第 12 章 高级查找

跳表、红黑树、伸展树

郑新、徐鹏飞、李健

2025 年 12 月 15 日

本节内容概览

跳表：概率快速索引 (Skip List)

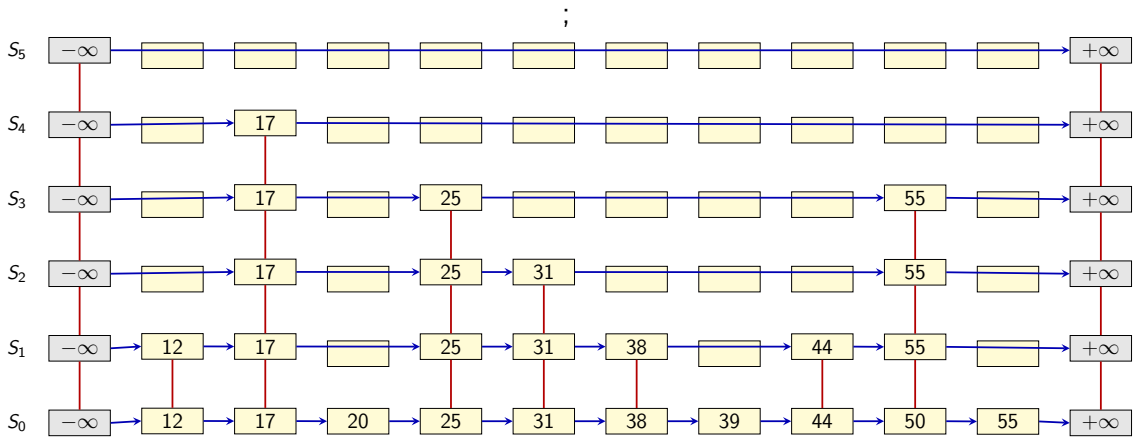
- **概念:** 跳表是一种巧妙地利用概率构建的多层有序链表数据结构。

- **目标:** 实现像平衡树一样快速的查找，但其代码实现相比平衡树**更简单**。
- **核心思想:** 通过随机机制提升某些节点到上层，形成“快速通道”。

核心结构与层级概念

- **最底层 (Level 0):** 包含所有数据元素，是一个完整的、**有序的**链表。
- **上层 (Higher Levels):** 它们是下层的“快速通道”，**只包含一部分元素**。

跳表 (skip list) 结构示例



说明: • 蓝色箭头：水平指针 • 红色竖线：垂直指针 • 灰色节点：头尾哨兵节点

概率规律：固定的比例 p

- **核心原则:** 每个节点独立决定是否提升，且遵循一个固定的概率 p。
- **概率 p:** 通常设定为 $p = 1/2$ 或 $p = 1/4$。

几何分布带来的性能

设 N 为总节点数：

- Level 0 (最底层): 约 N 个节点
- Level 1: 约 $N \cdot p$ 个节点
- Level 2: 约 $N \cdot p^2$ 个节点
- Level k: 约 $N \cdot p^k$ 个节点

$$\text{跳表高度} H \approx \log_{1/p} N = O(\log N)$$

结论: 这种概率分布保证了**平均查找性能**与平衡树相当，为 $O(\log N)$。

核心操作：查找 (Search)

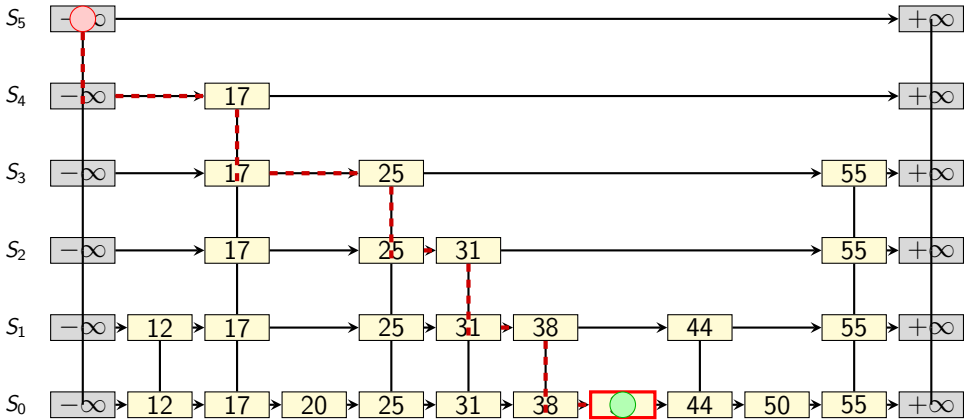
- **起点:** 从**头部节点的最高层级**开始。
- **目标:** 利用高层指针实现大步跳跃，模仿二分搜索。

查找 X 的策略 (从上到下，从左到右)

1. **起始:** 设置当前指针 p 为头部节点的最高层指针。
2. **横向遍历 (跳跃):** 在当前层级，当 p.next $\leq X$，就移动 `p=p.next`。
3. **纵向下降 (降层):** 否则，沿着 p 节点的垂直指针下降到下一层。
4. **重复:** 重复步骤 2 和 3，直到到达 Level 0。
5. **结果:** 在 Level 0，检查 `p.next` 是否等于 X。

效率: 通过每层排除大量节点，平均查找路径长度为 $O(\log N)$。

路径演示：查找值 39



查找收尾：Level 0 的最终判断（强化版）

- **策略回顾:** 在所有层级，我们都遵循向右移动，直到下一个节点 $\geq X$的原则。
- **最终定位:** 指针 p 最终停在 Level 0，它是目标值 X 的**精确前驱节点**。

判断 X 是否存在的逻辑

`p->next->value==x` (C 语言)

重要性: 查找操作不仅仅是为了“查找”，
它还为后续的插入和删除操作精确找到了所有层级的前驱节点列表。

查找收尾：Level 0 的最终判断（强化版）

- **策略回顾:** 在所有层级，我们都遵循向右移动，直到下一个节点 $\geq X$的原则。
- **最终定位:** 指针 p 最终停在 Level 0，它是目标值 X 的**精确前驱节点**。

判断 X 是否存在的逻辑

`p->next->value==x` (C 语言)

重要性: 查找操作不仅仅是为了“查找”，
它还为后续的插入和删除操作精确找到了所有层级的前驱节点列表。

数据结构选择

新节点 X 可能随机生成 k 个层高，我们需要记录 k 个前驱节点的指针。
我们应用哪种结构来临时存储不同层级的前驱节点，以便进行插入操作呢？

核心操作：插入 (Insertion) Level 0

待修改指针： `x.next` 和 `p.next`。

1. **新节点连接 (Step 1):**
   让新节点 x 继承 p 的原后继节点。

$$\texttt{x.next} = \texttt{p.next}$$

2. **前驱节点连接 (Step 2):**
   让 p 指向新节点 x。

$$\texttt{p.next} = \texttt{x}$$

为什么是这个顺序？

- 如果先执行 Step 2，原 `p.next` 的地址将丢失（悬空），导致链表断裂。

核心操作：向上层层链接 (Promotion)

- **前提:** 已确定新节点 x 的随机高度 h, 栈中存储了所有层的前驱节点 pred_i。
- **顺序:** 必须 **自底向上** $(i = 1 \to h)$ 进行链接。

利用栈完成 S_1 到 S_h 的插入

1. **循环:** 从 $i = 1$ 到 h，从栈中取出 Level i 的前驱节点 pred_i。
2. **水平链接 (Level i):**

$$\texttt{new\_node}_i\texttt{.next} = \texttt{pred}_i\texttt{.next};\ \texttt{pred}_i\texttt{.next} = \texttt{new\_node}_i$$

3. **垂直链接 (Level $i-1$ 至 i):** 将 x 的底层副本与 x 的当前层副本链接。

$$\texttt{new\_node}_{i-1}\texttt{.up} = \texttt{new\_node}_i$$

向上链接：垂直方向指针 (错误)

- **目标:** 将新节点 x 的不同层级副本连接起来，形成一个垂直链。
- **纠正:** 通常使用 `.down` 指针（从高层指向低层）。

错误的垂直链接表示

$$\texttt{new\_node}_{i-1}\texttt{.up} = \texttt{new\_node}_i$$

注意：这种 `.up` 指针表示，需要从底层节点出发才能找到上层节点。

问题: 跳表遍历通常从高层向下，因此**从上到下**的指针更符合查找逻辑。

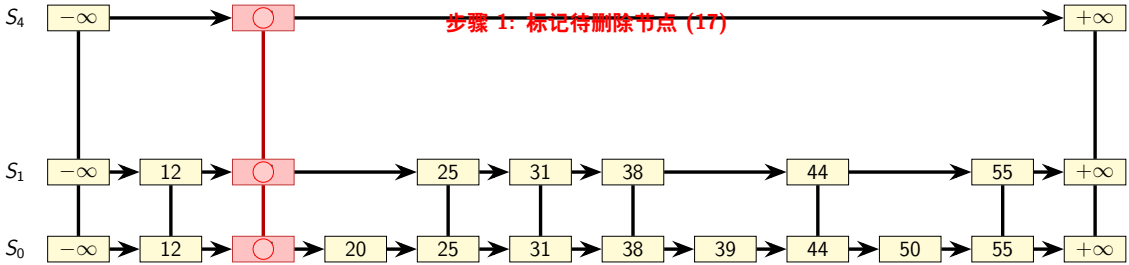
核心操作：删除 (Deletion) 流程

- **目标:** 安全地移除目标值 x 的所有层级副本，并维护所有链表
- **依赖:** 与插入操作一样，依赖于查找过程获得的前驱节点栈

三步删除流程

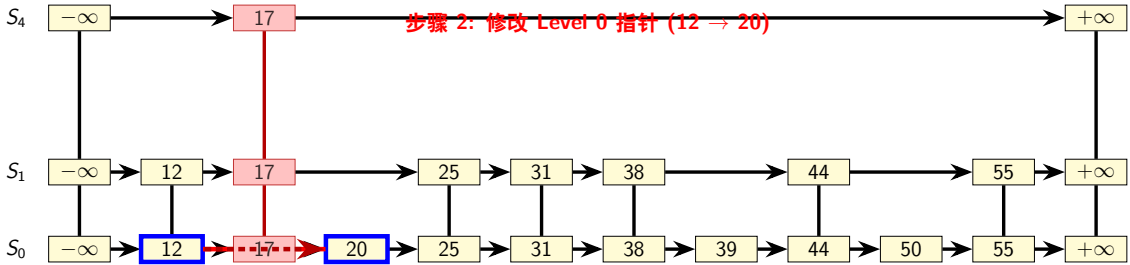
1. **定位与压栈:** 查找 x，并将所有层的前驱节点 pred_i 压入栈中
2. **Level 0 验证和移除:**
   - 验证 $\texttt{pred}_0\texttt{.next.value} = x$
   - 执行 `Level 0 删除`：$\texttt{pred}_0\texttt{.next} = \texttt{pred}_0\texttt{.next.next}$
3. **向上层级清理 ($S_1 \to S_h$):**
   - 从栈中取出 pred_i，并执行 `Level i 删除`：$\texttt{pred}_i\texttt{.next} = \texttt{pred}_i\texttt{.next.next}$
   - 检查并更新头节点的高度。

路径演示：删除节点 17



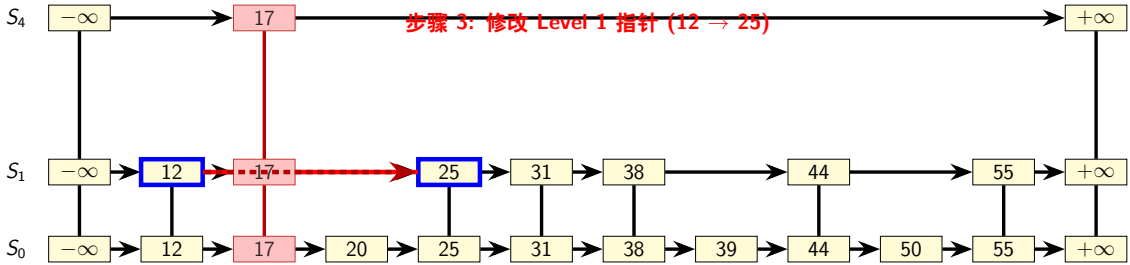
- **目标：** 删除 Level 0 到 S_4 上的所有节点 17。

路径演示：删除节点 17



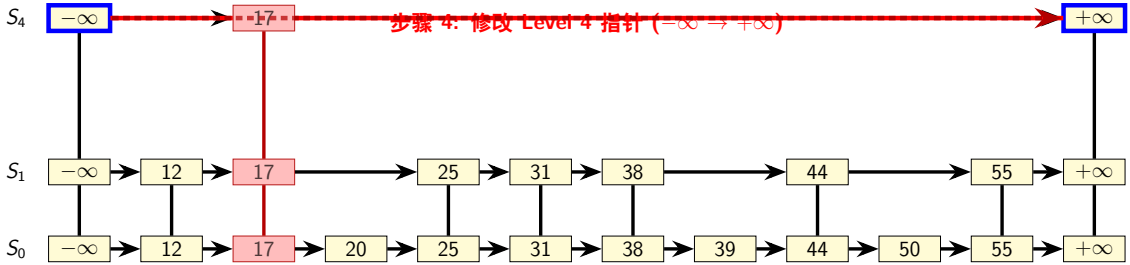
- **目标：** 删除 Level 0 到 S_4 上的所有节点 17。
- **Level 0 清理：** 前驱 12 将被修改，跳过 17。

路径演示：删除节点 17



- **目标:** 删除 Level 0 到 S_4 上的所有节点 17。
- **Level 1 清理:** 前驱 12 将被修改，跳过 17。

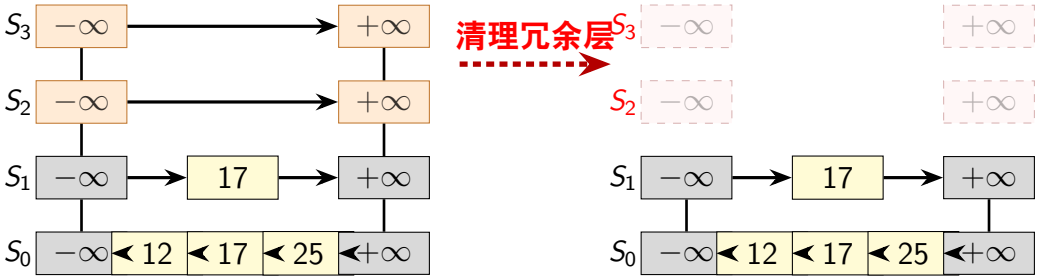
路径演示：删除节点 17



- **目标:** 删除 Level 0 到 S_4 上的所有节点 17。

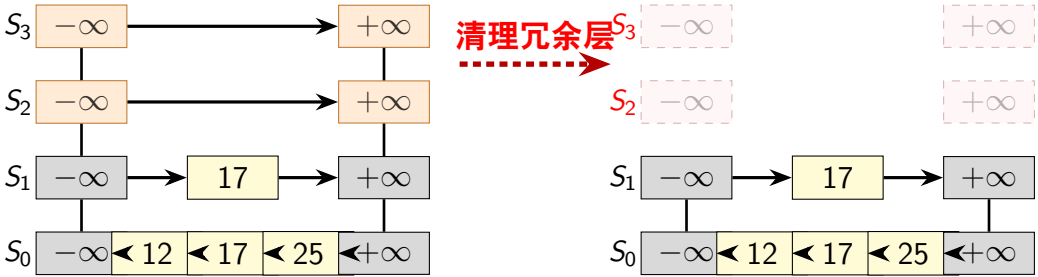
- **Level 4 清理:** 前驱 $-\infty$ 将被修改，跳过 17。

跳表冗余层清理



界定冗余: 一层中只包含哨兵节点，没有实际数据节点
清理策略: 从上向下删除所有冗余层，直到遇到包含数据节点的层

跳表冗余层清理



界定冗余: 一层中只包含哨兵节点，没有实际数据节点
清理策略: 从上向下删除所有冗余层，直到遇到包含数据节点的层

真删，还是假删 (逻辑上)? 譬如，维护变量 `max_height` 表示当前最高（有效）层。

练习 1/3：空间复杂度分析

问题： 跳表相比于普通有序链表，虽然操作效率更高，但空间复杂度略高。跳表的平均空间复杂度是多少？

A. $O(1)$

B. $O(n)$

C. $O(n \log n)$

D. $O(n\sqrt{n})$

练习 1/3：空间复杂度分析

问题： 跳表相比于普通有序链表，虽然操作效率更高，但空间复杂度略高。跳表的平均空间复杂度是多少？

A. $O(1)$

B. $O(n)$

C. $O(n \log n)$

D. $O(n\sqrt{n})$

答案是 B. $O(n)$

练习 1/3：空间复杂度分析

- **数学原理：** 为什么有 $\mathbf{O}(\log \mathbf{n})$ 层，但不是 $\mathbf{O}(\mathbf{n} \log \mathbf{n})$ 的空间？

$$\text{总指针数} = N_0 + N_1 + N_2 + \cdots + N_h$$

- **总指针数量的期望值** $E[S]$**：** 这是一个几何级数求和：

$$E[S] = \sum_{k=0}^{h} N_k \approx \sum_{k=0}^{\infty} n \cdot p^k = n \cdot \sum_{k=0}^{\infty} p^k$$

练习 1/3：空间复杂度分析

- **数学原理：** 为什么有 $\mathbf{O}(\log \mathbf{n})$ 层，但不是 $\mathbf{O}(\mathbf{n} \log \mathbf{n})$ 的空间？

$$\text{总指针数} = N_0 + N_1 + N_2 + \cdots + N_h$$

- **总指针数量的期望值** $E[S]$**：** 这是一个几何级数求和：

$$E[S] = \sum_{k=0}^{h} N_k \approx \sum_{k=0}^{\infty} n \cdot p^k = n \cdot \sum_{k=0}^{\infty} p^k$$

- 根据无穷等比级数求和公式（当 $|p| < 1$ 时）：$\sum_{k=0}^{\infty} p^k = \frac{1}{1-p}$

练习 1/3：空间复杂度分析

- **数学原理：** 为什么有 $\mathbf{O}(\log \mathbf{n})$ 层，但不是 $\mathbf{O}(\mathbf{n} \log \mathbf{n})$ 的空间？

$$\text{总指针数} = N_0 + N_1 + N_2 + \cdots + N_h$$

- **总指针数量的期望值** $E[S]$**：** 这是一个几何级数求和：

$$E[S] = \sum_{k=0}^{h} N_k \approx \sum_{k=0}^{\infty} n \cdot p^k = n \cdot \sum_{k=0}^{\infty} p^k$$

- 根据无穷等比级数求和公式（当 $|p| < 1$ 时）：$\sum_{k=0}^{\infty} p^k = \frac{1}{1-p}$
- 因此，总指针数量 $E[S]$ 的期望值为：$E[S] \approx n \cdot \frac{1}{1-p}$

练习 1/3：空间复杂度分析

- **数学原理：** 为什么有 $\mathbf{O}(\log \mathbf{n})$ 层，但不是 $\mathbf{O}(\mathbf{n} \log \mathbf{n})$ 的空间？

$$\text{总指针数} = N_0 + N_1 + N_2 + \cdots + N_h$$

- **总指针数量的期望值** $E[S]$**：** 这是一个几何级数求和：

$$E[S] = \sum_{k=0}^{h} N_k \approx \sum_{k=0}^{\infty} n \cdot p^k = n \cdot \sum_{k=0}^{\infty} p^k$$

- 根据无穷等比级数求和公式（当 $|p| < 1$ 时）：$\sum_{k=0}^{\infty} p^k = \frac{1}{1-p}$
- 因此，总指针数量 $E[S]$ 的期望值为：$E[S] \approx n \cdot \frac{1}{1-p}$
- 当 $p = 1/2$ 时，$\frac{1}{1-p} = 2$。跳表的平均总指针数约为 $2n$。

练习 1/3：空间复杂度分析

- **数学原理：** 为什么有 $\mathbf{O}(\log \mathbf{n})$ 层，但不是 $\mathbf{O}(\mathbf{n} \log \mathbf{n})$ 的空间？

$$\text{总指针数} = N_0 + N_1 + N_2 + \cdots + N_h$$

- **总指针数量的期望值** $E[S]$**：** 这是一个几何级数求和：

$$E[S] = \sum_{k=0}^{h} N_k \approx \sum_{k=0}^{\infty} n \cdot p^k = n \cdot \sum_{k=0}^{\infty} p^k$$

- 根据无穷等比级数求和公式（当 $|p| < 1$ 时）：$\sum_{k=0}^{\infty} p^k = \frac{1}{1-p}$
- 因此，总指针数量 $E[S]$ 的期望值为：$E[S] \approx n \cdot \frac{1}{1-p}$
- 当 $p = 1/2$ 时，$\frac{1}{1-p} = 2$。跳表的平均总指针数约为 $2n$。
- 由于 $2n$ 属于 $O(n)$，因此跳表的平均空间复杂度是线性的 $\mathbf{O}(\mathbf{n})$。

练习 2/3：垂直链接的方向

问题： 在插入操作中，为什么新节点的垂直链接必须从 Level 0 向上 进行？

A. **栈的特性**：因为栈（Stack）存储的前驱节点指针是从高层到低层压入的，只有从下到上弹出才能正确对应。

B. **指针方向**：因为跳表的核心结构要求高层节点通过 `down` 指针指向低层节点，必须先创建 `down` 指针。

C. **Level 0 基础**：因为 `Level 0` 的插入操作是所有操作的基础，必须先完成。

D. **删除准备**：这是为了保证删除操作时，可以从高层向下高效地移除节点。

练习 2/3：垂直链接的方向

正确答案解析

答案是 A. 栈的特性。

练习 2/3：垂直链接的方向

正确答案解析

答案是 A. 栈的特性。

- **查找与压栈：** 在查找待插入位置时，我们是从最高层 (S_h) 向下遍历到 S_0，并将沿途的所有前驱节点 pred_i 按顺序 ($\text{pred}_h, \ldots, \text{pred}_1, \text{pred}_0$) 压入栈中
- **插入与弹栈：** 栈是 LIFO（后进先出）。如果我们想从 Level 0 向上插入和链接，我们必须按顺序 $\text{pred}_0, \text{pred}_1, \ldots, \text{pred}_h$ **弹出前驱节点**
- **结论：** 栈的 LIFO 特性将从上到下的查找顺序，逆转为从下到上的插入顺序。

关于干扰项的说明

练习 2/3：垂直链接的方向

正确答案解析

答案是 A. 栈的特性。

- **查找与压栈：** 在查找待插入位置时，我们是从最高层 (S_h) 向下遍历到 S_0，并将沿途的所有前驱节点 pred_i 按顺序 ($\text{pred}_h, \ldots, \text{pred}_1, \text{pred}_0$) 压入栈中
- **插入与弹栈：** 栈是 `LIFO`（后进先出）。如果我们想从 Level 0 向上插入和链接，我们必须按顺序 $\text{pred}_0, \text{pred}_1, \ldots, \text{pred}_h$ **弹出前驱节点**
- **结论：** 栈的 `LIFO` 特性将从上到下的查找顺序，逆转为从下到上的插入顺序。

关于干扰项的说明

- **B 错误：** 尽管 `.down` 指针是标准做法，但是栈（而非指针方向）决定操作顺序
- **C 错误：** 尽管底层是基础，但向上链接是为了利用栈中的前驱信息。

练习 3/3: 性能权衡与实践选择

问题: 尽管跳表的最坏情况复杂度是 $O(n)$，为什么在实际应用中（如 Redis、LevelDB）仍倾向于使用它，而不是使用保证 $O(\log n)$ 的平衡树？

A. **缓存局部性**：跳表的节点通常按顺序存储在内存中，在查找时能更好地利用 CPU 缓存，因此常数因子比平衡树低。

B. **实现简易性与并发性**：跳表的实现代码更简单、维护更容易；并且在并发环境下，其局部修改特性使其更容易实现非阻塞同步。

C. **严格 $O(\log n)$ 保证**：只有跳表的查找操作能严格保证 $O(\log n)$ 复杂度，而平衡树的最坏情况复杂度也是 $O(n)$。

D. **空间效率**：跳表比平衡树（如红黑树）更节省内存空间，因为它只需要存储 `next` 指针，而平衡树需要存储更多的颜色和旋转元数据。

练习 3/3: 性能权衡与实践选择

答案是 B. 实现简易性与并发性

关于干扰项的说明

练习 3/3: 性能权衡与实践选择

答案是 B. 实现简易性与并发性

- **实现简易性:** 跳表的指针操作比平衡树复杂的旋转和着色规则要简单得多，减少了开发和维护成本。
- **并发优势:** 平衡树的结构修改（旋转）都可能影响到树的大部分，难以并发；而跳表的修改是局部的，天然支持细粒度锁，甚至更容易实现无锁数据结构。

关于干扰项的说明

练习 3/3: 性能权衡与实践选择

答案是 B. 实现简易性与并发性

- **实现简易性:** 跳表的指针操作比平衡树复杂的旋转和着色规则要简单得多，减少了开发和维护成本。
- **并发优势:** 平衡树的结构修改（旋转）都可能影响到树的大部分，难以并发；而跳表的修改是局部的，天然支持细粒度锁，甚至更容易实现无锁数据结构。

关于干扰项的说明

- **A 错误:** 跳表基于链表，节点的内存位置通常是分散的（局部性差）；平衡树节点通常更紧凑（局部性相对更好）。
- **C 错误:** 平衡树始终保证 $O(\log n)$ 最坏情况；跳表的最坏情况是 $O(n)$。
- **D 错误:** 跳表因为有多层指针，空间开销通常比平衡树更大。

对比分析：跳表 (Skip List) vs. 平衡树 (Balanced Tree)

跳表 (Skip List)

- **核心机制:** 链表 + 随机概率提升
- **实现复杂度:** → 较低
  只需简单的指针操作和随机数生成
- **并发支持:** → 优秀。
  天然支持细粒度锁和无锁操作
- **性能保证:** → 平均 $O(\log n)$
  依赖于概率，最坏情况 $O(n)$
- **空间开销:** 较高（多 $O(n)$ 个指针）

平衡树 (AVL/RB)

- **核心机制:** 二叉树 + 平衡规则
- **实现复杂度:** → 较高
  涉及复杂的旋转、着色等平衡逻辑
- **并发支持:** → 较差
  结构修改通常需要锁定较大范围
- **性能保证:** → 最坏 $O(\log n)$
  结构性保证，性能稳定
- **空间开销:** 较低（仅 $O(n)$ 额外空间）

对比分析：跳表 (Skip List) vs. 平衡树 (Balanced Tree)

跳表 (Skip List)

- **核心机制:** 链表 + 随机概率提升
- **实现复杂度:** → 较低
  只需简单的指针操作和随机数生成
- **并发支持:** → 优秀。
  天然支持细粒度锁和无锁操作
- **性能保证:** → 平均 $O(\log n)$
  依赖于概率，最坏情况 $O(n)$
- **空间开销:** 较高（多 $O(n)$ 个指针）

平衡树 (AVL/RB)

- **核心机制:** 二叉树 + 平衡规则
- **实现复杂度:** → 较高
  涉及复杂的旋转、着色等平衡逻辑
- **并发支持:** → 较差
  结构修改通常需要锁定较大范围
- **性能保证:** → 最坏 $O(\log n)$
  结构性保证，性能稳定
- **空间开销:** 较低（仅 $O(n)$ 额外空间）

本节内容概览

红黑树 (Red-Black Tree)：四大性质

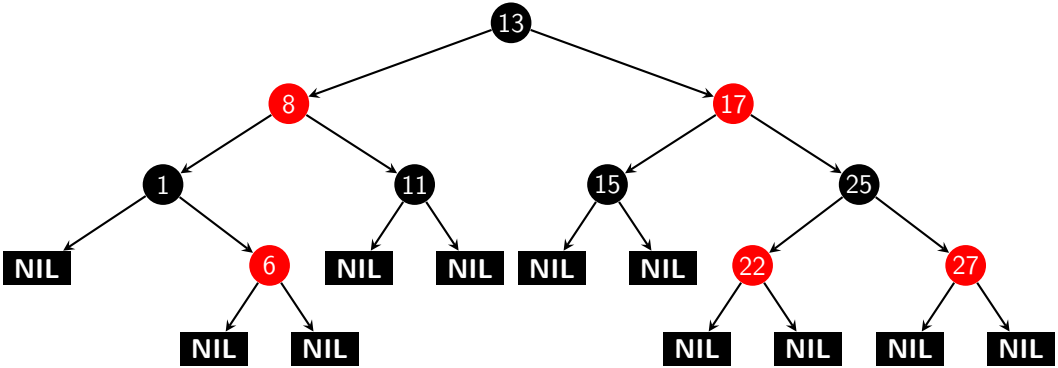
目标: 通过严格的颜色和排列规则，确保树的高度始终维持在 $O(\log n)$。

红黑树的四大性质 (核心约束)

1. 每个节点是红色或黑色
2. (根节点和) 所有的叶节点（NIL 节点，通常用空指针表示）都是黑色
3. 如果一个节点是红色，则它的两个子节点都必须是黑色
4. 从任一节点到其所有后代叶节点（NIL）的简单路径所含的黑色节点数量相同

关键点: 性质 3 和性质 4 是保证红黑树平衡的结构性基础。

Red-Black Tree Example



1. Every node is either **R** or **B**.
2. Root & NIL nodes are **B**.
3. **R** nodes cannot have **R** children.
4. Every path from node to descendant NIL has same number of **B** nodes.

RBT 四大性质 $\rightarrow O(\log n)$ 的树高

步骤 1: 路径长度约束

- **最短路径:** $h_{\min} = h_b$, 全是黑色 ($\mathbf{B} \rightarrow \mathbf{B} \rightarrow \dots$)
- **最长路径:** $h_{\max} = 2 \cdot h_b$, 颜色交替 ($\mathbf{B} \rightarrow \mathbf{R} \rightarrow \mathbf{B} \rightarrow \dots$)

步骤 2: 节点数 n 与 h_b 的关系

- **完美二叉树:** 给定 h_b，节点数最少的树是高度为 h_b 的完美二叉树。
- **节点数下界:** 具有 h_b 黑色高度的树，至少包含 $n \geq 2^{h_b} - 1$ 个节点:
- **推导 h_b 上限:** 通过对上述不等式变形，可以推导出 h_b 的上限:
  1. $n + 1 \geq 2^{h_b}$
  2. $\log_2(n+1) \geq h_b$

$$h_b \leq \log_2(n+1)$$

步骤 3: 代入步骤 1 的结论

$$h \leq 2 \cdot h_b \leq 2\log_2(n+1)$$

RBT 插入操作概览

核心思想: 首先遵循标准 BST 规则，然后通过**变色和旋转**修复红黑树性质。

RBT 插入操作概览

核心思想: 首先遵循标准 BST 规则，然后通过**变色和旋转**修复红黑树性质。

步骤 1: 遵循 BST 插入

1. **查找:** 从根节点开始向下查找新节点 X 的正确插入位置
2. **连接:** 将 X 插入到树中

步骤 2: 赋予初始颜色

- **颜色:** 新节点 X 初始为**红色**

为什么必须是红色？

RBT 插入操作概览

核心思想: 首先遵循标准 BST 规则，然后通过**变色和旋转**修复红黑树性质。

步骤 1: 遵循 BST 插入

1. **查找:** 从根节点开始向下查找新节点 **X** 的正确插入位置
2. **连接:** 将 **X** 插入到树中

步骤 2: 赋予初始颜色

- **颜色:** 新节点 **X** 初始为**红色**

为什么必须是红色？

- 若设为 **B**，则 **X** 路径上的 h_b 会加 1，这几乎必然导致树的整体 h_b 失衡
- 设为 **R** 不会破坏黑色高度 (h_b) 性质
- 设为 **R** 仅影响性质 3：无 **R** $\rightarrow$ **R** 连续节点

RBT 插入操作概览

核心思想: 首先遵循标准 BST 规则，然后通过**变色和旋转**修复红黑树性质。

步骤 1: 遵循 BST 插入

1. **查找:** 从根节点开始向下查找新节点 **X** 的正确插入位置
2. **连接:** 将 **X** 插入到树中

步骤 2: 赋予初始颜色

- **颜色:** 新节点 **X** 初始为**红色**

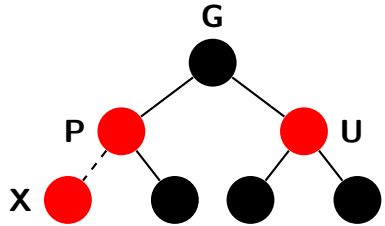
为什么必须是红色？

- 若设为 **B**，则 **X** 路径上的 h_b 会加 1，这几乎必然导致树的整体 h_b 失衡
- 设为 **R** 不会破坏黑色高度 (h_b) 性质
- 设为 **R** 仅影响性质 3：无 **R** $\rightarrow$ **R** 连续节点

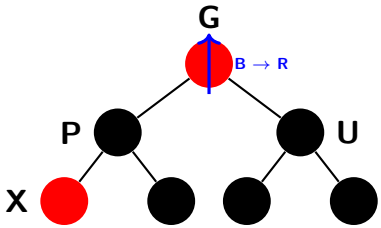
完成插入后，检查 **X** 的父节点 **P**。
如果 **P** 也是**红色**，则 **R** $\rightarrow$ **R** 冲突发生，
必须执行**插入修复算法**(Case 1，2，3) 来恢复红黑树的全部四条性质。

插入修复 Case 1：叔叔节点 U 为红色 (Recoloring)

修复前：R → R 冲突

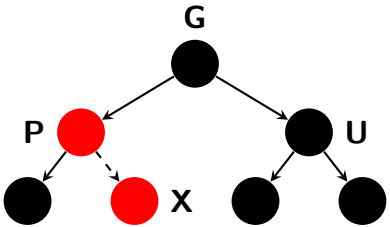


修复后：变色 (Recoloring)



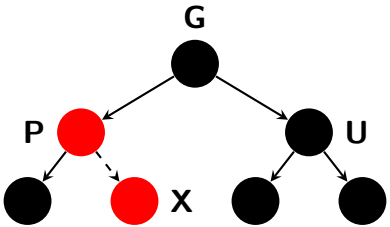
- **情况：** X 和父节点 P 连续，叔叔节点 U 为红色
- **效果：** 局部冲突解决，性质 3 得以恢复
- **转移：** 冲突可能转移到 G 及其新的父节点（需向上递归）
- **黑色高度：** h_b 保持不变，因为 G 处少了一个 B，但 P 和 U 处增加了 B

插入修复 Case 2 (1/2)：三角结构与旋转



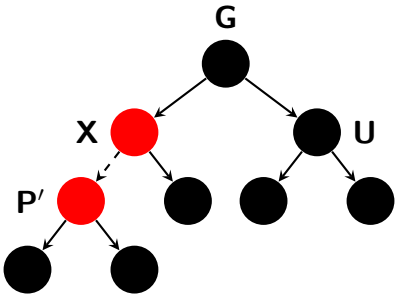
- **情况：** X 和父节点 P 连续，叔叔节点 U 为黑色，
  - **三角结构：** X 是父节点 P 的内侧子节点，形成 G-P-X 的三角结构
- **目标：** 将冲突结构从三角（G-P-X）转换为直线（G-X-P'）。

插入修复 Case 2 (1/2)：三角结构与旋转



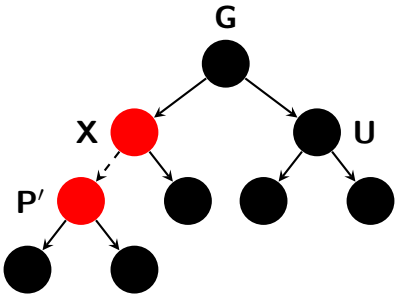
- **情况:** X 和父节点 P 连续，叔叔节点 U 为黑色，
  - **三角结构:** X 是父节点 P 的内侧子节点，形成 G-P-X 的三角结构
- **目标:** 将冲突结构从三角 (G-P-X) 转换为直线 (G-X-P')。
- **操作:** 对 P 进行 ↶ 左旋。
- **结果:** 节点 X 上升，取代 P 的位置；P 下沉成为 X 的左子节点。

插入修复 Case 2 (2/2)：转化为 Case 3 (直线结构)



- **转化结果:** 节点 **X** 现在是 **G** 的子节点，节点 **P′** 是 **X** 的子节点。
- **Case 3:** 现在的结构是直线 (G-X-P')，这正是 Case 3 的特征。

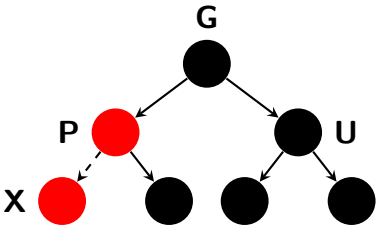
插入修复 Case 2 (2/2)：转化为 Case 3 (直线结构)



- **转化结果:** 节点 **X** 现在是 **G** 的子节点，节点 **P′** 是 **X** 的子节点。
- **Case 3:** 现在的结构是直线 (G-X-P')，这正是 Case 3 的特征。

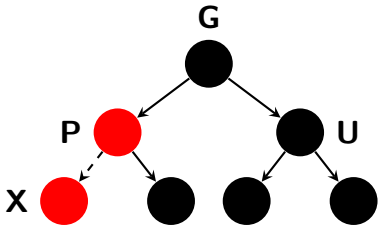
接下来我们只需要按 Case 3 处理即可

插入修复 Case 3 (1/2)：直线结构与旋转



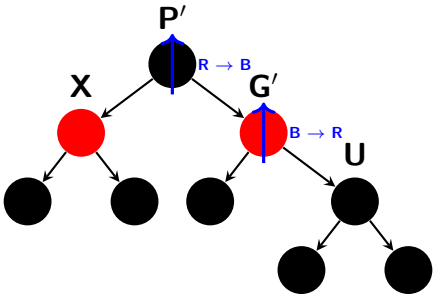
- **情况：** X 和父节点 P 连续，叔叔节点 U 为黑色，
  - **直线结构：** X 是父节点 P 的外侧子节点，形成 `G-P-X` 的直线结构

插入修复 Case 3 (1/2)：直线结构与旋转



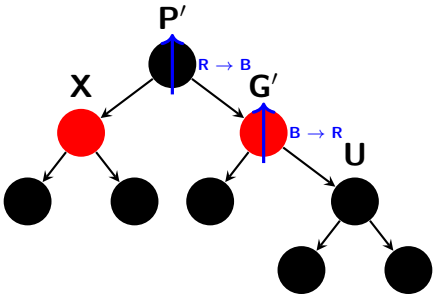
- **情况:** X 和父节点 P 连续，叔叔节点 U 为黑色，
  - **直线结构:** X 是父节点 P 的外侧子节点，形成 G-P-X 的直线结构
- **目标:** 将 P 提升到 G 的位置，消除 P 和 X 之间的 R $\rightarrow$ R 冲突
- **操作:** 对 G 进行右旋 $\curvearrowright$
- **结果:** P 成为新的局部根；G 成为 P 的右子

插入修复 Case 3 (2/2)：最终变色与平衡



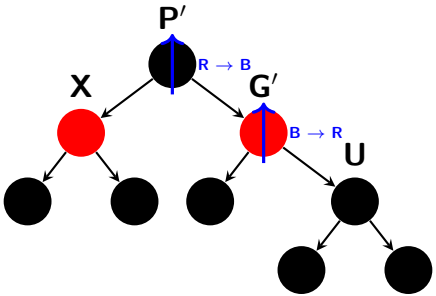
- **状态**：经过对 G 的右旋后，结构调整已完成，接下来进行变色
- **变色**：$P' \Rightarrow P'$；　$G' \Rightarrow G'$
- **效果**：冲突解决，性质 3 得以恢复

插入修复 Case 3 (2/2)：最终变色与平衡



- **状态**：经过对 G 的右旋后，结构调整已完成，接下来进行变色
- **变色**：$P' \Rightarrow P'$；$G' \Rightarrow G'$
- **效果**：冲突解决，性质 3 得以恢复
- **传递**：新子树根 G' 为黑，不会形成新的 $R \to R$ 冲突
- **黑色高度**：h_b 在所有路径上保持不变

插入修复 Case 3 (2/2)：最终变色与平衡



- **状态**：经过对 G 的右旋后，结构调整已完成，接下来进行变色
- **变色**：$P' \Rightarrow P'$；　$G' \Rightarrow G'$
- **效果**：冲突解决，性质 3 得以恢复
- **传递**：新子树根 G' 为黑，不会形成新的 $R \to R$ 冲突
- **黑色高度**：　h_b 在所有路径上保持不变

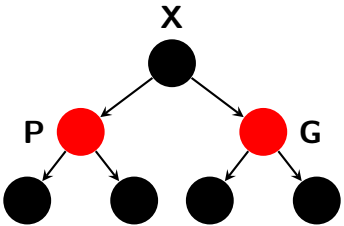
修复在 P' 处终止，红黑树性质完全恢复

练习 1/1：红黑树双旋转后的颜色修复 (LR 情况)

问题（双旋转后的颜色）：
在左-右 (**LR**) 双旋转完成并变色后，**X** 的新孩子 **P** 和 **G** 的最终颜色应该是什么？

A. **P** 是 **B**，**G** 是 **R**
B. 两者都保持原来的颜色
C. 两者都变成 **B**
D. 两者都变成 **R**

练习 1/1：红黑树双旋转后的颜色修复 (LR 情况)



LR 双旋后的最终状态

红黑树：删除操作概览

核心挑战： 移除一个黑色节点会破坏黑色高度 (h_b) 性质，必须修复。

阶段一：确定实际删除节点 N

1. **节点 N 有两个非 NIL 子节点**

红黑树：删除操作概览

核心挑战： 移除一个黑色节点会破坏黑色高度 (h_b) 性质，必须修复。

阶段一：确定实际删除节点 N

1. **节点 N 有两个非 NIL 子节点**
   - 找到 N 的后继节点 S
   - 交换 N 和 S 的数据，将 N 设为 S
   - 现在 N 最多一个非 `NIL` 子节点
2. **节点 N 最多有一个非 NIL 子节点**

红黑树：删除操作概览

核心挑战： 移除一个黑色节点会破坏黑色高度 (h_b) 性质，必须修复。

阶段一：确定实际删除节点 N

1. **节点 N 有两个非 NIL 子节点**
   - 找到 N 的后继节点 S
   - 交换 N 和 S 的数据，将 N 设为 S
   - 现在 N 最多一个非 NIL 子节点
2. **节点 N 最多有一个非 NIL 子节点**
   - N 就是实际要移除的节点
3. 该实际删除节点 **N**，将由它的唯一非 NIL 子节点 **S** 或 NIL 来取代

阶段二：确定修复类别

- **红 N，则必有两个黑子节点 NIL**

红黑树：删除操作概览

核心挑战： 移除一个黑色节点会破坏黑色高度 (h_b) 性质，必须修复。

阶段一：确定实际删除节点 N

1. **节点 N 有两个非 NIL 子节点**
   - 找到 N 的后继节点 S
   - 交换 N 和 S 的数据，将 N 设为 S
   - 现在 N 最多一个非 NIL 子节点
2. **节点 N 最多有一个非 NIL 子节点**
   - N 就是实际要移除的节点
3. 该实际删除节点 **N**，将由它的唯一非 NIL 子节点 **S** 或 NIL 来取代

阶段二：确定修复类别

- **红 N，则必有两个黑子节点 NIL**
  - 直接删，不影响 h_b，修复结束 ■
- **黑 N，且有一个红子节点 S**

红黑树：删除操作概览

核心挑战： 移除一个黑色节点会破坏黑色高度 (h_b) 性质，必须修复。

阶段一：确定实际删除节点 N

1. **节点 N 有两个非 NIL 子节点**
   - 找到 N 的后继节点 S
   - 交换 N 和 S 的数据，将 N 设为 S
   - 现在 N 最多一个非 NIL 子节点
2. **节点 N 最多有一个非 NIL 子节点**
   - N 就是实际要移除的节点
3. 该实际删除节点 **N**，将由它的唯一非 NIL 子节点 **S** 或 NIL 来取代

阶段二：确定修复类别

- **红 N，则必有两个黑子节点 NIL**
  - 直接删，不影响 h_b，修复结束 ■
- **黑 N，且有一个红子节点 S**
  - **S** 代替 **N**，并将 **S** 染成黑色
  - h_b 恢复平衡，修复结束 ■
- **黑 N，且有两个黑子节点 NIL**

红黑树：删除操作概览

核心挑战： 移除一个黑色节点会破坏黑色高度 (h_b) 性质，必须修复。

阶段一：确定实际删除节点 N

1. **节点 N 有两个非 NIL 子节点**
   - 找到 N 的后继节点 S
   - 交换 N 和 S 的数据，将 N 设为 S
   - 现在 N 最多一个非 NIL 子节点
2. **节点 N 最多有一个非 NIL 子节点**
   - N 就是实际要移除的节点
3. 该实际删除节点 **N**，将由它的唯一非 NIL 子节点 **S** 或 NIL 来取代

阶段二：确定修复类别

- **红 N，则必有两个黑子节点** NIL
  - 直接删，不影响 h_b，修复结束 ■
- **黑 N，且有一个红子节点 S**
  - **S** 代替 **N**，并将 **S** 染成黑色
  - h_b 恢复平衡，修复结束 ■
- **黑 N，且有两个黑子节点** NIL
  - **N** 被删后，相关路径亏欠一个黑色
  - 换言之，**N** 的继承者是**双重黑色**

红黑树：删除操作概览

核心挑战： 移除一个黑色节点会破坏黑色高度 (h_b) 性质，必须修复。

阶段一：确定实际删除节点 N

1. **节点 N 有两个非 NIL 子节点**
   - 找到 N 的后继节点 S
   - 交换 N 和 S 的数据，将 N 设为 S
   - 现在 N 最多一个非 NIL 子节点
2. **节点 N 最多有一个非 NIL 子节点**
   - N 就是实际要移除的节点
3. 该实际删除节点 **N**，将由它的唯一非 NIL 子节点 **S** 或 NIL 来取代

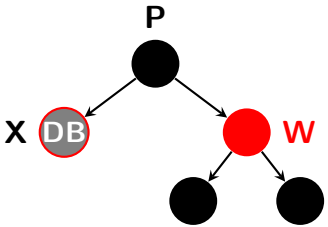
阶段二：确定修复类别

- **红 N，则必有两个黑子节点 NIL**
  - 直接删，不影响 h_b，修复结束 ■
- **黑 N，且有一个红子节点 S**
  - **S** 代替 **N**，并将 **S** 染成黑色
  - h_b 恢复平衡，修复结束 ■
- **黑 N，且有两个黑子节点 NIL**
  - **N** 被删后，相关路径亏欠一个黑色
  - 换言之，**N** 的继承者是**双重黑色**

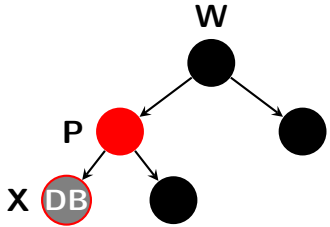
双重黑色情况，必须执行删除修复算法 (Case 1 到 Case 4) 来消除

删除修复 Case 1：兄弟节点 W 为红色

Case 1 (修复前结构)

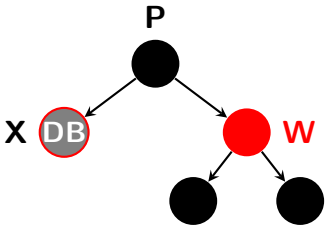


转化后：进入 Case 2/3/4

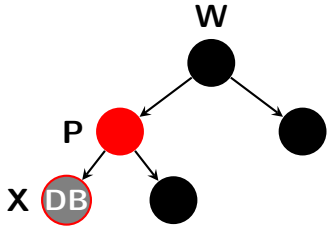


删除修复 Case 1：兄弟节点 W 为红色

Case 1 (修复前结构)



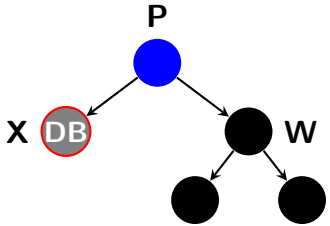
转化后：进入 Case 2/3/4



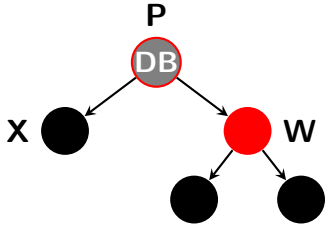
- **转化操作**：①对 P 进行左旋；② W 染成黑色，P 染成红色。
- **转化结果**：兄弟为黑色（Case 2/3/4）.

删除修复 Case 2：兄弟 W 和侄子节点均为黑色

Case 2 (修复前结构)



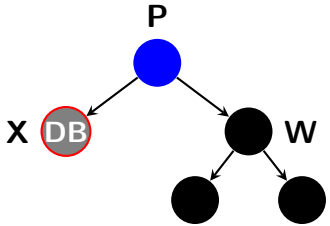
转化后：修复，但可能需要递归



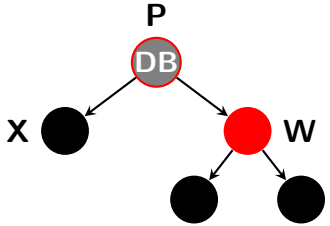
1. **兄弟转红:** 黑色 W → 红色 W

删除修复 Case 2：兄弟 W 和侄子节点均为黑色

Case 2 (修复前结构)



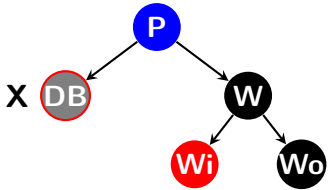
转化后：修复，但可能需要递归



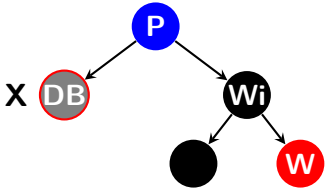
1. **兄弟转红:** 黑色 W $\rightarrow$ 红色 W
2. **向上传黑：** X 将多余的黑色传给 P
   1. 若 P 原色为红，则 P 则改为黑色，修复结束 ■
   2. 若 P 原色为黑，则 P 成为新的双黑，向上递归修复操作

删除修复 Case 3：兄弟及内外侄子为黑红黑

Case 3 (修复前结构)

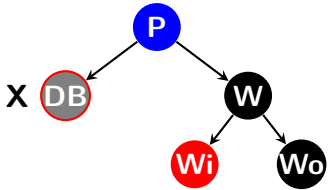


转化后：进入 Case 4

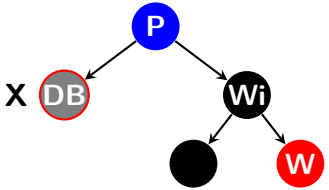


删除修复 Case 3：兄弟及内外侄子为黑红黑

Case 3 (修复前结构)



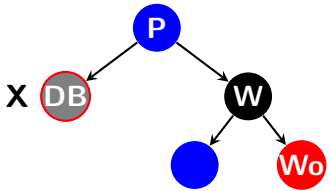
转化后：进入 Case 4



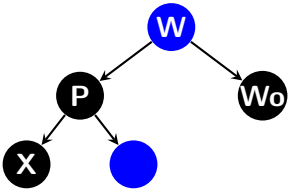
- **操作:** ① 对 W 右旋； ② W 和 W_l 交换颜色
- **结果:** X 的新兄弟是黑色 W_l，新外侄是红色 W（Case 4）.

删除修复 Case 4：兄弟及内外侄子为黑黑红或黑红红

Case 4 (修复前结构)



最终修复后的结构



- **修复操作:** ①对 P 左旋，并让 P 和 W 交换颜色；② 红色W_O → 黑色 W_O
- **修复结果:**
  1. W_O 变黑弥补了其路径上消失的一个黑节点（原 W）
  2. 下降的 P 分担了 X 多余的那个黑
  3. 性质 3 和 4 都得以修复，且根节点颜色未变，无需递归 ■

总结：红黑树删除修复的四种情况 (消除 DB)

兄弟-内外侄	主要操作	结果
1: 红黑黑	对 **P** **左旋**，并交换颜色	→ Case 2/3/4
2: 黑黑黑	**X**, **W** 各减一黑，**P** 加一黑	取决于 **P** 的原色，■ 或递归
3: 黑红黑	对 **W** **右旋**，并交换颜色	→ Case 4
4: 黑某红	对 **P** **左旋**，并交换颜色；$\mathbf{W_o}$ 变黑	■

表: **X** 双黑，$\mathbf{P}, \mathbf{W}, \mathbf{W_i}, \mathbf{W_o}$ 分别是父亲、兄弟、内侄和外侄

练习 1/1：删除修复的终止条件

问题：红黑树的删除修复过程（双黑的消除）在什么时候可以保证终止？

A. 当双黑标记向上转移到根节点时。

B. 当双黑标记向上转移到父节点 **P** 时，且 **P** 是红色.

C. 当执行了两次旋转（双旋转）时。

D. 选项 **A** 和 **B** 都是修复终止的条件。

练习 1/1：删除修复的终止条件

解

正确答案：*D.* 选项 *A* 和 *B* 都是修复终止的条件。

原理分析

- **直接消除终止**（Case 4，Direct Termination）
- **传播终止**（Case 2，Propagation Termination）
  1. DB 到达根节点，此时可以直接移除 DB 标记
  2. Case 2 + **P**

练习 1/1：删除修复的终止条件

解

正确答案：*D.* 选项 *A* 和 *B* 都是修复终止的条件。

原理分析

- **直接消除终止**（Case 4，Direct Termination）
- **传播终止**（Case 2，Propagation Termination）
  1. DB 到达根节点，此时可以直接移除 DB 标记
  2. Case 2 + **P**
- Case 1 → Case 2/3/4
- Case 3 → Case 4

练习 1/1：删除修复的终止条件

解

正确答案：*D.* 选项 *A* 和 *B* 都是修复终止的条件。

原理分析

- **直接消除终止**（Case 4，Direct Termination）
- **传播终止**（Case 2，Propagation Termination）
  1. DB 到达根节点，此时可以直接移除 DB 标记
  2. Case 2 + **P**
- Case 1 $\rightarrow$ Case 2/3/4
- Case 3 $\rightarrow$ Case 4
- Case 2 $\rightarrow$ Case 1/2/3/4 with $\Uparrow$

红黑树 (RBT) 与伸展树 (Splay Tree) 对比

特性	红黑树 (RBT)	伸展树 (Splay Tree)
平衡机制	**基于颜色** (黑高)，增删后局部修复	**基于访问** (自适应)，访问节点旋转至根
平衡类型	**严格平衡**，最坏 $O(\log n)$	**摊还平衡**，最坏情况可能 $O(n)$
查找/插入/删除	$O(\log n)$ (最坏)	$O(\log n)$ (摊还平均)
空间复杂度	$O(n)$ (额外的 **1** bit 颜色信息)	$O(n)$ (仅需存储节点数据和指针)
局部性/缓存	不关注局部性，结构相对稳定	**优化局部性**，常访问节点靠近根
实现复杂度	you know it	**相对简单** (zig, zag 等旋转)
核心应用	std::map/set，内核调度，内存管理	缓存系统，垃圾回收器，文件系统

关键差异总结

- **红黑树**提供严格的最坏情况性能保证，适用于对时间波动敏感的场景
- **伸展树**依赖访问频率，实现简洁，适用于访问模式具有高度局部性的场景

本节内容概览

1. 跳表
2. 红黑树
3. 伸展树
   - Splay: Zig, Zig-Zig, Zig-Zag
   - 查找、插入与删除
   - 摊还分析

伸展树核心：Zig (单旋转) 操作 (1/2)

伸展操作 (Splay) 的目标是将最近访问的节点 x 通过一系列旋转移动到树的根部 r，以实现摊还 $O(\log n)$ 平衡

场景：Zig 旋转 (单旋转)

触发条件: 节点 x 是根节点 r 的直接孩子

旋转方向与目的

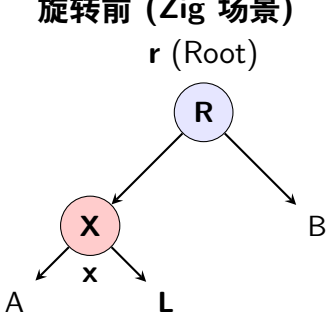
1. **旋转操作:** 如果 x 是左孩子，执行右旋。如果 x 是右孩子，执行左旋
2. **旋转目的:** 将被访问的 x 节点立即提升为树的新根

关键问题回顾

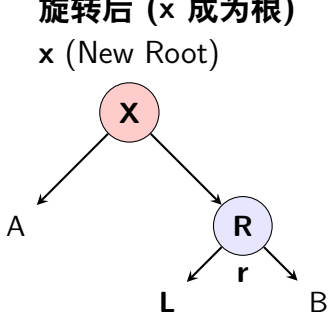
如果 x 是 r 的左孩子，对 r 右旋后，x 和 r 的相对位置和父子关系会如何变化？

伸展树核心：Zig 操作 (2/2)

旋转前 (Zig 场景)



旋转后 (x 成为根)



- **父子关系变化:** 旋转后，原根 **r** 成为 **x** 的右孩子
- **子树 L 转移:** **x** 的原右子树 **L** 转移成为 **r** 的左孩子，以维持 BST 性质。

伸展树核心：Zig-Zig 操作 (1/2)

场景： 节点 x、p 和 g 呈直线排列（同为左子或同为右子）

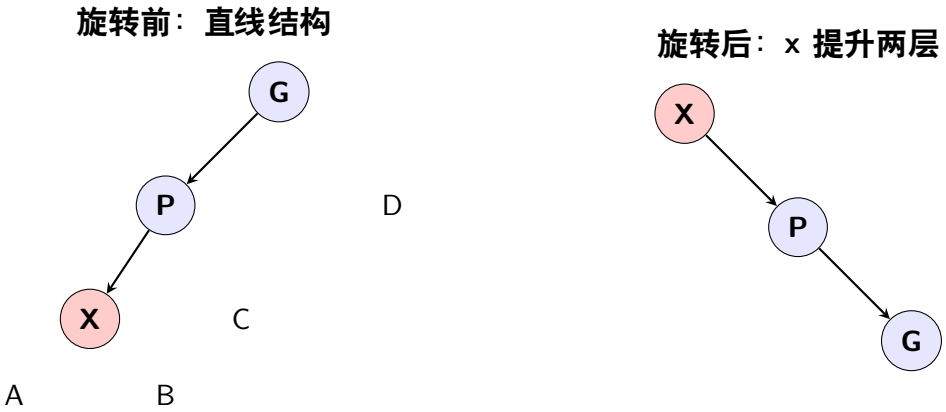
关键序列：两次同向旋转（先转 g 后转 p）→ x 连升两级

1. **第一次旋转 (g)：** p 成为 g 的父节点
2. **第二次旋转 (p)：** x 成为整个局部结构的新根

顺序不能弄反

- Zig-Zig 序列是 Splay Tree 实现摊还 $O(\log n)$ 性能的核心
- 如果先 p 后 g，在最坏情况下性能会退化。

伸展树核心：Zig-Zig 操作 (2/2)



- **最终结构:** x 成为局部的新根，p 成为 x 的右子，g 成为 p 的右子。
- **子树转移:** 原本的子树 B 和 C 转移到新 P 和新 G 的左侧，保持 BST 性质。

伸展树核心：Zig-Zag 操作 (1/2)

场景: Zig-Zag 序列用于处理 **x**、**p** 和 **g** 呈折线排列的情况。

与 Zig-Zig 的区别

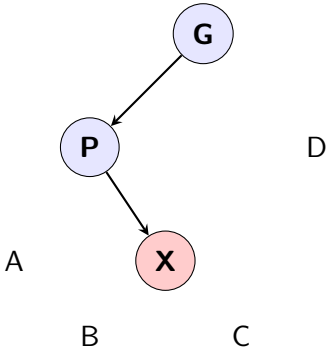
- **Zig-Zig (直线):** 两次旋转方向相同，且顺序是先 **g** 后 **p**
- **Zig-Zag (折线):** 两次旋转方向相反，且顺序是先 **p** 后 **g**
- 目标: Zig-Zag 通过两次方向相反的旋转，将 **x** 一次性提升两层

操作序列：两次方向相反的旋转

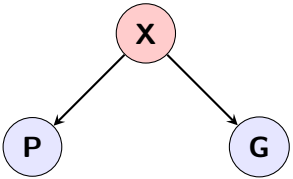
1. **第一次旋转 (在 p 处):** 旋转 **p**，将 **x** 提升到 **p** 的位置。(即 Zig 操作)
2. **第二次旋转 (在 g 处):** 旋转 **g**，将 **x** 提升到 **g** 的位置。(即 Zag 操作)

伸展树核心：Zig-Zag 操作 (2/2)

旋转前：折线结构



旋转后：x 提升两层



- **双旋转后:** x 成为局部的新根，p 和 g 成为 x 的两个孩子
- **子树转移:** 原本位于 x 左右两边的子树 (B 和 C) 转移成为 p 和 g 的子树

伸展树的查找操作

核心机制

- **基础过程**：与标准二叉搜索树（BST）**完全相同**
- **关键差异**：查找结束后，立即对**最后一个非空访问节点**执行伸展

关键特性

- **自调整性**：最近访问的节点移至根部，优化后续操作
- **摊还效率**：单次最坏 $O(n)$，但连续操作摊还 $O(\log n)$
- **统一处理**：无论查找成功（找到目标）或失败（到达空指针），均执行伸展

注：伸展操作通过 zig/zig-zig/zig-zag 旋转序列实现

Splay 树插入操作

插入算法

1. 像普通 BST 一样插入新节点 x
2. 对节点 x 执行 `splay(`x`)` 操作，将其移到根位置
3. 插入完成后，x 成为新的根节点

时间复杂度： $O(\log n)$ 摊还时间复杂度，与查找操作相同

Splay 树删除操作

删除算法

1. 通过查找定位要删除的节点 x
2. 对节点 x 执行 `splay(`x`)` 操作，将其移到根位置
3. 移除根节点，将左右子树**合并**
   - 对左子树执行 `splay(max)`，将最大元素移到左子树根
   - 将右子树作为新根的右子树

时间复杂度： $O(\log n)$ 摊还时间复杂度，包括查找、`splay` 和合并

问题：昂贵操作 vs 长期收益

核心矛盾

- 伸展操作最坏情况：$O(n)$ 时间复杂度
- 但序列操作的摊还成本：$O(\log n)$

关键问题

当我们将深度为 d 的节点 x 移至根部时，
如何证明这昂贵的 $O(d)$ 操作能减少未来操作的总成本？

摊还分析工具：势能方法 (Potential Method)

势能函数定义

对伸展树 T，定义势能函数：

$$\Phi(T) = \sum_{x \in T} \log_2(\text{size}(x))$$

其中 $\text{size}(x) =$ 以 x 为根的子树中的节点数

完整形式化证明： https://oiwiki.org/ds/splay/# 时间复杂度

摊还分析工具：势能方法 (Potential Method)

关键概念

- **势能**：衡量树的"不平衡程度"，值越大表示树越不平衡
- **摊还成本** = 实际成本 + 势能变化 $\Delta\Phi$
- **序列总成本** = 总摊还成本 − 净势能变化

$$\sum \text{实际成本}_i = \sum \text{摊还成本}_i - (\Phi_{\text{final}} - \Phi_{\text{initial}})$$

操作	实际成本	摊还成本
访问浅层节点	低	低
访问深层节点	高	中等

注：势能函数的设计使伸展操作显著降低树的势能，抵消其高实际成本

局部性原理：摊还优势的现实基础

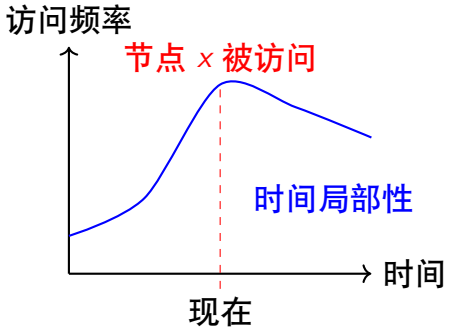
时间局部性

- **定义**：刚访问的节点很可能再被访问
- **伸展树优化**：
  - 将访问节点移至根部
  - 下次访问只需 $O(1)$ 时间
  - 示例：缓存、编译器符号表

空间局部性

- **定义**：其附近节点很可能被访问
- **伸展树优化**：
  - 伸展操作提升整个访问路径
  - 路径上的节点深度平均减少 50%
  - 示例：数据库范围查询

局部性原理：摊还优势的现实基础



实证数据: 实际工作负载中，80% 的访问集中在 20% 的数据上（帕累托原则）

序列操作：从最坏情况到平衡状态

初始状态：退化为链表的树 (最坏情况)

累积效果

- **第一次访问**：实际成本高 ($O(n)$)，但将树重构为平衡状态
- **后续访问**：所有操作成本降至 $O(\log n)$
- **总摊还成本**：m 次操作的总成本 $\leq O(m \log n + n)$

$$\text{平均成本} = O(\log n + n/m)$$

当 $m \gg n$ 时，平均成本 $\approx O(\log n)$

总结：伸展操作的摊还价值

核心价值主张：预付重构成本

- **短期**：支付 $O(d)$ 重构成本
- **长期**：
  - 目标节点下次访问：$O(1)$
  - 访问路径上所有节点：深度减少 30-50%
  - 树的整体平衡：高度降低，方差减少

- **缓存友好**：自动将热点数据提升至树顶
- **简单高效**：实现简单，常数因子小
- **适用场景**：缓存、垃圾回收、网络路由表等具有局部性的工作负载

摊还分析的本质：为未来操作购买保险
—— 今天的重构成本，是明天高效访问的预付款

练习 1/2：摊还时间复杂度

问题

伸展树的摊还时间复杂度 $O(\log n)$ 意味着什么？

选项

A. 所有操作的运行时间都精确地等于 $O(\log n)$。
B. 树的高度始终保持在 $O(\log n)$，因此每次操作都是 $O(\log n)$。
C. 对于任意 m 个操作的序列，总时间是 $O(m \log n)$。
D. 在随机输入的情况下，平均运行时间是 $O(\log n)$。

练习 1/2：正确答案与解析

正确答案：C

对于任意 m 个操作的序列，总时间是 $O(m \log n)$。

- 摊还分析的核心：**序列总成本** $= O(m \log n)$
- 单次操作可能为 $O(n)$，但 m 次操作的平均成本为 $O(\log n)$
- 数学表达：$\sum_{i=1}^{m} \text{cost}(op_i) \leq c \cdot m \log n$（$c$ 为常数）

错误选项分析

A 伸展树单次操作最坏情况为 $O(n)$，不是所有操作都精确为 $O(\log n)$

B 伸展树不保证高度始终为 $O(\log n)$，可能暂时退化

D 摊还分析对**任意**操作序列成立，不依赖随机输入假设

平均情况 vs. 摊还分析

分析类型	假设	伸展树性能
最坏情况	无	$O(n)$ 单次操作
平均情况	随机输入	$O(\log n)$
摊还分析	**任意序列**	$\mathbf{O(\log n)}$

关键洞见：摊还分析是”长期保证”，而非”每次保证”。它允许短期波动，确保长期平均高效。

练习 2/2：Zig-Zag 与 Zig-Zig 优势对比

问题

Zig-Zag 旋转序列（x 是 p 的左孩子，p 是 g 的右孩子）相比于 Zig-Zig，它的主要区别和优势是什么？

选项

A. Zig-Zag 比 Zig-Zig 旋转次数更少，更高效。

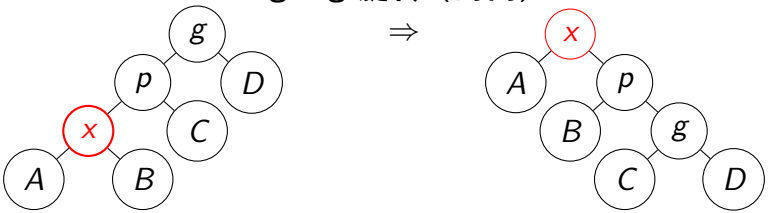
B. Zig-Zag 可以更好地降低 x 原始子树的高度。

C. Zig-Zag 只用于删除操作，Zig-Zig 只用于查找操作。

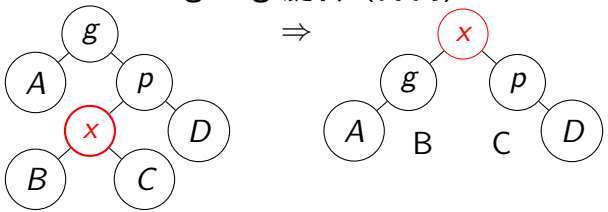
D. Zig-Zag 旋转后，g 始终成为 x 的父节点。

练习 2/2：Zig-Zag 与 Zig-Zig 优势对比

Zig-Zig 旋转（同侧）



Zig-Zag 旋转（异侧）



欢迎提问！

你的疑惑，我的动力