

第**2**讲
高级数据结构

刘汝佳

目录

- 平衡二叉树
- 可并优先队列
- 线段树和树状数组基础
- RMQ与LCA

一、平衡二叉树

基本BST

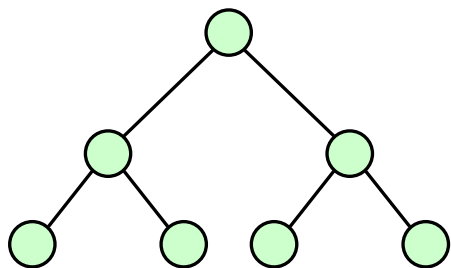
- 基本BST(Binary Search Tree)定义
 - 二叉树
 - 左子树 $<$ 根 $<$ 右子树
 - **递归定义**: 左子树和右子树均是BST
- 基本实现方式
 - 查找: 从根往下走 $O(h)$
 - 插入: 查找失败后 $O(1)$. 总 $O(h)$
 - 删除: 分情况讨论, $O(h)$

为何要平衡

- 越平衡, 各种操作的速度越快
- 什么是平衡: 渐进意义下树高 $h=O(\log n)$
 - 固定结点数, 则树高越小越好

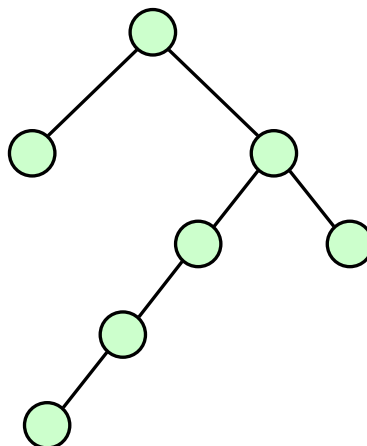
balanced

$$\text{height}(T) = O(\log n)$$



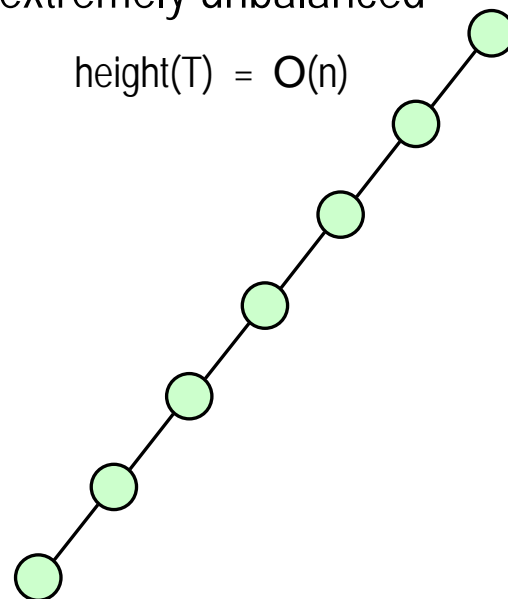
unbalanced

$$\text{height}(T) = O(n^c), 1 > c > 0$$



extremely unbalanced

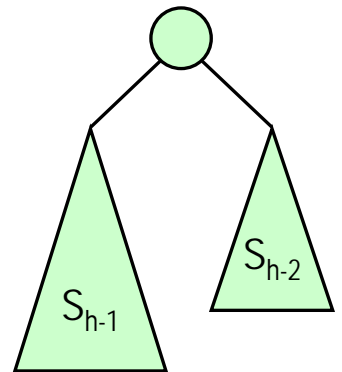
$$\text{height}(T) = O(n)$$



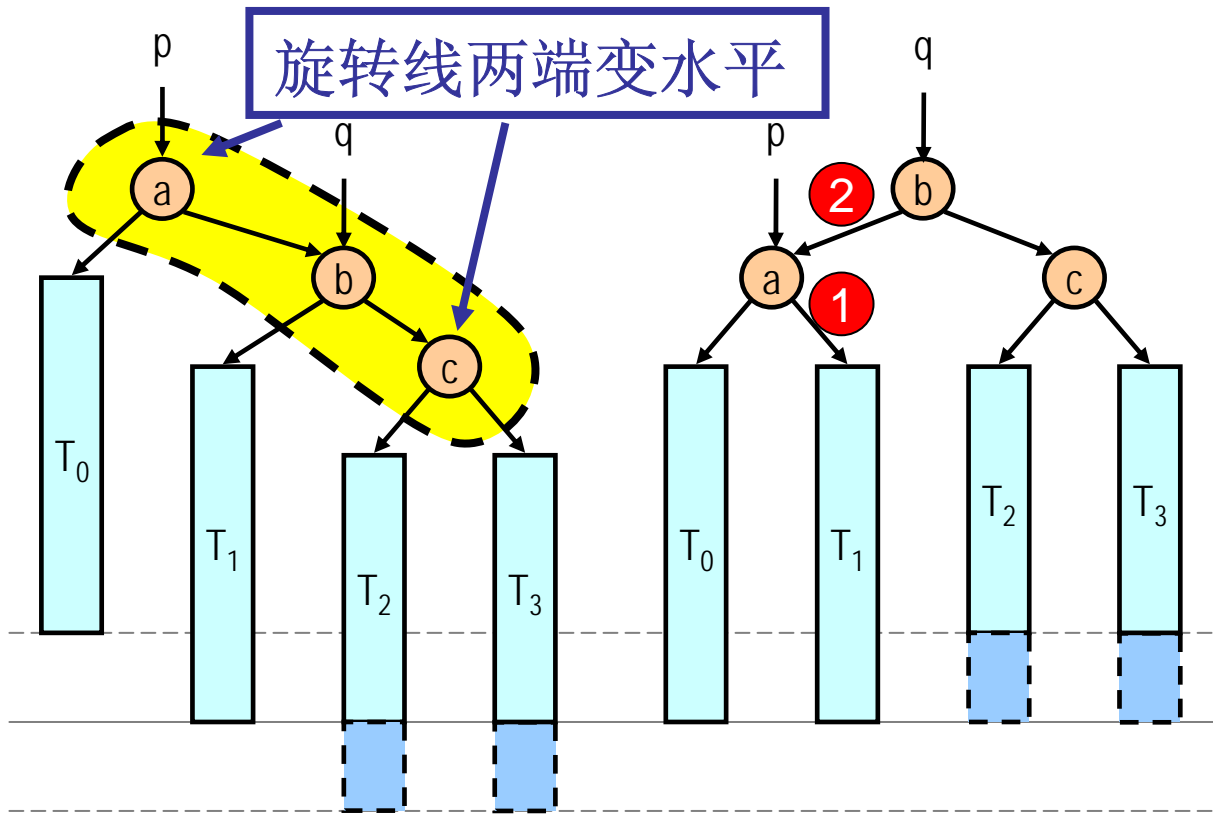
一、AVL树

- **基本思想:** 限制树的形状, 使得满足该限制的树一定是平衡的
- **定义:** 每个结点的左右子树高度差不超过1(空树高度为-1)的排序二叉树称为AVL树
- **AVL树的高度.** 设 $S(n)$ 为高度为 n 的AVL树的最少结点数, 则 $S(n)=S(n-1)+S(n-2)+1$, 而 $S(0)=1$, $S(1)=2$, 归纳得 $S(n) \geq F_{n+3}-1$, 因此

$$h_{\max} \approx 1.44 \log_2 n$$



AVL的单旋转



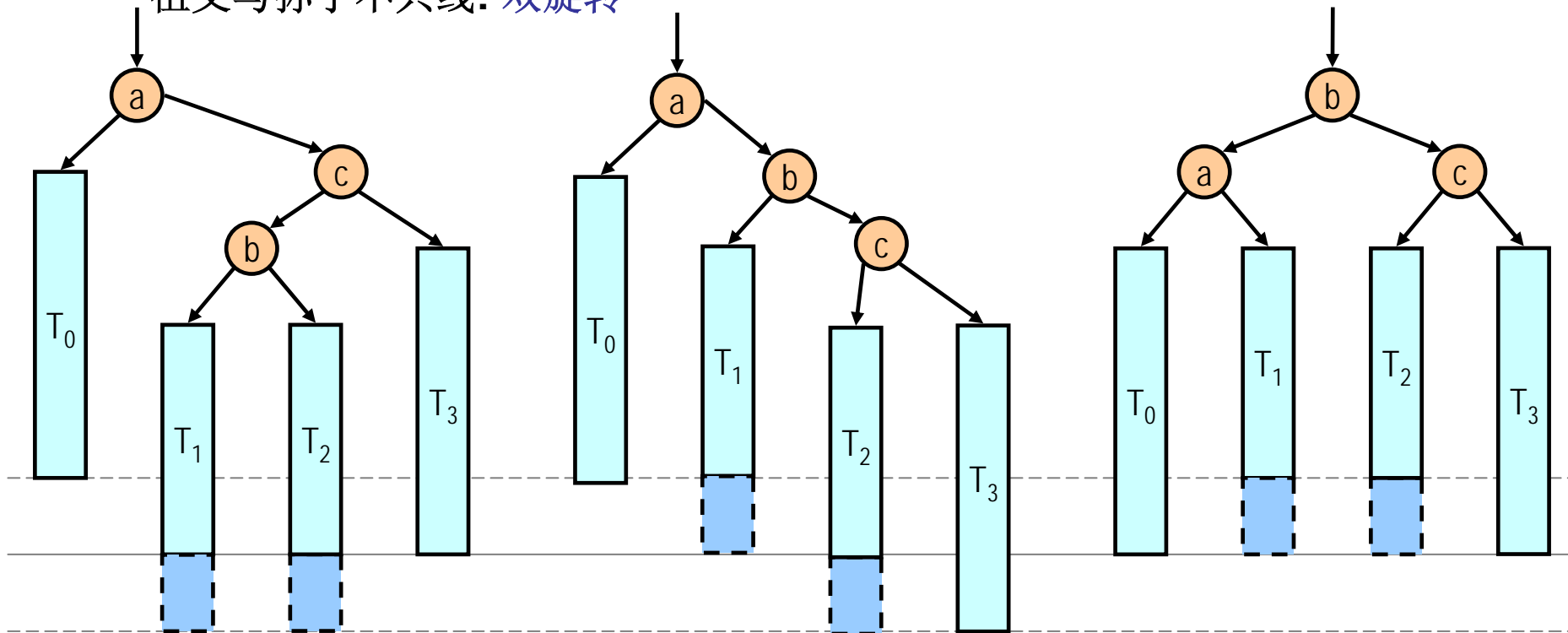
```
void rotatewithRightChild (int& p){  
    int q = right[p]; right[p] = left[q]; left[q] = p;  
    height[p] = max(height[left[p]], height[right[p]]) + 1;  
    height[q] = max(height[right[q]], height[p]) + 1;  
}
```

不平衡的点一定有孙子

祖父与孙子共线: 单旋转

祖父与孙子不共线: 双旋转

AVL的双旋转



```
rotatewithLeftChild(right[p]);  
rotatewithRightChild(p);
```

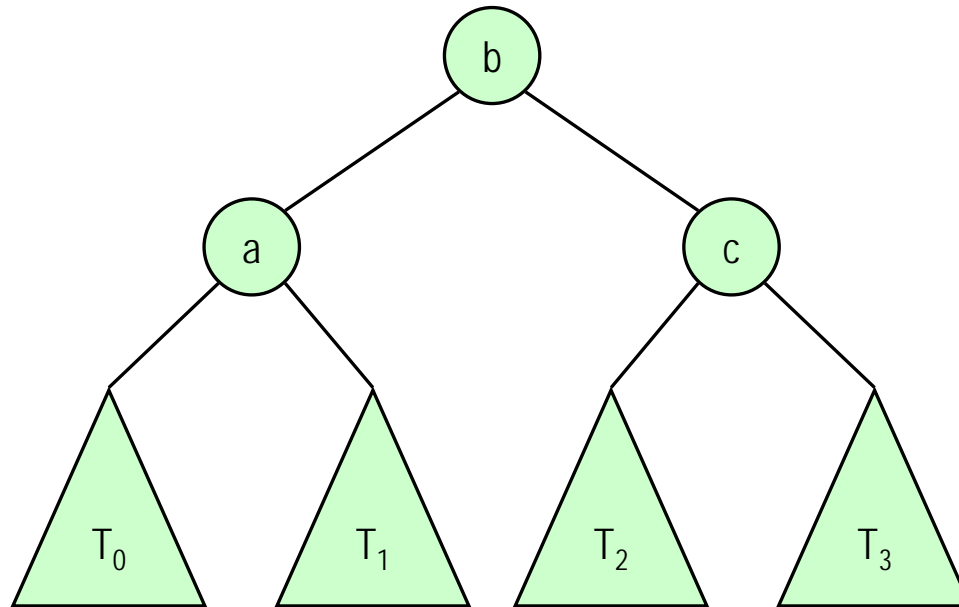
插入算法: 从插入元素开始
向上找第一个祖父不平衡
的点 x , 设父亲为 $p(x)$, 祖父
为 $g(x)$...

AVL树的插入

- 插入元素后，所有不平衡结点只可能在该元素到根的路径上
- 从插入元素开始向上找第一个祖父不平衡的点 x ，设父亲为 $p(x)$ ，祖父为 $g(x)$ ，则
 - $x, p(x), g(x)$ “共线”：单旋转
 - $x, p(x), g(x)$ 不“共线”：双旋转
- 统一算法：不考虑旋转的种类，直接处理
- 重要结论：旋转后 $g(x)$ 的高度恢复到插入以前，因此 $g(x)$ 的不平衡祖先（如果有）也一起恢复平衡

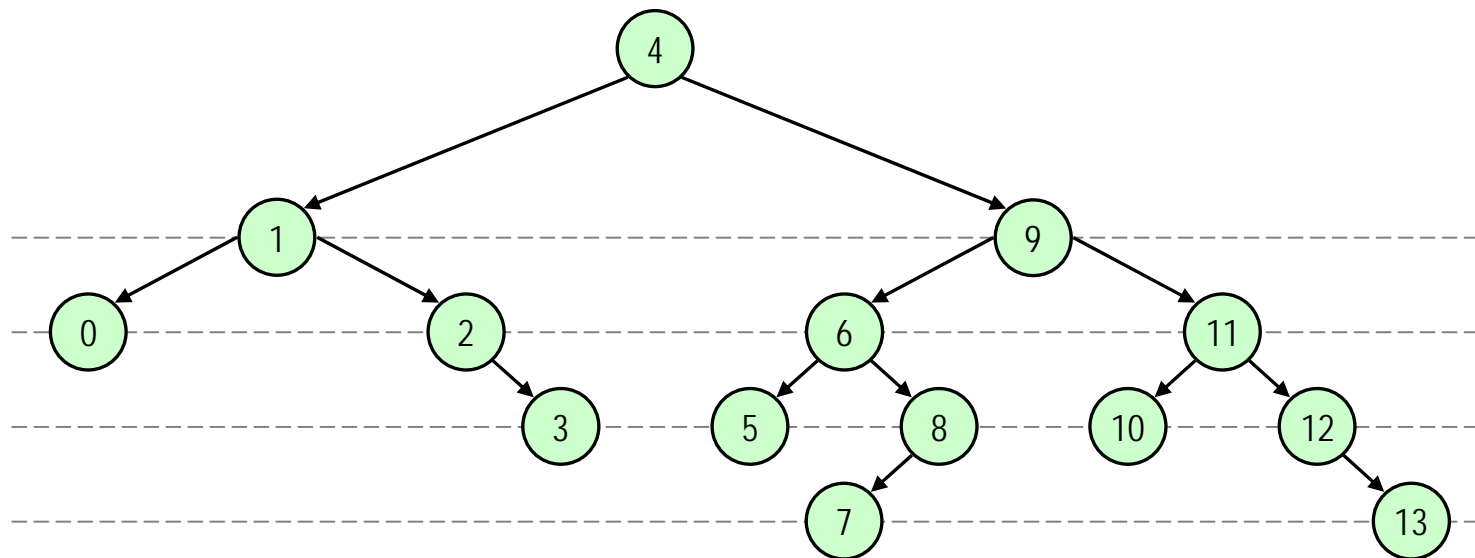
插入的统一算法

- 设 x , $p(x)$, $g(x)$ 在排序为 $a < b < c$; 它们的四棵不相交子树排序为 $T_0 < T_1 < T_2 < T_3$ (四棵子树中可能有空树), 则将以 $g(x)$ 为根的子树替换为如下子树后, **$g(x)$ 的高度恢复到插入前**



AVL树的删除

- 删除叶子后，它的父亲可能不平衡，在处理后可能祖父仍不平衡...
 - 从被删除叶子的父亲开始依次考虑各个祖先，只要不平衡就要旋转
 - 最坏情况： $O(\log n)$ 次旋转
- 删除中间结点时，先套用一般**BST**删除
 - 单儿子结点 u ：用惟一的儿子取代（注意：该儿子一定是叶子，否则 u 不平衡）
 - 双儿子结点 u ：用 u 的前驱取代后删除前驱



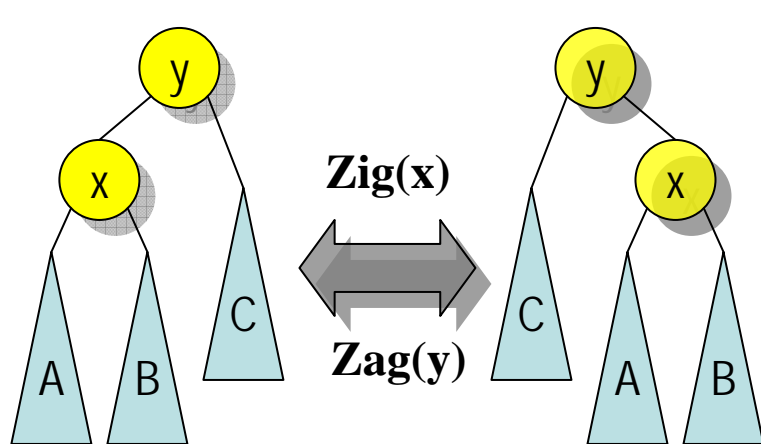
AVL = insert {4 1 9 0 2 6 11 3 5 8 10 12 7 13}

key	节点类型		删除方法	需重新平衡?
7、13	叶子		直接摘除	不需要
0、3、5、10				需要
12	内部节点	无左孩子 (右子树至多包含单个节点)	代之以右孩子	不需要
2				需要
8		无右孩子 (左子树至多包含单个节点)	代之以左孩子	不需要
				需要
9		左、右孩子都有	代之以直接前驱	不需要
1、6、11、4				需要

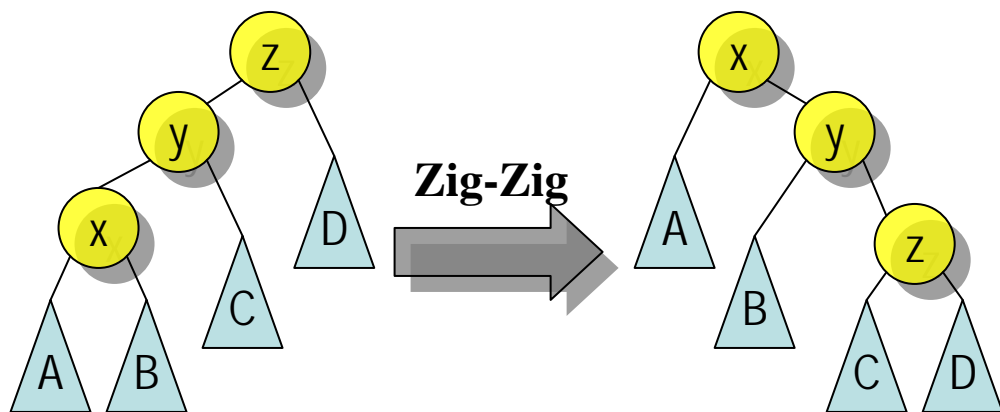
二、伸展树

- **基本思想**: 不严格限制树的形状, 而是假设访问有局部性, 让数据在访问后不久**再次访问**时变快
- **伸展操作****Splay(x,S)**: 把x旋转到树根, 同时保持树是一棵合法的BST
- 设 $y = \text{father}(x)$, $z = \text{father}(y)$
 - **情况一**: y是根结点: zig或zag
 - **情况二**: y不是根结点, 且x与y同是(自己父亲的)左孩子或同是右孩子: zig-zig或zag-zag
 - **情况三**: y不是根结点, 且x与y中一个是左孩子一个是右孩子: zig-zig或zag-zig

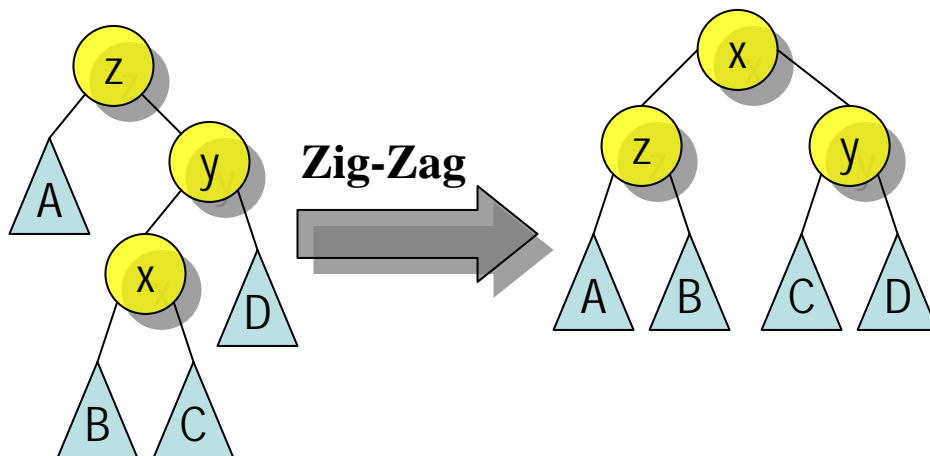
三种情况的处理



情况1. y 是根



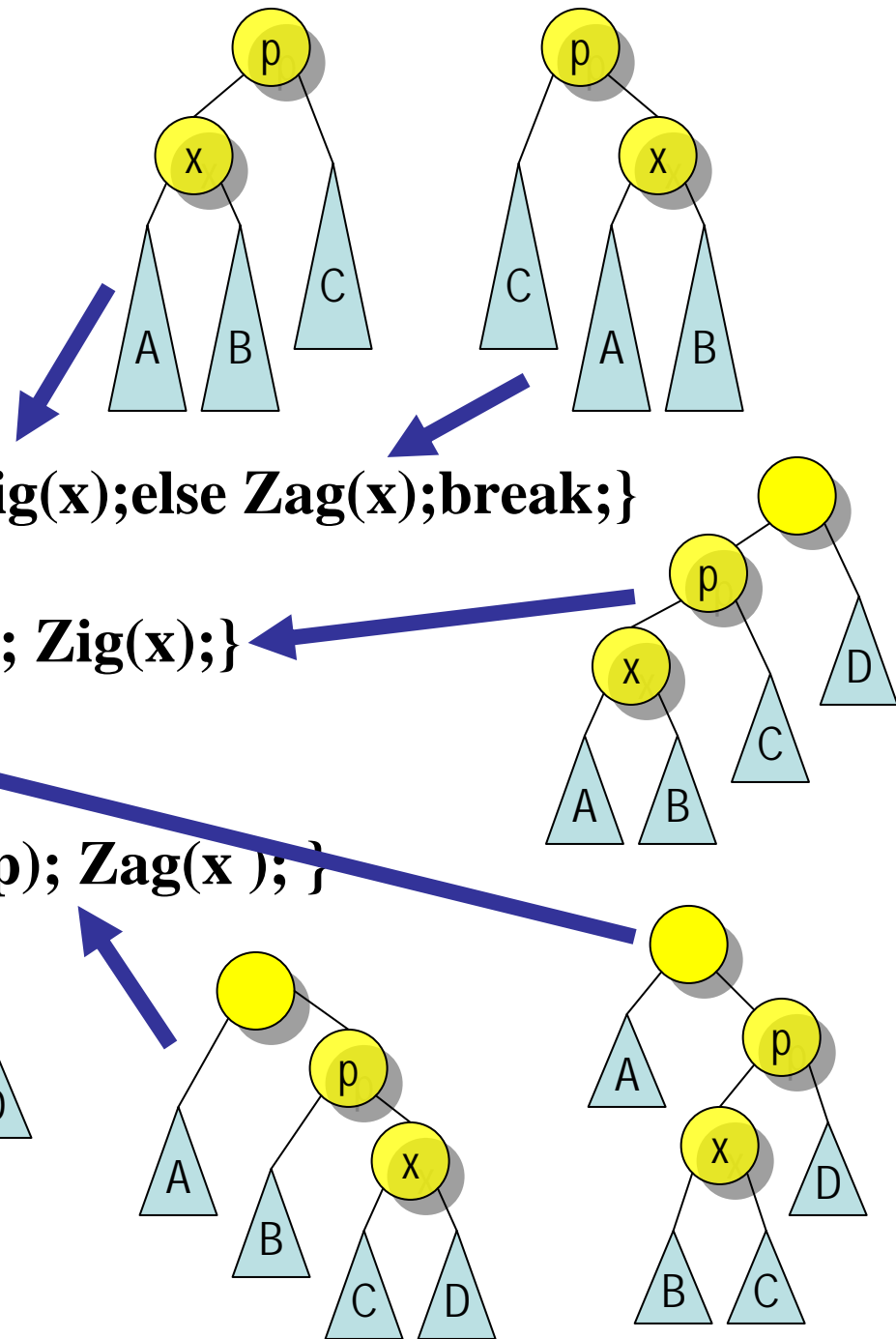
情况2. y 不是根, x, y, z 共线



情况3. y 不是根, x, y, z 不共线

伸展操作的实现

```
void splay (int& x , int& s){  
    int p;  
    while (father[x]){  
        p = father[x];  
        if (!father[p]){ if(x == left[p]) Zig(x);else Zag(x);break;}  
        if(x == left[p]){  
            if(p == left[father[p]]){ Zig (p); Zig(x);}  
            else{ Zig(x); Zag(x );}  
        }else{  
            if(p == right[father[p]]){ Zag(p); Zag(x ); }  
            else{ Zag(x); Zig(x); }  
        }  
    }  
    s = x;  
}
```

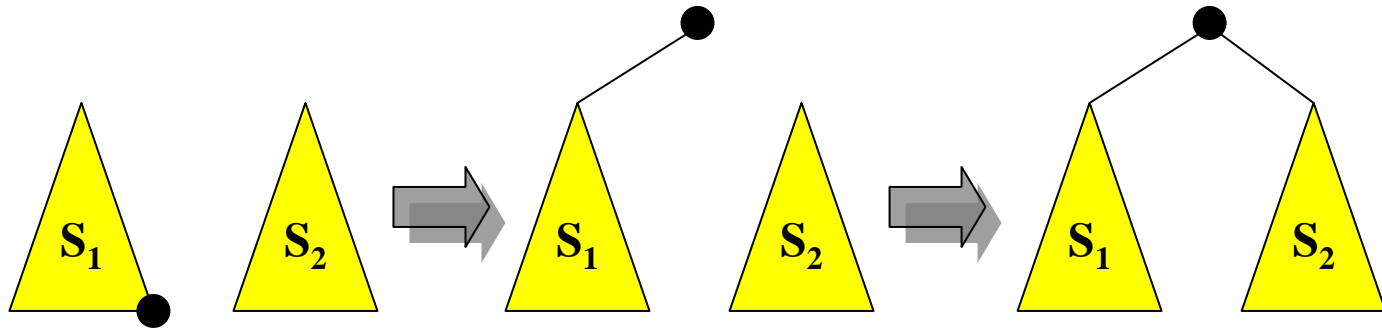


基本操作

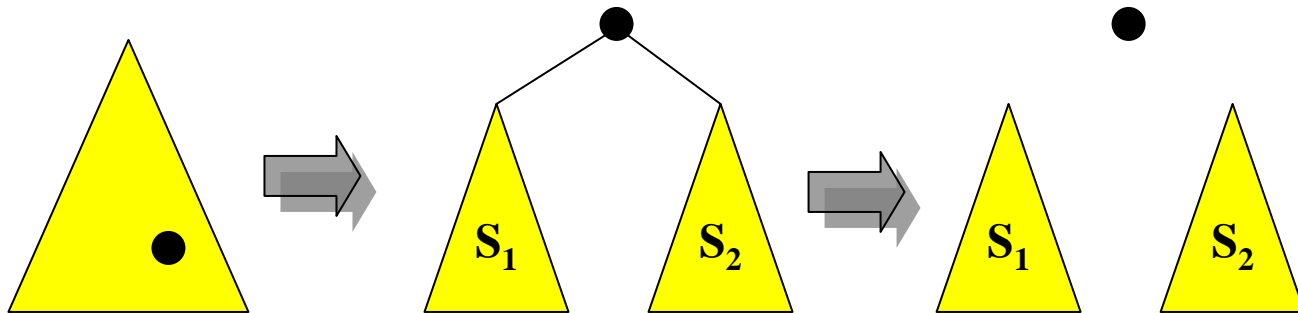
- Find(x , S): BST查找, 然后Splay
- Insert(x , S): BST插入, 然后Splay
 - 方法二: 查找 x 后Join (留作练习)
- Delete(x , S)
 - Find(x , S), 合并 x 的左右儿子
- Join(S_1 , S_2): 见后
- Split(x , S): 见后
- 伸展操作是基础!

合并与分离

- $\text{Join}(S_1, S_2)$: 伸展 S_1 中的最大元素



- $\text{Split}(x, S)$: 伸展 x , 断开儿子



```

int find( int x , int s)
{ int p = BST_Search (x, s); splay (p , s); return p; }
void insert(int x , int & s)
{ int p = BST_Insert (x, s); splay (p , s); return p; }
void remove(int x , int & s)
{ int p = find(x,s); join (left[p], right[p]); }
int maximum(int s)
{ int p = s; while(right[p]) p = right[p]; splay(p,s); return p; }
int minimum(int s)
{ int p = s; while (left[p]) p = left[p]; splay(p,s); return p; }
int prev( int x , int& s)
{ int p = find(x , s); p = left[p]; return maximum(p); }
int next( int x , int& s)
{ int p = find(x , s); p = right[p]; return minimum (p); }
int join ( int& s1 , int& s2){
    if (!s1) return s2 ; if (!s2) return s1;
    int p = maximum(s1); right[p ] = s2;
    return p;
}
void split (int x , int&s , int& s1 , int& s2){
    int p = find (x, s); s1 = left[p]; s2 = right[p];
}

```

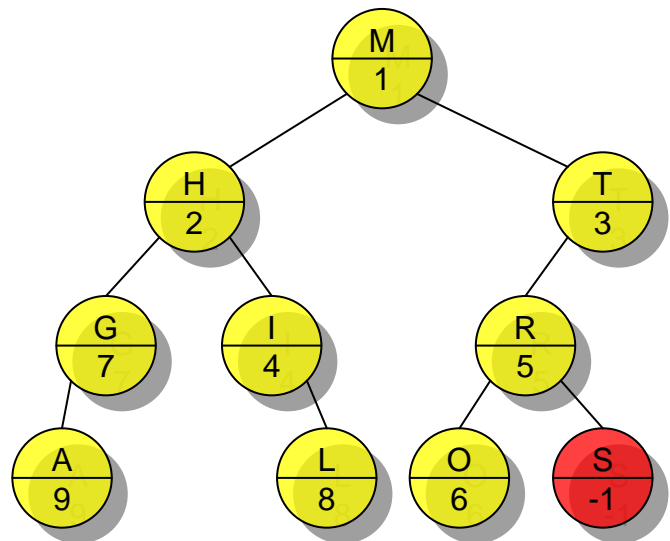
小结

- 伸展: 三种情况, 两种基本旋转(zig, zag)
- 五种基本操作: 以伸展为基础
- 时间复杂度: n 个结点的伸展树, 每个操作的平摊时间复杂度为 $O(\log n)$ (不证)
- 下面的讨论均假定所有key均不相同. 如果有相同的key呢?
 - 方法一: 加计数器
 - 方法二: 允许多个相同结点, 注意各种操作

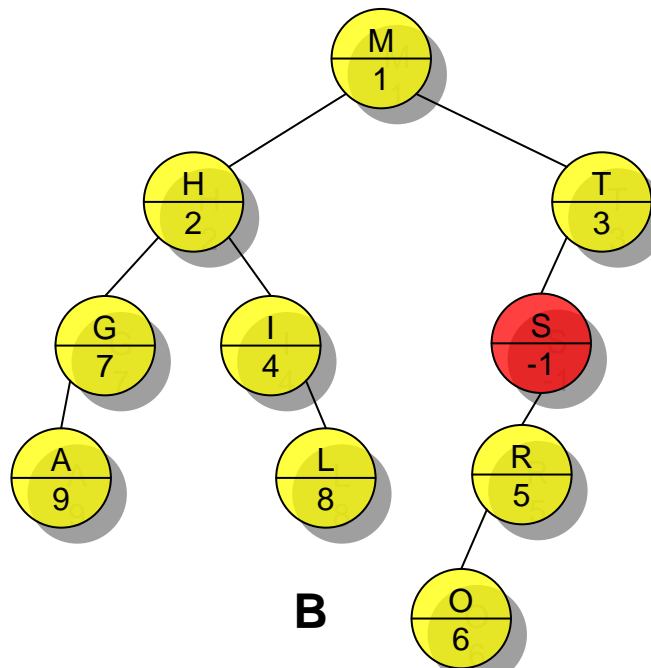
三、Treap

- 有一种实现相对容易的平衡二叉树称为
Treap = Tree + Heap
 - 每个结点有两个键值
 - treap关于key是排序二叉树
 - treap关于priority是堆（除了形态不一定是完全二叉树外）
- 把Treap作为平衡二叉树的方法是：插入时
随机取优先级，则期望树高为 $O(\log n)$
- 重要结论：key和priority确定时，treap惟一

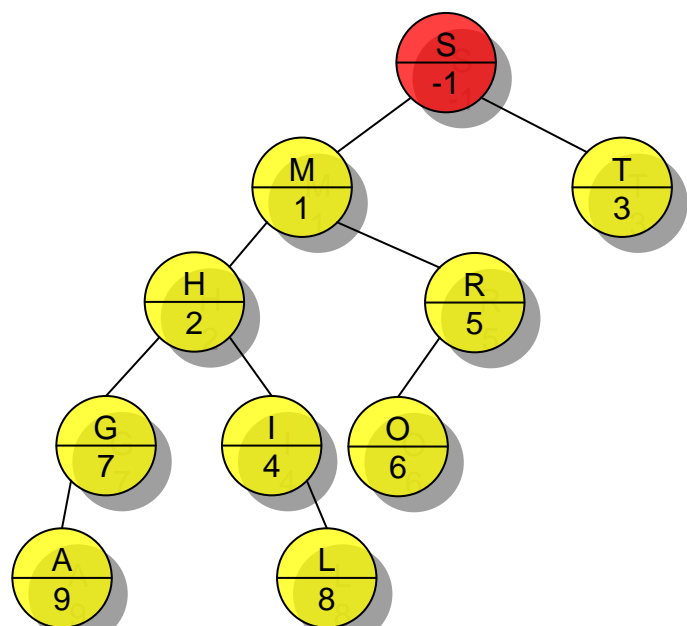
插入: **ABCD**
删除: **DCBA**



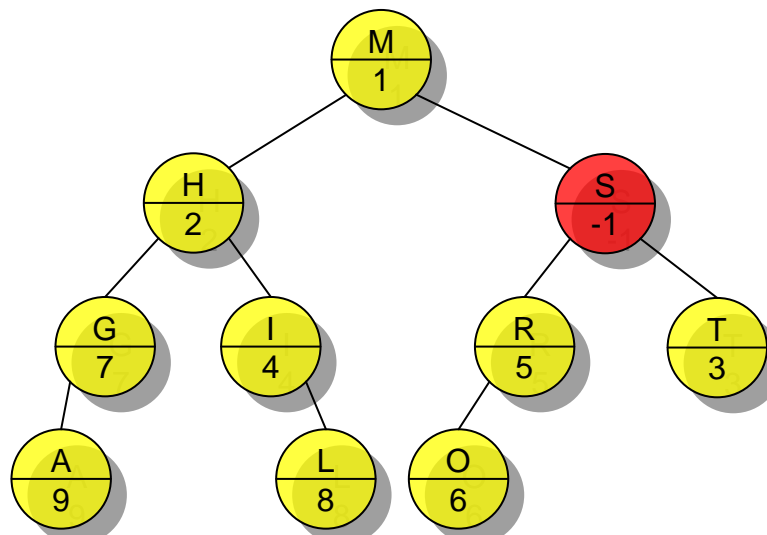
A



B



D



C

```

void insert (int x , int& p){
    if (!p)
        p = newnode(x, myrand());
    else if(x < key[p])
        { insert(x, left[p]); if(pr[left[p]] < pr[p]) rotateWithLeftChild(p); }
    else if(x > key[p])
        { insert(x, right[p]); if(pr[right[p]] < pr[p]) rotateWithRightChild(p); }
}

```

```

void remove (int x , int& p){
    if(p){
        if(x < key[p]) remove(x, left[p]);
        else if(x > key[p]) remove(x, right[p]);
        else {
            if (!left[p] && !right[p]) deletenode(p);
            if(pr[left[p]] < pr[right[p]]) { rotateWithLeftChild(p); remove(right[p]); }
            else { rotateWithRightChild(p); remove(left[p]); }
        }
    }
}

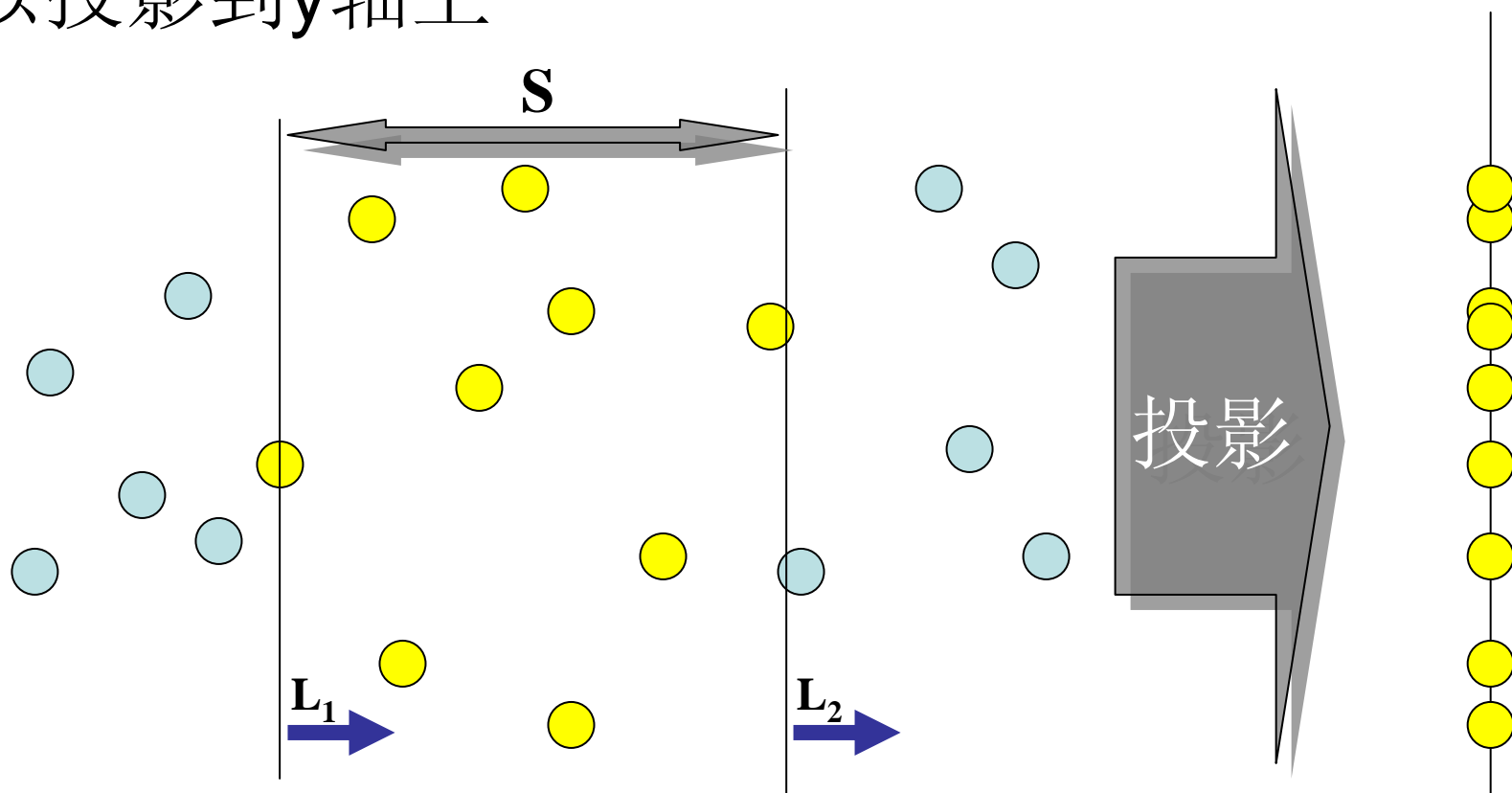
```

例1. 采矿

- 金矿的老师傅年底要退休了。经理为了奖赏他的尽职尽责的工作，决定在一块包含 n ($n \leq 15\ 000$) 个采金点的长方形土地中划出一块长度为 S ，宽度为 W 的区域奖励给他 ($1 \leq S, W \leq 10000$)
- 老师傅可以自己选择这块地的位置，显然其中包含的采金点越多越好。你的任务就是计算最多能得到多少个采金点。如果一个采金点的位置在长方形的边上，它也应该被计算在内。

平面扫描

- 用间隔固定的两条线 L_1 和 L_2 从左到右扫描，则两线之间的所有点可以忽略 x 坐标，即可以投影到 y 轴上



投影后的处理

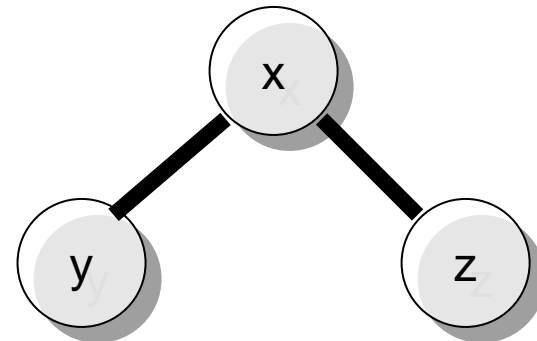
- 投影之后的问题变为了：在数轴上找一个长度为 W 的区间，包含尽量多的点
- 每个点 y 拆成两个事件 $(y, +1)$, $(y+w-1, -1)$ ，表示区间右端点在 y 和 $y+w-1$ 时该点进入/退出扫描区间，则扫描过程变成了累加过程，点数最多的位置对应了最大前缀和
- 主算法：从左到右处理各个点进入/退出 L_1 - L_2 间区域的事件，维护最大前缀和

最大前缀和

- 要求: 支持以下操作
 - INSERT(key)
 - DELETE(key)
 - FIND(key)
 - **MAXSUM**: 求最大的 k 前缀和
- 用BST或线段树均可, 结点附加信息: $m(p)$ 表示以结点 p 为根的子树中的最大前缀和
 - 如何用 $m(p)$ 计算MAXSUM? 答: 就是 $m(\text{root})$
 - 如何维护 $m(p)$? 答: 利用所有结点和 $s(p)$

$m(p)$ 的计算

- 不管是**BST**插入还是伸展过程中的维护，都需要利用用儿子计算父亲的递推式
- 最大前缀和的终点 u 有三种情况
 - u 在以 y 为根的子树中，则 $m(x)=m(y)$
 - $u=x$ ，则 $m(x)=s(y)+x$
 - u 在以 z 为根的子树中，则 $m(x)=s(y)+x+m(z)$



二、可并优先队列

可并优先队列

- 二叉堆很好的实现了优先队列的各种操作，但是却很难将两个堆合并起来（只能将其中一个堆的元素一个一个取出来插入另一个堆）。如果两个堆的元素都有 n 个，需要花 $n\log n$ 的时间
- 常用的可并优先队列有四种，其中左偏树和斜堆实现简单，实用性高。二项堆是一种非常巧妙的数据结构，实现难度适中，也是Fibonacci堆的基础。Fibonacci堆的理论时间复杂度非常优秀，特别是基于层叠提升法的decreaseKey操作，不仅思想巧妙，而且平摊时间复杂度仅为 $O(1)$ 。

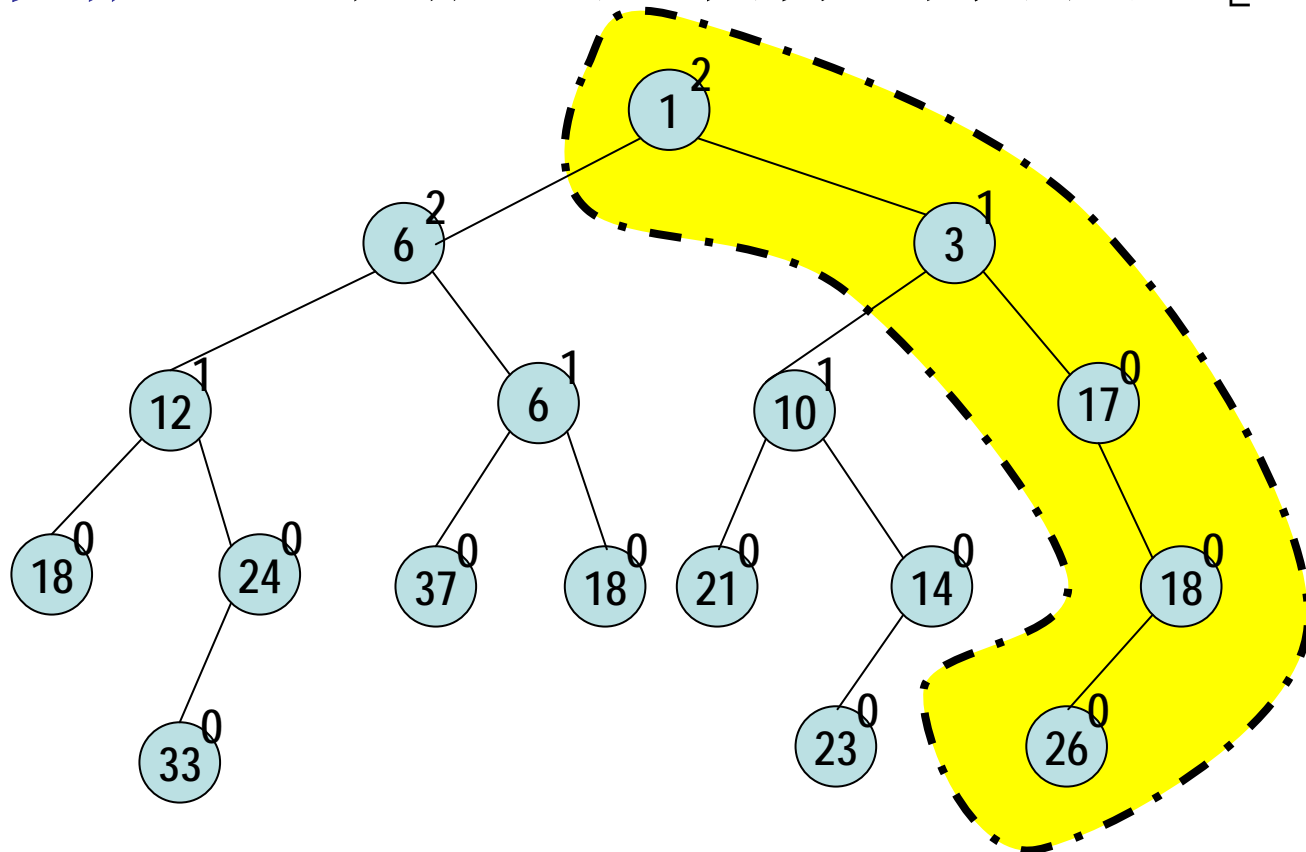
左偏树

外结点

- 左偏树是一棵二叉树，每个结点定义了距离 (dist)，即它到后代中**最多一个儿子**的结点的最小距离。叶子的距离为0，而空结点的距离为-1
- 左偏树满足两个性质：
 - **堆性质**：根的键值小于儿子键值
 - **左偏性质**：根的左儿子的距离大于或等于右儿子的距离。这个性质是递归的，即左偏树根的左右子树分别是左偏树
- 左偏树的距离定义为根的距离，根据左偏性质，它是树中**最右路径**的长度。左偏树并不是高度越小越好，而是越**左偏**越好，这样距离才会小

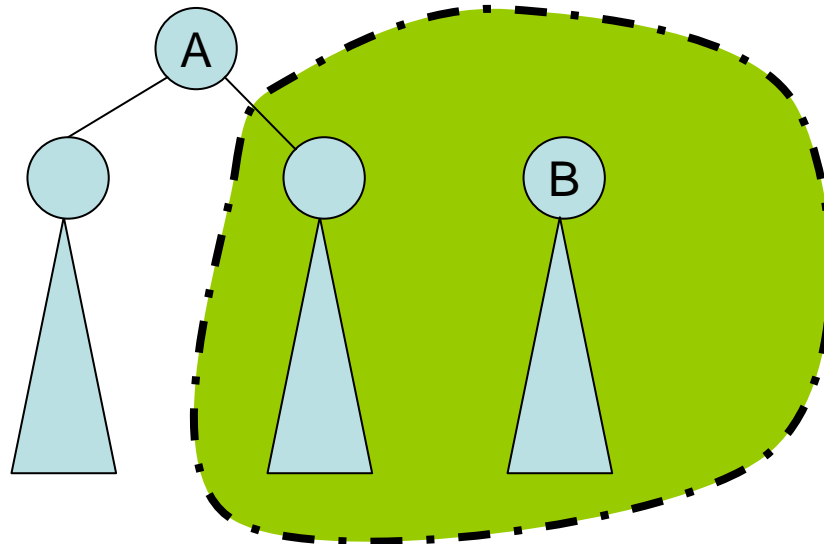
左偏树的高度

- 由左偏性质，**结点距离等于右儿子距离加1**。由于左儿子的距离至少和右儿子一样大，因此：
- **基本结论：** n 个结点的左偏树距离不超过 $\lfloor \log_2(n+1) \rfloor - 1$



基本操作：合并

- 若**A**或**B**为空，要返回另外一棵树，否则
 - 第一步：假设**A**的根 \leq **B**的根（否则交换**A**和**B**），把**A**的根作为新树的根，合并 $\text{right}(A)$ 和**B**
 - 第二步：如果合并后 $\text{right}(A) > \text{left}(A)$ ，交换
 - 第三步：更新 $\text{dist}(A) = \text{dist}[\text{right}[A]] + 1$



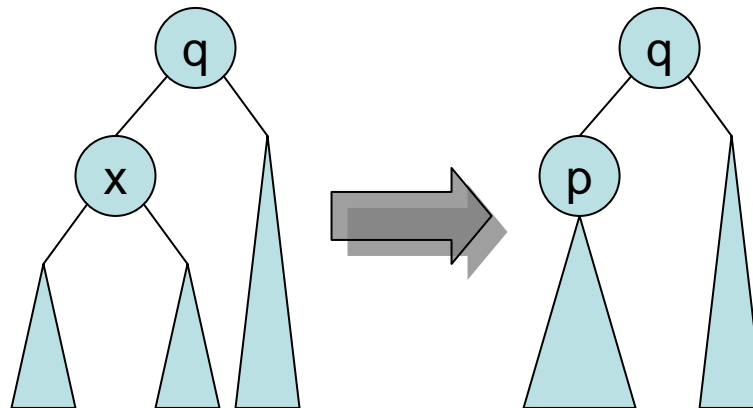
合并操作的分析

- 每次递归合并时分解右子树，它的距离至少减少1，时间复杂度为 $O(\log N_1 + \log N_2)$

```
int merge ( int a , int b){  
    if (!a) return b; else if (!b) return a;  
    if(key[b]<key[a]) swap(a, b);  
    right[a] = merge (right[a], b);  
    if(dist[right[a]] > dist[left[a]]) swap(left[a], right[a]);  
    if (!right[a]) dist[a] = 0;  
    else dist[a] = dist[right[a]] + 1;  
    return a;  
}
```

其他操作

- 插入：与单结点堆合并
- 删除最小值：合并根的左右子树
- 建立：将 n 个结点放入**FIFO**队列，每次取出队首两个堆进行合并，放入队尾，时间 $O(n)$
- 删除已知结点 x ：首先合并 x 的左右子树，得到子树 p ，然后进行讨论

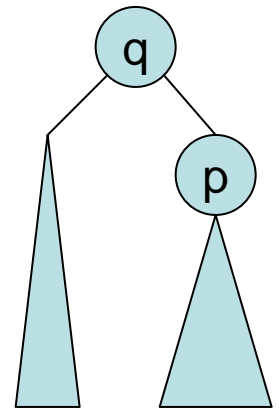
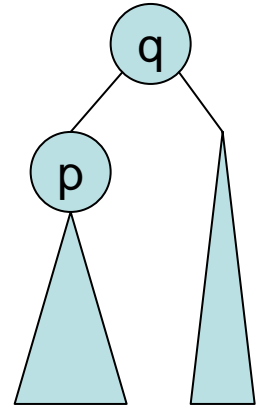


删除已知结点

- 设合并后 p 的距离为 $\text{dist}(p)$, 则
 - $\text{dist}(p)+1=\text{dist}(q)$, 删除结束
 - $\text{dist}(p)+1<\text{dist}(q)$, 更新 $\text{dist}(q)=\text{dist}(p)+1$. 若 p 是 q 的左子树, 交换子树后**处理 q 的父亲**
 - $\text{dist}(p)+1>\text{dist}(q)$, p 是左子树则不更新, 否则
 - p 的距离仍不超过 q 的左子树距离, 只更新 q 的距离
 - p 的距离大于 q 的左子树距离, 交换 q 的子树并调整 q 的距离。如果交换后 q 的距离增大, **处理 q 的父亲**

q距离变小

q距离变大



始终从右子树升上来; 变大变小不会交替出现; 时间复杂度 $O(\log n)$

```
void remove (int x)  
{  
    q = father[x];  
    p = merge(left[x], right[x]);  
    father[p] = q;  
    if(q && left[q] == x) left[q] = p;  
    if(q && right[q] == x) right[q] = p;  
    while(q)  
    {  
        if(dist[left[q]] < dist[right[q]]) swap(left[q], right[q]);  
        if(dist[right[q]]+1 == dist[q]) break ;  
        dist[q] = dist[right[q]] + 1;  
        p = q;  
        q = father[q];  
    }  
}
```

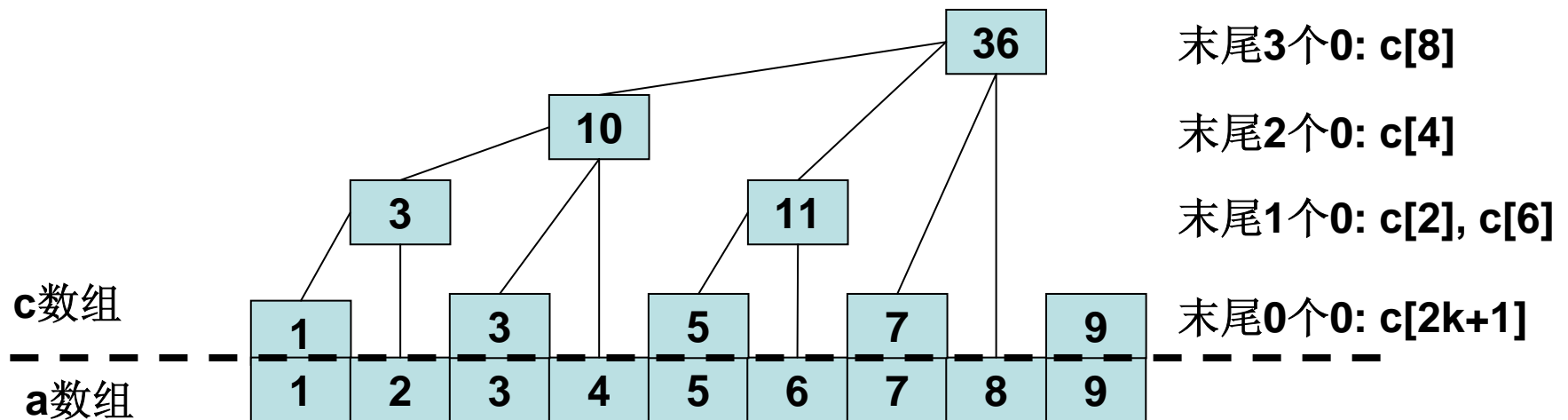
三、线段树和树状数组基础

动态统计问题I

- 有一个包含 n 个元素的整数数组 A ，每次可以修改一个元素，也可以询问一个区间 $[l, r]$ 内所有元素之和
- 如何设计算法，使得修改和询问操作的时间复杂度尽量低？

树状数组

- 设数组 $c[i] = a[i-2^k+1] + a[i-2^k+2] + \dots + a[i]$
 - k 为 i 在二进制形式下末尾 0 的个数
 - 起点是把 i 的最后一个 1 变为 0 再加 1
- c 数组的分层表示和递推关系如下图

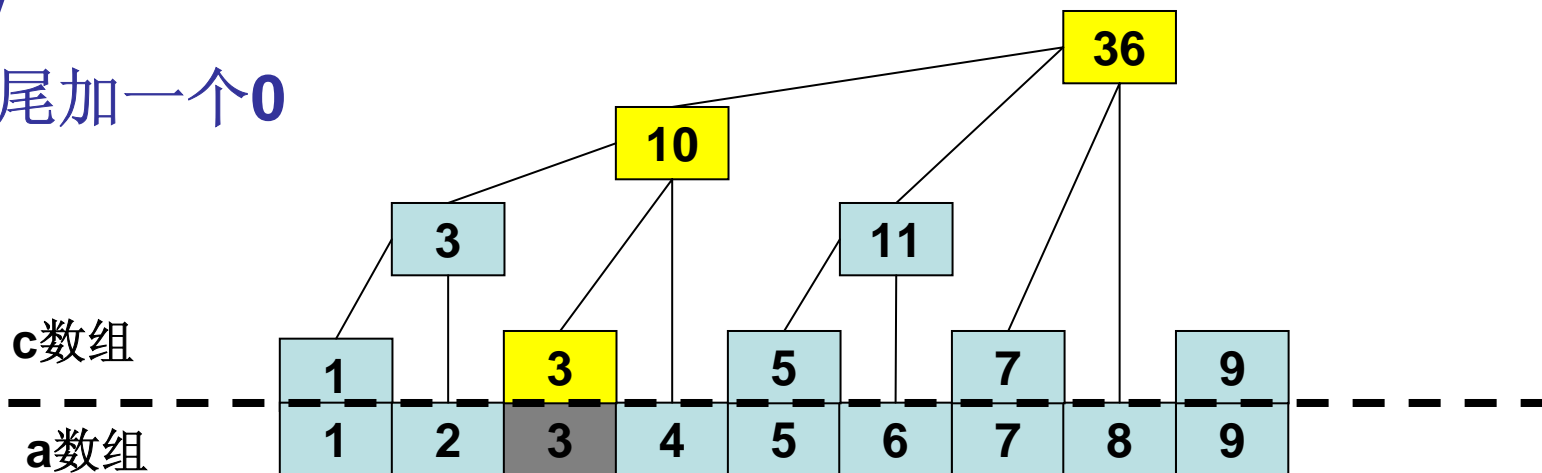


第 i 层末尾有 i 个零，度数为 $i+1$ ，定义式 2^i 项

修改操作

- 修改 $a[k]$ 后， c 数组的哪些元素受到影响？
 - $p_1=k$ 肯定受到影响，设 p_i 的父亲为 p_{i+1} ，则
 - $p_{i+1}=p_i+2^L$ ， L 为 p_i 二进制中末尾0的个数
 - 给 $a[3]$ 增加 x ，则 $p_1=3$ ， $p_2=3+2^0=4$ ， $p_3=4+2^2=8$ ， $p_4=8+2^3=16>9$ ，因此修改 $c[3], c[4], c[8]$

给末尾加一个0



求和操作

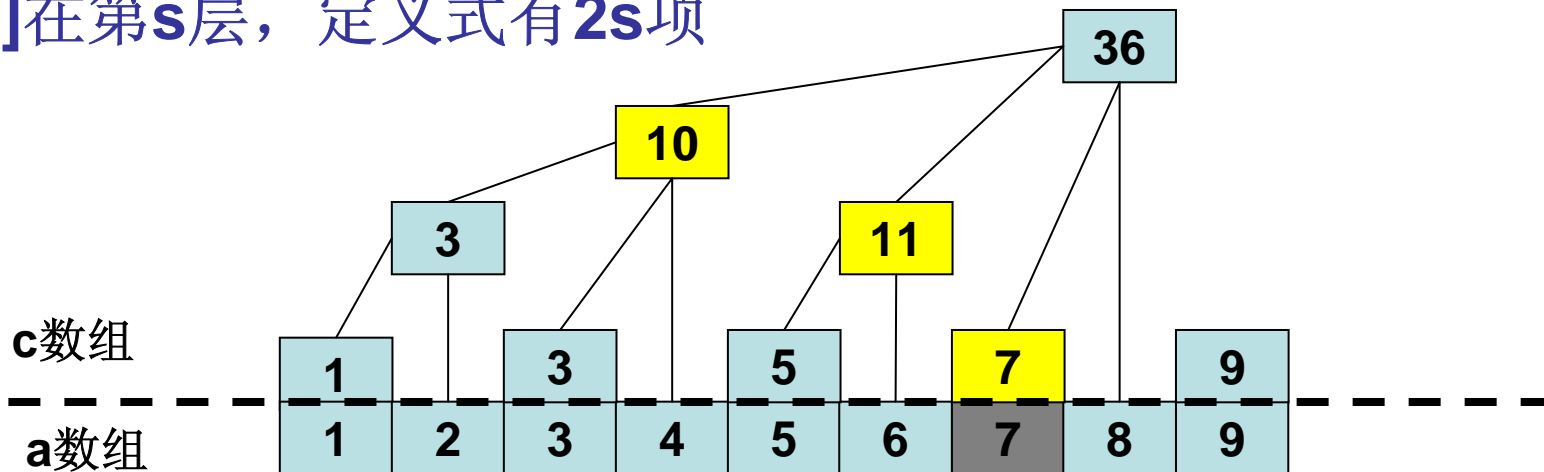
- 设要求前k项的和， 则从最后一项开始

$$-k_1 = k$$

– $k_{j+1} = k_j - 2^s$, s 为 k_j 二进制末尾 0 的个数

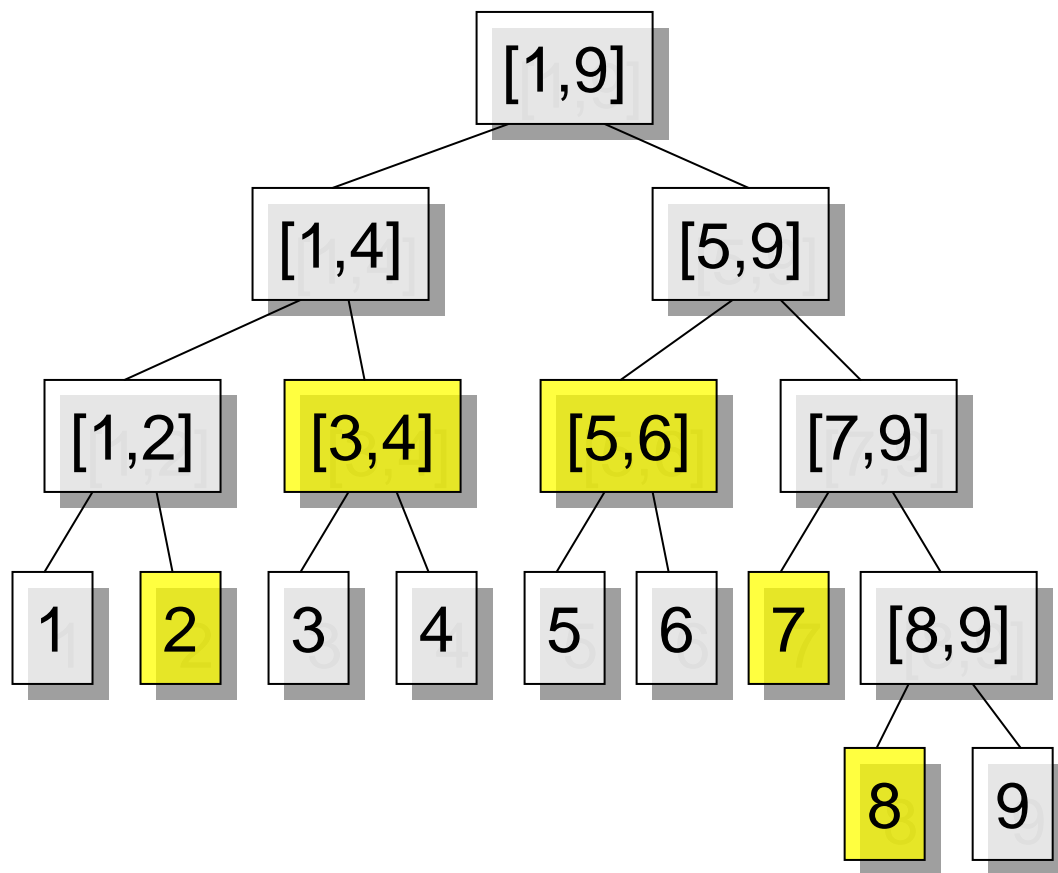
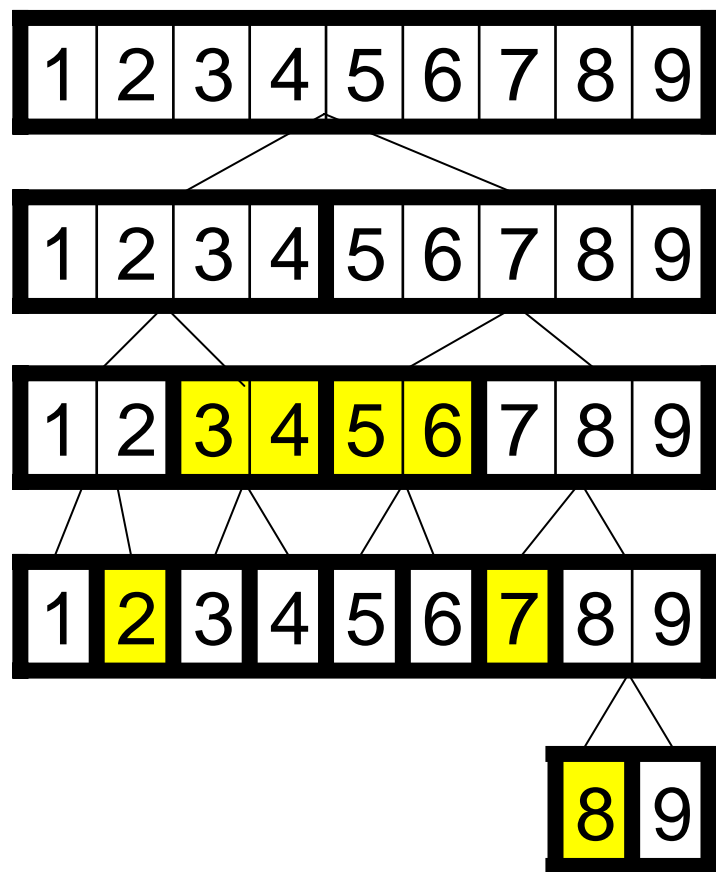
$k=7$, 则 $k_1=7$, $k_2=k_1-2^0=6$, $k_3=k_2-2^1=4$, $k_4=k_3-2^2=0$, 因此前7项和为 $c[7]+c[6]+c[4]$

$c[k_i]$ 在第s层，定义式有2s项



线段树

- 线段 $[1, 9]$ 的线段树和 $[2, 8]$ 的分解

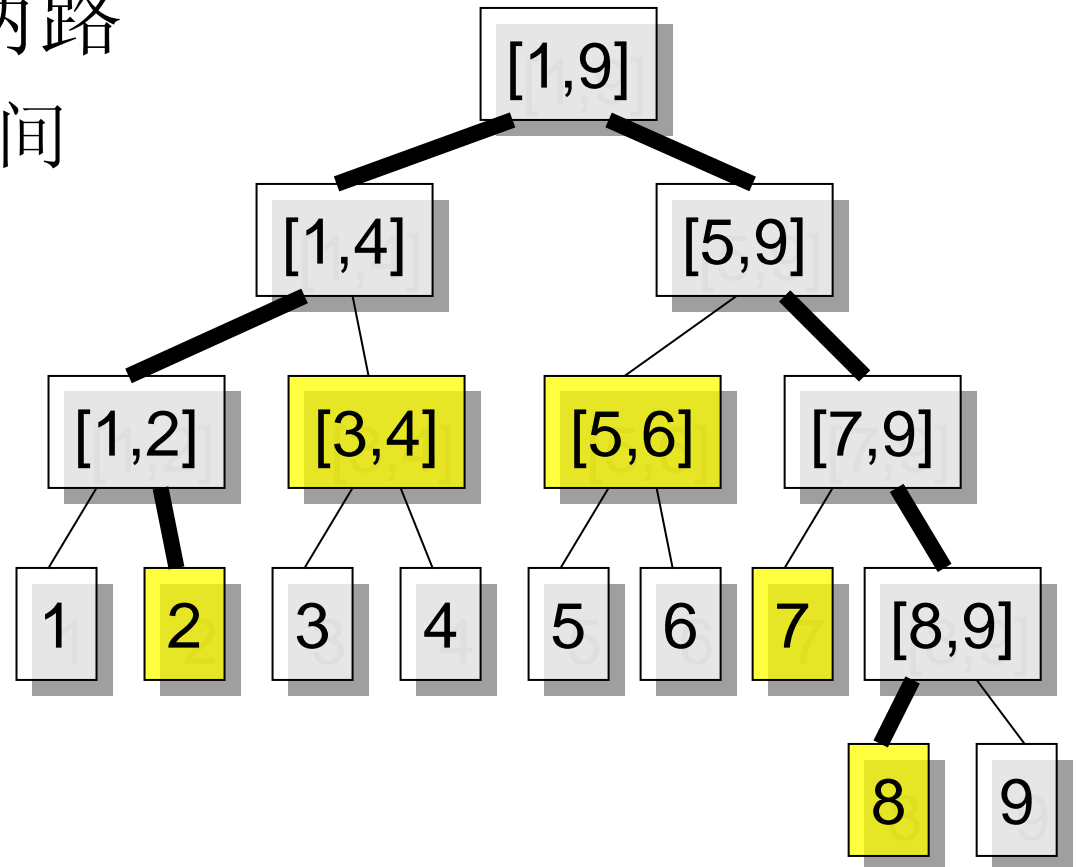


性质

- 每层都是 $[a, b]$ 的划分. 记 $L=b-a$, 则共 $\log_2 L$ 层
- 任两个结点要么是包含关系要么没有公共部分, 不可能部分重叠
- 给定一个叶子 p , 从根到 p 路径上所有结点 (即 p 的所有直系祖先)代表的区间都包含点 p , 且其他结点代表的区间都不包含点 p
- 给定一个区间 $[l, r]$, 可以把它分解为不超过 $2\log_2 L$ 条不相交线段的并

基本算法

- 找点: 根据定义, 从根一直走到叶子 $\log L$
- 区间分解: 兵分两路
 - 每层最多两个区间
 - 总时间 $4\log_2 L$



线段树的关键

- 用线段树解题的关键
 - 得到讨论区间(可能要先离散化)
 - 设计区间附加信息和维护/统计算法
- 线段树自身没有任何数据, 不像**BST**一样有一个序关系
- 警告: **想清楚**附加信息的**准确含义**, 不能有半点含糊!
- 建议: 先设计**便于解题**的附加信息, 如果难以维护就加以修改

再谈动态统计问题I——解法2

- 附加信息: $s(p)$ 表示结点 p 所代表区间内所有元素之和
- 维护算法
 - **ADD**: 给 i 对应结点的所有直系祖先 s 值增加 k
 - **SUM**: 做区间分解, 把对应结点的 s 值相加
- 时间复杂度和解法1一样, 但系数更大
- 那为什么还要用线段树? ? ?

动态统计问题II

- 包含 n 个元素的数组 A
 - $\text{ADD}(i, j, k)$: 给 $A[i], A[i+1], \dots A[j]$ 均增加 k
 - $\text{QUERY}(i)$: 求 $A[i]$
- 先看看是否可以沿用刚才的附加信息
 - $\text{QUERY}(i)$ 就是读取 i 对应的结点上的 s 值
 - ADD 呢? 极端情况下, 如果是修改整个区间, 则所有结点都需要修改!
- 需要新的附加信息

新的附加信息

- **Lazy**思想: 记录有哪些指令, 而不真正执行它们. 等到需要计算的时候再说
- 假设结点 p 对应的区间是 $[i, j]$, $a(p)$ 表示所有形如 $\text{ADD}(i, j, k)$ 的所有 k 之和
 - 如果 $[i, j]$ 不对应任何结点怎么办? 区间分解!
 - 这样的信息实质上是把所有 ADD 指令合并到了一起. 可以吗? 可以的, 因为 ADD 具有叠加性
- **QUERY**: 把所有直系祖先的 a 值相加, 就是 $A[i]$ 的增加量

继续讨论

- 附加信息 $a(p)$ 到底是什么？
 - 首先要在同一条指令中被增加
 - 但在同一条指令中被增加的结点却不能都被修改, 否则 $ADD(1, n)$ 仍然要修改所有结点
 - 正确的理解是: 先把指令 ADD 分解为不超过 $2\log_2 L$ 条指令, 每条指令的区间 $[i, j]$ 都在树中有 **单一的结点**与之对应, 然后每条原子 ADD 操作只修改该结点本身的计数器

动态统计问题III

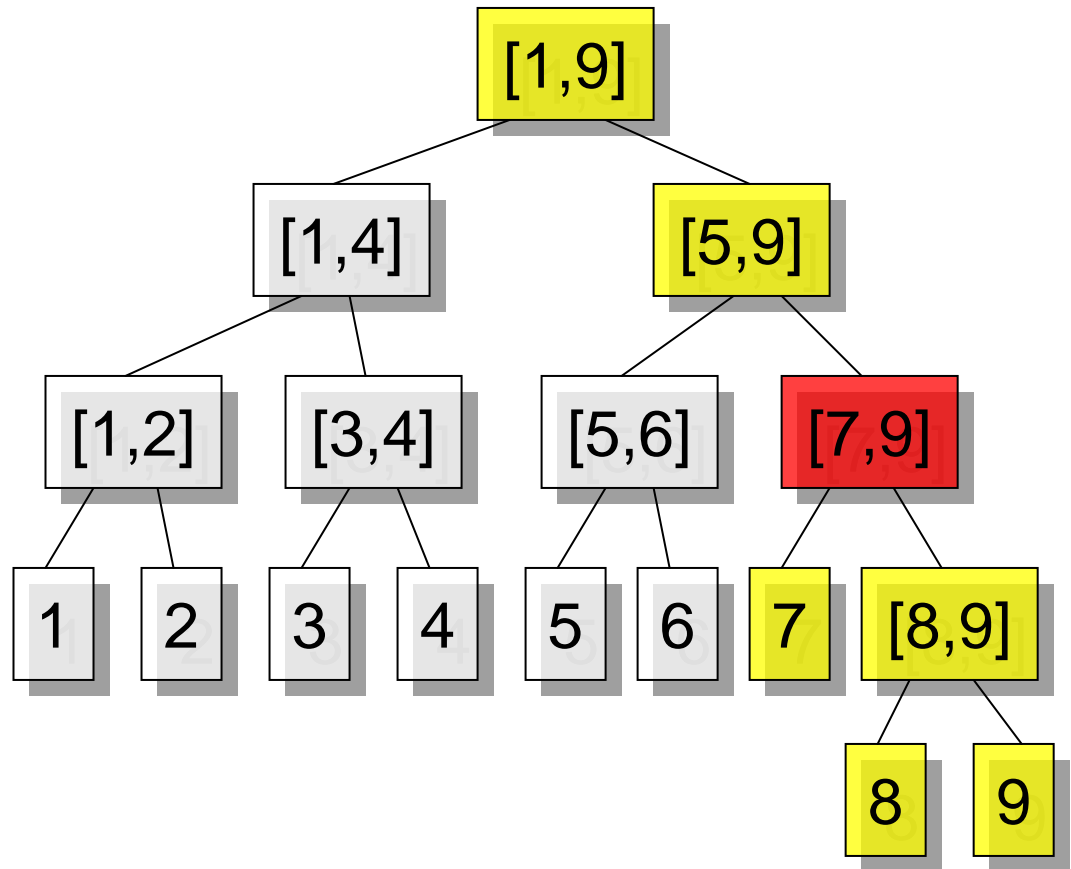
- 包含 n 个元素的数组 A
 - $\text{ADD}(i, j, k)$: 给 $A[i], A[i+1], \dots, A[j]$ 均增加 k
 - $\text{SUM}(p, q)$: 求 $A[p]+A[p+1]+\dots+A[q]$
- 显然动态统计问题I和II都是它的特殊情况
 - 问题I中, ADD 操作的 $i=j$
 - 问题II中, SUM 操作的 $p=q$
- 由于 ADD 操作和问题II一样, 这里沿用它的 ADD 实现, 那 SUM 怎么办?

SUM的实现

- 前面曾经提到, 区间统计的一般做法是把查询区间进行分解, 一一统计然后加起来
- 在本题中, 需要计算每个原子区间的数之和. 它们的和是多少呢? 这取决于有多少 **ADD** 操作影响到它
- 回忆: 任何两个树中区间要么相互包含要么没有公共部分. 因此影响一个原子区间的 **ADD** 操作都是它的直接祖先和后代

SUM的计算

- 右图表示影响
 $SUM(7, 9)$ 的所有区间
 - 影响全部: $[1,9]$,
 $[5,9]$, $[7,9]$
 - 影响部分: 7,
 $[8,9]$, 8, 9



完整的算法

- 至此, 算法轮廓已经出来
 - 再附加一个 $sa(p)$, 表示以 p 为根的子树所有结点的 a 值之和
 - **ADD**: 区间分解后除了修改各原子区间的 a 值外, 还要沿途修改 sa 值
 - **SUM**: 在区间分解的同时统计经过的 a 值, 然后把原子区间的 sa 值累加进来
- 两个操作均为 $O(\log n)$

例1. 动态区间最小值

- 包含 n 个元素的数组 A
 - $\text{MODIFY}(i, j)$: 设 $A[i] = j$
 - $\text{MIN}(p, q)$: 求 $\min\{A[p], A[p+1], \dots, A[q]\}$
- 和动态统计问题I很类似, 因此考虑设计附加信息: $m(p)$ 表示结点 p 所代表区间内所有元素的最小值, 那么 MIN 仍可以通过区间分解做. 但 MODIFY 呢?

递推法

- **MODIFY**操作仍然只需要修改从根到叶子的一条路径上所有 m 值, 但关键是: 如何修改?
- 回忆: 动态统计问题I中, 区间 $[l, j]$ 中任何一个元素增加了 k , 则区间综合增加 k . 但最小值呢? 只根据原来的 $m(p)$ 自身无法计算出新的 $m(p)$
- 方法: 递推. 设 p 的儿子为 l 和 r , 则
$$m(p) = \min\{m(l), m(r)\}$$
- 前提: 计算 $m(p)$ 时 $m(l)$ 和 $m(r)$ 已经算出.
- 保证: 自底向上递推

例2. 区间并的长度

- 实现一个区间集合
 - $\text{Add}(x, y)$: 增加区间 $[x, y]$ ($1 \leq x < y \leq n$)
 - $\text{Delete}(x, y)$: 删除区间 $[x, y]$, 它一定在集合中
 - **Total**: 区间并的长度(即被至少一个区间覆盖到的总长度)
 - 约定: 删除的区间 $[x, y]$ 一定是以前插入过
- 增加区间时进行分解, 设置计数器 $c(p)$, 表示分解后指令 $\text{Add}(x, y)$ 的条数

覆盖长度可以维护么？

- 考虑根结点. 如果 $c(\text{root}) > 0$, 则整个区间都被覆盖, 返回 L , 但如果 $c(\text{root}) = 0$ 呢? 需要根据左右儿子递推
- 是否可以定义 $l(p)$, 表示结点 p 对应的区间内被覆盖到的总长度呢? 不可以!
 - 初始为空时进行 $\text{Add}(1, n)$, 则树中所有结点对应的 $l(p)$ 都应被修改!
 - 怎么办? 修改 $l(p)$ 的定义

新的维护信息

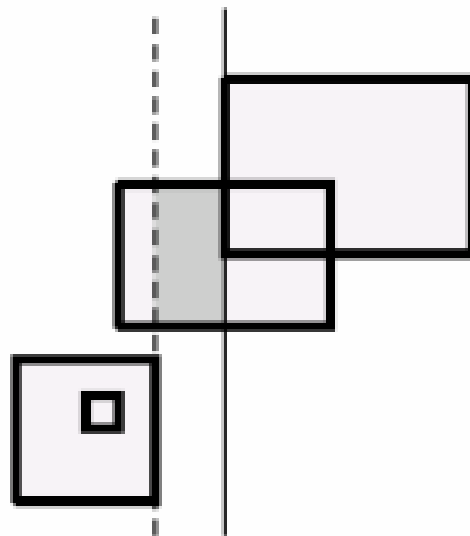
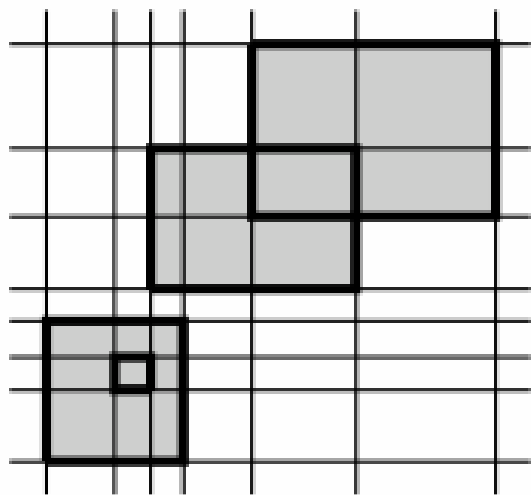
- 设 $l(p)$ 表示以 p 为根的子树中的所有Add操作在 p 对应的区间中覆盖了多大长度, 则 $Add(1, n)$ 只需要修改根
- 每个原子区间的插入、删除都只影响它的直接祖先, 插入/删除后自底向上用递推法维护即可

例3. 火星地图

- 2051年，科学家们探索出了火星上 N （ $N \leq 10\,000$ ）个不同的矩形（坐标为不超过109的正整数）区域并绘制了这些局部的地图，如图1-61所示。波罗的海太空研究所希望绘制出火星的完整地图。
- 科学家们首先需要知道这些矩形共占了多大的面积，你能帮助他们写一个程序计算出结果吗？

分析

- 离散化 + 水平扫描
- 维护**线段集合**，支持插入、删除、统计线段并的长度
- 线段树：每次操作 $O(\log n)$ ，共 $O(n \log n)$



例4. 01矩阵

- 给 $n \times n$ 的01矩阵, 支持
 - $C(x_0, y_0, x_1, y_1)$: 改变矩形 (每个元素取反)
 - $Q(x, y)$: 查询 (x, y) 的值

分析

- 构造辅助01矩阵 C' ，初始为0
- 矩形分解: $C(x_0, y_0, x_1, y_1)$ 等价于改变以下4点的值
 - $C'(x_0, y_0)$, $C'(x_0, y_1)$, $C'(x_1, y_0)$, $C'(x_1, y_1)$
- 元素 (x, y) 的最终值完全取决于在 C' 中 (x, y) 的右下方的元素和的奇偶性

例5. 动态区间k大数

- 维护一个数组 $A[1 \dots n]$
- 实现两个操作
 - $\text{Modify}(i, j)$, 设 $A[i] = j$
 - $\text{Query}(i, j, k)$, 返回 $A[i \dots j]$ 第 k 大元素

分析

- 首先考虑没有修改的情形
- 预处理：建立线段树，每个线段保存该区间内元素排序好的序列
- 查询 $\text{Query}(i,j,k)$
 - 把 $[i,j]$ 进行区间分解
 - 二分 W ，每次统计这些区间内一共有多少个数比 W 大，用 $\log W$ 次统计可求出第 k 大元素
- 如何统计原子区间内比 W 大的元素总个数？

分析

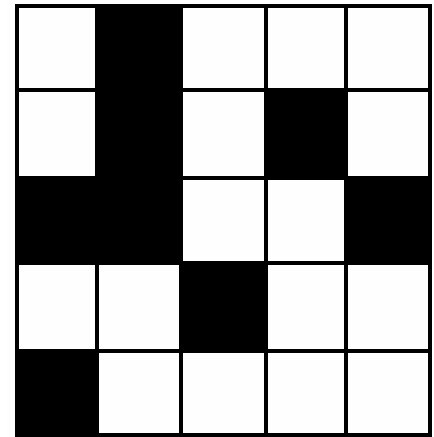
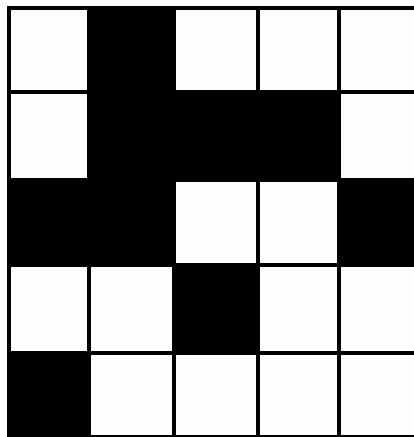
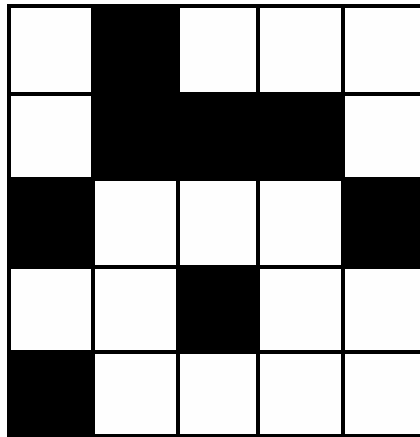
- 统计原子区间内一共有多少个数比 **W** 大
 - 区间内的数已排序，用二分每个区间求比 W 大的数 $\log n$
 - 累加所有 $2\log n$ 个区间比 W 大的数，共 $\log^2 n$
 - 总时间： $\log W * \log^2 n$
- 实现：一个归并排序可以同时构造线段树和每个节点内的排序数组. 空间： $O(n \log n)$

分析

- 有修改的情形: 每个结点不能用有序表了, 而需要是一棵平衡树
- 每次Modify需要修改 $O(\log n)$ 棵平衡树, 总时间为 $O(\log^2 n)$

例6. 动态连通块

- 给出 $n \times n$ 棋盘，有黑有白。每次改变其中一个格子颜色，输出黑白连通块的个数
- 左图，翻转(3,2)和(2,3)后分别得到中图和右图，应依次输出"4,3"、"5,2"



分析

- 对行集合建立线段树，区间 $[i,j]$ 保存内部的黑白连通块个数以及第 i 行和第 j 行每个格子所属于的连通块编号
- 由 $[i, \text{mid}]$ 和 $[\text{mid}+1, j]$ 可以合并成为 $[i, j]$ ，时间为 $O(n)$ （对交界线进行合并操作，修改内部连通块个数）
- 根据指令 (x,y) 所在行修改叶子区间，并往上递推。最多修改 $\log n$ 个区间，因此每次操作时间复杂度为 $O(n \log n)$

四、**RMQ**问题与**LCA**问题

RMQ问题

- RMQ (范围最小值查询)问题是一种动态查询问题，它不需要修改元素，但要及时回答出数组A在区间[l, r]中最小的元素值
- 对于RMQ，我们通常关心两方面的时间：预处理时间 $f(n)$ 和查询时间 $g(n)$ ，并用 $f(n)-g(n)$ 来描述算法的时间效率。
- RMQ可以用线段树来解决：每个树中区间记录该区间的最小元素，则每次询问只需要比较分解后的 $\log n$ 个树中区间。建树需要 $O(n)$ 的时间，询问需要 $O(\log n)$ 时间，换句话说，刚才基于线段树的算法是 $O(n)-O(\log n)$ 的

ST算法——预处理

- 基于线段树的算法虽然比暴力法好了很多，但对于RMQ这样的特殊问题并不算好
- ST(Sparse Table)算法是 $O(n \log n)$ - $O(1)$ 的，对于查询很多大的情况下比线段树好
- ST算法：用 $d[i, j]$ 表示从 i 开始的，长度为 j 的区间的RMQ，则有递推式

$$d[i, j] = \min\{d[i, j-1], d[i + 2^{j-1}, j-1]\}$$

- 因此，预处理时间复杂度为 $O(n \log n)$

ST算法——查询

- 取 $k = \lfloor \log_2(j-i+1) \rfloor$ ，那么令A为从i开始的长度为 2^k 的块，B为到j结束的块，那么A和B都是 $[i, j]$ 的子区间，但是A和B一起将覆盖整个 $[i, j]$ ，即：

$$RMQ(i, j) = \min\{d[i, k], d[j - 2^k + 1, k]\}$$

- 算法反映出了min和sum操作的本质不同：在分治时min可以分割成有重复的块，而sum不行

±1-RMQ问题

- 如果相邻两个元素 a_i 和 a_{i+1} 满足： $a_i - a_{i+1} = 1$ 或者 -1 （ a_i 和 a_{i+1} 不能相同），则这样的RMQ问题称为±1-RMQ问题
- **重要结论：**把 a 的所有元素同时减去一个相同的数，则修改后的数组对于询问将给出和原数组相同的最小值**位置**！
- 换句话说：在±1-RMQ问题中，长度为 n 的**本质不同**的数组只有 2^n 个！

±1-RMQ问题

- 把数组A划分成每部分长度为 $L=\log_2 n/2$ 的小块，则一共有 $2n/\log_2 n$ 个块
- 用 $O(n)$ 时间求出每个小块的最小值，令 $A'[i]$ 表示第 i 个小块的最小值，对 A' 做ST算法的预处理，时间为

$$O\left(\frac{2n}{\log n} \times \log\left(\frac{2n}{\log n}\right)\right) = O\left(\frac{2n}{\log n} \times (\log 2 + \log n - \log \log n)\right) = O(n)$$

- 对于一般的询问 $\text{RMQ}(i,j)$ ，可以分成三部分
 - 若干完整小块的RMQ: $\text{RMQ}(A', i', j')$
 - 小块内部的RMQ: $\text{In-RMQ}(x, y)$
- 关键：求出 $\text{in-RMQ}(x, y)$



O(1)

O(??)

In-RMQ

- 由于所有小块长度均为 $L = \log_2 n / 2$ ，所有它们最多只有 $2^L = n^{1/2}$ 种本质不同的数组，每个数组有不超过 $L^2 = O(\log^2 n)$ 种询问
- 用完全递推法用 $O(n^{1/2} \log^2 n)$ 的时间事先求出所有可能数组的所有询问的答案，再用 $O(n)$ 时间计算出每个小块属于哪个数组，就可以用查表在 $O(1)$ 时间内求出In-RMQ
- 查询显然仍为 $O(1)$ ，而预处理时间也不变，因为

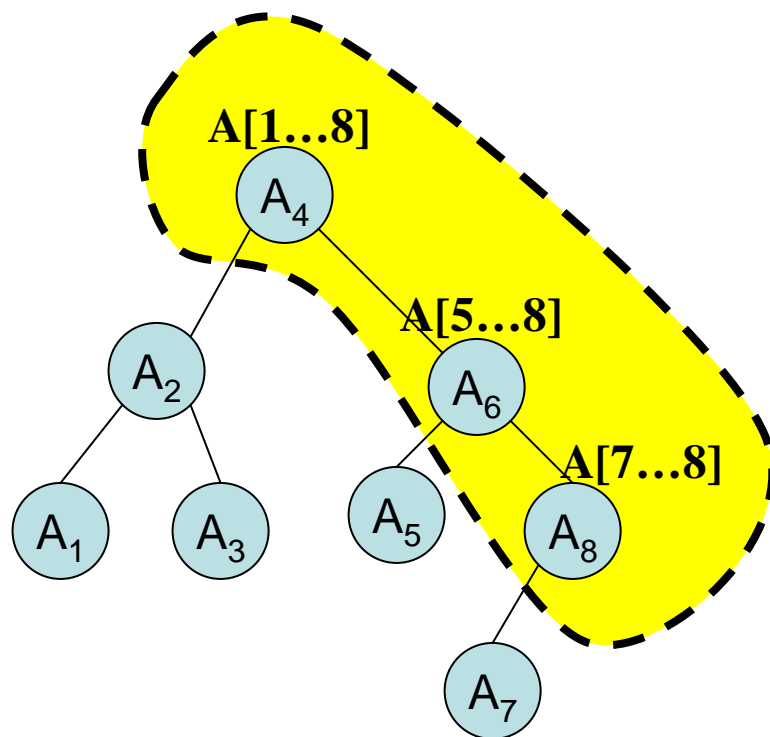
$$O(n + n^{1/2} \log^2 n) = O(n)$$

一般RMQ问题

- 一般RMQ问题也可以做到 $O(n)-O(1)$ ，但是需要进行一个迂回的转化
- **Cartesian Tree:** 根据长度为 n 的数组 A 建立，根是 A 的最小元素位置 i ，左右子树分别为 $A[1...i-1]$ 和 $A[i+1...n]$ 的Cartesian Tree
- **递归构造算法:** 每次先顺序查找到最小位置，然后递归建立，最坏 $O(n^2)$
- 增量构造可以做到 $O(n)$

Cartesian Tree的增量构造

- 从 $A[1..1]$ 的树 C_1 开始，每次加入一个数 $A[i]$ ，把 C_{i-1} 修改为 C_i
- 新数 $A[i]$ 一定在旧树 C_{i-1} 的最右路径上，而且一定没有右儿子，因此只要沿着最右路径**自底向上**把各个结点 p 和 $A[i]$ 做比较



$p < A[i]$, 则 $A[i]$ 作为 p 的右儿子**插入**，否则 p 作为 $A[i]$ 的左儿子**继续**

RMQ定理

- 由于每个结点最多进入和退出最右路径各一次，因此增量建立算法的时间为 $O(n)$
- 把数组 A 的Cartesian树记为 $C(A)$ ，则
RMQ定理： $RMQ(A,i,j)=LCA(C(A),i,j)$
- 定理很直观，这里不再叙述
- 这样，问题转化为了如下的
- **LCA问题：**在树 T 上两点 u 和 v 的最近公共祖先记作 $LCA(T, u, v)$

Tarjan算法

- Tarjan的离线算法利用并查集，初始时所有结点为白色，调用LCA(root)即可得到输出

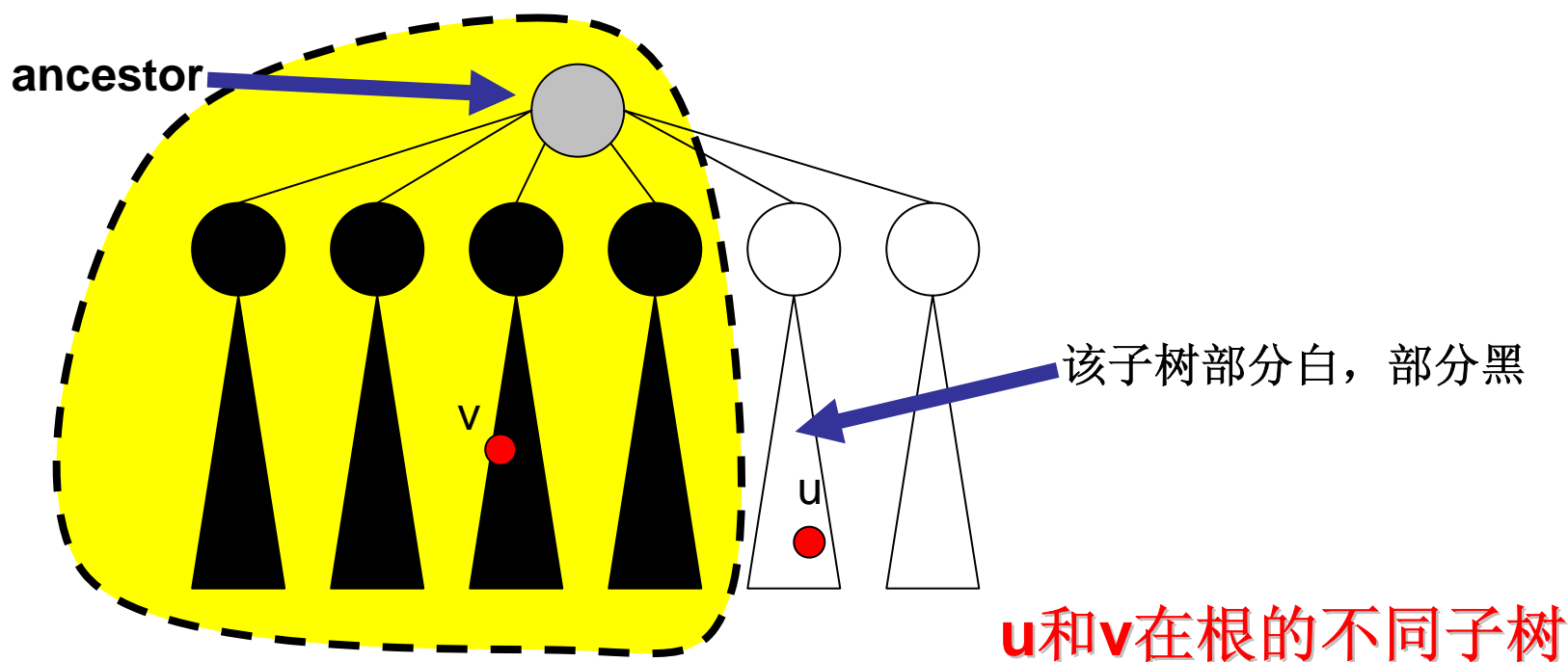
```
void LCA(u){
    makeset(u);
    ancestor[findset(u)] = u;
    v = firstS[u];
    while(v)
        { LCA(v); unionset(u, v); ancestor[findset(u)] = u; v = nextS[v]; }
    color[u] = BLACK;
    v = firstQ[u];
    while(v)
        { if(color[v] != BLACK) answer(u, v, ancestor[findset(v)]); v = nextQ[v]; }
}
```

处理u的所有儿子v

处理存在询问LCA(u,v)的所有结点v

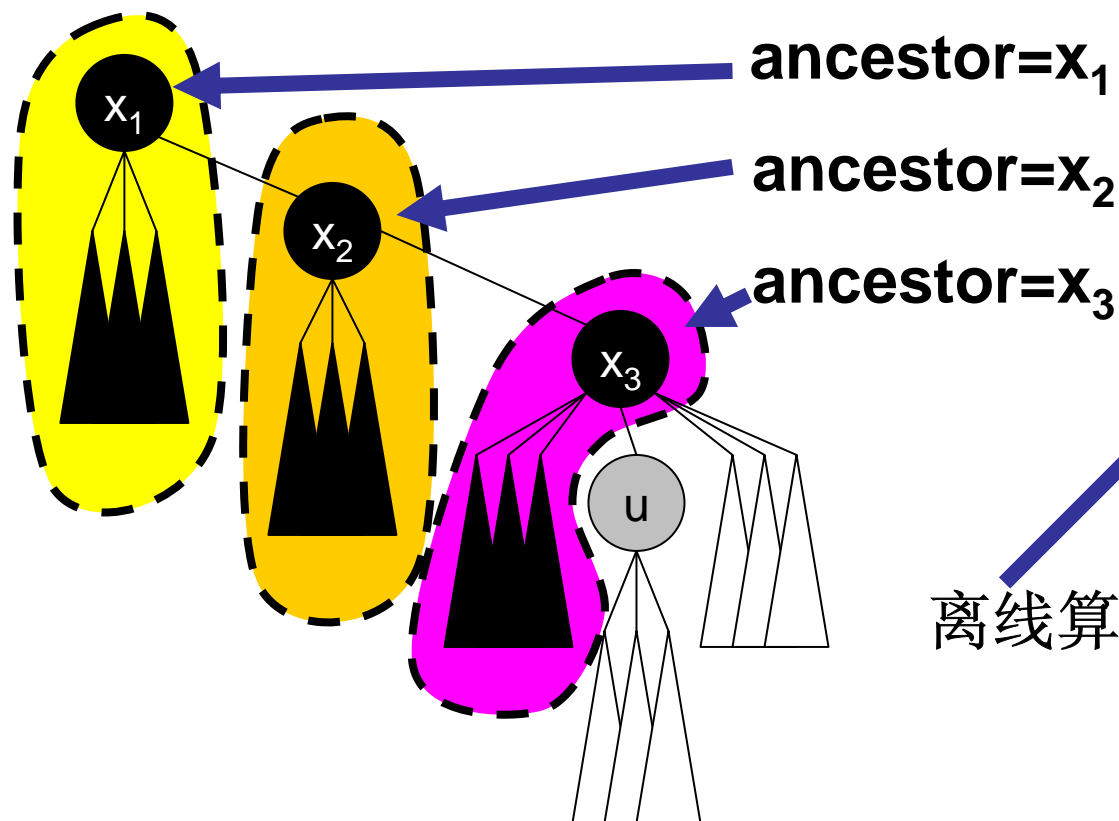
算法动机

- 忽略并查集操作和输出，算法对T进行的是深度优先遍历，每处理完一棵子树就把它合并到u所在集合中，**这个集合的元素v都满足 $\text{ancestor}[\text{findset}(v)] = u$**



集合的临时性

- u 和 v 的相对位置有多种情况
 - v 在 x_1 的**不包含 x_2** 的黑色子树中，则 $LCA(u,v)=x_1$



这些集合是临时的，
因为访问完 x_3 后 x_2 和 x_3 所在集合会**合并**，但
这没关系，因为对于
未处理的 u ， x_2 和 x_3 不
再有区别

离线算法才有可能

在线LCA算法：预处理

- 用栈显式地保存路径，则dfs可算出每个结点u的第 2^i 级祖先(父亲为0级) $anc[u][i]$ ，然后很方便的求出第k级祖先 $ancestor(u, k)$

```
void dfs(int d , int p){
    int i=1;
    stack[d] = p; depth[p] = d;
    for(j=0; i<=d; j++, i<=<1) anc[p][j] = stack[d-i];
    for(j=son[p]; j; j=next[j]) dfs(d+1 , j);
}

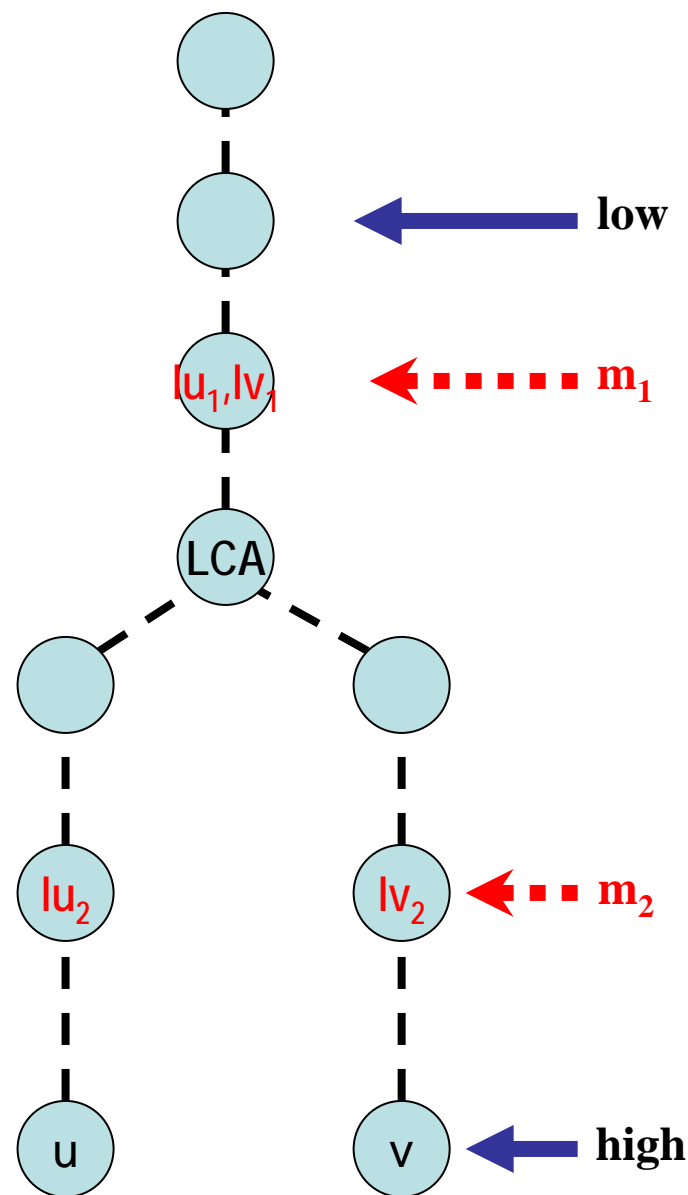
int ancestor(int u, int k){
    while(k>0){ u = anc[u][log[k]]; k -= (1<<log[k]); }
}
```

在线LCA算法： 查询处理

- 若非祖先后代关系， $LCA(u,v)$ 是满足下式的最大 w
 $\text{ancestor}(u, \text{depth}[u]-w) = \text{ancestor}(v, \text{depth}[v]-w)$
- 算法一：直接二分 w ，每次检查两边是否相等。由于计算 ancestor 的时间是 $O(\log n)$ ，因此询问的总时间为 $O(\log^2 n)$
- 算法二：令 $D = \min\{\text{depth}[u], \text{depth}[v]\}$ ，首先令 $u = \text{ancestor}(u, \text{depth}[u]-D)$, $v = \text{ancestor}(v, \text{depth}[v]-D)$ ，即把 u 和 v 上升到同一个高度。显然上升后LCA不变

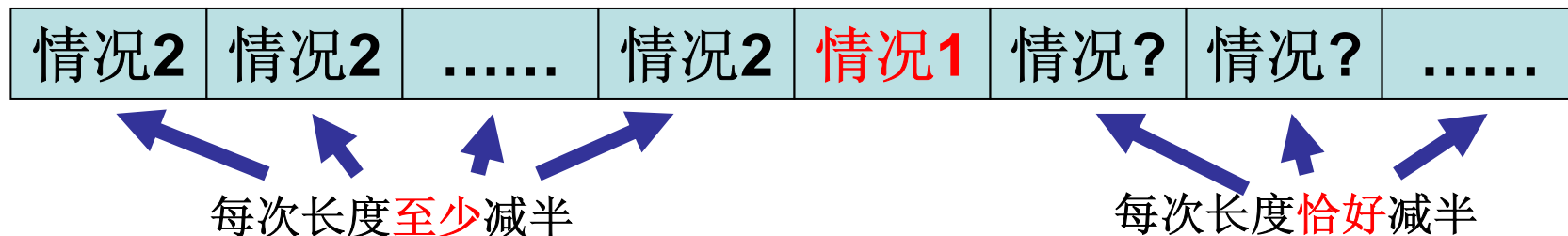
算法二

- 设 w 的范围在 $[low, high)$ （左闭右开），上升到同一位置 d 后区间初值设为 $[0, d)$
- 每次取 $lvl = \log(high - low - 1)$ ，计算 $lu = anc[u][lv]$ ， $lv = anc[v][lv]$ ，设 $m = high - 2^{lv}$
 - 若 $lu = lv$ ，说明 m 处为公共祖先，设 $low = m$
 - 否则设 $high = m$ ，更新 $u = lu$ ， $v = lv$



算法二分析

- 算法二的正确性是显然的，但时间复杂度却不是显然的：区间长度为 $n=2^k+1$ 时， $lvl=k$ ，因此 $m=high-2^k=low+1$ ，若设 $low=m$ 则区间长度只减1，但这样的情况最多一次
 - 情况1：设 $low=m$ ，则区间长度一定变为2的幂，今后区间长度每次减半($2^k \rightarrow 2^{k-1}$)
 - 情况2：设 $high=m$ ，本次至少把区间长度减半



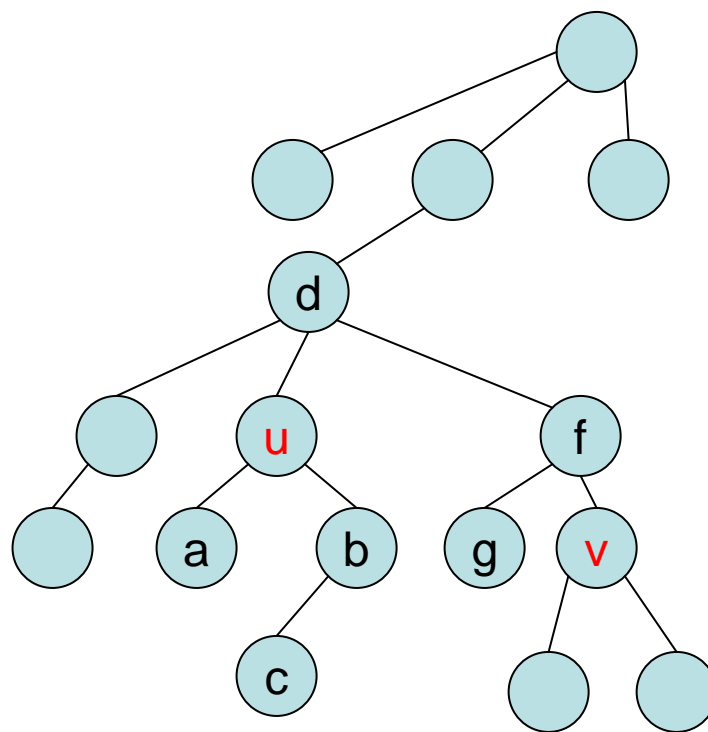
DFS序列

- 刚才的两个在线算法比较巧妙，但效率仍不令人满意。下面把LCA转化为 ± 1 -RMQ，从而得到一个 $O(n \log n) - O(1)$ 算法
- **DFS序列**：给树 T 做深度优先遍历，记录下每次到达的结点。第一个记录的结点是根 $\text{root}(T)$ ，每经过一条边都记录它的端点。由于每条边恰好经过了两次，因此一共将记录 $2n-1$ 个结点，用 $E[1 \dots 2n-1]$ 表示

问题的转化

$u \rightarrow v$ 的路径为 **u**aubcbudfg**v**

- 用 $R[i]$ 表示 E 数组中第一个值为 i 的元素下标，那么对于任何 $R[u] < R[v]$ 的结点 u, v 来说，DFS 从第一次访问 **u** 到第一次访问 **v** 所经过的路径应该是 $E[R[u], \dots, R[v]]$ ，令 $L[i]$ 表示结点 $E[i]$ 的深度，则



$$LCA(T, u, v) = \begin{cases} RMQ(L, R[u], R[v]) & R[u] \leq R[v] \\ RMQ(L, R[v], R[u]) & R[u] > R[v] \end{cases}$$

总结

- ± 1 -RMQ问题可以在 $O(n)-O(1)$ 时间解决
- 借助于**DFS序列**，LCA问题可以在 $O(n)$ 时间内转化为 ± 1 -RMQ问题
- 借助于**Cartesian Tree**，一般RMQ问题可以在 $O(n)$ 时间内转化为LCA问题
- **结论：**LCA问题和一般RMQ问题都可以在 $O(n)-O(1)$ 时间内解决，达到了理论下界