统计的力量
——线段树全接触

清华大学 张昆玮

* 根据 D. E. Knuth 的分类方法
  计算机算法可以分为两类：
* 数值算法与非数值算法
* 其中的非数值算法包括：
  * 索引
  * 分类
  * **统计**
  * ……

* 许多算法的本质是统计

* 大家都说：
* ……
* 常数很大？
* 不好写？
* 难调试？
* 想不到？
* ……

* 线段树？

* POJ 上的某题，时限很紧……
* 大家都用树状数组，但是有人只会用线段树呢？
* 而且我可以轻易改出一道不能用树状数组的题
* 在线段树一次次 TLE 后，有一个 ID 发帖抱怨
* “下次写一个**汇编**版**非递归**线段树，再超时？”
* 可是大家都知道，超时的代码已经 2k 了。

* 其实我写的就是线段树。很快，而且不到 1k 。

* 一个悲剧

* 运行速度快
* 适应能力强
* 编写方便
* 结构简单
* 容易调试
* 关键在于**灵活**实现

* 线段树用于统计

*线段树，从何而来？

为什么在《算法导论》和黑书中都难见到其踪迹？

* 计算几何在长期的发展中，
  诞生了许多实用的数据结构。
* 区间查询，穿刺查询都是计算几何解决的问题
* 作为特例中的特例，线段树解决的问题是：
  * 一维空间上的几何统计
* 高维推广（ kd-tree & … ）可以进行正交查询

* 由于竞赛中涉及计算几何的内容有限，不展开

* 计算几何！

* 线段树把数轴分成区间来处理
* 如 [8,10), [10,11), …
* 实际上我们面对的往往是离散量
* 竞赛中出现的线段树往往因此退化为“点树”
* 即，最底层的线段只包含一个点：
* 如： [8,9) 中只有整点 8 而已

* 在之后的讨论中，我们不再区别“点树”与线段树

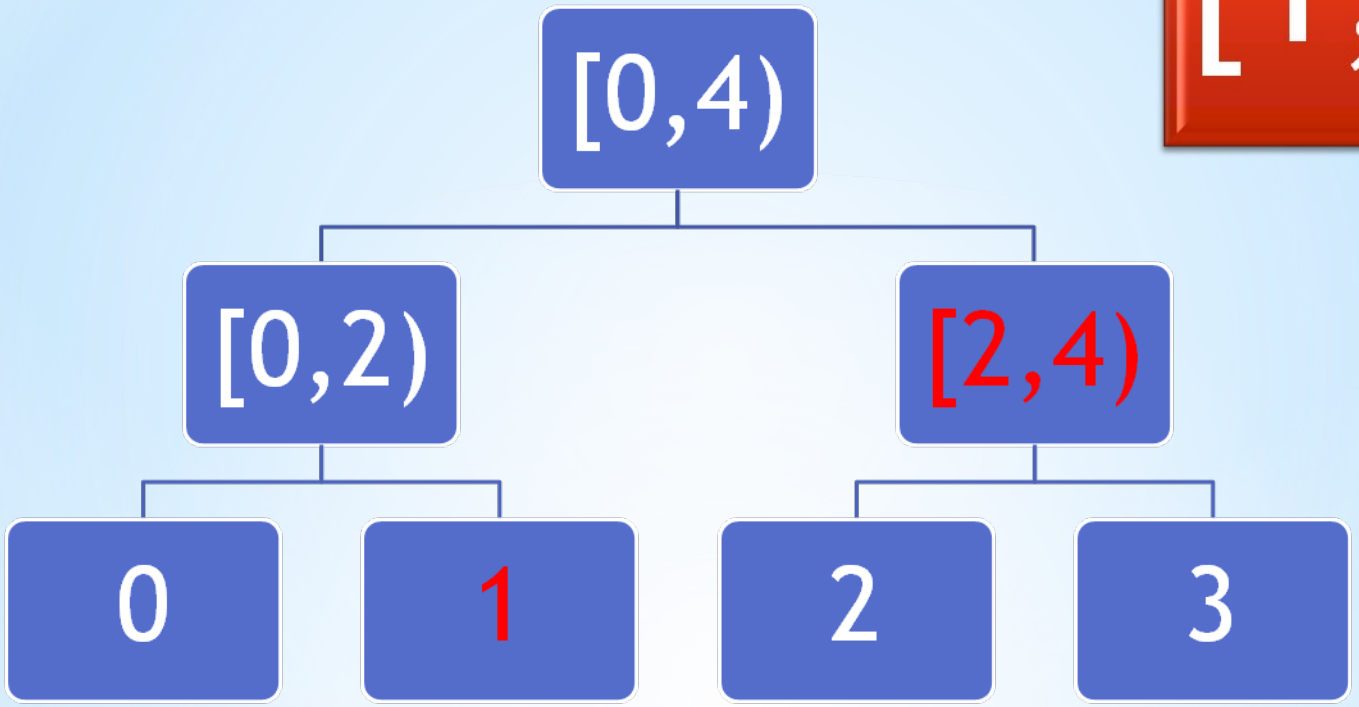
* 真正有用的是“点树”

*第一印象

如果我给 MM 的第一印象不到 80 分的话……

是不是老老实实地早点罢手比较好？

[1,4) ?



* 最经典的问题：区间和

* [1,4) in [0,4)
  * 左边， [1,2) in [0,2)
    * Get 1
  * Get [2,4) in [2,4)
* 虽然每次都有可能同时递归进入两棵子树，
  但每层实际上只访问了两个节点。为什么？
* 因为查询是连续的啊……

* 核心思想：分治

其实还有别的核心思想
后面再讲……

如果同一层有三个被访问，依次为 A， B， C
查询是连续的，有了 A 和 C，就一定包括 B 在树中的兄弟
那我直接用 B 的父亲来计算好了，为什么要用 B ？

*因为查询是连续的？

*为什么用线段树？

功利点说，没啥用的东西咱不学.....

* 且慢

区区区间和，用的着线段树？

* 直接把原数组处理成前缀和
* For i=2 to n do
  * A[i] += A[i-1]
* Ans(a,b) = A[a] - A[b-1]

* 原因是区间和的性质非常好
* 满足**区间减法**
* 区间减法什么的最讨厌了！后面再说！
* 不过直接前缀和不是万能的！
* 如果修改元素的话……

* 这是为什么呢？

数据结构	修改元素	取前缀和
直接存储原数组	O(1)	O(n)
直接存储前缀和	O(n)	O(1)
线段树	O(logn)	O(logn)

沟通原数组与前缀和的桥梁

*真正的作用

其实……（其实什么，后面再讲）

*怎么写？

这个问题，**本来**是仁者见仁，智者见智的啊

但是我要讲一点不那么“**本来**”的东西

* 访问线段
  * 如果要的是整条线段就直接返回
  * 如果与左子线段相交就递归处理
  * 如果与右子线段相交就递归处理
  * 合并所得到的解
* 遗憾的是，这种朴素的方法**并不是那么容易实现**
* 而且存在严重的效率问题（常数太大）

* 朴素的递归算法

TLE

* 怎么办？

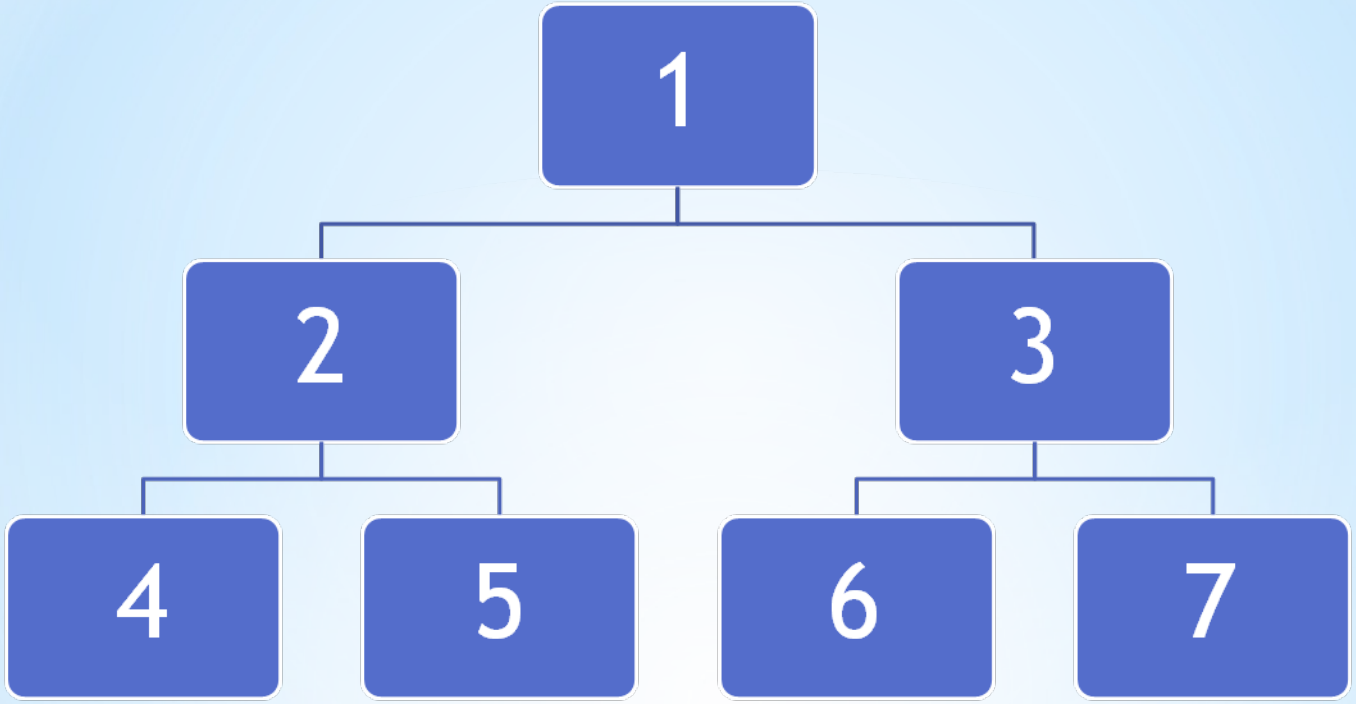
*从存储方式讲起

工欲善其事，必先利其器。

* 虽然可以设计出三叉，四叉，……线段树
* 但是我们先从二叉树开始
* 登高必自迩，行远必自卑
* 多叉怎么办~~后面再讲~~（这还要讲？自己研究去）

* 为了不处理各种特殊情况，假设二叉树是满的！
* 不是满的？你的总区间是[0,1000)？
* 你就当作[0,1024)不就好了？

* 几个不那么重要的问题



* 堆式存储是关键

指针退休了？
后面再讲.....

*N 的左儿子是 2N

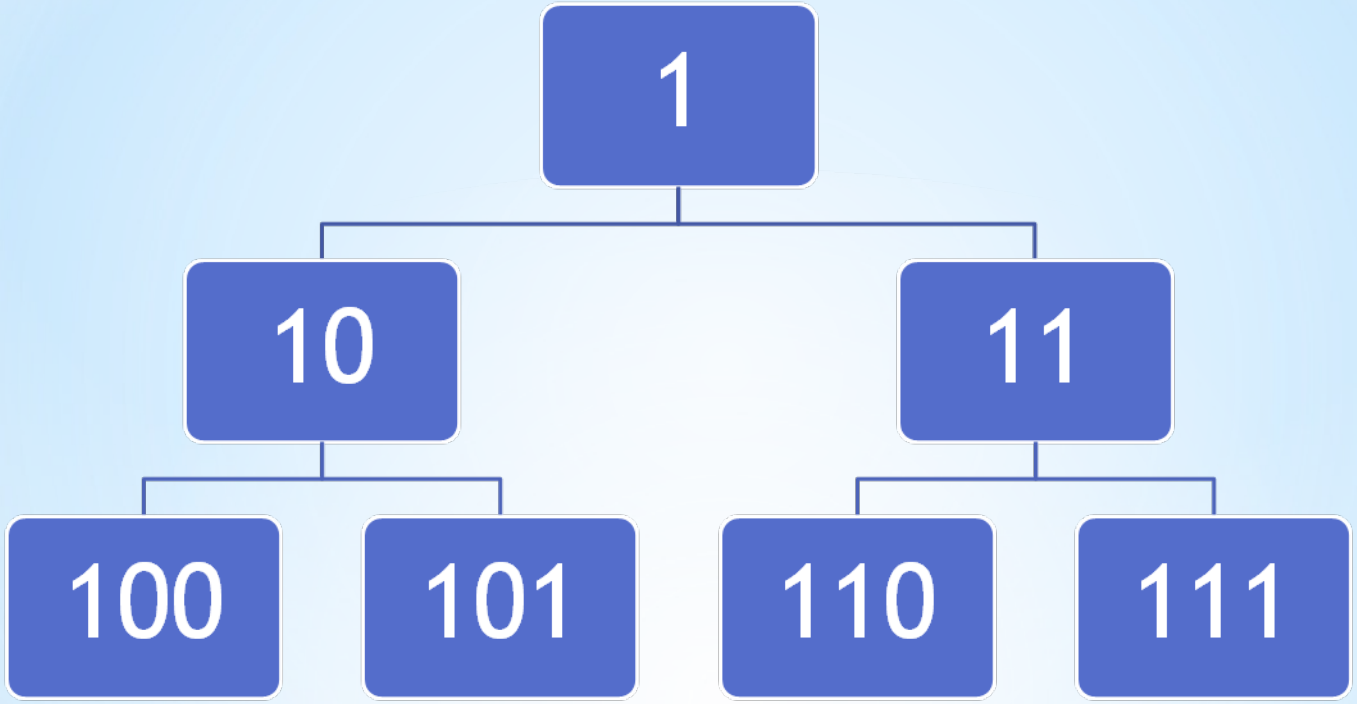
*N 的右儿子是 2N + 1

*N 的父亲是 N >> 1

*……

*不就是个堆存储么？不用讲了吧？

*一些简单的问题



*换成二进制看看吧！

* 字母树！
* 存放的是 100,101,110,111 四个串！
* 每个节点存的是以这个节点为前缀的所有节点和
* 例： 1 中存的是所有四个以 1 开头的和。

* 似乎从 100 到 111 就正好是原数组
* 貌似……下标减 4 就行了？

* 似曾相识？

* 我们可以直接找到一个数对应的叶子
* 不用自顶向下慢慢地找啊找
* “直接加 4”多简单！

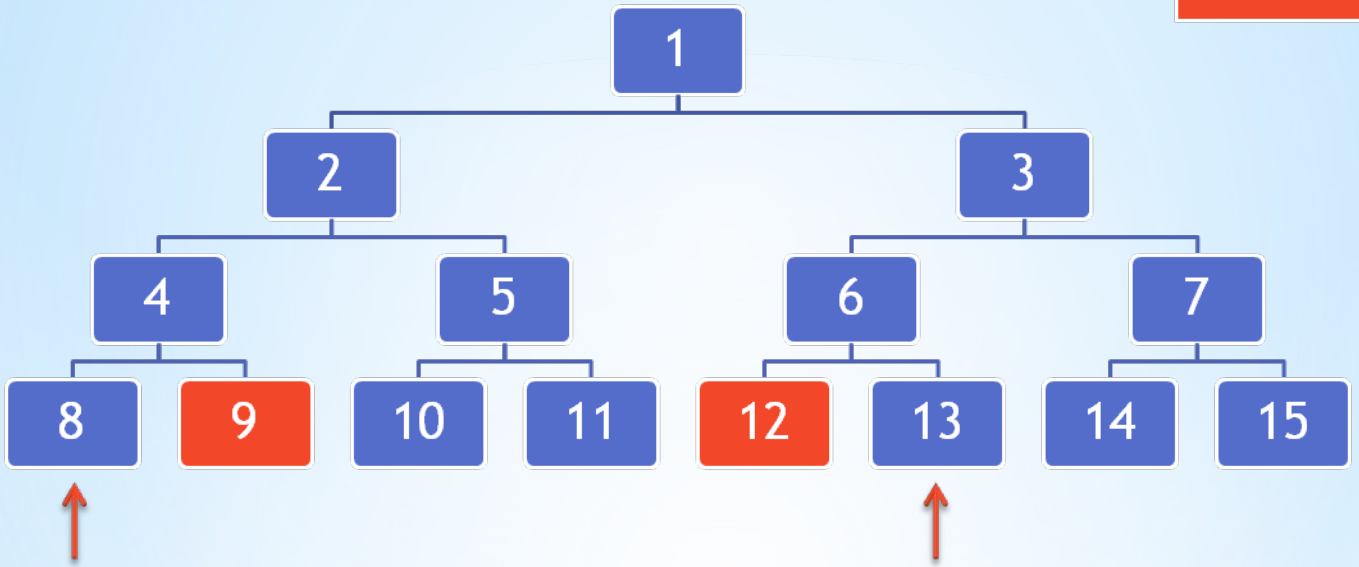
* ……
* 直接找到叶子？
* 无限遐想中……

* 好多性质啊，有用么？

*存下来了，然后呢？

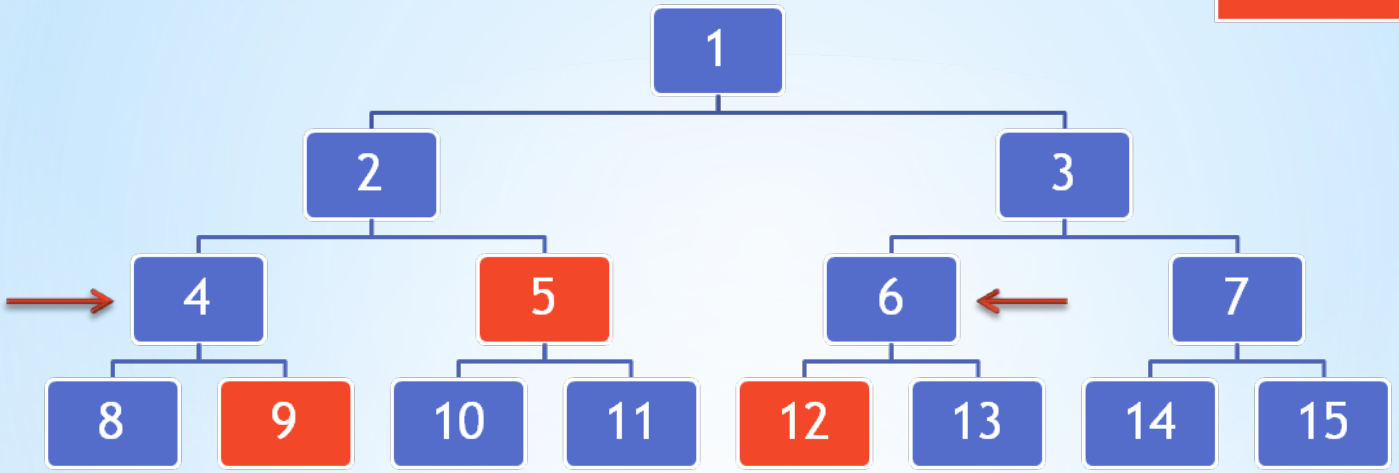
可以直接找到叶子？

(0,5)?



*天然的非递归方法！

(0,5)?



*天然的非递归方法！

* Func Query(s,t) // 询问从 s 到 t 闭区间
  * s = s - 1, t = t + 1; // 变为开区间
  * s += M, t += M; // 找到叶子位置
  * While not ((s xor t) == 1) do
    * If ((s and 1) == 0) Answer += Tree[s + 1];
    * If ((t and 1) == 1) Answer += Tree[t - 1];
    * s = s >> 1, t = t >> 1;

* 这么简单？

```
*for (s=s+M-1,t=t+M+1;s^t^1;s>>=1,t>>=1) {
   *if (~s&1) Ans+=T[s^1];
   *if (  t&1) Ans+=T[t^1];
*}
```

*C语言更简单！

* 在将闭区间转化成开区间的时候可能越界 1
* 理论上能放 [0,2^N) 的树
* 其实只能查询 [1,2^N - 2] 的范围
* 切记切记

* Warning

* 如果需要查询 0 就整个向后平移一下
* 所有下标加一！

* 如果需要在 [0,1024) 中查询 1023 结尾的区间？
* 慢！你的数据规模不是 1000 么？
* ……
* 如果真的要到 1023，直接把总区间变成 [0,2048)
* 就是这么狠！

* 不要紧张

* Func Change(n,NewValue)
  * n += M;
  * Tree[n] = NewValue;
  * While n > 1 do
    * n = n >> 1;
    * Tree[n] = Tree[2n] + Tree[2n+1];

* 修改就更简单

* for(T[n+=M]=V, n>>=1;n;n>>=1)
  * T[n]=T[n+n]+T[n+n+1];

* 没了？
* 没了。

* C语言更简单

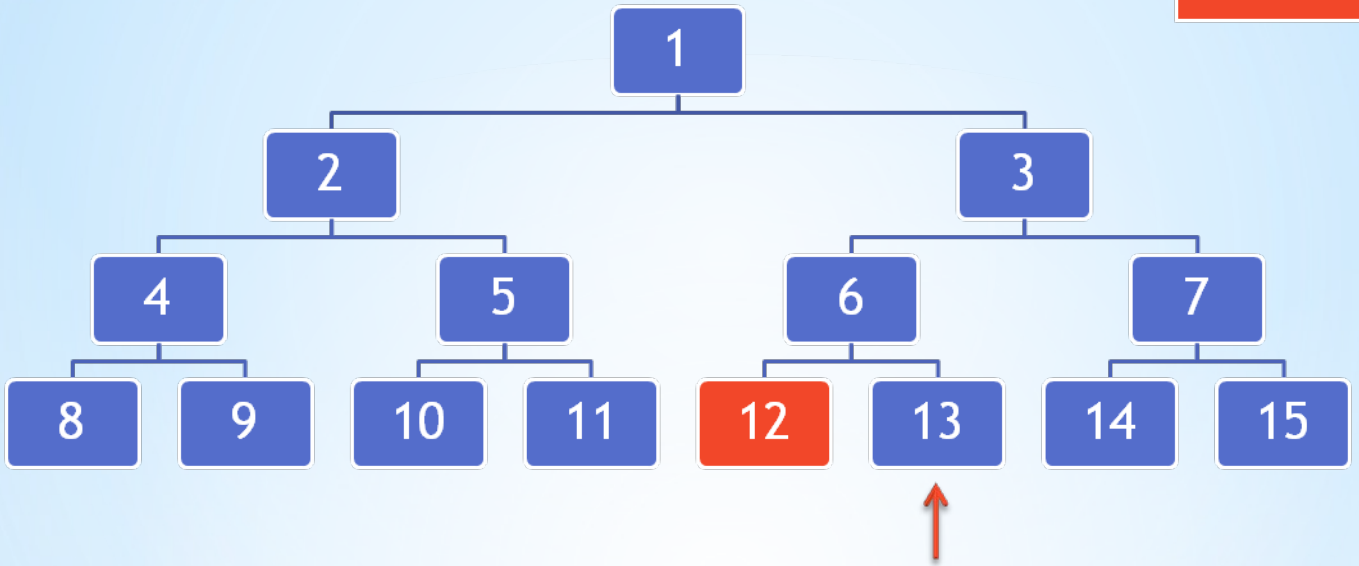
* 仅使用了两倍原数组的空间
* 其中还完整地包括了原数组
* 构造容易：
  * For i=M-1 downto 1 do T[i]=T[2i]+T[2i+1];
* 太好写了！好理解！
* 自底向上，只访问一次，而且不一定访问到顶层
* 实践中非常快，与树状数组接近
* 为什么呢？后面再讲。

* 技术参数？

* 因为树状数组依赖于区间减法
* 区间减法什么的，最讨厌了……后面再讲，再讲
* 反正如果只问问前缀和，不问区间和的话
* 那我也可以只用一倍空间！

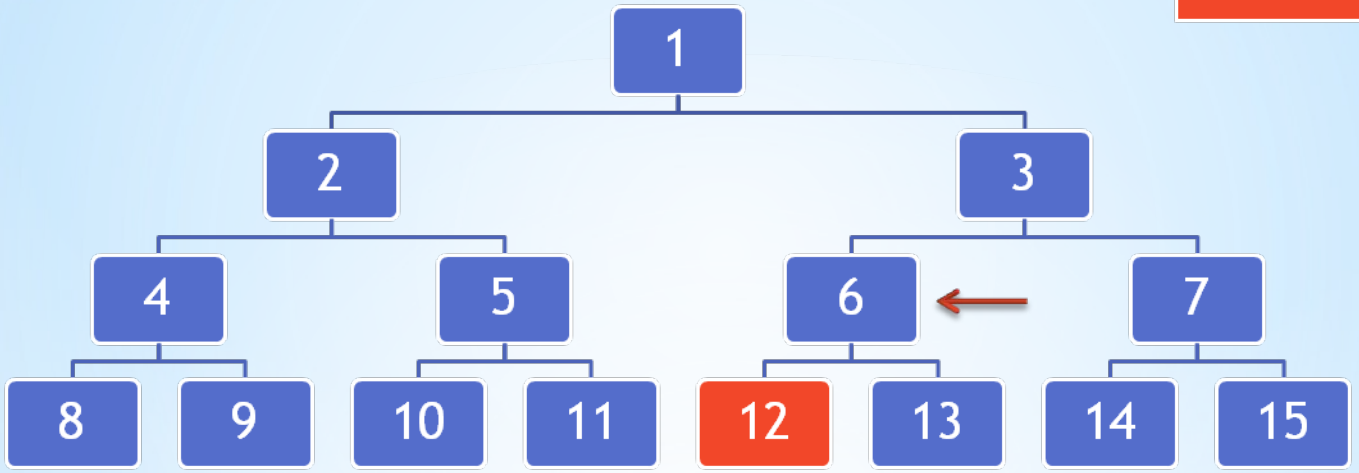
* 人家树状数组只用一倍空间

(…,5)?



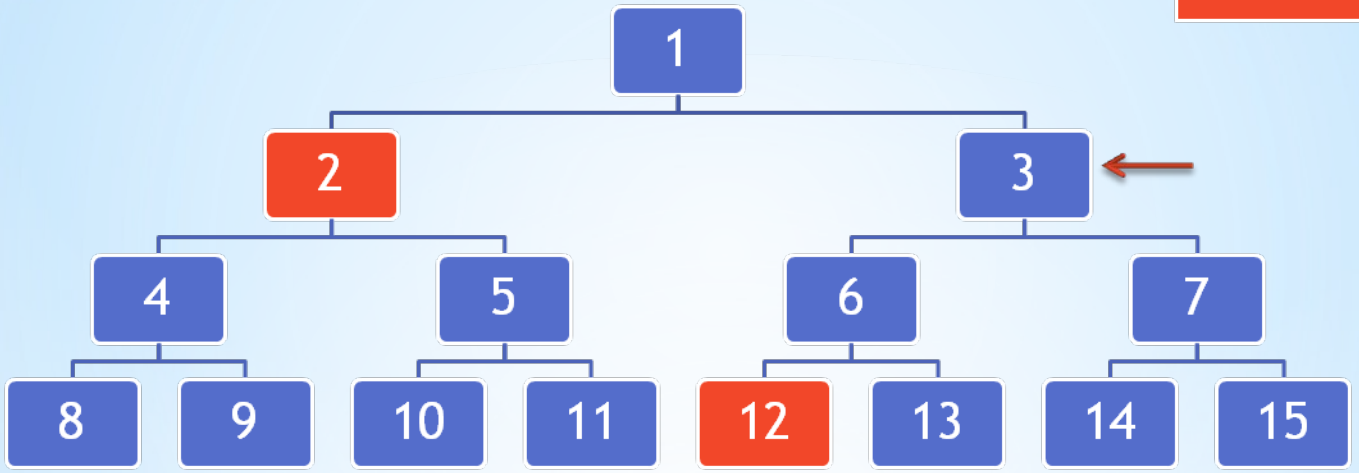
*天然的非递归方法！

(…,5)?

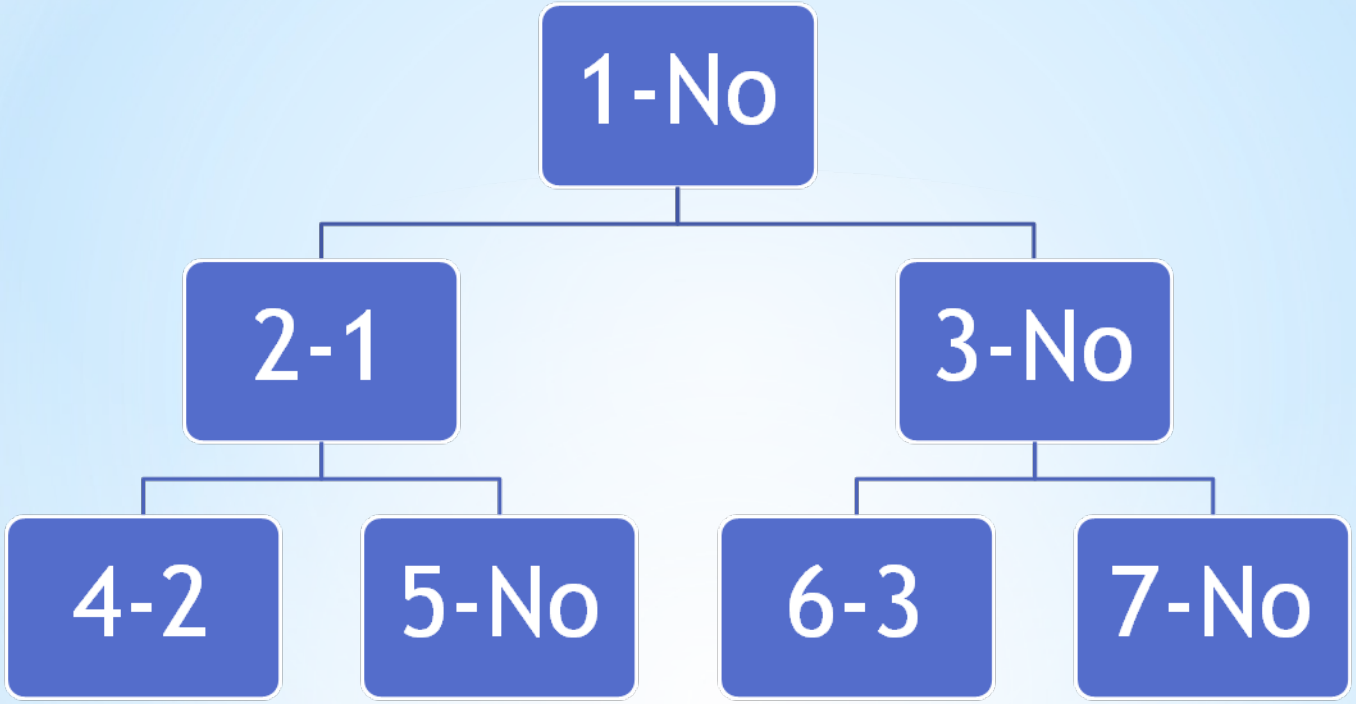


*天然的非递归方法！

(…,5)?



*天然的非递归方法！



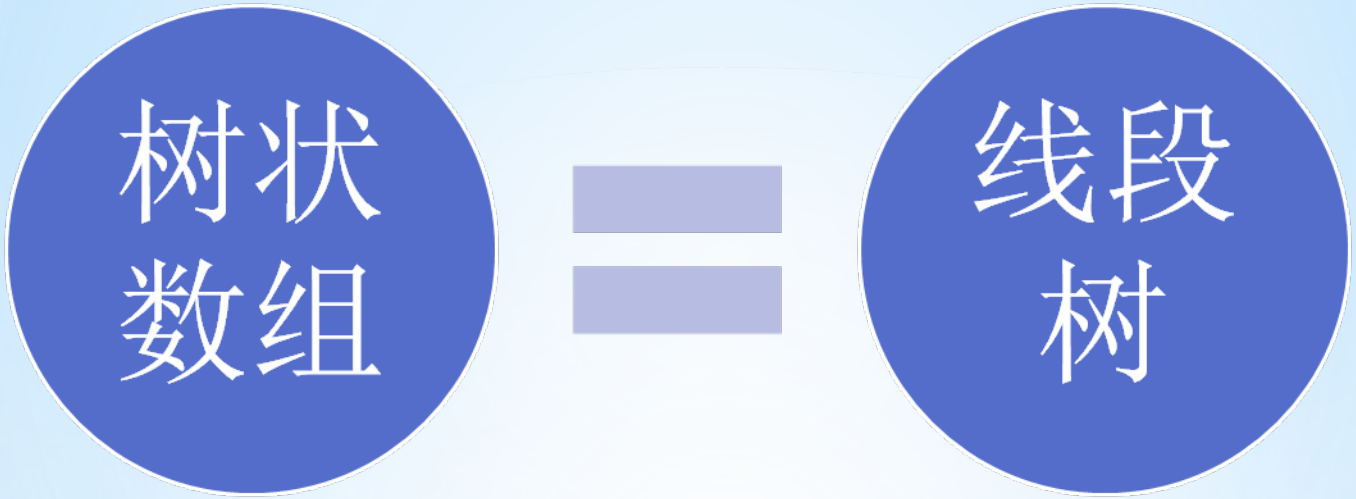
* 所有右儿子没有用了

* 这不就和树状数组一样了？
* 本来就应该一样。

* 回过头说，树状数组究竟是什么？
* 就是省掉一半空间后的线段树加上中序遍历
* 计算位置需要 lowbit 什么的
* 我们用的是先序遍历，符合人的思考习惯。

* 省了一半空间吧

但是，这个空间没必要省。
费点空间能换来实现的绝对简单。



*哈哈

* 我之前用这种写法做过不少题……
* 大家都说我的代码看不懂
* 我说这就是一个树状数组
* 写树状数组的人说没有 lowbit
* 我说那就算是线段树吧
* 大家不相信非递归的线段树这么短……
* 我：……

* Just For Fun

*标记的传递与收集

懒惰即美德。

*区间修改

噩梦的开始

* RMQ (Range Minimum Query)
* 区间最小值查询
* 线段树
* 支持区间修改：
  * 某一区间的数值全部增加一个可正可负的数
* 暴力修改不灵了！

* 带区间修改的RMQ

* 在线段树上的每个节点增加一个标记
* 表示这一区间被增加过多少
* 在自顶而下的递归过程中
* 如果看到标记，就更新当前节点的值
* 并将标记下传

* 嗯？又要自顶向下？

* 四两拨千斤

* 其实堆式存储也可以自顶向下访问
* 就是上下各走一次而已
* 但是我们有更好的办法（使劲想想就知道了）
* **不再下传标记**，而是随用随查
* 在自底向上的查询过程中
* 每向上走一层，就按照对应的标记调整答案
* 仔细想想很有道理。我们愿意把尽可能多的信息存放在树的根部，所以下传标记做什么？

* 标记永久化

常数很小：One Pass

永久化的标记就是值

庄周梦蝶而已

染色问题

- 一根线段，支持区间染色。
  询问区间是否同色？
- 区间染色需要使用染色标记
  1 表示红色， 2 表示蓝色， 3 绿色， 0 杂色
- 询问的时候就直接看标记嘛

* 2 为红色， 3 为蓝色， 1 为杂色
  修改 3 为红色
  查询 1 ，杂色？
* **永久化的标记就是值啊！**值是自动维护的啊！
* 其实我们的修改算法在修改 3 的时候已经维护了 1
* 自底向上顺便重算所有遇到的节点标记即可
* If (Mark[x]==0 and Mark[2x]==Mark[2x+1])
  * Mark[x]=Mark[2x];

* 直接看标记？

* 回到区间修改的 RMQ
* 通俗地讲，标记就是一个相对的量
* 既然有了标记，值还有什么用？
* 或者说，如果值本身就是相对的，还需要标记？
* 如果我让所有的数都从零开始变化，
  那标记永久化之后，所有值都恒为零啊！
* 于是我们连值也不存了。永久化的标记就是值。

* 狗拿耗子，猫下岗了

* 每个节点不存区间最大值 T[n] 了
  存放 M[n]=T[n]-T[n>>1]
* 让每一个节点的值都减除它父亲的值
* 区间修改就直接改 M[n] 。
* 查询就是从要查的节点开始一直加到根。
  T[n]=M[n]+M[n>>1]+...+M[1];
* 遇到节点 x ， 则 A=min(M[2x],M[2x+1])
  M[x]+=A,M[2x]-=A,M[2x+1]-=A

* 什么意思？

```
Func Add_x(s,t,x)
  for (s=s+M-1,t=t+M+1;s^t^1;s>>=1,t>>=1) {
    if (~s&1) T[s^1]+=x;
    if (  t&1) T[t^1]+=x;
    A=min(T[s],T[s^1]),T[s]-=A,T[s^1]-=A,T[s>>1]+=A;
    A=min(T[t],T[t^1]),T[t]-=A,T[t^1]-=A,T[t>>1]+=A;
  }
```

* 简单……

* Func Max(s,t)
  * for (s=s+M-1,t=t+M+1;s^t^1;s>>=1,t>>=1) {
    * LAns+=T[s],Rans+=T[t];
    * if (~s&1) LAns=max(LAns,T[s^1]);
    * if (t&1) RAns=max(RAns,T[t^1]);
  * }
  * Ans = max(LAns,RAns);
  * while (s>1) Ans += T[s>>=1];

* too 简单，too

* 差分是化绝对为相对的重要手段
* 标记永久化后就是值，只不过这种值是相对的
* 计算答案时可以利用从节点到根部的信息

* 启示

* 截至 PPT 制作时，大家用的线段树自顶向下居多
* 在自顶向下的线段树中，标记往往不是永久化的
* 对于 RMQ，需要下传标记
* 对于染色问题，需要下传并收集标记
* 思想与这里我的方法差不多，实现上差别较大
* 至少上下各访问一次，较慢
* 参见其他资料

*alternative

其实什么……现在可以讲了

* 线段树是连接原数组和前缀和的桥梁
* 桥梁两边同时取差分
* 前缀和与差分互为逆运算
* 因此线段树也是连接差分和原数组的桥梁

* 如果要支持区间修改，单点查询
* 无需使用标记，直接将原数组差分
* 用线段树维护差分数组的前缀和即可。

* 还一个欠账

* 不借助标记，支持区间修改和区间求和（原创）
* 先差分后变成维护一个前缀和的前缀和.....
* 别被吓到，前缀和的前缀和是什么
* $SS_1 = S_1 = x_1$
* $SS_2 = S_1 + = 2x_1 + x_2$
* $SS_3 = S_1 + S_2 + S_3 = 3x_1 + 2x_2 + x_3$
  $= 4(x_1 + x_2 + x_3) - (1x_1 + 2x_2 + 3x_3)$
* 多维护一个 $\{nx_n\}$ 的前缀和就行了。

* 前缀和的前缀和？

数学啊数学！

* 最长上升“区间列”
* 在一个区间列中按顺序找出最多区间
  使得不重叠，单调增
* 如 [1,3] [2,4] [4,5] 答案是 [1,3]+[4,5]
* 动态规划的可行决策是什么呢？
  如果要使上升列长度大于 x，
  最后一个数最小是多少，记为 f[x]
* 维护 f[x] 支持点查询和区间修改。

* 最长上升区间列

* 点查询：查询 x 处 f[x] 的值
* 区间修改： x 左边的所有超过 K 的值，变为 K
* 把 x 的左右换一下……（囧）
* 整个 f[-x] 就是单调减的
* 一个单调减的序列可以看作
  是由一个普通序列经过前缀 min 得到的！
* 前缀 min 的逆运算是什么呢？
* 我们并不关心

* 前缀min的逆运算

* 我们现在要维护的就是
  前缀 min 的逆运算后的原序列！
* 可是我们甚至不知道前缀 min 的逆运算是什么
* 不要紧，反正用不到。
* 点查询：查询 x 处 f[x] 的值
  直接返回维护序列的前缀 min
* 区间修改： x 左边的所有超过 K 的值，变为 K
  把维护序列中的 f[x] 变为 K

* 这样也行？

*线段树，太灵活了！

*但是不要迷信线段树

不要迷恋哥，哥只是个传说……

* 说了这么多，能使用线段树解决问题的关键：
* 区间加法，即区间的“性质”由子区间完全决定
* 包括但不仅限于求和，求最值，求染色状态
* 这里的“性质”有点像动态规划的状态表示
* 有时候，求的更多反而更容易
* 最简单的例子：求区间第二最值
* 如果实在不满足区间加法，就全完了

* 重要条件：区间加法

* 我们都知道线段树求区间平均值不难
* 那求一个区间中位数试试？
* 什么，还不难？
* 那你再求个众数？
* ……

* 不满足区间加法？

* 越来越难的原因很简单
* 知道两区间的中位数，就知道和区间的中位数？
* 知道各自众数有什么用？
* ……

* 不满足区间加法！

* 给定一列数，反复求区间第 k 大数。
* 要求的更多反而更容易……
  更容易……
* 线段树的每个区间必须保留更多的信息！
* 每个区间中存下区间所有数的有序数组
* 通过归并完成区间加法

* 超越中位数
K-th number

* 如果每做一次查询就归并若干个线段
* 复杂度就会达到 O(n)

* 离散化！二分答案！
* 变为求： x 是区间第几大数？
* 这可以分别二分查找若干线段得到。
* 总复杂度 $O(n\log n+Q*\log^2 n)$

* 归并很慢

另一种原创方法，后面再讲

* 如果有了区间减法……
* 线段树就能如虎添翼
* 如“区间和”变成“前缀和”
* 操作能简单不少
* 同时也是能够使用树状数组的条件
* 但这不是必需的（和区间加法比一比）

* 区间减法

另一种核心思想

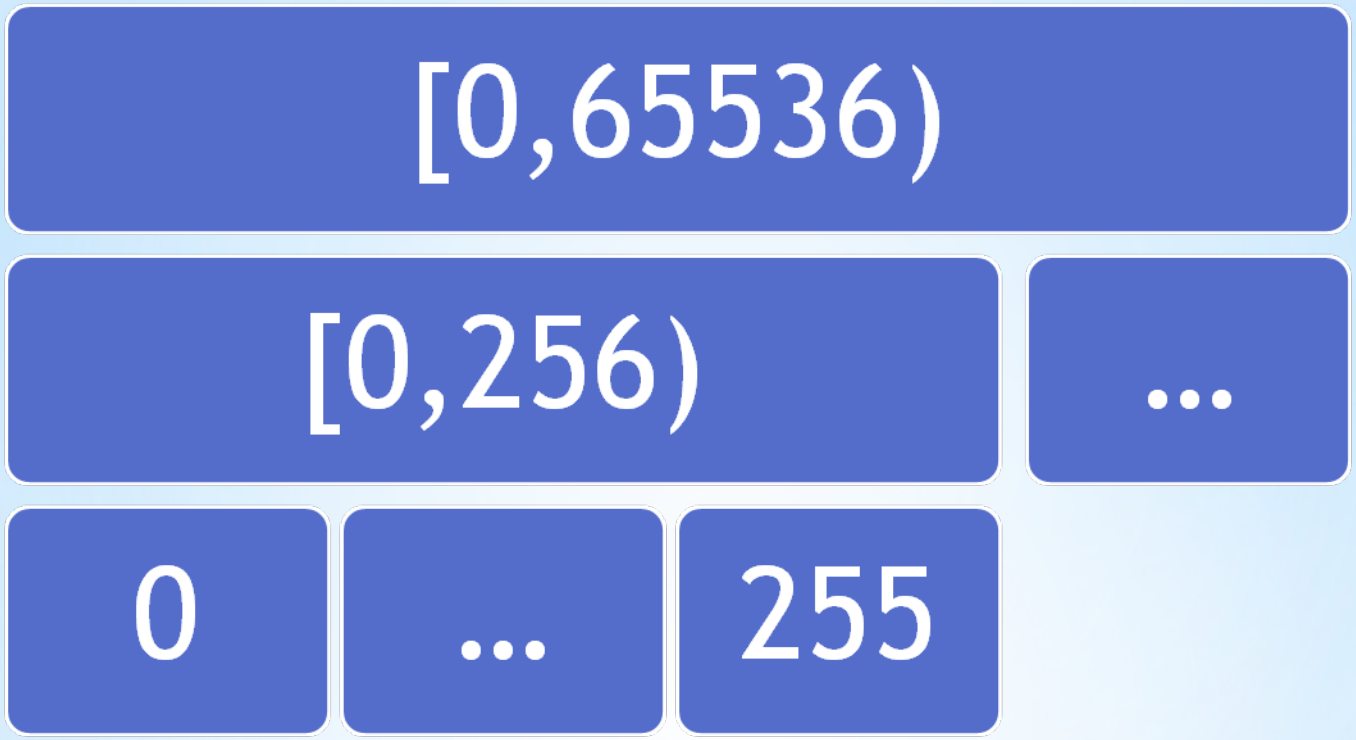
我说过后面要讲的嘛

* 维护一个数据结构支持
  * 整数插入
  * 取最大
  * 整数范围是 0~65535 好了

* 堆？

* 堆当然可以
* 但是刘汝佳老师的黑书上有大招！
* “分段哈希”……
* 分成若干段，存下“段里面有没有数”信息

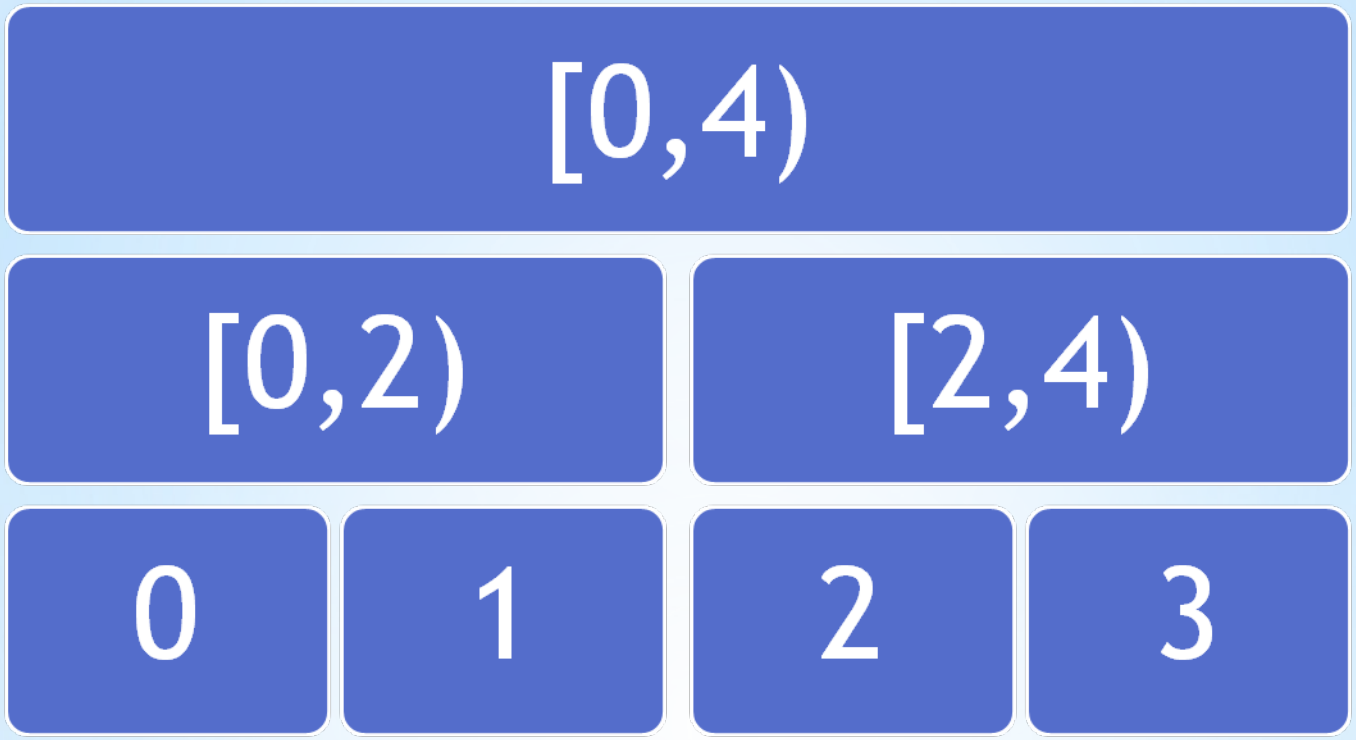
* 刘汝佳老师的大招



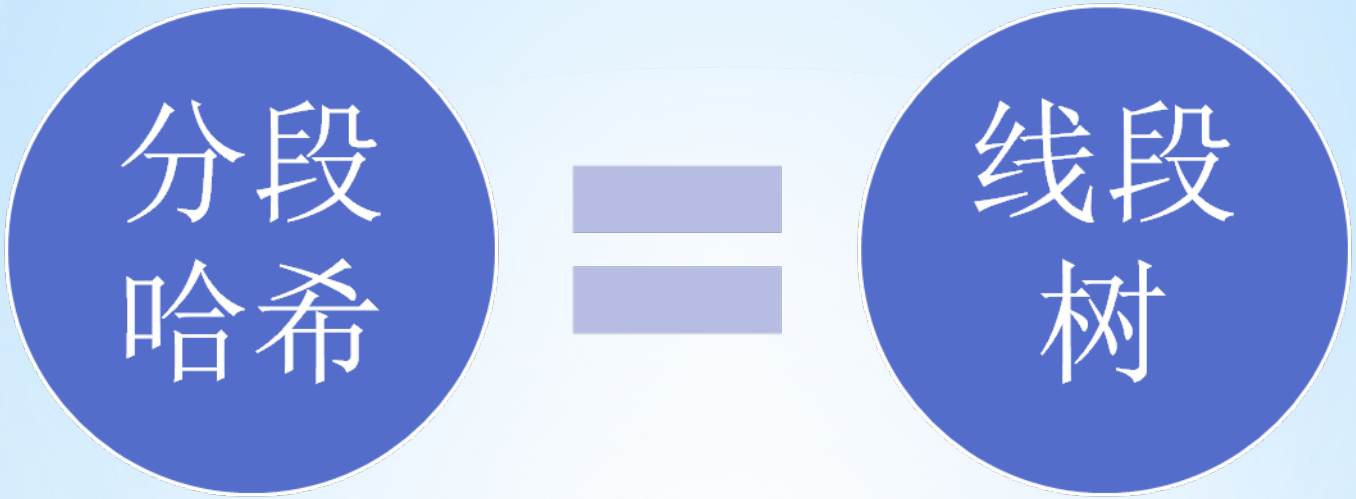
*分段哈希

* 如果多来几层呢？
* 3 层？ 4 层？……
* 到每个节点下面都只有两个点为止！
* ……
* 我们得到了什么……

* 多来几层如何



*这也是线段树



*哈哈

*平衡树 vs 线段树

不要折腾……

* 维护一个数据结构支持
  * 整数插入
  * 整数删除
  * 取第 k 大的数（取最大最小什么的就不说了）
  * 查询数的排名
  * 查某数是否存在
  * 允许元素重复（为了难一点）
  * 整数范围是 0~65535 好了

* 一种似曾相识的感觉

* 平衡树么？线段树！
* 统计 [0,65536) 每个数的出现频率，记为 f[x]
  * 整数插入， f[x]++
  * 整数删除， f[x]--
  * 查询数的排名，求前缀和
  * 取第 k 大的数（取最大最小什么的就不说了）
    给定前缀和，查找
    自顶向下，左边不够就向右走，否则向左。

* 统计的力量！

* 实测得到线段树用来处理这类问题非常快
* 平衡树中最快的 Size Balanced Tree 用了 4 秒多
* 线段树不到半秒全部出解。
* 这还没有分别减去读入数据的时间。
* 线段树使用刚刚所讲的实现，
* 代码量极小，且调试非常简单。

* 事半功倍

* 离散化。

* 不能离散化？
* 呵呵……后面再讲

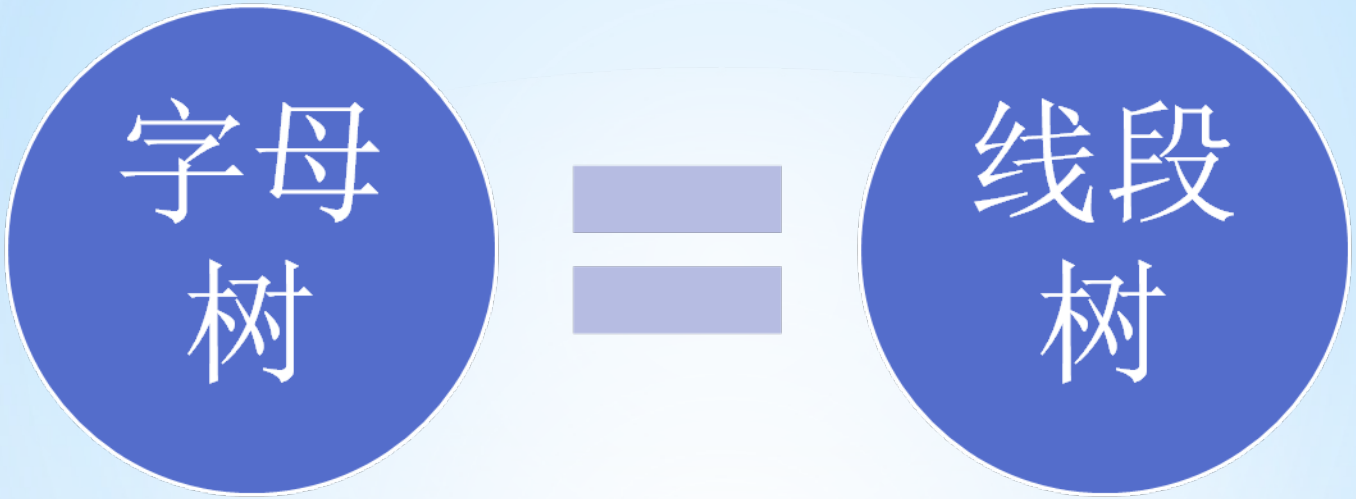
* 如果数据范围大呢？

* 维护一个数据结构支持
  * 字符串插入
  * 字符串删除
  * 取第 k 大的字符串（取最大最小什么的就不说了）
  * 查询字符串的排名
  * 查某字符串是否存在

* 一种似曾相识的感觉

* 这里的 size 域的维护方式
* 和线段树如出一辙！
* 排名的计算方法，和之前整数的线段树完全一样
* 如果把字符串看作 26 进制数
* 那字母树就是线段树？

* 带size域的字母树



*哈哈

* 所有节点用指针记录儿子
* 空间随用随开
* 不是满二叉树，甚至不完全
* 自顶向下处理所有问题

* 线段树也可以这样
* 数据范围再大，能比 26^N 还大？

* 那为啥字母树省空间？

* 就是一棵三十二层高的线段树
  指针式存储，节点像字母树一样动态生成
* 支持插入删除选择等等……
* 复杂度是 O(NlogM)， M 是最大的数
* 对于 long int， M=32
* 实测用了一秒多
* （还记得平衡树四秒多么？）
* 自顶向下，只算前缀和，也不难写

* 线段树处理long int

不就是个二进制的字母树？

* 像压缩字母树一样，会节约大量空间
  代价：不好写了
* 删除一个数之后尝试回收已经分配的节点
  代价：略慢，不好写了
* 树高动态化
  初始树高是 1，只能放 0
  每一次如果要放的数太大，增加一个根
  根的左儿子是当前的根，右儿子空。
  这个还实用！

* 可能的改进

*平衡树 with 线段树

在天愿作比翼鸟，在地愿为连理枝

* 平衡树上很多信息可以像线段树一样维护

* 平衡树就是一个会旋转的动态线段树！

* 最简单的，比如 size 域

* 记得Size域么？

* 块状链表的伤心题，标准程序 5k+
* 维护一个数列，支持：
  * 区间增加一个数
  * 区间删除
  * 区间插入
  * 区间求和
  * 区间翻转

* NOI2005 维护数列

* 平衡树 splay 可以支持：
  * 区间删除
  * 区间插入
* 线段树可以支持
  * 区间增加一个数
  * 区间求和
* 把线段树的值放在平衡树的节点上

* 平衡树与线段树

* 每个节点表示它和子树的信息总和
* 平衡树旋转时更新线段树的域
* 哇，会动的线段树……

* 转起来的线段树

* 既要区间修改又要区间求和
  使用自顶向下的标记下传即可
* 为了处理区间反转
  增设一个 bool 值表示当前节点左右子树已经互换
* 先把区间从树中 splay 出再处理
* 要同时更改所有节点的反转标记？
  不记录反转标记，记录反转标记的标记（！）

* 标记啊标记

* 但是所有部分都是相对简单的！

* 一点点写，只是很多很多小题而已……

* 好综合的一道题

*返璞归真

关于线段树，我们讲的已经太多了……

* K-th number 的另一个方法
* 如果区间互不包含，
  将所有要求的区间排个序来算。
* 用平衡树或线段树存下当前区间中的数
* 然后向下一个区间移动
* 左端点增加是数的删除
* 右端点增加是数的添加
* 每个数进出各一次而已

* 再还一个欠账

* 关键在于合理的计算方式
  使得相邻区间的差异尽量小
* 从一个区间变为另一个区间的代价是多少？
* 把区间看作二维平面的坐标
* 代价就是两个平面点的 Manhattan 距离！
* 然后呢？ Hamilton 路？
* 不！一个已知的区间可以用来算很多个未知的！
* 平面图 Manhattan 距离最小生成树。

* 如果相互可以包含呢？

* 平面图 Manhattan 距离 MST 可以在 O(nlogn) 求出
* 先对 Q 个问题用这个方法处理
* 再按照 MST 的顺序和方法实际计算
* 求数学大牛分析总复杂度

* 虽然绕了很多弯路，但是有一种用模型解决实际问题的感觉。居然 MST 还能用来做预处理呢……

* 然后呢？

* 听了这么久，一起做一道练习题吧.....
* 给定一列 n 个数，和 m 个区间，求每个区间里的众数出现了多少次。
* 对于 10% 的数据 n<100, m<100
  对于 30% 的数据 n<1000,m<1000
  对于 50% 的数据 n<100000,m<100000
  对于 70% 的数据 n<1000000,m<1000000
  以上数据中区间互不包含。
* 对于其余 30%， n<10000,m<10000，区间可包含

* 最后的硬骨头，众数

谢谢大家

E-mail:aceeca.135531@gmail.com