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

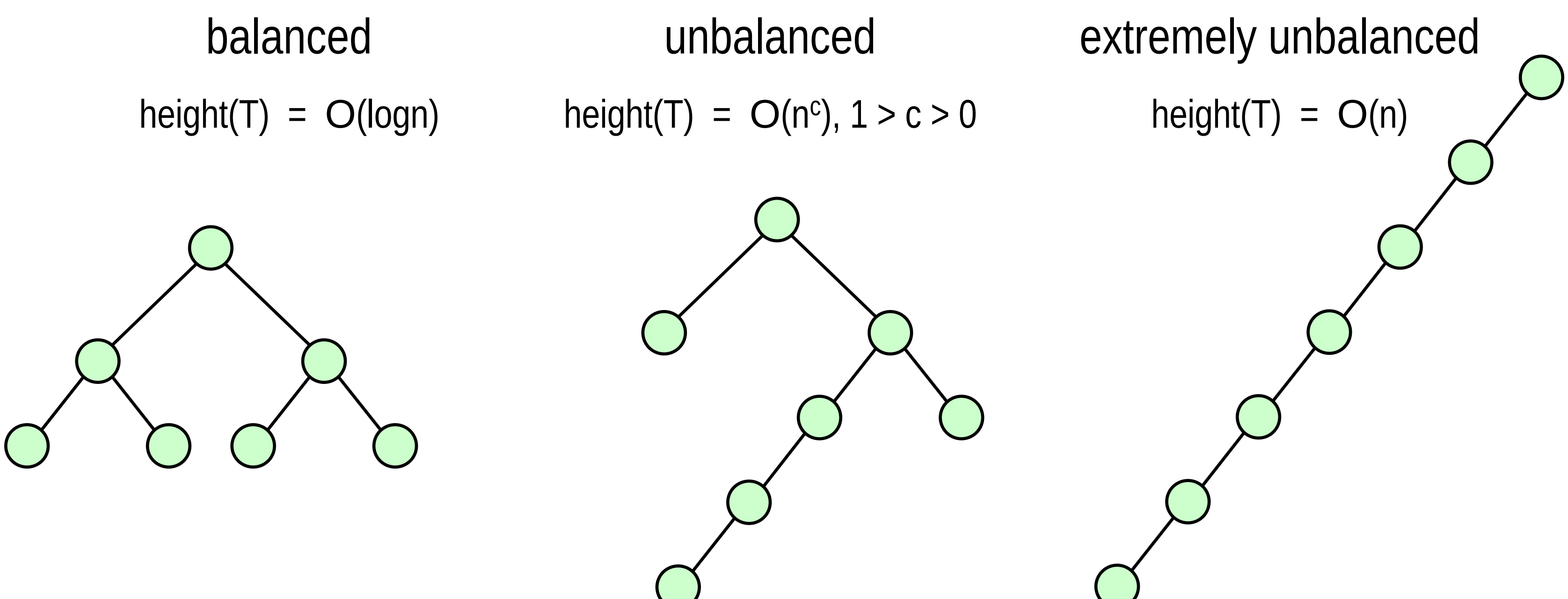
一、平衡二叉树

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

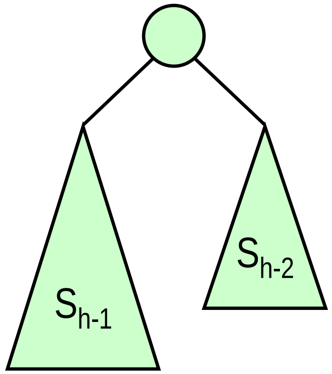
- 越平衡, 各种操作的速度越快
- 什么是平衡: 渐进意义下树高$h=O(\log n)$
  - 固定结点数, 则树高越小越好



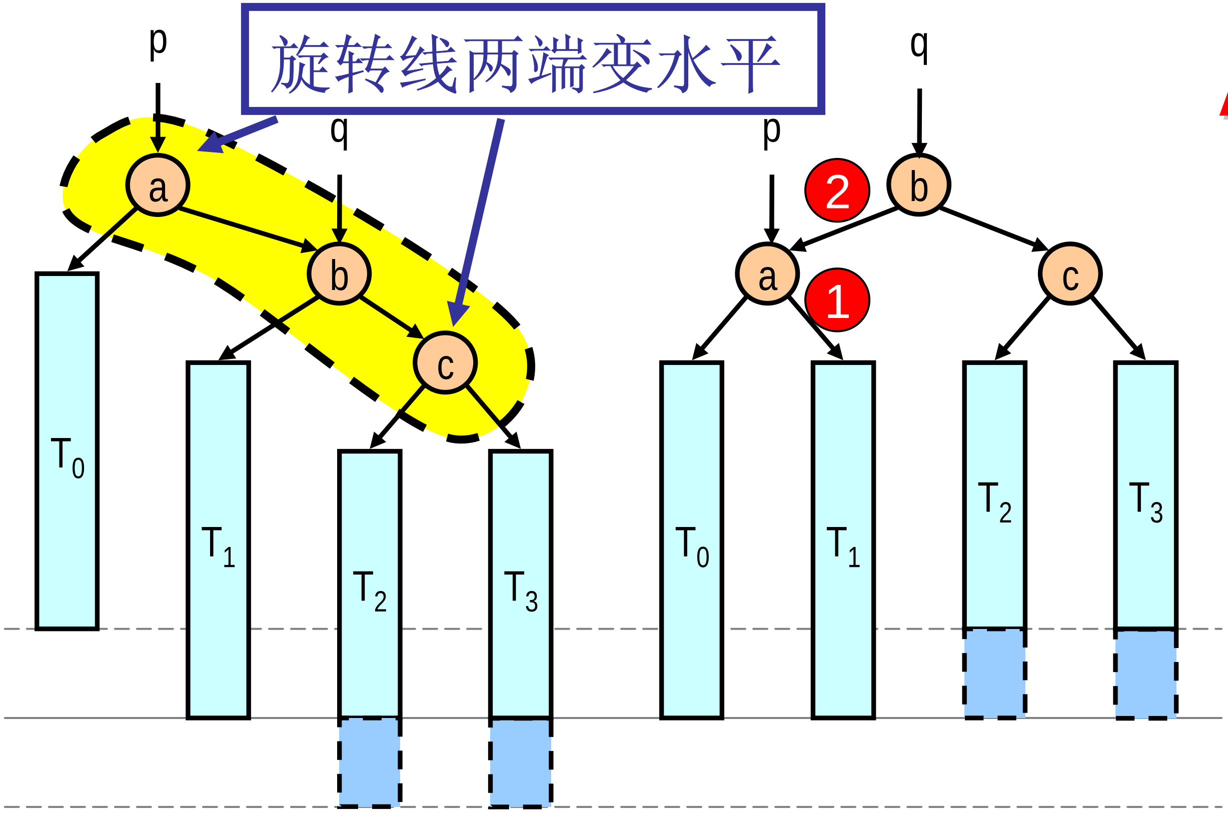
一、AVL树

- **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)>=F_{h+3}-1, 因此

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



AVL的单旋转



哪棵子树高度过大
就以哪个儿子为轴

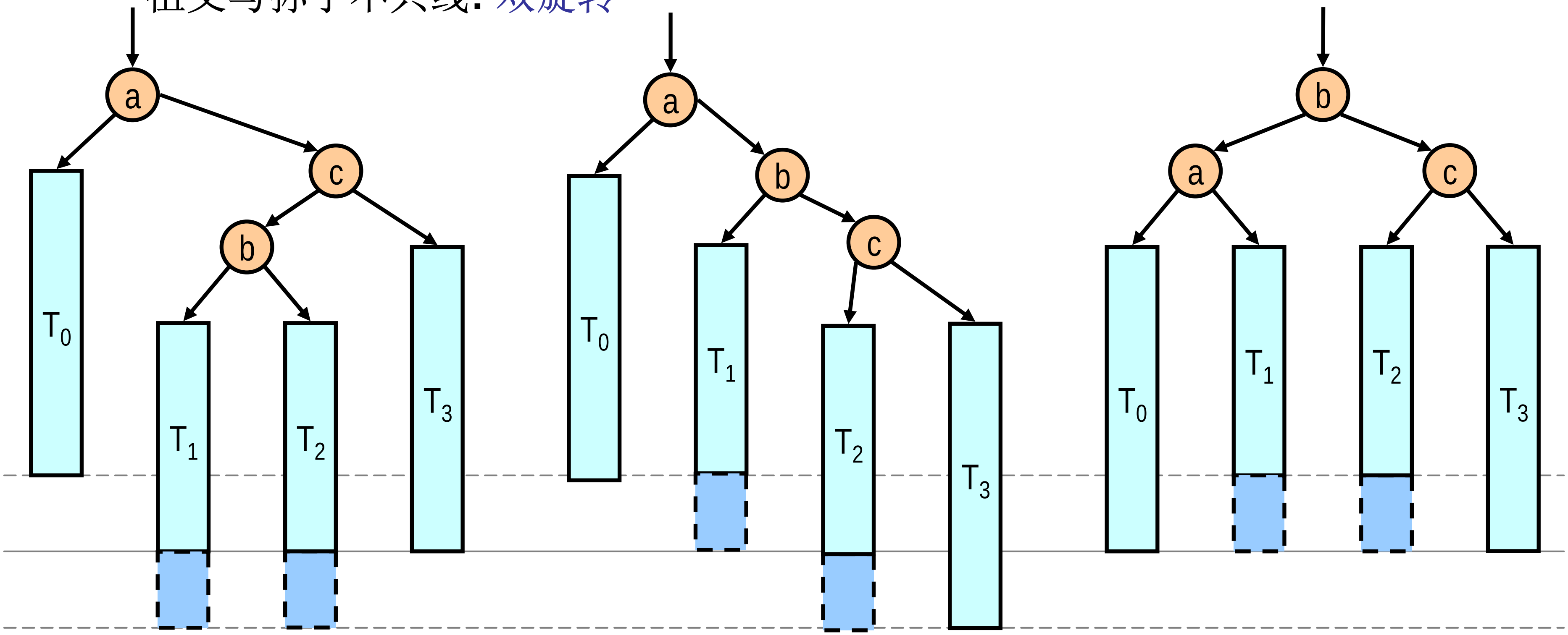
```
void rotatewithRightChild (int& p){
    int q = right[p]; right[p] = left[q]; left[q] = p;
    height[p] = max(height[left[p]], height[right[p]]) + 1;
    height[q] = max(height[right[q]], height[p]) + 1;
}
```

AVL的双旋转

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




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

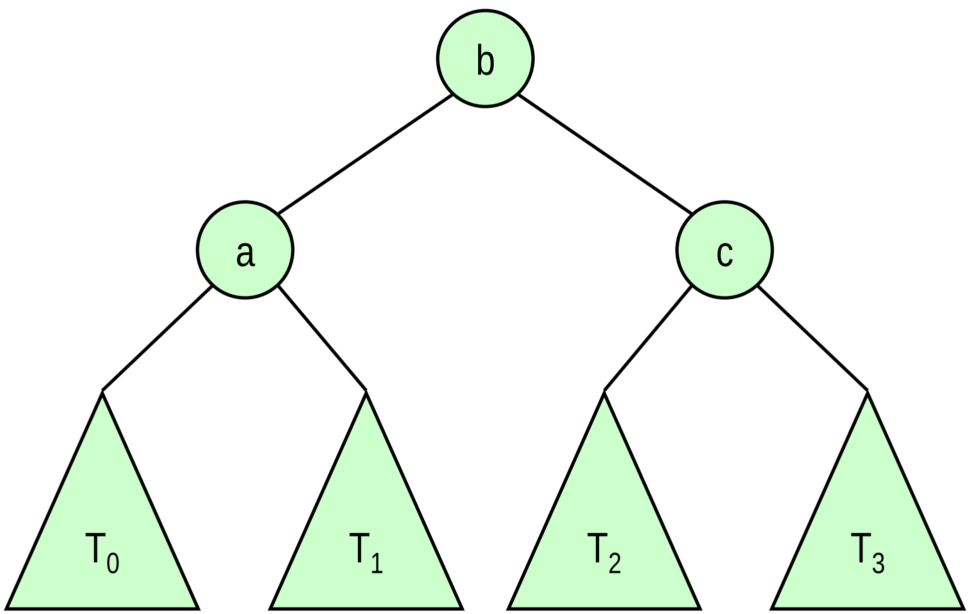
插入算法: 从插入元素开始向上找第一个祖父不平衡的点**x**, 设父亲为**p(x)**, 祖父为**g(x)**…

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

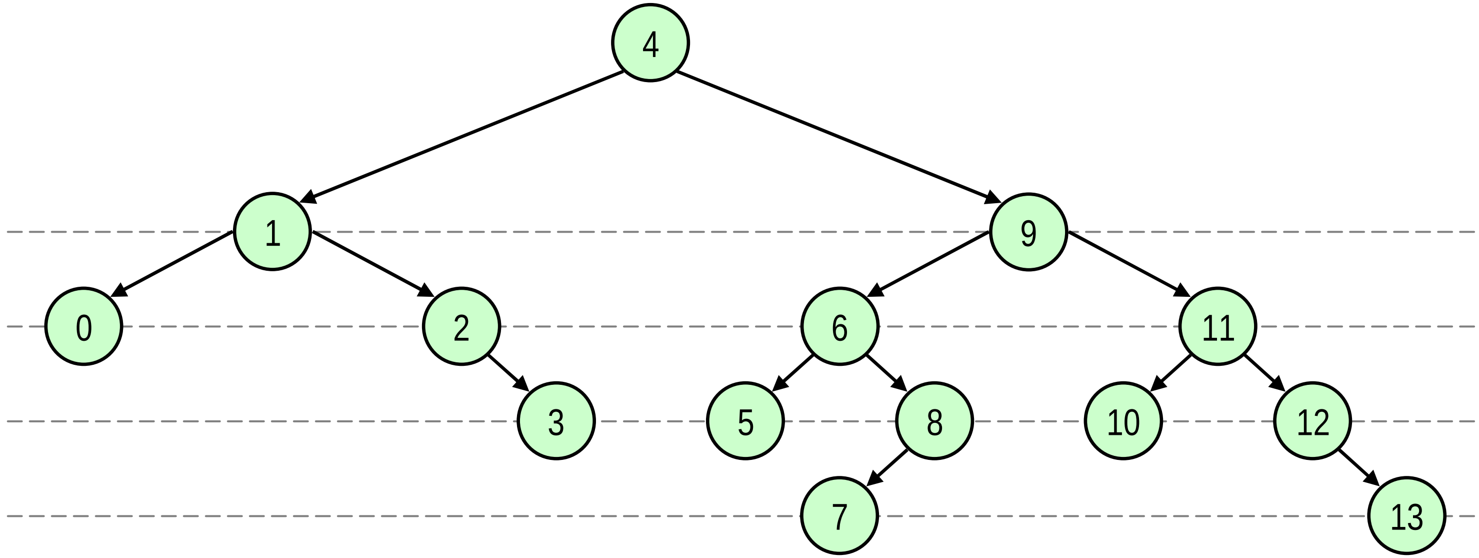
插入的统一算法

- 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(logn)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



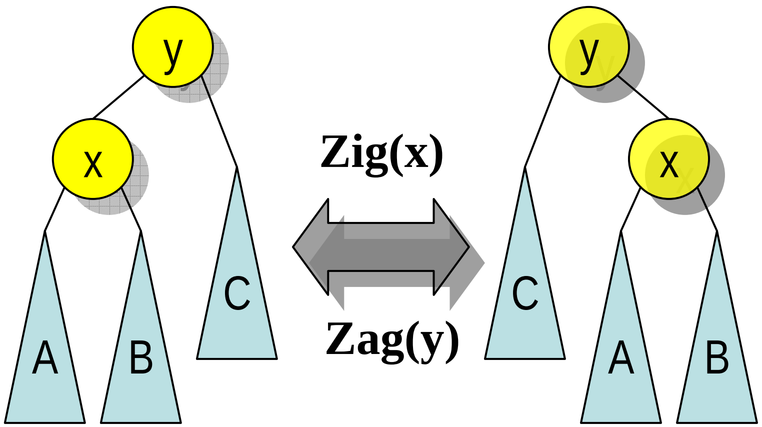
AVL = insert {4 1 9 0 2 6 11 3 5 8 10 12 7 13}

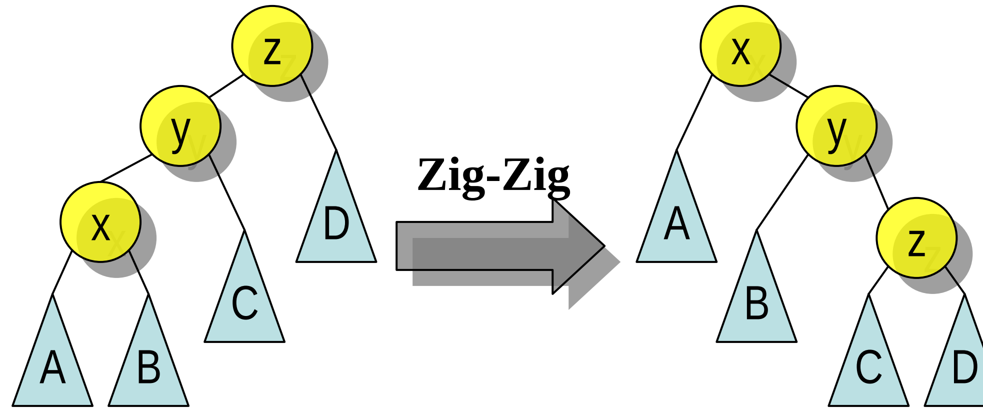
<table>
<tr><th>key</th><th colspan="2">节点类型</th><th>删除方法</th><th>需重新平衡?</th></tr>
<tr><td>7、13</td><td colspan="2" rowspan="2">叶子</td><td rowspan="2">直接摘除</td><td>不需要</td></tr>
<tr><td>0、3、5、10</td><td>需要</td></tr>
<tr><td>12</td><td rowspan="6">内部节点</td><td rowspan="2">无左孩子
（右子树至多包含单个节点）</td><td rowspan="2">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

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=father(x), z=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

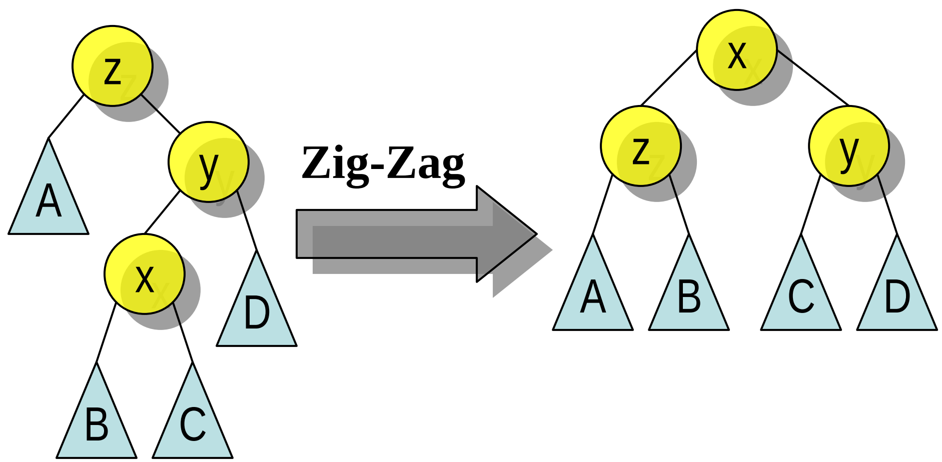
三种情况的处理





情况1. **y**是根

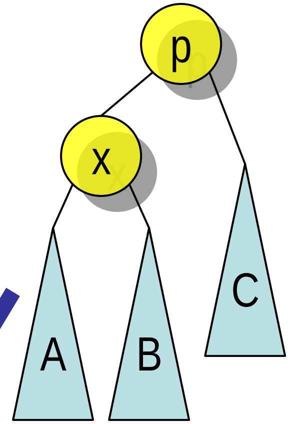
情况2. **y**不是根，**x, y, z**共线

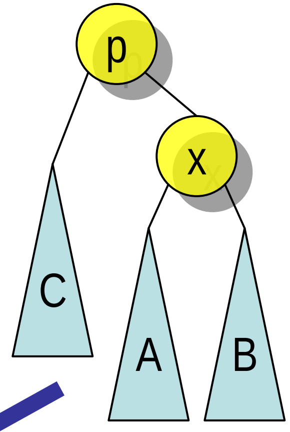


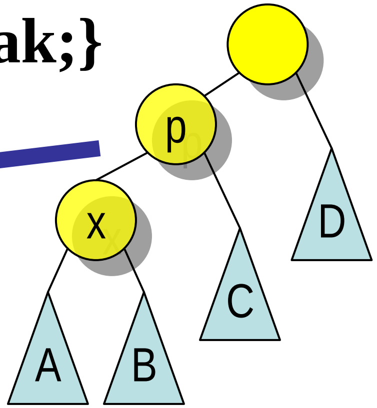
情况3. **y**不是根，**x, y, z**不共线

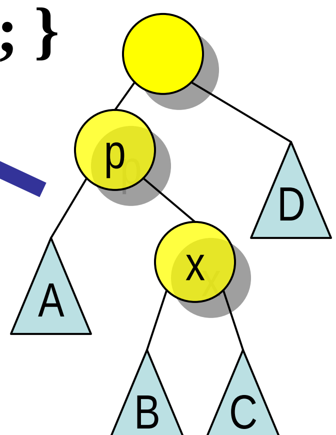
伸展操作的实现

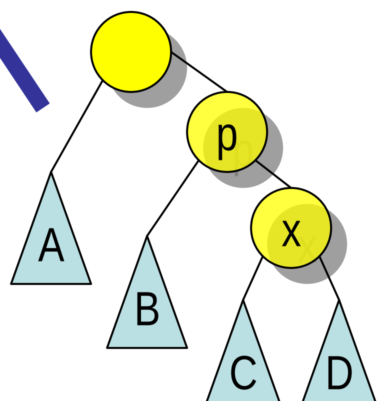
```
void splay (int& x , int& s){
  int p;
  while (father[x]){
    p = father[x];
    if (!father[p]){ if(x == left[p]) Zig(x);else Zag(x);break;}
    if(x == left[p]){
      if(p == left[father[p]]){ Zig (p); Zig(x);}
      else{ Zig(x); Zag(x ); }
    }else{
      if(p == right[father[p]]){ Zag(p);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

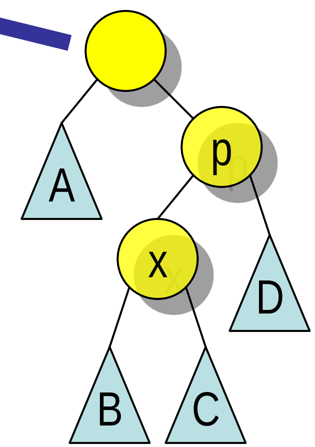










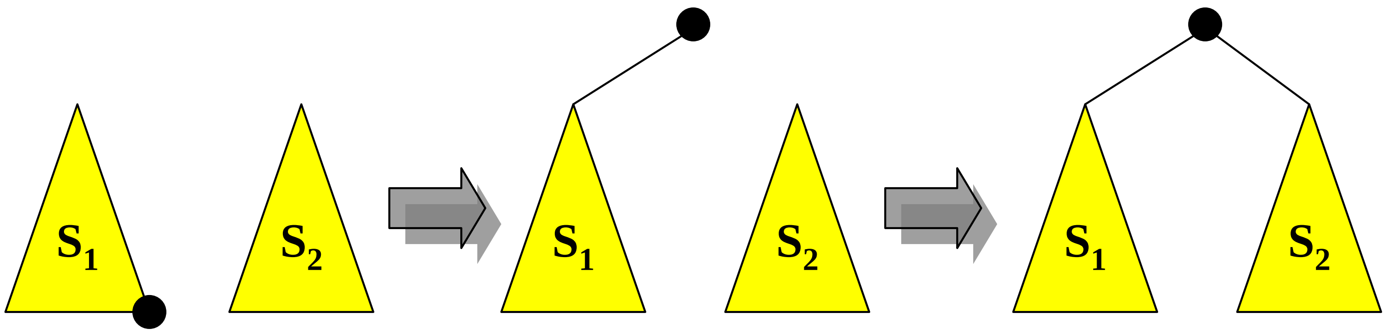


基本操作

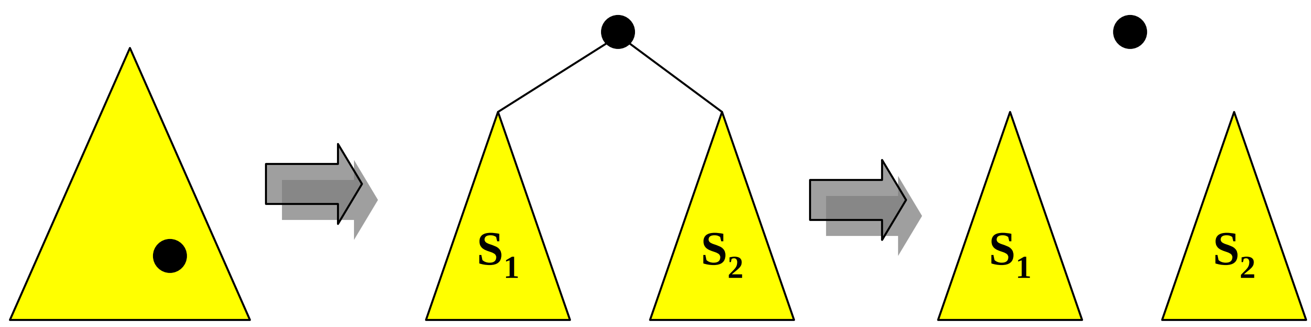
- Find(x, S): BST查找, 然后Splay
- Insert(x, S): BST插入, 然后Splay
  – 方法二：查找x后Join (留作练习)
- Delete(x, S)
  – Find(x, S), 合并x的左右儿子
- Join(S_1, S_2): 见后
- Split(x, S): 见后
- 伸展操作是基础!

合并与分离

- Join(S_1, S_2): 伸展S_1中的最大元素



- 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];
}
```

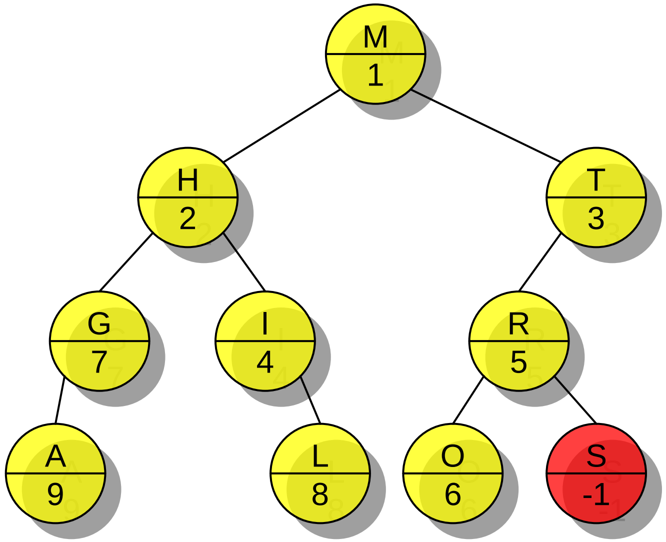
小结

- 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(logn) (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

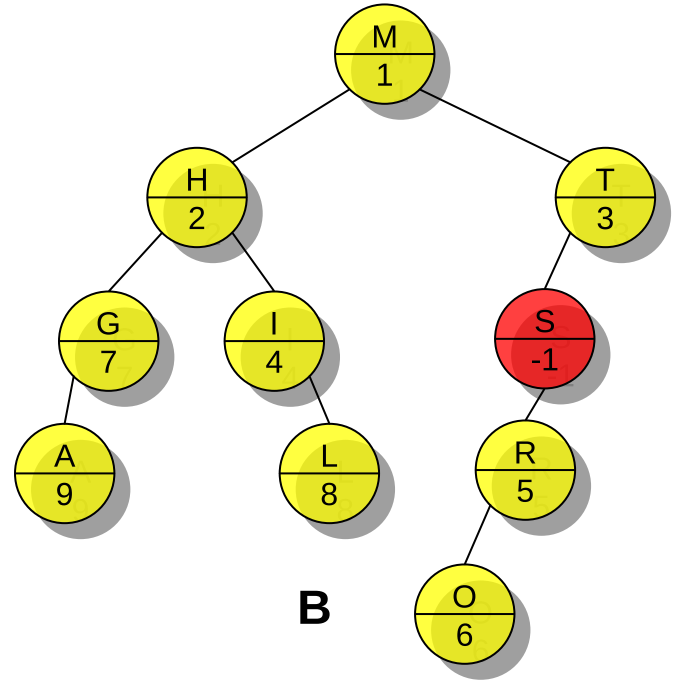
三、Treap

- 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(logn)
- 重要结论：key和priority确定时，treap惟一

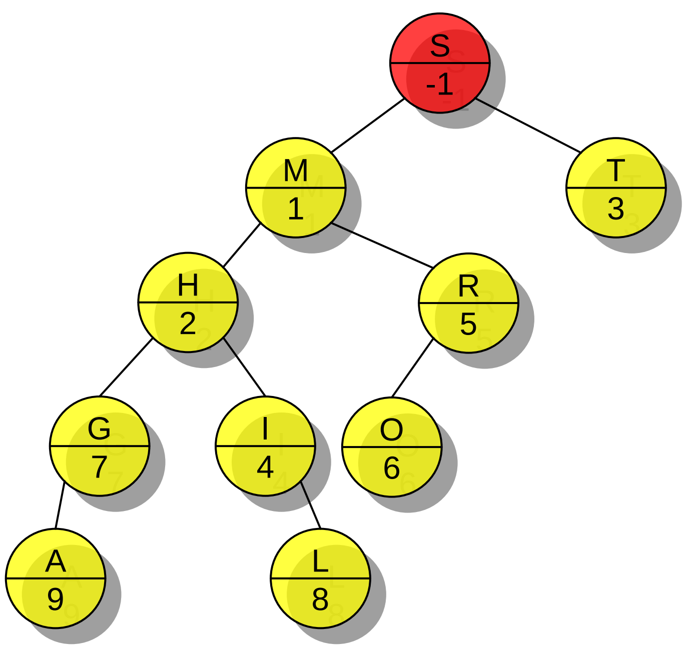
插入：ABCD
删除：DCBA



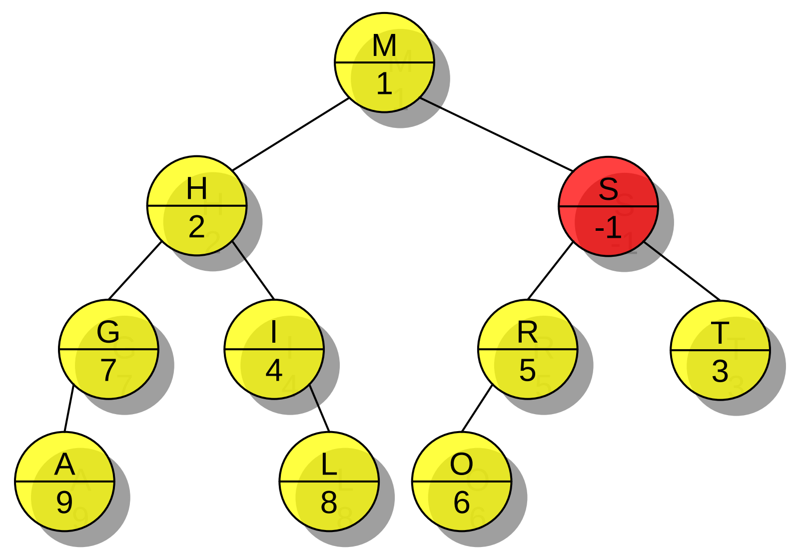
A



B



D



C

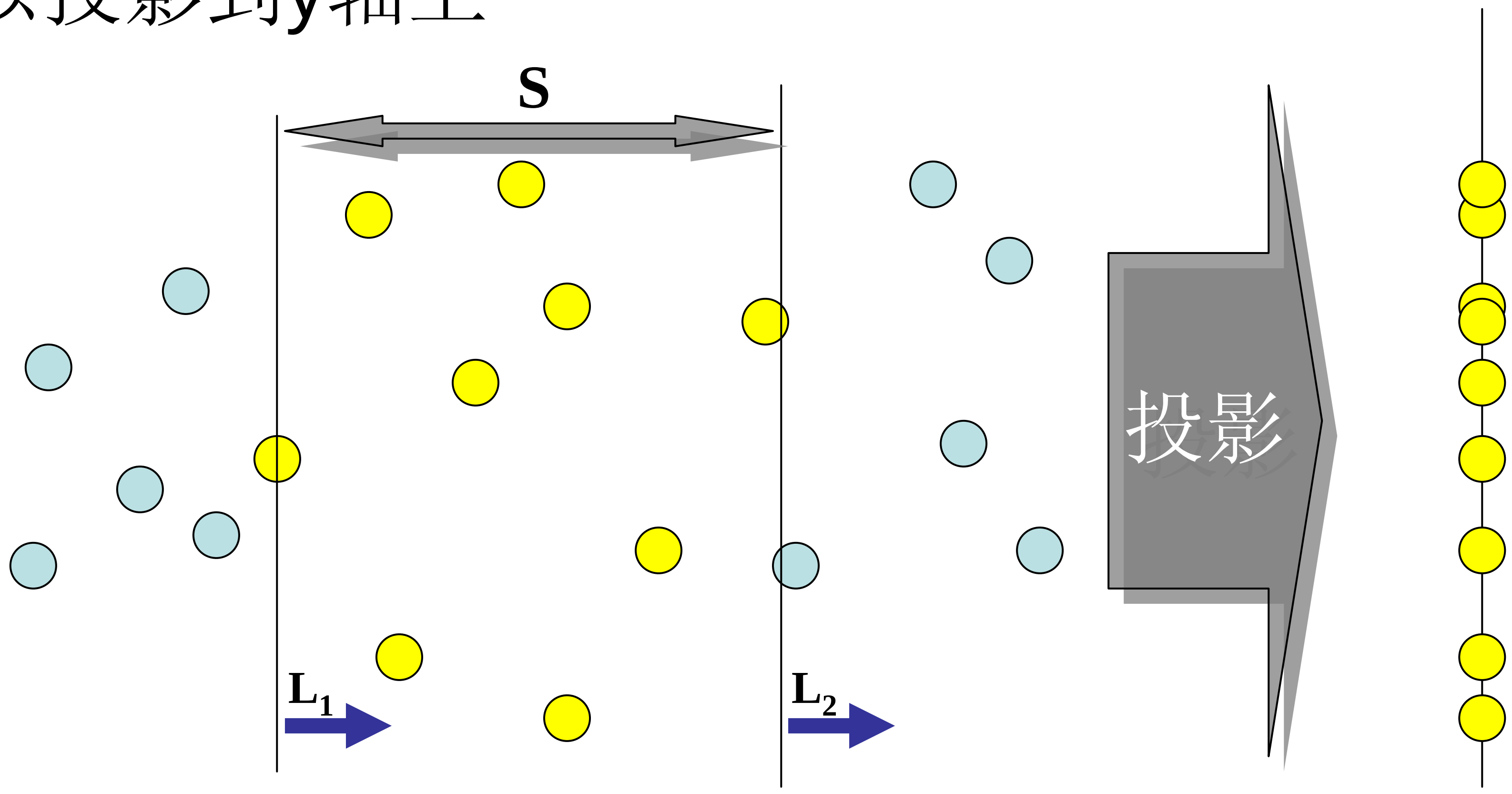
```
void insert (int x , int& p){
  if (!p)
      p = newnode(x, myrand());
  else if(x < key[p])
      { insert(x, left[p]); if(pr[left[p]] < pr[p]) rotateWithLeftChild(p); }
  else if(x > key[p])
      { insert(x, right[p]); if(pr[right[p]] < pr[p]) rotateWithRightChild(p); }
}

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<=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<=s，w<=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，它也应当被计算在内。

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

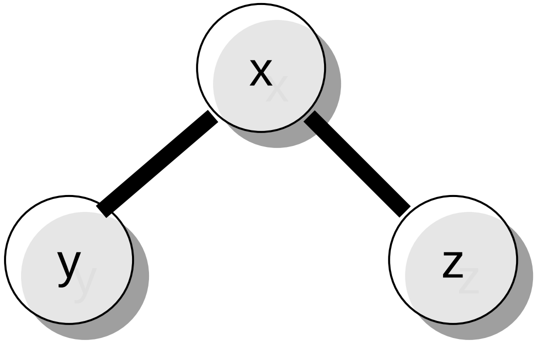
- 投影之后的问题变为了：在数轴上找一个长度为W的区间，包含尽量多的点
- 每个点y拆成两个事件(y, +1), (y+w-1, -1)，表示区间右端点在y和y+w-1时该点进入/退出扫描区间，则扫描过程变成了累加过程，点数最多的位置对应了最大前缀和
- 主算法：从左到右处理各个点进入/退出L_1-L_2间区域的事件，维护最大前缀和

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(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nlogn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)。

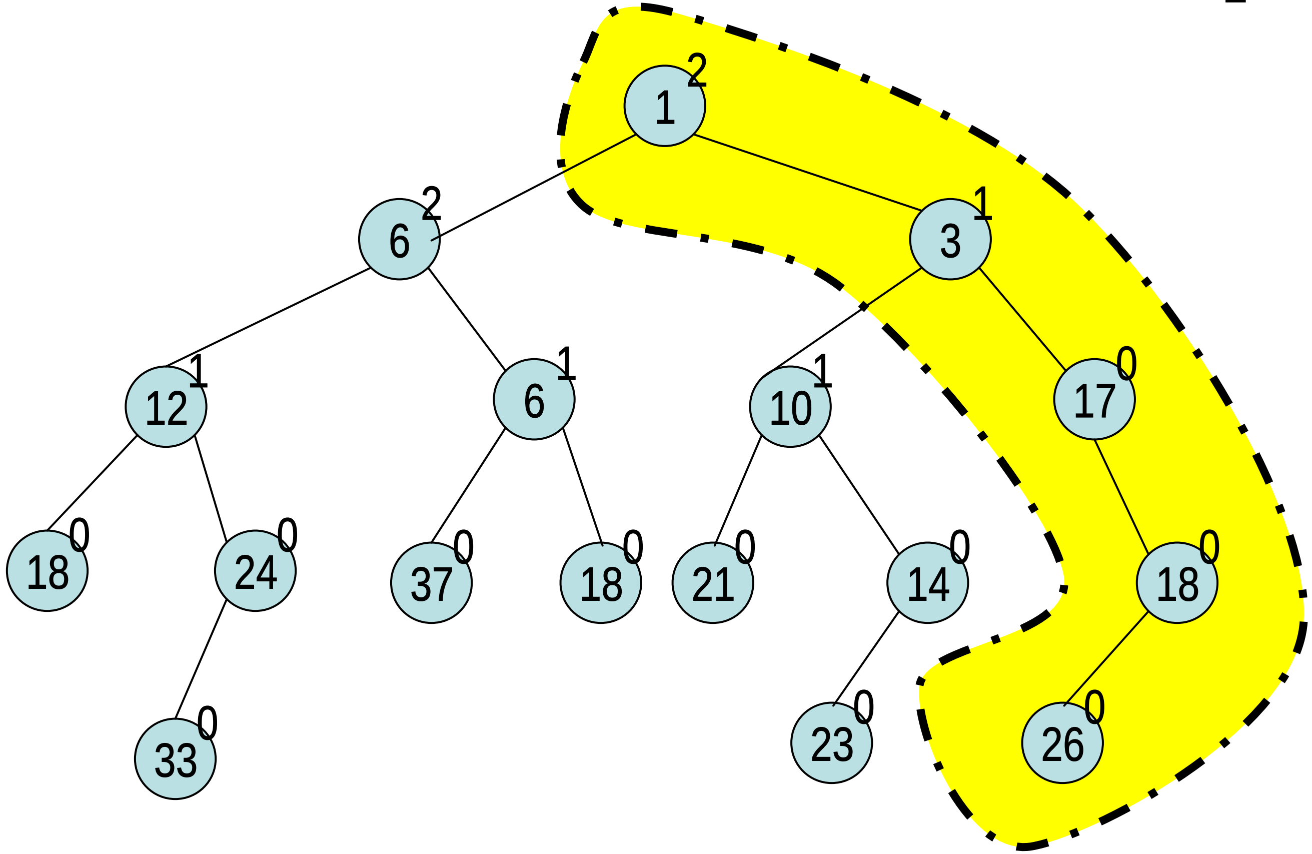
左偏树

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

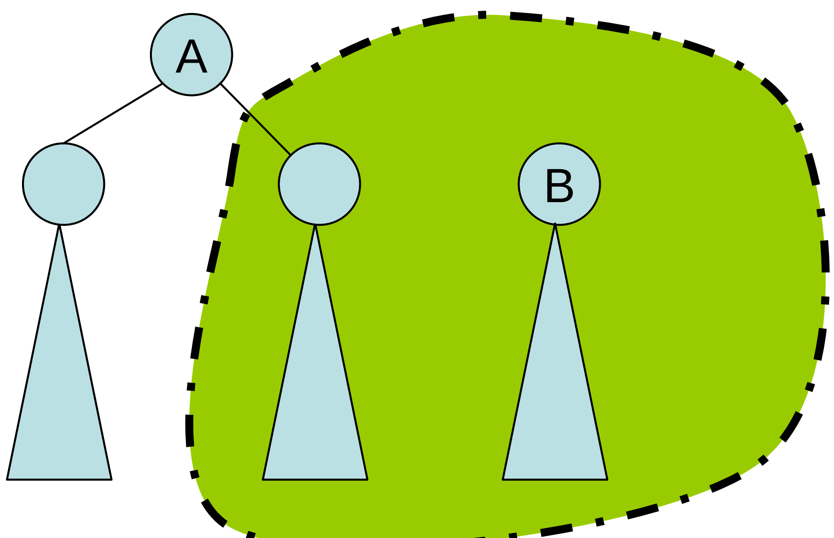
左偏树的高度

- 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<=B的根（否则交换A和B），把A的根作为新树的根，合并right(A)和B
  - 第二步：如果合并后right(A)>left(A)，交换
  - 第三步：更新dist(A)=dist[right[A]]+1



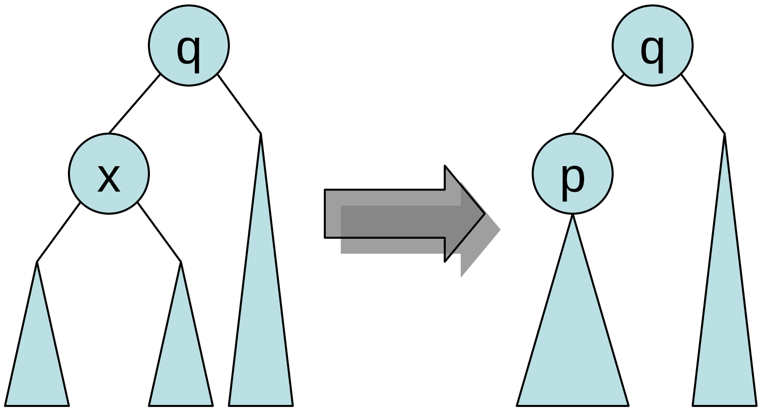
合并操作的分析

- 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$O(\log N_1+\log N_2)$

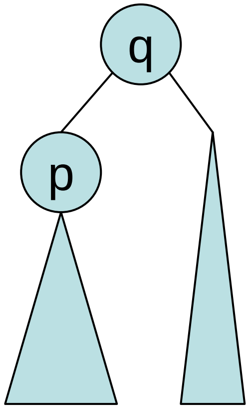
```
int merge ( int a , int b){
  if (!a) return b; else if (!b) return a;
  if(key[b]<key[a]) swap(a, b);
  right[a] = merge (right[a], b);
  if(dist[right[a]] > dist[left[a]]) swap(left[a], right[a]);
  if (!right[a]) dist[a] = 0;
  else dist[a] = dist[right[a]] + 1;
  return a;
}
```

其他操作

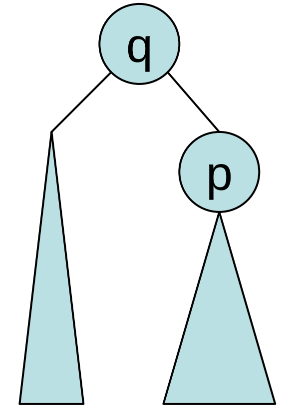
- 插入：与单结点堆合并
- 删除最小值：合并根的左右子树
- 建立：将n个结点放入FIFO队列，每次取出队首两个堆进行合并，放入队尾，时间O(n)
- 删除已知结点**x**：首先合并x的左右子树，得到子树p，然后进行讨论



删除已知结点



- 设合并后p的距离为dist(p)，则
  - dist(p)+1=dist(q)，删除结束
  - dist(p)+1<dist(q)，更新dist(q)=dist(p)+1. 若p是q的左子树，交换子树后处理**q**的父亲
  - dist(p)+1>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(logn)**

```
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];
  }
}
```

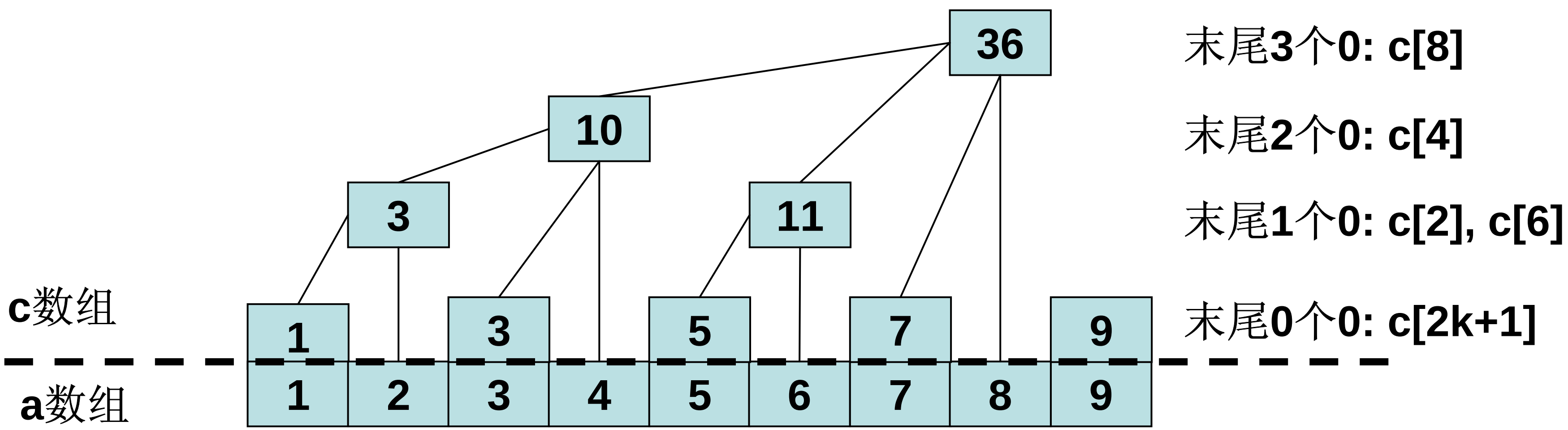
三、线段树和树状数组基础

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$c[i] = a[i-2^k+1]+a[i-2^k+2]+\ldots+a[i]$
  - 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

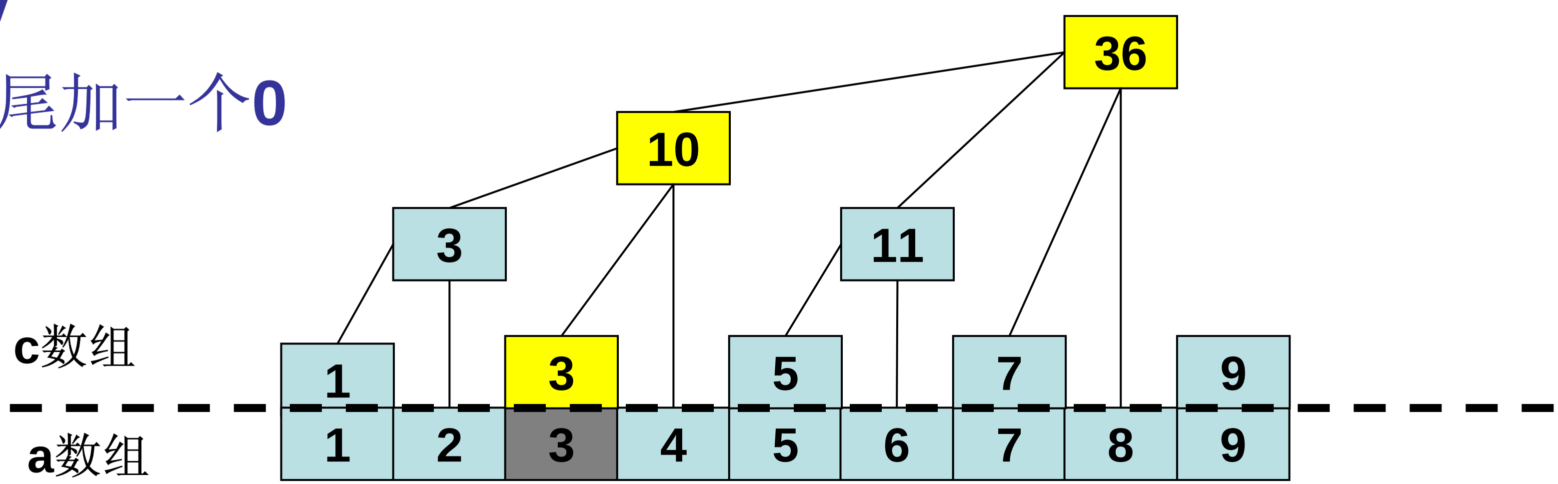


第**i**层末尾有**i**个零，度数为**i+1**，定义式**2^i**项

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]

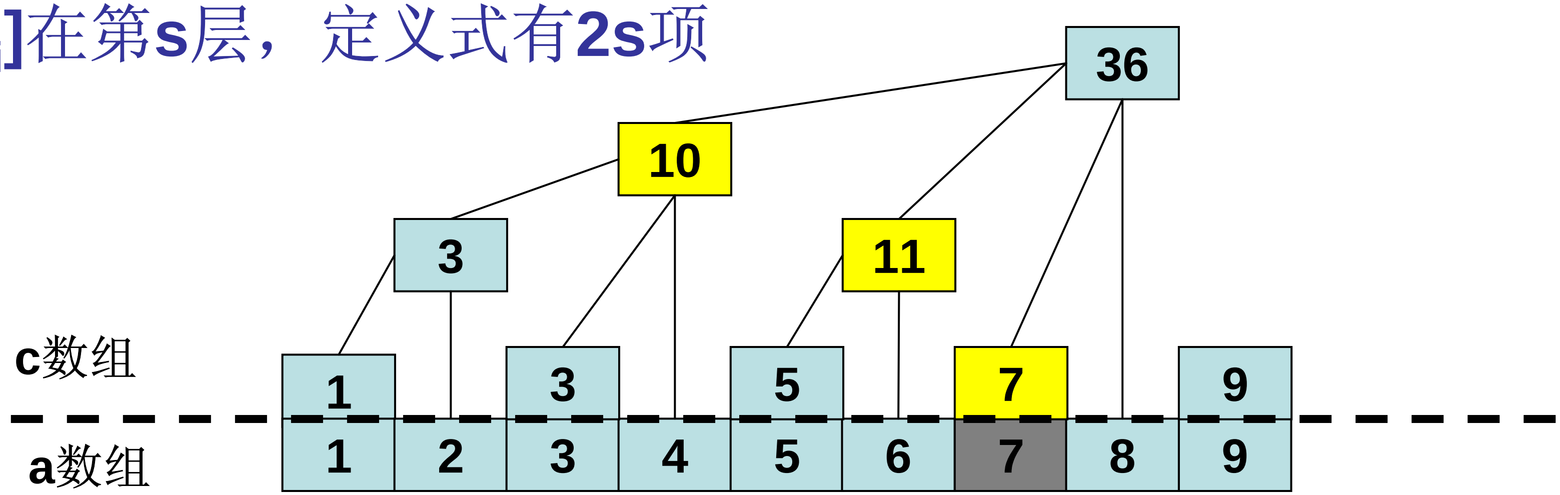
给末尾加一个**0**



求和操作

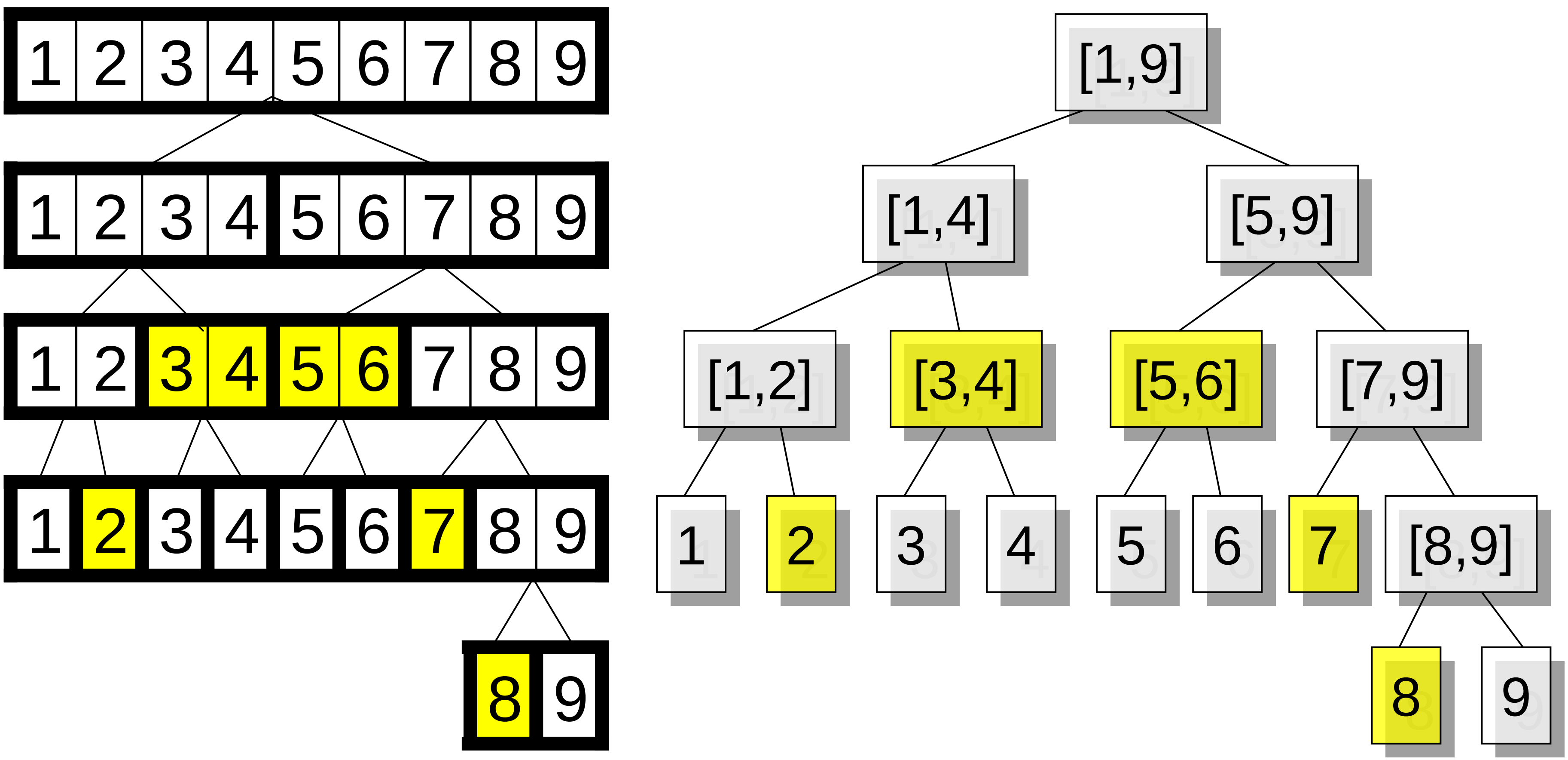
- 设要求前k项的和，则从最后一项开始
  - $k_1=k$
  - **$k_{i+1}=k_i-2^s$**, s为k_i二进制末尾0的个数
  - k=7, 则$k_1=7$, $k_2=k_1-2^0=6$, $k_3=k_2-2^1=4$, $k_4=k_3-2^2=0$，因此前7项和为c[7]+c[6]+c[4]

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

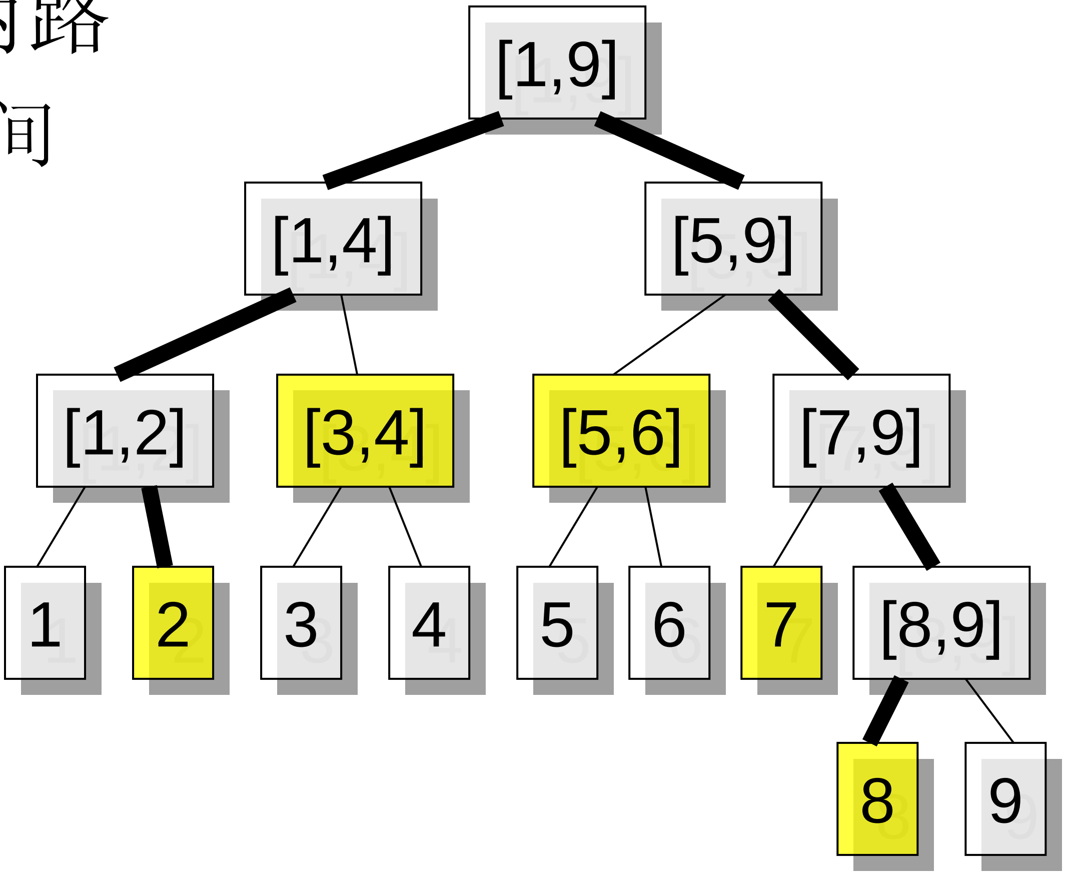


性质

- 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logL
- 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
  - ADD(i, j, k): 给A[i], A[i+1], … A[j]均增加k
  - QUERY(i): 求A[i]
- 先看看是否可以沿用刚才的附加信息
  - 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

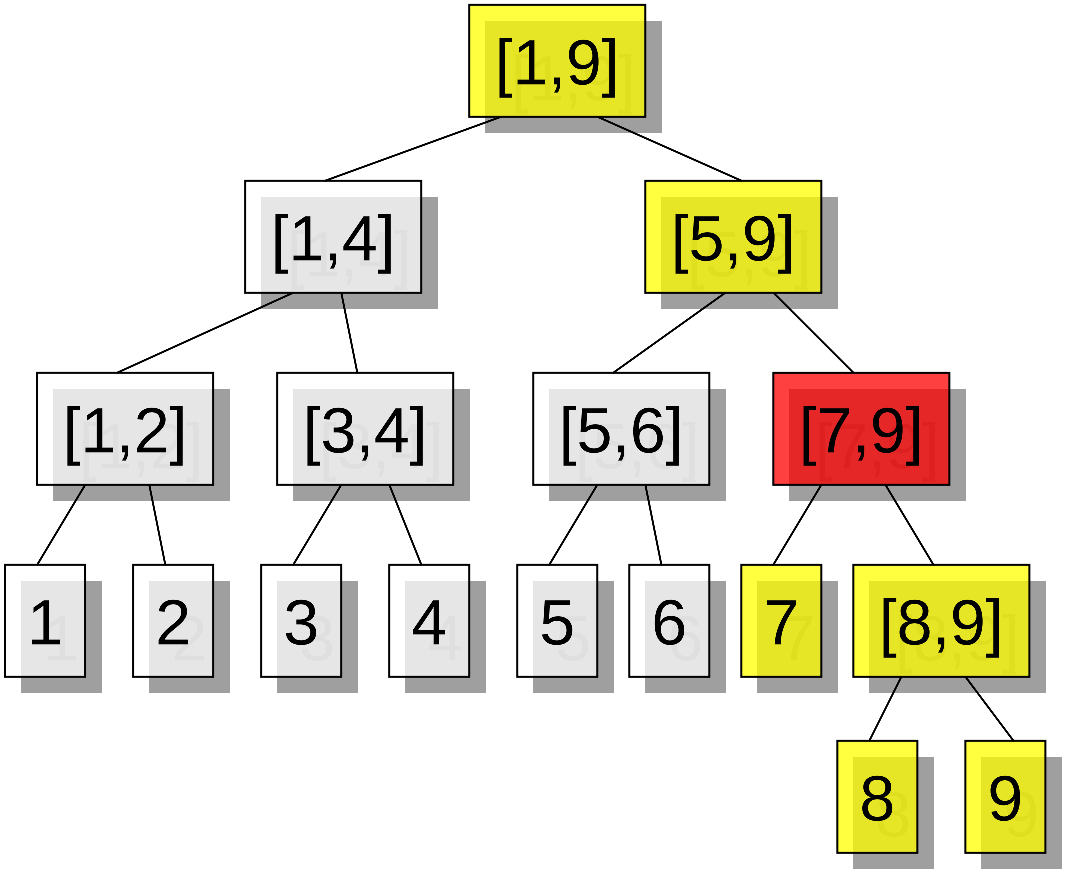
- 包含n个元素的数组A
  - ADD(i, j, k): 给A[i], A[i+1], … A[j]均增加k
  - SUM(p, q): 求A[p]+A[p+1]+…+A[q]
- 显然动态统计问题I和II都是它的特殊情况
  - 问题I中, ADD操作的i=j
  - 问题II中, SUM操作的p=q
- 由于ADD操作和问题II一样, 这里沿用它的ADD实现, 那SUM怎么办?

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(logn)

例1. 动态区间最小值

- 包含n个元素的数组A
  - MODIFY(i, j): 设A[i] = j
  - MIN(p, q): 求min{A[p], A[p+1],…,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[I, 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
  - Add(x, y): 增加区间[x, y] (1<=x<y<=n)
  - 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Add(x, y)的条数

覆盖长度可以维护么?

- 考虑根结点. 如果c(root)>0, 则整个区间都被覆盖, 返回L, 但如果c(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

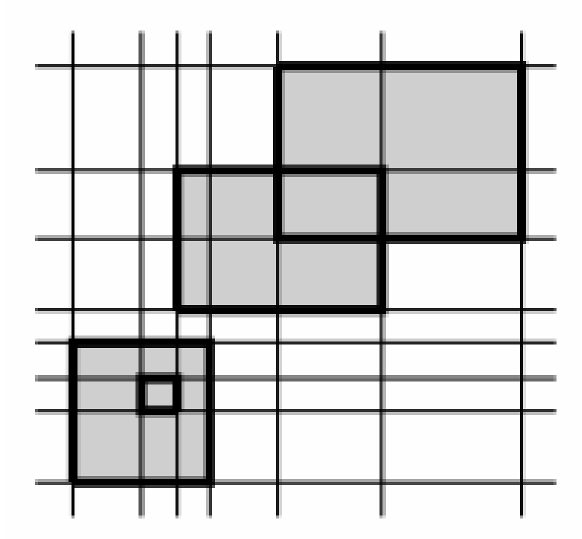
- 设l(p)表示**以p为根的子树中的所有Add操作**在p对应的区间中覆盖了多大长度, 则Add(1, n)只需要修改根
- 每个原子区间的插入、删除都只影响它的直接祖先, 插入/删除后自底向上用递推法维护即可

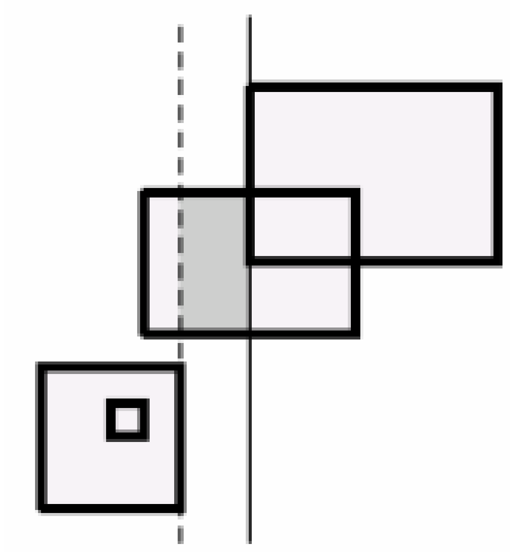
例3. 火星地图

- 2051年，科学家们探索出了火星上*N*（*N*≤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(logn)，共O(nlogn)





例4. 01矩阵

- 给n*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: $C(x_0,y_0,x_1,y_1)$等价为改变以下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…n]
- 实现两个操作
  - Modify(i,j), 设A[i] = j
  - Query(i,j,k), 返回A[i..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logW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

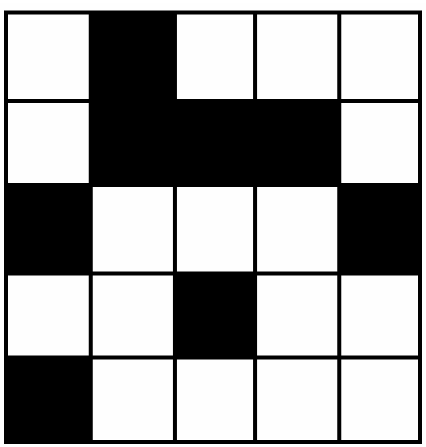
- 统计原子区间内一共有多少个数比**W**大
  - 区间内的数已排序，用二分每个区间求比W大的数logn
  - 累加所有2logn个区间比W大的数，共$\log^2 n$
  - 总时间： $\log W * \log^2 n$
- 实现: 一个归并排序可以同时构造线段树和每个节点内的排序数组. 空间:O(nlogn)

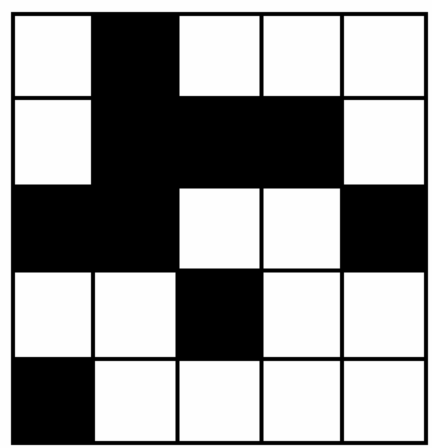
分析

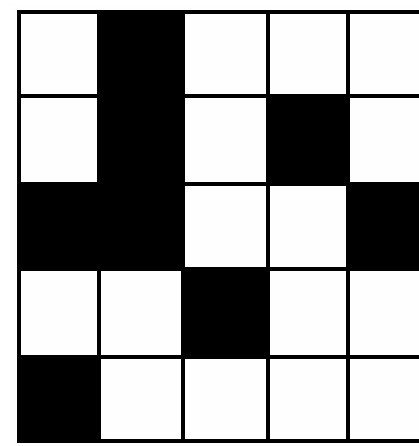
- 有修改的情形: 每个结点不能用有序表了，而需要是一棵平衡树
- 每次Modify需要修改O(logn)棵平衡树, 总时间为$O(\log^2 n)$

例6. 动态连通块

- 给出n*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,mid]和[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logn个区间，因此每次操作时间复杂度为O(nlogn)

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logn个树中区间。建树需要O(n)的时间，询问需要O(logn)时间，换句话说，刚才基于线段树的算法是O(n)-O(logn)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(nlogn)-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(nlogn)

ST算法——查询

- 取k=[$\log_2(j-i+1)$]，那么令A为从i开始的长度为2^k的块，B为到j结束的长度为2^k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

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

- 对于一般的询问RMQ(i,j)，可以分成三部分
  – 若干完整小块的RMQ：RMQ(A', i', j') **O(1)**
  – 小块内部的RMQ：In-RMQ(x, y) **O(??)**
- 关键：求出in-RMQ(x, y)

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

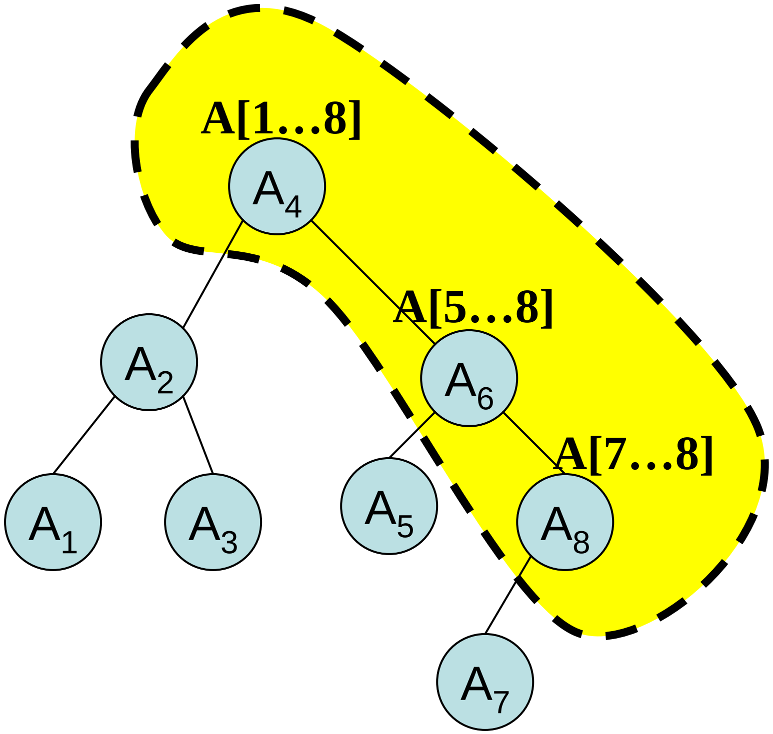
$$O(n+n^{1/2}\log^2 n)=O(n)$$

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

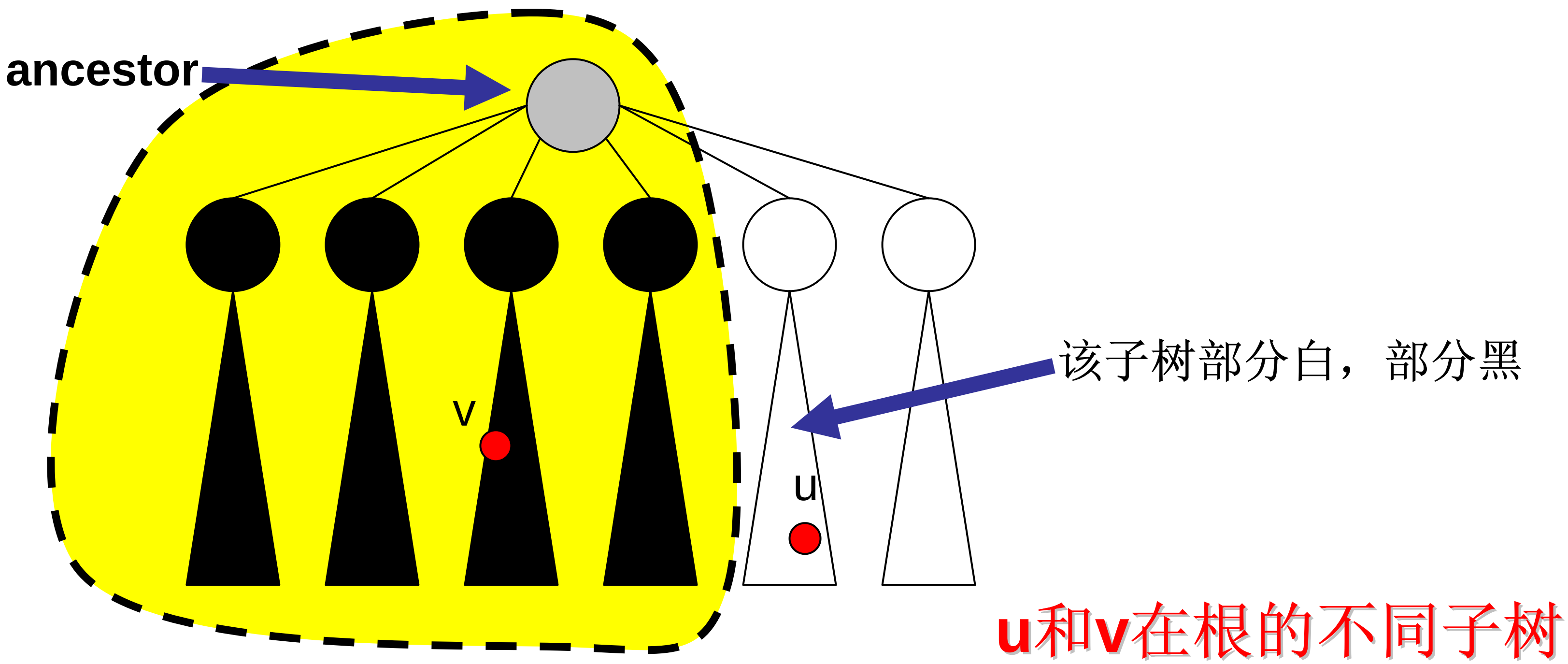
```
void LCA(u){
  makeset(u);
  ancestor[findset(u)] = u;
  v = firstS[u];
  while(v)
     { LCA(v); unionset(u, v); ancestor[findset(u)] = u; v = nextS[v]; }
  color[u] = BLACK;
  v = firstQ[u];
  while(v)
     {if(color[v] == BLACK) answer(u, v, ancestor[findset(v)]); v = nextQ[v]; }
}
```

处理**u**的所有儿子**v**

处理存在询问**LCA(u,v)**的所有结点**v**

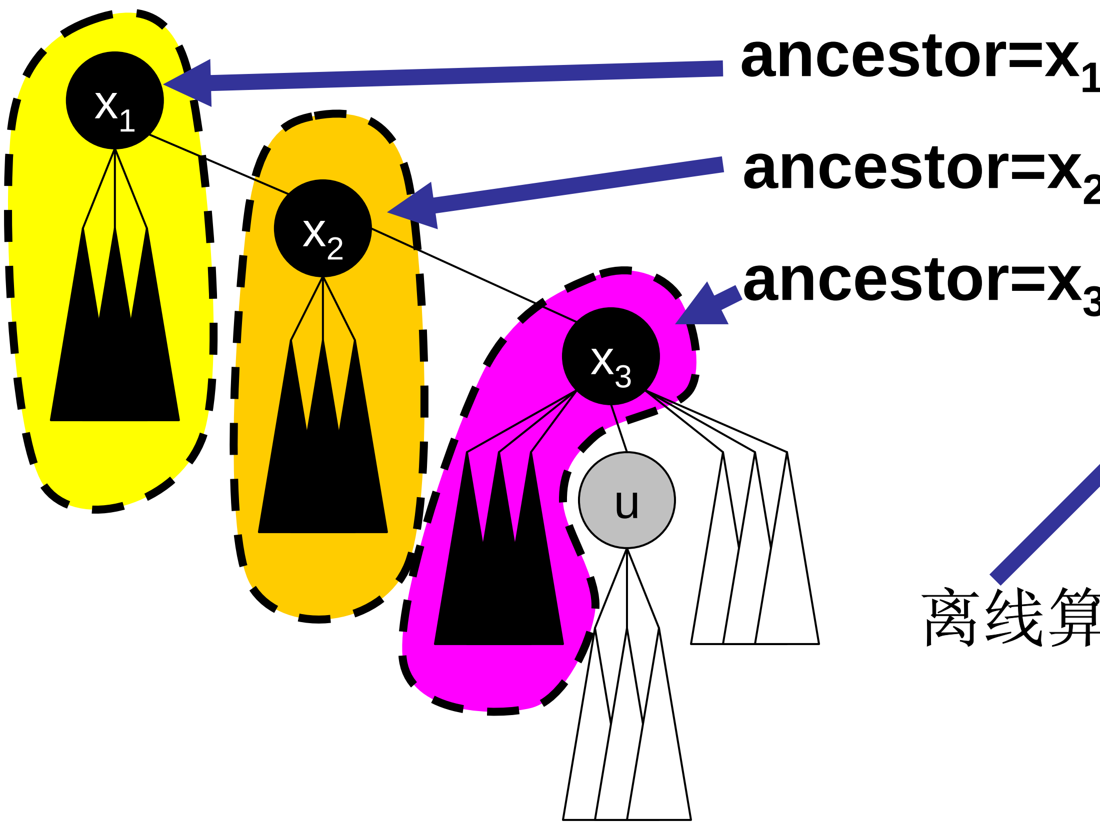
算法动机

- 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ancestor[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$\mathbf{x_2}$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$\mathbf{x_3}$后$\mathbf{x_2}$和$\mathbf{x_3}$所在集合会合并，但这没关系，因为对于未处理的**u**，$\mathbf{x_2}$和$\mathbf{x_3}$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

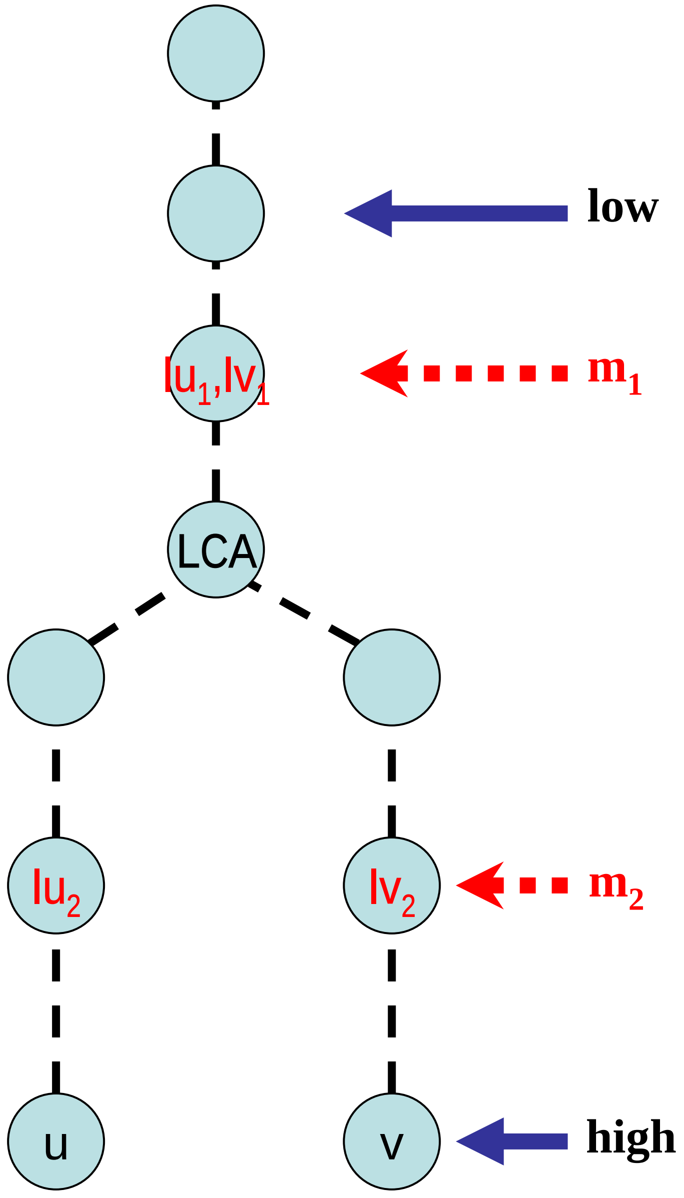
- 若非祖先后代关系，LCA(u,v)是满足下式的最大w

ancestor(u,depth[u]-w)=ancestor(v,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(logn)，因此询问的总时间为$O(\log^2 n)$
- 算法二：令D=min{depth[u], depth[v]}，首先令u=ancestor(u, depth[u]-D), v=ancestor(v, depth[v]-D)，即把u和v上升到同一个高度。显然上升后LCA不变

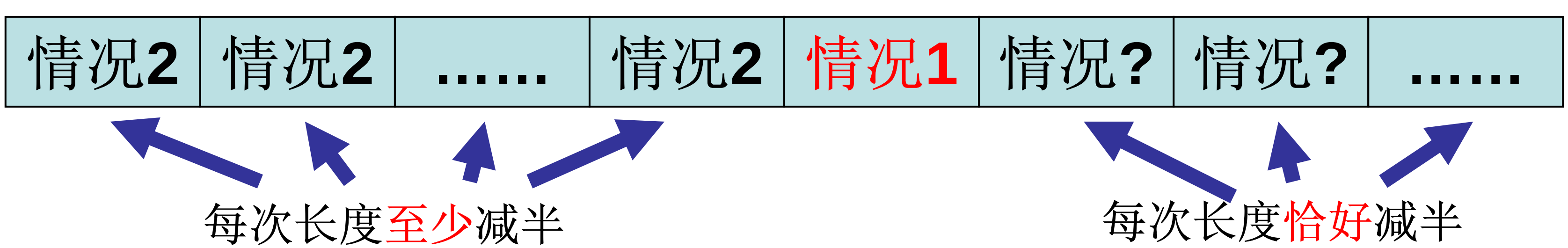
算法二

- 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][lvl], lv=anc[v][lvl]，设m=high-2^{lvl}
  - 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->2^{k-1})
  – 情况2：设high=m，本次至少把区间长度减半



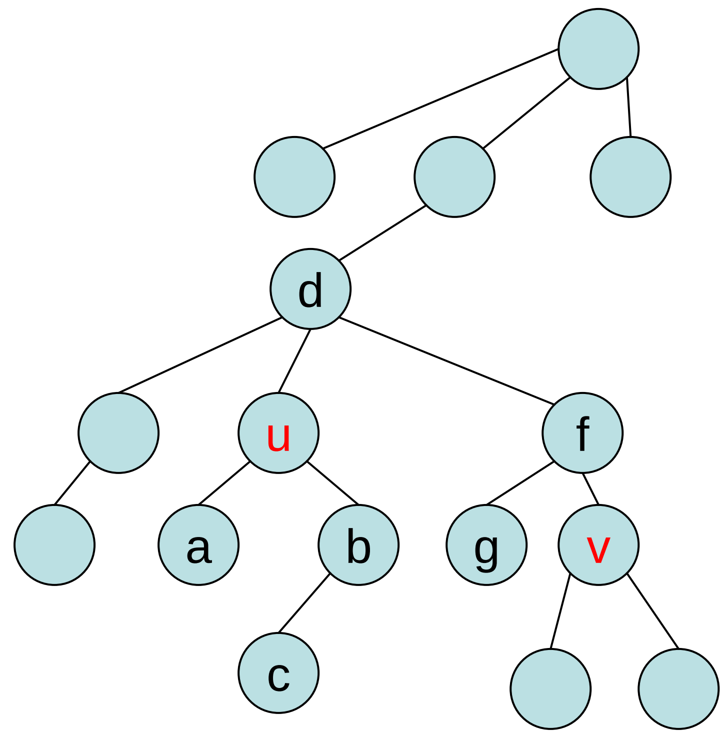
DFS序列

- 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(nlogn)-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…2n-1]表示

问题的转化

- 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], …, R[v]]，令L[i]表示结点E[i]的深度，则

u→v的路径为uaubcbudfgfv



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

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