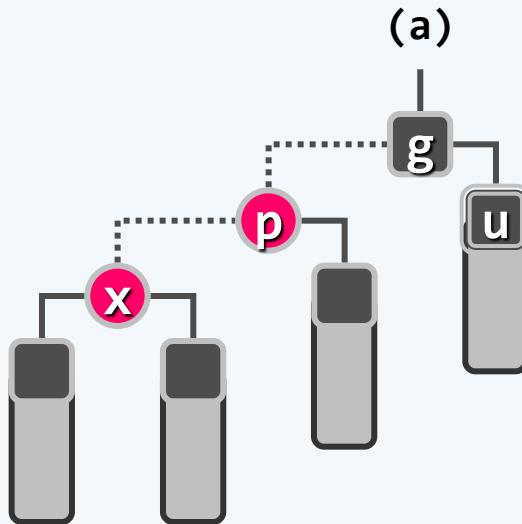
RR-1 : $u \rightarrow \text{color} == B$

❖ 此时：

x 、 p 、 g 的四个孩子

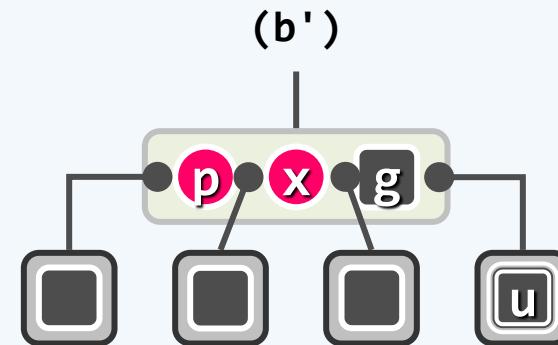
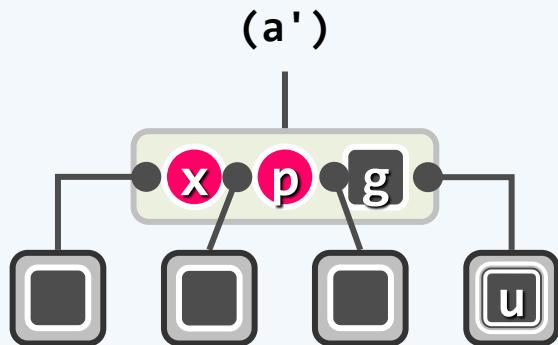
(可能是外部节点)

全为黑，且
黑高度相同



❖ 另两种对称情况

自行补充



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

1) 参照AVL树算法，做局部[3+4]重构

2) 染色：b转黑，a或c转红

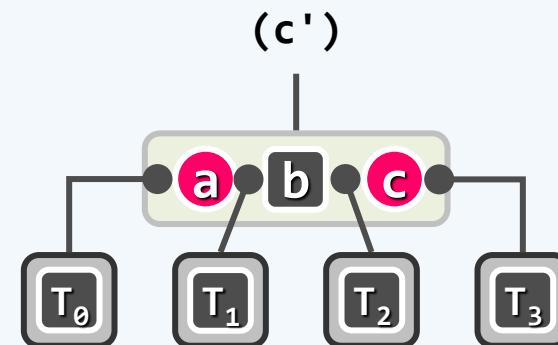
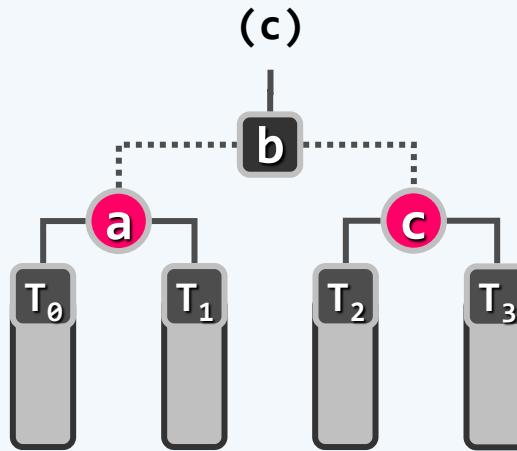
❖ 从B-树的角度，如何理解这一情况？

❖ 调整前之所以非法，是因为

- 在某个三叉节点中插入红关键码，使得
- 原黑关键码不再居中 // RRB 或 BRR，出现相邻的红关键码

❖ 调整之后的效果相当于 // B-树的拓扑结构不变，但

- 在新的四叉节点中，三个关键码的颜色改为RBR



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

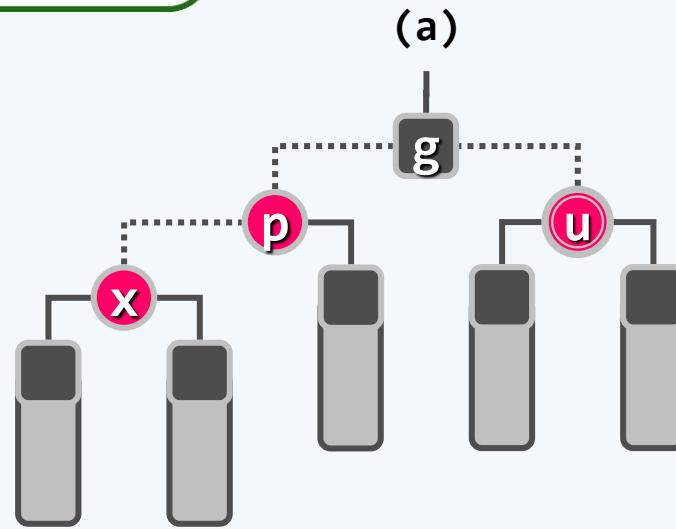
❖ 在B-树中，等效于

超级节点发生上溢

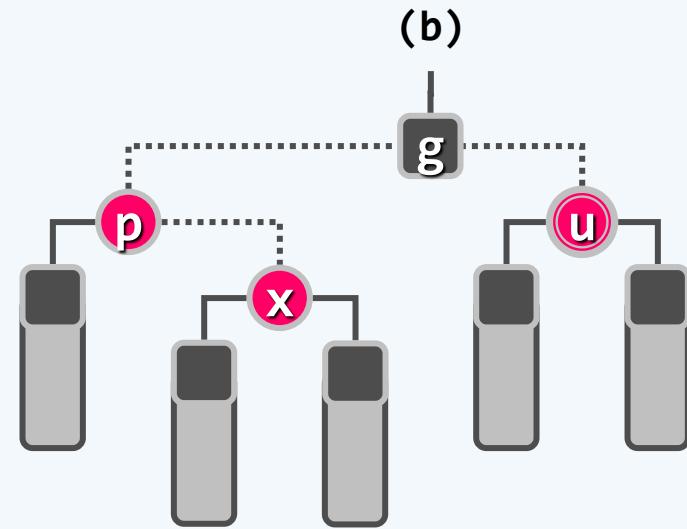
//另两种对称情况

//请自行补充

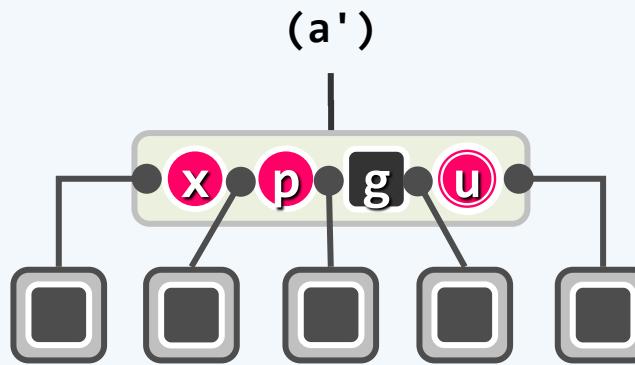
(a)



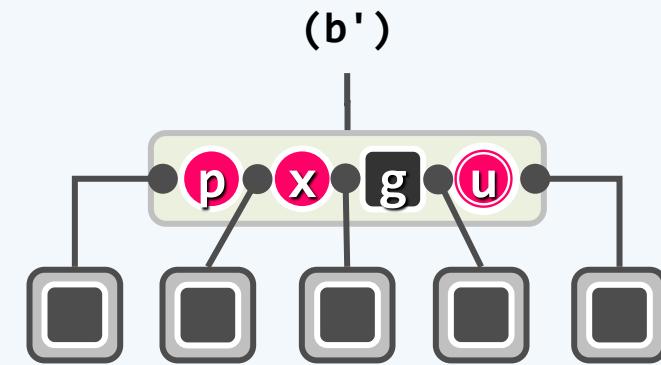
(b)



(a')



(b')



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

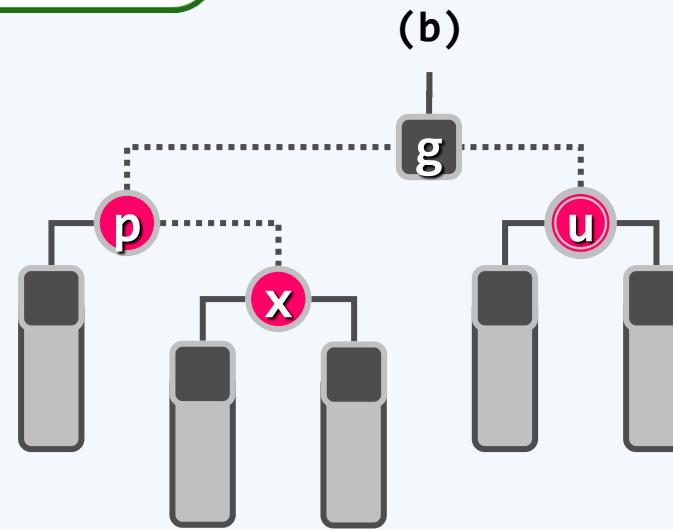
❖ p与u转黑，g转红

❖ 在B-树中，等效于

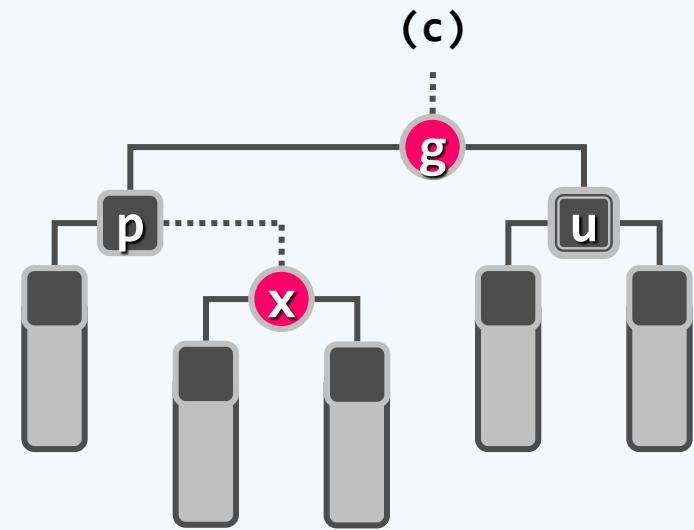
节点分裂

关键码g上升一层

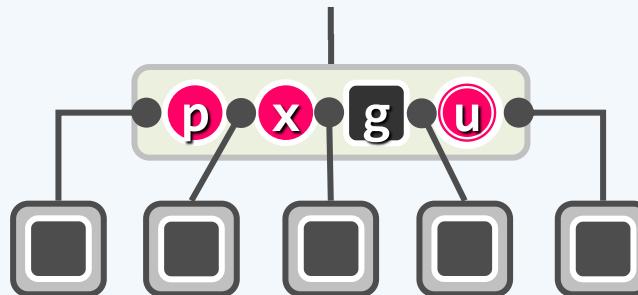
(b)



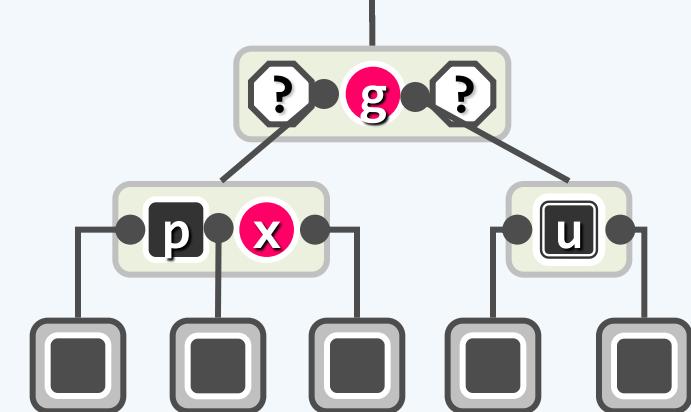
(c)



(b')



(c')



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

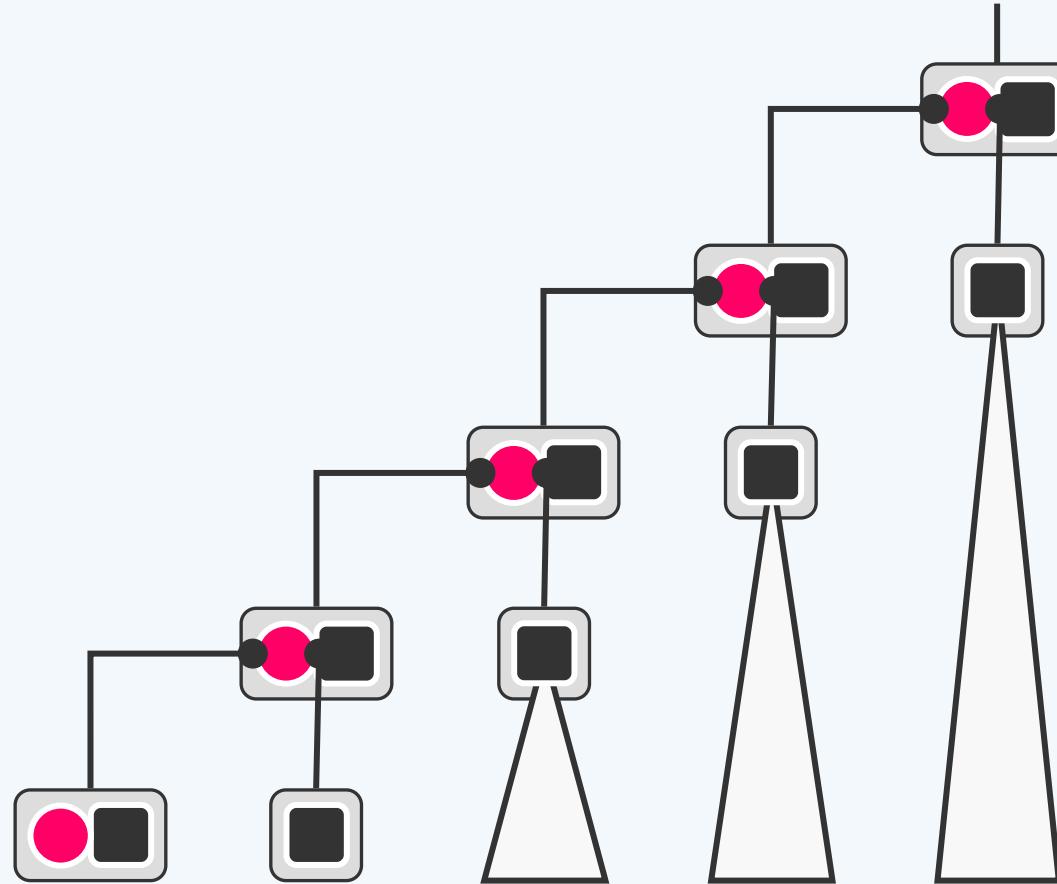
❖ 既然是分裂，也应有可能继续向上传递

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
1. 强行将 g 转为黑色
2. 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（或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  
  
        if ( ! IsRoot( *g ) ) g->color = RB_RED; //g若非根则转红  
        solveDoubleRed( g ); //继续调整g（类似于尾递归，可优化）  
    }  
}
```

复杂度

❖ 重构、染色均属常数时间的局部操作

故只需统计其总次数

❖ 红黑树的每一次插入操作

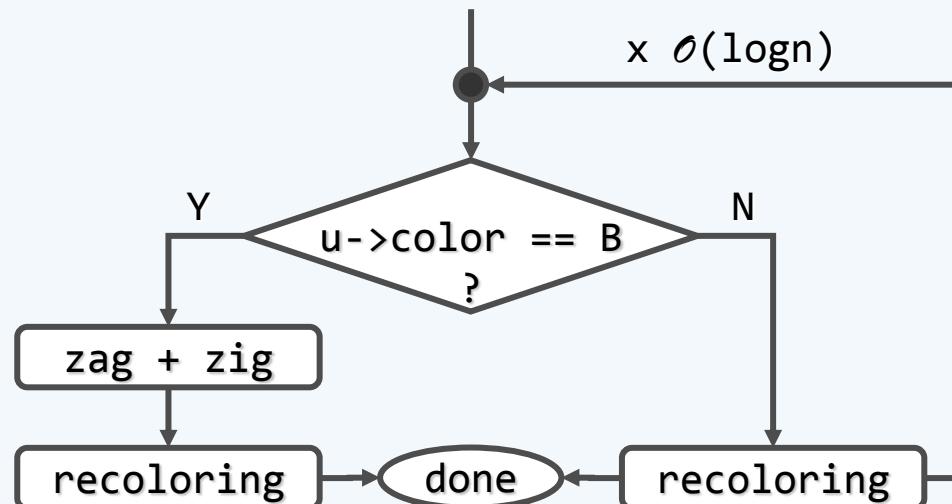
都可在 $\mathcal{O}(\log n)$ 时间内完成

❖ 其中至多做：

1. $\mathcal{O}(\log n)$ 次 节点染色

2. 一次 “3+4” 重构

情况	旋转次数	染色次数	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



8. 高级搜索树

红黑树

删除

邓俊辉

变白以为黑兮，倒上以为下

deng@tsinghua.edu.cn

算法

❖ 首先按照BST常规算法，执行

`r = removeAt(x, _hot)`

❖ x 由孩子 r 接替，另一孩子记作 w

w 首次必为NULL，但可能在随后的调整过程中逐层上升

——如何统一？

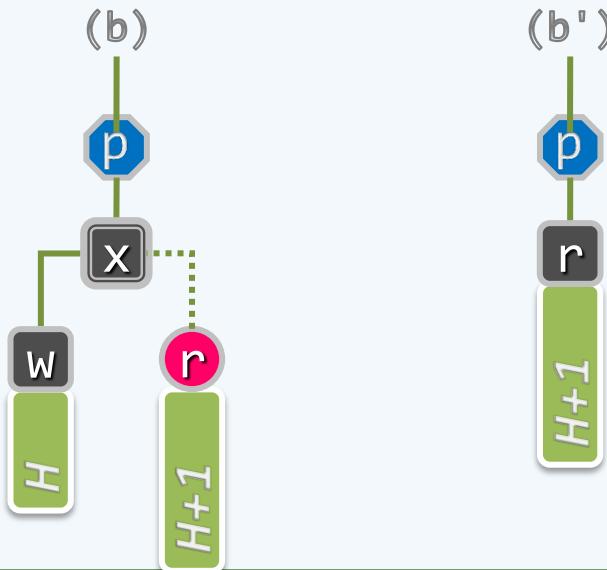
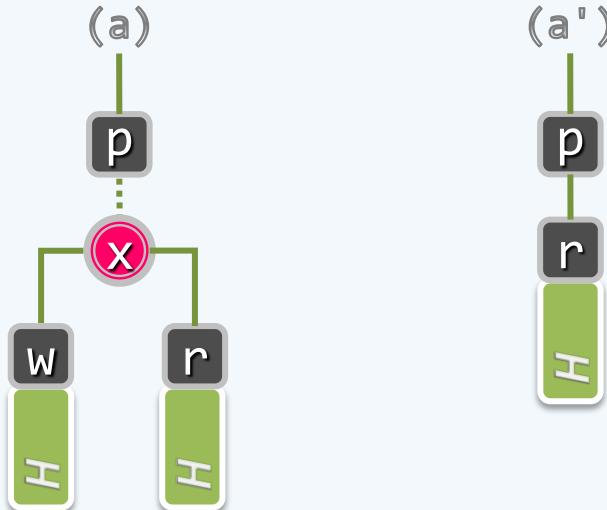
❖ 更通用、更便捷的等效理解：

w 为一棵与 r 等高的子红黑树，且随 x 一并被摘除

❖ 条件①和②依然满足；但③和④不见得

❖ 在原树中，考查 x 与 r ...

❖ 若二者之一为红，则③和④不难满足 //删除遂告完成！



算法

❖ 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 // 高效地...

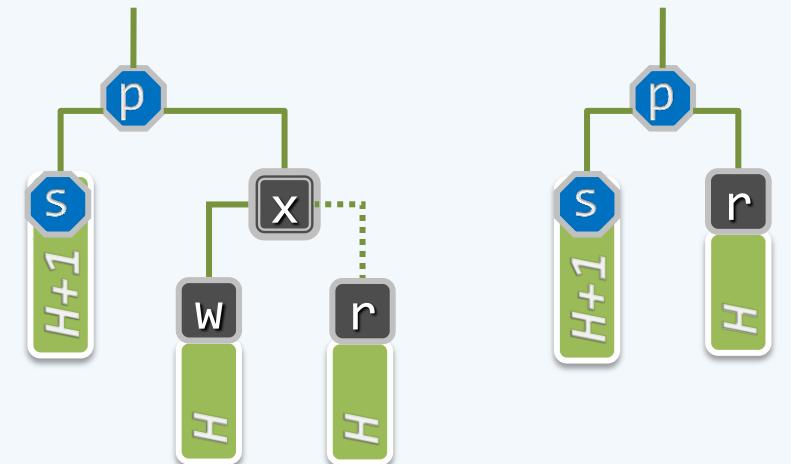
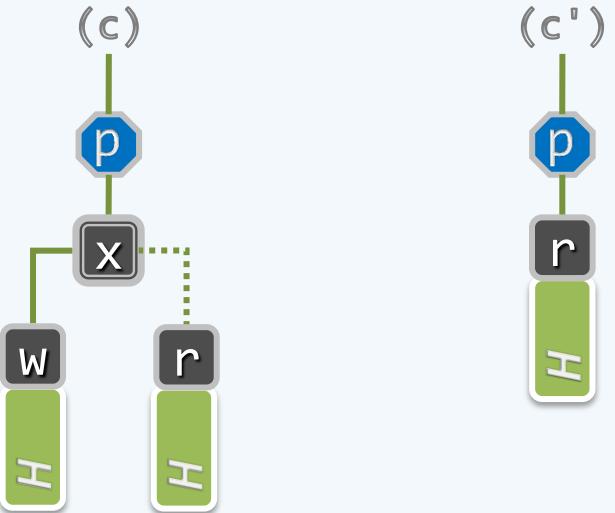
原B-树中 x 所属节点 下溢

❖ 在新树中，考查

r 的父亲 $p = r \rightarrow \text{parent}$ // 亦即原树中 x 的父亲

r 的兄弟 $s = [r == p \rightarrow \text{lc}] ? p \rightarrow \text{rc} : p \rightarrow \text{lc}$

❖ 以下分四种情况处理...



实现

```
❖ 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
```

实现

❖ // 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;
```

```
}
```

双黑修正

```

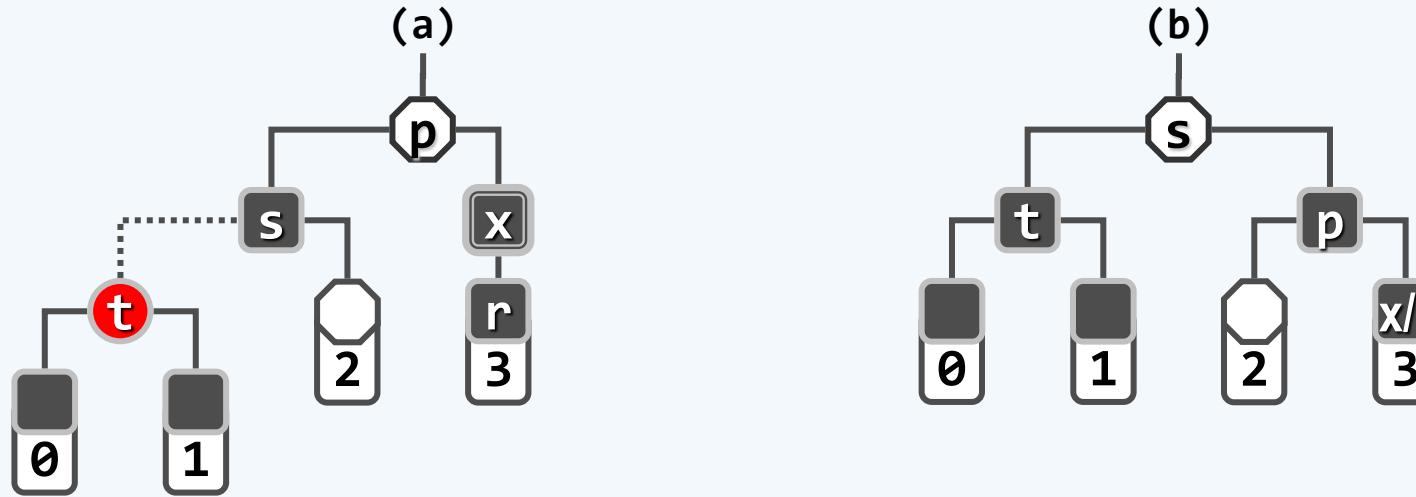
❖ template <typename T> void RedBlack<T>::solveDoubleBlack( BinNodePosi(T) r ) {
    BinNodePosi(T) p = r ? r->parent : _hot; if ( !p ) return; //r的父亲
    BinNodePosi(T) s = (r == p->lc) ? p->rc : p->lc; //r的兄弟
    if ( IsBlack( s ) ) { //兄弟s为黑
        BinNodePosi(T) t = NULL; //s的红孩子（若左、右孩子皆红，左者优先；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**、**s**、**p**重命名为**a**、**b**、**c**

r保持黑；**a**和**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**

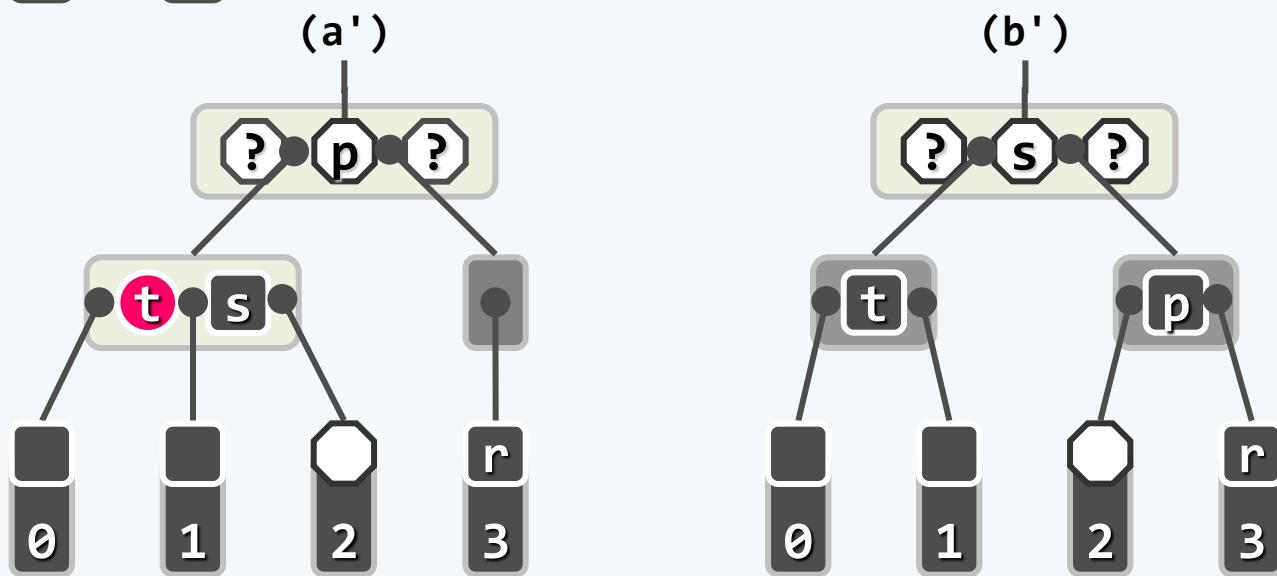
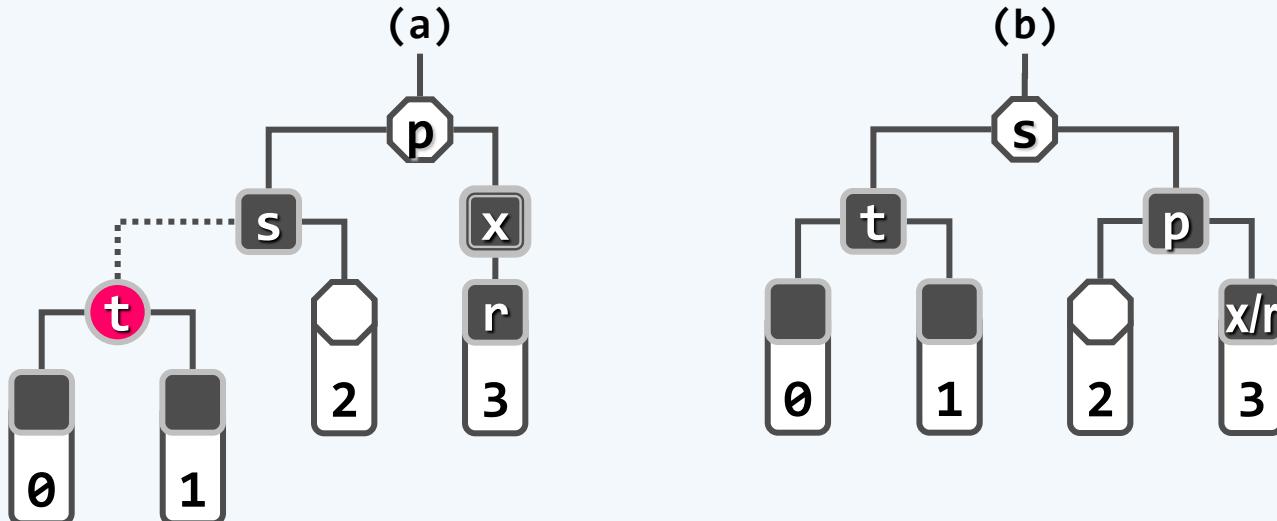
❖ 通过关键码的 **旋转**

消除超级节点的下溢

❖ **问号节点**

可同时存在

颜色不定



BB-1 : 实现

```

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

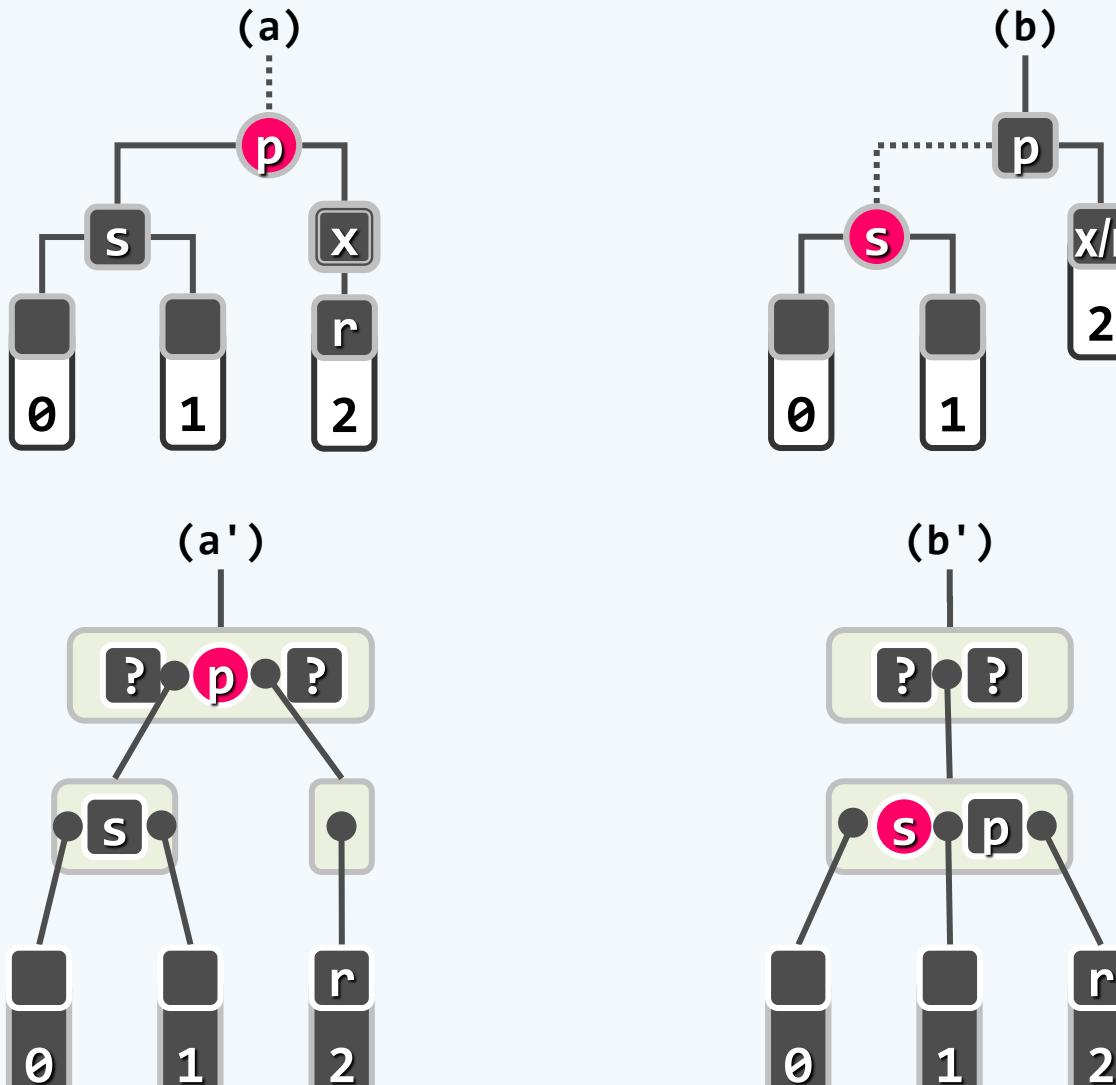
    if ( t ) { //黑s有红孩子：BB-1
        RBColor oldColor = p->color; //备份p颜色，并对t、父亲、祖父
        BinNodePosi(T) b = FromParentTo( *p ) = rotateAt( t ); //旋转
        if ( HasLChild( *b ) ) //新子树左子染黑后
            { b->lC->color = RB_BLACK; updateHeight( b->lC ); }
        if ( HasRChild( *b ) ) //新子树右子染黑后
            { 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 ... */ }

```

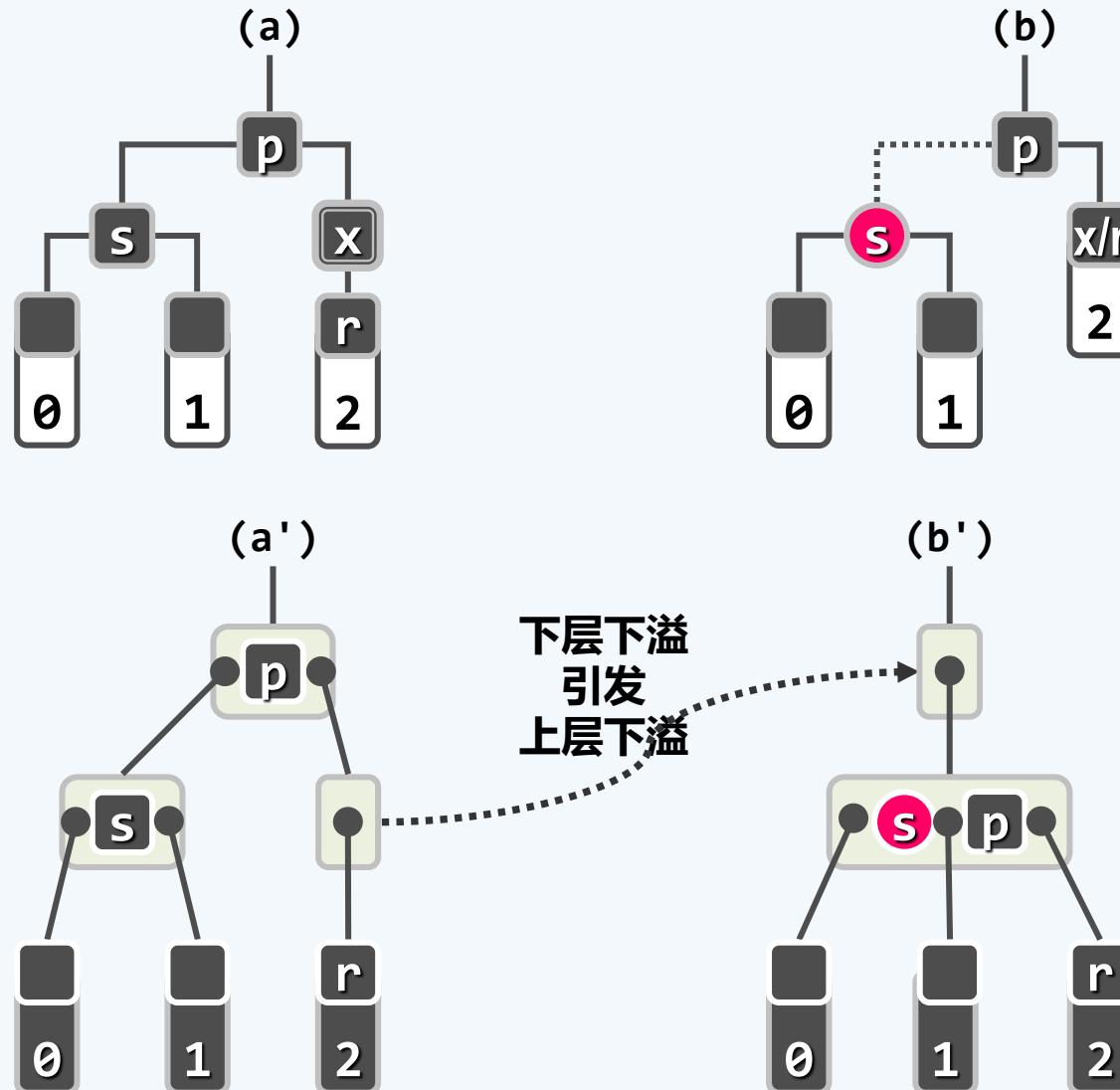
BB-2R : [s]为黑，且两个孩子均为黑；[p]为红

- ❖ 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（问号）关键码必为黑色
有且仅有[一个]



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
- ❖ 合并之前, **p和s均对应于单关键码节点**
- ❖ 失去关键码p后
上层节点必然继而下溢
- ❖ 好在可继续分情况处理
高度递增, 至多 **$O(\log n)$** 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**)

❖ $\text{zag}(p)$ 或 $\text{zig}(p)$ ；红**s**转黑，黑**p**转红

❖ 黑高度依然异常，但...

❖ **r**有了一个新**黑**兄弟**s'**

故转化为前述情况，而且...

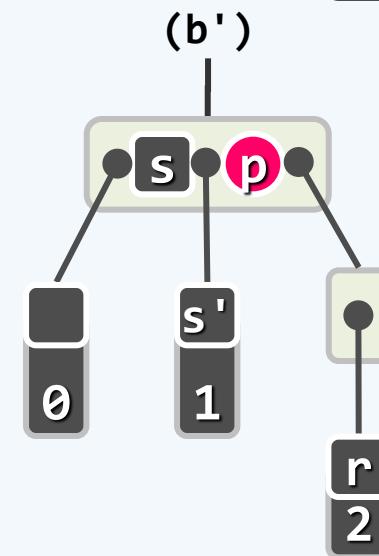
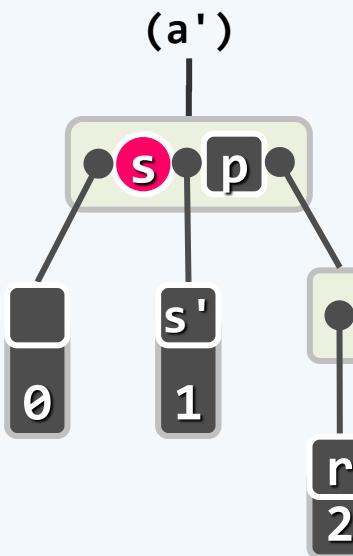
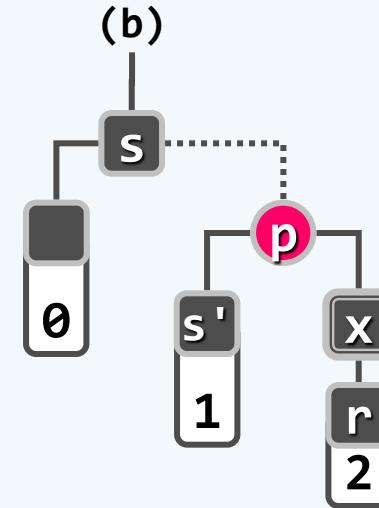
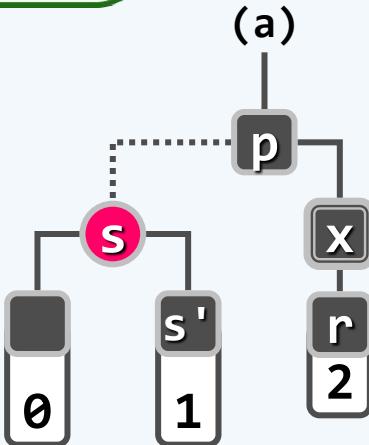
❖ 既然**p**已转**红**，接下来

绝不会是情况**BB-2B**

而只能是**BB-1**或**BB-2R**

❖ 于是，再经**一轮**调整之后

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
}

```

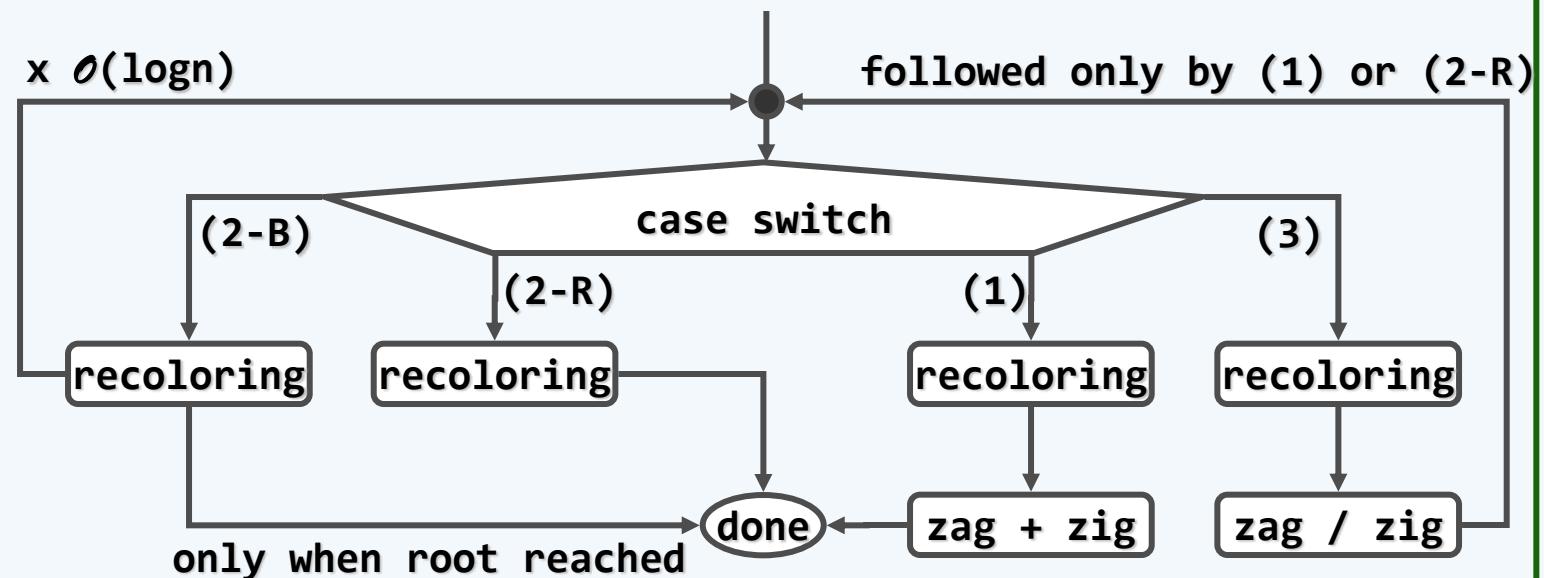
复杂度

- ❖ 红黑树的每一删除操作都可在 $\mathcal{O}(\log n)$ 时间内完成

- ❖ 其中，至多做

1. $\mathcal{O}(\log n)$ 次重染色
2. 一次“3+4”重构
3. 一次单旋

情况	旋转次数	染色次数	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	必然再次双黑 但将上升一层
(3) 红s	1	2	转为(1)或(2R)



Advanced Balanced Search Tree

Range Query

1-Dimensional Range Query

邓俊辉

deng@tsinghua.edu.cn

Query Types

❖ Let $P = \{ p_1, \dots, p_n \}$ be a set of n points on the x -axis



❖ For any given interval $I = (x_1, x_2)$ //left open & right closed

- **Counting** : how many points of P lies in the interval?
- **Reporting** : enumerate all points in $I \cap P$ (if not empty)

Online Query

- ❖ P is **fixed** while I is **randomly and repeatedly** given



- ❖ How to **preprocess** P into a certain data structure s.t.

the queries can be answered efficiently?

Advanced Balanced Search Tree

Range Query

Brute Force

When all else fails, try brute-force.

邓俊辉

- Anonymous

deng@tsinghua.edu.cn

Brute-force

- ❖ For each point p of P , test if $p \in (x_1, x_2]$
- ❖ Thus each query can be answered in $\Theta(n)$ time



- ❖ Can we do it faster?

It seems we can't, for ...

Lower Bound?

❖ In the worst case,

the interval contains up to $\Theta(n)$ points,

which need $\Theta(n)$ time to enumerate



❖ However, how if we

- ignore the time for enumerating and

- count only the searching time?

Geometric Range Search

Range Query

Binary Search

邓俊辉

deng@tsinghua.edu.cn

Preprocessing



❖ **Sort** all points into a sorted vector and

add an extra stencil point $p[0] = -\infty$

Binary Search



❖ For any interval $I = (x_1, x_2]$

- Find $t = \text{search}(x_2) = \max\{ i \mid p[i] \leq x_2 \} // O(\log n)$
 - Traverse the vector **backward** from $p[t]$ and report each point $// O(r)$
- until escaping from I at point $p[s]$
- return $r = t - s // \text{output size}$

Advanced Balanced Search Tree

Range Query

Output Sensitivity

邓俊辉

deng@tsinghua.edu.cn

❖ A **enumerating** query can be answered in $\mathcal{O}(r + \log n)$ time

⦿ $p[s]$ can also be found by binary search in $\mathcal{O}(\log n)$ time

∴ Hence for **counting** query, $\mathcal{O}(\log n)$ time is enough //independent to r



❖ Can this simple strategy be extended to **planar** range query?

TTBOMK, unfortunately, no!

Advanced Balanced Search Tree

Range Query

- Planar Range Query

邓俊辉

deng@tsinghua.edu.cn

❖ Let $P = \{ p_1, \dots, p_n \}$ be a planar set

❖ Given $R = [x_1, x_2] \times [y_1, y_2]$

- **Counting** : $|P \cap R| = ?$

- **Reporting** : $P \cap R = ?$

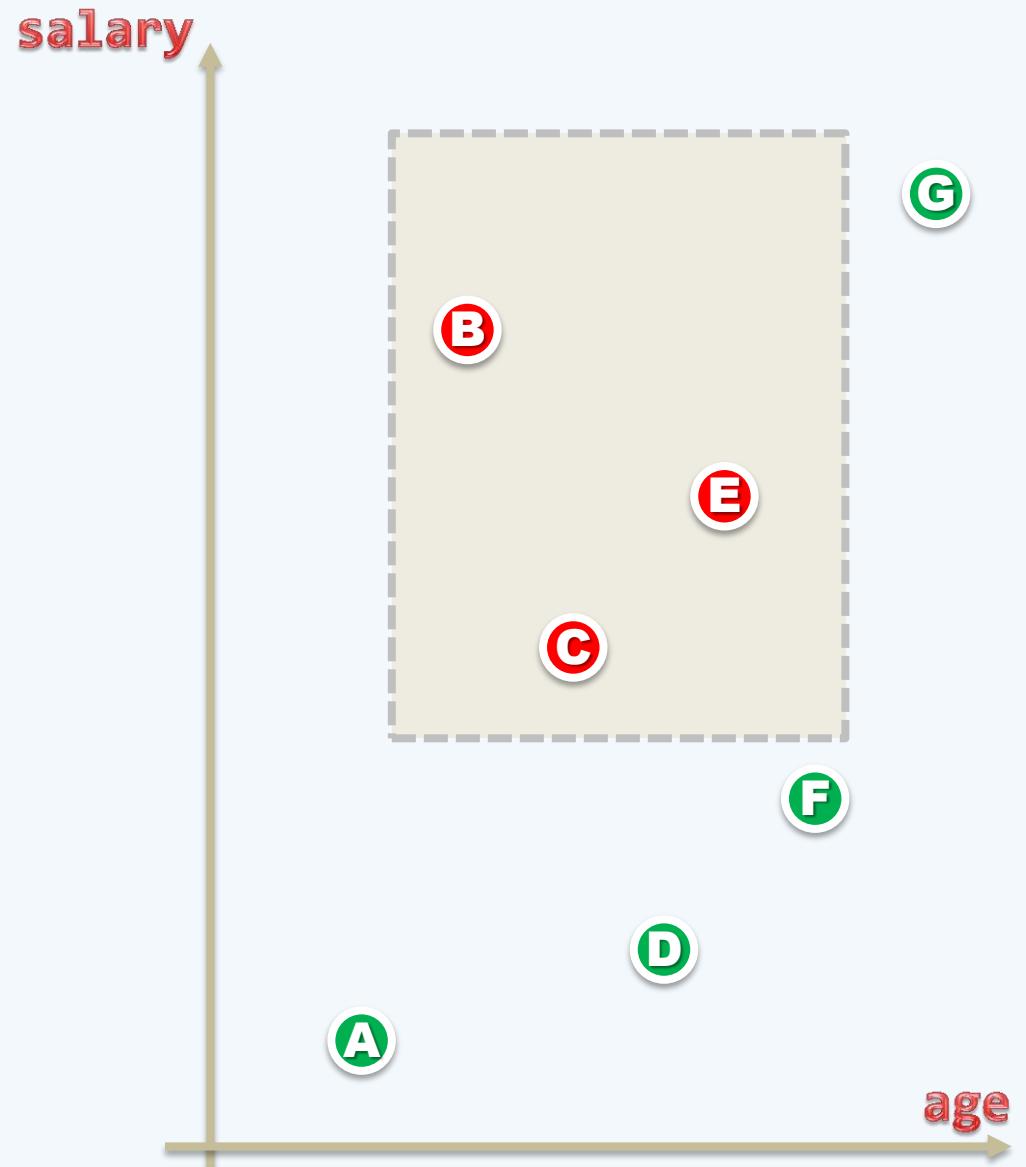
❖ Binary search

doesn't help this kind of query

❖ You might consider to

expand the counting method using

the **Inclusion-Exclusion Principle**



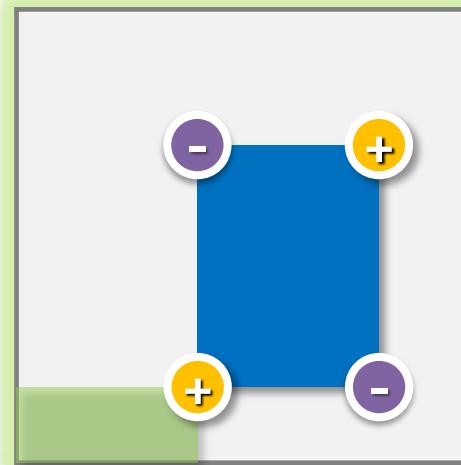
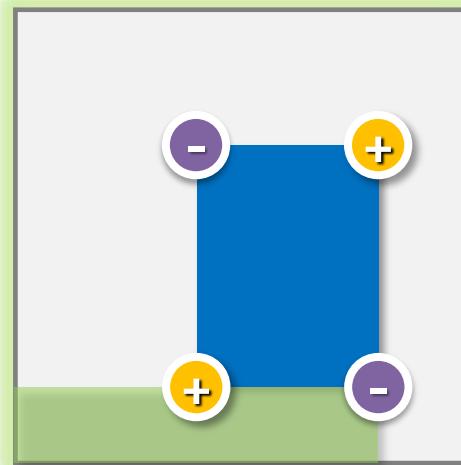
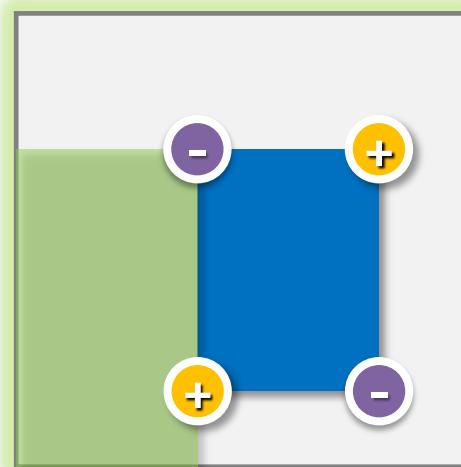
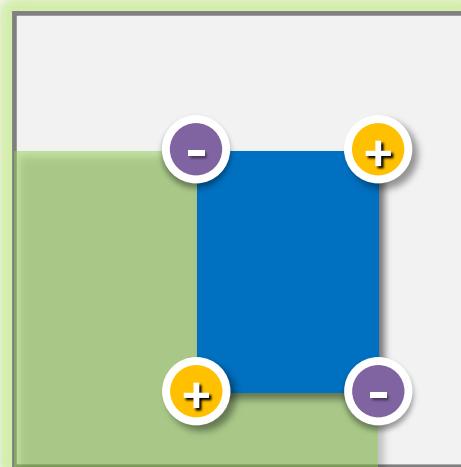
Preprocessing

- ❖ For each point (x, y) , let

$$n(x, y) = | \{(\theta, x] \times (\theta, y] \cap P\} |$$

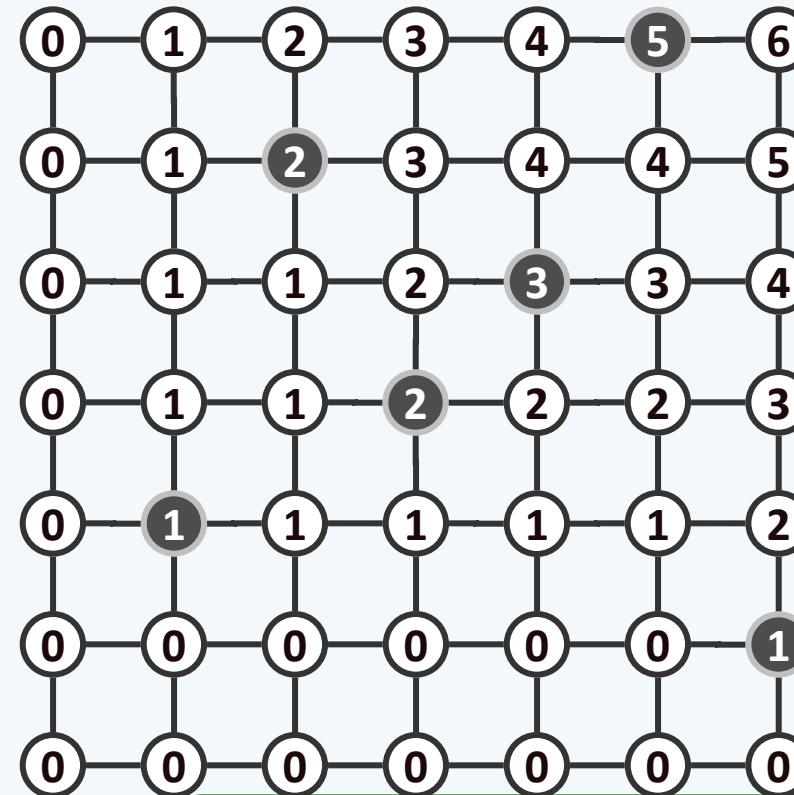
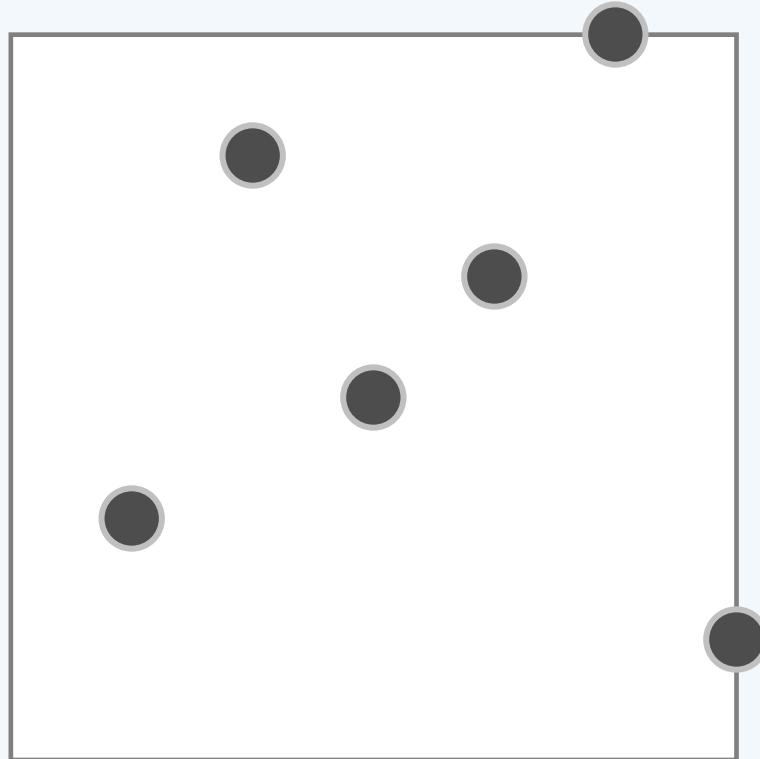
- ❖ This requires

$\Theta(n^2)$ time/space



Domination

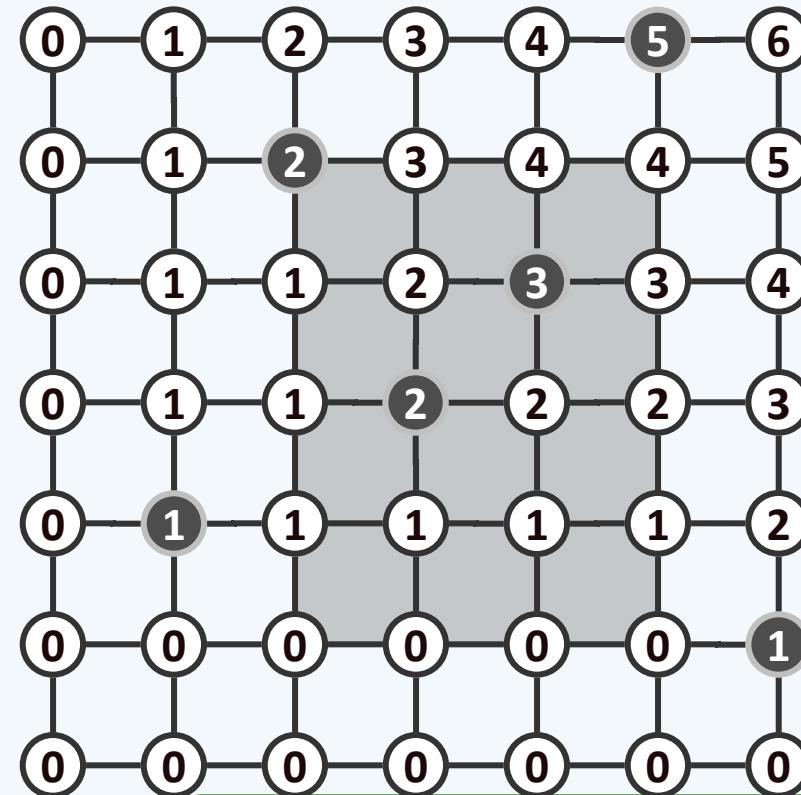
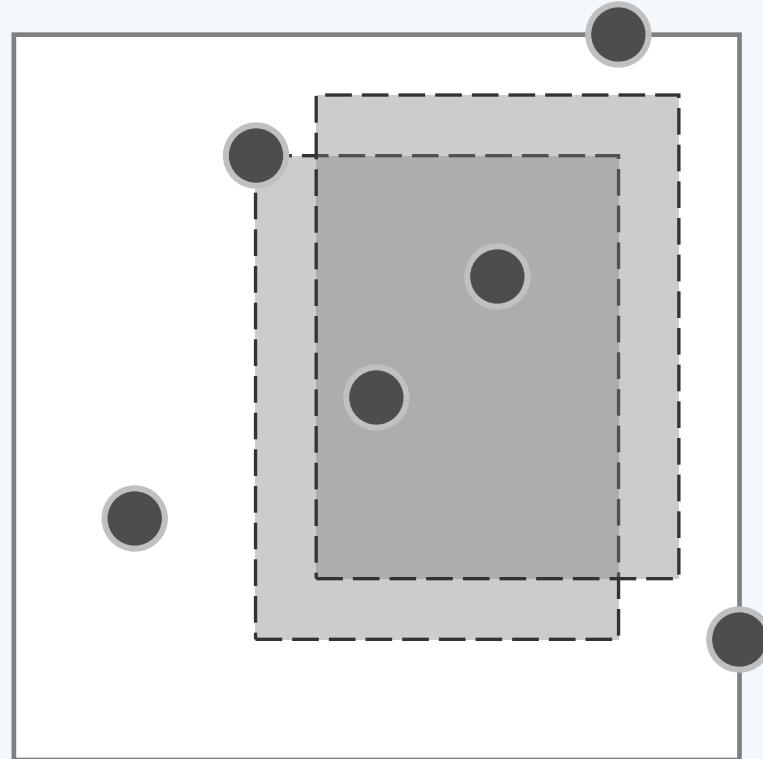
- ❖ A point (u, v) is called to be **dominated** by point (x, y) if
 $u \leq x$ and $v \leq y$



Inclusion-Exclusion Principle

❖ Then for any rectangular range $R = (x_1, x_2] \times (y_1, y_2]$, we have

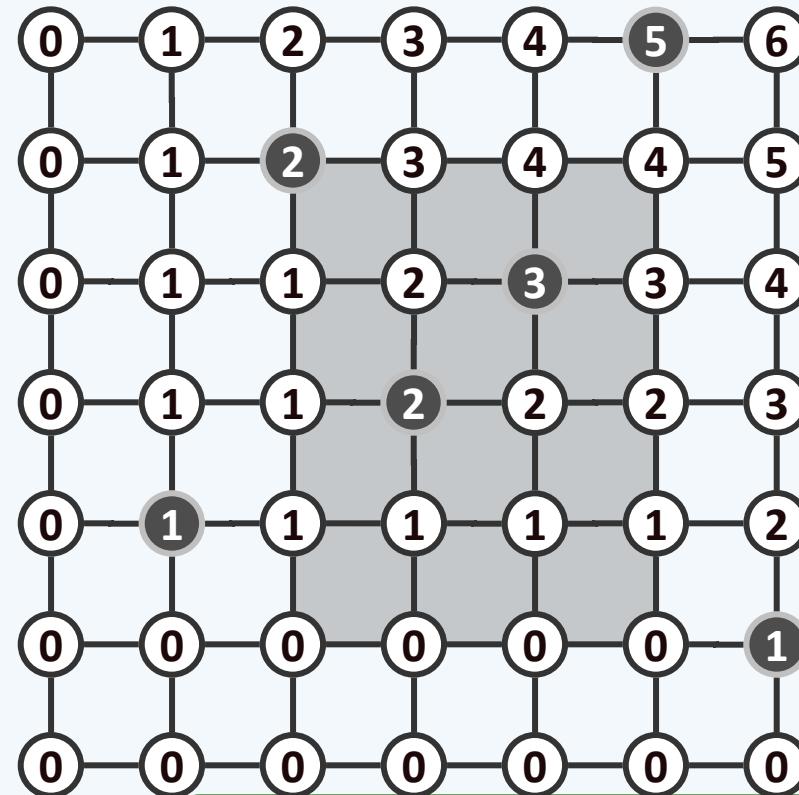
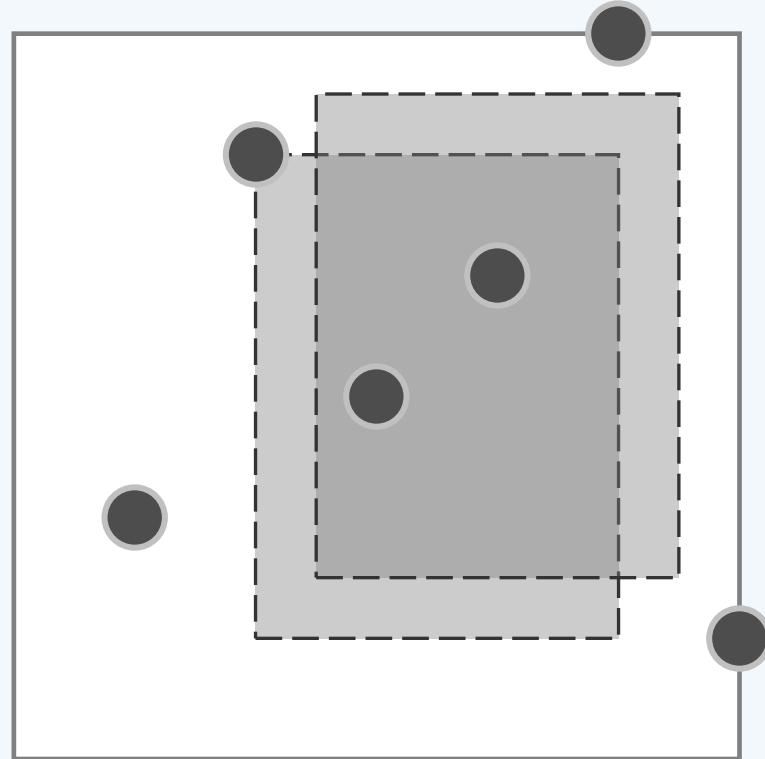
$$|R \cap P| = n(x_1, Y_1) + n(x_2, Y_2) - n(x_1, Y_2) - n(x_2, Y_1)$$



How If ...

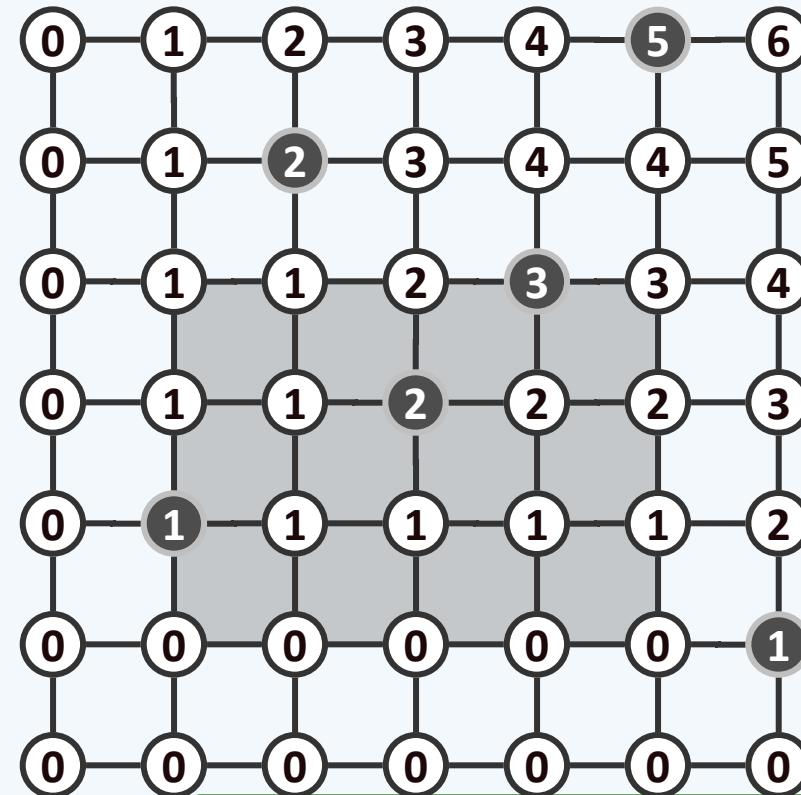
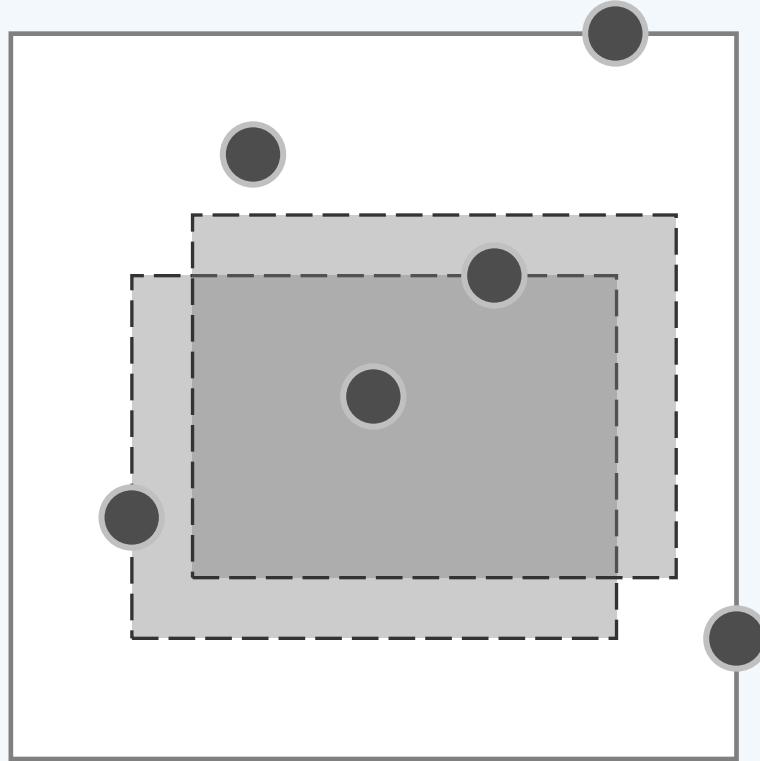
❖ the query range is closed?

there exist points sharing a same x or y coordinate?



Performance

- ❖ Each query needs only $\Theta(\log n)$ time
- ❖ Uses $\Theta(n^2)$ storage and even more for higher dimensions
- ❖ To figure out a better solution, let's go back to the 1D case



Advanced Balanced Search Tree

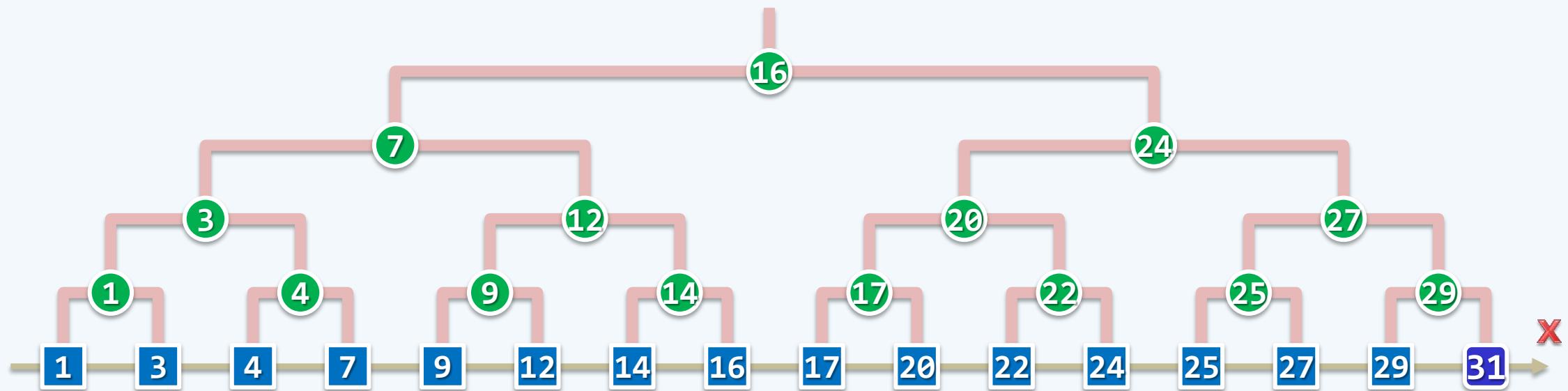
1d-Tree

- Structure

邓俊辉

deng@tsinghua.edu.cn

- ❖ For each v , $v.key = \max\{ u.key \mid u \in L\text{-Tree}(v) \} = v.\text{pred}().key$
- ❖ For each u in $L/R\text{-Tree}(v)$, $x(u) \leq / > x(v)$
- ❖ $\text{search}(x)$: returns the **maximum key not greater than x**



Advanced Balanced Search Tree

1d-Tree

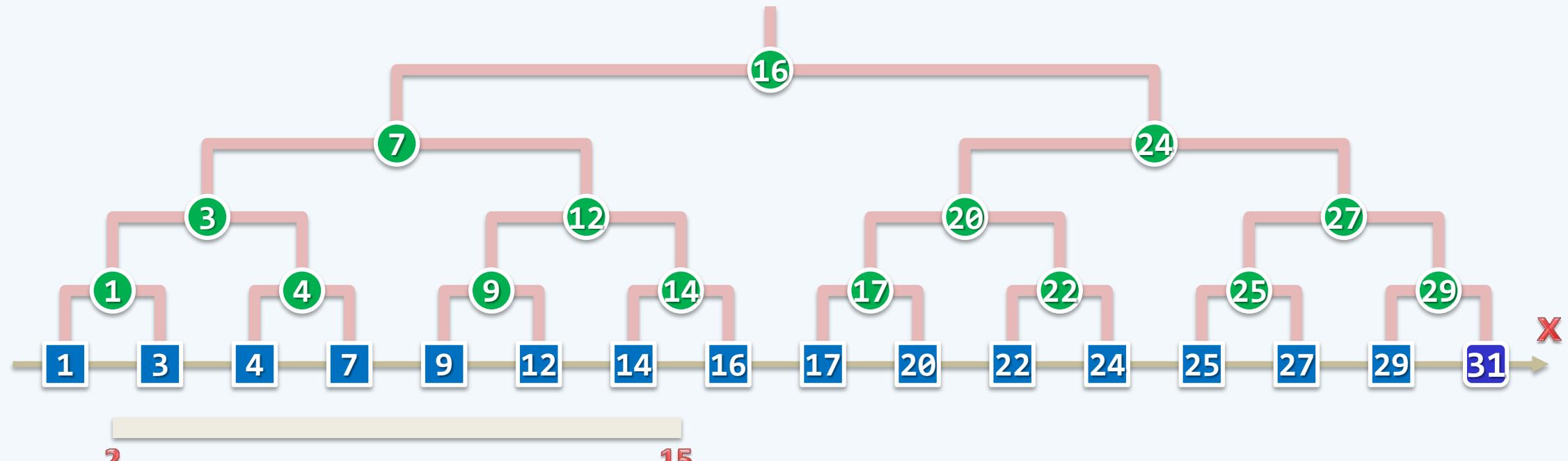
Query

邓俊辉

deng@tsinghua.edu.cn

Lowest Common Ancestor

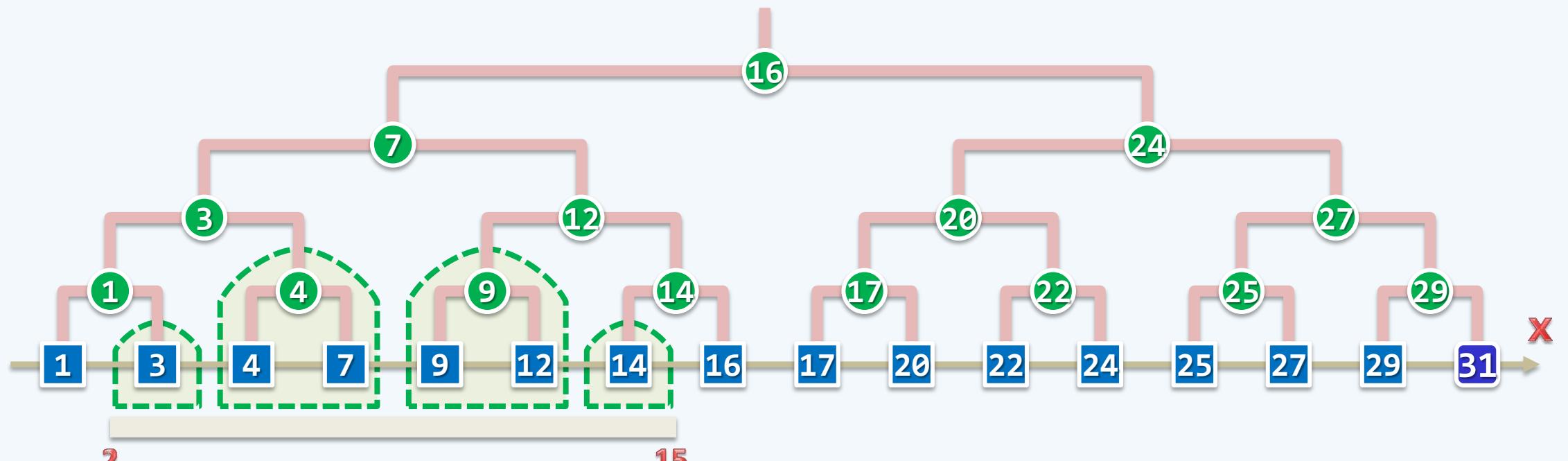
- ❖ Consider, as an example, the query for (2, 15]
- ❖ Binary search:
 - `search(2).succ() = 1.succ() = [3]`
 - `search(15) = [14]`
- ❖ Find: $\text{LCA}(3, 14) = [7]$



Traversal

❖ Starting from the LCA, traverse path(3) and path(14) once more resp.

- All right/left-turns along path(3/14) are ignored and
- the right/left subtree at each left/right-turn is reported



Advanced Balanced Search Tree

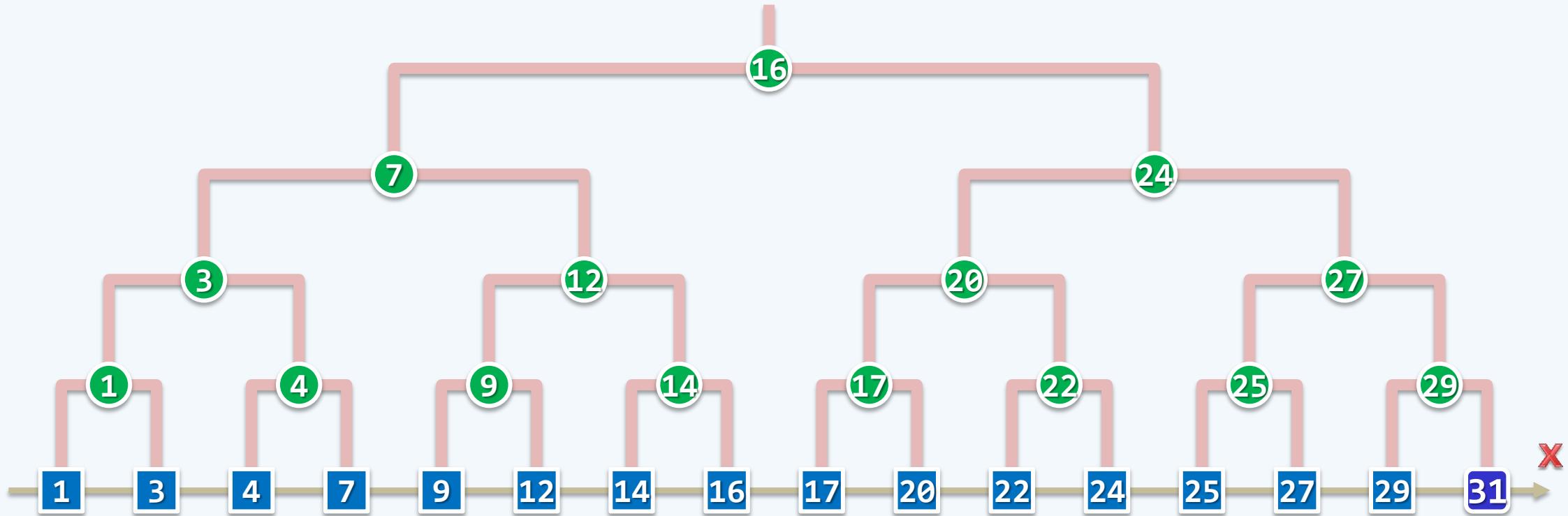
1d-Tree

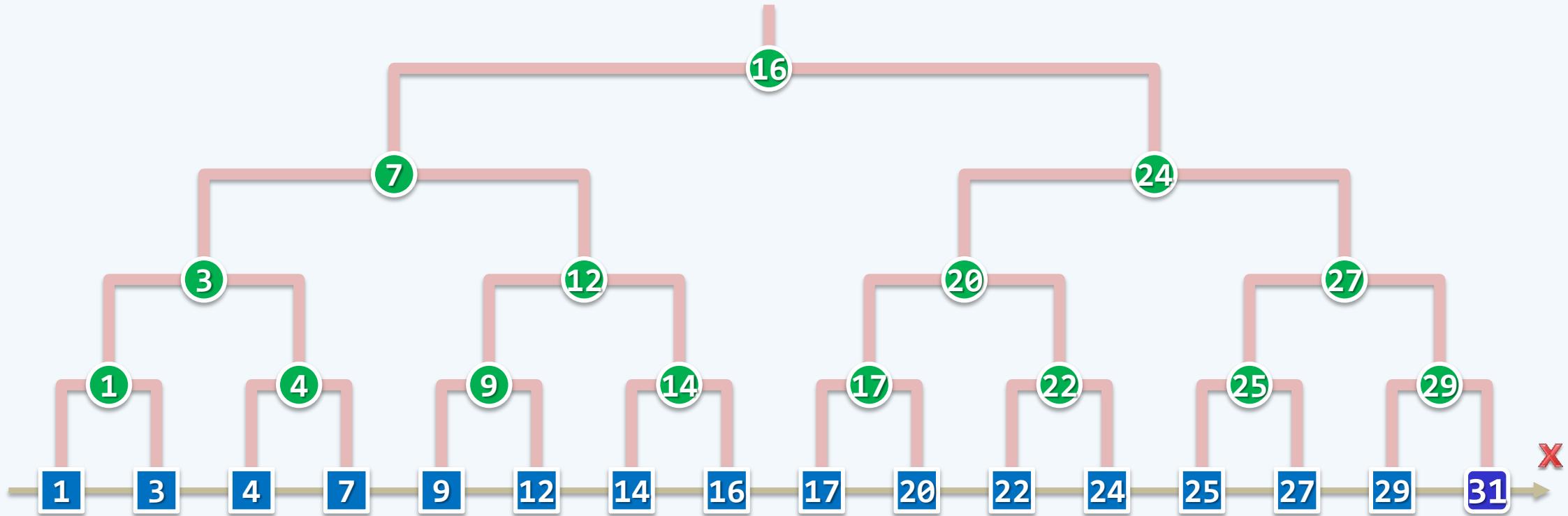
- Complexity

邓俊辉

deng@tsinghua.edu.cn

$\mathcal{O}(n \log n)$ Preprocessing Time

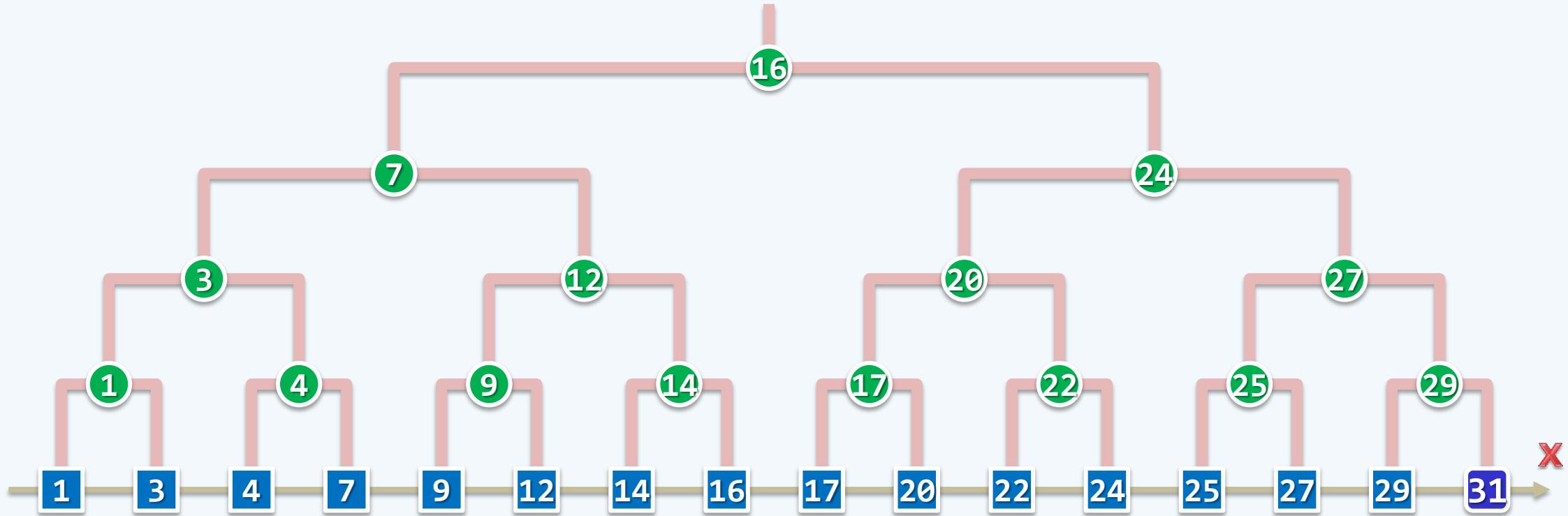


$\Theta(n)$ Storage

$\mathcal{O}(r + \log n)$ Query Time

❖ r = number of points being reported

//Output sensitive!



❖ But, wait for a minute ...

How much time is needed to find the LCA?

//To think about ...

Advanced Balanced Search Tree

2d-Tree

Structure

凡见字数，如停匀，即平分一半为上卦，一半为下卦。如字数不均，即少一字为上卦，取天轻清之义，以多一字为下卦，取地重浊之义

邓俊辉

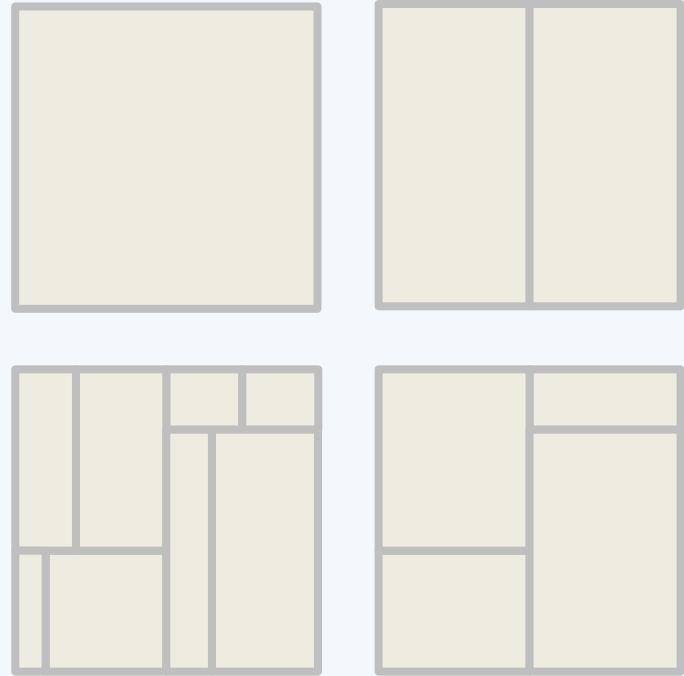
deng@tsinghua.edu.cn

Divide-And-Conquer

❖ To extend the BBST method to planar GRS, we

- **divide** the plane recursively and
- **arrange** the regions into a kd-tree

❖ Start with a single region (the entire plane)



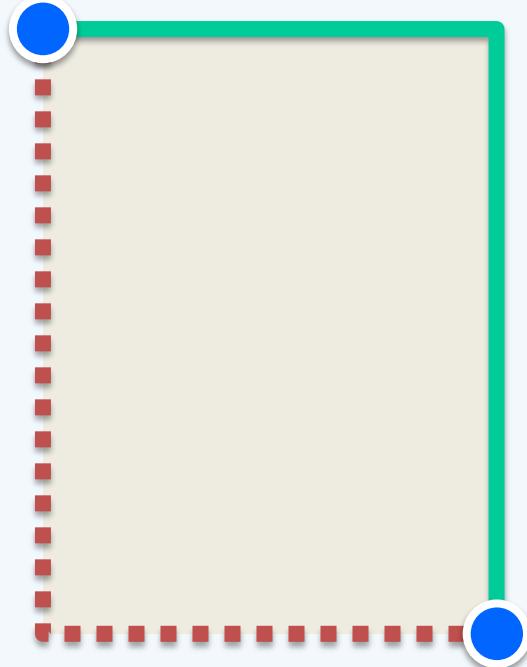
Partition the region vertically/horizontally on each even/odd level

Partition the sub-regions recursively

More Details

❖ To make it work,

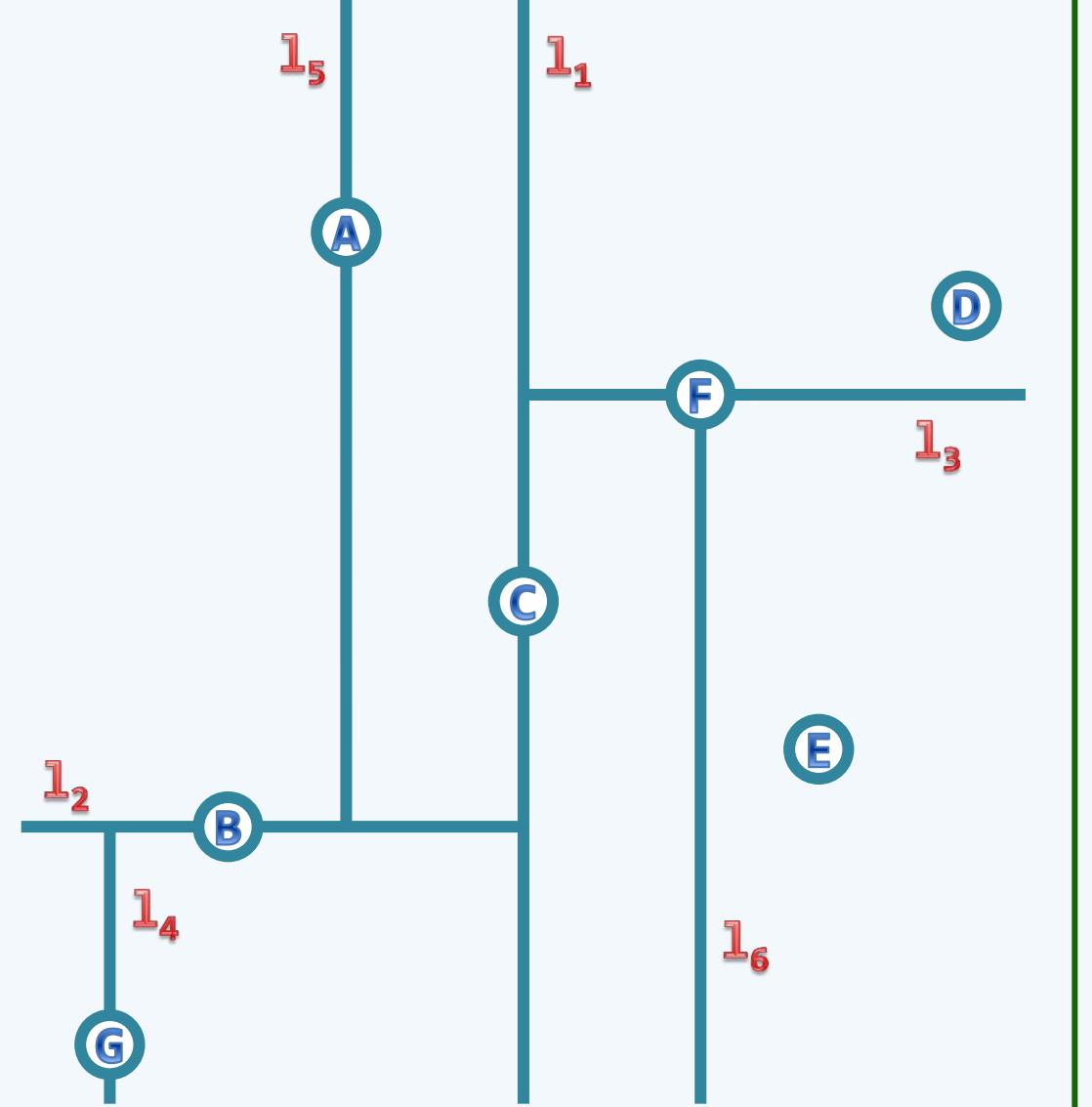
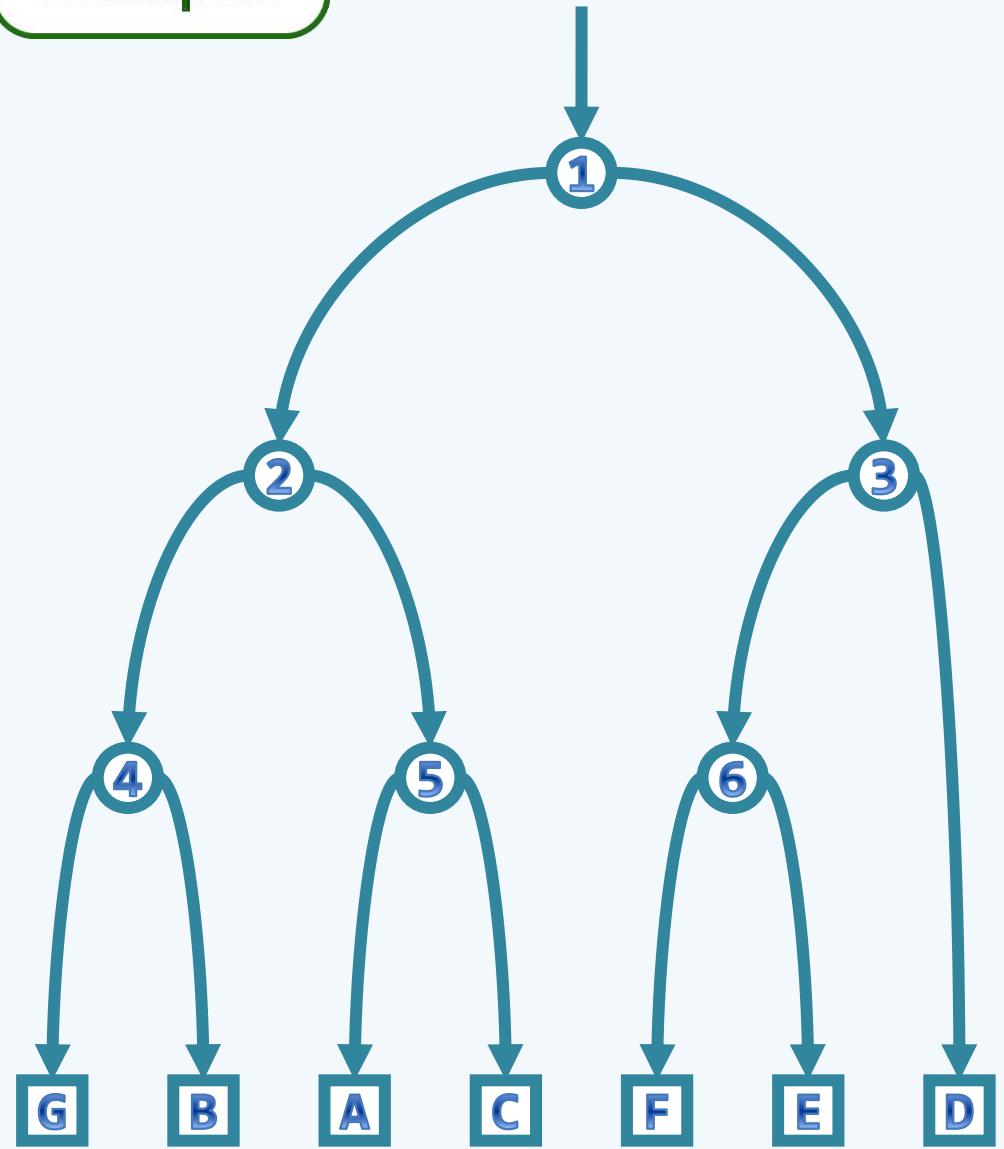
- each partition should be done as **evenly** as possible (at median)
- each region is defined to be **open/closed** on the **left-lower/right-upper** sides



❖ Degeneracy assumption:

no two input points lie on a same vertical/horizontal line

Example



Advanced Balanced Search Tree

2d-Tree

Construction

邓俊辉

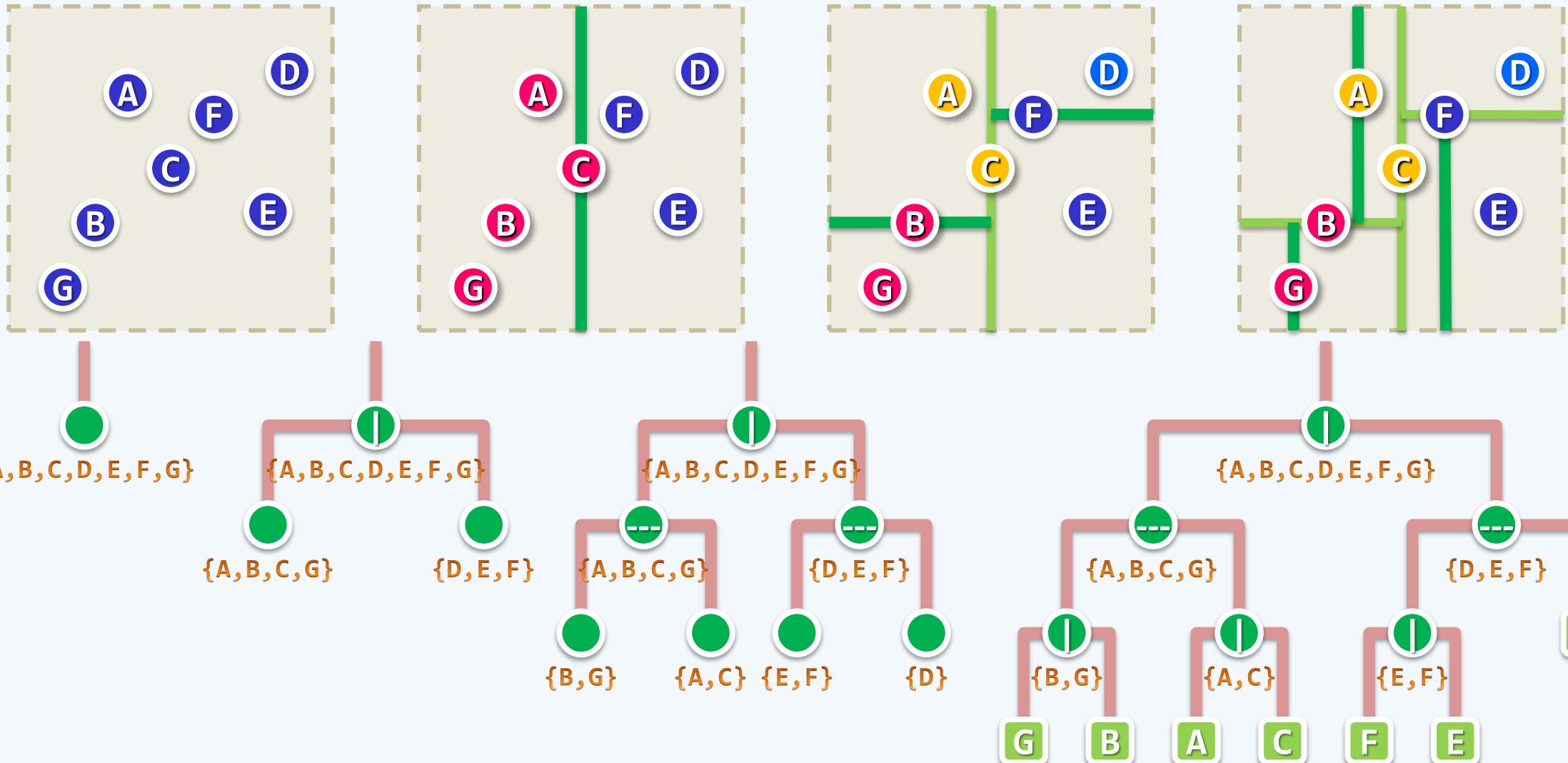
God found himself by creating.

deng@tsinghua.edu.cn

KdTree * buildKdTree (P , d)

```
❖ // construct a 2d-(sub)tree for point (sub)set P at depth d  
  
if ( P == {p} ) return CreateLeaf(p) //base  
  
root = CreateKdNode()  
  
root->splitDirection = Even(d) ? VERTICAL : HORIZONTAL  
  
root->splitLine = FindMedian( root->splitDirection, P ) // $\Theta(n)$ !  
  
( P1, P2 ) = Divide( P, root->splitDirection, root->splitLine ) //DAC  
  
root->lChild = buildKdTree( P1, d + 1 ) //recurse  
  
root->rChild = buildKdTree( P2, d + 1 ) //recurse  
  
return( root )
```

Example



Advanced Balanced Search Tree

2d-Tree

Canonical Subsets

韦小宝跟著她走到桌边，只见桌上大白布上钉满了几千枚绣花针，几千块碎片已拼成一幅完整无缺的大地图，难得的是几千片碎皮拼在一起，既没多出一片，也没少了一片。

邓俊辉

deng@tsinghua.edu.cn

⦿ Each node corresponds to

- a rectangular sub-region of the plane, as well as
- the subset of points contained in the sub-region

❖ Each of these subsets is called a **canonical set**

⦿ For each internal node X with children L and R,

$$\text{region}(X) = \text{region}(L) \cup \text{region}(R)$$

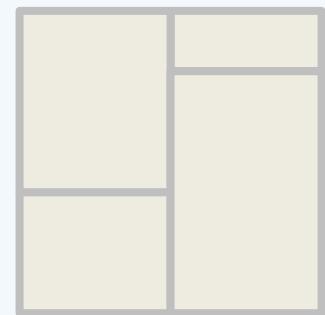
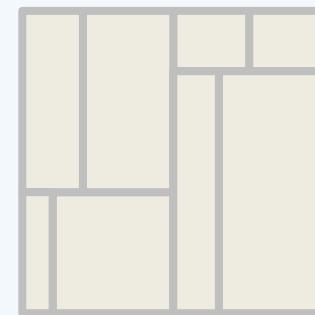
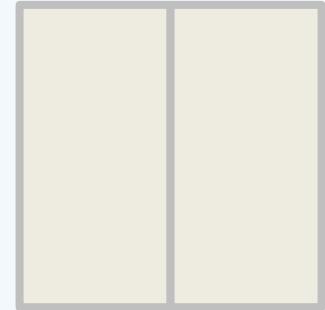
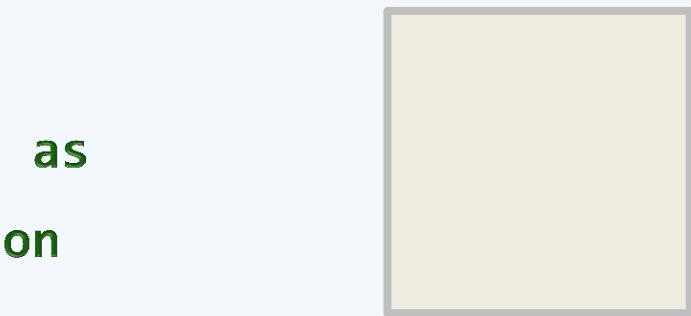
⦿ Sub-regions of nodes at a same depth

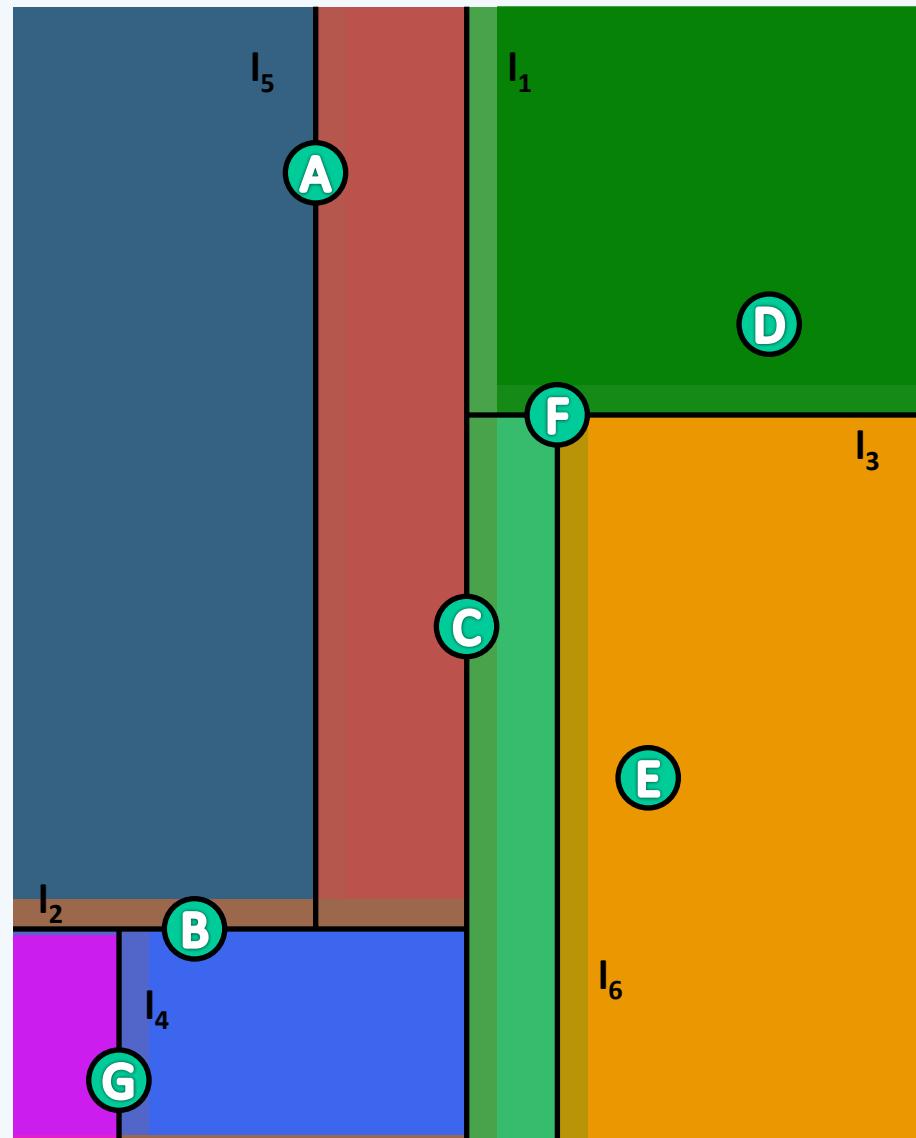
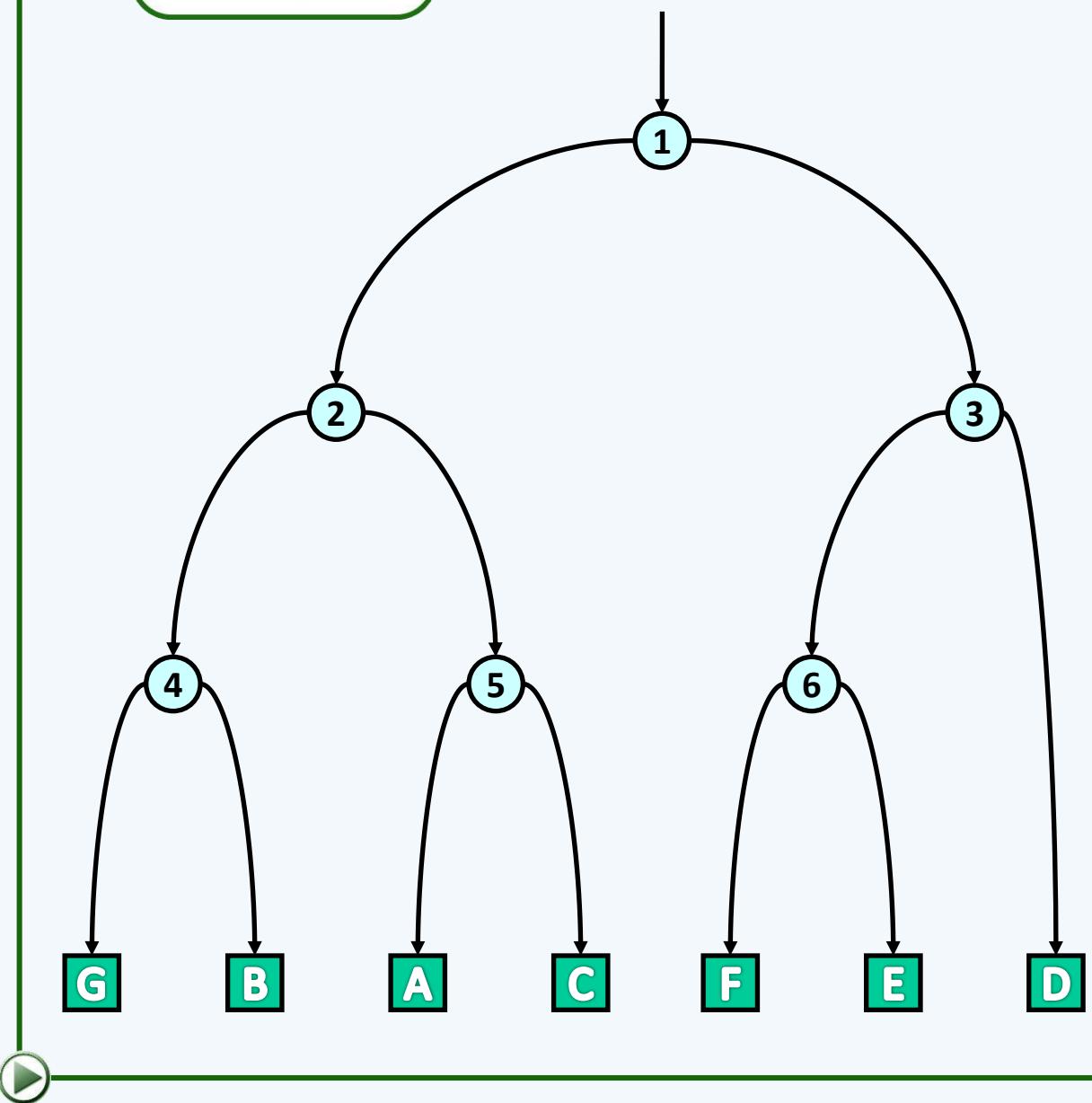
- never intersect with each other, and
- their union covers the entire plane

❖ We will see soon that

each 2D GRS can be answered by

the **union** of a number of CS's



Example

Advanced Balanced Search Tree

2d-Tree

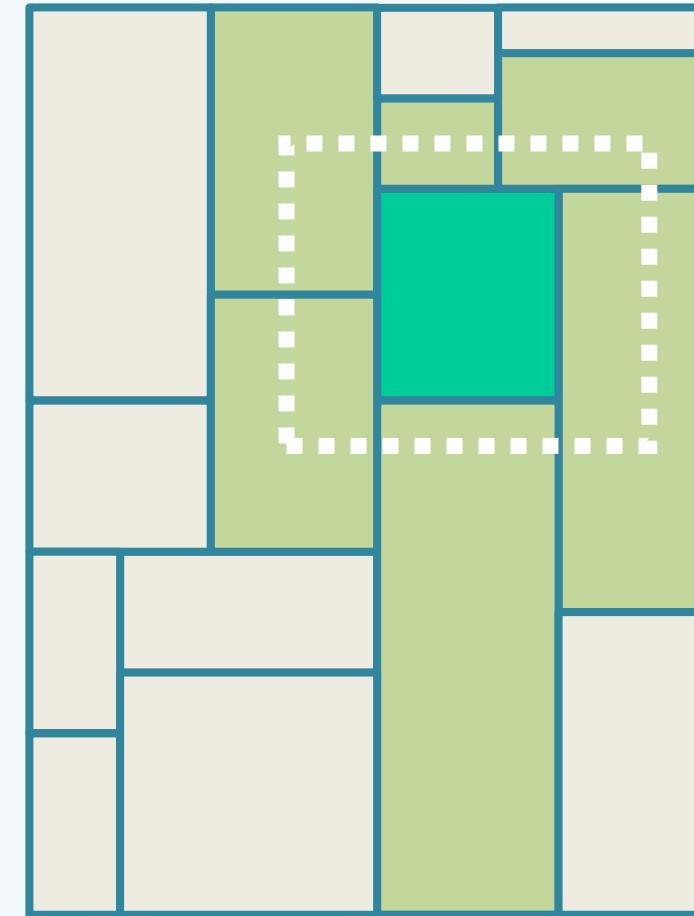
Query

邓俊辉

deng@tsinghua.edu.cn

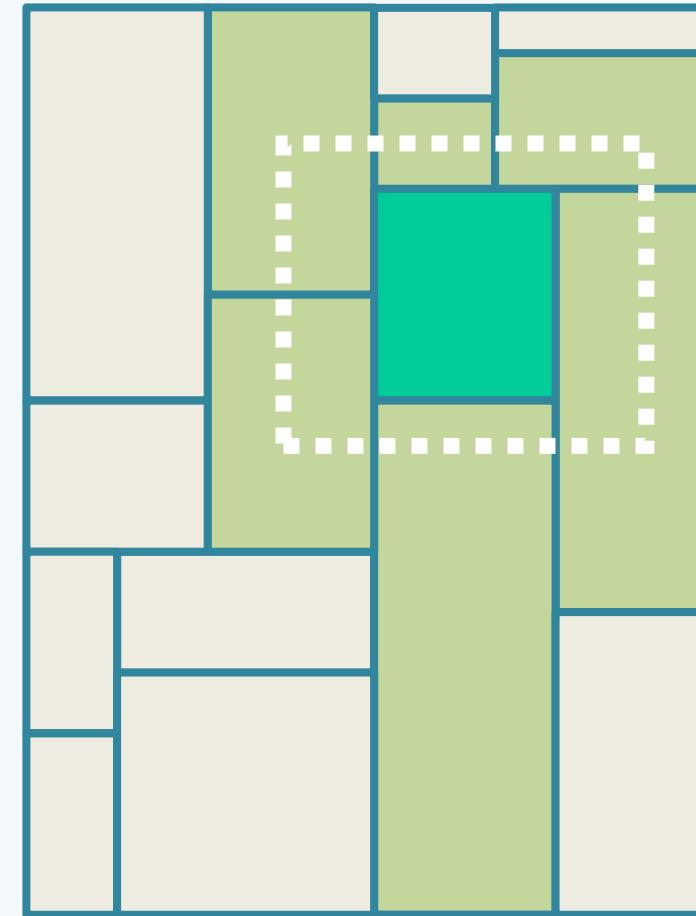
kdSearch(v , R)

```
❖ if ( isLeaf( v ) ) {  
    if ( inside( v, R ) )  
        report(v);  
  
    return;  
}  
}
```



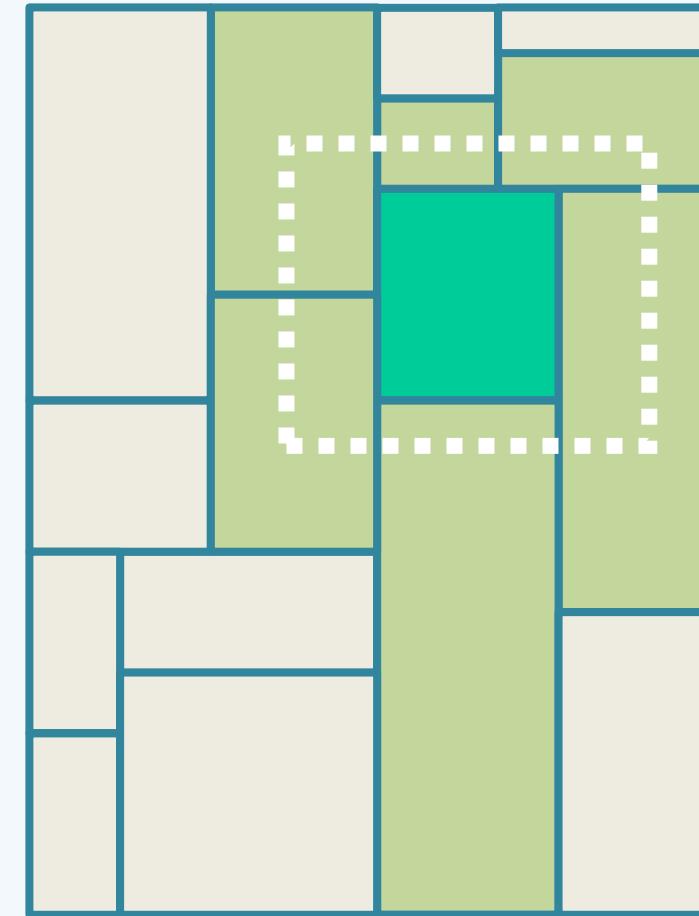
kdSearch(v , R)

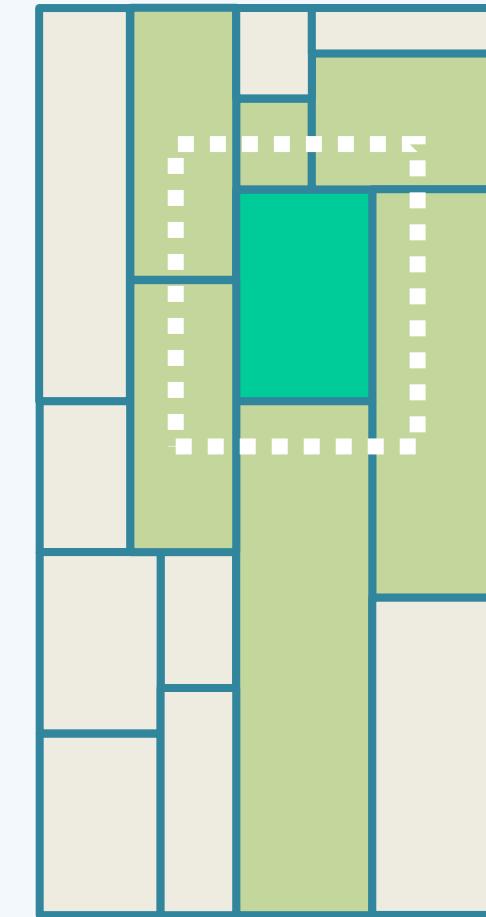
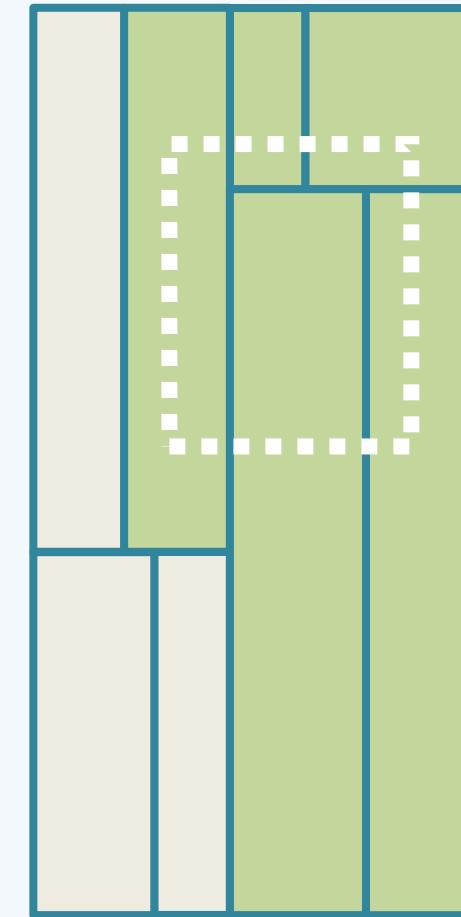
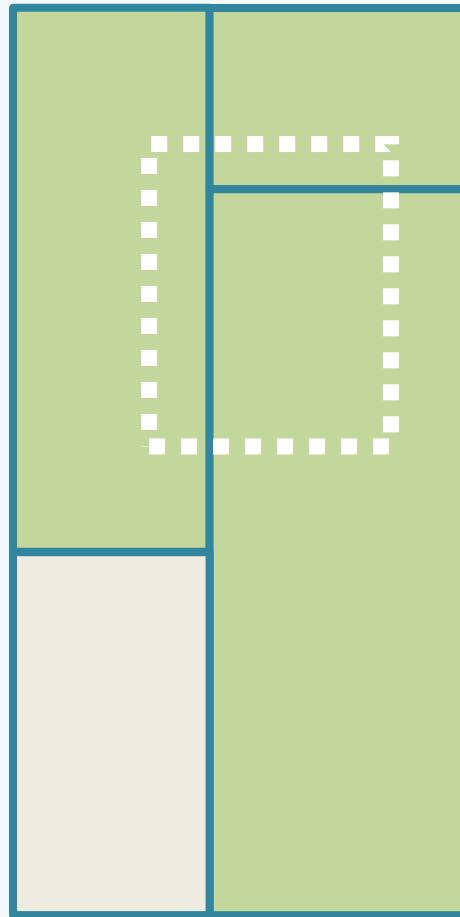
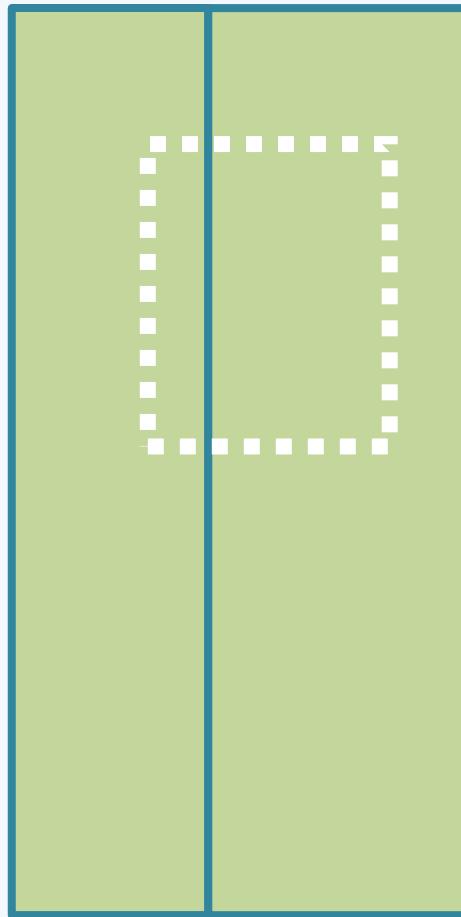
```
❖ if ( region( v->lc ) ⊆ R )  
    reportSubtree( v->lc );  
  
else if ( intersect( region( v->lc ), R ) )  
    kdSearch( v->lc, R );
```



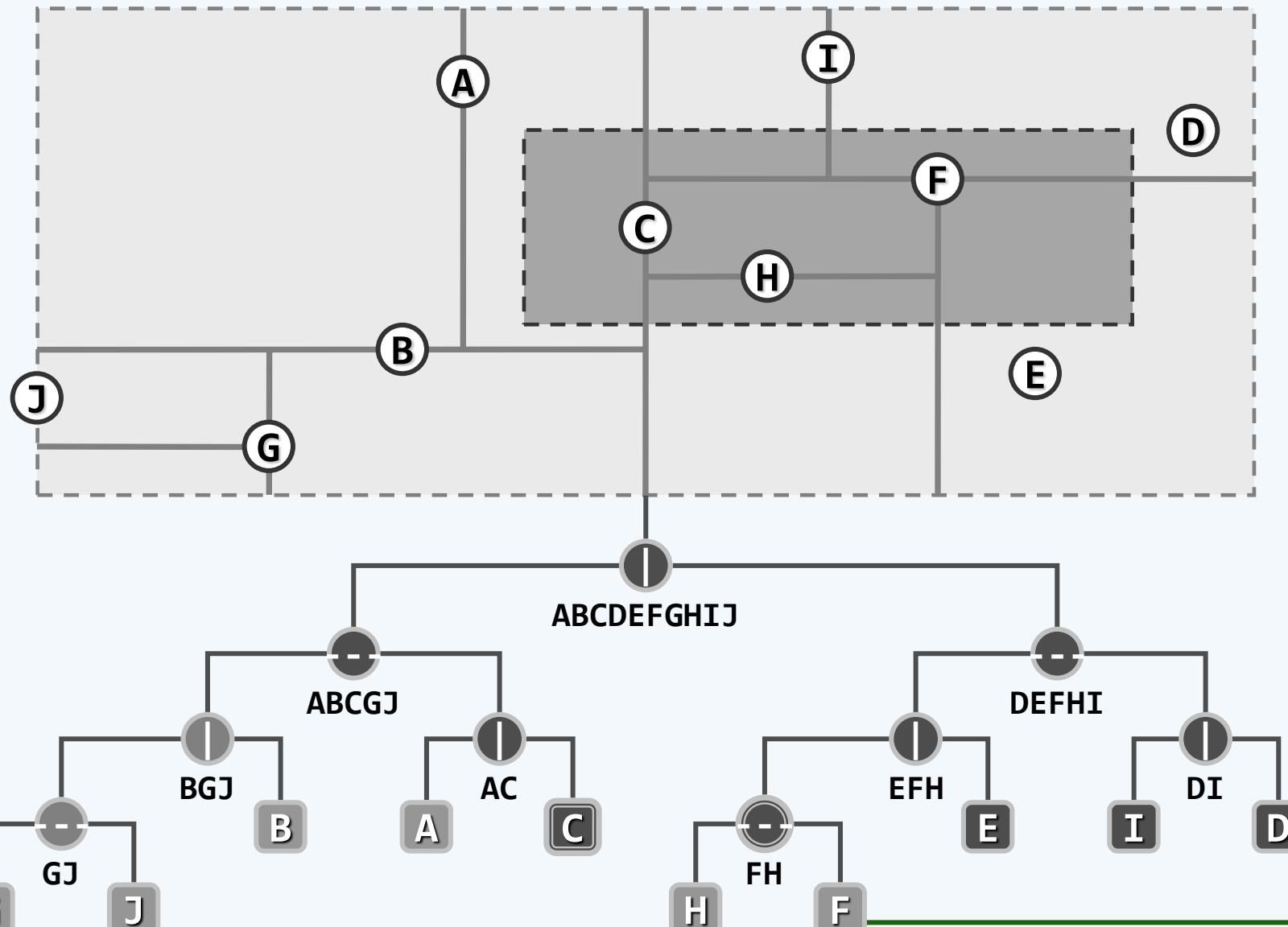
kdSearch(v , R)

```
❖ if ( region( v->rc ) ⊆ R )  
    reportSubtree( v->rc );  
  
else if ( intersect( region( v->rc ), R ) )  
    kdSearch( v->rc, R );
```



Example

Example



Advanced Balanced Search Tree

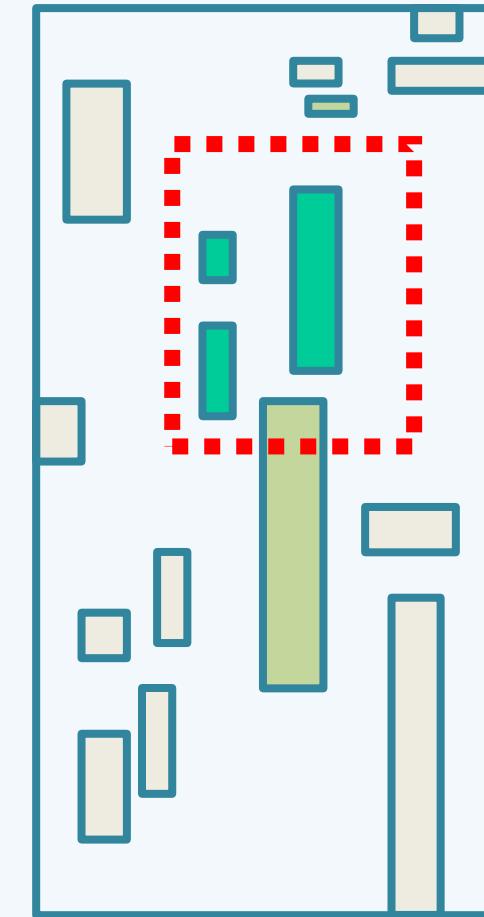
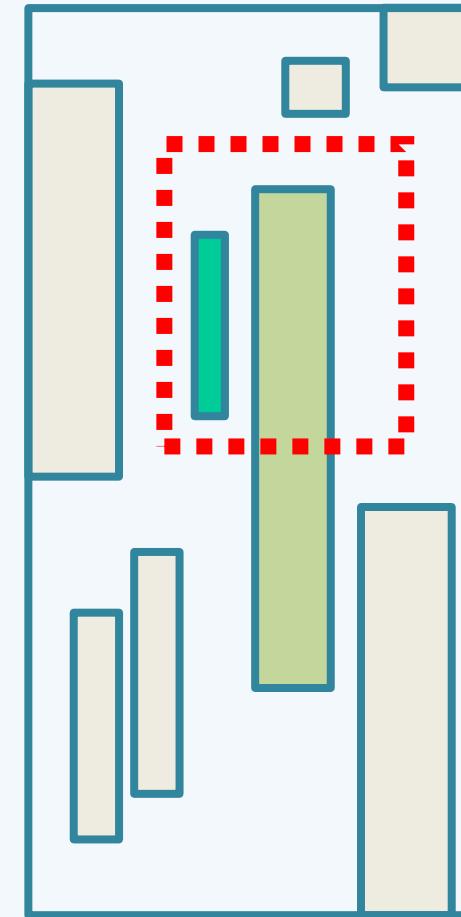
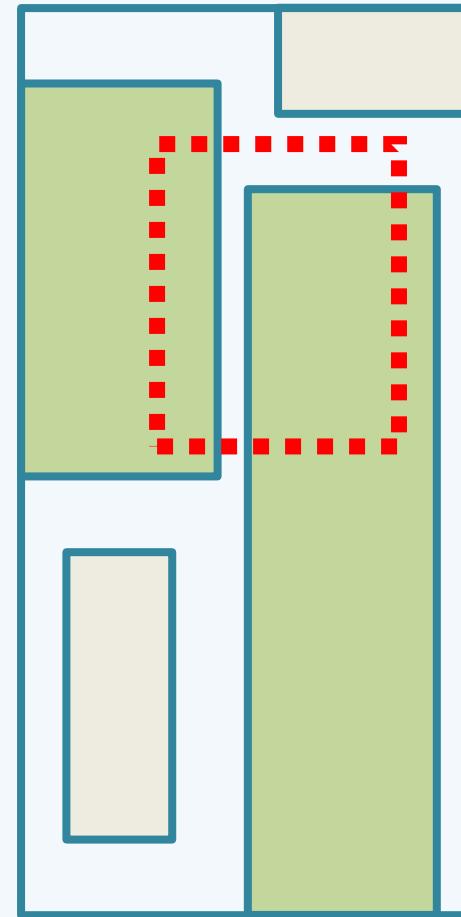
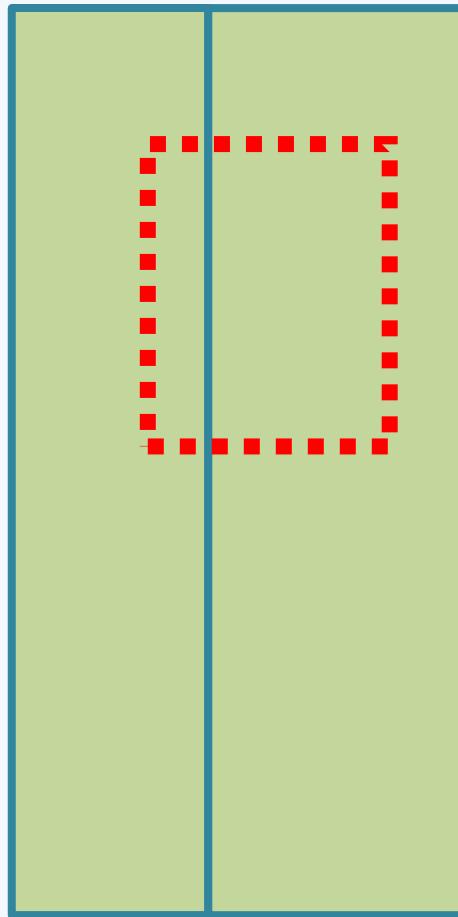
2d-Tree

- Optimization

邓俊辉

deng@tsinghua.edu.cn

Bounding Box



Advanced Balanced Search Tree

2d-Tree

Complexity

邓俊辉

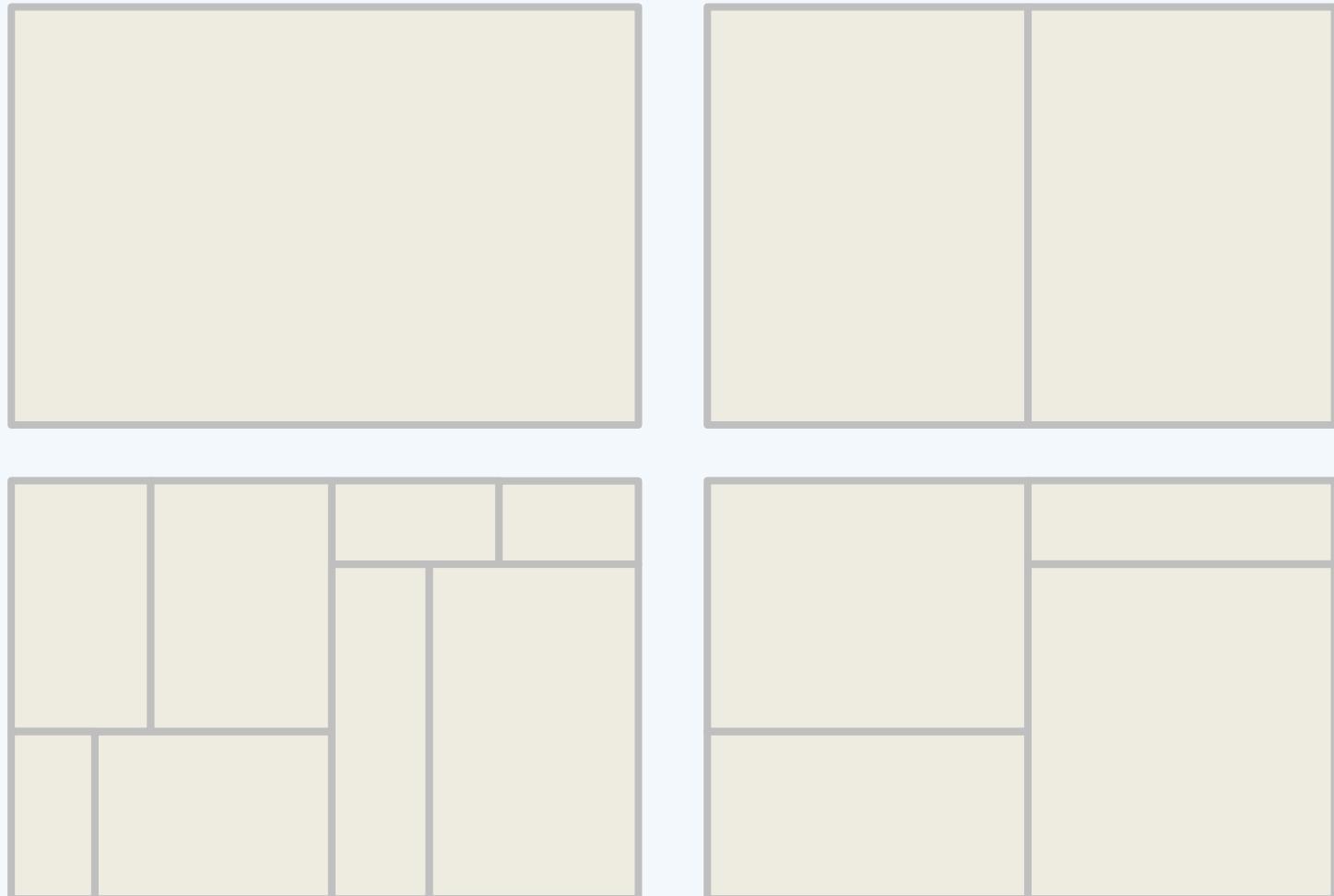
deng@tsinghua.edu.cn

Preprocessing

❖ $T(n)$

$$= 2*T(n/2) + O(n)$$

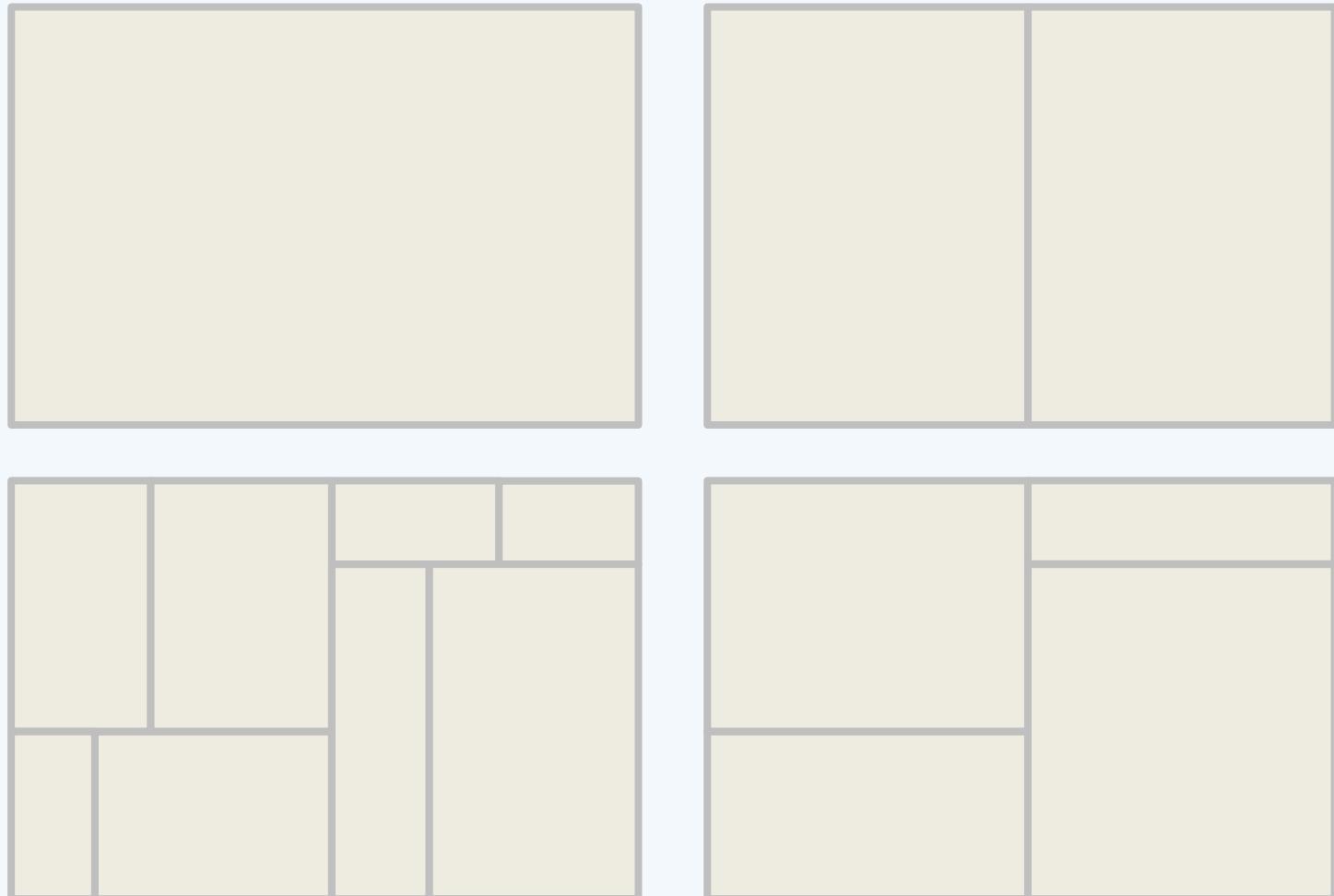
$$= O(n \log n)$$



Storage

❖ The tree has a height
of $\Theta(\log n)$

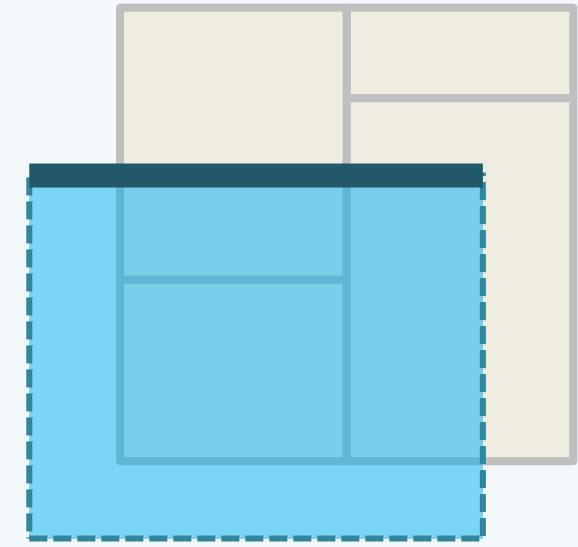
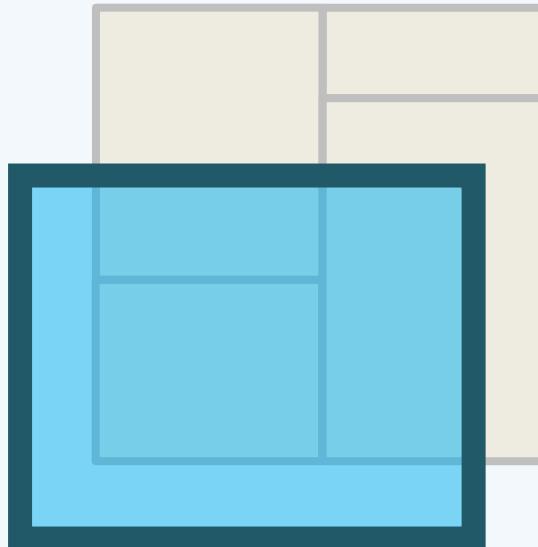
❖ 1
 $+ 2$
 $+ 4$
 $+ \dots$
 $+ \Theta(2^{\log n})$
 $= \Theta(n)$



Query Time

❖ Claim:

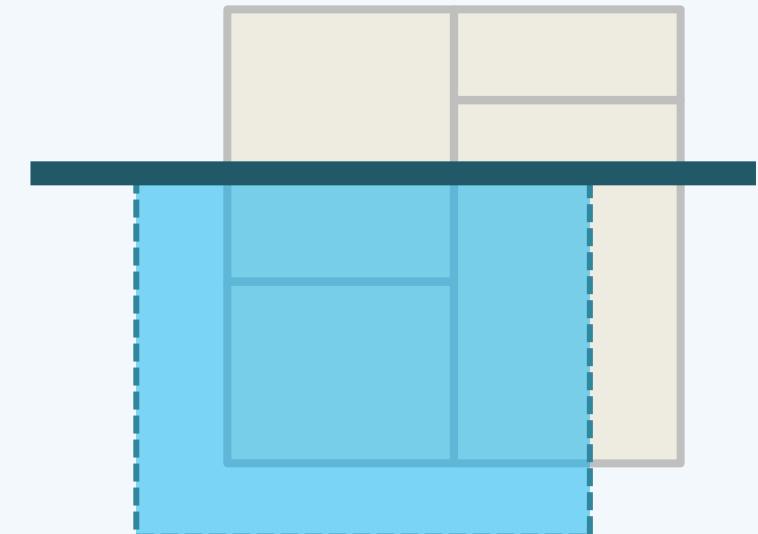
$$\text{Report + Search} = \mathcal{O}(r + \sqrt{n})$$



❖ The searching time depends on

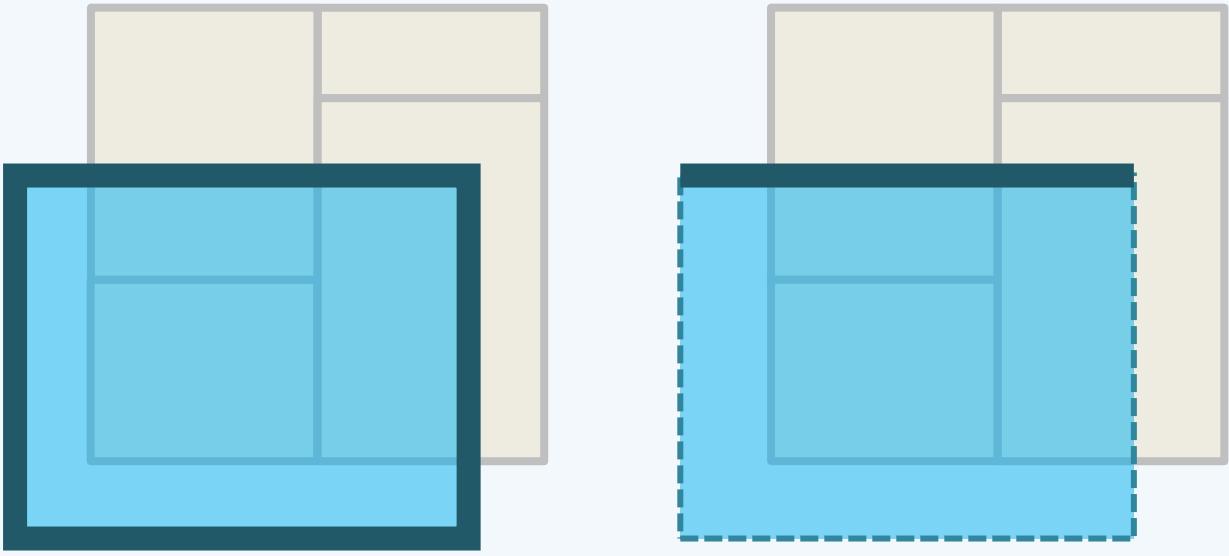
- the number of recursive calls, or
- $Q(n)$, the number of sub-regions

intersecting with R (at all levels)



Query Time

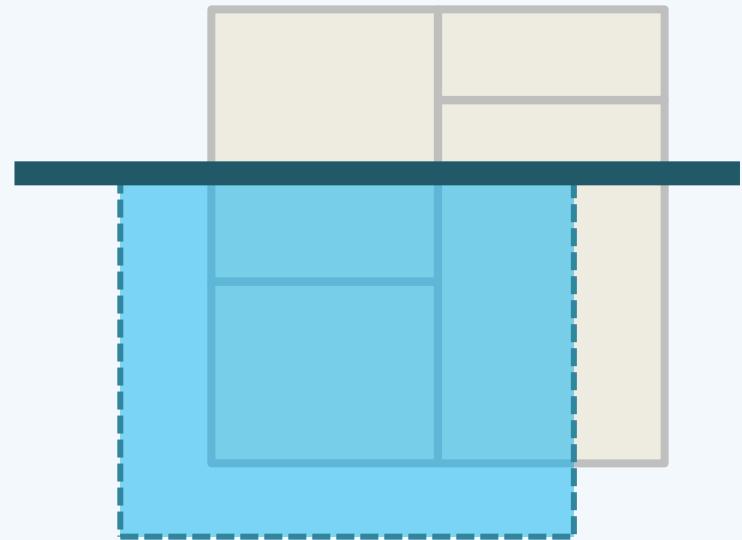
- For each node,
 - no more than 2 of its 4 grandchildren will recurse



❖ Recurrence

$$\begin{aligned} - Q(1) &= O(1) \\ - Q(n) &= 2 + 2 * Q(n/4) \end{aligned}$$

$$\diamond \text{Solve to } Q(n) = O(\sqrt{n})$$



Advanced Balanced Search Tree

2d-Tree

Beyond 2D

邓俊辉

deng@tsinghua.edu.cn

kd-Tree

❖ Can 2d-tree be extended to kd-tree and help higher dimensional GRS?

If yes, how efficiently can it help?

❖ A kd-tree in k-dimensional space

is constructed by

recursively divide \mathcal{E}^k

along the $1^{\text{st}}, 2^{\text{nd}}, \dots, k^{\text{th}}$ dimensions

$$\mathcal{O}(r + n^{1 - 1/d})$$

❖ An orthogonal range query on a set of n points in \mathbb{E}^d

- can be answered in $\mathcal{O}(r + n^{1 - 1/d})$ time,
- using a kd-tree of size $\mathcal{O}(n)$, which
- can be constructed in $\mathcal{O}(n \log n)$ time

Advanced Balanced Search Tree

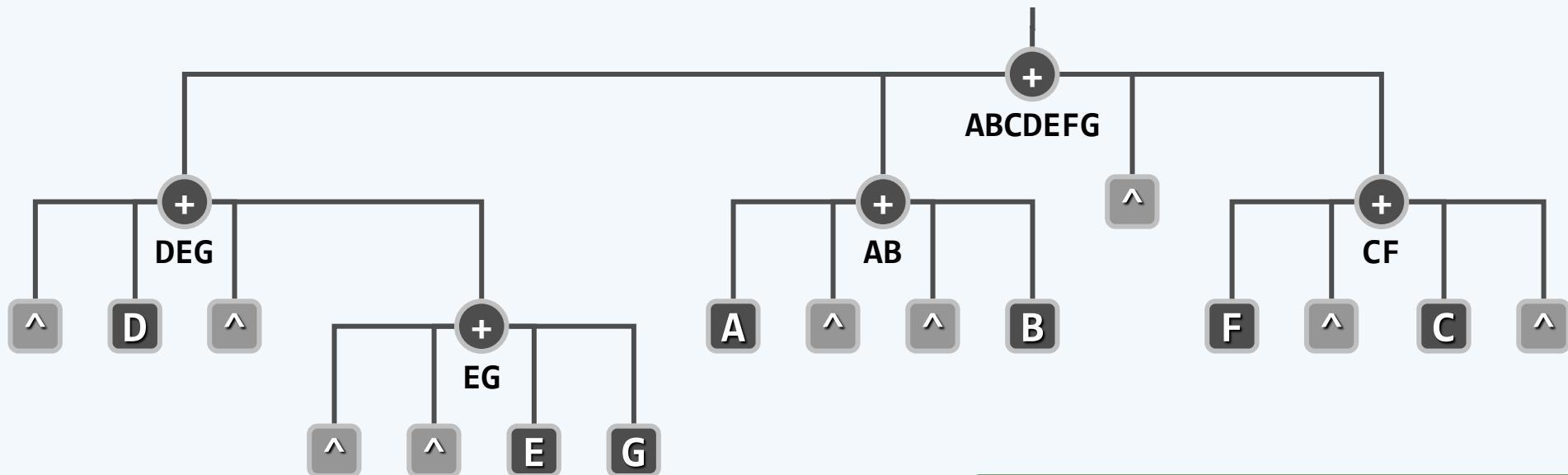
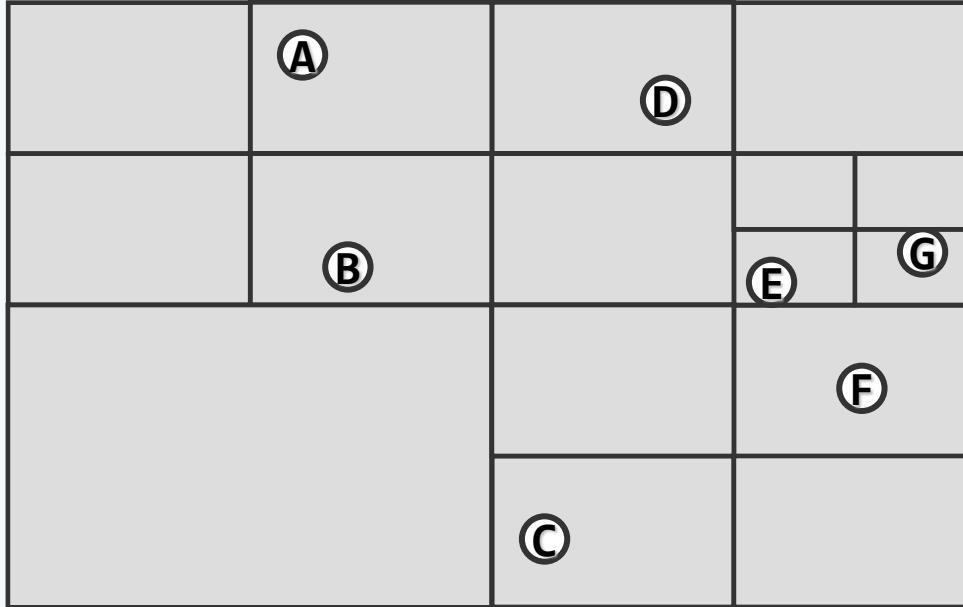
2d-Tree

Quadtree

邓俊辉

deng@tsinghua.edu.cn

Quadtree



Advanced Balanced Search Tree

Multi-Level Search Tree

x-Query + y-Query

邓俊辉

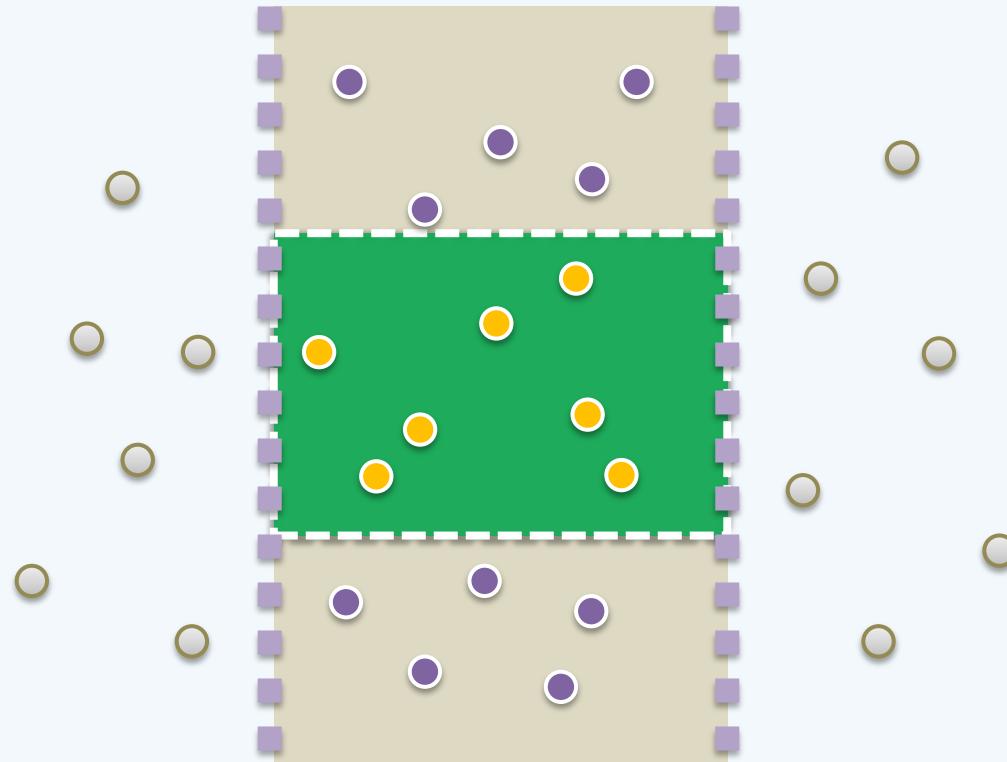
deng@tsinghua.edu.cn

2D Range Query = x-Query + y-Query

? Is there any structure
which answers range query

faster than kd-trees?

⦿ An m-D orthogonal range query
can be answered by
the intersection of
m 1-D queries



2D Range Query = x-Query + y-Query

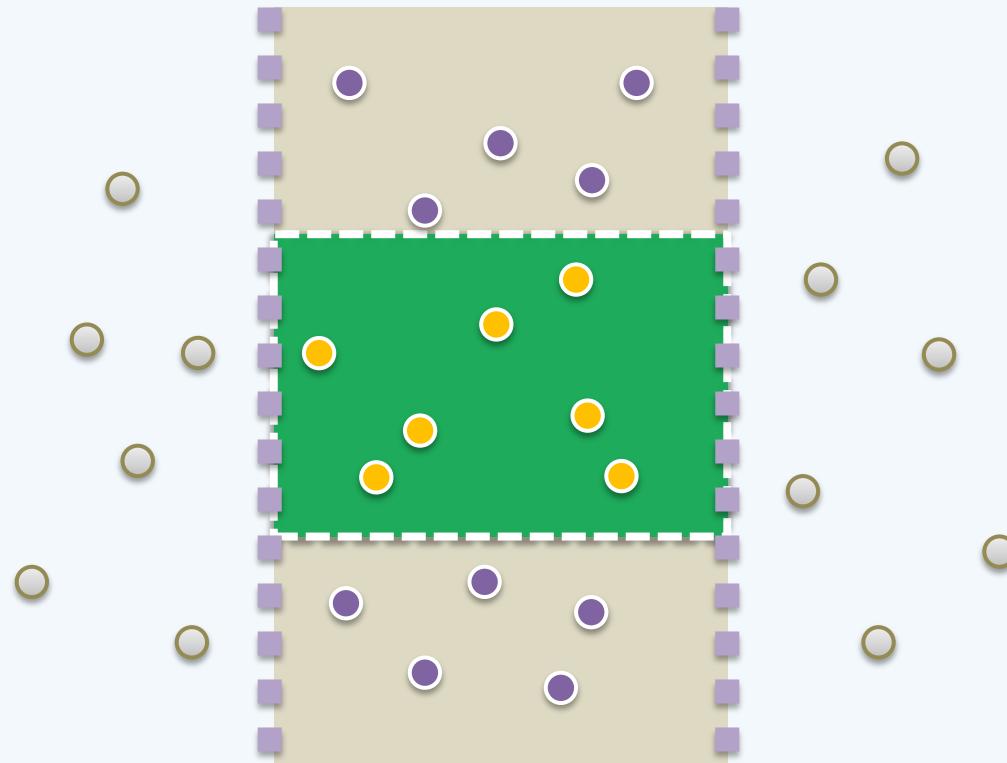
❖ For example ...

❖ A 2D range query

can be divided into

two 1D range queries:

- find all points
in $[x_1, x_2]$; and then
- find from these candidates
those lying in $[y_1, y_2]$



Advanced Balanced Search Tree

Multi-Level Search Tree

Worst Case

邓俊辉

deng@tsinghua.edu.cn

❖ Using kd-trees,

we need $\mathcal{O}(1 + \sqrt{n})$ time

But here ...

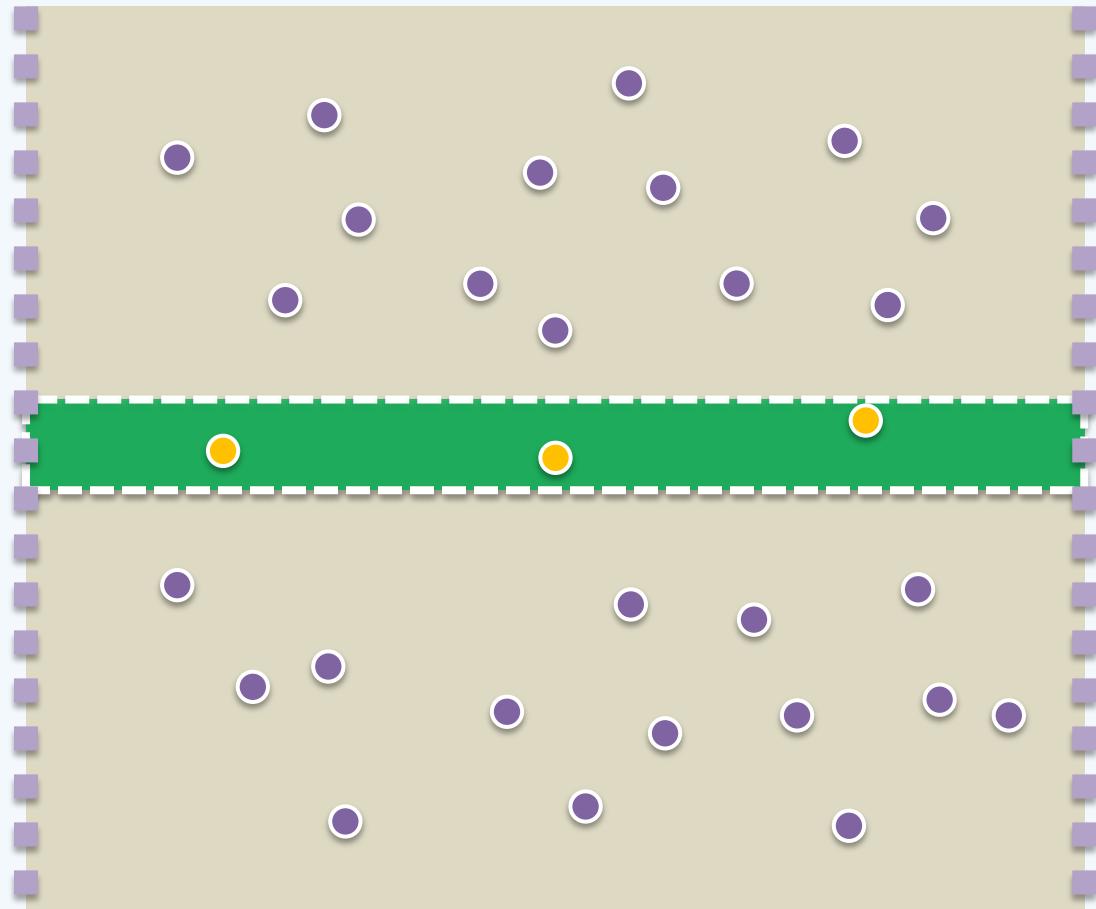
❖ The x-query returns

(almost) all points whereas

the y-query rejects

(almost) all

❖ We spent $\Omega(n)$ time before getting $r = 0$ points



Geometric Range Search

Multi-Level Search Tree

x-Query * y-Queries

邓俊辉

deng@tsinghua.edu.cn

2D Range Query = x-Query * y-Queries

❖ This idea can be implemented

in the following manner:

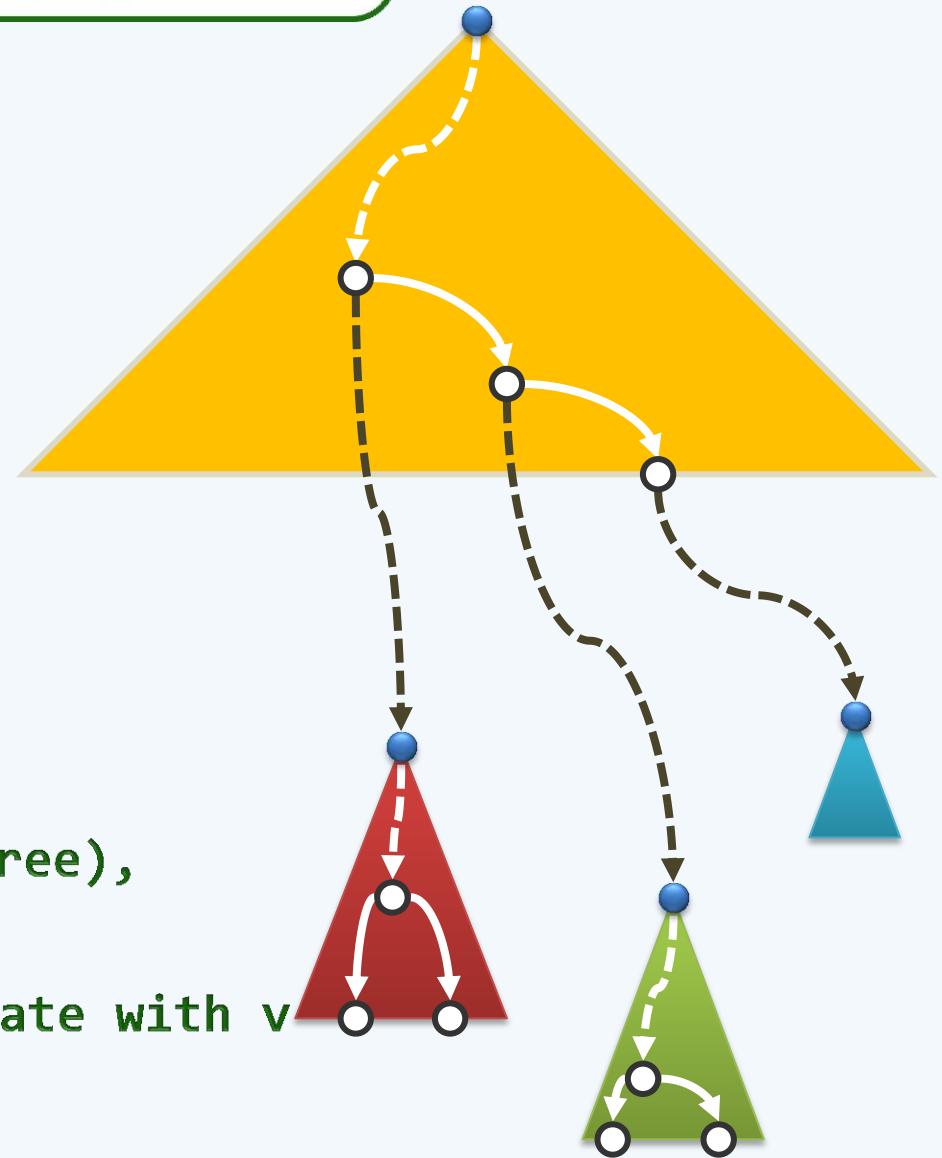
- build a 1D BBST (called \boxed{x} -tree)

- for the first range query (\boxed{x} -query);

- for each node v in the x -range tree,

- build a y -coordinate BBST (called \boxed{y} -tree),

- containing the canonical subset associate with v



Tree of Trees

❖ Hence we have built

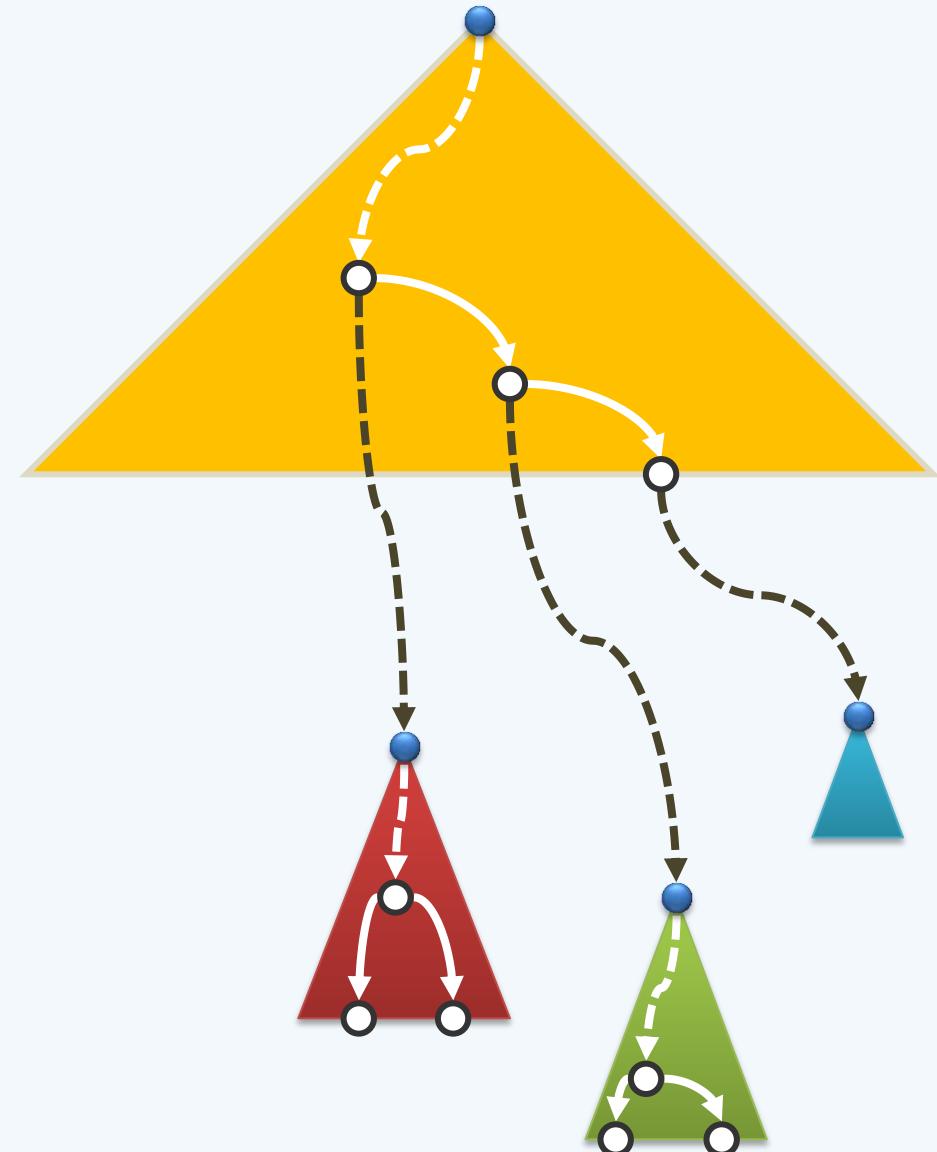
an x -tree of (a number of) y -trees,

which is called

a multi-level search tree

? How to answer range queries

with such an MLST?



Advanced Balanced Search Tree

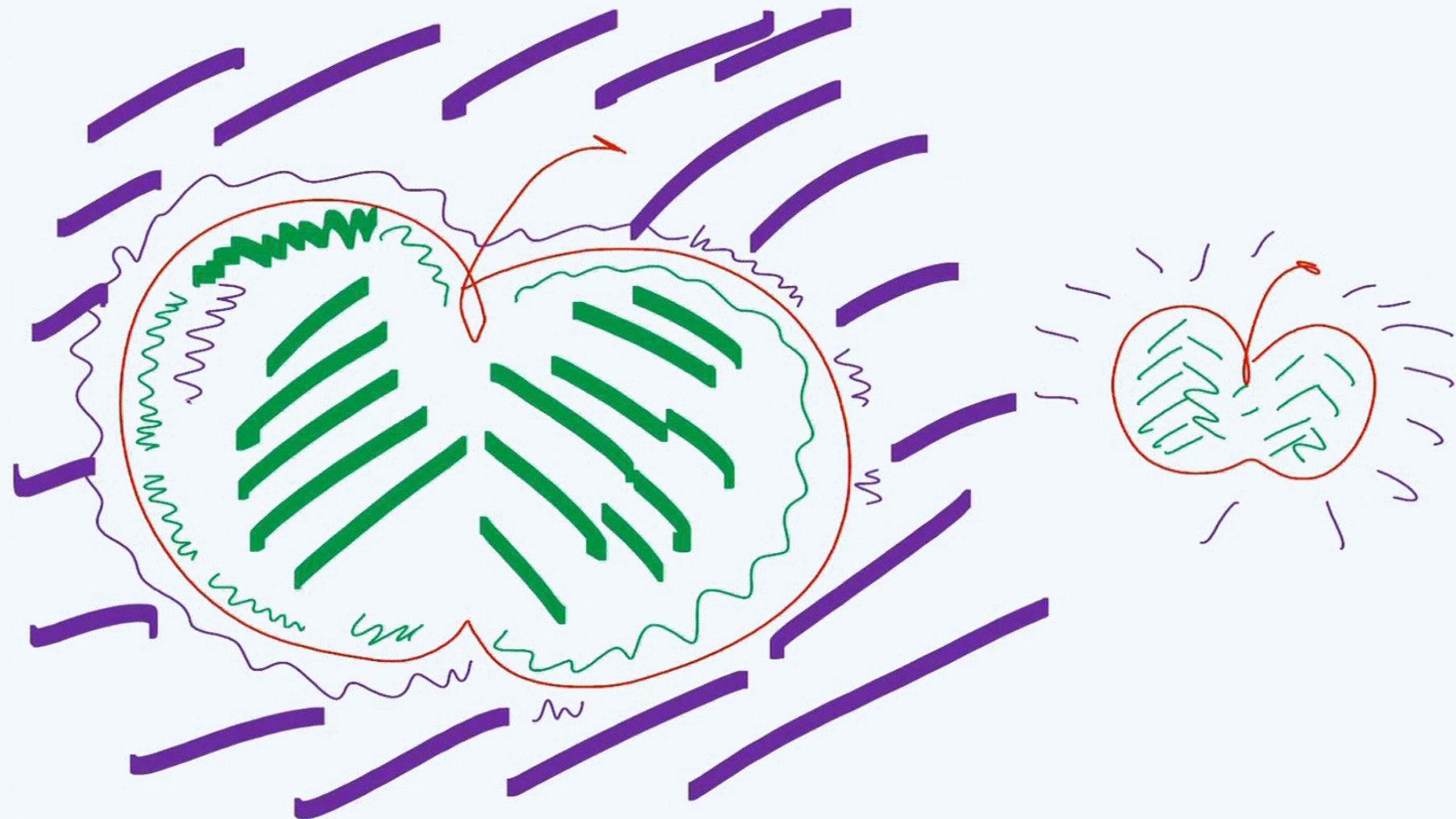
Multi-Level Search Tree

Query

邓俊辉

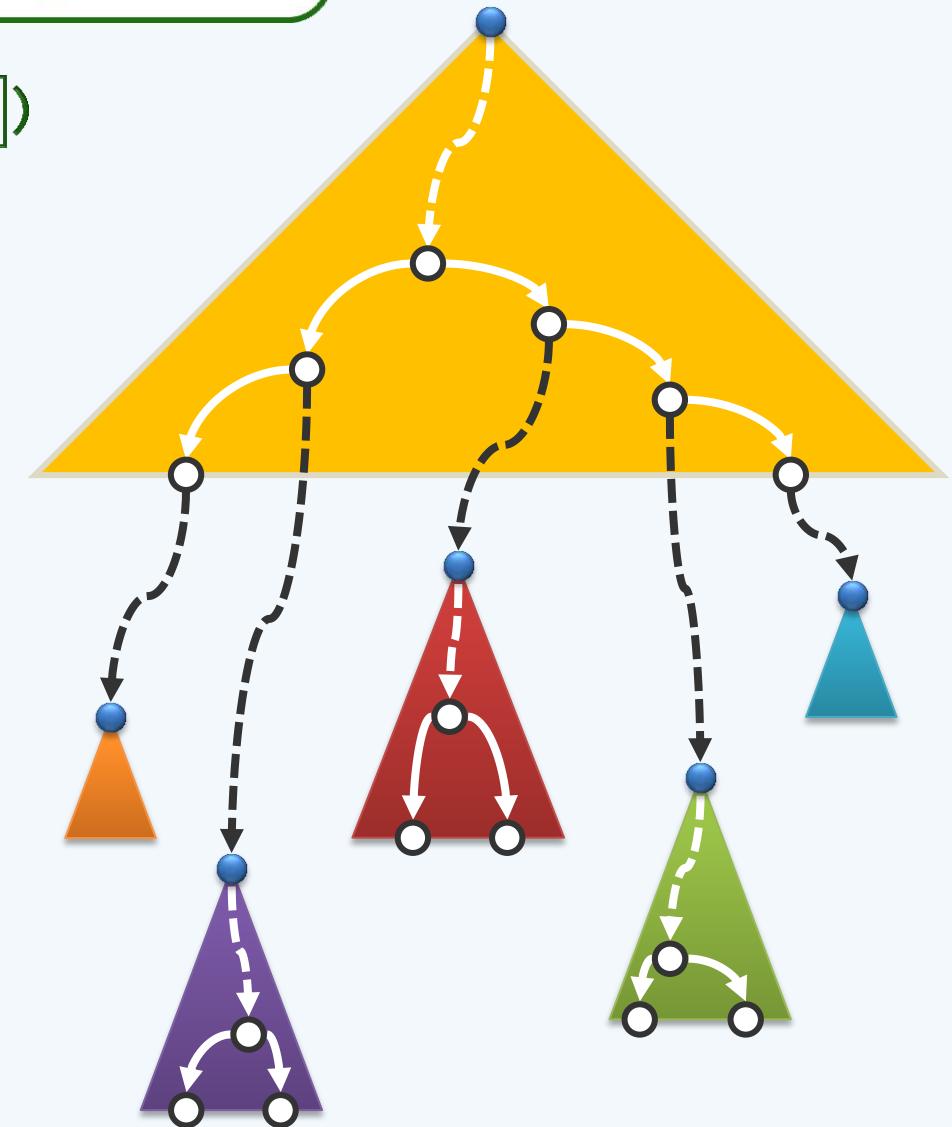
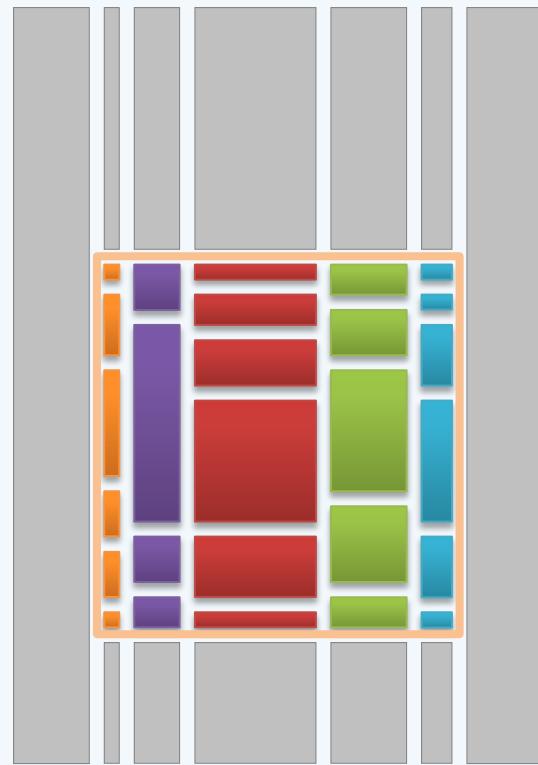
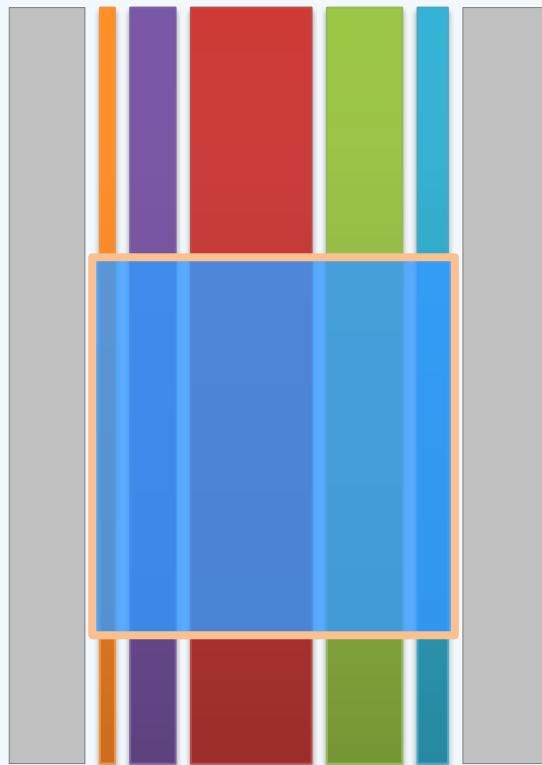
deng@tsinghua.edu.cn

Painters' Strategy



2D Range Query = x-Query * y-Queries

❖ Query Time = $\mathcal{O}(r + \lceil \log^2 n \rceil) \sim \mathcal{O}(r + \lceil \log n \rceil)$



Algorithm

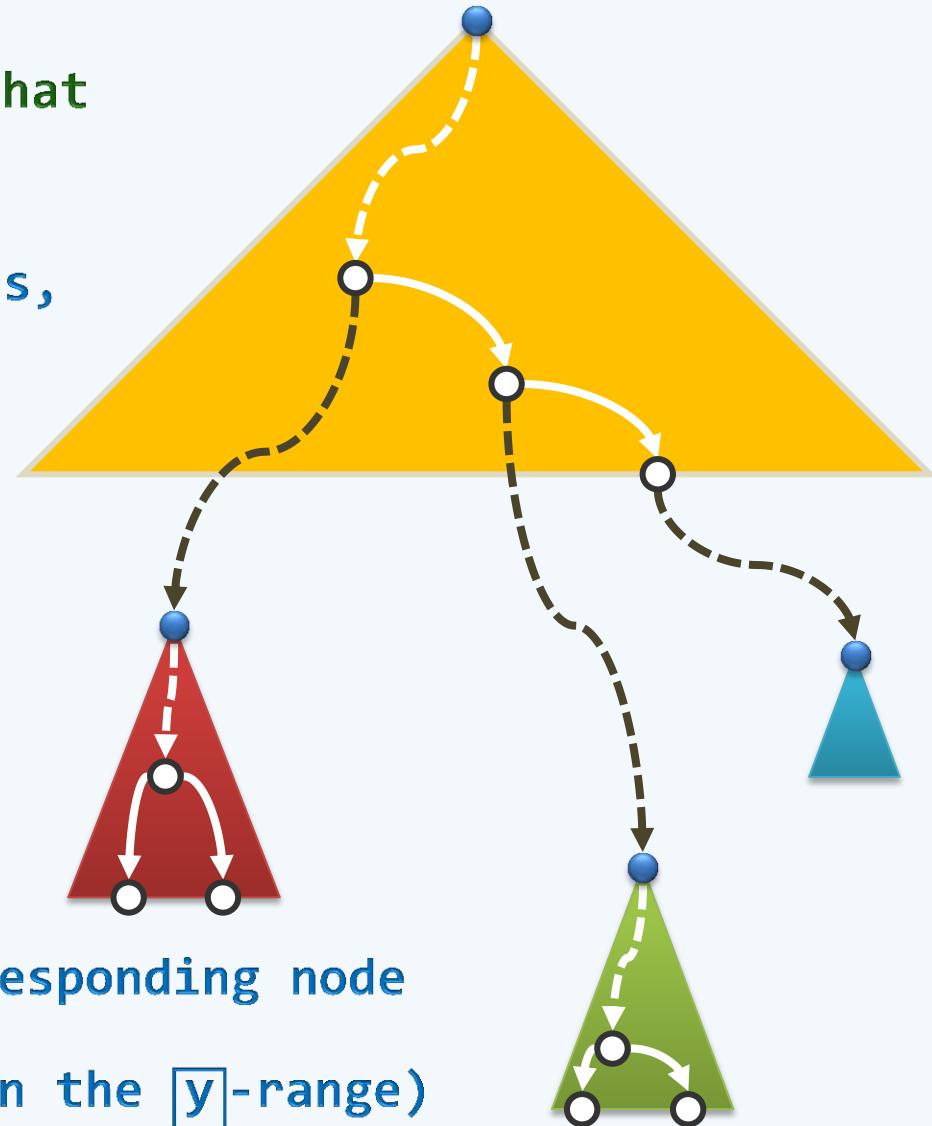
- Determine the canonical subsets of points that satisfy the first query

```
// there will be  $\Theta(\log n)$  such canonical sets,  
// each of which is just represented as  
// a node in the  $\boxed{x}$ -tree
```

- Find out from each canonical subset

which points lie within the \boxed{y} -range

```
// To do this,  
// for each canonical subset,  
// we access the  $\boxed{y}$ -tree for the corresponding node  
// this will be again a  $\boxed{1D}$  range search (on the  $\boxed{y}$ -range)
```



Advanced Balanced Search Tree

Multi-Level Search Tree

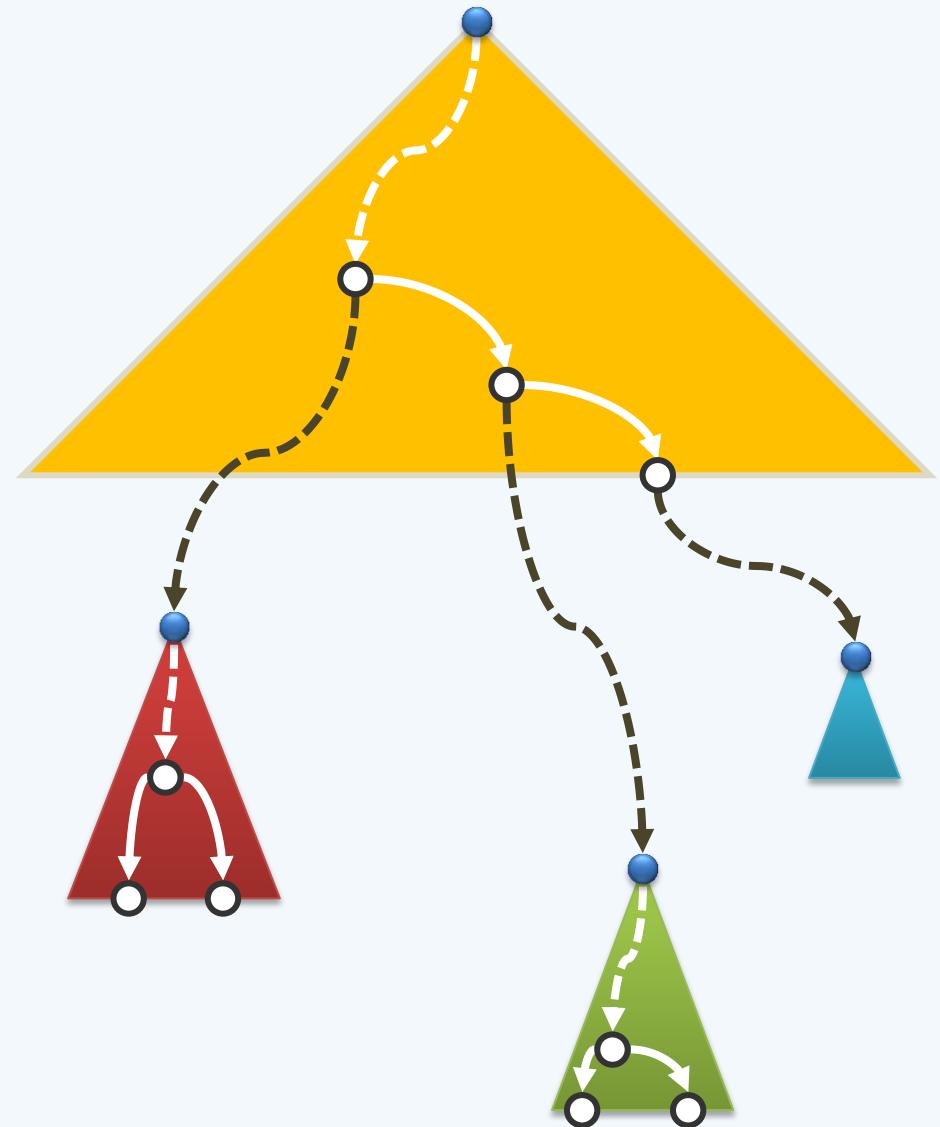
Complexity

邓俊辉

deng@tsinghua.edu.cn

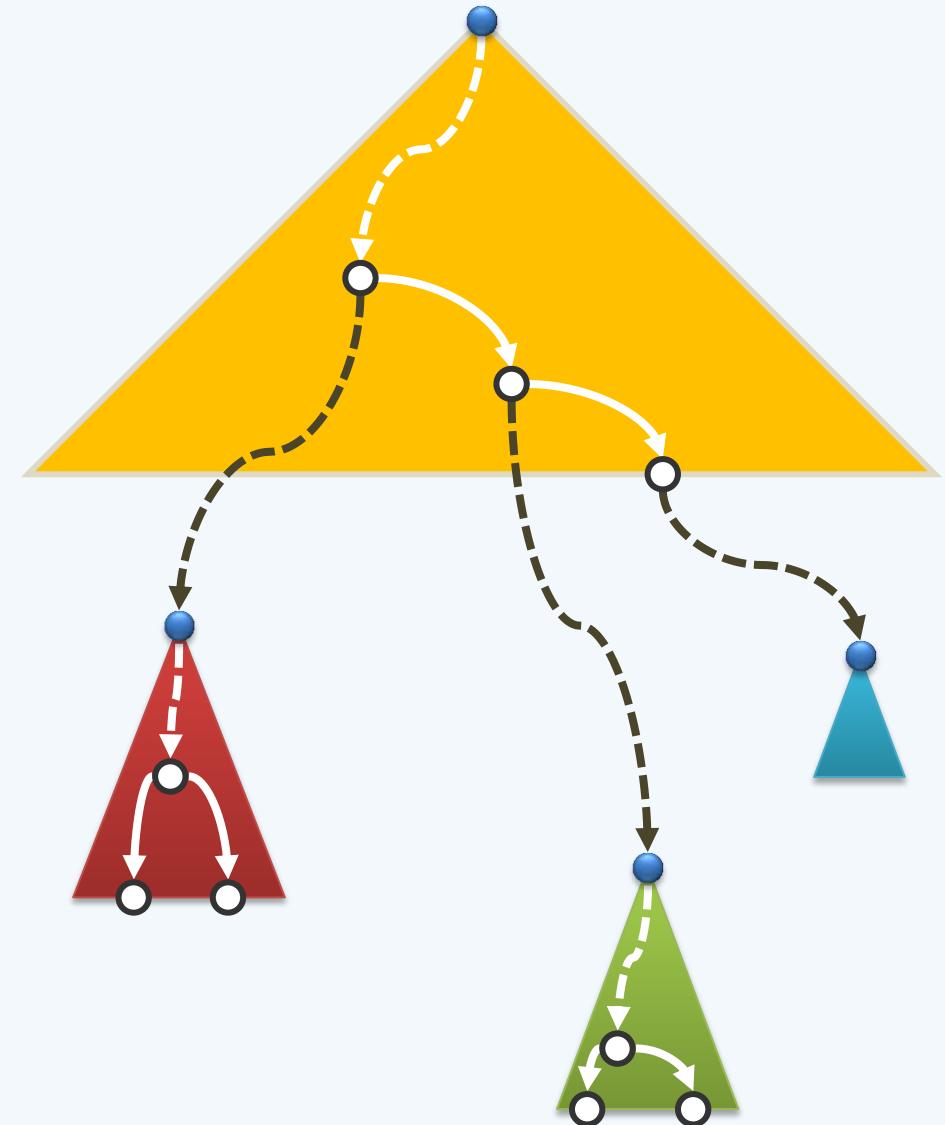
Storage

- ❖ Each input point
is stored in
 $\mathcal{O}(\log n)$ y-trees
- ❖ A 2-level search tree
for n points in the plane
needs $\mathcal{O}(n \log n)$ space



Preprocessing Time

- ❖ A 2-level search tree
for n points in the plane
can be built
in $\Theta(n \log n)$ time



Query Time

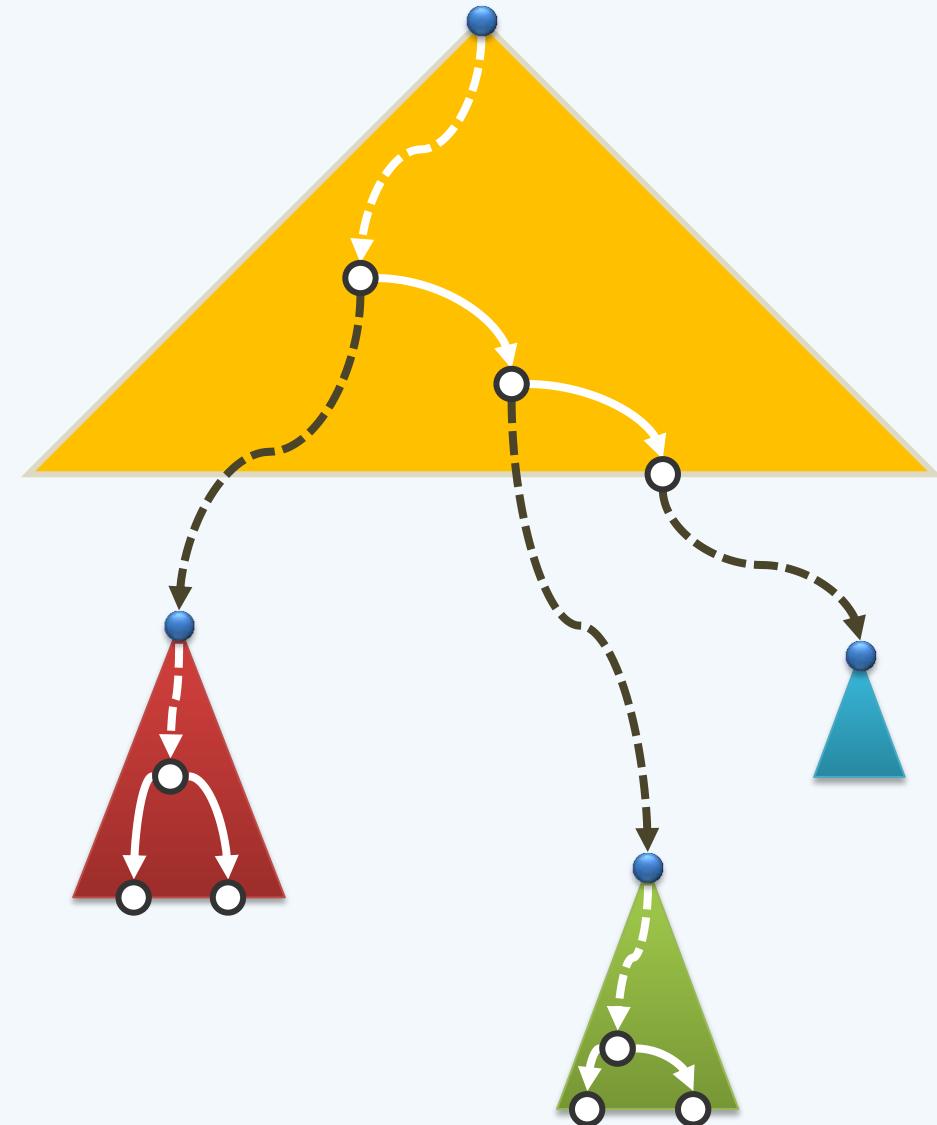
Claim:

A 2-level search tree

for n points in the plane

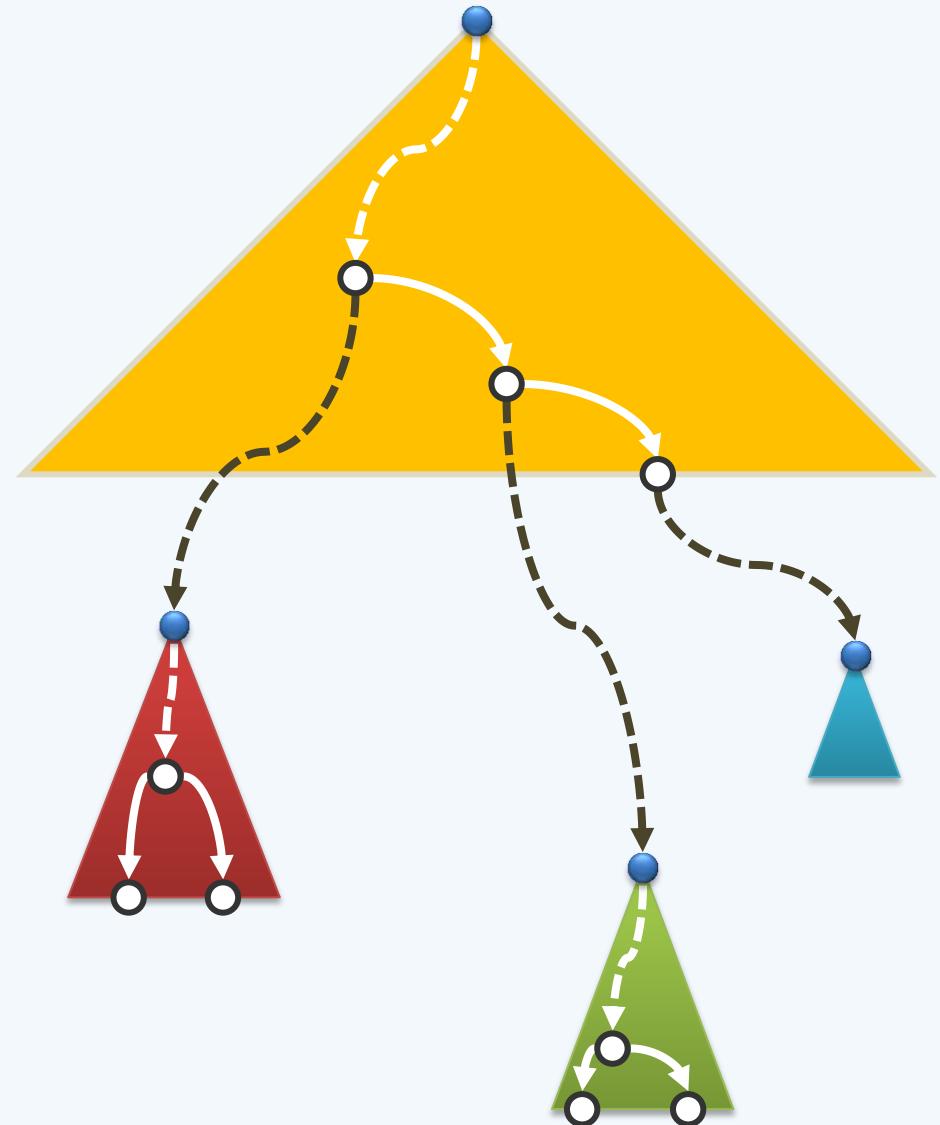
answers each planar range query

in $\mathcal{O}(r + \log^2 n)$ time



Query Time

- ❖ The x -range query needs $\Theta(\log n)$ time to locate the $\Theta(\log n)$ nodes representing the canonical subsets
- ❖ Then for each of these nodes, a y -range search is invoked, which needs $\Theta(\log n)$ time



Beyond 2D

❖ Let S be a set of n points in \mathcal{E}^d , $d \geq 2$

- A d -level tree for S uses $\mathcal{O}(n \log^{d-1} n)$ storage

- Such a tree can be constructed

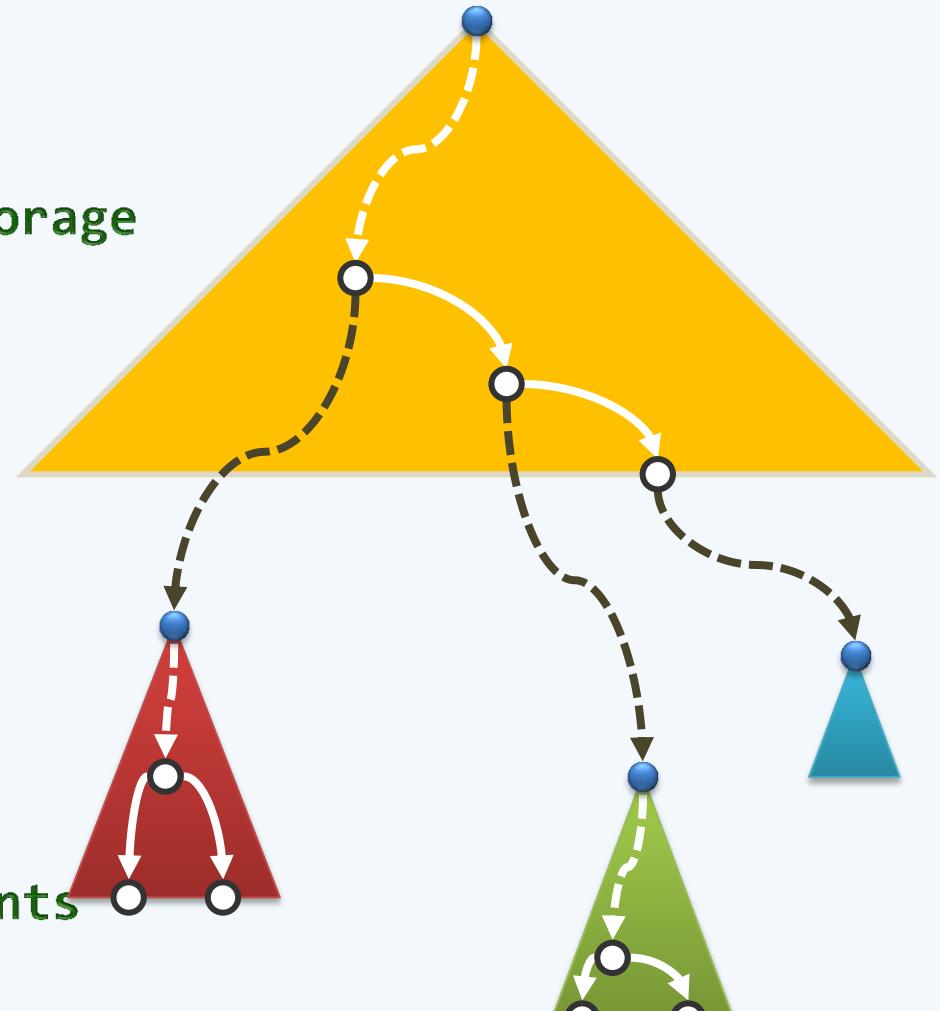
- in $\mathcal{O}(n \log^{d-1} n)$ time

- Each rectangular range query of S can be answered

- in $\mathcal{O}(r + \log^d n)$ time, where

- r is the number of reported points

❖ For planar case, can the query time be improved to, say, $\mathcal{O}(\log n)$?



Advanced Balanced Search Tree

Range Tree

Y-Lists

邓俊辉

deng@tsinghua.edu.cn

BBST<BBST<T>>

While we descend the search in the x -tree,

for each node, we need to

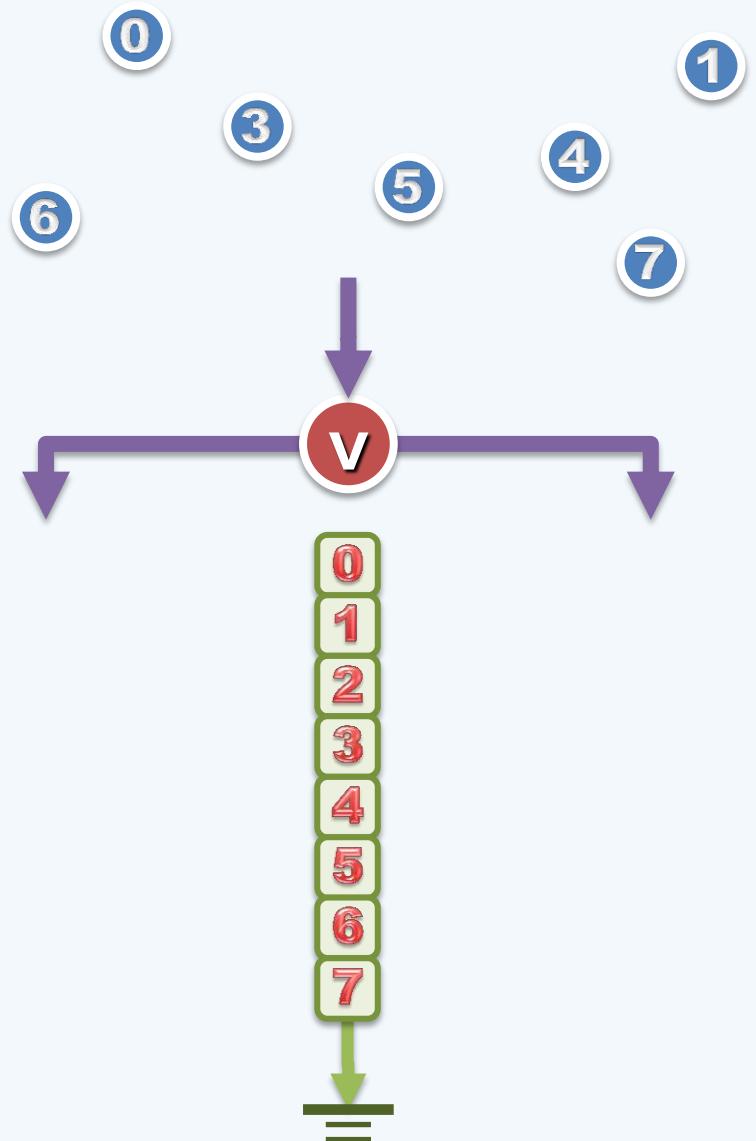
search the corresponding y -tree

It is this **combination** that leads to

the squaring of the logarithms

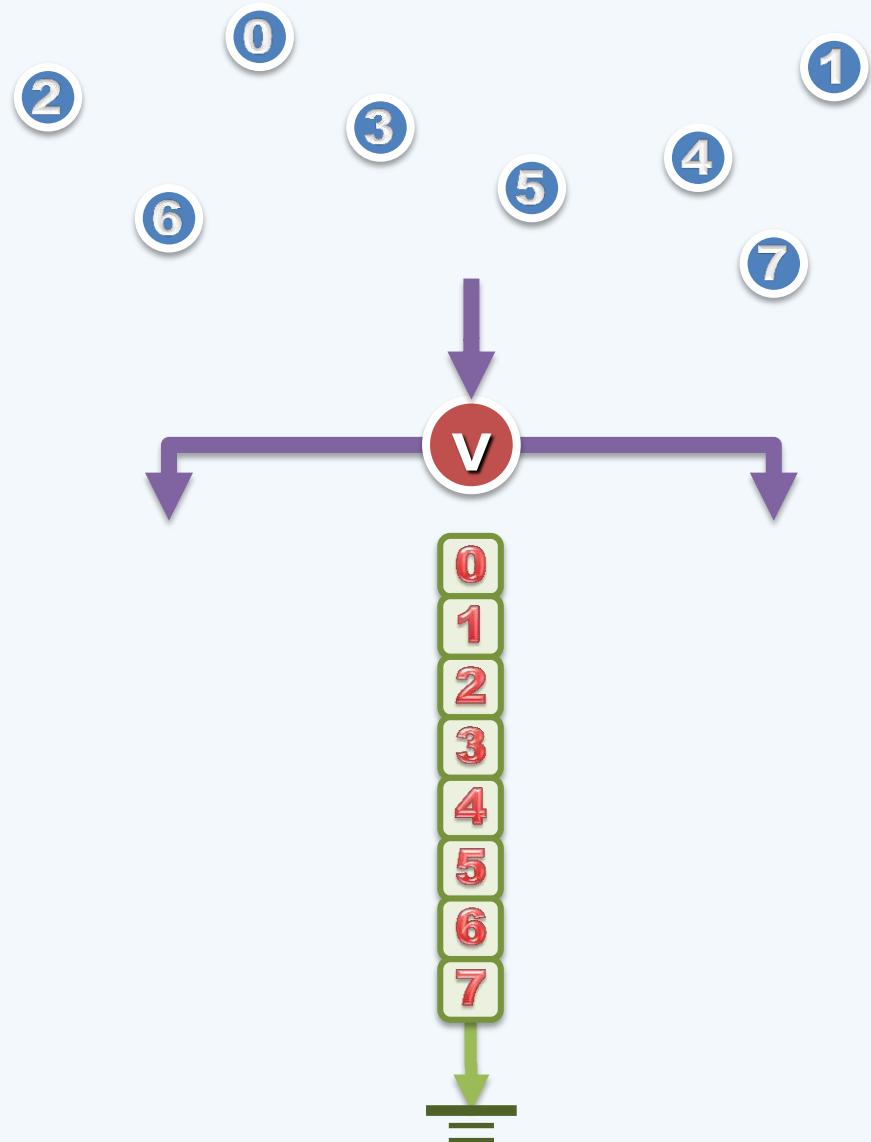
If each y -tree can be searched in $\Theta(1)$ time,

the second **logn** factor will be eliminated!



BBST<Vector<T>>

- ❖ For an easier visualization,
let's regard each y -tree, equivalently,
as a sorted array (called y -list)
- ❖ Actually, this is also
a more efficient way
to implement all y -trees



Advanced Balanced Search Tree

Range Tree

Coherence

邓俊辉

deng@tsinghua.edu.cn

⦿ Observe further that, for each query

- we need to **repeatedly** search

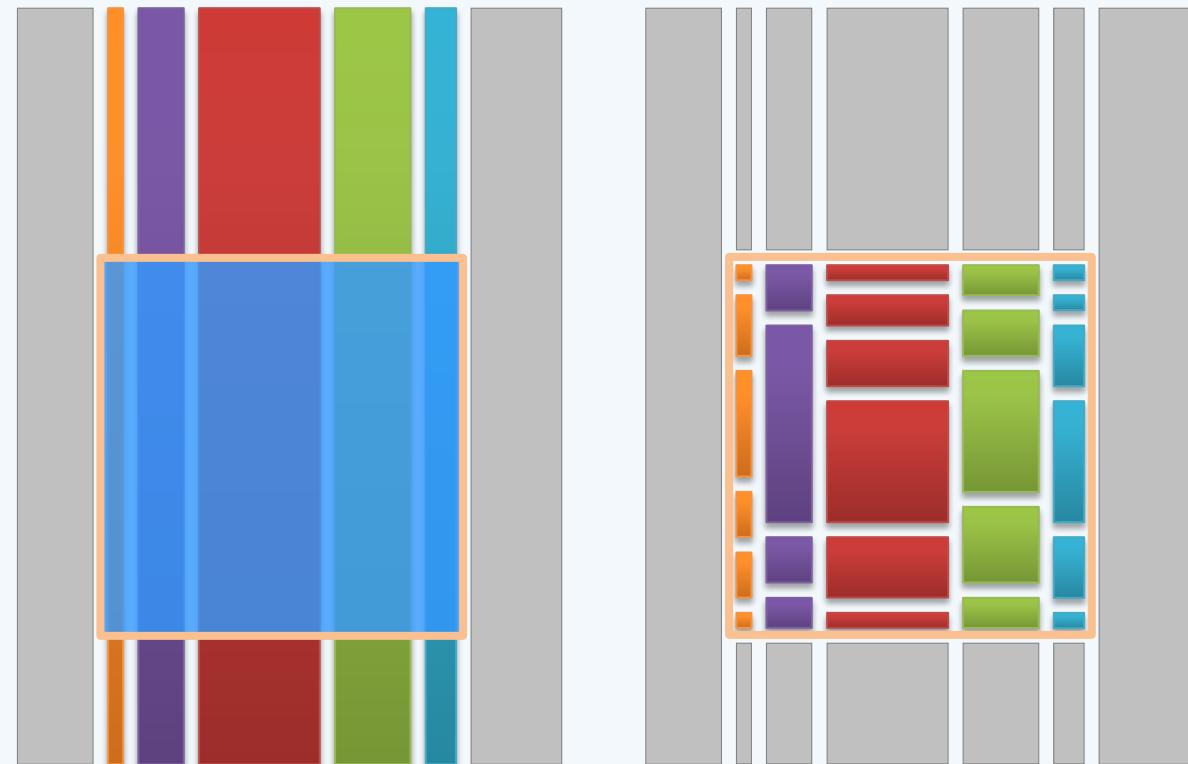
different lists/y-trees,

- but always with

the **same** key

❖ However, such an essential fact

is not used yet



Advanced Balanced Search Tree

Range Tree

Idea

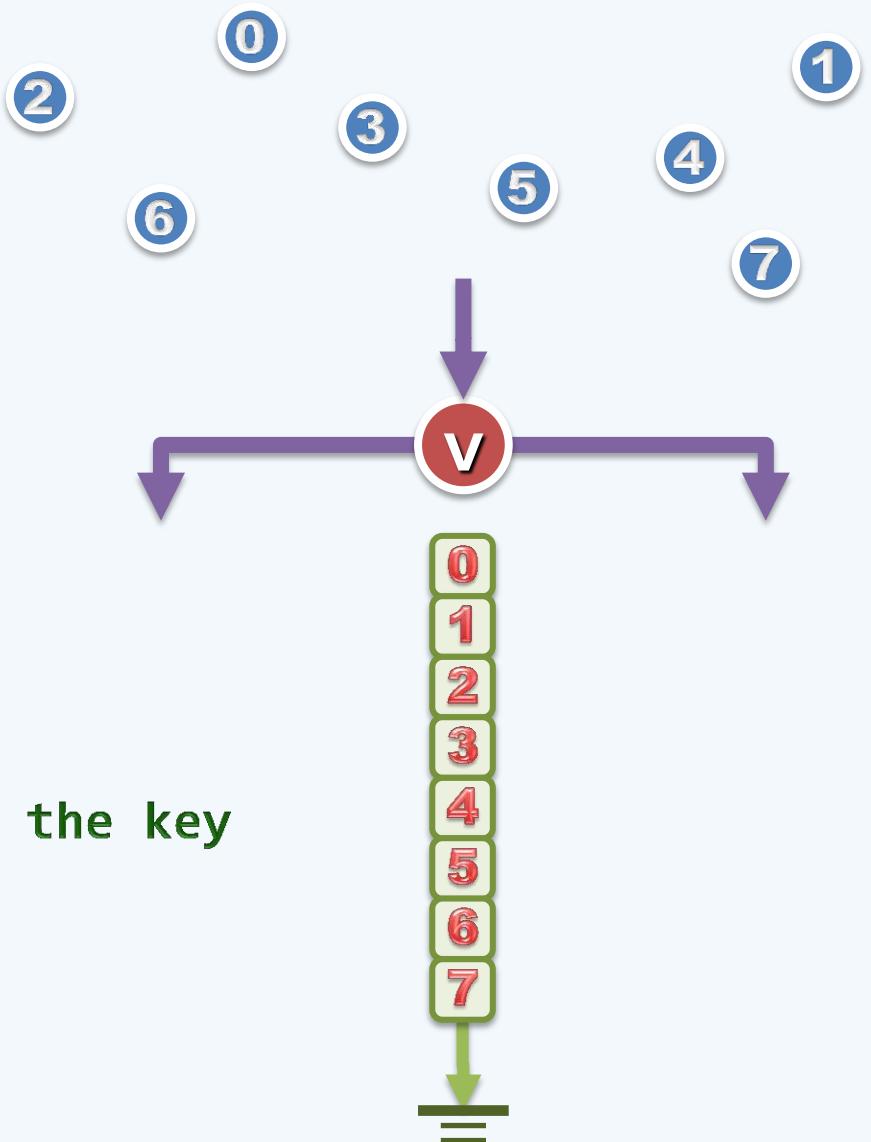
邓俊辉

deng@tsinghua.edu.cn

❖ So the idea for an improvement is that we merge all the different lists into a **single massive** list

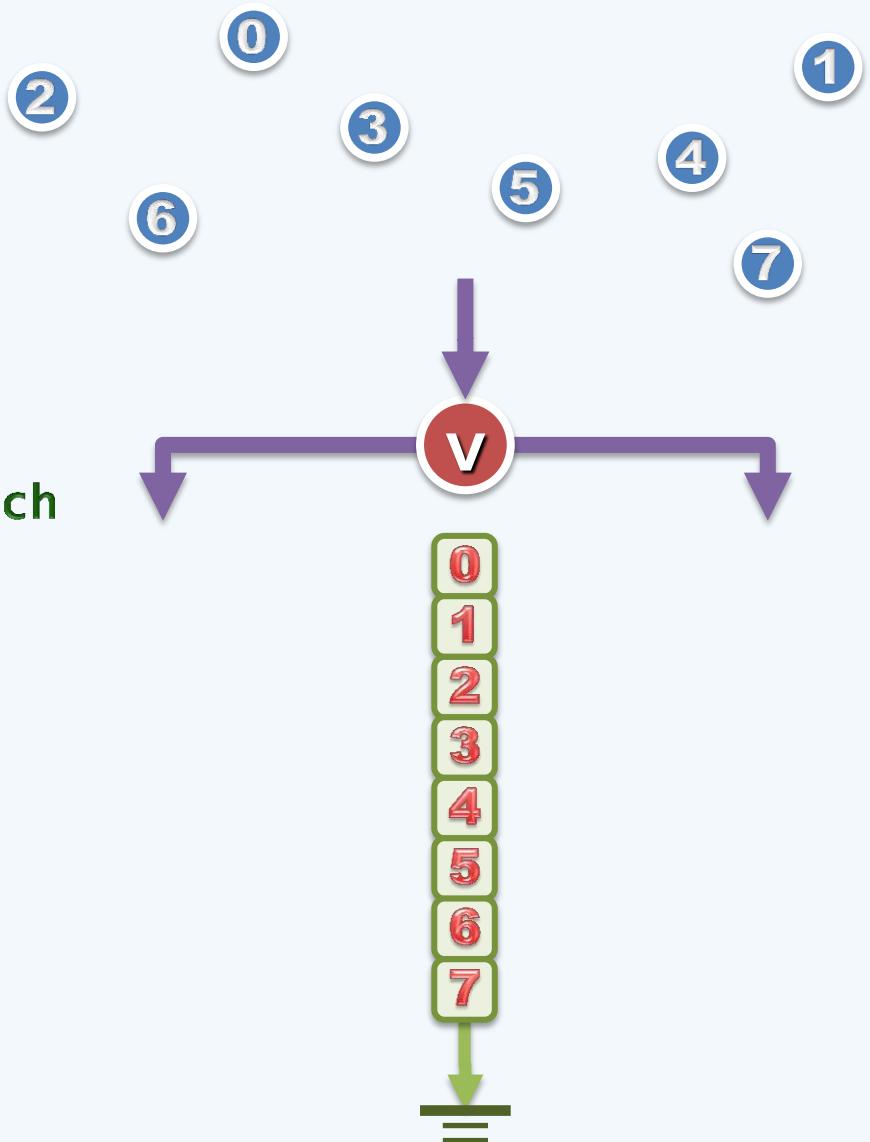
∴ Thus, to answer a planar query, we can

- do a **global search** in this list in **$O(\log n)$** time, and then
- use the information about the location of the key to answer each of the remaining y-queries in **$O(1)$** time



- ❖ In our case, the massive list on which we will do one search is the entire set of points, sorted by their y-coordinates
- ❖ We will do one expensive ($\mathcal{O}(\log n)$ time) search on the **y**-list for the root

// after this, however, we claim that ...
- ❖ While descending the **x**-tree, we can keep track of the position of **y**-range in each auxiliary list in **constant** time



Advanced Balanced Search Tree

Range Tree

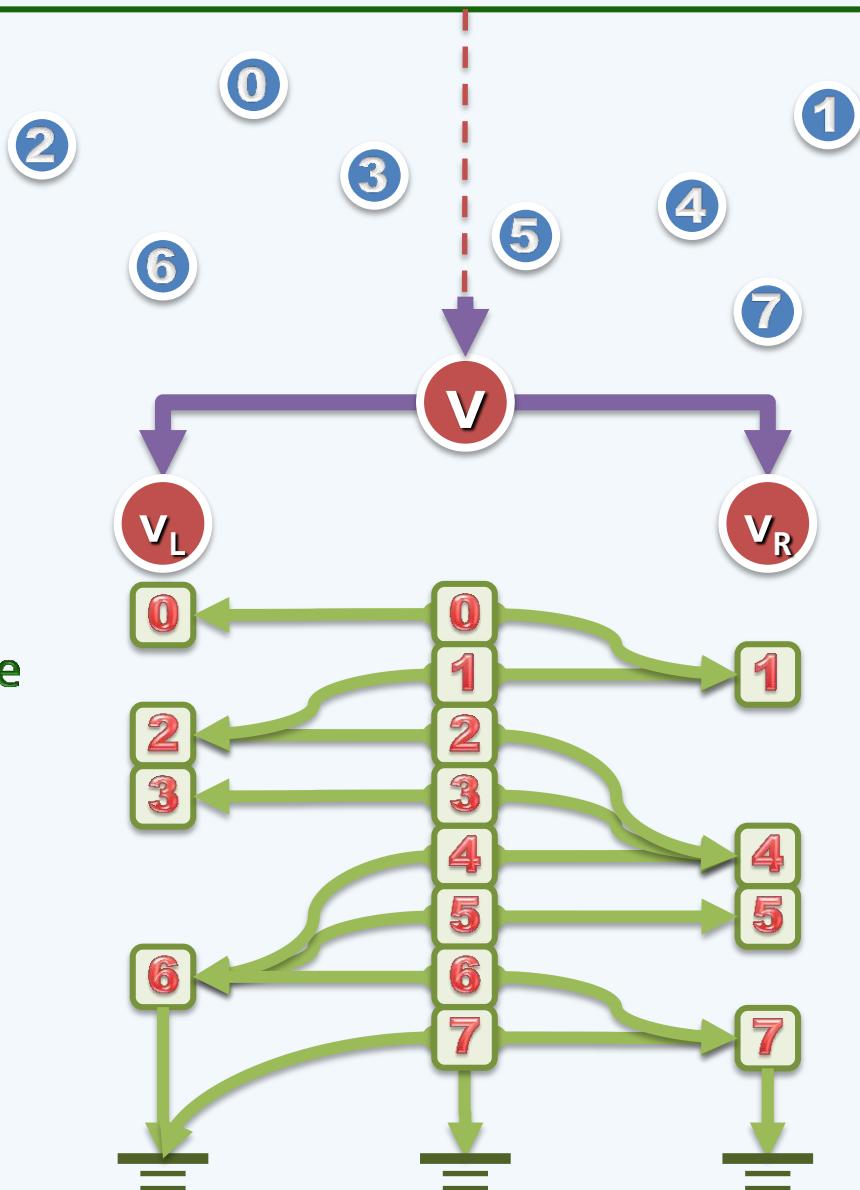
Fractional Cascading

邓俊辉

deng@tsinghua.edu.cn

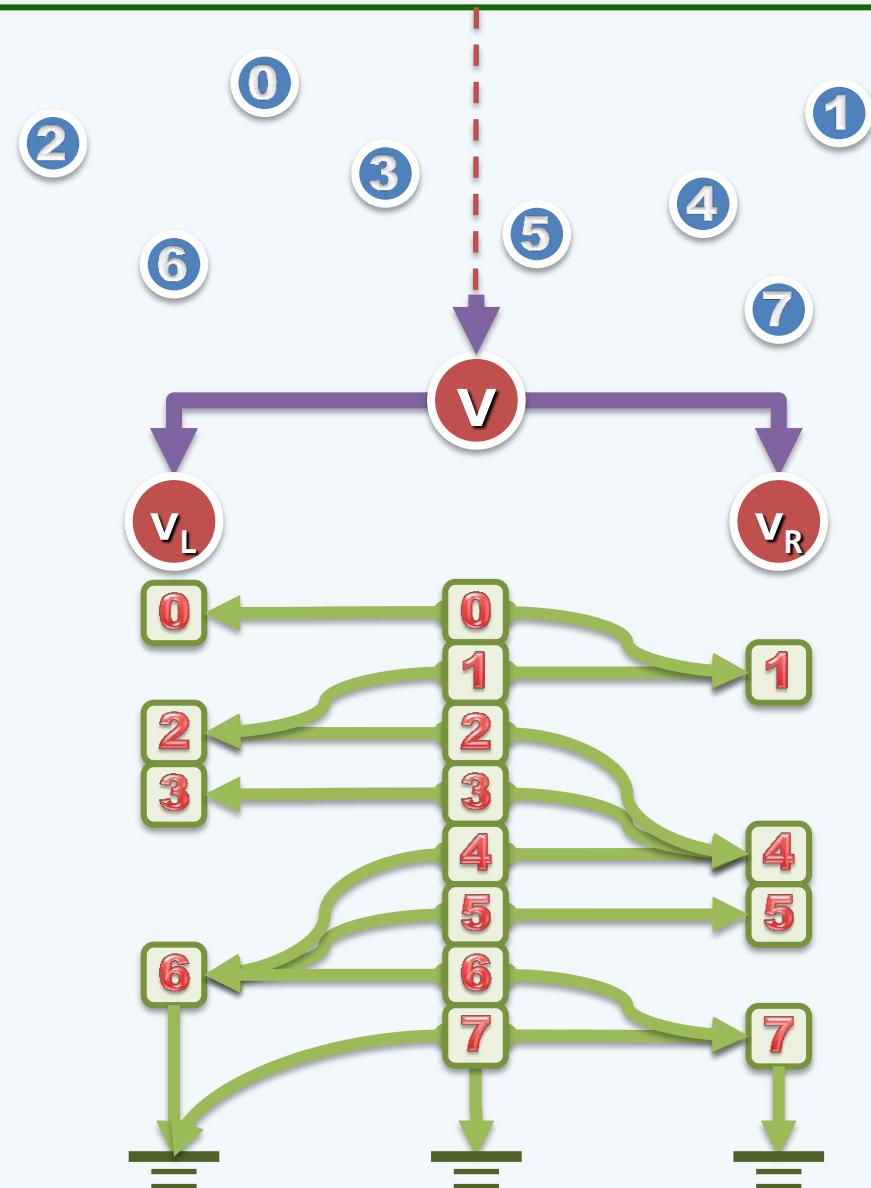
Merge/Split

- ❖ Let v be an internal node in the x -tree with v_L/v_R its left/right child resp.
- ❖ Let A_v be the y -list for v and A_L/A_R be the y -lists for its children
- ❖ Assuming no duplicate y -coordinates, we have
 - A_v is the disjoint union of A_L and A_R , and hence
 - A_v can be obtained by merging A_L and A_R (in linear time)



Structure

- ❖ For each item in A_v ,
we store two pointers to
the item of
equal or larger value
in A_L and A_R resp.
- ❖ When there is no such item,
the pointer is NULL



Fractional Cascading

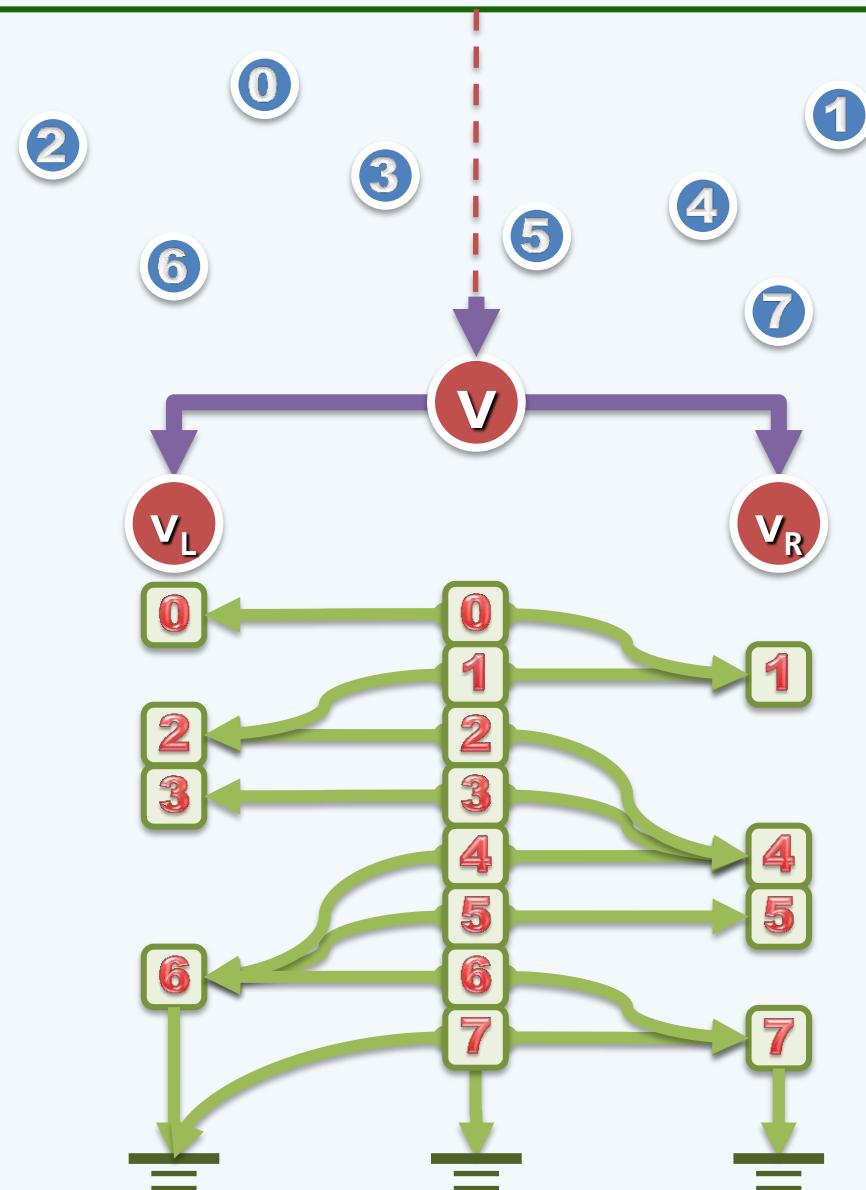
For any y -query with q_y ,

once we know its entry in A_v ,

we can determine its entry

in either A_L or A_R

in $\Theta(1)$ additional time



Advanced Balanced Search Tree

Range Tree

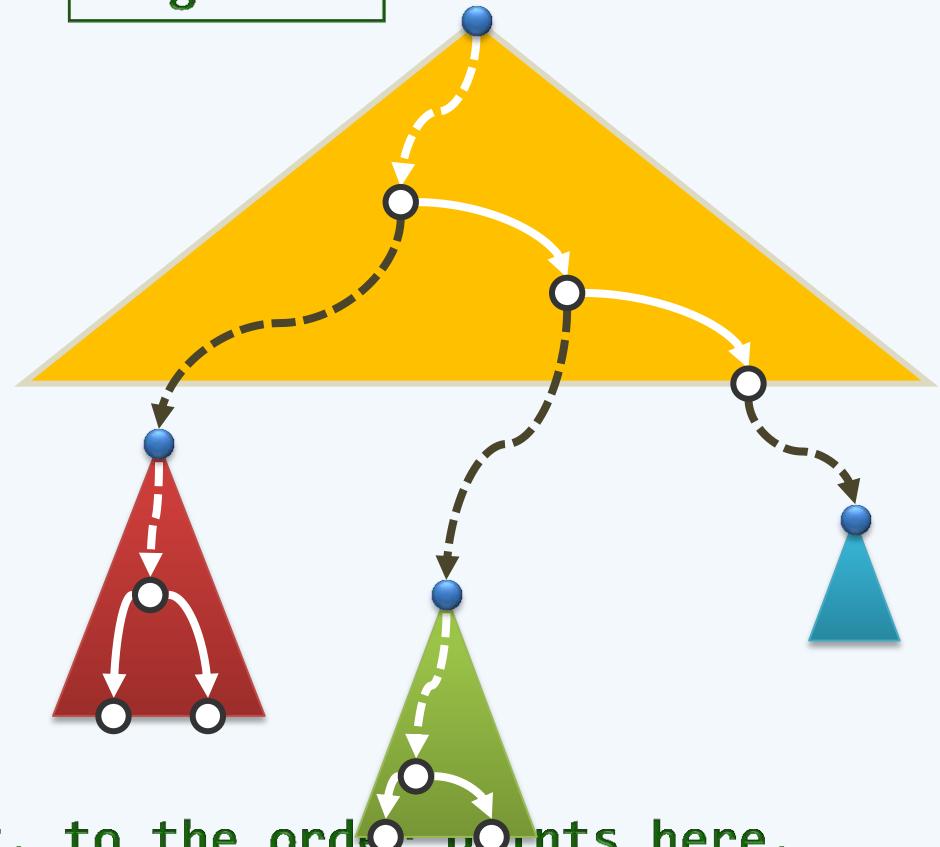
Complexity

邓俊辉

deng@tsinghua.edu.cn

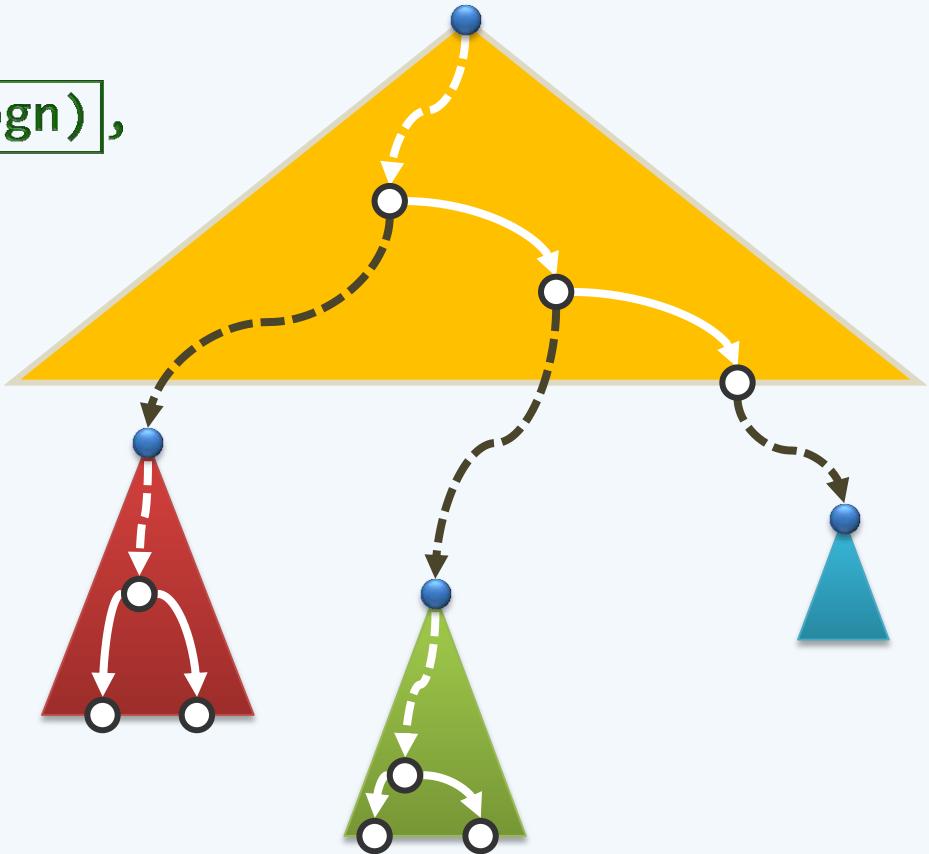
Complexity

- ❖ An MLST with fractional cascading is called a range tree
- ❖ At the root of the main tree,
 - we need to perform a binary search with all the y -values to determine which points lie within this interval, and
 - it requires $\mathcal{O}(\log n)$ time
- ❖ For all subsequent levels, once we know where the y -interval falls w.r.t. to the order points here, we can drop down to the next level in $\mathcal{O}(1)$ time



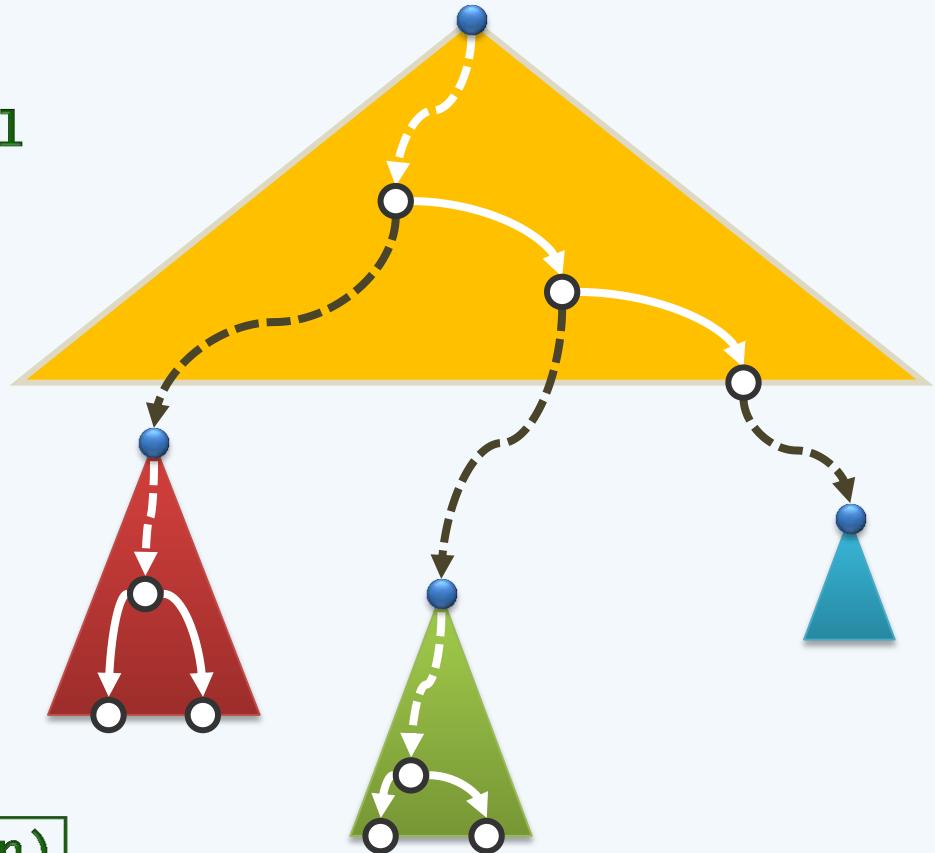
Complexity

- ❖ Thus, as with fractional cascading,
the time for cascading searches is $\Theta(2\log n)$,
rather than $\Theta(\log^2 n)$
- ❖ Given a set of n points in the plane,
orthogonal range queries
 - can be answered in $\Theta(r + \log n)$ time
 - from a data structure of size $\Theta(n \log n)$,
 - which can be constructed in $\Theta(n \log n)$ time



Beyond 2D

- ❖ Unfortunately, it turns out that the trick of fractional cascading can **only** be applied to the **last** level of the search structure, because all other levels need the **full** tree search to compute canonical sets
- ❖ Given a set of n points in \mathcal{E}^d , an orthogonal range query
 - can be answered in $\mathcal{O}(r + \log^{(d-1)}n)$ time
 - from a data structure of size $\mathcal{O}(n * \log^{(d-1)}n)$,
 - which can be constructed in $\mathcal{O}(n * \log^{(d-1)}n)$ time



Advanced Balanced Search Tree

Interval Tree

Stabbing Query

邓俊辉

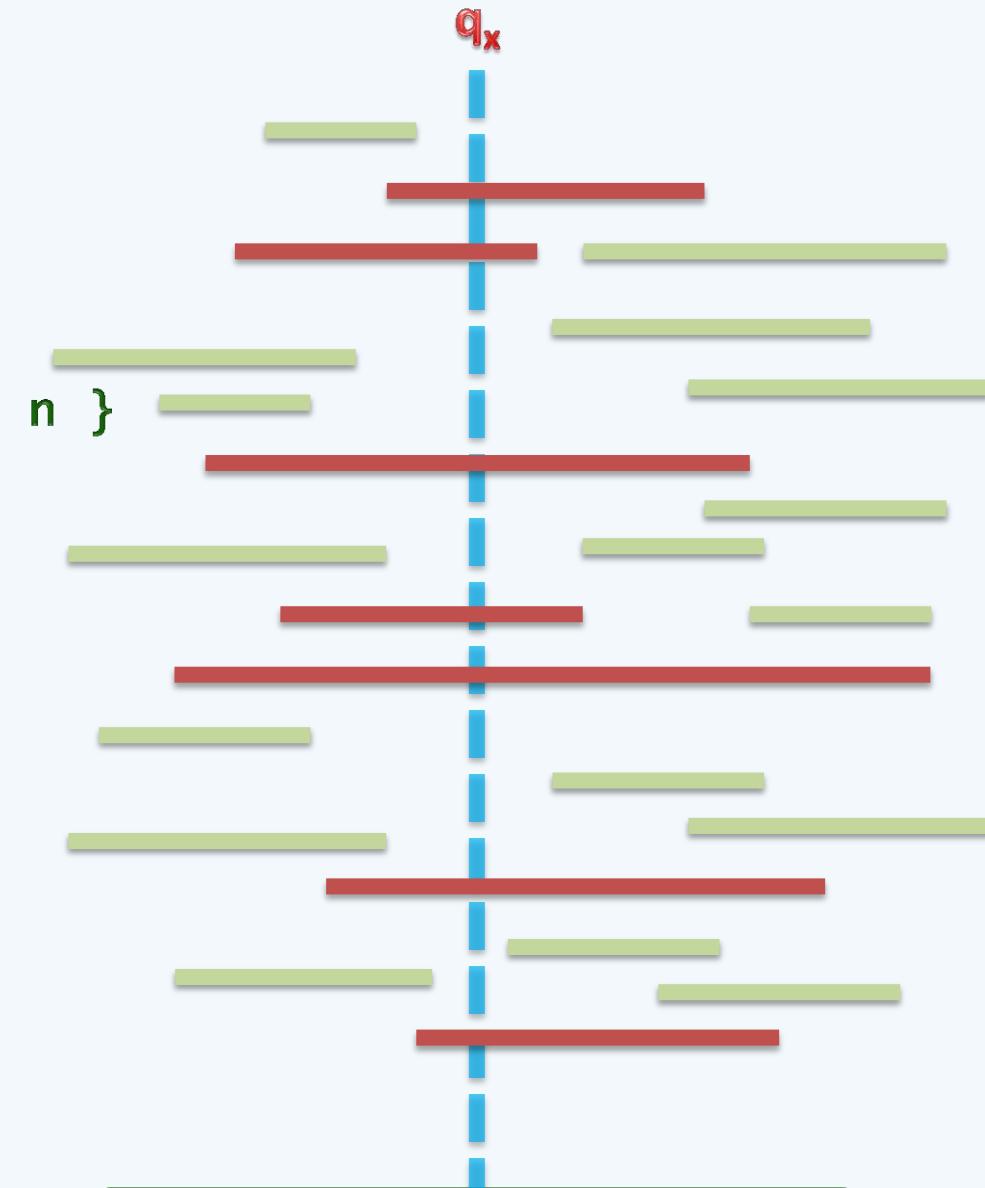
deng@tsinghua.edu.cn

Input

- ❖ A set of **intervals** in general position on the **x**-axis

$$S = \{ s_i = [x_i, x'_i] \mid 1 \leq i \leq n \}$$

- ❖ A query **point**: q_x



Output

❖ All intervals that contain q_x

$$\{ s_i = [x_i, x_i'] \mid x_i \leq q_x \leq x_i' \}$$

❖ To solve this query,

we will use

a structure named

interval tree ...



Advanced Balanced Search Tree

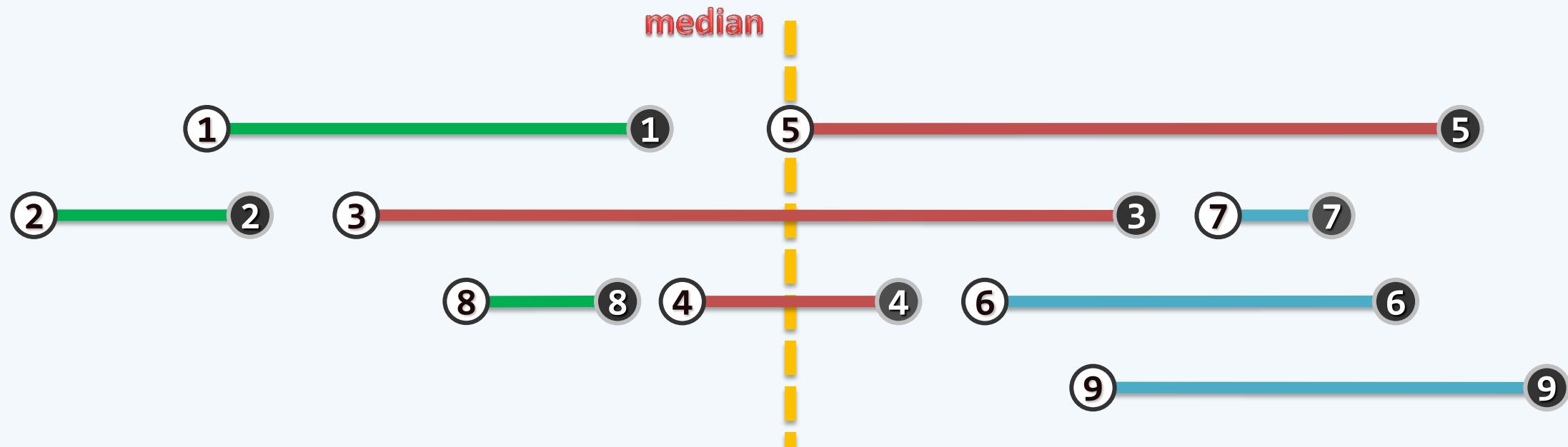
Interval Tree
Construction

邓俊辉

deng@tsinghua.edu.cn

Median

- ❖ Let $S = \{ s_1, \dots, s_n \}$ be the set of intervals
- ❖ Let $P = \partial S$ be the set of all endpoints
 - // by general position assumption, $|P| = 2n$
- ❖ Let $\text{median}(P) = x_{\text{mid}}$ be the median of P



Partitioning (1)

❖ All intervals can be then categorized into 3 subsets

$$S_{\text{left}} = \{ S_i \mid x_{i'} < x_{\text{mid}} \}$$

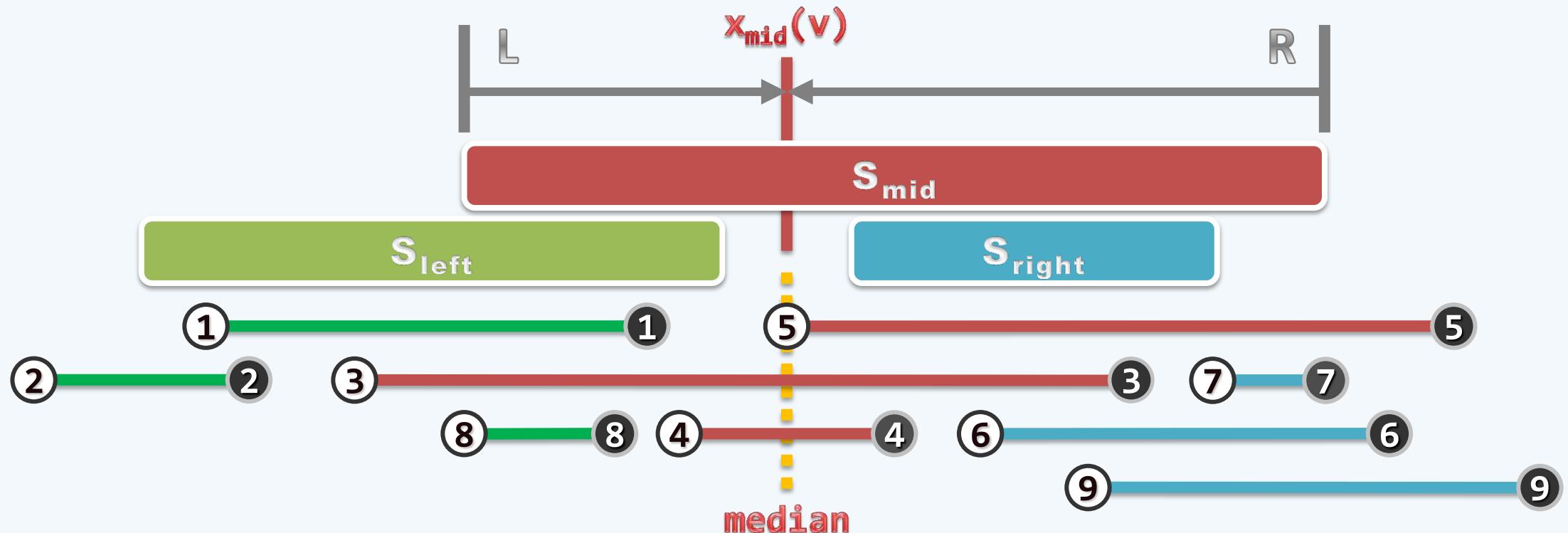
//intervals lying entirely **left** to x_{mid}

$$S_{\text{right}} = \{ S_i \mid x_{\text{mid}} < x_i \}$$

//intervals lying entirely **right** to x_{mid}

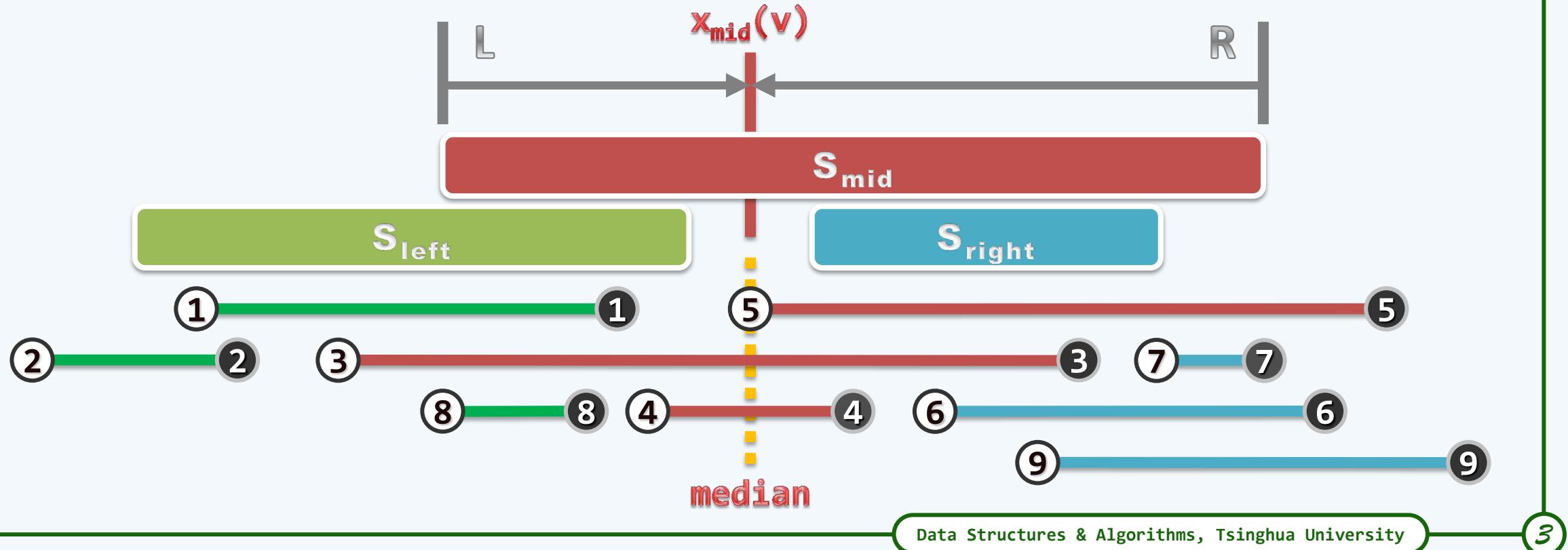
$$S_{\text{mid}} = \{ S_i \mid x_i \leq x_{\text{mid}} \leq x_{i'} \}$$

//intervals that **contain** x_{mid}



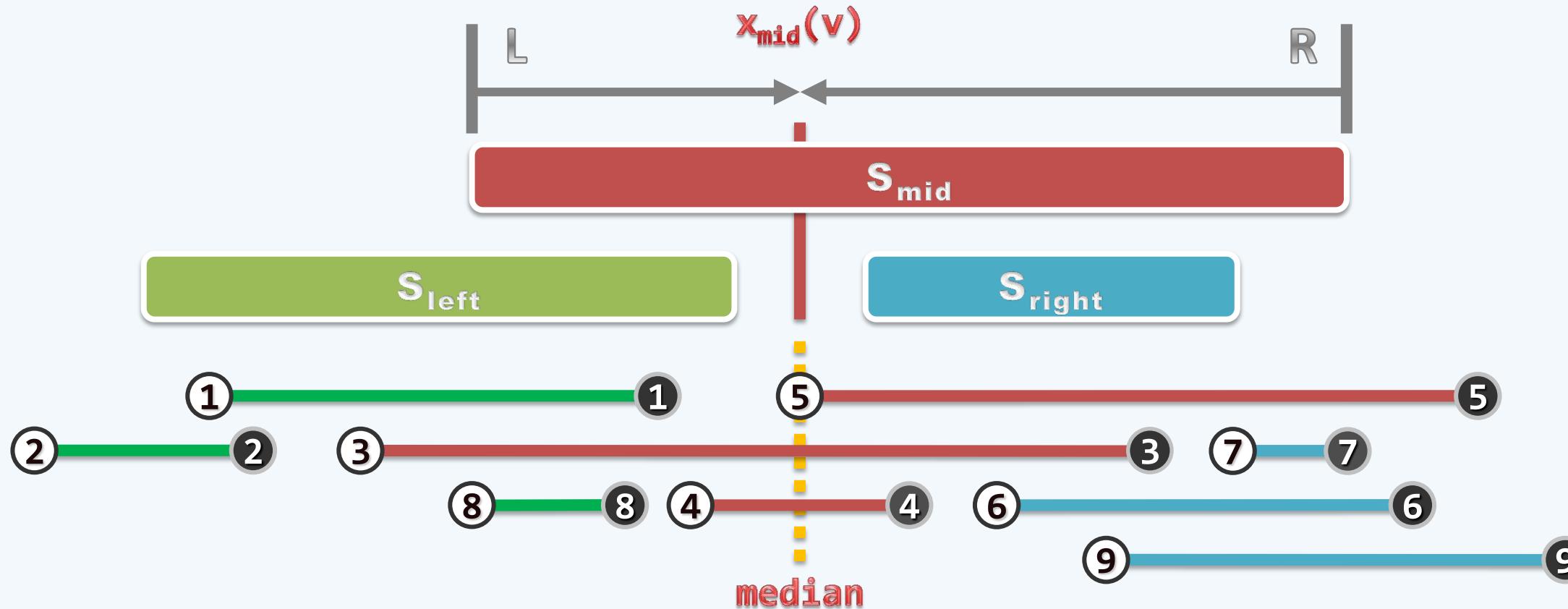
Partitioning (2)

❖ $S_{left/right}$ will be recursively partitioned until they are empty (leaves)



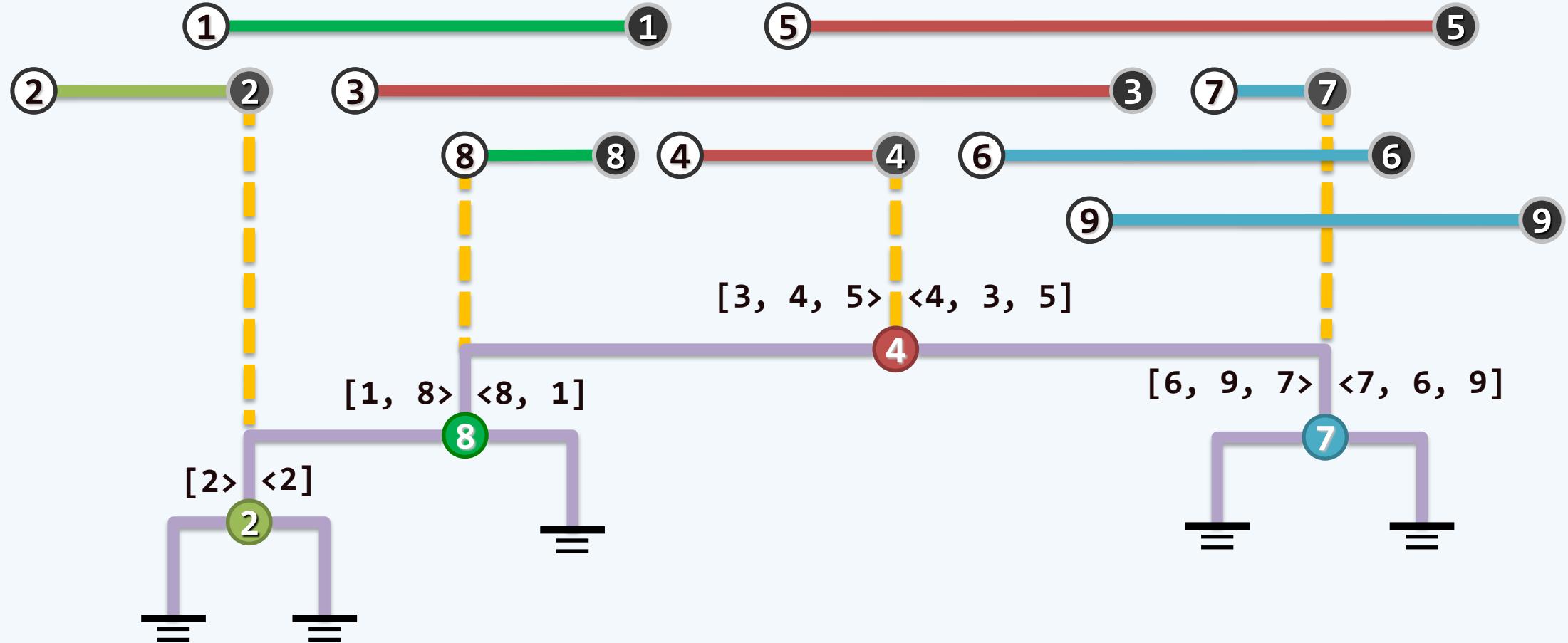
Balance

- ❖ $\max \{ |S_{\text{left}}|, |S_{\text{right}}| \} \leq n/2$
- ❖ Best/worst case: $|S_{\text{mid}}| = n/1$



Associative Lists

❖ $L_{\text{left/right}} = \text{all intervals of } S_{\text{mid}}$ sorted by the **left**/**right** endpoints



Advanced Balanced Search Tree

Interval Tree

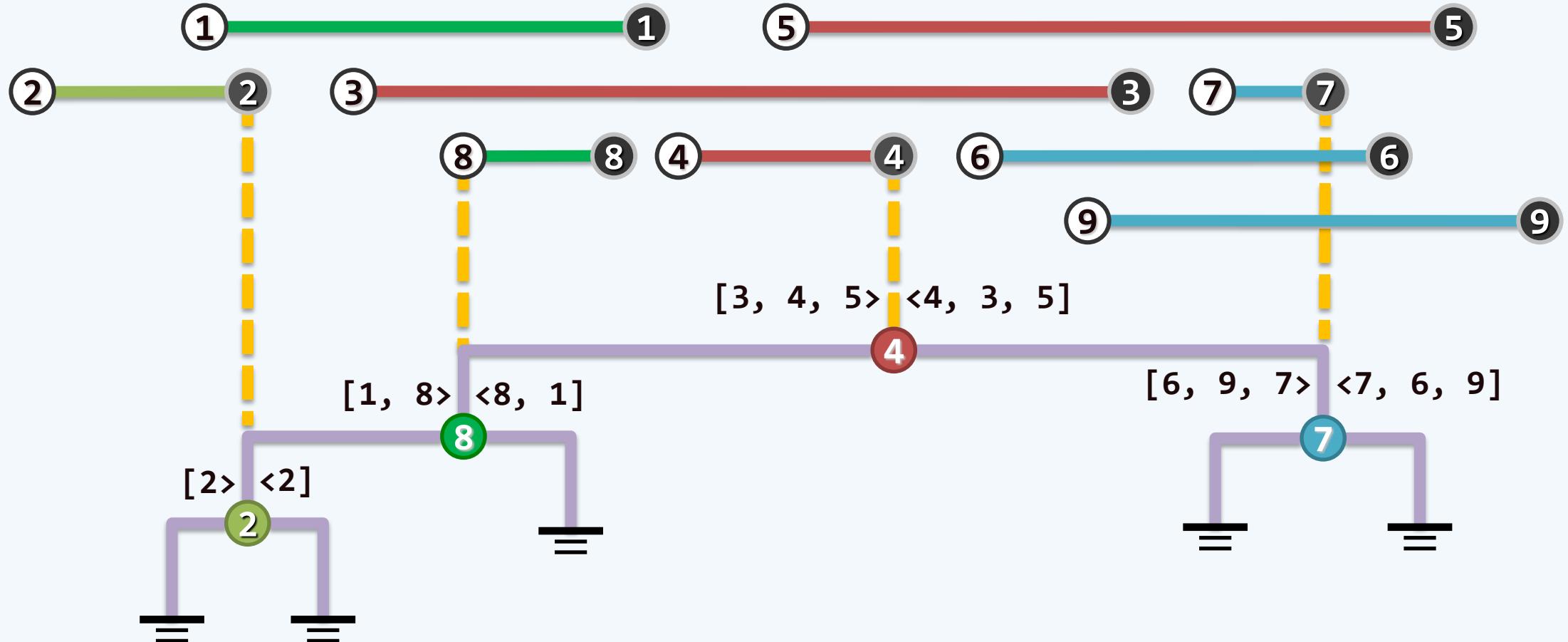
Complexity (1)

邓俊辉

deng@tsinghua.edu.cn

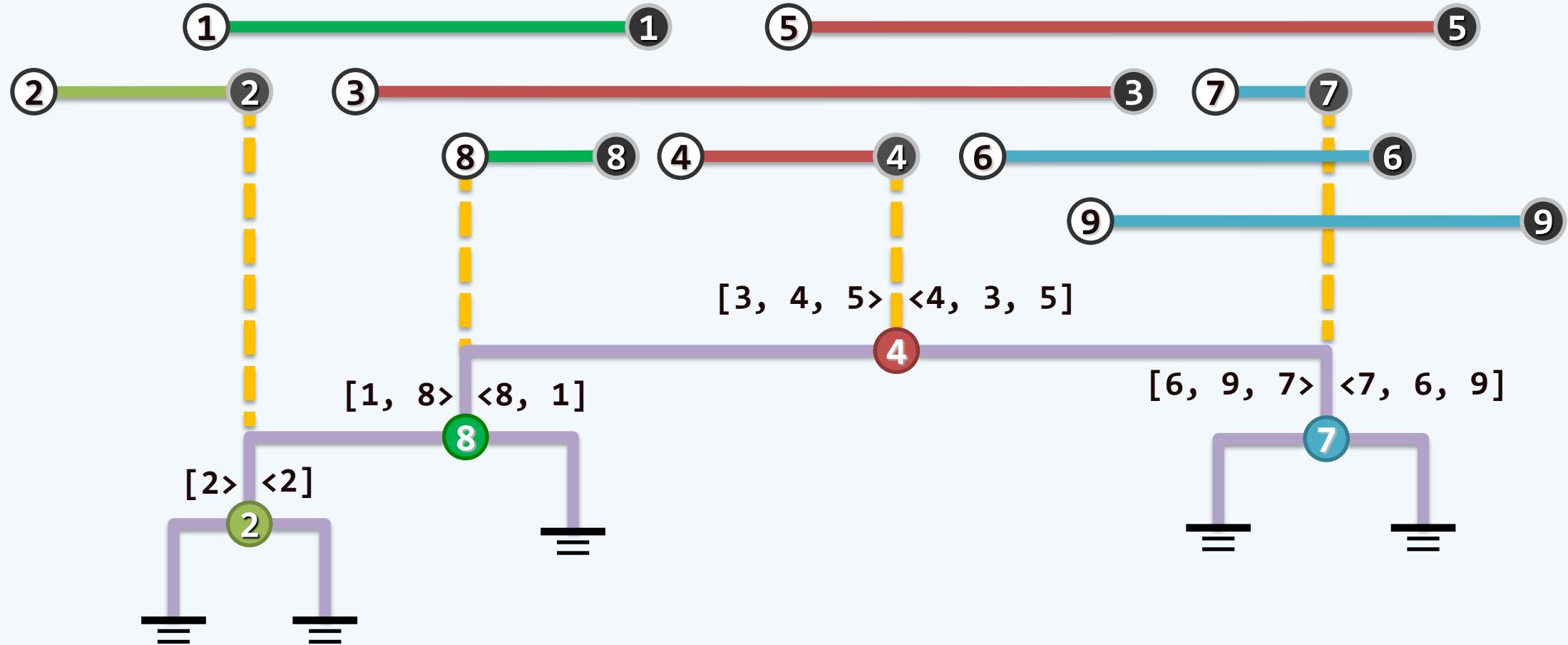
$\mathcal{O}(n)$ Size

❖ Each segment appears twice



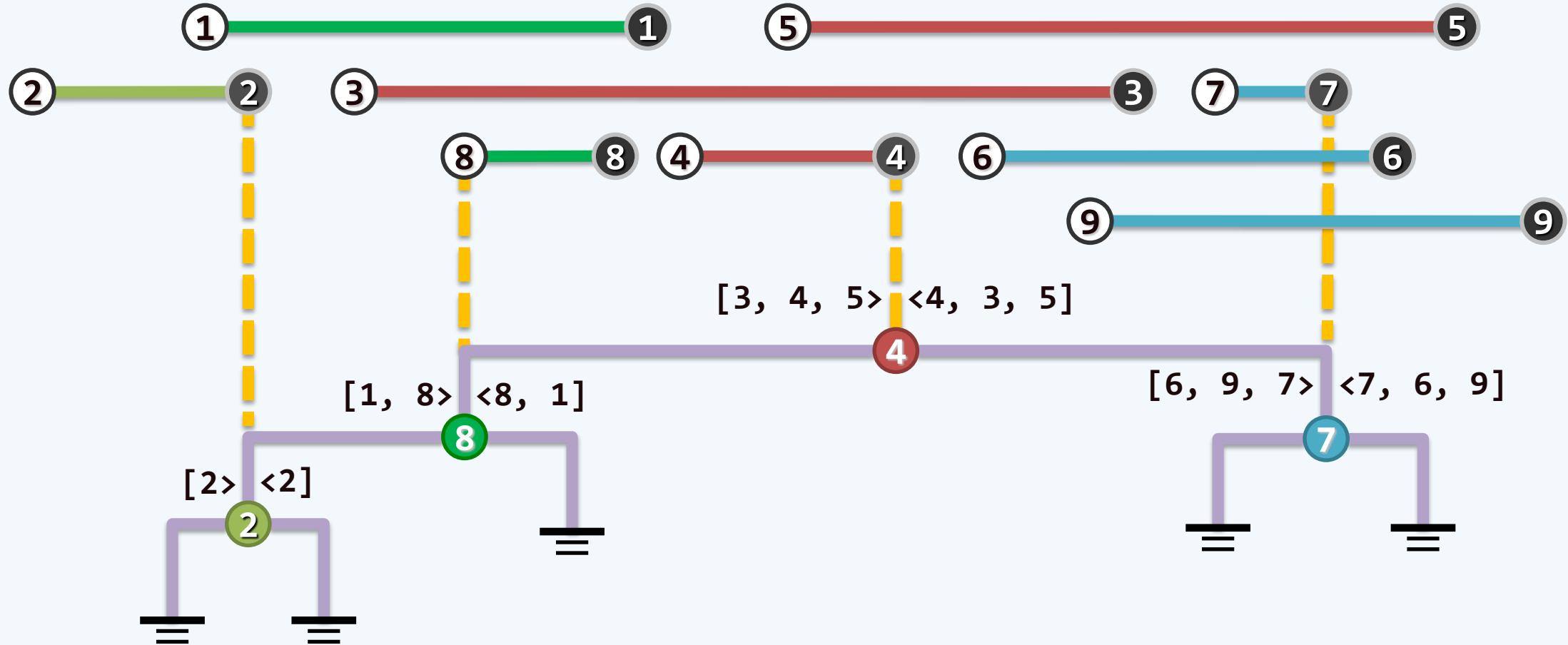
$\mathcal{O}(\log n)$ Depth

❖ Partitionings are done evenly



$\mathcal{O}(n \log n)$ Construction Time

❖ Hint: avoid repeatedly sorting



Windowing Query

Interval Tree
Query

邓俊辉

deng@tsinghua.edu.cn

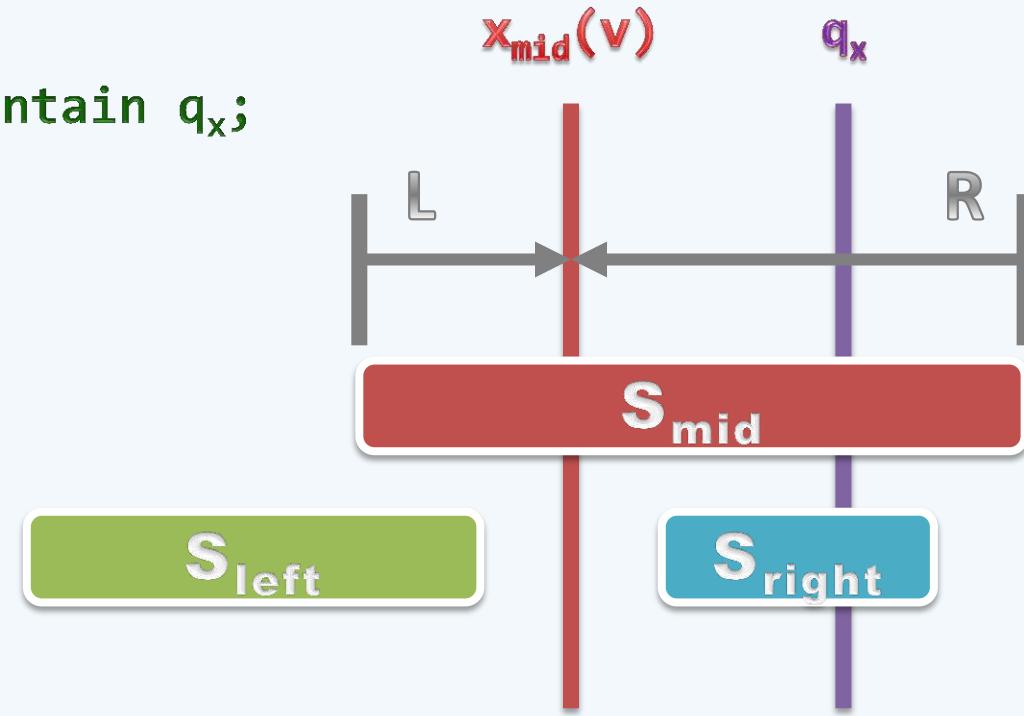
queryIntervalTree(v, q_x)

```

if ( ! v ) return; //base

if ( qx < xmid(v) ) {
    report all segments of Smid(v) that contain qx;
    //by a scan of Lleft(v)
    queryIntervalTree( lc(v), qx );
    //rc(v) can be ignored
}
...

```



queryIntervalTree(v, q_x)

...

```
} else if ( xmid(v) < qx ) {
```

report all segments of S_{mid}(v) that contain q_x;

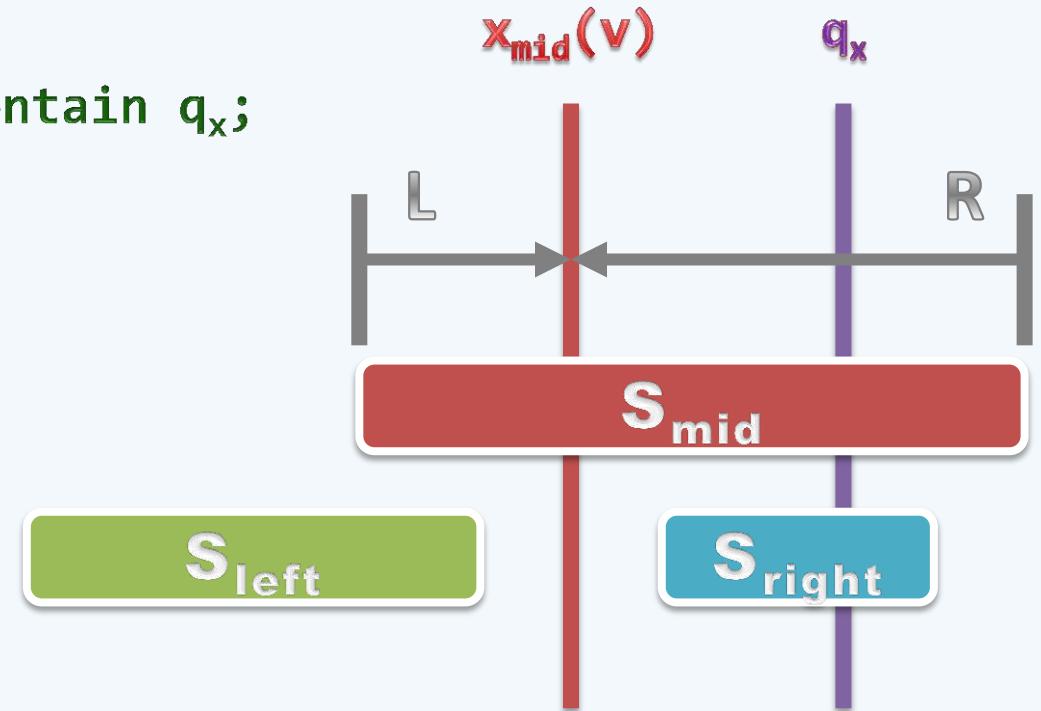
//by a scan of L_{right}(v)

```
queryIntervalTree( rc(v), qx );
```

//lc(v) can be ignored

```
} else //with a probability ≈ 0
```

report all segments of S_{mid}(v); //both rc(v) & lc(v) can be ignored



Windowing Query

Interval Tree

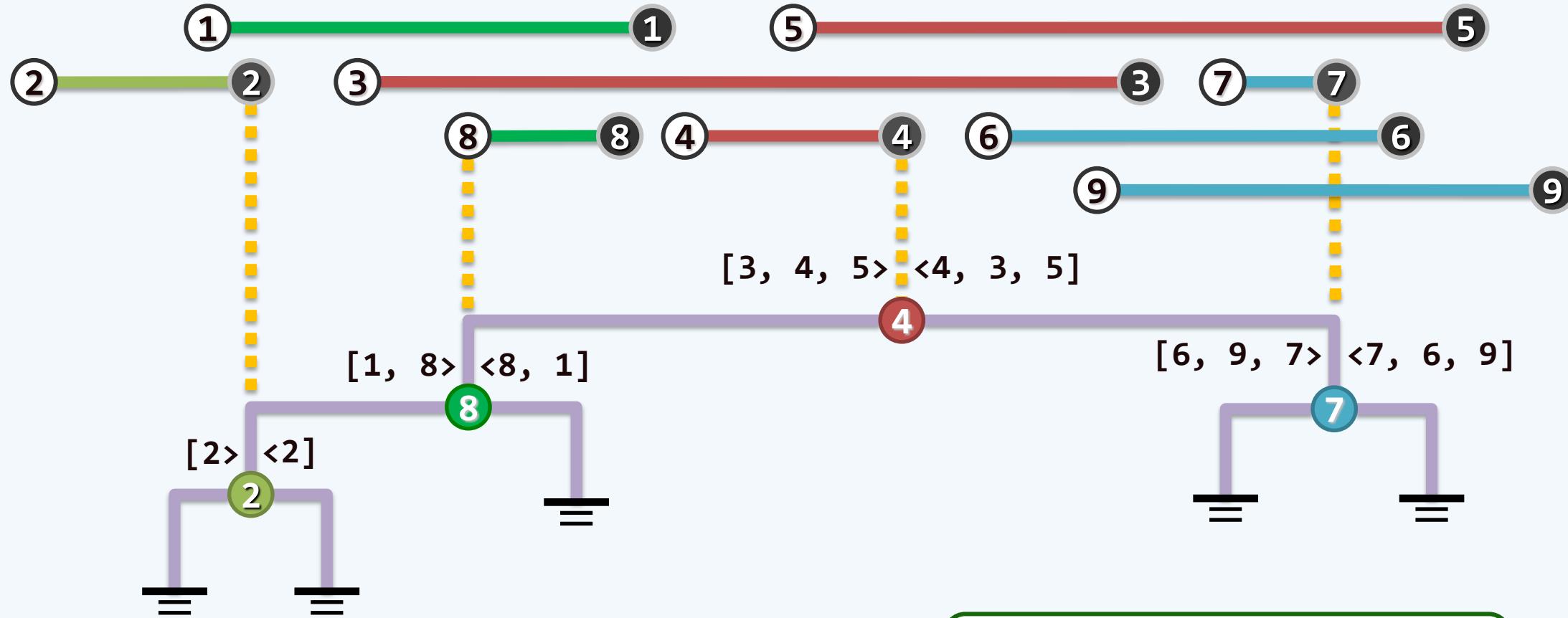
Complexity (2)

邓俊辉

deng@tsinghua.edu.cn

❖ Each query visits $O(\log n)$ nodes

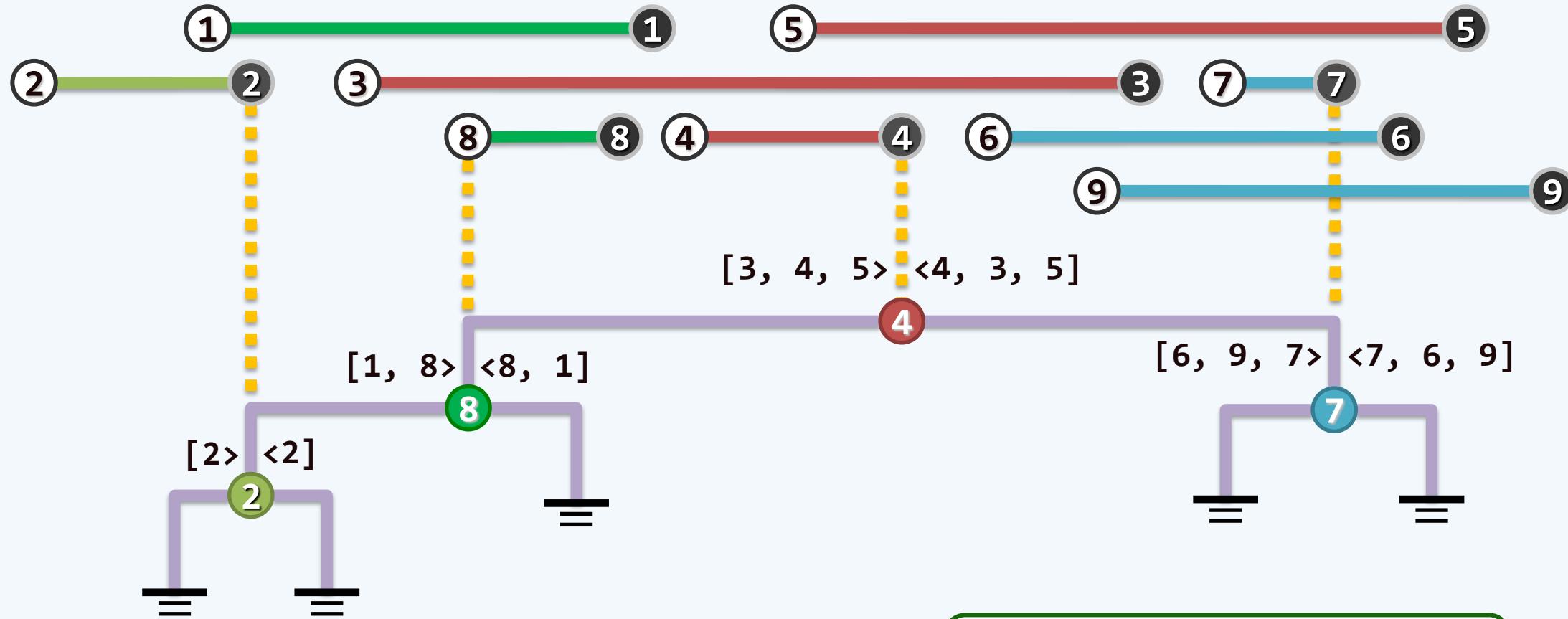
// linear recursion



❖ Query time = $\Theta(r + \log n)$

// r = # intervals visited & reported

❖ It's time now to go back to the 2D version ...



Advanced Balanced Search Tree

Segment Tree

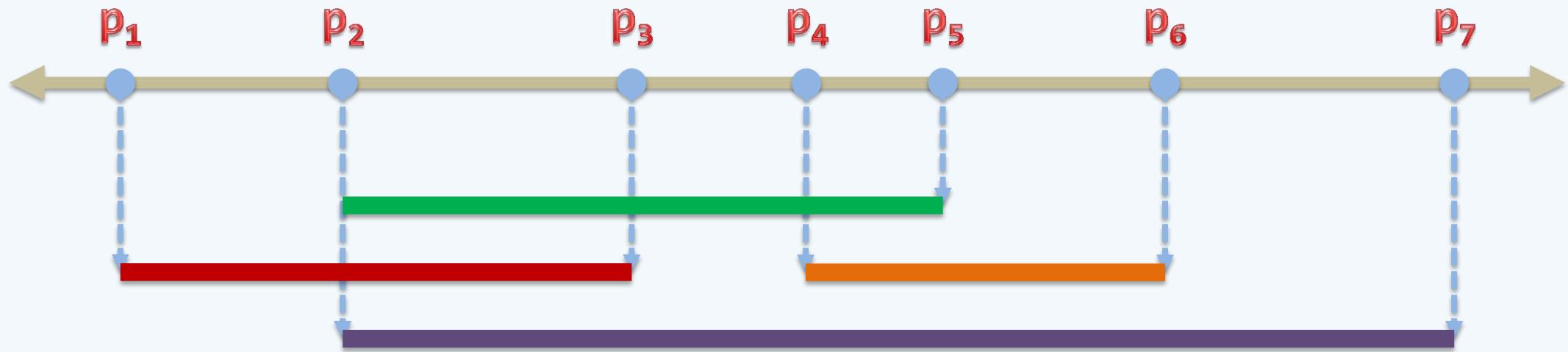
Discretization

邓俊辉

deng@tsinghua.edu.cn

Elementary Interval

- ❖ Let $I := \{ [x_i : x'_i] \mid i = 1, 2, \dots, n \}$ be n intervals on the x -axis
Sort all the endpoints into $\{ p_1, \dots, p_m \}$, $m \leq 2n$

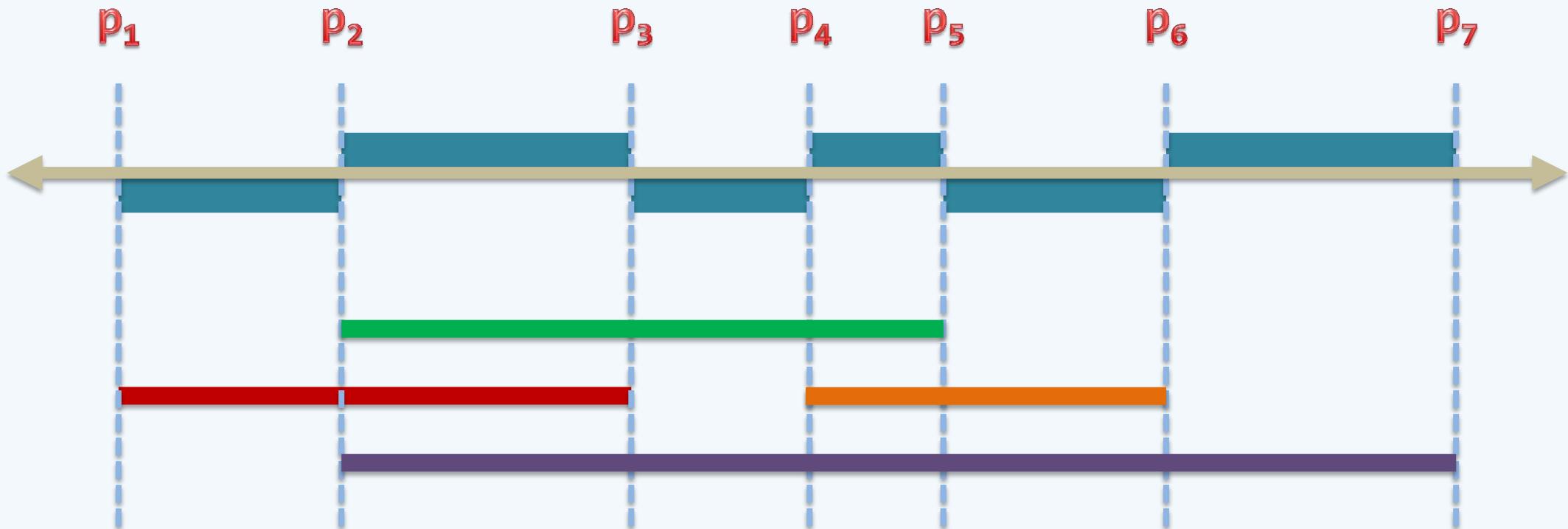


- ❖ $m + 1$ **elementary intervals** are hence defined:

$(\infty, p_1], (p_1, p_2], (p_2, p_3], \dots, (p_{m-1}, p_m], (p_m, +\infty)$

Discretization

Within each EI, all stabbing queries share a same output

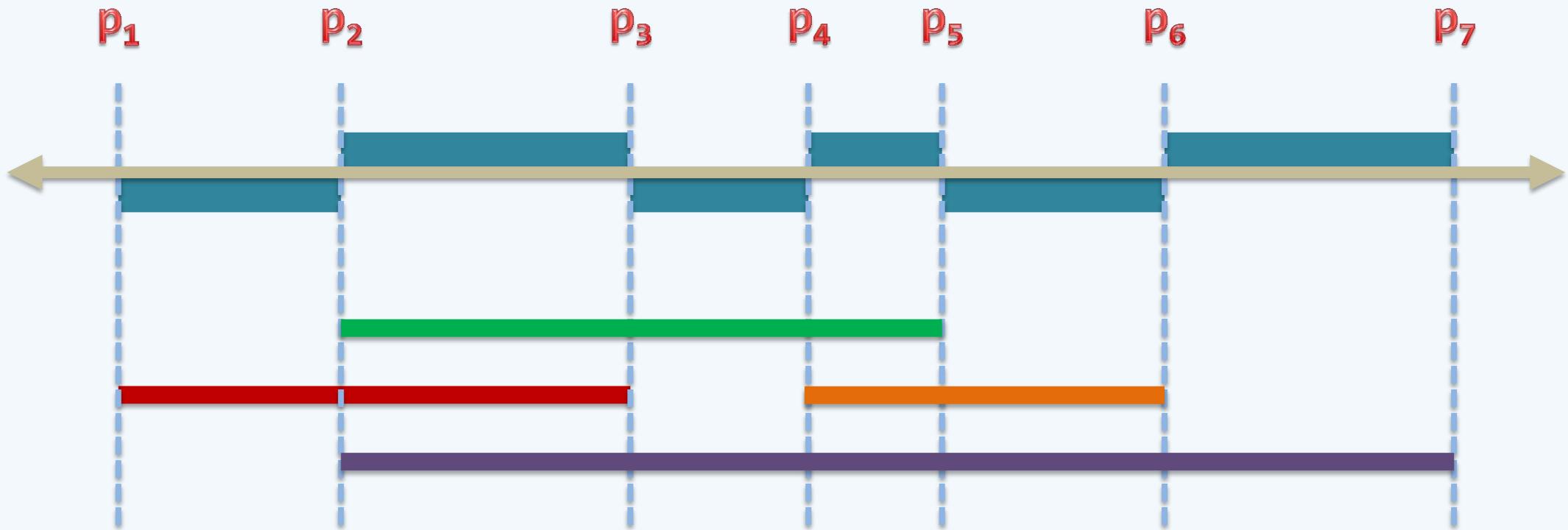


∴ If we **sort** all EI's into a vector and

store the corresponding **output** with each EI, then ...

Sorted Vector

∴ Once a query position is determined, //by an $\mathcal{O}(\log n)$ time binary search

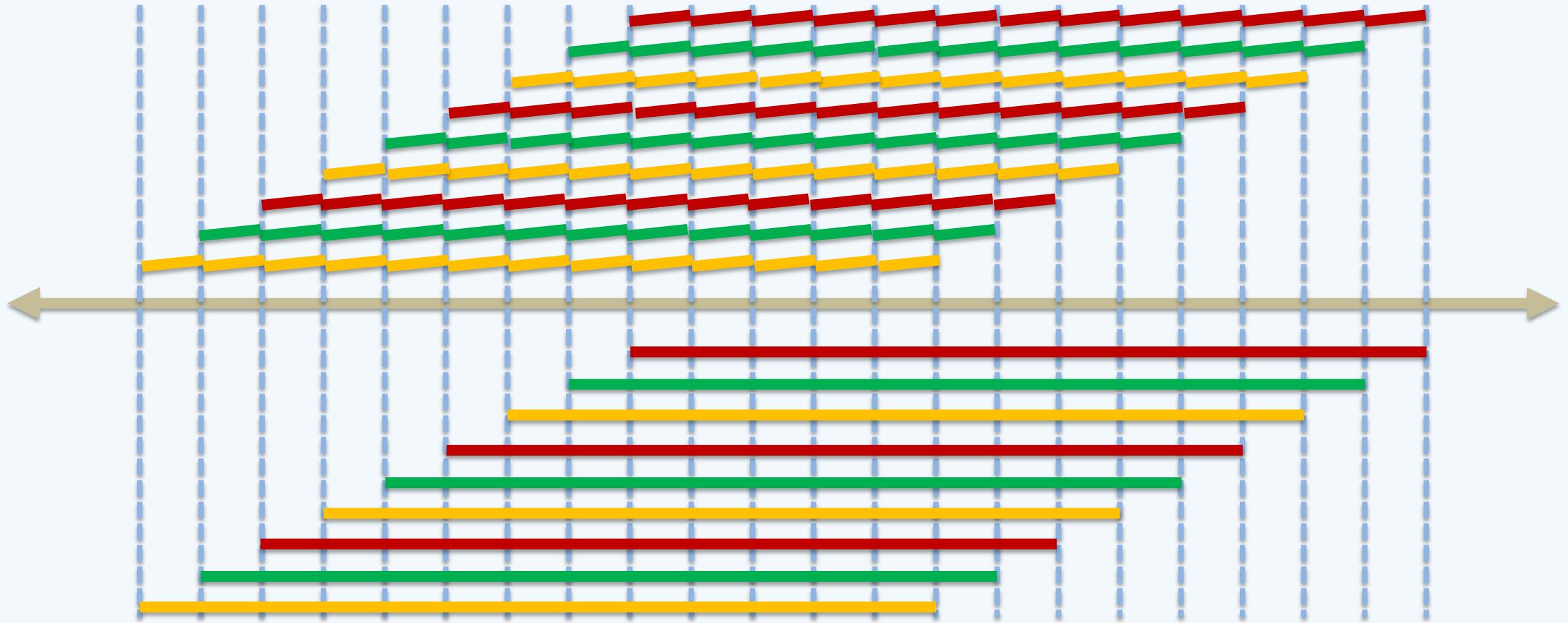


the output can then be returned directly

// $\mathcal{O}(r)$

Worst Case

- ❖ Every interval spans $\Omega(n)$ EI's and a total space of $\Omega(n^2)$ is required



Advanced Balanced Search Tree

Segment Tree

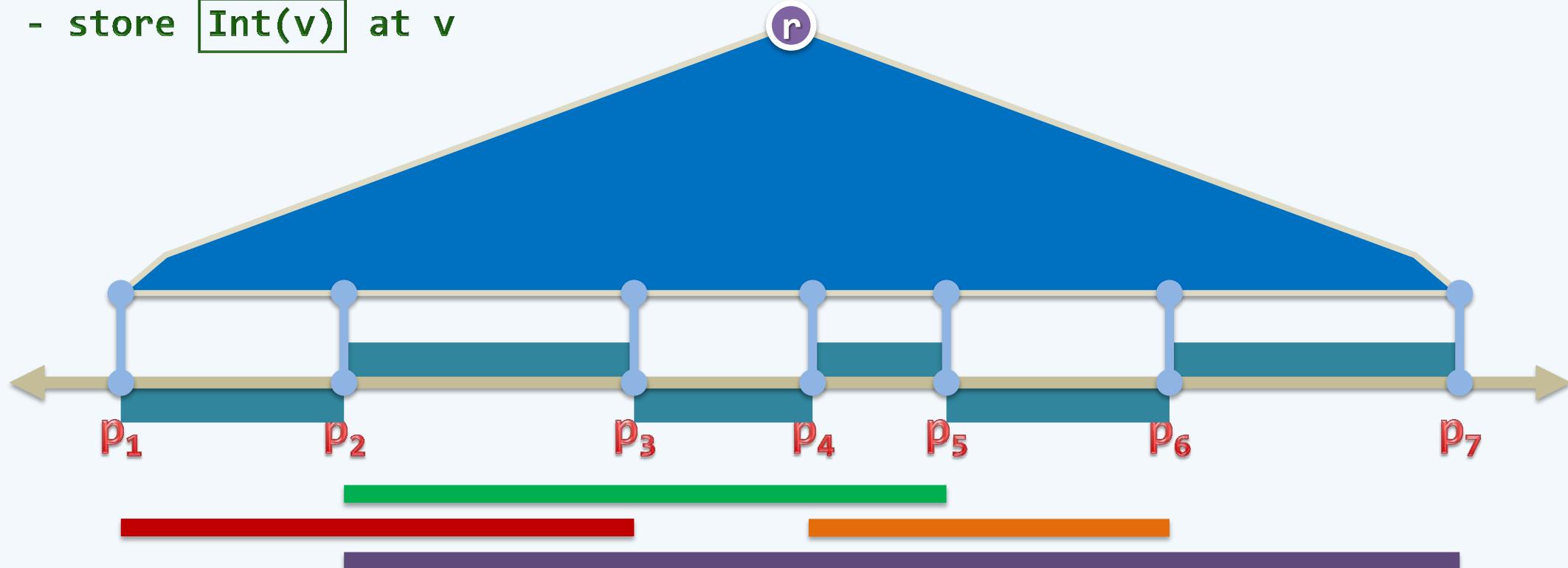
BBST

邓俊辉

deng@tsinghua.edu.cn

Replacing The Sorted Vector With A BBST

- ❖ For each leaf v ,
 - denote the corresponding elementary interval as $EI(v)$
 - denote the subset of intervals containing $EI(v)$ as $Int(v)$ and
 - store $Int(v)$ at v



1D Stabbing Query

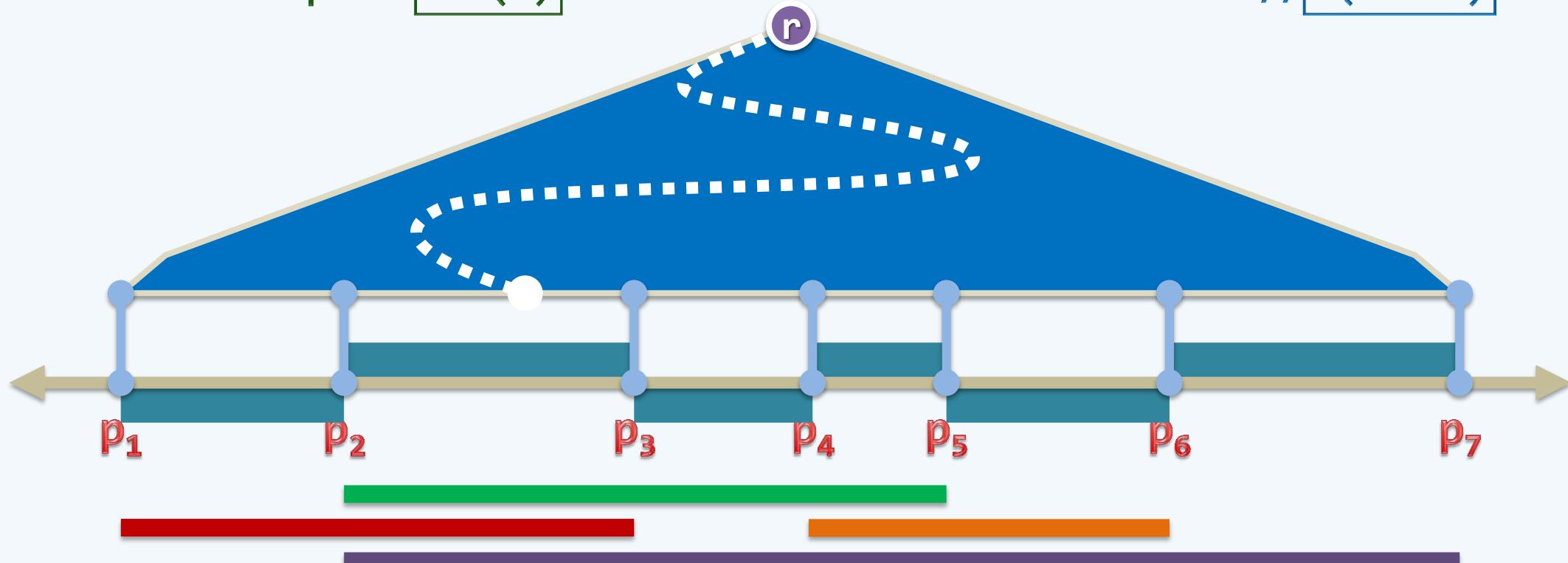
❖ To find all intervals containing q_x , we can

- find the $EI(v)$ containing q_x

// $\Theta(\log n)$ time for a BBST

- and then report $\text{Int}(v)$

// $\Theta(1 + k)$ time



Advanced Balanced Search Tree

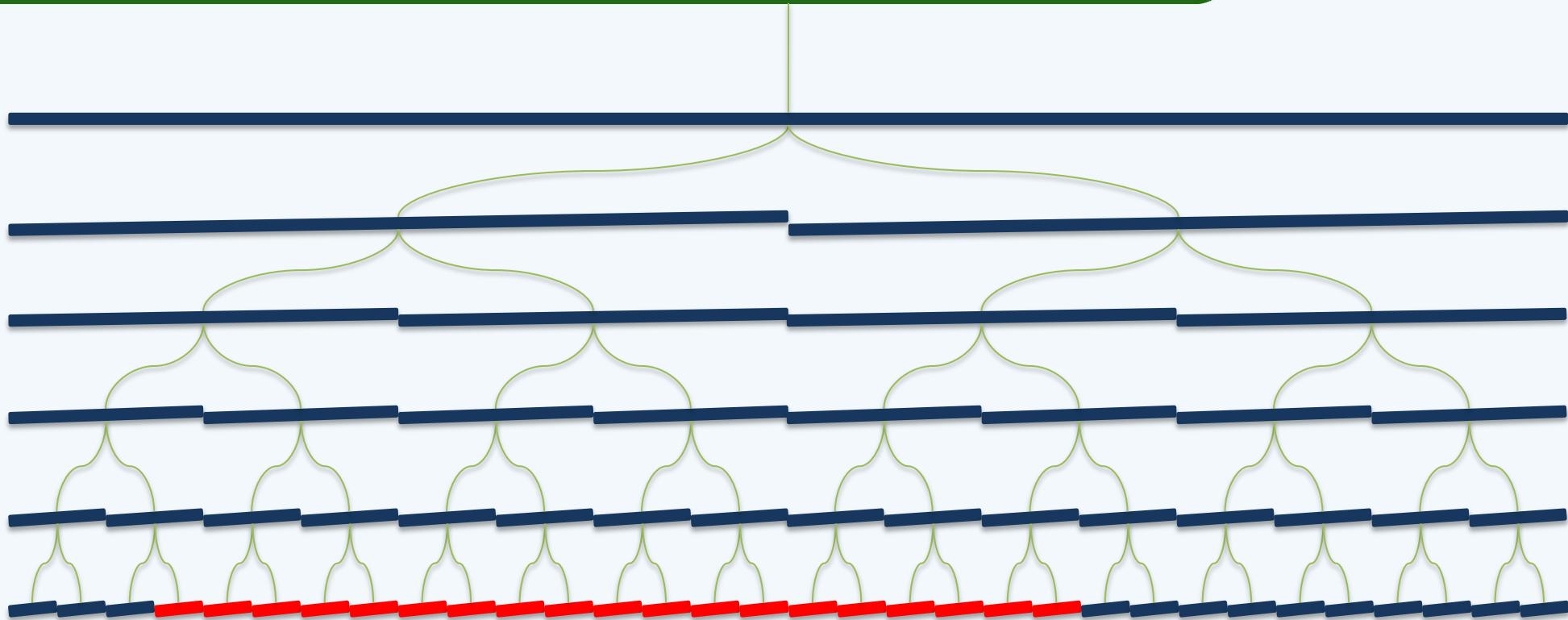
Segment Tree

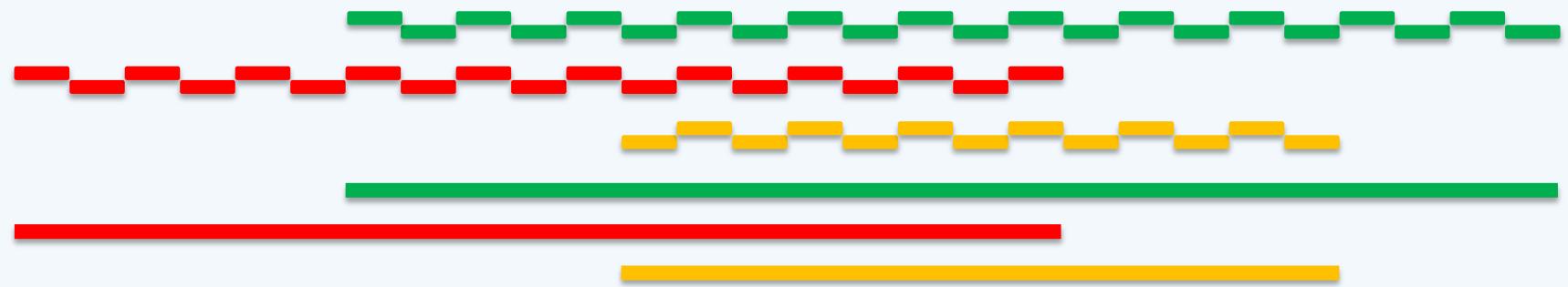
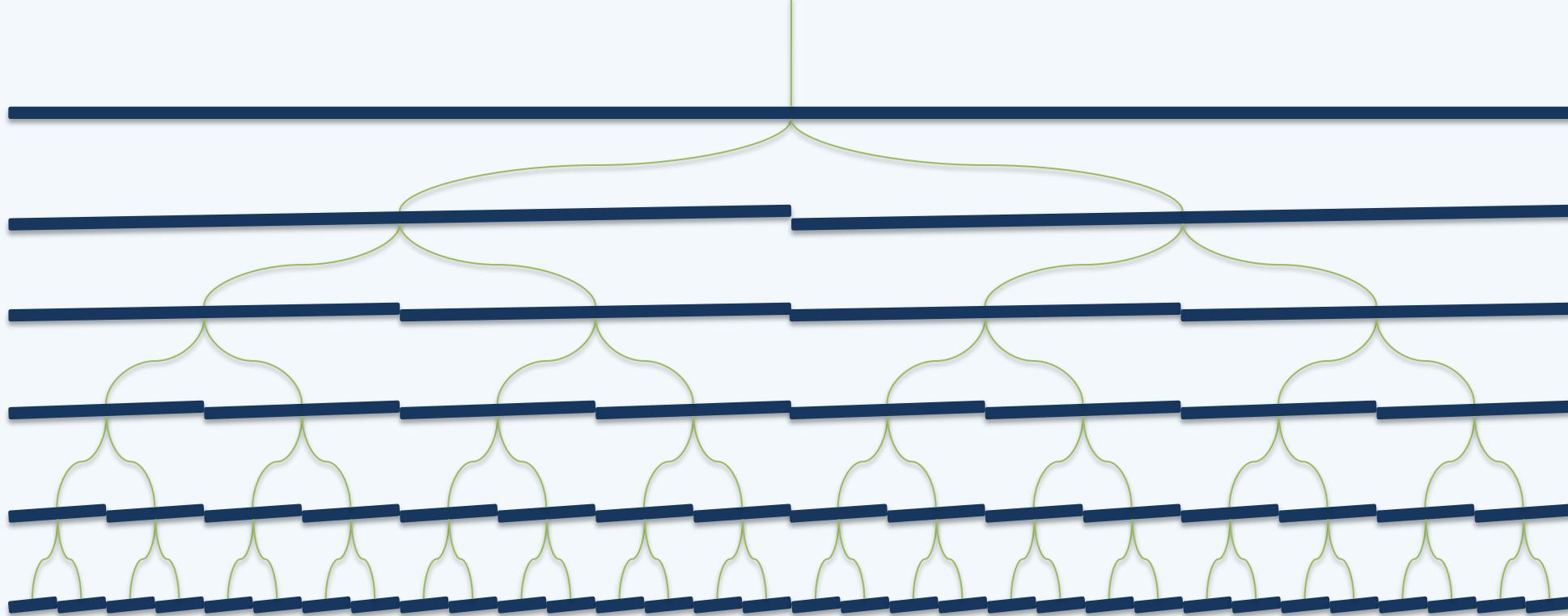
Worst Case

邓俊辉

deng@tsinghua.edu.cn

Still, Up To $\Omega(n)$ Space For Each Interval



$\Omega(n^2)$ Total Space In The Worst Cases

Advanced Balanced Search Tree

Segment Tree

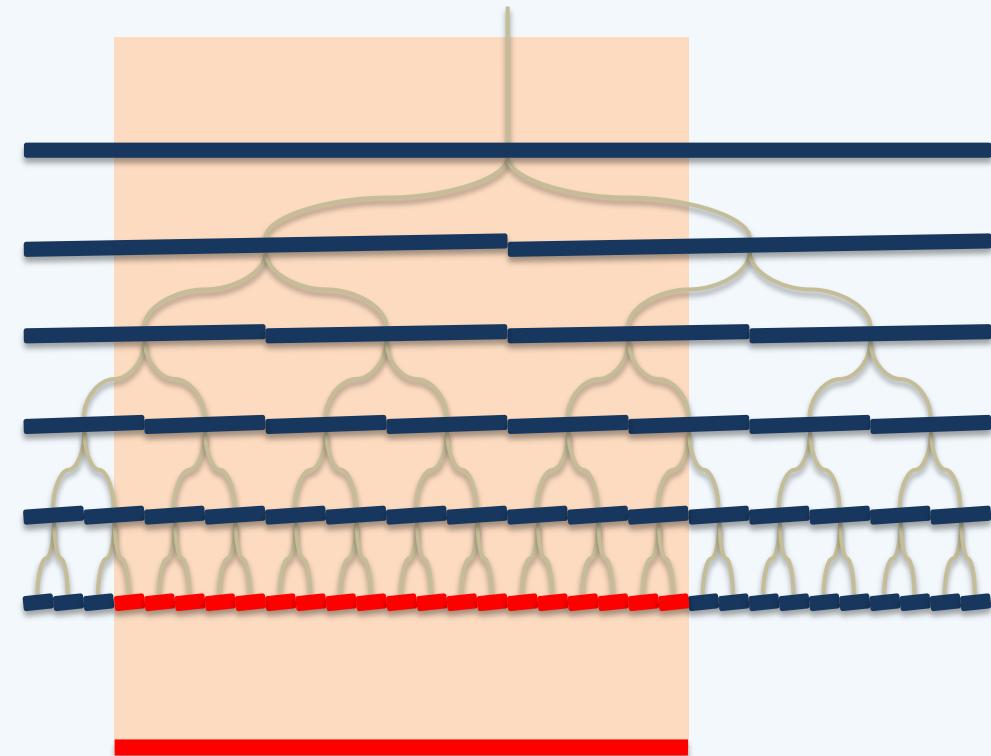
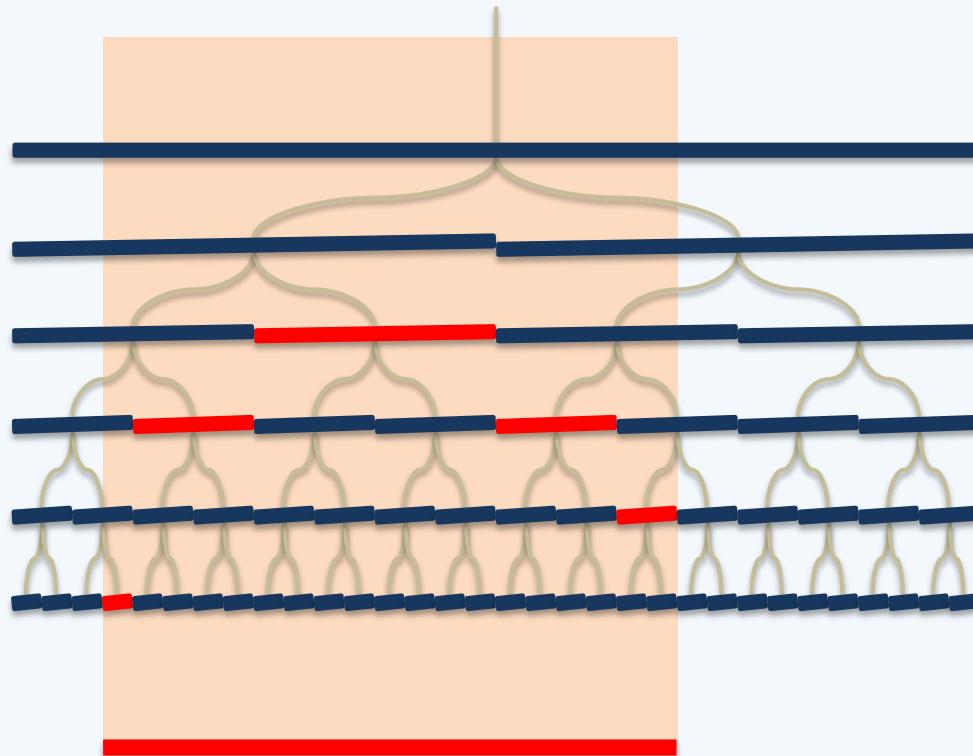
Common Ancestor

邓俊辉

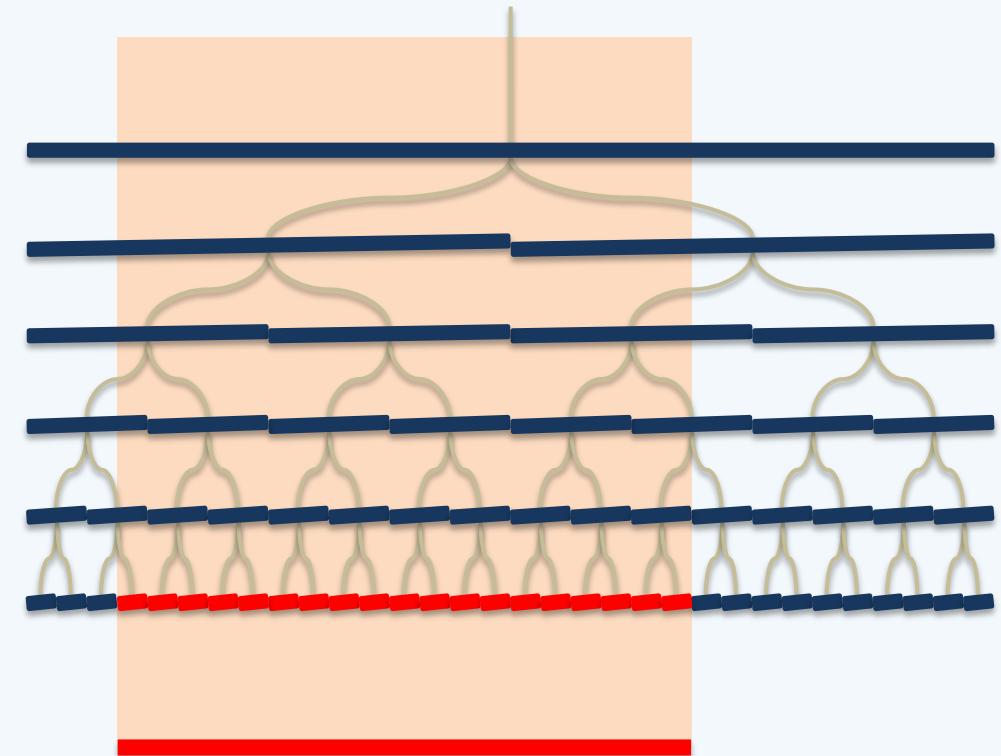
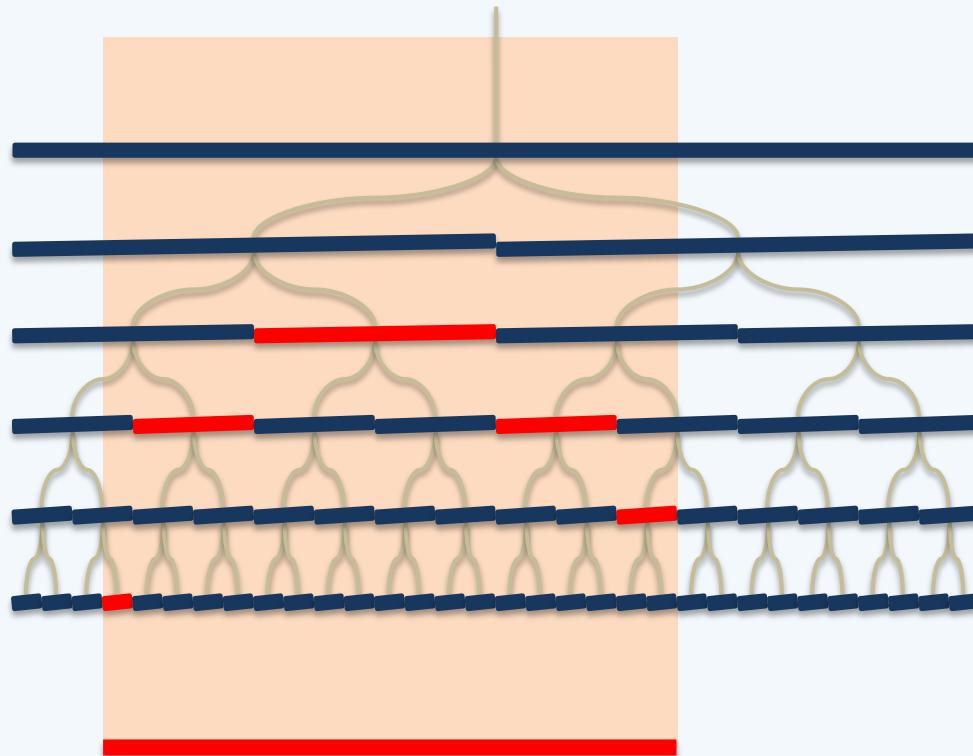
deng@tsinghua.edu.cn

⌚ If some of the leaves share a CA, then

a search path ends in any of them iff the CA is visited



∴ So, instead, why not just store a **single** copy at the **CA**?



Advanced Balanced Search Tree

Segment Tree

Canonical Subsets

邓俊辉

deng@tsinghua.edu.cn

❖ For each internal node u

- let $EI(u) = EI(u.lc) + EI(u.rc)$

- denote the CS for u as $\text{Int}(u)$...

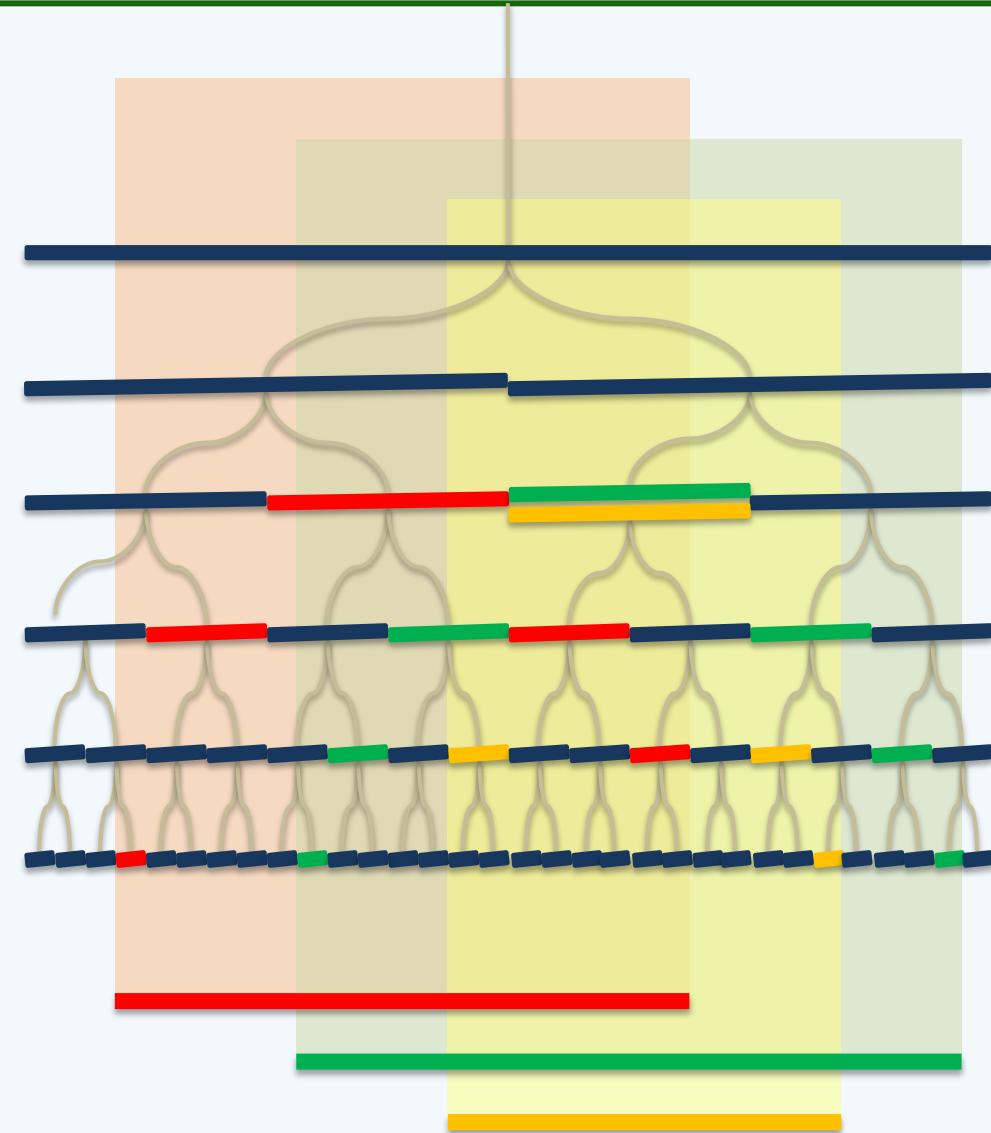
❖ An input interval X

belongs to $\text{Int}(u)$

iff

it covers $EI(u)$ but

not $EI(u.parent)$



Advanced Balanced Search Tree

Segment Tree

- $\mathcal{O}(n \log n)$ Space

邓俊辉

deng@tsinghua.edu.cn

❖ Consider any input interval X

- X is stored

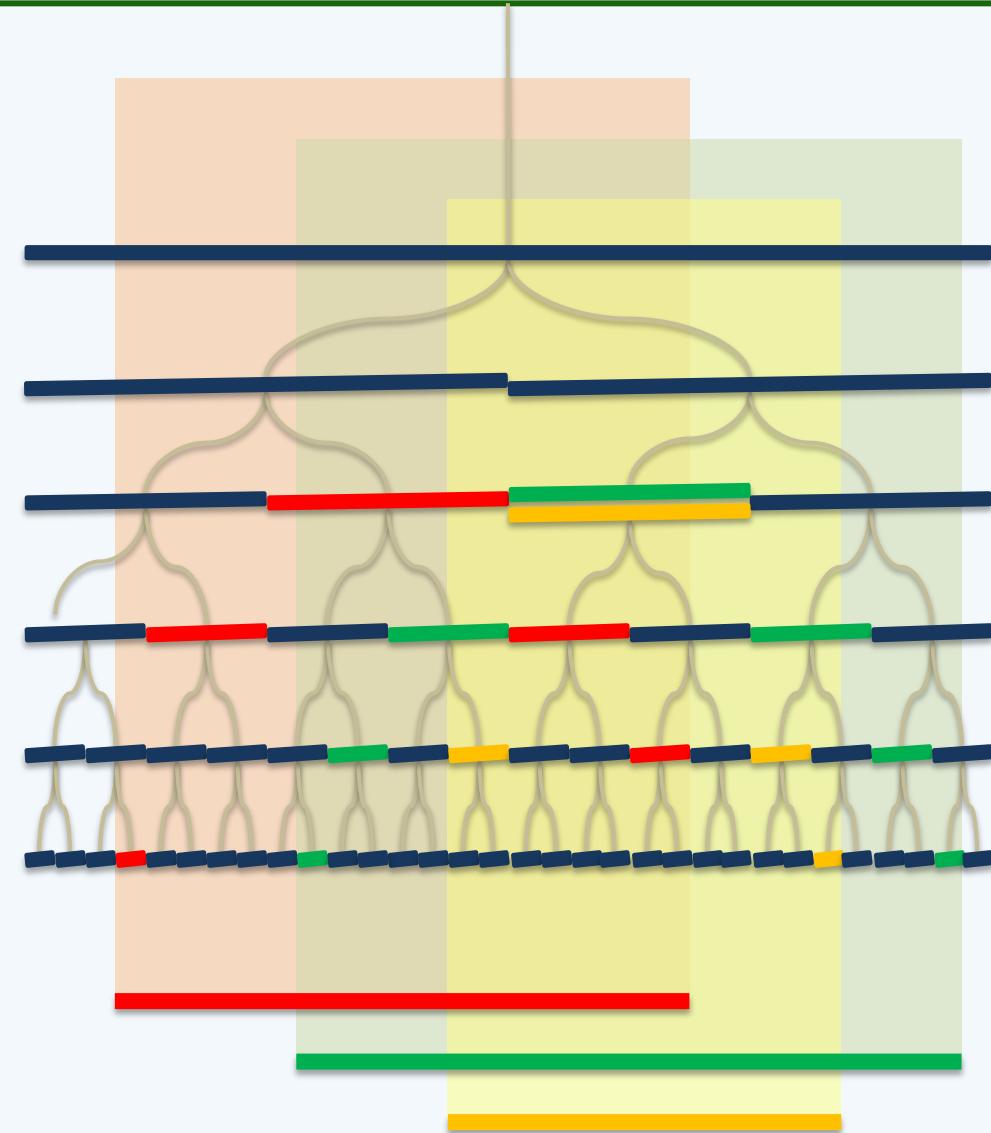
no more than **twice**

on each level

- X costs **$\Theta(\log n)$** space

❖ Such an optimized BBST

is called a **segment tree**



Advanced Balanced Search Tree

Segment Tree

Construction

邓俊辉

deng@tsinghua.edu.cn

BuildSegmentTree(I)

```

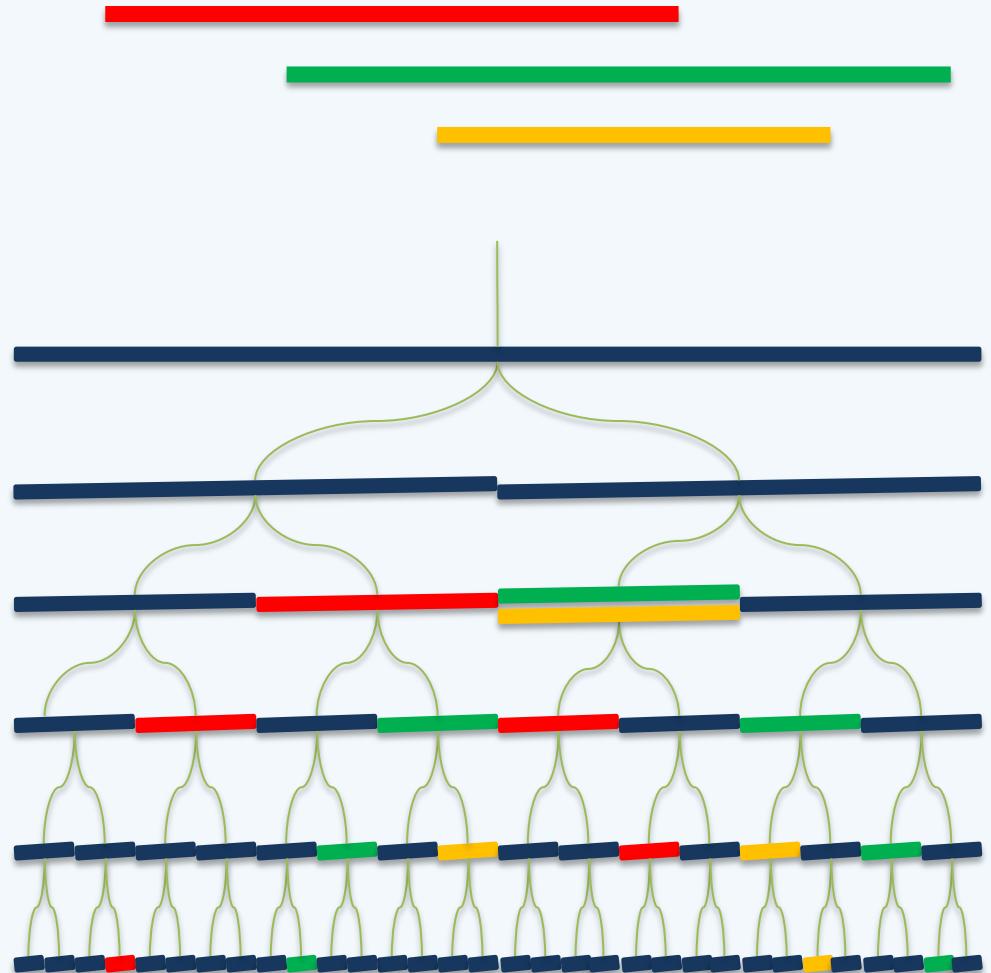
❖ // Construct a segment tree on
    // a set I of n intervals
Sort all endpoints in I before
    determining all EI's // $\Theta(n \log n)$ 

Create T a BBST on all the EI's // $\Theta(n)$ 

Determine Int(v) for each node v
    // $\Theta(n)$  if done in a bottom-up manner

For each s of I
    call InsertSegmentTree( T.root , s )

```



InsertSegmentTree(v , s)

❖ // Insert an interval s into a segment (sub)tree rooted at v

if ($\text{Int}(v) \subseteq s$) store s at v and return;

if ($\text{Int}(\boxed{\text{lc}(v)}) \cap s \neq \emptyset$) //recurse

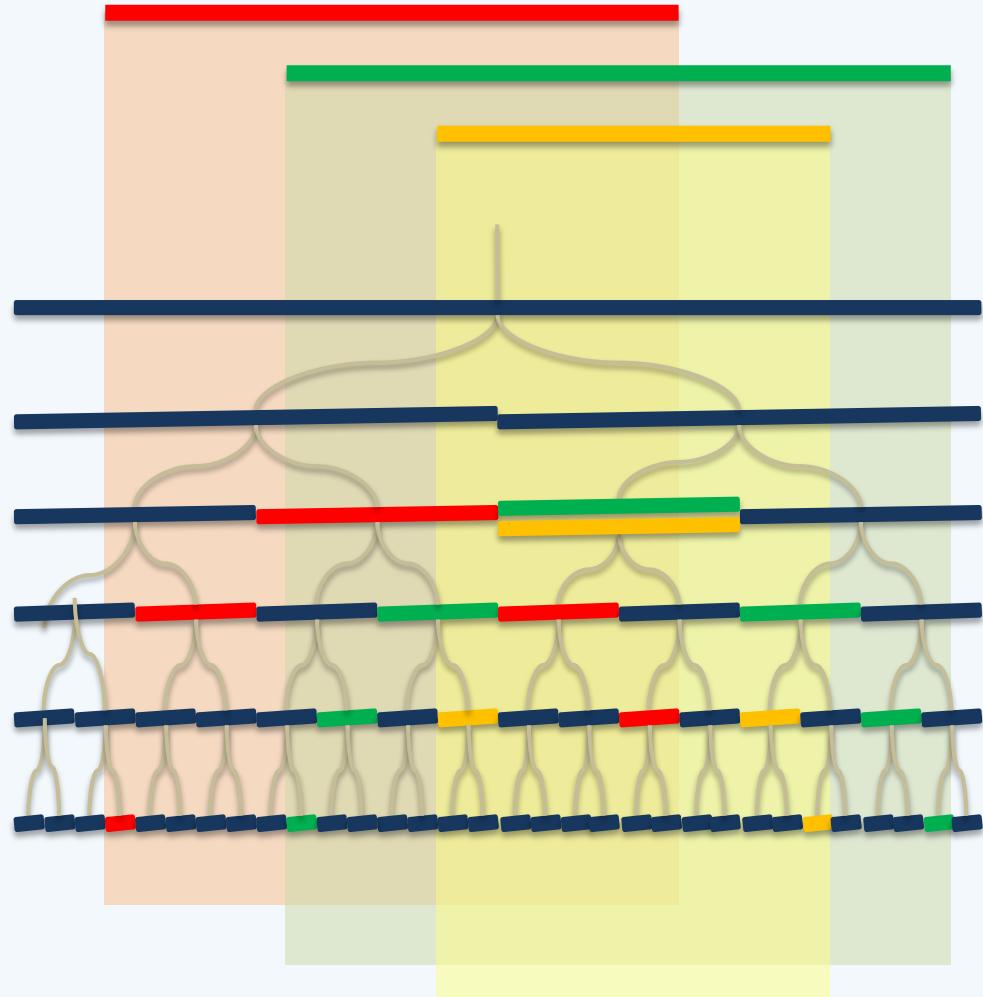
 InsertSegmentTree($\text{lc}(v)$, s);

if ($\text{Int}(\boxed{\text{rc}(v)}) \cap s \neq \emptyset$) //recurse

 InsertSegmentTree($\text{rc}(v)$, s);

⦿ At each level, ≤ 4 nodes are
visited (2 stores + 2 recursions)

$\therefore \Theta(\log n)$ time



Advanced Balanced Search Tree

Segment Tree

Query

邓俊辉

deng@tsinghua.edu.cn

QuerySegmentTree(v , q_x)

```

❖ // Find all intervals
    // in the (sub)tree rooted at  $v$ 
    // that contain  $q_x$ 

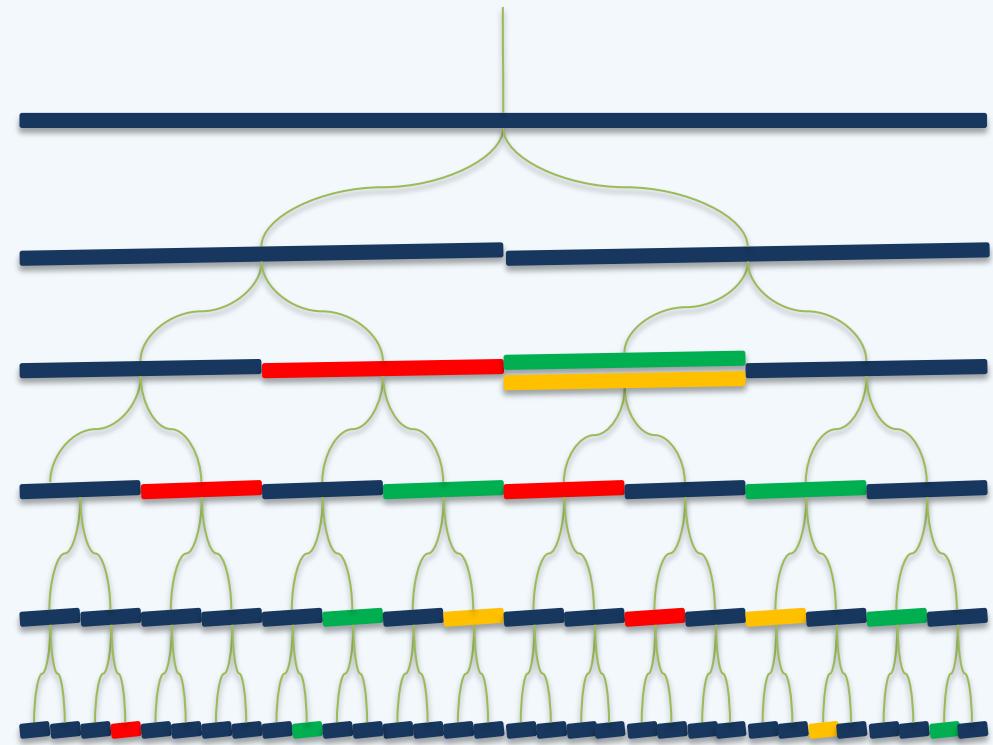
report all the intervals in  $\text{Int}(v)$ 

if (  $v$  is a leaf )
    return

if (  $q_x \in \text{Int}(\text{lc}(v))$  )
    QuerySegmentTree(  $\text{lc}(v)$ ,  $q_x$  )

else
    QuerySegmentTree(  $\text{rc}(v)$ ,  $q_x$  )

```



$\mathcal{O}(r + \log n)$

Only one node is visited per level,

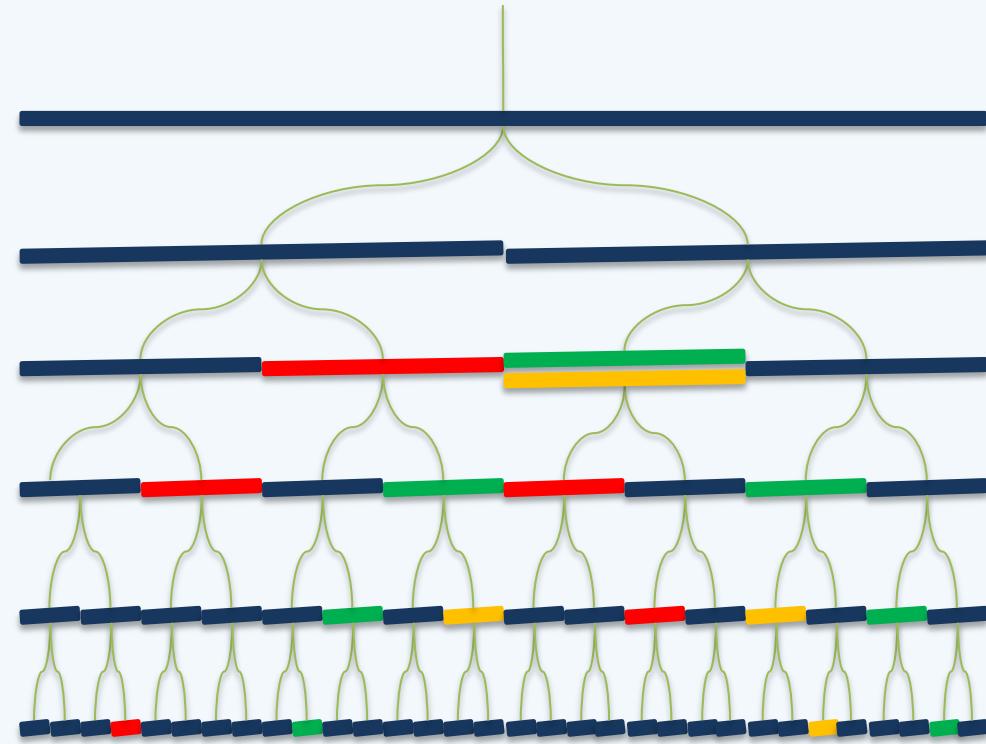
altogether $\mathcal{O}(\log n)$ nodes

At each node v

- the CS $\text{Int}(v)$ is reported
- in time

$$1 + |\text{Int}(v)| = \mathcal{O}(1 + r_v)$$

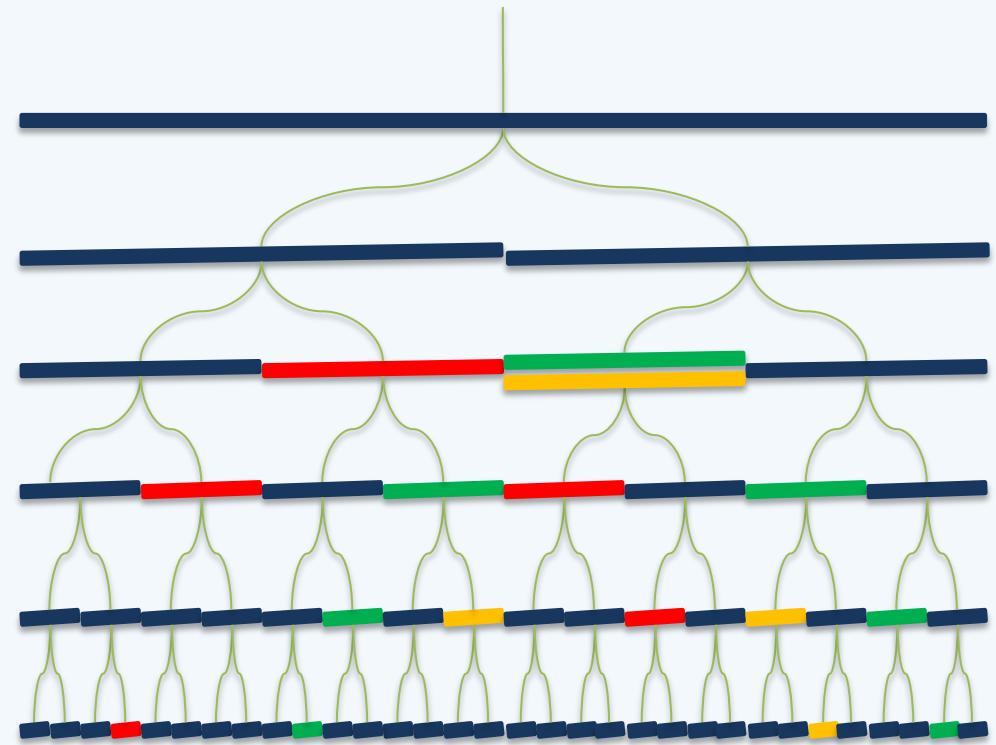
∴ Reporting all the intervals costs $\mathcal{O}(r)$ time



Conclusion

❖ For a set of n intervals,

- a segment tree of size $\Theta(n \log n)$
- can be built in $\Theta(n \log n)$ time
- which reports all intervals containing a query point in $\Theta(r + \log n)$ time



We are shaped by our thoughts;
we become what we think.

- Buddha

9. 词典

散列

循值访问

Man's thought is shaped by his tongue.

- Anonymous

The diversity of languages is not
a diversity of signs and sounds, but
a diversity of views of the world.

- Wilhelm von Humboldt, 1820

邓俊辉

deng@tsinghua.edu.cn

联合数组

- ❖ 数组？再常见不过，比如：`fib[0] = 0, fib[1] = 1, fib[2] = 1, fib[3] = 2, ...`
- ❖ **associative array**：与此前的数组有何区别？
- ❖ 根据数据元素的取值，直接访问！

```
style["关羽"] = "云长"
```

```
style["张飞"] = "翼德"
```

```
style["赵云"] = "子龙"
```

```
style["马超"] = "孟起"
```

下标不再是整数，甚至没有大小次序——更为直观、便捷

- ❖ 支持的语言：

`Snobol4、MUMPS、SETL、Rexx、AWK、Java、Python、Perl、Ruby、PHP、...`

映射 + 词典 = 符号表

❖ 词条 : `entry = (key, value)`

❖ 映射 : `Map` = 词条的集合 , 其中各词条 (的关键码) 互异

ADT : `get(key)` , `put(key, value)` , `remove(key)`

❖ 较之 BST , 关键码之间未必可比较 `call-by-value`

较之 PQ , 查找对象更广泛 , 不限于最大、最小词条

❖ 词典 : `Dictionary` : 与映射基本相同 , 但允许词条 (的关键码) 雷同

`Sorted dictionary` : 关键码之间可定义全序关系的词典

❖ 映射和词典都是动态的 , 统称符号表 `symbol table`

Dictionary

❖ template <typename K, typename V> //key、value

struct Dictionary { //Dictionary模板类

virtual int size() = 0; //查询当前的词条总数

virtual bool put(K) = 0; //插入词条(key, value)

virtual V* get(K) = 0; //查找以key为关键码的词条

virtual bool remove(K) = 0; //删除以key为关键码的词条

};

❖ 尽管诸如**Java:::TreeMap**等实现仍然要求支持**比较器**，但实际上

词典中的词条，只需支持**比对**判等操作，而不必支持大小**比较**

Dictionary

- ❖ 在这里，无论对外的访问方式，还是内部的存储方式都更直接地依据数据对象自身的取值
key与value地位等同，不必区分



- ❖ 循值访问 **call-by-value**：方式更为自然，适用范围也更广泛
- ❖ 回忆一下，**初次接触**程序设计时，你首先想到的应该就是这种方式

Java: HashMap + Hashtable

```
import java.util.*;  
  
public class Hash {  
  
    public static void main(String[] args) {  
  
        HashMap HM = new HashMap(); //Map  
  
        HM.put("东岳", "泰山"); HM.put("西岳", "华山"); HM.put("南岳", "衡山");  
        HM.put("北岳", "恒山"); HM.put("中岳", "嵩山"); System.out.println(HM);  
  
        Hashtable HT = new Hashtable(); //Dictionary  
  
        HT.put("东岳", "泰山"); HT.put("西岳", "华山"); HT.put("南岳", "衡山");  
        HT.put("北岳", "恒山"); HT.put("中岳", "嵩山"); System.out.println(HT);  
    }  
}
```

Perl: %Hash Type

❖由字符串 (string) 标识的一组无序标量 (scalar) //亦即MAP

❖`my %hero = ("云长"=>"关羽", "翼德"=>"张飞", "子龙"=>"赵云", "孟起"=>"马超") ;`

`foreach $style (keys %hero) # Hash类型的变量由%引导`

`{ print "$style => $hero{$style}\n"; }`

❖`$hero{"汉升"} = "黄忠";`

`foreach $style (keys %hero)`

`{ print "$style => $hero{$style}\n"; }`

`foreach $style (reverse sort keys %hero)`

`{ print "$style => $hero{$style}\n"; }`

Python: Dictionary Class

```
❖ beauty = dict # Python dictionary (hashtable)
    ( { "沉鱼": "西施", "落雁": "昭君", "闭月": "貂蝉", "羞花": "玉环" } )
print beauty

❖ beauty["红颜"] = "圆圆"
print beauty

❖ for alias, name in beauty.items():
    print alias, ":", name

❖ for alias, name in sorted(beauty.items()):
    print alias, ":", name

❖ for alias in sorted(beauty.keys(), reverse = True):
    print alias, ":", beauty[alias]
```

Ruby: Hash Table

```
scarborough = { # declare and initialize a hash table  
    "P"=>"parsley", "S"=>"sage",  
    "R"=>"rosemary", "T"=>"thyme"  
}  
  
puts scarborough # output the hash table  
  
for k in scarborough.keys # output hash table items  
    puts k + "=>" + scarborough[k] # 1-by-1  
end  
  
for k in scarborough.keys.sort # output hash table items  
    puts k + "=>" + scarborough[k] # 1-by-1 in order  
end
```

课后

❖ 了解Java中HashMap与Hashtable的异同

❖ 安装JDK (<http://www.java.com>)

尝试HashMap和Hashtable类

❖ 安装Perl (<http://www.perl.org>)

尝试%Hash类型

❖ 安装Python (<http://www.python.org>)

尝试Dictionary类

❖ 安装Ruby (<http://www.ruby-lang.org>)

尝试Hash Table

9. 词典

散列

原理

书者，散也。

邓俊辉

欲书先散怀抱，任情恣性，然后书之。

deng@tsinghua.edu.cn

服务 ~ 电话**General inquiries**

Tel: Toll Free: 1-800-IBM-4YOU
E-mail: askibm@vnet.ibm.com
www.ibm.com/us/en/

Shopping

Tel: Toll Free: 1-888-SHOP-IBM

Sales Center

1-855-2-LENOVO (1-855-253-6686)
Mon - Fri: 9am-9pm (EST)
Sat - Sun: 9am-6pm (EST)

Customer Service

1-855-2-LENOVO (1-855-253-6686)
Mon - Fri: 9am-9pm (EST)
Sat - Sun: 9am-6pm (EST)



85001

❖ TV channel → s/n

CCTV-1 [1]	BJTV-1 [21]	CETV-1 [31]
CCTV-2 [2]	BJTV-2 [22]	CETV-2 [32]
CCTV-3 [3]	BJTV-3 [23]	CETV-2 [33]

...



❖ <http://85001.tsinghua.edu.cn/>

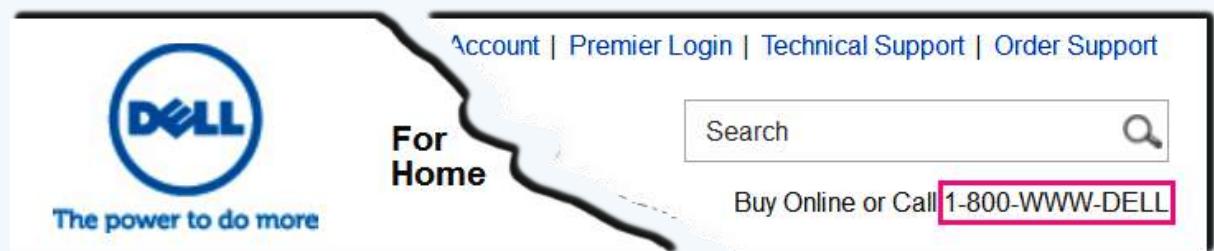
```
owner = phonebook.get(number)
```



❖ 动态性：每天的电话簿都是不同的

```
phonebook.put(number, owner)
```

```
phonebook.remove(number)
```



电话簿

❖ 需求：为一所学校制作电话簿

号码 ~ 个人（教员、学生、员工）或办公室

▶ ❖ 蛮力：使用数组，按电话号码索引

时间 = $O(1)$

❖ 以清华为例（2003）

$$\# \text{可能的电话} = R = 10^8 = 100M$$

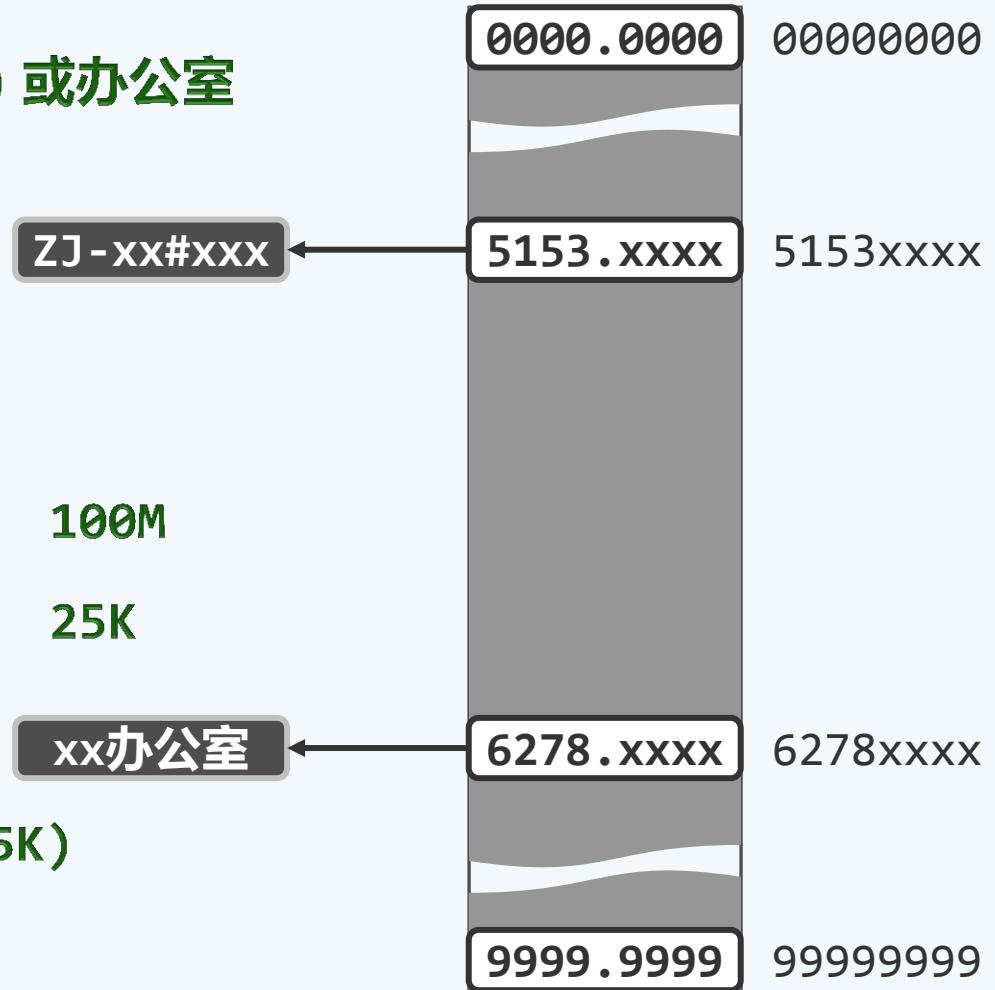
$$\# \text{实有的电话} = N = 25,000 = 25K$$

❖ 问题

$$\text{空间} = O(R + N) = O(100M + 25K)$$

$$\text{效率} = 25K / 100M = 0.025\%$$

其它方面...



IP Dictionary

❖ IP → Domain name + Host info

C-Term WRY.dat

<http://www.ip138.com>

❖ IPv4 (32bit) → IPv6 (128bit)

128-bit

$$R = \# \text{可能的IP} = 2^{128} = 256 \times 10^{36}$$

32-bit

$$N = \# \text{实际的IP} = 2^{32} = 4 \times 10^9$$

$$N / R = 0.\boxed{000} \boxed{000} \boxed{000} \boxed{000} \boxed{000} \boxed{000} \boxed{000} \boxed{000} \boxed{000} \boxed{015} \boxed{625}$$

❖ 事实：实际的词条数 N << 可能的词条数 R

//相对于THU电话簿，此时的N与R相差更为悬殊

❖ 如何在保持查找速度的同时，降低存储消耗？

散列表

❖ 桶 bucket : 直接存放或间接指向一个词条

❖ 桶数组 bucket array / 散列表 hash table , 容量为 M



$$\text{空间} = O(N + M) = O(N)$$

❖ 定址/杂凑/散列 :

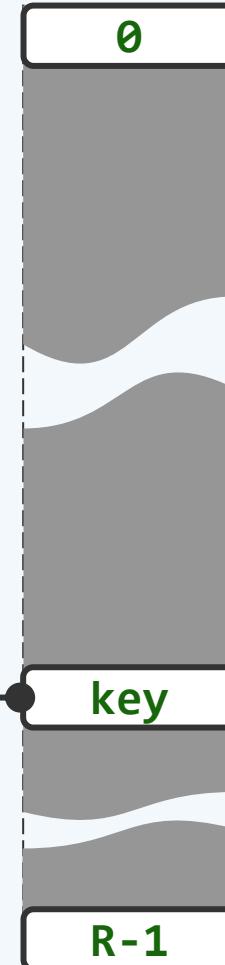
根据词条的key (未必可比较)

直接确定散列表入口

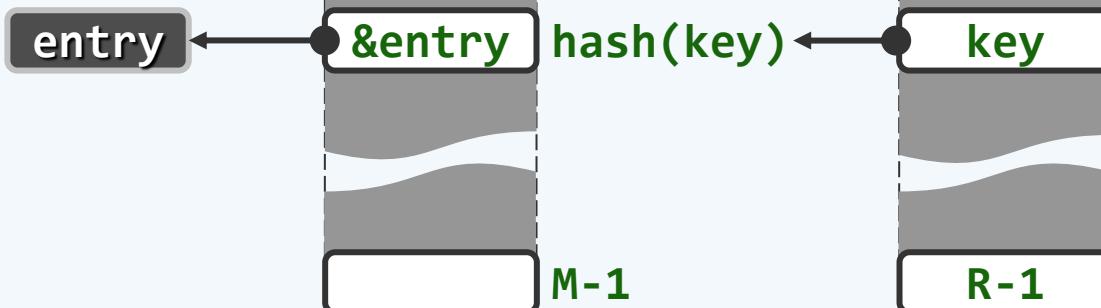
❖ 散列函数 : $\text{hash}() : \text{key} \mapsto \&\text{entry}$

❖ 直接 = $\text{expected} - O(1) \neq O(1)$

关键码空间



散列地址空间



实例

散列函数 : $\text{hash}(\text{key}) = \text{key \% M}$

6278.5001

5153.1876

6277.0211

% 90001

散列表长 : $M = 90001$

0

39,514

51,304

54,304

90,000

校长办公室

ZJ-09#403

查号台

时空效率

❖ 定址：通过散列函数，将任一key映射至 $[0, M)$

除余法： $\text{hash}(\text{key}) = \text{key} \% M$

//例： $M = 90001$

定址效率：基于取模运算——常数时间！

//机器字长无限制？

❖ 无论散列表多大，定址、查询、插入和删除均只需expected- $O(1)$ 时间

❖ 装填因子load factor

$\lambda = N / M = \# \text{存放的词条} / |\text{桶数组}|$

// λ ，选多大才合适？

❖ λ 越大，空间利用率越高

当然， λ 不可能超过100%

//否则，根据鸽巢原理...

反之，是否只要 $\lambda \leq 1$ 就行了？

//比如，就取 $\lambda = 1$

❖ 实际上，即便 $\lambda \ll 1$ ，依然会有问题...

9. 词典

散列

冲突

宝玉道：“已经完了，怎么又作揖？”袭人笑道：
“这是他来给你拜寿。今儿也是他的生日，你也
该给他拜寿。”宝玉听了，喜的忙作下揖去，说：
“原来今儿也是姐姐的芳诞。”

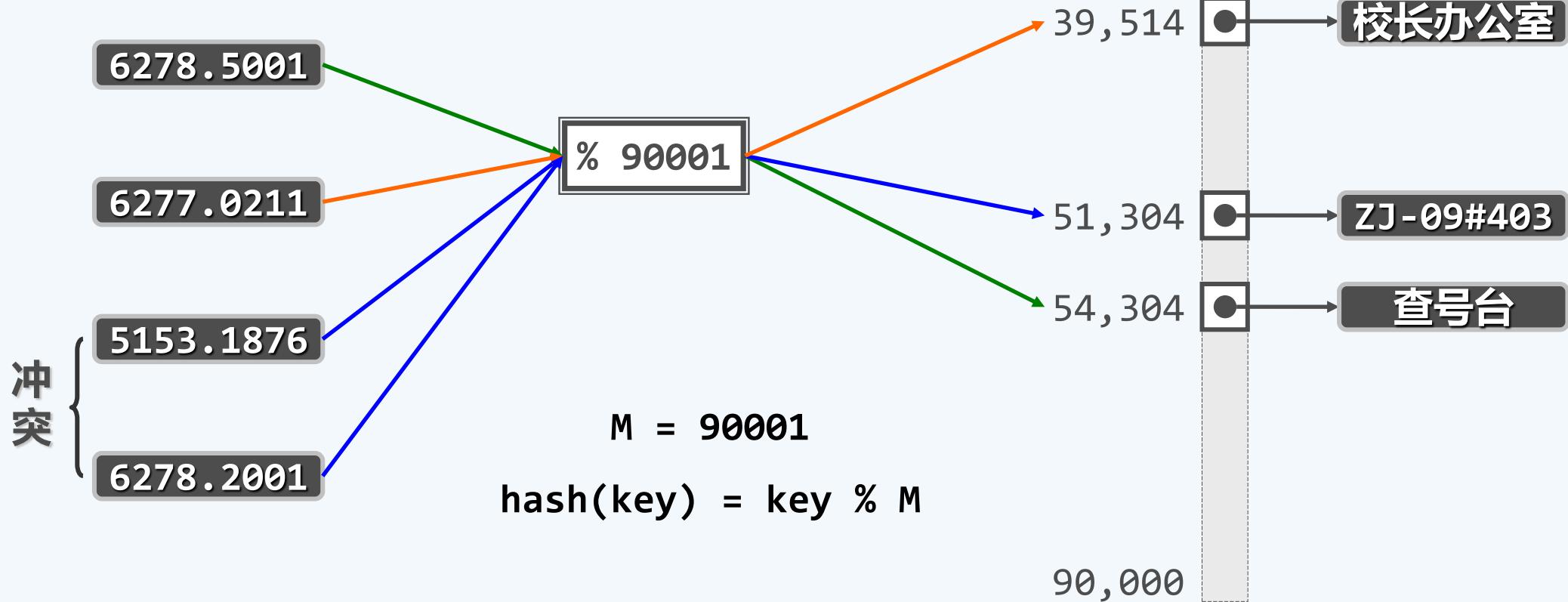
邓俊辉

deng@tsinghua.edu.cn

实例

❖ $\text{key1} \neq \text{key2}$, 但

$\text{hash}(\text{key1}) = \text{hash}(\text{key2})$



完美散列

- ❖ 是否存在某种定址方法，能保证不出现冲突？
亦即，散列函数等效于一个单射 injection？
- ❖ 在关键码满足某些条件时，的确可以实现单射式散列，比如...
- ❖ 对已知且固定的关键码集（比如CD）
可实现完美散列 perfect hashing
 - 采用两级散列模式
 - 仅需 $\Theta(n)$ 空间
 - 关键码之间互不冲突
 - 即便在最坏情况下，查找时间也不过 $\Theta(1)$ 时间
- ❖ 不过，在一般情况下，完美散列无法保证存在...

生日悖论

❖ 将在座同学（对应的词条）按生日（月/日）做散列存储

散列表长固定为 $M = 365$ ，装填因子 = 在场人数 $N / 365$

❖ 冲突（至少有两位同学生日相同）的可能性 $P_{365}(n) = ?$

// 概率论与数理统计讲义第一章，清华大学数学系王晓峰

$P_{365}(21) = 44.4\%$, $P_{365}(22) = 47.6\%$, ..., $P_{365}(23) = 50.7\%$ // $23/365 = 6.3\%$

❖ 100人的集会： $1 - p_{365}(100) = 0.000,031\%$

自7岁起，不吃不喝、无休无息，每小时参加四次

到100岁，才有可能遇到一次没有冲突的集会

❖ 因此，在装填因子确定之后，散列策略的选取将至关重要，散列函数的设计也很有讲究...

9. 词典

散列函数

基本

此刻他就在占卜，方法是要从办公室到法庭扶手椅座位的步数可以被三除尽，那么新的疗法肯定能治好他的胃炎；要是除不尽，那就治不好。走下来是二十六步，但他把最后一步缩小，这样就正好走了二十七步。

邓俊辉

deng@tsinghua.edu.cn

无法杜绝的冲突

◆ 散列函数 $\text{hash}(): S \rightarrow A$, 不可能是单射

//词条空间 S ~ 可能的词条

$$|S| = R \gg M = |A|$$

//地址空间 A ~ 散列表



两项基本任务

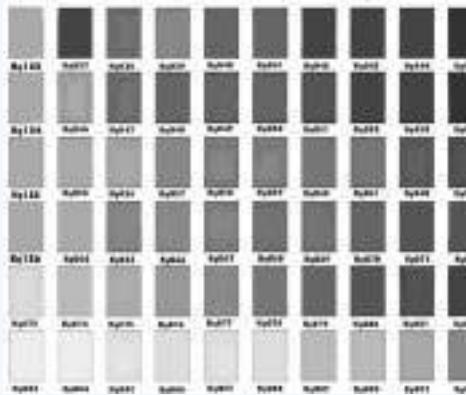
- ❖ 近似的单射，往往可行...
- ❖ 因此，我们需要...

1) **本节**：精心设计**散列表**及**散列函数**

以**尽可能**降低冲突的概率；更需要...

2) **下节**：制定可行的**预案**

以便在发生冲突时，能够尽快予以**排解**



评价标准与设计原则

❖ 什么样的散列函数hash()更好？

- 1) 确定determinism : 同一关键码总是被映射至同一地址
- 2) 快速efficiency : expected- $\mathcal{O}(1)$
- 3) 满射surjection : 尽可能充分地覆盖整个散列空间
- 4) 均匀uniformity : 关键码映射到散列表各位置的概率尽量接近

可有效避免聚集clustering现象

除余法

❖ $\text{hash}(\text{key}) = \text{key \% } M$

前例中，为何选 $M = 90001$?

❖ 若取 $M = 2^k$ ，其效果相当于

截取 key 的最后 k 位 (bit)，前面的 $n - k$ 位对地址没有影响

$$M - 1 = \boxed{0 \ 0 \ 0 \ \dots \ 0} \mid \boxed{1 \ 1 \ 1 \ \dots \ 1}$$

$$\text{key \% } M = \text{key \& } (M - 1)$$

推论：发生冲突 iff 最后 k 位相同 //发生冲突的概率大

❖ 若取 $M \neq 2^k$ ，缺陷有所改善

❖ M 为 素数 时，数据对散列表的覆盖最 充分，分布最 均匀 // 蝉的哲学

MAD法

❖ 除余法的缺陷...

1) 不动点 : 无论表长 M 取值如何 , 总有 $\text{hash}(0) \equiv 0$

2) 零阶均匀 : $[0, R)$ 的关键码 , 平均分配至 M 个桶 ; 但相邻关键码的散列地址也必相邻

❖ 一阶均匀 : 邻近的关键码 , 散列地址不再邻近

// 更高阶的均匀性呢 ?

❖ $\text{MAD} = \text{multiply-add-divide}$

取 M 为素数 , $a > 0$, $b > 0$, $a \% M \neq 0$

$$\text{hash(key)} = ((a \times \text{key} + b) \% M)$$

❖ 当然 , 某些特定场合下 , 未必需要高阶的均匀性

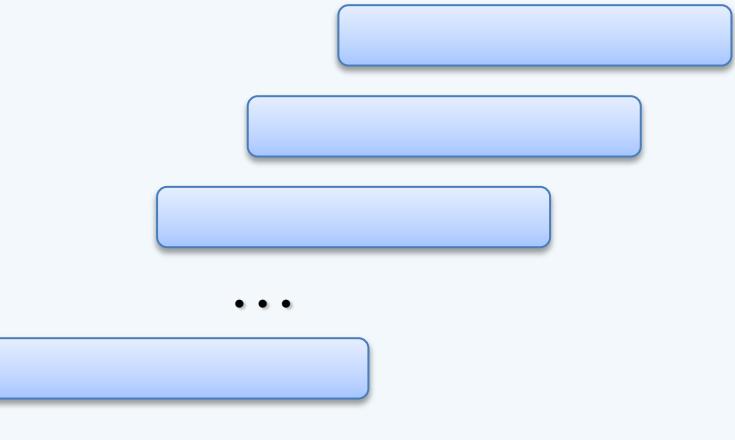
更多散列函数

❖ 数字分析 selecting digits

抽取key中的某几位，构成地址

比如，取十进制表示的奇数位

$$\text{hash}(\begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array}) = \begin{array}{ccccccc} 1 & 3 & 5 & 7 & 9 \end{array}$$



❖ 平方取中 mid-square

取key²的中间若干位，构成地址

$$\text{hash}(123) = 512 // \text{保留key}^2 = 123^2 = 1\boxed{512}9 \text{的中间3位}$$

$$\text{hash}(1234567) = 556 // 1234567^2 = 15241\boxed{556}77489$$

更多散列函数

❖ 折叠法 folding : 将key分割成等宽的若干段，取其总和作为地址

$\text{hash}(123456789) = 1368 // 123 + 456 + 789$, 自左向右

$\text{hash}(123456789) = 1566 // 123 + 654 + 789$, 往复折返

❖ 位异或法 XOR : 将key分割成等宽的二进制段，经异或运算得到地址

$\text{hash}(110011011_b) = 110_b // 110 \wedge 011 \wedge 011$, 自左向右

$\text{hash}(110011011_b) = 011_b // 110 \wedge 110 \wedge 011$, 往复折返

❖ ...

❖ 总之，越是随机，越是沒有规律，越好

9.词典

散列函数

更多

邓俊辉

有意整齐与有意变化，皆是一方死法。

deng@tsinghua.edu.cn

(伪)随机数法

❖ (伪)随机数发生器

循环： $\text{rand}(x + 1) = [a \times \text{rand}(x)] \% M$ //M素数， $a \% M \neq 0$

$$a = 7^5 = 16,807 = \boxed{100000110100111}_b$$

$$M = 2^{31} - 1 = 2,147,483,647 = 01111111 \boxed{11111111} 11111111 \boxed{11111111}_b$$

❖ (伪)随机数法

选取： $\text{hash(key)} = \text{rand(key)} = [\text{rand}(0) \times a^{\text{key}}] \% M$

种子： $\text{rand}(0) = ?$

❖ 把难题推给伪随机数发生器，但是...

❖ (伪)随机数发生器的实现，因具体平台、不同历史版本而异

创建的散列表 **移植性差**——故需慎用此法！

(伪)随机数法

❖ `unsigned long int next = 1; //The C Programming Language (2nd edn), p46`

```
void srand(unsigned int seed) { next = seed; }
```

```
int rand(void) { //1103515245 = 3^5 * 5 * 7 * 129749
```

```
next = next * 1103515245 + 12345;
```

```
return (unsigned int)(next/65536) % 32768;
```

```
}
```

rand  2^{15}

next  2^{15} 2^{32}

❖ 另类随机数(串)算法

```
int rand() { int uninitialized; return uninitialized; }
```

```
char* rand( t_size n ) { return ( char* ) malloc( n ); }
```

❖ 以上方法，可否用于散列？

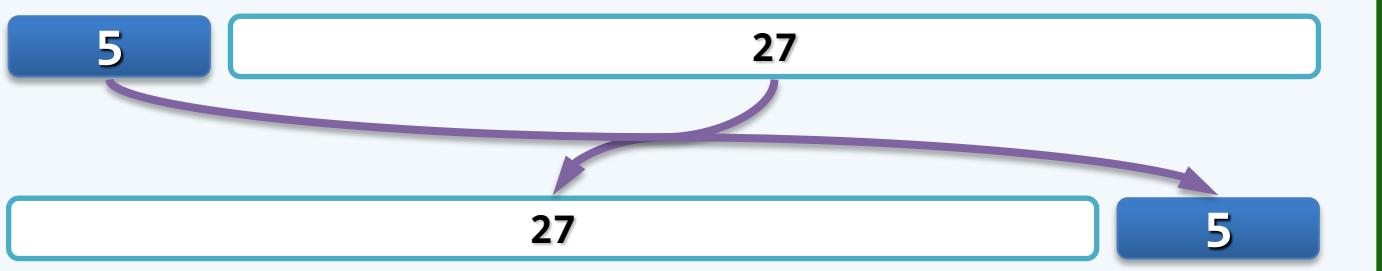
多项式法

❖ hash($s = [x_0 \ x_1 \ \dots \ x_{n-1}]$) = $x_0 \cdot a^{n-1} + x_1 \cdot a^{n-2} + \dots + x_{n-2} \cdot a^1 + x_{n-1}$

$$= (\dots ((x_0 * a + x_1) * a + x_2) * a + \dots x_{n-2}) * a + x_{n-1}$$

❖ static size_t hashCode(char s[]) { //近似多项式，但更快捷
 unsigned int h = 0;
 for (size_t n = strlen(s), i = 0; i < n; i++)
 { h = (h << 5) | (h >> 27); h += (int)s[i]; }
 return (size_t)h;
} //有必要如此复杂吗？

❖ 能否使用更简单的散列，比如...



多项式法

❖ 字符分别映射为数字 : $f(c) = \text{CODE}(\text{UPPER}(c)) - 64$

$//['A', 'Z'] \sim [1, 26]$

再简单相加 : $\text{hash}(s) = \sum_{c \in s} f(c)$

hash $\sim 8 + 1 + 19 + 8 = 36$

❖ 字符相对次序信息丢失, 将引发大量冲突

I am Lord Voldemort

Tom Marvolo Riddle

❖ 即便字符不同、数目不等...

He's Harry Potter

❖ Key to improving your programming skills

Learning Tsinghua Data Structures & Algorithms



Java: hashCode()

❖ hashCode()方法

适用于Java中的所有对象

将任意类的对象转换为（32位int型）整数

对于非整型的key，先转换为整数（散列码），然后再做散列

❖ hashCode()：效果如何？效率如何？是如何实现的？

❖ object.hashCode() = object在内存中的地址

❖ 问题：相关的对象地址也相近，冲突概率高 //更糟糕的是...

❖ 散列码与对象的内容无关

比如，完全相等的两个字符串对象，散列码居然不同

❖ 有何替代方案？

9.词典

排解冲突

开放散列

Every mistake I've ever made
Has been rehashed and then replayed
As I got lost along the way.

邓俊辉

deng@tsinghua.edu.cn

多槽位

❖ multiple slots

桶单元细分成若干槽位 slot

存放（与同一单元）冲突的词条

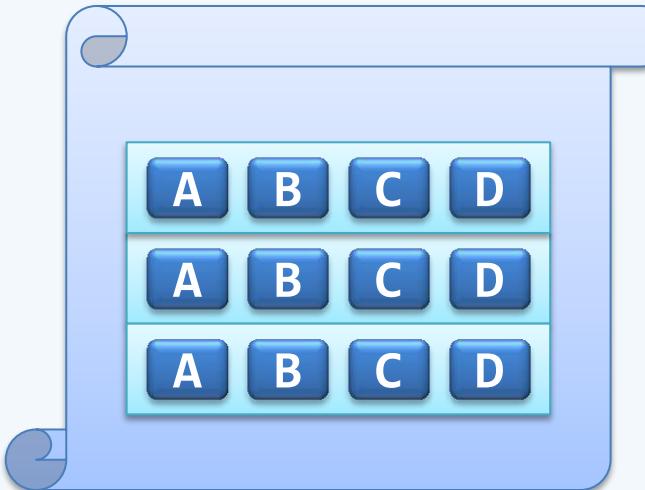
❖ 只要槽位数目不多

依然可以保证 $\mathcal{O}(1)$ 的时间效率

❖ 但是，需要为每个桶配备多少个槽，方能保证 $\mathcal{O}(1)$ ？ //难以预测

预留过多，空间浪费

无论预留多少，极端情况下仍有可能不够



bucket[n + 1]

bucket[n]

bucket[n - 1]

独立链

❖ linked-list chaining / separate chaining

每个桶存放一个指针

冲突的词条，组织成列表

❖ 优点 无需为每个桶预备多个槽位

任意多次的冲突都可解决

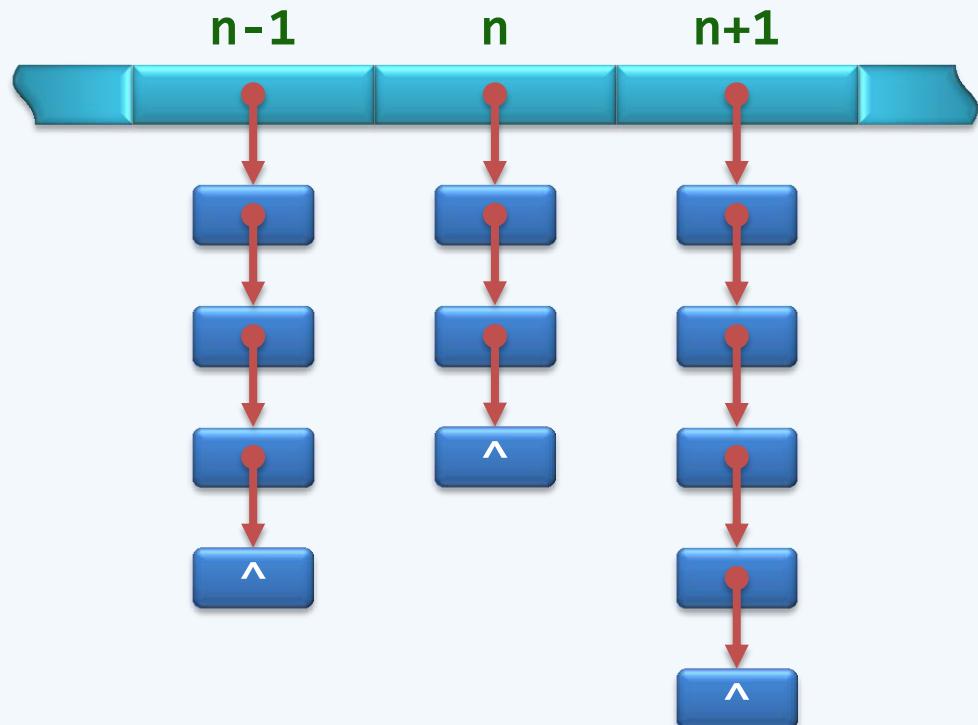
删除操作实现简单、统一

❖ 但是 指针需要额外空间

节点需要动态申请

更重要的是...

❖ 空间未必连续分布，系统缓存几乎失效



公共溢出区

❖ overflow area

单独开辟一块连续空间

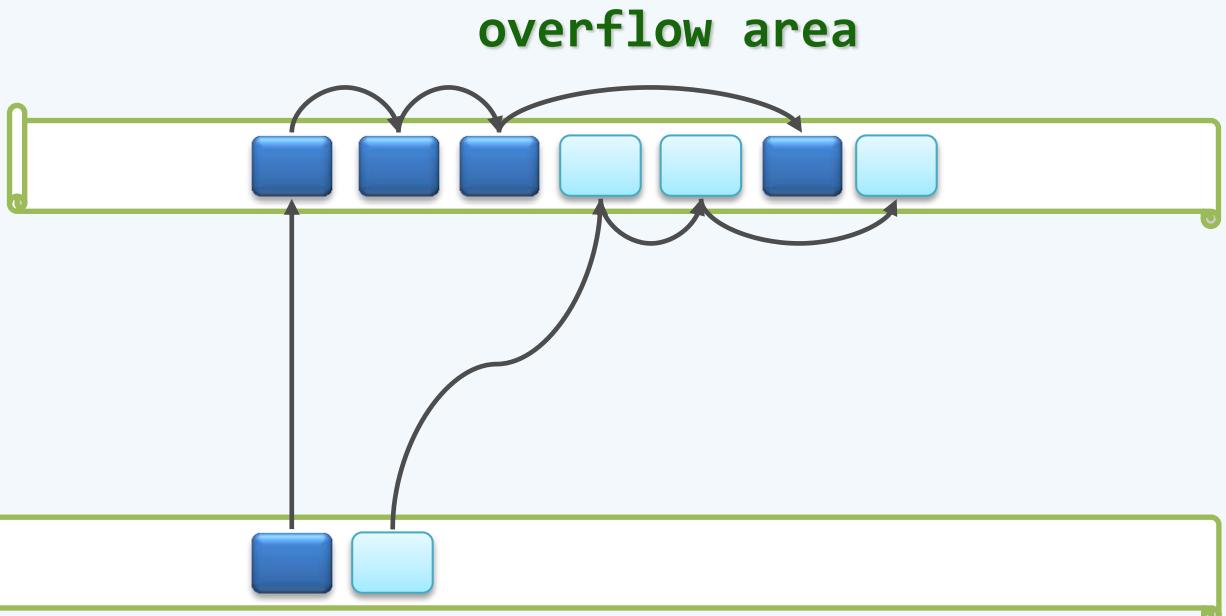
发生冲突的词条，顺序存入此区域

❖ 结构简单

算法易于实现

❖ 但是，不冲突则已，一旦发生冲突

最坏情况下，处理冲突词条所需的时间正比于溢出区的规模



9. 词典

排解冲突

封闭散列

邓俊辉

deng@tsinghua.edu.cn

开放定址

❖ **closed hashing** 必然对应于 **open addressing** :

只要有必要，任何散列桶都可以接纳任何词条

❖ **probing sequence/chain** :

为每个词条，都需 **事先约定** 若干 **备用桶**，优先级逐次下降

❖ 查找：沿 **查找链**，逐个转向 **下一桶单元**，直到...

- 命中 **成功**，或者
- 抵达一个空桶（已遍历所有冲突的词条）**失败**

❖ 具体地，**查找链**应如何**约定**？

线性试探：策略

- ❖ **Linear probing** 一旦冲突，则试探后一紧邻桶单元

$[\text{hash}(\text{key}) + \boxed{1}] \% M$

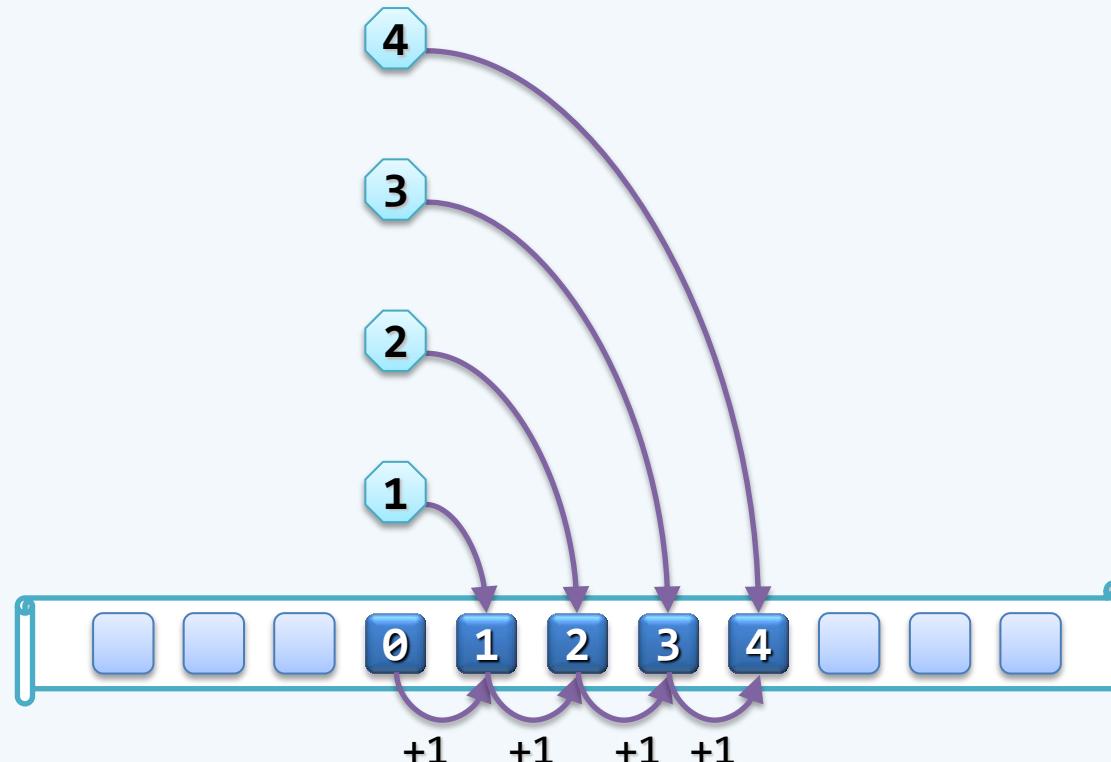
$[\text{hash}(\text{key}) + \boxed{2}] \% M$

$[\text{hash}(\text{key}) + \boxed{3}] \% M$

$[\text{hash}(\text{key}) + \boxed{4}] \% M$

...

直到命中**成功**，或抵达空桶**失败**



线性试探：评估

❖ 在散列表内部解决冲突

- 无需附加的（指针、链表或溢出区等）空间
- 结构本身保持简洁

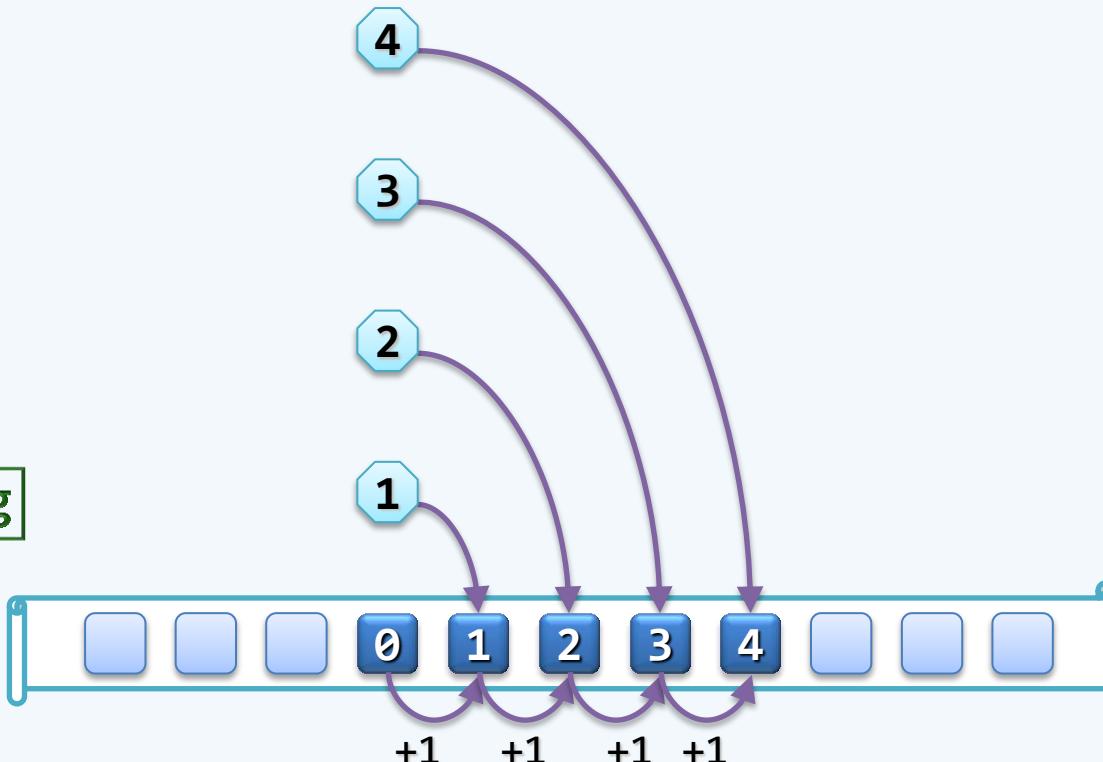
❖ 只要还有空桶，迟早会找到

❖ 但是，已发生过（并已排解）的冲突

会导致本不必发生的冲突 clustering

❖ 好在，查找链具有局部性

可充分利用系统缓存，有效减少 I/O



插入 + 删 除

❖ 插入：新词条若尚不存在，则存入查找终止处的空桶

❖ 查找链：可能因而彼此重叠

❖ 删除：简单地清除命中的桶？

- 某条查找链可能因此被切断，后续词条可能丢失——明明存在，却访问不到
- 多条查找链可能因此被切断...

❖ 具体地，此时的查找链应如何约定？

9.词典

排解冲突

懒惰删除

邓俊辉

deng@tsinghua.edu.cn

策略

- ❖ **lazy removal** : 仅做**删除标记**，查找链**不必续接**
- ❖ 此后，带有删除标记的桶所扮演的**角色**，因具体的操作类型而异
 - **查找**词条时
被视作“**必不匹配的非空桶**”，查找链在此得以**延续**
 - **插入**词条时
被视作“**必然匹配的空闲桶**”，可以用来**存放**新词条
- ❖ 具体过程的实现...

算法

```

❖ template <typename K, typename V> int Hashtable<K, V>::probe4Hit(const K& k)
{
    int r = hashCode(k) % M; //从首个桶起沿查找链，跳过所有冲突的和被懒惰删除的桶
    while ( ht[r] && ( k != ht[r]->key ) || !ht[r] && lazilyRemoved(r) )
        r = ( r + 1 ) % M; //线性试探（注意并列判断的次序，命中可能性更大者前置）
    return r; //调用者根据ht[r]是否为空，即可判断查找是否成功
}

```

```

❖ template <typename K, typename V> int Hashtable<K, V>::probe4Free(const K& k) {
    int r = hashCode(k) % M; //从首个桶起
    while ( ht[r] ) r = (r + 1) % M; //沿查找链找到第一个空桶（无论是否懒惰删除）
    return r; //调用者根据ht[r]是否为空，即可判断查找是否成功
}

```

9.词典

排解冲突

平方试探

我真的以为

这样何尝不是一种所谓的解脱

要背负的辛苦又有谁能够清楚

那内心的冲突

邓俊辉

deng@tsinghua.edu.cn

平方试探

❖ Quadratic probing

以**平方数**为距离，确定下一试探桶单元

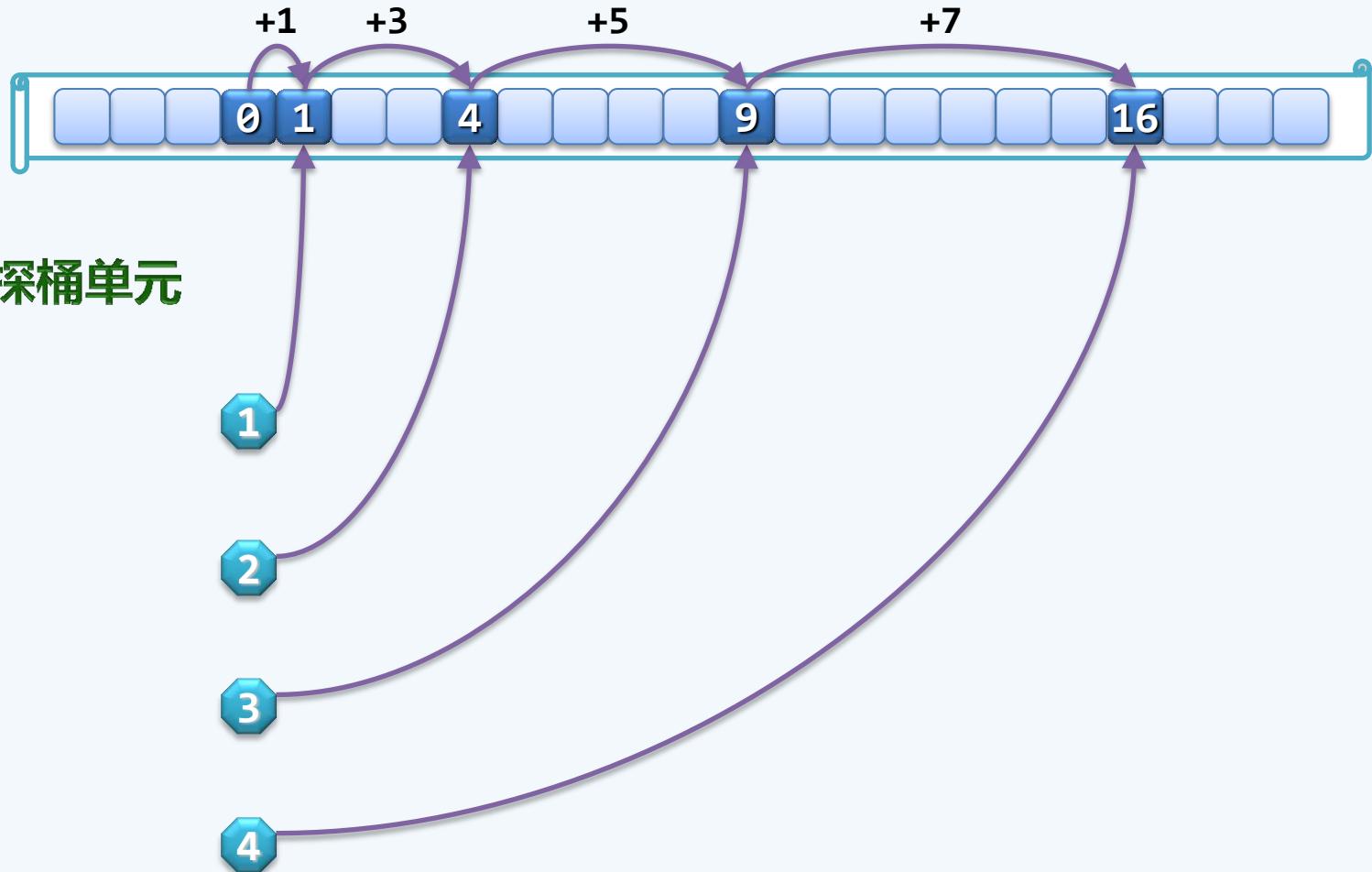
$$[\text{hash(key)} + [1^2]] \% M$$

$$[\text{hash(key)} + [2^2]] \% M$$

$$[\text{hash(key)} + [3^2]] \% M$$

$$[\text{hash(key)} + [4^2]] \% M$$

...



优点、缺点及疑惑

◆ 数据聚集现象有所缓解

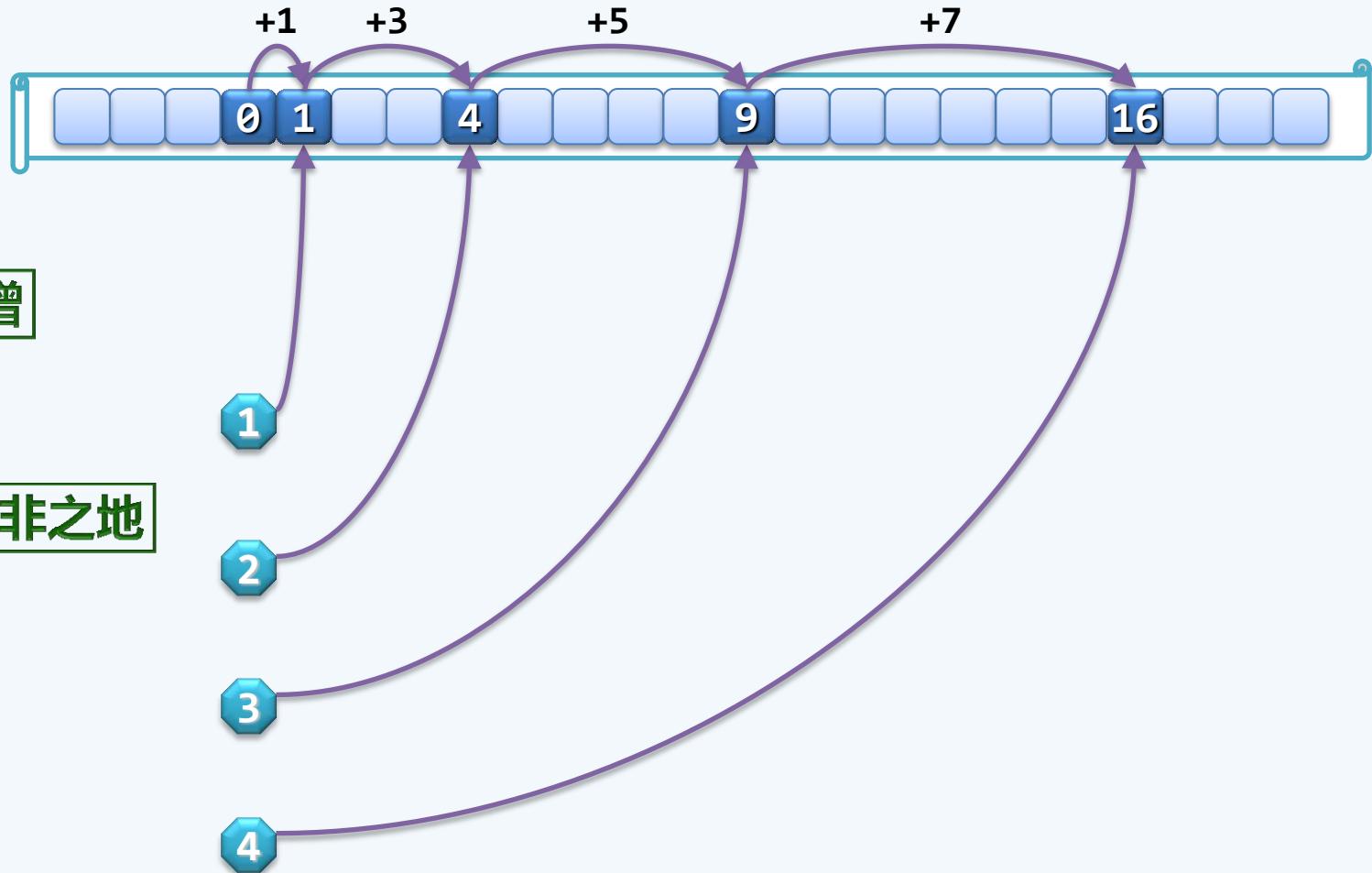
查找链上，各桶间距线性递增

一旦冲突，可聪明地跳离是非之地

◆ 若涉及外存，I/O将激增

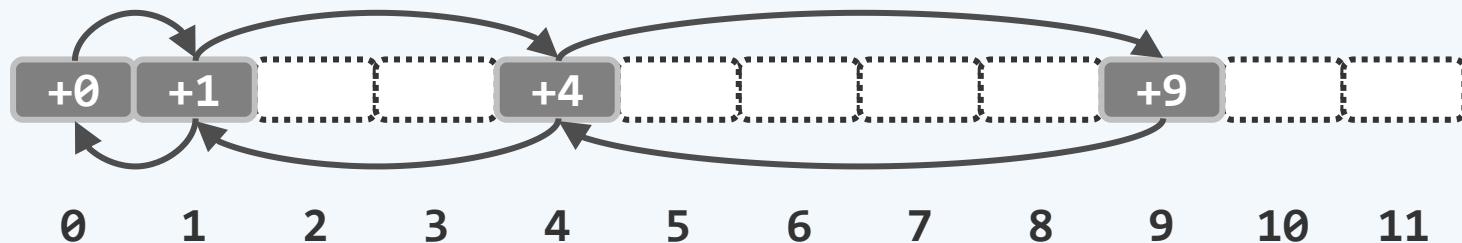
◆ 只要有空桶，就...一定能...找出来吗？

//毕竟不是挨个试探



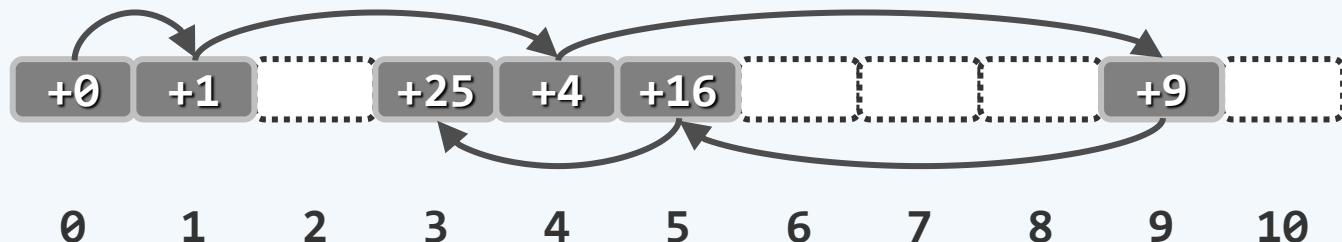
装填因子，须足够小！

❖ $\{ 0, 1, 2, 3, 4, 5, \dots \}^2 \% 12 = \{ 0, 1, 4, 9 \}$



M若为合数： $n^2 \% M$ 可能的取值必然少于 $[M/2]$ 种——此时，只要对应的桶均非空...

❖ $\{ 0, 1, 2, 3, 4, 5, \dots \}^2 \% 11 = \{ 0, 1, 4, 9, 5, 3 \}$



M若为素数： $n^2 \% M$ 可能的取值恰好会有 $[M/2]$ 种——此时，恰由查找链的前 $[M/2]$ 项取遍

❖ 定理：若M是素数，且 $\lambda \leq 0.5$ ，就一定能够找出；否则，不见得

查找链前缀，必足够长！

❖ 反证：假设存在 $0 \leq a < b < \lceil M/2 \rceil$ ，使得

沿着查找链，第a项和第b项彼此冲突

❖ 于是： a^2 和 b^2 自然属于M的某一同余类，亦即

$$a^2 \equiv b^2 \pmod{M}$$

$$b^2 - a^2 = (b + a) \cdot (b - a) \equiv 0 \pmod{M}$$

❖ 然而： $0 < b - a \leq b + a < M$ —— 这与M为素数矛盾

❖ 那么，另一半的桶，可否也利用起来呢...

9.词典

排解冲突

双向平方试探

邓俊辉

deng@tsinghua.edu.cn

策略

❖ 自冲突位置起，依次向后试探

[hash(key) $+ 1^2$] % M

[hash(key) $- 1^2$] % M

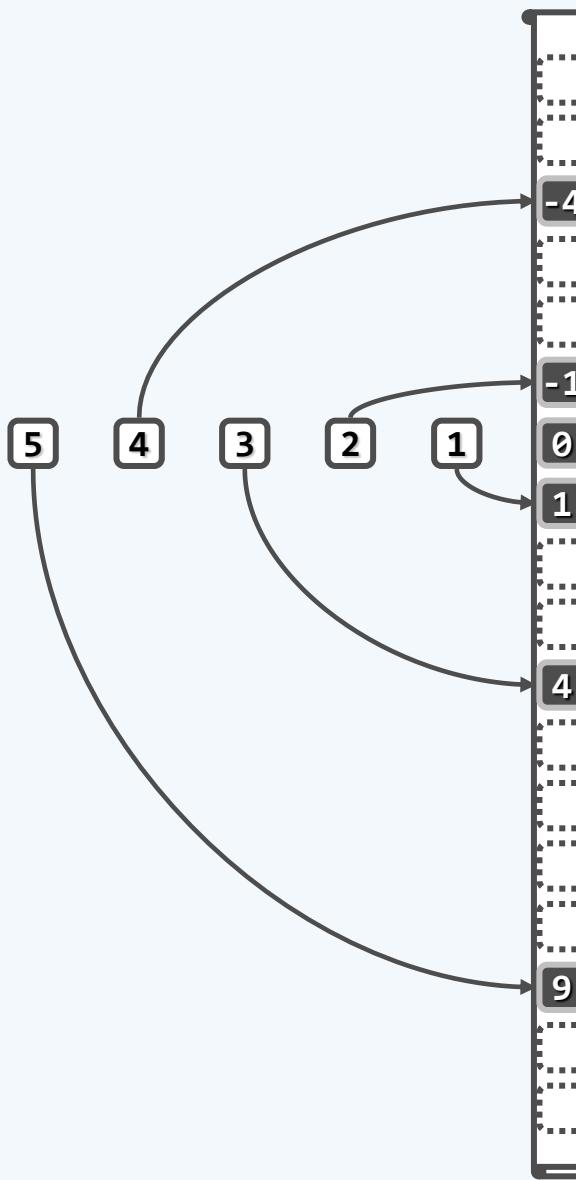
[hash(key) $+ 2^2$] % M

[hash(key) $- 2^2$] % M

[hash(key) $+ 3^2$] % M

[hash(key) $- 3^2$] % M

...



查找链，彼此独立？

❖ 正向和逆向的子查找链，各包含 $\lceil M/2 \rceil$ 个互异的桶

$-\lfloor M/2 \rfloor, \dots, -2, -1, \boxed{0}, 1, 2, \dots, \lfloor M/2 \rfloor$

$\pm i^2$		-36	-25	-16	-9	-4	-1	0	1	4	9	16	25	36
M	5				1	4	0	1	4					
	7				5	3	6	0	1	4	2			
	11		8	6	2	7	10	0	1	4	9	5	3	
	13	3	1	10	4	9	12	0	1	4	9	3	12	10

❖ 除了 $\boxed{0}$ ，这两个序列是否还有...其它公共的桶？

4k + 3

❖ 两类素数：

3 5 7 11 13 17 19 23 29 31 ...

❖ 表长取作素数 $M = 4 \times k + 3$ ，必然可以保证查找链的前 M 项均互异

$\pm i^2$		-36	-25	-16	-9	-4	-1	0	1	4	9	16	25	36
M	5				1	4	0	1	4					
	7				5	3	6	0	1	4	2			
	11		8	6	2	7	10	0	1	4	9	5	3	
	13	3	1	10	4	9	12	0	1	4	9	3	12	10

❖ 反之， $M = 4 \times k + 1$ 就... 必然不可使用？

Two-Square Theorem of Fermat

❖ 任一素数 p 可表示为一对整数的平方和，当且仅当 $p \% 4 = 1$

❖ 只要注意到：

$$\begin{aligned}(u^2 + v^2) \cdot (s^2 + t^2) &= (us + vt)^2 + (ut - vs)^2 \\ &= (vs + ut)^2 + (vt - us)^2\end{aligned}$$

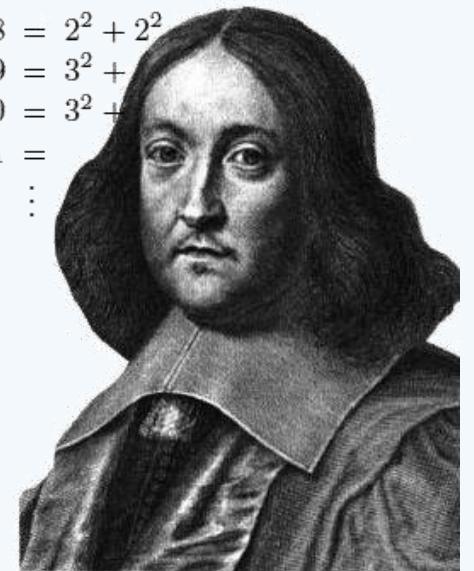
$$\begin{aligned}(2^2 + 3^2) \cdot (5^2 + 8^2) &= (10 + 24)^2 + (16 - 15)^2 \\ &= (15 + 16)^2 + (24 - 10)^2\end{aligned}$$

❖ 就不难推知：

任一自然数 n 可表示为一对整数的平方和，当且仅当

在其素分解中，形如 $M = 4 \times k + 3$ 的每一素因子均为偶数次方

$$\begin{aligned}1 &= 1^2 + 0^2 \\ 2 &= 1^2 + 1^2 \\ 3 &= \\ 4 &= 2^2 + 0^2 \\ 5 &= 2^2 + 1^2 \\ 6 &= \\ 7 &= \\ 8 &= 2^2 + 2^2 \\ 9 &= 3^2 + 0^2 \\ 10 &= 3^2 + 1^2 \\ 11 &= \\ &\vdots\end{aligned}$$



9.词典

排解冲突

再散列

邓俊辉

deng@tsinghua.edu.cn

Double Hashing

- ❖ 预先约定第二散列函数： **hash2()**
- ❖ 一旦发生冲突，以 **hash2(key)** 为偏移 **增量**，重新确定地址
 - [$\text{hash}(\text{key}) + [1] \times \text{hash2}(\text{key})$] % M
 - [$\text{hash}(\text{key}) + [2] \times \text{hash2}(\text{key})$] % M
 - [$\text{hash}(\text{key}) + [3] \times \text{hash2}(\text{key})$] % M
 - ...
- 直到发现一个空桶
- ❖ 比如， $\text{hash2}() = 1$ 时，即是线性试探

9.词典

排解冲突

重散列

邓俊辉

deng@tsinghua.edu.cn

Rehashing

❖ template <typename K, typename V> //随着装填因子增大，冲突概率、排解难度都将激增

```
void Hashtable<K, V>::rehash() { //此时，不如“集体搬迁”至一个更大的散列表  
    int old_capacity = M; Entry<K, V>** old_ht = ht; N = 0;  
  
    ht = new Entry<K, V>*[ M = primeNLT( 2*M ) ]; //新表容量至少加倍  
    memset( ht, 0, sizeof( Entry<K, V>* ) * M ); //初始化各桶  
    release(lazyRemoval); lazyRemoval = new Bitmap(M); //新建位图，容量至少加倍  
    for ( int i = 0; i < old_capacity; i++ ) //扫描原桶数组  
        if ( old_ht[i] ) //将非空桶中的词条逐一  
            put( old_ht[i]->key, old_ht[i]->value ); //插入至新表  
    release( old_ht ); //释放原桶数组  
}
```

9. 词典

桶排序

算法

Don't put all your eggs
in one basket.

邓俊辉

deng@tsinghua.edu.cn

简单情况

❖ 给定 $[0, m)$ 内的n个互异整数，如何高效地排序？ //必有 $n \leq m$

❖ 借助散列表 $E[0, m)$ //各元素仅需1个bit

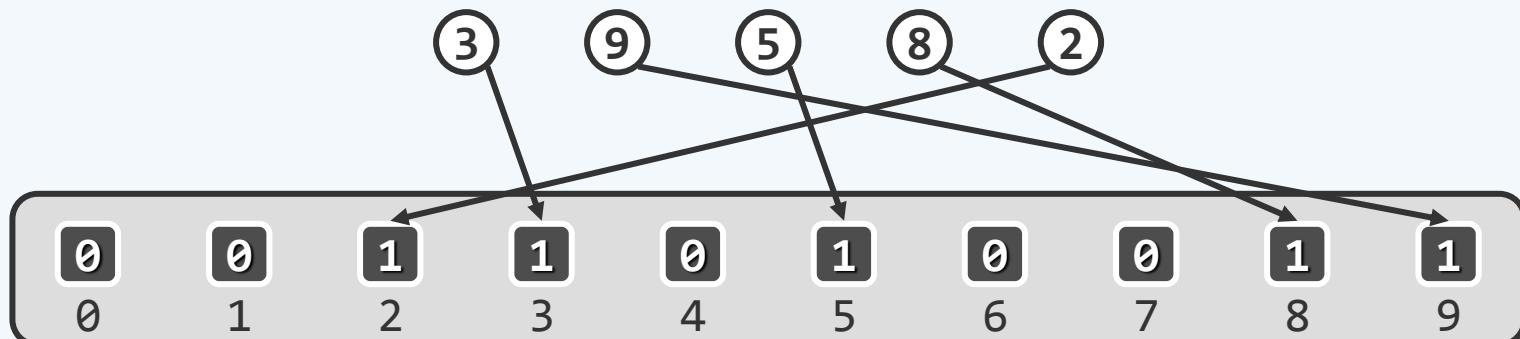
initialization `for i = 0 to m - 1, let E[i] = 0` // $O(m)$ ，可优化至 $O(1)$

distribution `for each key in the input, let E[key] = 1` // $O(n)$

enumeration `for i = 0 to m - 1, output i if E[i] = 1` // $O(m)$

❖ 空间： $O(m)$

时间： $O(n + m)$



一般情况

❖ 进一步地，若允许关键码 **重复** //此时未必 $n \leq m$ ，甚至可能 $m \ll n$

比如，清华大学2013级本科生按**生日**排序，则有 $n = 3300$, $m = 365$

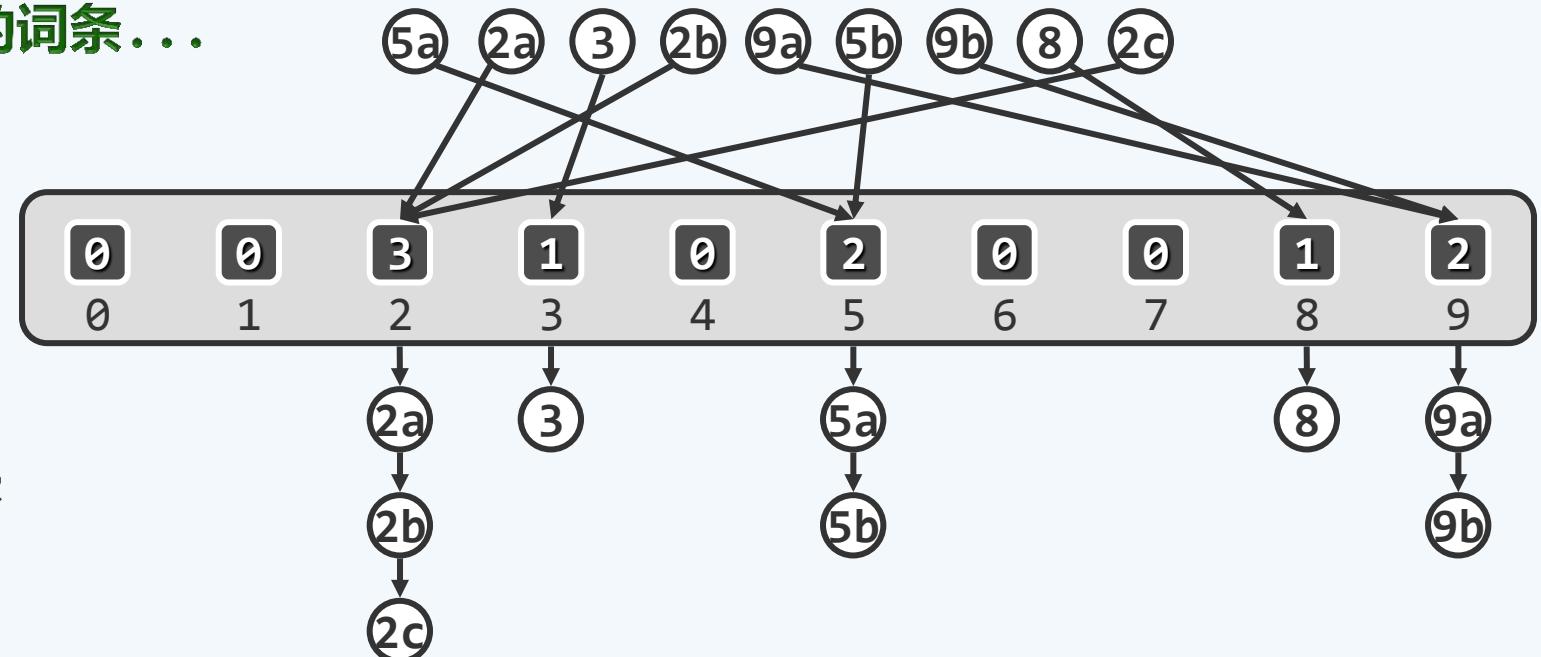
❖ 依然使用散列表，相互冲突的词条...

组成**独立链**

❖ 空间复杂度

= 散列表长 + 所有链表总长

= $\theta(m + n)$ //改用向量呢



一般情况

- ❖ **initialization** 初始化散列表（开辟空间、设置各桶的表头） //如有必要，可以优化
- ❖ **distribution** 扫描各词条，散列并插至对应桶的链表 //插入位置有讲究
- ❖ **collection** 扫描各桶，串接所有非空链表 //串接次序和方向也有讲究
- ❖ 只要实现得当，必能保证稳定性，即雷同词条的次序与输入相同 //其重要性，远超直觉
- ❖ 时间复杂度 = $\mathcal{O}(m) + \mathcal{O}(n) + \mathcal{O}(m) = \mathcal{O}(n + m)$
- ❖ 大量词条重复时 $m \ll n$ ，性能接近于线性
- ❖ 关键码均匀分布时，亦是如此

9. 词典

桶排序

最大缝隙

“恒纪元能持续多长时间？”

“一天或一个世纪，每次多长谁都说不准。”

“那乱纪元会持续多长时间呢？”

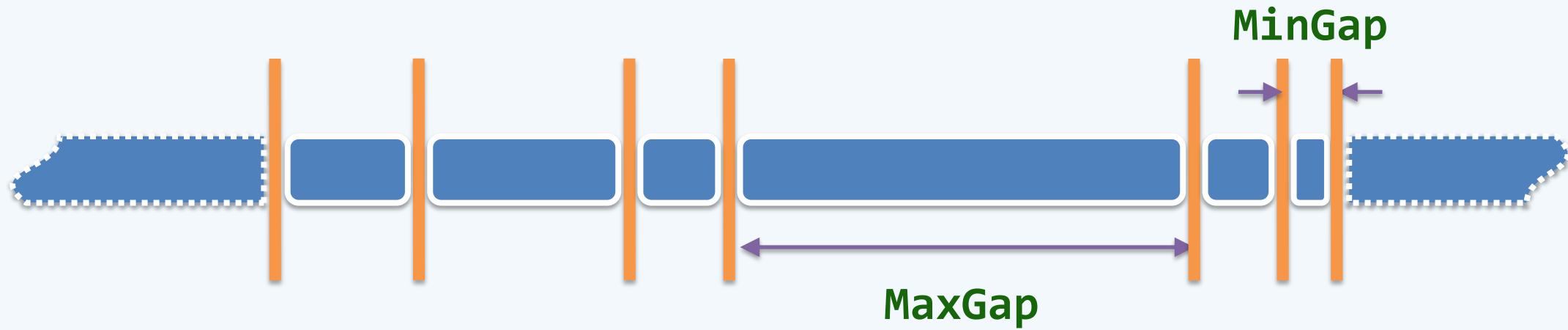
“不是说过嘛，除了恒纪元都是乱纪元，两者互为对方的间隙。”

邓俊辉

deng@tsinghua.edu.cn

MaxGap

❖ 任意 n 个互异点均将实轴分为 $n - 1$ 段有界区间，其中哪一段最长？

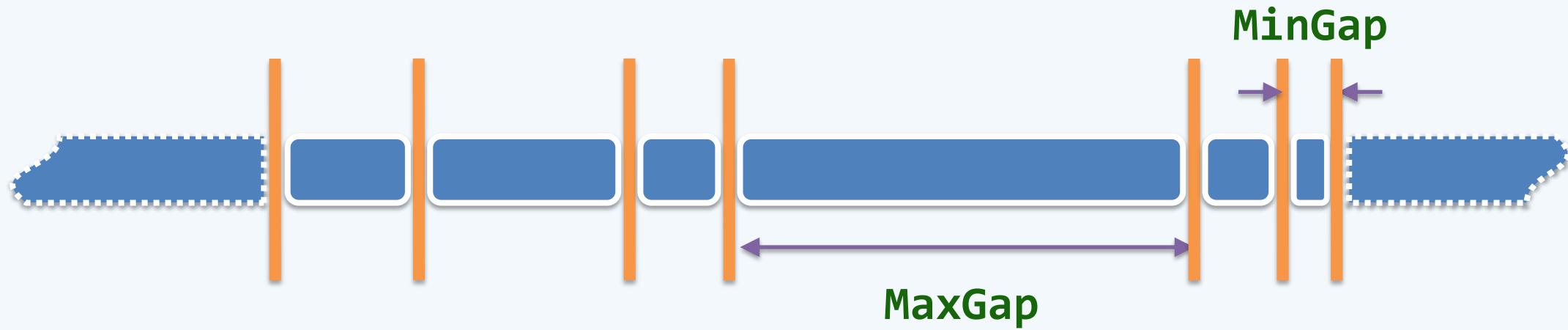


❖ 如果不追求效率，显而易见的方法是...

平凡算法

❖ 对所有点排序 //最坏情况下 $\Omega(n \log n)$

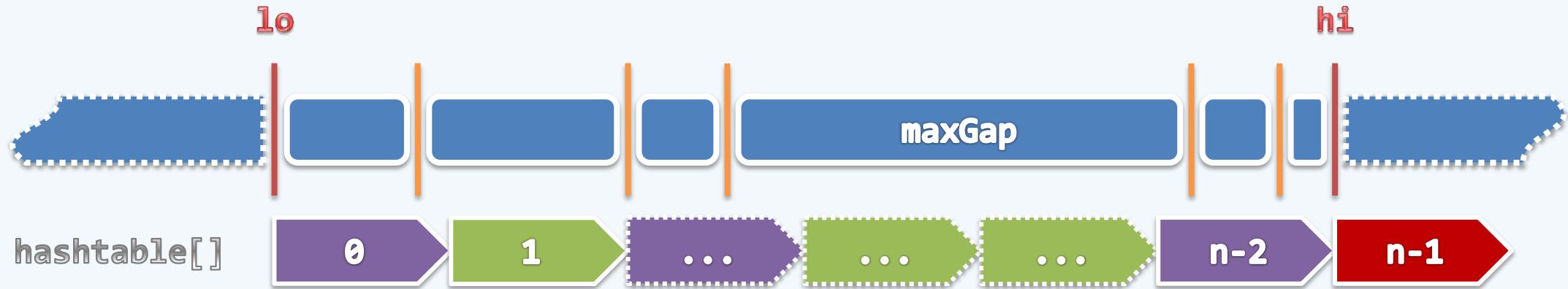
依次计算各相邻点对的间距，保留最大者 // $\Theta(n)$



❖ 可否更快？

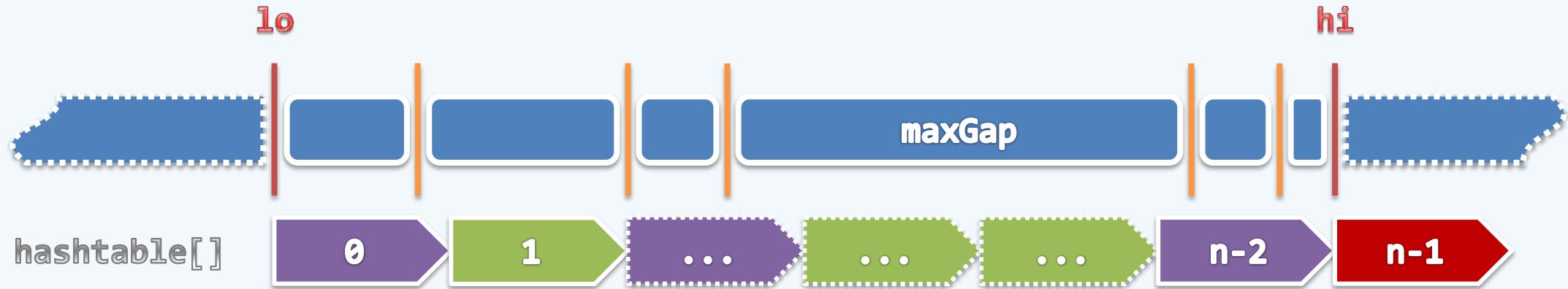
❖ 采用分桶策略，可改进至 $O(n)$ 时间...

线性算法



- ❖ 找到最左点、最右点 $\mathcal{O}(n)$ //一趟线性扫描
- 将有效范围均匀地划分为 $n-1$ 段 (n 个桶) $\mathcal{O}(n)$ //相当于散列表
- 通过散列，将各点归入对应的桶 $\mathcal{O}(n)$ //模余法
- 在各桶中，动态记录最左点、最右点 $\mathcal{O}(n)$ //可能相同甚至没有
- 算出相邻 (非空) 桶之间的 “距离” $\mathcal{O}(n)$ //一趟遍历足矣
- 最大的距离即MaxGap $\mathcal{O}(n)$ //画家算法

正确性



❖ 正确性：MaxGap至少与相邻的两个桶相交

等价地，定义MaxGap的点不可能属于同一个桶

❖ 对称的MinGap问题： $n - 1$ 段有界区间中，何者最短？

可否沿用上法，以突破 $\Omega(n \log n)$ 下界？

9. 词典

基数排序

算法

邓俊辉

deng@tsinghua.edu.cn

词典序

❖ 有时，关键码由多个域组成： k_t, k_{t-1}, \dots, k_1

❖ 若将各域视作字母，则关键码即单词

❖ 于是，可按照词典方式排序 lexicographic order

❖  A >  K >  Q >  J >  10 > ... >  2 >

 A >  K >  Q >  J >  10 > ... >  2 >

 A >  K >  Q >  J >  10 > ... >  2 >

 A >  K >  Q >  J >  10 > ... >  2

算法

❖ 低位优先：自 k_1 到 k_t ，依次以各域为序，做一趟桶排序

3 280	5 1 12	51 1 2	432 0	6441
4 320	7 2 14	72 1 4	328 0	5276
4 698	5 2 76	43 2 0	644 1	4320
5 112	3 2 80	64 4 1	511 2	7214
5 276	4 3 20	52 7 6	721 4	4698
6 441	6 4 41	32 8 0	527 6	3280
7 214	4 6 98	46 9 8	469 8	5112

9. 词典

基数排序

算法

邓俊辉

deng@tsinghua.edu.cn

正确性

❖ Radixsort的正确性何以见得？**数学归纳**！

❖ 归纳假设：在经过算法的**前*i*趟**之后，所有词条关于**最低的*i*位**有序——**第1趟**易见成立

❖ 假设**前*i - 1*趟**均成立，现考查**第*i*趟**排序之后的时刻

无非两种情况：1) 凡第*i*位**不同的**词条：即便此前**曾是**逆序，现在亦必已**转为**有序

2) 凡第*i*位**相同**的词条：得益于桶排序的**稳定性**，必**保持**原有次序

❖ 由此也可看出，只要实现**得法**，基数排序同样**稳定**

时间成本

❖ = 各趟桶排序所需时间之和

$$= n + 2m_1$$

$$+ n + 2m_2$$

+ ...

+ n + 2m_t // m_k 为各域的取值范围

$$= O(t \times (n + m)) //m = \max\{m_1, \dots, m_t\}$$

❖ 当 $m = O(n)$ 且 t 为常数时， $O(n)$ ！

❖ 在一些特定场合，Radixsort非常高效，比如...

整数排序

- ❖ 对于 $[0, n^d]$ 内的任意 n 个整数，如何高效排序？ // 常数 $d > 1$
- ❖ 理解： $1/d = \log(n) / \log(n^d)$ // 常对数密度，实际应用中不难满足
- ❖ 预处理：将所有关键码转换为 n 进制 形式 $x = (x_d, \dots, x_2, x_1)$
- ❖ 于是，原问题转化为 d 个域 的基数排序问题，可套用前述算法
- ❖ 排序时间 = $d(n + n) = O(n)$ // “突破”了此前确定的下界！
- ❖ 原因：1) 整数 取值范围 有限制；2) 不再是 基于比较 的计算模式
- ❖ 预处理 的用时尚未计入，它本身需要多少时间？回忆一下，此前的相关内容...

9. 词典

基数排序

整数排序

邓俊辉

deng@tsinghua.edu.cn

常对数密度的整数集

◊ 设 $1 < d$ 为常数，考查取自 $[0, n^d)$ 内的 n 个整数

$$\text{- 常规密度} = \frac{n}{n^d} = \frac{1}{n^{d-1}} \mapsto 0$$

$$\text{- 对数密度} = \frac{\ln n}{\ln n^d} = \frac{1}{d} = O(1)$$

◊ 亦即，这类整数集的对数密度不超过常数

◊ 这一附加条件，在实际应用中不难满足

若取 $d = 4$ ，则即便是就64位整数而言，也只需 $n > (2^{64})^{1/4} > 2^{16} = 65,536$

◊ 对于这类整数集，有无效率为 $\Theta(n \log n)$ 的排序算法？

线性排序算法

❖ 预处理：将所有元素转换为 n 进制形式：

$$x = (x_d, \dots, x_2, x_1)$$

❖ 于是，每个元素均转化为 d 个域，故可直接套用 Radixsort 算法

❖ 排序时间 = $d * (n + n) = O(n)$ //“突破”了此前确定的下界！

❖ 原因

- 整数取值范围有限制
- 不再是基于比较的计算模式

❖ 稍等！预处理本身需要多少时间？回忆一下，此前的相关内容...

9. 词典

计数排序

邓俊辉

deng@tsinghua.edu.cn

算法

❖ 回忆：基数排序中反复做的桶排序...

❖ 亦属 **小集合 + 大数据** 类型，是否可以更快？

❖ 仍以纸牌排序为例 $n \gg m = 4$ ，假设已按**点数**排序，以下对**花色**排序

1) 扫描 (方向**无所谓**) 所有纸牌，统计各种花色的**数量**

$\text{// } O(n)$



♣ * 3

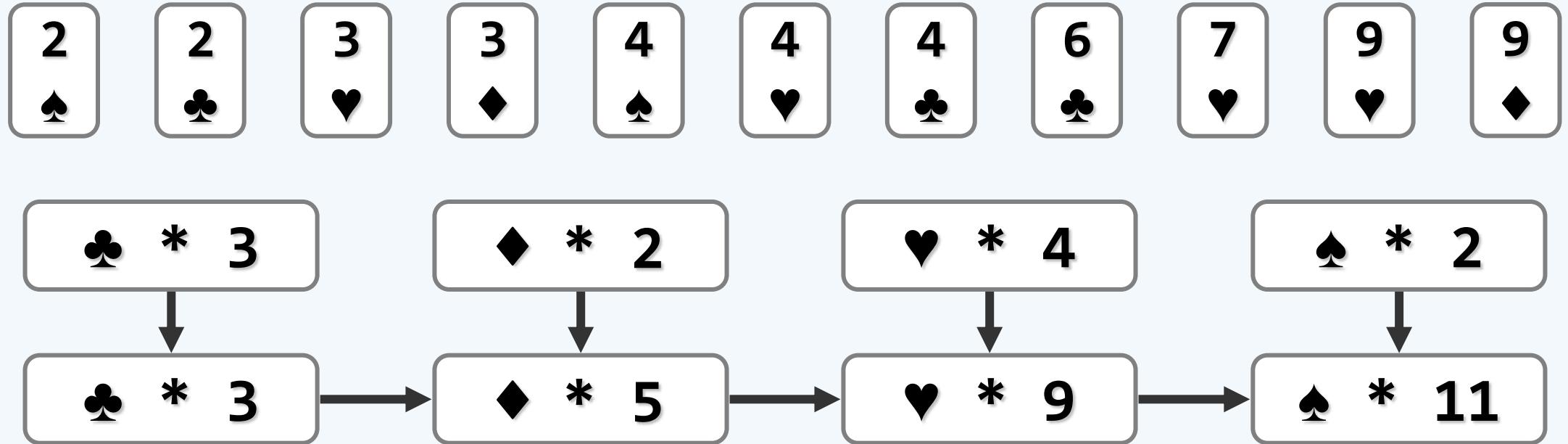
♦ * 2

♥ * 4

♠ * 2

算法

2) 自前向后扫描各桶，依次累加 $\text{// } O(m)$

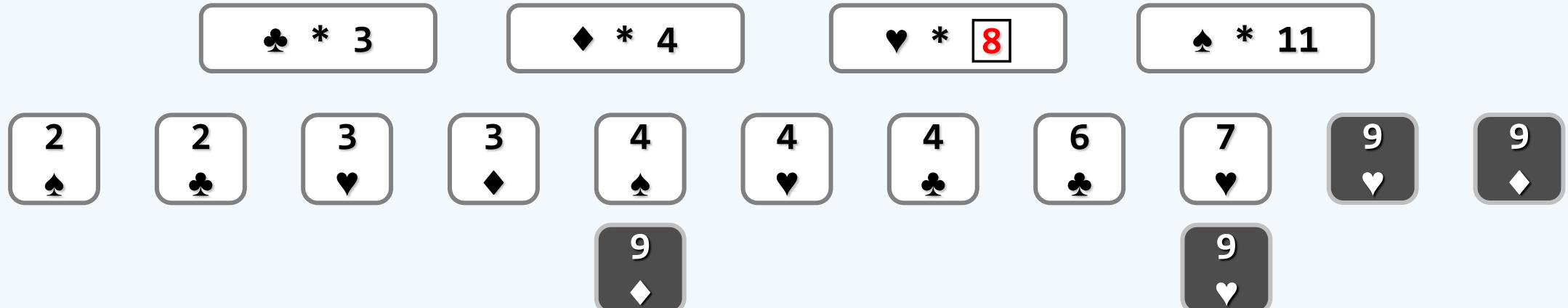


3) 自后向前扫描所有纸牌 $\text{// } O(n)$

对应桶的计数 减一

根据对应桶的 计数，确定其在最终有序序列中对应的 秩

实例



实例

♣ * 3

♦ * 4

♥ * 7

♠ * 11

2
♠2
♣3
♥3
♦4
♠4
♥4
♣6
♣7
♥9
♥9
♦9
♦7
♥9
♥

♣ * 2

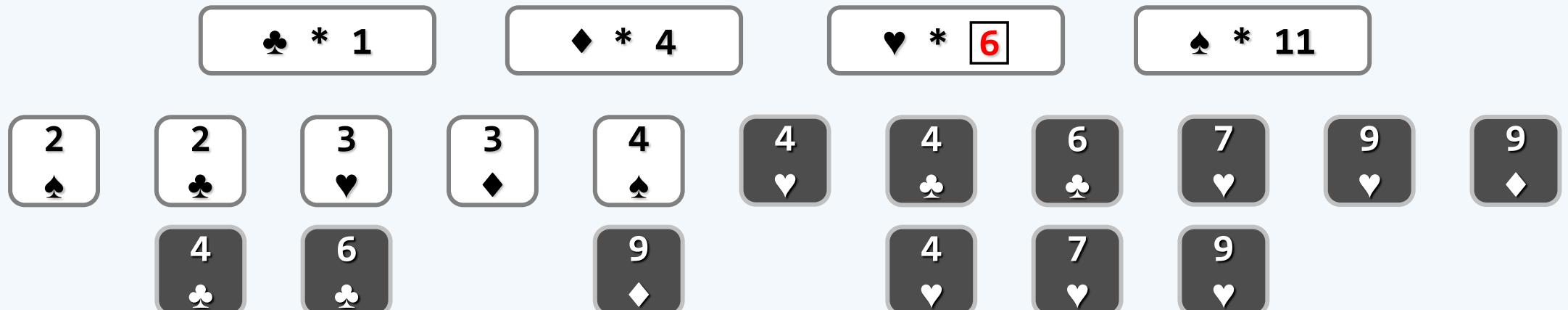
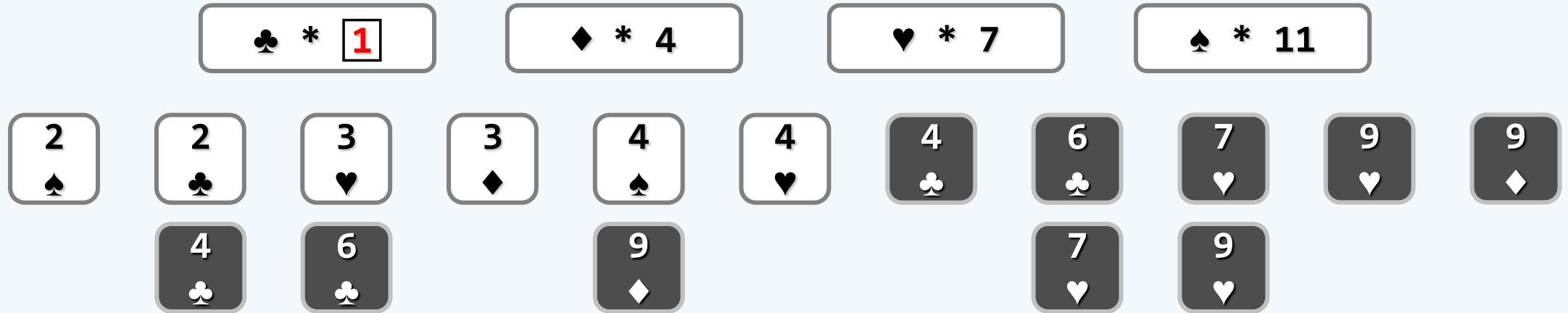
♦ * 4

♥ * 7

♠ * 11

2
♠2
♣3
♥3
♦4
♠4
♥4
♣6
♣7
♥9
♥9
♦6
♣9
♦7
♥9
♥

实例



实例

♣ * 1

♦ * 4

♥ * 6

♠ * 10

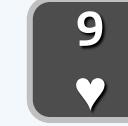


♣ * 1

♦ * 3

♥ * 6

♠ * 10



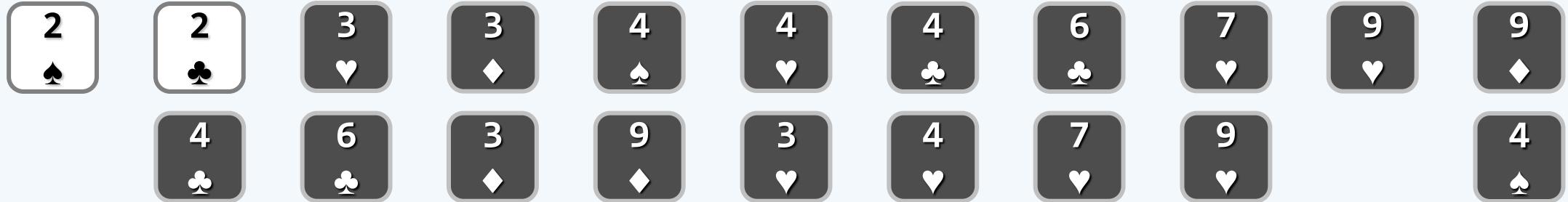
实例

♣ * 1

♦ * 3

♥ * 5

♠ * 10

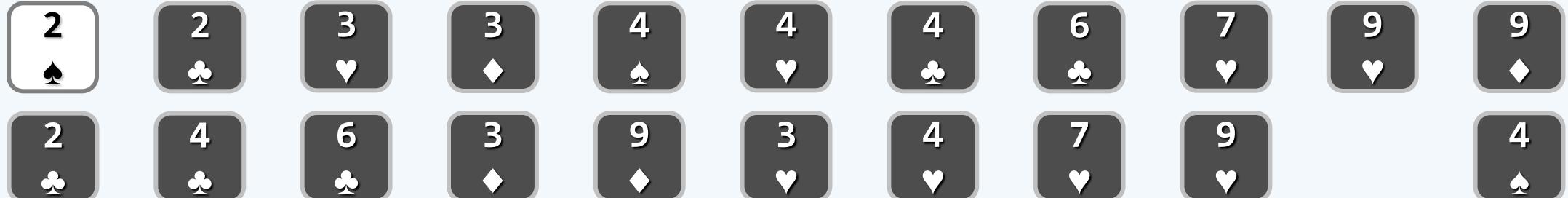


♣ * 0

♦ * 3

♥ * 5

♠ * 10



分析

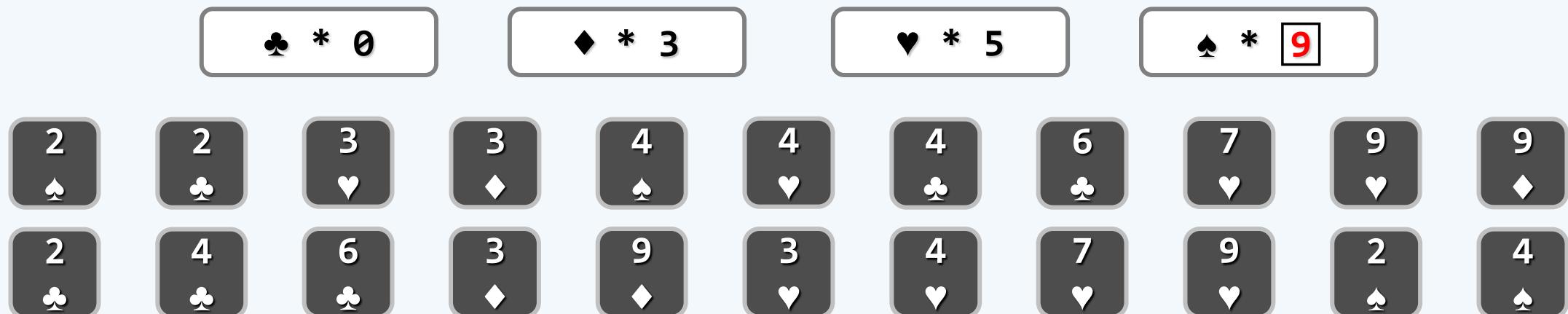
❖ 时间复杂度 = $\Theta(n + m + n) = \Theta(n)$

——高效处理大规模数据

❖ 空间复杂度 = $\Theta(m)$

——充分利用多重复特点

❖ 最后一步的扫描次序，可否改为自前向后？



9. 词典

跳转表

结构

去沿江上下，或二十里，或三十里，选高阜处置一
烽火台，每台用五十军守之。

邓俊辉

deng@tsinghua.edu.cn

动机与思路

❖ 可否综合向量与列表的优势，高效地实现词典接口？

具体地，如何使得各接口的效率均为 $\mathcal{O}(\log n)$ ？

❖ [William Pugh, 1989] Skip Lists: A Probabilistic Alternative to Balanced Trees

Skip lists are data structures that use

probabilistic balancing rather than

strictly enforced balancing

Algorithms for insertion and deletion in skip lists

are much simpler and significantly faster than

equivalent algorithms for balanced trees



❖ 本节介绍非确定型跳转表，确定型 deterministic 跳转表可自学

结构

分层次、相互耦合的多个列表： $s_0, s_1, s_2, \dots, s_h$

//层高 = h

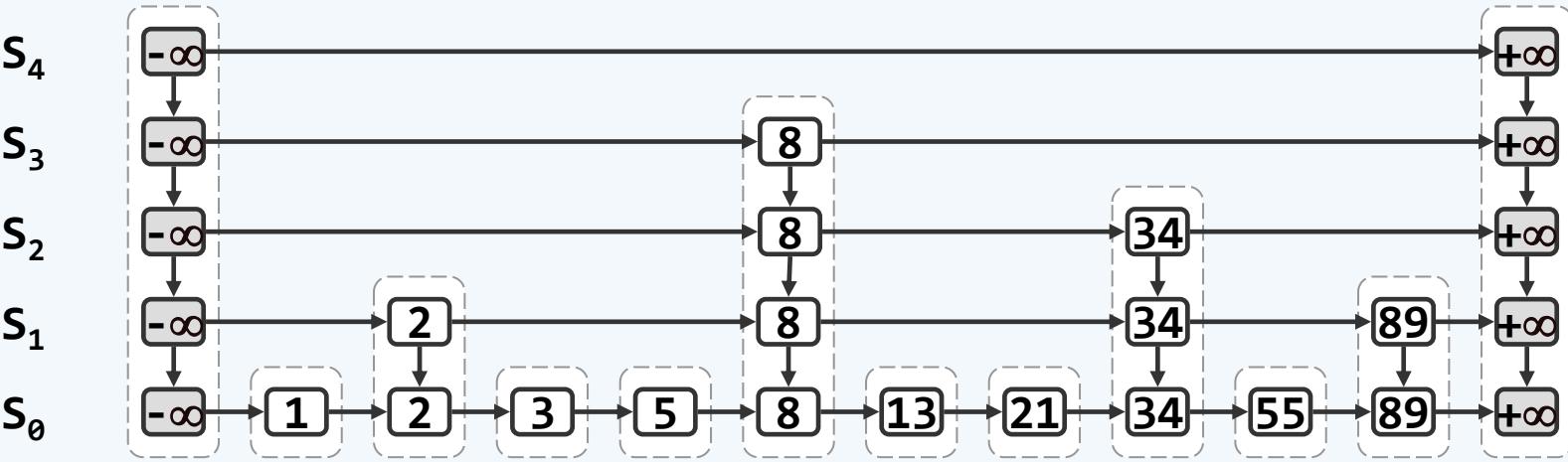
s_h 称作顶层 top

s_0 称作底层 bottom

各节点至多拥有四个引用：

横向为层 (level) : prev()、next() //设有头、尾哨兵

纵向成塔 (tower) : above()、below()



空间性能

❖ 较之常规的单层列表

每次操作过程中，需访问的节点是否会实质地增多？

每个节点都至多可能重复 h 次，空间复杂度是否因此有实质增加？ //先来回答后者...

❖ 生长概率逐层减半： S_k 中的每个关键码，在 S_{k+1} 中依然出现的概率，均为 $p = 1/2$

——暂且假设成立，稍后说明如何保证

❖ 可见，各塔的高度符合几何分布：

$$\Pr(h = k) = p^{k-1} \cdot (1 - p)$$

❖ 于是，期望的塔高 $E(h) = 1 / (1 - p) = 2$

❖ 什么，没有学过概率？不要紧，有初等的解释...

空间性能



既然生长概率逐层减半： S_0 中任一关键码在 S_k 中依然出现的概率均为 2^{-k}

$$\text{第 } k \text{ 层节点数的期望值 } E(|S_k|) = n \cdot 2^{-k} = n/2^k$$

于是，所有节点期望的总数（即各层列表所需空间总和）为

$$E(\sum_k |S_k|) = \sum_k E(|S_k|) = n \times (\sum_k 2^{-k}) < 2n = O(n)$$

定理：跳转表所需空间为 $O(n)$

类比：半衰期为 1 年的放射性物质中，各粒子的平均寿命不过 2 年

更为细致地，平均塔高的方差或标准差是否足够地小？

// 比照稍后对层高的分析

QuadList

```

template <typename T> class Quadlist { //四联表
private: int _size; QuadlistNodePosi(T) header, trailer; //规模、哨兵
protected: void init(); int clear(); //初始化、清除所有节点
public: QuadlistNodePosi(T) first() const { return header->succ; } //首节点
        QuadlistNodePosi(T) last() const { return trailer->pred; } //末节点
        T remove( QuadlistNodePosi(T) p ); //删除p
        QuadlistNodePosi(T) insertAfterAbove( //插入
            T const & e, //数据项e，使之成为
            QuadlistNodePosi(T) p, //p的后继，以及
            QuadlistNodePosi(T) b = NULL ); //b的上邻
};

    
```

Skiplist

```

template < typename K, typename V > class Skiplist : //多重继承
public Dictionary< K, V >, public List< Quadlist< Entry< K, V > > * > {
protected:
    bool skipSearch( ListNode< Quadlist< Entry< K, V > > * > * & qlist,
                      QuadlistNode< Entry< K, V > > * & p, K & k );
public:
    int size() { return empty() ? 0 : last()->data->size(); } //词条总数
    int level() { return List::size(); } //层高，即Quadlist总数
    bool put( K, V ); //插入 (Skiplist允许词条重复，故必然成功)
    V * get( K ); //读取 (基于skipSearch()直接实现)
    bool remove( K ); //删除
};

```

9. 词典

跳转表

查找

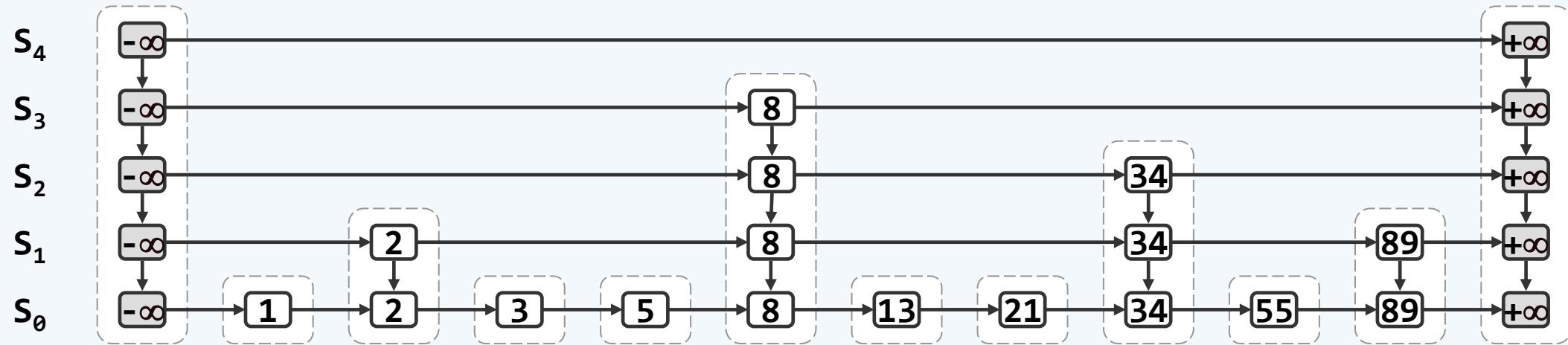
只见参仙老怪梁子翁笑嘻嘻的站起身来，向众人拱了拱手，缓步走到庭中，忽地跃起，左足探出，已落在欧阳克插在雪地的筷子之上，拉开架子，……，把一路巧打连绵的“燕青拳”使了出来，脚下纵跳如飞，每一步都落在竖直的筷子之上。

邓俊辉

deng@tsinghua.edu.cn

策略与算法

❖ 减而治之：由粗到细 = 由高至低



❖ 实例：成功 (21、34、1、89)，失败 (80、0、99)

❖ 观察：查找时间取决于横向、纵向的累计跳转次数

那么，是否可能因层次过多，首先导致纵向跳转过多？

实现

```

template <typename K, typename V> bool SkipList<K, V>::skipSearch(
    ListNode< QuadList < Entry< K, V > > * > * & qlist, //从指定层qlist的
    QuadListNode< Entry< K, V > > * & p, [K & k] { //首节点p出发，向右、向下查找k
        while ( true ) { //在每一层从前向后查找，直到出现更大的key，或者溢出至trailer
            while ( p->succ && ( p->entry.key <= k ) ) p = p->succ; p = p->pred;
            if ( p->pred && ( k == p->entry.key ) ) return [true]; //命中则成功返回
            qlist = qlist->succ; //否则转入下一层
            if ( ! qlist->succ ) return [false]; //若已到穿透底层，则意味着失败；否则...
            p = p->pred ? p->below : qlist->data->first(); //转至当前塔的下一节点
        } //确认：无论成功或失败，返回的p均为其中不大于e的最大者？
    } //体会：得益于哨兵的设置，哪些环节被简化了？
}

```

纵向跳转 / 层高

❖ 观察：一座塔高度不低于 | 低于 k 的概率 = p^k | $1 - p^k$

❖ 引理：随着 k 的增加， S_k 为空 / 非空 的概率急剧上升 / 下降

$$\Pr(|S_k| = 0) = (1 - p^k)^n \geq 1 - n \cdot p^k$$

$$\Pr(|S_k| > 0) \leq n \cdot p^k$$

❖ 推论：跳转表高度 $h = \mathcal{O}(\log n)$ 的概率极大

❖ 比如：若 $p = 1/2$ ，则第 $k = 3\log n$ 层非空（当且仅当 $h \geq k$ ）的概率为

$$\Pr(|S_k| > 0) \leq n \cdot p^k = n \cdot n^{-3} = 1/n^2 \rightarrow 0$$

❖ 结论：查找过程中，纵向跳转的次数，累计不过 $\text{expected-}\mathcal{O}(\log n)$

横向跳转 / 紧邻塔顶

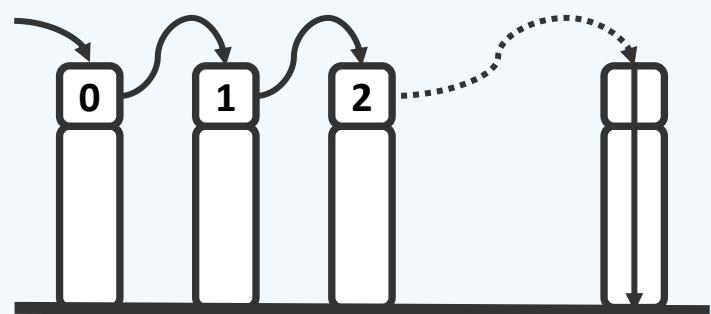
❖ 那么： 横向跳转是否可能很多次？比如 $\omega(\log n)$ ，甚至 $\Omega(n)$ ？

► ❖ 观察： 在同一水平列表中，横向跳转所经节点必然依次紧邻，而且每次抵达都是塔顶

❖ 于是： 若将沿同层列表跳转的次数记作 Y ，则有 几何分布：

$$\Pr(Y = k) = (1 - p)^k \cdot p$$

❖ 定理： $E(Y) = (1 - p)/p = (1 - 0.5)/0.5 = 1$ 次



同层列表中紧邻的塔顶节点，平均不过 $1 + 1 = 2$ 个

❖ 推论： 查找过程中横向跳转所需时间 $\leq \text{expected}(2h) = \text{expected-}\mathcal{O}(\log n)$

❖ 结论： 跳转表的每次查找可在 $\text{expected-}\mathcal{O}(\log n)$ 时间内完成

9.词典

跳转表

插入

如果一个人遇到不可解之事，把脑子想穿了，也找不到其中的原因，怎么办呢？他或许会去庙里烧香，把自己的难题交给算命先生，听人他们的摆布。

邓俊辉

deng@tsinghua.edu.cn

算法 & 实现

```

template <typename K, typename V> bool Skiplist< K, V >::put( K k, V v ) {
    Entry< K, V > e = Entry< K, V >( k, v ); //将被随机地插入多个副本的新词条

    if ( empty() ) insertAsFirst( new Quadlist< Entry< K, V > > ); //首个Entry

    ListNode< Quadlist< Entry< K, V > > * > * qlist = first(); //从顶层列表的

    QuadlistNode< Entry< K, V > > * p = qlist->data->first(); //首节点开始

    if ( skipSearch( qlist, p, k ) ) //查找适当的插入位置——若已有雷同词条，则

        while ( p->below ) p = p->below; //强制转到塔底

    qlist = last();

    QuadlistNode< Entry< K, V > > * b = qlist->data->insertAfterAbove( e, p );

    /* ... 以下，紧邻于p的右侧，以新节点b为基座，自底而上逐层长出一座新塔 ... */

```

算法 & 实现

```

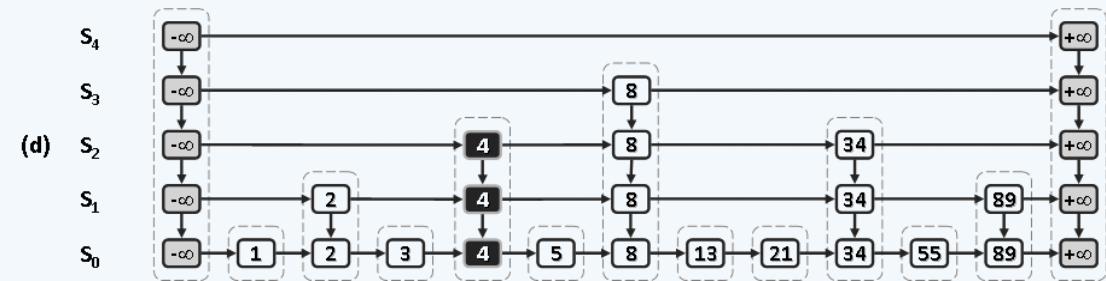
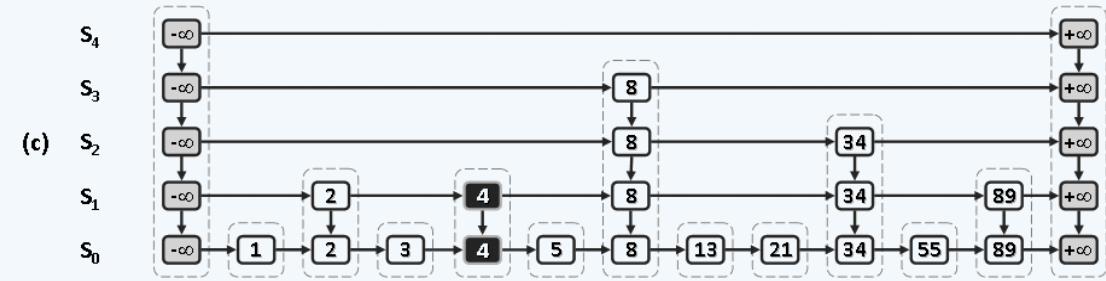
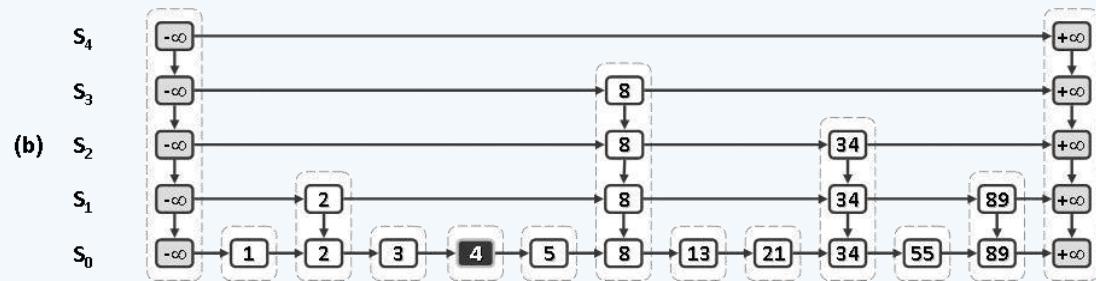
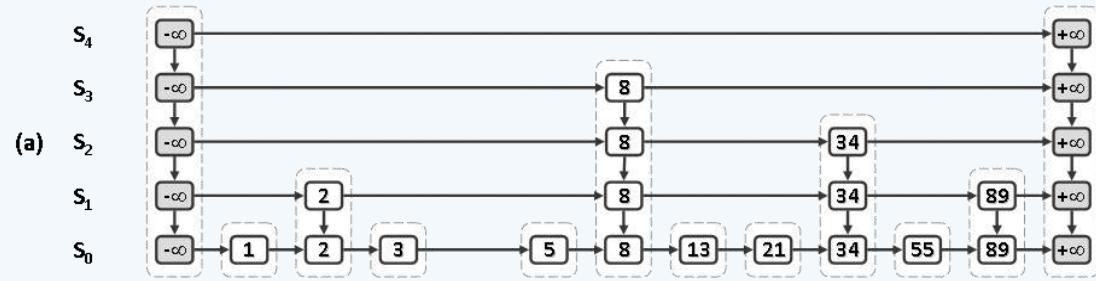
while ( rand() & 1 ) { //经投掷硬币，若新塔需再长高，则先找出不低于此高度的...
    while ( qlist->data->valid(p) && ! p->above ) p = p->pred; //最近前驱
    if ( ! qlist->data->valid(p) ) { //若该前驱是header
        if ( qlist == first() ) //且当前已是最顶层，则意味着必须
            insertAsFirst( new Quadlist< Entry< K, V > > ); //先创建新层，再
        p = qlist->pred->data->first()->pred; //将p转至上一层的header
    } else p = p->above; //否则，可径自将p提升至该高度
    qlist = qlist->pred; //上升一层，并在该层将新节点
    b = qlist->data->insertAfterAbove( e, p, b ); //插至p之后、b之上
} //while ( rand() & 1 )

return true; //SkipList允许重复元素，故插入必成功
}

```

实例

❖ 实例：一般 (4、20、40)，边界 (0、99)



9.词典

跳转表

删除

邓俊辉

deng@tsinghua.edu.cn

//插入的逆过程

```
template <typename K, typename V> bool Skiplist< K, V >::remove( K k ) {  
    if ( empty() ) return false; //空表  
  
    ListNode< Quadlist< Entry< K, V > > * > *  qlist = first(); //从顶层Quadlist  
  
    QuadlistNode< Entry< K, V > > *  p = qlist->data->first(); //的首节点开始  
  
    if ( ! skipSearch( qlist, p, k ) ) //目标词条不存在，则  
        return false; //直接返回  
  
    /* ... */
```

do { //若目标词条存在，则逐层拆除与之对应的塔

```
QuadlistNode< Entry< K, V > > * lower = p->below; //记住下一层节点
```

```
qlist->data->remove(p); //删除当前层的节点后，再
```

```
p = lower; qlist = qlist->succ; //转入下一层
```

```
} while ( qlist->succ ); //如上不断重复，直到塔基
```

```
while ( ! empty() && first()->data->empty() ) //逐一地
```

```
List::remove( first() ); //清除已可能不含词条的顶层Quadlist
```

```
return true; //删除操作成功完成
```

```
} //体会：得益于哨兵的设置，哪些环节被简化了？
```

9. 词典

位图

数据结构

邓俊辉

deng@tsinghua.edu.cn

集合及其接口

- ❖ 集合，也可视作一种抽象数据类型
- ❖ 这里不妨考查其中的特例——整数集
- ❖ 依散列的思想观之，所有离散集

或者显式地本身就是整数集

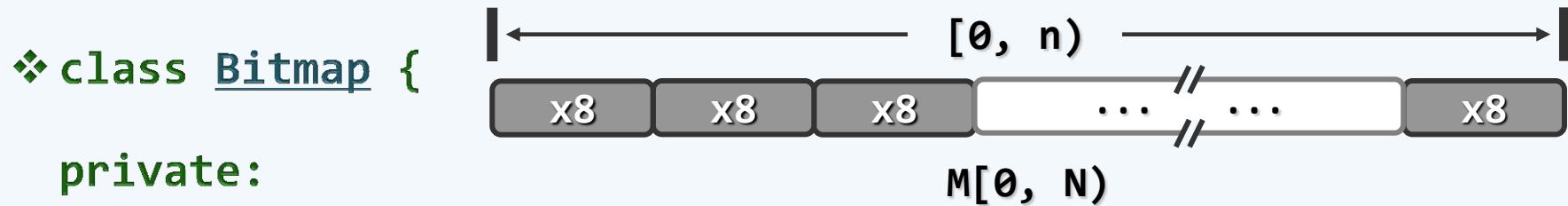
或者隐式地可转换为整数集

- ❖ **ADT**
 - `void set(int k); // 将整数k加入当前集合`
 - `void clear(int k); // 从当前集合中排除整数k`
 - `bool test(int k); // 判断整数k是否属于当前集合`

- ❖ 如何具体实现？各接口的效率如何？

思路

❖ 使用物理地址连续的一段空间，各元素依次对应于一个比特位



char * M; //以char(8比特)为单位的比特位图

int N; //位图长度(以sizeof(char)为单位)

public:

Bitmap(int n = 8) { M = new char[N = (n+7)/8]; memset(M, 0, N); }

~Bitmap() { delete [] M; M = NULL; } //析构

void set(int k); void clear(int k); bool test(int k); //ADT

};

实现

```

❖ void set(int k) { expand(k); M[k >> 3] |= (0x80 >> (k & 0x07)); }

❖ void clear(int k) { expand(k); M[k >> 3] &= ~(0x80 >> (k & 0x07)); }

❖ bool test(int k) { expand(k); return M[k >> 3] & (0x80 >> (k & 0x07)); }

❖ void expand( int k ) { //Bitmap[k]出界时扩容，分摊O(1)
    if ( k < 8 * N ) return; //仍在界内，无需扩容
    int oldN = N; char * oldM = M;
    M = new char[ N = (2 * k + 7) / 8 ]; //与向量类似，加倍策略
    memcpy_s( M, N, oldM, oldN );
    delete [] oldM;
}

```

9. 词典

位图

典型应用

邓俊辉

deng@tsinghua.edu.cn

小集合 + 大数据

- ❖ 老问题：`int A[n]` 的元素均取自 $[0, m)$ ，如何剔除其中的重复者？
- ❖ 仿照 `vector::deduplicate()` 改进版：先排序，再扫描 —— $O(n \log n + n)$ —— 毫无压力
- ❖ 新特点：数据量虽大，但重复度极高 —— 想想我们电脑里的 MP3，不难理解
- ❖ 比如， $2^{24} = m \ll n = 10^{10}$
亦即， $10,000,000,000$ 个 24 位 无符号整数
- ❖ 如果采用内部排序算法，你至少需要 $4 * n = 40\text{GB}$ 内存
—— 否则，频繁的 I/O 将导致整体效率的低下
- ❖ 那么， $m \ll n$ 的条件，又应如何加以利用？

小集合 + 大数据

❖ Bitmap B(\boxed{m}); // $\theta(\boxed{m})$

```
for (int i = 0; i <  $\boxed{n}$ ; i++) B.set( A[i] ); // $\theta(n)$ 
```

```
for (int k = 0; k <  $\boxed{m}$ ; j++) if ( B.test( k ) ) /* ... */; // $\theta(m)$ 
```

❖ 总体运行时间 = $\theta(\boxed{n} + \boxed{m}) = \theta(\boxed{n})$

❖ 空间 = $\theta(\boxed{m})$ 就上例而言，降至： $m/8 = 2^{21} = 2\text{MB} \ll 40\text{GB}$

即便 $m = 2^{32}$ ，也不过： $2^{29} = 0.5\text{GB}$

❖ 拓展：搜索引擎的使用规律亦是如此，词表规模不大，但重复度极高

——如何从中剔除重复的索引词？

❖ 关键在于，如何将查询词表转换为某一集合——留作习题

筛法

❖ 如何计算出[0, n)之间的所有素数？

❖ 与其说是**计算**，不如说是**筛选**

❖ void Eratosthenes(int n, char * file) {

 Bitmap B(n); //创建位图

 B.set(0); B.set(1); //0和1都不是素数

 for (int i = 2; i < n; i++) //反复地，从下一

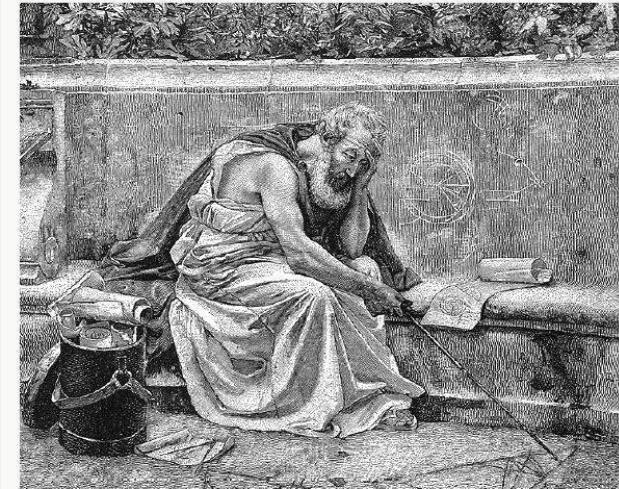
 if (! B.test(i)) //可认定的素数i起

 for (int j = 2 * i; j < n; j += i) //以i为间隔

 B.set(j); //将下一个数标记为合数

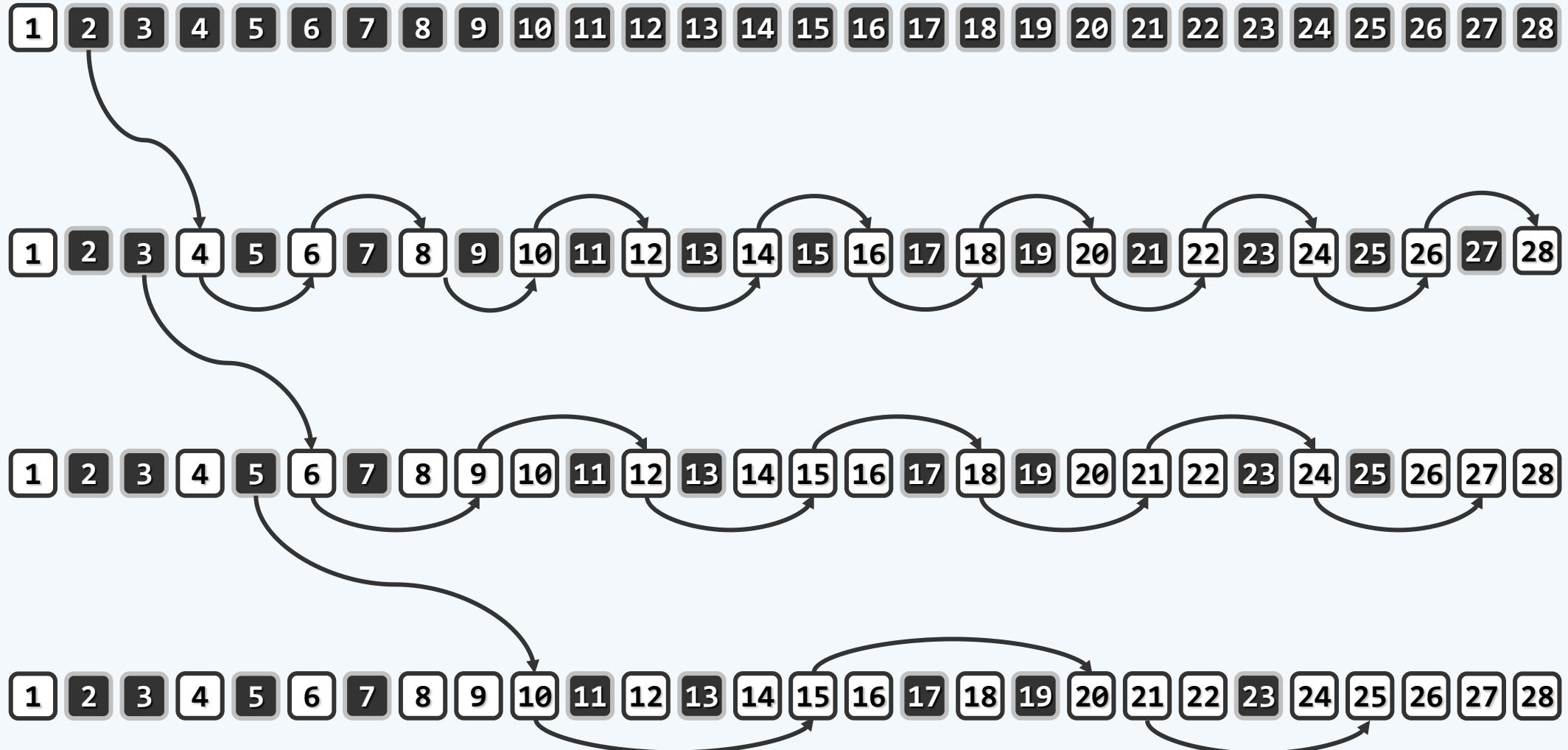
 B.dump(file); //将筛选出的素数转储至文件，以便日后使用

}



Eratosthenes
(276 ~ 194 B.C.)

实例



实例

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28

1	2
3	4
5	6
7	8
9	10
11	12
13	14
15	16
17	18
19	20
21	22
23	24
25	26
27	28
29	30

1	2	3
4	5	6
7	8	9
10	11	12
13	14	15
16	17	18
19	20	21
22	23	24
25	26	27
28	29	30

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25
26	27	28	29	30

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

效率

❖ 不计内循环，外循环自身每次仅一次加法、两次判断，累计 $\mathcal{O}(n)$

❖ 内循环每趟迭代 $\mathcal{O}(n/i)$ 步，由素数定理至多 $n/\ln(n)$ 趟，累计耗时不过

$$n/2 + n/3 + n/5 + n/7 + n/11 + \dots$$

$$< n/2 + n/3 + n/4 + n/5 + n/6 + \dots + n/(n/\ln(n))$$

$$= \mathcal{O}(n * [\ln(n/\ln(n)) - 1])$$

$$= \mathcal{O}(n\ln(n) - n\ln(\ln(n)))$$

$$= \mathcal{O}(n\log(n))$$

改进

- ❖ 循环起点 $i + i$, 可进一步改作 $i * i$

——为什么？

- ❖ 如此，每次内循环的长度，由

$$\mathcal{O}(n/i)$$

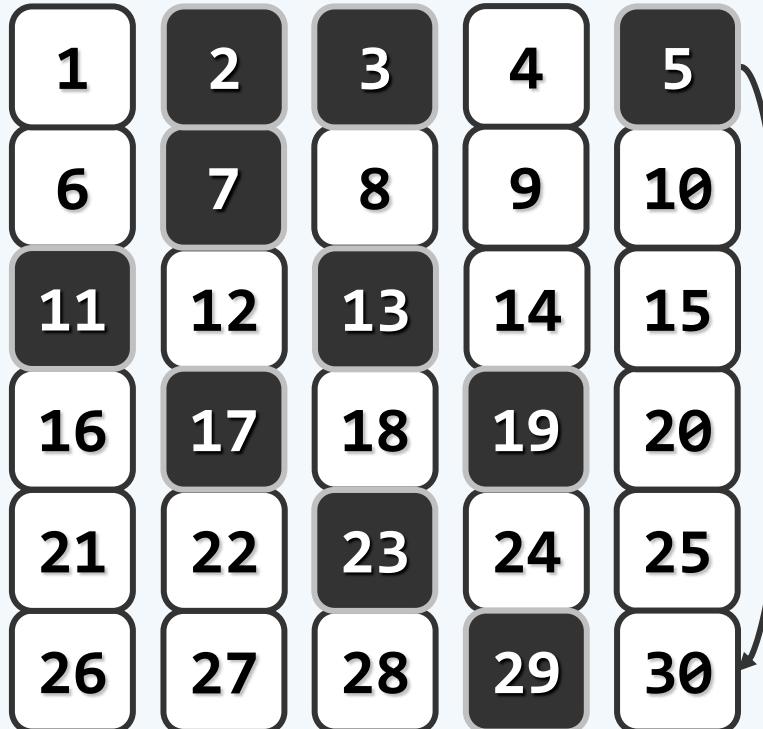
降至

$$\mathcal{O}(\max(1, n/i - i))$$

- ❖ 效率有所提高！

——从渐进的角度看，是否实质改进？

- ❖ 基于以上，如何实现 primeNLT(int low) ?



9. 词典

位图

快速初始化

邓俊辉

deng@tsinghua.edu.cn

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

❖ Bitmap的构造函数中，通过`memset(M, 0, N)`统一清零

这一步只需 $\mathcal{O}(1)$ 时间？不，实际上仍等效于诸位清零， $\mathcal{O}(N) = \mathcal{O}(n)$ ！

❖ 尽管这并不会影响上例的渐进复杂度，但并非所有问题都是如此

❖ 有时，对于大规模的散列表，初始化的效率直接影响到实际性能

例如：第11章中`bc[]`表的构造算法，需要 $\mathcal{O}(|\Sigma| + m) = \mathcal{O}(s + m)$ 时间

若能省去`bc[]`表各项的初始化，则可严格地保证是 $\mathcal{O}(m)$

❖ 有时，甚至会影响到算法的整体渐进复杂度

例如，为从 $n = 10^8$ 个32位整数中找出重复者，可仿造剔除算法... //但这里无需回收

因此，若能省去Bitmap的初始化，则只需 $\mathcal{O}(n)$ 时间

算法

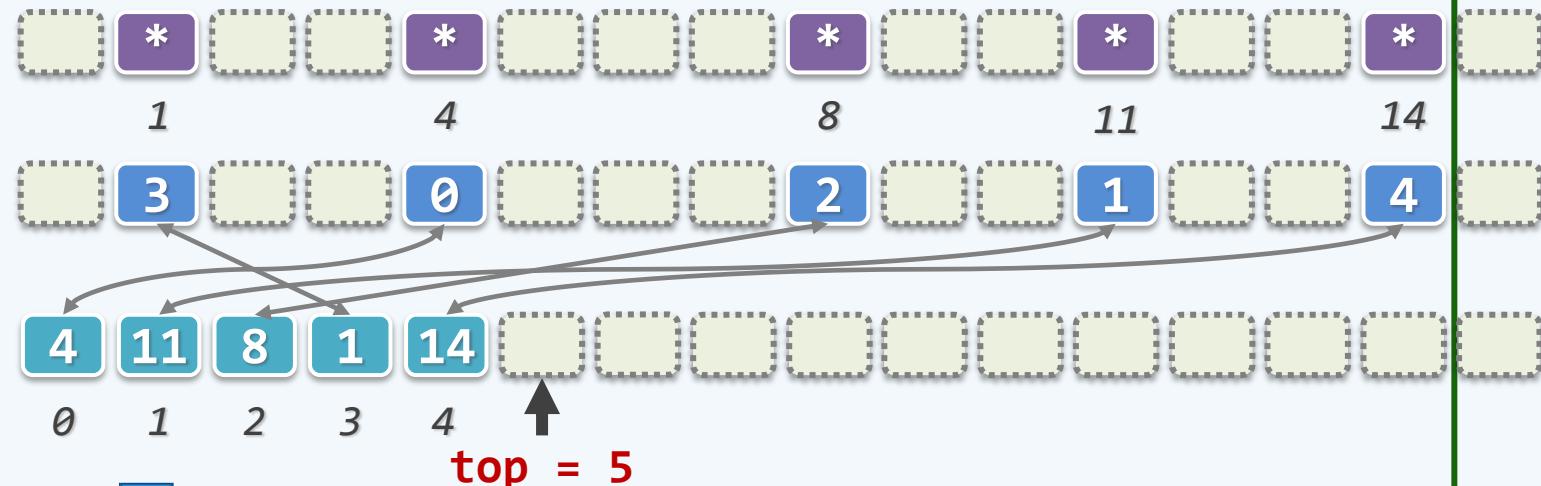
❖ // [J. Hopcroft, 1974] 为B[m]增设一对等长的Rank型向量

```
Rank F[ m ]; //向量from
```

```
Rank T[ m ]; //向量(栈)to
```

```
Rank top = 0; //to的栈顶
```

//总体空间复杂度仍为 $O(m)$



❖ bool Bitmap::test(int k) // $O(1)$

```
{ return 0 <= F[ k ] && F[ k ] < top && T[ F[ k ] ] == k; }
```

❖ void Bitmap::set(int k) // $O(1)$

```
{ if ( ! B.test( k ) ) { T[ top ] = k; F[ k ] = top++; } }
```

void Bitmap::clear(int k) { /* 更为复杂，习题[2-34] */ } // $O(1)$

9.词典

MD5

以小明大，见一叶落而知岁之将暮，
睹瓶中之冰，而知天下之寒

邓俊辉

deng@tsinghua.edu.cn

MD5

❖ 网络下载：除了数据包本身，你是否还注意过对应的**MD5**？

MD5？有何作用？

❖ UNIX：**/etc/passwd**

#LOGNAME:**PASSWORD**:UID:GID:USERINFO:HOME:SHELL

root:**zPDeHbougaPpA**:0:0:Super User:/bin/sh

ftp:**xyDfccTrt180xMy8**:3:3:FTP User:/usr/spool/ftp:

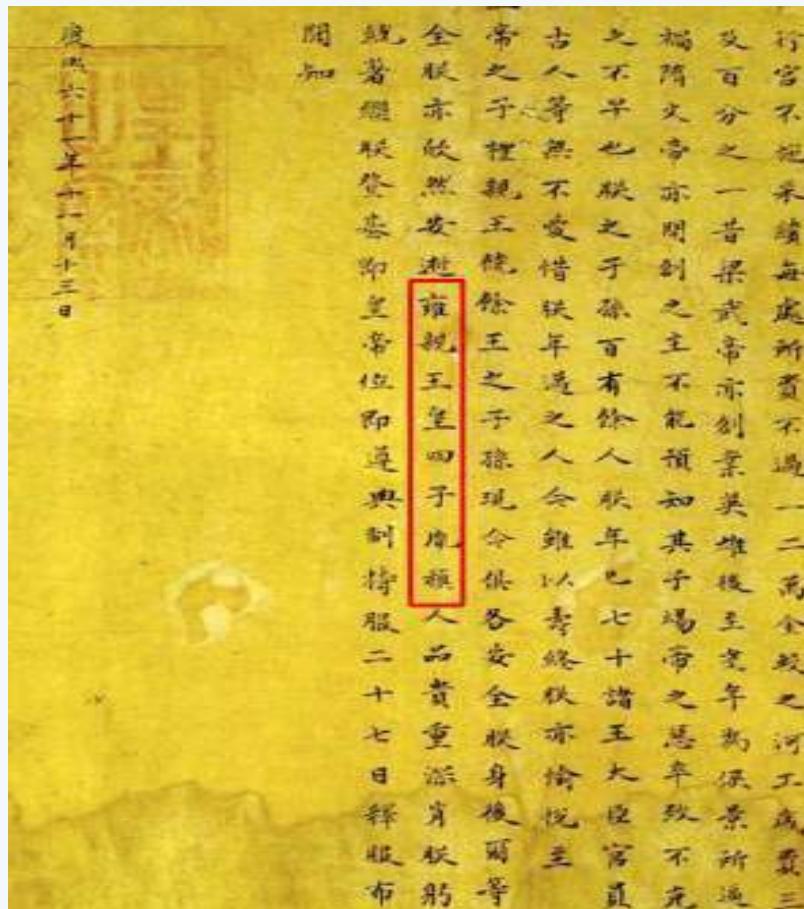
smart:**xmotTVoyumjls**:6:4:THUDSA Student:/bin/csh

#.....

❖ **PASSWORD**域，是如何**计算**出来的？如何**验证**的？反过来，会暴露**原文**吗？

Note: The i32 Intelligent Updater package cannot be used to update Symantec AntiVirus Corporate Edition [servers](#), but can be used to update Corporate Edition [clients](#).

File Name	Creation Date	Release Date	File Size	MD5 all
20101113-003-v5i32.exe FTP	11/13/2010	11/13/2010	80.1 MB	6DEBC39595EFC409C308AA0CA79667D0



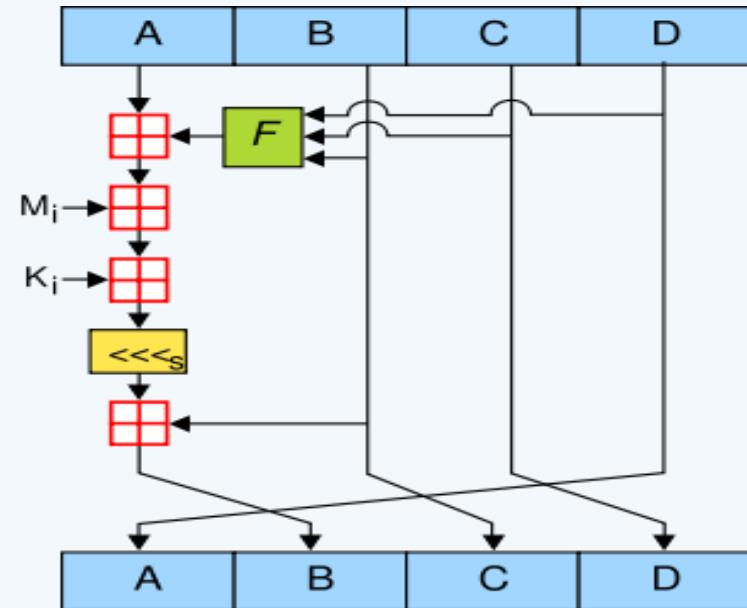
傳位于四子

以小见大——MD5

数字签名

$$16^{32} = 2^{128} = 10^{38}$$

正向易，逆向难



- ❖ 在很多场合，MD5都可大显身手…
- ❖ 数字签名：授权文档一经篡改随即失效 // 电子形式签署的法律文书
- ❖ 身份验证：OS或应用软件，根据口令做账号授权 // 充分而不必要
- ❖ 隐私通知：“你我知道是指谁，别人不知道是指谁” // 手机尾号
- ❖ 寻渊探源：文档之间的（局部、逻辑）复制关系 // 学术诚信鉴别
- ❖ 信道校验：带宽有限时，快速确认大数据的一致性 // 分布式存储、云存储
- ❖ 甄别赝伪：系统文件是否被病毒篡改？被何种病毒篡改？ // 特征
- ❖ ...

散列指纹

❖ 方法：在文档中选取若干序列，通过散列生成指纹

❖ 简单的累加？极易冲突！

I am Lord Voldemort = Tom Marvolo Riddle = He's Harry Potter

❖ 问题：序列越多越好吗？

技巧：在足以刻画文档的前提下，如何选取更少的序列？

❖ 单向函数： $f()$ 可快速计算，但 $f^{-1}()$ 的计算却十分...十分地耗时

❖ 整数乘法： $f(x, y) = x \cdot y$ // $\Theta(1)$ 时间——RAM模型

$f^{-1}(x \cdot y) = \langle x, y \rangle$ // $\Theta(?)$ ——质因数分解

算法

❖ Message-Digest Algorithm version.5, 1991

MIT Lab for Computer Science & RSA Data Security Inc.

将信息统一视作比特流，每 $512 = 16 \times 32$ bit 作为一个处理分组

经过四轮位运算变换，最终得到一个 128 bit 的大整数，即为MD5指纹

❖ 初始话： 如有必要，追加一个1及多个0，使总长形如 $512*k - 64$

将原信息的长度附加到末尾，使总长形如 $512*k$

❖ 128 bit，足够复杂？ $// 2^{128} = 3.4 \times 10^{38}$

要重构（破译）符合指定指纹的原始信息，似乎不那么简单...

课后

- ❖ 了解MD5算法的更多细节 //google("MD5 algorithm")
- ❖ 学习MD5相关软件的使用 //google("MD5 tool")
- ❖ 验证：某个文件的MD5，可能与另一个文件的MD5雷同
- ❖ 思考：如果有人掌握了MD5冲突的原理，则意味着什么？
- ❖ 了解MD5安全性的最新结论 //google("MD5 'WANG Xiaoyun'")
- ❖ 提交作业包时，尝试同时提交MD5指纹
- ❖ 了解分布式散列表的
 - 原理、方法与 //Overlay Network, Distributed Hash Table
 - 实现 //Napster, Gnutella; Chord, CAN, Tapestry, Pastry, ...

10. 优先级队列

概述

需求与动机

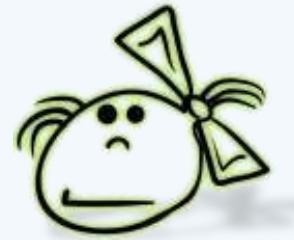
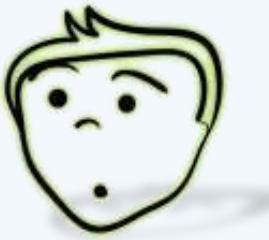
I cannot choose the best.

邓俊辉

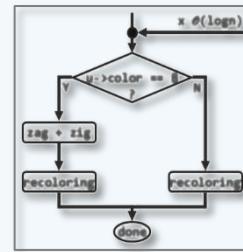
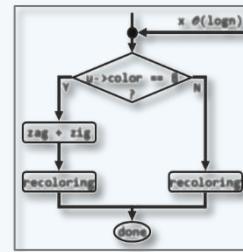
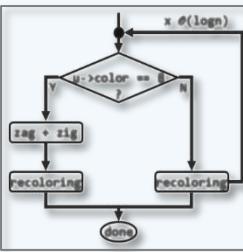
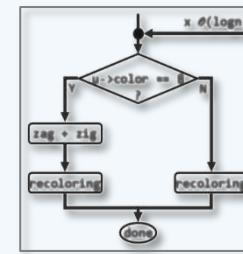
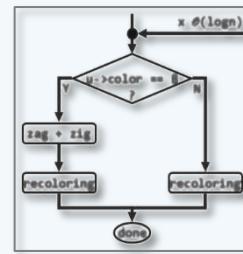
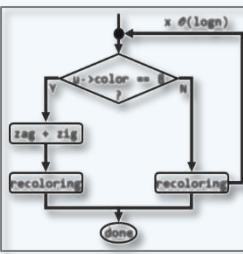
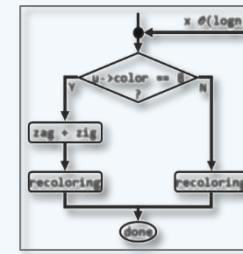
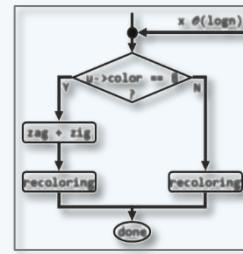
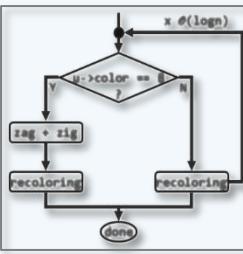
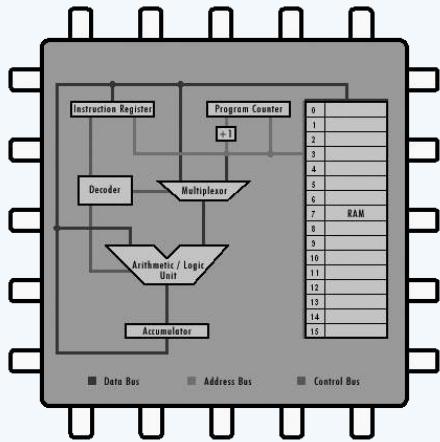
The best chooses me.

deng@tsinghua.edu.cn

夜间门诊



多任务调度



应用、算法与特点

❖ 应用 离散事件模拟

操作系统：任务调度、中断处理、GUI的MRU、...

输入法：词频调整

❖ 作为底层数据结构所支持的高效操作，是很多高效算法的基础

内部、外部、在线排序

贪心算法：Huffman编码、Kruskal

平面扫描算法中的事件队列

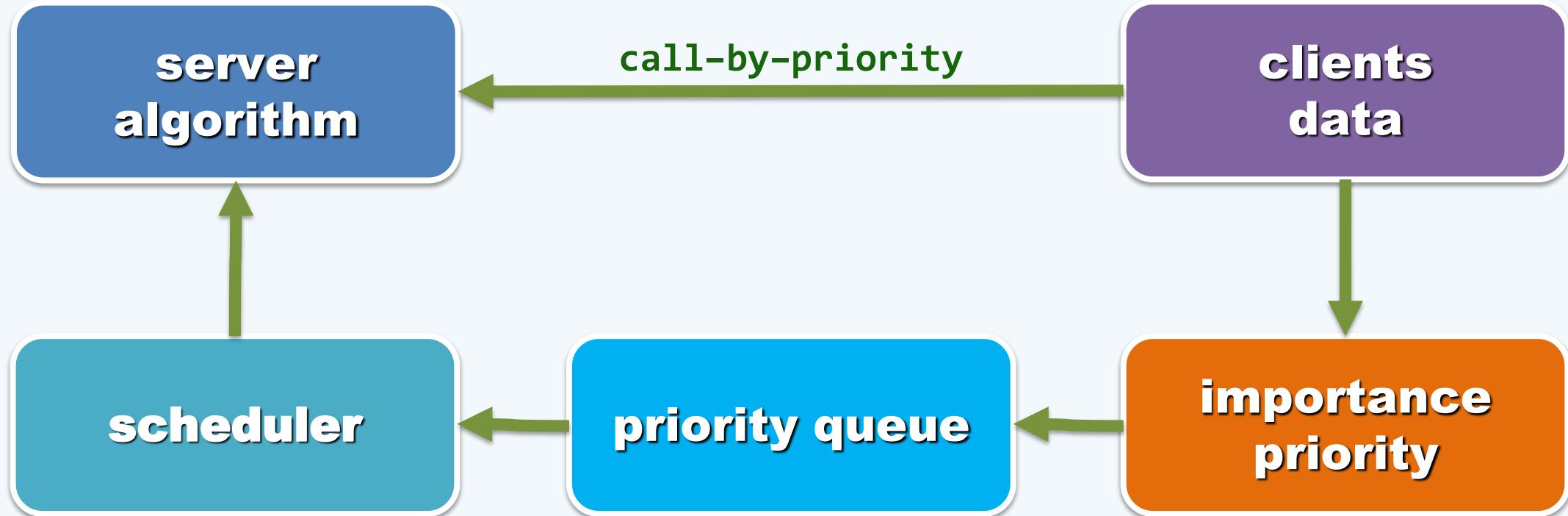
...

❖ 极值元素：须反复地、快速地定位

集合组成：可动态变化

元素优先级：可动态变化

问题模式



优先级队列

- ❖ `template <typename T> struct PQ { //priority queue`
 - `virtual void insert(T) = 0; //按照优先级次序插入词条`
 - `virtual T getMax() = 0; //取出优先级最高的词条`
 - `virtual T delMax() = 0; //删除优先级最高的词条`
- }; //与其说PQ是数据结构，不如说是ADT；其不同的实现方式，效率及适用场合也各不相同
- ❖ Stack和Queue，都是PQ的特例——优先级完全取决于元素的插入次序
- ❖ Steap和Queap，也是PQ的特例——插入和删除的位置受限

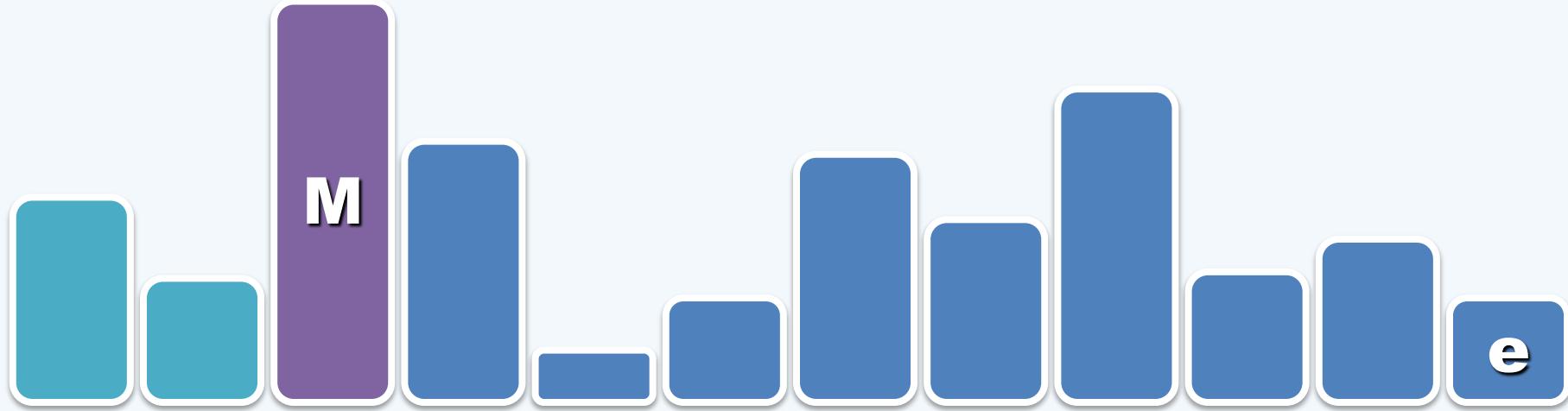
10. 优先级队列

概述

基本实现

邓俊辉

deng@tsinghua.edu.cn

Vector**getMax()****delMax()****insert()****traverse()** $\Theta(n)$ **remove(traverse())** $\Theta(n) + \Theta(n) = \Theta(n)$ **insertAsLast(e)** $\Theta(1)$

Sorted Vector



getMax()

[n - 1]

$\mathcal{O}(1)$

delMax()

remove(n - 1)

$\mathcal{O}(1)$

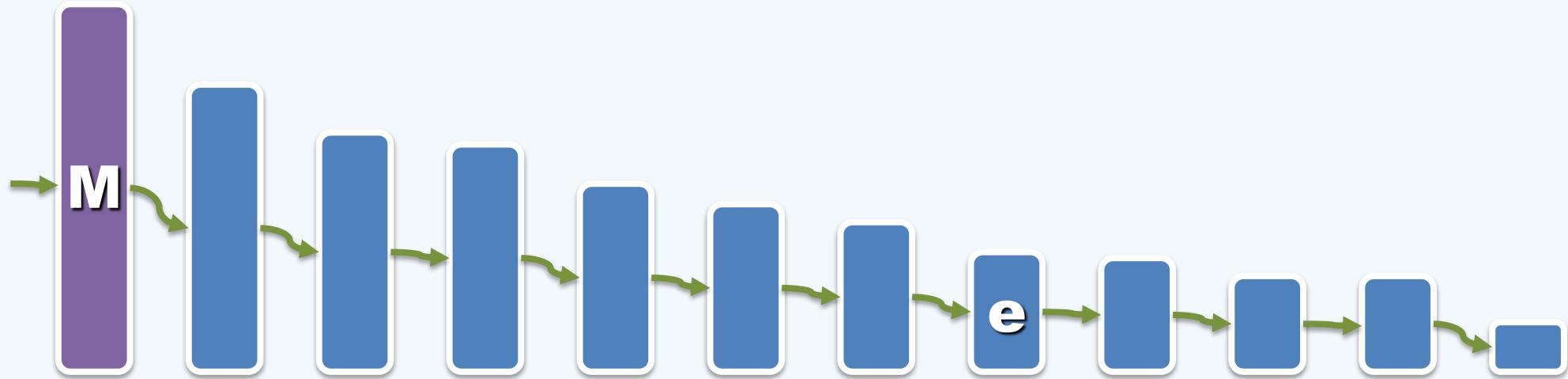
insert()

insert(1 + search(e), e)

$\mathcal{O}(\log n) + \mathcal{O}(n) = \mathcal{O}(n)$

List**getMax()****delMax()****insert()****traverse()** $\Theta(n)$ **remove(traverse())** $\Theta(n) + \Theta(1) = \Theta(n)$ **insertAsFirst(e)** $\Theta(1)$

Sorted List



getMax()

first()

$\mathcal{O}(1)$

delMax()

remove(first())

$\mathcal{O}(1)$

insert()

insertA(search(e), e)

$\mathcal{O}(n) + \mathcal{O}(1) = \mathcal{O}(n)$

❖ AVL、Splay、Red-black：三个接口均只需 $\mathcal{O}(\log n)$ 时间

但是，BBST的功能远远超出了PQ的需求...

$$\text{❖ PQ} = \boxed{1 \times \text{insert}()} + \boxed{0.5 \times \text{search}()} + \boxed{0.5 \times \text{remove}()} = \frac{2}{3} \times \text{BBST}$$

❖ 若只需查找极值元，则不必维护所有元素之间的全序关系，偏序足矣

❖ 因此有理由相信，存在某种更为简单、维护成本更低的实现方式

使得各功能接口 时间复杂度依然为 $\mathcal{O}(\log n)$ ，而且

实际效率更高

❖ 当然，就最坏情况而言，这类实现方式已属最优——为什么？

统一测试

```
❖ template <typename PQ, typename T> void testHeap( int n ) {  
    T* A = new T[ 2 * n / 3 ]; //创建容量为2n/3的数组，并  
    for ( int i = 0; i < 2 * n / 3; i++ ) A[i] = dice( (T) 3 * n ); //随机化  
    PQ heap( A + n / 6, n / 3 ); delete [] A; //Robert Floyd  
    while ( heap.size() < n ) //随机测试  
        if ( dice( 100 ) < 70 ) heap.insert( dice( (T) 3 * n ) ); //70%概率插入  
        else if ( ! heap.empty() ) heap.delMax(); //30%概率删除  
    while ( ! heap.empty() ) heap.delMax(); //清空  
}
```

10. 优先级队列

完全二叉堆

结构

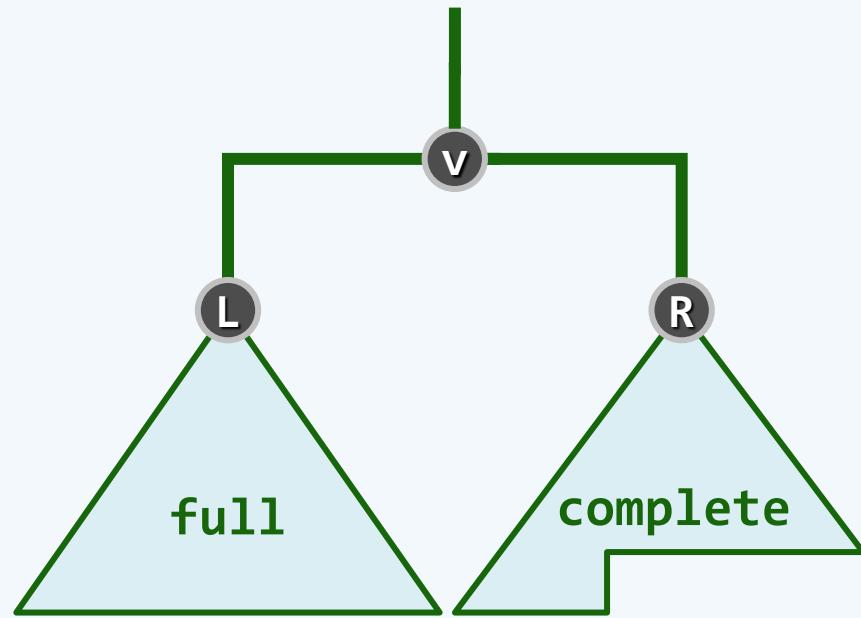
逊问曰：“何人将乱石作堆？
如何乱石堆中有杀气冲起？”

邓俊辉

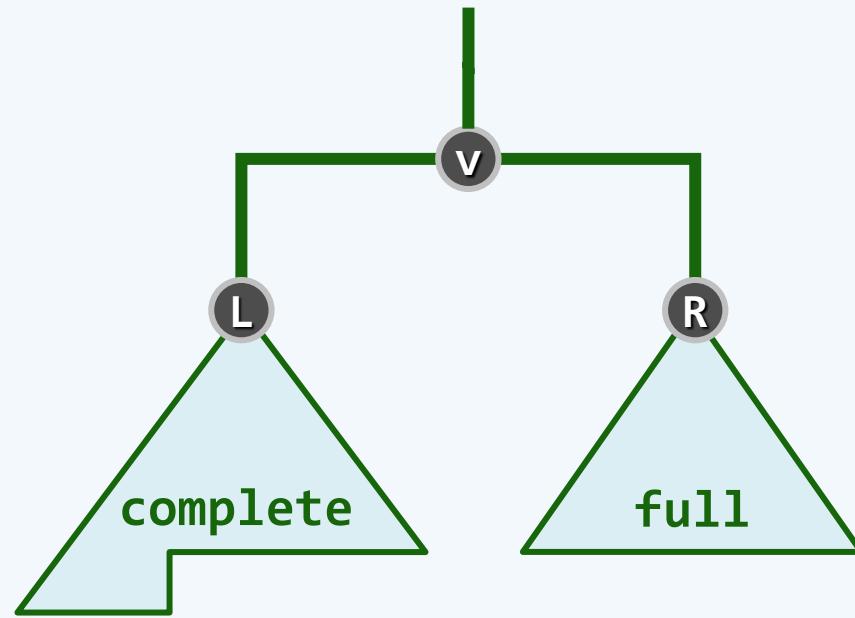
deng@tsinghua.edu.cn

完全二叉树

❖ Complete Binary Tree : 平衡因子处处非负的AVL，而且...



若 $bf(v) = 0$, 则 $lc(v)$ 满



若 $bf(v) = 1$, 则 $rc(v)$ 满

结构性

❖ 逻辑上，等同于完全二叉树

物理上，直接借助向量实现

❖ 逻辑节点与物理元素

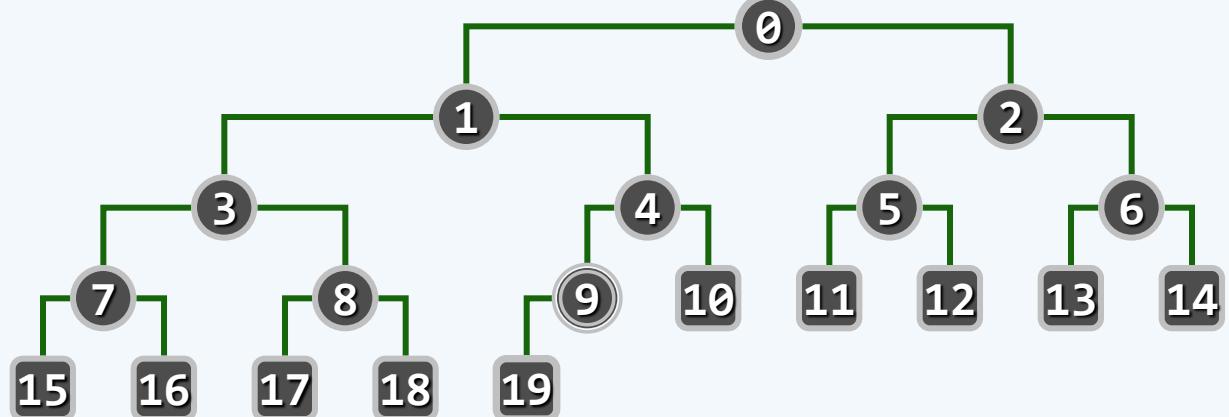
依层次遍历次序彼此对应

```
#define Parent( i ) ( ( i - 1 ) >> 1 )
```

```
#define LChild( i ) ( 1 + ( ( i ) << 1 ) ) //奇数
```

```
#define RChild( i ) ( ( 1 + ( i ) ) << 1 ) //偶数
```

❖ 共n个节点时，内部节点的最大秩 = $\lfloor (n - 2)/2 \rfloor = \lceil (n - 3)/2 \rceil$ //比如，这里的[9]



PQ_CmplHeap = PQ + Vector

```

❖ template <typename T> class PQ_CmplHeap : public PQ<T>, public Vector<T> {

protected: Rank percolateDown( Rank n, Rank i ); //下滤
            Rank percolateUp( Rank i ); //上滤
            void heapify( Rank n ); //Floyd建堆算法

public:    PQ_CmplHeap( T* A, Rank n ) //批量构造
            { copyFrom( A, 0, n ); heapify( n ); }

            void insert( T ); //按照比较器确定的优先级次序，插入词条
            T getMax(); //读取优先级最高的词条
            T delMax(); //删除优先级最高的词条
};

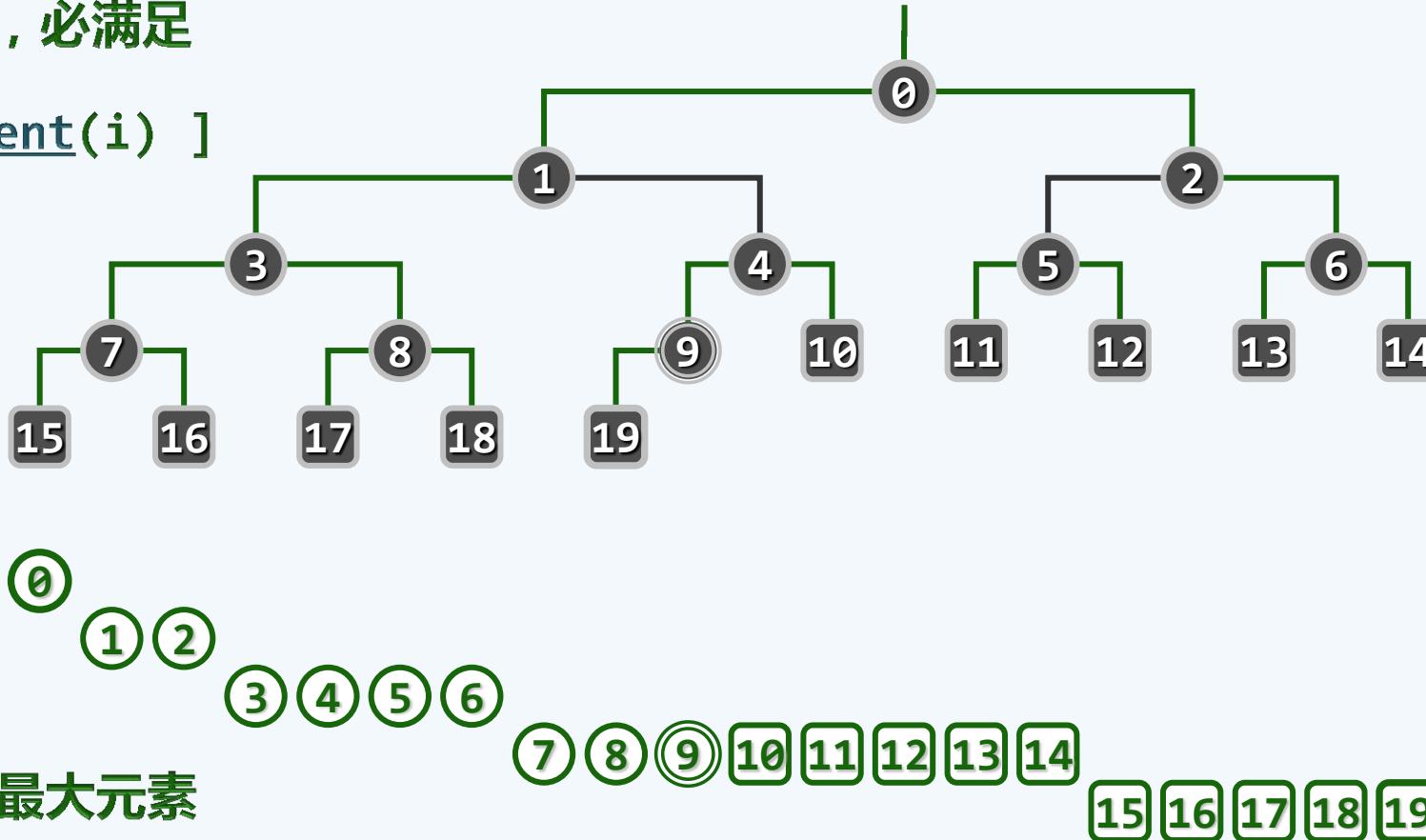
}

```

堆序性

❖ 数值上，只要 $\theta < i$ ，必满足

$$H[i] \leq H[\text{Parent}(i)]$$



❖ 故 $H[0]$ 即是全局最大元素

```
template <typename T> T
```

```
PQ_CmplHeap<T>::getMax() { return _elem[0]; }
```

10. 优先级队列

完全二叉堆

插入

时迁看见土地庙后一株大柏树，便把两只腿夹定，
一节节爬将上去树头顶，骑马儿坐在枝柯上。

邓俊辉

deng@tsinghua.edu.cn

算法

❖ 为插入词条e，只需将e作为**末元素**接入向量

//结构性自然保持

//若堆序性也亦未破坏，则完成

❖ 否则 //只能是e与其父节点违反堆序性

e与其父节点换位 //若堆序性因此恢复，则完成

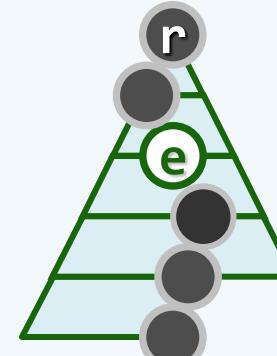
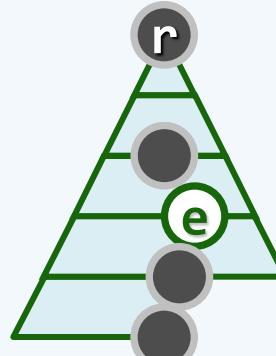
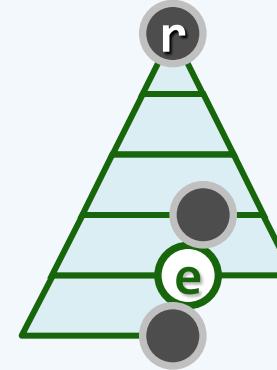
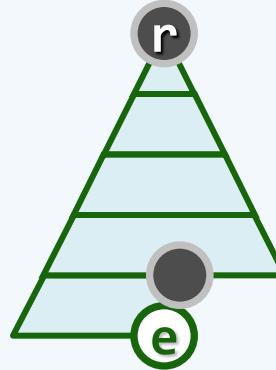
❖ 否则 //依然只可能是e与其（新的）父节点...

e再与父节点换位

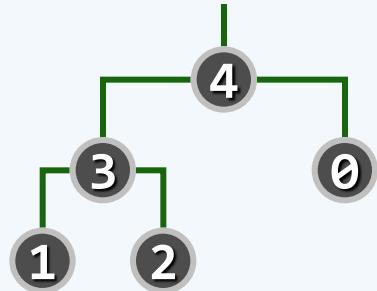
❖ 不断重复...直到

e与其父亲满足堆序性，或者

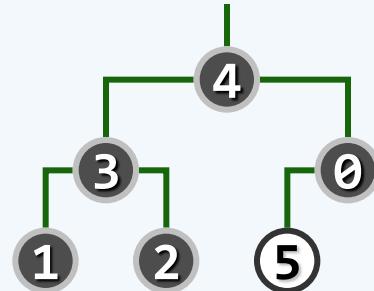
e到达堆顶（没有父亲）



实例

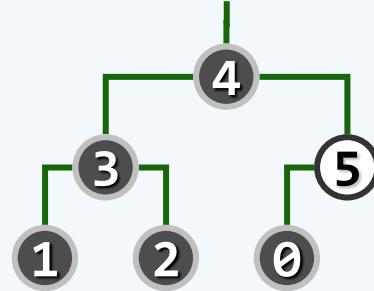


4 3 0 1 2



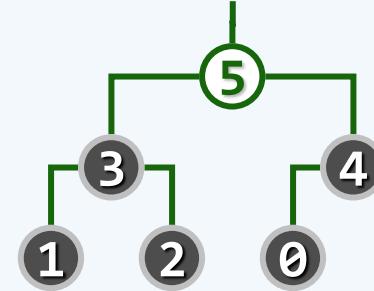
4 3 0 1 2 5

swap



4 3 5 1 2 0

swap



5 3 4 1 2 0

实现

```

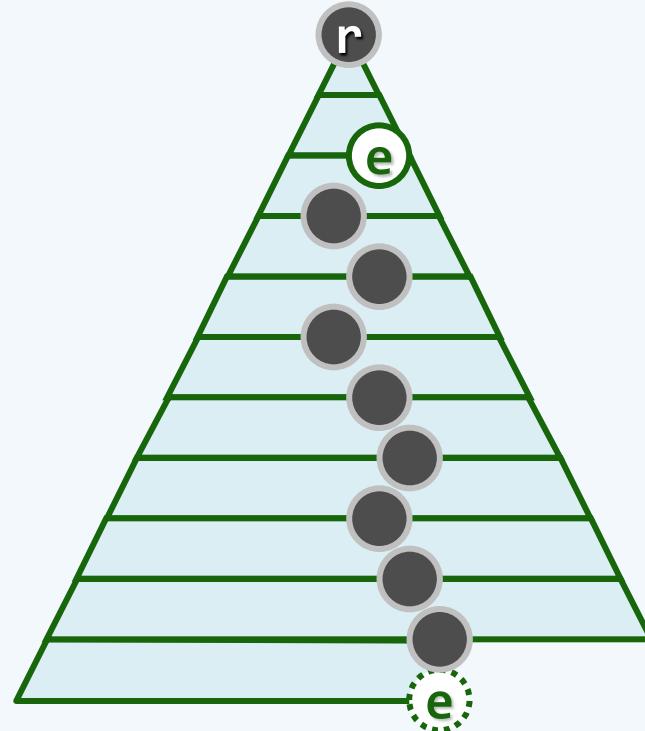
❖ template <typename T> void PQ_CmplHeap<T>::insert( T e ) //插入
{ Vector<T>::insert( e ); percolateUp( _size - 1 ); }

❖ template <typename T> //对第i个词条实施上滤，i < _size
Rank PQ_CmplHeap<T>::percolateUp( Rank i ) {
    while ( ParentValid( i ) ) { //只要i有父亲（尚未抵达堆顶），则
        Rank j = Parent( i ); //将i之父记作j
        if ( lt( _elem[i], _elem[j] ) ) break; //一旦父子不再逆序，上滤旋即完成
        swap( _elem[i], _elem[j] ); i = j; //否则，交换父子位置，并上升一层
    } //while
    return i; //返回上滤最终抵达的位置
}

```

效率

- ❖ e与父亲的交换，每次只需 $\Theta(1)$ 时间，且每经过一次交换，e都会上升一层
- ❖ 在插入新节点e的整个过程中，只有e的祖先们，才有可能需要与之交换
- ❖ 这里的堆以完全树实现，必平衡，故e的祖先至多 $\Theta(\log n)$ 个
- ❖ 结论：通过上滤，可在 $\Theta(\log n)$ 时间内插入一个新节点，并整体地重新调整为堆



10. 优先级队列

完全二叉堆

删除

I have scaled the peak and
found no shelter in
fame's bleak and barren height.

邓俊辉

deng@tsinghua.edu.cn

算法

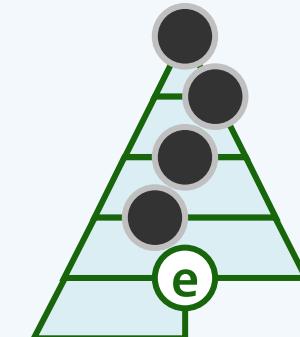
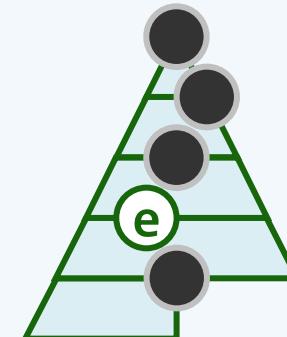
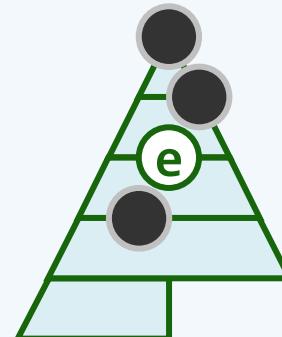
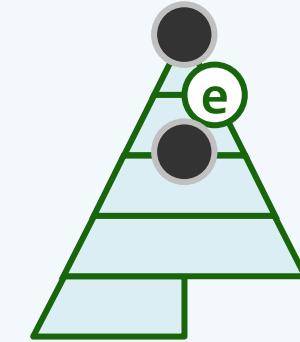
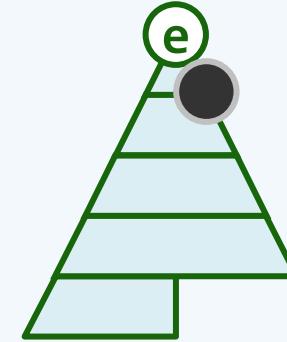
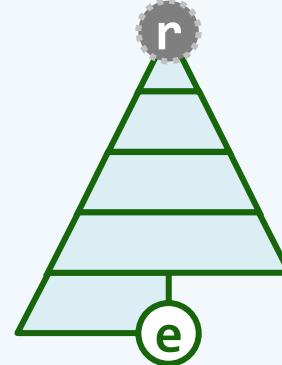
❖ 最大元素始终在堆顶，故为删除之，只需...

❖ 摘除向量首元素，代之以末元素e
//结构性保持；若堆序性依然保持则完成

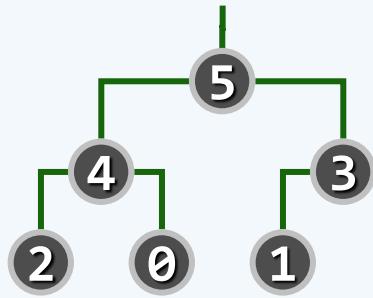
❖ 否则 //即e与孩子们违背堆序性
e与孩子中的大者换位
//若堆序性因此恢复，则完成

❖ 否则 //即e与其（新的）孩子们...
e再次与孩子中的大者换位

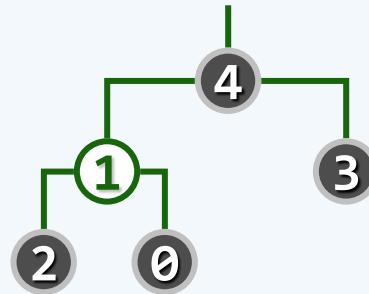
❖ 不断重复...直到
e满足堆序性，或者已是叶子



实例

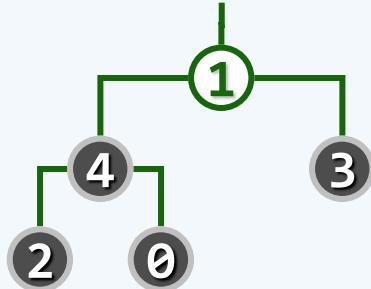


5 4 3 2 0 1



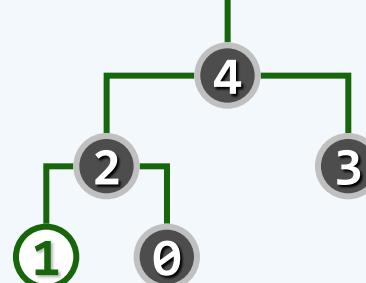
4 1 3 2 0

swap



1 4 3 2 0

Swap



4 2 3 1 0

实现

```

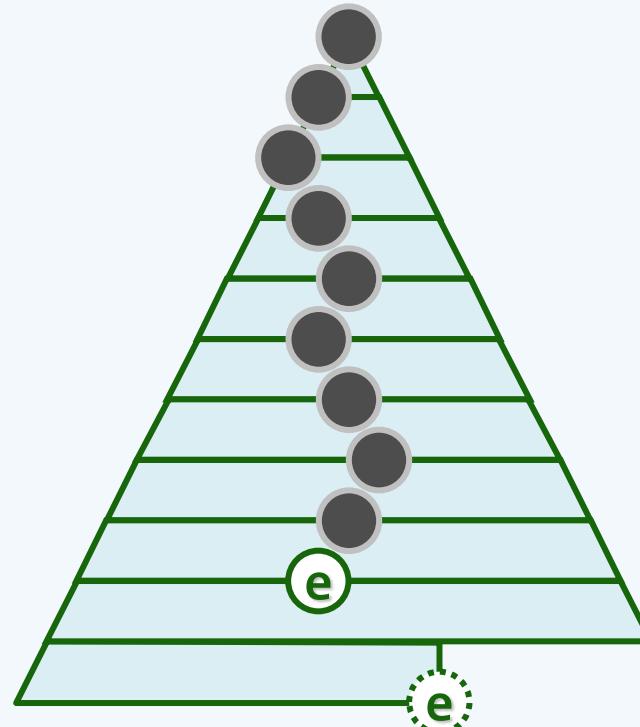
❖ template <typename T> T PQ_CmplHeap<T>::delMax() { //删除
    T maxElem = _elem[0]; _elem[0] = _elem[ --_size ]; //摘除堆顶，代之以末词条
    percolateDown( _size, 0 ); //对新堆顶实施下滤
    return maxElem; //返回此前备份的最大词条
}

❖ template <typename T> //对前n个词条中的第i个实施下滤，i < n
Rank PQ_CmplHeap<T>::percolateDown( Rank n, Rank i ) {
    Rank j; //i及其（至多两个）孩子中，堪为父者
    while ( i != ( j = ProperParent( _elem, n, i ) ) ) //只要i非j，则
        { swap( _elem[i], _elem[j] ); i = j; } //换位，并继续考察i
    return i; //返回下滤抵达的位置（亦i亦j）
}

```

效率

- ❖ 在下滤的过程中
每经过一次交换
 e 的高度都降低一层
 - ❖ 故此，在每层至多需要一次交换
 - ❖ 再次地，由于堆是完全树，故其高度为 $\lceil \log n \rceil$
 - ❖ 结论：通过下滤，可在 $\Theta(\log n)$ 时间内
删除堆顶节点，并
整体重新调整为堆



10. 优先级队列

完全二叉堆

批量建堆

邓俊辉

deng@tsinghua.edu.cn

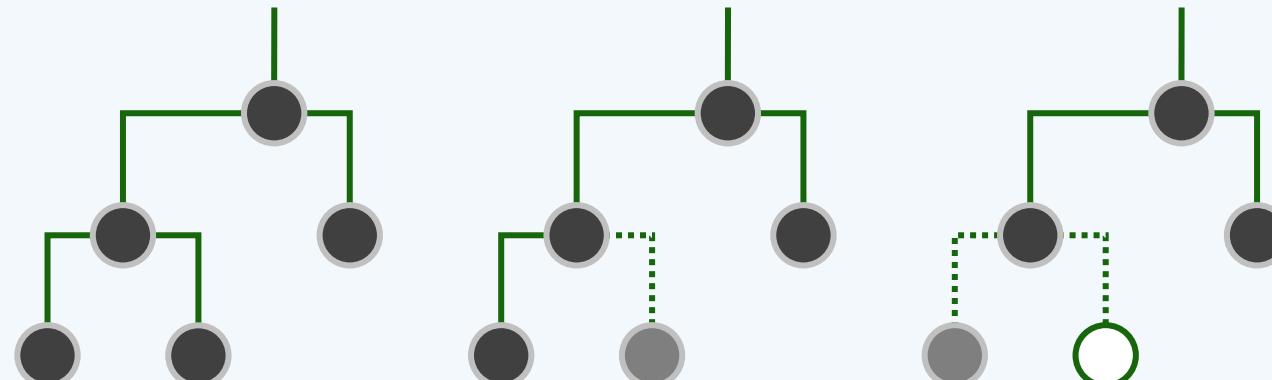
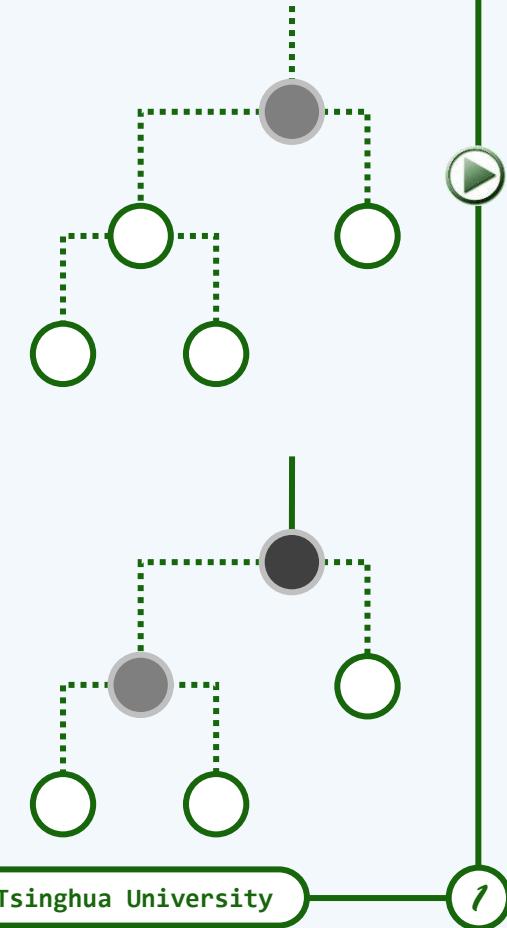
自上而下的上滤

```

❖ PQ_CmplHeap( T* A, Rank n ) { copyFrom( A, 0, n ); heapify( n ); } //如何实现?

❖ template <typename T> void PQ_CmplHeap<T>::heapify ( Rank n ) { //蛮力
    for ( int i = 1; i < n; i++ ) //按照层次遍历次序逐一
        percolateUp ( i ); //经上滤插入各节点
}

```



效率

❖ 最坏情况下

每个节点都需上滤至根

所需成本线性正比于其深度

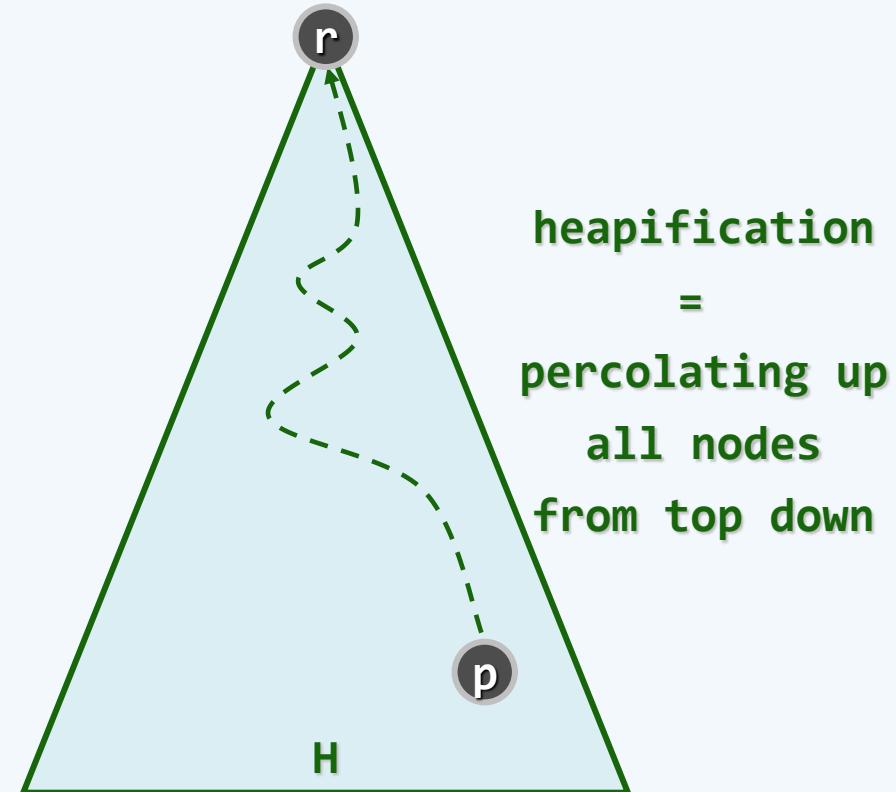
❖ 即便只考虑底层

$n/2$ 个叶节点，深度均为 $\Theta(\log n)$

累计耗时 $\Theta(n \log n)$

❖ 这样长的时间，本足以全排序！

应该，能够更快的...



自下而上的下滤

❖ 任意给定堆 H_0 和 H_1 ，以及节点 p

❖ 为得到堆 $H_0 \cup \{p\} \cup H_1$ ，只需

将 r_0 和 r_1 当作 p 的孩子，对 p 下滤

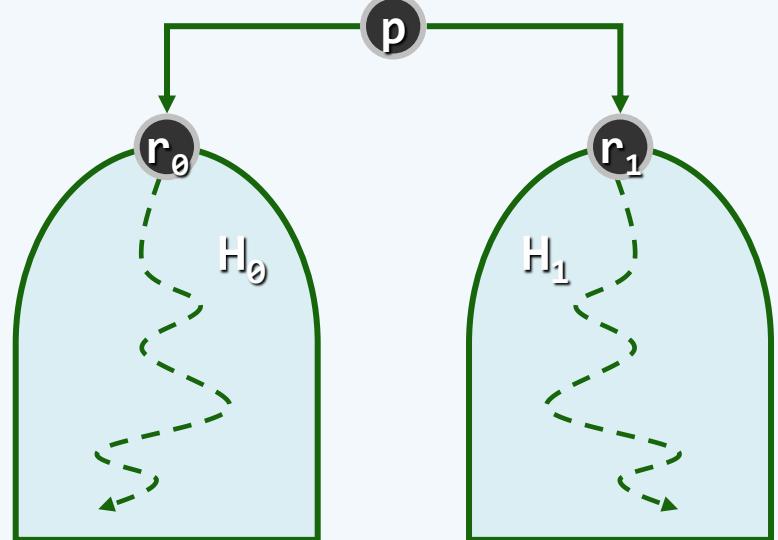
❖ template <typename T>

```
void PQ_CmplHeap<T>::heapify( Rank n ) { //Robert Floyd, 1964
```

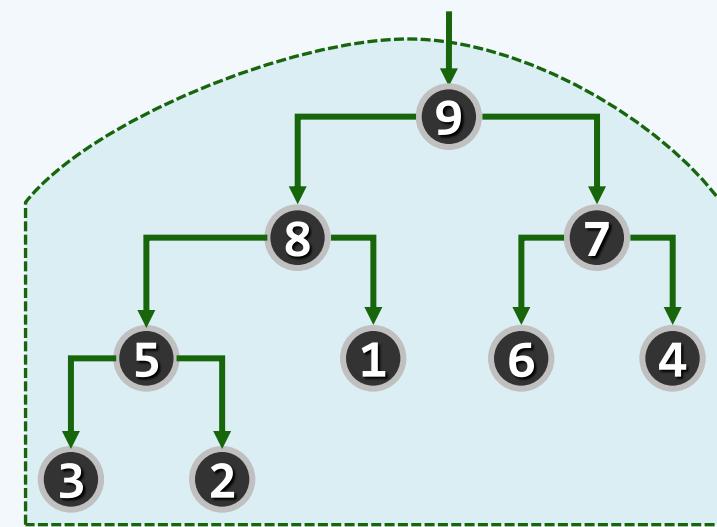
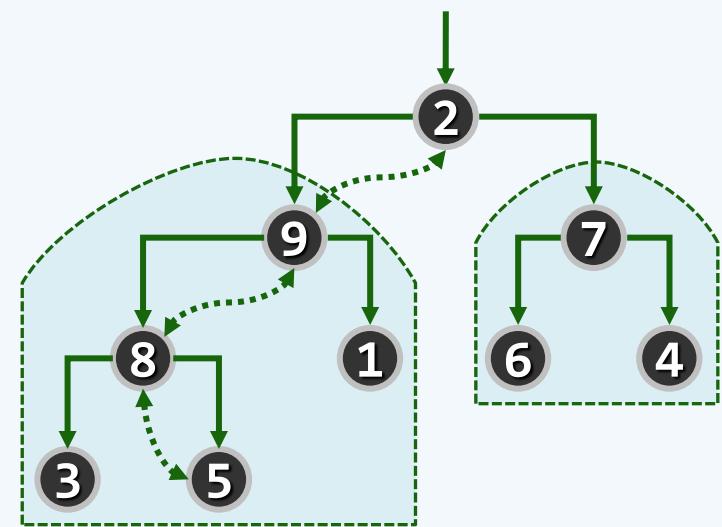
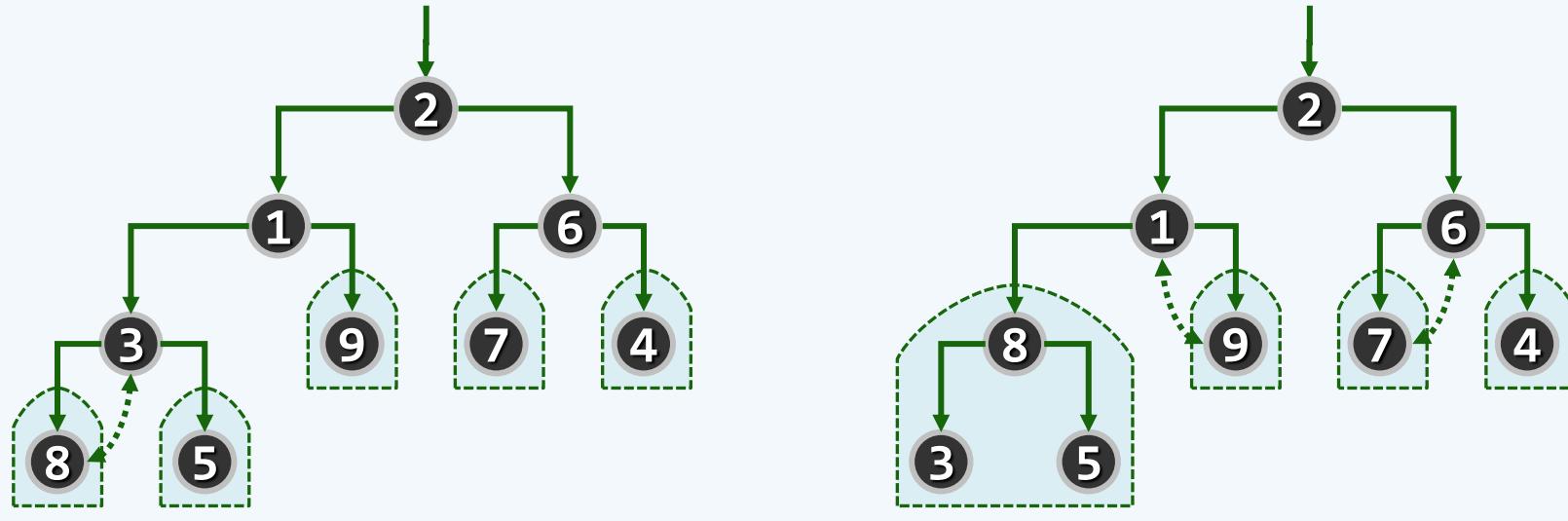
```
for ( int i = LastInternal(n); i >= 0; i-- ) //自下而上，依次
```

```
percolateDown( n, i ); //下滤各内部节点
```

```
} //可理解为子堆的逐层合并 —— 由以上性质，堆序性最终必然在全局恢复
```



实例



效率

❖ 每个内部节点所需的调整时间，正比于其高度而非深度

❖ 不失一般性，考查满树： $n = 2^{d+1} - 1$

❖ $S(n) = \text{所有节点的} [\text{高度}] \text{总和}$

$$= \sum_{i=0..d} ((d - i) \times 2^i)$$

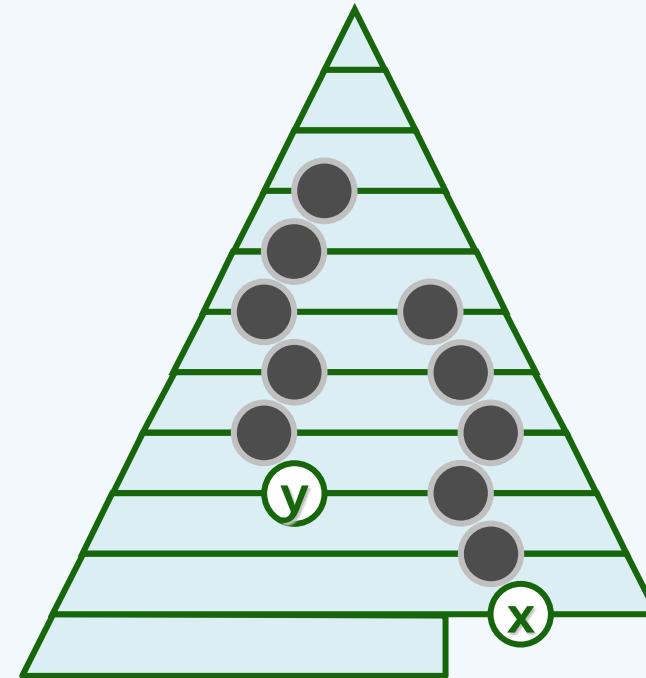
$$= d \times \sum_{i=0..d} 2^i - T(n)$$

$$= d \times (2^{d+1} - 1) - [(d - 1) \times 2^{d+1} + 2]$$

$$= 2^{d+1} - (d + 2)$$

$$= n - \log_2(n + 1)$$

$$= O(n)$$



课后

- ❖ `insert()`：最坏情况下效率为 $\Theta(n \log n)$ ，平均情况呢？
- ❖ `heapify()`：构造次序颠倒后，为什么复杂度会实质性地降低？
这一算法在哪些场合不适用？
- ❖ 扩充接口：
`decrease(i, delta) //任一元素 elem[i] 的数值减小delta`
`increase(i, delta) //任一元素 elem[i] 的数值增加delta`
`remove(i) //删除任一元素 elem[i]`
- ❖ 借助完全堆，在 $\Theta(n \log n)$ 时间内构造 Huffman 树
- ❖ 在大顶堆中，`delMin()` 操作能否也在 $\Theta(n \log n)$ 时间内完成？
难道，为此需要同时维护一个小顶堆？

10. 优先级队列

堆排序

邓俊辉

deng@tsinghua.edu.cn

选取

❖ 在 `selectionSort()` 中...

❖ 将 `U` 替换为 `H`

❖ J. Williams, 1964

初始化 : `heapify()`, $\mathcal{O}(n)$, 建堆

迭代 : `delMax()`, $\mathcal{O}(\log n)$, 取出堆顶并调整复原

不变性 : $H \leq S$

❖ 等效于常规选择排序, 正确无疑

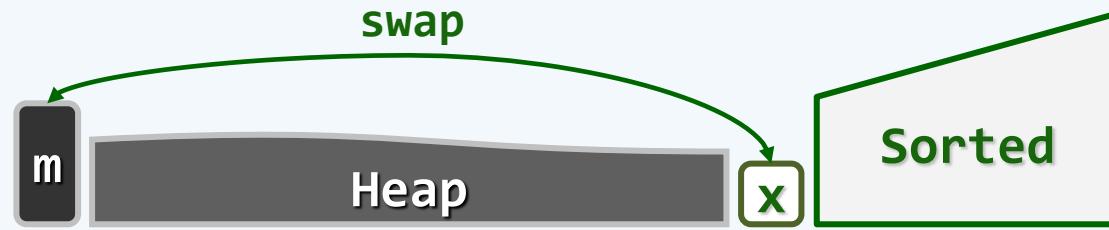
❖ $\mathcal{O}(n) + n \times \mathcal{O}(\log n) = \mathcal{O}(n \log n)$



就地

❖ 在物理上

完全二叉堆 即是 向量



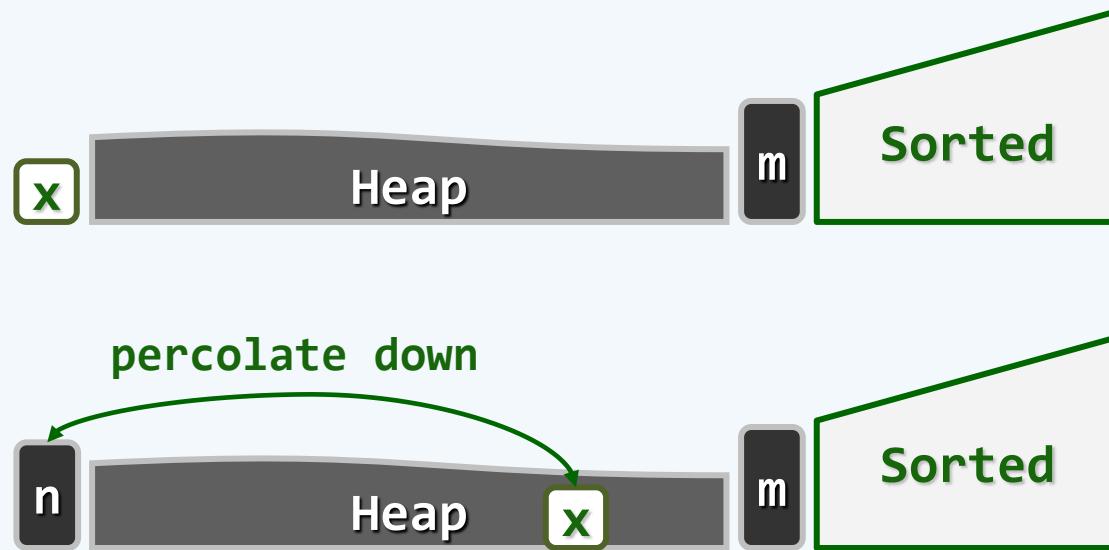
❖ 既然此前有：

$m = H[0]$

$x = S[-1]$

不妨随即就：

$\text{swap}(m, x) = H.\text{insert}(x) + S.\text{insert}(m)$



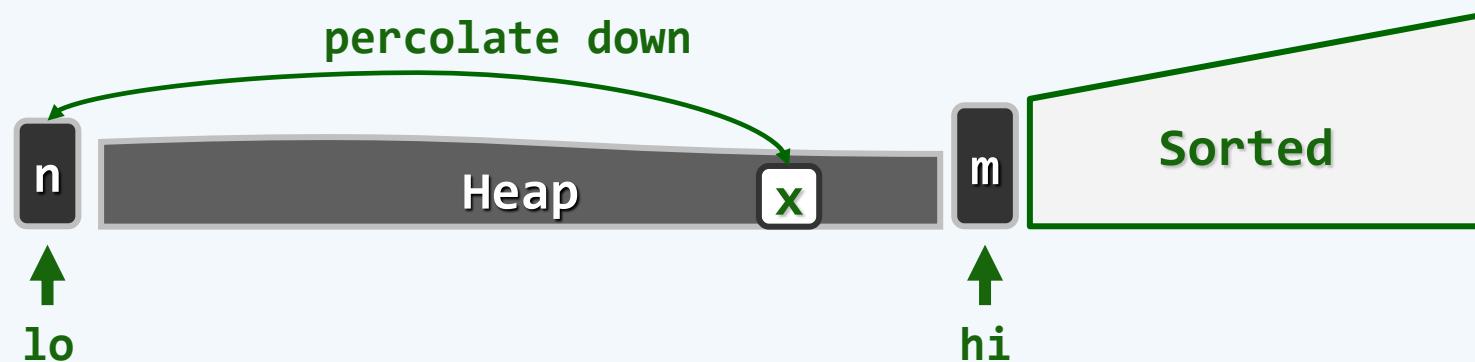
实现

❖ template <typename T> //对向量区间[lo, hi)做就地堆排序

```
void Vector<T>::heapSort( Rank lo, Rank hi ) {
    PQ_CmplHeap<T> H( _elem + lo , hi - lo ); //待排序区间建堆，O(n)

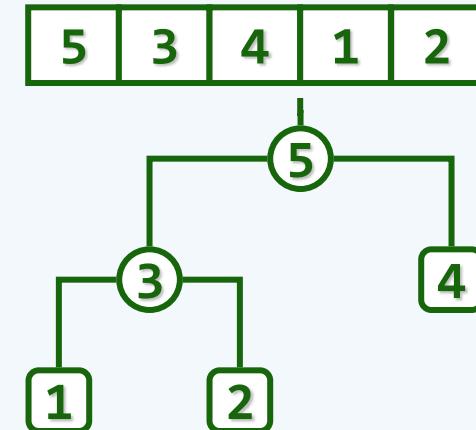
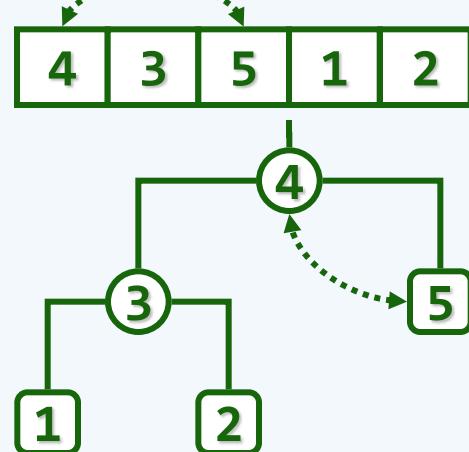
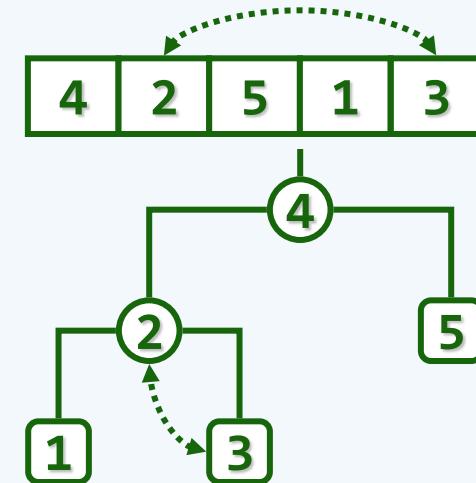
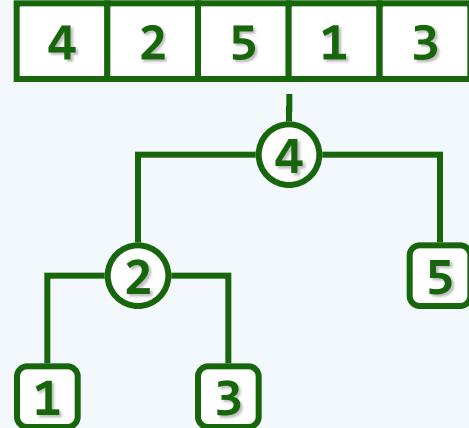
    while ( ! H.empty() ) //反复地摘除最大元并归入已排序的后缀，直至堆空

        _elem[ --hi ] = H.delMax(); //等效于堆顶与末元素对换后下滤
}
```

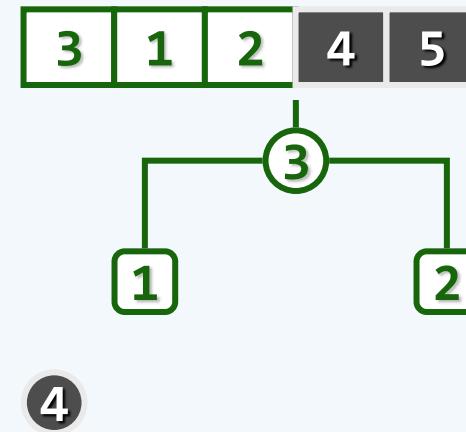
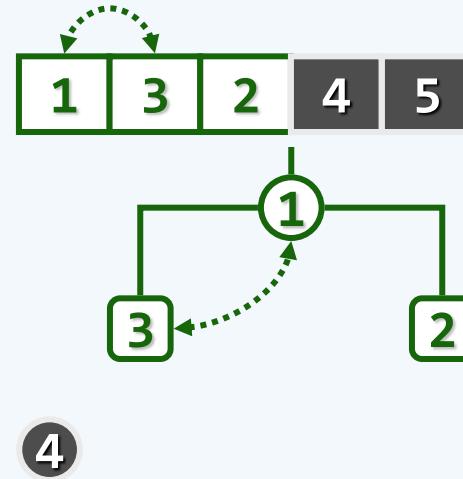
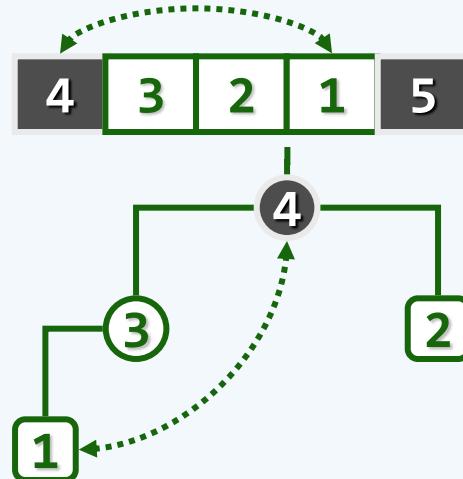
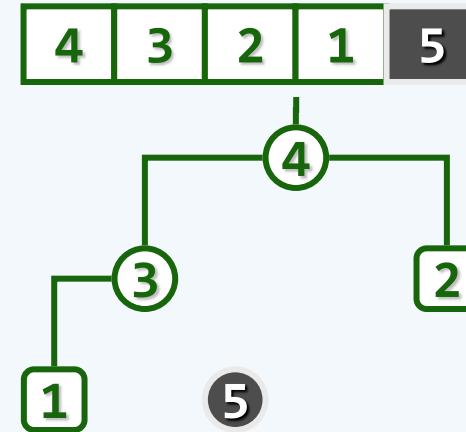
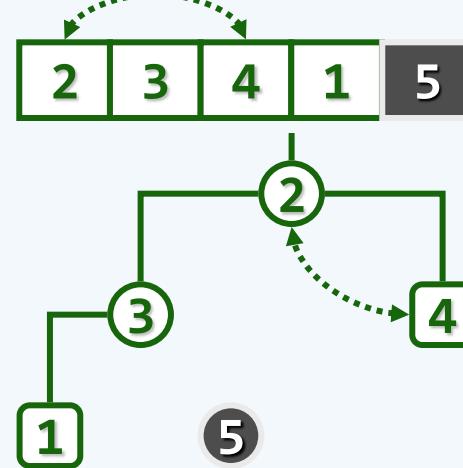
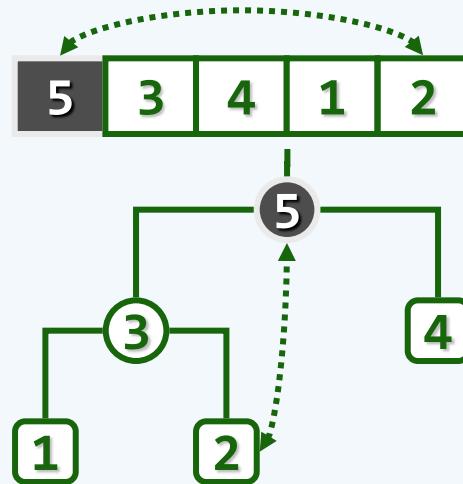


实例：建堆

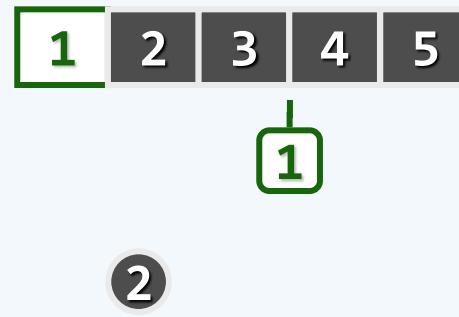
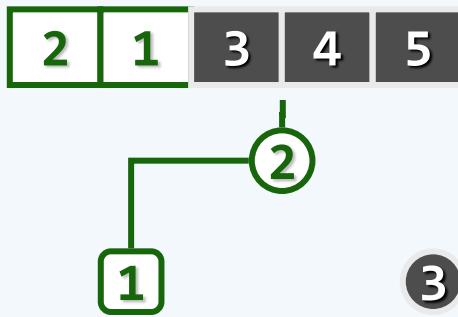
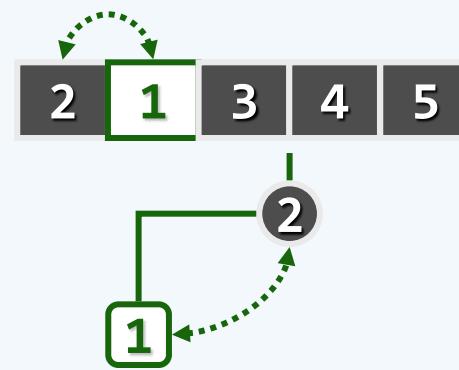
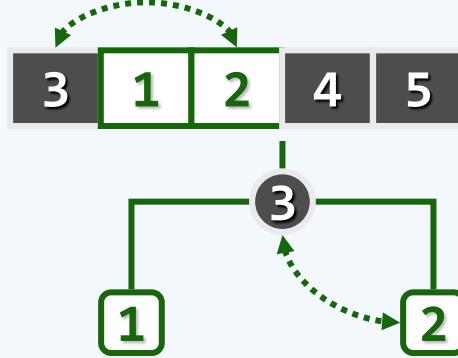
4	2	5	1	3
---	---	---	---	---



实例：选取 + 调整



实例：选取 + 调整



评价

❖ 易于理解，便于实现 //完全基于二叉堆结构及其操作接口

快速高效 //尤其适用于大规模数据

可就地运转

不需全排序即可找出前k个词条 // $\Theta(k \log n)$ 的selection算法

❖ 不稳定 //为什么？可否克服？

❖ 权衡：采用就地策略，是否值得？

固然可以节省一定的空间

但对换操作因此须涉及两个完整的词条，操作的单位成本增加

10. 优先级队列

锦标赛排序

锦标赛树

老妖道：“怎么叫做分瓣梅花计？”

小妖道：“如今把洞中大小群妖，点将起来，千中选百，百中选十，十中只选三个...”

邓俊辉

deng@tsinghua.edu.cn

锦标赛树

❖ Tournament Tree : 完全二叉树

叶节点：待排序元素（选手）

内部节点：孩子中的胜者

胜负判定原则：小者为胜

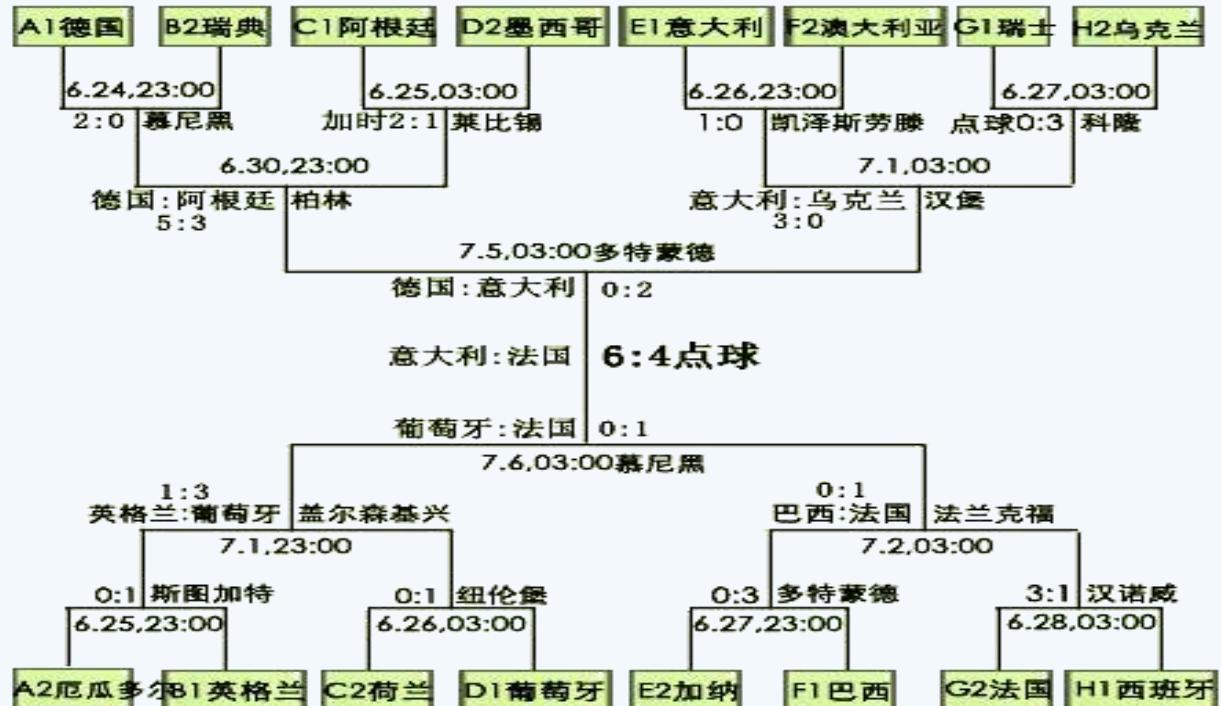
❖ create() // $O(n)$

remove() // $O(\log n)$

insert() // $O(\log n)$

❖ 树根总是全局冠军——类似于最小堆

有重复：在任一子树中，从根通往优胜者的沿途，所有节点都是优胜者



❖ Tournamentsort()

CREATE a tournament tree for the input list

while there are active leaves

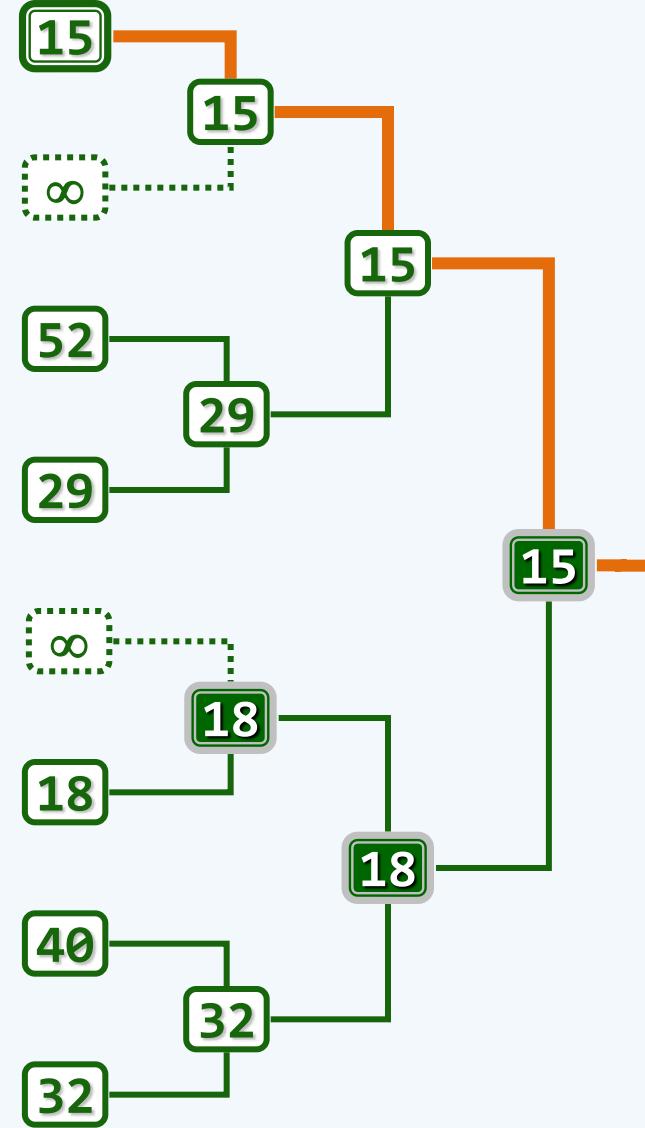
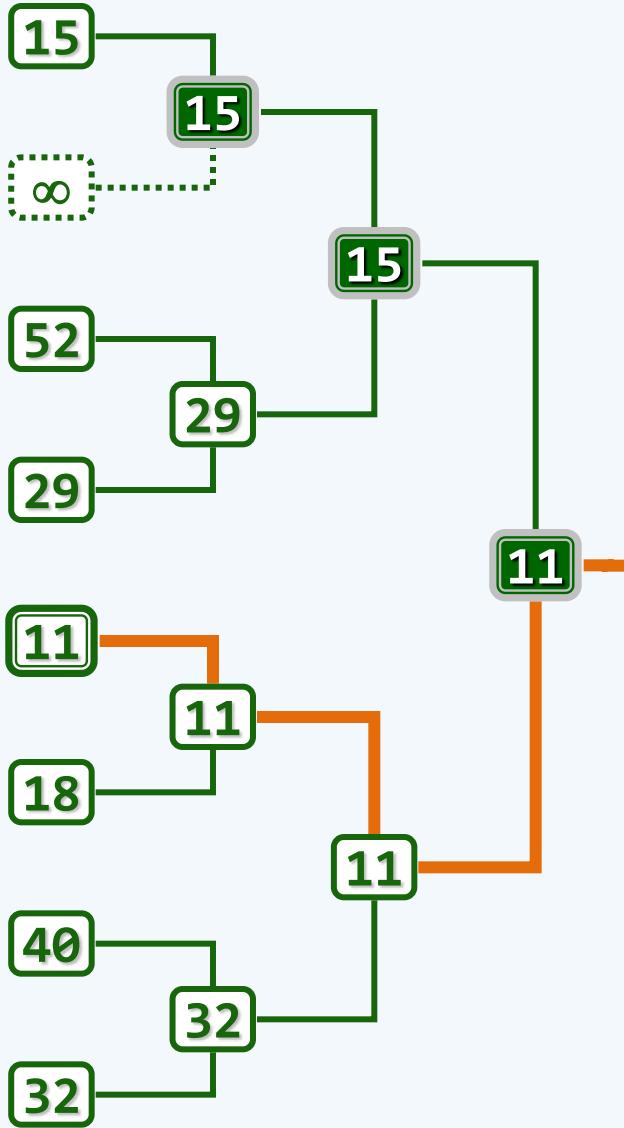
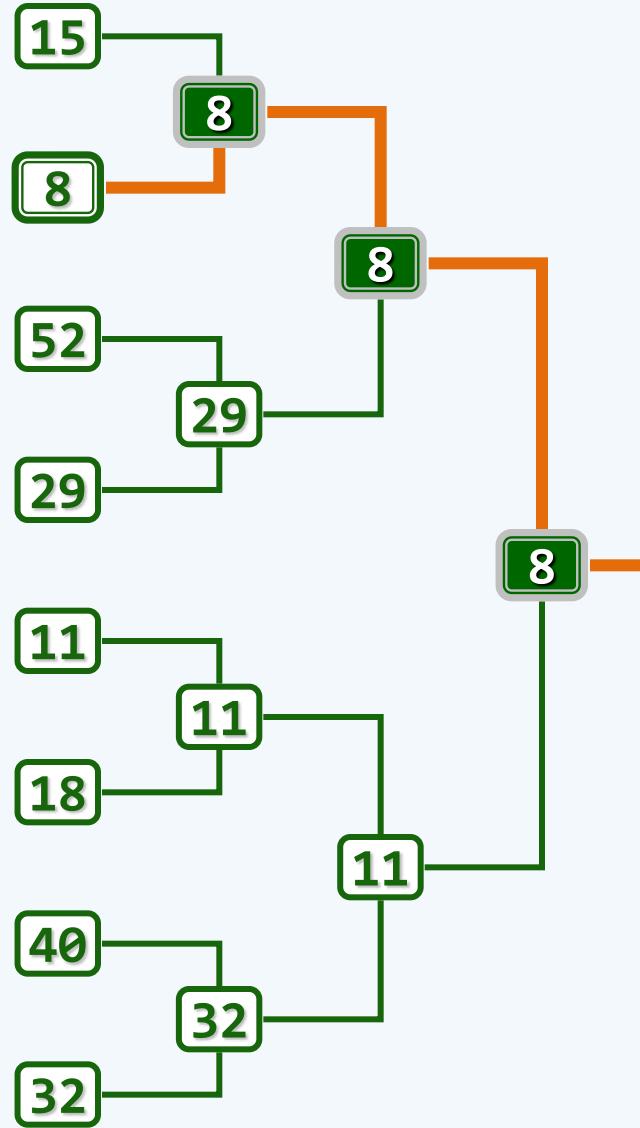
REMOVE the root

RETRACE the root down to its leaf

DEACTIVATE the leaf

REPLAY along the path back to the root

实例



效率

❖ 空间： $\mathcal{O}(\text{节点数}) = \mathcal{O}(\text{叶节点数}) = \mathcal{O}(n)$

❖ 构造：仅需 $\mathcal{O}(n)$ 时间

❖ 更新：每次都须全体重赛 replay ?

唯上一优胜者的祖先，才有必要参加！

❖ 为此 只需从其所在叶节点出发，逐层上溯直到树根

如此 为确定各轮优胜者，总共所需时间仅 $\mathcal{O}(\log n)$

❖ 时间： $n\text{轮} \times \mathcal{O}(\log n) = \mathcal{O}(n \log n)$ ，达到下界

选取

❖ 从 n 个元素中找出最小的 k 个， $k \ll n$ // 回忆第 01 章的 `max2()`，即是 $k = 2 \ll n$

❖ 借助锦标赛树 初始： $\theta(n)$ // $n - 1$ 次比较

迭代 k 步： $\theta(k * \log n)$ // 每步 $\log n$ 次比较

与 `小顶堆` 旗鼓相当？

❖ 渐进意义上，的确如此

但就常系数而言，区别不小...

❖ `Floyd 算法` 中的 `percolateDown()`，在每一层需做 2 次比较，累计略少于 $2n$ 次

`delMax()` 中的 `percolateDown()`，亦是如此

10. 优先级队列

锦标赛排序

败者树

邓俊辉

deng@tsinghua.edu.cn

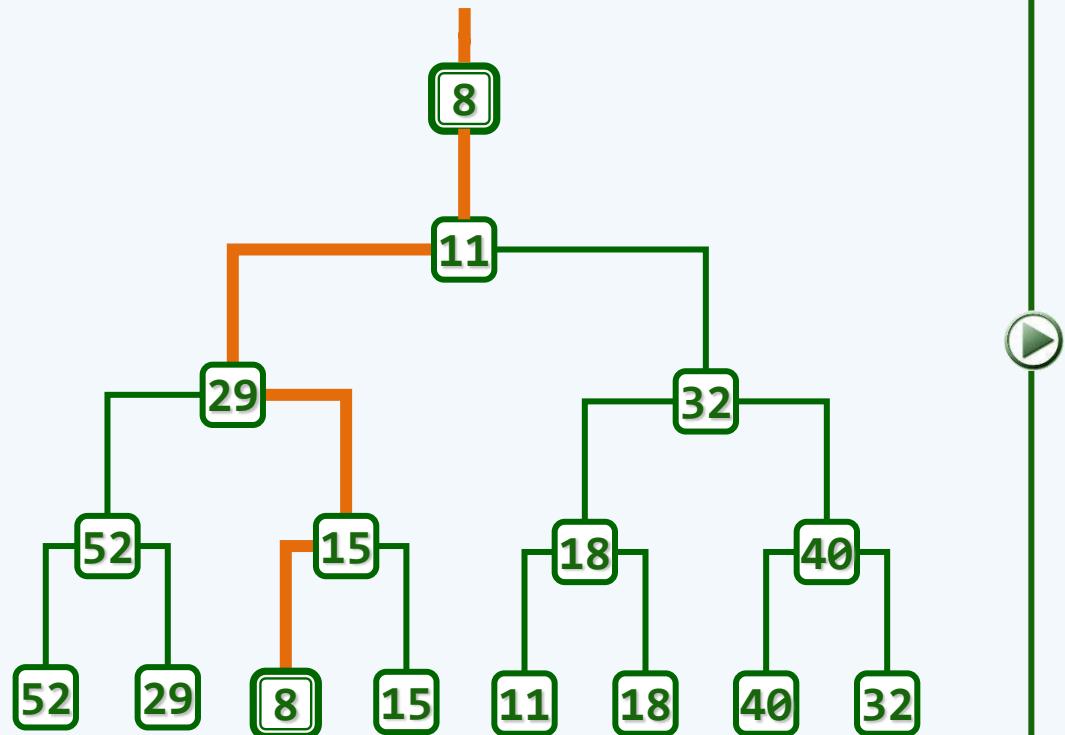
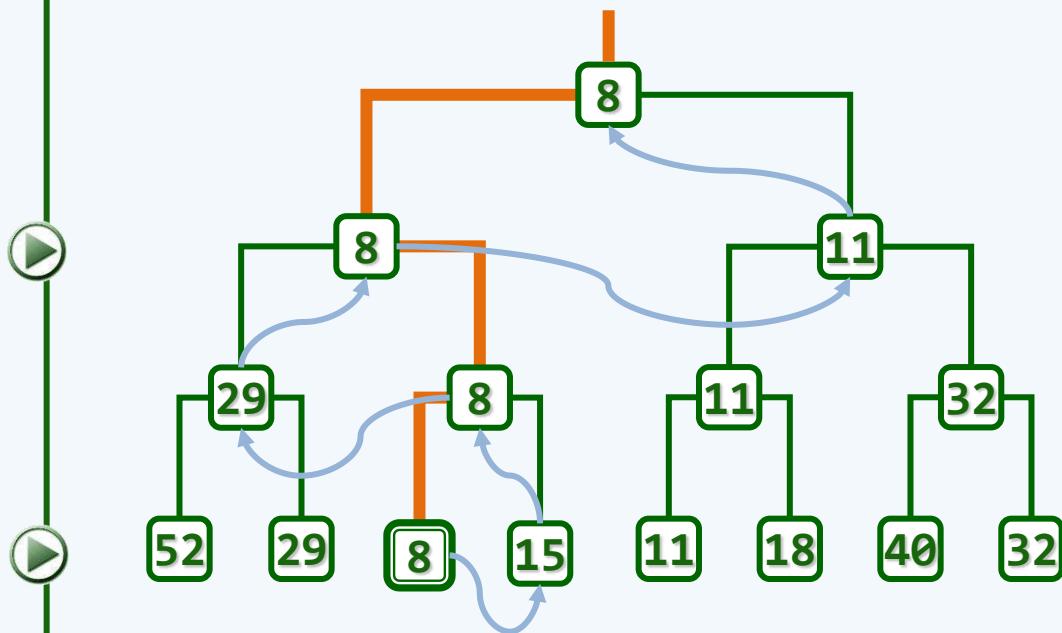
败者树

❖ 重赛过程中，除了沿途各节点，还需间隔地访问兄弟

//如何避免这类迂回？

❖ loser tree : 内部节点，记录对应比赛的败者

增设根的“父节点”，记录冠军



- ❖ 利用锦标赛树，可否实现**稳定的排序**？
- ❖ 利用锦标赛树，如何实现**多路归并**？

10. 优先级队列

多叉堆

邓俊辉

deng@tsinghua.edu.cn

优先级搜索

- ❖ 回顾图的优先级搜索以及统一框架：`g->pfs()`...
- ❖ 无论何种算法，差异仅在于所采用的优先级更新器 `prioUpdater()`
 - Prim算法：`g->pfs(0, PrimPU());`
 - Dijkstra算法：`g->pfs(0, DijkstraPU());`
- ❖ 每一节点引入遍历树后，都需要
 - `更新`树外顶点的优先级（数），并
 - `选出`新的优先级最高者
- ❖ 若采用邻接表，两类操作的累计时间，分别为 $\mathcal{O}(n + e)$ 和 $\mathcal{O}(n^2)$
- ❖ 能否 `更快` 呢？

优先级队列

❖ 自然地，PFS中的各顶点可组织为**优先级队列**形式

❖ 为此需要使用**PQ**接口

heapify():

由n个顶点创建初始PQ

总计 $\Theta(n)$

delMax():

取优先级最高（极短）跨边(u, w)

总计 $\Theta(\boxed{n} * \log n)$

increase():

更新所有关联顶点到u的距离，提高优先级

总计 $\Theta(\boxed{e} * \log n)$

❖ 总体运行时间 = $\Theta(\boxed{(n + e)} * \log n)$

对于**稀疏图**，处理效率很高；对于**稠密图**，反而不如常规实现的版本

❖ 有无更好的算法？如果PQ的接口效率能够更高的话...

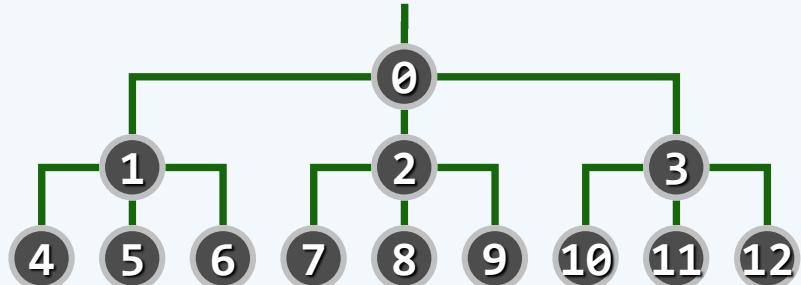
❖ 不太现实？异想天开？不妨先试试...

多叉堆

❖ `heapify()`: $\Theta(n)$ 不可能再快了 //直接写入，亦不过如此

`delMax()`: $\Theta(\log n)$ 实质就是`percolateDown()` //已是极限了

`increase()`: $\Theta(\log n)$ 实质就是`percolateUp()` //似乎仍有余地



多叉堆

❖ 若将二叉堆改成多叉堆 (d-heap)

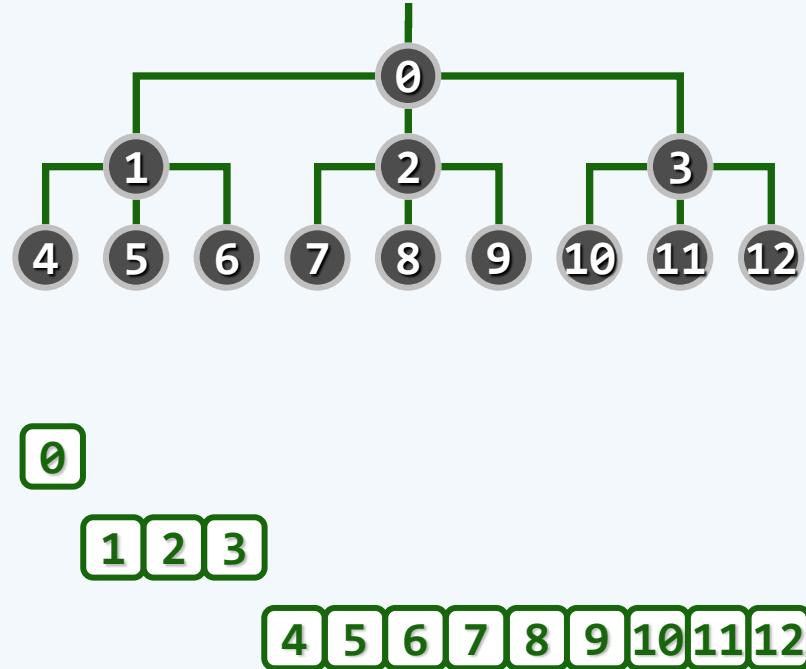
则堆高降至 $\Theta(\log_d n)$

❖ 上山容易下山难：

上滤成本可降至 $\log_d n$ ，但

下滤成本却增至

$$d * \log_d n > (d * \ln 2 / \ln d) * \log_2 n$$



❖ 对于稠密图，两类操作的次数相差悬殊 —— 故而利大于弊...

多叉堆

❖ 如此，PFS的运行时间将是： $n \cdot d \cdot \log_d n + e \cdot \log_d n = (n \cdot d + e) \cdot \log_d n$

❖ 两相权衡，大致取 $d = e/n + 2$ 时

总体性能达到最优的 $\mathcal{O}(e \cdot \log_{(e/n+2)} n)$

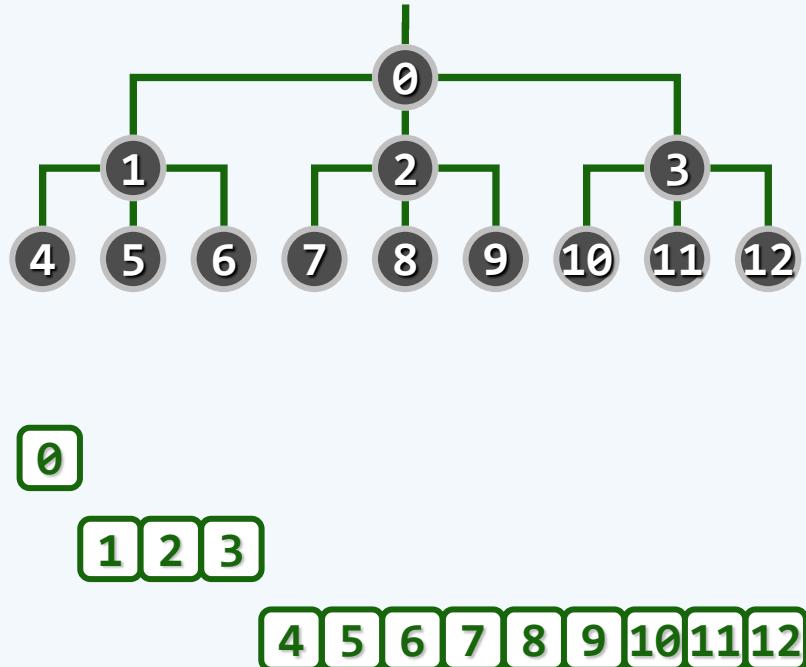
❖ 对于稀疏图保持高效：

$$e \cdot \log_{(e/n+2)} n \approx n \cdot \log_{(n/n+2)} n = \mathcal{O}(n \log n)$$

对于稠密图改进极大：

$$e \cdot \log_{(e/n+2)} n \approx n^2 \cdot \log_{(n^2/n+2)} n \approx n^2 = \mathcal{O}(e)$$

对于一般的图，会自适应地实现最优



多叉堆

实现方面，依然可以基于向量

$$\text{parent}(k) = \lfloor (k - 1) / d \rfloor$$

$$\text{child}(k, i) = kd + i, \quad 0 < i \leq d$$

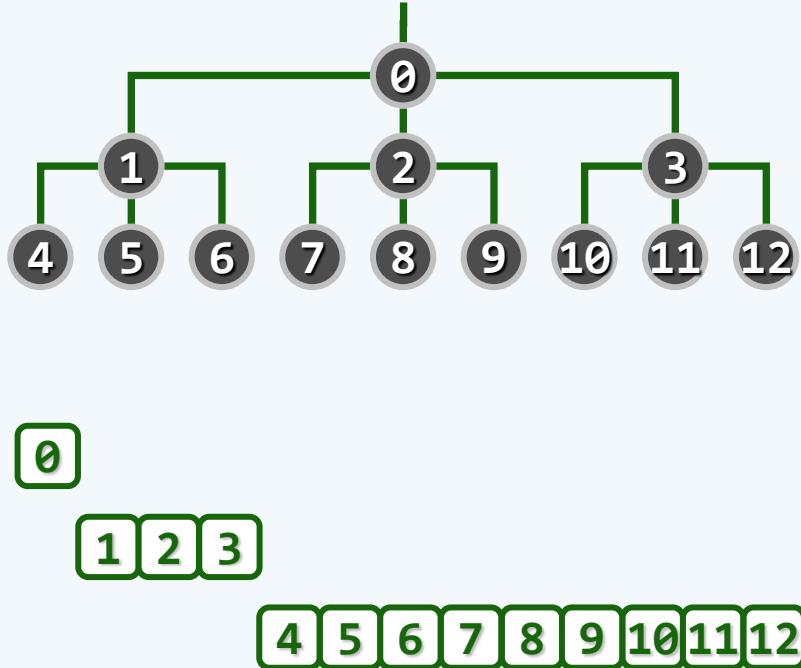
当然，d不再是2的幂时

将不再能够借助**移位**加速秩的换算

不过反过来，特别适用于

不主要取决于**秩换算**效率的场合

比如，数据**规模大**到需要**跨越**存储层次时 //策略上，与B-树完全一致



Fibonacci堆

❖ 左式堆 \times (新的上滤算法 + 懒惰合并)

❖ 各接口的分摊复杂度

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

`insert()` $O(\boxed{1})$

`merge()` $O(\boxed{1})$

`increase()` $O(\boxed{1})$

❖ 于是，基于 **PFS框架** 的算法采用 Fibonacci堆后，运行时间自然就是

$$n \cdot \mathcal{O}(\log n) + e \cdot \mathcal{O}(1) = \mathcal{O}(e + n \log n)$$

10. 优先级队列

God's right hand is gentle

But terrible is his left hand

左式堆

结构

君子居则贵左，用兵则贵右

吉事尚左，凶事尚右

偏将军居左，上将军居右

邓俊辉

deng@tsinghua.edu.cn

堆合并

❖ $H = \text{merge}(A, B)$: 将堆A和B合二为一 //不妨设 $|A| = n \geq m = |B|$

❖ 方法一 : $A.\text{insert}(B.\text{delMax}())$

$$\begin{aligned} & O(m * (\log m + \log(n + m))) \\ &= O(m * \log(n + m)) \end{aligned}$$

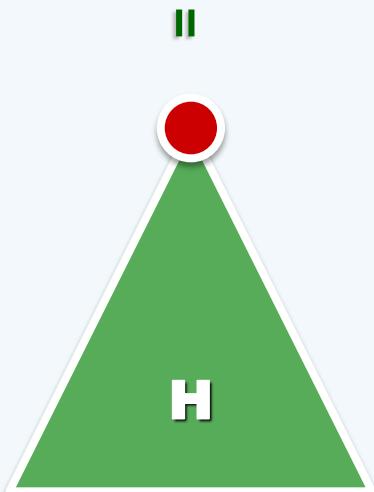


❖ 方法二 : $\text{union}(A, B).\text{heapify}(n + m)$

$$O(m + n)$$

❖ 有没有更好的办法 ? 比如...

❖ 可否奢望在... $O(\log n)$...时间内实现 $\text{merge}()$?



单侧倾斜

❖ C. A. Crane, 1972 : 保持堆序性，附加新条件，使得

在堆合并过程中，只需调整很少部分的节点 $\Theta(\log n)$

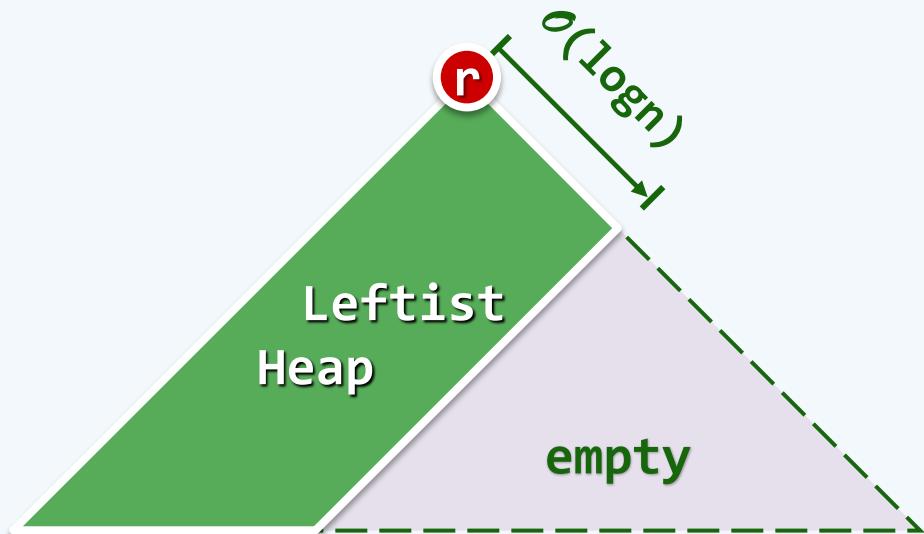
❖ 新条件 = 单侧倾斜 : 节点分布偏向于左侧

合并操作只涉及右侧

❖ 可是，果真如此，则...

❖ 拓扑上不见得是完全二叉树，结构性无法保证！？

❖ 是的，实际上，结构性并非堆结构的本质要求



空节点路径长度

❖ 引入所有的外部节点

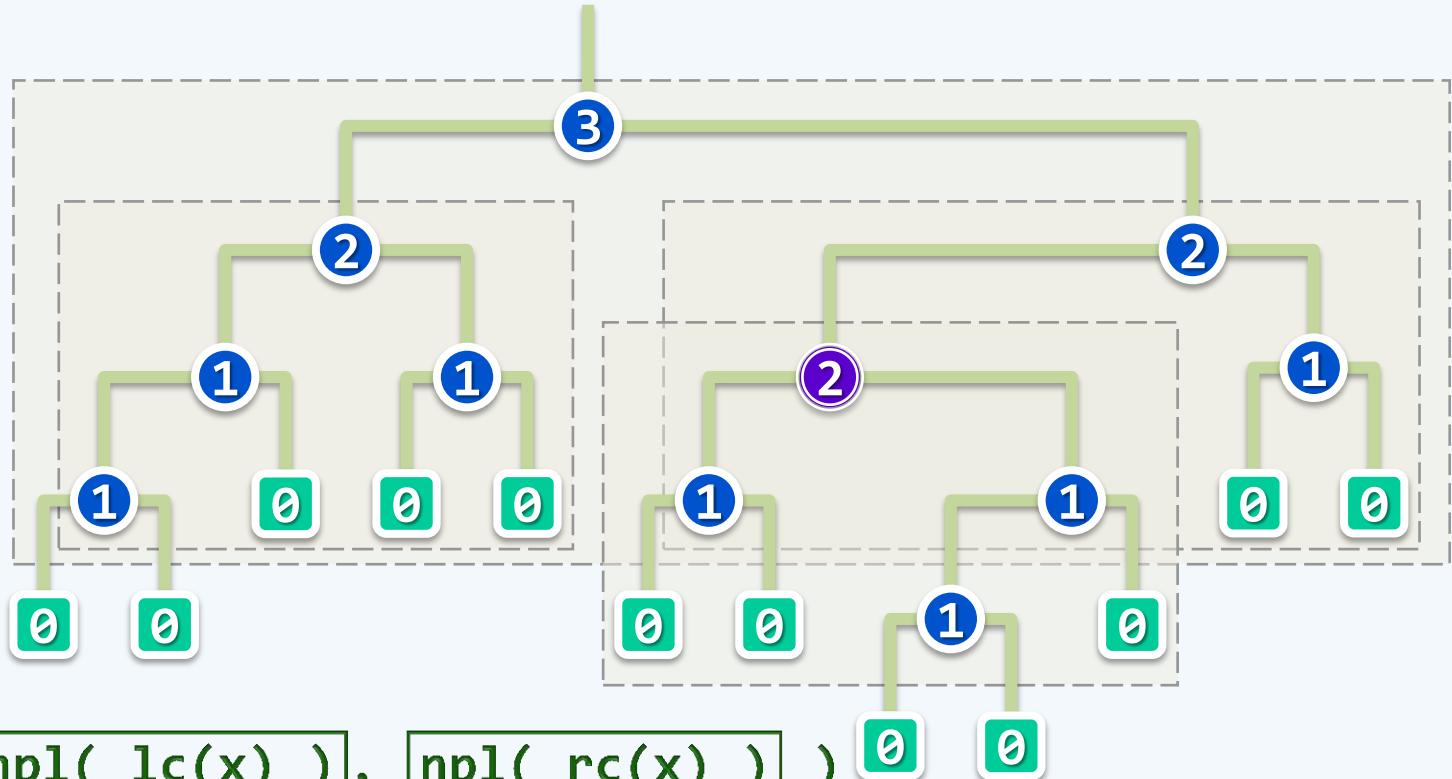
消除一度节点

转为真二叉树

❖ Null Path Length

$$0) \text{npl(} \text{NULL} \text{)} = 0$$

$$1) \text{npl(} x \text{)} = 1 + \min(\text{npl(} \text{lc}(x) \text{)}, \text{npl(} \text{rc}(x) \text{)})$$



❖ 验证： $\text{npl}(x) = x$ 到外部节点的最近距离

$\text{npl}(x) = \text{以 } x \text{ 为根的最大满子树的高度}$

左倾性 & 左式堆

❖ 左倾：对任何内节点 x ，都有 $\text{np1}(\boxed{\text{lc}(x)}) \geq \text{np1}(\boxed{\text{rc}(x)})$

推论：对任何内节点 x ，都有 $\text{np1}(\boxed{x}) = \boxed{1} + \text{np1}(\boxed{\text{rc}(x)})$

❖ 满足左倾性 **leftist property** 的堆，即是**左式堆**

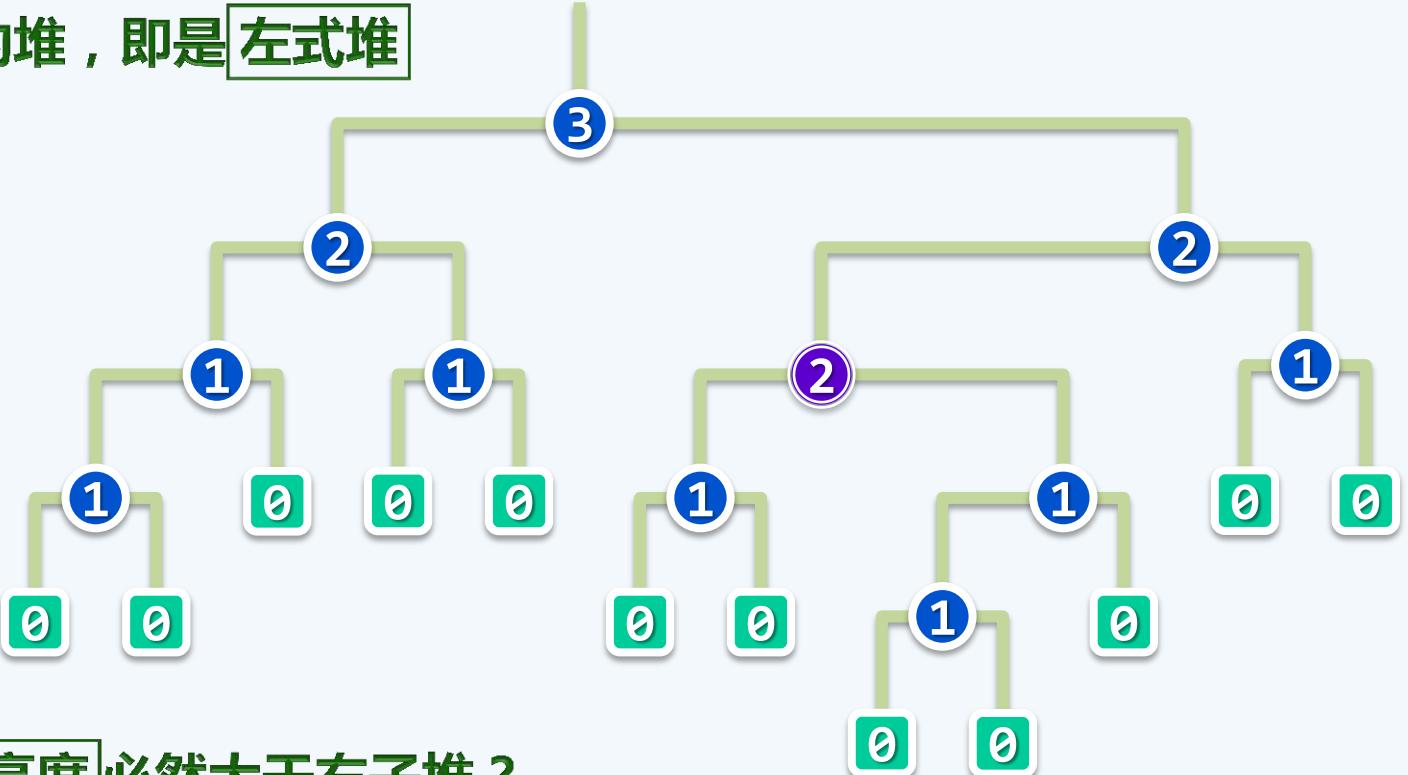
❖ 左倾性与堆序性，**相容而不矛盾**

❖ 左式堆的子堆，**必是左式堆**

❖ 左式堆倾向于

更多节点分布于**左侧**分支

❖ 这是否意味着，左子堆的**规模**和**高度**必然大于右子堆？



右侧链

❖ **rChain(x)** : 从节点x出发，一直沿**右分支**前进

❖ 特别地，**rChain(root)**的终点，必为全堆中最**浅**的外部节点

$$npl(r) = |rChain(r)| = d$$

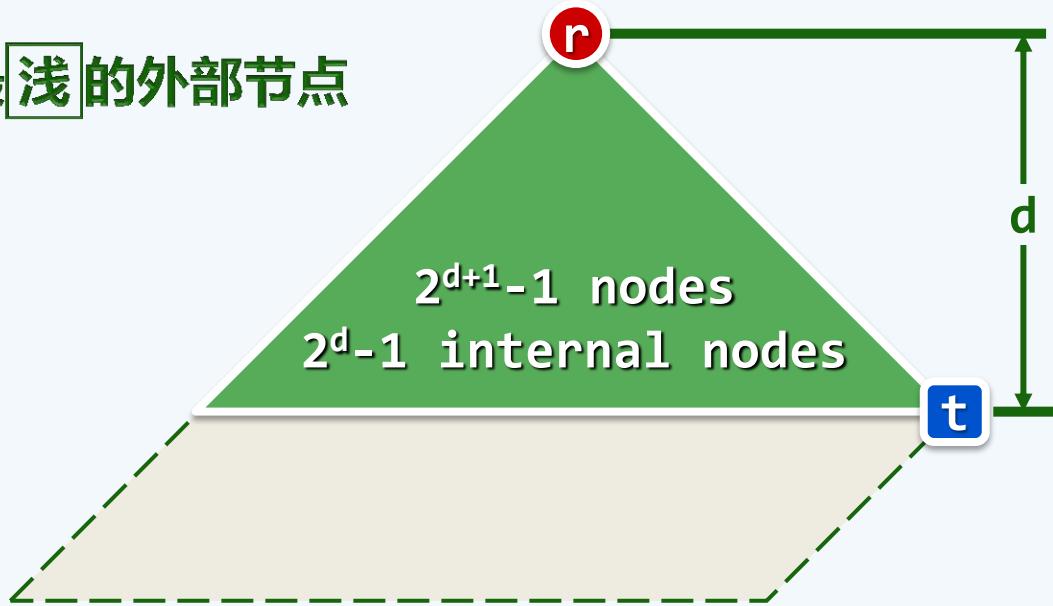
存在一棵以r为根、高度为d的满子树

❖ 右侧链长为d的左式堆，**至少**包含

$2^d - 1$ 个内部节点、 $2^{d+1} - 1$ 个节点

❖ 反之，在包含n个节点的左式堆中

$$\text{右侧链的长度 } d \leq \lfloor \log_2(n + 1) \rfloor - 1 = O(\log n)$$



10. 优先级队列

左式堆

合并

左之左之，君子宜之

右之右之，君子有之

邓俊辉

deng@tsinghua.edu.cn

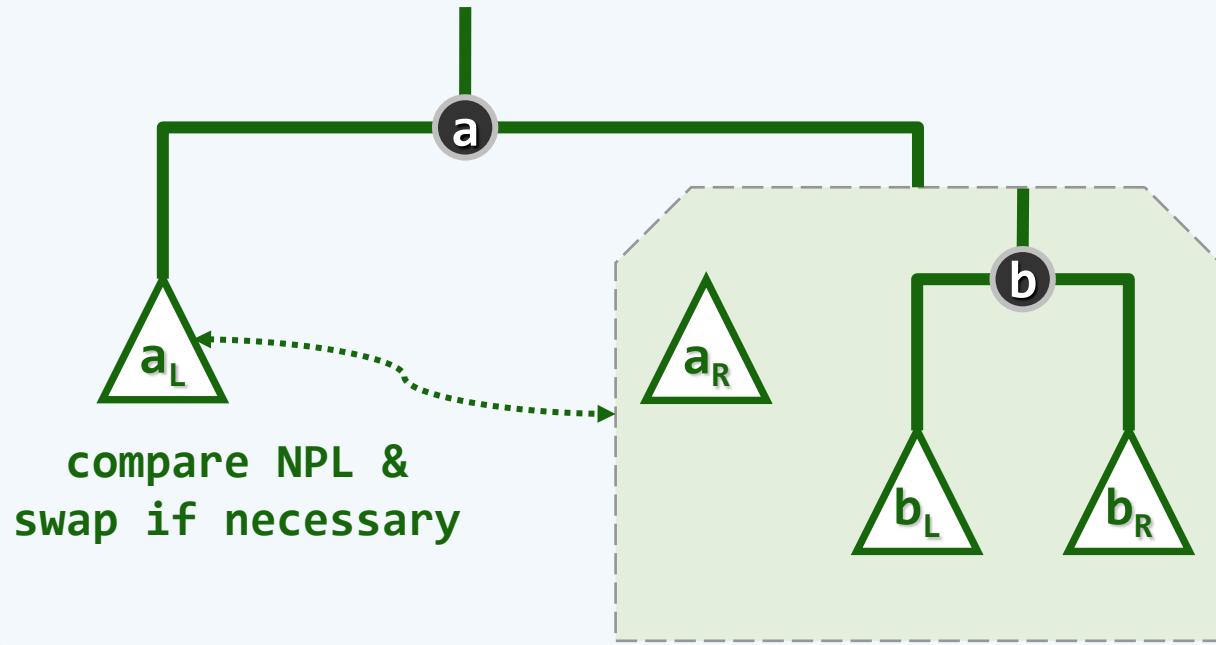
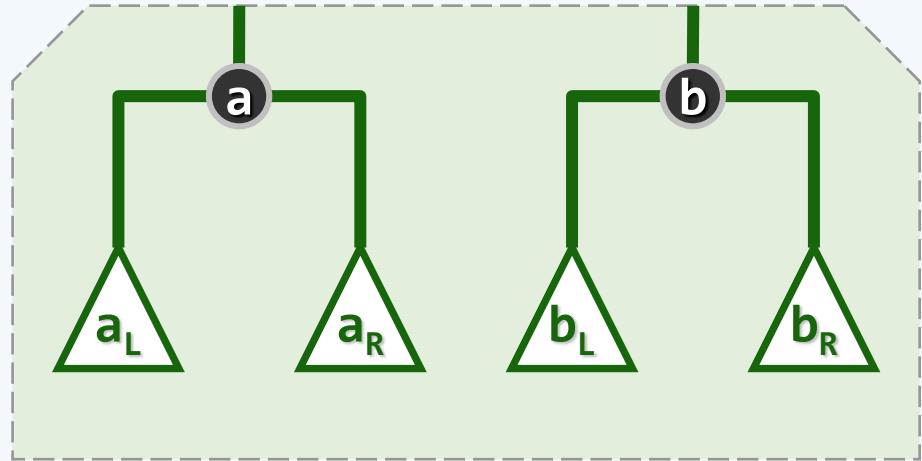
LeftHeap

❖ template <typename T> //基于二叉树，以左式堆形式实现的优先级队列

```
class PQ_LeftHeap : public PQ<T>, public BinTree<T> {  
  
public:  
  
    void insert(T); // (按比较器确定的优先级次序) 插入元素  
  
    T getMax() { return _root->data; } // 取出优先级最高的元素  
  
    T delMax(); // 删除优先级最高的元素  
  
}; // 主要接口，均基于统一的合并操作实现...
```

❖ template <typename T>

```
static BinNodePosi(T) merge( BinNodePosi(T), BinNodePosi(T) );
```



实现

```

❖ template <typename T>

    static BinNodePosi(T) merge( BinNodePosi(T) a, BinNodePosi(T) b ) {

        if ( ! a ) return b; //递归基
        if ( ! b ) return a; //递归基

        if ( lt( a->data, b->data ) ) swap( b , a ); //一般情况：首先确保b不大
a->rc = merge( a->rc, b ); //将a的右子堆，与b合并
a->rc->parent = a; //并更新父子关系

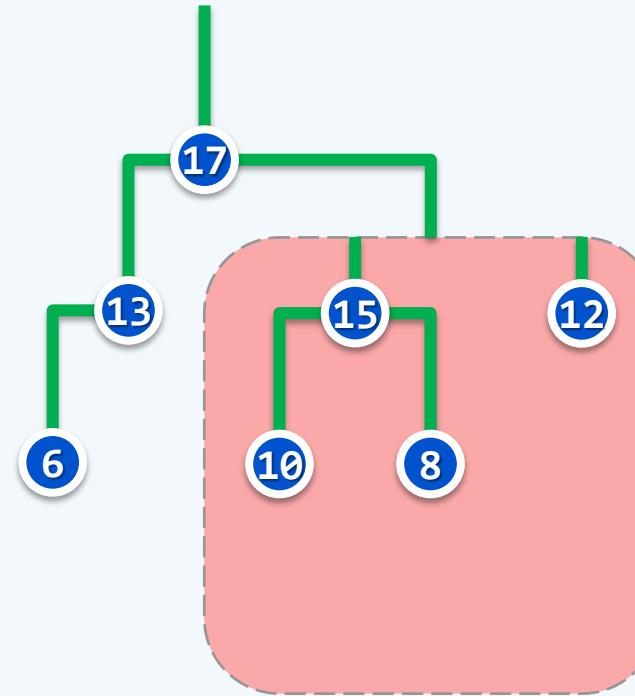
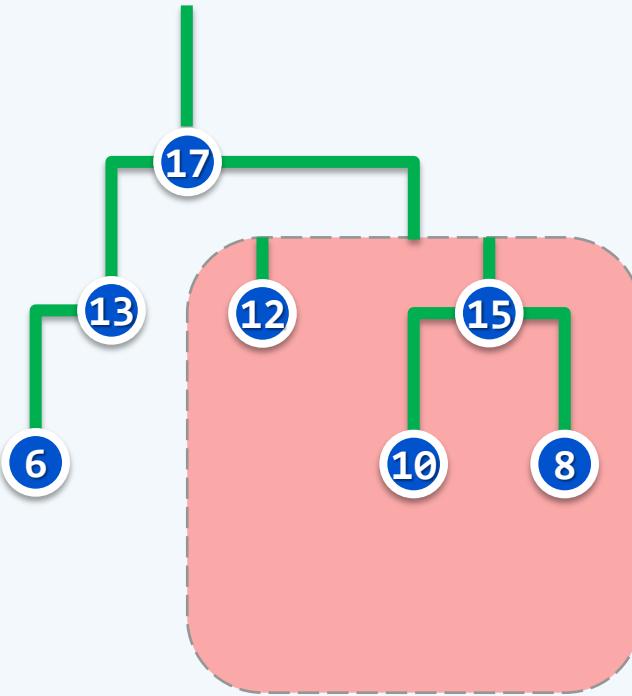
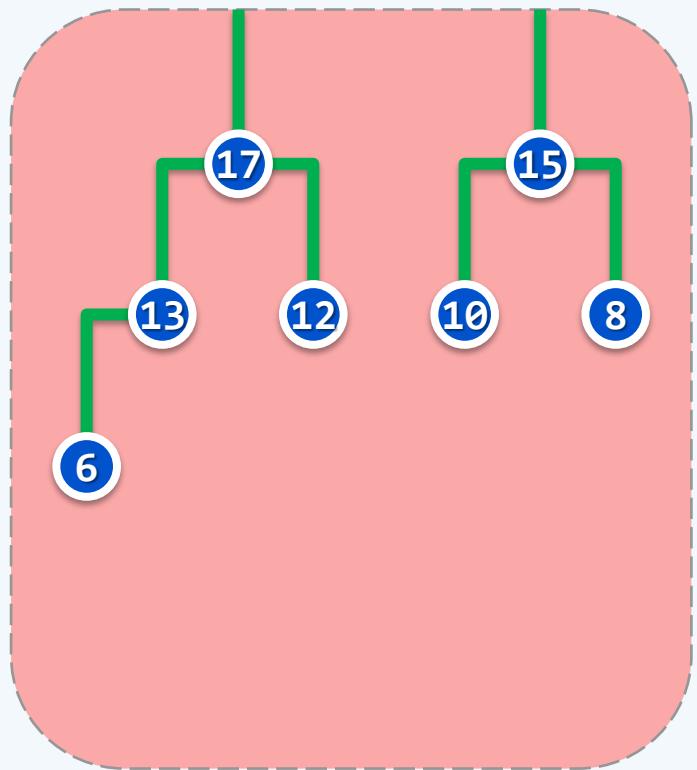
        if ( ! a->lc || a->lc->npl < a->rc->npl ) //若有必要
            swap( a->lc, a->rc ); //交换a的左、右子堆，以确保右子堆的npl不大

        a->npl = a->rc ? a->rc->npl + 1 : 1 ; //更新a的npl

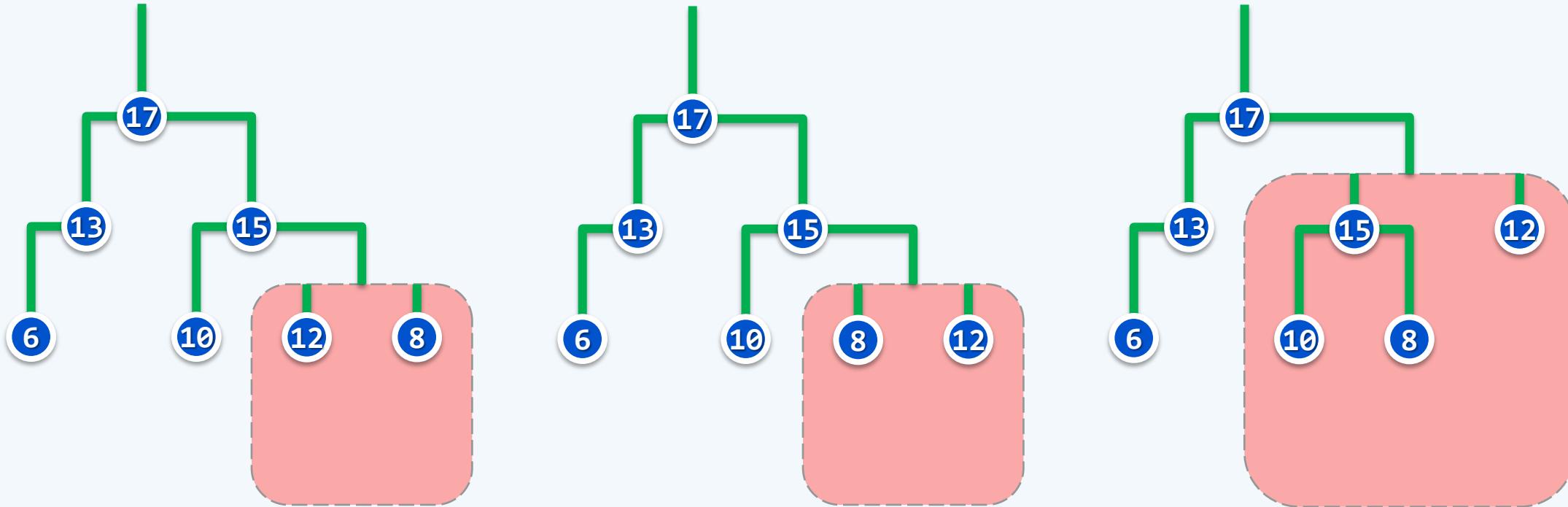
        return a; //返回合并后的堆顶
    }
}

```

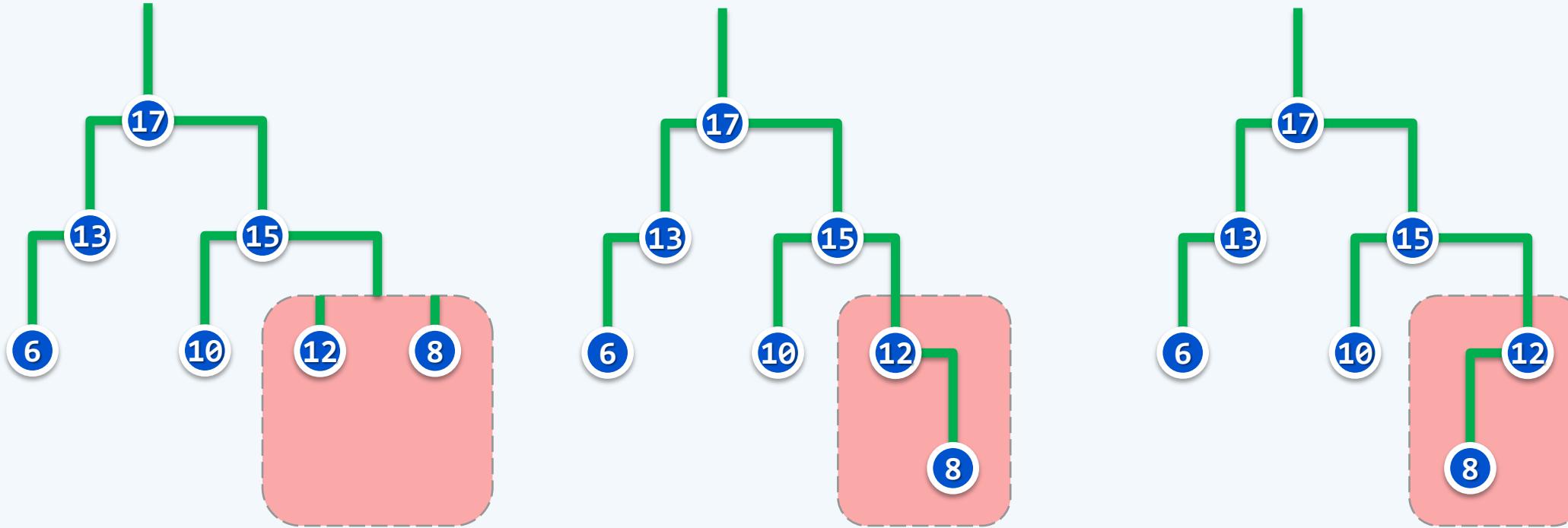
实例 (1/4)



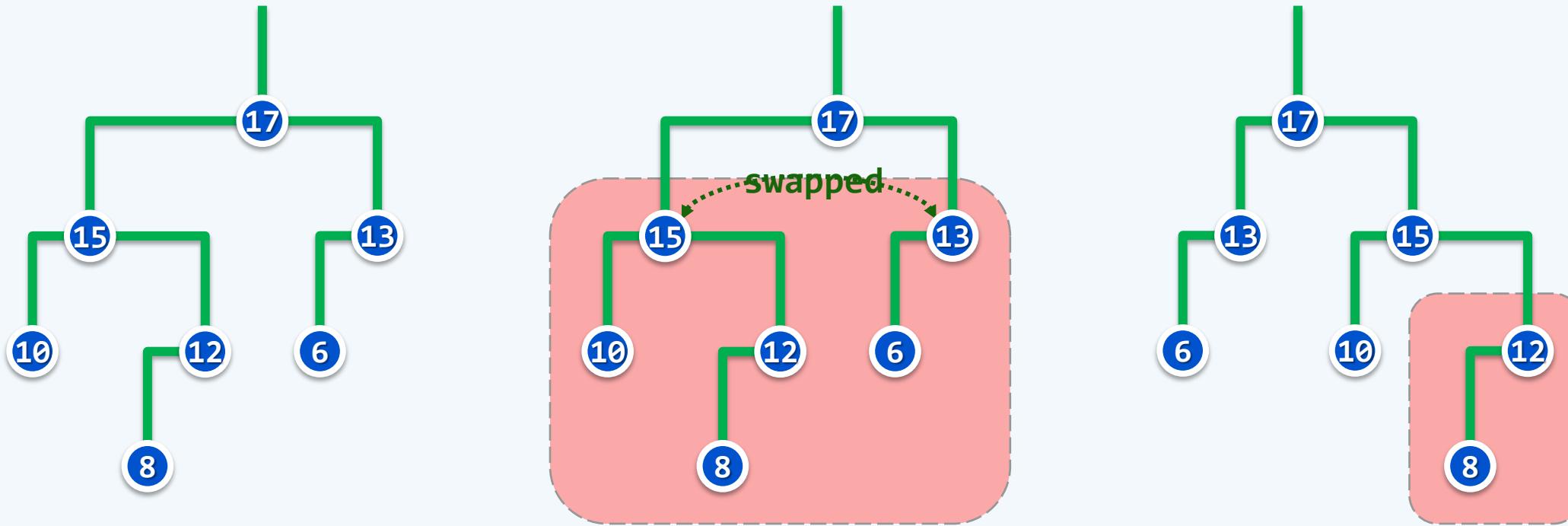
实例 (2/4)



实例 (3/4)



实例 (4/4)



10. 优先级队列

左式堆

插入与删除

一招鲜，吃遍天

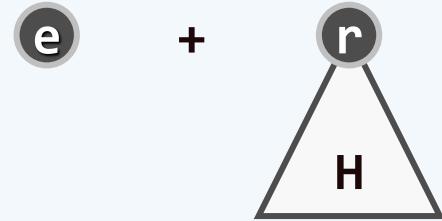
邓俊辉

体虽宣斜，而字心必正

deng@tsinghua.edu.cn

insert()

❖ template <typename T>



```
void PQ_LeftHeap<T>::insert( T e ) { //O(logn)
```

`BinNodePosi(T) v = new BinNode<T>(e); //为e创建一个二叉树节点`

`_root = merge(_root, v); //通过合并完成新节点的插入`

`_root->parent = NULL; //既然此时堆非空，还需相应设置父子链接`

`_size++; //更新规模`

}

delMax()

❖ template <typename T> T PQ_LeftHeap<T>::delMax() { // $O(\log n)$

BinNodePosi(T) lHeap = _root->lc; //左子堆

BinNodePosi(T) rHeap = _root->rc; //右子堆

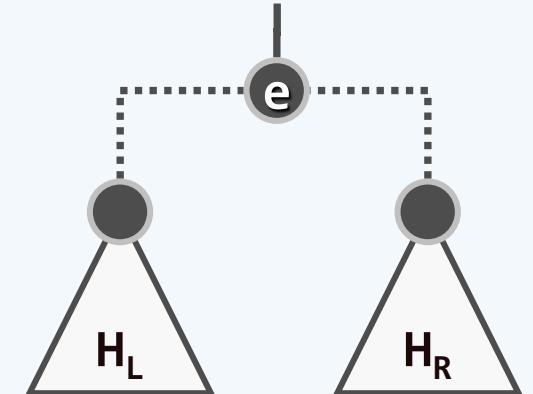
T e = _root->data; //备份堆顶处的最大元素

delete _root; _size--; //删除根节点

_root = merge(lHeap, rHeap); //原左、右子堆合并

if (_root) _root->parent = NULL; //更新父子链接

return e; //返回原根节点的数据项



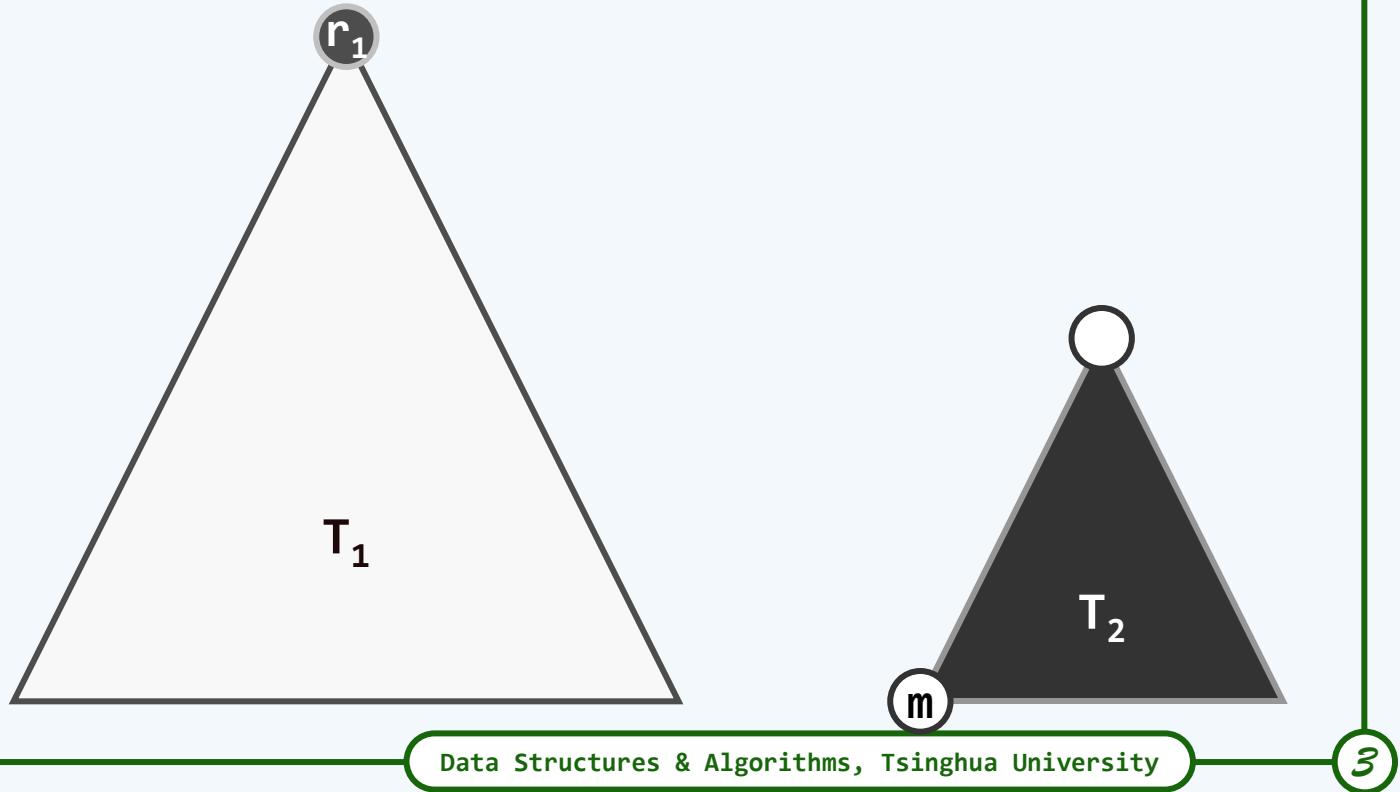
}

AVL::merge()

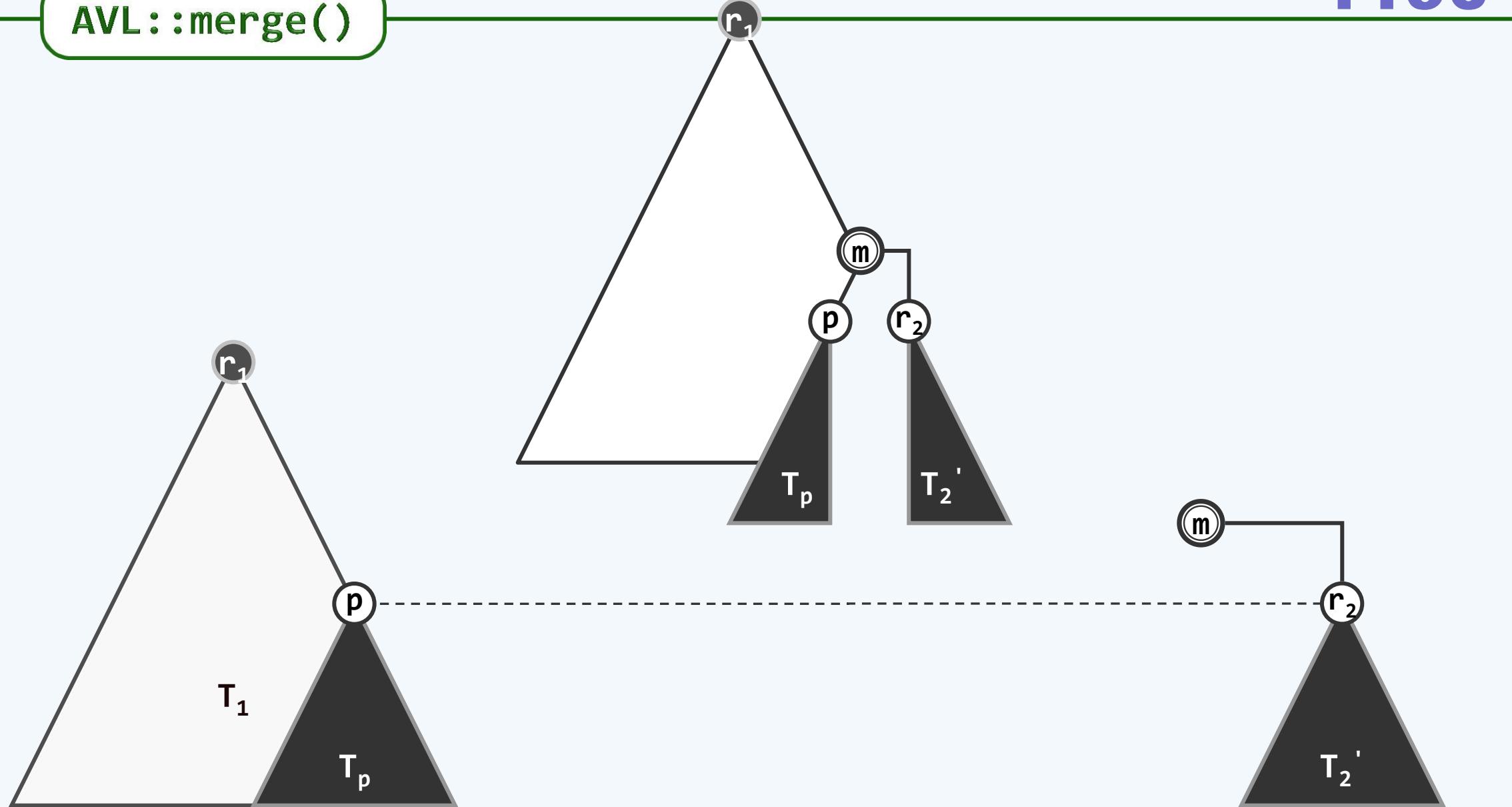
设 T_1 和 T_2 为两棵AVL树，且 $\max(T_1) < m = \min(T_2)$

如何尽快地将其合并为一棵AVL树？

WLOG， $\text{height}(T_1) \geq \text{height}(T_2)$



AVL::merge()



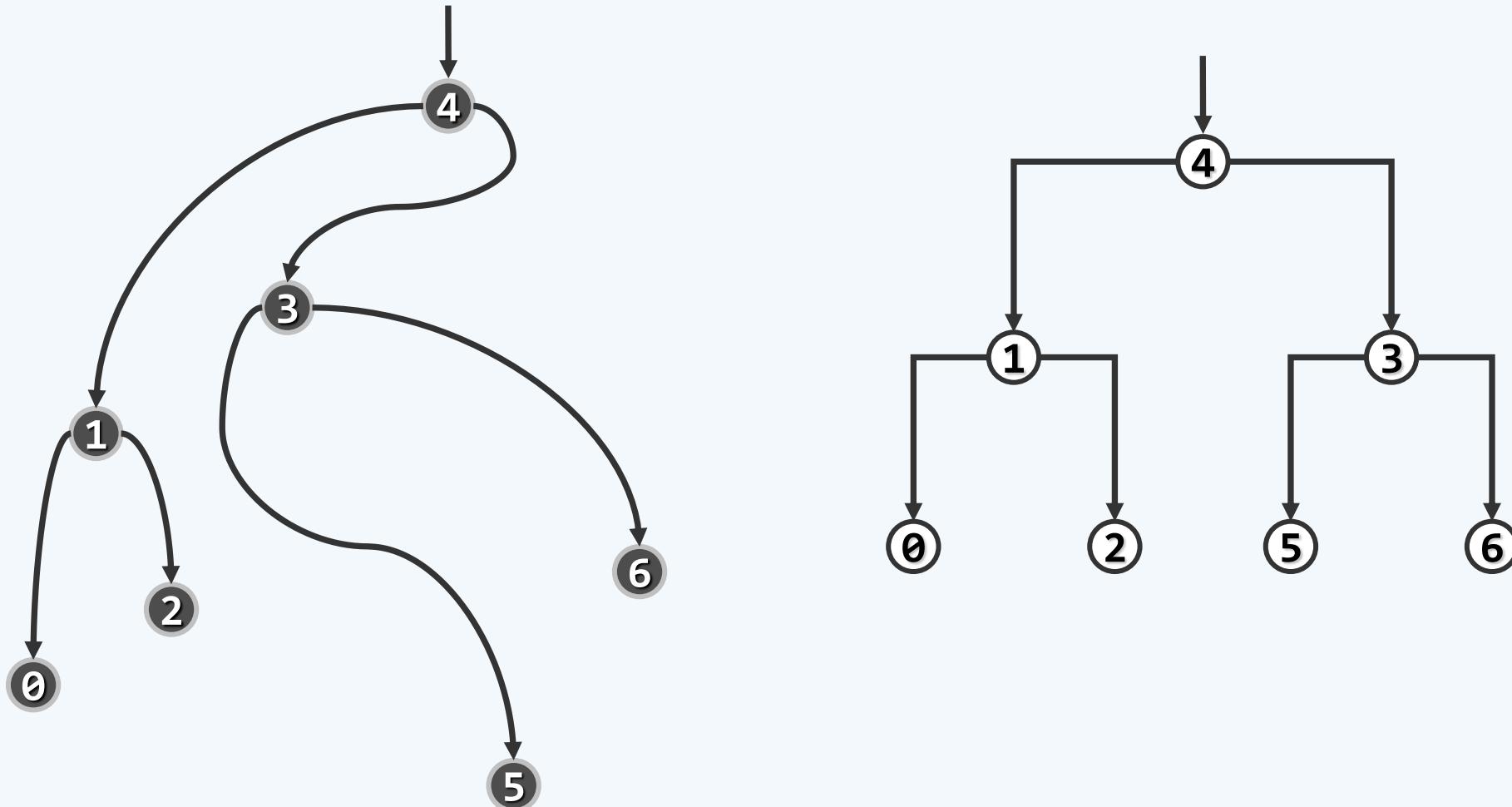
10. 优先级队列

优先级搜索树

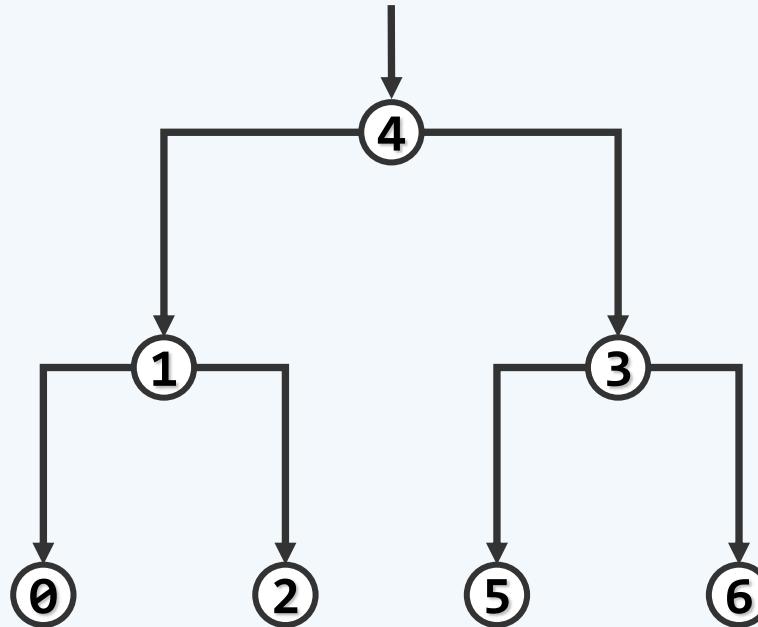
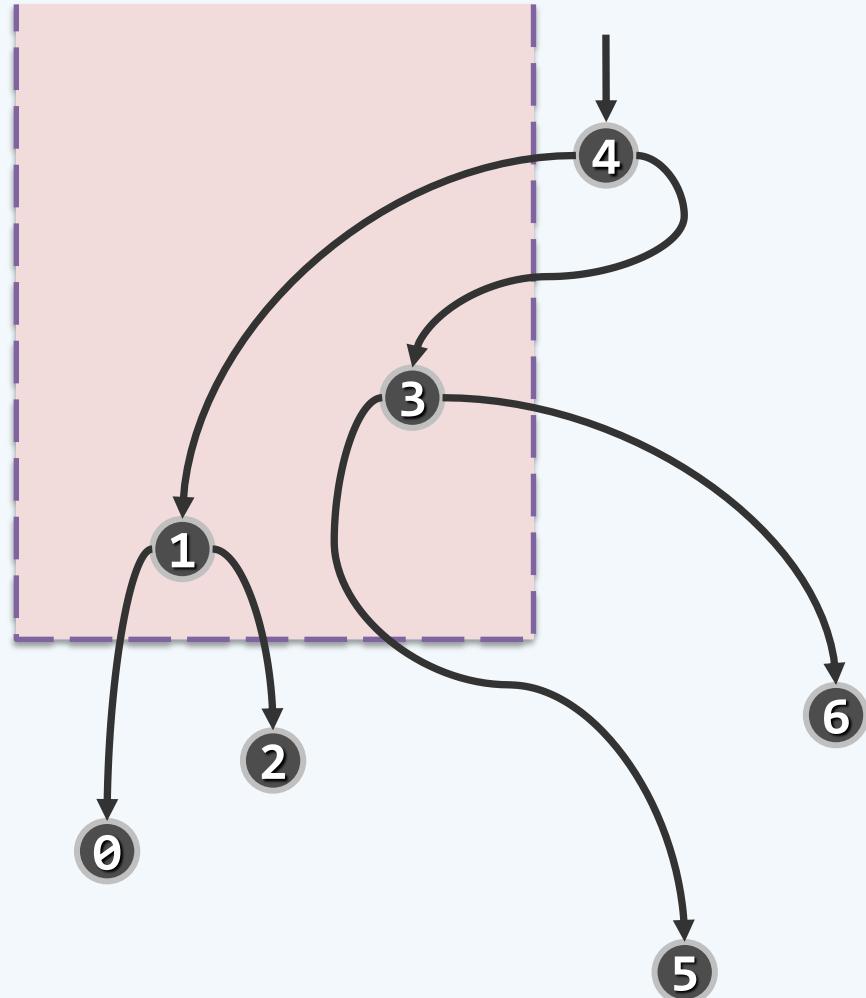
邓俊辉

deng@tsinghua.edu.cn

Priority Search Tree = BST + PQ



Grounded Range Query

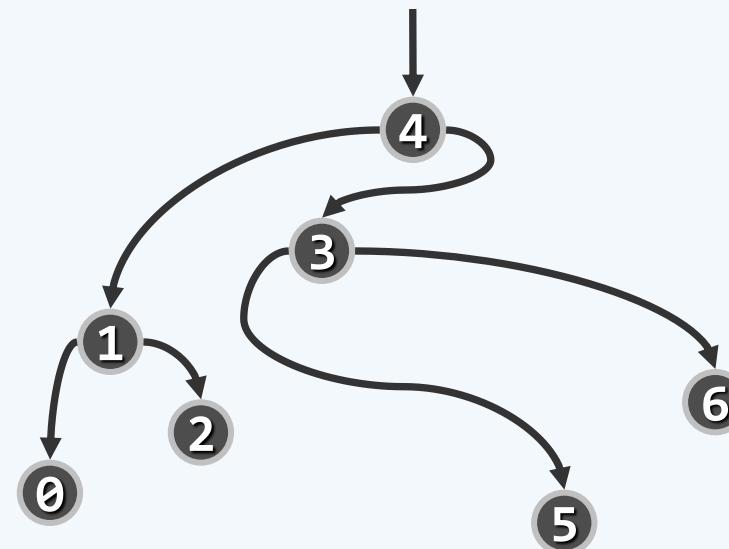
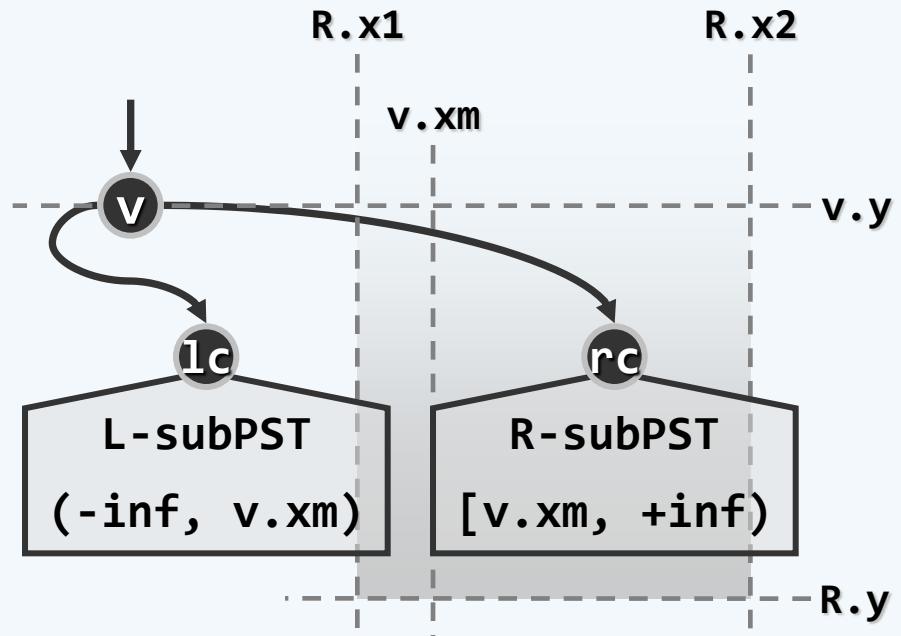


Query Algorithm

```

❖ queryPST( PSTNode v, SemiInfRange R ) {
    if ( !v || (R.y < v.y) ) return; //prune
    if ( (R.x1 < v.x) && (v.x < R.x2) ) output(v); //hit
    if ( (R.x1 < v.xm) queryPST( v.lc, R ); //recursion
    if ( (v.xm <= R.x2) queryPST( v.rc, R ); //recursion
}

```



1195

11.串

ADT

邓俊辉

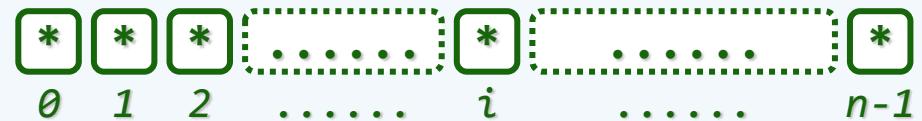
deng@tsinghua.edu.cn

定义

❖ 由来自**字母表 Σ** 的字符所组成的**有限序列**：

$$s = [a_0 \ a_1 \ a_2 \ \dots \ a_{n-1}] \in \Sigma^*$$

❖ 既然如此，为何不直接用**序列**来实现串？



❖ 通常，字符的种类不多，而串长 $= n \gg |\Sigma|$

❖ 比如： 英文文章 $(['A' - 'Z'] \cup ['a' - 'z'] \cup \{ '.', ',', '!', ... \})^*$

C/C++程序 $(\{95\text{个可打印字符}\} \cup \{\text{LF, CR}\})^*$

天然蛋白质 $\{21\text{种氨基酸}\}^*$

DNA $\{A, C, G, T\}^*$

RNA $\{A, C, G, U\}^*$

二进制 $\{0, 1\}^*$

术语

❖ 相等： $s[0, n) = t[0, m)$

 长度相等 ($n = m$)，且对应的字符均相同 ($s[i] = t[i]$)

❖ 子串： $s.substr(i, k) = s[i, i + k), 0 \leq i < n, 0 \leq k$

 亦即，从 $s[i]$ 起的连续 k 个字符 [0, i) [i, i + k) [i + k, n)

❖ 前缀： $s.prefix(k) = s.substr(0, k) = s[0, k), 0 \leq k \leq n$

 亦即， s 中最靠前的 k 个字符 [0, k) [k, n)

❖ 后缀： $s.suffix(k) = s.substr(n - k, k) = s[n - k, n), 0 \leq k \leq n$

 亦即， s 中最靠后的 k 个字符 [0, n - k) [n - k, n)

❖ 联系： $s.substr(i, k) = s.prefix(i + k).suffix(k)$

❖ 空串： $s[0, n = 0)$ ，也是任何串的子串、前缀、后缀

ADT

length()[0, n)**charAt(i)**

[0, i)

[i]

(i, n)

substr(i, k)

[0, i)

[i, i + k)

[i + k, n)

prefix(k)

[0, k)

[k, n)

suffix(k)

[0, n - k)

[n - k, n)

concat(T)

S

T

equal(T)

S

T

indexOf(P)

S

[k , k + m)

P[0, m)

实例

- ❖ "data structures".length() = 15
- "data structures".charAt(5) = 's'
- "data structures".prefix(4) = "data"
- "data structures".suffix(10) = "structures"
- "data structures".concat(" & algorithms") = "data structures & algorithms"
- "algorithms".equal("data structures") = false
- "data structures and algorithms".indexOf("string") = -1
- "data structures and algorithms".indexOf("algorithm") = 20
- ❖ <string.h> 中的对应功能 : strlen()、 strcpy()、 strcat()、 strcmp()、 strstr()
- ❖ 以下，直接利用字符数组实现字符串，转而重点讨论串匹配算法

11. 串

模式匹配

问题描述

邓俊辉

deng@tsinghua.edu.cn

模式匹配

% grep <pattern> <text>

文本 T = now is the time for all good people to come

模式 P = people

❖ 记 $n = |T|$ 和 $m = |P|$ ，通常有 $n \gg m \gg 2$ //比如， $100,000 \gg 100 \gg 2$

❖ Pattern matching

detection : P是否出现？

location : 首次在哪里出现？ //本章主要讨论的问题

counting : 共有几次出现？ //find /c "2013" students.txt

enumeration : 各出现在哪里？ //find "2013" students.txt

模式匹配

❖ 岐义： $T = "1\ 0\ 0\ 1\ \boxed{1\ 0\ 1\ 1}\ 0\ \boxed{1\ 0\ 1}\ \boxed{1}\ \boxed{0\ 1\ 1}\ 1\ 0\ 0\ 1"$

$P = " \boxed{1\ 0\ 1\ 1} "$

❖ 应用：文本编辑器、数据库检索、C++模板匹配、模式识别、搜索引擎、...

❖ 应用：生物序列分析 (biological sequence analysis)

通常不能完全匹配

——alignment：最接近的匹配在什么位置？

HBA_HUMAN vs. HBB_HUMAN

\boxed{G}	S	A	Q	$\boxed{V\ K}$	G	H	G	K	K	V	A	D	\boxed{A}	L	T	N	A	V	$\boxed{A\ H}$	V	\boxed{D}	D	M	P	N	A	L	S	A	$\boxed{L\ S}$	D	$\boxed{L\ H}$	A	H	
\boxed{G}	N	P	K	$\boxed{V\ K}$	A	H	G	K	K	V	L	G	\boxed{A}	F	S	D	G	L	$\boxed{A\ H}$	L	\boxed{D}	D	N	L	K	G	T	F	A	T	$\boxed{L\ S}$	E	$\boxed{L\ H}$	C	D

算法评测

❖ 如何客观地 **测量与评估** 串匹配算法的**性能** ? 具体采用什么**标准与策略** ?

❖ **随机T + 随机P** ? 不妥 !

❖ 以 $\Sigma = \{0, 1\}^*$ 为例 $| \{ \text{长度为 } m \text{ 的P} \} | = 2^m$

$$| \{ \text{长度为 } m \text{ 且在T中出现的P} \} | = n - m + 1 < n$$

匹配成功的概率 $= n/2^m \ll 100,000 / 2^{100} < 10^{-25}$

如此 , 将无法对算法做充分测试

❖ **随机T** , 对成功、失败的匹配**分别** 测试

成功 : 在T中 , 随机取出长度为m的**子串**作为P ; 分析平均复杂度

失败 : 采用**随机**的P ; 统计平均复杂度

11. 串

模式匹配

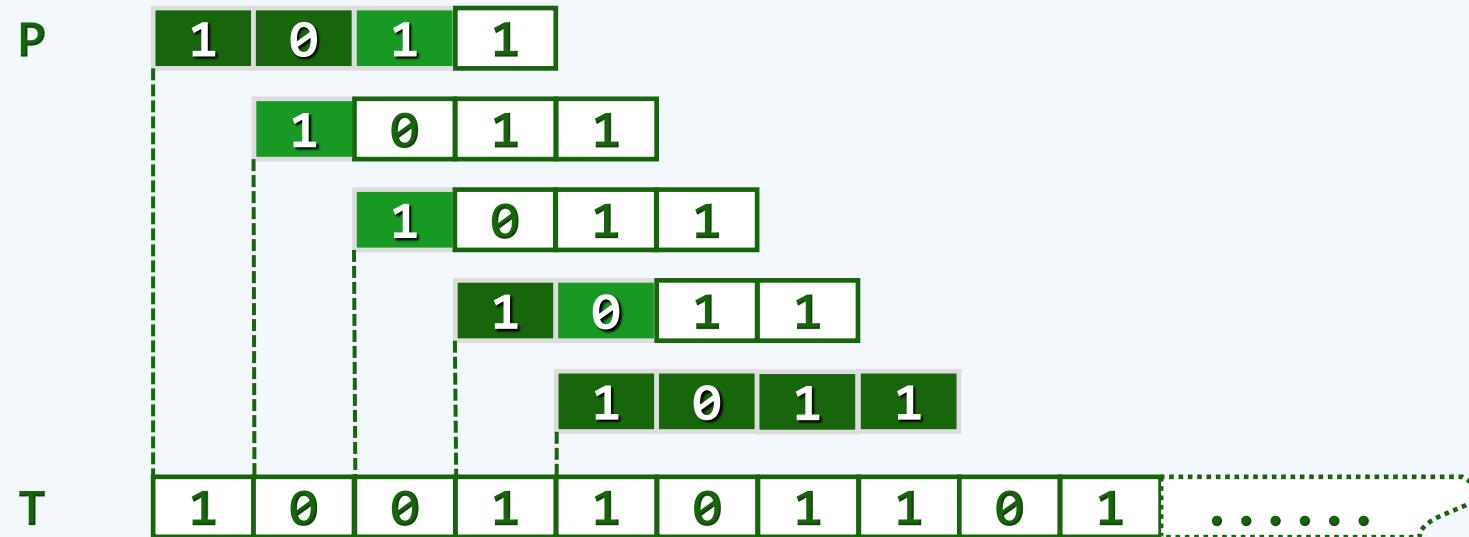
蛮力匹配

邓俊辉

deng@tsinghua.edu.cn

构思

❖ 自左向右，以字符为单位，依次移动模式串
直到在某个位置，发现匹配



❖ 如何：确定串T和P每次做比对的字符位置？并
在发现某对字符失配后，调整位置以继续比对？

实现

❖ 实现1：**i**和**j**分别指向T[]和P[]中待比对的字符

```

if ( T[i] == P[j] ) { i++; j++; }           //若匹配，则i和j同时右移

else { i -= j - 1; j = 0; }                   //若失配，则T回退、P复位

```

❖ 实现2：**i + j**和**j**分别指向T[]和P[]中待比对的字符

```

if ( T[i + j] == P[j] ) { j++; }             //若匹配，则i + j和j同时右移

else { i++; j = 0; }                          //若失配，则i右移，j回溯

```

❖ 这两种实现方法，各有什么优缺点？

版本1

```
❖ int match( char * P, char * T ) {
```

```
    size_t n = strlen(T), i = 0;
```

```
    size_t m = strlen(P), j = 0;
```

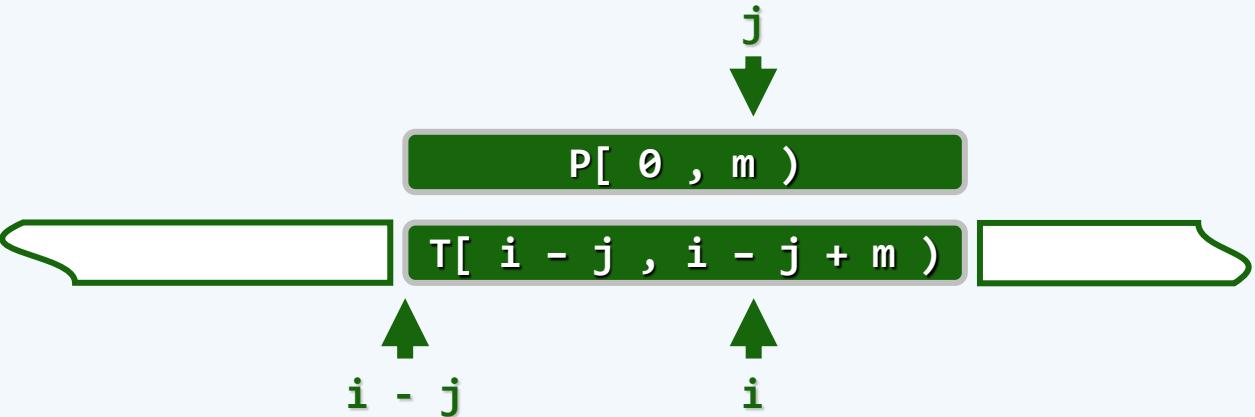
//自左向右逐个比对字符

```
    if ( T[i] == P[j] ) { i++; j++; } //若匹配，则转到下一对字符
```

```
    else { i -= j - 1; j = 0; } //否则，T回退、P复位
```

```
return i - j;
```

//如何通过返回值，判断匹配结果？



版本2

```

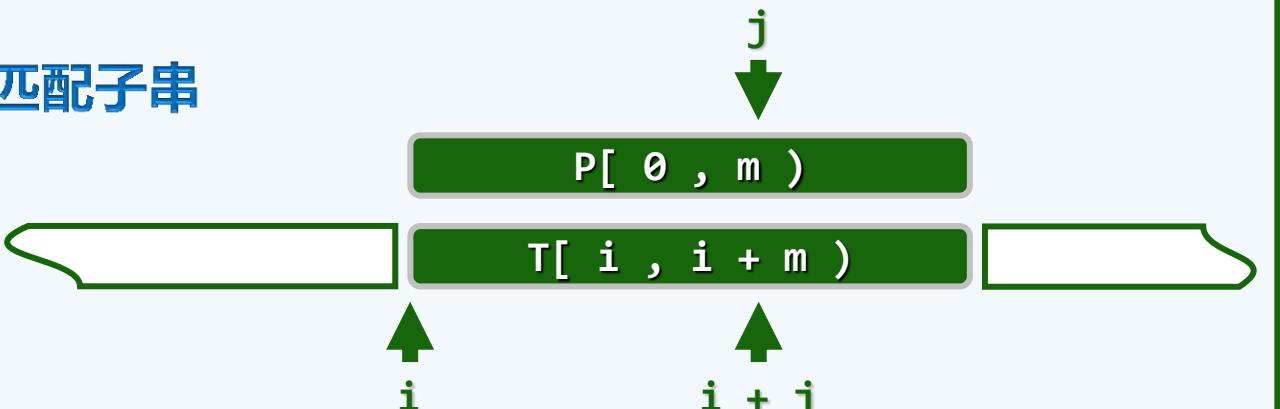
❖ int match( char * P, char * T ) {

    size_t n = strlen(T), i = 0; //T[i]与P[0]对齐
    size_t m = strlen(P), j; //T[i + j]与P[j]对齐

    for ( i = 0; i < n - m + 1; i ++ ) { //T从第i个字符起，与
        for ( j = 0; j < m; j ++ ) //P中对应的字符逐个比对

            if ( T[i + j] != P[j] ) break; //若失配，P整体右移一个字符，重新比对
            if ( m <= j ) break; //找到匹配子串
    }
    return i;
} //如何通过返回值，判断匹配结果？

```



复杂度

❖ 最好情况（只经过一轮比对，即可确定匹配）：#比对 = m = $O(m)$

❖ 最坏情况（每轮都比对至P的末字符，且反复如此）

每轮循环：#比对 = m - 1(成功) + 1(失败) = m

循环次数 = n - m + 1

一般地有 $m \ll n$

故总体地，#比对 = m × (n - m + 1) = $O(n \times m)$

❖ 最坏情况，真会出现？



是的！



❖ $|\Sigma|$ 越小，最坏情况出现的【概率】越高；m 越大，最坏情况的【后果】更加严重

11. 串

KMP算法

记忆法

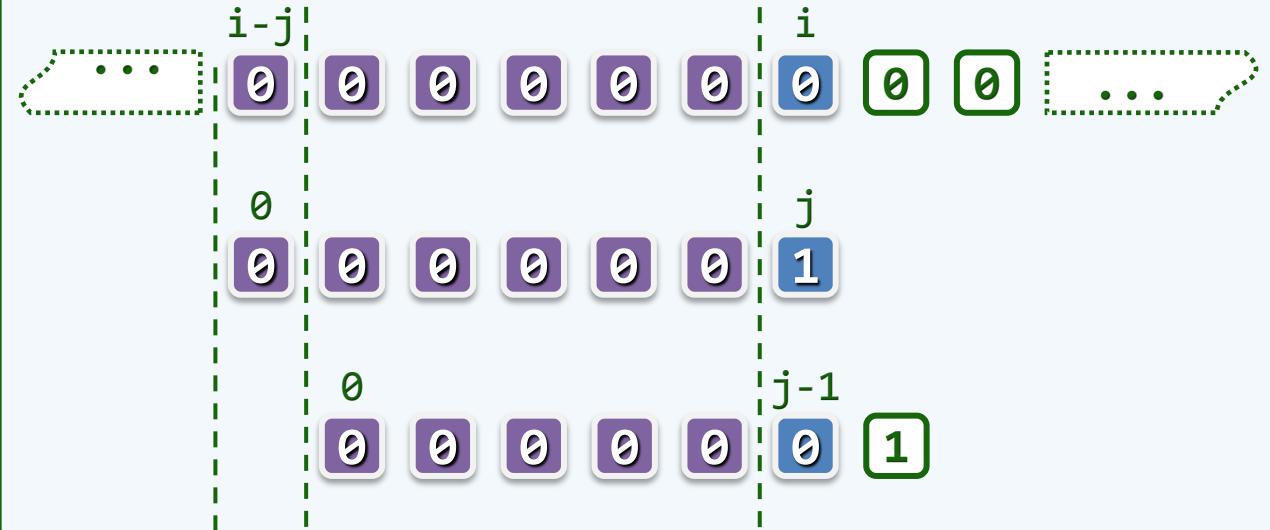
邓俊辉

知易者不占，善易者不卜

deng@tsinghua.edu.cn

蛮力，为何低效

- ❖ T回退、P复位之后，此前比对过的字符，将再次参与比对



- ▶ ❖ 最坏情况下，T/P中每个字符平均参加m/n次比对——累计 $\mathcal{O}(m*n)$ 次
- ❖ 于是，只要局部匹配很多，效率必将很低
- ❖ 其实，这类比对大多是不必要的，因为...

无论如何，还是不变性

❖ $T[i - j, i] == P[0, j]$

$T[i] == P[j]$

?

$T[i - j, i]$

*

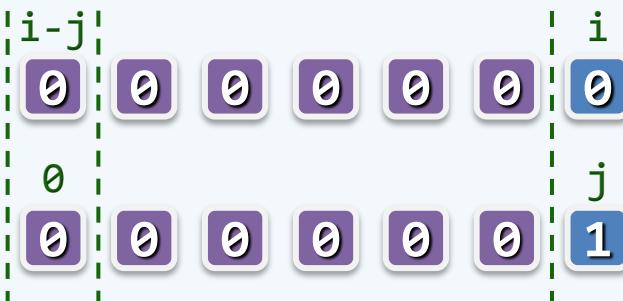
$T(i, n)$

$P[0, j]$

*

$P(j, m)$

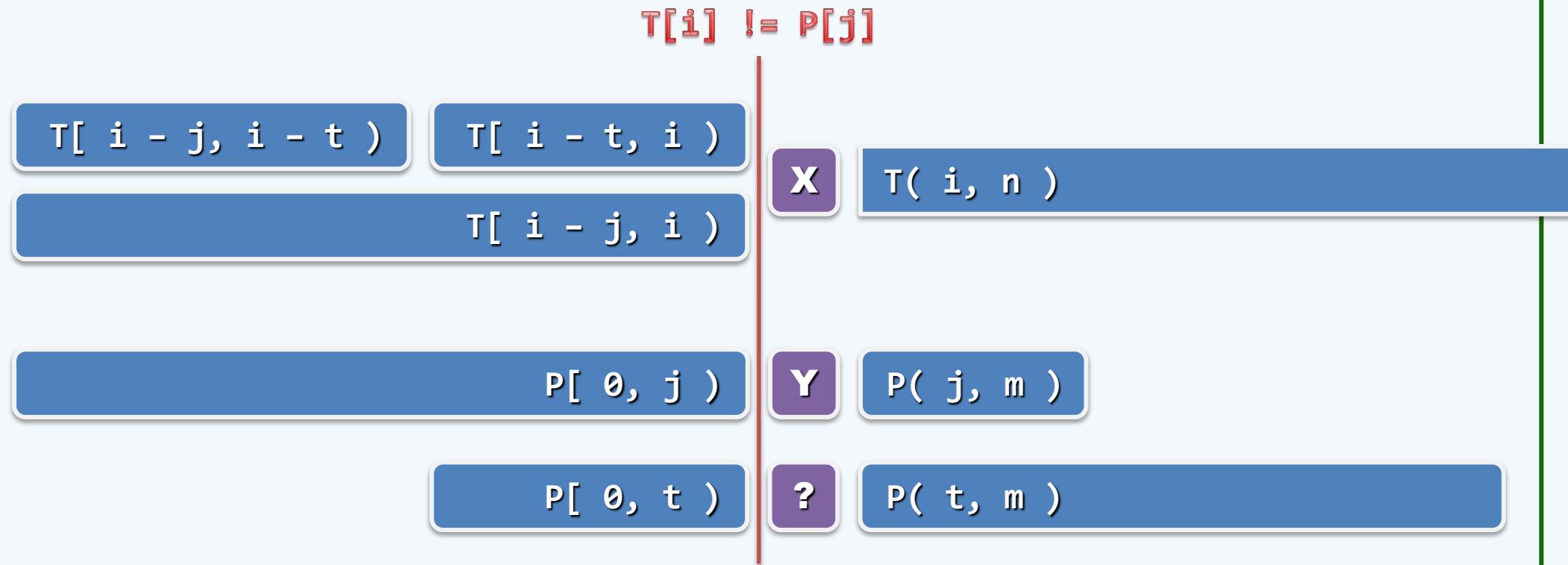
❖ 亦即，我们业已掌握 $T[i - j, i]$ 的全部信息——其中的字符各是什么



❖ 既如此...

只要**记忆力**足够强

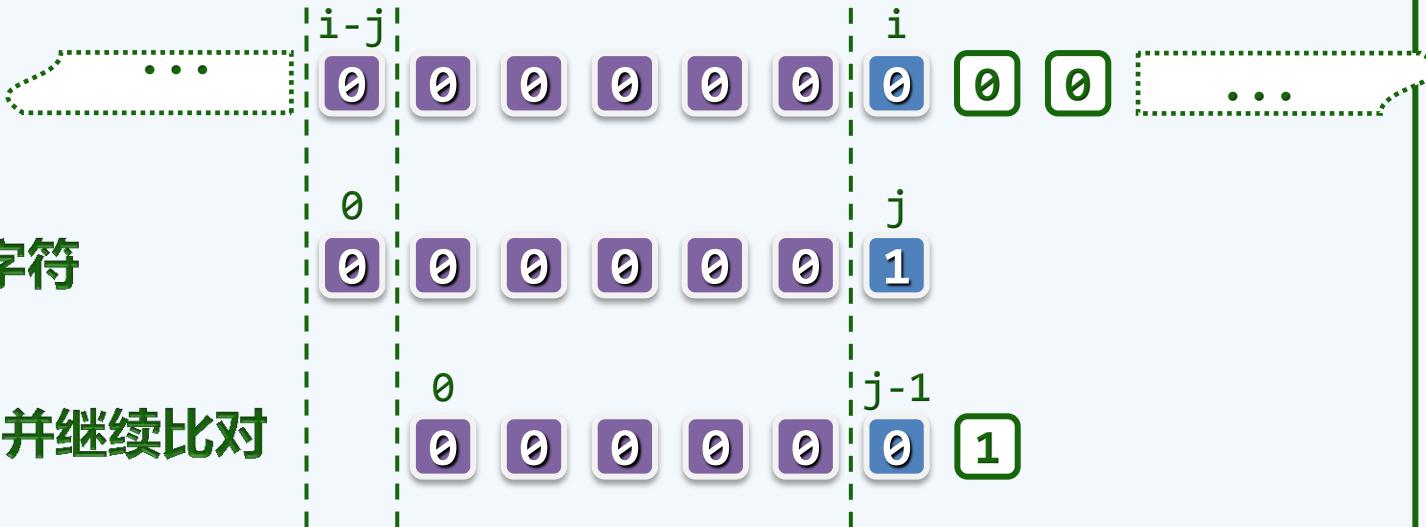
- ❖ 在失败之后我们已知道：哪些位置不必对齐，哪些值得对齐 //而且对于后一类位置...
- ❖ 在下一轮比对中， $T[i - j, i)$ 将**不必再次**接受比对，而是可以直接地...



将记忆力，转化为预知力

◆如此，**i**将完全不必回退！

- 比对成功，则与 j 同步前进一个字符
 - 否则， j 更新为 $\text{某个更小的 } t$ ，并继续比对

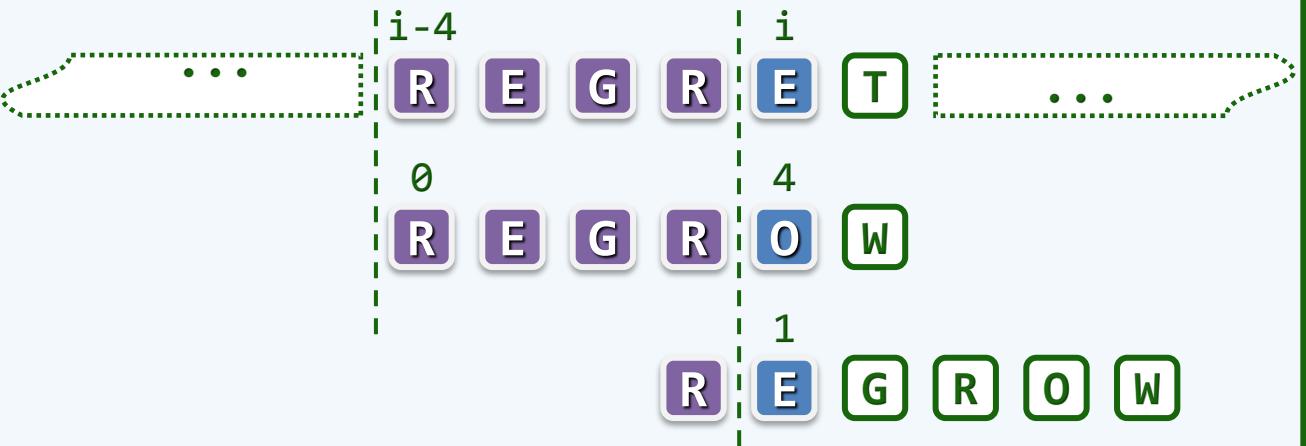


❖ 即便是更为复杂的情况，依然可行

◆ 优化 = P可快速右移 + 避免重复比对

❖ 为确定 t ，需花费多少时间和空间？

更重要地，可否在事先就确定？



11. 串

KMP算法

查询表

邓俊辉

好记性不如烂笔头

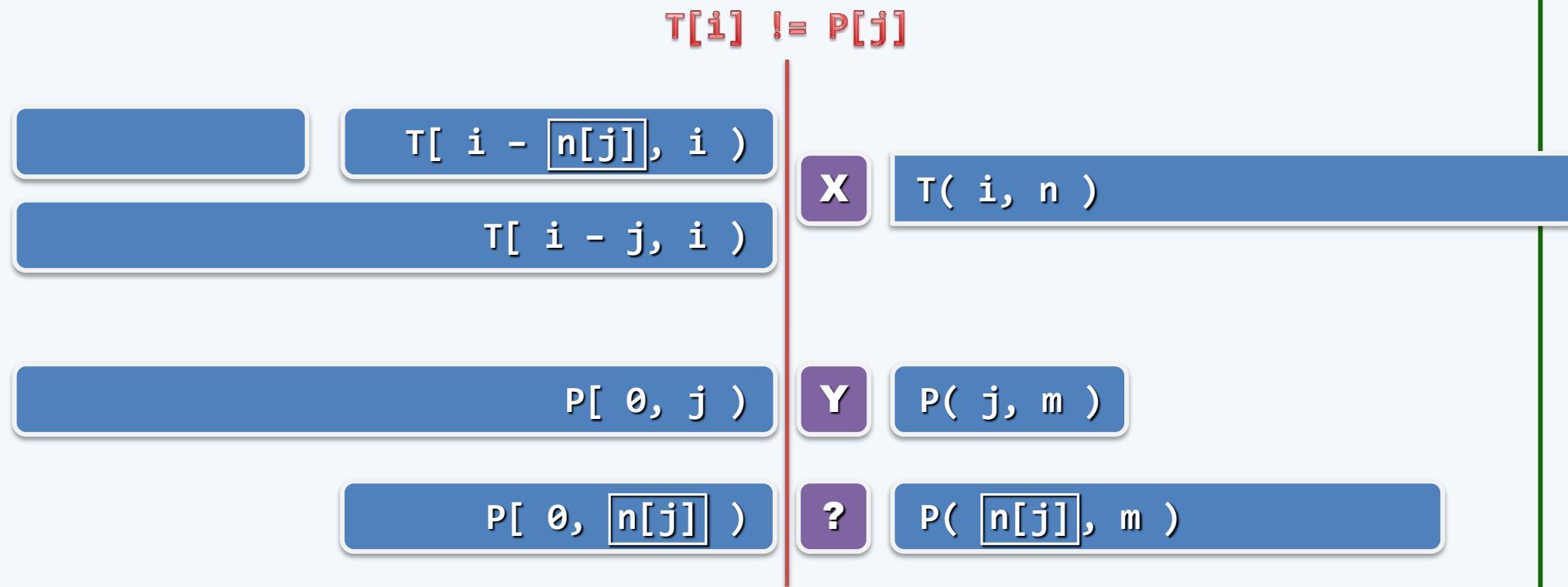
deng@tsinghua.edu.cn

事先确定 t

不仅可以事先确定，而且仅根据 P 即可确定

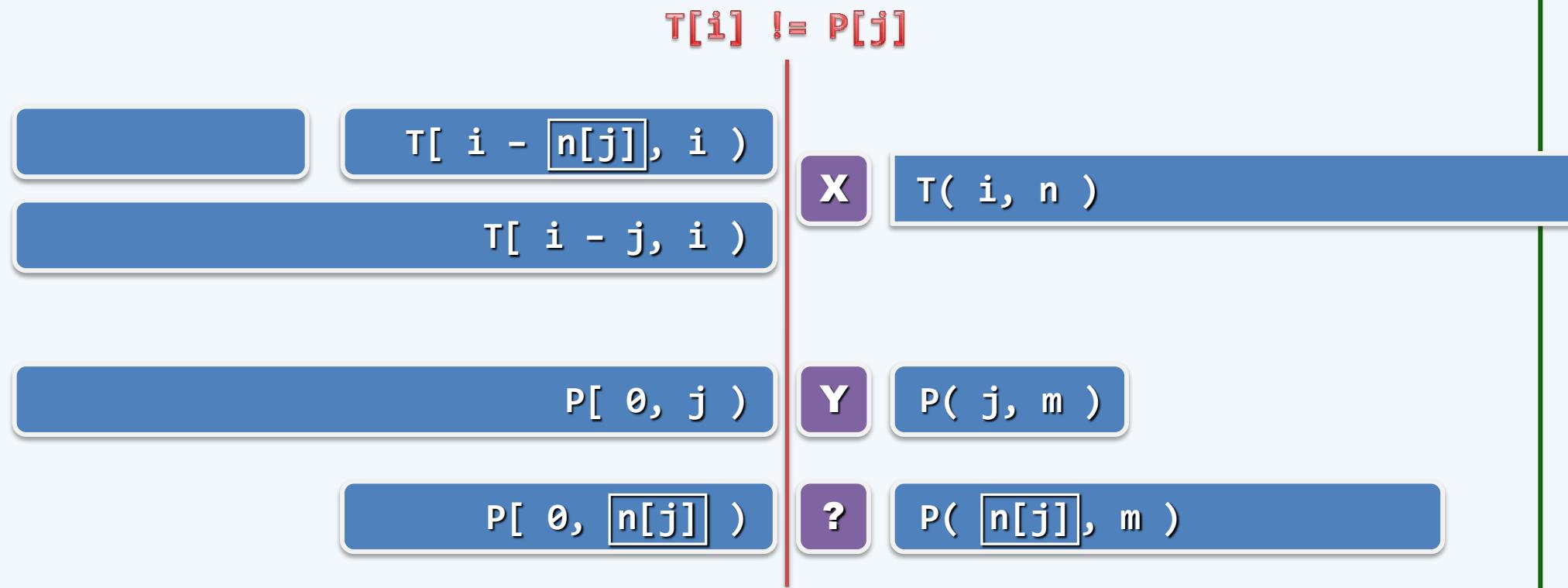
//与 T 无关，仅依赖于 P 的前缀

根据失败位置 $P[j]$ ，无非 m 种情况…



事先确定 t

- 构造查询表 $\text{next}[0, m)$ ：在任一位置 $P[j]$ 处失败之后，将 j 替换为 $\text{next}[j]$
- 与其说是借助强大的记忆，不如说是做好充分的预案



KMP算法

```

❖ int match( char * P, char * T ) {
    int * next = buildNext(P); //构造next表
    int n = (int) strlen(T), i = 0; //文本串指针
    int m = (int) strlen(P), j = 0; //模式串指针
    while ( j < m && i < n ) //自左向右，逐个比对字符
        if ( 0 > j || T[i] == P[j] ) { //若匹配
            i++; j++; //则携手共进
        } else //否则，P右移，T不回退
            j = next[j];
    delete [] next; //释放next表
    return i - j;
}

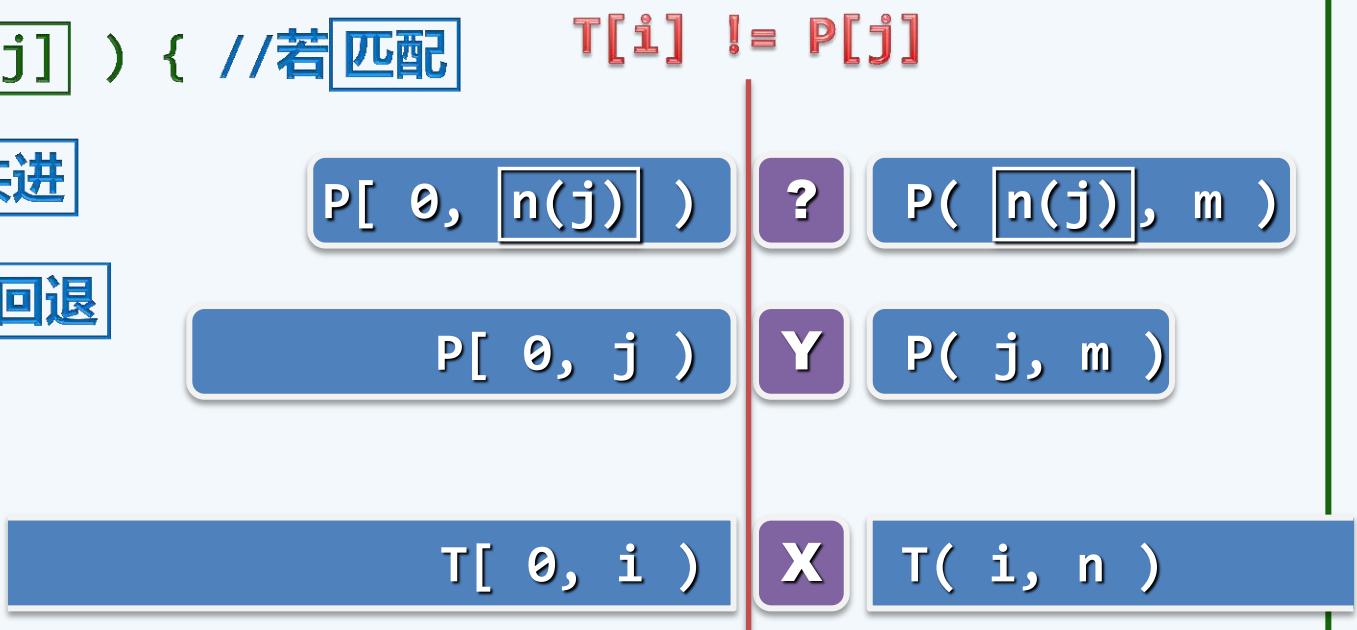
```



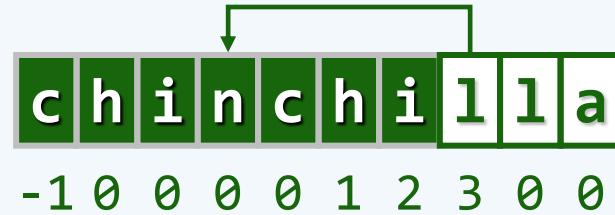
D. E. Knuth

J. H. Morris

V. R. Pratt



实例



自动机

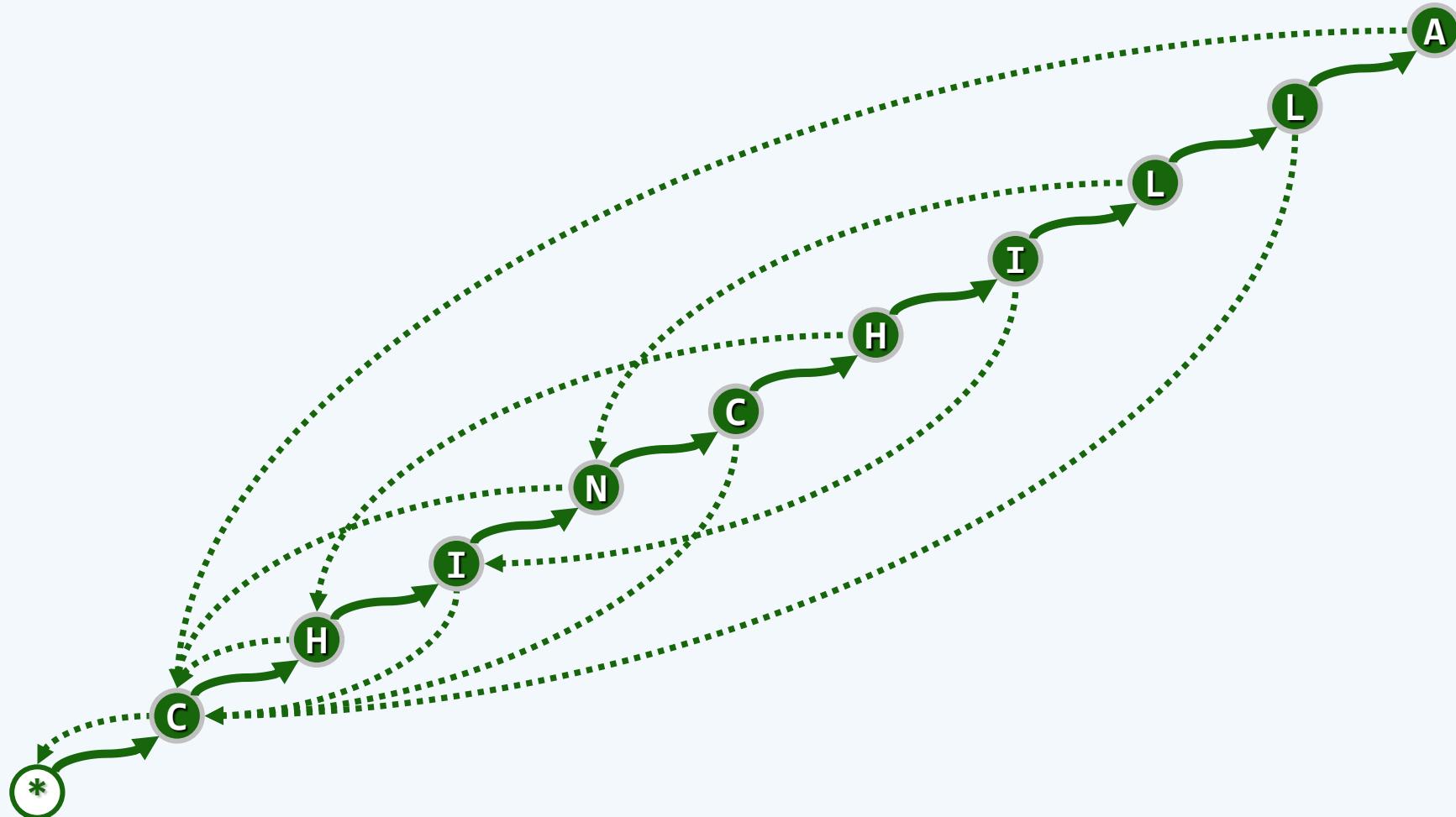
❖ int match(char * T) { //对任一模式串（比如P = chinchilla），可自动生成如下代码

```
int n = strlen(T); int i = -1; //文本串对齐位置
```

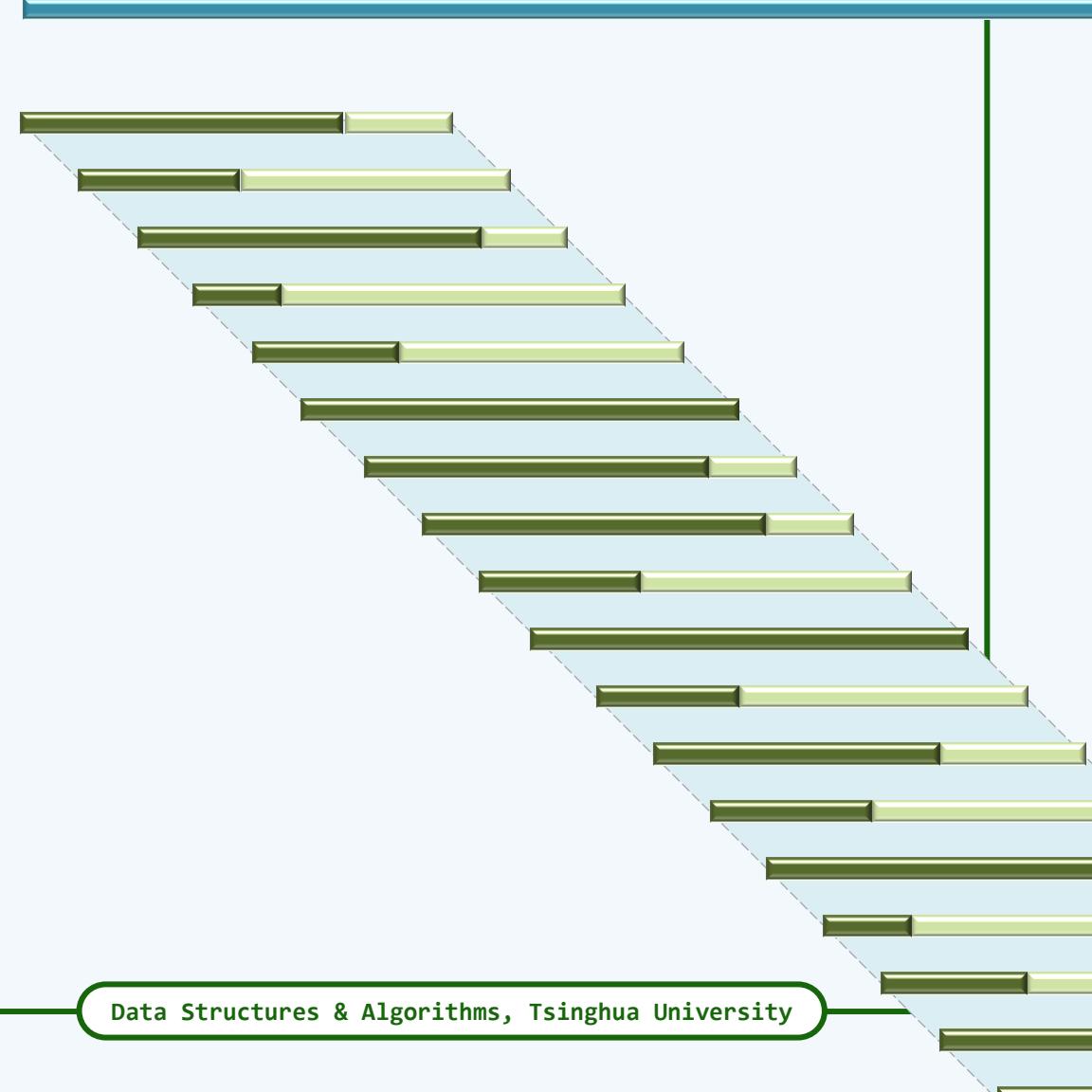
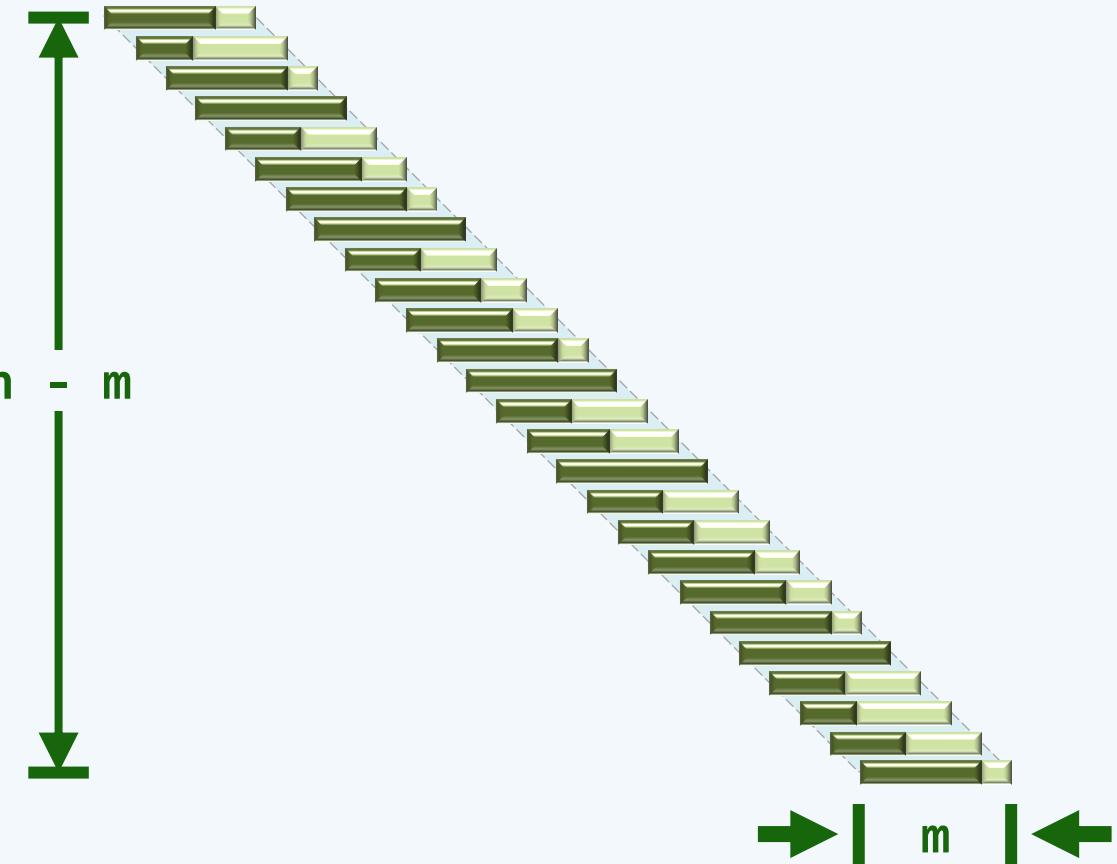
```
s_-: ++i;                                // ↑
s0: (T[i] != 'C') ? goto s_- : if (n <= ++i) return -1; // *      ~ ↑
s1: (T[i] != 'H') ? goto s0 : if (n <= ++i) return -1; // *C     ~ *
s2: (T[i] != 'I') ? goto s0 : if (n <= ++i) return -1; // *CH    ~ *
s3: (T[i] != 'N') ? goto s0 : if (n <= ++i) return -1; // *CHI   ~ *
s4: (T[i] != 'C') ? goto s0 : if (n <= ++i) return -1; // *CHIN  ~ *
s5: (T[i] != 'H') ? goto s1 : if (n <= ++i) return -1; // *CHINC ~ *C
s6: (T[i] != 'I') ? goto s2 : if (n <= ++i) return -1; // *CHINCCH ~ *CH
s7: (T[i] != 'L') ? goto s3 : if (n <= ++i) return -1; // *CHINCCHI ~ *CHI
s8: (T[i] != 'L') ? goto s0 : if (n <= ++i) return -1; // *CHINCCHIL ~ *
s9: (T[i] != 'A') ? goto s0 : if (n <= ++i) return -1; // *CHINCCHILL ~ *
                                         // *CHINCCHILLA
```

}

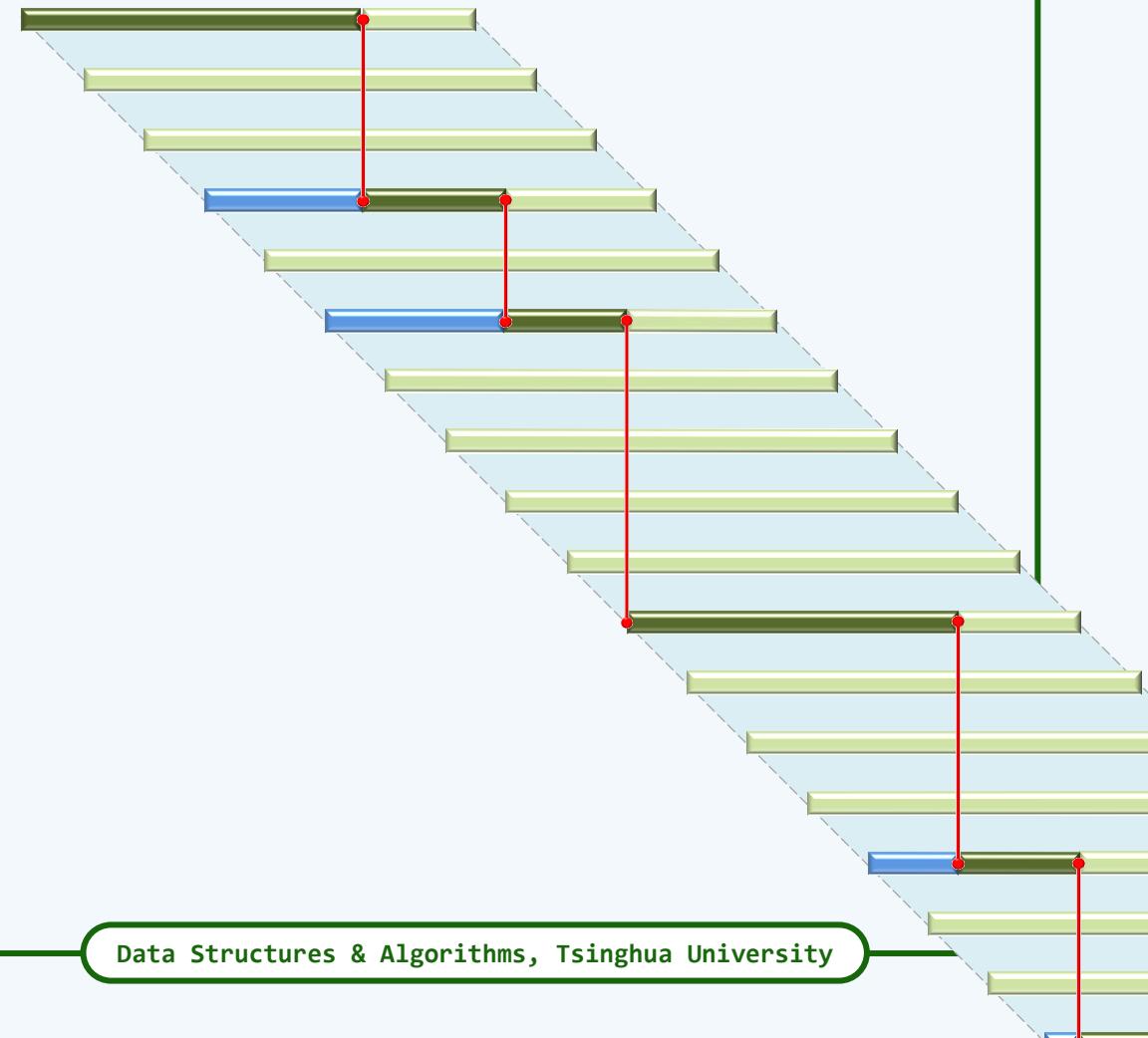
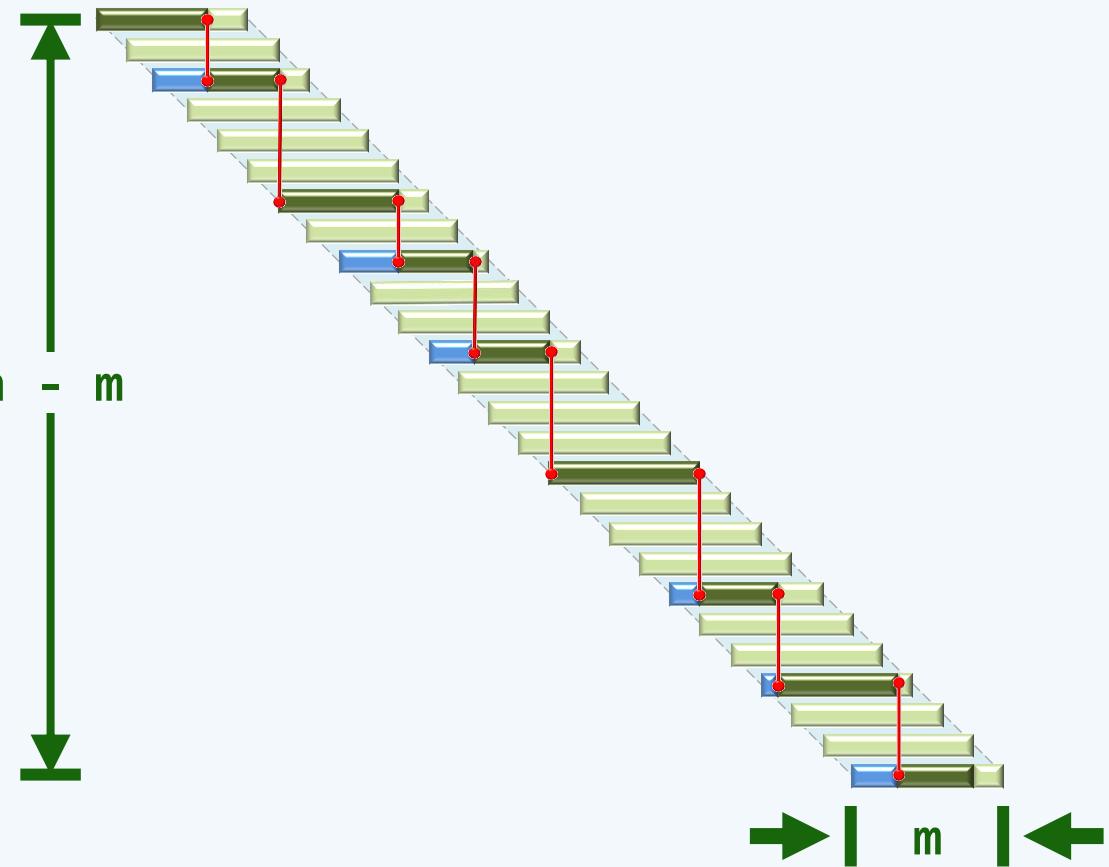
自动机



失败情况 : Brute-force



失败情况 : KMP



11. 串

KMP算法

理解next[]表

吴用再使时迁扮作伏路小军，去曾头市寨中，探听他不出何意，所有陷坑，暗暗地记着，离寨多少路远，总有几处。时迁去了一日，都知备细，暗地使了记号，回报军师。

邓俊辉

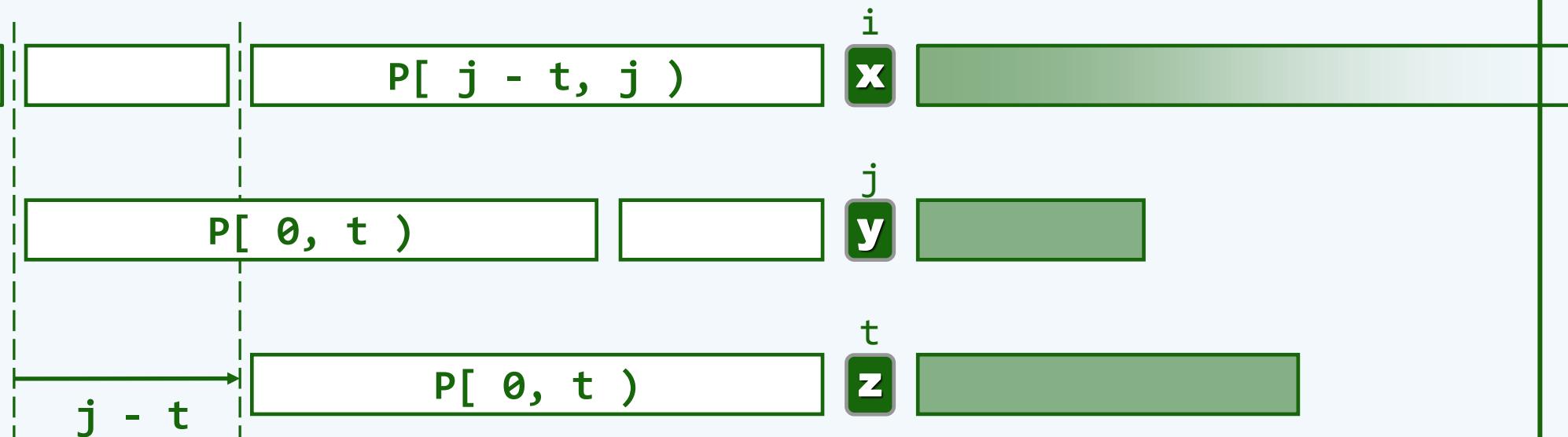
deng@tsinghua.edu.cn

自匹配 = 快速右移

❖ 对任意 j ，考察集合： $N(P, j) = \{ 0 \leq t < j \mid P[0, t] == P[j-t, j] \}$

亦即，在 $P[j]$ 的前缀 $P[0, j]$ 中，所有匹配真前缀和真后缀的长度

❖ 因此，一旦 $T[i] \neq P[j]$ ，可从 $N(P, j)$ 中取某个 t ，令 $P[t]$ 对准 $T[i]$ ，并继续比对

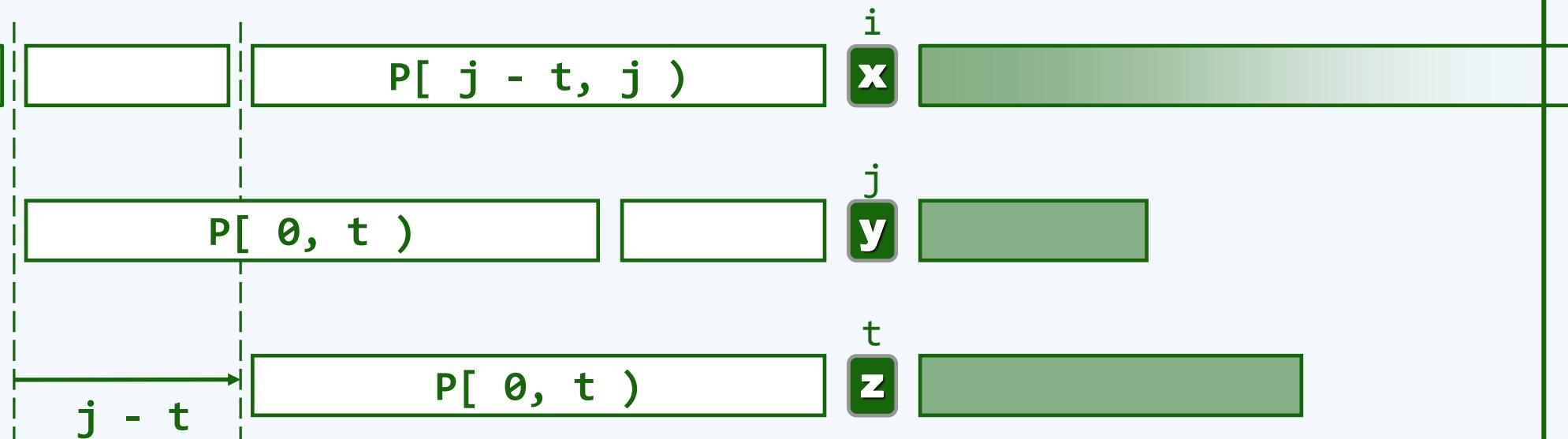


最长自匹配 = 快速右移 + 绝不回退

❖ $|N(P, j)| > 1$ 时，难道需要遍历其中的每一个 t ? 不必 !

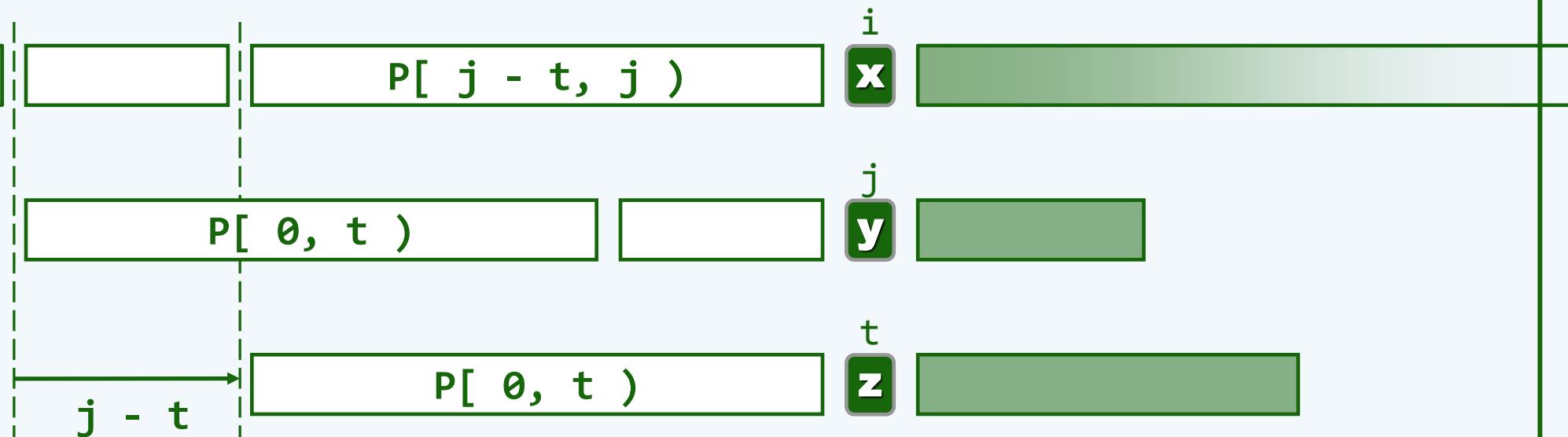
❖ 观察：位移量 = $j - t$ ，与 t 成反比

❖ 因此，若选用最大的 t ，则必然最安全



next[θ]

- ❖ 只要 $j > \theta$, 必有 $\theta \in N(P, j)$ // 空串是任何非空串的真子串
- ❖ 但若 $j = \theta$, 则有 $N(P, \theta) = \emptyset$ // 空串没有真子串
- ❖ 不妨取: $next[\theta] = -1 \dots$ // 回顾主算法: 行之有效! 如何理解?



11. 串

KMP算法

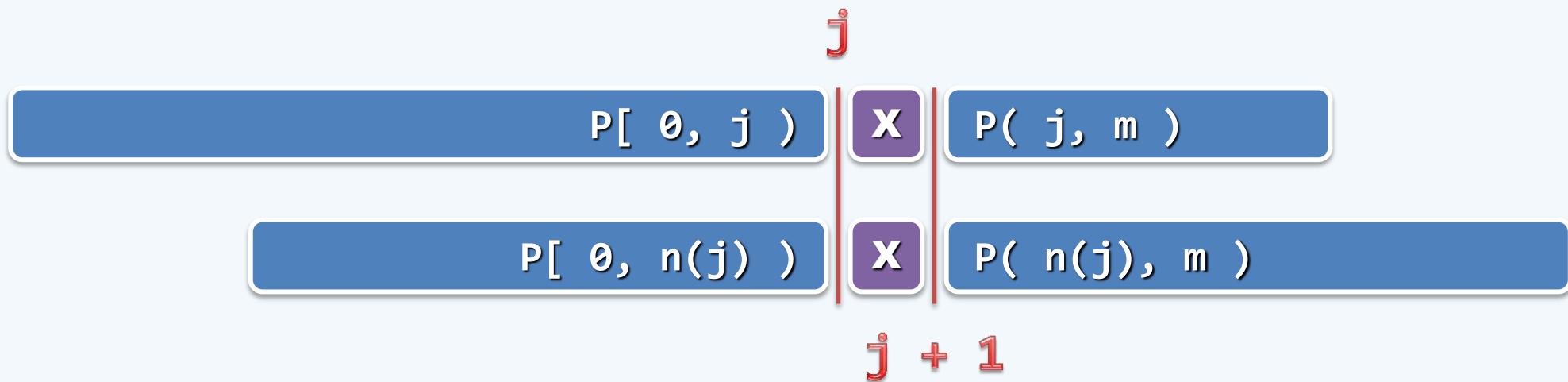
构造next[]表

邓俊辉

deng@tsinghua.edu.cn

递推

- ❖ 根据已知的 $\text{next}[0, j]$ ，如何高效地计算 $\text{next}[j + 1]$ ？
- ❖ 所谓 $\text{next}(j)$ ，即是在 $P[0, j]$ 中，最大自匹配的真前缀和真后缀的长度
- ❖ 故： $\text{next}[j + 1] \leq \text{next}[j] + 1$ //当且仅当 $P[j] == P[\text{next}[j]]$ 时取等号



- ❖ 一般地， $P[j] != P[\text{next}[j]]$ 时，又该如何得到 $\text{next}[j + 1]$ ？

算法

❖ $\text{next}[j + 1]$ 的候选者

依次应该是：

$1 + \boxed{\text{next}[j]}$

$1 + \boxed{\text{next}[\boxed{\text{next}[j]}]}$

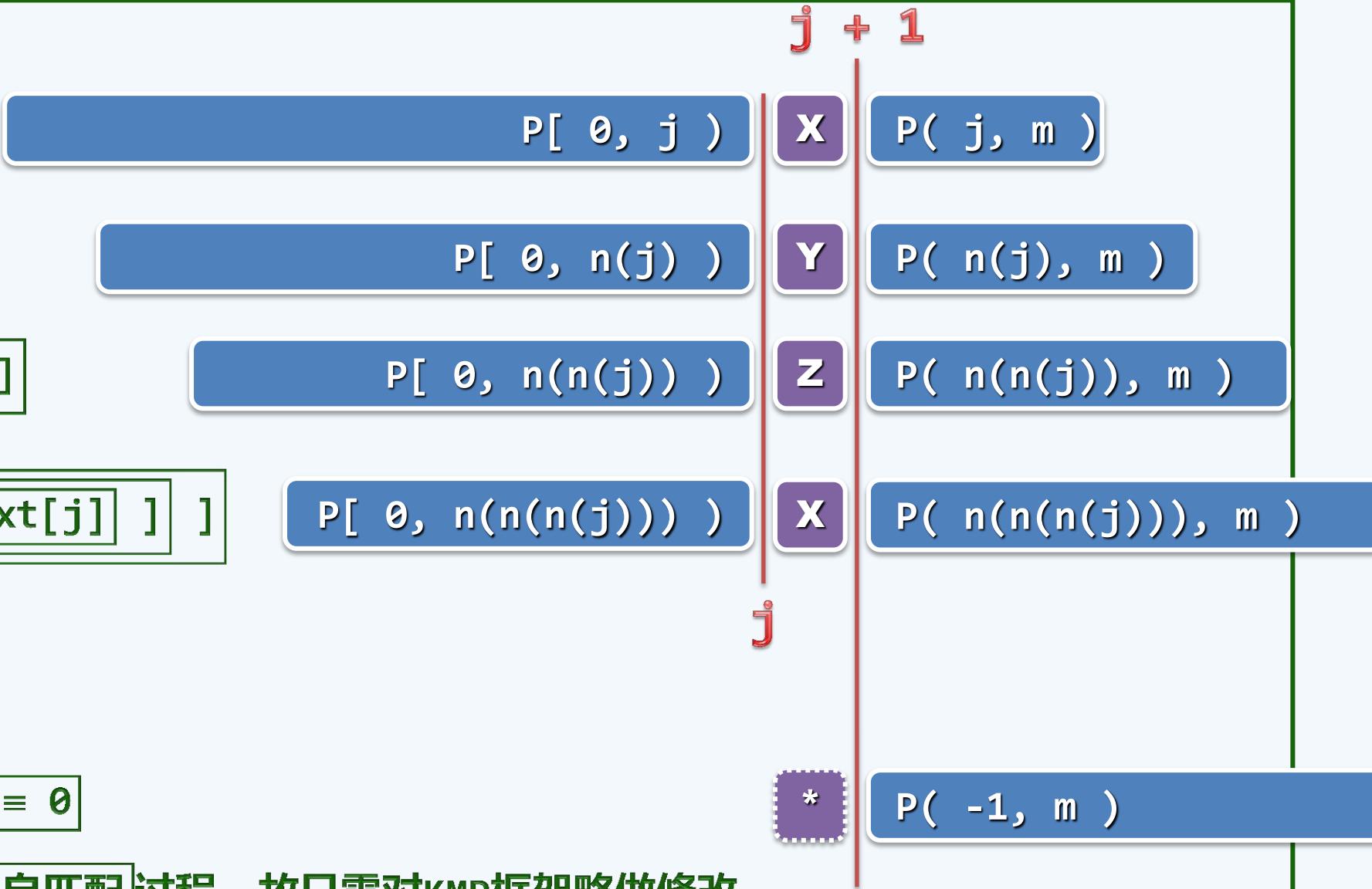
$1 + \boxed{\text{next}[\boxed{\text{next}[\boxed{\text{next}[j]}]}]}$

...

❖ 这个序列严格递减，且

必收敛于 $1 + \text{next}[0] = 0$

❖ 以上递推过程，即是 P 的 **自匹配** 过程，故只需对 KMP 框架略做修改...



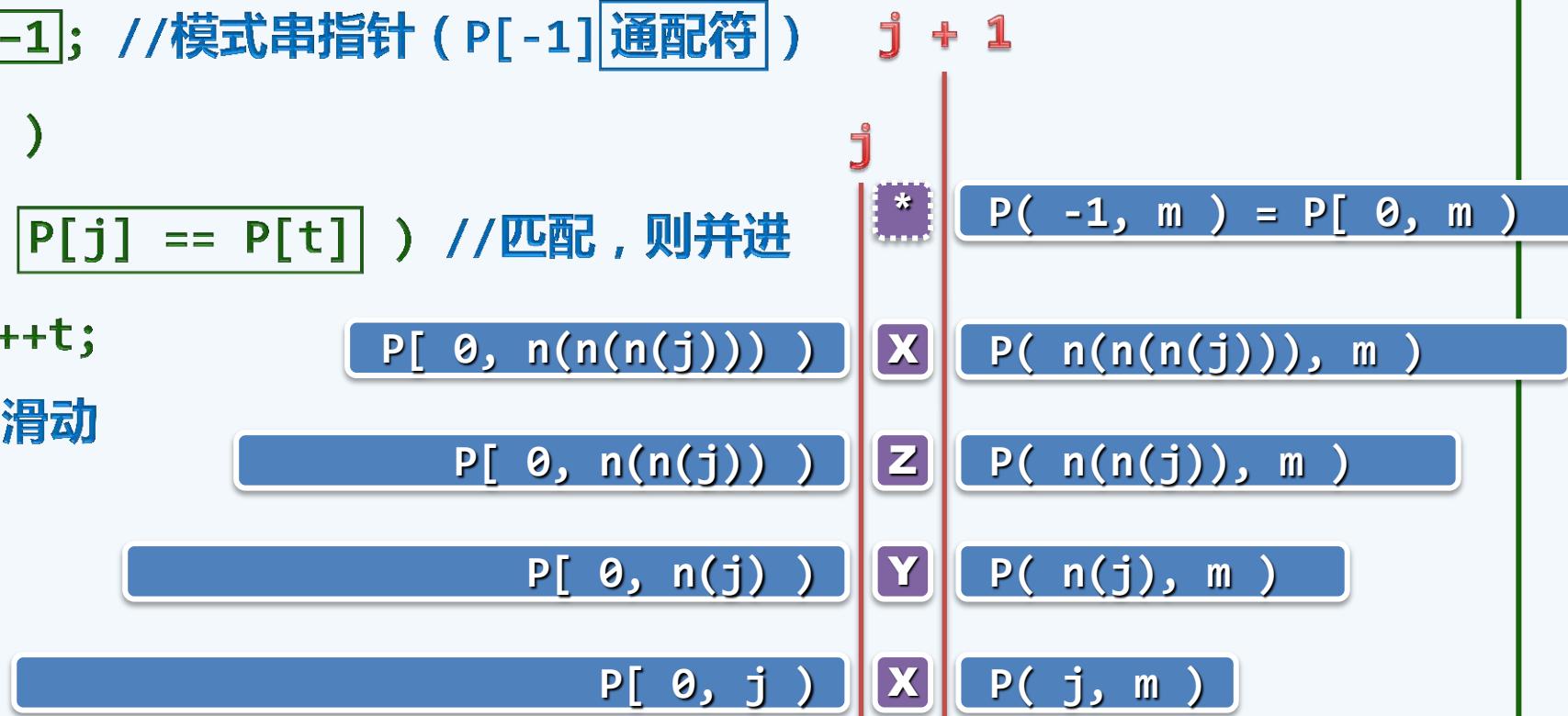
实现

```

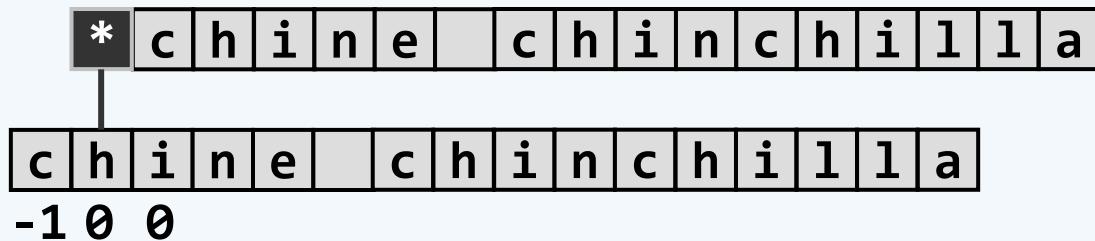
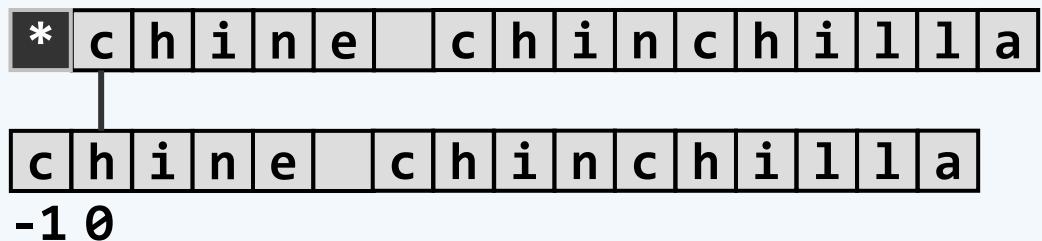
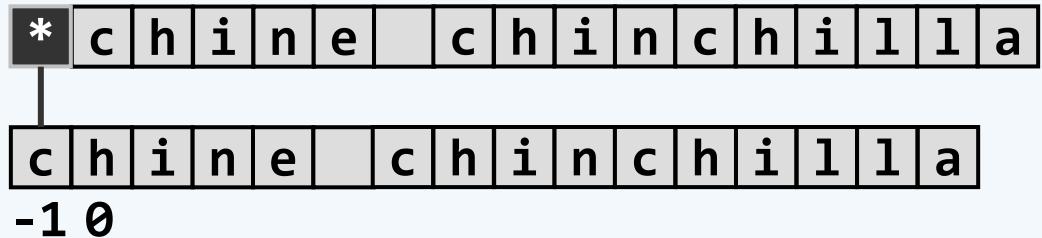
❖ int * buildNext( char * P ) { //构造模式串P的next[]表
    size_t m = strlen(P), j = 0; //“主”串指针
    int * N = new int[m]; //next[]表

    int t = N[0] = -1; //模式串指针 (P[-1]通配符)
    while ( j < m - 1 )
        if ( 0 > t || P[j] == P[t] ) //匹配，则并进
            N[ ++j ] = ++t;
        else //失配，则滑动
            t = N[t];
    return N;
}

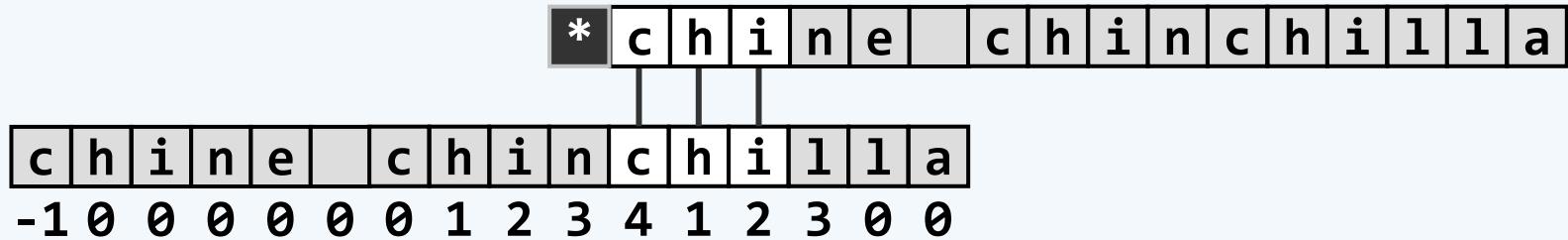
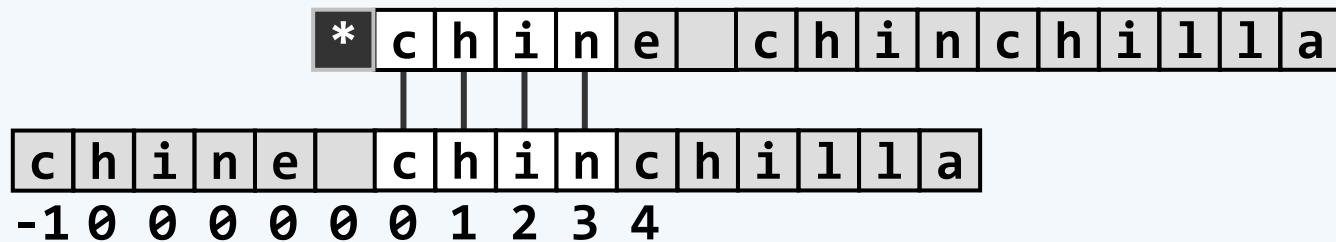
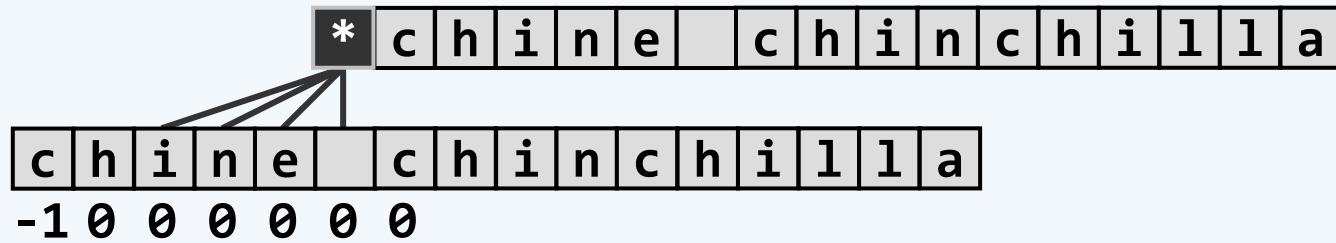
```



实例



实例



11. 串

KMP算法

分摊分析

邓俊辉

失之东隅，收之桑榆

deng@tsinghua.edu.cn

$\Omega(n * m)$?

❖ 观察：KMP算法的确可以节省多次比对

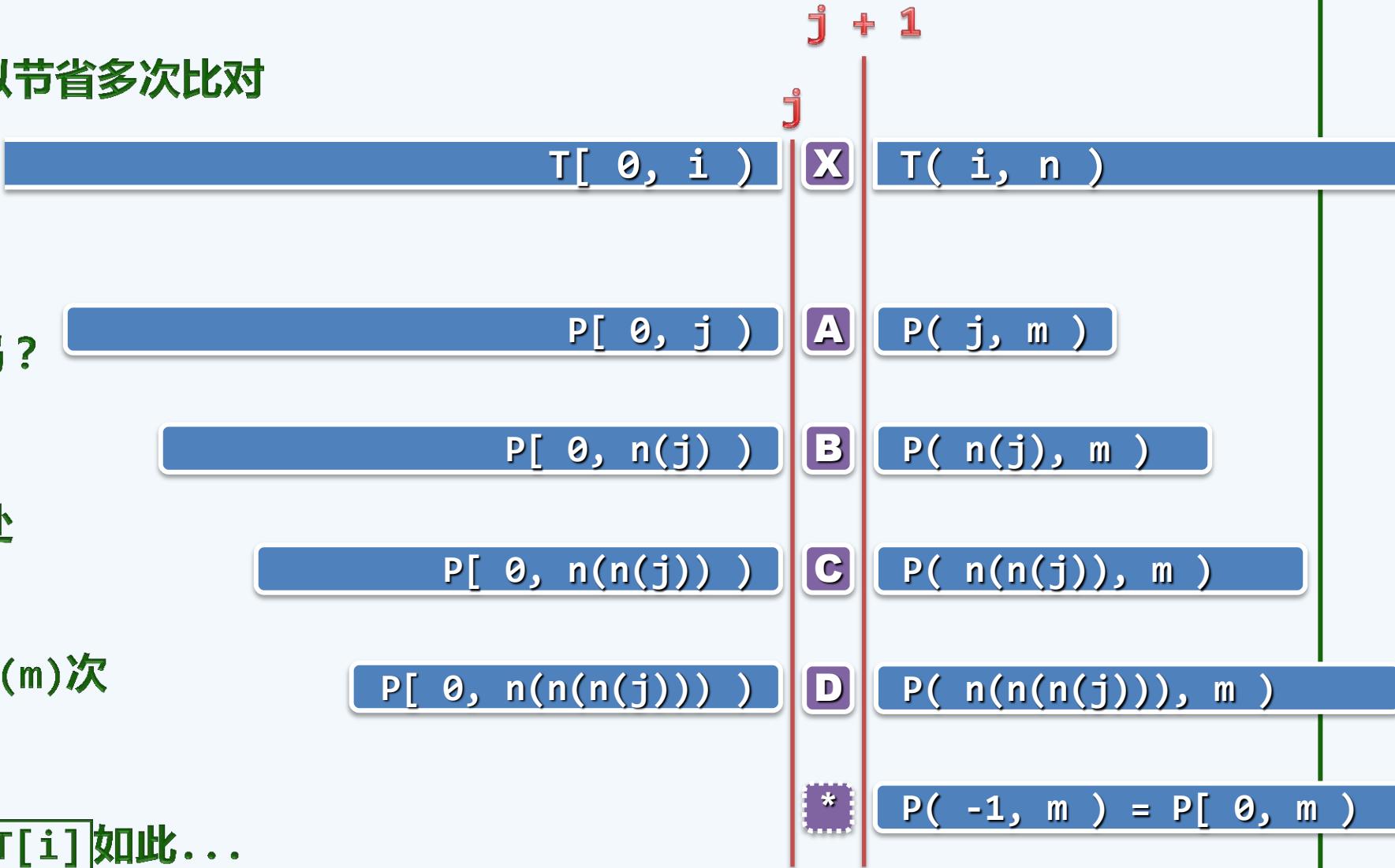
❖ 然而：就渐进意义而言

有**实质**的节省吗？

❖ 观察：在每一个 $T[i]$ 处

P 都**可能**比对 $\Omega(m)$ 次

❖ 于是，倘若有 $\Omega(n)$ 个 $T[i]$ 如此...



$\Omega(n * m)$?

❖ 难道，总体复杂度还是

$\Omega(n * m)$?

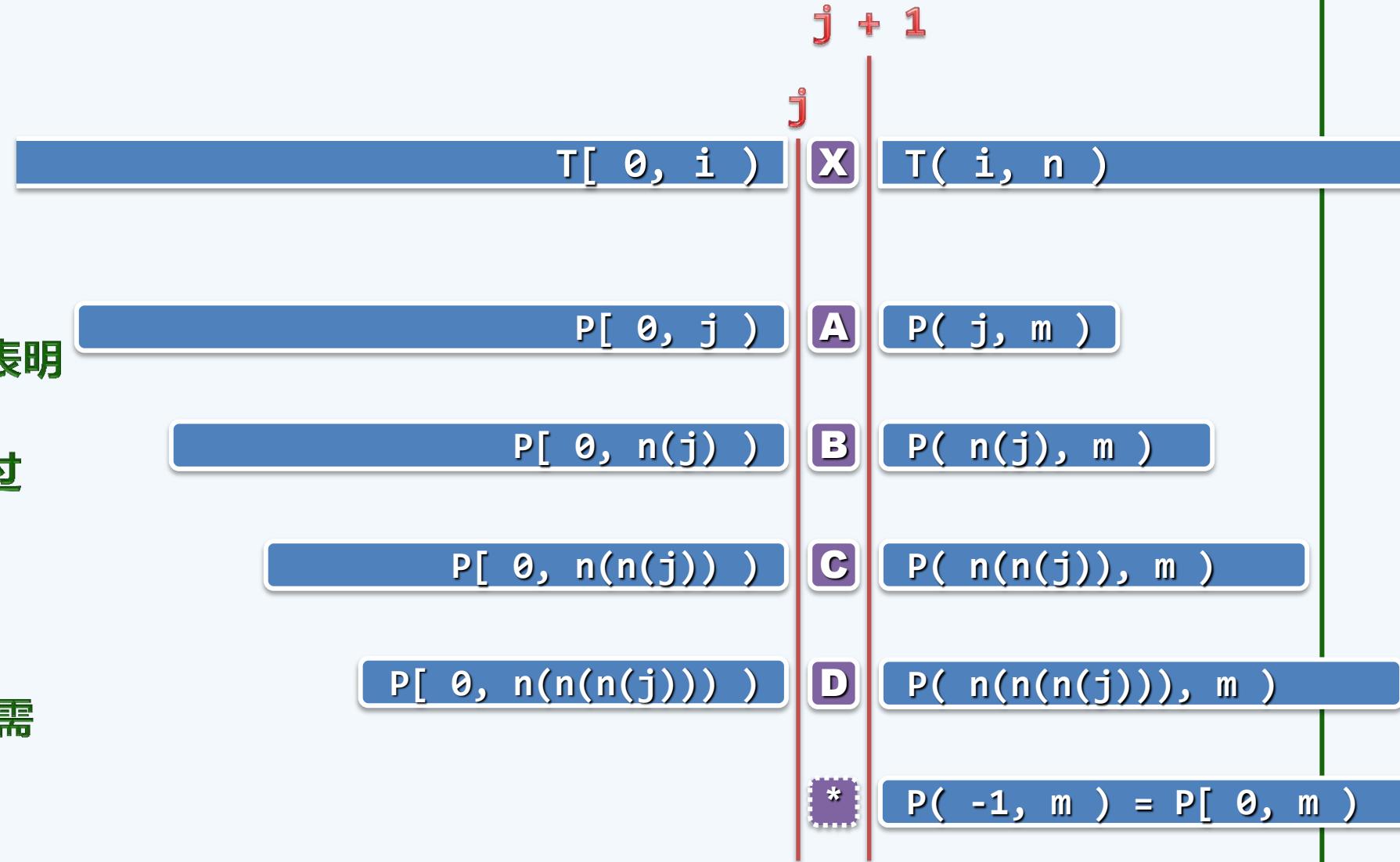
❖ 然而，更细致的分析将表明

即便是最坏情况，也不过

$O(n)$ 时间

❖ 同理，建立`next[]`也只需

$O(m)$ 时间



$\theta(n + m)!$ ❖ 令 $k = 2*i - j$

// 具体含义，详见习题[11-4]

```
while ( j < m && i < n ) // k 必随迭代而单调递增，故也是迭代步数的上界
```

```
if ( 0 > j || T[i] == P[j] )
```

```
{ i++; j++; } // i、j同时加1，故k恰好加1
```

```
else
```

```
j = next[j]; // i不变，j至少减1，故k至少加1
```

❖ k 的初值为 0；算法结束时，必有：

$$k = 2*i - j \leq 2(n - 1) - (-1) = 2n - 1$$

11. 串

KMP算法

再改进

来来 思前想后 差一点忘记了怎么投诉

来来 从此以后 不要犯同一个错误

邓俊辉

前车之覆，后车之鉴

deng@tsinghua.edu.cn

反例

❖ $T = [0\ 0\ 0\ \boxed{1}\ 0\ 0\ 0\ 0\ 1]$

$P = [0\ 0\ 0\ \boxed{0}\ 1]$

❖ $T[\boxed{3}] :$

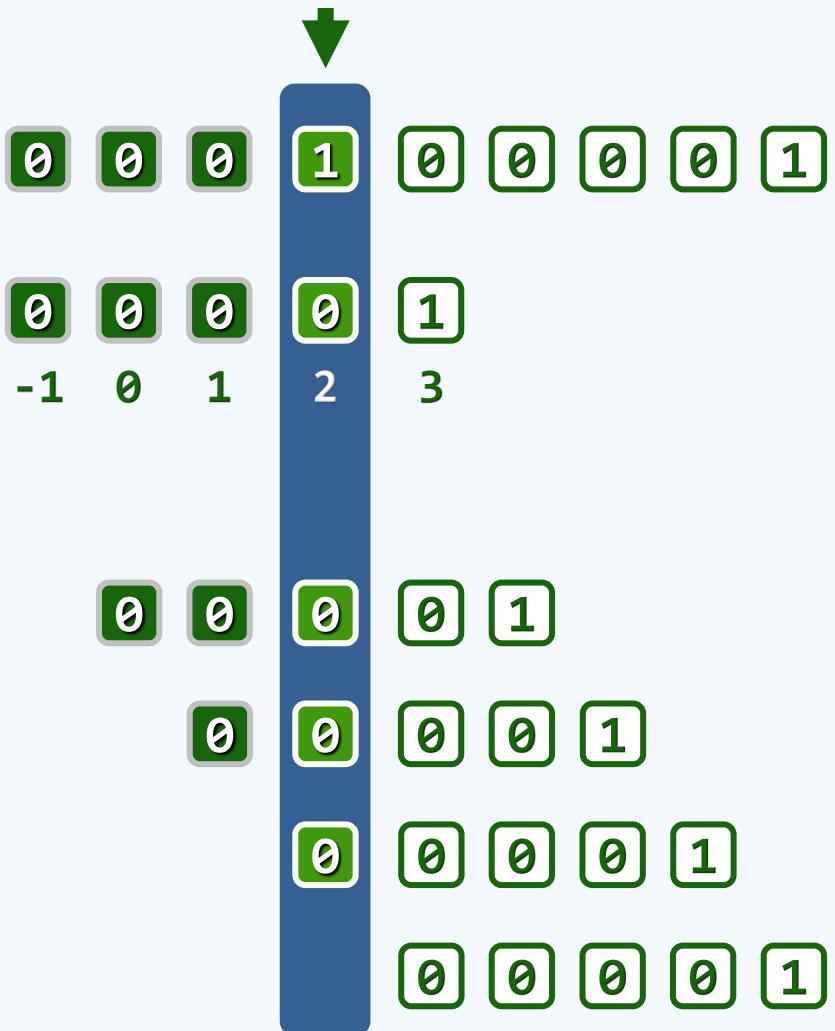
与 $P[\boxed{3}]$ 比对，失败

与 $P[\boxed{2}] = P[\boxed{\text{next}[3]}]$ 继续比对，失败

与 $P[\boxed{1}] = P[\boxed{\text{next}[2]}]$ 继续比对，失败

与 $P[\boxed{0}] = P[\boxed{\text{next}[1]}]$ 继续比对，失败

最终，才前进到 $T[\boxed{4}]$



根源

❖ 无需T串，即可在事先确定：

$P[3] =$

$P[2] =$

$P[1] =$

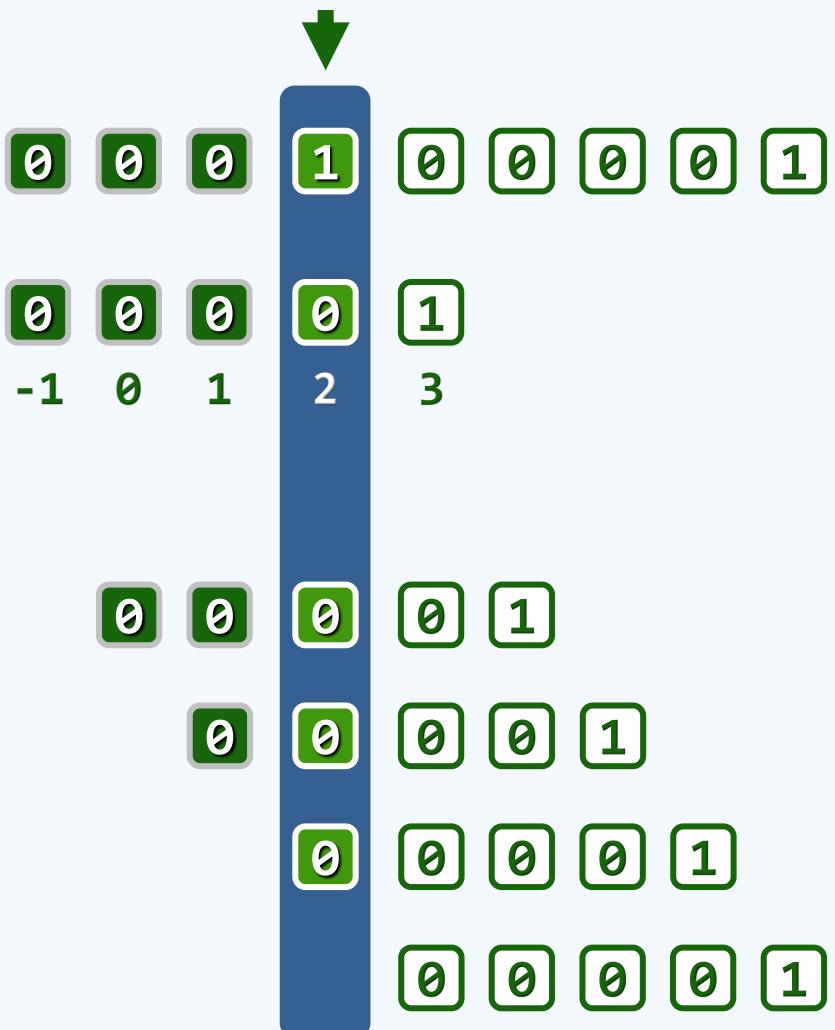
$P[0] = \boxed{0}$

既然如此...

❖ 在发现 $T[3] != P[3]$ 之后

为何还要一错再错？

❖ 事实上，后三次比对本来都是可以避免的！

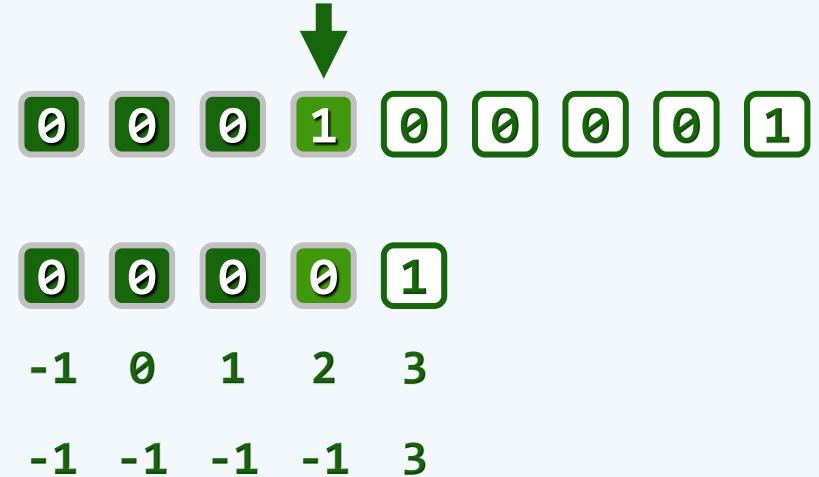


改进

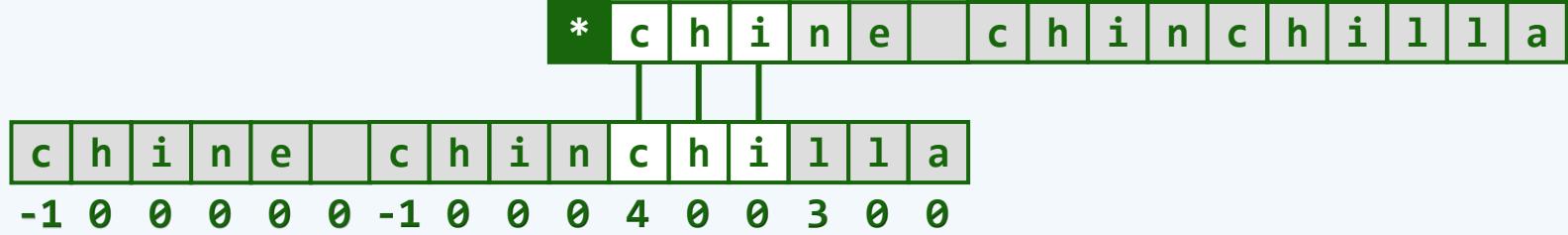
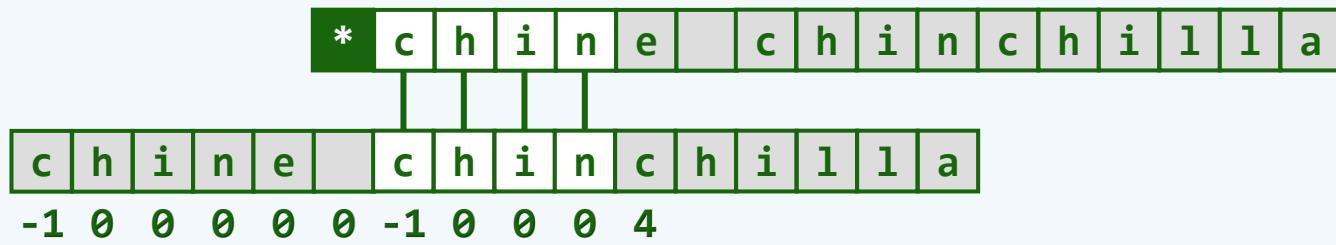
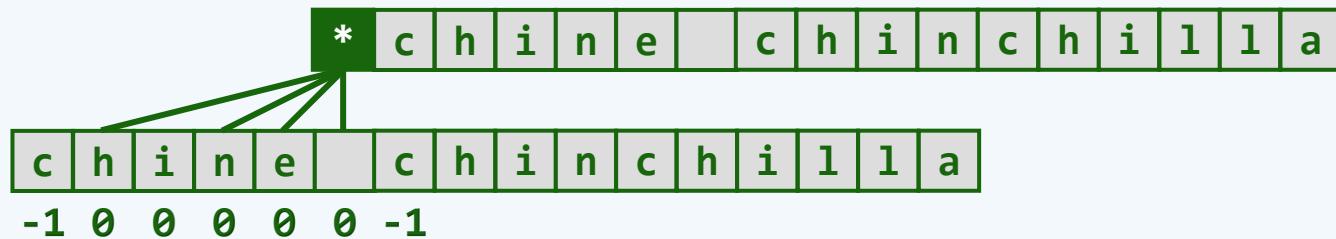
```

❖ int * buildNext( char * P ) {
    size_t m = strlen(P), j = 0; //“主”串指针
    int * N = new int[m]; //next表
    int t = N[0] = -1; //模式串指针
    while ( j < m - 1 )
        if ( 0 > t || P[j] == P[t] ) { //匹配
            j++; t++; N[j] = P[j] != P[t] ? t : N[t];
        } else //失配
            t = N[t];
    return N;
}

```



实例



小结

❖ 充分利用以往的比对所提供的信息

模式串快速右移，文本串无需回退

❖ 经验：以往成功的比对 —— $T[i - j, i)$ 是什么

教训：以往失败的比对 —— $T[i]$ 不是什么

❖ 特别适用于顺序存储介质

❖ 单次匹配概率越大 ($|\Sigma|$ 越小) 的场合，优势越明显

// 比如二进制串

否则，与蛮力算法的性能相差无几...

11. 串

BM算法：BC策略

以终为始

邓俊辉

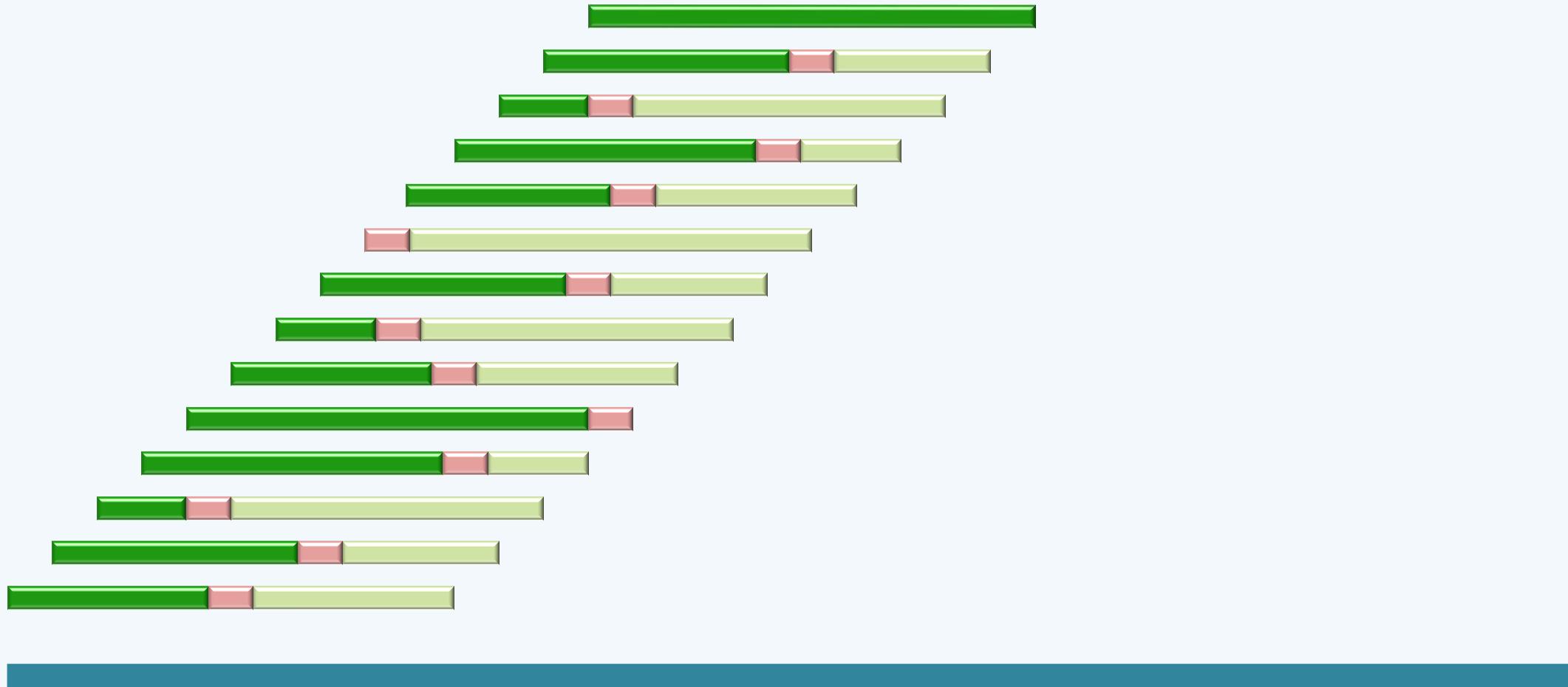
Begin with the end in mind.

deng@tsinghua.edu.cn

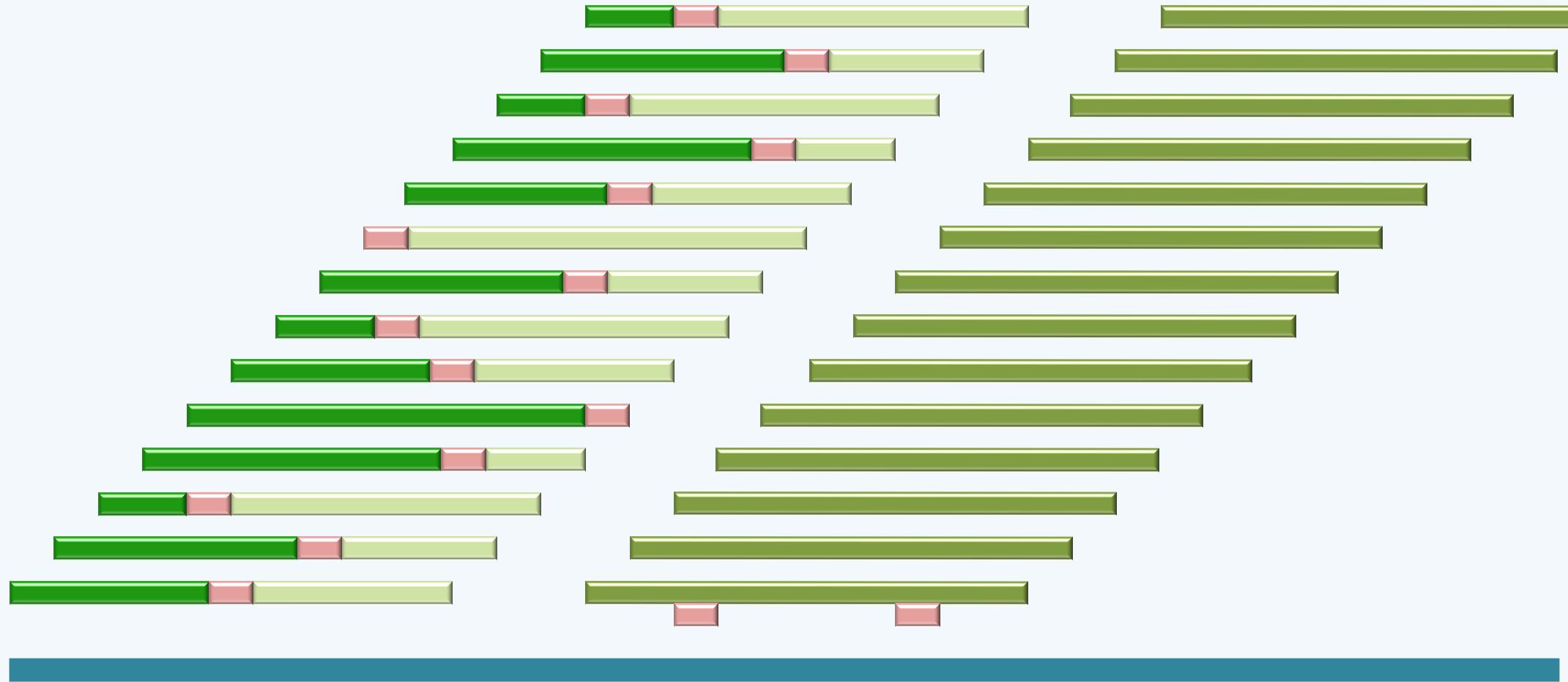
经验 + 教训

- ❖ 串匹配 = x 次失败 的对齐 + 0/1次成功 的对齐
- ❖ 与其说要加速匹配，不如说是加速失败 —— 尽快排除失败的对齐
- ❖ 就单个对齐位置的排除而言
 - 平均仅需常数次比对（只要 $|\Sigma|$ 不致太小，单次比对成功概率足够低）
 - 且具体的比对位置及次序无所谓
- ❖ 然而就排除更多后续对齐位置而言
 - 不同的对比位置及次序，作用差异极大
- ❖ 通常，越是靠前/后的位置，作用越小/大

善待教训



前轻后重



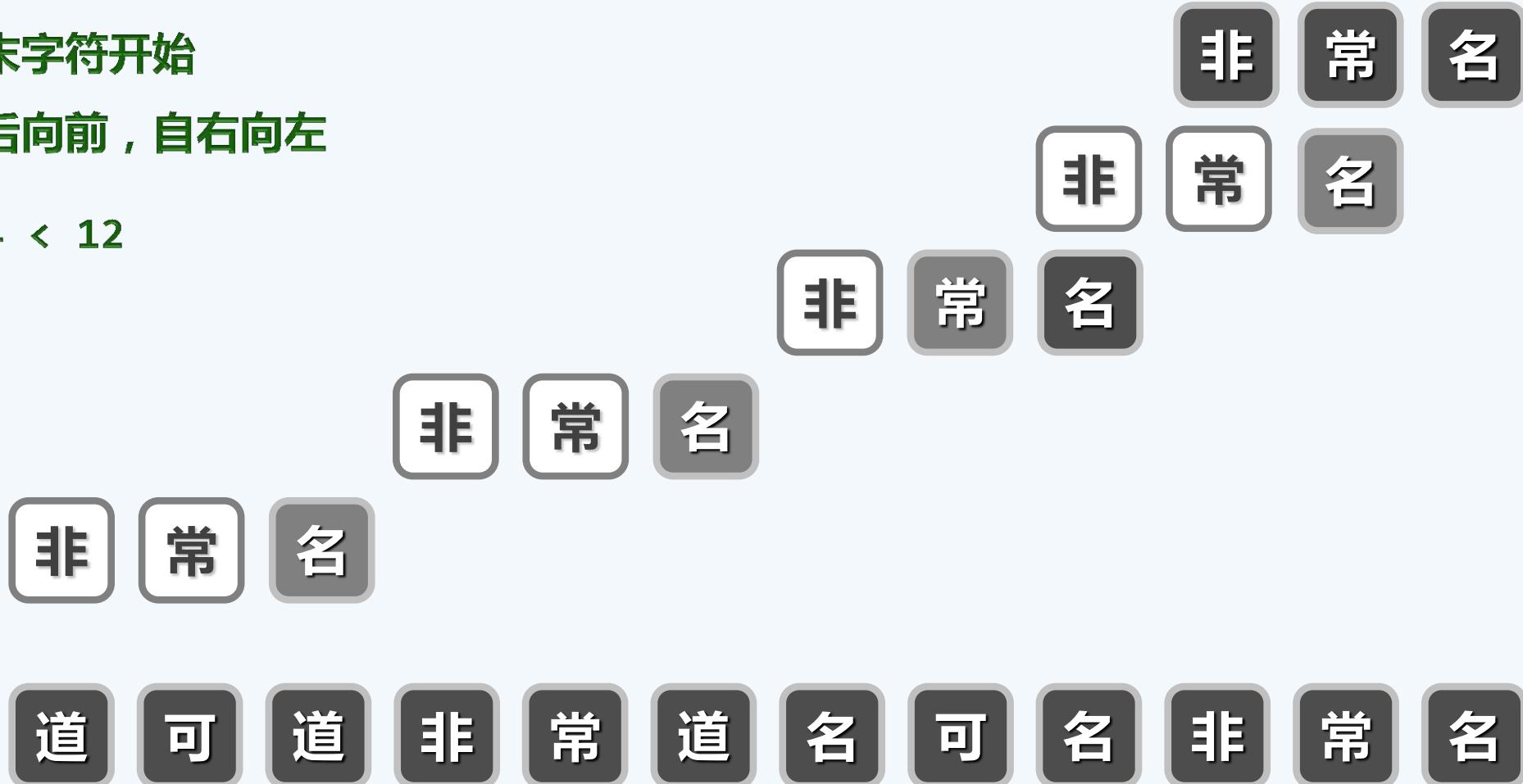
以终为始

❖ 既如此，每一趟比对都更应该

从未字符开始

自后向前，自右向左

❖ $4 + 4 < 12$



以终为始

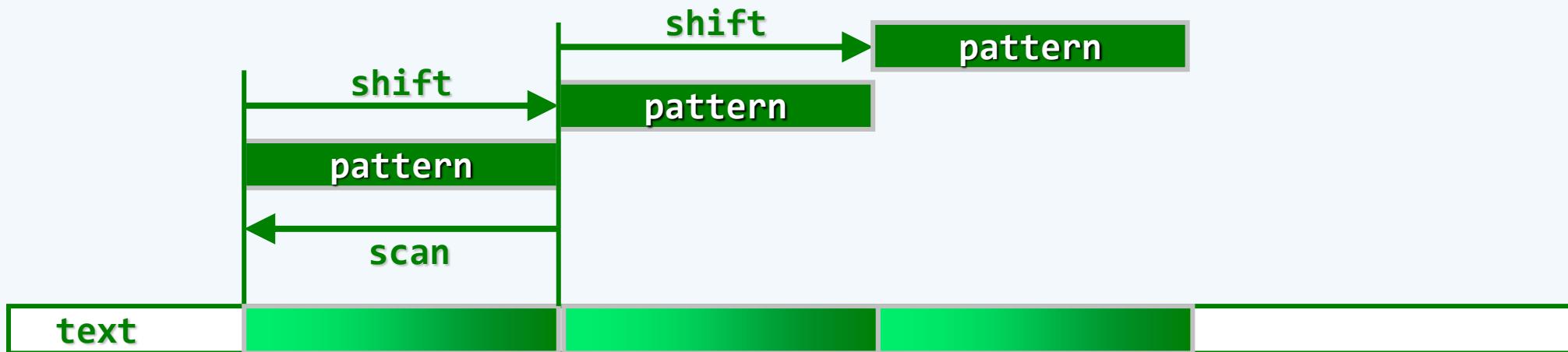
❖ [Boyer + Moore, 1977] A fast string searching algorithm

❖ 预处理：根据模式串P，预先构造 $gs[]$ 表和 $bc[]$ 表

迭代：自右向左依次比对字符，找到极大的匹配后缀

若完全匹配，则返回位置

否则，根据 $gs[]$ 和 $bc[]$ ，P适当右移，并重新自右向左比对



11.串

BM算法：BC策略

坏字符

墩儿了个墩儿啊	大了棉地墩儿啊
怎么这么厚啊？	亲娘絮的呀
后娘当的呀	当多少钱啊？
五六毛啊	一块一啊
我给你两块二呀	我找你一块一呀
我给你三块八呀	我找你两块七呀
我给你一百八十六块三毛二分五哇	我留下一块一啊，剩下全给你呀！

邓俊辉

deng@tsinghua.edu.cn

Bad-Character

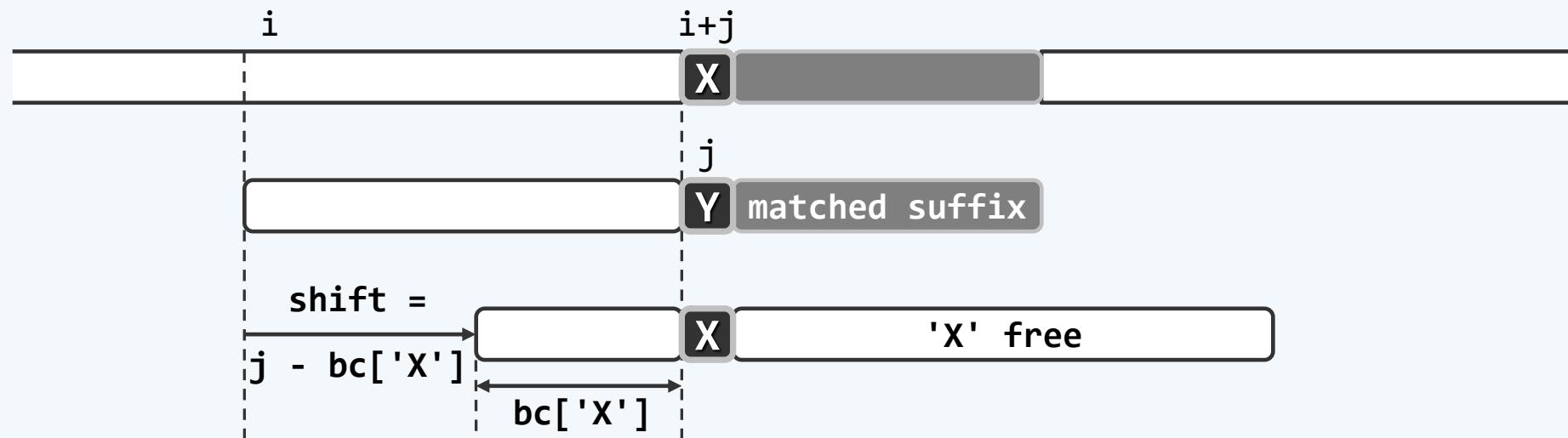
❖ 某趟扫描中，一旦发现

$$T[i + j] = X \neq Y = P[j]$$

// Y 称作坏字符

则 P 相应地右移，并启动新一轮扫描比对

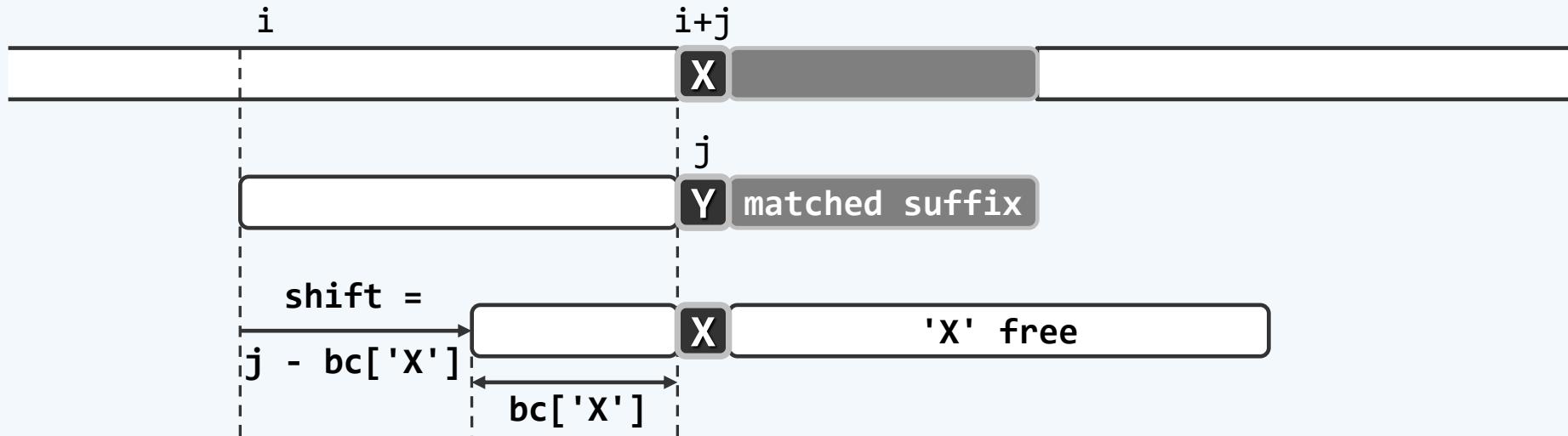
// 具体如何右移？



❖ 必要条件：至少坏字符本身应得以恢复匹配——因此...

Bad-Character Shift

❖ 只需：找出P中的 \boxed{X} ，使之与 $T[i + j] = \boxed{X}$ 对准，并做新一轮比对



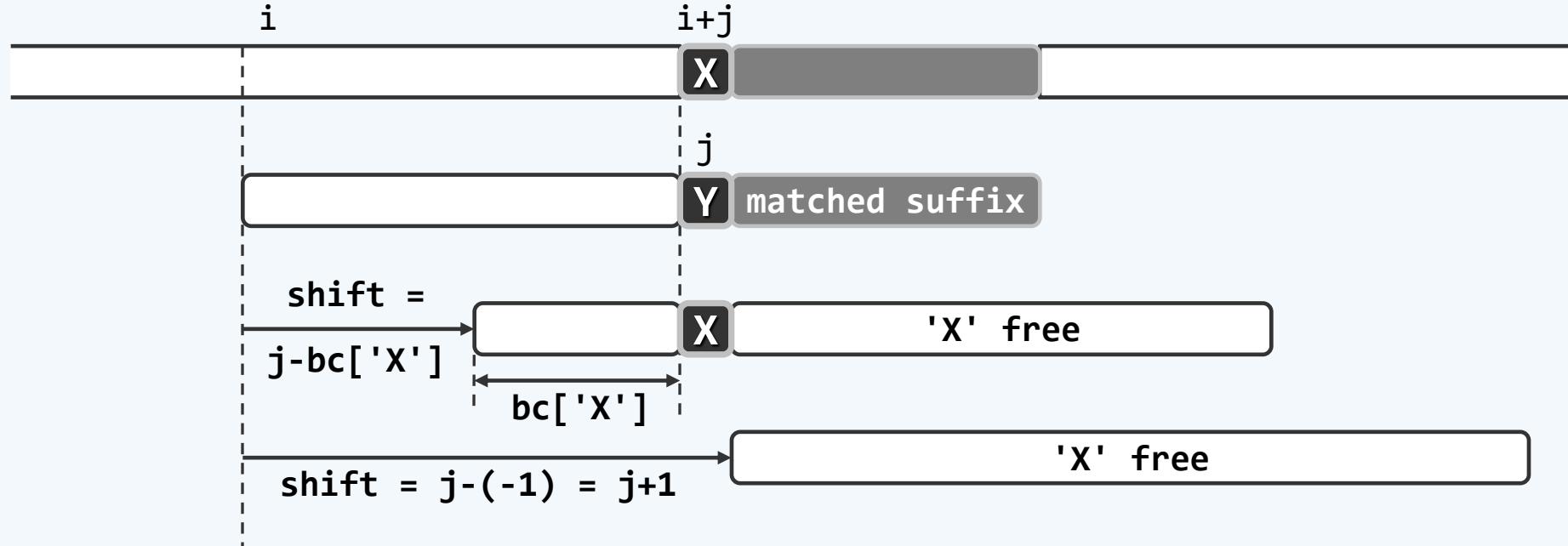
❖ 注意：位移量取决于失配位置 j ，以及 \boxed{X} 在P中的 $\boxed{\text{秩}}$ ，而与 T 和 i 无关！ //与KMP类似

❖ 若：令 $\text{bc}[\boxed{X}] = \text{rank}[\boxed{X}] = j - \text{shift}$

则： $\text{bc}[]$ 总计有 $s = |\Sigma|$ 项，且可事先计算，并制表待查

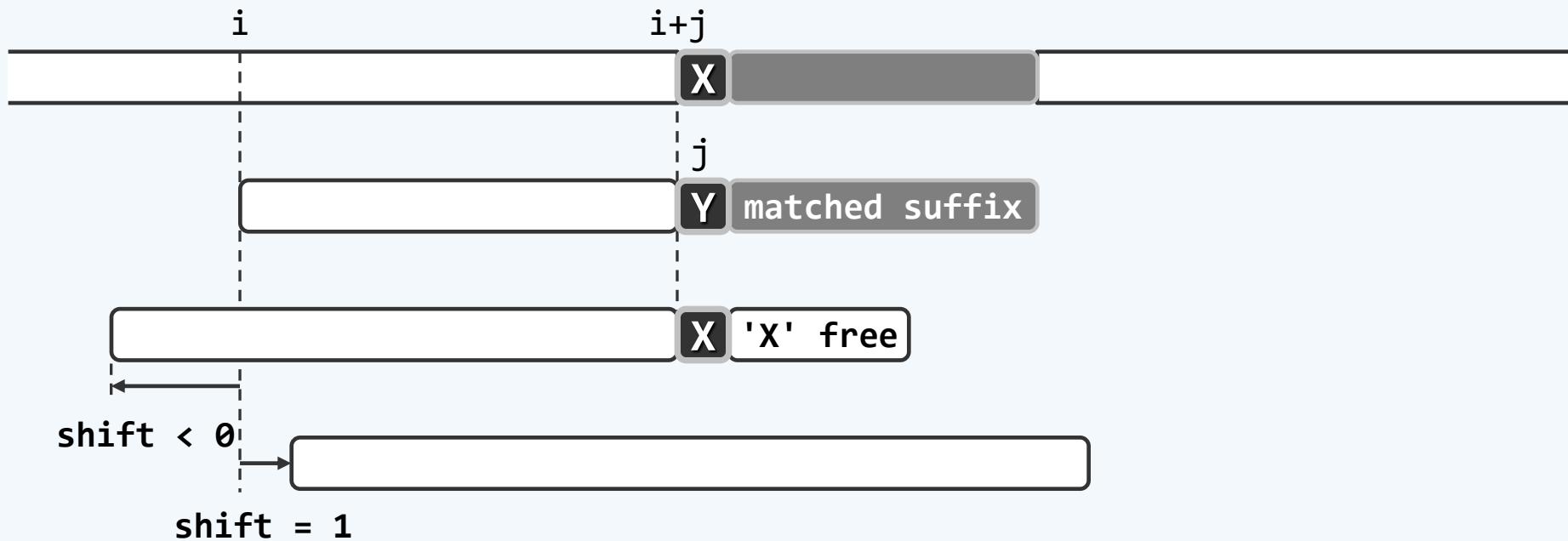
Bad-Character Shift

- 若P包含多个X，取其中最靠右者 //如此可确保无需回溯
- 若P不含任何X，将P整体移过T[i + j] //与KMP类似，这等效于，与假想的通配符对齐

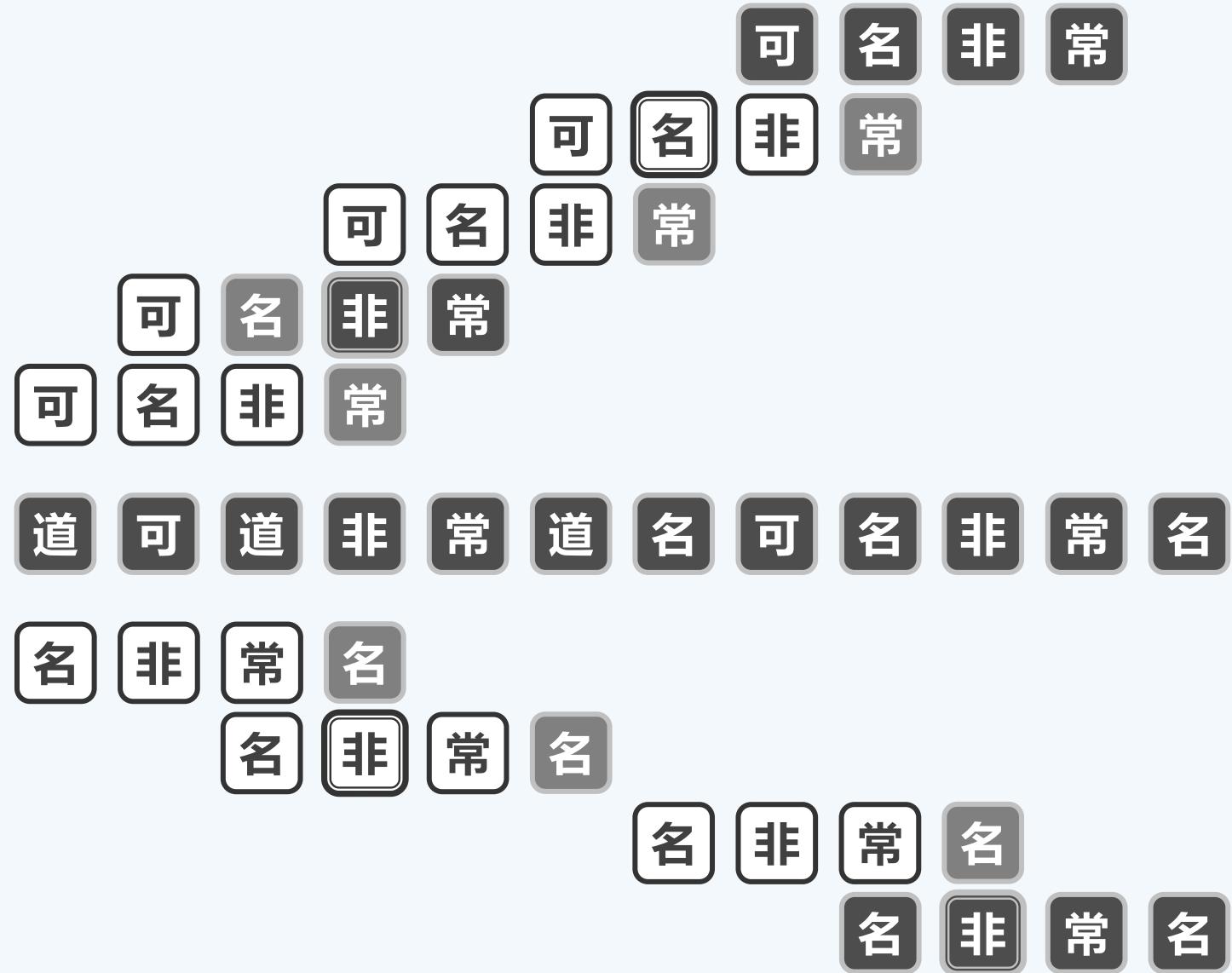


Bad-Character Shift

- 即使'x'在P中出现，但最右侧者过于靠右（以至于位移量为负）呢？
- 将P[]右移一个字符！
- 为何...不选用在P[j]左侧并与之最近的那个X？



实例



11. 串

BM算法：BC策略

构造bc[]

邓俊辉

deng@tsinghua.edu.cn

画家算法

```

❖ int * buildBC( char * P ) {

    int * bc = new int[ 256 ]; //bc[]表，与字母表等长

    for ( size_t j = 0; j < 256; j++ ) bc[j] = -1; //初始化（统一指向通配符）

    for ( size_t m = strlen(P), j = 0; j < m; j++ ) //自左向右扫描
        bc[ P[ j ] ] = j; //刷新P[j]的出现位置记录（painter：后来覆盖以往）

    return bc;
}

} //第二个循环，通过引入临时变量m，避免反复调用strlen()

```

❖ 附加空间 = | bc[] | = $\Theta(|\Sigma|)$ = $\Theta(s)$

❖ 时间 = $\Theta(|\Sigma| + m)$ = $\Theta(s + m)$ //习题解析：可改进至 $\Theta(m)$

11. 串

BM算法：BC策略

性能分析

邓俊辉

deng@tsinghua.edu.cn

最好情况

❖ $\theta(n / m)$ —— 除法？没错！

❖ 比如： $T = \boxed{x \ x \ x \ x \ 1} \ \boxed{x \ x \ x \ x \ 1} \ \boxed{x \ x \ x \ x \ 1} \ \dots \ \boxed{x \ x \ x \ x \ 1} \ \boxed{x \ x \ x \ x \ 1}$

$P = \boxed{0 \ 0 \ 0 \ 0 \ 0}$

❖ 一般地：只要 P 不含 $T[i+j]$ ，即可直接移动 m 个字符

仅需单次比较，即可排除 m 个对齐位置

❖ 单次匹配概率越小，性能优势越明显

// 大字母表：ASCII + Unicode

❖ P 越长，这类移动的效果越明显

最差情况

❖ 最坏 = $\Theta(n \times m)$ —— 退化为蛮力算法？是的！

❖ 比如：T = 

P = 

❖ 每轮迭代，都要在扫过整个P之后，方能确定右移一个字符

此时，须经m次比较，方能排除单个对齐位置

❖ 单次匹配概率越大的场合，性能越接近于蛮力算法 // 小字母表Bitmap + DNA

❖ 反思：借助以上bc表，仅仅利用了失配比对提供的信息（教训）！

类比：可否仿照KMP，同时利用起匹配比对提供的信息（经验）？

11.串

BM算法：GS策略

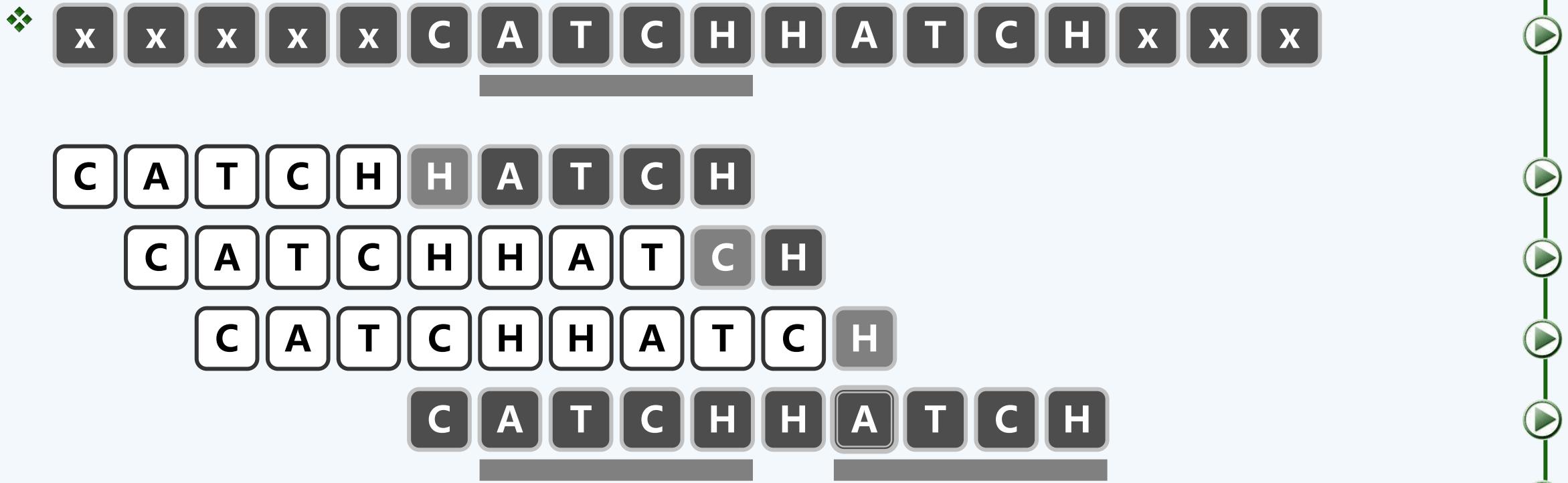
好后缀

Wrong cannot afford defeat,
but Right can.

邓俊辉

deng@tsinghua.edu.cn

经验 = 匹配的后缀



首趟比对虽以失败结束，却积累了足够的**经验**（**匹配的后缀****ATCH**） //好后缀

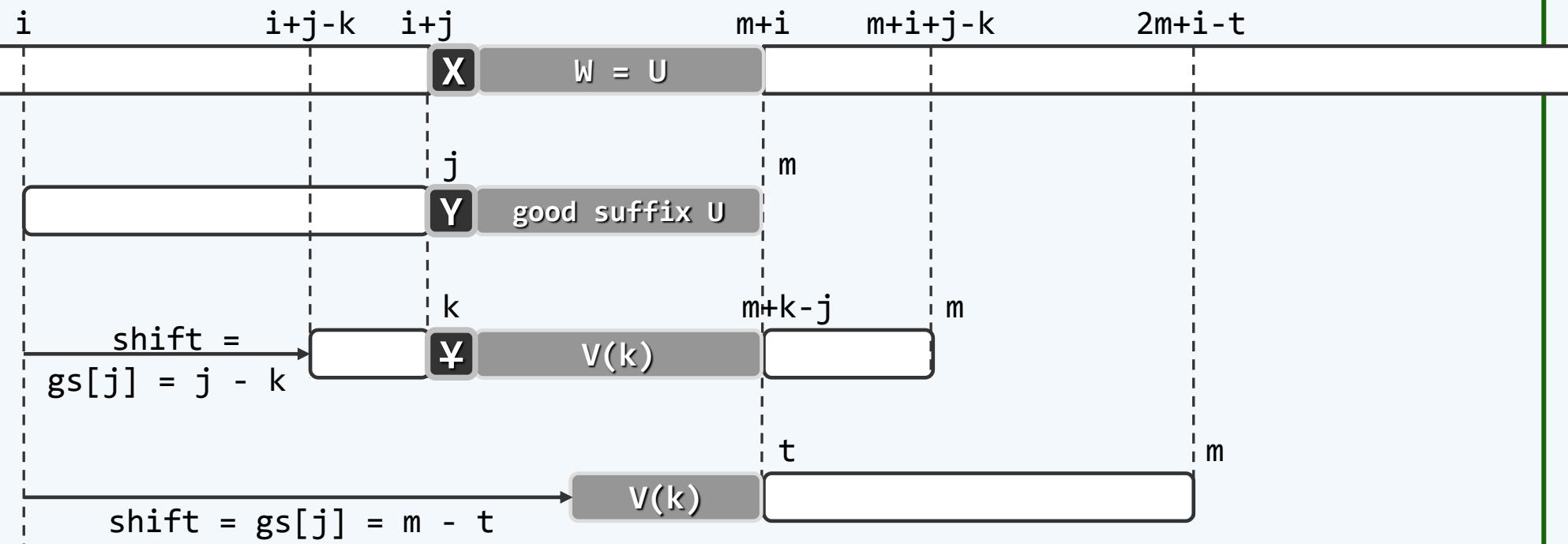
❖ 据此，可以省去中间的两趟，而**直接**转至最后一趟（P右移5个字符）

❖ 这一规律与技巧，与**KMP**完全一致，只不过**前后颠倒而已...**

Good-Suffix Shift

扫描比对中断于 $T[i + j] = \text{X} \neq \text{Y} = P[j]$ 时， $U = P(j, m)$ 必为好后缀

故下一对齐位置必须使：
 1) U 重新与 $V(k) = P(k, m + k - j)$ 匹配，且 //经验
 2) $P[k] = \text{Y} \neq \text{Y} = P[j]$ //教训

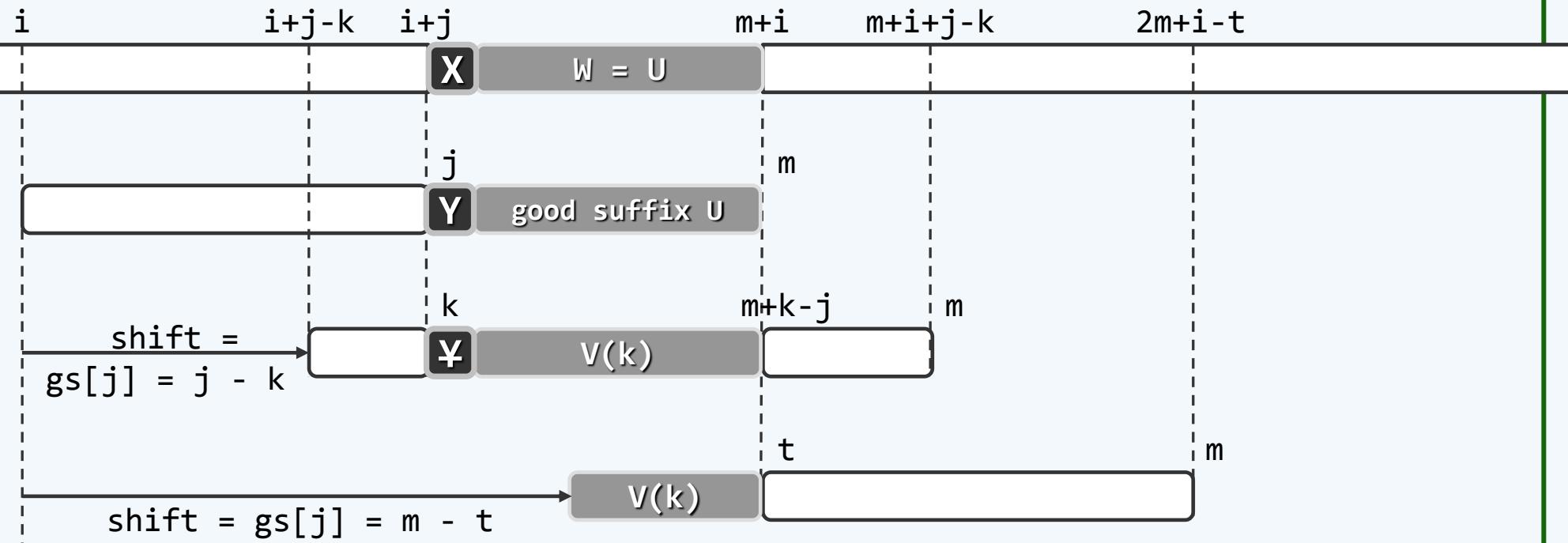


Good-Suffix Shift

若P中的确存在这样的子串 $V(k)$ ，则可

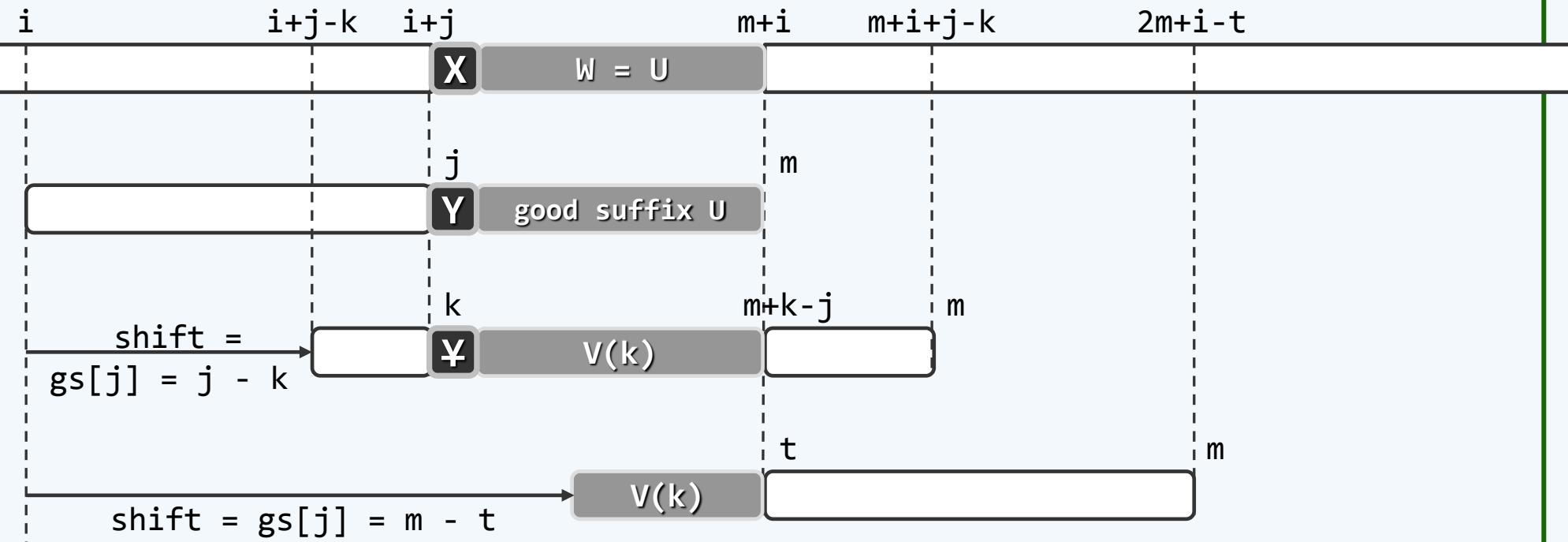
选择其中 k 最大者（尽可能靠后）

然后，通过右移使之与 U 对齐（移动距离尽可能小）

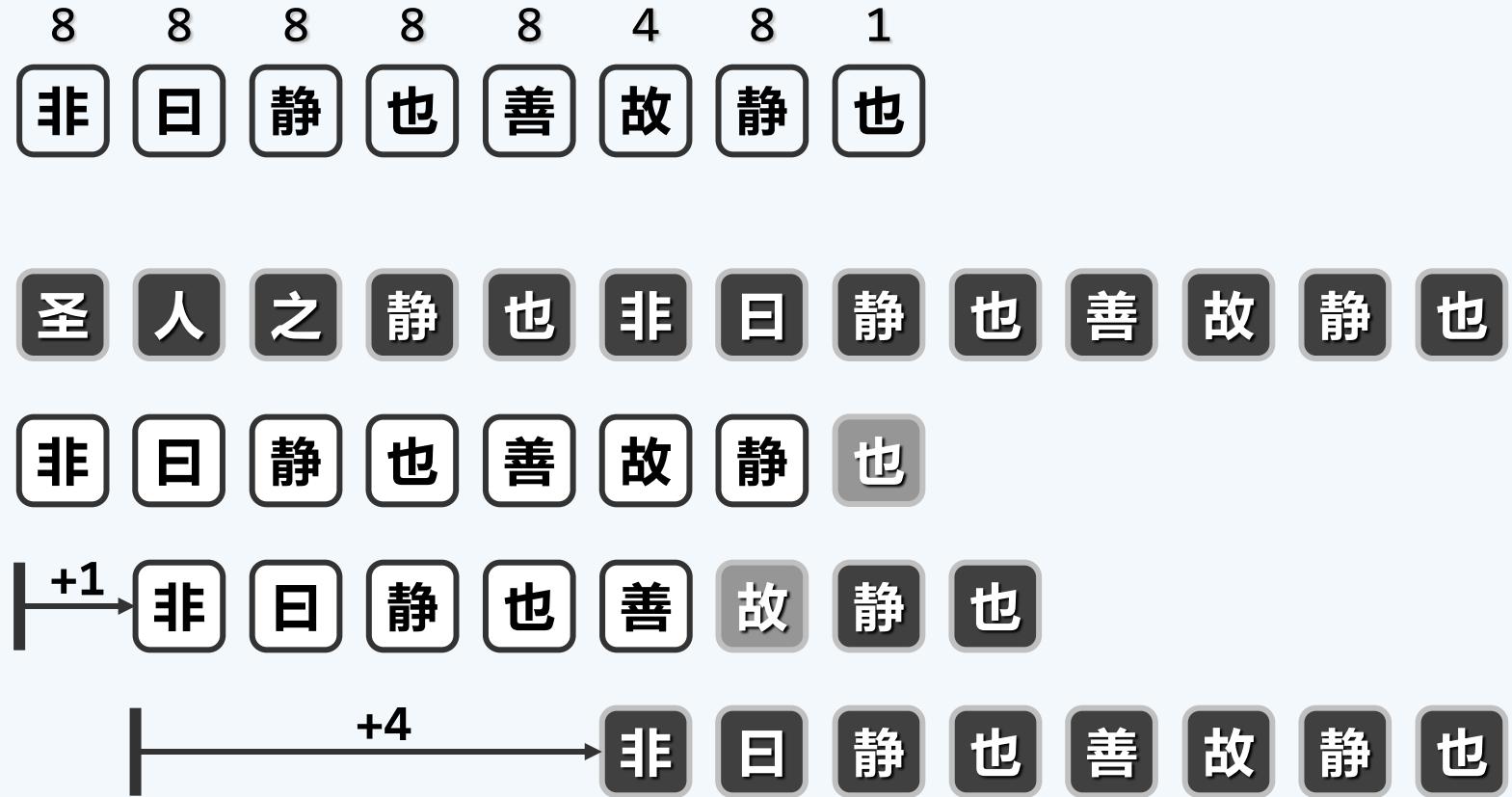


Good-Suffix Shift

- 否则，在所有前缀 $P[0, t]$ 中，取与 U 的后缀匹配的最长者 // 注意：有可能 $t = 0$
- 无论如何，位移量仅取决于 j 和 P 本身——亦可预先计算，并制表待查



实例



11. 串

BM算法：GS策略

构造gs表

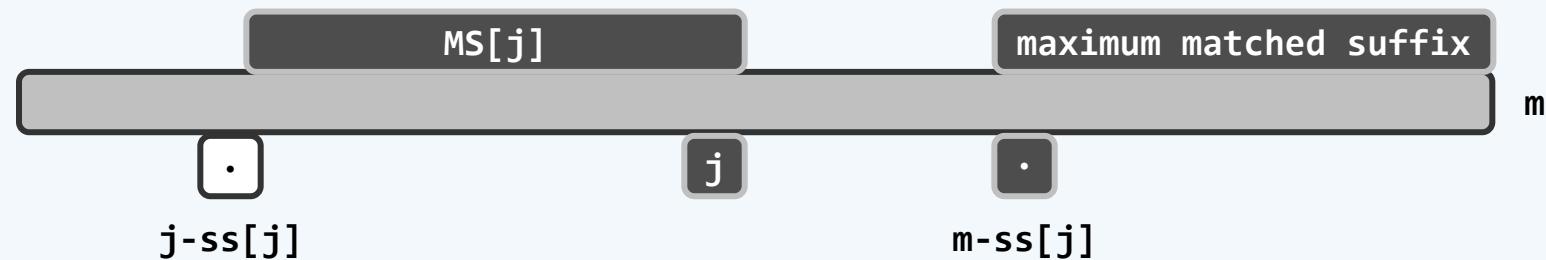
邓俊辉

deng@tsinghua.edu.cn

MS[] → ss[]

❖ MS[j] : P[0, j]的所有后缀中，与P的某一后缀匹配的**最长者**

❖ $ss[j] = |MS[j]| = \max\{ s \leq j + 1 \mid P(j - s, j) = P[m - s, m] \} \leq j + 1$



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I	C	E	D	R	I	C	E	P	R	I	C	E		
0	0	3	0	0	0	0	0	4	0	0	0	0	0	15

❖ 实际上，ss[]表中蕴含了gs[]表的所有信息

//无非两种情况...

ss[] → gs[]

- a) 若 $ss[j] = j + 1$, 则 对于任一字符 $P[i]$ ($i < m - j - 1$)

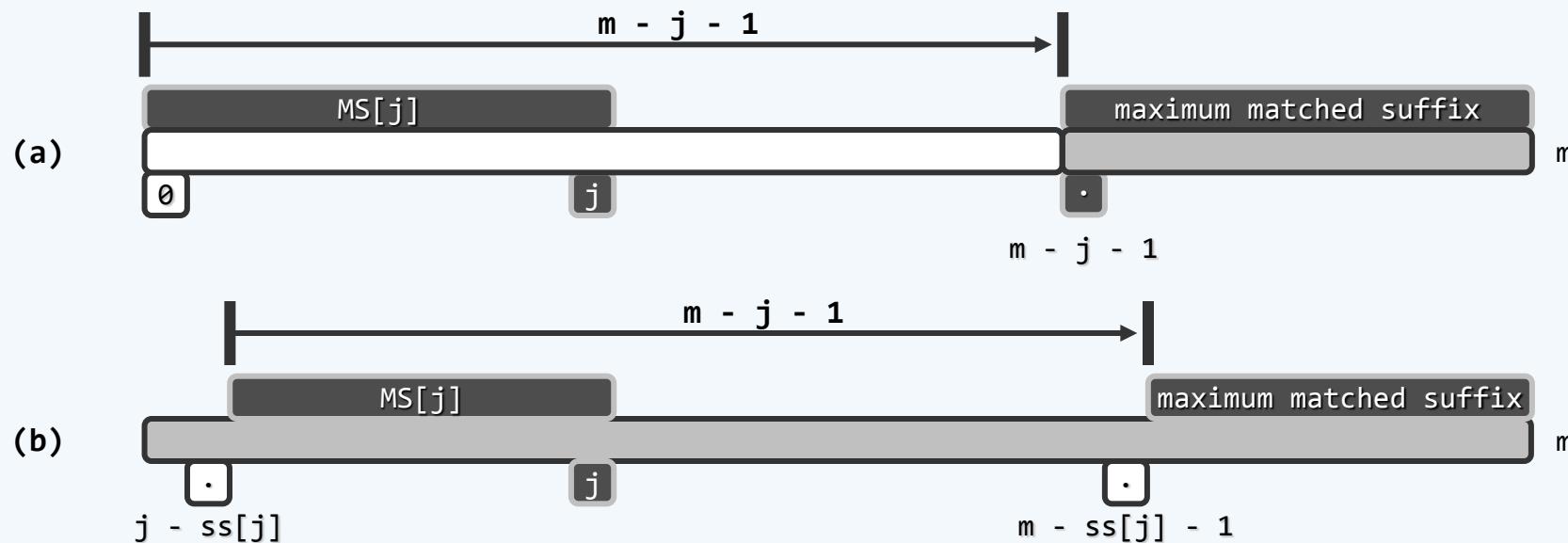
对于任一字符 $P[i]$ ($i < m - j - 1$)

$m - j - 1$ 必是 $gs[i]$ 的一个候选

- b) 若 $ss[j] \leq j$, 则 对于字符 $P[m - ss[j] - 1]$

对于字符 $P[m - ss[j] - 1]$

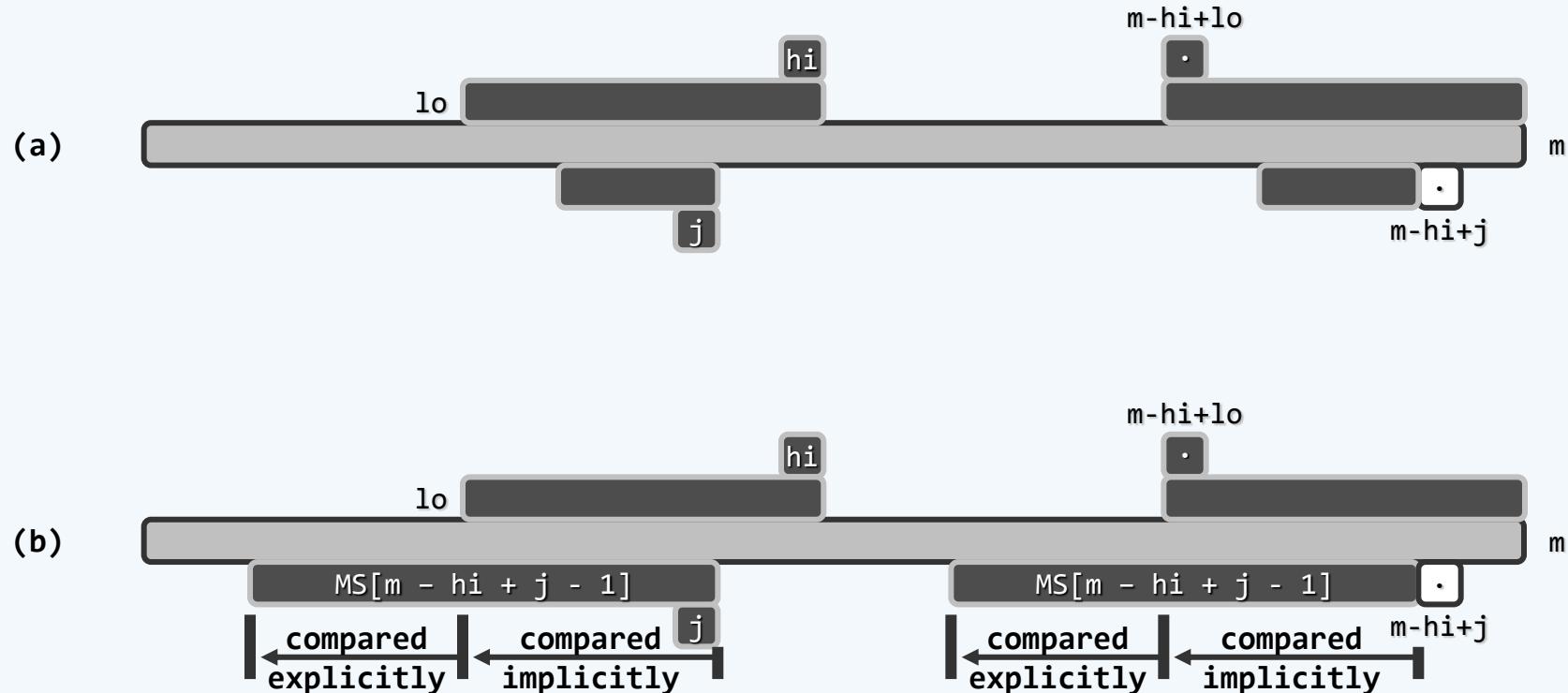
$m - j - 1$ 必是 $gs[m - ss[j] - 1]$ 的一个候选



构造 $ss[]$

❖ 采用蛮力策略，每个字符都需一趟扫描，累计 $O(m^2)$ 时间

❖ 自后向前逆向扫描，只需 $O(m)$ 时间 —— 习题[11-6]



11. 串

BM算法：GS策略

综合性能

邓俊辉

deng@tsinghua.edu.cn

性能

❖ 空间

$$|bc[]| + |gs[]| = O(|\Sigma| + m)$$

❖ 预处理

$$O(|\Sigma| + m)$$

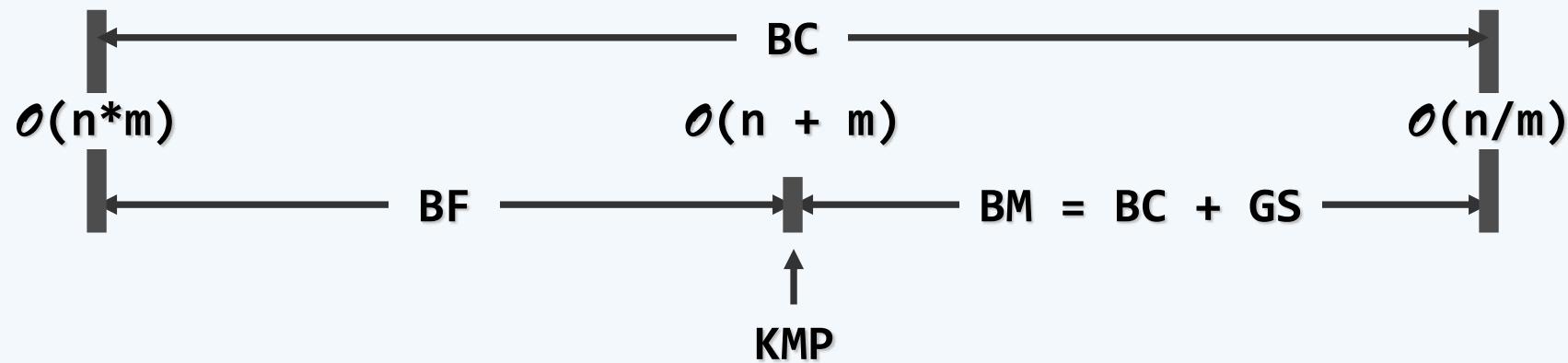
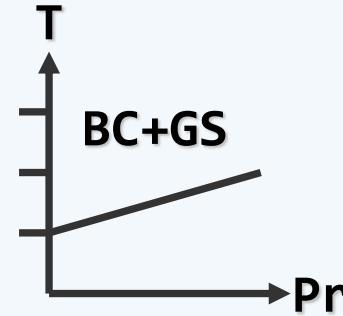
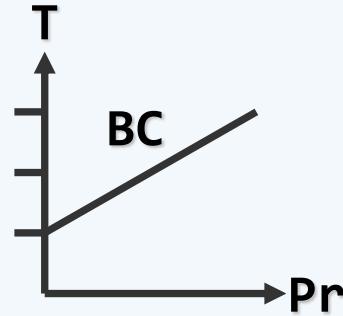
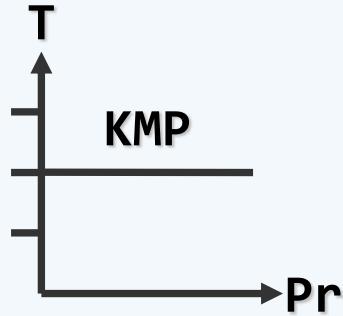
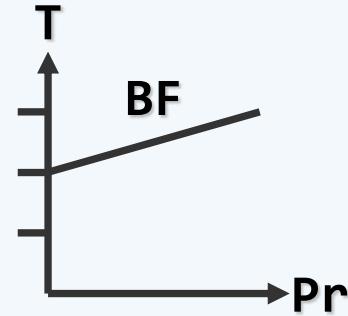
❖ 查找

最好 = $O(n / m)$

最差 = $O(n + m)$

// 参照KMP算法的分析

纵览



11. 串

Karp-Rabin 算法

串即是数

All things are numbers.

- Pythagoras (570 ~ 495 BC)

God made the integers;
all else is the work of man.

- L. Kronecker (1823 ~ 1891)

邓俊辉

deng@tsinghua.edu.cn

凡物皆数



Gödel numbering

逻辑系统的符号、表达式、公式、命题、定理、公理等

均可以不同的自然数标识



素数序列： $p(k)$ = 第 k 个素数

2 , 3 , 5 , 7 , 11 , ...



K. Gödel : 每个有限维的自然数向量，唯一对应于一个自然数

$$\langle a_1, a_2, \dots, a_n \rangle \sim p(1)^{1+a_1} \times p(2)^{1+a_2} \times \dots \times p(n)^{1+a_n}$$

$$\langle 3, 1, 4, 1, 5, 9, 2, 6 \rangle$$

$$= 2^{\boxed{4}} \times 3^{\boxed{2}} \times 5^{\boxed{5}} \times 7^{\boxed{2}} \times 11^{\boxed{6}} \times 13^{\boxed{10}} \times 17^{\boxed{3}} \times 19^{\boxed{7}}$$

凡物皆数

❖ K. Gödel : 给定可数字母表，有限长度的字符串均唯一对应于自然数

$$\{ \text{ a }, \text{ b }, \text{ c }, \dots, \text{ z }, \dots \}$$

$$= \{ 1, 2, 3, \dots, 26, \dots \}$$

$$\boxed{\text{godel}} = \boxed{2^1 + 7} \times \boxed{3^1 + 15} \times \boxed{5^1 + 4} \times \boxed{7^1 + 5} \times \boxed{11^1 + 12}$$

$$= \boxed{139 \ 869 \ 560 \ 310 \ 664 \ 817 \ 087 \ 943 \ 919 \ 200 \ 000}$$

❖ 若果真如RAM模型所假设，字长无限

则只需一个寄存器即可...

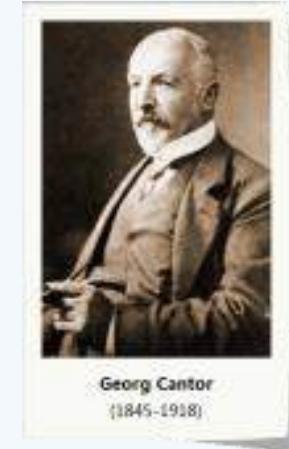
凡物皆数

- ❖ 反过来，由Gödel编号可否唯一确定原字符串？
- ❖ Book IX of *The Elements of Geometry* (ca 300 B.C.)
Euclid: every number factors uniquely into primes
- ❖ 因此，由合法的编号，经素因子分解再排序，即可唯一确定原字符串
- ❖ 素因子分解，至今尚无有效算法！
——不幸？幸运？

凡物皆数

❖ Cantor numbering

$$\diamond \text{cantor}_2(i, j) = ((i + j)^2 + 3*i + j) / 2$$



$$\text{cantor}_2(2, 3) = ((2 + 3)^2 + 3*2 + 3) / 2 = 17$$

$$\text{cantor}_2(3, 2) = ((3 + 2)^2 + 3*3 + 2) / 2 = 18$$

$$\diamond \text{cantor}_{n+1}(a_1, \dots, a_{n-1}, [a_n, a_{n+1}]) =$$

$$\text{cantor}_n(a_1, \dots, a_{n-1}), \text{cantor}_2([a_n, a_{n+1}]))$$

0	1	3	6	10	15
2	4	7	11	16	
5	8	12	17		
9	13	18			
14	19				
20					

凡物皆数

❖ 长度有限的字符串，都可视作 $d = 1 + |\Sigma|$ 进制的自然数

$$\text{decade} = 453145_{(10)}$$

$$//d = 1 + (\boxed{i} - \boxed{a}) = 10$$



❖ 长度无限的字符串，都可视作 $[0, 1)$ 内的 d 进制小数

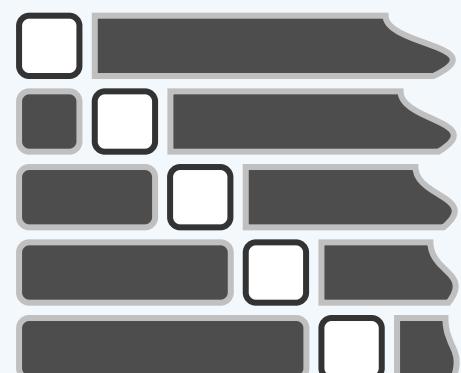
$$\text{bgahbhahbhdei...} = 0.2718281828459\dots$$

$$\text{fade} = \boxed{0.6145} = \boxed{0.6144999\dots} = \text{fad}\boxed{d}ii\dots$$

❖ Cantor：集合是否可数，与其维度无关

有理数集与自然数集一样可数 // (分子, 分母), \aleph_0

无理数集不可数 // Cantor's diagonal, $\aleph_0 \triangleleft \aleph_1$



串亦为数

❖ 十进制串，可直接视作自然数 //指纹（fingerprint），等效于多项式法

$$P = \boxed{82818}$$

$$T = 271 \boxed{82818} 284590452353602874713527$$

❖ 一般地，随意对字符编号 { 0, 1, 2, …, d - 1 } //设 $d = |\Sigma|$

于是，每个字符串都对应于一个 d 进制自然数 //尽管不是单射

$$\boxed{\text{CAT}} = 2 \ 0 \ 19_{(26)} = 1371_{(10)} // \Sigma = \{ A, B, C, \dots, Z \}$$

$$\boxed{\text{ABBA}} = 0 \ 1 \ 1 \ 0_{(26)} = 702_{(10)}$$

❖ P 在 T 中出现 仅当 T 中某一子串与 P 相等 //为什么不是当？

❖ 这，不已经就是一个 算法 吗？！ //具体如何实现？

❖ 问题似乎解决得很顺利，果真如此简单吗？ //复杂度？

11.串

Karp-Rabin算法

散列

“我有些明白了：如果把要指明的恒星与周围恒星的相对位置信息发送出去，接收者把它与星图进行对照，就确定了这颗恒星的位置。”

邓俊辉

deng@tsinghua.edu.cn

数位溢出

❖ 如果 $|\Sigma|$ 很大，模式串 P 较长，其对应的 **指纹** 将 **很大**

比如，若将 P 视作 $|P|$ 位的 $|\Sigma|$ 进制 **自然数**，并将其作为指纹...

❖ 仍以 ASCII 字符集为例 // $|\Sigma| = 128 = 2^7$

只要 $|P| > 9$ ，则指纹的长度将至少是： $7 \times 10 = 70$ bits

❖ 然而，目前的字长一般也不过 **64 位** // 存储不便

❖ 而更重要地，指纹的 **计算** 及 **比对**，将不能在 $O(1)$ 时间内完成 // RAM!?

准确地说，需要 $O(|P| / 64) = O(m)$ 时间；总体需要 $O(n * m)$ 时间 // 与蛮力算法相当

❖ 有何高招...

散列压缩

❖ 基本构思：通过对比压缩后的指纹，确定匹配位置 //更接近于人类的指纹

❖ 关键技巧：借助散列，将指纹压缩至存储器支持的范围

比如，采用模余函数： $\text{hash}(\text{ key }) = \text{key \% M}$ //若取M = 97...

❖ P = [8 2 8 1 8] // $\text{hash}(P) = \text{hash}(82818) = 77$

T = 2 7 1 [8 2 8 1 8] 2 8 4 5 9 0 4 5 2 3 5 3 6

[2 7 1 8 2] //22

[7 1 8 2 8] //48

[1 8 2 8 1] //45

[8 2 8 1 8] //77

散列冲突

❖ 注意：hash()值相等，并非匹配的充分条件... //好在必要

因此，通过hash()筛选之后，还须经过严格比对，方可最终确定是否匹配...

❖ $P = [1 \ 8 \ 2 \ 8 \ 4] //$ 仍取 $M = 97$ ，则 $\text{hash}(18284) = 48$

$T = 2 [7 \ 1 \ 8 \ 2 \ 8] [1 \ 8 \ 2 \ 8 \ 4] 5 \ 9 \ 0 \ 4 \ 5 \ 2 \ 3 \ 5 \ 3 \ 6$

$[2 \ 7 \ 1 \ 8 \ 2] //22$

$[7 \ 1 \ 8 \ 2 \ 8] //48$

...

$[1 \ 8 \ 2 \ 8 \ 4] //48$

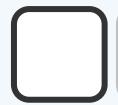
❖ 既然是通过散列压缩，指纹冲突就在所难免

适当选取散列函数，可以大大降低冲突的可能

快速指纹计算

❖ hash()的计算，似乎每次均需 $\Theta(|P|)$ 时间

有可能加速吗？



suffix(T_{i-1})

❖ 回忆一下，进制转换算法...

prefix(T_i)

❖ 观察 相邻的两次计算之间，存在着某种相关性

相邻的两个指纹之间，也有着某种相关性

❖ 利用上述性质，即可在 $\Theta(1)$ 时间内，由上一指纹得到下一指纹...

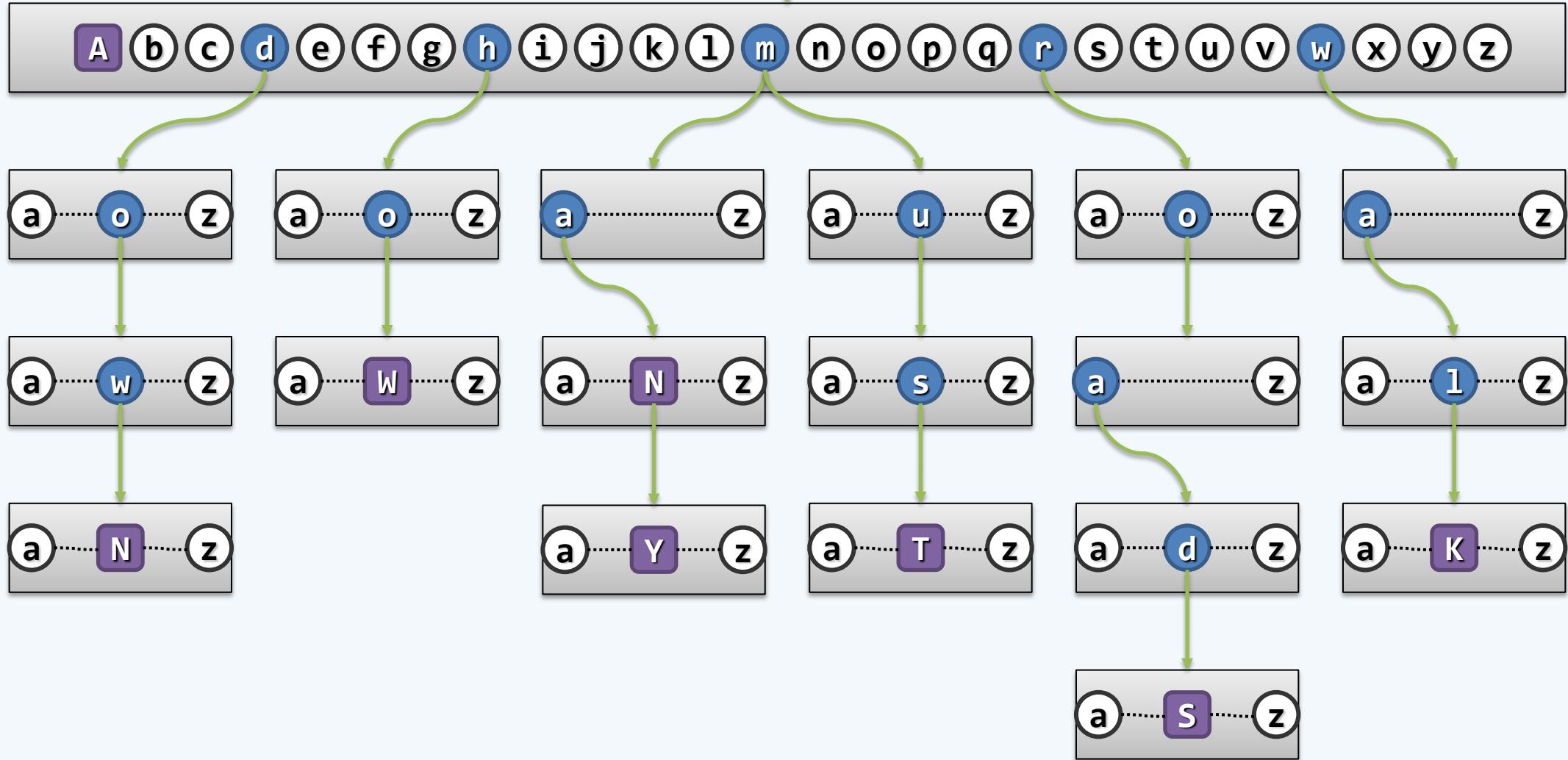
11.串

键树

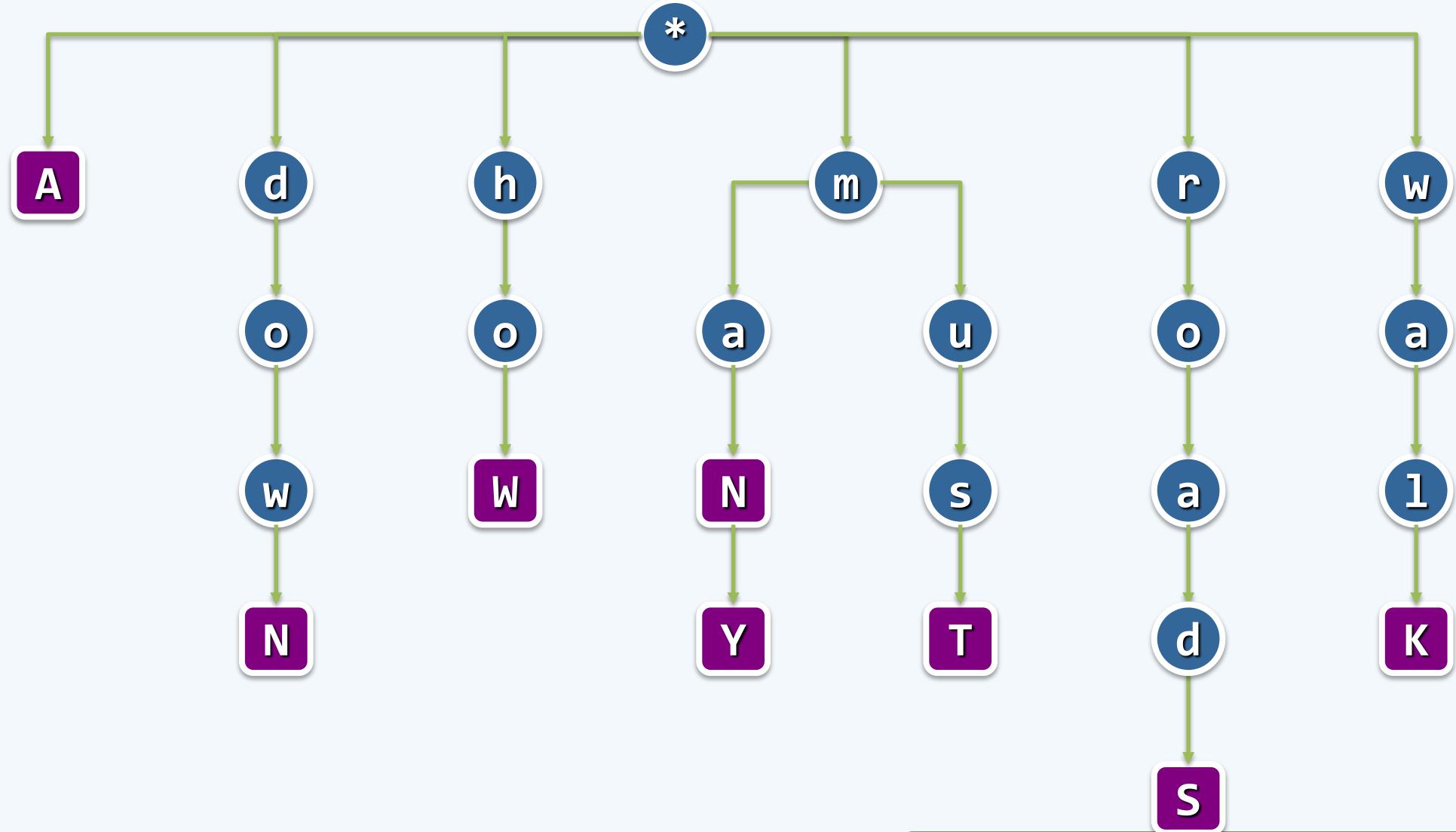
邓俊辉

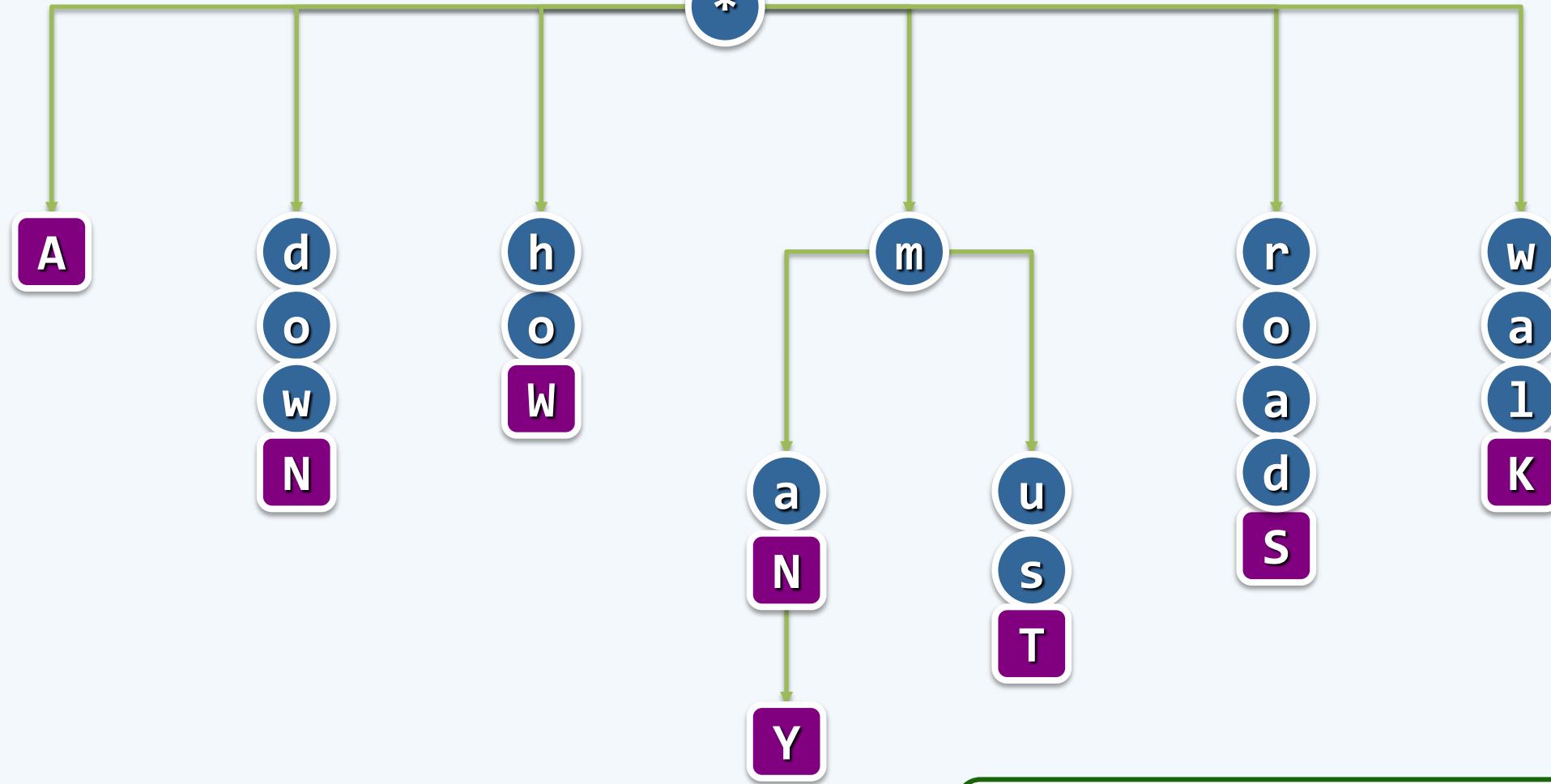
deng@tsinghua.edu.cn

how many roads must a man walk down

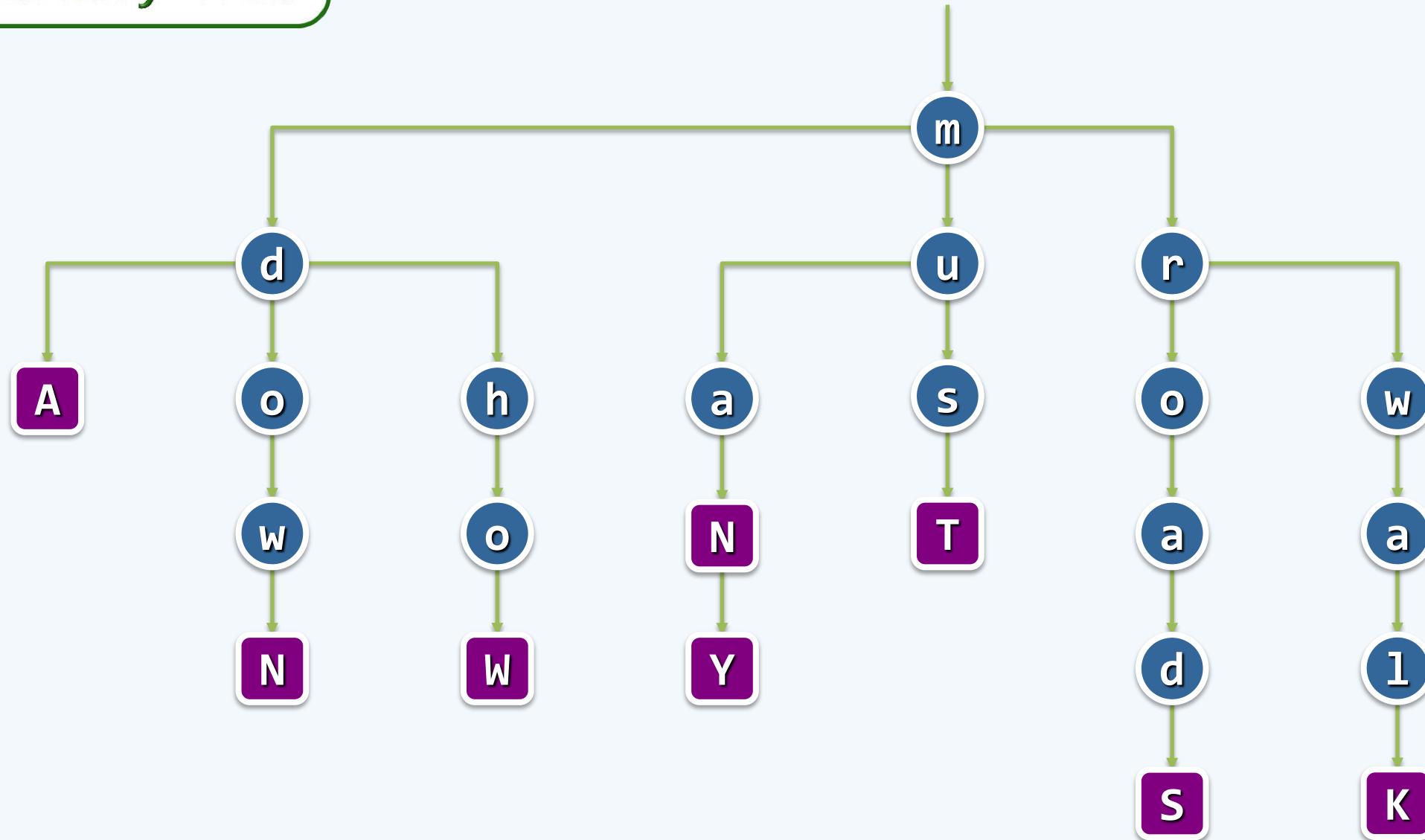


Trie = re[TRIE]val



PATRICIA Tree

Ternary Trie



12. 排序

快速排序

算法A

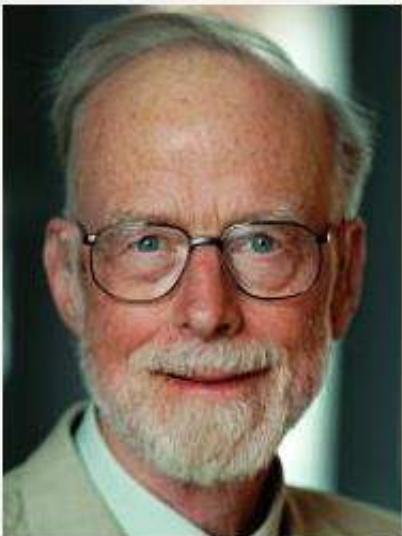
左朱雀之芨芨兮

邓俊辉

右苍龙之躍躍

deng@tsinghua.edu.cn

分而治之



C. A. R. Hoare
(1934 ~)
Turing Award, 1980

❖ 将序列分为两个子序列： $S = S_L + S_R$ // $\theta(n)$

规模**缩小**： $\max\{|S_L|, |S_R|\} < n$

彼此**独立**： $\max(S_L) \leq \min(S_R)$

❖ 在子序列分别**递归地**排序之后，原序列自然有序

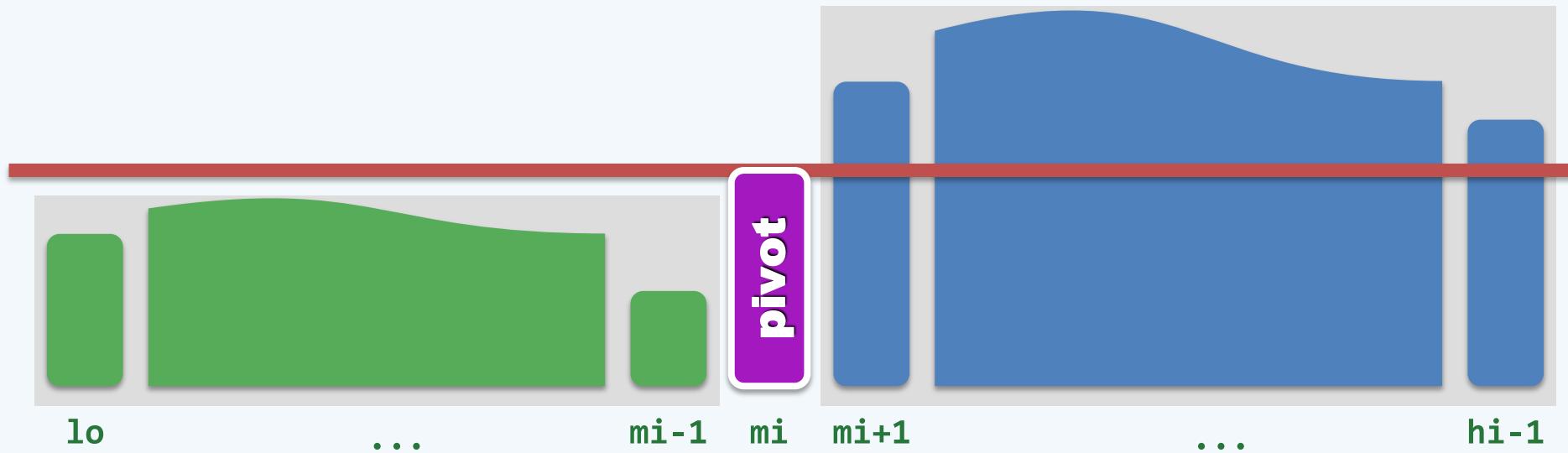
$\text{sorted}(S) = \text{sorted}(S_L) + \text{sorted}(S_R)$

❖ 平凡解：只剩**单个**元素时，本身就是解

❖ mergesort的计算量和难点在于**合**，而quicksort在于**分** //如何实现上述划分呢？

轴点

❖ pivot : 左/右侧的元素，均不比它更大/小



❖ 以轴点为界，原序列的划分自然实现：

$$[lo, hi) = [lo, mi) + [mi] + (mi, hi)$$

快速排序

```

❖ template <typename T> void Vector<T>::quickSort( Rank lo, Rank hi ) {

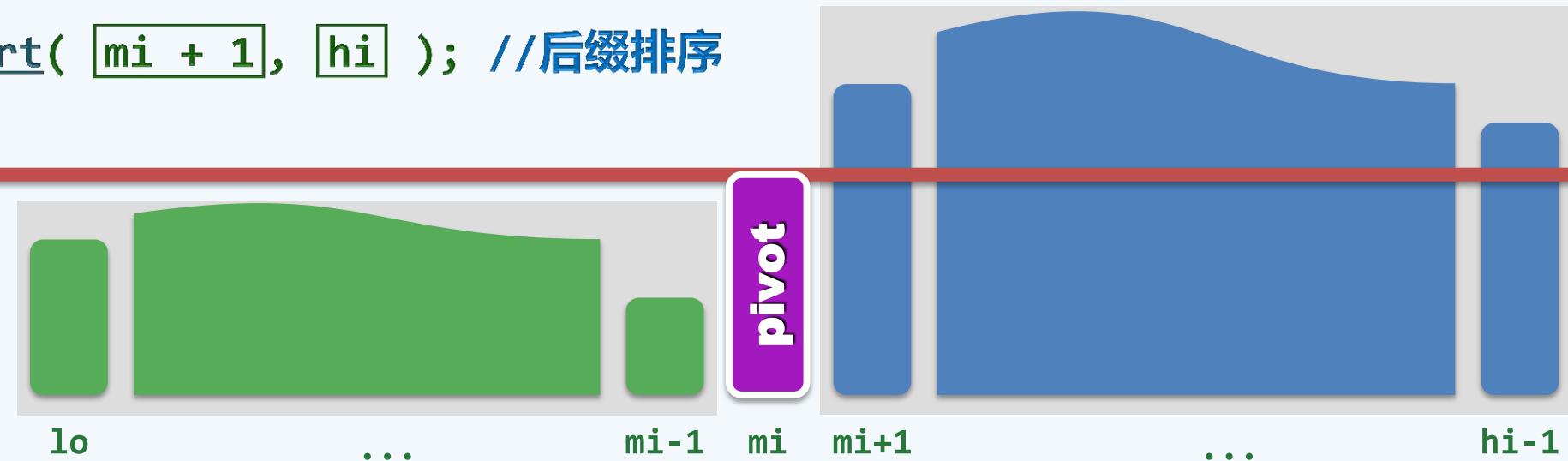
    if ( hi - lo < 2 ) return; //单元素区间自然有序，否则

    Rank mi = partition( lo, hi - 1 ); //先构造轴点，再

    quickSort( lo, mi ); //前缀排序

    quickSort( mi + 1, hi ); //后缀排序
}

```



轴点

- ❖ 坏消息： 在原始序列中，轴点未必存在...
- ❖ 必要条件： 轴点必定已然就位 //尽管反之不然
- ❖ derangement： 2 3 4 ... n 1
- ❖ 特别地： 在有序序列中，所有元素皆为轴点；反之亦然
- ❖ 快速排序： 就是将所有元素逐个转换为轴点的过程
- ❖ 好消息： 通过适当交换，可使任一元素转换为轴点
- ❖ 问题： 如何交换？成本多高？

构造轴点

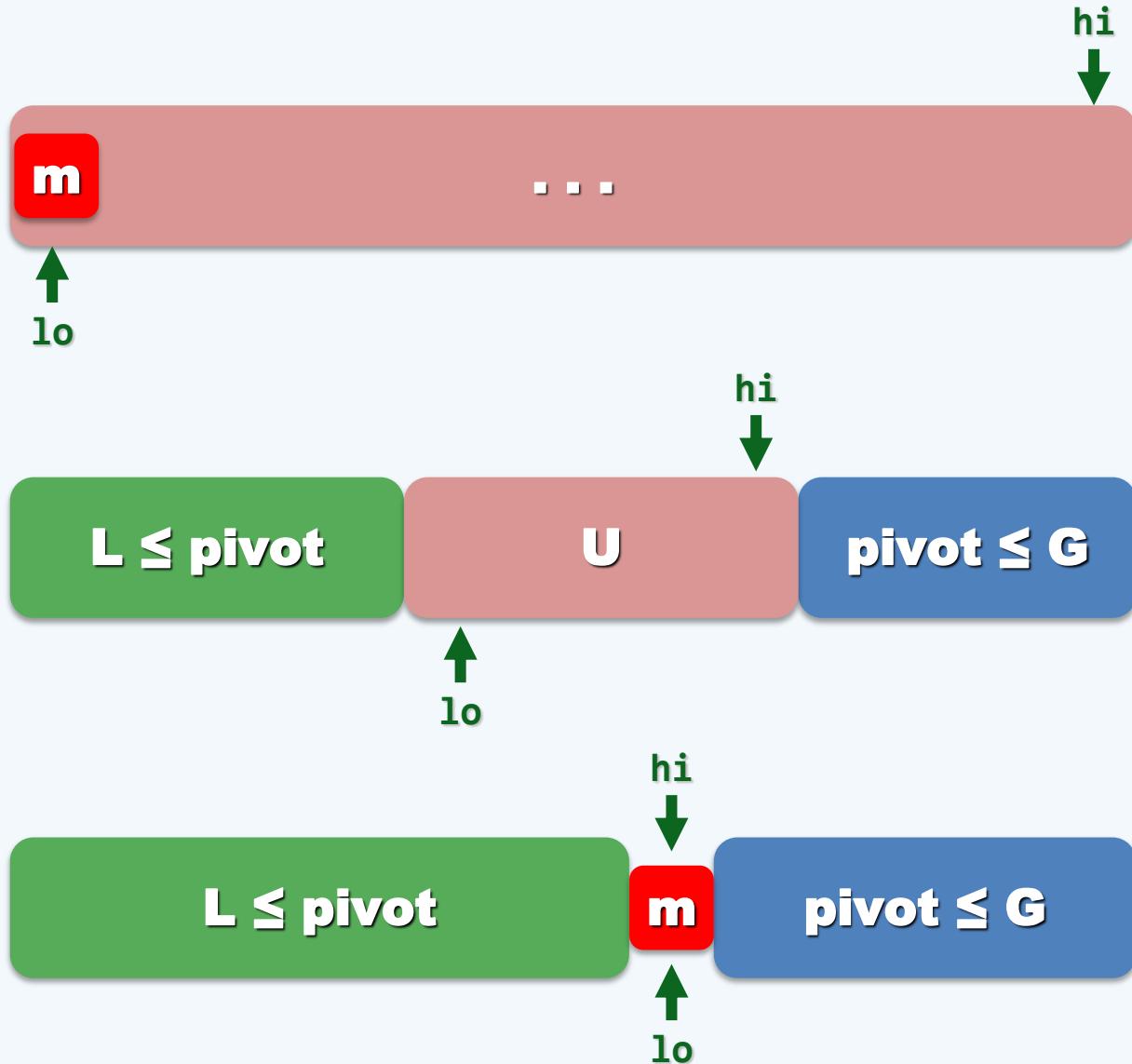
❖ 任取一候选者（如 $[0]$ ）

❖ 2个指针，3个子序列

前缀L： \leq 候选者，初始为空

后缀G： \geq 候选者，初始为空

中段U：? 待确定，初始为全集



构造轴点

❖ 交替地向内移动 lo 和 hi

❖ 逐个检查当前元素：

若更小/大，则转移归入 L/G

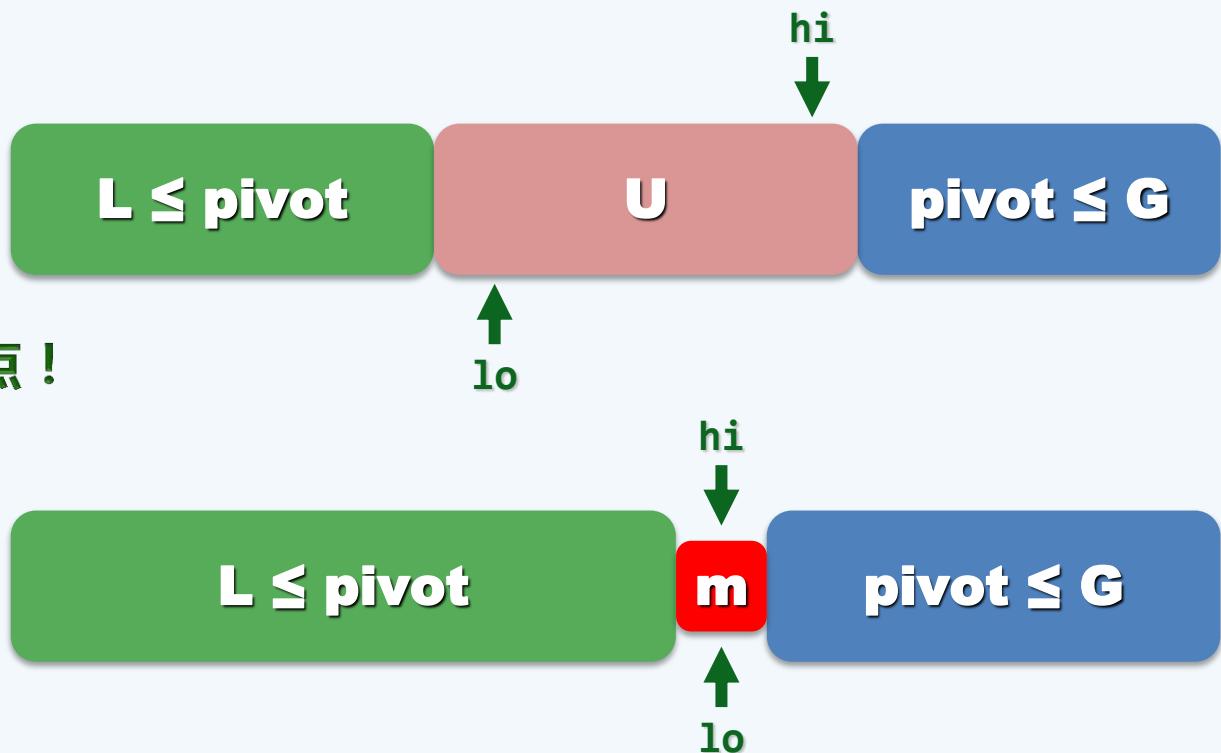
❖ 当 $lo = hi$ 时，只需

将候选者嵌入于 L/G 之间，它即是轴点！

❖ 整个过程中

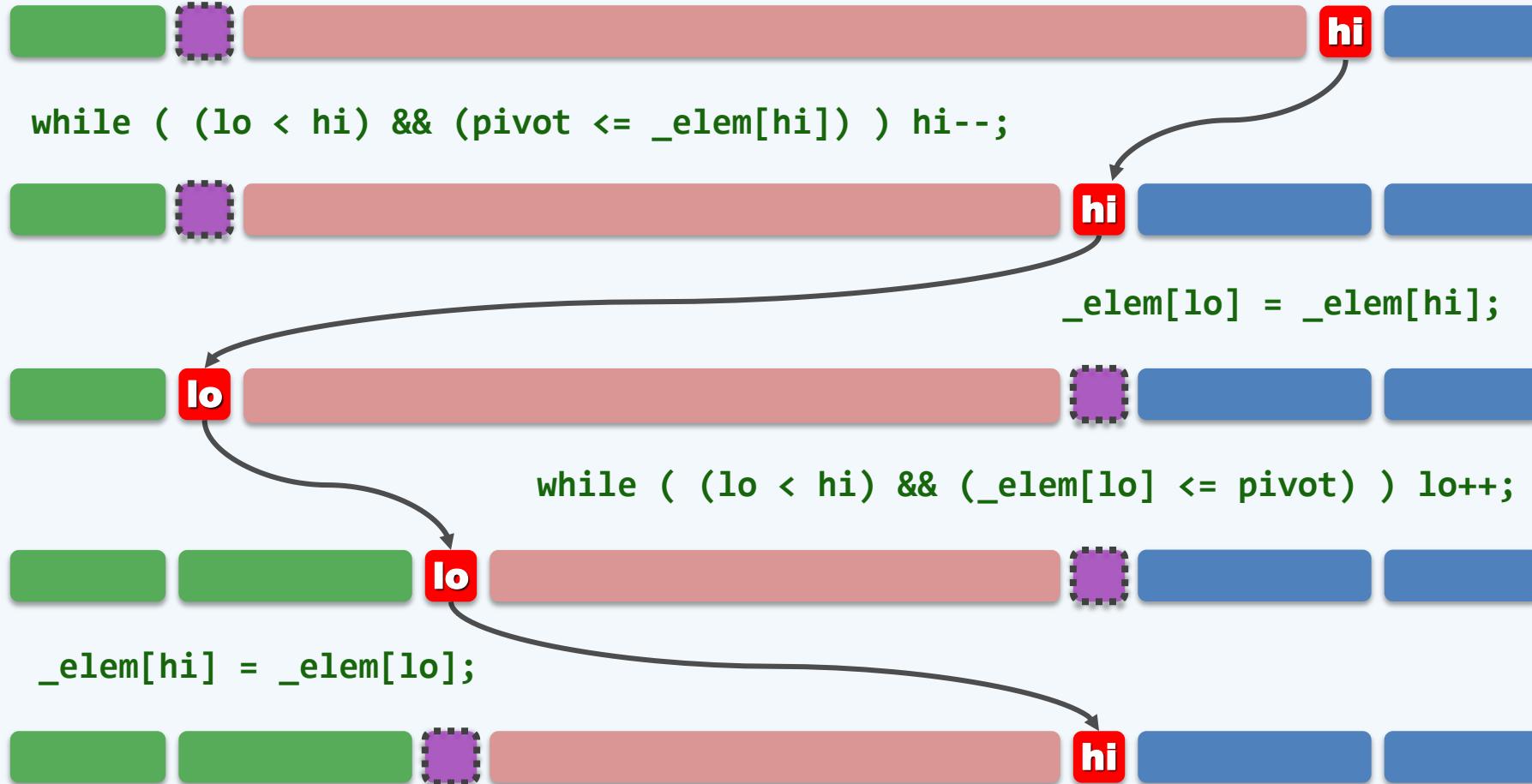
各元素最多移动一次（候选者两次）

——累计 $O(n)$ 时间、 $O(1)$ 辅助空间



不变性 + 单调性

❖ $L \leq pivot \leq G$; $U = [lo, hi]$ 中， $[lo]$ 和 $[hi]$ 交替空闲



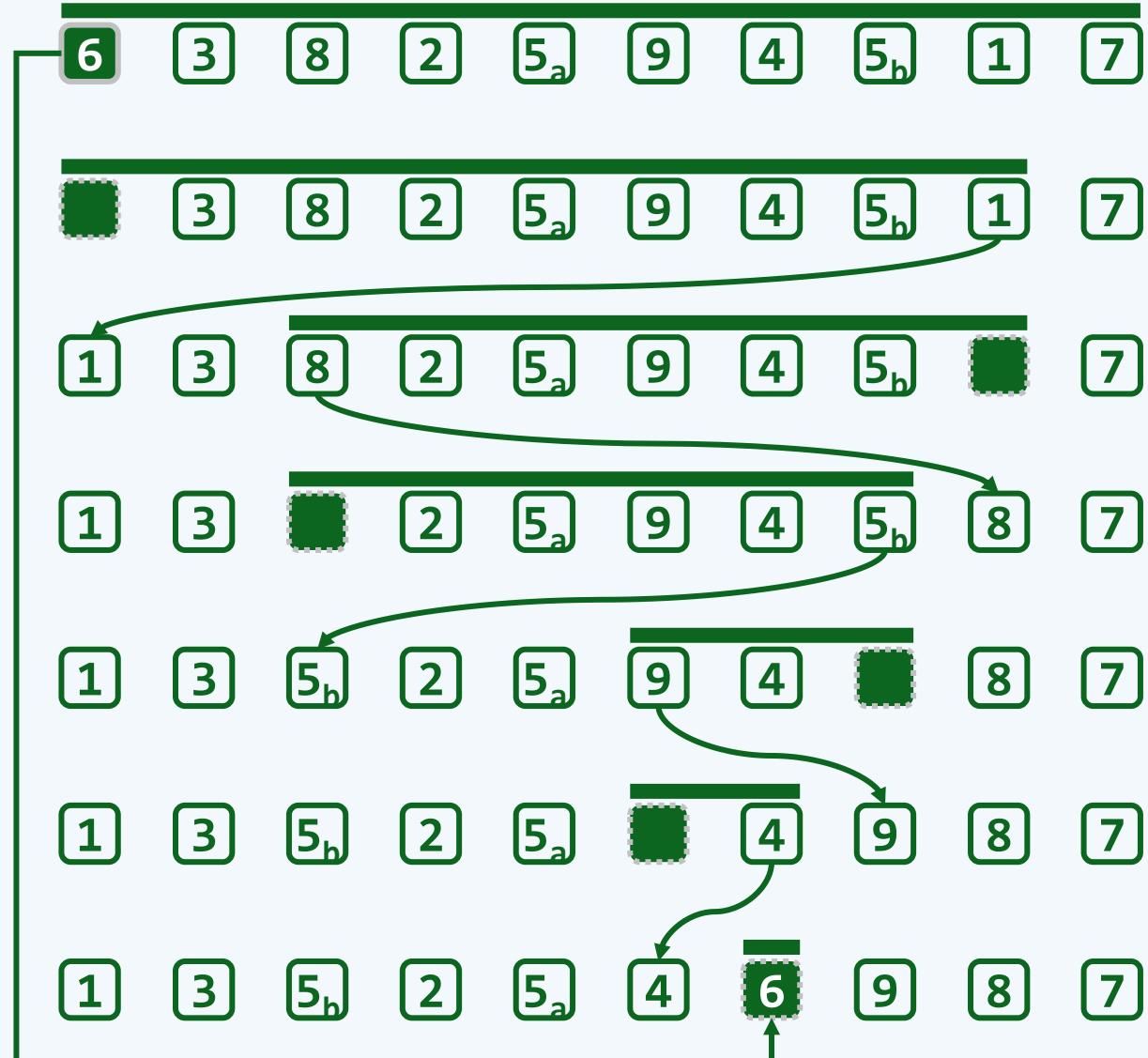
实现

```

template <typename T> Rank Vector<T>::partition( Rank lo, Rank hi ) { // [lo, hi]
    swap( _elem[ lo ], _elem[ lo + rand() % ( hi - lo + 1 ) ] ); // 随机交换
    T pivot = _elem[ lo ]; // 经以上交换，等效于随机选取候选轴点
    while ( lo < hi ) { // 从两端交替地向中间扫描，彼此靠拢
        while ( [lo < hi] && [pivot <= _elem[ hi ]] ) hi--; // 向左拓展 G
        _elem[ lo ] = _elem[ hi ]; // 凡小于轴点者，皆归入 L
        while ( [lo < hi] && [_elem[ lo ] <= pivot] ) lo++; // 向右拓展 L
        _elem[ hi ] = _elem[ lo ]; // 凡大于轴点者，皆归入 G
    } // assert: lo == hi
    _elem[ lo ] = pivot; return lo; // 候选轴点归位；返回其秩
}

```

实例



12. 排序

快速排序

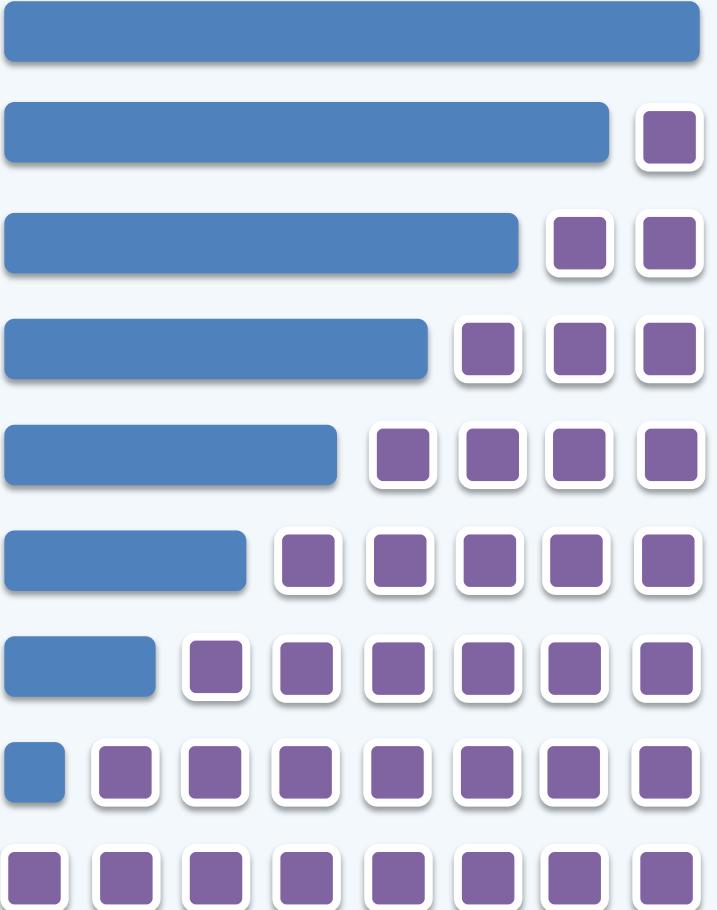
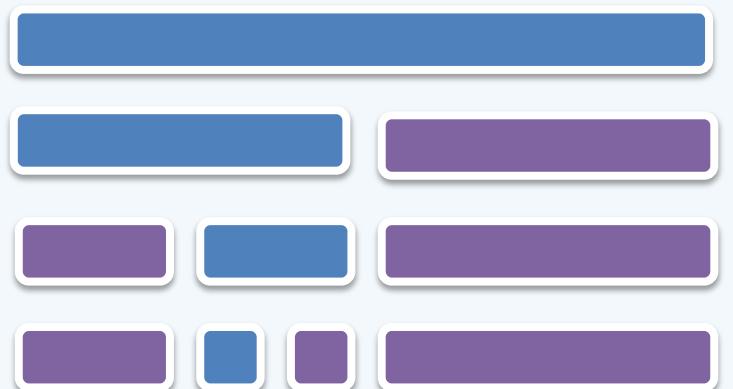
性能分析(1): 基本

邓俊辉

deng@tsinghua.edu.cn

非时间性能

- ❖ **unstable** : lo/hi的移动方向相反，左/右侧的大/小重复元素可能前/后颠倒
- ❖ **in-place** : 只需 $\mathcal{O}(1)$ 附加空间
- ❖ **overhead for recursion** : 最坏情况下需要 $\Omega(n)$ 空间
- ❖ **greedy (smaller first) + tail recursion removal** : 可控制在 $\mathcal{O}(\log n)$, Sedgewick



时间性能

❖ 最好情况：每次划分都（接近）**平均**，轴点总是（接近）**中央**

$$T(n) = 2 \cdot T\left(\frac{n-1}{2}\right) + \mathcal{O}(n) = \mathcal{O}(n \log n)$$
//到达下界！

❖ 最坏情况：每次划分都**极不均衡** **//比如，轴点总是最小/大元素**

$$T(n) = T(n-1) + T(0) + \mathcal{O}(n) = \mathcal{O}(n^2)$$
//与起泡排序相当！

❖ 即便采用**随机选取**、**(Unix) 三者取中**之类的策略

也只能降低最坏情况的概率，而无法杜绝

❖ 既然如此，为何还称作**快速排序**？

12. 排序

快速排序

性能分析(2): 固定策略 + 随机序列

邓俊辉

deng@tsinghua.edu.cn

准居中

- ❖ 出现的概率： 最坏情况（ $\Omega(n)$ 递归深度）极低
平均情况（ $\mathcal{O}(\log n)$ 递归深度）极高
- ❖ 实际上： 除非过于侧偏的pivot，都会有效地缩短递归深度

(1 - λ) / 2width = λ = Pr.(1 - λ) / 2

- ❖ **准居中**：pivot的秩落在宽度为 λn 的居中区间 // λ 也是这种情况出现的概率
- ❖ 每一递归路径上，至多出现 $\log_{\frac{2}{1+\lambda}} n$ 个**准居中**的pivots ...

期望深度

❖ 每递归一层，都有 $\lambda \mid 1 - \lambda$ 的概率 准居中 | 准侧偏

❖ 深入 $\frac{1}{\lambda} \cdot \log_{\frac{2}{1+\lambda}} n$ 层后，即可期望出现 $\log_{\frac{2}{1+\lambda}} n$ 次 准居中，且有极高的概率出现

❖ 相反情况的概率 $< (1 - \lambda)^{(\frac{1}{\lambda} - 1) \cdot \log_{\frac{2}{1+\lambda}} n}$

$(1 - \lambda) / 2$

width = λ = Pr.

$(1 - \lambda) / 2$

❖ 以 $\lambda = 1/3$ 和 $n = 128$ 为例，此概率 $< \left(\frac{2}{3}\right)^{2 \cdot \log_{\frac{3}{2}} 128} < \left(\frac{2}{3}\right)^{2 \cdot \log_2 128} < \left(\frac{2}{3}\right)^{14} < 0.4\%$

❖ 因此有极高的概率，递归深度不超过 $\frac{1}{\lambda} \cdot \log_{\frac{2}{1+\lambda}} n = 3 \cdot \log_{\frac{3}{2}} n$

平均性能

❖ $\Theta(n \log n)$ —— 以均匀独立分布为例 . . .

L

k

G

 $T(n)$

$$= (n + 1) + \frac{1}{n} \times \sum_{k=0}^{n-1} [T(k) + T(n - k - 1)] \quad // \text{对称}$$

$$= (n + 1) + \frac{2}{n} \times \sum_{k=0}^{n-1} T(k) \quad // \text{乘以 } n \dots$$

$$n \cdot T(n) = n \cdot (n + 1) + 2 \times \sum_{k=0}^{n-1} T(k) \quad // \text{下推至 } n-1 \dots$$

$$(n - 1) \cdot T(n - 1) = (n - 1) \cdot n + 2 \times \sum_{k=0}^{n-2} T(k) \quad // \text{相减} \dots$$

平均性能

❖ $n \cdot T(n) - (n - 1) \cdot T(n - 1) = 2 \cdot n + 2 \times T(n - 1)$ //整理 ...

$n \cdot T(n) = 2 \cdot n + (n + 1) \cdot T(n - 1)$ //除以 $n(n+1)$...

$$\begin{aligned}
 T(n)/(n + 1) &= 2/(n + 1) + T(n - 1)/n \\
 &= 2/(n + 1) + 2/n + T(n - 2)/(n - 1) \\
 &= 2/(n + 1) + 2/n + 2/(n - 1) + \dots + 2/2 + T(0)/1 \\
 &< 2 \cdot \ln n
 \end{aligned}$$

$$T(n) \approx 2 \cdot n \cdot \ln n = (2 \cdot \ln 2) \cdot n \log n \approx 1.39 \cdot n \log n = \mathcal{O}(n \log n)$$

12. 排序

快速排序

性能分析(3) : 固定序列 + 随机策略

邓俊辉

deng@tsinghua.edu.cn

- ❖ 随机选取pivot，随机选择优先递归方向
- ❖ quicksort的每一次运行，都对应于一棵BST //假设递归实例执行次序固定，BFS或DFS

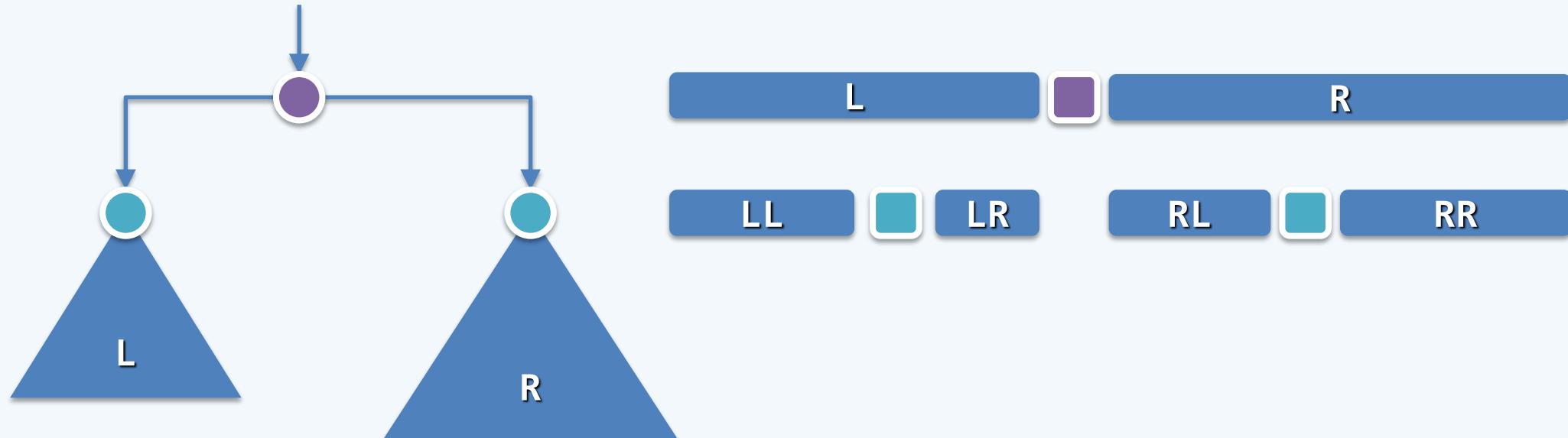
全序列 中被选取的轴点，对应于**全树** 的根节点；左、右子序列，对应于左、右子树

子序列 中被选取的轴点，对应于**子树** 的根节点；.....



期望比较次数

- ❖ 在 quicksort 过程中执行的 比较操作 次数，等于
按对应的随机序列 生成该 BST 过程中，执行的 比较操作 次数
- ❖ quicksort 所有可能的运行过程中，所做比较次数的 平均 | 期望 值，等于
通过随机输入序列构造 BST 过程中，所做比较次数的 平均 | 期望 值



数学期望的线性律

◆ 生成BST的随机序列，记作： $v_1 \ v_2 \ v_3 \ \dots \ v_n$

◆ 相应的计算成本（比较操作的次数），记作： $T = \sum_i \sum_{j < i} t(i, j)$

◆ 其中： $t(i, j) = \begin{cases} 1 & v_i \text{在插入过程中，与 } v_j \text{ 做过比较；或等价地， } v_j \text{ 是 } v_i \text{ 的祖先} \\ 0 & \text{否则} \end{cases}$

◆ 于是： $E(T) = \sum_i \sum_{j < i} E(t(i, j)) \quad // \text{by linearity of expectation}$
 $= \sum_i \sum_{j < i} Pr(1 = t(i, j))$

平均性能

❖ 固定：任意一对 v_j 与 v_i

忽略：介于二者之间插入的所有节点

//它们存在与否，与 v_i 与 v_j 是否做过比较无关

于是：作为等效的此前最后一个被插入者， v_j 必是叶子

❖ 观察： $\{ v_1 \ v_2 \ v_3 \ \dots \ v_j \}$ 将整个取值区间分为 $j+1$ 个子区间 //无论是否已排序

❖ 观察： v_i 与 v_j 做过比较 iff v_i 与落在以 v_j 为边界的两个子区间之一

❖ 因此： $Pr(1 = t(i, j)) = \frac{2}{j + 1}$ // $\{ v_1 \ \dots \ v_j \}$ 亦是随机序列，各子区间概率均等

❖ 于是： $E(T) = \sum_i \sum_{j < i} Pr(1 = t(i, j)) = \sum_i \sum_{j < i} \frac{2}{j + 1} = \sum_i \mathcal{O}(log n) = \mathcal{O}(n log n)$

12. 排序

快速排序

重复元素

邓俊辉

左见兮鸣鳩，右睹兮呼枭

deng@tsinghua.edu.cn

重复元素

❖ 大量甚至全部元素重复时

轴点位置总是接近于 lo

子序列的划分极不均匀

二分递归退化为线性递归

递归深度接近于 $\Theta(n)$

运行时间接近于 $\Theta(n^2)$

❖ 移动 lo 和 hi 的过程中，同时比较相邻元素：若属于相邻的重复元素，则不再深入递归

但一般情况下，如此计算量反而增加，得不偿失

❖ 对算法 A 略做调整，即可解决问题——为便于对比，先给出等价形式 A1...

算法A1

```

template <typename T> Rank Vector<T>::partition( Rank lo, Rank hi ) { // [lo, hi]
    swap( _elem[ lo ], _elem[ lo + rand() % ( hi - lo + 1 ) ] ); // 随机交换
    T pivot = _elem[ lo ]; // 经以上交换，等效于随机选取候选轴点
    while ( lo < hi ) { // 从两端交替地向中间扫描，彼此靠拢
        while ( [lo < hi] && [pivot <= _elem[ hi ]] ) hi--; // 向左拓展G
        if (lo < hi) _elem[ lo++ ] = _elem[ hi ]; // 凡小于轴点者，皆归入L
        while ( [lo < hi] && [_elem[ lo ] <= pivot] ) lo++; // 向右拓展L
        if (lo < hi) _elem[ hi-- ] = _elem[ lo ]; // 凡大于轴点者，皆归入G
    } // assert: lo == hi
    _elem[ lo ] = pivot; return lo; // 候选轴点归位；返回其秩
}

```

算法B1

```

template <typename T> Rank Vector<T>::partition( Rank lo, Rank hi ) { // [lo, hi]
    swap( _elem[ lo ], _elem[ lo + rand() % ( hi - lo + 1 ) ] ); // 随机交换
    T pivot = _elem[ lo ]; // 经以上交换，等效于随机选取候选轴点
    while ( lo < hi ) { // 从两端交替地向中间扫描，彼此靠拢
        while ( [lo < hi] && pivot < _elem[ hi ] ) hi--; // 向左拓展G
        if (lo < hi) _elem[ lo++ ] = _elem[ hi ]; // 凡不大于轴点者，皆归入L
        while ( [lo < hi] && _elem[ lo ] < pivot ) lo++; // 向右拓展L
        if (lo < hi) _elem[ hi-- ] = _elem[ lo ]; // 凡不小于轴点者，皆归入G
    } // assert: lo == hi
    _elem[ lo ] = pivot; return lo; // 候选轴点归位；返回其秩
}

```

算法B

```

template <typename T> Rank Vector<T>::partition( Rank lo, Rank hi ) { // [lo, hi]
    swap( _elem[ lo ], _elem[ lo + rand() % ( hi - lo + 1 ) ] ); // 随机交换
    T pivot = _elem[ lo ]; // 经以上交换，等效于随机选取候选轴点
    while ( lo < hi ) { // 从两端交替地向中间扫描，彼此靠拢
        while ( lo < hi )
            if ( pivot < _elem[ hi ] ) hi--; // 向左拓展G，直至遇到不大于轴点者
            else { _elem[ lo++ ] = _elem[ hi ]; break; } // 将其归入L
        while ( lo < hi )
            if ( _elem[ lo ] < pivot ) lo++; // 向右拓展L，直至遇到不小于轴点者
            else { _elem[ hi-- ] = _elem[ lo ]; break; } // 将其归入G
    } // assert: lo == hi
    _elem[ lo ] = pivot; return lo; // 候选轴点归位；返回其秩
}

```

性能

❖ 可以正确地处理一般情况，而且复杂度并未实质增高

❖ 处理重复元素时

- lo 和 hi 会交替移动
- 二者移动的距离大致相当

最终，轴点被安置于 $(\text{lo} + \text{hi}) / 2$ 处，实现划分均匀

❖ 相对于算法A的勤于拓展、懒于交换，转为懒于拓展、勤于交换，因此：

- 交换操作有所增多——尤其是雷同元素，在版本A中多不移动
- 更不稳定

12. 排序

快速排序

变种

The Great walks with the Small without fear.

邓俊辉

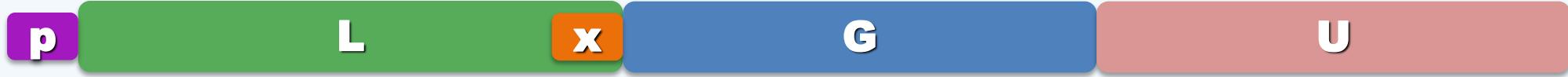
The Middle keeps aloof.

deng@tsinghua.edu.cn

不变性

❖ 四部分： $S = [lo] + L(lo, mi] + G(mi, k) + U[k, hi]$

$L < pivot \leq G$



单调性

❖ [k]不小于轴点 ? 直接G拓展 : G滚动后移，L拓展

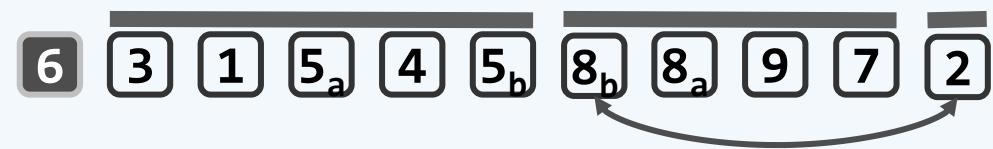
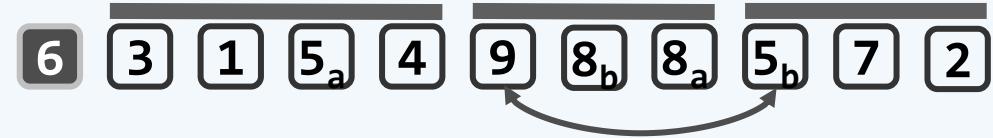
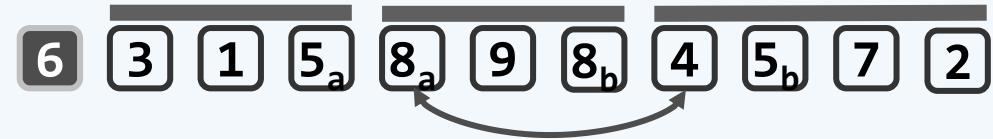
`pivot <= S[k] ? k++ : swap(S[++mi], S[k++])`



实现

```
template <typename T> Rank Vector<T>::partition( Rank lo, Rank hi ) { // [lo, hi]  
    swap( _elem[ lo ], _elem[ lo + rand() % ( hi - lo + 1 ) ] ); // 随机交换  
    T pivot = _elem[ lo ]; int mi = lo;  
  
    for ( int k = lo + 1; k <= hi; k++ ) // 自左向右考查每个[k]  
        if ( _elem[ k ] < pivot ) // 若 [k] 小于轴点，则将其  
            swap( _elem[ ++mi ], _elem[ k ] ); // 与 [mi] 交换，L 向右扩展  
    swap( _elem[ lo ], _elem[ mi ] ); // 候选轴点归位（从而名副其实）  
    return mi; // 返回轴点的秩  
}
```

实例



12. 排序

选取
众数

邓俊辉

deng@tsinghua.edu.cn

选取与中位数

❖ **k-selection** 在任意一组可比较大小的元素中，如何由小到大，找到次序为k者？

亦即，在这组元素的非降排序序列s中，找出 $s[k]$

`// Excel : large(range, rank)`

❖ **median** 长度为n的有序序列s中，元素 $s[\lfloor \frac{n}{2} \rfloor]$ 称作中位数 //数值上可能有重复

在任意一组可比较大小的元素中，如何找到中位数？

`// Excel : median(range)`



❖ 中位数是k-选取的一个特例；稍后将看到，也是其中难度最大者

众数

❖ **majority** 无序向量中，若有一半以上元素同为 m ，则称之为众数

在{ 3, 5, 2, 3, 3 }中，众数为3；然而

在{ 3, 5, 2, 3, 3, 0 }中，却无众数

❖ 平凡算法 排序 + 扫描

但进一步地 若限制时间不超过 $O(n)$ ，附加空间不超过 $O(1)$ 呢？

❖ 必要性 众数若存在，则亦必中位数

❖ 事实上 只要能够找出中位数，即不难验证它是否众数

```
template <typename T> bool majority( Vector<T> A, T & maj )
```

```
{ return majEleCheck( A, maj = median( A ) ); }
```

必要条件

- ❖ 然而 在高效的中位数算法未知之前，如何确定众数的候选呢？
- ❖ `mode` 众数若存在，则亦必频繁数 //Excel : mode(range)

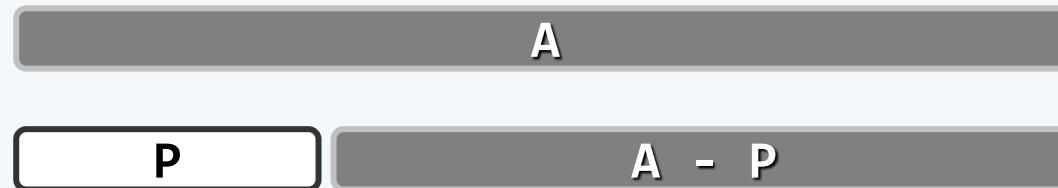
```
template <typename T> bool majority( Vector<T> A, T & maj )
{
    return majEleCheck( A, maj = mode( A ) );
}
```
- ❖ 同样地 mode() 算法难以兼顾时间、空间的高效
- ❖ 可行思路 借助更弱但计算成本更低的必要条件，选出唯一的候选者

```
template <typename T> bool majority( Vector<T> A, T & maj )
{
    return majEleCheck( A, maj = majEleCandidate( A ) );
}
```

减而治之

❖ 若在向量A的前缀P（ $|P|$ 为偶数）中，元素x出现的次数恰占半数，则

A有众数仅当，对应的后缀A - P有众数m，且m就是A的众数



❖ 既然最终总要花费 $\Theta(n)$ 时间做验证，故而只需考虑A的确含有众数的两种情况：

1. 若 $x = m$ ，则在排除前缀P之后，m与其它元素在数量上的差距保持不变

（从浓度50%的盐水中渗析出50%的一部分，剩余部分的浓度仍为50%）

2. 若 $x \neq m$ ，则在排除前缀P之后，m与其它元素在数量上的差距不致缩小

算法

```

❖ template <typename T> T majEleCandidate( Vector<T> A ) {
    T maj; //众数候选者

    // 线性扫描：借助计数器c，记录maj与其它元素的数量差额
    for ( int c = 0, i = 0; i < A.size(); i++ )
        if ( 0 == c ) { //每当c归零，都意味着此时的前缀P可以剪除
            maj = A[i]; c = 1; //众数候选者改为新的当前元素
        } else //否则
            maj == A[i] ? c++ : c--;
    return maj; //至此，原向量的众数若存在，则只能是maj —— 尽管反之不然
}

```

❖ 若将众数的**标准**从**一半以上**改作**至少一半**，算法需做什么调整？

12. 排序

选取

中位数

你去问问你琏二婶子，正月里请吃年酒的日子拟了没有。

若拟定了，叫书房里明白开了单子来，咱们再请时，就

不能重犯了。旧年不留心重了几家，不说咱们不留神，

倒象两宅商议定了送虚情怕费事一样。

邓俊辉

deng@tsinghua.edu.cn

归并向量的中位数

❖ 任给已经排序的有序向量 S_1 和 S_2
如何快速找出有序向量 $S = S_1 \cup S_2$ 的中位数？

❖ 蛮力： 归并 S_1 和 S_2 ， 得到有序向量 S
取出 $S[(|S_1| + |S_2|) / 2]$ ， 即是 S 的中位数

❖ 如此， 共需 $\Theta(|S_1| + |S_2|)$ 时间
❖ 这一效率虽不算低， 但毕竟未能充分利用 S_1 和 S_2 的有序性

❖ 以下， 先介绍 $|S_1| = |S_2| = n$ 情况下的算法
然后， 再将该算法推广至一般情况

❖ 新的算法， 依然采用减而治之策略...

等长子向量：构思

♦ 考查： $m_1 = s_1[\lfloor n/2 \rfloor]$, $m_2 = s_2[\lceil n/2 \rceil - 1] = s_2[\lfloor (n-1)/2 \rfloor]$



♦ 若 $m_1 = m_2$ ，则它们同时是 s_1 、 s_2 和 s 的中位数

♦ 若 $m_1 < m_2$ ，则无论 n 为偶为奇，灰色区间或者不是 s 的中位数；或者与 m_1 或 m_2 同为 s 的中位数

这意味着，剪除这些区间之后， s 中位数的数值保持不变

// $m_1 > m_2$ 同理

♦ 总之，每经一次比较，原问题的规模即大致减半 —— 整体不过 $O(\log n)$

等长子向量：实现

❖ template <typename T> //尾递归，可改写为迭代形式

```
T median( Vector<T> & S1, int lo1, Vector<T> & S2, int lo2, int n ) {  
    if ( n < 3 ) return trivialMedian( S1, lo1, n, S2, lo2, n ); //递归基  
    int mi1 = lo1 + n/2, mi2 = lo2 + (n - 1)/2; //长度减半  
    if ( S1[ mi1 ] < S2[ mi2 ] ) //取S1右半、S2左半  
        return median( S1, mi1, S2, lo2, n + lo1 - mi1 );  
    else if ( S1[ mi1 ] > S2[ mi2 ] ) //取S1左半、S2右半  
        return median( S1, lo1, S2, mi2, n + lo2 - mi2 );  
    else  
        return S1[ mi1 ];  
}
```

任意子向量：实现 (1/2)

```
template <typename T>

T median ( Vector<T> & S1, int lo1, int n1, Vector<T> & S2, int lo2, int n2 ) {

    if ( n1 > n2 )

        return median( S2, lo2, n2, S1, lo1, n1 ); //确保n1 <= n2

    if ( n2 < 6 )

        return trivialMedian( S1, lo1, n1, S2, lo2, n2 ); //递归基

    if ( 2 * n1 < n2 )

        return median( S1, lo1, n1, S2, lo2 + (n2-n1-1)/2, n1+2-(n2-n1)%2 );
```

任意子向量：实现 (2/2)

```

int mi1 = lo1 + n1/2, mi2a = lo2 + (n1 - 1)/2, mi2b = lo2 + n2 - 1 - n1/2;

if ( s1[ mi1 ] > s2[ mi2b ] ) //取s1左半、s2右半

    return median( s1, [lo1], n1 / 2 + 1, s2, [mi2a], n2 - (n1 - 1) / 2 );

else if ( s1[ mi1 ] < s2[ mi2a ] ) //取s1右半、s2左半

    return median( s1, [mi1], (n1 + 1) / 2, s2, [lo2], n2 - n1 / 2 );

else //S1保留，S2左右同时缩短

    return median( s1, [lo1], n1, s2, [mi2a], n2 - (n1 - 1) / 2 * 2 );

} //O( log( min( n1, n2 ) ) )——可见，实际上等长版本才是难度最大的

```

12. 排序

选取

QuickSelect

世兄的才名，弟所素知的。在世兄是数万人里头选出来最清最雅的，至于弟乃庸庸碌碌一等愚人，忝附同名，殊觉玷辱了这两个字。

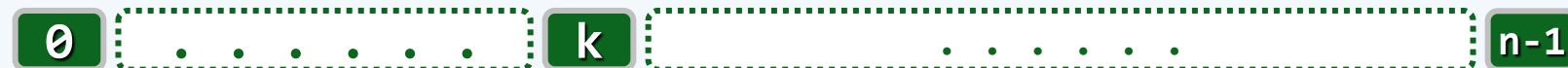
邓俊辉

deng@tsinghua.edu.cn

尝试：蛮力

❖ 对A排序 // $O(n \log n)$

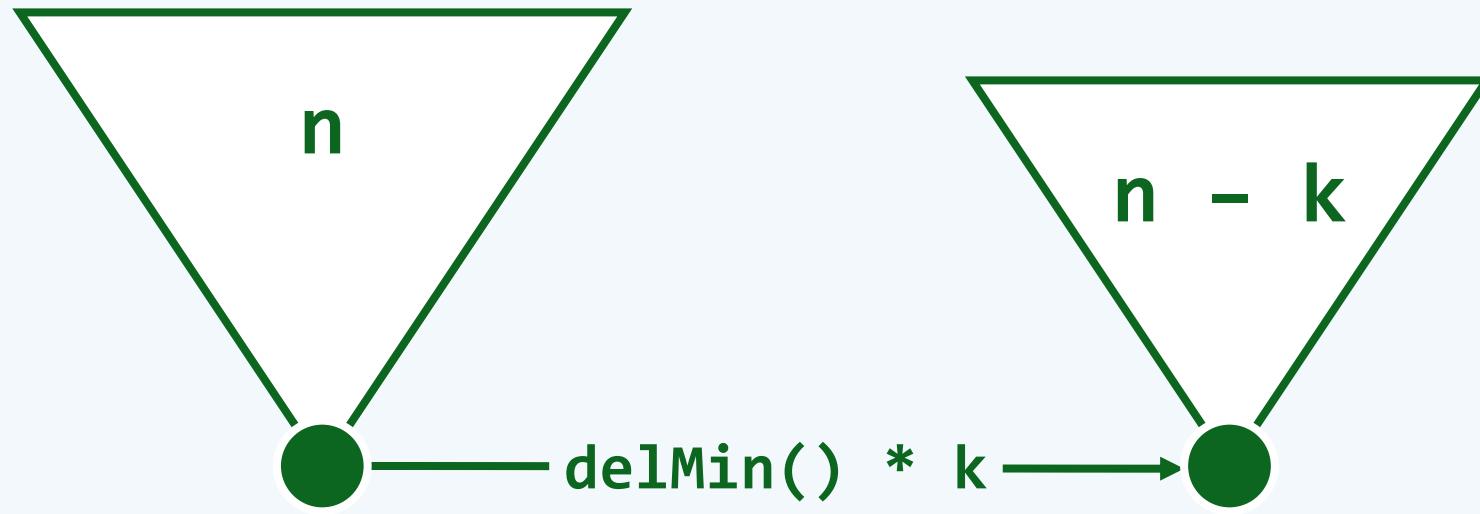
从前向后行进k步 // $O(k) = O(n)$



尝试：堆 (A)

❖ 将所有元素组织为小顶堆 $// O(n)$

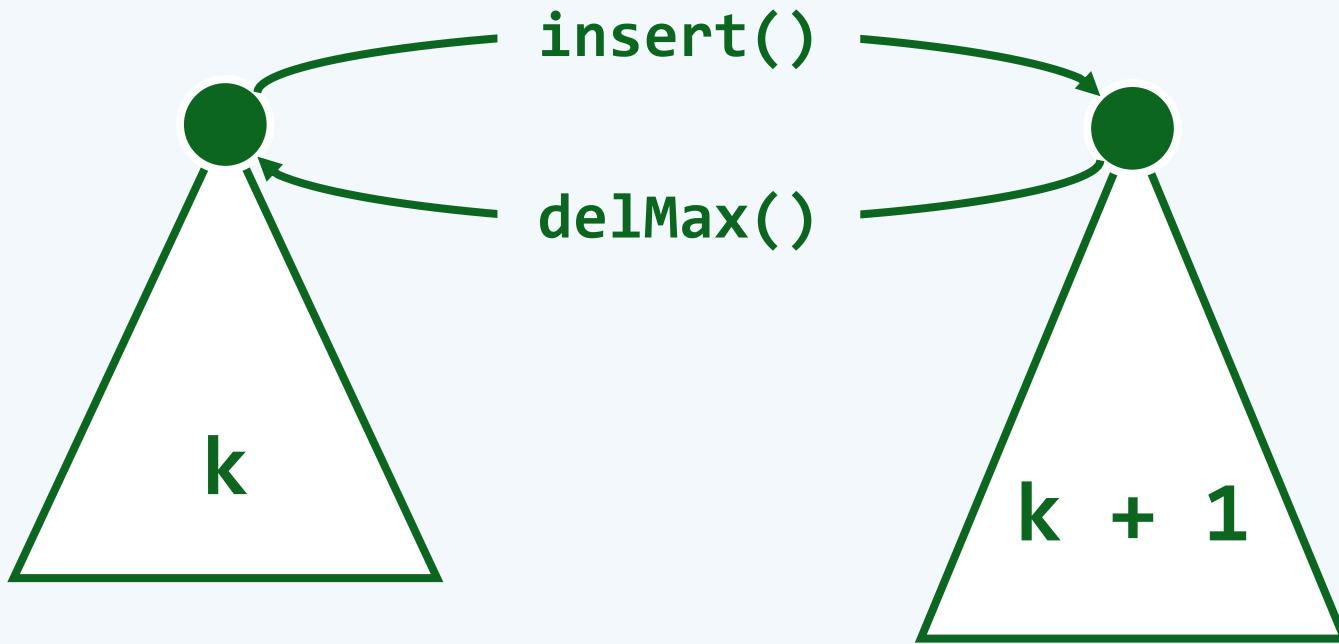
连续调用 k 次 `delMin()` $// O(k \log n)$



尝试：堆 (B)

❖ 任选（比如前） k 个元素，组织为大顶堆 // $\Theta(k)$

对于剩余的 $n - k$ 个元素，各调用一次`insert()`和`delMax()` // $\Theta(2*(n - k)*\log k)$



尝试：堆 (c)

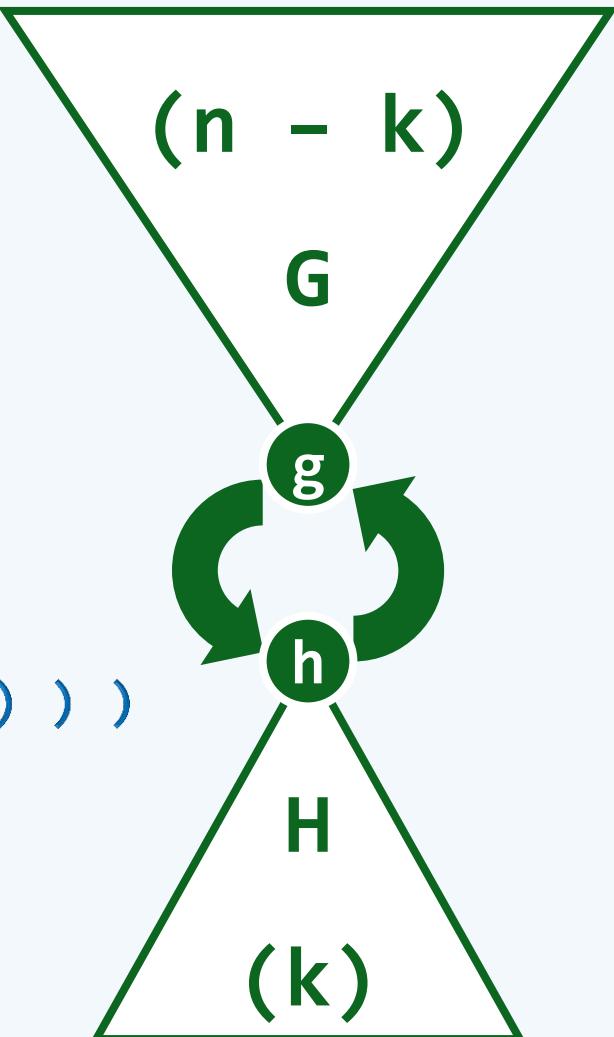
❖ **H**: 任取 k 个元素，组织为 **大顶堆** $\text{//} \theta(k)$

G: 其余 $n - k$ 个元素，组织为 **小顶堆** $\text{//} \theta(n - k)$

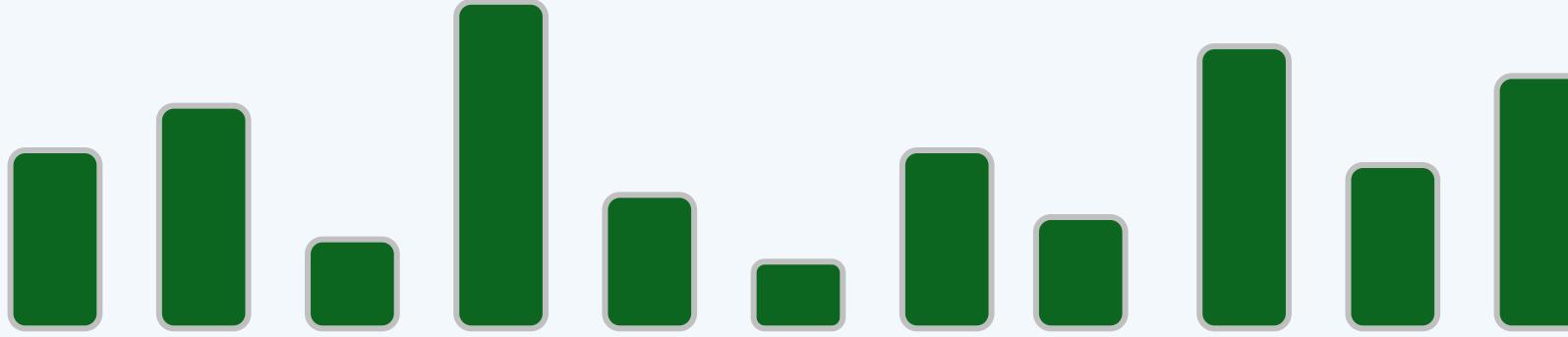
❖ 反复地： 比较 **h** 和 **g** $\text{//} \theta(1)$

如有必要，**交换**之 $\text{//} \theta(2 \times (\log k + \log(n - k)))$

直到： **$h \leq g$** $\text{//} \theta(\min(k, n - k))$



尝试：计数排序



下界与最优

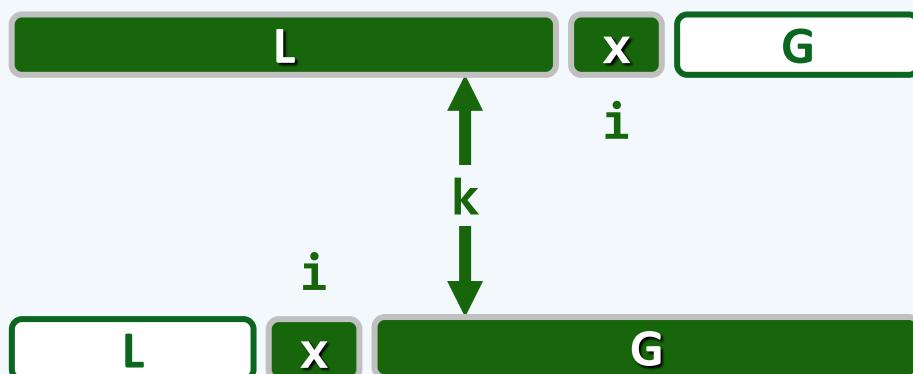
- ❖ 是否存在更快的算法？
- ❖ $\Omega(n)$ ！
- ❖ 所谓第k大，是相对于序列整体而言
在访问每个元素至少一次之前，绝无可能确定
- ❖ 反过来，是否存在 $O(n)$ 的算法？

quickSelect()

```

template <typename T> void quickSelect( Vector<T> & A, Rank k ) {
    for ( Rank lo = 0, hi = A.size() - 1; lo < hi; ) {
        Rank i = lo, j = hi; T pivot = A[lo];
        while ( i < j ) { // $\theta(hi - lo + 1) = \theta(n)$ 
            while ( i < j && pivot <= A[j] ) j--; A[i] = A[j];
            while ( i < j && A[i] <= pivot ) i++; A[j] = A[i];
        } //assert: i == j
        A[i] = pivot;
        if ( k <= i ) hi = i - 1;
        if ( i <= k ) lo = i + 1;
    } //A[k] is now a pivot
}

```



12. 排序

选取

LinearSelect

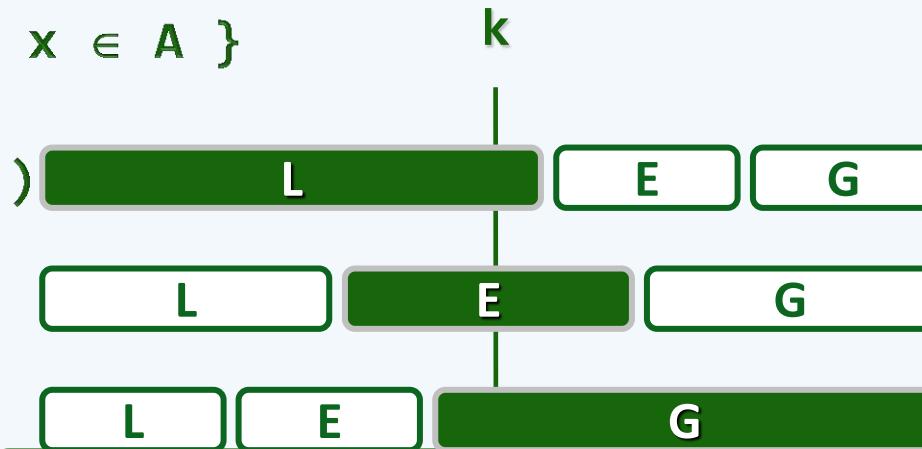
邓俊辉

deng@tsinghua.edu.cn

linearSelect()

Let Q be a small constant

0. if ($n = |A| < Q$) return trivialSelect(A, k)
1. else divide A evenly into $\lceil n/Q \rceil$ subsequeces (each of size Q)
2. Sort each subsequence and determine $\lceil n/Q \rceil$ medians //e.g. by insertionsort
3. Call linearSelect to find M , median of the medians //by recursion
4. Let $L / E / G = \{ x \mid x < M \text{ or } x = M \text{ or } x > M \mid x \in A \}$
5. if ($k \leq |L|$) return linearSelect(L, k)
- if ($k \leq |L| + |E|$) return M
- return linearSelect($G, k - |L| - |E|$)



复杂度

- ❖ 将`linearSelect()`算法的运行时间记作 $T(n)$
- ❖ 第0步 : $\mathcal{O}(1) = \mathcal{O}(Q \log Q)$ //递归基：序列长度 $|A| \leq Q$
- ❖ 第1步 : $\mathcal{O}(n)$ //子序列划分
- ❖ 第2步 : $\mathcal{O}(n) = \mathcal{O}(1) \times n/Q$ //子序列各自排序，并找到中位数
- ❖ 第3步 : $T(\lfloor n/Q \rfloor)$ //从 $\lfloor n/Q \rfloor$ 个中位数中，递归地找到全局中位数
- ❖ 第4步 : $\mathcal{O}(n)$ //划分子集 `L/E/G`，并分别计数 —— 一趟扫描足矣
- ❖ 第5步 : $T(\lfloor 3n/4 \rfloor)$ //为什么...

复杂度

❖ 在某种意义上，如上所确定的 M 必然 不偏不倚

至少各有 $n/4$ 个元素， 不小于 / 不大于 M

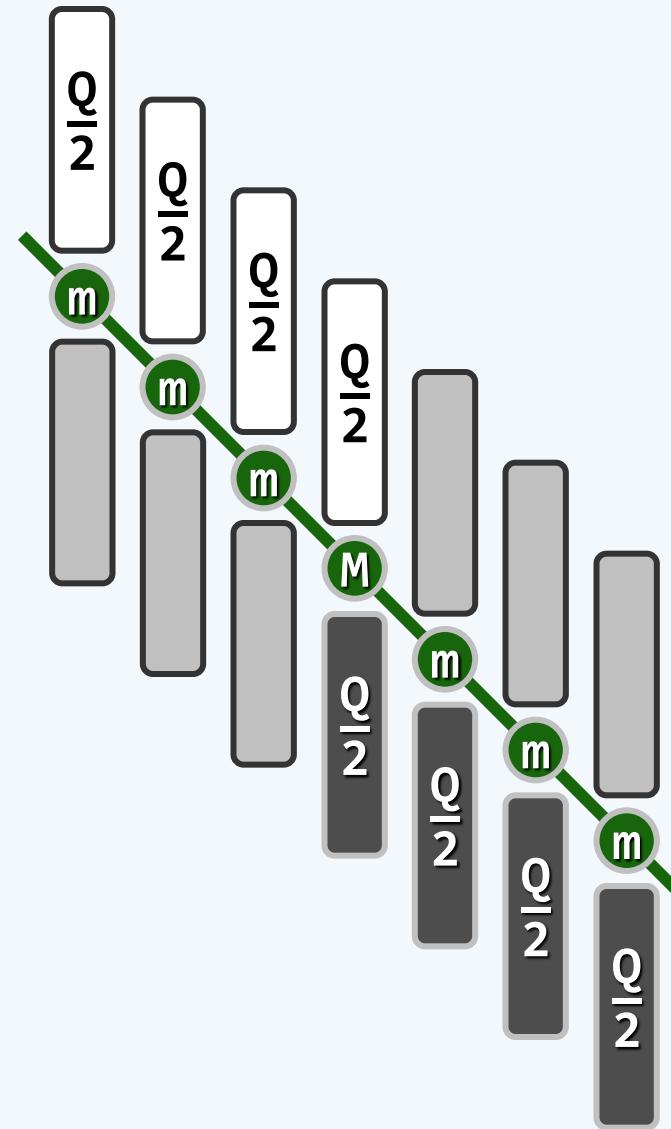
❖ n/Q 个中位数中， 至少半数 不小于 M

而它们又各自不大于至少 $Q/2$ 个元素

$$\frac{n}{Q} / 2 * \frac{Q/2}{2} = n/4$$

❖ $\min(|L|, |G|) + |E| \geq n/4$

$\max(|L|, |G|) \leq 3n/4$



复杂度

❖ $T(n) = \Theta(n) + T(\lfloor n/Q \rfloor) + T(\lceil 3n/4 \rceil)$

❖ 为使之解作线性函数，只需保证

$$\lfloor n/Q \rfloor + \lceil 3n/4 \rceil < n$$

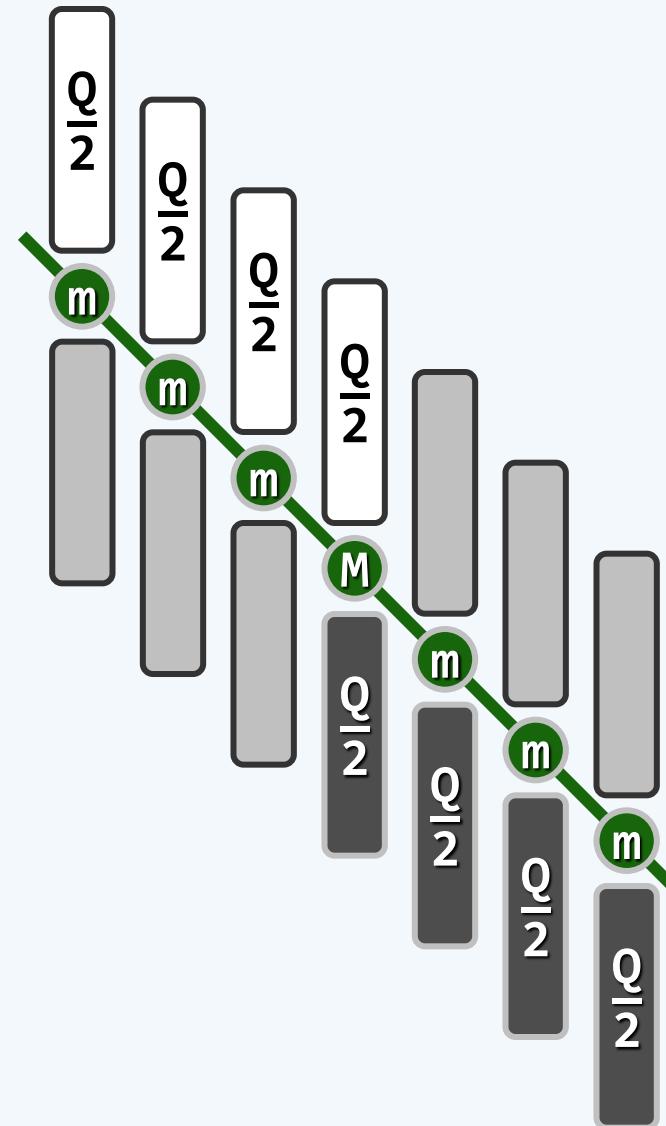
或等价地

$$\frac{1}{Q} + \frac{3}{4} < 1$$

❖ 比如，若取 $Q = 5$ ，则存在常数 c ，使得

$$T(n) = cn + T(\lfloor n/5 \rfloor) + T(\lceil 3n/4 \rceil)$$

$$T(n) = \Theta(20c n) = \Theta(n)$$



12. 排序

希尔排序

框架+实例

邓俊辉

deng@tsinghua.edu.cn

Shellsort

❖ Donald L. Shell, 1959 : 将整个序列视作一个矩阵，逐列各自排序 w-sorting

❖ 递减增量 diminishing increment

由粗到细：重排矩阵，使其更窄，再次逐列排序 w-ordered

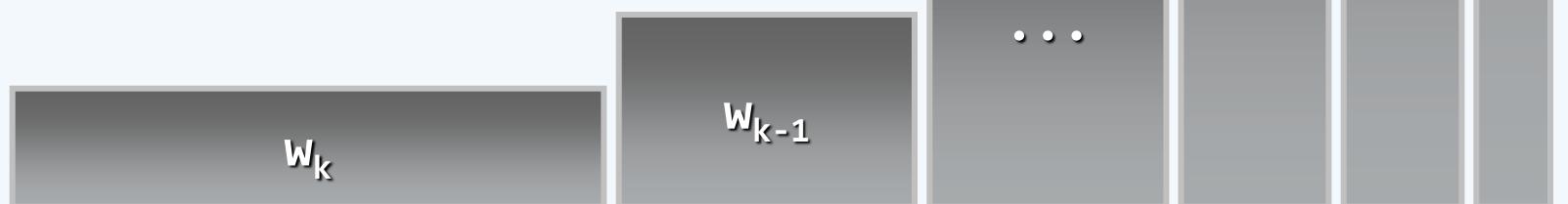
逐步求精：如此往复，直至矩阵变成一列 1-sorting

❖ 步长序列 step sequence：由各矩阵宽度构成的逆序列

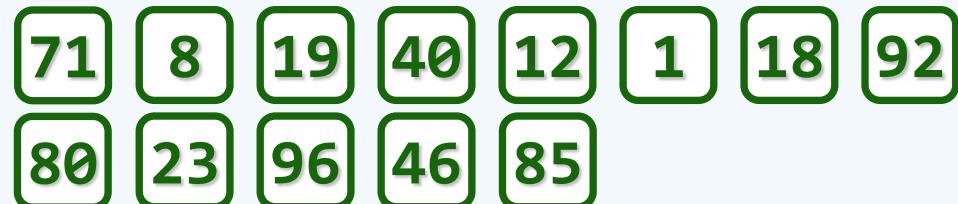
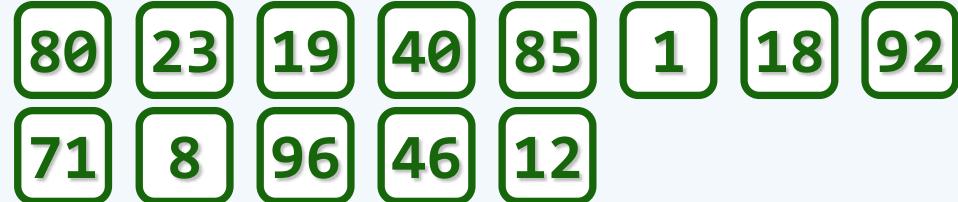
$$\pi = \{ w_1 = 1, w_2, w_3, \dots, w_k, \dots \}$$

❖ 正确性：最后一次迭代，等同于全排序

1-ordered = sorted



实例 : $w_5 = 8$



实例 : $w_4 = 5$

1 8 19 40 12 71 18 85 80 23 96 46 92

71 8 19 40 12
1 18 92 80 23
96 46 85

1 8 19 40 12
71 18 85 80 23
96 46 92

71 8 19 40 12 1 18 92 80 23 96 46 85

实例 : $w_3 = 3$

1 8 19 40 12 71 18 85 80 23 96 46 92

1 8 19
40 12 71
18 85 80
23 96 46
92

1 8 19
18 12 46
23 85 71
40 96 80
92

1 8 19 18 12 46 23 85 71 40 96 80 92

实例 : $w_2 = 2$

1 8 12 18 19 40 23 46 71 80 92 85 96

1 8
19 18
12 46
23 85
71 40
96 80
92

1 8
12 18
19 40
23 46
71 80
92 85
96

1 8 19 18 12 46 23 85 71 40 96 80 92

实例 : $w_1 = 1$

1

8

12

18

19

40

23

46

71

80

92

85

96

1

8

12

18

19

23

40

46

71

80

85

92

96

12. 排序

希尔排序

输入敏感性

邓俊辉

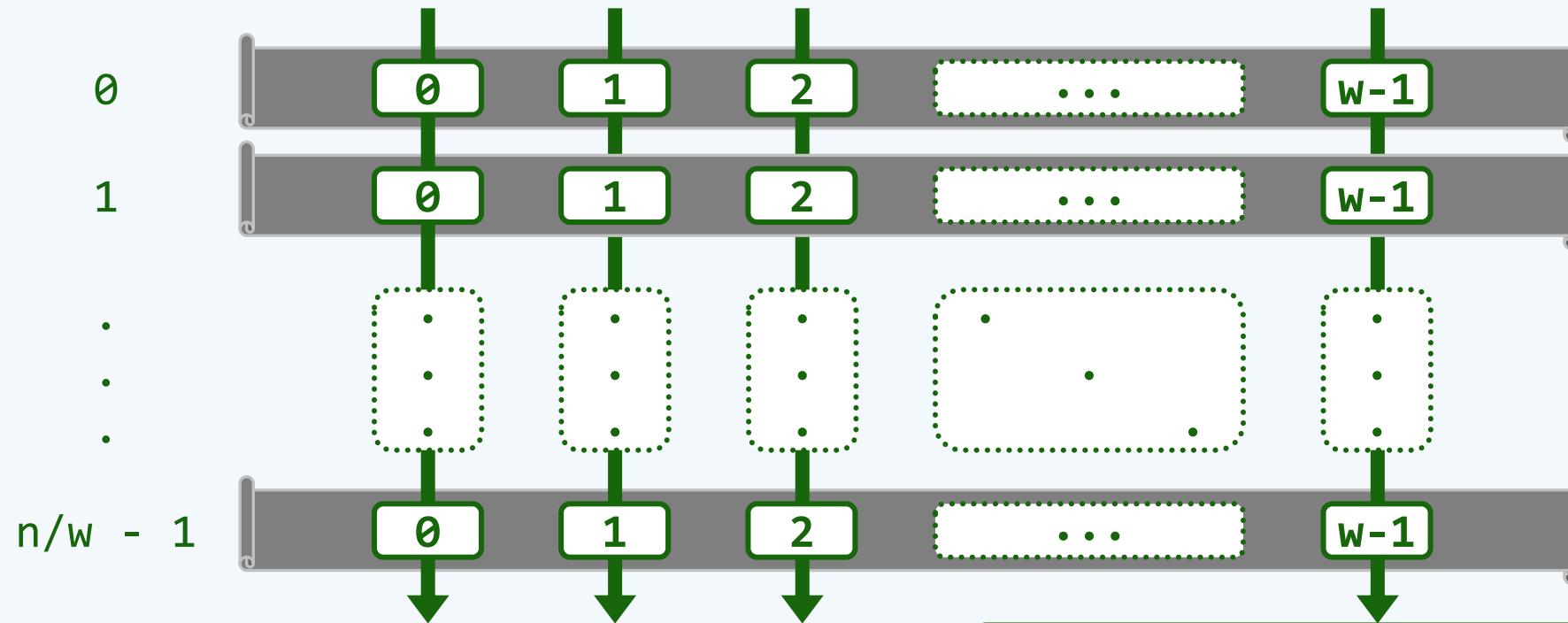
deng@tsinghua.edu.cn

Call-by-rank

❖ 如何实现矩阵重排？莫非，需要使用二维向量？实际上，借助一维向量足矣

❖ 在每步迭代中，若当前的矩阵宽度为 w ，则

$$B[i][j] = A[iw + j] \quad \text{或} \quad A[k] = B[k / w][k \% w]$$



Input-sensitivity

- ❖ 各列内部的排序如何实现？
- ❖ 必须采用**输入敏感**的算法，以保证
有序性可**持续**改善，且**总体**成本足够**低廉**
- ❖ 比如，**insertionsort**
其实际运行时间，更多地取决于
输入序列所含**逆序对**的总数



Step Sequences

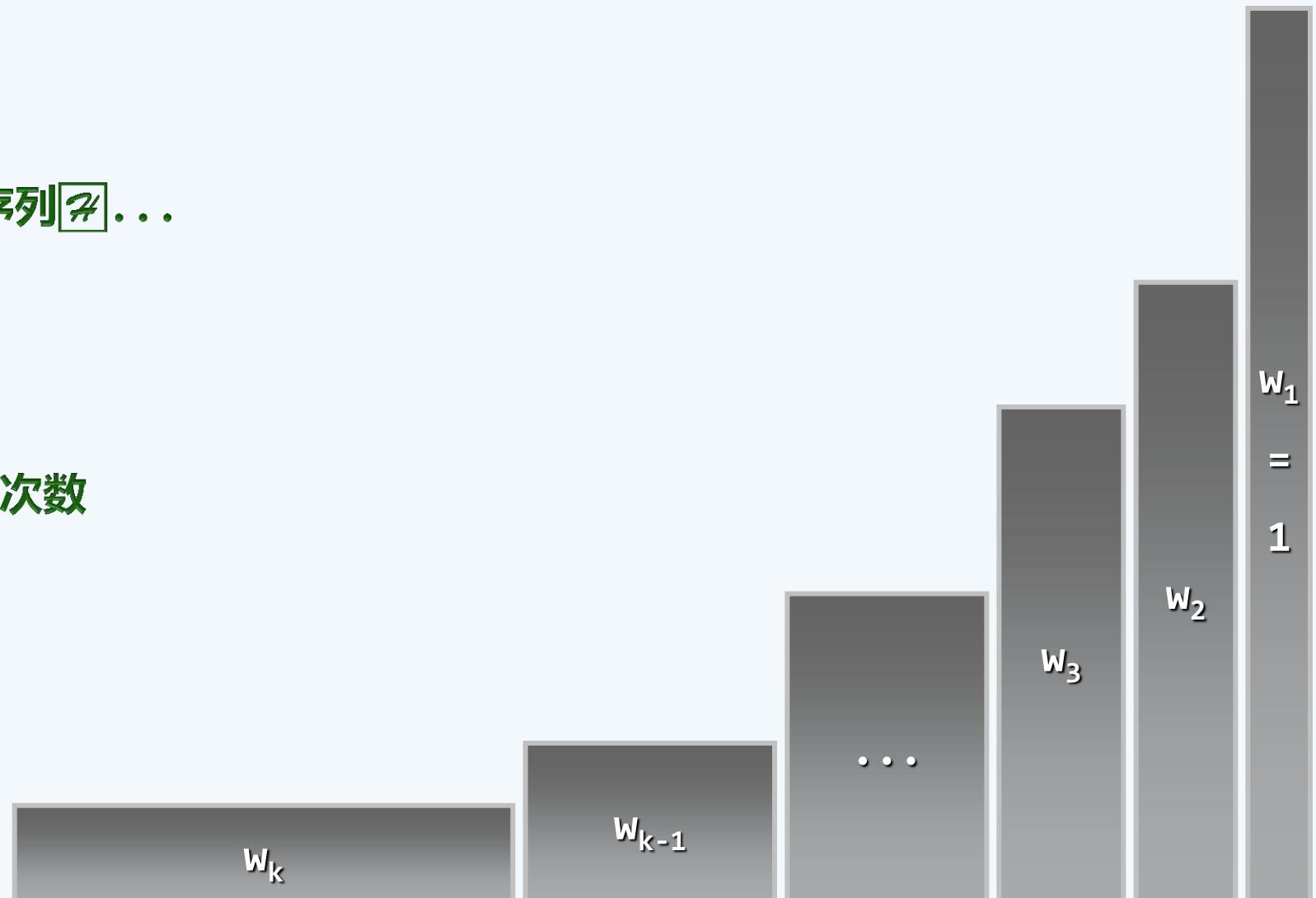
❖ Shellsort的总体效率

取决于具体使用何种步长序列 $\boxed{?} \dots$

❖ 主要考察和评测

1. 比较操作、移动操作的次数
2. 收敛的速度，或反过来

迭代的轮数



12. 排序

希尔排序

Shell序列

邓俊辉

deng@tsinghua.edu.cn

Shell's Sequence

❖ $\pi_{\text{shell}} = \{ 1, 2, 4, 8, \dots, 2^k, \dots \}$ //Shell 1959

❖ 采用 π_{shell} ，在最坏情况下需要运行 $\Omega(n^2)$ 时间

❖ 考查由子序列 $A = \text{unsort}[0, 2^{N-1})$ 和 $B = \text{unsort}[2^{N-1}, 2^N)$ 交错而成的：

<u>11</u>	4	<u>14</u>	3	<u>10</u>	0	<u>15</u>	1	<u>9</u>	6	<u>8</u>	7	<u>13</u>	2	<u>12</u>	5
-----------	---	-----------	---	-----------	---	-----------	---	----------	---	----------	---	-----------	---	-----------	---

❖ 在 **2-sorting** 刚结束时， A 和 B 必然各自有序：

<u>8</u>	0	<u>9</u>	1	<u>10</u>	2	<u>11</u>	3	<u>12</u>	4	<u>13</u>	5	<u>14</u>	6	<u>15</u>	7
----------	---	----------	---	-----------	---	-----------	---	-----------	---	-----------	---	-----------	---	-----------	---

❖ 其中的逆序对仍然很多，**1-sorting** 仍需 $1 + 2 + \dots + 2^{N-1} = \Omega(n^2)$ 时间

❖ 根源在于， π_{shell} 中各项并不互素，甚至相邻项也非互素

12. 排序

希尔排序

逆序对

邓俊辉

deng@tsinghua.edu.cn

Postage Problem

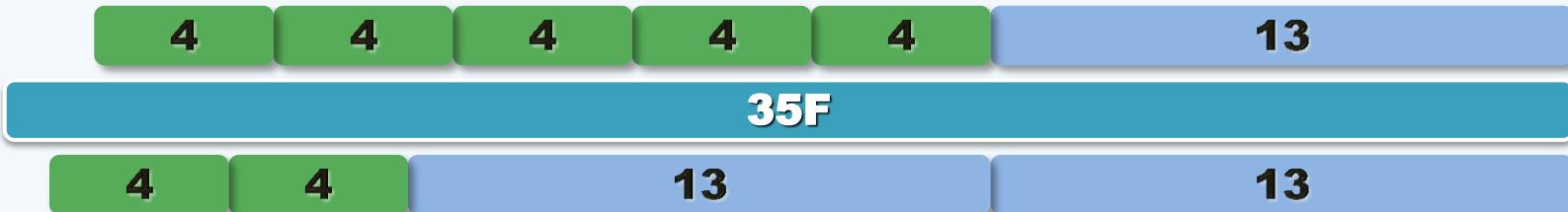
❖ The postage for a letter is **50F**, and a postcard **35F**

But there are only stamps of **4F** and **13F** available

❖ Possible to stamp the letter and the postcard **EXACTLY**?



❖ How about other postages?



❖ For each postage P , determine whether $P \in \{ n \cdot 4 + m \cdot 13 \mid n, m \in \mathbb{N} \}$

Linear Combination

- ❖ Let $g, h \in \mathcal{N}$
- ❖ For any $n, m \in \mathcal{N}$, $n \cdot g + m \cdot h$ is called a **linear combination** of g and h
- ❖ Denote $\mathbf{C}(g, h) = \{ ng + mh \mid n, m \in \mathcal{N} \}$
 $\mathbf{N}(g, h) = \mathcal{N} \setminus \mathbf{C}(g, h)$ //numbers that are **NOT** combinations of g and h
 $\mathbf{x}(g, h) = \max\{ \mathbf{N}(g, h) \}$ //always exists?
- ❖ Theorem: when g and h are **relatively prime**, we have

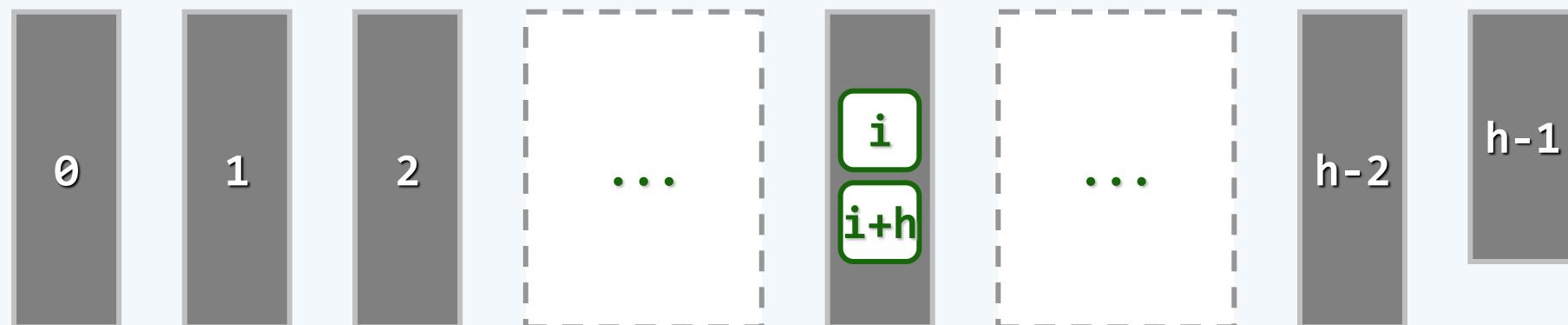
$$\mathbf{x}(g, h) = (g - 1) \cdot (h - 1) - 1 = gh - g - h$$
- ❖ e.g. $\mathbf{x}(3, 7) = 11$, $\mathbf{x}(4, 9) = 23$, $\mathbf{x}(\boxed{4}, \boxed{13}) = \boxed{35}$, $\mathbf{x}(5, 14) = 51$

h-sorting & h-ordered

- ❖ Let $h \in \mathbb{N}$. A sequence $S[0, n)$ is called **h-ordered** if

$$S[i] \leq S[i + h] \text{ holds for } 0 \leq i < n - h$$

- ❖ A **1-ordered** sequence is sorted
- ❖ **h-sorting**: an h-ordered sequence is obtained by
 - arranging S into a 2D matrix with h columns and
 - sorting each column respectively

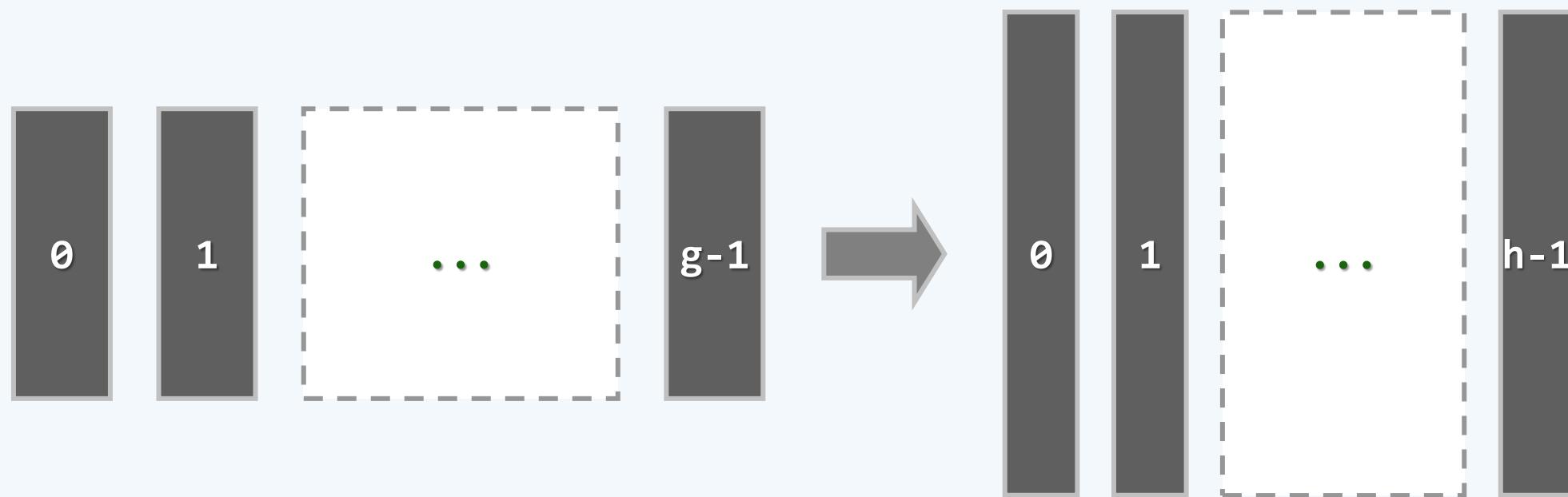


Theorem K

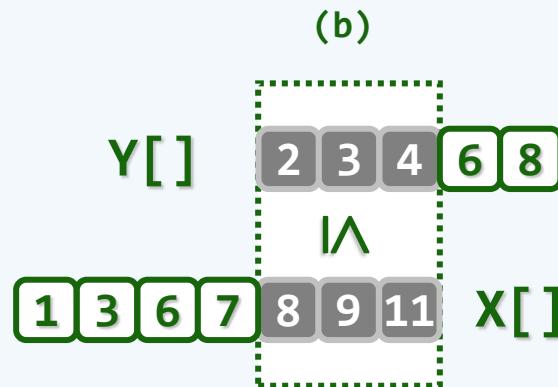
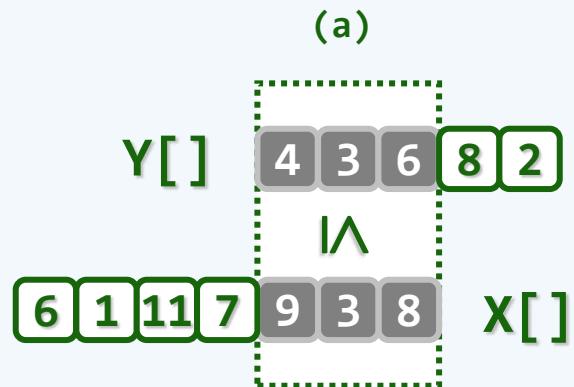
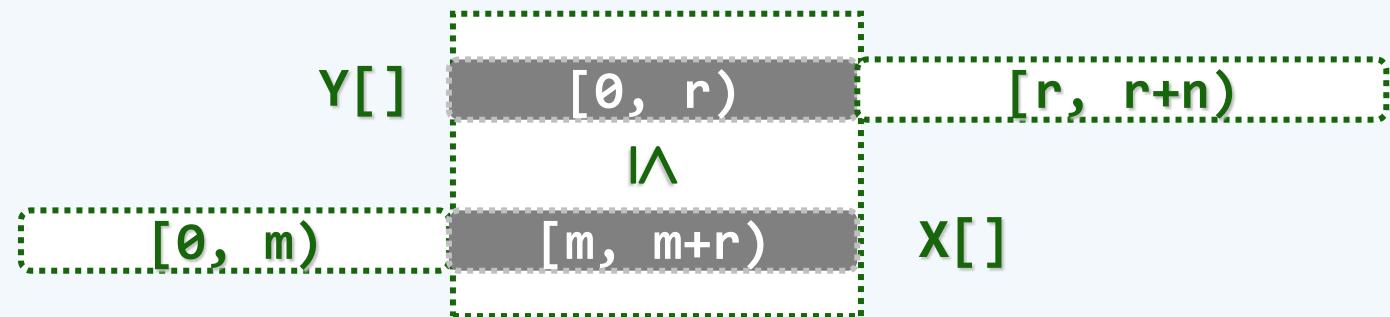
❖ [Knuth, ACP Vol.3 p.90]

//习题解析[12-12, 12-13]

A **g-ordered** sequence **REMAINS** g-ordered after being **h-sorted**

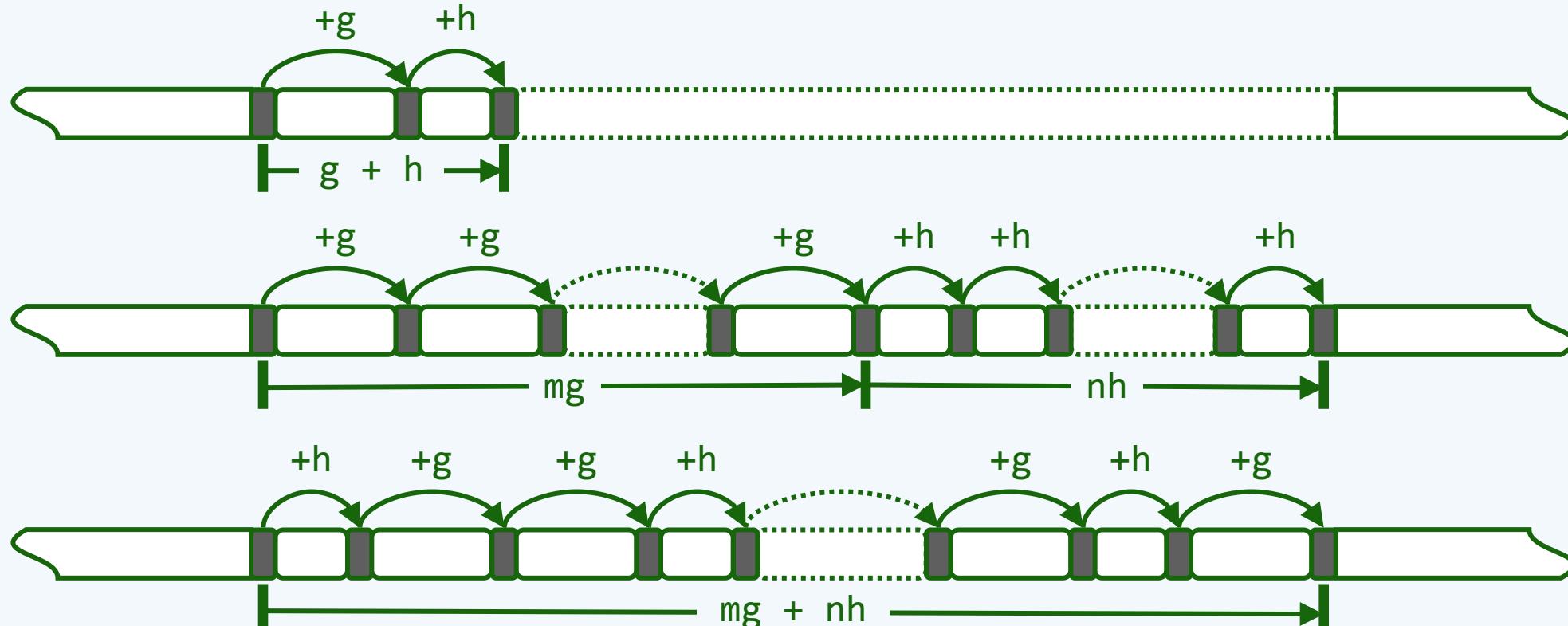


Lemma L



Linear Combination

- ❖ A sequence that is both g -ordered and h -ordered is called (g, h) -ordered, which must be both $(g + h)$ -ordered and $(mg + nh)$ -ordered for any $m, n \in \mathbb{N}$



Inversion

❖ Let $S[0, n)$ be a (g, h) -ordered sequence, where g & h are relatively prime

❖ Then for all elements $S[i]$ and $S[j]$, we have

$$j - i \geq x(g, h) + 1 = (g - 1) \cdot (h - 1) \quad \text{only if} \quad S[i] \leq S[j]$$

❖ This implies that to the **RIGHT** of each element,

only the next $x(g, h)$ elements could be smaller



❖ There would be no more than $n \cdot x(g, h)$ inversions altogether

12. 排序

希尔排序

PS序列

邓俊辉

deng@tsinghua.edu.cn

d-Sorting an $\mathcal{O}(d)$ -Ordered Sequence in $\mathcal{O}(dn)$ Time

❖ If g and h are relatively prime and are both in $\mathcal{O}(d)$

we can d-sort the sequence in $\mathcal{O}(dn)$ time ...

- re-arrange the sequence as a 2D matrix with d columns
- each element is swapped with $\mathcal{O}((g - 1) \cdot (h - 1)/d) = \mathcal{O}(d)$ elements

❖ Since this holds for all elements, $\mathcal{O}(dn)$ steps are enough



PS Sequence

❖ Papernov & Stasevic, 1965

//also called Hibbard's sequence

$$\mathcal{H}_{PS} = \mathcal{H}_{Shell} - 1 = \{ 2^k - 1 \mid k \in \mathcal{N} \} = \{ 1, 3, 7, 15, 31, 63, 127, 255, \dots \}$$

❖ Different items **may not** be relatively prime, e.g. $h_{2k} = h_k \cdot (h_k + 2)$

But **adjacent** items **must** be, since $h_{k+1} - 2 \cdot h_k \equiv 1$

❖ Shellsort with \mathcal{H}_{ps} needs

- $\mathcal{O}(\log n)$ outer iterations and
- $\mathcal{O}(n^{3/2})$ time to sorts a sequence of length n //Why ...

$t < k$

❖ Let h_t be the h closest to \sqrt{n} and hence $h_t \approx \sqrt{n} = \Theta(n^{1/2})$

1) Consider those iterations for $\{ h_k \mid t < k \} = \{ \overleftarrow{h_{t+1}, h_{t+2}, \dots, h_m} \}$

$k \leq t$

$h_k \leq h_t$

\therefore there would be $\mathcal{O}(n/h_k)$ elements in each of the h_k columns

\therefore we can **insertionsort** each column in $\mathcal{O}((n/h_k)^2)$ time

\therefore each h_k -sorting costs $\mathcal{O}(n^2/h_k)$ time

\therefore all these iterations cost time of

$$\mathcal{O}(2 \times n^2/h_t) = \mathcal{O}(n^{3/2})$$

$t < k$
 $h_t < h_k$

$k = t$

$h_k = h_t$

$k \leq t$

2) Consider those iterations for $\{ h_k \mid k \leq t \} = \{ \overleftarrow{h_1, h_2, \dots, h_t} \}$

$\because [h_{k+1}]$ and $[h_{k+2}]$ are relatively prime and are both in $\mathcal{O}([h_k])$

$k \leq t$

\therefore each h_k -sorting costs $\mathcal{O}(n \times h_k)$ time

$h_k \leq h_t$

\therefore all these iterations cost $\mathcal{O}(n \times 2 \cdot h_t) = \mathcal{O}(n^{3/2})$ time

❖ This upper bound is **TIGHT**

❖ How about the average cases?

- $\mathcal{O}(n^{5/4})$ based on simulations
- but not proved yet

$t < k$
 $h_t < h_k$

$k = t$
 $h_k = h_t$

12. 排序

希尔排序

Pratt序列

邓俊辉

deng@tsinghua.edu.cn

Pratt's Sequence, 1971

$$\mathcal{H}_{\text{pratt}} = \{2^p \cdot 3^q \mid p, q \in \mathbb{N}\} = \{1, 2, 3, 4, 6, 8, 9, 12, 16, \dots\}$$

❖ With $\mathcal{H}_{\text{pratt}}$,

Shellsort sorts a sequence of length n in $\mathcal{O}(\log^2 n)$ time //Why?

❖ And, wait a minute, but ...

adjacent numbers in $\mathcal{H}_{\text{pratt}}$

are NOT all relatively prime ...

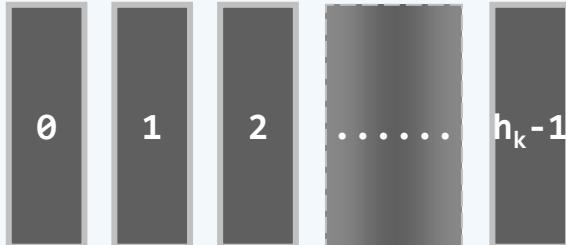
$\mathcal{O}(n \log^2 n)$

❖ From $(2, 3)$ -ordered to 1 -ordered

$$\therefore x(2, 3) = 1$$

\therefore to the right of each element in a $(2, 3)$ -ordered sequence,
only the next element can be smaller

\therefore it costs $\mathcal{O}(n)$ time to sort such a sequence



❖ From $(2h_k, 3h_k)$ -ordered to h_k -ordered

divide S into h_k subsequences, each of which is $(2, 3)$ -ordered

\therefore it costs altogether $\mathcal{O}(n)$ time to sort them resp.

❖ \therefore there are altogether $\mathcal{O}(\log^2 n)$ iterations //why

$$\therefore \text{we need time } \mathcal{O}(n) \times \mathcal{O}(\log^2 n) = \mathcal{O}(n \log^2 n)$$

12. 排序

希尔排序

Sedgewick序列

邓俊辉

deng@tsinghua.edu.cn

Sedgewick's Sequence

❖ Pratt's sequence performs best asymptotically but

- requires too many iterations and hence
- is not good for pre-sorted sequences

❖ Sedgewick's sequence: a combination of PS's sequence with Pratt's

$$\{ 1 \quad 5 \quad 19 \quad 41 \quad 109 \quad 209 \quad 505 \quad 929 \quad 2161 \quad 3905 \quad 8929 \quad 16001 \quad 36289 \quad 64769 \quad \dots \}$$

$$\{ 9 \times 4^k - 9 \times 2^k + 1 \mid k \geq 0 \} \cup \{ 4^k - 3 \times 2^k + 1 \mid k \geq 2 \}$$

- worst $\mathcal{O}(n^{4/3})$ & average $\mathcal{O}(n^{7/6})$
- best performance in practice

❖ Is there a step sequence with $\mathcal{O}(n \log n)$ worst-case performance?

