平衡树

zjp_shadow

2023 年 12 月 2 日

平衡树的前置——二叉搜索树

二叉搜索树

- 空树是二叉搜索树。
- 若二叉搜索树的左子树不为空，则其左子树上所有点的附加权值均小于其根节点的值。
- 若二叉搜索树的右子树不为空，则其右子树上所有点的附加权值均大于其根节点的值。
- 二叉搜索树的左右子树均为二叉搜索树。

二叉搜索树上的基本操作所花费的时间与这棵树的高度成正比。对于一个有 n 个结点的二叉搜索树中，这些操作的最优时间复杂度为 $\mathcal{O}(\log n)$，最坏为 $\mathcal{O}(n)$。随机构造这样一棵二叉搜索树的期望高度为 $\mathcal{O}(\log n)$。

BST 的基本操作

- 遍历二叉搜索树：由二叉搜索树的递归定义可得，二叉搜索树的中序遍历权值的序列为非降的序列。时间复杂度为 $\mathcal{O}(n)$。
- 查找最小/最大值：由二叉搜索树的性质可得，二叉搜索树上的最小值为二叉搜索树左链的顶点，最大值为二叉搜索树右链的顶点。时间复杂度为 $\mathcal{O}(h)$。
- 搜索元素：在以 root 为根节点的二叉搜索树中搜索一个值为 value 的节点，在每个点处进行分类讨论即可。时间复杂度为 $\mathcal{O}(h)$。插入，删除，修改都需要先在二叉搜索树中进行搜索。
- 插入元素：找到对应元素的节点后，就 $+1$，如果为空就新建。
- 删除元素：找到对应元素的节点后，然后 -1，如果为 0 那么找到它对应的 pre 或者 suf 替换它。

BST 的基本操作

- 求元素的排名：
  排名定义为将数组元素升序排序后第一个相同元素之前的数的个数加一。
  查找一个元素的排名，首先从根节点跳到这个元素，若向右跳，答案加上左儿子节点个数加当前节点重复的数个数，最后答案加上终点的左儿子子树大小加一。
- 查找排名为 k 的元素：
  在一棵子树中，根节点的排名取决于其左子树的大小。
  - 若其左子树的大小大于等于 k，则该元素在左子树中；
  - 若其左子树的大小在区间 $[k - count, k - 1]$（count 为当前结点的值的出现次数）中，则该元素为子树的根节点；
  - 若其左子树的大小小于 $k - count$，则该元素在右子树中。

平衡树简介

使用搜索树的目的之一是缩短插入、删除、修改和查找（插入、删除、修改都包括查找操作）节点的时间。
关于查找效率，如果一棵树的高度为 h，在最坏的情况，查找一个关键字需要对比 h 次，查找时间复杂度（也为平均查找长度 ASL，Average Search Length）不超过 $\mathcal{O}(h)$。一棵理想的二叉搜索树所有操作的时间可以缩短到 $\mathcal{O}(\log n)$（n 是节点总数）。
然而 $\mathcal{O}(h)$ 的时间复杂度仅为理想情况。在最坏情况下，搜索树有可能退化为链表。想象一棵每个结点只有右孩子的二叉搜索树，那么它的性质就和链表一样，所有操作（增删改查）的时间是 $\mathcal{O}(n)$。
可以发现操作的复杂度与树的高度 h 有关。由此引出了平衡树，通过一定操作维持树的高度（平衡性）来降低操作的复杂度。

平衡性的定义

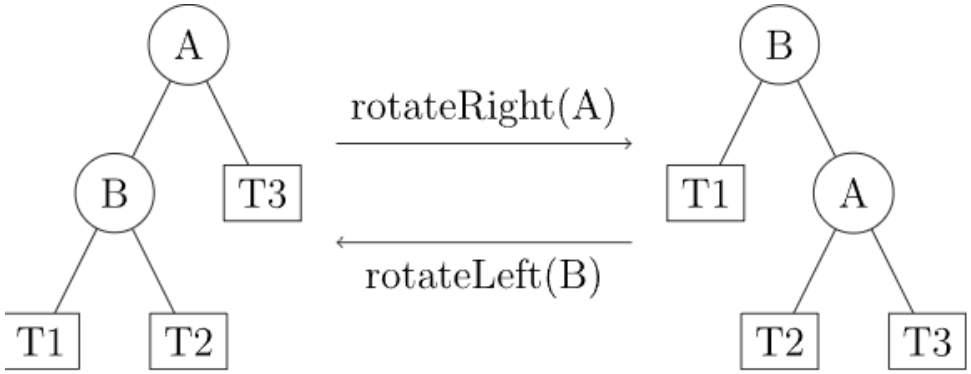
关于一棵搜索树是否「**平衡**」，不同的平衡树中对「**平衡**」有着不同的定义。比如以 T 为根节点的二叉搜索树，左子树和右子树的高度相差很大，或者左子树的节点个数远大于右子树的节点个数，这棵树显然不具有平衡性。平衡状态不一定唯一。
对于二叉搜索树来说，常见的平衡性的定义是指：**以 T 为根节点的树，每一个结点的左子树和右子树高度差最多为 1** 。

- Splay 树：对于任意节点的访问操作（搜索、插入还是删除），都会将被访问的节点移动到树的根节点位置。
- AVL 树：每个节点 N 维护以 N 为根节点的树的高度信息。AVL 树对平衡性的定义：如果 T 是一棵 AVL 树，当且仅当左右子树也是 AVL 树，且 $|height(T \to left) - height(T \to right)| \leq 1$。
- Size Balanced Tree：每个节点 N 维护以 N 为根节点的树中节点个数 *size*。对平衡性的定义：任意节点的 *size* 不小于其兄弟节点的所有子节点的 *size*。
- B 树：对平衡性的定义：每个节点应该保持在一个预定义的范围内的关键字数量。

平衡的调整过程

关于二叉平衡树，平衡的调整操作分为包括 **左旋（Left Rotate 或者 zag）**和 **右旋（Right Rotate 或者 zig）**两种。由于二叉平衡树在调整时需要保证中序遍历序列不变。这两种操作均不改变中序遍历序列。

在这里先介绍右旋，对于结点 A 的右旋操作是指：将 A 的左孩子 B 向右上旋转，代替 A 成为根节点，将 A 结点向右下旋转成为 B 的右子树的根结点，B 的原来的右子树变为 A 的左子树。



图：二叉搜索树的右旋操作

平衡的调整过程

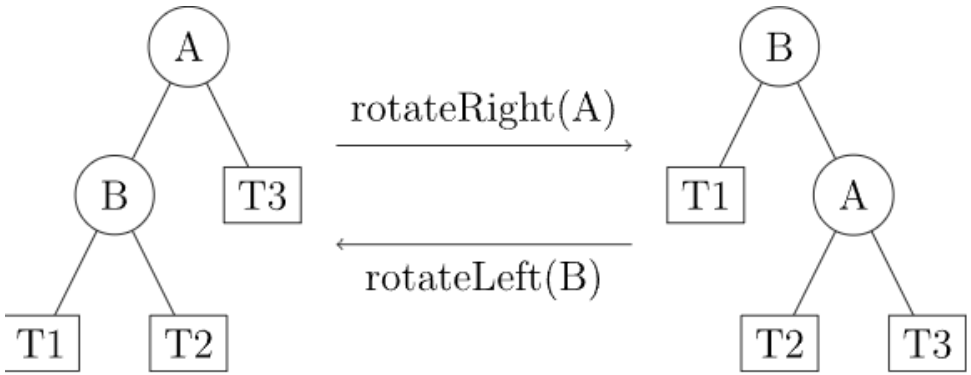


图: 二叉搜索树的右旋操作

右旋操作只改变了三组结点关联，相当于对三组边进行循环置换一下，因此需要暂存一个结点再进行轮换更新。
对于右旋操作一般的更新顺序是：暂存 B 结点（新的根节点），让 A 的左孩子指向 B 的右子树 $T2$，再让 B 的右孩子指针指向 A，最后让 A 的父结点指向暂存的 B。
完全同理，有对应的左旋操作，左旋操作与右旋操作互为镜像。

平衡性破坏的情况

虽然不同的二叉平衡树的定义有所区别，不同二叉平衡树区别只在于节点维护的信息不同，以及旋转调整后节点更新的信息不同。二叉平衡树平衡性被破坏的情况只有四种。进行平衡性调整的操作只包括左旋和右旋。

这里介绍两种，其他两种类似：T 的左孩子的左子树过长导致平衡性破坏。调整方式：右旋节点 T。

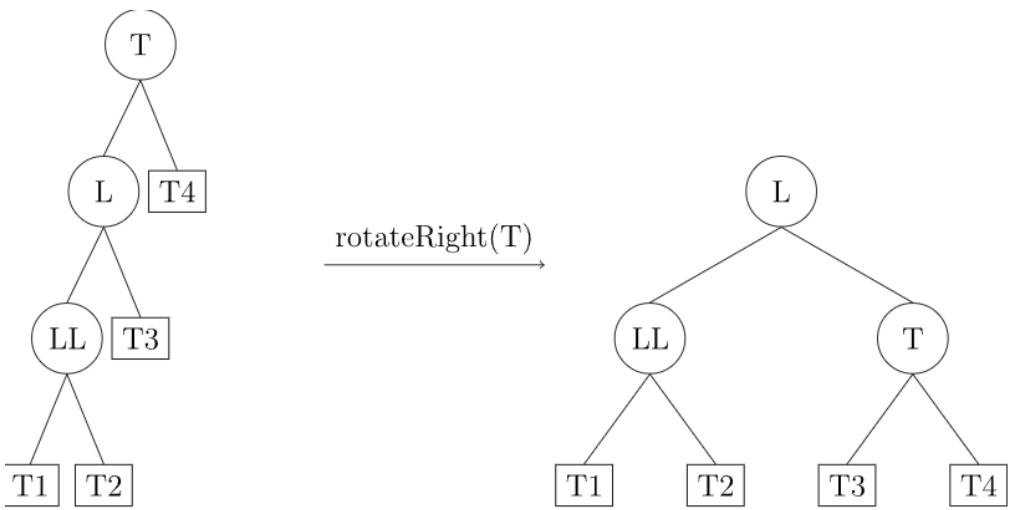
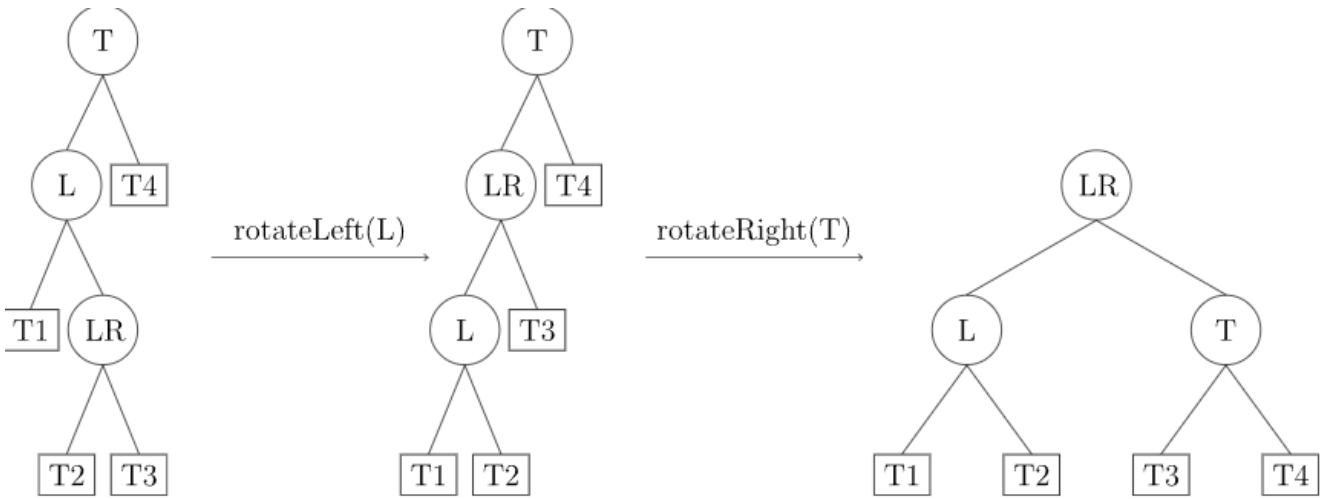


图: LL 型调整

平衡性破坏的情况

T 的左孩子的右子树过长导致平衡性破坏。
调整方式：先左旋节点 L，成为 LL 型，再右旋节点 T。



图：LR 型调整

Treap

Treap（树堆）是一种 **弱平衡的 二叉搜索树**。它同时符合二叉搜索树和堆的性质，名字也因此为 tree（树）和 heap（堆）的组合。其中，二叉搜索树的性质是：

- 左子节点的值（*val*）比父节点大
- 右子节点的值（*val*）比父节点小（当然这也是可以反过来的）

堆的性质是：

- 子节点值（*priority*）比父节点大或小（取决于是小根堆还是大根堆）

不难看出，如果用的是同一个值，那这两种数据结构的性质是矛盾的，所以我们再在搜索树的基础上，引入一个给堆的值 *priority*。对于 *val* 值，我们维护搜索树的性质，对于 *priority* 值，我们维护堆的性质。其中 *priority* 这个值是随机给出的。

Treap

Treap 通过随机化的 *priority* 属性，以及维护堆性质的过程，「打乱」了节点的插入顺序。从而让二叉搜索树达到了理想的复杂度，避免了退化成链的问题，期望深度为 $\mathcal{O}(\log n)$ 。

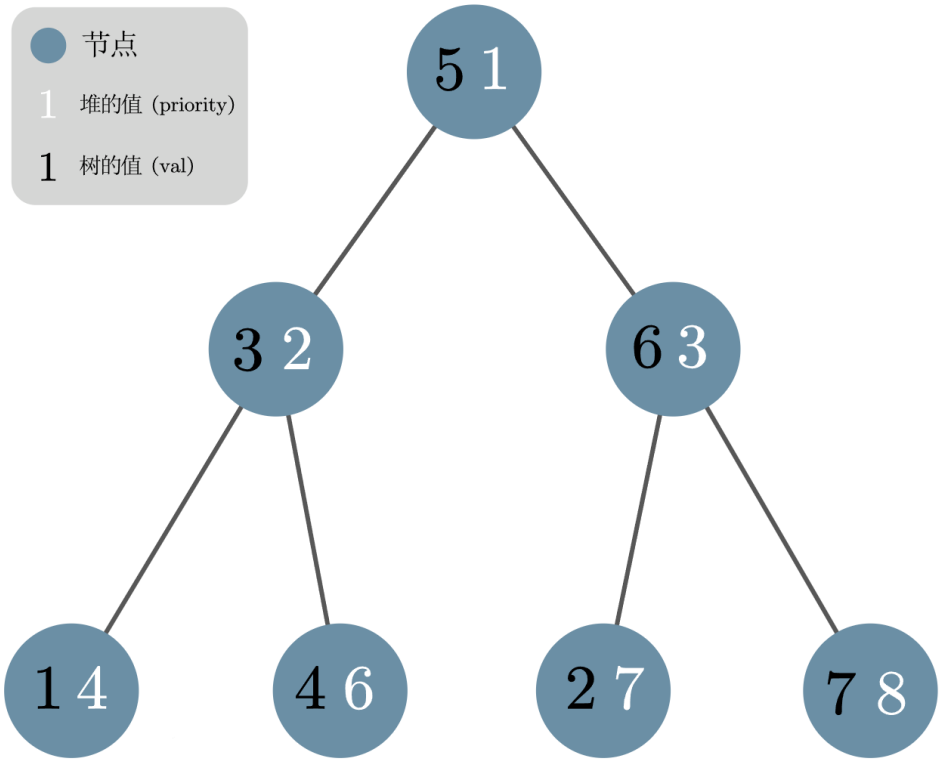


图: Treap

旋转 Treap

旋转 treap 维护平衡的方式为旋转，和 AVL 树的旋转操作类似，分为**左旋**和 **右旋**。即在满足二叉搜索树的条件下根据堆的优先级对 treap 进行平衡操作。
旋转 treap 在做普通平衡树题的时候，是所有平衡树中常数较小的。因为普通的二叉搜索树会被递增或递减的数据卡，用 treap 对每个节点定义一个由 `rand` 得到的权值，从而防止特殊数据卡。同时在每次删除/插入时通过这个权值决定要不要旋转即可，其他操作与二叉搜索树类似。
大部分的树形数据结构都有指针和数组模拟两种实现方法。

注意

代码中的 `rank` 代表前面讲的 *priority* 变量（堆的值）。并且，维护的堆的性质是小根堆（*priority* 小的在上面）。

旋转 Treap 的维护

旋转操作是 treap 的一个非常重要的操作，主要用来在保持 treap 树性质的同时，调整不同节点的层数，以达到维护堆性质的作用。
旋转操作的左旋和右旋可能不是特别容易区分，以下是两个较为明显的特点：

- 在不影响搜索树性质的前提下，把和旋转方向相反的子树变成根节点（如左旋，就是把右子树变成根节点）
- 不影响性质，并且在旋转过后，跟旋转方向相同的子节点变成了原来的根节点（如左旋，旋转完之后的左子节点是旋转前的根节点）

注意

有时候经常用 `rank` 代表前面讲的 *priority* 变量（堆的值）。并且，维护的堆的性质是小根堆（*priority* 小的在上面）。

旋转 Treap 插入

- **寻找插入位置**：按照二叉搜索树的规则找到插入点。
- **处理重复元素**：如果插入值已存在，增加该节点的重复计数。
- **维护堆性质**：如果新插入节点的 `rank` 小于父节点，执行旋转。
- **旋转操作**：
  - 如果新节点在左侧，进行右旋。
  - 如果新节点在右侧，进行左旋。
- **旋转后处理**：更新树的大小信息。

旋转 Treap 删除

- **定位要删除的节点**：遵循二叉搜索树属性找到目标节点。
- **处理重复元素**：若目标节点重复，减少重复计数。
- **叶子节点直接删除**：如果是叶子节点，直接删除。
- **有子节点的处理**：
  - 根据子节点的 `rank` 决定旋转方向。
  - 旋转以提升 `rank` 较小的子节点。
- **递归删除**：如果节点不是叶子节点，通过旋转和递归处理。
- **更新大小信息**：删除操作后更新树的大小。

[NOI2004] 郁闷的出纳员

你需要写一种数据结构，并维护以下 n 个操作（ k 已经给出）：

I. 给出 k，如果 $k \geq min$，则插入 k 。

A. 给定每个数加上 k 。

S. 给定每个数减去 k ，并删除所有元素 $< min$ 的数。

F. 查询第 k 大的数。

$0 \leq n \leq 3 \times 10^5, 0 \leq \min \leq 10^9$

[NOI2004] 郁闷的出纳员

- I 操作，单点插入，不过要判断能否加入，低于工资的最低值时，直接淘汰掉。
- 这里我们把 A,S,F 操作一起考虑：
  我们可以用一个 delta 把工资的调整记录下来。
  - 那么当我们添加一个新节点时添加的就不是 x，而是 x-delta，这样我们就可以直接利用 delta 进行 S 操作了。
  - S 操作时，我们调整 delta 后，我们就要把小于 minn-delta 的员工全部删掉，我们把树分裂后直接删掉左子树即可。
  - F 操作要注意的只有查询后要输出 val[x]+delta，这样才可以输出现在的值

  复杂度为 $\mathcal{O}(n \log n)$ 。

[THUPC2019] 不等式

已知两个 n 维实向量 $\vec{a} = (a_1, a_2, \ldots, a_n), \vec{b} = (b_1, b_2, \ldots, b_n)$，定义 n 个定义域为 $\mathbb{R}$ 函数 $f_1, f_2, \ldots, f_n$：

$$f_k(x) = \sum_{i=1}^{k} |a_i x + b_i| \quad (k = 1, 2, \ldots, n)$$

现在，对于每个 $k = 1, 2, \ldots, n$，试求 f_k 在 $\mathbb{R}$ 上的最小值。可以证明最小值一定存在。

$1 \le n \le 5 \times 10^5, |a_i|, |b_i| < 10^5$

[THUPC2019] 不等式

做一个简单转化，原式可以化为

$$\sum_{i=1}^{k} |a_i||x + \frac{b_i}{a_i}|$$

那么等价于将一个点转化为 $|a_i|$ 个点，考虑实际意义，一条数轴上有 $m = \sum |a_i|$ 个点，要求数轴上一点，使该点到各点的距离和最小。
所以用平衡树动态维护中位数就可以了，复杂度 $\mathcal{O}(n \log n)$ 。

无旋 Treap

无旋 Treap 又称分裂合并 Treap。它通过两种核心操作——**分裂**与**合并**——实现功能，这使得它在某些情况下比旋转 Treap 更加方便。

核心操作

- **分裂（Split）**：按照某种规则将一个 Treap 分裂成两个。
- **合并（Merge）**：将两个 Treap 合并成一个。

注释

无旋 Treap 也被称为 FHQ-Treap，由范浩强提出。这种数据结构支持可持久化操作和区间操作，详细内容可参照《范浩强谈数据结构》。

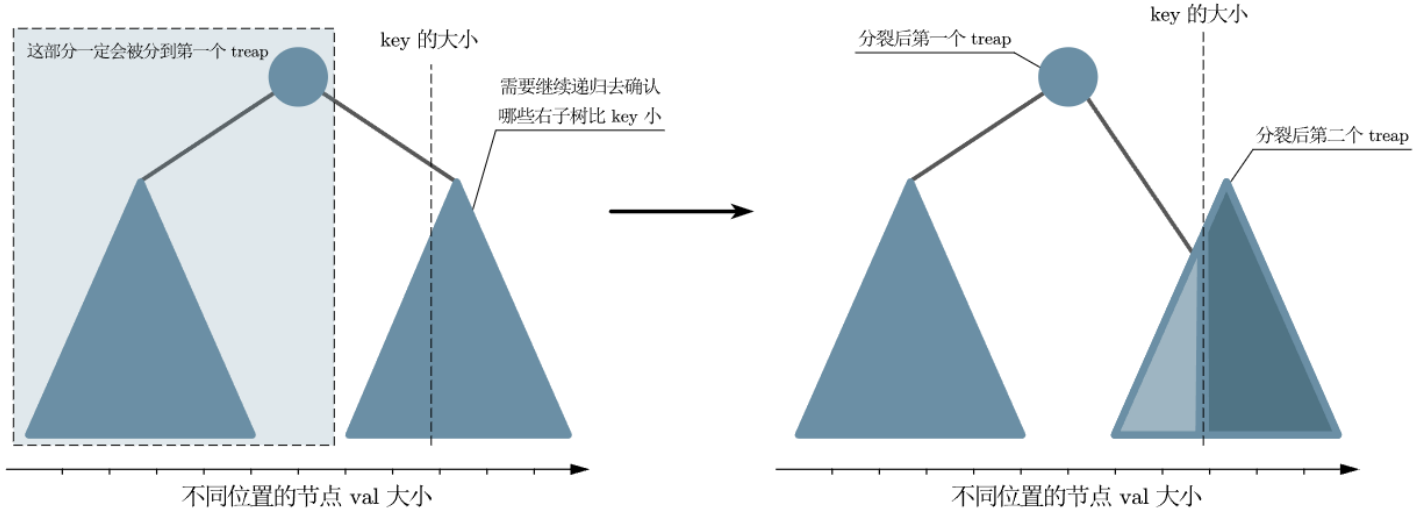
无旋 Treap - 按值分裂

分裂操作接受两个参数：根指针 *cur* 和关键值 *key*。分裂的结果是将 treap 分为两部分：

- 第一个 treap 包含所有值小于等于 *key* 的节点。
- 第二个 treap 包含所有值大于 *key* 的节点。

分裂逻辑

- 若 $key < cur$ 的值，则 *cur* 和其右子树属于第二个 treap。继续分裂左子树。
- 若 $key \geq cur$ 的值，则 *cur* 和其左子树属于第一个 treap。继续分裂右子树。



图：FHQ Treap

无旋 Treap - 按排名分裂

此操作基于排名来分裂 treap，并接受两个参数：节点指针 *cur* 和排名 *rk*。分裂的结果是三个不同的 treap：

- 第一个 treap 包含排名小于 *rk* 的所有节点。
- 第二个 treap 只包含排名等于 *rk* 的单个节点。
- 第三个 treap 包含排名大于 *rk* 的所有节点。

操作重点

关键在于确定排名等于 *rk* 的节点在树中的位置。这一过程与旋转 treap 中根据排名查询值操作类似，需要仔细处理递归分裂逻辑。

注意：与按值分裂的递归逻辑相似，但侧重于排名而非节点值。

合并（Merge）

合并操作接受两个参数：左 treap 的根指针 u 和右 treap 的根指针 v。必须满足 u 中所有节点的值小于等于 v 中所有节点的值。

合并逻辑

- 若 u 的根节点 *priority* 小于 v，u 成为新根节点。将 v 与 u 的右子树合并。
- 若 v 的根节点 *priority* 小于或等于 u，v 成为新根节点。将 u 与 v 的左子树合并。

注意：合并过程中，需保持搜索树的性质，并根据小根堆的性质确定新的根节点。

插入操作

插入操作在无旋 Treap 中可以通过分裂和合并来实现。这种方法虽然简洁，但可能比标准二叉搜索树的方法稍慢。

插入步骤

1. 根据插入值 val 分裂当前 treap，得到两个子树 T_1 和 T_2，满足 $T_1 \leq val$ 和 $T_2 > val$。
2. 再次分裂 T_1，根据 $val - 1$ 得到两个子树 $T_{1\text{ left}}$ 和 $T_{1\text{ right}}$，满足 $T_{1\text{ left}} \leq val - 1$ 和 $T_{1\text{ right}} = val$。
3. 如果 $T_{1\text{ right}}$ 非空，增加重复次数；否则新建节点。
4. 使用合并操作将分裂的子树重新组合。

注意：合并时需确保第一个子树的所有节点值小于第二个子树。

删除操作

在无旋 Treap 中，删除操作找到值等于 *val* 的节点，并将其删除。

删除步骤

- 首先按照 *val* 分裂 treap。
- 再分裂得到的左子树，得到的 $T_{1\ \text{left}}$ 和 $T_{1\ \text{right}}$。
- 如果 $T_{1\ \text{right}}$ 的重复次数大于 1，减少计数。
- 否则，删除 $T_{1\ \text{right}}$ 节点，并处理空节点情况。
- 使用合并操作将分裂的子树重新组合。

和插入操作基本相同。

无旋 Treap - 查询操作

根据值查询排名：

- 进行分裂操作：根据给定的值 $val - 1$ 分裂 Treap。
- 分裂后的第一个 Treap 包含所有小于 val 的节点。
- 计算排名：排名等于第一个 Treap 的大小加一（因为排名是比这个值小的节点数量加一）。
- 重组 Treap：查询后，将分裂的 Treap 合并回原来的状态。

根据排名查询值：

- 分裂操作：按排名 rk 分裂 Treap。
- 分裂得到三个 Treap，中间的 Treap 包含排名为 rk 的单个节点。
- 提取值：返回中间 Treap 的根节点值，即为所求的值。
- 重组 Treap：查询后，将分裂的 Treap 合并回原状态。

无旋 Treap - 查询操作

查询第一个比 *val* 小的节点：

- 分裂 Treap：根据 $val - 1$ 分裂，获取所有小于 *val* 的节点。
- 查找节点：在小于 *val* 的子树中找排名最大的节点，即值最大的节点。
- 重组 Treap：分裂后的子树重新合并。

查询第一个比 *val* 大的节点：

- 分裂 Treap：根据 *val* 分裂，获取所有大于 *val* 的节点。
- 查找节点：在大于 *val* 的子树中找排名最小的节点，即值最小的节点。
- 重组 Treap：分裂后的子树重新合并。

无旋 Treap - 建树操作

将一个有 n 个节点的序列 $\{a_n\}$ 转化为一棵 treap。
方法概述：

1. **暴力插入**：依次插入节点，分裂再合并，时间复杂度 $\mathcal{O}(n \log n)$。
2. **递归建树（方法一）**：选取区间中点作为根，钦定优先值，保证树高 $\mathcal{O}(\log n)$，同时满足堆性质。
3. **递归建树（方法二）**：选取区间中点作为根，随机优先级，保证树高 $\mathcal{O}(\log n)$，不必严格满足堆性质。
4. **笛卡尔树建树（方法三）**：利用笛卡尔树的 $\mathcal{O}(n)$ 建树方法，使用单调栈维护右链。

注意：不同方法适用于不同场景，选择方法时要考虑实际需求和数据特点。

无旋 Treap - 区间操作

无旋 Treap 的一大优点是能够有效实现区间操作，如区间翻转。以文艺平衡树的模板题为例，我们探讨如何建立初态 Treap 来维护有序数列。

建树过程

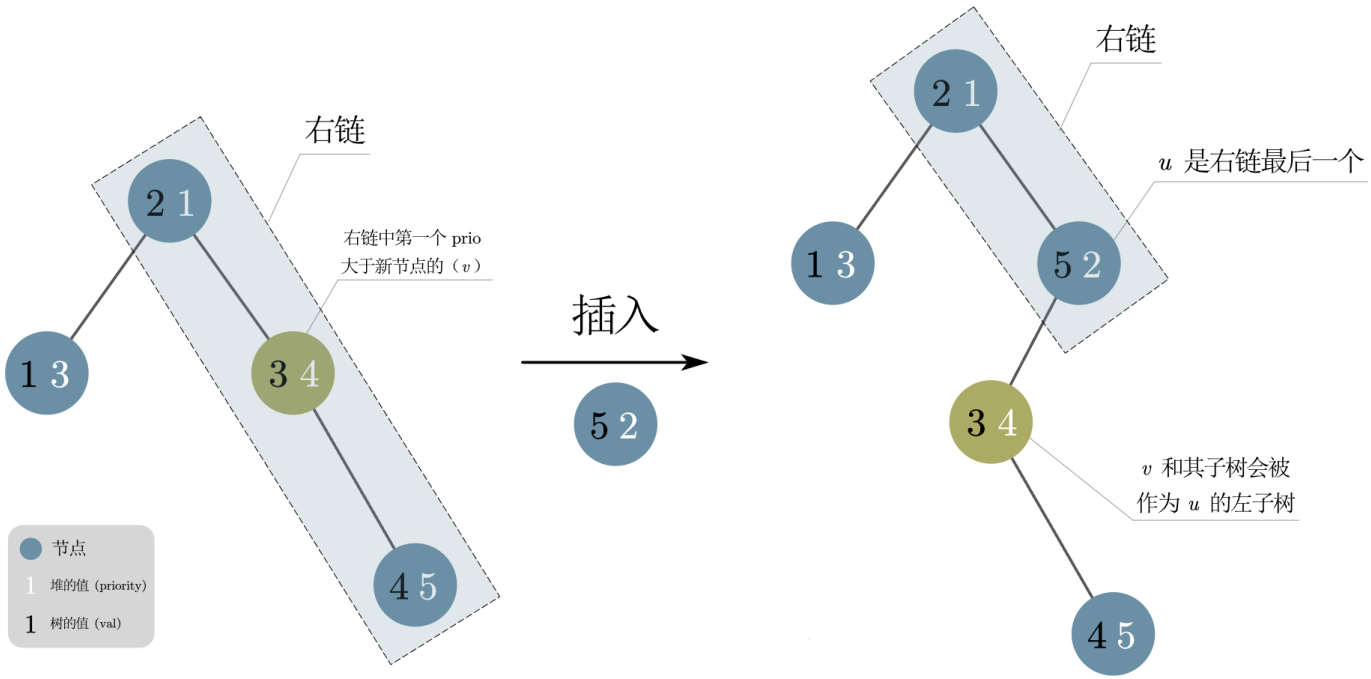
- 目标：维护一个有序数列，并提供区间翻转操作。
- 方法：将区间的下标依次插入 Treap。
- 结果：中序遍历 Treap 时，得到初始区间。

类似于朴素二叉查找树，按递增顺序插入节点可以建立一个长链。中序遍历该树即可得到原始的区间序列。

无旋 Treap - 区间操作

说明中序遍历仍然正确：当在 Treap 中按递增顺序插入节点时，每个新节点将连接到 Treap 的右链上，形成由根节点到最右节点的链。

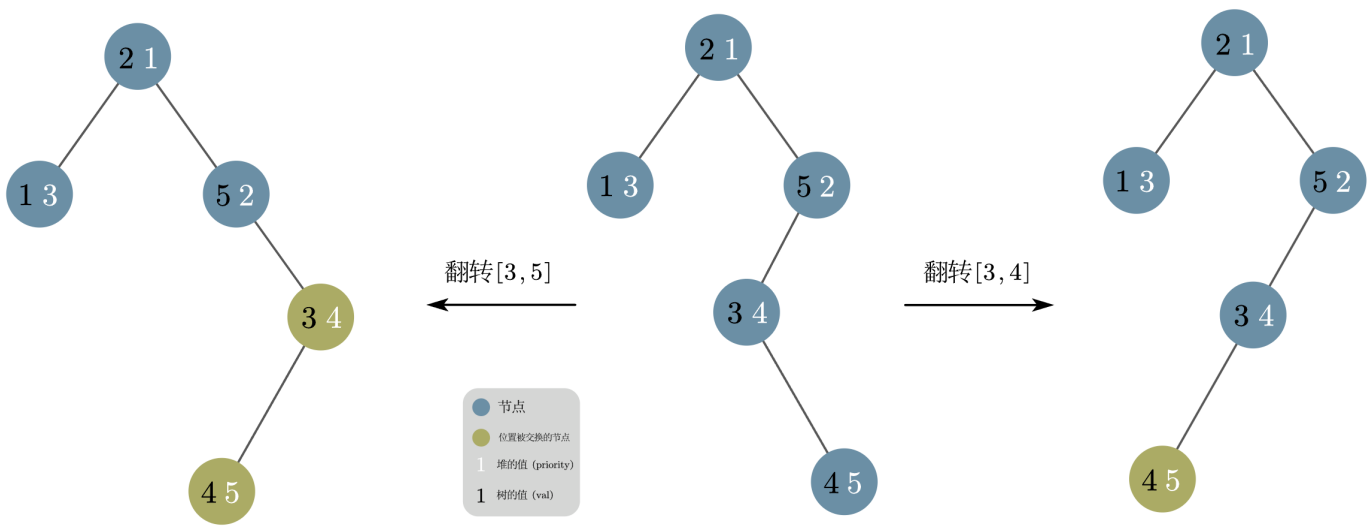
- 新插入 u 会替换右链上第一个优先级 *priority* 大于 u 的节点 v。
- v 及其子树成为 u 的左子树，u 没有右子树。
- 在中序遍历中，u 总是最后被遍历到。



区间翻转操作

翻转区间 $[l, r]$ 的基本思路是将 Treap 分裂成三个区间：$[1, l-1]$，$[l, r]$，$[r+1, n]$，然后对中间区间 $[l, r]$ 进行翻转。

- 分裂 Treap 为三个区间。
- 对中间区间 $[l, r]$ 的每个子树交换左右子节点，使用懒标记放置翻转操作：因为我们想要翻转的区间和懒标记代表的区间不一定重合，所以要在分裂时下传标记。并且，分裂和合并操作会造成每个节点及其懒标记所代表的节点发生变动，所以也需要在合并前下传懒标记。
- 合并三个区间的 Treap 。



中序遍历顺序：$[1, 2, \mathbf{5}, \mathbf{4}, \mathbf{3}]$　　中序遍历顺序：$[1, 2, 3, 4, 5]$　　中序遍历顺序：$[1, 2, \mathbf{4}, \mathbf{3}, 5]$

[TJOI2007] 书架

Knuth 先生家里有个精致的书架，书架上有 N 本书，如今他想学到更多的知识，于是又买来了 M 本不同的新书。现在他要把新买的书依次插入到书架中，他已经把每本书要插入的位置标记好了，并且相应的将它们放好。由于 Knuth 年龄已大，过几天他已经记不清某些位置上放的到底是什么书了，现在想求出来。
第一行为整数 N，接下来 N 行分别是书架上依次放着的 N 本书的书名。接下来的 M 行分别为这本书的书名和要插入的位置。
接下来共有 Q 次询问，每行都是一个整数表示询问的位置。(书架上位置的编号从 0 开始)
对于 100% 的数据，$1 \leqslant N \leqslant 200, 1 \leqslant M \leqslant 10^5, 1 \leqslant Q \leqslant 10^4$ ，所有数据都符合题目中所描述的限制关系。

[TJOI2007] 书架

对于插入操作，加入我们想把一个元素插入后，让它在序列的第 k 个位置，那么我们分裂出大小为 $k-1$ 的子树，把它与新结点合并，再和剩余部分合并。
对于查询操作，既然它按照原序列保持二叉搜索树性质，那么我们就可以把它当做查询第 k 小。先拆出大小为 $k-1$ 的子树，再在剩余部分拆出大小为 1 的子树，取这个结点的 val ，再原样拼回去。
复杂度 $\mathcal{O}(n \log n)$ 。

Splay 树（伸展树）

Splay 树，也称为**伸展树**，是一种自平衡的二叉查找树。由 Daniel Sleator 和 Robert Tarjan 于 1985 年发明。

主要特性

- 通过 **Splay（伸展）操作**将任一节点移动到根节点。
- 保持二叉查找树的性质。
- 插入、查找和删除操作的均摊时间复杂度为 $\mathcal{O}(\log N)$。
- 自平衡，避免树退化为链。

Splay 树的核心优势在于其伸展操作，通过不断旋转某个节点，使树保持平衡，从而实现高效的数据操作。

Splay 操作

Splay 操作的目的是将访问的节点 x 旋转到根节点。

Splay 步骤

- **zig**：当 x 的父节点 p 是根节点时进行。直接将 x 左旋或右旋。
- **zig-zig**：当 p 非根节点且 x 和 p 同侧时进行。首先旋转 p 和其父节点 g，然后旋转 x 和 p。
- **zig-zag**：当 p 非根节点且 x 和 p 异侧时进行。首先旋转 x 和 p，然后旋转 x 和新的父节点。

Splay 树 - 时间复杂度分析

势能分析基础

- 势能函数 $\varphi = \sum_x w(x)$，其中 $w(x) = [\log(\text{size}(x))]$。
- 每次 splay 操作后，势能变化量可通过势能函数变化来计算。

Splay 步骤的势能变化分析

- **zig 操作**：势能变化量 $\Delta\varphi_{\text{zig}} = 1 + w'(x) - w(x)$。
- **zig-zig 操作**：势能变化量 $\Delta\varphi_{\text{zig-zig}} = 1 + w'(x) + w'(fa) + w'(g) - w(x) - w(fa) - w(g)$。简化为 $\leq 3(w'(x) - w(x))$。
- **zig-zag 操作**：势能变化量 $\Delta\varphi_{\text{zig-zag}} = 1 + w'(x) + w'(fa) + w'(g) - w(x) - w(fa) - w(g)$。简化为 $\leq 2(w'(x) - w(x))$。

Splay 树 - 时间复杂度分析

总体时间复杂度

- 初始势能 $\varphi(0) \leq n \log n$。

  由此可见，三种 splay 步骤的势能全部可以缩放为 $\leq 3(w'(x)-w(x))$. 令 $w^{(n)}(x) = w'^{(n-1)}(x), w^{(0)}(x) = w(x)$, 假设 splay 操作一次依次访问了 $x_1, x_2, \cdots, x_n$, 最终 x_1 成为根节点，我们可以得到：

$$3\left(\sum_{i=0}^{n-2}\left(w^{(i+1)}(x_1) - w^{(i)}(x_1)\right) + w(n) - w^{(n-1)}(x_1)\right) + 1$$
$$= 3(w(n) - w(x_1)) + 1 \leq \log n$$

- 对于 n 个节点的 splay 树，每次 splay 操作的均摊复杂度为 $\mathcal{O}(\log n)$。

合并两棵 Splay 树

合并两棵 Splay 树的过程，假设根节点分别为 x 和 y，且 x 树中最大值小于 y 树中最小值。

合并过程

- **空树情况**：如果 x 或 y 为空，返回非空树的根节点，或空树。
- **非空树情况**：
  1. 对 x 树执行 Splay 操作，使得 x 树中的最大值成为根节点。
  2. 将 x 树根节点的右子树设置为 y。
  3. 更新节点信息。
  4. 返回新的根节点。

注意：合并操作确保了 Splay 树的性质，同时保持了树的平衡。

删除操作

删除操作在 Splay 树中需要先将待删除的节点旋转到根的位置。

删除过程

1. 将待删除的节点 x 通过 Splay 操作旋转到根节点。
2. 判断 x 的重复次数 $cnt[x]$：
   - 如果 $cnt[x] > 1$，则减少 x 的重复次数，即 $cnt[x] = cnt[x] - 1$，然后结束操作。
   - 如果 $cnt[x] = 1$，合并 x 的左右子树。
3. 最后，返回合并后的子树根节点，作为新的树根。

注意：删除操作在 Splay 树中是通过旋转和合并子树实现的，每次操作结束之后都要记得 splay 。

[ICPC2022 Xi'an R] Bridge

Erathia 大陆上有 n 个国家，从 1 到 n 编号。每个国家可以看成由 $m+1$ 个结点组成的链，结点从 1 到 $m+1$ 编号。结点 (a,b) 和 $(a,b+1)$ 由一条街道连接，其中 (a,b) 表示国家 a 的第 b 个结点。一开始，国家之间没有桥。

你需要处理 q 个操作：

- $1\ a\ b$ $(1 \le a < n,\ 1 \le b \le m)$：在 (a,b) 和 $(a+1,b)$ 之间建造一座桥。**保证每个结点最多和一座桥相连**。
- $2\ a$ $(1 \le a \le n)$：一名英雄走过 Erathia 大陆。他从 $(a,1)$ 出发。如果这名英雄当前在结点 (x,y) 且有一座未被访问过的桥与之连接，那么他会走过这个桥到达桥的另一端，否则他会走到 $(x,y+1)$。一旦他到达某个国家的第 $m+1$ 个结点，他就会停下来。注意两个询问之间的“未被访问过的桥”是独立的。

你的任务是对每个操作 2 求出英雄最终所在的国家。

$1 \le n, m, q \le 10^5$。

[ICPC2022 Xi'an R] Bridge

不难发现，最开始有 n 条链，并且由于每个点最多有一个桥，所以我们的交换操作实际上等价于将相邻的两条链断开，然后将它们后半部分交换。并且每个点在路径中的相对位置不变。
有一个很直观的思路就是维护点对 (i,j) 表示最开始第 i 条链的第 j 个点在哪条链中，我们需要快速改变它以及它后面点的引索，所以考虑把在一条链中的点对放到一棵以 j 为键值的 Splay 上并维护每个点所属的 Splay 然后操作就变成了分裂之后交换子树以及查询最大值，这个很好维护，可问题是点数是 $\mathcal{O}(nm)$ 级别的。
所以考虑维护三元组表示最开始第 i 条链上第 l 到第 r 个点当前所在的链，不难发现最开始有 n 个三元组且一次操作最多使一个三元组分裂为 (x,l,t) 与 $(x,t+1,r)$ 故三元组数量为 $\mathcal{O}(n+m)$ 级别。
为了快速检索每个点在那个三元组中，把 x 相同的三元组放在一个可以支持快速插入删除寻找前驱后继的平衡树中，在平衡树中储存三元组所在的节点的编号即可。复杂度 $\mathcal{O}(n\log n)$

B 树简介

在计算机科学中，**B 树（B-tree）**是一种自平衡的树，能够保持数据有序。这种数据结构能够让查找数据、顺序访问、插入数据及删除的动作，都在对数时间内完成。
在 B 树中，有两种节点：

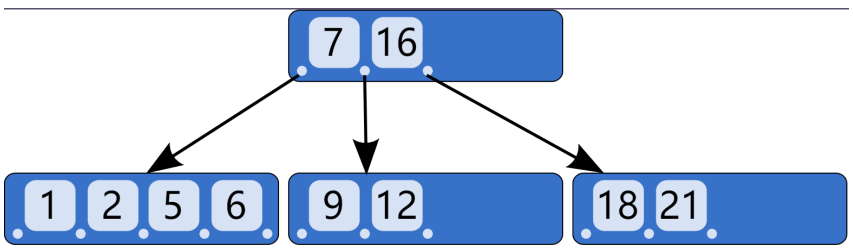
1. 内部节点（internal node）：存储了数据以及指向其子节点的指针。
2. 叶子节点（leaf node）：与内部节点不同的是，叶子节点只存储数据，并没有子节点。

树是一种数据结构。树用多个节点储存元素。某些节点存在一定的关系，用连线表示。二叉树是一种特殊的树，每个节点最多有两个子树。二叉树常用于实现二叉搜索树和二叉堆。

B 树的性质

首先介绍一下一棵 m 阶的 B 树的特性。m 表示这个树的每一个节点最多可以拥有的子节点个数。一棵 m 阶的 B 树满足的性质如下：

1. 每个节点最多有 m 个子节点。
2. 每一个非叶子节点（除根节点）最少有 $\lceil \frac{m}{2} \rceil$ 个子节点。
3. 如果根节点不是叶子节点，那么它至少有两个子节点。
4. 有 k 个子节点的非叶子节点拥有 $k-1$ 个键，且升序排列，满足 $k[i] < k[i+1]$。
5. 每个节点至多包含 $2k-1$ 个键。
6. 所有的叶子节点都在同一层。



图：B-Tree

B 树的插入

在 m 阶 B 树中插入一个值为 v 的新节点，主要步骤如下：

- 新节点始终插入到叶子节点。从根节点向下遍历，找到合适的叶子节点插入新值。
- 检查插入后的叶子节点是否超出最大可容纳节点个数。
- 插入分为三种情况：
  1. 若叶子节点未满（关键字数小于 $m-1$），直接插入。
  2. 若叶子节点已满，进行分裂操作：
     - 选择中位数。
     - 分裂为两个节点：左节点包含小于中位数的元素，右节点包含大于中位数的元素。
     - 中位数上移至父节点，可能导致父节点的进一步分裂。
  3. 如需分裂到根节点，则创建新的根节点，增加树的高度。

B 树的删除

B 树的删除操作要确保不违背 B 树的基本特性，特别是关于节点键数的限制。简化的删除步骤如下：

- 首先定位并删除要移除的元素。如果该元素在非叶节点，将其替换为左子树最大元素或右子树最小元素。
- 然后，根据节点键数进行调整：
  1. 如果节点键数小于 $\lceil m/2 \rceil - 1$，需要调整以保证每个节点的键数不少于 $\lceil m/2 \rceil - 1$。
  2. 如果相邻兄弟节点键数足够（大于 $\lceil m/2 \rceil - 1$），从兄弟节点借一个键。
  3. 如果相邻兄弟节点都不足够，则与一个兄弟节点合并。

B 树的优势

相较于二叉查找树（BST），B 树在处理磁盘存储的数据时显示出明显的优势。这主要由于磁盘读写的特点，与内存读写相比有所不同：

- 磁盘读写速度相对较慢。
- 磁盘读写的单位大小通常大于内存读写的最小单位。

基于这些特点，理想的数据结构应当尽量满足局部性原理，即逻辑上相邻的数据在物理上也应该靠近。因此，对比单一数据节点的 BST，B 树的设计目标是形状更「胖」、更「扁平」，具体表现为：

- 每个节点能够容纳更多数据。
- 降低树的高度，使得逻辑上相邻的数据在物理上也靠近，从而减少磁盘读写操作。

替罪羊树

替罪羊树是一种依靠重构操作维持平衡的重量平衡树。替罪羊树会在插入、删除操作时，检测途经的节点，若发现失衡，则将以该节点为根的子树重构。

替罪羊树——重构

首先，如前所述，我们需要判定一个节点是否应重构。为此，我们引入了一些关键条件：

- 引入一个比例常数 α（取值在 $0.5, 1$ 之间，一般采用 0.7 或 0.8）。
- 若某节点的子节点大小占它本身大小的比例超过 α，则重构。
- 由于采用惰性删除，已删除节点过多也会影响效率。
- 因此，若未被删除的子树大小占总大小的比例低于 α，则亦进行重构。

重构分为两个步骤：先中序遍历展开存入数组，再二分重建成树。

Leafy Tree

Leafy Tree 是一种依靠旋转维持重量平衡的平衡树。

- **通过判断树的数据存储位置，我们可以将树分为 Nodey 和 Leafy 的。**
- **Leafy Tree 被定义为一种信息全部储存在叶子节点上的二叉树。在这种结构中，每个叶子存储值，而每个非叶节点则负责维护树的形态而不维护树的信息。**
- **非叶节点通常会维护孩子信息，从而加速查询。**
- **线段树的结构属于一种 Leafy Tree，因此 Leafy Tree 也被称为平衡线段树。**

Leafy Tree 的特点

1. 所有的信息维护在叶子节点上。
2. 类似 Kruskal 重构树的结构，每个非叶子节点一定有两个孩子，且非叶子节点统计两个孩子的信息（类似线段树上传信息），所以维护 n 个信息的 Leafy Tree 有 $2n-1$ 个节点。
3. 可以完成区间操作，比如翻转，以及可持久化等。

注意到，一个 Leafy 结构的每个节点必定有两个孩子。对其进行插入删除时，在插入删除叶子时必定会额外修改一个非叶节点。常见的平衡树均属于每个节点同时维护值和结构的 Nodey Tree。如果将一个 Nodey 结构的所有孩子的空指针指向一个维护值的节点，那么这棵树将变为一个 Leafy 结构。

Leafy Tree 是一个纯函数化的数据结构，因此其在实现函数化数据结构和可持久化效率上具有先天优势，时间效率极高。

Leafy Tree 的例子

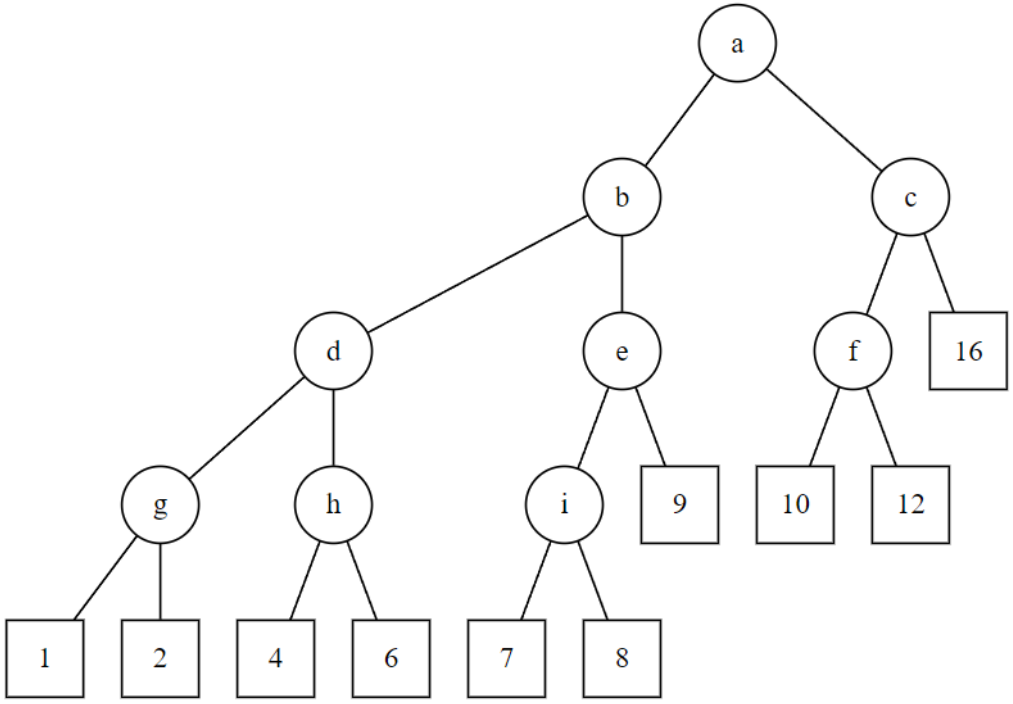


图: Leafy Tree

Leafy Tree 的旋转操作类似于替罪羊树，仅需一次旋转。其余操作和普通 BST 几乎一致。

全局平衡二叉树

由于树链剖分的时间复杂度为 $\mathcal{O}(n\log^2 n)$，而我们熟知的 LCT 虽然时间复杂度为 $\mathcal{O}(n\log n)$，但常数较大，可能比树链剖分还慢。那么有什么既是 $\mathcal{O}(n\log n)$ 的，常数又相对较小的方法呢？这个时候全局平衡二叉树就出现了。

全局平衡二叉树实际上是一颗二叉树森林，其中的每颗二叉树维护一条重链。但是这个森林里的二叉树又互有联系，其中每个二叉树的根连向这个重链链头的父亲，就像 LCT 中一样。但全局平衡二叉树是静态树，区别于 LCT，建成后树的形态不变。

全局平衡二叉树是一种可以处理树上链修改/查询的数据结构，可以做到：

- $\mathcal{O}(\log n)$ 一条链整体修改。
- $\mathcal{O}(\log n)$ 一条链整体查询。
- $\mathcal{O}(\log n)$ 求最近公共祖先，子树修改，子树查询等，这些复杂度和重链剖分是一样的。

全局平衡二叉树—主要性质

全局平衡二叉树的主要性质包括：

1. 全局平衡二叉树由很多棵二叉树通过轻边连起来组成，每一棵二叉树维护了原树的一条重链，其中序遍历的顺序就是这条重链深度单调递增的顺序。每个节点都仅出现在一棵二叉树中。
2. 边分为重边和轻边，重边是包含在二叉树中的边，维护的时候就像正常维护二叉树一样，记录左右儿子和父节点。轻边从一颗二叉树的根节点指向它所对应的重链顶端节点的父节点。轻边维护的时候”认父不认子”，即只能从子节点访问到父节点，不能反过来。注意，全局平衡二叉树中的边和原树中的边没有对应关系。
3. 算上重边和轻边，全局平衡二叉树的高度是 $\mathcal{O}(\log n)$ 级别的。这条是保证全局平衡二叉树时间复杂度的性质。

全局平衡二叉树

下面是一个全局平衡二叉树建树的例子。第一张图是原树，以节点 1 为根节点。实线是重边。
第二张图是建出来的全局平衡二叉树，其中虚线是轻边，实线是重边，每一棵二叉树用红圈表示。

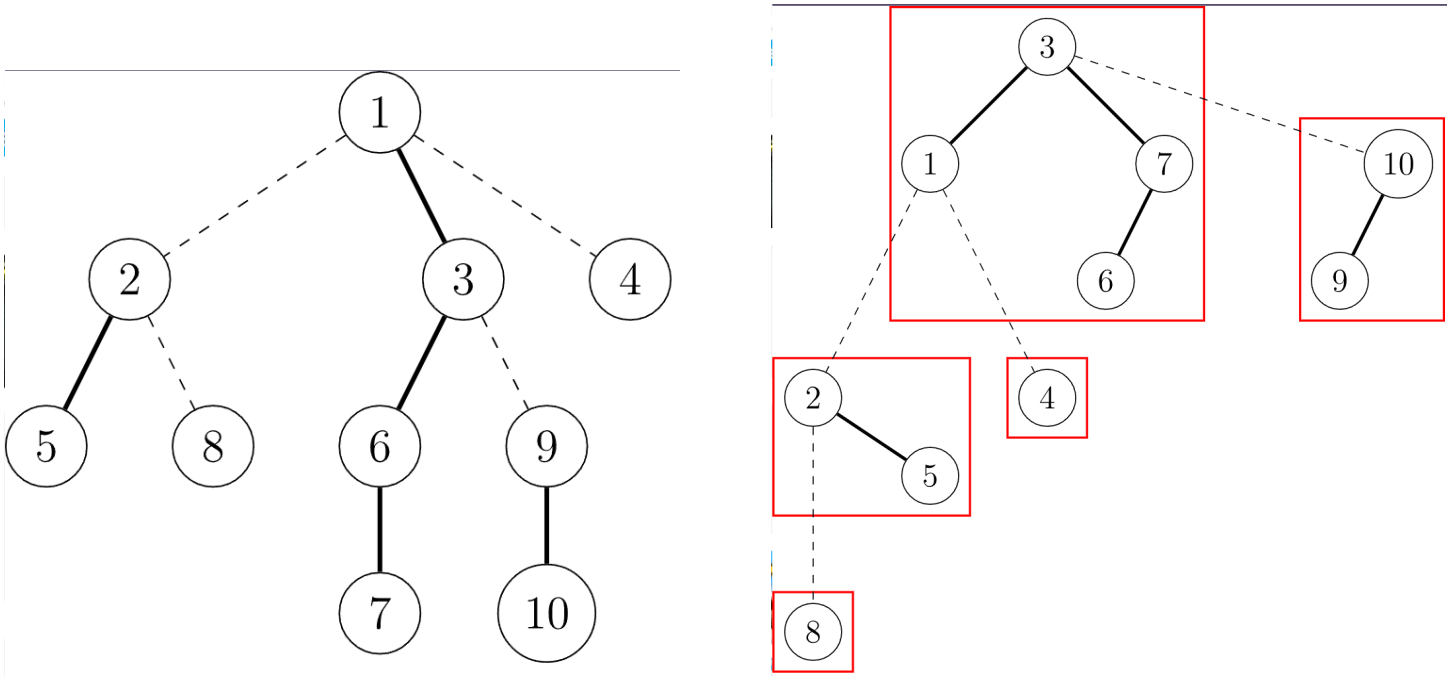


图: 全局平衡二叉树的例子

全局平衡二叉树———建树

- 首先是像普通重链剖分一样，一次 DFS 求出每个节点的重儿子。
- 然后从根开始，找到根节点所在的重链，对于这些点的轻儿子递归建树，并连上轻边。然后我们需要给重链上的点建一棵二叉树。
- 我们先把重链上的点存到数组里，求出每个点轻儿子的子树大小之和加一（即该点本身所贡献的 size）。
- 然后我们按照这个求出这条重链的加权中点，把它作为二叉树的根，两边递归建树，并连上重边。

可以看出建树的时间复杂度是 $\mathcal{O}(n \log n)$。接下来我们可以证明树高是 $\mathcal{O}(\log n)$ 的：考虑从任意一个点跳父节点到根。跳轻边就相当于在原树中跳到另一条重链，由重链剖分的性质可得跳轻边最多 $\mathcal{O}(\log n)$ 条；因为建二叉树的时候根节点找的是算轻儿子的加权中点，那么跳一次重边算上轻儿子的 size 至少翻倍，所以跳重边最多也是 $\mathcal{O}(\log n)$ 条。整体树高就是 $\mathcal{O}(\log n)$ 的。

全局平衡二叉树———查询

1. 关于链修改和链查询的操作方法相对简单，只需要从要操作的点出发，一直跳跃到根节点。
2. 要操作某个点所在的重链上比它深度小的所有点，本质上等同于在这条重链的二叉树中操作目标节点左侧的所有节点。
3. 这些操作可以分解成一系列子树操作，与普通二叉树的维护方法类似，其中涉及到维护子树和以及打子树标记。
4. 在这一过程中，使用的是标记永久化。也可以用 pushdown 来打标记，用 pushup 维护子树和。
5. 不过这种方式可能相对复杂，因为通常情况下，处理二叉树是自上而下进行操作，但在这里，需要首先确定跳跃路径，然后再从上到下进行 pushdown，可能导致常数较大。

Count on a Tree II Striking Back

给定一棵 n 个节点的树。每个节点都有颜色。
每次询问 a, b 路径上的颜色种类数是否大于 c, d 路径上的颜色种类数。
带单点颜色修改且强制在线。
数据保证每次询问 $2f(a, b) \leq f(c, d)$ 或者 $2f(c, d) \leq f(a, b)$ 。
$n \leq 5 \times 10^5$

Count on a Tree II Striking Back

一个非常有意思的随机化算法。
首先如果链上颜色越多那么可能的答案就越小，$[0,1]$ 区间选 k 个数的最小值的期望是 $\frac{1}{k+1}$，我们就把每个颜色随机一个对应值，两条链取 $\min$ 后的值比下大小就行了（多取几次样本减少错误概率）。
利用全局平衡二叉树能在 $\mathcal{O}(\log n)$ 查询某条链上的最小值。
复杂度 $\mathcal{O}(mn\log n)$，其中 m 为随机的次数。