# 从大象讲起——说说各种数据结构。

范浩强

# 自我介绍？

- 略吧。

# 先从大象说起。

- 先不着急说数据结构的事。
- 先从大象那天说起。
- 23rd IOI Day2

# 和动物在一起的一天。

- 鳄鱼 crocodile
- 大象 elephant
- 鹦鹉 parrot

# 最‘‘简单’’的鹦鹉。

- 为什么说是最‘‘简单’’呢？因为用到的数据结构只有数组。
- 为什么要给‘‘简单’’加引号呢？因为它并不简单。

# 鹦鹉。

- 你有一群鹦鹉。每个鹦鹉可以记住 8 个二进制位，即，一个 0 ~ 255 之间的自然数。
- 你告诉了一些鹦鹉一些数，之后，让它们飞到另一个人那里。每个鹦鹉忠实地告诉了那个人（接收者）它记住的数是多少。
- 你想通过这个方法传递一个消息。但是，鹦鹉有个小问题，就是鹦鹉到达的顺序是随机的。

# 鹦鹉。

- 例如：
- 你发送了 3 只鹦鹉：
- 1 9 9 8
- 到达接收者那里很可能变成了：
- 9 8 1 9
- 你要在这种状况下，通过发送最少个数的鹦鹉来传递消息。消息可以视作一个 N 位的二进制数。

# 鹦鹉？

- 这题出得不错。。。但是，怎么做呢？
- 主要矛盾：乱序到达？
-
- 想法 1：每个鹦鹉记住自己是第几个（发送位置码）。
- 每个鹦鹉记住一个数 4x+y，其中 y 表示一个 2 位的消息 (0/1/2/3)， x 表示这个消息在原文中的位置。

# 鹦鹉。

- 0 1 3 1
- => 0 5 11 13
- 到达接收者那里之后
- 5 13 0 11
- 接收者把它们从小到大排序
- 0 5 11 13
- 之后分别模 4
- 0 1 3 1，很神奇吧。。。

# 嗯，很和谐。

- 通过这种方法最长能发送多长的消息呢？
- X 的取值是 [0,64)，能最多标记 64 个位置，每个位置 2 位，所以最长 128 位，即 16 字节。
- 如果要发送 N 个字节（ 8N 位）的消息，要用 4N 个鹦鹉。
- 这么做有多少分？
- 子任务 1 ， 2 ：很水，一共 32 分。
- 子任务 3 （ 18 分）
- N<=16
- 鹦鹉数不超过 10N
- 哈哈，一共 50 分到手了！

# 鹦鹉？。

- 子任务 4（29 分）
- 1<=N<=32
- 最多发送 10N 只鹦鹉。
- 这个怎么办？
- 如果延续上面的思路，能否不用 6 个位来标识位置，而是用使用更多 / 更少的位来编码位置信息？

# 但是。

- 计算一下各种情况下发送的最大长度。
- 0 个位置码，1*8=8
- 1 个位置码，2*7=14
- 2 个位置码，4*6=24
- 3 个位置码，8*5=40
- 4 个位置码，16*4=64
- 5 个位置码，32*3=96
- 6 个位置码，64*2=128
- 7 个位置码，128*1=128（囧了！）

# 鹦鹉！。

- 于是，要另辟蹊径。
- 重新想想，接收者得到的信息的本质是什么？
- 2 1 3 3 0
- 接收者：我得到了 1 只说“ 0” 的鹦鹉， 1 只说“ 1” 的鹦鹉， 1 只说“ 2” 的鹦鹉， 2 只说“ 3” 的鹦鹉，没有其他的鹦鹉了。
- ->1 1 1 2 0 0 0...<repeat 252 times>...0
- 发送的信息的本质是一个 unsigned int[256] ！
- 使用的鹦鹉数 = 数组里的元素之和

# 啊哈！

- 我有想法了。
- 把 0~255 视作 256 个频道。派出一个说 x 的鹦鹉 -> 在频道 x 上发送 1 。
- 'ac' (01100001011000011)
- ->0 1 5 6 8 13 14
- 这样，不就可以发送 256 位的信息了？最多使用 256 只鹦鹉！

# 相当和谐。

- 一共 79 分到手了。
- 还有哪里可以改进呢？
-
-
- 我干嘛在 1 个频道上只发送 1 只鹦鹉呢？

# 相当和谐。

- 一共 79 分到手了。
- 还有哪里可以改进呢？
-
-
- 我干嘛在 1 个频道上只发送 1 只鹦鹉呢？

# 鹦鹉！。

- 想象：在 3 个频道上发送 2 只鹦鹉，一共有 10 种方法。
- 0 0 0
- 1 0 0
- 0 1 0
- 0 0 1
- 2 0 0
- 0 2 0
- 0 0 2
- 1 1 0
- 1 0 1
- 0 1 1
-

# 鹦鹉。

- 每组频道可以表示一个 10 进制位，我一共可以用最多 170 只鹦鹉发送 85 组频道，即 85 个十进制位，即 282 个 2 进制位。
-
- 很好！不光鹦鹉用得少，消息也传得长了。

# 再接再厉！。

- 4 个频道一组，每组发送最多 7 只鹦鹉
- 一组有 330 种方法，可以用来编码一个字节。
- 一共可以发送 64 字节，使用鹦鹉数是 7N
- 
- 子任务 4：
- N<=64

| P= 鹦鹉数 /N | 分数 |
| --- | --- |
| 5 | 19 |
| 6 | 18 |
| 7 | 17( 哈哈，98 分了 ) |
| 15 | 31-2P |

# 剩下 2 分怎么办？

- 8 个频道一组，发送 11 只鹦鹉，一共 75582 种方法，编码 16 个位，鹦鹉数 /N＝5.5 。可以用 bfs 来产生各种方法。
- 如何满分？
- 16 个频道一组，发送 20 只鹦鹉，一共 7307872110 种方法，可以编码 32 位，鹦鹉数 /N＝5
- 但是，只能用动态规划来编码，没法 bfs ，来不及写了。。。
- 这样，我就得了 99 分。。。

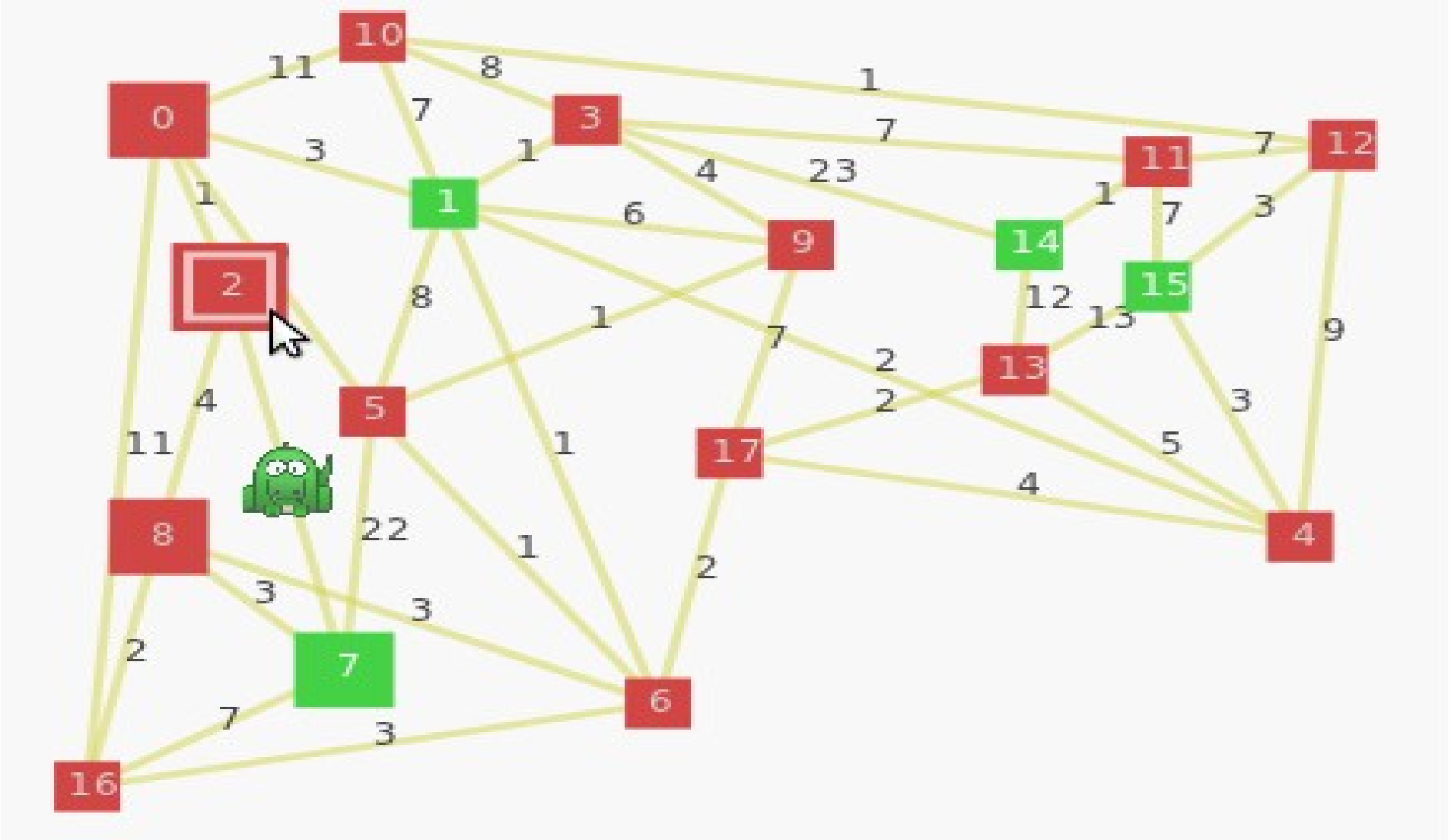
# 最牛可以做到多少？

- 在 256 个频道上发送 261 只鹦鹉，方法数是
- 1468967922881709002768861963936950325459008154875039435044134579530363844059555156904717150263042177073817939138714442648118923394773509148500010204596960 6
- 可以编码 64 个字节
- 鹦鹉数 /N=4.078125
- 嗯，很好，很好。。。

# 啊，鹦鹉可算做完了。

- 鳄鱼？
-
- 一只阻拦你出去的鳄鱼，每次会智能地挡在你的一条出路上，让你到达出口的时间最长。

# 鳄鱼

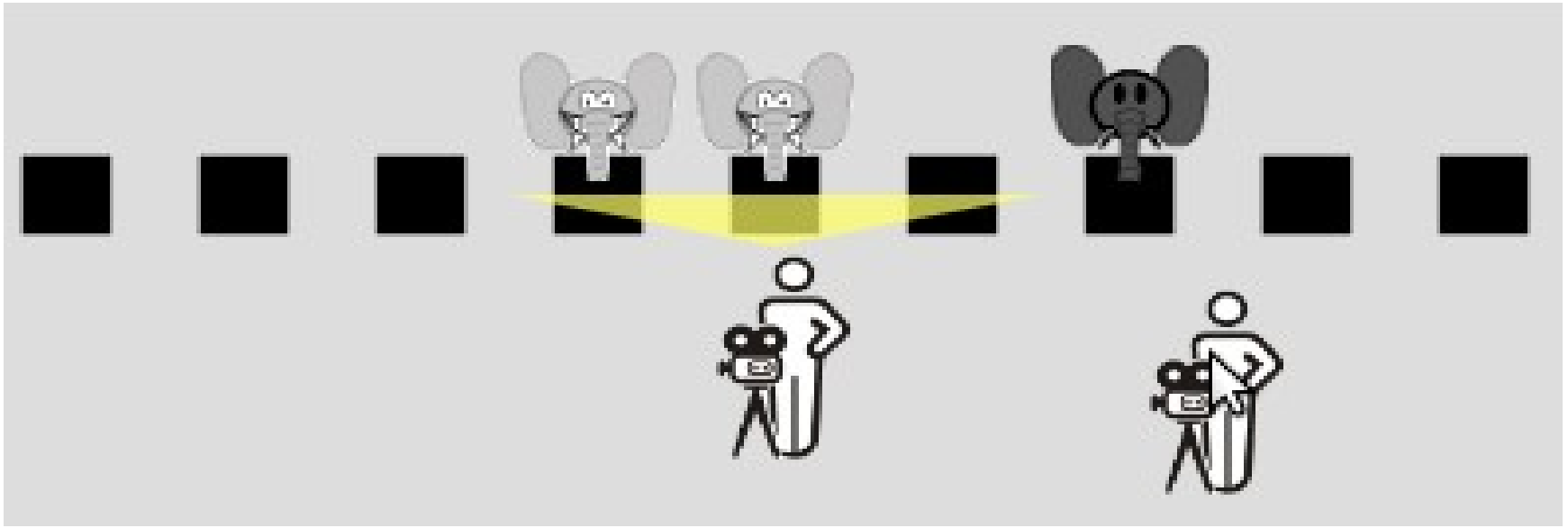


# 嗯，这题是不是我们已经见过了？。

- 前一阵子的集训。。。
- 老虎的故事。。。
-
- 解法简述：
- 扩展的 Dijkstra，每个点记录最近的两条边，每次挑一个第 2 短路最短的进行扩展。
-
- 程序很短。。。（感谢 STL 的 priority_queue ）

# 主角登场了：大象！

- 你见过大象跳舞吗？（我在泰国见过。。。）
- 你见过 N 只大象在数轴上跳舞，有若干范围为 L 的摄像机拍摄它们吗？（这个真没有。。。）
- 你见过还有人想知道最少用多少个摄像机来覆盖所有的大象的吗？（只有题里有这种人。。。）

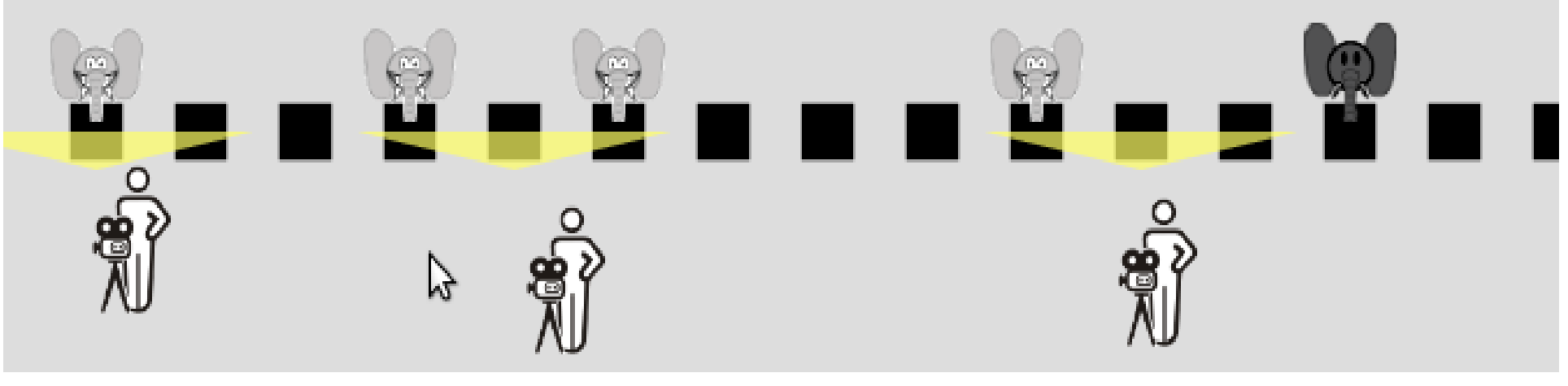


# 吓一跳。

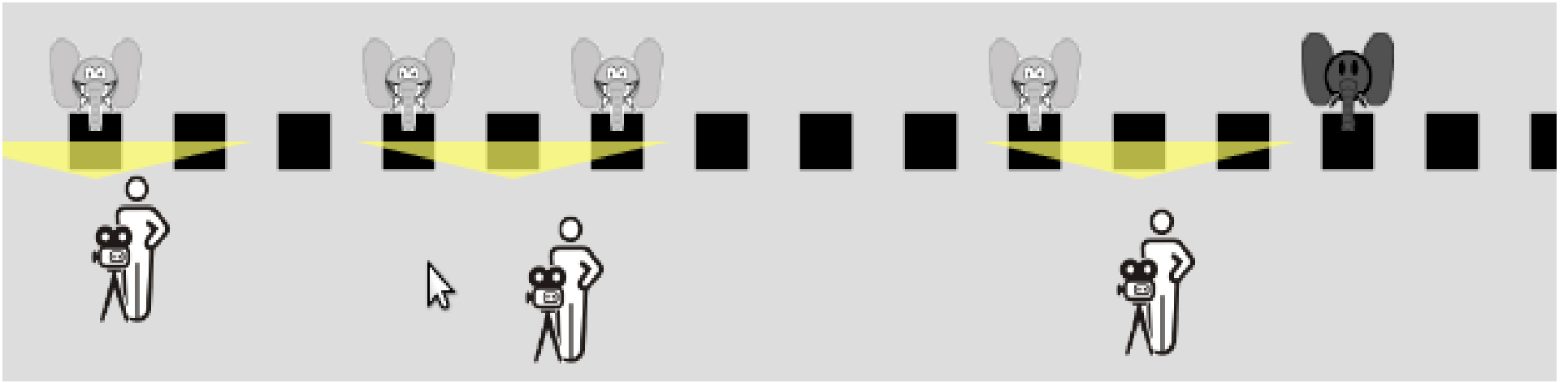
- 大象数有多少？
- 最多 15 万。
- 数轴有多长？
- 最长 10 亿（大象在整点上）。
-
- 如果真有那么多大象在一起，这将是一场灾难。。。

# 灾难？在后面。

- 可怕的是，这些大象还在移动！每个时刻，一个大象从一个位置 x 跑到位置 y 。
- 更可怕的是，有一个人，想知道：
- **每个时刻**，最少用多少个摄像机来覆盖所有的大象？



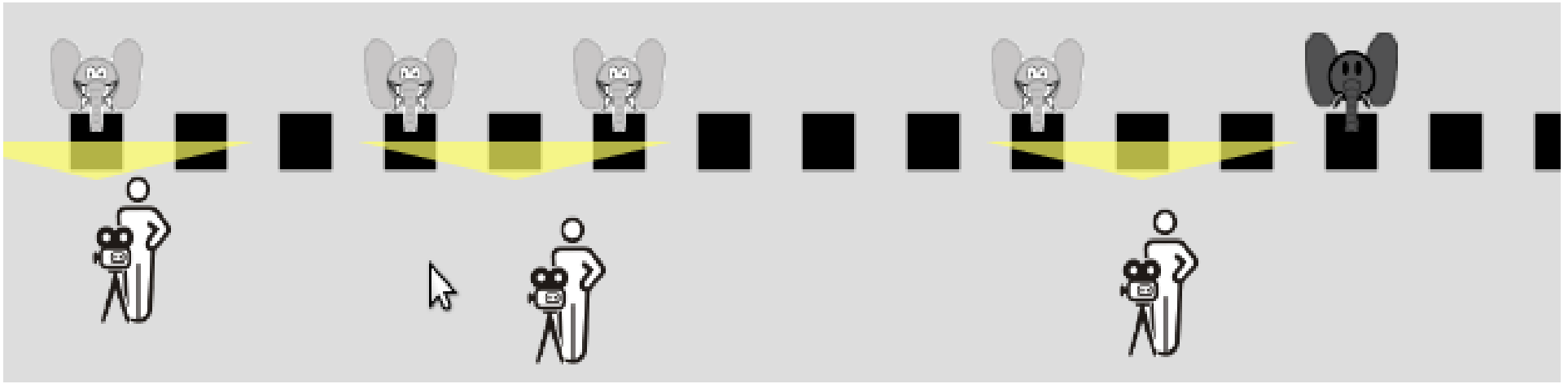
# 冷静一下



- 让我们先在那些可怕的大象面前做一个深呼吸。
- 如果大象不动，能否快速求出最小的覆盖数？
- 
- 啥？动态规划？

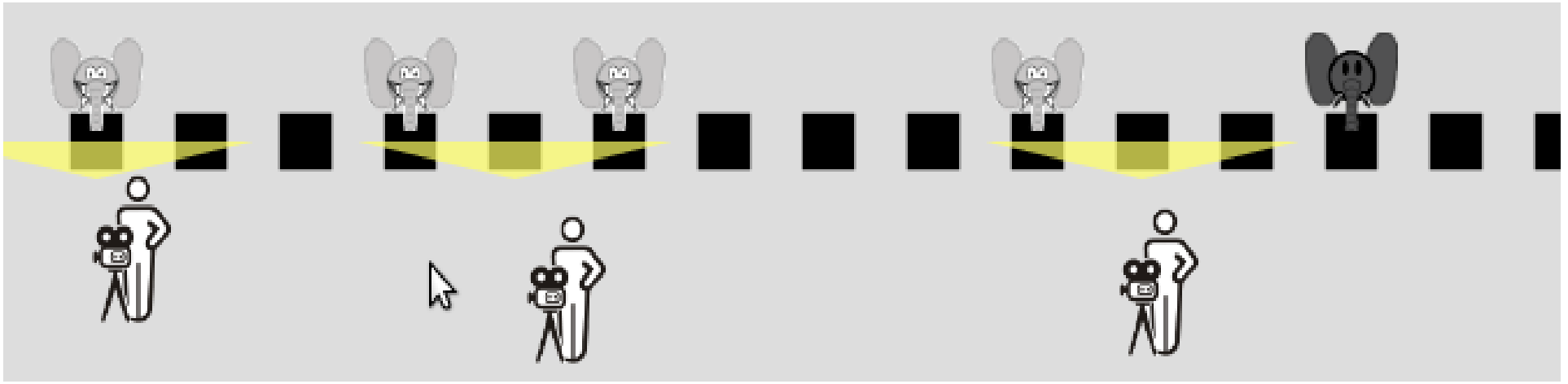
# 动态规划？。。。

- 先把大象从左到右排序。
- 设，前 i 个大象最少用 f[i] 个摄像机覆盖。考察第 i 个大象的摄像机覆盖到哪里，有
- f[i]=min(f[j]，大象 i 到大象 j 的距离不超过 L)+1，



# dp 看起来挺傻的。

- 为什么非要 dp 呢？
- 通过 dp 方程的分析，我们似乎得到了这样一个结论：可以贪心！
- 每次，找最靠左的没有被覆盖的大象，以它为左端点架一个摄像机。



# 代码很简洁啊。

- #A= 大象们的位置
- s,x=0,-1
- for i in sorted(A):
- 　if i>x:
- 　　s,x=s+1,i+L
- return s

# 时间复杂度？

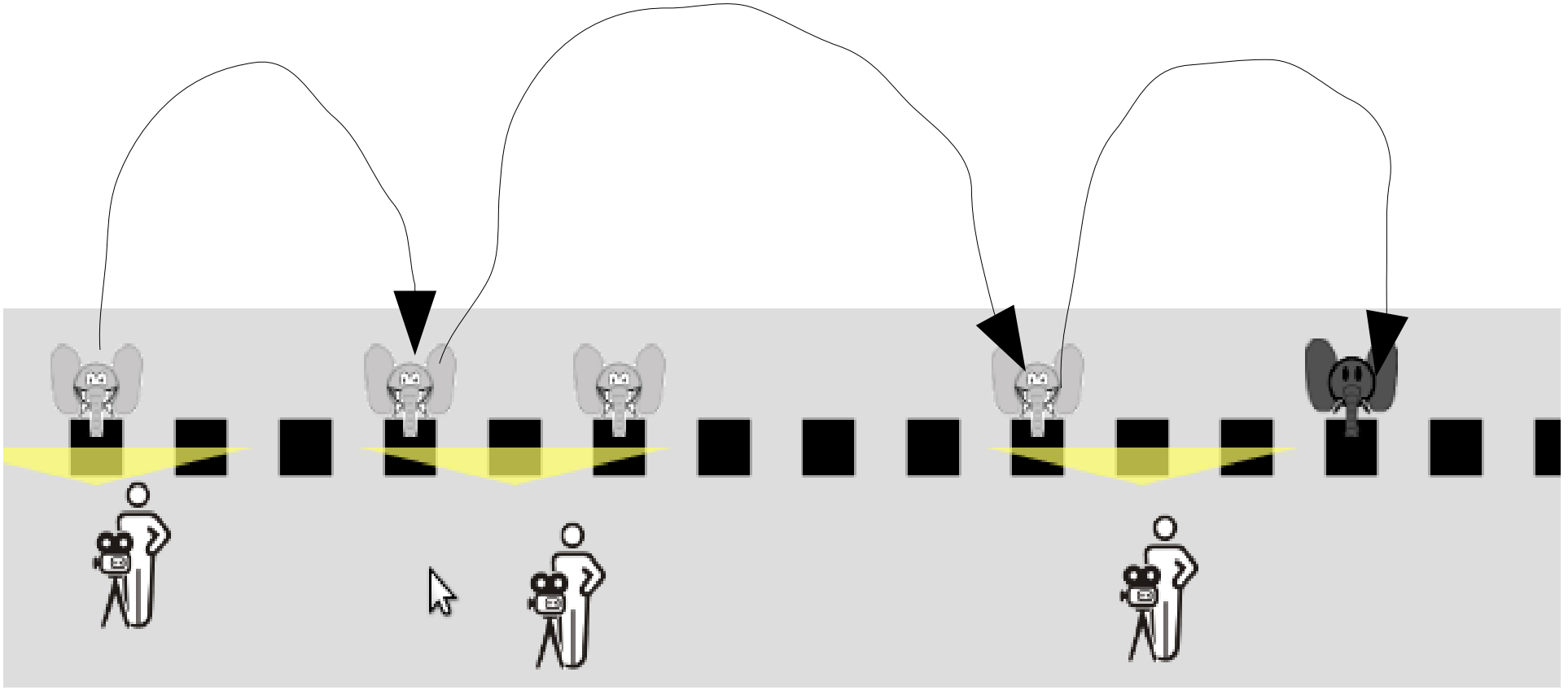
- 不算排序，时间复杂度是 O(N)，常数很小。
- 如果每次移动都排序一遍，那么时间复杂度是 O(NlogN*M)， M 是移动次数
- 很显然，我们可以第一次的时候排序，以后每次移动的时候冒泡一下。
- 时间复杂度是 O(NlogN+NM)
- 嗯，有多少分呢？

# 26分。可怜啊。。。

| | N | M |
|---|---|---|
| Task 1（10分） | 2 | 100 |
| Task 2（16分） | 100 | 100 |
| Task 3（24分） | 50000 | 50000 |
| Task 4（47分） | 70000 | 70000 |
| Task 5（3分，这。。。） | 150000 | 150000 |

# 加油！！！

- 再把问题转化一下。
- 想想这段代码的意思：
- for i in sorted(A):
- 　if i>x:
- 　　s,x=s+1,i+L
- 实际上，我们做的工作是：从最左边的大象开始，每次往右跳到 L 的距离以外最左边的那只大象。答案就是这个链的长度。



# 可以这么搞吗？

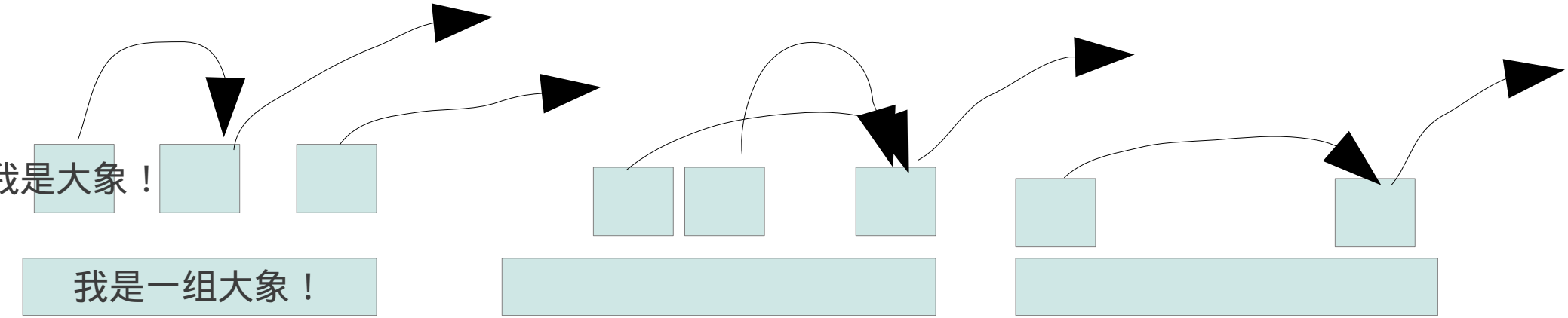
- multiset<int> A= 大象们
- int s=0;
- for (multiset<int>::iterator i=A.begin();i!=A.end();){
- s++;
- i=A.upper_bound(*i+L);
- }
- 很合理，尤其是当 L 很大的时候。
- 但是，不解决本质问题。 L 很小怎么办？

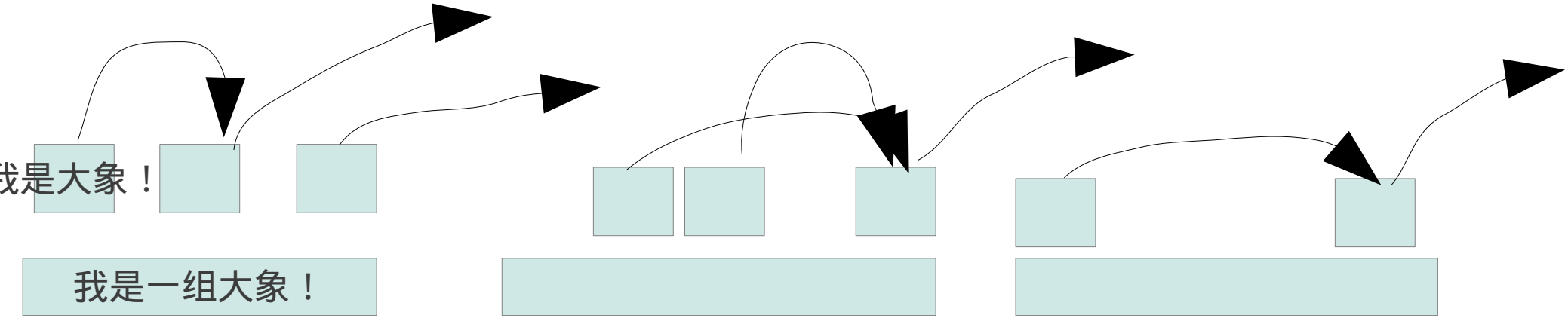
# 大象？。

- 我们能否维护每个大象的“后继”呢？即，每个大象后面距离超过 $\llcorner$ 的最近的大象。
- 似乎可以。
- 还可以在每个大象上记录它打头的链的长度。
-
- 大象移动了，就去更新它涉及的链。
- $\llcorner$ 很小的时候**似乎**可行。
- 但是，链的长度真的很难维护。

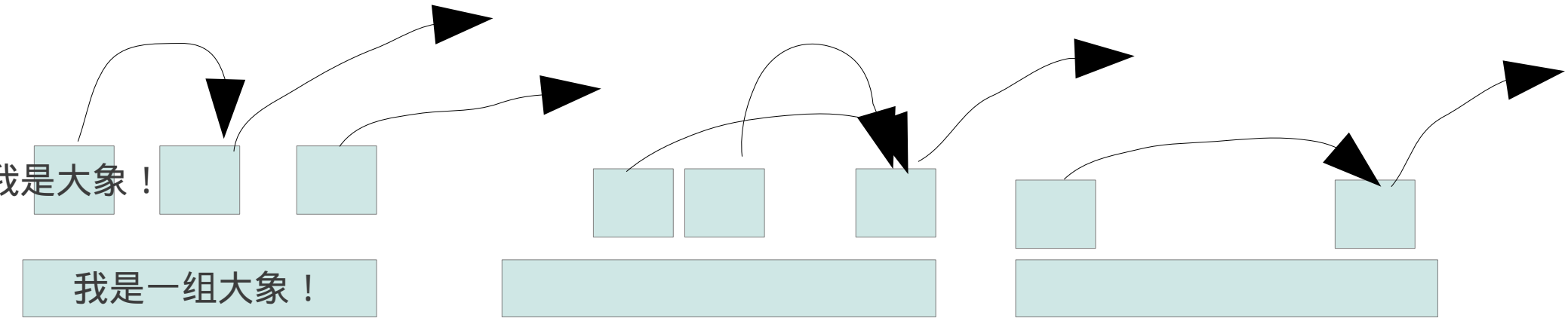
# 矛盾啊。

- 修改：后继关系：⌊小点好。
- 查询：从最左边一直往后跳：⌊大点好。
-
- 于是，我们有超级武器：分块！
-
- 把连续的 sqrt(N) 个大象分成一组。

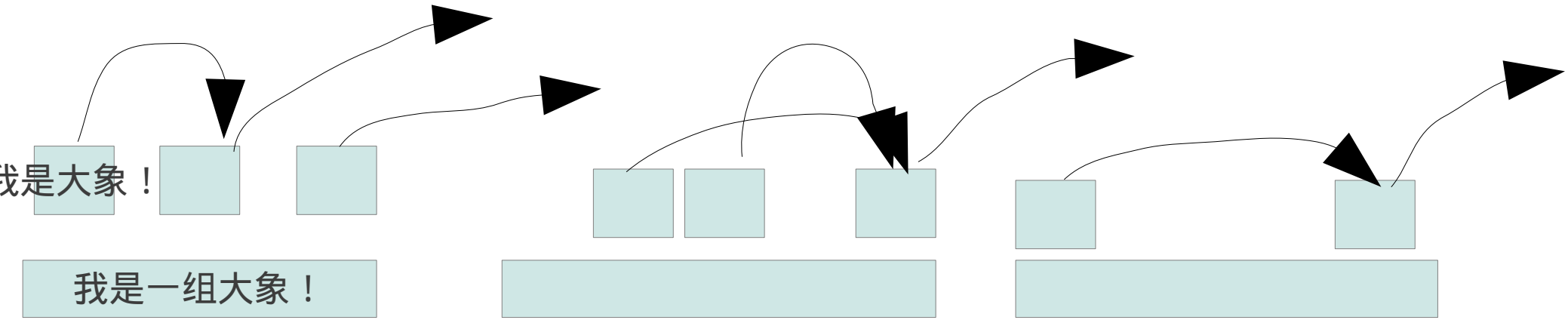

我是大象！
我是一组大象！



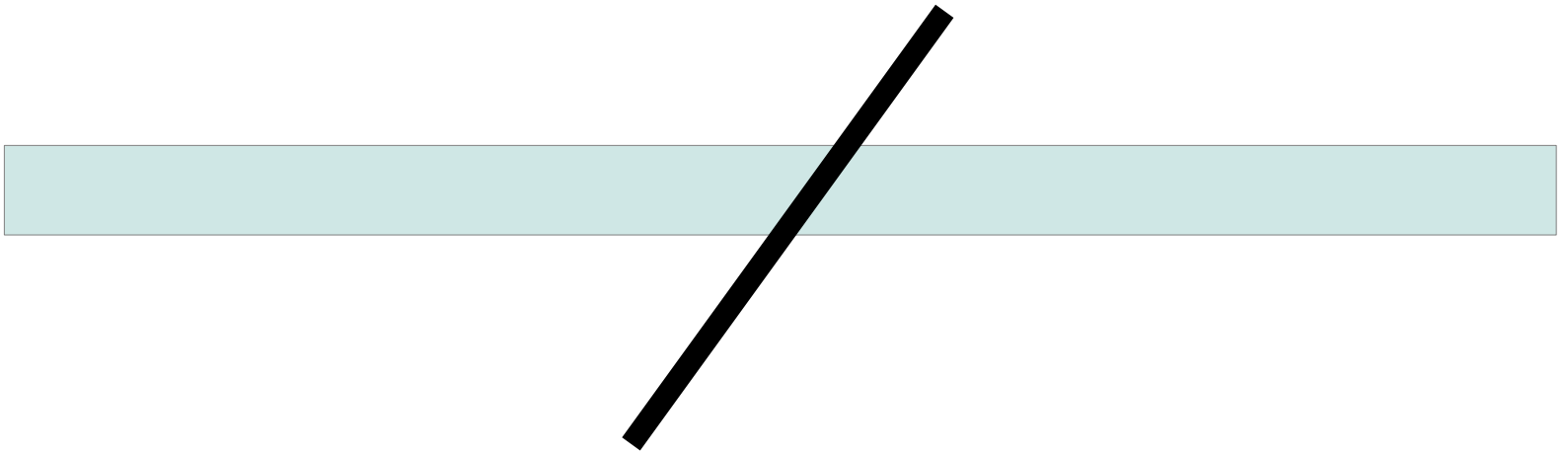

我们要维护哪些量呢？。
每个大象的位置。
每个大象在组内的链的长度。
每个大象在组内的链的末尾。




查询是怎么回事？。
从最左边的组的最左边的大象开始。
在组内，利用记录的信息直接跳到组的末尾。
到下一个组，通过二分查找来计算后继，之后接着“跳”。
记录路过的大象的个数 = 答案。
时间复杂度 =sqrt(N)logN




修改是怎么回事？
修改 = 插入 + 删除
找到大象所在的组。
用线性的时间更新每个大象的后继，
计算链长、链尾。
问题是，组的大小不再是 sqrt(N) 了！
怎么办？



如果一个组的大小超过 2sqrt(N)-> 分裂成两个。

如果相邻两个组的大小和不超过 sqrt(N)-> 合并成一个。

可以证明，每个组的大小、组数都是 O(sqrt(N))，而每 sqrt(N) 次操作才会引发一次合并 / 分裂。一次合并、分裂的时间复杂度是 O(sqrt(N))，均摊下来就是 O(1)。

# 能不能不写分裂 / 合并？。

- 可以啊！
-
- 每 sqrt(N) 次操作之后，销毁整个数据结构，重新构建一次！
-
- 均摊 ->O(sqrt(N))

# 总的时间复杂度

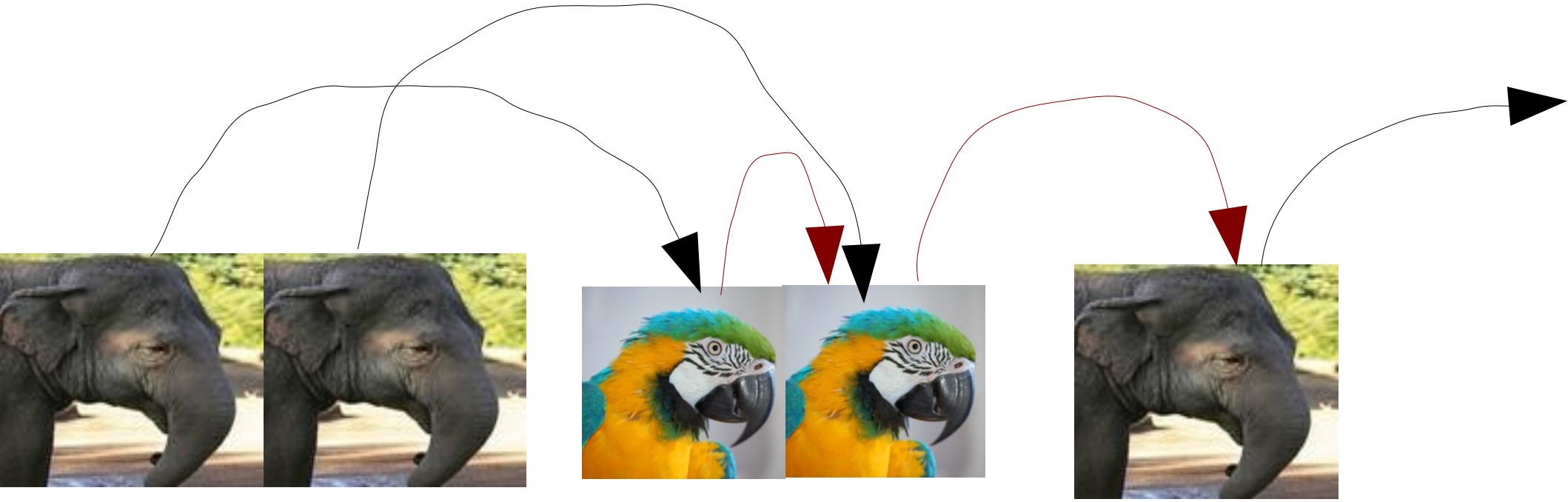
- 查找： O(logNsqrt(N))
- 更新： O(sqrt(N))
- 嗯，其实可以每 sqrt(NlogN) 个大象一段，这样时间复杂度可以更好。。。
-
- 这就满分啦。
-
- 但，这就结束了吗？

# 一个想法。

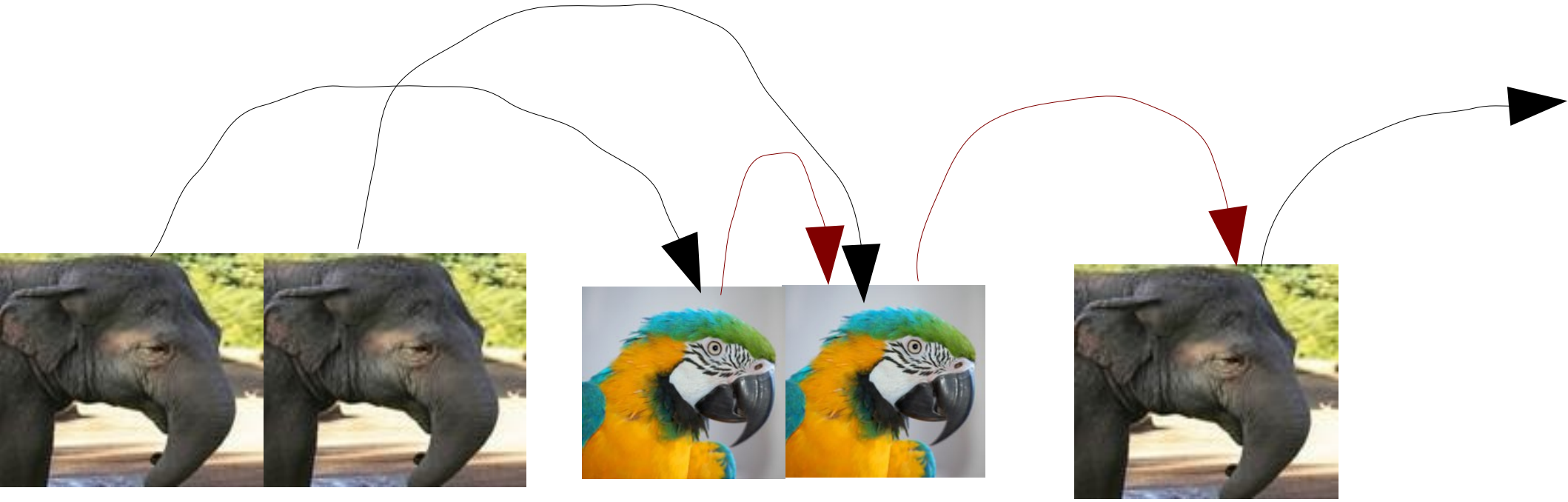
- 我们能否做到 O(logN) 呢？
-
- 这是一个很自然的想法。

# 有了！。

- 我们要维护每个点往后 $L$ 个距离的后继。
- $L$ 的距离，后继，这两个关系放在一起就费劲了。
- 能否转化成要么是 $L$ 的距离，要么是后继？
-
- 要请鹦鹉来帮忙啦！



我在每个大象后面 L 的位置设置一个鹦鹉。
每个大象连一条边到它对应的鹦鹉。
每只鹦鹉连一条边到它后面紧跟着的动物（可以是鹦鹉，也可以是大象）。
最小摄像机数 = 最左边的大象所在的链内大象的个数。



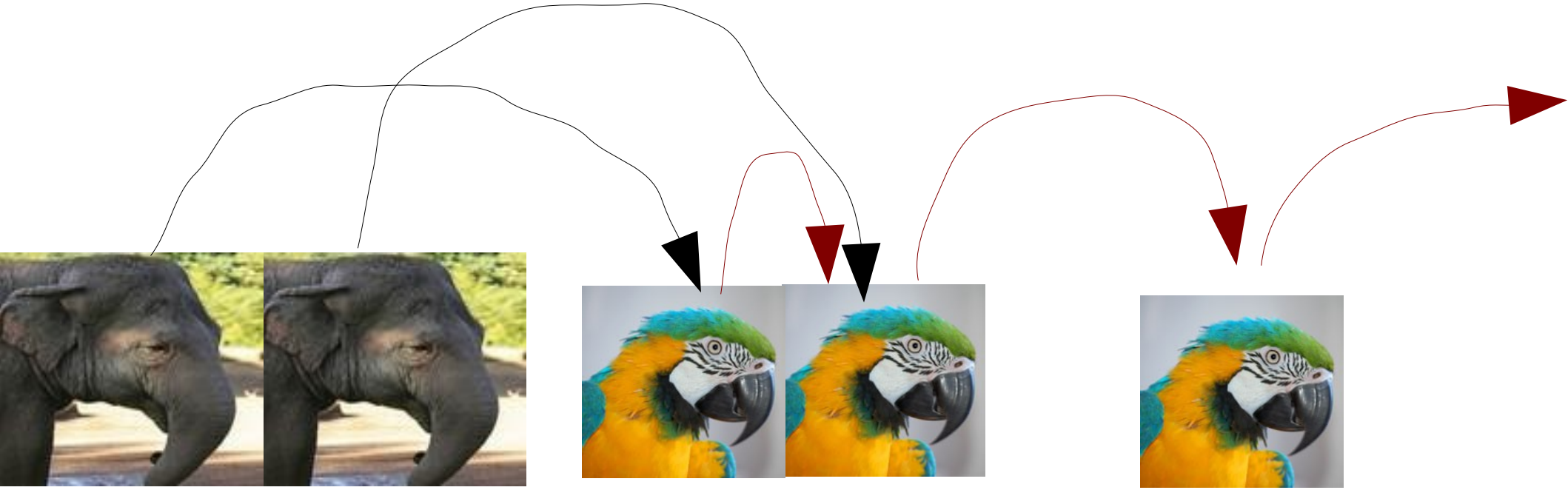
我在每个大象后面 L 的位置设置一个鹦鹉。
每个大象连一条边到它对应的鹦鹉。
每只鹦鹉连一条边到它后面紧跟着的动物（可以是鹦鹉，也可以是大象）。
最小摄像机数 = 最左边的大象所在的链内大象的个数。

# 一个 2N 个点，2N-1 条边的图

- 树！。
- 没错，这些动物形成了一个树。还是一个有根树，从左边指向右边。
- 每次查询，我们的工作：
- 询问从一个点到树根的路径上大象的个数。
- 这。。。好熟悉的模型。

# 先说说修改怎么办

- 修改 = 插入 + 删除
- 插入：
- 同时插入一对鹦鹉和大象。用一个 set 来维护所有的动物的位置。设置大象的后继为鹦鹉，利用 lower_bound 查找鹦鹉的后继，同时，看看它是否修改了其他鹦鹉的后继。
- 删除：
- 把一个大象变成鹦鹉即可！



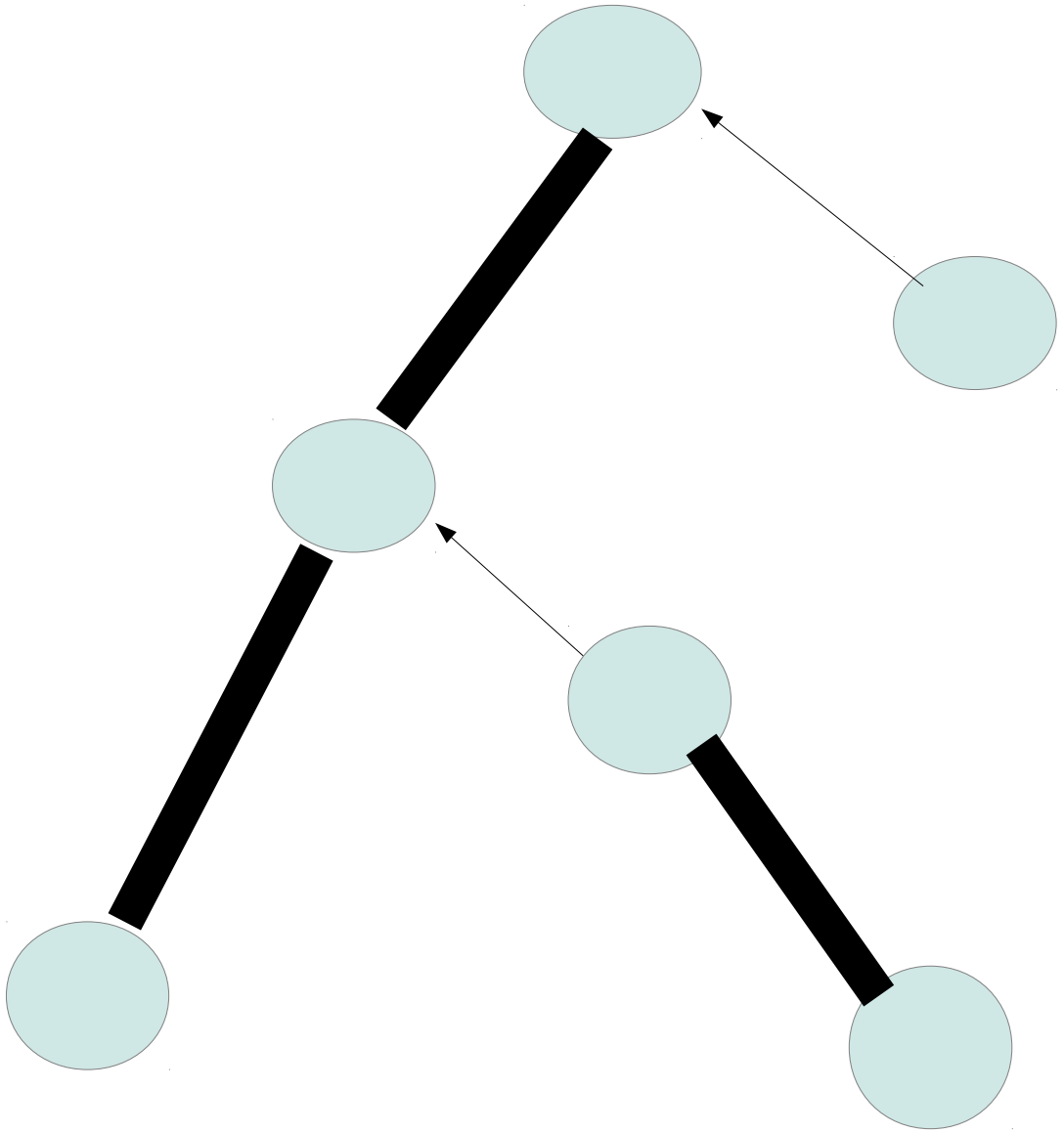
多来几只鹦鹉并无大碍！。

# 树啊。

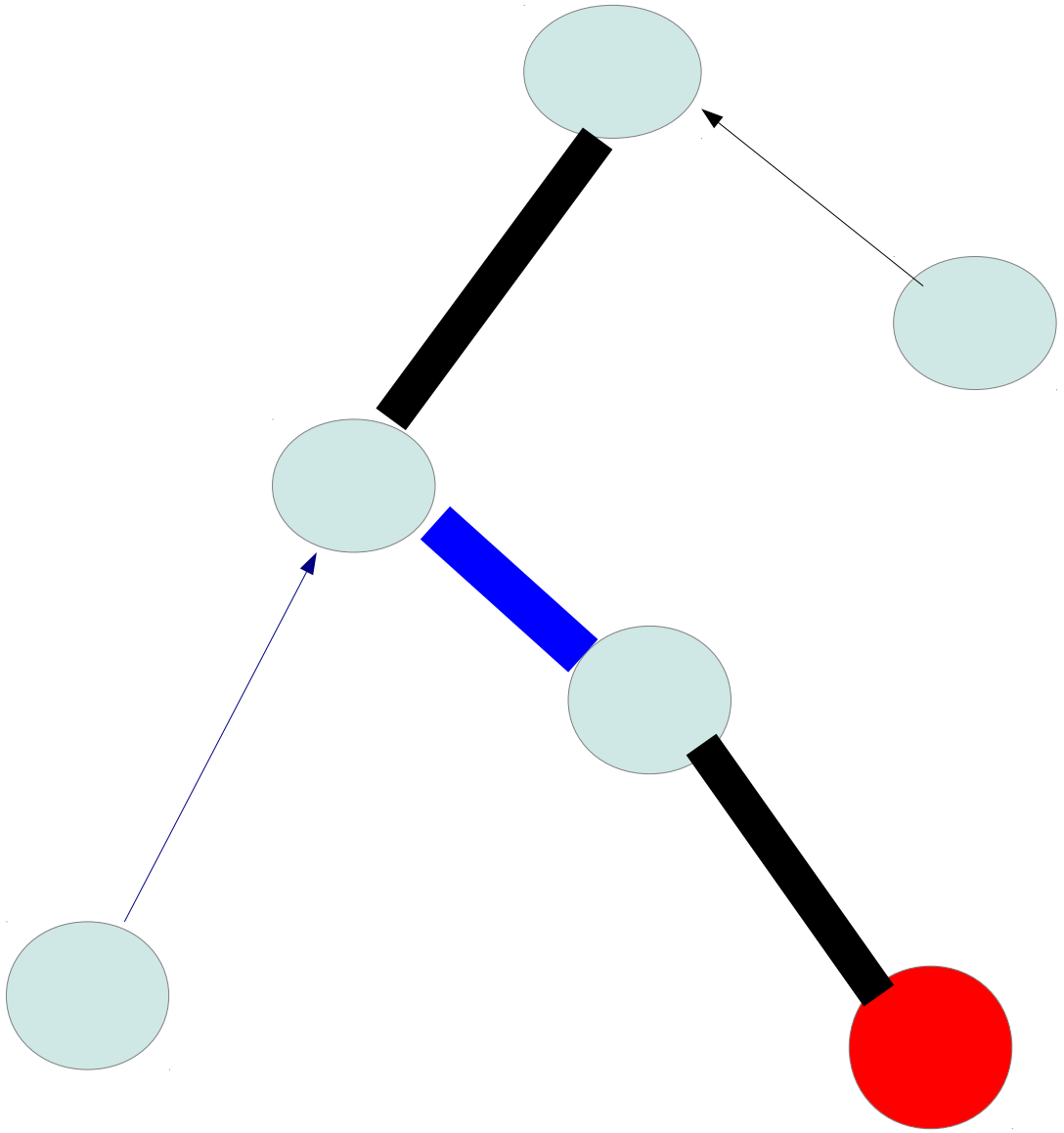
- 问题已经长到树上了。
-
- 维护一个树，支持以下操作（ O(logN) 每次）。
- 插入一个点。
- 修改一个点的父亲。
- 修改一个点上的权值。
- 查询一个点到根的路径上的权值和。

# 这好像叫：动态树！

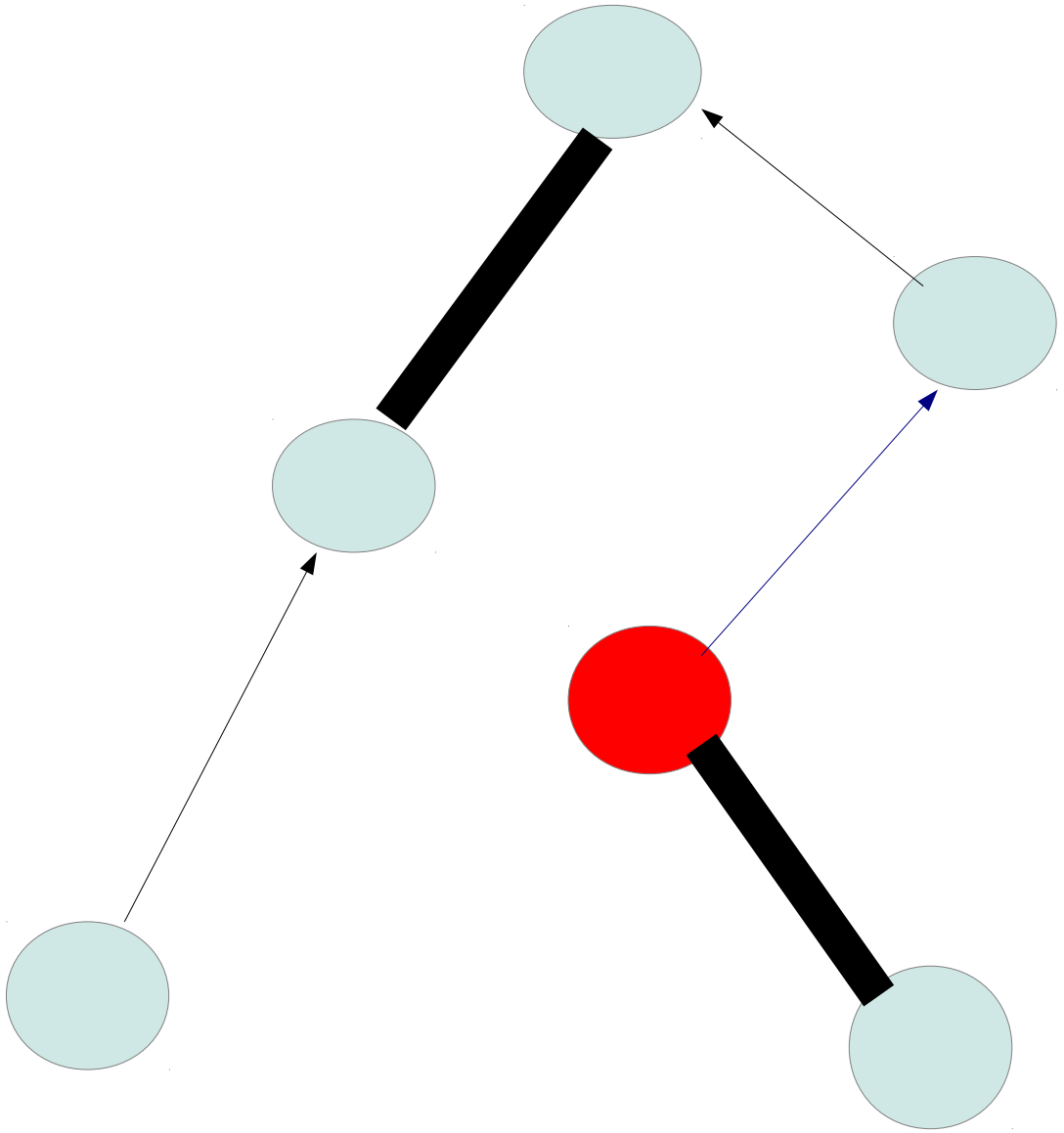
- 啥是动态树？定义请自己 google/baidu/bing
-
- 有一种动态树的实现，叫 link-cut tree 。
-
- 似乎以前已经有人讲过了吧？



核心思想：
把树剖分成若干个链。链与链之间通过父子关系相连。
每个链用一个数据结构来维护，同时记录链上点的权值和。



查询一个点：
把这个点到根的路径用一条链连接起来，在链上查询（途中要进行链的拆分、连接）。
修改一个点的权值：
同理。



修改一个点的父亲：
把它所在的链断开，
之后设置新的父亲。

# 链？时间复杂度？。

- 用什么数据结构来维护链呢？。
- **不妨**用 splay 吧。
- 时间复杂度呢？
- 有的论文说，是 O(logN)。
-
- 啥？你说这篇论文是错的？不能用 splay 来维护？
- 那你再换篇论文看看。
- 不管是不是真的 O(logN)，反正我用了。

# 嗯，要开始说数据结构了！

- 啥叫数据结构呢？
- 下定义是一个痛苦的事情。
-
- 我们生活在数据结构的世界里。
- 随机存取线性表（数组，vector）
- 优先队列（ priority_queue ）
- 有序集合（各种平衡树， set ）
- 映射（ map， hashmap ）

# 矛盾。

- 信息学中充满了矛盾。
- 语言：C++——Pascal
- 算法：Dijkstra——BellmanFord
- Dinic——SAP
- 数据结构：Treap——Splay
- 自顶向下线段树——自底向上线段树
- 甲：我写的比你短！乙：我写的比你更短！甲：我写的比你还短！乙：我常数小！甲：我常数更小……

# 动态树：新兴的阵地。

- 一般来说，动态树维护一个树，动态地支持几种操作：
- 修改一个点的信息
- 查询一个点的父亲 / 查询一个树的根
- 查询两点间路径上的信息 / 查询子树上的信息
- 修改一个点的父亲 / 把一个点设成根

# 都有什么题呢？。

- ‘‘ 裸’’动态树。
- 查询两点间最小值。
- 查询两个间最……值，……和，……的个数。
- 查询一个可以看作树的图上的最……路径。
-
- 等等等。

# 有没有不裸的？。

- 嗯，有啊！
- 例如，大象就是一个极好的例子。
- 不过，需要进行模型的转化，把树给挖出来。
-
- 还有吗？

# 动态树作为一个工具。

- 可以用来优化网络流 / 费用流算法。
- 不过，别想在考试的时间里写出来。。。
-
- 可以作为其他数据结构的一部分。

# 动态图联通性？。

- 给一个无向图，动态添加 / 删除边，问两个点之间的联通性。
- 很复杂。。。最终的结论是，可以 $O(\log^2 N)$ 维护。
- 核心思想是把图拆成 logN 层，每层分别是大小不同的动态树组成的森林，层级越高森林里的树越大，查询 / 修改都是分层进行。
- 几乎不具备可写性。

# 其他的呢？。

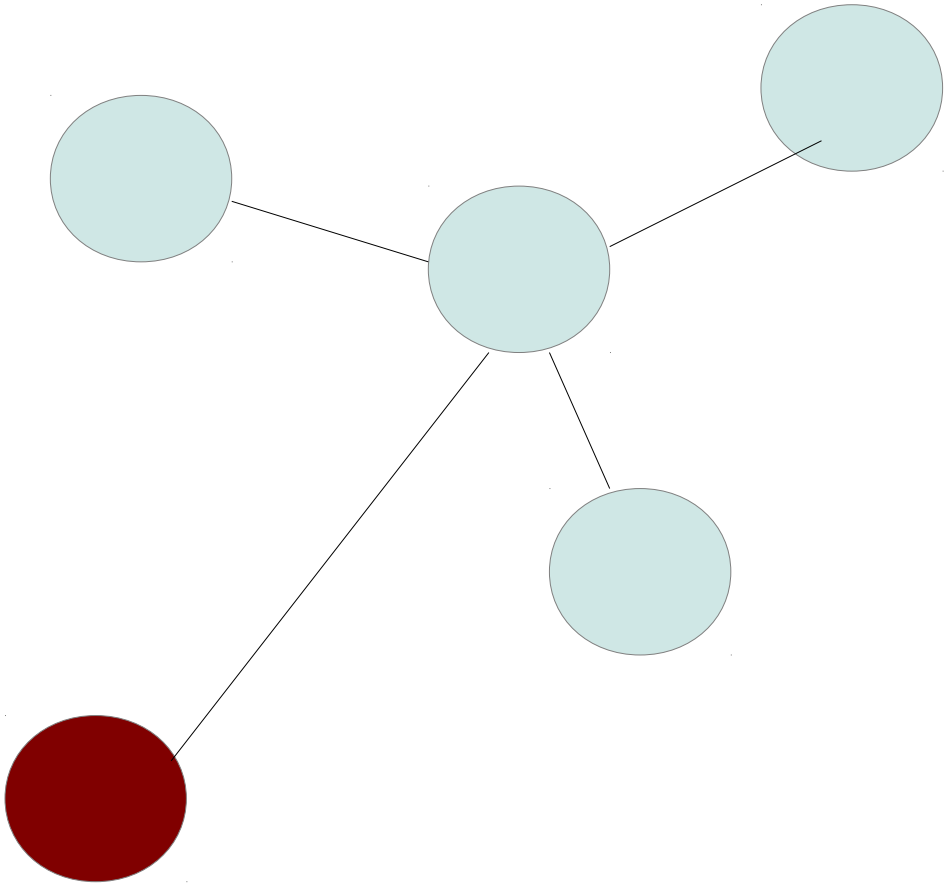
- 很多，很复杂。

# 来点简单的吧。

- 还记得 NOI 道路修建那题吗？
- 给一个树，求每个点到某一个点的距离和的最小值。
- 这个东西能动态维护吗？
-
- 什么叫动态？

# 动态。

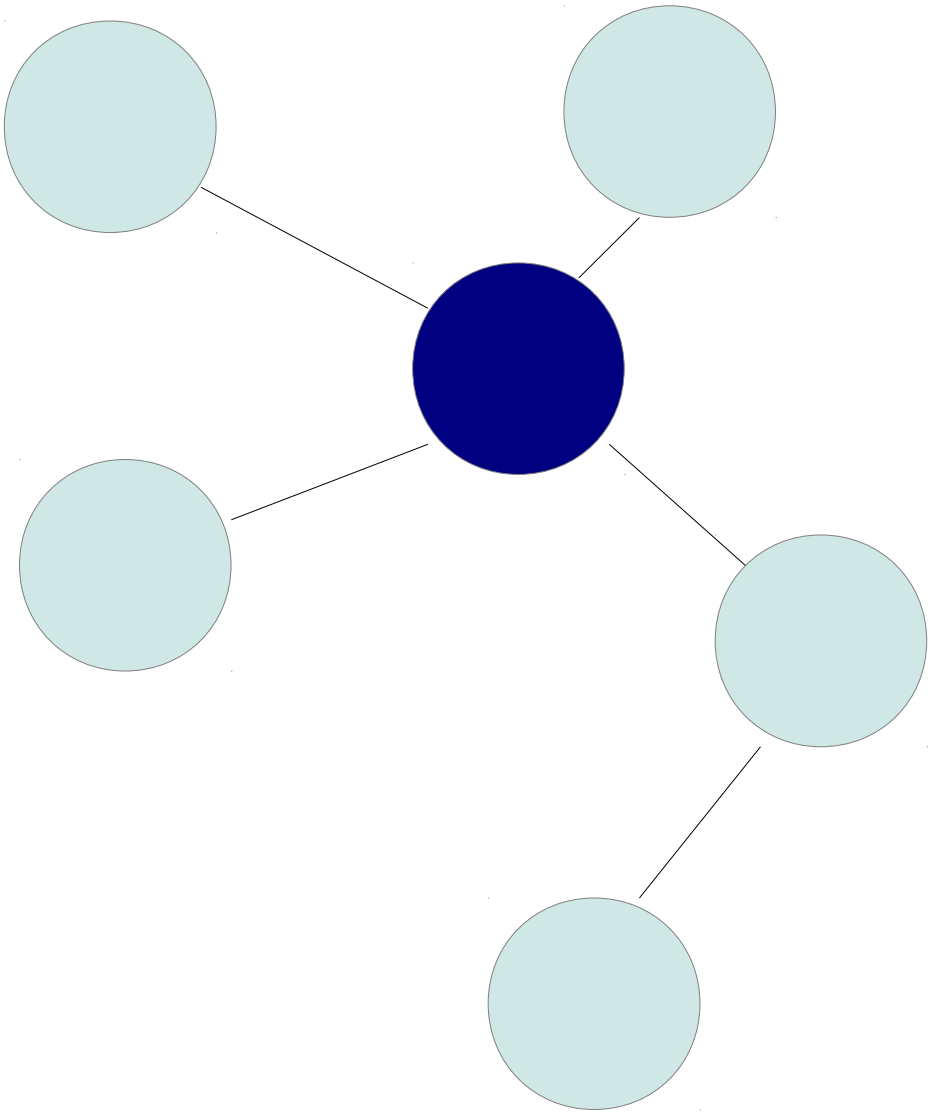
- 每次加一个叶子怎么样？。每次添一个叶子，查询最小距离和。



# 不是很显然了？。

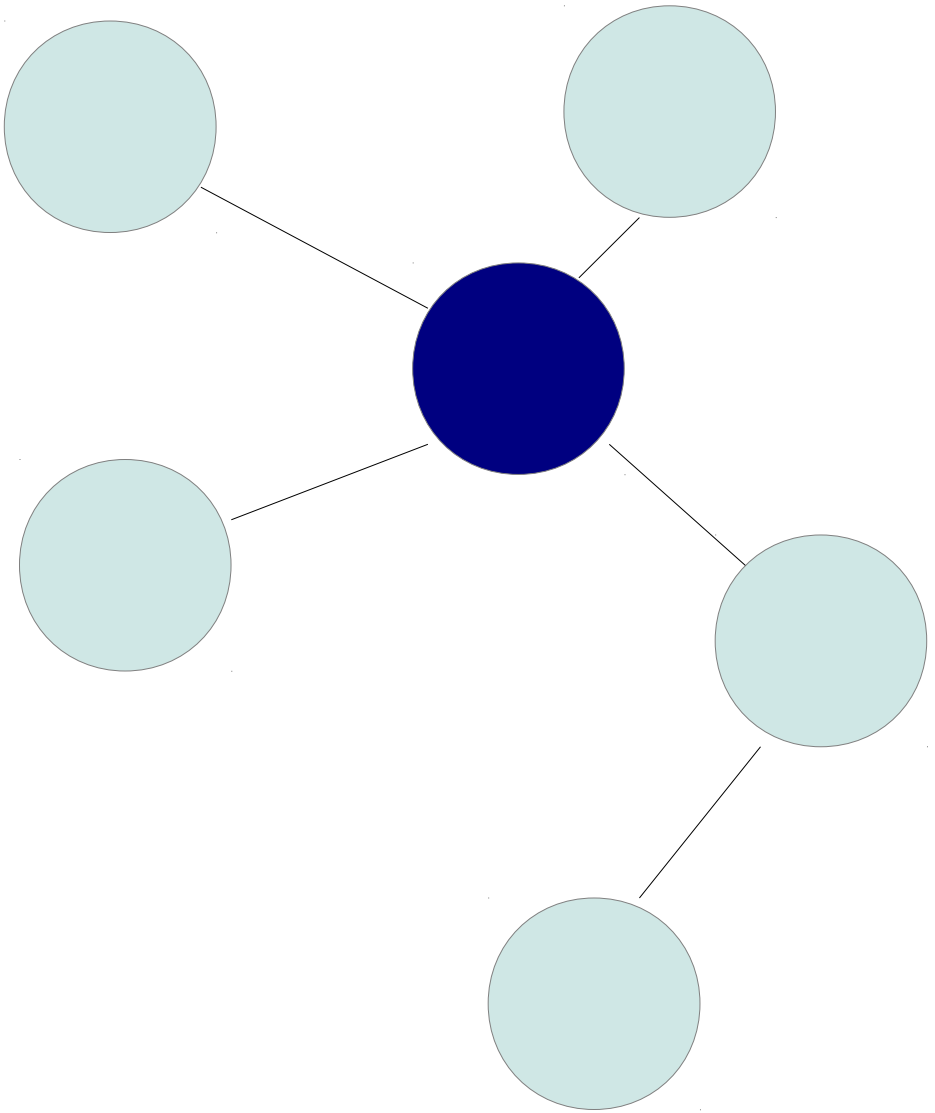
- 这题怎么用动态树呢？
-
- 想用动态树，要先搞清楚维护什么。
-
- 树上所有点到哪个点的距离最小？
- 树的重心！。

# 树的重心？。



树上，如果以一个点为根，所有的子树的大小都不超过整个树的一半，就叫这个点“树的重心”。树的重心最多两个（如果是两个，它们一定挨着）。

# 树的重心。



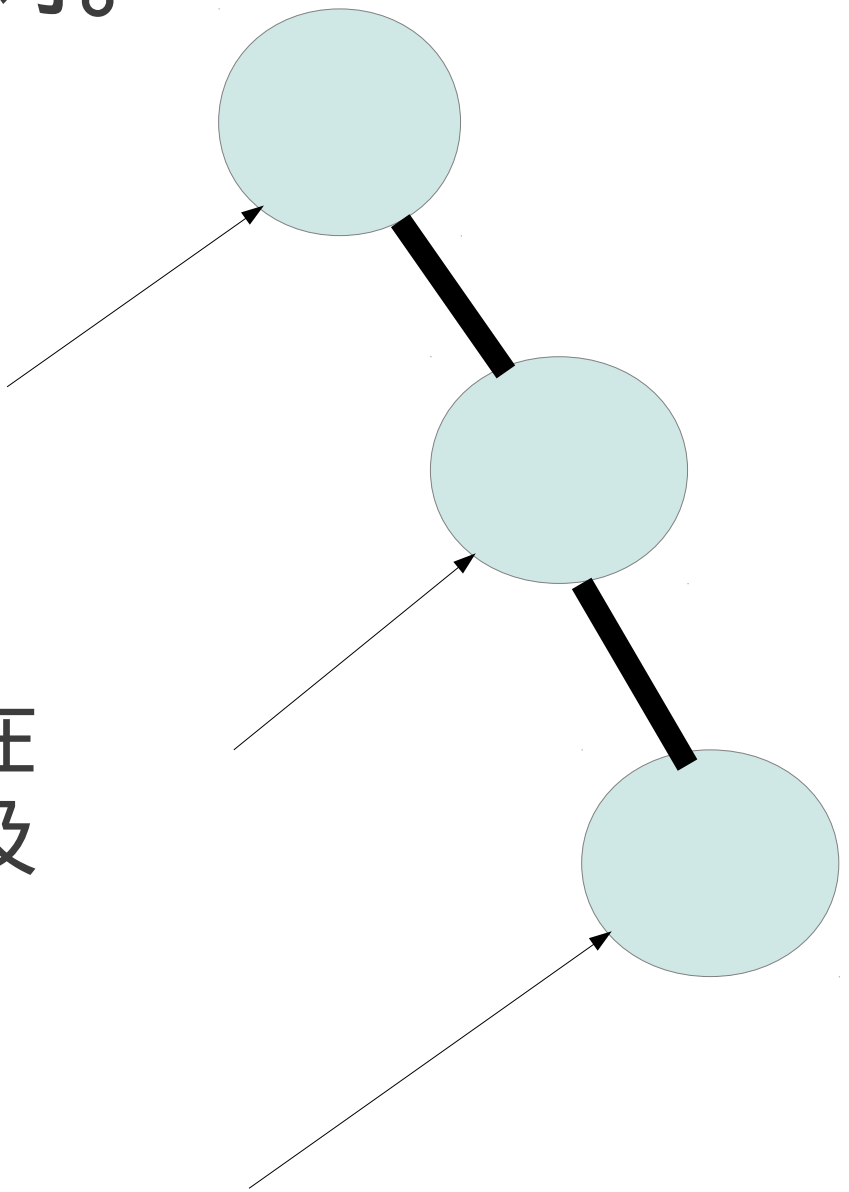
树的重心的性质：
1. 树上所有点到树的重心的距离和最短
2. 任取一条边把树分成两部分，重心一定在点多的那边。
3. ？？？有没有对动态维护有用的性质？

# 有！。

- 添加一片叶子，树的重心最多移动一个点。
- 我们以树的重心为根组织这个树。
- 新的重心一定在添加的点到根的路径上。
-
- 嗯，动态树出来了！

# 动态树。

- 谁是根？树的重心是根。
- 维护什么？
-
- 链的长度；
- 以某点为根的子树，不在链上的部分的大小，以及部分和；
- 距离和。



# 具体实现挺复杂的。

- 不过，也不是特别复杂。
- 也就几百行的代码。

# 数据结构啊。

- 动态树先说到这里吧。
-
- 还有没有新鲜玩意呢？。

# 你听说过函数式编程吗？。

- Haskell、 Erlang……
-
- 函数式编程的思想：
- 所谓程序，是把输入映射成输出的函数。
- 所谓函数，就是定义域 + 对应法则。
- 一个函数可以由其他函数拼起来。

# 举个例子。

- fac 0=1
- fac x=x*fac (x-1)
- 嗯，这就是阶乘的写法。
- 如果用命令式呢？
- s=1
- for i in range(1,n+1):
- s*=i
- 你注意到一个区别没有？
- 函数式编程从不**修改**任何东西。它只做一件事：定义。

# 从不“修改”任何东西？。

- 从不“修改”东西？你不修改任何修改任何东西能写出一个快排出来吗？
- quicksort [] = []
- quicksort (x:xs) =quicksort (filter xs (<x)) ++ [x] ++ quicksort (filter xs (>=x))
- 神奇吧？

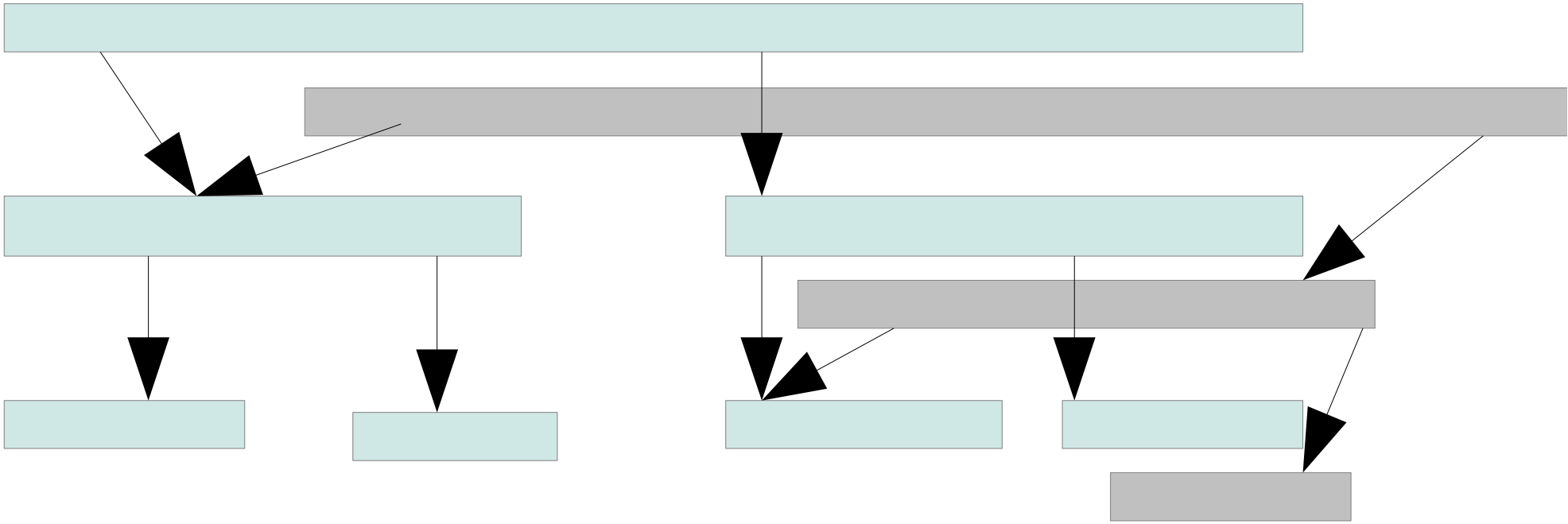
# 数据结构呢？

- 你能不“修改”任何东西写出一个线段树出来吗？
- 当然可以啦。
- 介于你对 Haskell 的语法了解比较少。。。我们就写伪代码吧。
- 支持：修改一个元素，查询一段和。

# "伪"代码。

- buildtree(l,r)= 递归构建一个从 l 到 r 的树。
- ask(a,b,e)= 递归地询问 a 节点对应的子树中从 b 到 e 的和
- change(a,b,y)= 返回**一棵新的树**的根，表示 a 节点对应的子树把位置 b 修改成 y 之后形成的树。
- 慢点，慢点：你怎么能每次返回一棵新的树！（空间 + 时间会爆的！）

# 为啥啊？。



既不爆空间，也不爆时间的秘诀在于：函数式编程中，我们从不“改变”什么，于是，可以放心大胆地“重用”以前的东西！。

# 说的挺好听。 So what?

- 这个给我们以想象空间：
- 如果有这样一个题：
- 维护一个序列，要求**在线**支持：
- 修改一个值
- 查询某个区间的最小值
- 查询 x **次修改之前**的某个区间的最小值。
- 这个怎么办？

# 暴力离线？。

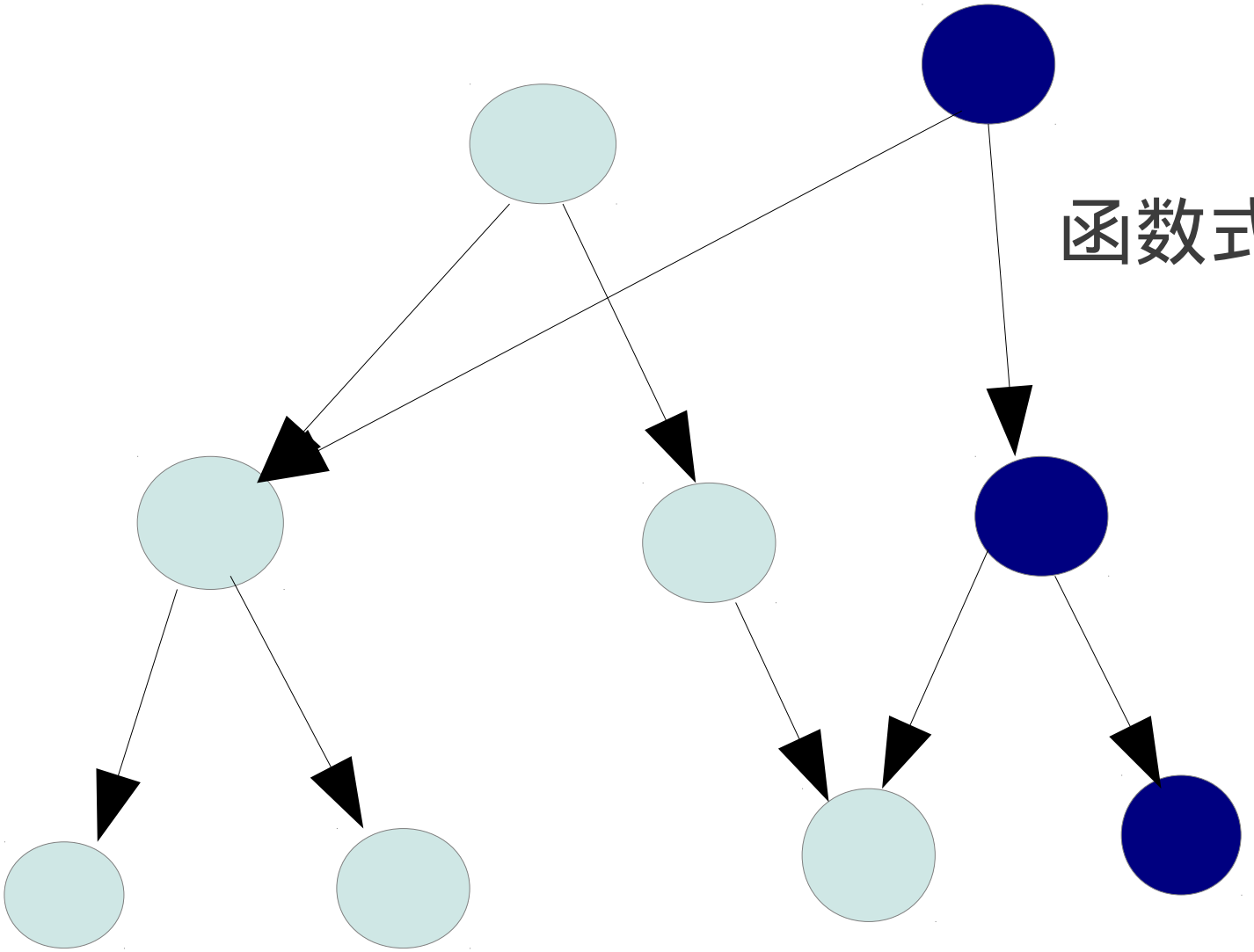
- 哼，总能有强迫你在线的方法的。

# ？？？怎么做？。

- 如果用命令式编程，这个问题将变得很困难……至少是不好做。（肯定可以做。）
- 但用前面介绍的函数式线段树，一切都和谐了！。。。
- 
- 我们从不"改变"什么，于是可以放心大胆地开一个表，记录下每次操作之后的线段树，直接查询以前的结果。

# 哇！好东西。

- 是的，是个好东西。
- 
- 除了函数式线段树，我们还可以有：
- 函数式 Treap
- 函数式 AVL
- ……
- 
- **函数式** Splay ？不可以！因为势能分析失效了！（访问以前的数据。。。）
- 函数式动态树？。。。慢慢研究吧。



# 函数式 Treap 假象图

# 等等！旋转！。

- 大多数平衡树（Treap，AVL，SBT）都要旋转以保持平衡。这个用函数式来表达就有点太痛苦了。
- 怎么办？
-
- 你干嘛非要旋转呢？
- Think Functional ！

# Treap 代码

- struct node{
- int key,weight;
- node *left,*right;
- node(int _key,int weight,node * left,node * right) : key( _key ), weight ( weight ) , left ( _left ) , right ( _right ){ }
- };
- node * newnode(int key){
- return new node(key,rand(),NULL,NULL);
- }

# 插入怎么写？不要想插入的事情。

- 我们定义三个函数：split_l，split_r，merge，表示把一棵树按 key 分割成两个树，以及把左、右子树合并成一个树。
- 所谓插入：
- insert(a,x)=merge(merge(split_l(a,x),newnode(x),split_r(a,x))
- 所谓删除：
- remove(a,x)=merge(split_l(a,x),split_r(a,x+1))

# merge

- node * merge(node *a,node *b){
- return (!a || !b)?(a?a:b):
- (a->weight<b->weight?
- new node(a->key,a->weight,a->l,merge(a->r,b):
- new node(b->key,b->weight,merge(a,b->l),b->r));
- }
- 没"旋转"什么事吧。。。

# split_l

- node * split_l(node *a,int key){
- return !a?NULL:
- (a->key<key?
- new node(a->key,a->weight,a->l,split_l(a->r,key):
- split_l(a->r,key));
- }
- split_r 是对偶的；依然没"旋转"什么事。

# 能不能不用 merge/split 来 insert 呢？

- node* insert(node *a,int x,int w){
- return
- (!a || a->weight>w)?
- new node(x,w,split_l(a,x),split_r(a,x)):
- x<a->key?
- new node(a->key,a->weight, insert(a->l,x,w),a->r):
- new node(a->key,a->weight,a->l, insert(a->r,x,w);
- }

# 看起来挺好的。

- 是挺好的。
-
- 在 STL 扩展中，有一个神奇的东西：
- rope
- 它是用类似的方法实现的一个字符串的数据结构。（只不过它使用了更复杂的数据结构。。。）
- 在 Haskell 的数据结构实现中，映射，优先队列都有对应的“专用”数据结构来实现。
-
- 能出什么题吗？能出不“裸”的题吗？。。。慢慢想吧。

# 再讲一个东西吧。

- Think Beyond logN

# 为什么总算是 logN 呢？

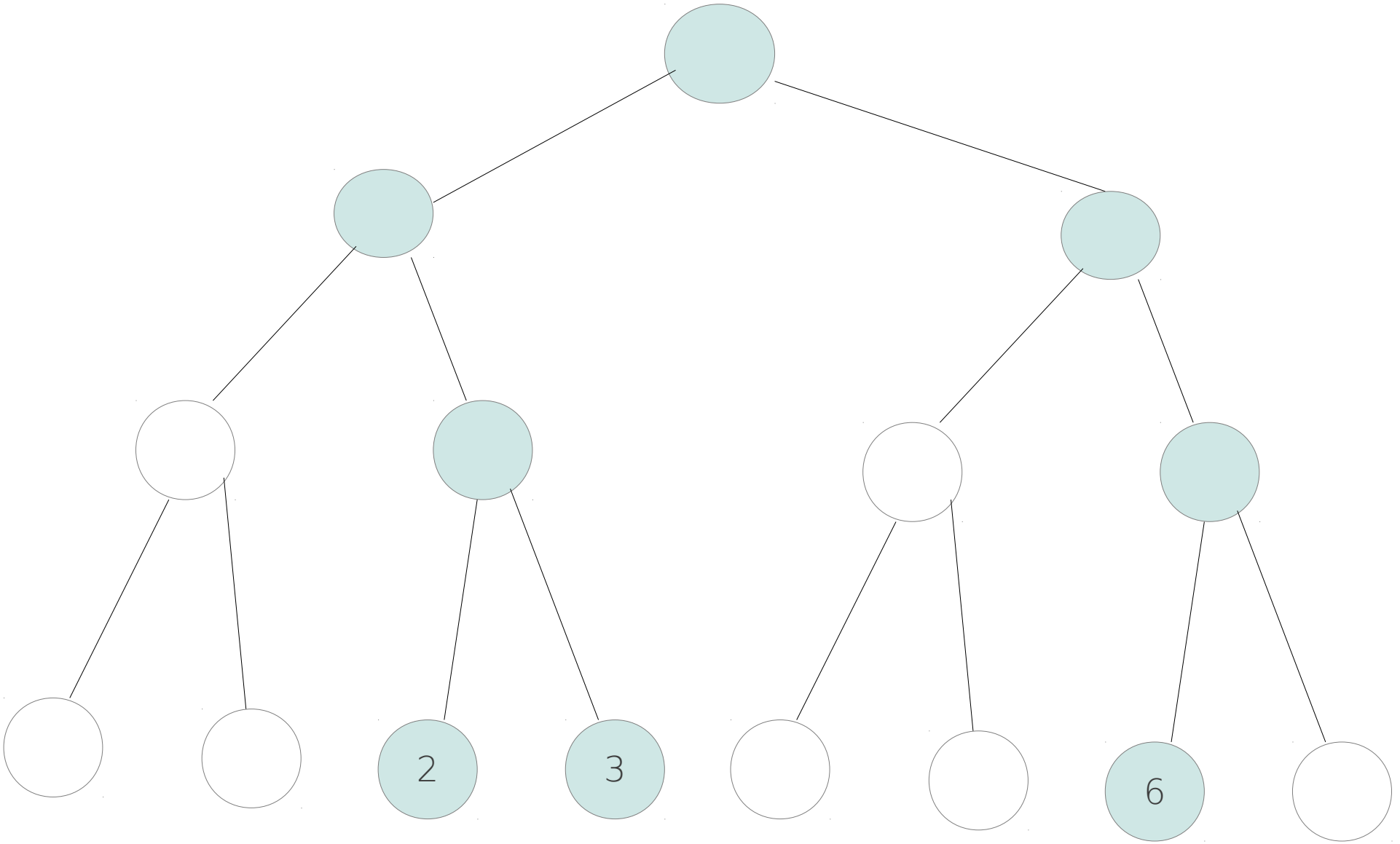
- 我们想当然地认为：
- 有序集合的操作的时间复杂度最好是 O(logN)
- 排序的时间复杂度最好是 O(NlogN)
- Dijkstra 的时间复杂度是 O(NlogN+M)
- ……
- 为什么总是 logN 呢？
- 因为不能做到 O(1) 对吧。。。

# logN 不是尽头

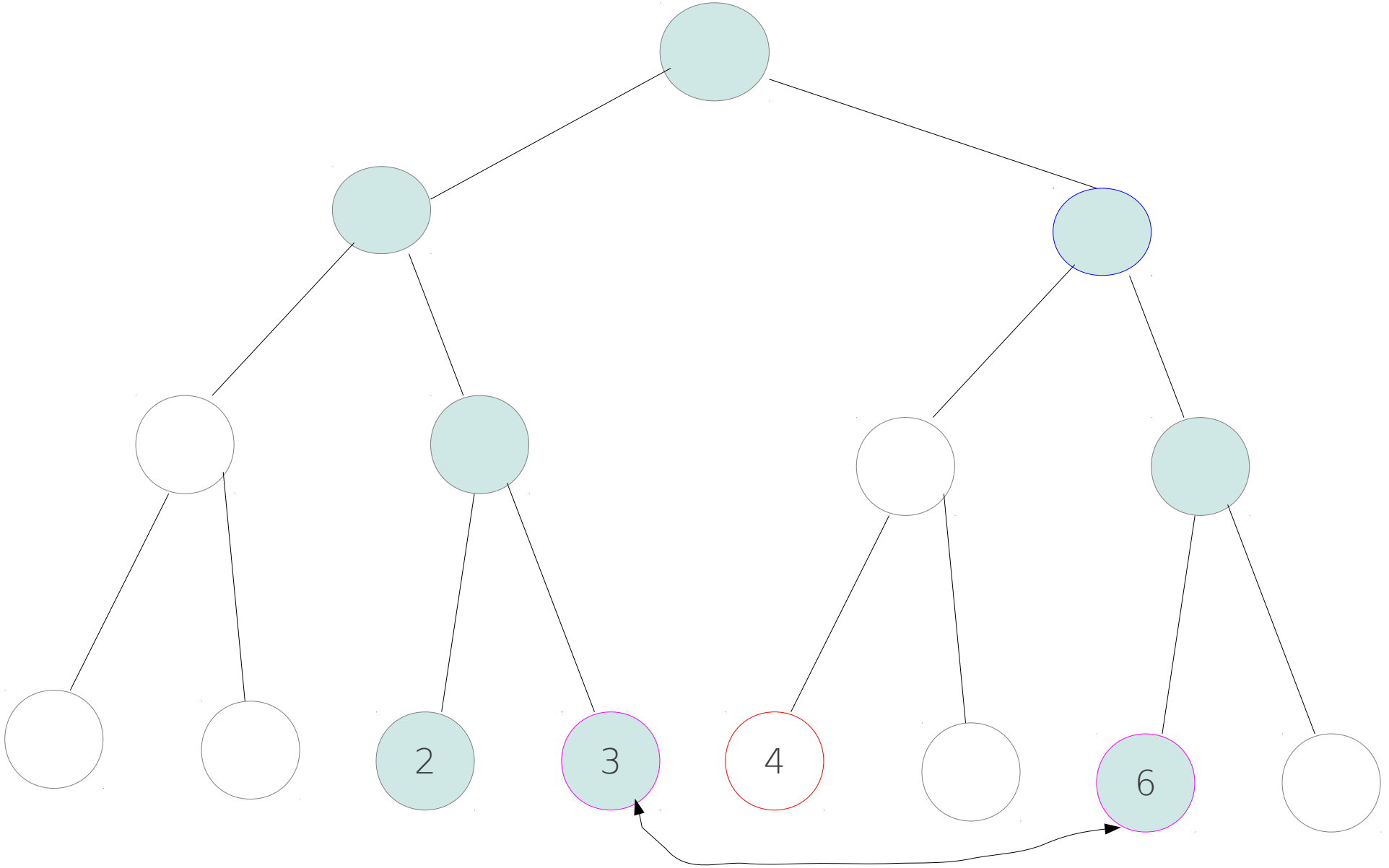
- 在合理的假设下，
- 排序 N 个 int 的时间可以是：
- NloglogN
- $N\sqrt{\log(\log(N))}(randomized)$
- set<int> 的每次操作可以是
- O(log w)(w 是字长，如果取 $w=\log^{O(1)}N$，就是 O(loglogN))

# 要不然举一个例子？。

- 介绍一下 y-fast tree 。
- 先说说基本的假设：
- 计算机的字长是 w ，所要处理的数据都是 w 位长的 int ； w 大于 logN 。
- set<int> 中的操作：
- find （这个用个 hash 可以 O(1) 搞定）
- ++ ， --（这个用个双向链表）
- lower_bound （这是 y-fast tree 要处理的）




想象一个 Trie；每片叶子代表一个数。
我们用一个 Hash 来存所有存在的节点。同时，每个节点也记录它下面的数的最大、最小值。

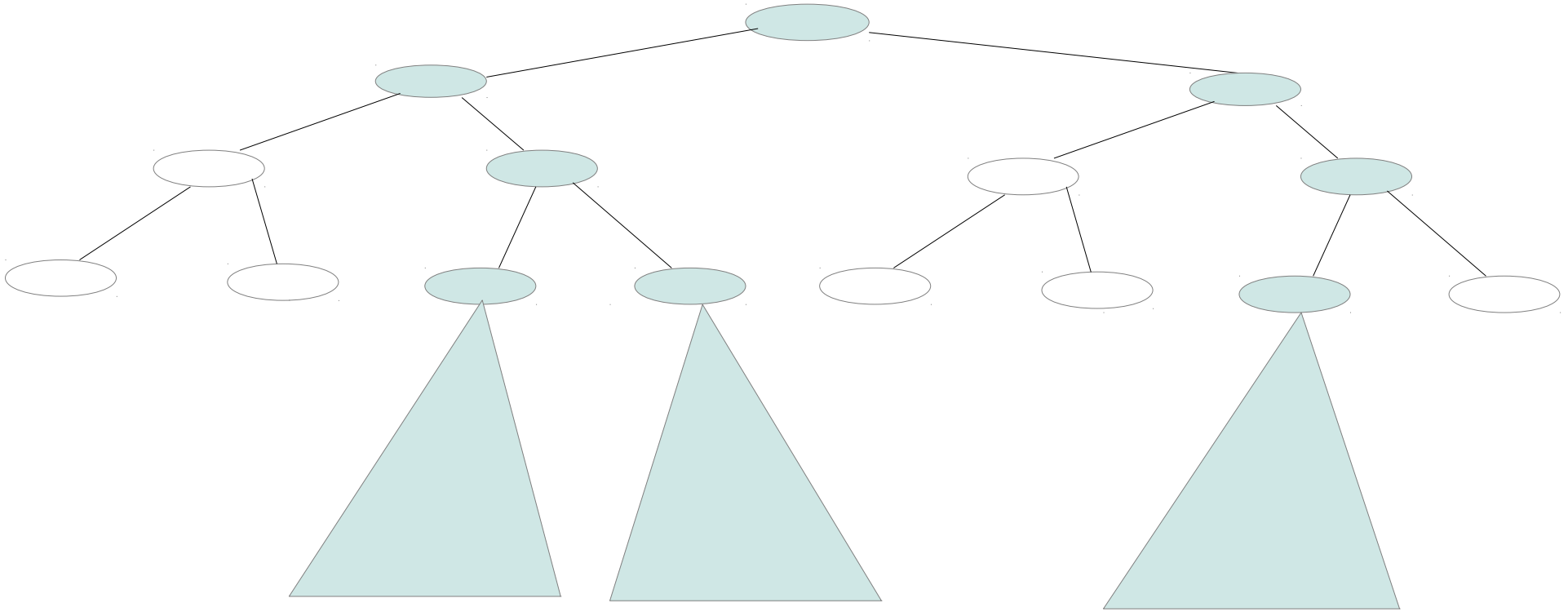



如何查找一个一个数的 lower_bound 呢？很简单，我们**二分查找**这个数所对应的路径上最低的存在的节点。之后查询这个节点对应的的 min/max，这样就能找到被查找数的前驱 / 后继。通过双向链表即可找到 lower_bound。（用时 O(log w)）

# 慢点，慢点。

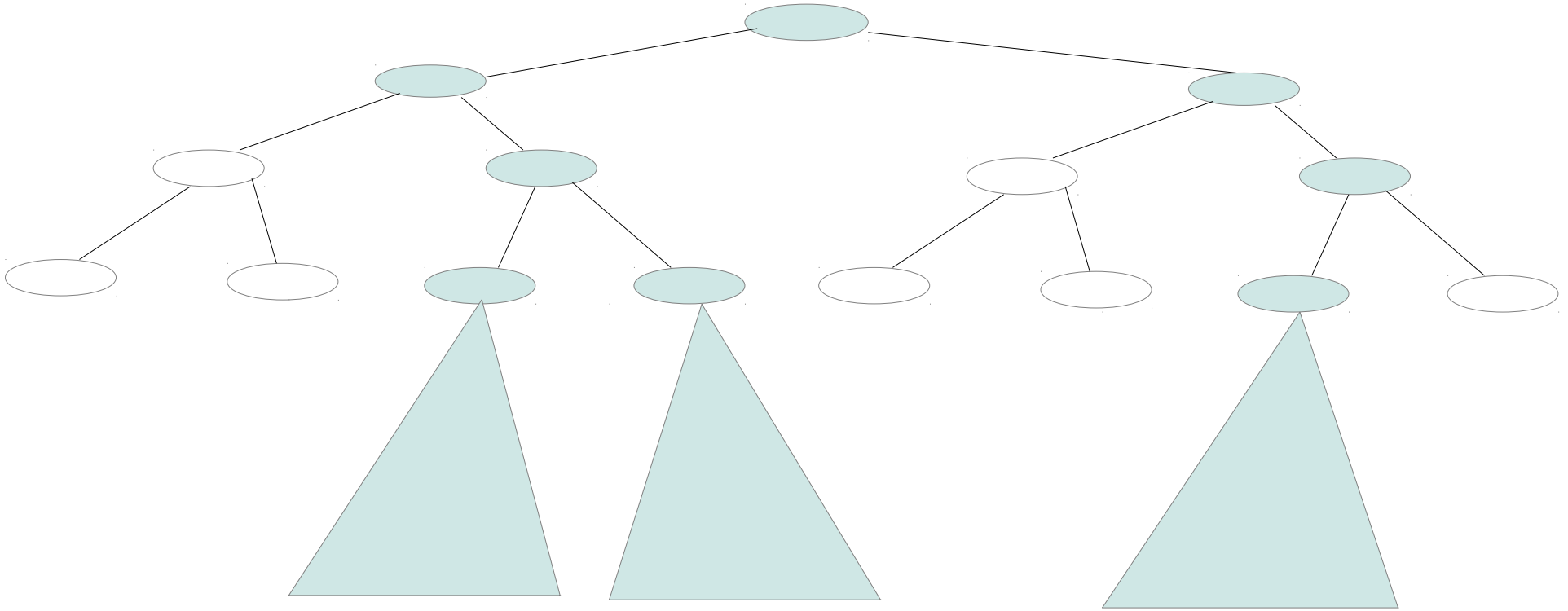
- 我怎么觉得，你这个 y-fast tree 的时间复杂度是：
- 插入 / 删除： O(w)
- 查找： O(log w)
- 空间： O(N*w)
-
- 嗯，是的，但是，不要着急。

# “重定向”。



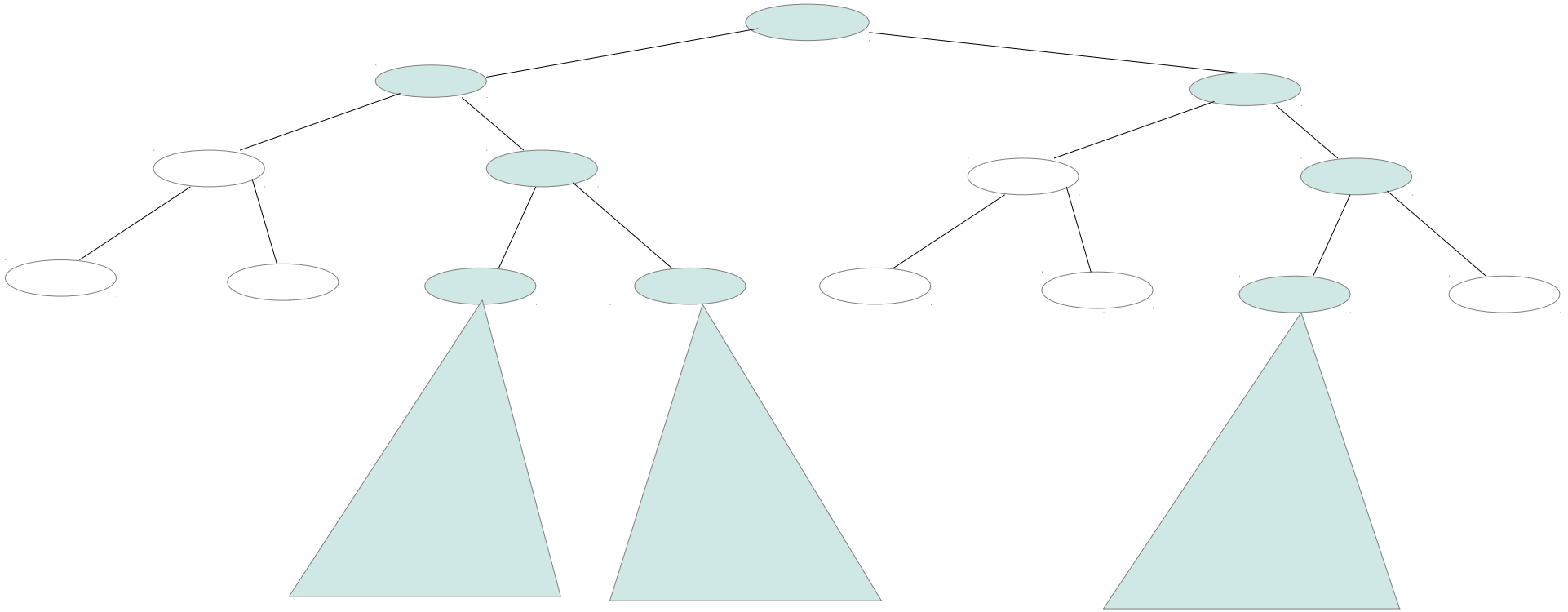
看时间复杂度是如何变戏法的：
我们把 Θ(w) 个相邻的数变成一组，每组使用一个比较“正常”的 set<int> 来维护；每组挑一个代表，存储到 y-fast tree 中。

# “重定向”。



查询的时候，先在 y-fast tree 中查到代表，找到所在的组，之后再在组内查询。时间复杂度依然是 O(log w)

# “重定向”。



修改的时候，先在 y-fast tree 中找到对应的组，在组内进行插入 / 删除。在组内数字个数超过 2w 或者相邻两个组的大小和小于 w ，就进行一次分裂 / 合并，同时修改 y-fast tree 。这样， y-fast tree 的修改的时间复杂度被均摊为 O(1) ，空间也变成了 O(N) 。

# 乱七八糟的。

- 嗯，这样你信了吧， set<int> 可以在 O(log sizeof(int)) 的时间内实现。
- 不过。。。这个玩意只有理论上的价值。
-
- 但它至少告诉我们：
- 做到 logN 远不是极限。

？。

- 什么？
- 你想到大象了？
-
- 很好。。。

超越 logN，我在路上等你。

# The End

Time for lunch!