



Weight balanced-tree

Cat-shao

二叉搜索树

- 二叉搜索树本质上是把“二分查找”的结构存下来。考虑我们是怎么二分查找的。
- 序列 a 已经排好序，在 $[l, r]$ 中找 v ， $mid = \frac{(l+r)}{2}$ ，比较 v, mid ，如果 $v > a_{mid}$ 那么 v 在 $[mid + 1, r]$ ，如果 $v < a_{mid}$ 那么 v 在 $[l, mid - 1]$ ，如果 $v = a_{mid}$ 那么就找到了。
- 二叉搜索树存储了“排好序”的序列。求二叉搜索树的中序遍历，能得到这个排好序的序列。
- 我们不能随便交换两个结点，因为交换后，中序遍历可能不是“排好序”的了。唯一的一种操作，就是左旋/右旋。因为左旋/右旋后，树的中序遍历不变。

重量平衡树

- 二叉搜索树可能会退化成链，树高不再是 $\Theta(\log n)$ ，我们需要想个办法让它平衡，使得树高是 $\Theta(\log n)$ 。
- 一个思路：对于每一个子树，在它的左子树与右子树中，结点数多的子树的结点数与结点数少的子树的结点个数的比值不超过某个常数，那么树高就是 $\Theta(\log n)$ 的。
- $\text{size}(T)$ 表示树 T 的结点数，上面的思路可以写成对于任意子树 T ，都有

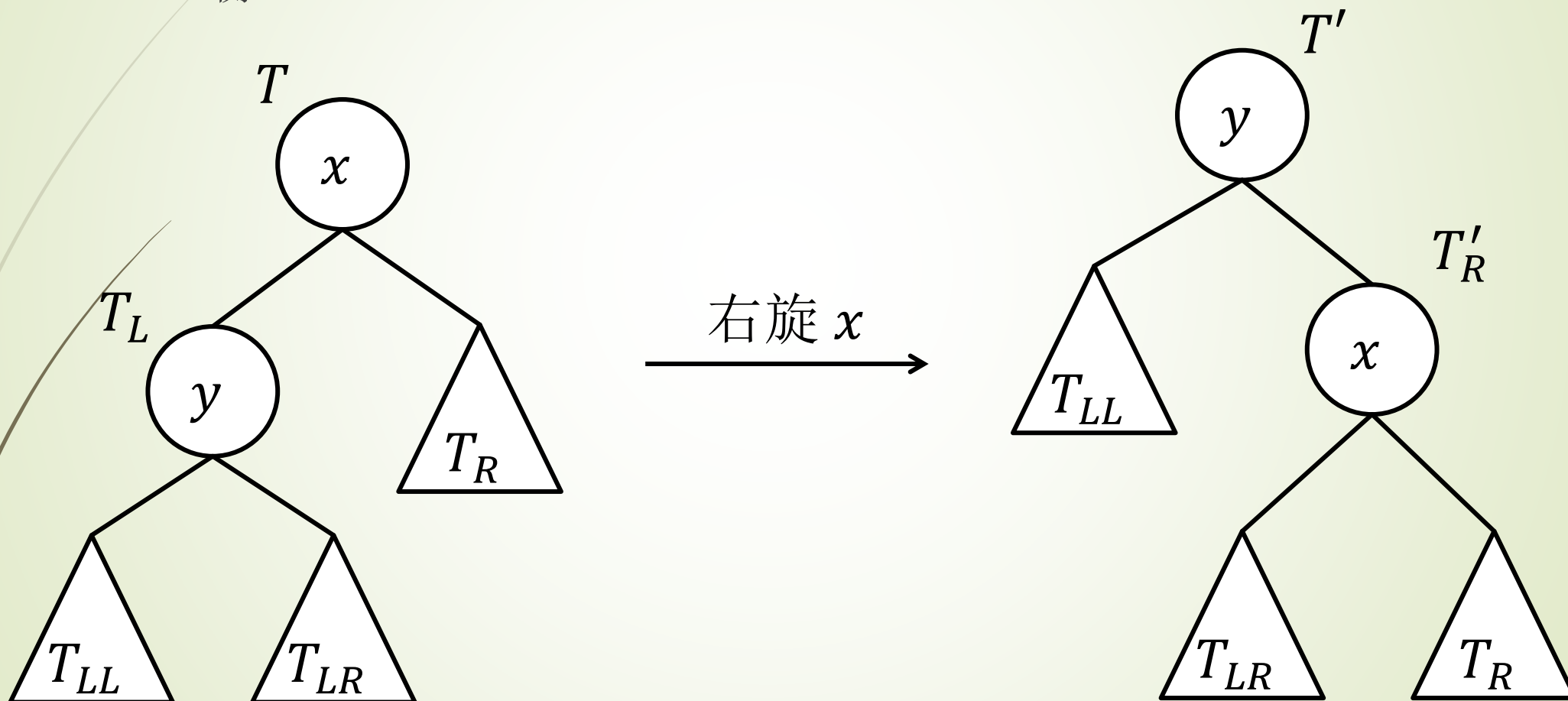
$$\frac{\max\{\text{size}(T_L), \text{size}(T_R)\}}{\min\{\text{size}(T_L), \text{size}(T_R)\}} < \text{const}$$

- 其中 T_L, T_R 分别是 T 的左右子树，那么这棵树平衡。
- 这类平衡树叫做 **weight balanced-tree**，简称 **WBT**，可译为“重量平衡树”。刚刚老师讲过的“替罪羊树”就属于 WBT。

原始 WBT

- 为方便叙述，这两页 PPT 中平衡指的是“子树的比值不超过一个常数”。
- 各位可以想一下怎么用左旋/右旋维护 WBT 的平衡。
- 如果插入/删除后，整棵树依旧平衡，那就太好了，我们什么都不需要做。
- 点的一侧子树太大，不平衡，我们把沉的子树转上去，这样原来沉的子树的一部分就给到了另一侧轻的子树，可能就平衡了。

- 子树 T_L 沉，把 y 旋上去后可能就平衡了。
- 但是仔细一想，发现右旋 y 后， T' 可能不平衡，以 T'_R 为根的树也可能不平衡。

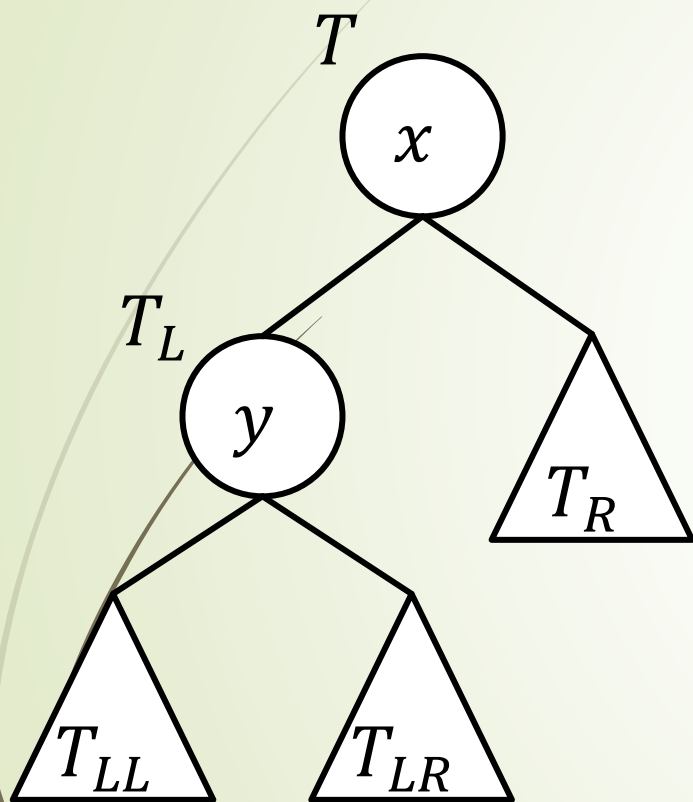


原始 WBT

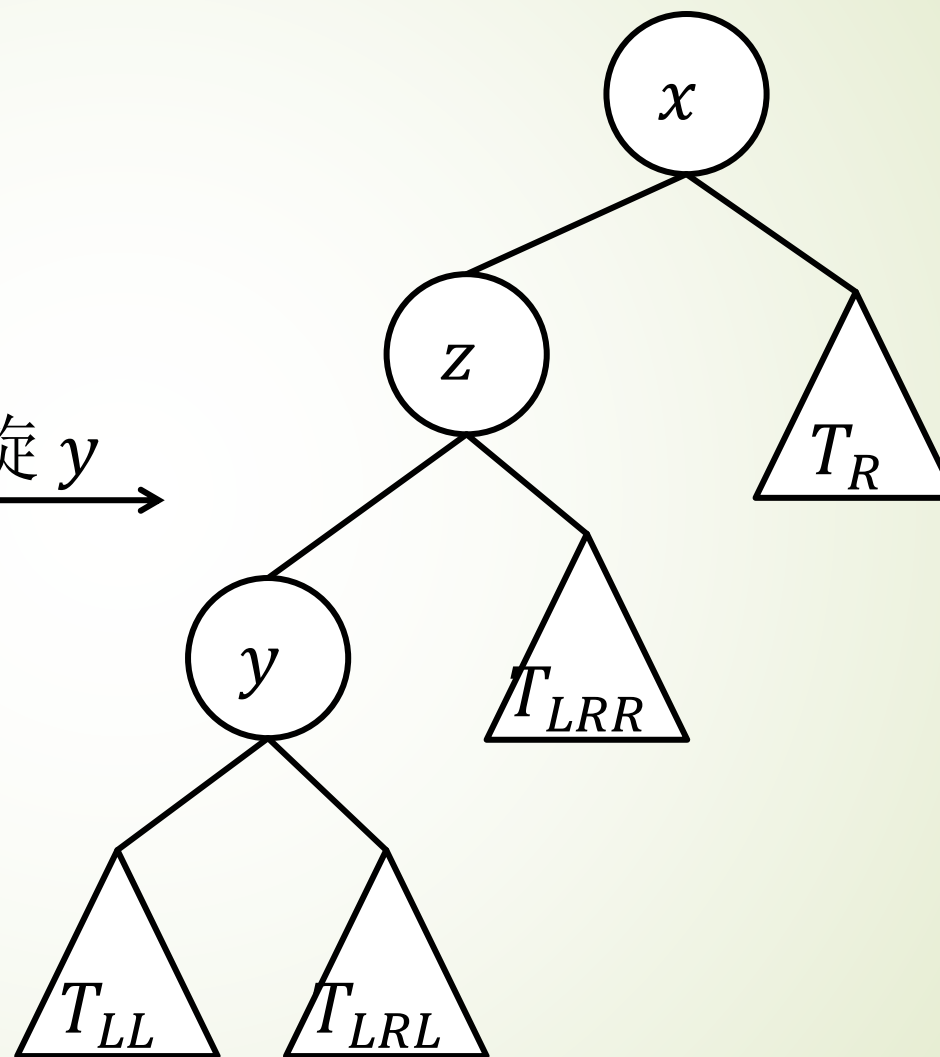
- 基于以上想法，Nievergelt & Reingold 于 1972 年发表了《Binary Search Trees of Bounded Balance》。它们用的名称是 $BB[\alpha]$ ，我们在这里称其为“原始 WBT”。
- 原始 WBT 通过单旋和双旋来维护树的平衡。
- 一棵原始 WBT 有两个参数 $\langle \Delta, \Gamma \rangle$ ， Δ 用于判定平衡， Γ 用于选择双旋。
- 定义“重量”： $\text{weight}(T) = \text{size}(T) + 1$ 。我们基于“重量”判断平衡，而非“结点个数”。空结点的重量是 1。
- 定义“平衡”： T_L, T_R 是 T 的左子树和右子树，如果 $\Delta \text{weight}(T_L) \geq \text{weight}(T_R)$ 且 $\Delta \text{weight}(T_R) \geq \text{weight}(T_L)$ ，且 T_L, T_R 平衡，那么 T 平衡。空结点平衡。
- 每次插入/删除，自底向上维护平衡，每到一个结点 x 看看以 x 为根的子树 T 是否平衡，如果不平衡就旋转。

原始 WBT

- 单旋：把沉的子树旋上去。
- 为什么需要双旋？
- 看第六页 PPT，如果 Δ 在一个合适的区间中取值，可以证明 T'_R 是平衡的。此时，直接单旋的问题就只有 T' 可能不平衡，不平衡只可能是 $\text{weight}(T'_R) > \Delta \text{weight}(T_{LL})$ 。就是因为 T_{LR} 太沉了。
- 为处理这种情况，我们引入 Γ 来判断是否需要双旋。还是举之前 $\text{weight}(T_L) > \Delta \text{weight}(T_R)$ 的例子（毕竟是对称的）。如果 $\text{weight}(T_{LR}) \geq \Gamma \text{weight}(T_{LL})$ ，那么左旋 y ，然后单旋。
- 感性理解下，左旋 y 相当于是把 T_{LR} 劈开，使得单旋转过过去的部分的重量变小。
- 可以证明， Δ, Γ 在一个合理的范围内取值，上述过程能够使整棵树再次平衡。



左旋 y



右旋 x

原始 WBT

- $\langle \Delta, \Gamma \rangle$ 的取值范围？
- 注意到“一次就插入一个点，一次就删一个点”和“合并两棵树”的取值范围不同。对于前者，每个结点每次的重量顶多 $+1$ ，而对于后者，有的结点的重量可能直接多了几倍。
- 仅仅考虑前者， $\langle \Delta, \Gamma \rangle$ 的取值范围也相当复杂，一直以来大家只能用 $\langle \Delta, \Gamma \rangle = \langle 1 + \sqrt{2}, \sqrt{2} \rangle$ ，是最初那篇论文给的。直到 2011 年 Y Hirai 和 K Yamamoto 证明了这个范围。用的是 Coq，证明有 15775 行。它们发的论文中，给出了唯一的整数解： $\langle \Delta, \Gamma \rangle = \langle 3, 2 \rangle$ 。这对整数解同样适用于“合并两棵树”，也就是集合操作。这个我们接下来讲。

集合操作

- 顾名思义，用平衡树来维护有限集合，并支持交集、并集、差集。
- 实现它们，需要用到两个函数 $\text{join}(T_L, k, T_R)$ 和 $\text{split}(T, k)$ 。前者的意思是合并 T_L, k, T_R ， T_L, T_R 是树， k 是一个元素， $\max\{T_L\} < k < \min\{T_R\}$ 。后者的意思是分裂 T 为三个部分： T_L, b, T_R ， T_L 中的元素都小于 k ， T_R 中的元素都大于 k ， b 表示 k 在不在 T 中。
- 怎么用 WBT 实现这两个函数？
- Join: 任意一个为空直接返回另外一个。其余情况：小的往大的里合并，顺着左链/右链找，直到找到一个位置使得在这里合并后当前子树平衡，再向上维护平衡，类似插入。

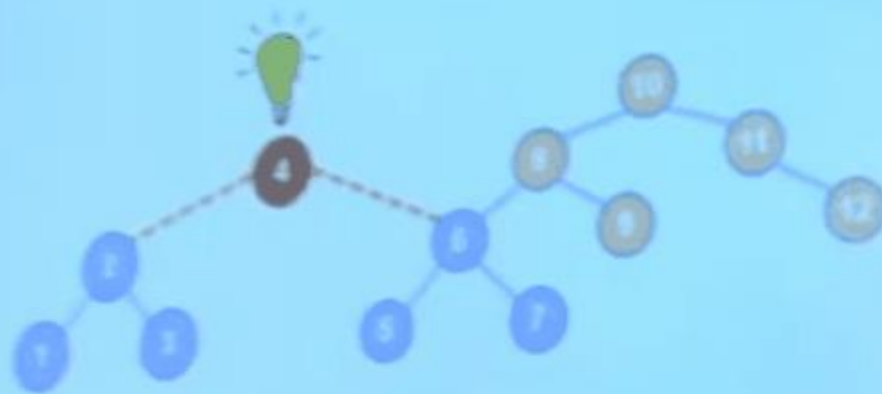
THE MAGIC JOIN FUNCTION

- Connect at a balancing point



THE MAGIC JOIN FUNCTION

- Connect at a balancing point



The same rank (or
differs by just a
small constant)

集合操作

- $\text{split}(T, k)$ ，记 T 的两个子树为 T_L, T_R ， T 的根的元素是 m 。如果 $k = m$ ，直接返回 $\{T_L, \text{true}, T_R\}$ 。如果 $k < m$ ，即 k 在 T_L 中，记 $\{T_1, b, T_2\} = \text{split}(T_L, k)$ ，返回 $\{T_1, b, \text{join}(T_2, k, T_R)\}$ 。
- join 的时间复杂度是 $O\left(\log\left(\frac{\text{weight}(T_L)}{\text{weight}(T_R)}\right)\right)$ ，其中 $\text{size}(T_L) < \text{size}(T_R)$ 。感性理解下，小的往大的合并，不会走完大的树的一个树高。
- 由于 join 的时间复杂度是上面那个，不难证 split 的复杂度是 $O(\log(\text{size}(T)))$ 。


集合操作

- $\text{union}(T_1, T_2)$ ，合并 T_1, T_2 ， T_1, T_2 值域有交集。
- T_1, T_2 有一个是空的，直接返回另外一个。
- L_1, k, R_1 分别是 T_1 的左子树、根的元素、右子树。 $\{L_2, b, R_2\} = \text{split}(T_2, k)$ ， $T_L = \text{union}(L_1, L_2)$ ， $T_R = \text{union}(R_1, R_2)$ ，将 T_1 的根的左右子结点替换成 T_L, T_R 并返回。
- 可以类比线段树合并。
- 时间复杂度 $O\left(n \log\left(\frac{m}{n} + 1\right)\right)$ ，其中 $n = \min\{\text{size}(T_1), \text{size}(T_2)\}$ ， $m = \max\{\text{size}(T_1), \text{size}(T_2)\}$ 。与线段树合并相比，时间多个 \log ，空间少个 \log 。适用于卡线段树合并空间的题目。

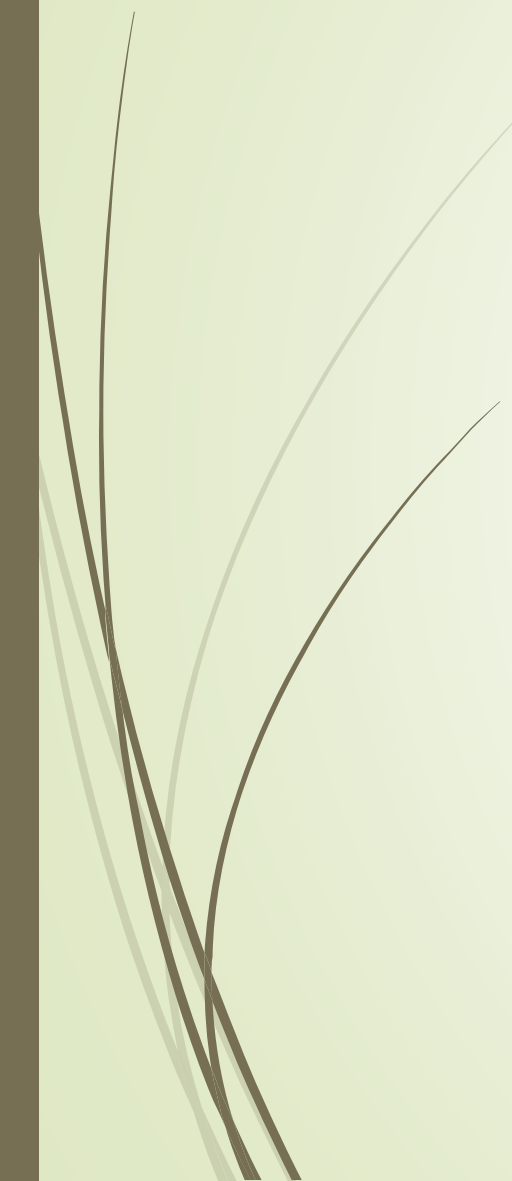
有什么用？

- WBT 的思路不仅仅局限于平衡树，还能用在其他数据结构中。（我个人认为 WBT 最基础的思路，也就是只有单旋，是易于想和理解的）
- WBT 的是我写过的平衡树中跑得次快的，第一是红黑树，那个思维难度更大，代码更长。P6136，我写的 WBT 能跑进 4s。用这个东西写几道毒瘤题，不需要卡常数（甚至还能跑到最优解）。
- 我实现前面讲的用分裂合并的原始 WBT，跑得跟无旋 Treap 一样慢，可能是因为我的实现用的是 `std::tuple`。
- 不难发现，前面的“集合操作”用无旋 Treap 同样能做，是个通用的方法，算不上 WBT 的优势。

排名	中文名称	英文名称	速度 (单位: 秒)
1	红黑树	red black tree	5.12
2	重量平衡树	weight balanced tree	5.31
3	AVL	AVL	5.79
4	替罪羊树	scapegoat tree	6.01
5	树堆	treap	6.29
6	范浩强的树堆	fhq treap	7.16 (7.81)
7	重量平衡树 (使用分裂和合并)	Parallel Ordered Sets Using Join	7.13 (9.03)
8	重量平衡树 (leafy tree实现)	weight balanced leafy tree	7.61
9	替罪羊树 (leafy tree实现)	scapegoat leafy tree	8.98



参考

- Balancing weight-balanced trees
 - Parallel Ordered Sets Using Join
- 



谢谢大家