替罪羊树 (Scapegoat Tree) 介绍

替罪羊树是一种二叉搜索树 (Binary Search Tree) 。通过对“非平衡”的子树进行重构变为平衡的完全二叉树的方式，替罪羊树可以在均摊 $O(\log n)$ 时间内完成对任意结点的查找、插入、删除操作，即从一棵空树开始连续的 n 次插入或删除操作的总时间复杂度为 $O(n\log n)$ 。

重量平衡

定义一个二叉树是 α**-重量平衡**的，当且仅当树上的所有结点 $node$ 都满足

$$\begin{gathered}\operatorname{size}(left) \le \alpha \cdot \operatorname{size}(node), \\ \operatorname{size}(right) \le \alpha \cdot \operatorname{size}(node).\end{gathered}$$

这里 $\operatorname{size}(node)$ 指的是以 $node$ 为根的子树所包含的结点个数（包括 $node$ 自己），如果 $node$ 为空则 $\operatorname{size}(node) = 0$ 。 $left$ 和 $right$ 分别表示 $node$ 结点的左孩子和右孩子。

当 $\alpha \ge 1$ 时，以上等式总是成立，因此我们接下来讨论 $0.5 \le alpha < 1$ 的情形。

如果一棵二叉树是 α-重量平衡的，则 $\operatorname{size}(node) \ge \dfrac{1}{\alpha}\operatorname{size}(left)$，那么每当树高增加 1，整个树的结点个数就要至少增加 $\dfrac{1}{\alpha}$ 倍，因此有以下结论：

$$\operatorname{height}(tree) \le \left\lfloor \log_{1/\alpha}(\operatorname{size}(tree)) \right\rfloor + 1.$$

因此一棵具有 n 个结点的 α-重量平衡的二叉树的树高是 $O(\log n)$ 的。

替罪羊树就是 α-重量平衡的二叉树。

α 的取值会影响替罪羊树的运行效率，这里建议可以取 $\alpha = 0.75$ 。

维持平衡

仅含 1 个或 0 个结点的二叉树一定是 α-重量平衡的，而对于树上结点的插入与删除可能会打破这个性质。为了使树在结点变化前后始终保持 α-重量平衡，我们采用重构子树的方式维持平衡。

插入操作

插入一个结点会使这个点所有的祖先结点的 size 增加 1 ，也就是说总共可能有 $O(\log n)$ 个结点的 α-重量平衡性被破坏。我们依次去检查这之中的哪些结点违反了 α-重量平衡的性质，取其中高度最低的那个结点作为“替罪羊”。然后我们将以“替罪羊”为根子树直接重构，使得它满足 α-平衡性质并且尽可能完美。

重构的策略是让左右子树的 size 尽可能平均。假设以“替罪羊”为根的子树中的结点序列为 $s_l, s_{l+1}, \cdots, s_r$ 。我们会让前一半的结点分到左子树当中，让后一半的结点分到右子树当中。

```
Node *rebuild(size_t l, size_t r, Node s[])
{
    if (l > r)
        return nullptr;
    size_t mid = (l + r) / 2;
    s[mid].left = rebuild(l, mid - 1, s);
    s[mid].right = rebuild(mid + 1, r, s);
    return &s[mid];
}
```

删除操作

额外记录一个变量 `MaxNodeCnt` 表示替罪羊树结点数量的最大值。令 `NodeCnt` 表示替罪羊树中所含的结点数量，在每次插入或删除过后更新 `MaxNodeCnt = max(MaxNodeCnt, NodeCnt)` 。并且，如果整棵树被重构，应当设置 `MaxNodeCnt = NodeCnt` 。

利用这个信息，经过一次删除过后，如果 `NodeCnt <= alpha * MaxNodeCnt` ，我们就将整棵树重构。

这意味着其实删除某些结点过后，某一些子树会不再具有 α-重量平衡的性质，只有当整棵树中被删除过的结点数量过多时，才会选择重构整棵树。

但从实际应用的角度来说，如果替罪羊树中的结点个数不超过 n ，那么替罪羊树进行删除重构的插入、查找、删除的最坏时间复杂度是 $O(\log n)$ ，而替罪羊树不进行删除重构的插入、查找、删除的最坏时间复杂度依然是 $O(\log n)$ 。其实是因为删除操作不增加树的高度，所以不进行删除重构并不会引起更糟糕的时间复杂度（你完全可以在此题中不实现删除重构）。

复杂度证明

插入重构的均摊复杂度证明

考虑一棵具有 n 个结点并且刚刚被重构过的二叉树，假设我们插入了 k 个结点过后，整棵树才不平衡并且需要重构，那么

$$\left\lfloor \frac{n}{2} \right\rfloor + k \geq \alpha(n+k) \implies n \leq \frac{1-\alpha}{\alpha - \frac{1}{2}} \cdot k. \tag{1}$$

由于我们插入了 k 个结点**才**使它不平衡，插入这 k 个结点的过程中整棵树的高度应该是保持在 $O(\log n)$ 的。于是，插入 k 的结点所需要的总时间为

$$T_{\text{insert}}(n,k) = kO(\log n) + \Theta(n+k),$$

其中 $kO(\log n)$ 是插入操作本身所需的时间， $\Theta(n+k)$ 是进行一次重构的时间。将 (1) 式代入，得

$$T_{\text{insert}}(n,k) = kO(\log n) + \Theta(n+k) \leq kO(\log n) + \Theta\left(\frac{\frac{1}{2}}{\alpha - \frac{1}{2}} k\right) = O(k \log n).$$

所以平均一次插入的时间复杂度为 $O(\log n)$ 。

删除重构的均摊复杂度证明

考虑一棵具有 n 个结点并且刚刚被重构过的二叉树，假设我们删除了 k 个结点过后，整棵树才会被删除重构，那么

$$n - k \le \alpha \cdot n \implies n \le \frac{k}{1-\alpha}.$$

这 k 次删除操作的总时间为

$$kO(\log n) + O(n) \le kO(\log n) + O\left(\frac{k}{1-\alpha}\right) = O(k \log n),$$

所以平均一次删除的时间复杂度为 $O(\log n)$ 。