最后开始说数据结构。

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……我们已经厌烦这种争吵了。

数据结构啊。

- 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。

- 如何实现：读入若干行，输出长度不超过 10 的行？
-
- main = interact $ unlines . filter ((<10) . length) . lines

举个容易懂的例子。。。

- 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**<u>修改</u>**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。

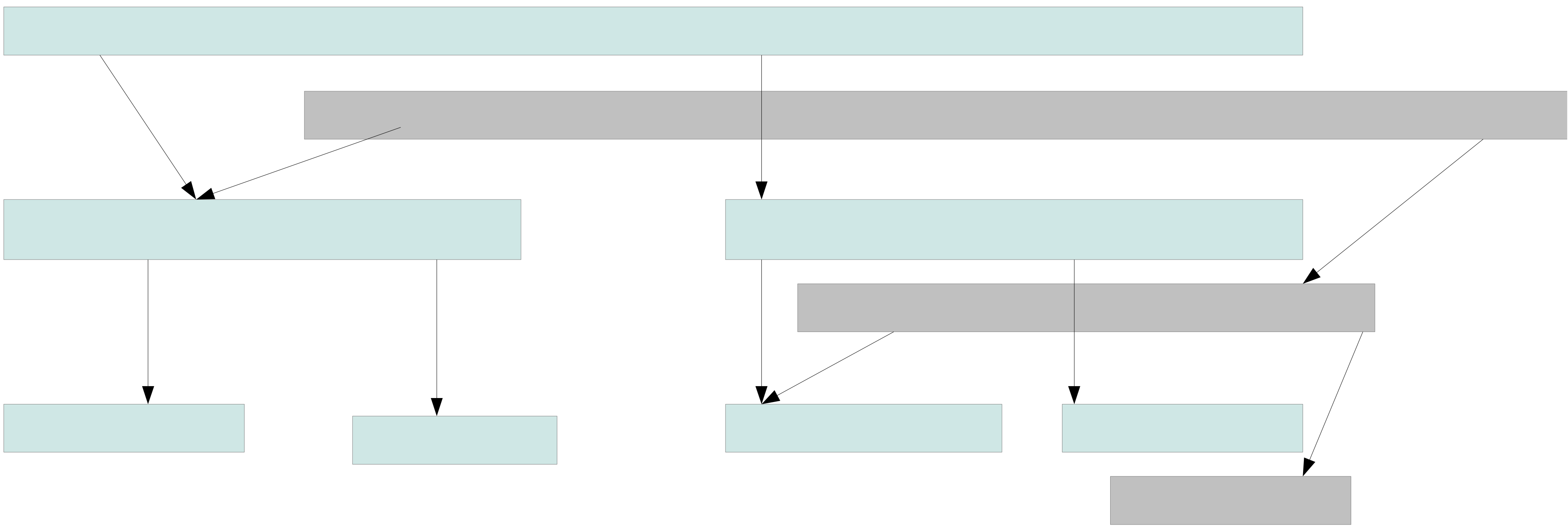
“伪”代码。

- 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！）

“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。

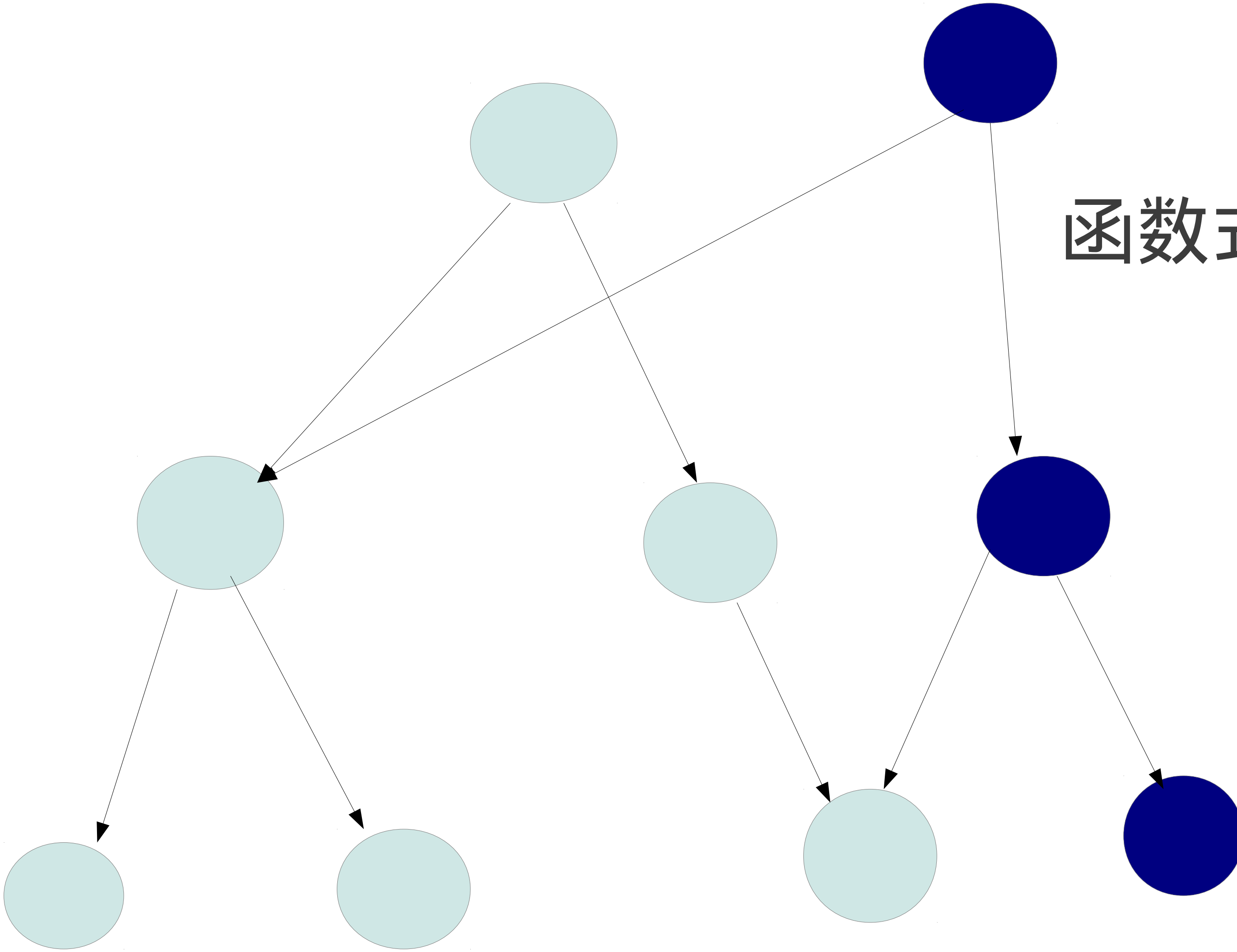
哇！好东西。

- 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。

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
- ....
- **<u>函数式</u>** <u>Splay</u> ？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？

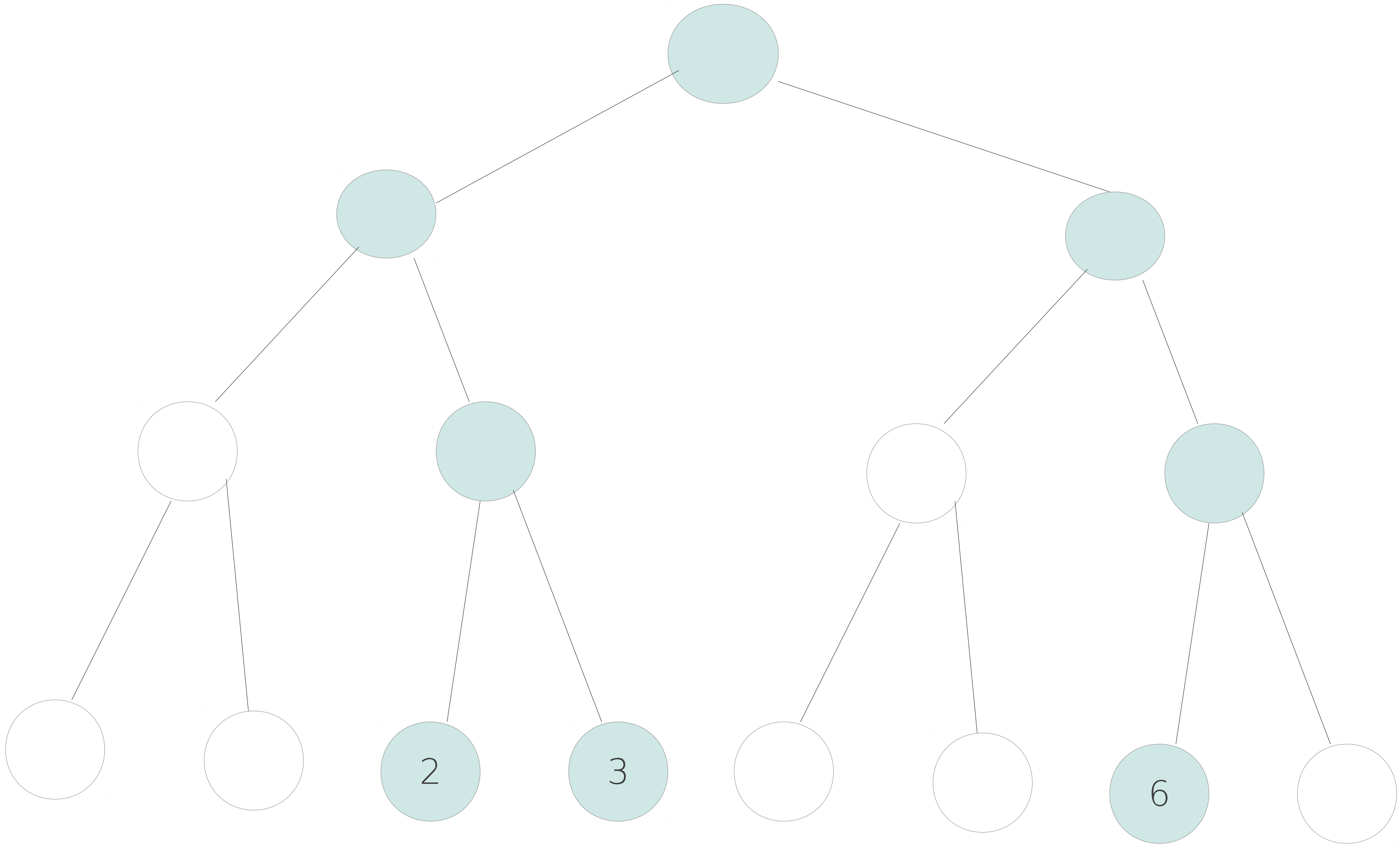
- 我们想当然地认为：
- 有序集合的操作的时间复杂度最好是 O(logN)
- 排序的时间复杂度最好是 O(NlogN)
- Dijkstra 的时间复杂度是 O(NlogN+M)
- …….
- 为什么总是 logN 呢？
- 因为不能做到 O(1) 对吧。。。

logN 不是尽头

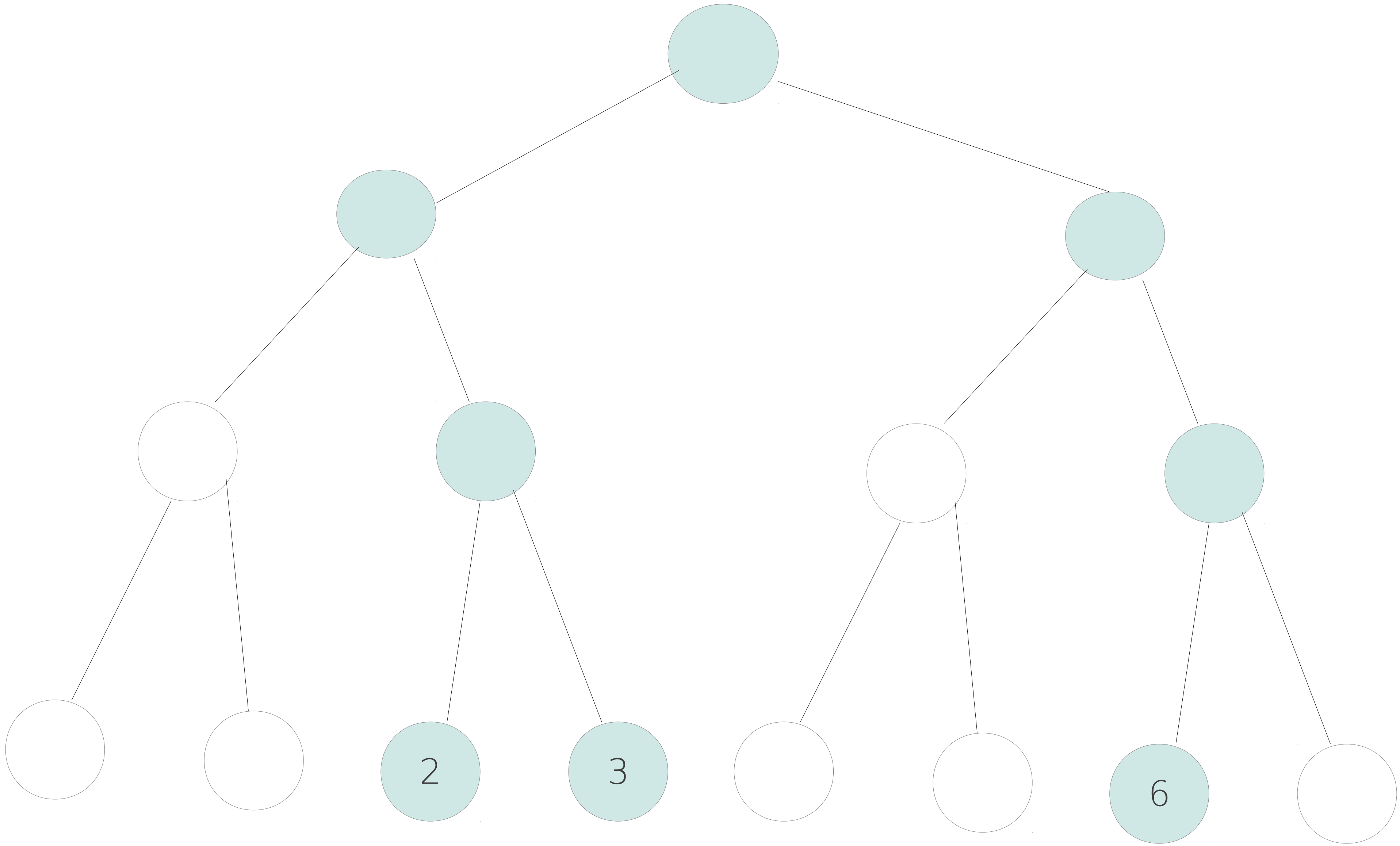
- 在合理的假设下，
- 排序 N 个 int 的时间可以是：
- NloglogN
- $N\sqrt{\log(\log(N))}(randomized)$
- set<int> 的每次操作可以是
- O(log w)(w 是字长，如果取 w=$\log^{O(1)}$N ，就是 O(loglogN))

要不然举一个例子？。

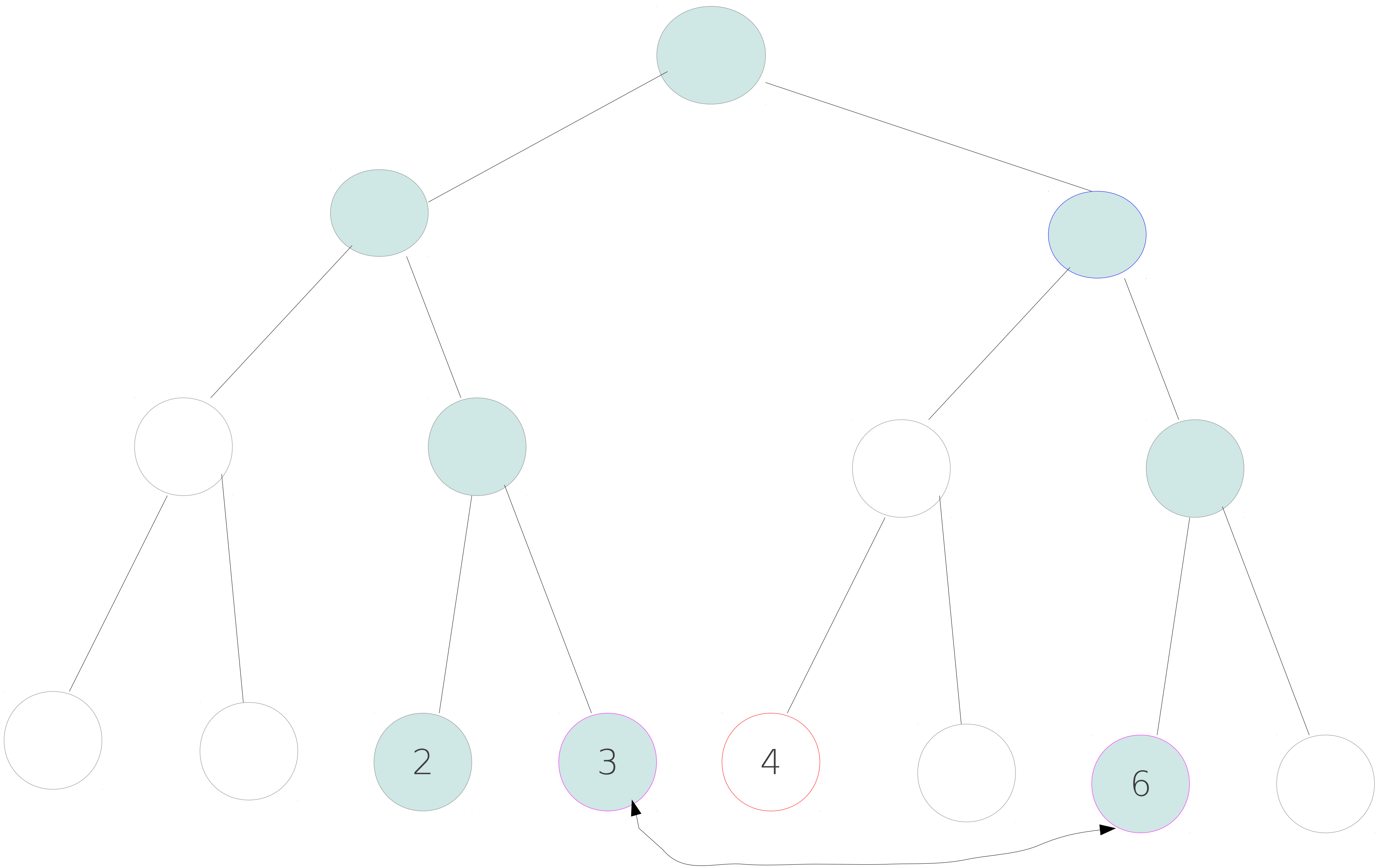
- 介绍一下 y-fast tree 。
- 先说说基本的假设：
- 计算机的字长是 w ，所要处理的数据都是 w 位长的 int ； w 大于 logN 。
- set<int> 中的操作：
- find （这个用个 hash 可以 O(1) 搞定）
- ++ ， -- （这个用个双向链表）
- lower_bound （这是 y-fast tree 要处理的）



你想到什么了？



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) ）

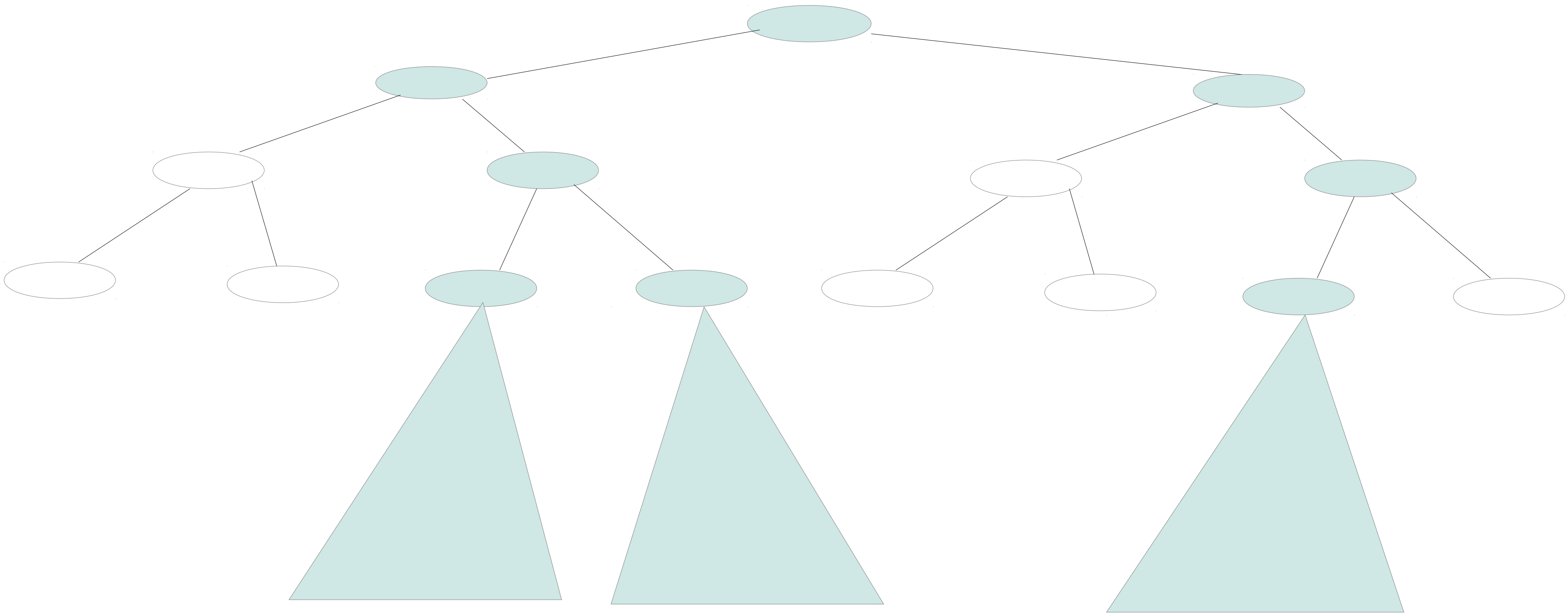
时间复杂度？。

- 插入？
- 删除？
- **查询**？。
- 还有空间？

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。

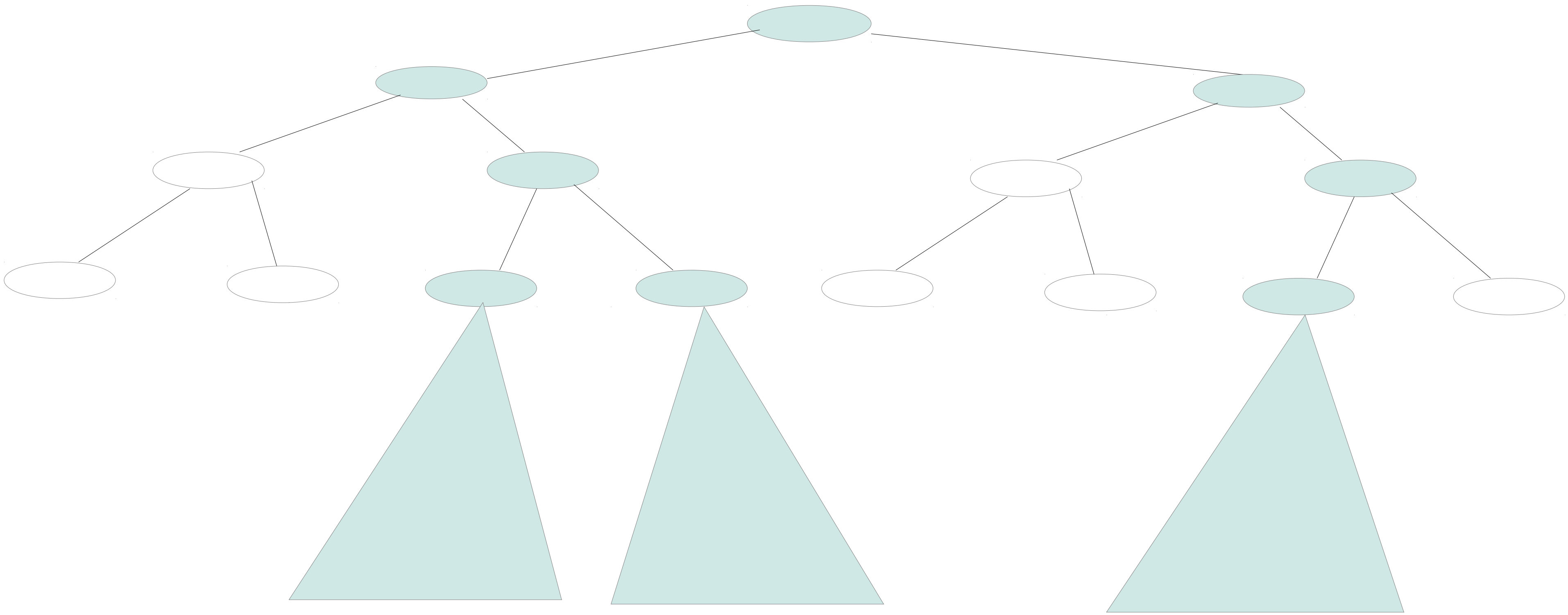
“重定向”。



看时间复杂度是如何变戏法的：

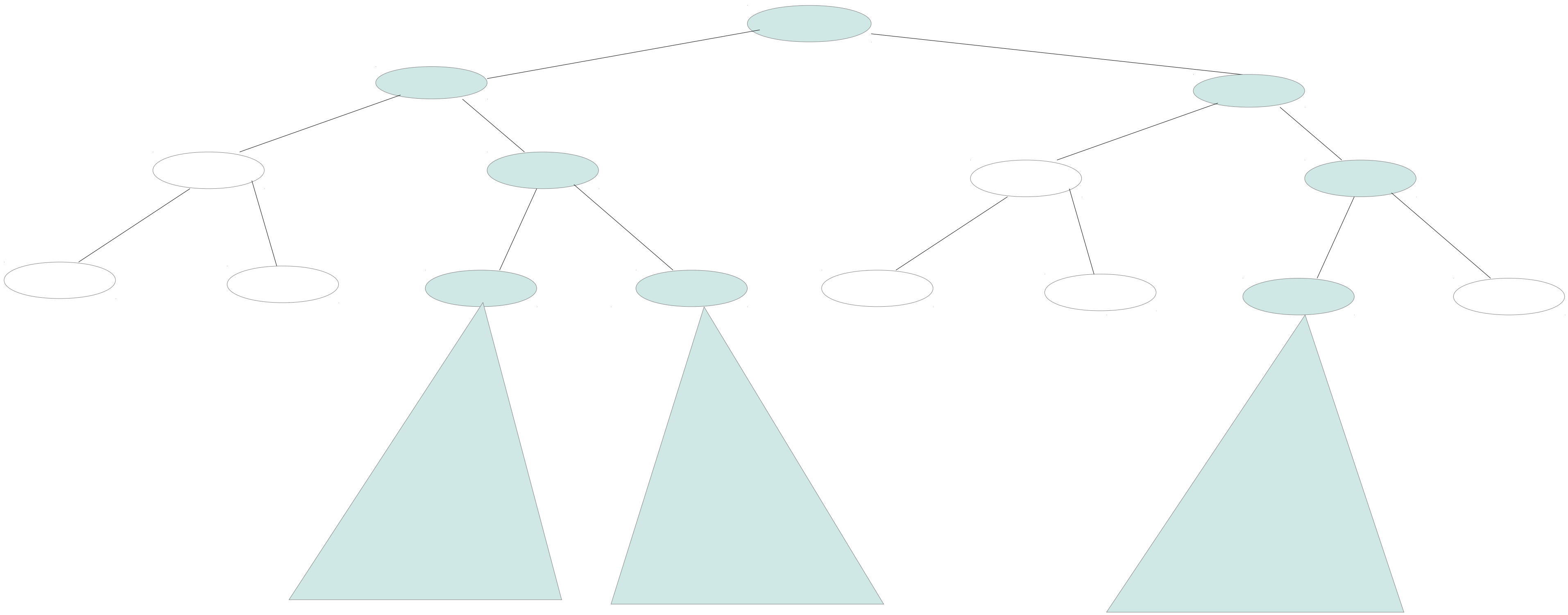
我们把 $\Theta(w)$ 个相邻的数变成一组，每组使用一个比较“正常”的 set<int> 来维护；每组挑一个代表，存储到 y-fast tree 中。

“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!