动态树

成都七中　nzhtl1477

动态树

- 指支持动态改变树形态，维护树上信息的数据结构
- 1.Link Cut Tree，主要维护链信息
- 2.Euler Tour Tree，主要维护子树信息
- 3.Top Tree，最强的树上分治信息维护工具

Link Cut Tree

- 维护一个有 n 个节点的森林，支持:
- 1. 在两个节点之间连一条边
- 2. 断开两个节点之间的边
- 3. 将一个点所在的树的根变为这个点
- 4. 查询以两个点为端点的简单路径的信息

Link Cut Tree

- 也叫 Sleator Tarjan Tree
- 原理是用一棵支持 finger search 的平衡树（如 splay）来维护所有重链，这里的重链是动态的
- 每次访问一个点的时候会将这个点到当前根的路径放在同一个重链中，这个操作叫做 access
- 同时支持换根

Link Cut Tree

- 可以证明在树上这样的重链段变化均摊是 O(mlogn) 次，平衡树复杂度为单次 split merge O(logs)，所以理应 O(log^2n) 单次
- 但是由于使用 splay 等结构可以和 lct 的结构很好地 match 起来，可以证明复杂度为 O(mlogn) 均摊的（然而比如 leafy tree 这种结构就很难和 lct 的结构 match 起来，因为天生性质冲突）

Link Cut Tree

- Link cut tree 可以高效地支持链信息维护
- 如查询 x 到 y 的简单路径信息，可以先把 x 节点设为根节点，然后访问 y 节点，这样 x 到 y 的节点就在同一条重链上了，也就被同一棵平衡树所维护
- 可以发现这样的操作类似于平衡树维护区间的时候提取一段区间

Link Cut Tree

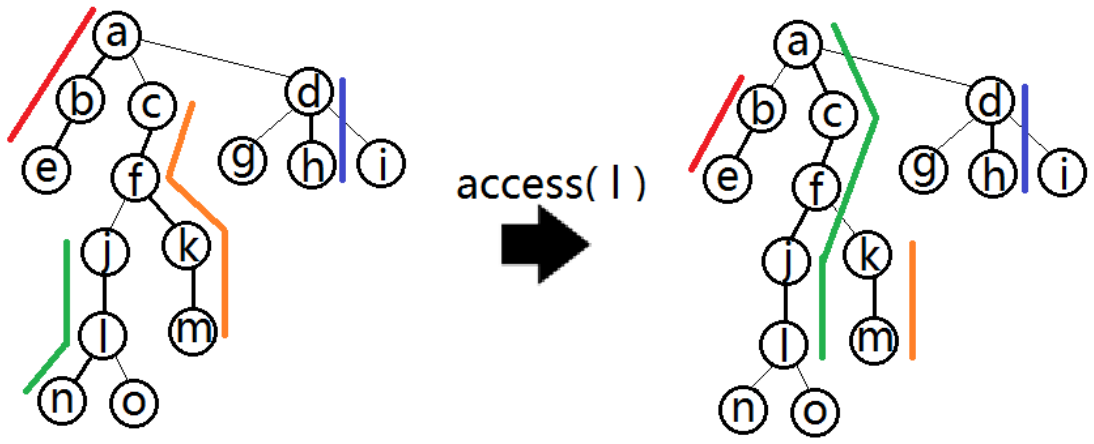
- 将原树划分为一些节点深度递增的路径，通过维护这些路径的信息来维护树上的简单路径的信息，具有以下性质：
- 1. 将树上的节点划分为一些深度递增的路径，这些路径覆盖了树上所有的节点，并且彼此间不相交。
- 2. 将树上的边分为普通边和偏重边，偏重边连接的两个节点必须在同一棵辅助树中，偏重边以外的边被称为普通边。

辅助树

- 将每条路径上的节点按照深度从小到大排序，可以将其看成一个序列。对每条路径使用一棵被称作辅助树的平衡二叉搜索树，用类似维护动态序列的方法去维护。在这个特化的问题上，使用效率最高的伸展树来维护可以达到最好的效果。

Access

- 这个是 `link cut tree` 的核心操作，所有的其他操作都是基于访问操作的。
- 当访问一个节点 `x` 的时候，我们将 `x` 到根的简单路径上所有点放进同一棵辅助树里面，并且将这些点从其他辅助树中删除，即这些点之间的边全部变成了偏重边，而这些点和其他点的边全部变成了普通边。



Link Cut Tree

- 如何查询 x 到 y 的信息呢?
- 我们先把根变成 x，然后 access(y)，然后 x 到 y 这一段就在一棵平衡树上了，查询平衡树信息即可
- 如何换根呢?
- Makeroot 操作

Makeroot 操作

- 设原来的根为 x，新的根为 y
- 先 access(y)，将 x 到 y 放到同一棵平衡树种
- 发现 x 到 y 这一段的深度顺序需要取反
- 然后其他节点的深度顺序不发生任何改变
- 如何取反呢?
- 打一个 rev 标记，和区间取反一个原理

Link

- Link(x,y)：先 makeroot(y)，然后 access(x)，然后 fa[y]=x

Cut

- Cut(x,y)：不妨设 y 在树上深度比 x 大， access(x)，然后直接 fa[y]=0

实现

- 网上一堆实现的很好的板子
- Hzwer 的就还不错?
- http://hzwer.com/3921.html
- 由于篇幅限制这里不详细讲实现了
- LCT 比 splay 还好写 233

Luogu3203 [HNOI2010] 弹飞绵羊

某天，Lostmonkey 发明了一种超级弹力装置，为了在他的绵羊朋友面前显摆，他邀请小绵羊一起玩个游戏。

游戏一开始，Lostmonkey 在地上沿着一条直线摆上 n 个装置，每个装置设定初始弹力系数 k_i，当绵羊达到第 i 个装置时，它会往后弹 k_i 步，达到第 $i+k_i$ 个装置，若不存在第 $i+k_i$ 个装置，则绵羊被弹飞。

绵羊想知道当它从第 i 个装置起步时，被弹几次后会被弹飞。为了使得游戏更有趣，Lostmonkey 可以修改某个弹力装置的弹力系数，任何时候弹力系数均为正整数。

Solution

- 如果将每个点 `i` 看做是和 `i+ki` 连边，则我们干的事情是:
- 1. 修改一个点的父亲
- 2. 查询一个点到根的距离
- 这个可以用 `lct` 直接维护

Luogu3703 [SDOI2017] 树点涂色

Bob 有一棵 n 个点的有根树，其中 1 号点是根节点。Bob 在每个点上涂了颜色，并且每个点上的颜色不同。

定义一条路径的权值是：这条路径上的点（包括起点和终点）共有多少种不同的颜色。

Bob可能会进行这几种操作：

- `1 x` 表示把点 x 到根节点的路径上所有的点染上一种没有用过的新颜色。
- `2 x y` 求 x 到 y 的路径的权值。
- `3 x` 在以 x 为根的子树中选择一个点，使得这个点到根节点的路径权值最大，求最大权值。

Bob一共会进行 m 次操作

Solution

- 注意到所有点颜色不同，而且每次是涂上一种新的颜色
- 根到一个点都染成同一个颜色，这个我们可以联想到 lct 的 access 操作，也是把根到一个点都变成同一个重链
- 然后我们转换一下题意
- 1.access(x)
- 2. 求 x 到 y 要跳多少次重链
- 3. 求 x 子树中一个点，使得其跳重链到根节点次数最多

LCT 的 Access 均摊

- 可以类比区间染色的颜色段均摊，这个是 O(n + m) 次的
- LCT 的 Access 均摊是 O((n+m)logn) 次的

Solution

- 相当于我们每次也是合并颜色段那样
- 然后维护一个数组 a[i] 表示 i 节点到根的颜色数
- 每次修改的时候就是链的 a[i]+1-1 这样的
- 然后每次查询是子树的 max(a[i])
- 由于树形态不变，可以用 HLD 实现，O(log^2n) 或 O(log n)

Luogu4172 [WC2006] 水管局长

每天供水公司可能要将一定量的水从 u 处送往 v 处，嘟嘟需要为供水公司找到一条从 u 至 v 的水管的路径，接着通过信息化的控制中心通知路径上的水管进入准备送水状态，等到路径上每一条水管都准备好了，供水公司就可以开始送水了。嘟嘟一次只能处理一项送水任务，等到当前的送水任务完成了，才能处理下一项。

在处理每项送水任务之前，路径上的水管都要进行一系列的准备操作，如清洗、消毒等等。嘟嘟在控制中心一声令下，这些水管的准备操作同时开始，但由于各条管道的长度、内径不同，进行准备操作需要的时间可能不同。

供水公司总是希望嘟嘟能找到这样一条送水路径，路径上的所有管道全都准备就绪所需要的时间尽量短。嘟嘟希望你能帮助他完成这样的一个选择路径的系统，以满足供水公司的要求。另外，由于 MY 市的水管年代久远，一些水管会不时出现故障导致不能使用，你的程序必须考虑到这一点。

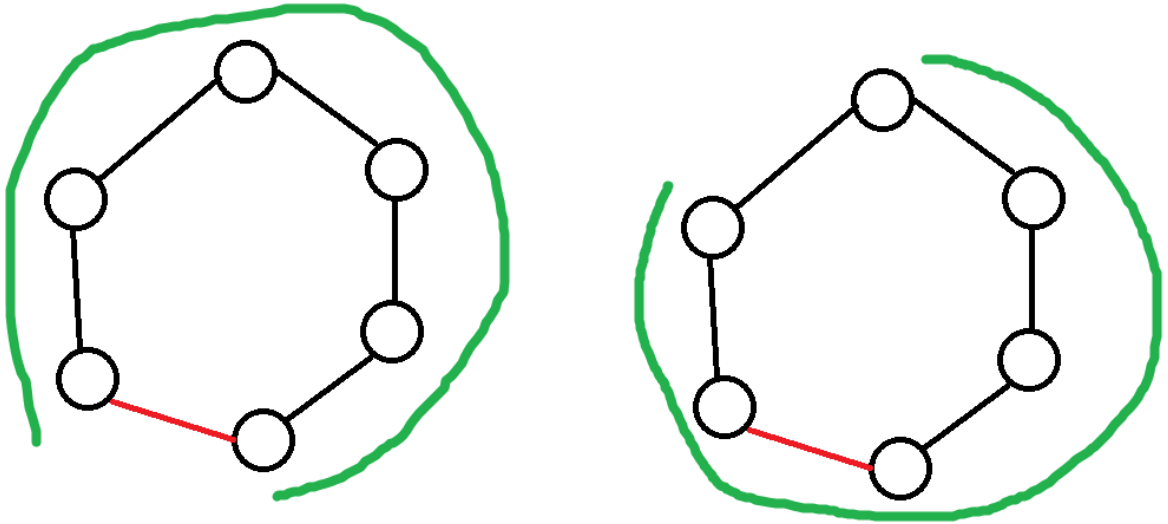
不妨将 MY 市的水管网络看作一幅简单无向图（即没有自环或重边）：水管是图中的边，水管的连接处为图中的结点。整张图共有 n 个节点和 m 条边，节点从 1 至 n 编号。

Solution

- 把删边离线掉变成加边
- 加边，维护 MST
- 如何维护呢?

Solution

- 加边的时候如果成了一个环，就在环上删掉边权最大的边
- LCT 维护链 max，每次可能进行 O(1) 次 link cut



Loj 121

- 维护一个 n 个节点的无向图
- 1. 加边
- 2. 删边
- 3. 查询 x 和 y 是否连通
- 允许离线

线段树时间分治

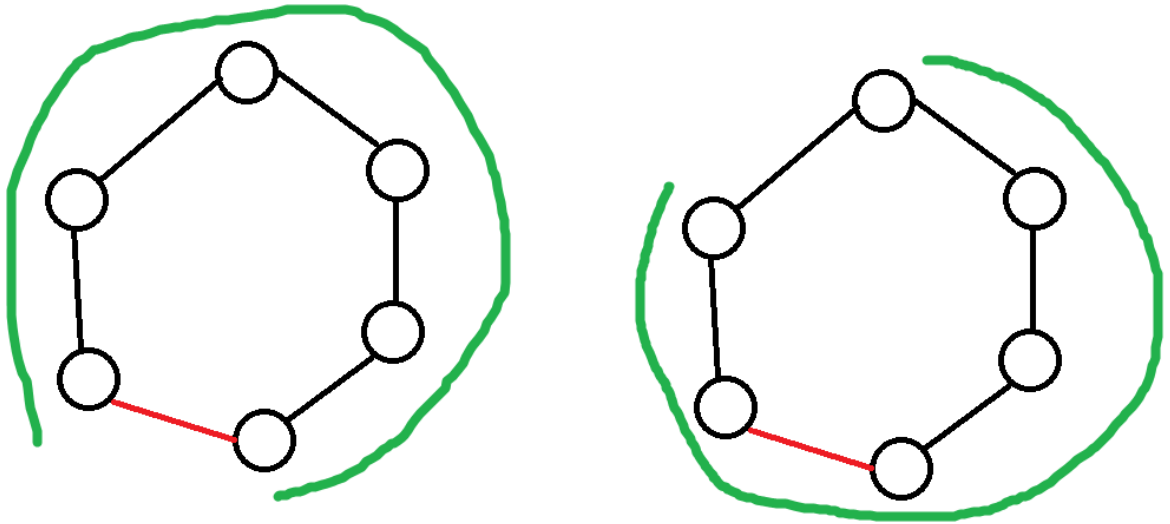
- 考虑对时间建一棵线段树，离线算出每条边在哪些时间中存在，然后将其插入线段树的一个区间中
- 这样每条边只被插入 O(logn) 个节点中
- 然后考虑 DFS 这棵线段树，每到一个节点的时候，将所有这个节点的边所对应的 x 和 y 合并

线段树时间分治

- 可以发现这样每条边只被插入一次
- 但是我们这个结构必须支持撤回，也就是一个子树空了之后必须能删除这个子树
- 所以需要使用可撤销的并查集，复杂度为 O(logn)
- 每次记录下哪些内存位置被改成了什么，这样维护一个栈撤回，和之前讲过的不删除莫队一样
- 总复杂度 O((n+m)log^2n + qlogn)

LCT 维护最小生成树

- 考虑使用 LCT 维护关于删除时间的最大生成树
- 由于我们离线了，所以知道每条边什么时候会被删掉
- 加边的时候如果成了一个环，就在环上删掉最早会被删掉的边
- LCT 维护链 `min`，每次可能进行 `O(1)` 次 `link cut`



LCT 维护最小生成树

- 删边的时候
- 如果这条边在 LCT 上，则意味着这个环上比这条边早删掉的也都删了，那在 LCT 上进行一次 cut
- 否则这条边反正也不在 LCT 上，不影响连通性，不管
- 每次查询就看两个点是否在 LCT 上连通

- 这样总复杂度 O((n+m+q)logn) 维护了动态连通性

Uoj274【清华集训2016】温暖会指引我们前行

任务描述

虽然小R住的宿舍楼早已来了暖气，但是由于某些原因，宿舍楼中的某些窗户仍然开着（例如厕所的窗户），这就使得宿舍楼中有一些路上的温度还是很低。

小R的宿舍楼中有n个地点和一些路，一条路连接了两个地点，小R可以通过这条路从其中任意一个地点到达另外一个地点。但在刚开始，小R还不熟悉宿舍楼中的任何一条路，所以他会慢慢地发现这些路，他在发现一条路时还会知道这条路的温度和长度。每条路的温度都是互不相同的。

小R需要在宿舍楼中活动，每次他都需要从一个地点到达另一个地点。小R希望每次活动时经过一条最温暖的路径，最温暖的路径的定义为，将路径上各条路的温度从小到大排序后字典序最大。即温度最低的路温度尽量高，在满足该条件的情况下，温度第二低的路温度尽量高，以此类推。小R不会经过重复的路。由于每条路的温度互不相同，因此只存在一条最温暖的路径。

对于小R的每次活动，你需要求出小R需要走过的路径总长度。如果小R通过当前发现的路不能完成这次活动，则输出 -1。

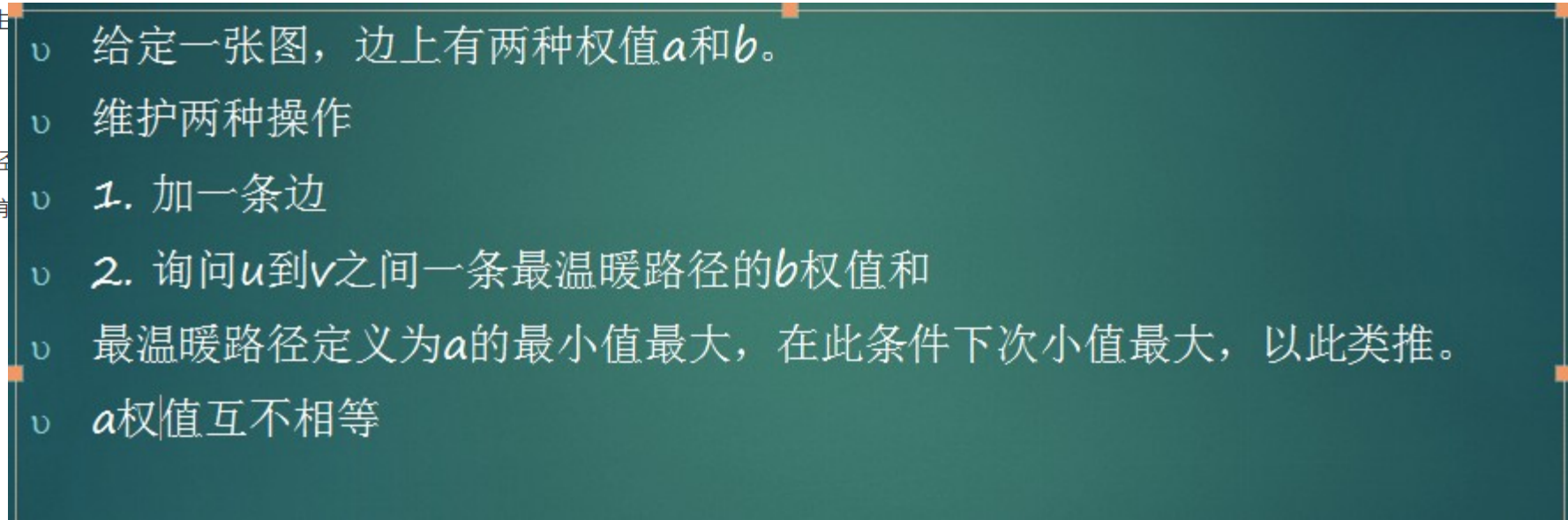
注意本题中的字典序与传统意义上的字典序定义有所不同，对于两个序列$a,b(a \neq b)$，若a是b的前缀则a的字典序较大，同时可以推出空串的字典序最大。

输入格式

第一行两个正整数 n,m。表示小R的宿舍楼中有 n 个地点，共发生

接下来 m 行，每行描述一个事件，事件分为三类。

1. `find id u v t l` 表示小R发现了一条连接u和v之间的路，
2. `move u v` 表示小R要从u到达v，你需要计算出最温暖的路径
3. `change id l` 表示从u到v这条边的长度变为了l（保证在当前



Solution

- 最小值最大，次小值最大，以此类推，这个是最大瓶颈树
- 和 MST 是等价的
- 维护的时候加入一条边的时候在环上删除比其大的边就行了

- O((n+m)logn)

Luogu2387 [NOI2014] 魔法森林

为了得到书法大家的真传，小 E 同学下定决心去拜访住在魔法森林中的隐士。魔法森林可以被看成一个包含 n 个节点 m 条边的无向图，节点标号为 $1, 2, 3, \ldots, n$，边标号为 $1, 2, 3, \ldots, m$。初始时小 E 同学在 1 号节点，隐士则住在 n 号节点。小 E 需要通过这一片魔法森林，才能够拜访到隐士。

魔法森林中居住了一些妖怪。每当有人经过一条边的时候，这条边上的妖怪 就会对其发起攻击。幸运的是，在 1 号节点住着两种守护精灵：A 型守护精灵与 B 型守护精灵。小 E 可以借助它们的力量，达到自己的目的。

只要小 E 带上足够多的守护精灵，妖怪们就不会发起攻击了。具体来说，无向图中的每一条边 e_i 包含两个权值 a_i 与 b_i 。若身上携带的 A 型守护精灵个数不少于 a_i ，且 B 型守护精灵个数不少于 b_i ，这条边上的妖怪就不会对通过这条边的人发起攻击。当且仅当通过这片魔法森林的过程中没有任意一条边的妖怪向小 E 发起攻击，他才能成功找到隐士。

由于携带守护精灵是一件非常麻烦的事，小 E 想要知道，要能够成功拜访到 隐士，最少需要携带守护精灵的总个数。守护精灵的总个数为 A 型守护精灵的个数与 B 型守护精灵的个数之和。

Solution

- 考虑先按照 A[i] 为关键字将每条边排序，然后按顺序动态加入每一条边。以下定一条路径的代价为一条路径经过的所有边的 B 的最大值。可以知道，加入第 i 条边之后的最小代价，就是 A[i] 加上从 1 到 n 的所有路径中，代价最小的路径的代价。
- 看到动态加边，可以想到使用 LCT 来完成。由于要维护边权，所以要把边在 LCT 中理解成点。
- 然后问题变成了一个动态维护 MST 的问题，用之前的方法即可维护
- O((n+m)logn)，现场数据很弱暴力可以随便过

Hdu 5398

- 给你一个 n, 让你输出一个最大生成树的价值。
- 有 n 个点，任意两点 (u,v) 有条边，边权为 gcd(u,v) 。
- n<=1e5

Solution

- 如果 (x,y) 连边，且 d=gcd(x,y) 使得 d<x 且 d<y，那么考虑 (d,x)，(d,y)，去掉边 (x,y) 后 d 必然和一个不在一起，那么连上这条边即可。
- gcd(x,y)>=gcd(x,d),gcd(y,d)
- 所以最后的任意边 (x,y) 都是 xly 或 ylx 的，我们只需要维护这样的边就可以了
- 然后从小到大枚举然后 LCT 维护最大生成树即可，这里每次枚举一个值的约数，所以总次数是 O(nlogn) 的

- O(nlog^2n)，听说暴力也能过

Bzoj4025 二分图

- 有 n 个点 m 条边，边会在 start 时刻出现在 end 时刻消失，求对于每一段时间，该图是不是一个二分图。

Solution

- 二分图等价于每个连通子图都没有奇环
- 还是考虑离线
- 用 LCT 维护删除时间的最大生成树。

Solution

- 加入一条边时，如果两点不连通则直接 link，否则肯定有一条边多余，若形成奇环则将多余的边加入集合。
- 删除一条边时，若这条边是树边则直接删除，否则若在集合中，则从集合中删除。
- 查询时，如果集合中没有边，则为二分图。
- 当然，我们也可以用线段树时间分治来做，和连通性类似

- O((n+m)logn)

Bzoj3514 Codechef MARCH14 GERA LD07 加强版

- n 个点 m 条边的无向图，询问保留图中编号在 [l,r] 的边的时候图中的联通块个数，强制在线

Solution

- 先用 LCT 预处理一下，依次加入每一条边
- 当前加入第 i 条边，如果这个边加入之后成环了，就弹出这个环上面编号最小的边 x，并且记录下 b[i] = x，默认 b[i] = 0
- 之后就是查询 n- 区间中 b 值小于 l 的数个数
- 因为如果 b 值小于 l，那这个边与 [l,r] 内边不成环，所以连通块数量 -1
- 否则没有贡献
- 时间复杂度 O((n+m)logn)，空间复杂度 O(nlogn)

Loj 6038

- 给你 n 个点，支持 m 次操作，每次为以下两种：
- 1. 连一条边，保证连完后是一棵树 / 森林；
- 2. 询问一个点能到达的最远的点与该点的距离。
- 强制在线。

Solution

- 与直径相关的结论 1：与一个点距离最大的点为任意一条直径的两个端点之一。
- 与直径相关的结论 2：两棵树之间连一条边，新树直径的两个端点一定为第一棵树直径的两个端点和第二棵树直径的两个端点这四者中之二。
- 于是问题就变简单了，用并查集维护每个连通块的直径即可。由于强制在线，所以必须用 LCT 维护树上距离。

Luogu4230 连环病原体

- 有 n 种病原体，它们之间会产生 m 种无方向性的影响，第 i 种影响发生在 u[i],v[i] **两种**病原体之间。
- 我们把所有的**影响**按编号顺序排成一个序列，如果某一个区间包含有环，那么这个区间被称作加强区间。
- 求每种影响分别在多少个加强区间中出现过。

Solution

- 考虑扫描线来处理出所有极长的无环区间，这样我们就方便处理出所有有环的区间了
- 这里使用 two-pointers 的方法
- 扫描线加入一个影响的时候，如果成环，则我们需要一直删左边端点直到不成环，或者我们在环上找到最小编号删掉也行
- 这样 O(nlogn) 预处理完之后随便搞搞就算出来了

静态 lct

- 很多问题并不涉及到树形态的修改，所以用 lct 维护会常数比较大
- 可以考虑将 lct 给”静态化”
- 发现 HLD 的问题是虽然跳轻边的次数是 O(logn) 的，但是在重链上表现不佳
- 因为 HLD 是对于每条重链局部平衡的，并不能和全局的结构很好地配合起来

静态 lct

- 对于重链我们不建一个平衡的线段树，而是建一个基于 biased dictionary problem 的 bst
- 将每个节点赋予额外权值为其轻儿子子树 size 和 +1
- 对于每条重链我们找该重链的带权重心，以该节点作为 bst 的根，两边做为左右子树建 bst
- 轻边不改变
- 可以证明这样的结构我们每次跳 a 层，子树 size 一定会增大 1/b，其中 a 和 b 是两个常数，好像是 bst 上跳三层一定增大一倍，所以深度是 O(logn) 的
- 这个挺好写的，缺点是静态结构无法动态改变树的形态

Link Cut Tree

- LCT 可以经过进一步拓展实现 LCT 维护子树，再进一步拓展成为 Top tree，由于 Top tree 具有维护子树动态 DP 信息的性质，所以实质上 Top tree 可以看做是动态化的树链剖分，即动态化的链分治

LCT 维护子树

- 对于每个节点，维护所有虚儿子的信息的合并
- 每次 access 的时候对一些点的一个儿子进行轻重边切换，这个时候可以 O(1) 维护这个切换对该点所有虚儿子信息的合并的影响
- 然后查询一个点子树的时候把其虚儿子的信息加上实儿子的信息，由于一个点最多有一个实儿子，所以这里复杂度为 O(logn)
- 缺点是必须支持子树信息可差分

LCT 维护子树

- 具体来说就是维护 f[x] 和 g[x]，分别代表所有虚儿子的信息与一个点所有的信息
- f[x]=sigma f[y]，y 为 x 的虚儿子
- g[x]=g[l]+g[r]+f[x]+v[x]，其中 x 是 splay 节点，l 和 r 为其左右儿子
- link(x,y) 时 f[x]+=g[y]
- 查询 x 子树信息时 access(x),splay(x), 此时 x 所有儿子都变成虚儿子了，所以返回 f[x] 即可

Luogu4299 首都

在 X 星球上有 n 个国家，每个国家占据着 X 星球的一座城市，城市从 1 至 n 编号。由于国家之间是敌对关系，所以不同国家的两个城市是不会有公路相连的。

X 星球上战乱频发，如果 A 国打败了 B 国，那么 B 国将永远从这个星球消失，而 B 国的国土也将归 A 国管辖。A 国国王为了加强统治，会在 A 国和 B 国之间修建一条公路，即选择原 A 国的某个城市和 B 国某个城市，修建一条连接这两座城市的公路。

同样为了便于统治自己的国家，国家的首都会选在某个使得其他城市到它距离之和最小的城市，这里的距离是指需要经过公路的条数，如果有多个这样的城市，编号最小的将成为首都。

现在告诉你发生在 X 星球的战事，需要你处理一些关于国家首都的信息，具体地，有如下3种信息需要处理：

- `A x y`：表示某两个国家发生战乱，战胜国选择了 x 城市和 y 城市，在它们之间修建公路（保证其中城市一个在战胜国另一个在战败国）。
- `Q x`：询问当前编号为 x 的城市所在国家的首都。
- `Xor`：询问当前所有国家首都编号的异或和。

Solution1

- 发现每次 link(a,b) 的时候新的重心一定是在 a 所在树的重心到 b 所在树的重心的连线上
- 使用 LCT 维护子树 size
- 每次在这个连线上二分出子树 size 最平衡的位置，这个就是我们的重心
- 这样是 O(logn) 的

Solution2

- 由于题目只有加边操作，所以可以启发式合并
- 启发式合并然后维护一个数据结构，由于没有强制在线所以可以预处理出树
- 每次还是在连线上二分出重心位置

- 这样是 O(log^2n) 的，感觉不会比 LCT 慢

Luogu5526 [Ynoi2012] 惊惶的 SCOI2016 &CF564E

- 树，点有颜色，带修改，每次修改后输出所有链颜色数的和

Solution

- 对每种颜色分别考虑不含该颜色的简单路径条数。

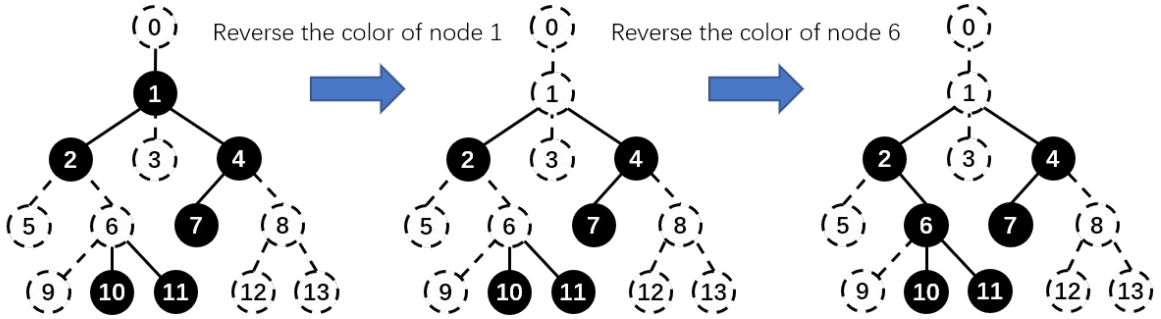
- 令当前处理的颜色为 c，把颜色为 c 的视为白色，不是 c 的视为黑色，那么不含 c 的路径条数就是每个黑联通块的大小的平方和，修改就是当颜色是 c 等价于 颜色不是 c 时翻转一个点的颜色。所以，问题转化成了黑白两色的树，单点翻转颜色，维护黑联通块大小的平方和。这个转化后的问题可以用很多数据结构做

Solution

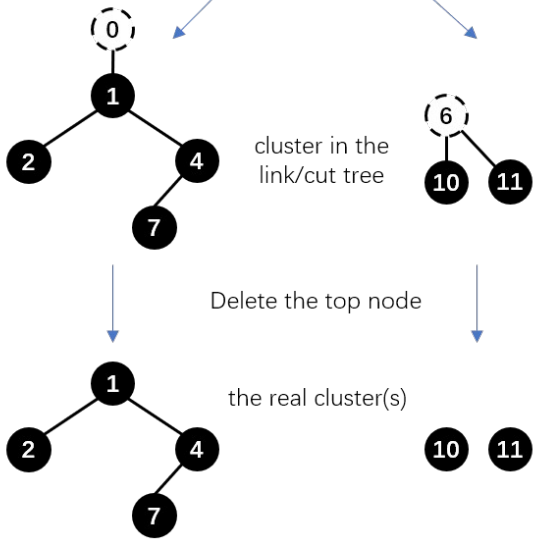
- 对每个点维护子树大小，儿子大小平方和，在 link/cut 的时候更新答案即可。有一个大家熟知的 trick，就是每个黑点向父亲连边，这样真正的联通块就是 Link/cut Tree 里的联通块删掉根。

- 具体看图吧，图讲的挺清楚的：

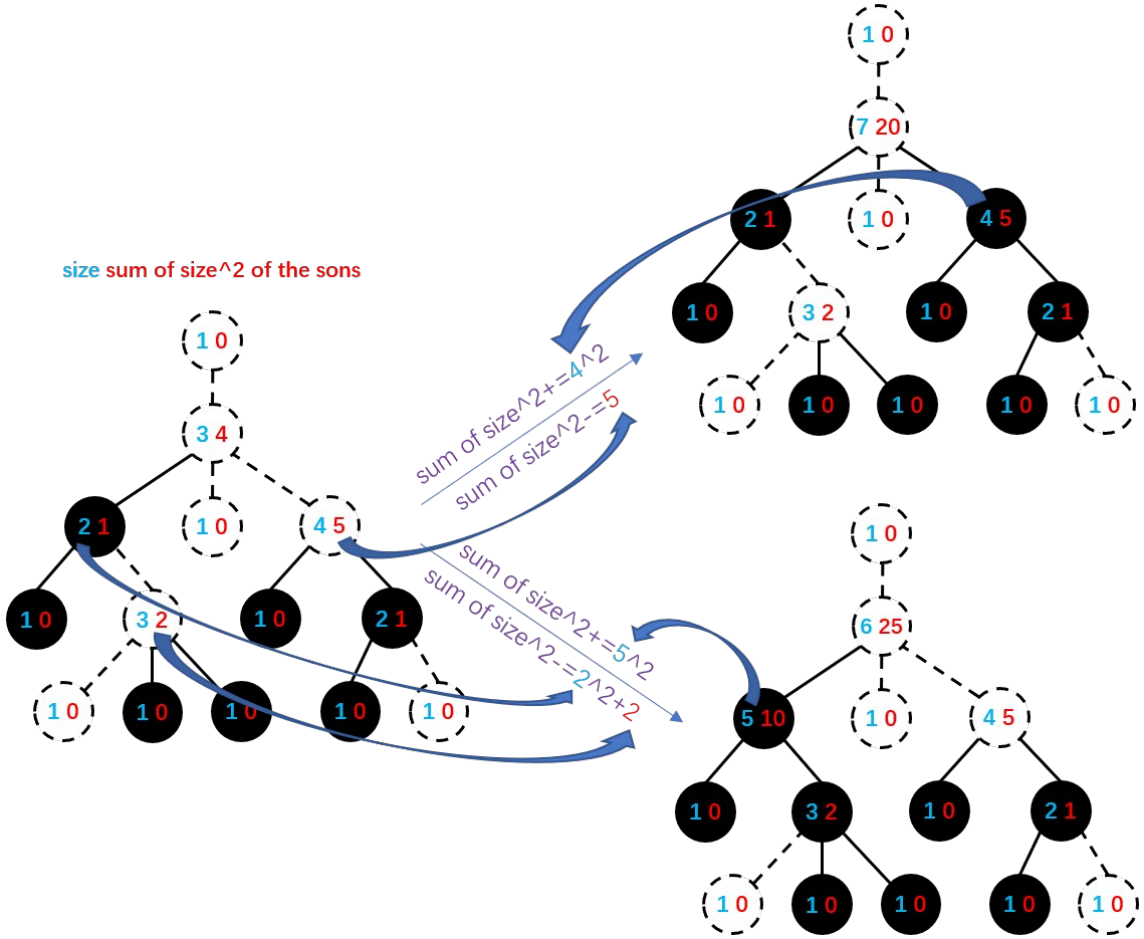
Solution

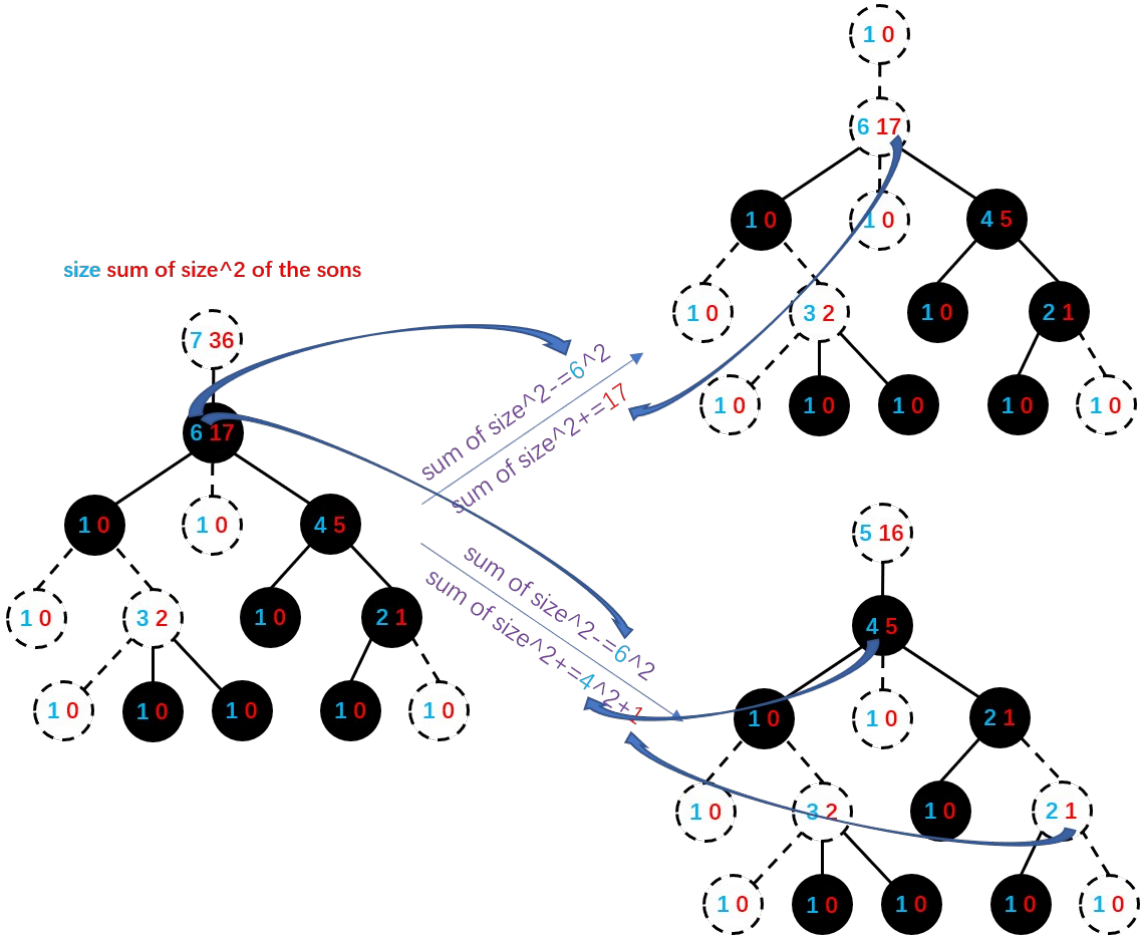


clusters: 1,2,4,7 ; 10 ; 11 2 ; 4,7 ; 10,11 2,6,10,11 ; 4,7



Solution


size sum of size^2 of the sons
sum of size^2+=4^2
sum of size^2-=5
sum of size^2+=5^2
sum of size^2-=2^2+2


size sum of size^2 of the sons
sum of size^2-=6^2
sum of size^2+=17
sum of size^2-=6^2
sum of size^2+=4^2+1

Solution

- 有个只用 HLD，不用 biased search tree，不用 lct 的做法
- 我感觉挺有趣的，如果感觉时间太多了就来讲讲吧

Uoj207

火车司机出秦川，跳蚤国王下江南，共价大爷游长沙。每个周末，勤劳的共价大爷都会开车游历长沙市。

长沙市的交通线路可以抽象成为一个 n 个点 $n-1$ 条边的无向图，点编号为 1 到 n，任意两点间均存在**恰好**一条路径，显然两个点之间最多也只会有一条边相连。有一个包含一些点对 (x,y) 的**可重**集合S，共价大爷的旅行路线是这样确定的：**每次他会选择 S 中的某一对点 (x,y)，并从 x 出发沿着唯一路径到达 y。**

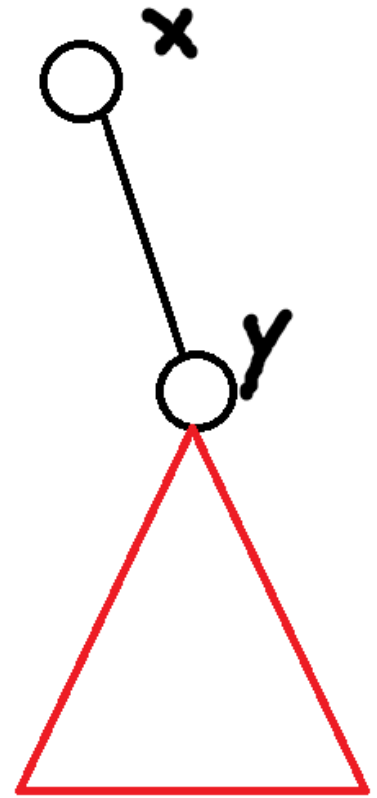
小L是共价大爷的脑残粉，为了见到共价大爷的尊容，小L决定守在这张图的某条边上等待共价大爷的到来。为了保证一定能见到他，**显然小L必须选择共价大爷一定会经过的边——也就是所有共价大爷可能选择的路径都经过的边。**

现在小L想知道，如果他守在某一条边，是否一定能见到共价大爷。

然而长沙市总是不断的施工，也就是说，**可能某个时刻某条边会断开，同时这个时刻一定也有某条新边会出现，且任意时刻图都满足任意两点间均存在恰好一条路径的条件。**注意断开的边有可能和加入的新边连接着相同的两个端点。共价大爷的兴趣也会不断变化，**所以S也会不断加入新点对或者删除原有的点对。**当然，**小L也有可能在任何时候向你提出守在某一条边是否一定能见到共价大爷的问题。你能回答小L的所有问题吗?**

Solution

- 考虑用 LCT 维护子树信息
- 假设询问的边是 (x,y)
- 选出这条边中深度最大的点，不妨设是 y
- 等价于查询 y 在以 x 为根的子树中是否把集合中每条链
- 恰好出现了一端
- 如何维护这个信息呢?



Solution

- 这种信息的维护是困难的，考虑设计一种哈希方法
- 对每个二元组我们把信息在两个点上各存一个
- 然后合并信息的时候使用 xor 运算，这样一条边存在两次就等同于不存在了
- 把每个二元组都随机一个 long long，然后变成维护子树 xor 和
- 如果子树 xor 和等价于集合每条边的 xor 和，就认为这条边符合条件，否则认为不符合
- 错误率低于 1e-9

Solution

- 使用 LCT 维护子树，每次集合中插入删除一个二元组的时候就变成两次单点修改，查询边的时候就变成查询集合 xor 和
- 由于 xor 可差分，所以可以直接使用 LCT 维护子树的方法

- O((n+m)logn)

静态 lct 维护子树

- 同理，把 lct 换成静态 lct 即可

使用 splay 维护轻儿子的 lct

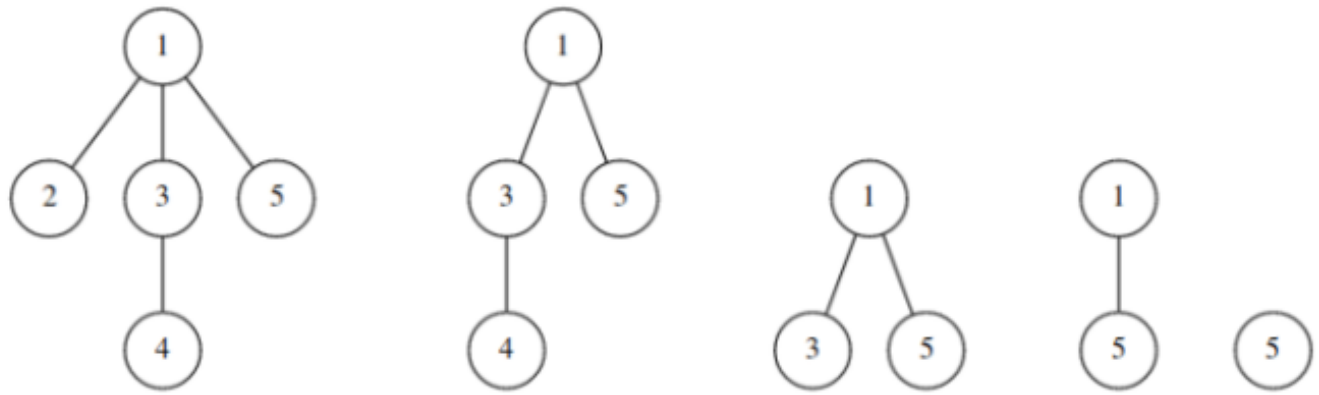
- 我们对每个 lct 节点，再开一棵平衡树维护所有轻儿子
- 这样可以维护子树任意信息，只需要处理边标记 pushdown 到轻边平衡树上的情况，以及轻边平衡树标记 pushdown 到边上的情况——即原 lct 上标记以及每个点的轻边平衡树标记互相的影响
- 这样便达成了 top tree 的全部功能

更多的 LCT 例题

- 有的比前面的稍微难一些

CF1137F Matches Are Not a Child's Play

我们定义一棵树的删除序列为：每一次将树中编号最小的叶子删掉，将该节点编号加入到当前序列的最末端，最后只剩下一个节点时将该节点的编号加入到结尾。



例如对于上图中的树，它的删除序列为：2 4 3 1 5

现在给出一棵n个节点的树，有m次操作：

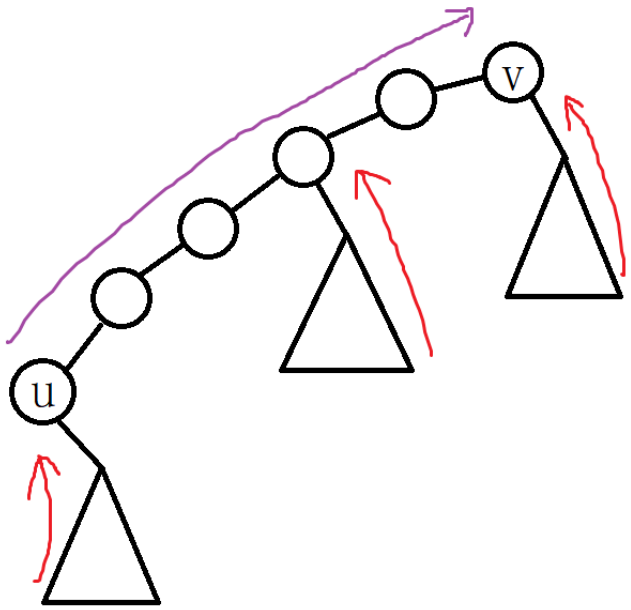
`up v`：将v号节点的编号变为当前所有节点编号的$\max+1$

`when v`：查询v在当前树的删除序列中是第几号元素

`compare u v`：查询u和v在当前树的删除序列中谁更靠前

Solution

- 3 操作可以由两次 2 操作代替
- 由于这个修改只能将一个点修改为最大值 max+1
- 考虑之前最大值 a[u]=max，修改后 a[v]=max+1
- 则这次修改的影响是，先烧完了除 u 到 v 路径上的所有点，然后烧 u，沿着 u 到 v 的路径烧到 v
- 考虑除了这条路径以外的点
- 可以发现这些点被烧的相对顺序不发生变化



Solution

- 考虑 LCT 上的 access 均摊，每次修改即
- makeroot(u)，然后 access(v)，之后 u 到 v 的路径按 u 到 v 的顺序处于删除序列的末端，即 [n-x+1,n]， x 为路径上点数
- 使用一个数据结构维护这样的连续段，支持 access 时将一些连续段删除，之后加入一个新的连续段
- 查询一个点在删除序列的位置时，在 LCT 上找到其属于哪个连续段，然后通过维护的数据结构查询其排名即可

- 总时间复杂度 O((n+m)log^2n)

Luogu6107 [Ynoi2010]Worst Case Top Tree&[Code+7] 六元环

给定序列 $a_0, a_1, \ldots, a_n, a_{n+1}$；满足 $a_0 = a_{n+1} = +\infty$，$a_1, a_2, \ldots, a_n$ 在输入中给出；

对 $1 \le x \le n$，称 $\max_{0 \le i < x, a_i \ge a_x} i$ 和 x 是相邻的，且 $\min_{x < i \le n+1, a_i > a_x} i$ 和 x 是**相邻**的；如果 x 和 y 相邻，则 y 和 x 也相邻；

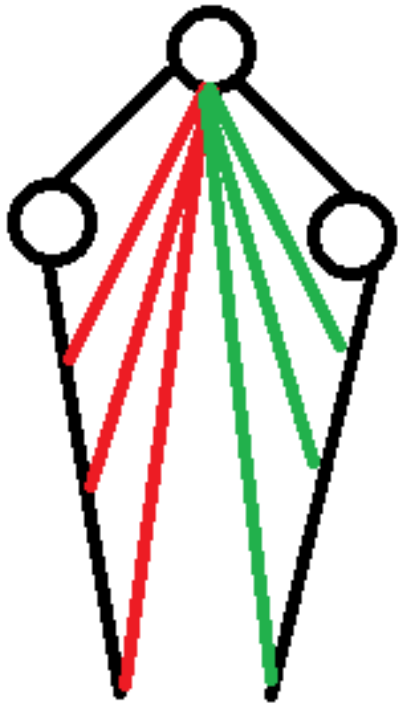
如果 $0 \le b_1, b_2, b_3, b_4, b_5, b_6 \le n+1$，且 b_i 和 b_{i+1} 相邻，b_1 和 b_6 相邻，b_i 互不相同，则称集合 $\{b_1, b_2, b_3, b_4, b_5, b_6\}$ 是一个六元环（即判断两个六元环是否相同时，不考虑 b_i 的顺序）。

共有 m 次修改操作，每次修改操作给出 x y，将 a_x 改为 $a_x + y$；每次修改后要求输出六元环的个数；

以上提到的所有数值为整数，且 $1 \le n, m \le 5 \times 10^5, 1 \le x \le n, 1 \le a_i, y \le 10^9$。

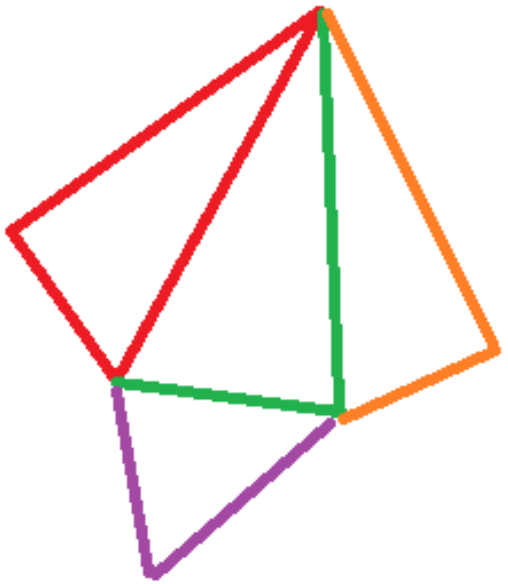
Solution

- 先把笛卡尔树建出来
- 可以发现这个图是把每个点 x 到其左儿子的最右链上每个点连边，到其右儿子的最左链上每个点连边，这样构成的一个平面图



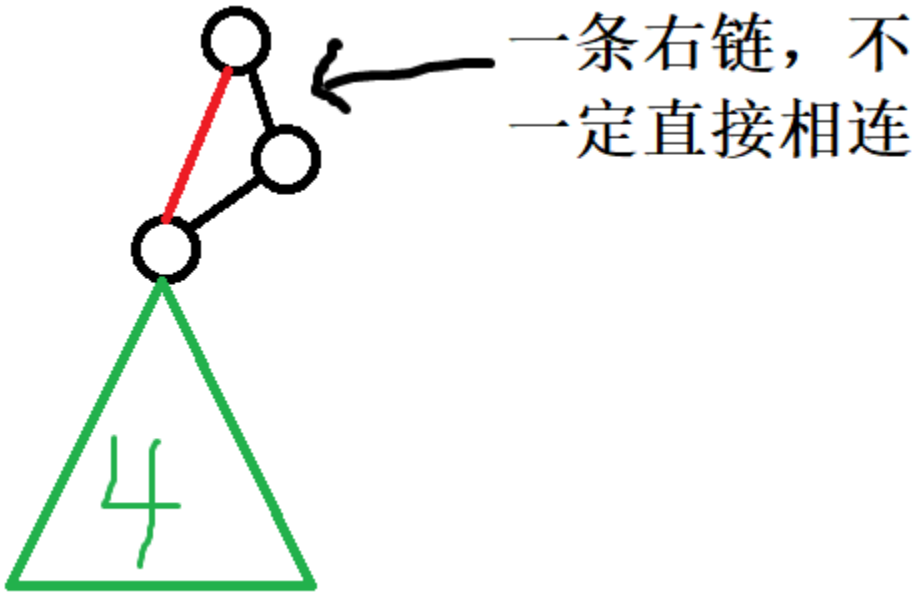
Solution

- 发现这个结构自然得构成了三角剖分结构
- 平面图的对偶图是一棵树
- 所以这个平面图上的 n 元环和相邻的 n-2 个三角形构成的连通子图一一对应



Solution

- 考虑这样的四个三角形在笛卡尔树上是什么样的形态构成的
- 只有下图中这种结构，以及其左右镜像结构
- 如何维护这样的结构呢?



Solution

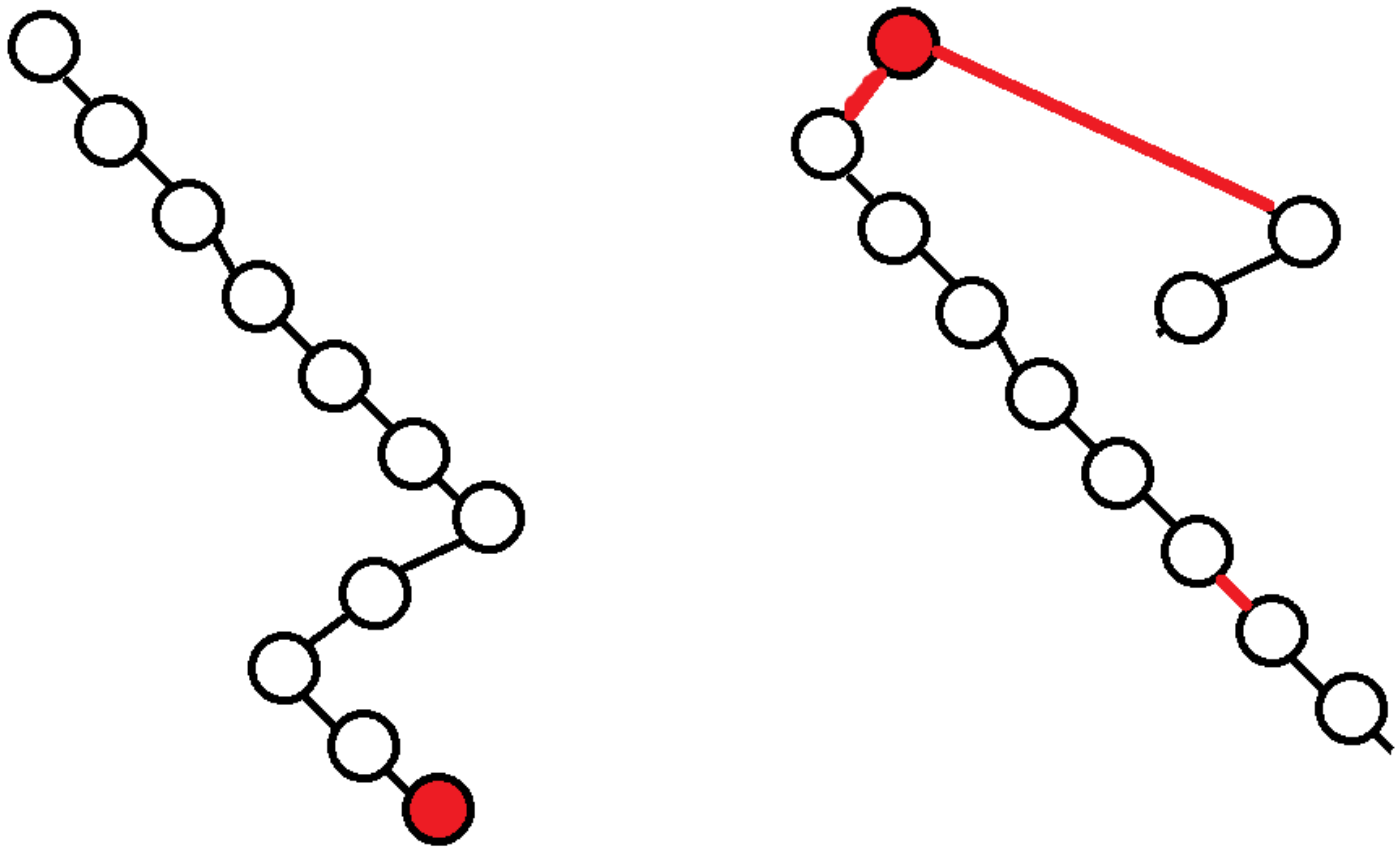
- 我们不用管最上面那个点，直接数大小为 4 的连通子图即可，相当于我们枚举那个 4 个节点的子树，然后唯一确定上面两个点，这里需要特判笛卡尔树的根
- 问题转换为维护大小为 4 的连通子图

Solution

- 指数级复杂度预处理一下每个可能的 4 个节点的连通子图形态
- 然后每个节点维护其下面 3 个点可能的方案数，每次就可以在根上合并了
- 问题转换为需要动态维护笛卡尔树上的局部信息，如果能维护动态笛卡尔树的形态，这个就可以维护了
- 这里只加正数，可能有均摊性质

Solution

- 笛卡尔树把一个点单旋上去的过程：



Solution

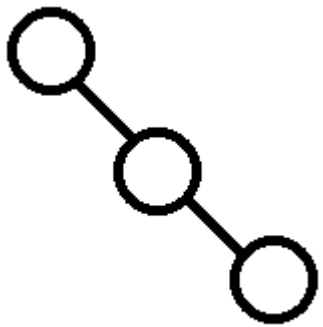
- 可以发现对这条路径，以每个拐点为划分，会把每条极长右链都拼到左边去，每条极长左链都拼到右边去
- 拼完之后拐点变成 O(1) 个
- 所以我们可以考虑 O(拐点个数) 地维护这样的结构，以拐点个数为均摊
- 这里 HLD 一下，轻边导致的拐点是平凡的，类似 access 的切换，重边导致的拐点可以均摊分析出
- 拐点个数变化量 O((n+m)logn)

Solution

- 于是我们均摊维护拐点，考虑直接用 LCT 维护整棵树，上旋 x->y 的时候，就提取出 x 到 y 的路径。
- 然后通过在 splay 上二分来将 x 到 y 的路径分割成若干个权值单调的连续段。
- 将这些连续段分别拆开来后，按照一定顺序合并起来即可。
- 这样时间复杂度可以证明 O((n+m)log^2n)

Solution

- 使用“极长直链剖分”的结构，即把每个 x，假设父亲是 y，父亲的父亲是 z，如果三者方向相同，则放到同一个极长链中
- 按这样的树链剖分方法可以简单地均摊维护拐点，时间复杂度 O((n+m)logn)



欧拉回路树

- ETT
- 在 OI 中把这个当成用平衡树维护 DFS 序即可，一般用来做根不变的问题
- 维护一棵有根树：
- 1. 把 x 的父亲变成 y
- 2. 查询 x 的子树信息

欧拉回路树

- 考虑用平衡树维护 DFS 序
- 构造 DFS 序的时候，DFS 进入 x 的时候插入 left[x]，出去的时候插入 right[x]
- 把 x 的父亲换成 y 也就是说把 left[x]~right[x] 在平衡树上这一段插入到 y 在 DFS 序上 left[y]~right[y] 这一段里，直接放到 right[y] 前面即可

平衡树维护 DFS 序

- 需要平衡树维护父亲，支持 split merge 区间
- 使用各种平衡树都可以维护
- 需要维护每个节点为根， DFS 序两端的指针
- 每次 split merge 区间的时候沿着这个指针向上走，从而提取出子树

不知道哪里看到的题

- 树木仙有一棵根节点为 1 的菊花树 (除 1 号节点外所有节点的父亲都是 1). 这棵树每个节点都有一个权值, 最开始 i 号节点的权值为 vali . 树木仙觉得菊花树不够美观, 决定修改这棵树 . 树木仙每次会将编号在 [l,r] 之间的节点的 父亲修改为 x. 因此, 这棵树的形态会不断改变 . 为了保证美观, 树木仙还会不断修改某个点的权值 . 为了证明你能维护这棵树, 在修改过程中树木仙会不断询问你某棵子树的点权和 .
- 保证不会改动 1 的父亲
- 原文: https://blog.csdn.net/qq_35649707/article/details/79631755

Solution

- father 可以套用区间染色的均摊分析
- 每次把一段 father 为同一个点的节点换父亲可以批量处理，因为他们在 DFS 序上都是连续的，这里可以看做每次加了一个虚点，把所有连接过去的子树都连上了虚点
- 因为只查询子树所以我们用 ETT 来维护就行了

- 总时间复杂度 O((n+m)logn)

Luogu5529 [Ynoi2012] 梦断 SCOI2017

- 给一棵树，点权，支持:
- 1. 单点修改
- 2. 修改一个点所在同色块的颜色
- 3. 查询一个点所在同色块的最浅点和最深点
- 颜色数 c=30

Luogu5529 [Ynoi2012] 梦断 SCOI2017

- 对这题的评价： 6/11

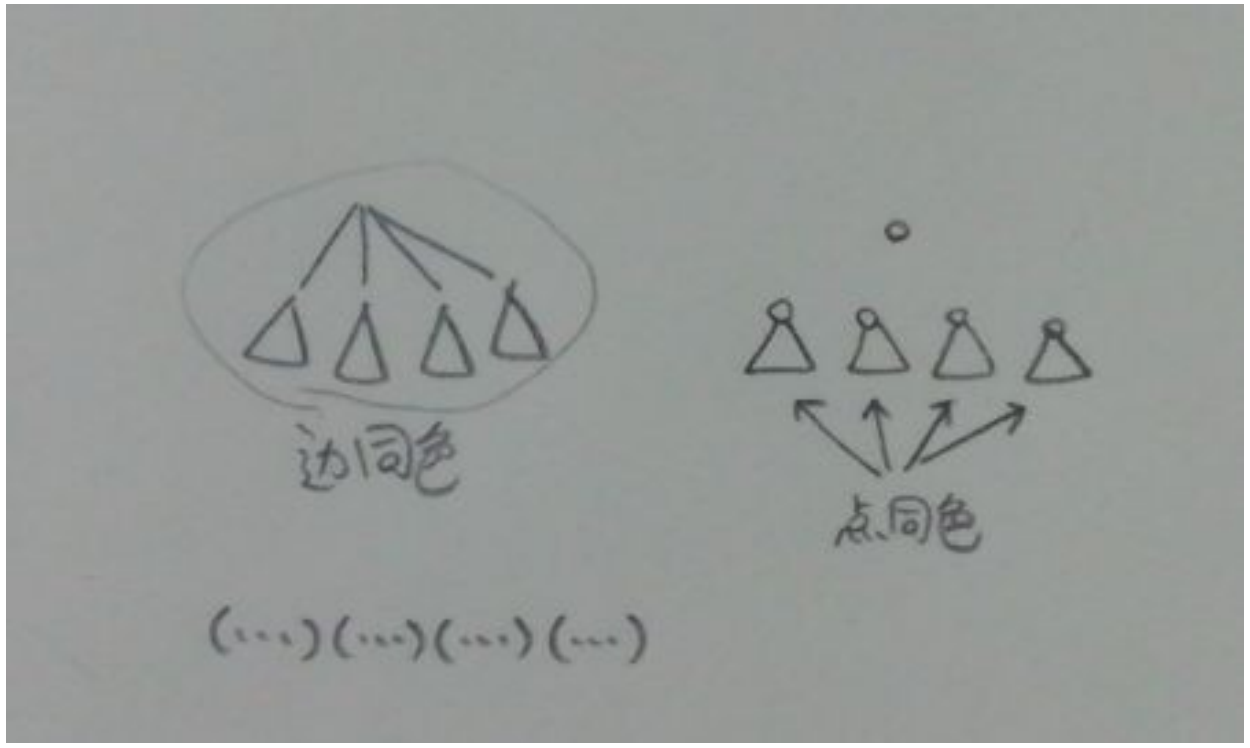
Solution1

- 暴力

- 总复杂度 O(nm)
- 期望得分: 20

Solution2

- 点同色连通块的括号序列对应于所在的边同色连通块的括号序列的一个区间，二者维护的点集信息完全一样，因为边同色联通块只比点同色联通块多一条边



Solution2

- 因为点的联通块没有均摊，但是边的联通块有均摊，所以维护边的联通块而不是点的联通块
- 可以发现改变一条边的颜色只影响 O(1) 个边同色连通块，而点同色连通块可能影响 O(n) 个
- 把点连通块转边连通块就是把每个点 x 的点权放在 x 到父亲 fa[x] 的边上，这样的边连通块

Solution2

- 于是问题可以转化为:
- 边权
- 1. 单边修改
- 2. 修改一条边所在同色块的颜色
- 3. 查询一条边所在同色块的最浅点和最深点

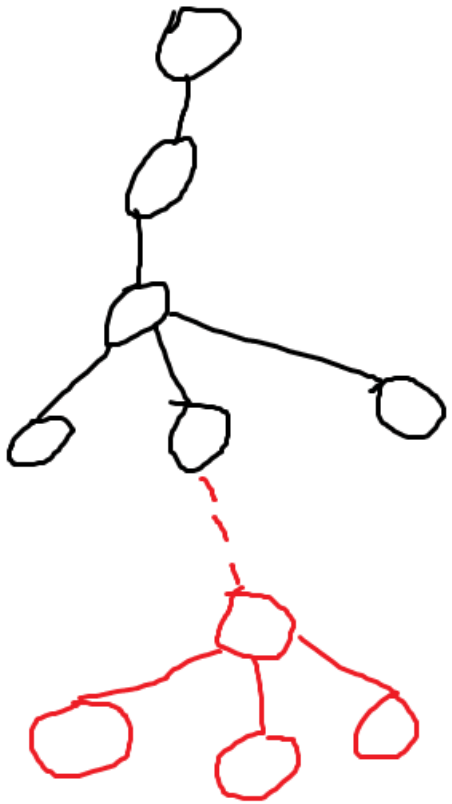
Solution2

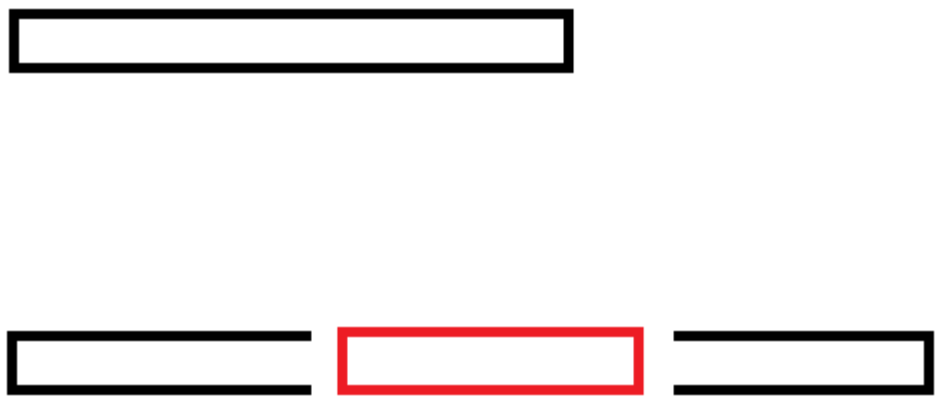
- 通过对每个极大边连通块开一个 ETT 来维护
- 然后每个 ETT 上面每个节点维护 O(c) 个信息，表示这个节点为根的子树里面有没有节点的儿子为 1,2…c 这种颜色

Solution2

- 当把这个连通块染色为某个颜色 x 的时候
- while(root 的 x 颜色信息为 1) 沿着 ETT 结构递归下去，直到找到一个点其满足至少有一个儿子颜色为 x 颜色
- 然后把这个点的所有颜色为 x 的儿子的 ETT 合并进这个 ETT 里面
- 这个通过对每个点开一个 O(x) 的数组来存下每个颜色的儿子节点就可以做到了
- 具体体现在操作上就是合并两棵平衡树，把其中一个平衡树完全合并进另一个平衡树，作为一个区间

Solution2





Solution2

- 这样以 O(clogn) 代价，我们减少了 O(1) 个联通块
- 每次最多加 2 = O(1) 个联通块
- 所以时间复杂度是均摊 O((n + m)clogn)

Solution2

- 点染色就转化为了边染色，可能会合并连通块，这个是平凡的
- 把 x 从 a 染成 b ，要把 x 为根的 b 色连通块和 fa(x) 为根的 b 色连通块和边 (x,fa(x)) 合并
- 查询的话， ETT 上面维护：最浅，最深的 dep ，这个联通块的颜色 size
- 这样就做完了

- 时间复杂度 O(nclogn) ，空间复杂度 O(nc)

Solution3

- 发现我们 ETT 每个节点维护了 O(c) 个信息
- 这 O(c) 个信息是 or 起来的
- 所以可以压位维护

- 时间复杂度 O(nclogn/w)，空间复杂度 O(nc/w)

Top tree

- 最强大的树分治结构
- 目前主要是有两种:
- Self Adjusting Top Tree：基于 path decomposition 的 top tree
- Worst Case Top Tree：基于 contraction 的 top tree

Top tree

- 二者很大程度上相通，这里解释几个术语
- 部分摘抄自
- https://etaoinwu.win/blog/top-tree-notes/

Top tree

- Top tree：一种维护无向动态森林的数据结构
- 将一棵树递归分成若干部分，top-down，称作“分治”；反过来的过程，bottom-up 地把小块合并上来的过程，不妨称作“合治”
- Top tree 其实是类似于 bst，线段树等的结构，二者有很大相似性，只不过一个建立在序列上，以区间为单位，另一个建立在树上，以 cluster 为单位

Cluster

- 为了实现这一目标，我们需要设立合治的单位： cluster （暂译“树簇”，对应物似乎是“区间“之于序列数据结构）。这玩意需要满足以下条件:
- 有一个 O(1) 的边界（否则信息存不下）
- 可以合并拆分（废话）
- 不仅能储存树链信息，还可以储存”树块”信息（子树信息）

相关术语

- **树簇**，简称**簇**：树上的一个连通子图，有至多两个点和全树的其他位置连接
- 这两个结点被称做**界点**（Boundary Node\Endpoint）。
- C 的界点记作 ∂C。不是界点的点被称做**内点** Internal Node，记作 $C\backslash\partial C$。
- 这两个界点之间的路径被称做**簇路径** Cluster Path，记作 $\pi(C)$。
- 有一个界点的被称做**叶簇**（Leaf cluster），其余的称作**路簇**（Path cluster）

相关术语

- 最小的（最基本的）簇是树的一条边，称作**簇元**（Edge cluster\Base cluster）。“元”字取其基本元素之意，或译“簇素”，然其音有误解之可能。如果连接的是叶子与其他，那么有一个界点，称作 leaf edge cluster；反之有两个，称作 path edge cluster （不出现，不译）。
- 如果两个树簇 A,B 有且仅有一个公共点，（显然此时这个公共点是二者的公共界点），两个连通子图的并也是一个簇，那么他们被称做**可合并**（Mergeable），合并的结果记作 $A \cup B$ 。

Cluster 的合并

- 相比于线段树，平衡树维护序列， top tree 的一大难点在于如何合并信息
- 维护序列时每次需要在分治结构上维护两个区间合并为一个区间，维护树上信息时即每次需要在分治结构上维护两个 cluster 合并为一个 cluster
- 类比与序列维护中只有两个相连的区间才可以合并，在树上只有有公共端点的一对 cluster 才是可合并的

Cluster 合并的可能形态

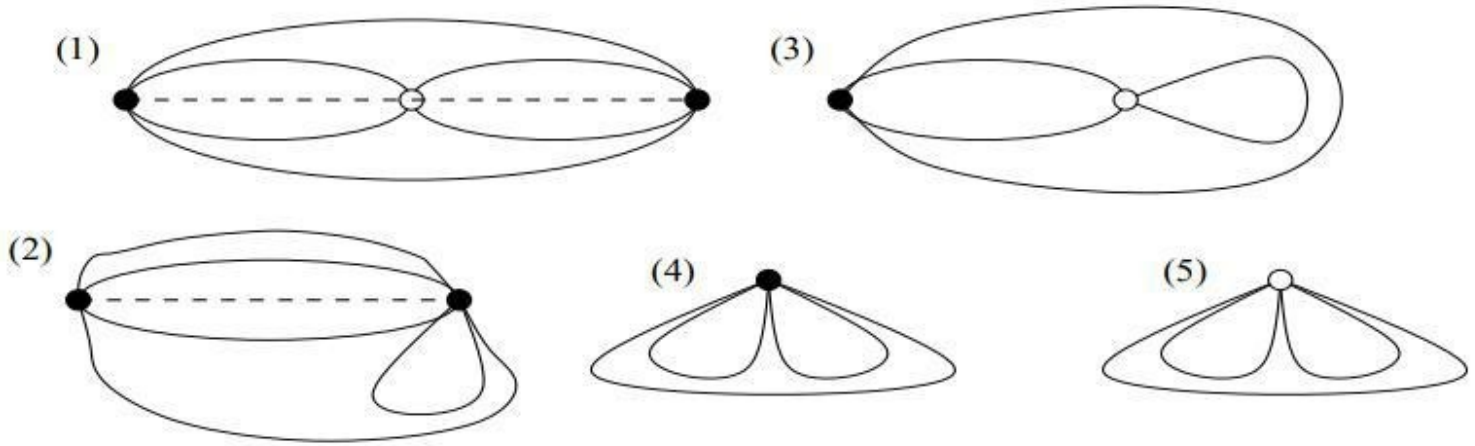


Figure 1: The cases of joining two neighboring clusters into one. The • are the boundary vertices of the joined cluster and the ∘ are the boundary vertices of the children clusters that did not become boundary vertices of the joined cluster. Finally the dashed line is the cluster path of the joined cluster.

Top tree 的树结构

- 类比于线段树的树结构：
- 最开始每个点都是一个最基本的区间 [i,i]
- 最后合并为唯一的一个区间 [1,n]
- 每次合并两个相邻的区间，形成一个树形结构，即线段树

Top tree 的树结构

- Top tree：
- 最开始每条边都是一个 cluster——base cluster
- 最后合并为了一个唯一的 cluster——root cluster
- 每次把两个 cluster 合并为一个 cluster，所以我们记录下 cluster 合并的这个树形结构，即得到了 top tree 的树结构

Top tree 的树结构

- 一个簇对应一个结点
- 每个簇要么是叶子结点（簇元），要么是由两个簇合并起来，将他们作为大簇的孩子
- （父亲 / 祖先 / 后代定义略）
- 如果一个簇是另一个簇的父 / 子 / 祖先 / 后代，且他们的簇路径有至少一条边的交，那么分别称作 path-ancestor/path-descendant/path-child 。

Top tree 的外部操作

- Link(v,w) ：假设 v,w 两个结点在不同的树 Tv,Tw 中。建立一颗新的 Top Tree T，代表 TvTw(u,v) 。 Cut(v,w)：在 T 中去除 (v,w) 这条边，建立 Tv 与 Tw 。 Expose(v,w) ：将 v 与 w 设定为 T 的例外界点。

Top tree 的内部操作

- Merge(A,B)：将可合并簇 A 与 B 合并，创建 C=A∪B 作为他们的父簇，并 ∂C=∂(A∪B) 。设定 Info(C):=Merge(C:A,B)，其中 Merge(C:A,B) 是用户定义的计算信息合并的过程。
  Split(C)：其中 C=A∪B 。调用 Split(C:A,B)，然后把 C 从 T 中删去。Split(C:A,B) 是用户定义的用 Info(C) 更新 Info(A) 和 Info(B) 的过程，可以理解为标记下传。
  Create((v,w))：创建一条新边。调用同名用户定义过程。
  Eradicate(C)：要求 C 是一个簇元。删除这条边。调用同名用户定义过程。

Top tree

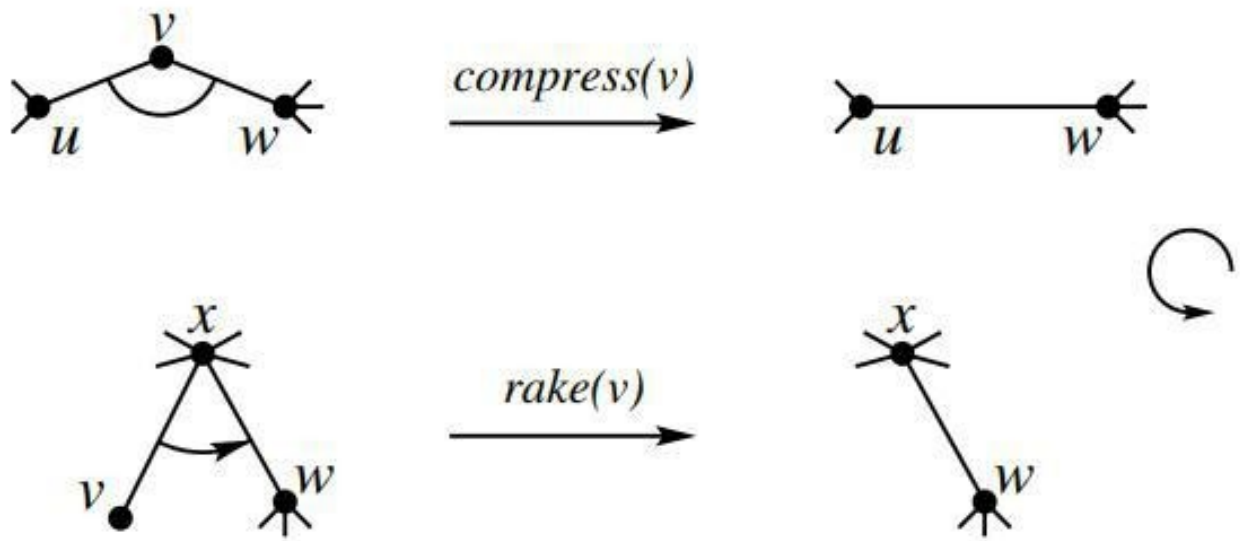
- 这样，我们就可以把 Top tree 描述成一个黑盒：用户提供四个内部操作同名过程负责信息的维护，然后调用外部操作进行修改树结构，然后对根节点进行查询。

Top tree 的构造

- 目前有的 top tree 构造都是 bottom up 的
- 我们考虑自底向上的构建：每次拿到一棵树，进行一些合并，合并完的 cluster 每个都作为一条边放到上层。
- 在同一层一个簇只会被合并一次，因此这里簇等价于底层的边，以下将混用。

Top tree 的构造

- 我们强行（随意地）给每个结点周围的边定一个极角序。这样，就可以定义出每条弧在顺向的前驱后继边；显然构成了一个边的循环欧拉序。
- 考虑两种基本的合并： Rake 和 Compress 。



Top tree 的构造

- Rake(x)： x 是一个叶子结点；将 x 合并到他往上连的顺向边上去。 Compress(x)： x 是一个度为 2 的结点；将 x 两侧的边合起来。
- 对于每一层：使用一个数据结构（欧拉序循环链表）维护尚未被合并的簇；一开始每个簇没有父亲；然后遍历链表，随意地选择合法位置（指那些要合并的两条簇都没有父亲的）进行 Rake
- 或 Compress，把结果作为两条簇的父亲；直到没有合法的位置为止。实现上，为了给每一个簇结点一个清晰的层次，可以给每一个孤儿建一个父亲，这个父亲没有其他孩子，且与孩子其维护的簇是相同的。然后，把所有父亲作为上一层的边使用。

Top tree 的构造

- 然后操作层数（Top Tree 的深度）就只有 O(logn) 了。这里的证明比较简单，分为三步：
- 读者自行验证：每棵树都有至少 12
- 的点度为 1 或 2 。 简单讨论一下，不难证明：每个 Rake/Compress 操作至多只会让两个相邻的原本合法的操作失效。 因此每层至少可以减少 1/6 的边。 事实上有一些反例违反了“每个 R/C 操作至多只会让两个相邻的原本合法的操作失效”，但我们可以证明，假设有 k 个反例，那么度为 1 或 2 的结点至少有 N/2+k 个。
- 然后这样我们就有一个 O(n) 的构建算法了，我们称作 Build 。在构建算法过程中，我们可以指定例外界点，以符合特殊需求。

非局部搜索

- Non-local Searching
- 即”线段树上二分”在 top tree 上的拓展
- 如果，存在一个判定程序 B:Cluster→Bool ，可以判断出“寻找目标”在这个 Cluster 的哪一侧，那么存在一个 B×O(logn) 的算法可以找到我们的“寻找目标”。这是线段树上二分的简单推广。注意这个 B 的输入是根节点（整棵树），因此实现并非平凡。

Sone1

- K=0 表示子树修改，后面 x,y，表示以 x 为根的子树的点权值改成 y
- K=1 表示换根，后面 x，表示把这棵树的根变成 x
- K=2 表示链修改，后面 x,y,z，表示把这棵树中 x-y 的路径上点权值改成 z
- K=3 表示子树询问 min，后面 x，表示以 x 为根的子树中点的权值 min
- K=4 表示子树询问 max，后面 x，表示以 x 为根的子树中点的权值 max
- K=5 表示子树加，后面 x,y，表示 x 为根的子树中点的权值 +y
- K=6 表示链加，后面 x,y,z，表示把这棵树中 x-y 的路径上点权值改成 +z
- K=7 表示链询问 min，后面 x,y，表示把这棵树中 x-y 的路径上点的 min
- K=8 表示链询问 max，后面 x,y，表示把这棵树中 x-y 的路径上点的 max
- K=9 表示换父亲，后面 x,y，表示把 x 的父亲换成 y，如果 y 在 x 子树里不操作。
- K=10 表示链询问 sum，后面 x,y,z，表示表示把这棵树中 x-y 的路径上点的 sum
- K=11 表示子树询问 sum，后面 x，表示以 x 为根的子树的点权 sum

Solution

- Top tree 维护路径信息：维护簇的路径点权信息
- Top tree 维护子树信息：维护簇的除簇路径外内点权信息
- 打标记：簇路径点标记，除簇路径外内点标记

Solution

- 每个簇维护以下东西：
- 除簇路径外的内点 点权和、最大值、最小值
- 簇路径点权和、最大值、最小值
- 除簇路径外的内点 点权标记，可以写成 $x \leftarrow kx+b$ 形式
- 簇路径点权标记，同理
- 在 Merge 中处理加和、取 max ，在 Split 中把标记扔到两个孩子簇上。

[ZJOI2015] 幻想乡战略游戏

- 带修改，维护带权重心

Solution

- 在 Top tree 上使用 Non-local search 的 trick
- O(logn)
- 具体怎么实现…这是论文里面的一个例题
- 但是我还没看完论文所以咕咕咕了

静态 top tree

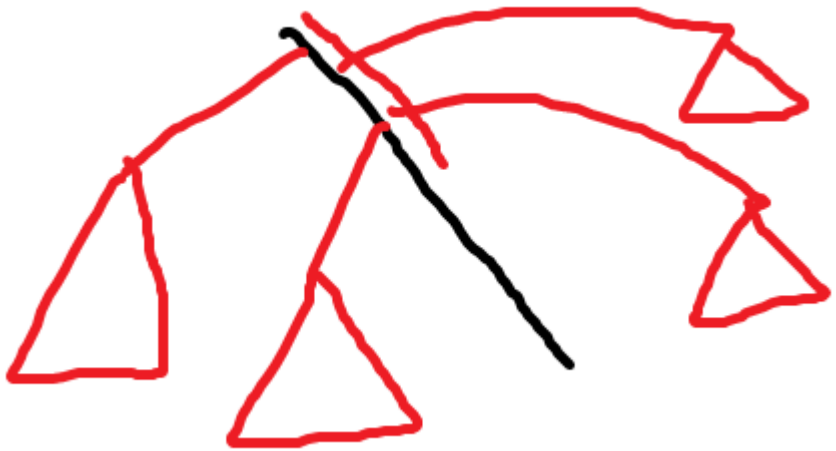
- 静态的 top tree 结构维护起来会方便很多，常数也会比 top tree 要好一些

基于 HLD 实现的静态 top tree

- 考虑静态 lct 维护子树和用平衡树维护每个点的轻儿子的静态 lct 的区别
- 发现区别就是后者进行了点度数的分治
- 实际上静态 lct 已经进行了两轮分治：
- 1. 对树进行分治——链分治
- 2. 对每条重链建一棵基于 biased dictionary problem 的 bst—— 对每条重链的分治
- 于是我们再在这个静态 lct 结构上进行一轮基于点度数的分治就可以让其功能达到静态 top tree 的功能

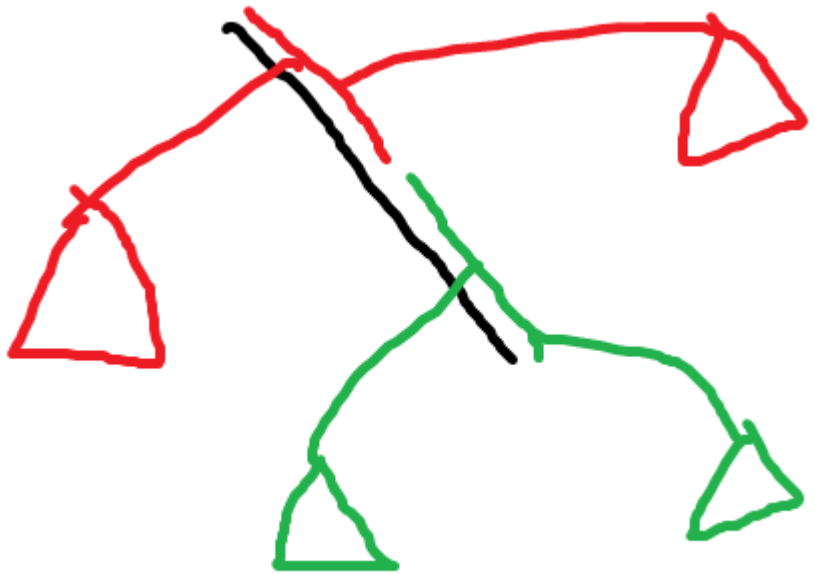
基于 HLD 实现的静态 top tree

- 定义 cluster 为所有满足有一个祖先为这个重链上连续的一段的点

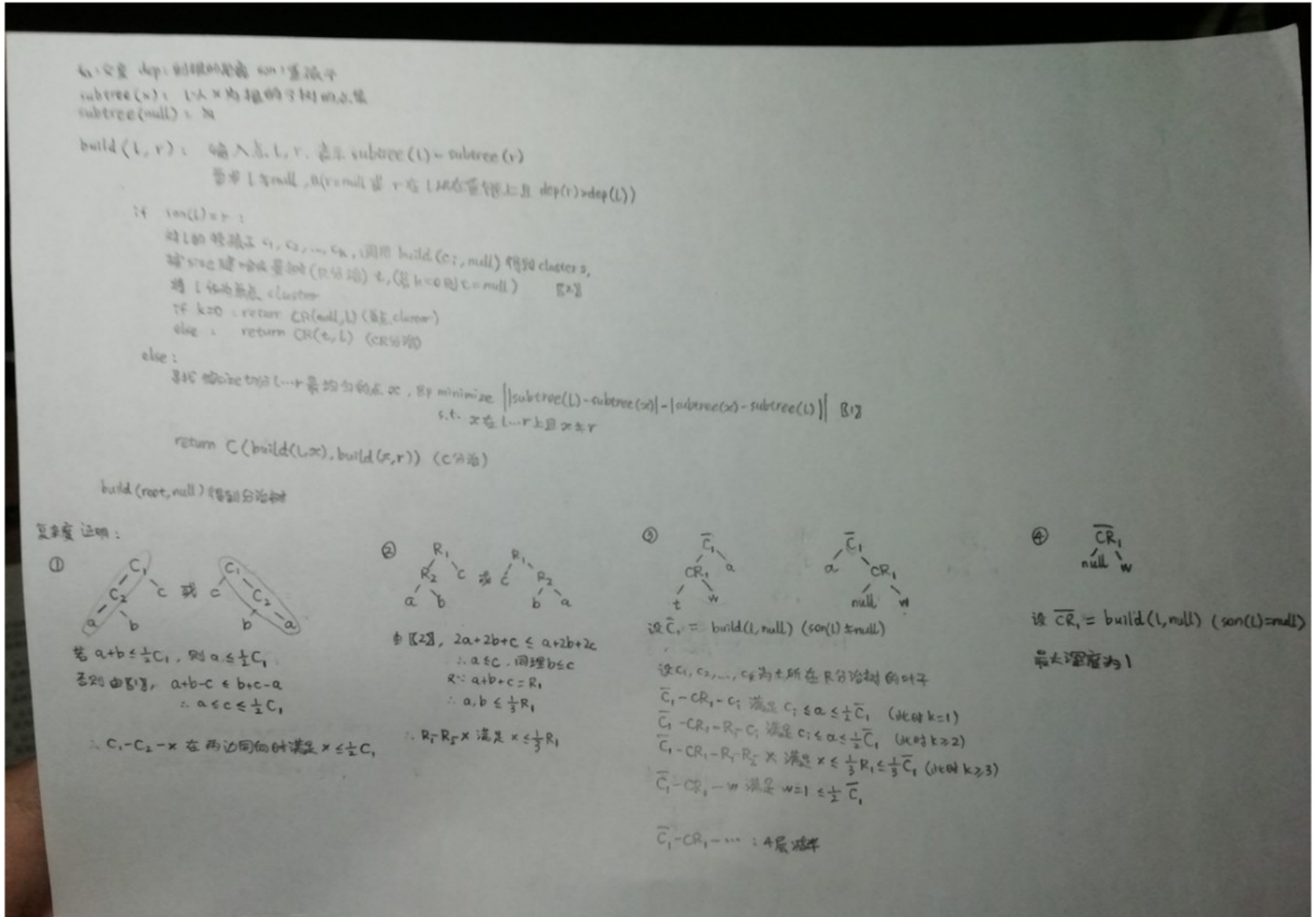


Rake&Compress

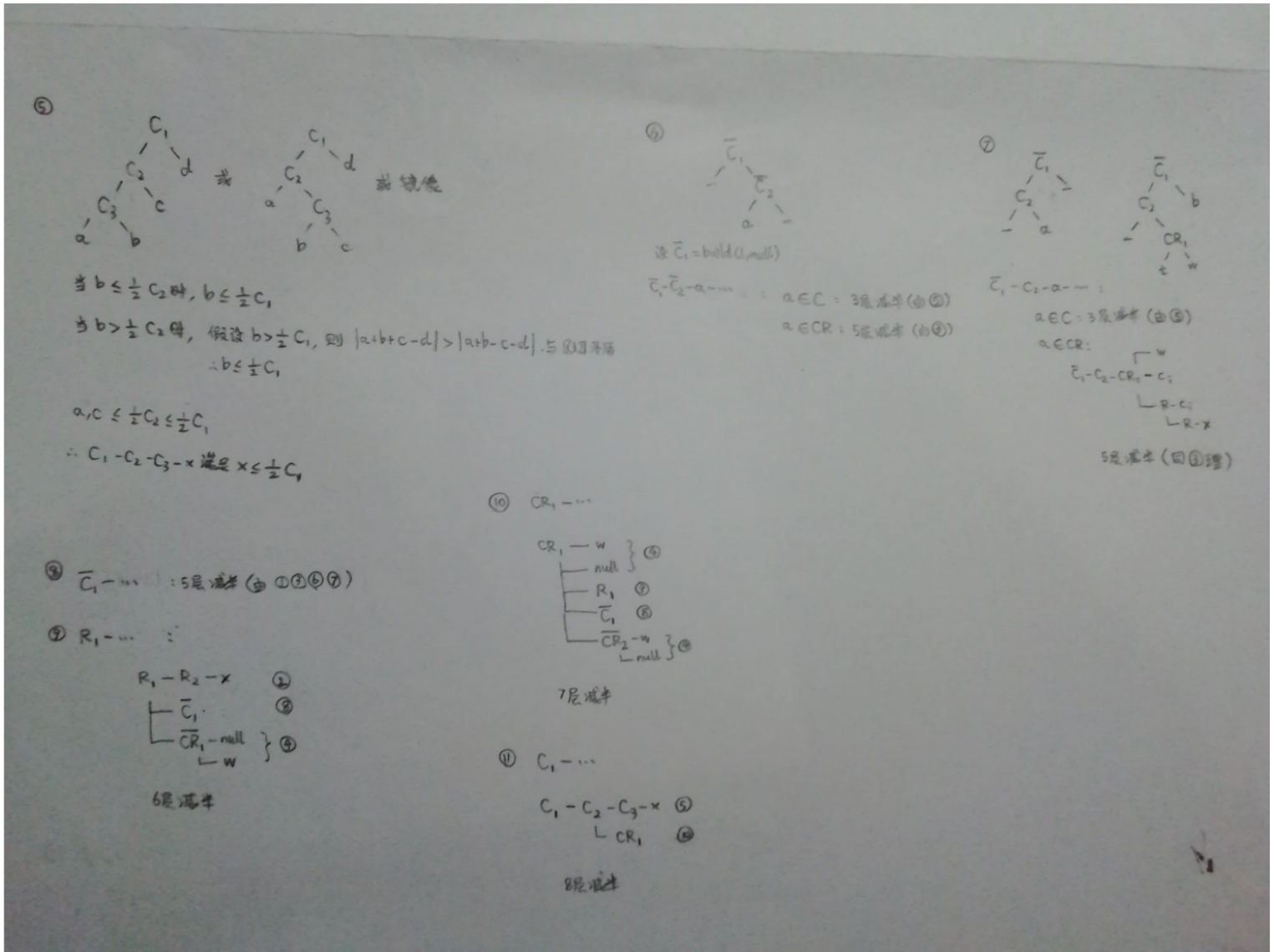
- Compress： 每次合并两段连续的重链的子链所对应的 cluster
- Rake： 每次在一个 cluster 中加入一个点



复杂度证明


subtree(x): 以x为根的子树的点集
subtree(null): ∅
build(l, r): 输入点 l, r, 表示 subtree(l) ~ subtree(r)
if son(l) ≠ r :
return C(build(l,x), build(x,r)) (C分治)
build(root, null) 得到分治树
复杂度证明:
①
若 a+b ≤ ½C1, 则 a ≤ ½C1
否则由[1], a+b-c ≤ b+c-a
∴ a ≤ c ≤ ½C1
∴ C1-C2-x 在两边同侧时满足 x ≤ ½C1
②
由[2], 2a+2b+c ≤ a+2b+2c
∴ a ≤ c, 同理 b ≤ c
又∵ a+b+c = R1
∴ a, b ≤ ⅓R1
∴ R1-R2-x 满足 x ≤ ⅓R1
③
设 C̄1 = build(l, null) (son(l) ≠ null)
设 c1, c2, ..., ck 为 t 所在 R 分治树的叶子
C̄1 - CR1 - ci 满足 ci ≤ a ≤ ½C̄1 (此时 k=1)
C̄1 - CR1 - R2 - ci 满足 ci ≤ a ≤ ½C̄1 (此时 k≥2)
C̄1 - CR1 - R2 - R3 - x 满足 x ≤ ⅓R1 ≤ ⅓C̄1 (此时 k≥3)
C̄1 - CR1 - w 满足 w=1 ≤ ½C̄1
C̄1 - CR1 - ... : 4层以上
④
设 C̄Rl = build(l, null) (son(l)=null)
最大深度为1

复杂度证明


⑤
或
或镜像
当 $b \le \frac{1}{2}C_2$ 时，$b \le \frac{1}{2}C_1$
当 $b > \frac{1}{2}C_2$ 时，假设 $b > \frac{1}{2}C_1$，则 $|a+b+c-d| > |a+b-c-d|$，与题目矛盾
$\therefore b \le \frac{1}{2}C_1$
$a, c \le \frac{1}{2}C_2 \le \frac{1}{2}C_1$
$\therefore C_1 - C_2 - C_3 - x$ 满足 $x \le \frac{1}{2}C_1$
⑥ $\overline{C_1}-\cdots$ ：5层减半（由①②⑤⑦）
⑦ $R_1-\cdots$ ：
$R_1 - R_2 - x$ ②
$\overline{C_1}$ ③
$\overline{CR_1}$ - null ④
w
6层减半
⑨
设 $\overline{C_1}$ = build(1, null)
$\overline{C_1}-\overline{C_2}-a-\cdots$ ：
$a \in C$：3层减半（由⑤）
$a \in CR$：5层减半（由⑦）
⑩ $CR_1-\cdots$
CR_1 — w ⑤
null
R_1 ⑦
$\overline{C_1}$ ⑥
$\overline{CR_2}$ - w ⑨
null
7层减半
⑪ $C_1-\cdots$
$C_1 - C_2 - C_3 - x$ ⑤
CR_1 ⑩
8层减半
⑦
$\overline{C_1}-C_2-a-\cdots$ ：
$a \in C$：3层减半（由⑤）
$a \in CR$：
w
$\overline{C_1}-C_2-CR_1-c_1$
R-c_1
R-x
5层减半（同⑥理）

Thanks for listening