

# 数据结构



假的



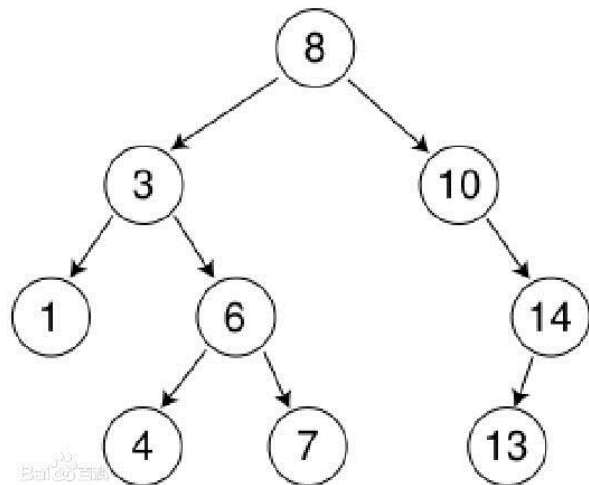
# 目录

- Splay
- 树链剖分
- lct
- 树分治
- 重量平衡树
- 可持久化
- 根号算法
- 仙人掌
- 支配树

# 声明

- 此ppt内出现的题目，不保证我全部做过，也不保证我吹得都对，欢迎打脸
- 此ppt仅为一个蒟蒻的学习总结篇，没有太多的题目
- 但题目大都是比较经典的
- 此ppt内容大部分借鉴抄袭大佬们的博客、论文等，如果你发现有什么东西很眼熟，请不要大声说出来，如果侵犯了你的著作权，请私下找我，谢谢。
- 如果对上述内容已经比较熟悉的大佬可以尽早离场

# Splay



- 前置技能
- 二叉排序树
- 二叉排序树或者是一棵空树，或者是具有下列性质的二叉树：
  - (1) 若左子树不空，则左子树上所有结点的值均小于它的根结点的值；
  - (2) 若右子树不空，则右子树上所有结点的值均大于它的根结点的值；
  - (3) 左、右子树也分别为二叉排序树；
  - (4) 没有键值相等的节点。



# Splay

## ► 简介

- 伸展树 (Splay) 是一种平衡二叉树，即优化后的二叉查找树。伸展树可以自我调整，这就要依靠伸展操作，使得提升效率。

# Splay

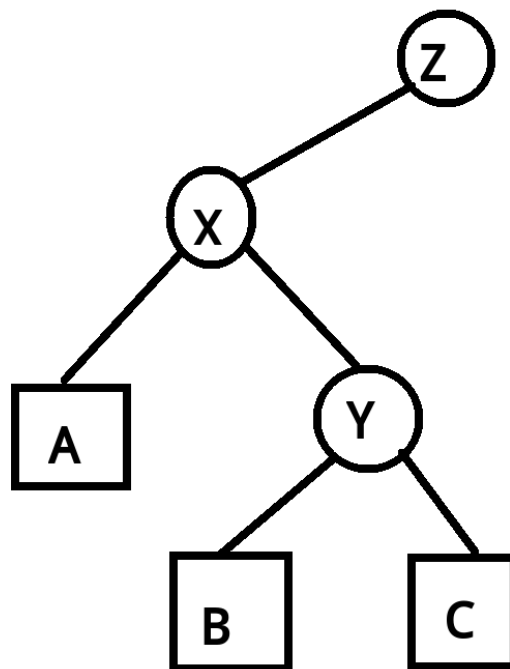
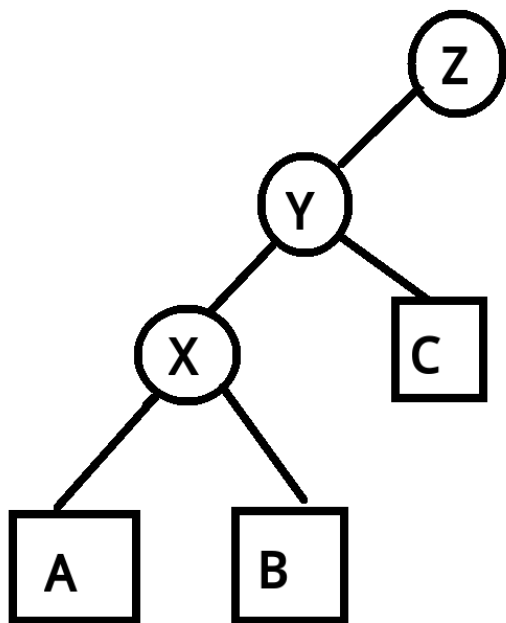
- ▶ 基本操作：
- ▶ 查找
- ▶ 插入
- ▶ 区间翻转
- ▶ 删除
- ▶ 寻找最大值
- ▶ 寻找最小值
- ▶ 求一个刚好比当前的值大的值
- ▶ 求一个刚好比当前的值小的值
- ▶ .....

Splay操作是基础

# Splay

- 旋转(rotate)
- Splay使用旋转保持平衡。所以旋转是最重要的操作，也是最核心的操作。
- Splay旋转后，中序遍历和Splay的合法性不变。
- 旋转操作可以认为就是把当前节点与父亲节点换个位置

# Splay



把x和它的父亲y换位置

容易发现，不管x是左或是右儿子，其过程是相似的。



# Splay

- 总结一下：
- 1.X变到原来Y的位置
- 2.Y变成了 X原来在Y的位置的 相对的那个儿子
- 3.Y的非X的儿子不变
- 4.X的 X原来在Y的位置 那个儿子不变
- （比如：x是y的左儿子，则x的左儿子不变）
- 5.X的 X原来在Y的 相对的 那个儿子 变成了 Y 原来是X的那个儿子
- （比如：x是y的左儿子，则x的右儿子变成y的左儿子）

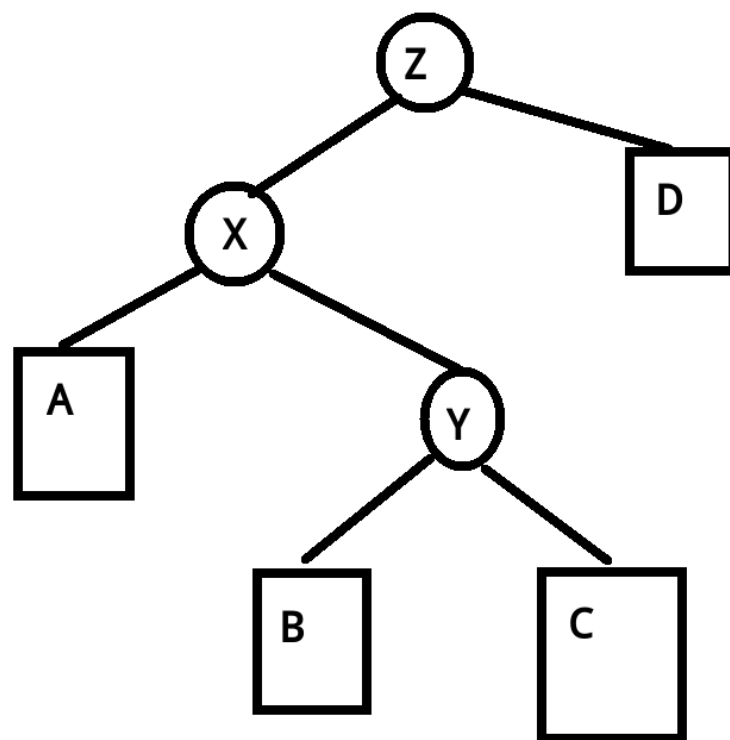
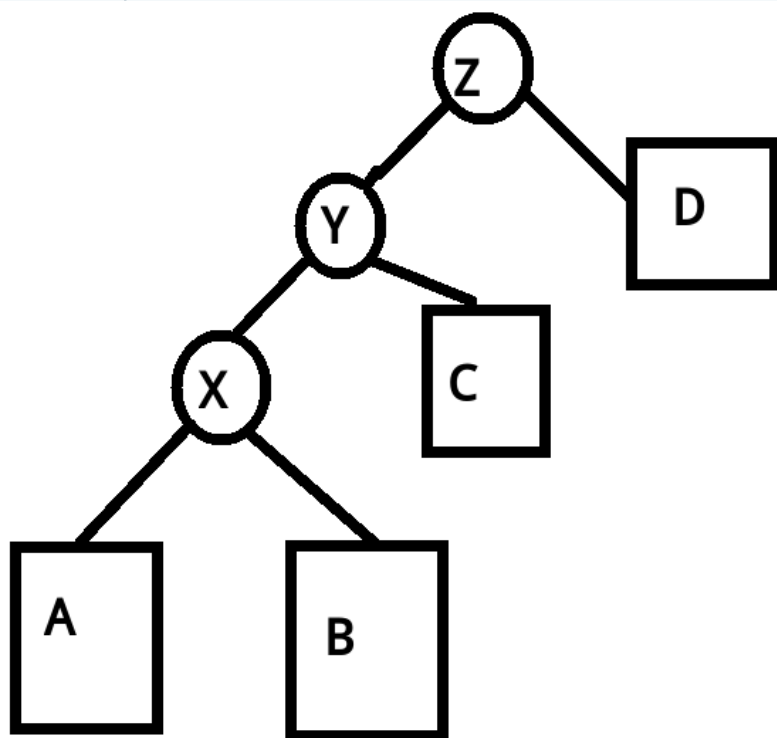


# Splay

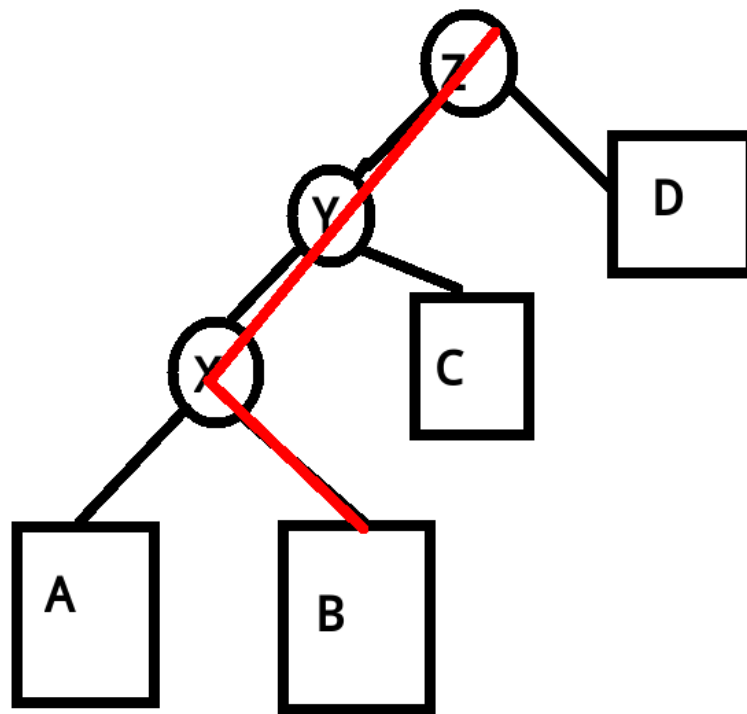
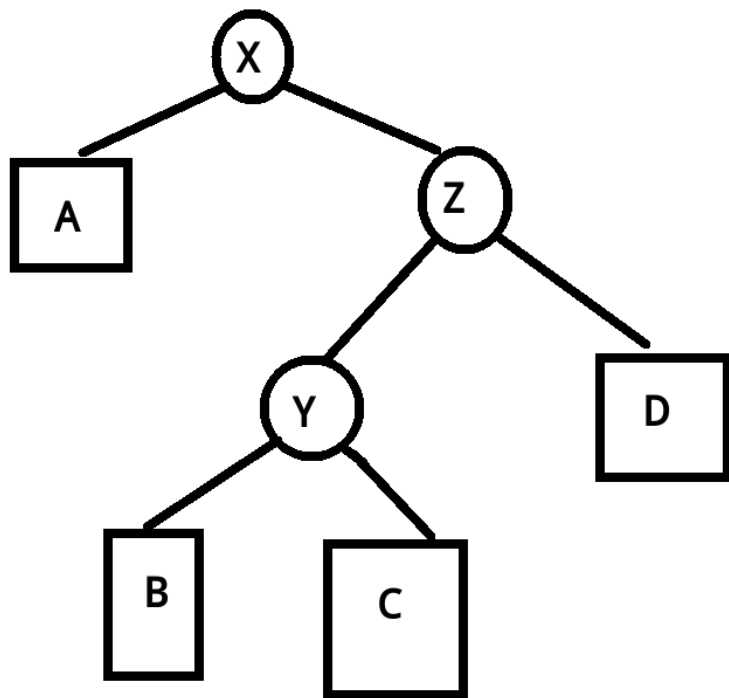
```
➤ void rotate(int x){  
➤   int y=f[x],z=son(x); f[x]=f[y];  
➤   if (f[x]) s[f[x]][son(y)]=x;  
➤   if (s[x][1-z]) f[s[x][1-z]]=y;  
➤   s[y][z]=s[x][1-z]; s[x][1-z]=y; f[y]=x;  
➤   update(y); update(x);  
➤ }
```

# Splay

➡ 单旋



# Splay



有一条链 $X \rightarrow Z \rightarrow Y \rightarrow B$   
如果你只对 $X$ 进行旋转的话,  
有一条链依旧存在,  
如果是这样的话, splay很可能会被卡。

# Splay

## ► 双旋

► 我们可以发现，当前节点与其父亲节点同属一边时，旋转父亲节点并不影响当前节点对其父亲节点的相对位置，所以我们可以先对其父亲进行单旋，再对当前节点单旋，这就是所谓的双旋操作。

► 具体的复杂度分析，可以参看

杨思雨 《伸展树的基本操作与应用》

# Splay

```
void splay(int x,int y){  
    while (f[x]!=y) {  
        if (f[f[x]]!=y) {  
            if (son(x)==son(f[x])) rotate(f[x]); else rotate(x);  
        } rotate(x);  
    }  
    if (!y) root=x;  
}
```

其实主要操作就这些，其余操作都是基于这个基础的

# Splay

- 插入操作
- 先在splay树上寻找，如果找不到，那么直接在当前位置新建节点。
- `void ins(int &x,int y,int z) {`
- `if (!x) {x=++m; siz[m]=1; f[m]=z; return; }`
- `if (y<x) ins(s[x][0],y,x); else`  
`ins(s[x][1],y,x);`
- `update(x);`
- `}`

# Splay

- 删除操作
- 现在就很简单啦
- 首先找到这个数的前驱，把他Splay到根节点
- 然后找到这个数后继，把他旋转到前驱的底下
- 比前驱大的数是后继，在右子树
- 比后继小的且比前驱大的有且仅有当前数
- 在后继的左子树上面，
- 因此直接把当前根节点的右儿子的左儿子删掉就可以啦
- 代码比较显然



# Splay

- ▶ 查找操作
- ▶ 辅助操作，将最大的小于等于的数所在的节点splay到根。
- ▶ 查询k大(kth)
- ▶ 从根节点开始，一路搜索下去。每次判断要走向哪个子树。注意考虑重复权值。
- ▶ 前驱(pre)
- ▶ 将该节点splay到根后返回左子树最右边的节点即可。
- ▶ 后继(succ)
- ▶ 同理，返回右子树最左边的节点即可。
- ▶ 查询rank(rank)
- ▶ 并不需要专门写操作。将该节点find到根后返回左子树的权值数即可。

以上操作都比较显然，这里不再一一叙述

# Splay

- 区间翻转
- 考虑线段树维护区间标记的方法，将其移植到Splay即可。
- 打标记时，将 $l$ 的前驱和 $r$ 的后继分别旋转到根节点和根节点右儿子处，那么左子树即是区间。在其根处打上标记然后在查询和输出中序遍历时下传标记即可。

# Splay

```
➤ void sign(int x){  
➤   bz[x]^=1; swap(s[x][1],s[x][0]);  
➤ }  
➤ void down(int x){  
➤   if (bz[x]) {  
➤     if (s[x][0]) sign(s[x][0]);  
➤     if (s[x][1]) sign(s[x][1]);  
➤     bz[x]=0;  
➤   }  
➤ }
```



# Splay

- 区间标记
- 平衡树像线段树一样，可以打标记。但是有一个不同点，就是平衡树的每个节点都有权值。所以更新标记时和线段树不一样，要考虑自身节点的权值。
- 因为Splay可以直接提取指定区间，所以Splay的区间操作在某些意义上比线段树还好写。
- 注意在改变父子关系时更新



# Splay

- 优化
- 手动实现垃圾回收，删除的时候，把要删除的节点全部加到一个队列里。等到要插入的时候，优先使用队列里的点。

## 题目


- [HNOI2010]弹飞绵羊
- JZOJ 3599 【CQOI2014】 排序机械臂
- BZOJ 1208 【HNOI2004】 宠物收养场
- JZOJ 2808. 【HNOI2012】 永无乡
- JZOJ 2413. 【NOI2005】 维护数列

# 树链剖分

- 链剖分，是指一类对树的边进行轻重划分的操作，这样做的目的是为了减少某些链上的修改、查询等操作的复杂度。
- 目前总共有三类：重链剖分，实链剖分和并不常见的长链剖分



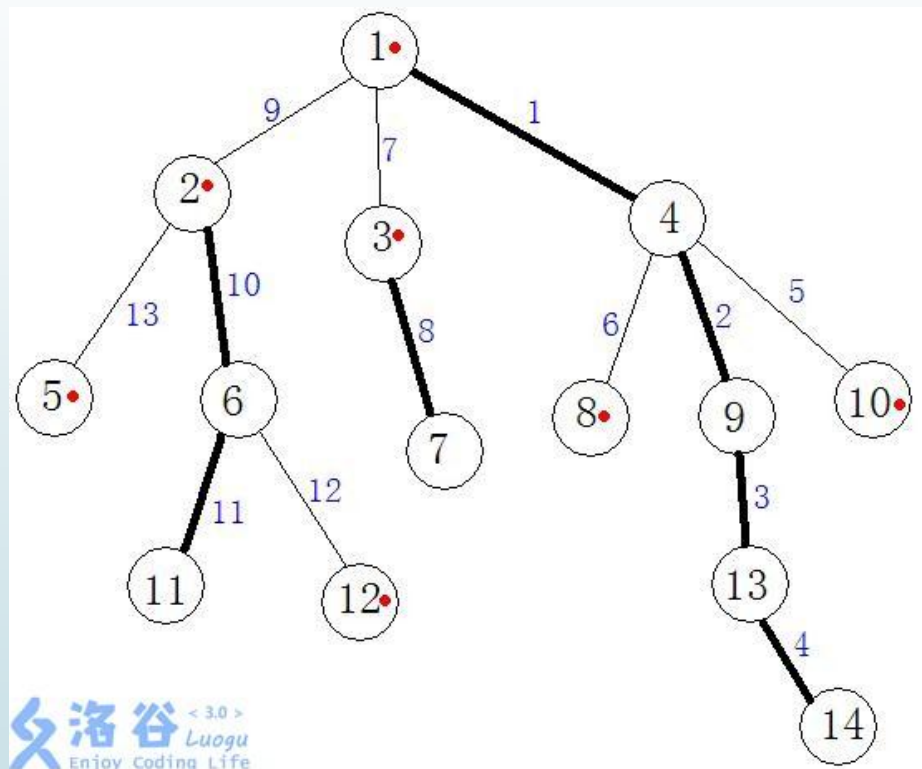
# 实链剖分

- ➡ 剖分方法：
  - ➡ 盲目剖分
  - ➡ 随机剖分
  - ➡ 启发式剖分
- 



# 实链剖分

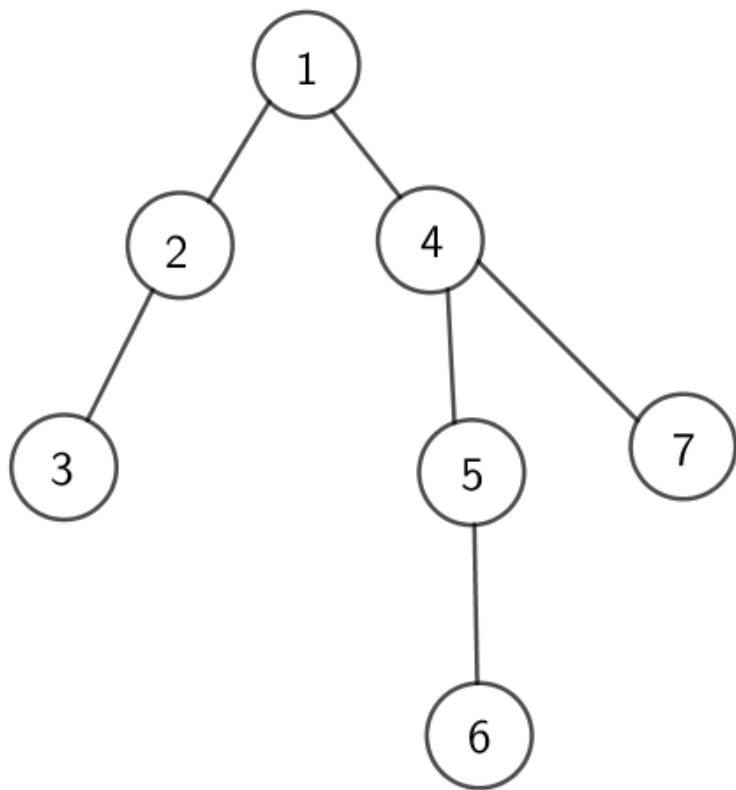
- 轻边、重边、轻儿子、重儿子
- 定义：size(x)为以x为根的子树的节点个数。
- 令y为x的儿子中，size值最大的节点，那么y就是x的重儿子，边(x,y)被称为重边，树中除重边以外的边被称作轻边。



# 实链剖分

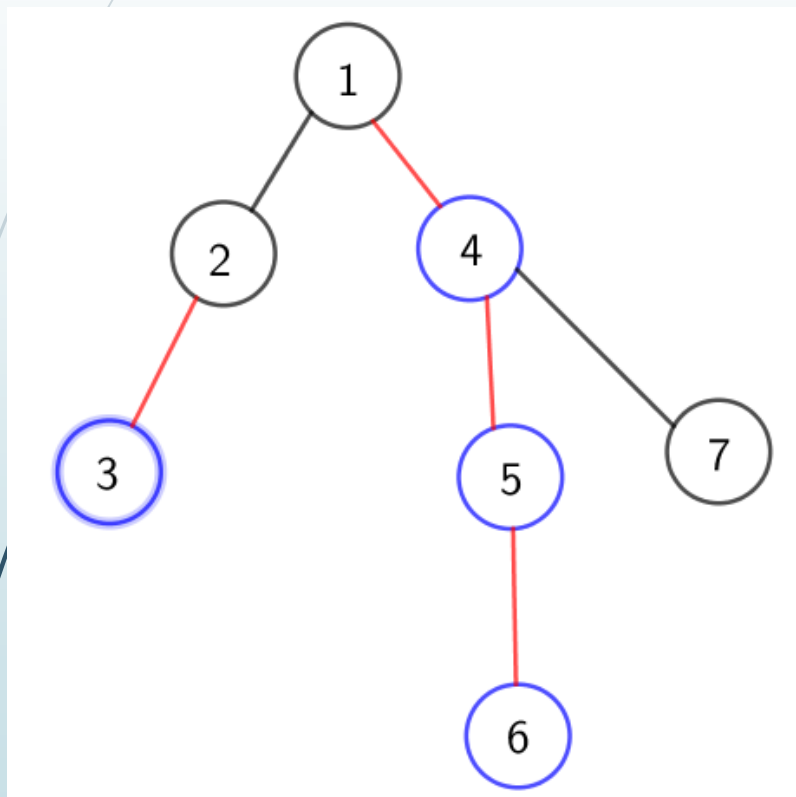
- 对于每一个节点，找出它的重儿子，那么这棵树就自然而然的被拆成了许多重链与许多轻链
- 如何对这些链进行维护？
- 首先，要对这些链进行维护，就要确保每个链上的节点都是连续的，因此我们需要对整棵树进行**重新编号**，然后利用dfs序的思想，用线段树或树状数组等进行维护
- 注意在进行重新编号的时候先访问重链
- 这样可以保证重链内的节点编号连续

# 实链剖分



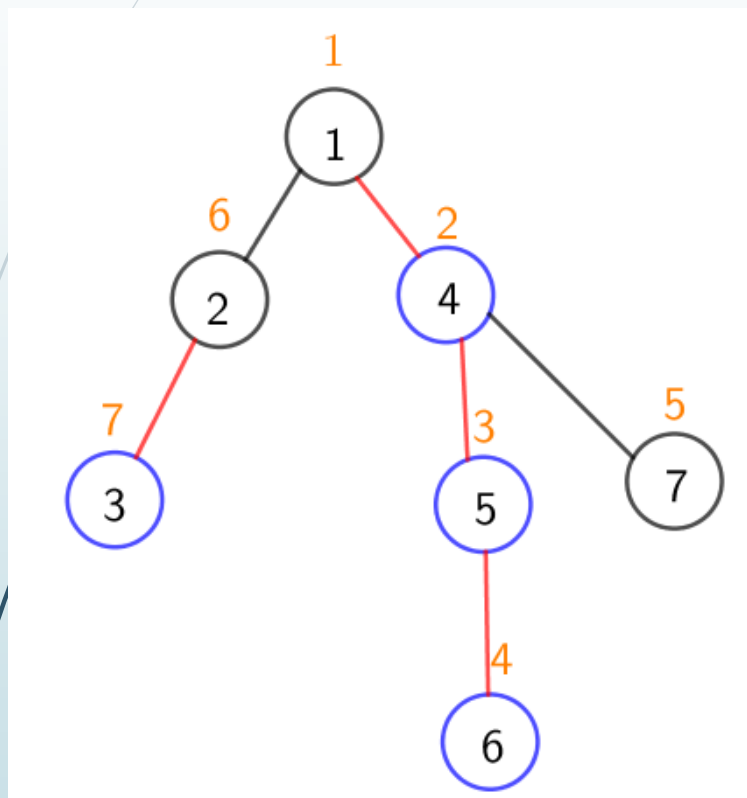
这是一棵基本的树

## 实链剖分



给他标记重儿子，  
蓝色为重儿子，红  
色为重边

# 实链剖分



然后对树进行重新编号

橙色表示的是该节点重新编号后的序号

不难看出重链内的节点编号是连续的

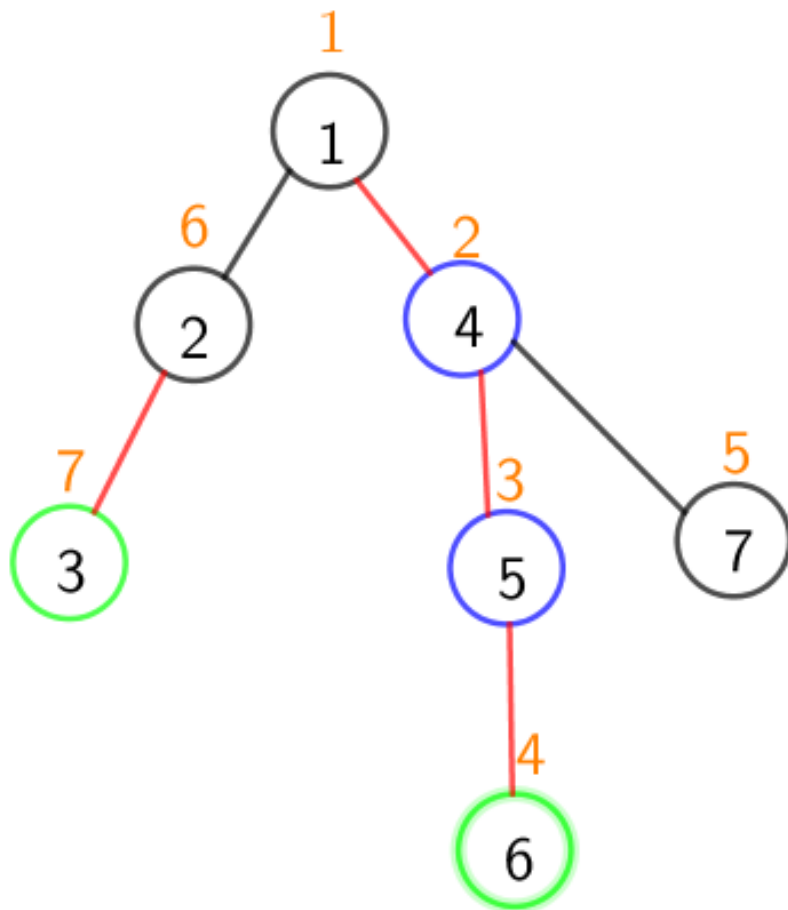
# 实链剖分

- 具体操作：
- 首先要对整棵树dfs一遍，找出每个节点的重儿子，顺便处理出每个节点的深度，以及他们的父亲节点
- 然后再对整棵树dfs一遍，将重边连成重链（对于每个节点，记录top数组，如top[x]表示x所在重链的深度最浅点），通过先遍历重儿子，重新编号（这样会使一条重链上的点编号连续）
- 接着，根据重新编完号的树，把这棵树的上每个点映射到数据结构上
- 最后实现树上的操作，对于两个不在同一重链内的节点，让他们不断地跳，使得他们处于同一重链上

# 实链剖分

- 那么我们应该如何“跳”呢？
- 因为 $x$ 到 $\text{top}[x]$ 中的节点在线段树上是连续的，
- 结合 $\text{deep}$ 数组
- 假设两个节点为 $x, y$
- 我们每次让 $\text{deep}[\text{top}[x]]$ 与 $\text{deep}[\text{top}[y]]$ 中大的(在下面的)往上跳(有点类似于树上倍增)
- 让 $x$ 节点直接跳到 $\text{top}[x]$ ,然后在线段树上更新
- 最后两个节点一定是处于同一条重链的，前面我们提到过重链上的节点都是连续的，直接在线段树上进行一次查询就好

# 实链剖分



举个例子



# 实链剖分

- 因为树中任意两个节点之间的路径中轻边的条数不会超过 $\log_2 n$ ,重路径的数目不会超过 $\log_2 n$
- 时间复杂度
- 由于重路径的数量的上界为 $\log_2 n$ ,
- 线段树中查询/修改的复杂度为 $\log_2 n$
- 那么总的复杂度就是 $(\log_2 n)^2$



# 实链剖分

- 题目
- 最近公共祖先
- [HAOI2015]树上操作
- JZOJ2271.[SDOI2011]染色
- BZOJ3531.[SDOI2014]旅行
- BZOJ3626.[LNOI2014] LCA



# lct

- 同样将某一个儿子的连边划分为实边，而连向其他子树的边划分为虚边。
- 区别在于虚实是可以动态变化的，因此要使用更高级、更灵活的Splay来维护每一条由若干实边连接而成的实链。
- 基于性质更加优秀的实链剖分，LCT(Link-Cut Tree)应运而生。

# lct

- 性质
- 1. 每一个Splay维护的是一条从上到下按在原树中深度严格递增的路径，且中序遍历Splay得到的每个点的深度序列严格递增。（每棵splay都没有相同深度的节点）
- 2. 每个节点包含且仅包含于一个Splay中。
- 3. 边分为实边和虚边，实边包含在Splay中，而虚边总是由一棵Splay指向另一个节点（指向该Splay中中序遍历最靠前的点在原树中的父亲）。那么为了保持树的形状，我们要让到其它儿子的边变为虚边，由对应儿子所属的Splay的根节点的父亲指向该点，而从该点并不能直接访问该儿子（认父不认子）。



# Lct

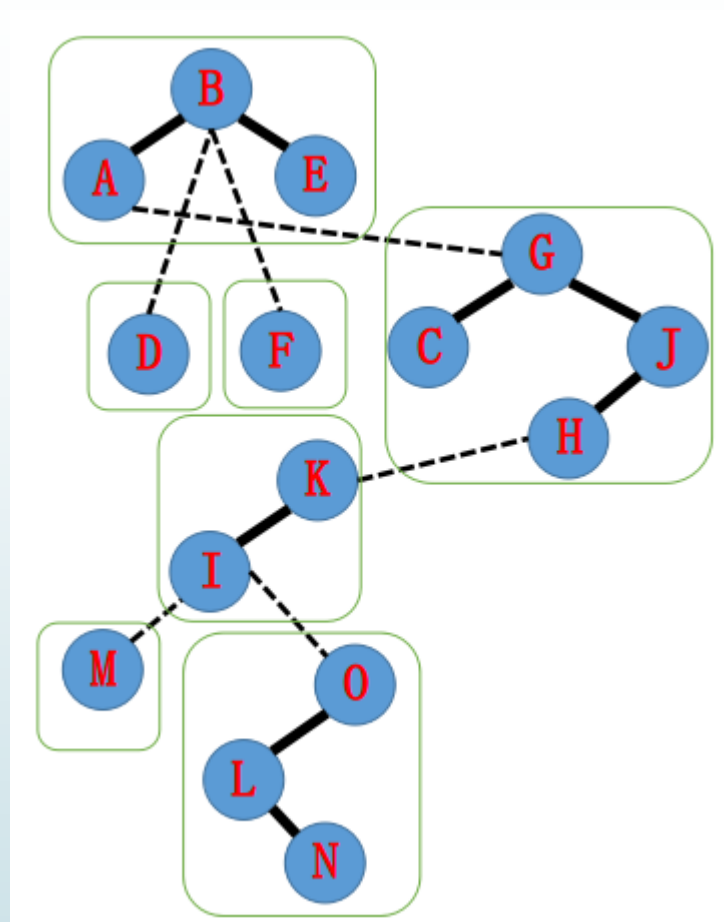
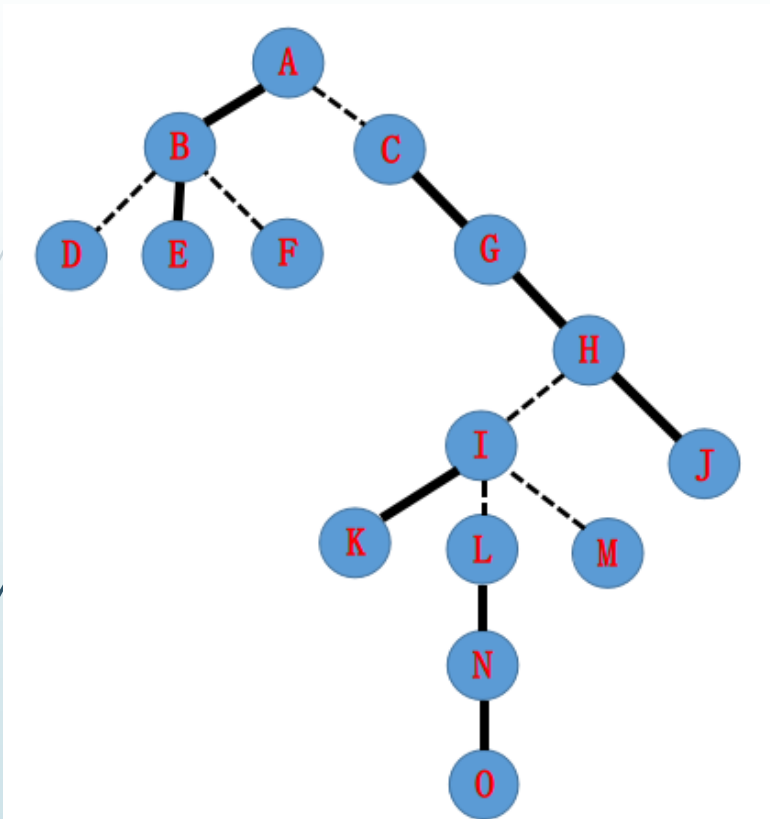
- 常用操作
- 查询、修改链上的信息（最值，总和等）
- 随意指定原树的根（即换根）
- 动态连边、删边
- 合并两棵树、分离一棵树
- 动态维护连通性



# lct

- `access(x)`
- LCT核心操作，也是最难理解的操作。其它所有的操作都是在此基础上完成的。
- `access`即定义为打通根节点到指定节点的实链，使得一条中序遍历以根开始、以指定点结束的Splay出现。

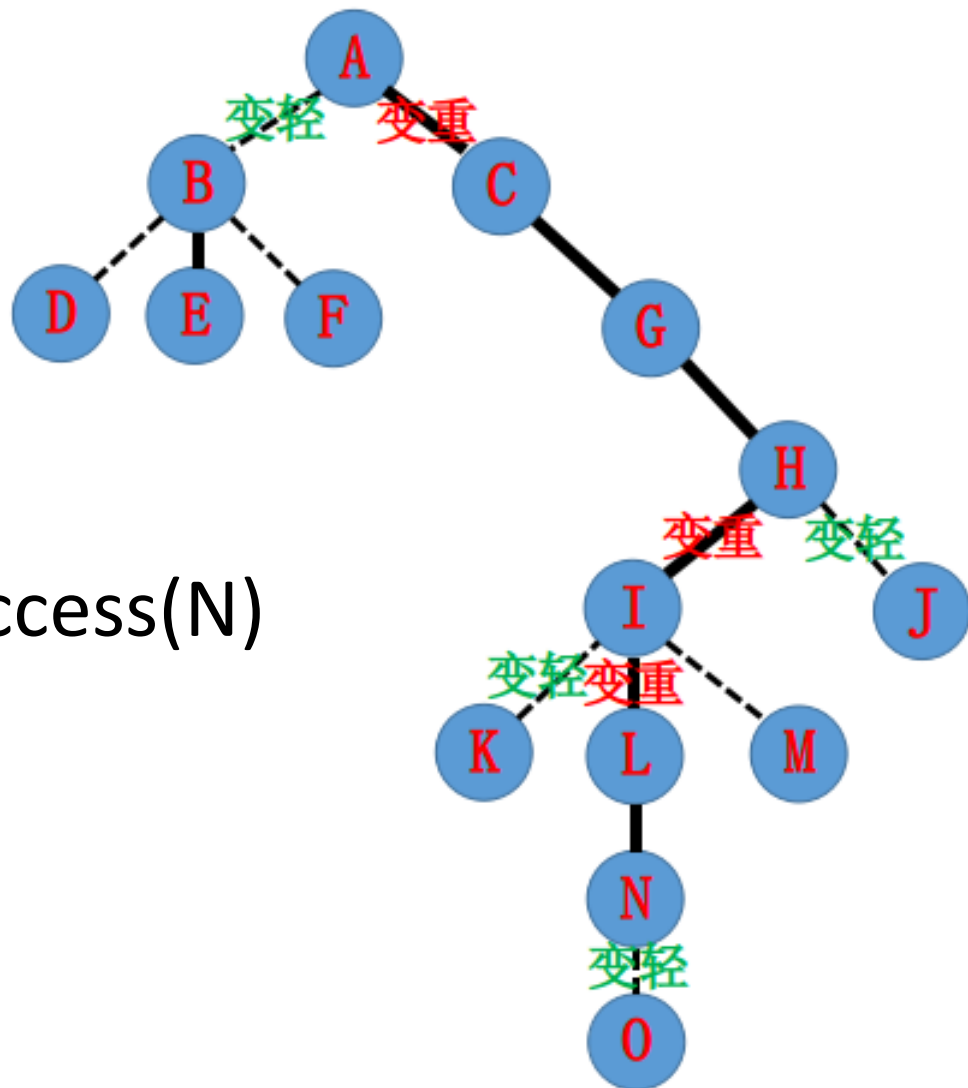
# lct



左边是一棵经过实链剖分后的树，右边是这棵树实际在同一棵splay上的划分

lct

现在我们要access(N)





# lct

- 具体实现
- 首先splay(N)，使之成为当前它所在的splay的根。因为此时它的父亲已经是在上一棵splay中，所以splay它的父亲，因为它的父亲正好比N深度大一，因为它们access后要在同一棵splay中，所以经过splay(N的父亲)后，把N作为它的父亲在splay中的右儿子。
- ```
void access(int x){
```
- ```
    for(int y=0;x;x=f[y=x])
```
- ```
        splay(x),s[x][1]=y,update(x);
```
- ```
}
```

# lct

- 换根
- 假设我们要使x变为根。
- 先`access(x)`，使x与根在同一棵splay中，然后`splay(x)`，那么由于x是在当前splay中深度最大的节点，所以x的右节点为空，我们翻转x的左右节点，那么x就成为了根。
- ```
void makeroot(int x){
```
- ```
    access(x); splay(x); flag(x);
```
- ```
}
```

# lct

- 找当前节点所在子树的根
- 假设当前节点为x
- 先access(x)，使其与根在同一棵splay中，然后splay(x)，使其成为splay的根，然后从x一直向左儿子走，直到没有左儿子为止。
- void access(int x){
- for(int y=0;x;x=f[y=x])
- splay(x),s[x][1]=y,update(x);
- }

# lct

- 连边
- 假设我们连接(x,y)。
- 首先，先使x变为它所在树的根，然后判断y所在子树的根是否使x，如若不是，则使x的父亲为y。
- `void link(int x,int y){`
- `makeroot(x);`
- `if (findroot(y)!=x) f[x]=y;`
- `}`

# lct

- 删边
- 假设我们要删除(x,y)这条边。
- 首先，使x变成它当前所在子树的根，然后判断y是否与x直接连边，如果是，则使y的父亲，x对应的儿子设为0。
- 注意这里有三个条件
- `void cut(int x,int y){`
- `makeroot(x);`
- `if (findroot(y)==x && f[y]==x && s[y][0]==0) {`
- `f[y]=s[x][son(y)]=0; update(x);`
- `}`
- `}`

# lct

- 小结
- 大致的基本操作就是以上了，还有一些比较细的细节这里就不再多赘述了（比如update（更新数值）、down（下传标记）都和splay差不多）。
- 常见题型
- 维护链信息（LCT上的平衡树操作）
- 动态维护连通性&双联通分量（可以说是并查集的升级，因为并查集只能连不能断）
- 维护边权（常用于维护生成树）
- 维护子树信息
- 维护树上染色联通块



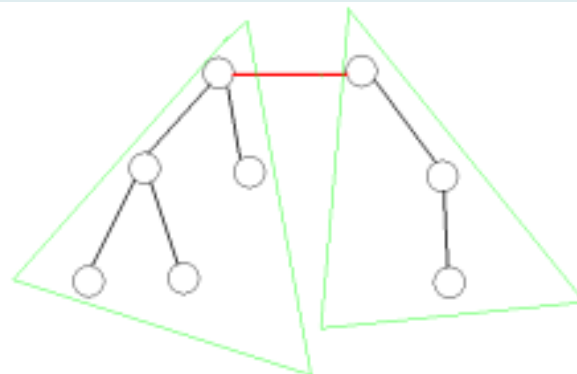
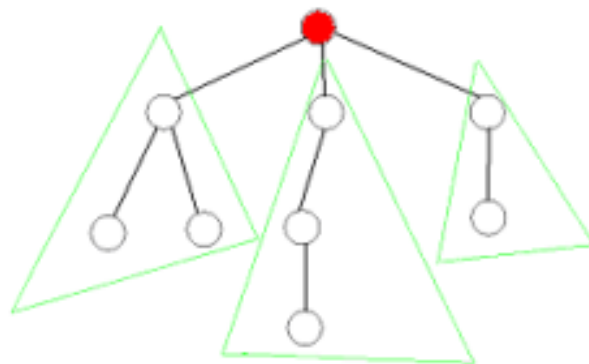
# lct



- 例题
- [HNOI2010]弹飞绵羊
- [NOI2014]魔法森林
- [SDOI2008]洞穴勘测

# 树分治

- 基于点的分治：
- 首先选取一个点将无根树转为有根树，再递归处理每一颗以根结点的儿子为根的子树
- 基于边的分治：
- 在树中选取一条边，将原树分成两棵不相交的树，递归处理。





# 树分治

- 点分治
- 因为每次都是先找到重心，然后再分治，所以点分治一共最多有 $\log n$ 层，每层遍历约为 $n$ ，所以时间复杂度为 $n \log n$ 。
- 边分治
- 可以证明在最坏情况下，边分治的递归层数会到 $n$ ，所以这个算法效率比较低，故这里只考虑使用点分治。



# 树分治

- 点分治
- 算法步骤
  - 第一步，求出当前树的重心。
  - 第二步，计算当前树。
  - 第三步，对当前树重心的相邻的，未成为任一棵树重心的节点执行第一步。

# 树分治

- ➡ JZOJ 1166. 树中点对距离
- ➡ 给出一棵带边权的树，问有多少对点的距离  $\leq \text{Len}$
- ➡  $N$ ,  $\text{Len}$  ( $2 \leq n \leq 10000, \text{len} \leq \text{maxlongint}$ )

因为每条路径要么过根结点，要么在一棵子树中，这启发了我们可以使用分治算法

# 树分治

- 对于这道题，首先，我们应该找到树的重心，由重心开始，递归分治求解， $O(n)$ 找到每个点到当前根的距离，然后计算。具体计算网上好像是简单容斥，但我自己写的时候是用两个桶记录，一个记录它到这个点距离为 $x$ 的有多少个，另一个记录当前距离为 $x$ 的个数，然后先加入`ans`再把第二个桶内的值加进去第一个桶里，并清空第二个桶。
- 其次，删除当前节点（其实标记即可）然后找到它的每一棵子树的重心，然后以重心为根继续上一个过程。
- 代码就不放了，过程还是很显然的。



# 树分治

- 例题
- JZOJ3234. 阴阳
- [SPOJ1825]Free tour II
- [WC2010]重建计划

# 重量平衡树

一般的平衡树依赖于一个旋转操作，旋转操作在统计一些可以快速合并的信息的时候是没有问题的，但是对于更加复杂的信息，就会遇到复杂度的问题。

考虑一个简单的例子，我们想在一个平衡树的每个节点上维护一个集合，存储子树内部所有的数，此时一次旋转操作的代价可能会达到 $O(n)$ ，传统的旋转平衡树就无法发挥作用了。

而重量平衡树则不同，他每次操作影响的最大子树的大小是最坏或者均摊或者期望的 $O(\log n)$ 。这样即使按照上面所说的方式维护，复杂度也是可以接受的。

有一些平衡树(或者能实现传统平衡树操作的数据结构)并不依赖于旋转机制，因此也就不会受到旋转机制弊端的影响。

本文选取在OI中有一定实际价值的两者加以介绍。

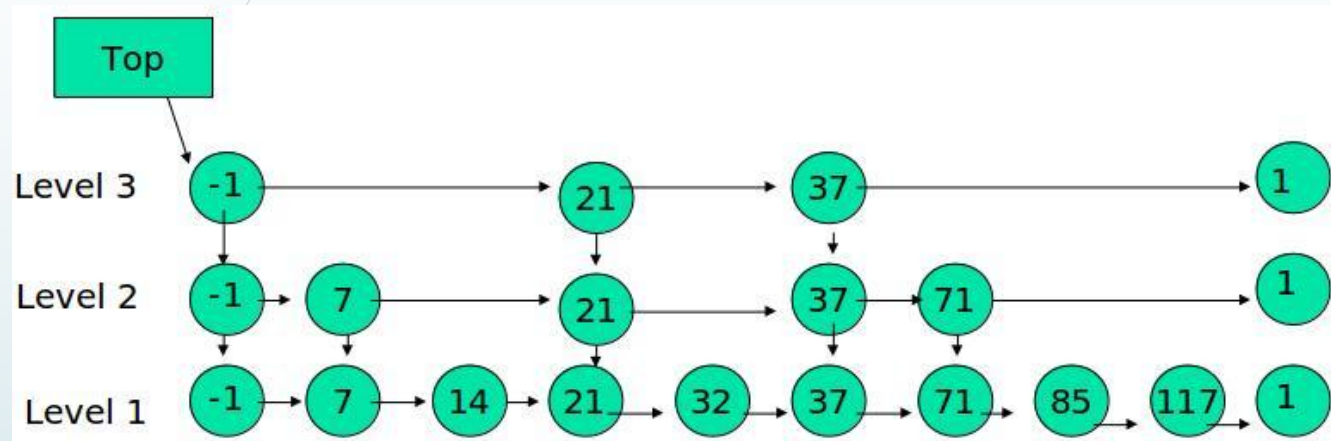
以上皆来自陈立杰论文



# 重量平衡树

- Treap
- Skip-List(跳表)
- Scapegoat Tree(替罪羊树)

# 重量平衡树



## 跳表

简单介绍一下，就是一个几层的链表，查找一层层往下找，对于加入一个数，就随机一个值，表示这个数存在于几层，然后在每一层都插入这个数，随机类似丢硬币，为1表示加入，为0停止。



# 重量平衡树

## ➡ 替罪羊树

替罪羊树依赖于一种暴力重构的操作。

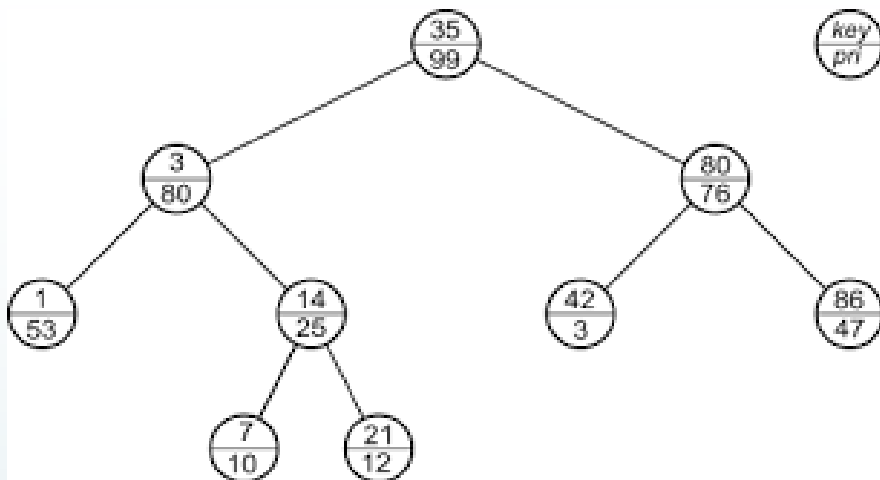
我们定义一个平衡因子 $a$ ，对替罪羊树的每个节点 $t$ ，我们都需要满足 $\max(size_l, size_r) \leq a \cdot size_t$  ( $l, r$ 分别表示左右子树)。

他的实现方式也非常的简洁明了，每次我们进行操作之后，找出往上最高的不满足平衡性质的节点，将它暴力重构为完全二叉树。

# 重量平衡树

- 替罪羊树
- 建树：递归类似线段树建树即可
- 重构：拍成链然后重建子树，之后还接回去
- 删除：用被删除节点的左子树的最后一个节点或者右子树的第一个节点来顶替被删除节点的位置
- 查询 $x$ 数的排名
- 查询排名为 $x$ 的数
- 求 $x$ 的前驱和后继

# 重量平衡树



- Treap
- key: 满足二叉搜索树性质，即中序遍历按照 key 值有序。
- val: 满足堆性质，即对于任何一颗以  $x$  为根的子树， $x$  的 val 值为该子树的最值，方便后文叙述，定义为最小值。为了满足期望，val 值是一个随机的权值，用来保证树高期望为  $\log n$ 。剩下的 key 值则是用来维护我们想要维护的一个权值，此为一个二叉搜索树的基本要素。
- Treap分为有旋和无旋两种

# 重量平衡树

- 有旋treap
- 添加节点：先按照普通二叉搜索树添加节点，再按照val满足堆的性质旋转。
- 旋转：与splay单旋相似（但只记子不记父，所以旋转时要注意是左旋还是右旋）
- 删除：将结点下旋，直到该节点不是满孩子的情況，节点赋0，将儿子结点顶上。
- 但是我们发现，有旋的treap除了代码简单，并没有太多的可取之处。

# 重量平衡树

- ➡ 无旋treap
- ➡ 简介：非旋转treap是一个很有实用性的平衡树，核心操作只有split（分离）和merge（合并）

支持：

1. 插入
2. 删除

3. 查询x的排名

4. 查询排名为x的数

5. 求x的前驱

6. 求x的后继

7. 可持久化

# Treap

- Split
- 一般有两种，一种是按子树大小，一种是按节点权值。这里只讲按子树大小的，因为按权值分类似。实际上，我们把一棵树分为了大小为 $k$ 的和大小为 $n-k$ 的。
- 其实这个分离的操作也可以理解为将一棵树先剖开,然后再按照一定的顺序连接起来,也就是将从 $x$ 节点一直到最坐下或是最右下剖出来,然后再继续处理剖出来链的剩余部分.

# Treap

```
pr split(int x,int k){
    pr y; y.l=y.r=0;
    if (!x) return y;
    if (k<=a[a[x].l].siz) {
        y=split(a[x].l,k);
        a[x].l=y.r; y.r=x; update(x);
    } else
    {
        y=split(a[x].r,k-a[a[x].l].siz-1);
        a[x].r=y.l; y.l=x; update(x);
    }
    return y;
}
```

# Treap

- **merge**
- 首先**merge**操作是有前提条件的,要求是必须第一颗树权值最大的节点要大于第二棵树权值最小的节点。
- 因为有了上面那个限制条件,所以右边的子树只要是插在左边的这颗树的右儿子上就可以维护它的中序遍历,那么我们就只需要考虑如何维护它平衡树的性质。
- 这里我们就需要通过玄学的随机值来维护这个树的性质了.我们在合并两颗树时,因为左边的权值是一定小于右边的,所以左边的那棵树一定是一个根和它的左子树的形式合并的,而右边的那棵树就是以根和右子树的形式合并的,那么如果这次选择的是将左边的树合并上来的话,那么下一次合并过来的位置一定是在这个节点位置的右儿子位置。
- 你可以把这个过程理解为在第一个**Treap**的左子树上插入第二个树,也可以理解为在第二个树的左子树上插入第一棵树。因为第一棵树都满足小于第二个树,所以就变成了比较随机值来确定树的形态。



# Treap

- (以上那段话是我copy的)
- 实际上我认为我们可以理解为，两棵值已经有大小之分的树合并，显然，我们只需要维护其关于堆的性质就可以了。

```
int merge(int x,int y){
    if (!x || !y) return (x^y);
    if (a[x].k<a[y].k) {
        int z=merge(a[x].r,y);
        a[x].r=z; update(x); return x;
    } else
    {
        int z=merge(x,a[y].l);
        a[y].l=z; update(y); return y;
    }
}
```

# Treap

- 其余操作的基础都是以上操作
- 删除：删除数 $x$ ，先找到 $x$ 所在的位置（平衡树，直接找即可），然后把 $x$ 的前面分离出来，然后把 $x$ 的后面分离出来，然后合并 $x$ 的前后， $x$ 就被删除了。（这里仅仅删除了值为 $x$ 的一个数）
- 插入：类似，先找到小于等于 $x$ 中最大的数的排名，记为 $y$ ，然后就把排名小于等于 $y$ 的分离出来，新建节点，合并原来的值较小的树与新节点，然后再合并新的树与原来值较大的树。
- 其余操作与普通平衡树无较大差别，这里不再赘述。

# Treap

```
void insert(int x){
    int k=rank(x);
    pr y=split(root,k);
    a[++mm].v=x; a[mm].k=rand(); a[mm].siz=1;
    root=merge(merge(y.l,mm),y.r);
}

void del(int x){
    int k=rank(x);
    pr y=split(root,k-1); pr z=split(y.r,1);
    root=merge(y.l,z.r);
}
```

# Treap

- 小结：无旋Treap大致就是这样的了。它比splay好写但常数可能稍大一些，而且支持的操作也没有那么多，但是它有一点splay做不到的，就是它可以可持久化，具体内容我会在下一节可持久化中阐述。
- 补充：无旋treap是支持区间操作的，在维护数列时，按照数列前后为顺序，你只需要把要操作的区间分离出来再操作即可。
- 题目
- BZOJ3224: Tyvj 1728 普通平衡树
- (好像没有再多的题目了……)
- 你可以尝试用无旋Treap做之前splay的题

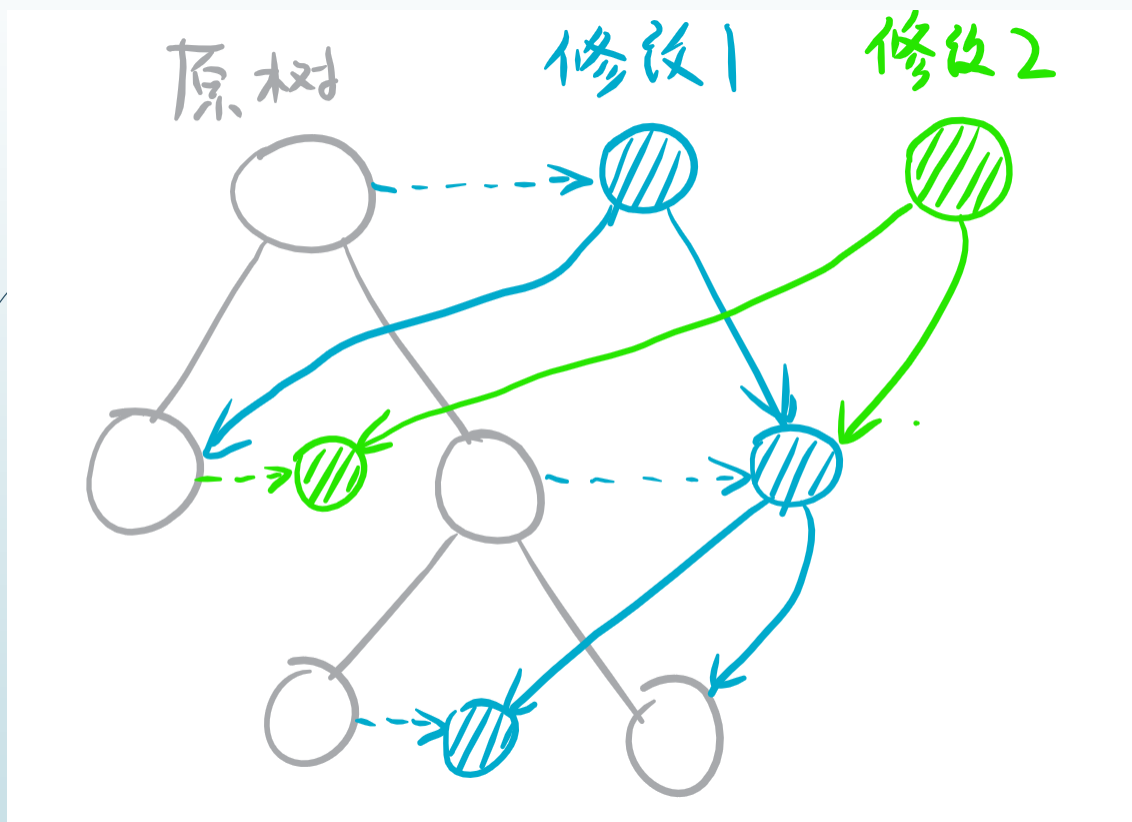
# 可持久化

- 简介
- 持久化(Persistent)数据结构又叫不可变(Immutable)数据结构，顾名思义，这类数据结构的内容是不可变的。也就是说，对于这类数据结构的修改操作，都会返回一个新的副本，而原来的数据结构保存的内容不会有任何改变。这样的数据结构是有意义的，比方说我们现在所编写的程序，都可以看作是一个状态机，也就是说在程序运行的过程的每一个时刻，程序本身可以被看作存在一个状态(State)，我们的语句作用在当前状态上，从而不断地产生出进一步的状态，由此循环往复。
- 可持久化数据结构，就是保存这个数据结构的所有历史版本，同时用它们之间的共用数据来减少时间和空间的损害。

# 可持久化线段树

- 通常遇到的线段树都是构建之后结构不变化的，所以在修改关键值时，只有节点内的值受到影响，而树本身的结构不发生变化（比如左右子节点所表示的区间）。这为线段树进行可持久化提供了便利。我们每次修改的时候不直接改动原来节点的值，而是创建一系列新的节点。如果整棵树复制的话不仅非常耗费时间，而且占用空间太大。在线段树的单次修改中，实际上受到影响的节点是有限的，原来的节点可以得到重复利用。

# 可持久化线段树



# 可持久化线段树

- 可持久化线段树，又名函数式线段树，主席树。
- 所谓主席树呢，就是对原来的数列 $[1 \dots n]$ 的每一个前缀 $[1..i]$  ( $1 \leq i \leq n$ ) 建立一棵线段树，每次增加一个节点 $i$ ，新的线段树 $[1 \dots i]$ 对比之前的 $[1 \dots i-1]$ 仅仅多了一个节点，包含此节点的在线段树上的区间最多为 $\log n$ 个，所以实际上我们只需要修改这 $\log n$ 个区间即可，其余的可以连向上一棵树代表的区间。



# 可持久化线段树

- 对于区间修改，我们可以通过打标记来处理，当前节点如果被全部覆盖，那就建一个节点修改一下打上标记就好了；如果当前节点没有被全部覆盖，那就把他的子节点里被全部覆盖的那个新建节点修改打标记。
- 代码比较简单，相信同学们都会了。

# 可持久化线段树

- ➡ 感受一道经典例题吧
- ➡ 在线求区间 $[l,r]$ 之间的第 $k$ 大的值。

显然我们可以发现，这道题我们只需要建一棵主席树（本质权值线段树）然后再以 $r$ 为根与以 $l-1$ 为根的线段树上做差，二分即可。

# 可持久化线段树

- 还是上一道题，如果添加了修改操作呢？
- 对于动态查询的主席树，如果修改一个点的值，我们肯定不能修改所有包含这个点的线段树，时间复杂度无法承受。
- 那么我们就可以套上树状数组，个人感觉这里比较难懂。
- 树状数组的每个节点都是一颗线段树，但这棵线段树不再保存每个前缀的信息了，而是由树状数组的sum函数计算出这个前缀的信息，那么显而易见这棵线段树保存的是辅助数组S的值，即 $S = A[i - \text{lowbit} + 1] + \dots + A[i]$ ，其中 $A[i]$ 表示值为i的元素出现的次数。

# 可持久化线段树

- 小结：
- 其实我觉得，可持久化线段树可以简单的被认为是利用了使用历史版本和动态开点思想的线段树，每一次修改，最多只涉及到 $\log n$ 个线段树上的节点，从而保证时间与空间复杂度。
- 题目：
- ZOJ 2112 Dynamic Rankings
- JZOJ 5295 Create

# 可持久化Treap

- 因为 FHQ Treap 不需要旋转，因此可以支持可持久化，与其他可持久化数据结构一样，我们使用一个 root 数组来表示不同版本对应的根
- FHQ Treap 的持久化只需要在 split 和 merge 过程中，基于原树复制一个一样的节点，再进行操作

# 可持久化Treap

➡ 例子：split

```
pr split(int x,int k){
    pr y; y.l=y.r=0;
    if (!x) return y;
    int xx=++m; a[xx]=a[x];
    if (a[a[x].l].siz>=k) {
        y=split(a[x].l,k);
        a[xx].l=y.r; update(xx); y.r=xx; return y;
    } else
    {
        y=split(a[x].r,k-a[a[x].l].siz-1);
        a[xx].r=y.l; update(xx); y.l=xx; return y;
    }
}
```

# 可持久化Treap

- 小结：
- 显然，它的过程与普通Treap差不多。与可持久线段树不同，可持久Treap没有动态静态之分，支持查询、修改、翻转等操作。
- 技巧：
- 在某些题目中，可能会有复制操作，这时，固定的随机值可能会使Treap丧失随机性，所以我们可以实时更改随机值，使用 $\text{rand}() \% (\text{size}[a] + \text{size}[b]) < \text{size}[a]$ 作为两树合并时的随机值，使树高的尽量放在上面。
- 题目：
- UOJ3658.文本编辑器

# 根号算法

简介：

根号算法，就是指复杂度带根号的算法。——wzd

分块

块状链表

分块套分块

莫队算法

带修改莫队

多种算法的平衡



# 分块

- 最常见的根号算法是分块。
- 这种算法会将序列进行分块，通常设置一个阈值 $B$ （ $B$ 一般设置为根号 $n$ ），每一块有至多 $B$ 个元素。在序列分块问题上，一般会严格要求每个块都要有 $B$ 个元素（最后一个块除外）
- 分块算法可以维护一些线段树维护不了的东西，例如单调队列等，线段树能维护的东西必须能够进行信息合并，而分块则不需要。不过，和线段树一样，分块需要支持类似标记合并的东西。
- （相信大家都会了）

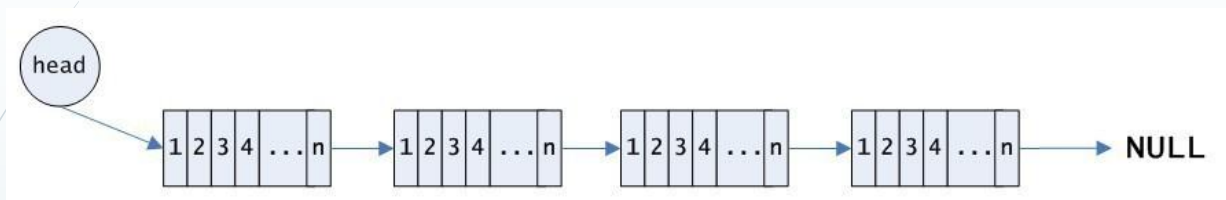
# 分块

- 算法步骤
- 我们首先把求出块的大小  $\text{block} = \sqrt{n}$
- 然后求出一共有多少个块  $\text{num} = n \div \text{block}$
- 再求出每一个块的左右端点
- $l[i] = (i-1) \times \text{block} + 1$
- $r[i] = i \times \text{block}$
- 最后求一下每个节点属于哪个块  
 $\text{belong}[i] = (i-1) \div \text{block} + 1$
- （有可能还要维护一下别的值） 这个步骤一般所有分块都有。

# 分块

- 小结：
- 相信以上的题目大家都切了吧。
- 题目：
- BZOJ 2002 弹飞绵羊
- BZOJ 2821 作诗
- 分块内容来源：
- [「分块」数列分块入门1 – 9 by hzwer](#)

# 块状链表



➡ 块状链表是结合了数组和链表的优缺点，块状链表本身是一个链表，但是链表储存的并不是一般的数据，而是由这些数据组成的顺序表。每一个块状链表的节点，也就是顺序表，可以被叫做一个块。块状链表通过使用可变的顺序表的长度和特殊的插入、删除方式，可以在达到的复杂度。块状链表另一个特点是相对于普通链表来说节省内存，因为不用保存指向每一个数据节点的指针。

# 块状链表

- 小结：
- 没有找到什么有趣的题，基本上我感觉普通分块很像，就是用链表维护了插入和删除而已。
- 题目：
- [bzoj3065]带插入区间k小值—(据说这题故意卡块状链表)

# 莫队算法

- 一个优雅的暴力——Alan\_Cty
- 简介：
- 莫队算法是由莫涛提出的算法，可以解决一类离线区间询问问题，适用性极为广泛。同时将其加以扩展，便能轻松处理树上路径询问以及支持修改操作。
- 莫队算法需要满足已知 $[l,r]$ 可以得到 $[l+1,r]$  $[l-1,r]$  $[l,r+1]$  $[l,r-1]$ 。

# 普通莫队

- 时间优化技巧：
- 奇偶化排序
- 即对于属于奇数块的询问， $r$  按从小到大排序，对于属于偶数块的排序， $r$  从大到小排序，这样我们的  $r$  指针在处理完这个奇数块的问题后，将在返回的途中处理偶数块的问题，再向  $n$  移动处理下一个奇数块的问题，优化了  $r$  指针的移动次数，一般情况下，这种优化能让程序快 30% 左右

# 普通莫队

- 小结：
- 其实莫队算法就是通过减少左右指针的移动步数来优化时间。
- 题目：
- BZOJ 2038 小Z的袜子
- JZOJ 3568 小纪的作业题
- BZOJ 3585 mex
- —(都是板题)



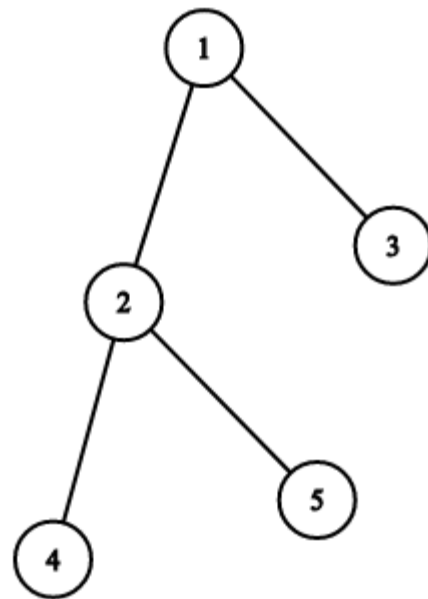
# 带修改莫队

- 简介：
- 考虑普通莫队加入修改操作，如果修改操作可以  $O(1)$  的应用以及撤销（同时也要维护当前区间的答案），那么可以在  $O(n^{3/5})$  的复杂度内求出所有询问的答案。
- 实现：
- 每个询问本来只有二元组  $(l,r)$  表示，现在我用一个三元组  $(l,r,x)$  表示，意思是对  $[l,r]$  进行询问， $x$  表示在此次询问操作之前经过了  $x$  次修改操作。
- 题目：
- BZOJ2120: 数颜色

# 树上莫队

- 简介：
- 一般的莫队只能处理线性问题，我们要把树强行压成序列
- 我们可以将树的括号序跑下来，把括号序分块，在括号序上跑莫队

# 树上莫队



- 询问树上一条链的答案。
- 首先对树进行DFS，每个点入一个时间戳，出一个时间戳。

得到每个点入和出的时间戳( $in[i], out[i]$ )，以及第 $i$ 个时间戳是哪个点。

- 从1开始DFS，得到的时间戳是：  
 $in=[1,2,8,3,5], out=[10,7,9,4,6]$ ，标记顺序是  
 $[1,2,4,4,5,5,2,3,3,1]$
- 我们定义，一个询问 $[l,r]$ 表示询问所有  $l$ 到 $r$ 这一段标记里，恰好出现了一次的节点的答案。

# 树上莫队

- 比如 $[1,5]$ 就是询问1-2-5这条路径的答案。
- 当 $u$ 和 $v$ 是同节点，则询问 $[in[u], in[u]]$ 就可以了。
- 当 $v$ 在 $u$ 的子树里面，我们就可以询问 $[in[u], in[v]]$ 区间，可以发现不在这条路径上的一定出现了0/2次，在这条路径上的一定出现了1次。
- $u$ 在 $v$ 子树中就相反。
- 否则，显然 $[in[u], out[u]], [in[v], out[v]]$ 不相交。
- 假设 $out[u] < in[v]$ ，那么我们可以询问 $[out[u], in[v]]$ 。此时除了LCA以外，这条路径上的节点恰好出现了一次。
- 再加上LCA的贡献就可以了。
- 否则就询问 $[out[v], in[u]]$ 即可。



# 总结

- 其实还有很多（题目）可以讲，但这里没有多说。
- 还有一些（神仙）数据结构没有讲。
- 以后完善吧。
- （未完待续……）