第 11 章 查找

静态查找、二叉查找树、AVL 树、散列查找

郑新、徐鹏飞、李健

2025 年 12 月 3 日

本节内容概览

什么是静态查找？

定义

静态查找是指在查找过程中，数据集合是固定不变的。

- **不能**：增加、删除或修改数据。
- **核心操作**：只进行**查询**（Query）操作。

现实场景

想象你在查阅一个不会变的目录或清单：

- **查字典**：查找一个已收录的单词。
- **查询学号**：在已录入的班级学号列表中查找特定学生。
- **查找商品目录**：在一个季度不变的商品清单中寻找商品。

方法一：顺序查找

查找思路：从头到尾 ($O(N)$)

步骤：从数组的第一个元素开始，依次向后比较，直到：

1. 找到目标元素，或
2. 遍历完整个数组。

代码思路

使用一个简单的 `for` 循环即可实现。

时间复杂度分析

- **最好情况**： $O(1)$ (第一个元素就是目标)。
- **最坏/平均情况**： $O(N)$ (需要遍历约 N 次)。
- **适用性**：数据量很小，或数据完全无序时。

方法二：二分查找

强制前提条件

数组必须是有序的！ （升序或降序均可）

查找思路：分治法

核心思想：每次查找都与当前查找区间的中间元素比较，然后排除一半的数据。

1. 比较中间元素 arr[M] 与 target。
2. 如果 arr[M] $>$ target，则到左半部分查找。
3. 如果 arr[M] $<$ target，则到右半部分查找。

最终，数据量会以 $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots$ 的速度快速缩小。

效率对比

性能对比：$O(N)$ vs $O(\log N)$

数据规模 N 越大，效率差距越是天壤之别！

表：查找操作次数对比

数据规模 (N)	顺序查找 ($O(N)$)	二分查找 ($O(\log N)$)
10^3 (1 千)	1,000 次	10 次
10^6 (1 百万)	1,000,000 次	20 次

方法三：索引查找

核心思想：分块索引

将数据分块，先查目录，再查内容。就像查阅一本有目录页（索引）的大部头书。

查找结构

1. **数据区**：存储所有实际数据，分成 K 个大小相近的块。
2. **索引区**：存储 K 个块的目录信息（如每块的最大值、起始位置）。

查找流程

1. **第一步（查目录）**：在索引区中快速定位目标可能位于哪个数据块。
2. **第二步（查内容）**：在找到的数据块内进行顺序查找（或二分查找）。

索引查找复杂度分析 ($O(\sqrt{N})$)

最优分块策略

假设：数组大小为 N，我们将其分成 K 个块，则每块大小约为 $\frac{N}{K}$。

索引查找 (查 K 个块)：
顺序：$O(K)$，二分：$O(\log K)$

块内查找 (查 $\frac{N}{K}$ 个元素)：$O(\frac{N}{K})$

总时间成本 $\approx O(K + \frac{N}{K})$

最优复杂度

当块数 K 取 $\sqrt{N}$ 时，总时间成本最低：

$$O(\sqrt{N} + \frac{N}{\sqrt{N}}) = O(2\sqrt{N}) = O(\sqrt{N})$$

效率：比 $O(N)$ 快，但比 $O(\log N)$ 慢。

效率对比与总结

表: 查找操作次数对比

方法	复杂度	$N = 10^6$ 次数	$N = 10^9$ 次数
顺序查找	$O(N)$	1,000,000 次	1,000,000,000 次
索引查找	$O(\sqrt{N})$	1,000 次	31,622 次
二分查找	$O(\log N)$	20 次	30 次

总结与应用原则

- **最优**：数据有序时，使用 $O(\log N)$ 的二分查找。
- **折衷**：数据分块且静态时，使用 $O(\sqrt{N})$ 的索引查找。
- **最慢**：数据无序且数据量大时，顺序查找不可取。

本节内容概览

为什么需要树？

回顾数组的局限性

数组在面对动态变化时效率很低：

- **有序数组查找**： $O(\log N)$ ✓(快，通过二分查找)
- **有序数组插入/删除**： $\boldsymbol{O(N)}$ × (慢，需要移动大量元素)

我们面临的问题

有没有一种数据结构，能同时实现高效的查找和高效的修改操作？

$$\text{查找}\boldsymbol{O(\log N)} + \text{修改}\boldsymbol{O(\log N)}$$

为什么需要树？

回顾数组的局限性

数组在面对动态变化时效率很低：

- **有序数组查找**： $O(\log N)$ ✓(快，通过二分查找)
- **有序数组插入/删除**： $\boldsymbol{O(N)}$ × (慢，需要移动大量元素)

我们面临的问题

有没有一种数据结构，能同时实现高效的查找和高效的修改操作？

$$\text{查找}\boldsymbol{O(\log N)} + \text{修改}\boldsymbol{O(\log N)}$$

答案揭晓

树结构，特别是二叉查找树，能够保持有序性，同时避免大规模数据搬移。

二叉查找树 (BST) 的核心定义

节点结构

每个节点包含：

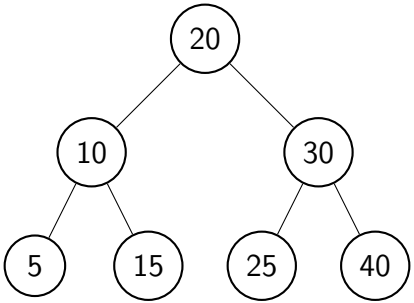
- **值**：存储的数据。
- **左子节点指针**：指向左子树的根节点。
- **右子节点指针**：指向右子树的根节点。

BST 核心性质

此性质对树中所有子树都成立：

- **左子树规则**：节点左子树中所有值都 $<$ 根节点的值。
- **右子树规则**：节点右子树中所有值都 $>$ 根节点的值。

示例：一个 BST



性质检验：$10 < 20,\ 30 > 20.\ 5 < 10,\ 15 > 10.$

BST 的查找操作 (search)

查找思路：与二分查找相似

目标：判断目标值是否在树中。

1. **起点**：从**根节点 (Root)** 开始。
2. **比较**：比较目标值与当前节点的值：
   - 如果相等，查找成功。
   - 如果目标值 $<$ 当前节点值，转向**左子树**。
   - 如果目标值 $>$ 当前节点值，转向**右子树**。
3. **结束**：如果遇到空指针，查找失败。

效率分析

- **时间复杂度**： $O(h)$，其中 h 是树的**高度**。
- **理想情况 (平衡树)**： $h \approx \log N$，复杂度为 $O(\log N)$。

BST 的插入操作 (insertion)

插入思路：一次查找 + 一个连接

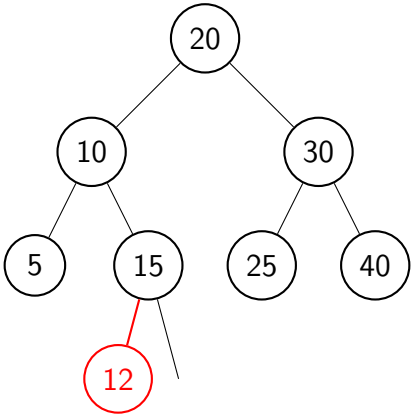
步骤：

1. **执行查找**：从根节点开始，执行一次查找操作，寻找新值应该在的位置。
2. **找到连接点**：沿着查找路径向下，直到找到一个应该连接新节点的**空指针**。
3. **挂载**：将新节点挂载到该空指针的位置。

效率分析

- **时间复杂度**： $O(h)$
- **理想情况**： $O(\log N)$

插入值 12 的过程



新节点 12 挂载在 15 的左侧，保持了 BST 性质。

BST 的删除操作 (deletion)

删除 D 的三种情况（假设已找到）

1. **情况 1：被删除节点 D 有 0 个子节点 (叶子节点)**
   - 找到父节点，将其指向 D 的指针设为 NULL。
   - 直接删除 D 节点。
2. **情况 2：被删除节点 D 有 1 个子节点**
   - 将 D 的子节点直接连接到 D 的父节点上，取代 D 的位置。
   - 删除 D 节点。
3. **情况 3：被删除节点 D 有 2 个子节点 (最复杂)**
   - **目标**：找到一个合适的值 S 来替换 D 的值，使得 BST 性质被保持。
   - **寻找**：寻找 D 的中序后继（即 D 的右子树中的最小节点）。
   - **替换**：用 S 的值替换 D 的值，然后**转为删除** S **节点**。

删除操作相对复杂，但也保持在 $O(h)$ 的效率。

BST 性能概览 (理想/平均情况)

BST 与数组操作复杂度对比

操作	BST 复杂度	有序数组复杂度
查找 (Search)	$\boldsymbol{O(\log N)}$	$O(\log N)$
插入 (Insertion)	$\boldsymbol{O(\log N)}$	$O(N)$
删除 (Deletion)	$\boldsymbol{O(\log N)}$	$O(N)$

注意：以上均为平均情况或平衡情况下的性能。

最坏情况

如果数据按顺序插入（例如 1, 2, 3, 4, 5...），会发生什么？

最坏情况

如果数据按顺序插入（例如 1, 2, 3, 4, 5...），会发生什么？

- **结果**：树会退化成一个链表！
- **最坏复杂度**：此时 $h = N$，所有操作复杂度退化为 $O(N)$！

最坏情况

如果数据按顺序插入（例如 1, 2, 3, 4, 5...），会发生什么？

- **结果**：树会退化成一个链表！
- **最坏复杂度**：此时 $h = N$，所有操作复杂度退化为 $O(N)$！

⇓

平衡二叉查找树

（如 AVL 树、红黑树），

它们能通过旋转操作保持 $h \approx \log N$！

本节内容概览

引入平衡：定义和目标

平衡二叉查找树 (Balanced BST)

这是一种能够在插入或删除操作后，通过自我调整机制（如旋转），将树的高度维持在 $O(\log N)$ 级别的 BST。

核心思想：局部维护

不是追求整体绝对平衡，而是确保局部失衡不会迅速传递到根节点。

- 每次操作后，从新插入/删除的节点向上检查。
- 如果检测到失衡，立即进行**局部调整** (旋转)。

严格平衡：AVL 树的定义

AVL 树：最早被发现的自平衡 BST，也是最严格的平衡

AVL 树是以发明者 G.M. Adelson-Velsky 和 E.M. Landis 命名的。

AVL 树的平衡条件

树中任意节点 N 的平衡因子 (Balance Factor, BF) 都必须满足：

$$-\mathbf{1} \leq \mathrm{BF}(\boldsymbol{N}) \leq \mathbf{1}, \text{ 其中}$$

$$\mathrm{BF}(N) = \mathrm{Height}(\mathrm{LeftSubtree}) - \mathrm{Height}(\mathrm{RightSubtree})$$

平衡因子状态

$\mathbf{BF} = \mathbf{0}$	$\mathbf{BF} = \mathbf{1}$	$\mathbf{BF} = \mathbf{-1}$
左右子树高度相等。	左子树比右子树高 1。	右子树比左子树高 1。

AVL 树插入算法总览

目标：保持平衡因子 $|BF| \leq 1$

AVL 插入操作在遵循 BST 规则后，必须向上回溯并检查所有祖先节点的平衡性。

1. **标准插入**：

AVL 树插入算法总览

目标：保持平衡因子 $|BF| \leq 1$

AVL 插入操作在遵循 BST 规则后，必须向上回溯并检查所有祖先节点的平衡性。

1. **标准插入**： 将新节点作为叶节点插入到正确位置（与普通 BST 一样）。
2. **向上遍历**：

AVL 树插入算法总览

目标：保持平衡因子 $|BF| \leq 1$

AVL 插入操作在遵循 BST 规则后，必须向上回溯并检查所有祖先节点的平衡性。

1. **标准插入**：将新节点作为叶节点插入到正确位置（与普通 BST 一样）。
2. **向上遍历**：从新插入的节点开始，沿父节点路径一直向上遍历到根节点。
3. **检查平衡**：

AVL 树插入算法总览

目标：保持平衡因子 $|\mathrm{BF}| \leq 1$

AVL 插入操作在遵循 BST 规则后，必须向上回溯并检查所有祖先节点的平衡性。

1. **标准插入**：将新节点作为叶节点插入到正确位置（与普通 BST 一样）。
2. **向上遍历**：从新插入的节点开始，沿父节点路径一直向上遍历到根节点。
3. **检查平衡**：在回溯路径上的每个节点 A 处，重新计算其平衡因子 $\mathrm{BF}(A)$。
4. **定位失衡**：

AVL 树插入算法总览

目标：保持平衡因子 $|\mathrm{BF}| \leq 1$

AVL 插入操作在遵循 BST 规则后，必须向上回溯并检查所有祖先节点的平衡性。

1. **标准插入**：将新节点作为叶节点插入到正确位置（与普通 BST 一样）。
2. **向上遍历**：从新插入的节点开始，沿父节点路径一直向上遍历到根节点。
3. **检查平衡**：在回溯路径上的每个节点 A 处，重新计算其平衡因子 $\mathrm{BF}(A)$。
4. **定位失衡**：找到第一个 $|\mathrm{BF}(A)| > 1$ 的祖先节点 A。
5. **应用旋转**：

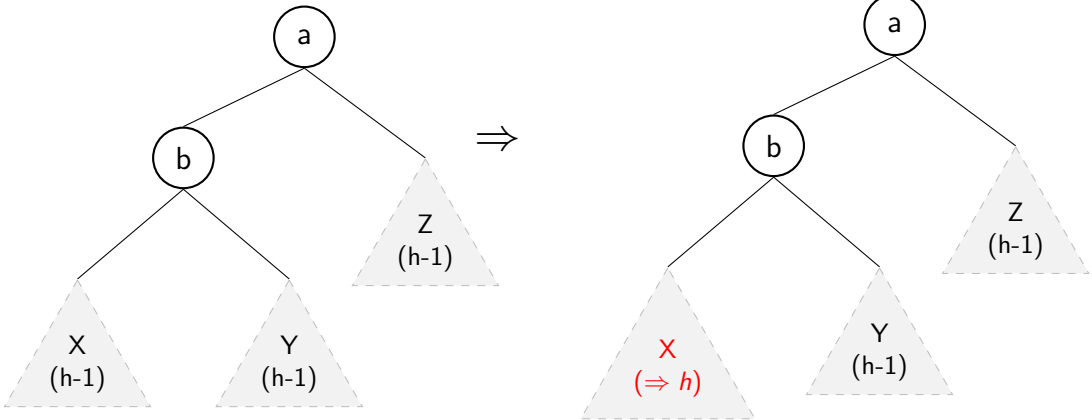
AVL 树插入算法总览

目标：保持平衡因子 $|\mathrm{BF}| \leq 1$

AVL 插入操作在遵循 BST 规则后，必须向上回溯并检查所有祖先节点的平衡性。

1. **标准插入**：将新节点作为叶节点插入到正确位置（与普通 BST 一样）。
2. **向上遍历**：从新插入的节点开始，沿父节点路径一直向上遍历到根节点。
3. **检查平衡**：在回溯路径上的每个节点 A 处，重新计算其平衡因子 $\mathrm{BF}(A)$。
4. **定位失衡**：找到第一个 $|\mathrm{BF}(A)| > 1$ 的祖先节点 A。
5. **应用旋转**：判断失衡类型并执行对应旋转：
   - **LL 失衡**：在 A 处执行 **单次右旋** ↷。
   - **RR 失衡**：在 A 处执行 **单次** ↶ **左旋** 。
   - **LR 失衡**：先 ↶ 左旋 (左孩节点)，后右旋 ↷ (关键节点)。
   - **RL 失衡**：先右旋 ↷ (右孩节点)，后 ↶ 左旋 (关键节点)。

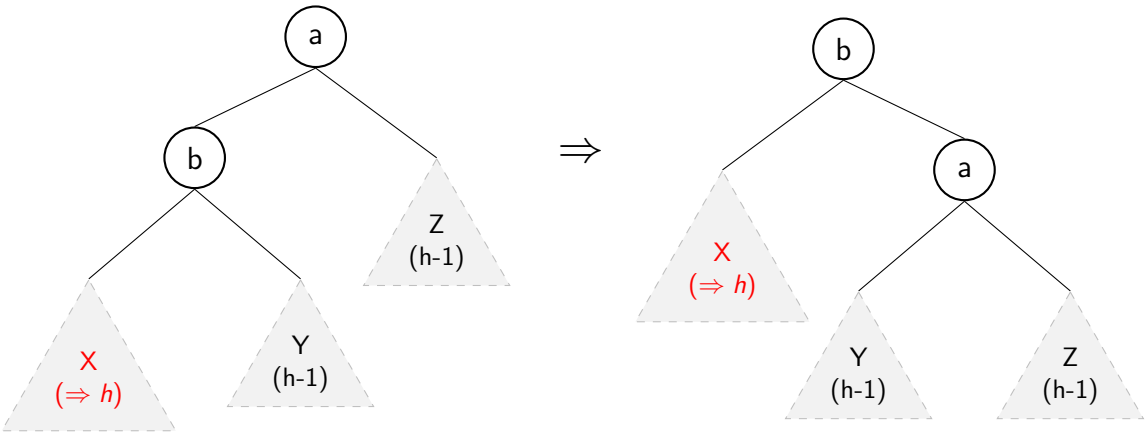
从平衡到失衡（左左情况）



BF(a)=1 （平衡）

BF(a)=2 （失衡）

通过“右旋 ↷”恢复 LL 失衡



BF(a)=2 （失衡）

BF(a)=BF(b)=0 （平衡）

若发生 LL 失衡，以“右旋 ↷”应对

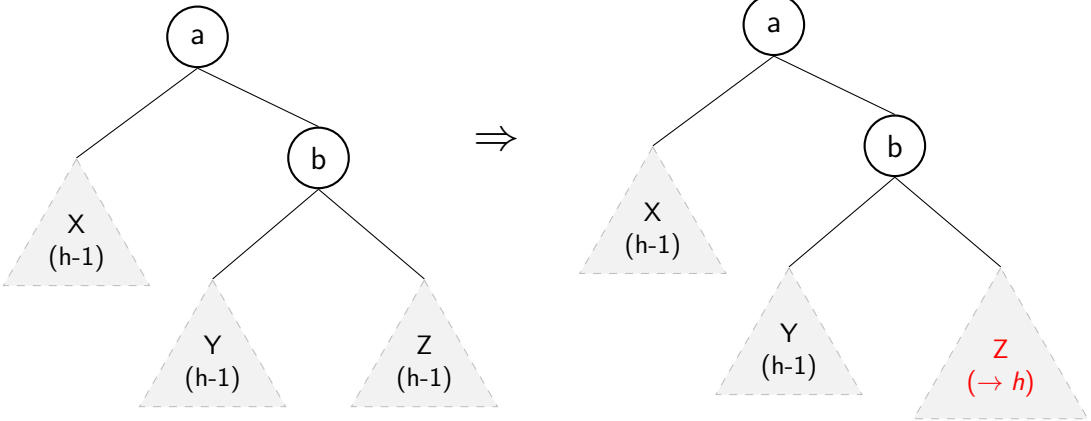
LL 失衡

触发条件：在节点 a 的左孩子 b 的左子树 X 中插入了新节点，导致 BF(a)=2.

右旋 ↷

- 修复操作应在不改变中序序列({X}b{Y}a{Z}) 的前提下恢复平衡。
- 右旋 ↷，效果是 a↓ b ↑.
  1. a 的左孩 := b 的右孩，即 Y 的根.
  2. b 的右孩 := a.
  3. a 的父节点指向 b（如果 a 是根，则更新根为 b）. (下文省略)

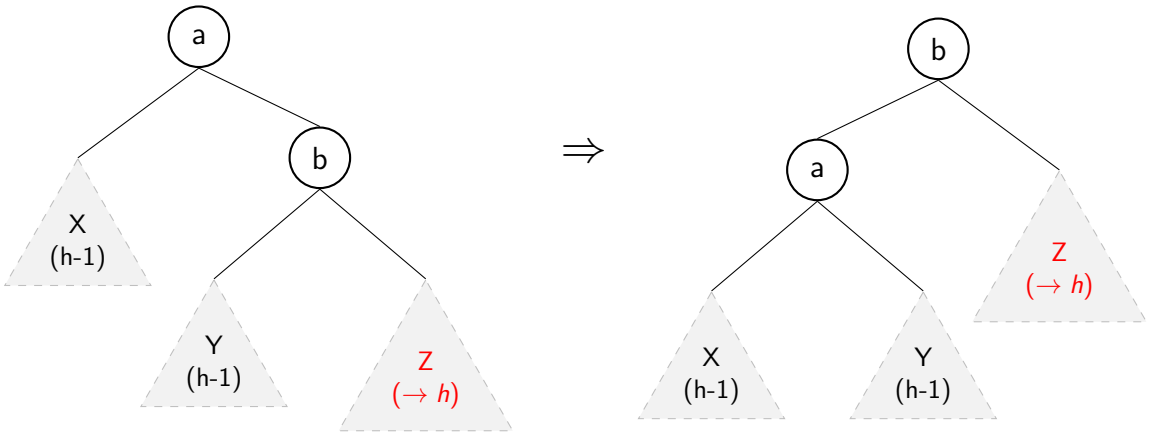
从平衡到失衡（右右情况）



BF(a)=-1 （平衡）

BF(a)=-2 （失衡）

通过“↶ 左旋”恢复 RR 失衡



BF(a)=-2 （失衡）

BF(a)=BF(b)=0 （平衡）

若发生 RR 失衡，以“↶ 左旋”应对

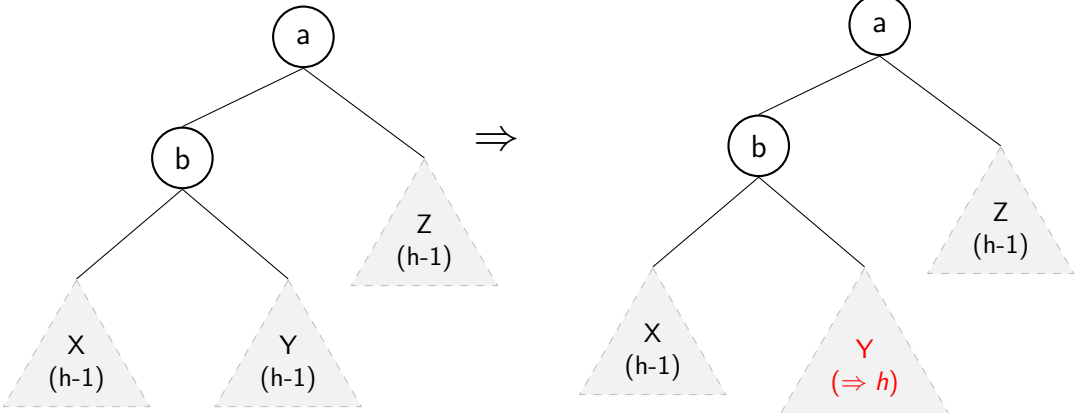
RR 失衡

触发条件：在节点 a 的右孩子 b 的右子树 Z 中插入了新节点，导致 BF(a)=-2.

↶ 左旋

- 修复操作应在不改变**中序序列**({X}a{Y}b{Z}) 的前提下恢复平衡。
- ↶ 左旋，效果是 a↓ b ↑.
  1. a 的右孩 := b 的左孩，即 Y 的根.
  2. b 的左孩 := a.

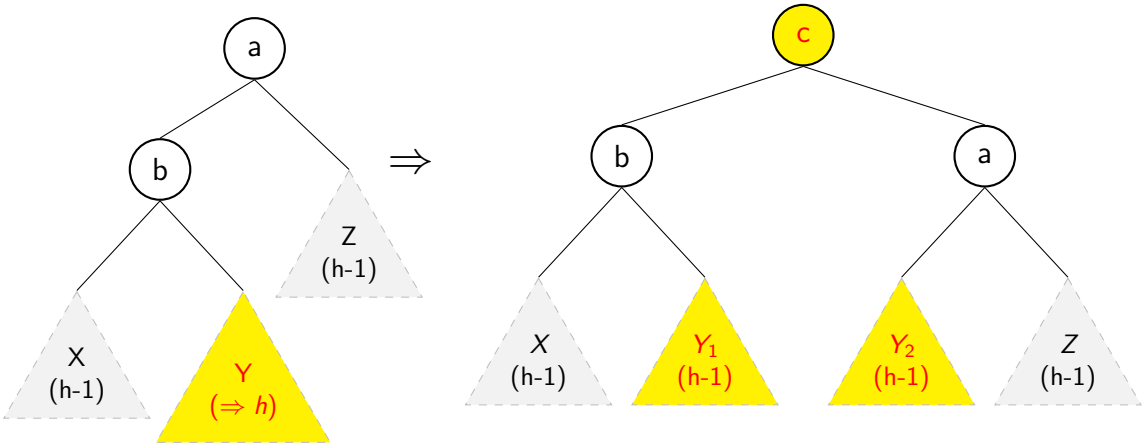
从平衡到失衡（左右情况）



BF(a)=1　（平衡）

BF(a)=2　（失衡）

通过 “↶ 左旋 b + 右旋 ↷ a” 恢复 LR 失衡



BF(a)=2 （失衡）

BF(c)=0, BF(a)=0/-1, BF(b)=0/1 （平衡）

Y_1, Y_2 其中之一的高度可能是 h-2。

若发生 LR 失衡，以 “↶ 左旋左孩 + 右旋 ↷ 自己” 应对

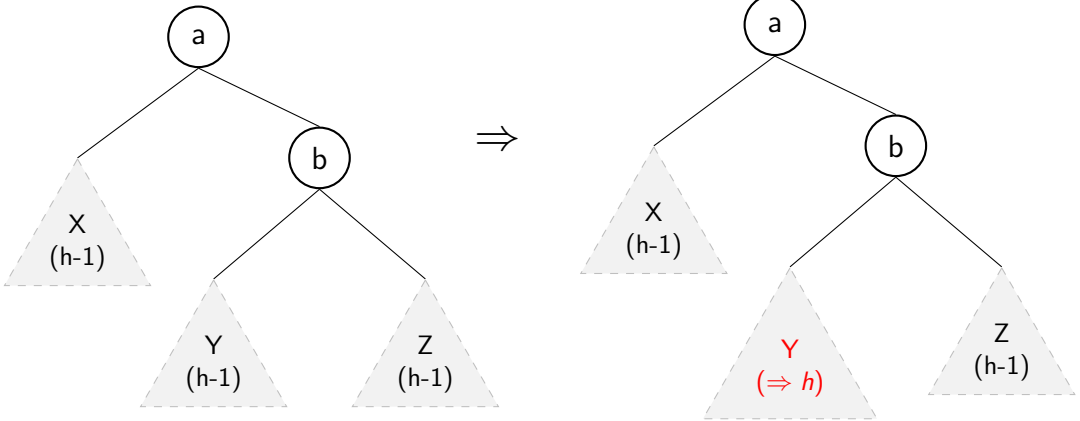
LR 失衡

触发条件：在节点 a 的左孩子 b 的右子树 Y 中插入了新节点，导致 BF(a)=2.

↶ 左旋左孩 + 右旋 ↷ 自己

- 修复操作应在不改变中序序列({X}b{Y1}c{Y2}a{Z}) 的前提下恢复平衡。
- 第一步：↶ 左旋左孩，效果是 b↓ c ↑.
  1. b 的右孩 := c 的左孩，即 Y1 的根.
  2. c 的左孩 := b.
- 第二步：右旋 ↷ 自己，效果是 a↓ c ↑.
  1. a 的左孩 := c 的右孩，即 Y2 的根.
  2. c 的右孩 := a.

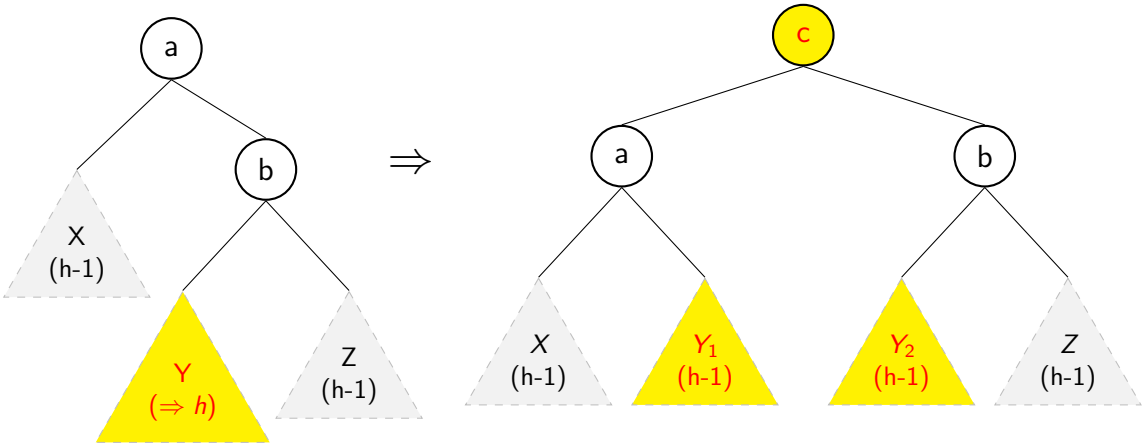
从平衡到失衡（右左情况）



BF(a)=-1 （平衡）

BF(a)=-2 （失衡）

通过“右旋 ↷b + ↶ 左旋 a”恢复 RL 失衡



BF(a)=-2 （失衡）

BF(c)=0, BF(a)=0/1, BF(b)=0/-1 （平衡）

Y_1, Y_2 其中之一的高度可能是 h-2。

若发生 RL 失衡，以“右旋 ↷ 右孩 + ↶ 左旋自己”应对

RL 失衡

触发条件：在节点 a 的右孩子 b 的左子树 Y 中插入了新节点，导致 BF(a)=-2.

右旋 ↷ 右孩 + ↶ 左旋自己

- 修复操作应在不改变中序序列({X}a{Y1}c{Y2}b{Z}) 的前提下恢复平衡。
- 第一步：右旋 ↷ 右孩，效果是 b↓ c ↑.
  1. b 的左孩 := c 的右孩，即 Y2 的根.
  2. c 的右孩 := b.
- 第二步：↶ 左旋自己，效果是 a↓ c ↑.
  1. a 的右孩 := c 的左孩，即 Y1 的根.
  2. c 的左孩 := a.

AVL 树插入算法总结

目标：保持平衡因子 $|\mathrm{BF}| \leq 1$

AVL 插入操作在遵循 BST 规则后，必须向上回溯并检查所有祖先节点的平衡性。

1. **标准插入**：将新节点作为叶节点插入到正确位置（与普通 BST 一样）。
2. **向上遍历**：从新插入的节点开始，沿父节点路径一直向上遍历到根节点。
3. **检查平衡**：在回溯路径上的每个节点 A 处，重新计算其平衡因子 $\mathrm{BF}(A)$。
4. **定位失衡**：找到第一个 $|\mathrm{BF}(A)| > 1$ 的祖先节点 A。
5. **应用旋转**：判断失衡类型并执行对应旋转：
   - **LL 失衡**：在 A 处执行 **单次右旋** ↷。
   - **RR 失衡**：在 A 处执行 **单次** ↶ **左旋** 。
   - **LR 失衡**：先 ↶ 左旋 (左孩节点)，后右旋 ↷ (关键节点)。
   - **RL 失衡**：先右旋 ↷ (右孩节点)，后 ↶ 左旋 (关键节点)。

AVL 树插入操作的复杂度分析

代码量有点大！失衡定位 + 分类讨论 + 旋转操作

AVL 树插入操作的复杂度分析

代码量有点大！失衡定位 + 分类讨论 + 旋转操作

☆ 插入元素、定位失衡的时间复杂度是 $O(\log n)$

AVL 树插入操作的复杂度分析

代码量有点大！失衡定位 + 分类讨论 + 旋转操作

☆ **插入元素、定位失衡的时间复杂度是** $O(\log n)$

最多只有一个失衡节点!

AVL 树插入操作的复杂度分析

代码量有点大！失衡定位 + 分类讨论 + 旋转操作

☆ **插入元素、定位失衡的时间复杂度是** $O(\log n)$

★ 最多只有一个失衡节点!

每种类型的失衡只需**一次单旋或双旋**即可恢复平衡。并且，

失衡节点的高度将被恢复到插入操作前，所以祖先节点的 BF 完全不受影响。

AVL 树插入操作的复杂度分析

代码量有点大！失衡定位 + 分类讨论 + 旋转操作

☆ **插入元素、定位失衡的时间复杂度是** $O(\log n)$

★ 最多只有一个失衡节点!

每种类型的失衡只需**一次单旋或双旋**即可恢复平衡。并且，

失衡节点的高度将被恢复到插入操作前，所以祖先节点的 BF 完全不受影响。

★ **因此，插入操作的旋转次数是** $O(1)$ **的**。

BST 的删除操作

删除 D 的三种情况（假设已找到）

1. **情况 1：被删除节点 D 有 0 个子节点 (叶子节点)**
   - 找到父节点，将其指向 D 的指针设为 NULL。
   - 直接删除 D 节点。
2. **情况 2：被删除节点 D 有 1 个子节点**
   - 将 D 的子节点直接连接到 D 的父节点上，取代 D 的位置。
   - 删除 D 节点。
3. **情况 3：被删除节点 D 有 2 个子节点 (最复杂)**
   - **目标**：找到一个合适的值 S 来替换 D 的值，使得 BST 性质被保持。
   - **寻找**：寻找 D 的中序后继（即 D 的右子树中的最小节点）。
   - **替换**：用 S 的值替换 D 的值，然后**转为删除 S 节点**（情况 1 或情况 2）。

☆ **虽有三种情况，但最后被删的（逻辑上）都是叶节点。**

AVL 树的删除算法总览

目标：保持平衡因子 $|BF| \leq 1$

AVL 删除操作在遵循 BST 规则后，必须向上回溯并检查所有祖先节点的平衡性。

1. **标准删除**：

AVL 树的删除算法总览

目标：保持平衡因子 $|BF| \leq 1$

AVL 删除操作在遵循 BST 规则后，必须向上回溯并检查所有祖先节点的平衡性。

1. **标准删除**： 按照 BST 删除规则移除节点（与普通 BST 一样）。
2. **向上遍历**：

AVL 树的删除算法总览

目标：保持平衡因子 $|BF| \leq 1$

AVL 删除操作在遵循 BST 规则后，必须向上回溯并检查所有祖先节点的平衡性。

1. **标准删除**： 按照 BST 删除规则移除节点（与普通 BST 一样）。
2. **向上遍历**： 从被删除节点的父节点开始，一路直向上遍历到根节点。
3. **检查平衡**：

AVL 树的删除算法总览

目标：保持平衡因子 $|\mathrm{BF}| \leq 1$

AVL 删除操作在遵循 BST 规则后，必须向上回溯并检查所有祖先节点的平衡性。

1. **标准删除**：按照 BST 删除规则移除节点（与普通 BST 一样）。
2. **向上遍历**：从被删除节点的父节点开始，一路直向上遍历到根节点。
3. **检查平衡**：在回溯路径上的每个节点 A 处，重新计算其平衡因子 $\mathrm{BF}(A)$。
4. **定位失衡**：

AVL 树的删除算法总览

目标：保持平衡因子 $|\mathrm{BF}| \leq 1$

AVL 删除操作在遵循 BST 规则后，必须向上回溯并检查所有祖先节点的平衡性。

1. **标准删除**：按照 BST 删除规则移除节点（与普通 BST 一样）。
2. **向上遍历**：从被删除节点的父节点开始，一路直向上遍历到根节点。
3. **检查平衡**：在回溯路径上的每个节点 A 处，重新计算其平衡因子 $\mathrm{BF}(A)$。
4. **定位失衡**：找到第一个 $|\mathrm{BF}(A)| > 1$ 的节点 A。
5. **应用旋转**：

AVL 树的删除算法总览

目标：保持平衡因子 $|\mathrm{BF}| \leq 1$

AVL 删除操作在遵循 BST 规则后，必须向上回溯并检查所有祖先节点的平衡性。

1. **标准删除**：按照 BST 删除规则移除节点（与普通 BST 一样）。
2. **向上遍历**：从被删除节点的父节点开始，一路直向上遍历到根节点。
3. **检查平衡**：在回溯路径上的每个节点 A 处，重新计算其平衡因子 $\mathrm{BF}(A)$。
4. **定位失衡**：找到第一个 $|\mathrm{BF}(A)| > 1$ 的节点 A。
5. **应用旋转**：判断失衡类型并执行对应旋转（与插入时相同的四种情况）。
6. **继续北上**：

AVL 树的删除算法总览

目标：保持平衡因子 $|\mathrm{BF}| \leq 1$

AVL 删除操作在遵循 BST 规则后，必须向上回溯并检查所有祖先节点的平衡性。

1. **标准删除**：按照 BST 删除规则移除节点（与普通 BST 一样）。
2. **向上遍历**：从被删除节点的父节点开始，一路直向上遍历到根节点。
3. **检查平衡**：在回溯路径上的每个节点 A 处，重新计算其平衡因子 $\mathrm{BF}(A)$。
4. **定位失衡**：找到第一个 $|\mathrm{BF}(A)| > 1$ 的节点 A。
5. **应用旋转**：判断失衡类型并执行对应旋转（与插入时相同的四种情况）。
6. **继续北上**：需继续向上检查并修复，直到根节点。(可能有多个失衡节点)

AVL 树四种失衡情况汇总

类型	描述	BF(A)	BF(B)	恢复操作
LL	左子树的左侧过深	$+2$	≥ 0	**单次右旋** ($\curvearrowright$)
RR	右子树的右侧过深	-2	≤ 0	**单次左旋** ($\curvearrowleft$)
LR	左子树的右侧过深	$+2$	-1	**双旋** (先左 $\curvearrowleft$ 后右 $\curvearrowright$)
RL	右子树的左侧过深	-2	$+1$	**双旋** (先右 $\curvearrowright$ 后左 $\curvearrowleft$)

注:

- 对于 LL/RR，插入时子节点 BF 通常为 ± 1，删除时可能为 0。
- 对于 LR/RL（之字形），必须执行双旋：先对子节点 B 旋转，再对 A 旋转。

删除与插入的区别

触发失衡的方式不同

- **插入时**：我们在较高的一侧插入，导致它太高。
- **删除时**：我们在**较矮的一侧**删除，导致它太矮，从而反衬出另一侧太高。

删除与插入的区别

触发失衡的方式不同

- **插入时**：我们在较高的一侧插入，导致它太高。
- **删除时**：我们在**较矮的一侧**删除，导致它太矮，从而反衬出另一侧太高。

与旋转侧的关系不同（旋转总在较高侧进行，以便降低高度）

- **插入侧**与**旋转侧**一致，而**删除侧**与**旋转侧**相反。

删除与插入的区别

触发失衡的方式不同

- **插入时**：我们在较高的一侧插入，导致它太高。
- **删除时**：我们在**较矮的一侧**删除，导致它太矮，从而反衬出另一侧太高。

与旋转侧的关系不同（旋转总在较高侧进行，以便降低高度）

- **插入侧**与**旋转侧**一致，而**删除侧**与**旋转侧**相反。

A 子树的高度变化不同

- **插入时**：A 子树的总高度满足 $h_{new} = h_{old}$，因此可以停止检查。
- **删除时**：A 子树的总高度满足 $h_{new} \leq h_{old}$，不等时需继续北上。

删除操作触发的四种失衡详表

前提：节点 **A** 因其某侧子树高度减小而失衡，**B** 为 A 的较高子节点。

删除发生位置	BF(A)	BF(B)	判定类型	所需旋转
在 **右子树删除** (导致左重)	$+2$	≥ 0 (左重/平)	**LL**	**右旋** (A)
		-1 (右重)	**LR**	**先左旋** (B) **后右旋** (A)
在 **左子树删除** (导致右重)	-2	≤ 0 (右重/平)	**RR**	**左旋** (A)
		$+1$ (左重)	**RL**	**先右旋** (B) **后左旋** (A)

重要提示：

- 插入时，没有 `BF(B)=0` 的失衡情况。
- 删除时，如果 `BF(B)=0`，我们将其视为同向失衡（LL 或 RR），执行单旋。

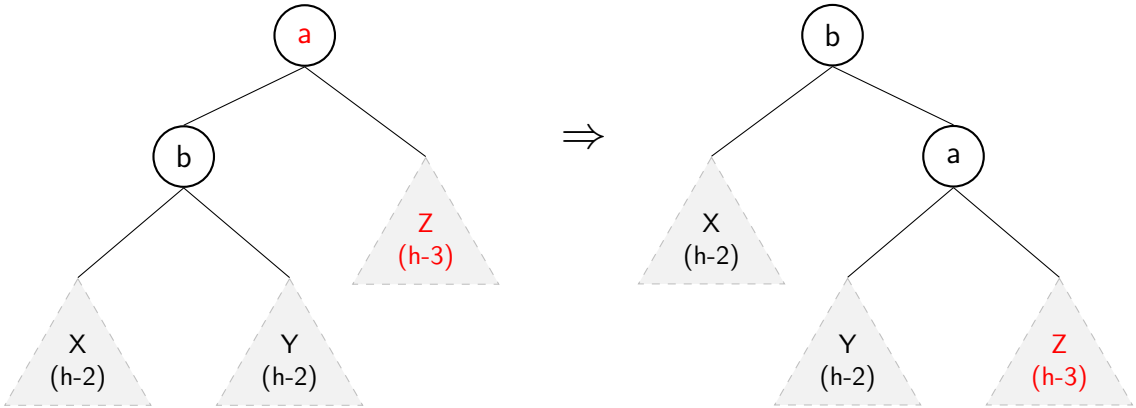
删除后 A 子树的高度变化，取决于 B 的平衡因子

失衡类型	BF(B)	子树高度变化	连锁反应
LR/RL	± 1 (交叉失衡)	$h \to h-1$	继续 (链式失衡)
LL/RR	± 1 (直线失衡)	$h \to h-1$	继续 (链式失衡)
LL/RR	0 (子节点平衡)	$h \to h$	停止 (平衡恢复)

注：

- **单次删除的最坏情况**：需要沿着路径从失衡点一直到根，**每层进行一次旋转**（$O(\log n)$）。

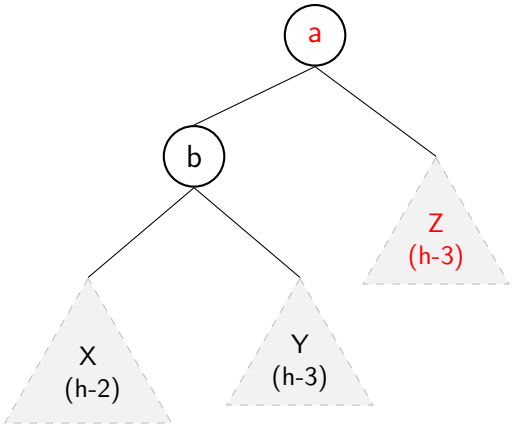
通过“右旋 ↷”恢复（插入不可能触发的）LL0 失衡



BF(a)=2 （失衡）

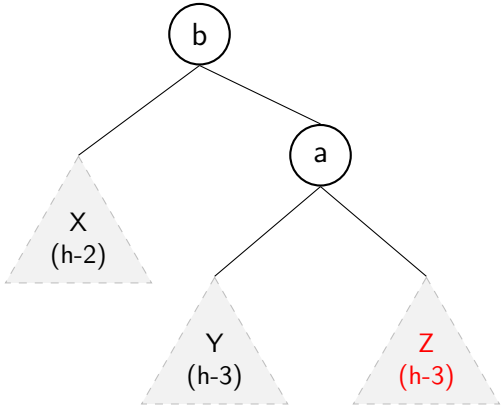
平衡恢复，且 $h_{new} = h_{old}$.

通过“右旋 ↷”恢复 LL1 失衡（直线失衡）



BF(a)=2 （失衡）

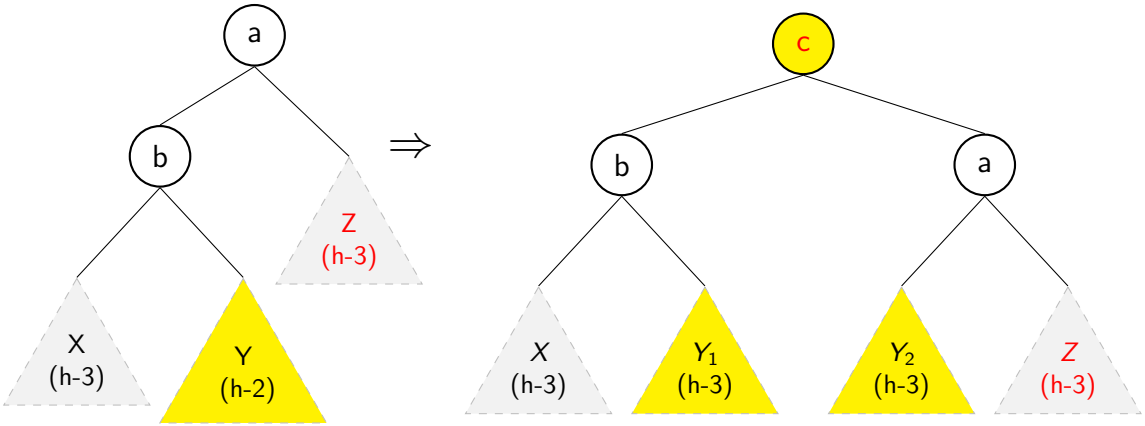
⇒



局部的平衡恢复，但是

$h_{new} = h_{old} - 1.$

通过“↶ 左旋 b + 右旋 ↷ a”恢复 LR 失衡（交叉失衡）



BF(a)=2 （失衡）

局部的平衡恢复，但是 $h_{new} = h_{old} - 1$.

AVL 总结：核心定义与性质

AVL 树的基本要求

1. **BST 性质**：必须首先满足二叉搜索树的所有性质。
2. **平衡因子约束**：任意节点的左子树高度与右子树高度差（平衡因子）必须满足：

$$\mathrm{BF}(\mathit{Node}) = \mathrm{Height}(\mathrm{Left}) - \mathrm{Height}(\mathrm{Right}) \in \{-1, 0, 1\}$$

树高与性能保障

- **性质**： AVL 树的最大高度始终为 $O(\log n)$。
- **证明**：用归纳法可以证明，树高的最小节点数由斐波那契数列决定。

AVL 总结：操作与平衡机制

表：AVL 树核心操作对比

操作	时间复杂度	平衡机制	关键区别
查找	$O(\log n)$	无	效率极高，性能稳定
插入	$O(\log n)$	向上查，**最多 1 次旋转**。	修复后子树高度恢复
删除	$O(\log n)$	向上查，**可能** $O(\log n)$ **次连旋**。	**可能变矮**，导致祖先失衡

AVL 总结：旋转类型与判断

表：删除操作后的修复与传播

类型	BF(B)	旋转动作	高度变化	继续检查？
LR/RL (双旋)	$\pm\mathbf{1}$	2 次旋转	-1	是
LL/RR (单旋)	$\neq \mathbf{0}$ (± 1)	1 次旋转	-1	是
LL/RR (单旋)	**0**	1 次旋转	不变	否

AVL 总结：性能与应用对比

AVL 树的优点

- **最严格的平衡**：树高控制最好，保证最佳的 $O(\log n)$ 查找性能。
- 适用于查询密集型场景（读多写少），如静态数据集和内存数据库索引。

表：与红黑树 (RBT) 的比较

特性	AVL 树	红黑树 (RBT)
查找性能	严格平衡，理论最优，常数小	弱平衡，$O(\log n)$
插入性能	**最多 1 次旋转**	最多 2 次旋转 + 颜色重涂
删除性能	**最多 $O(\log n)$ 次连旋**	最多 2 次旋转 + 颜色重涂
应用场景	查询密集型、删除少	动态数据、OS 内核、`std::map`

本节内容概览

突破 $O(\log n)$ 瓶颈：哈希查找

路径 1：基于比较的查找 (BST)

- **代表**：AVL 树、红黑树 (`std::set`)
- **原理**：沿着树边进行**比较判断**
- **限制**：理论最优复杂度为 $O(\log n)$
- **优点**：保持键的**有序性**

突破 $O(\log n)$ 瓶颈：哈希查找

路径 1：基于比较的查找 (BST)

- **代表**：AVL 树、红黑树 (`std::set`)
- **原理**：沿着树边进行**比较判断**
- **限制**：理论最优复杂度为 $O(\log n)$
- **优点**：保持键的**有序性**

路径 2：基于地址的查找 (Hashing)

- **代表**：哈希表 (`std::unordered_set`)
- **原理**：通过**哈希函数**得到存储位置
- **突破**：平均复杂度达到 $O(1)$
- **缺点**：键值存储**无序**、哈希冲突问题

核心转变：从比较到计算

- 基于比较的查找依赖于**逐层比较**来定位目标，受限于树的高度。
- 哈希查找则通过**计算哈希值**直接定位存储位置，跳过了查找路径。

哈希表的基本原理

什么是哈希 (Hashing)?

- 一种将任意大小的键 (**Key**) 映射到固定大小的数组索引 (**Index**) 的过程。
- 通过计算直接定位数据，实现 $O(1)$ 的平均查找、插入和删除时间复杂度。

哈希表的结构组成:

1. **哈希函数** $h(k)$**:** 将键 k 转换为索引 i。
2. **数组/桶 (Buckets):** 存储实际数据的底层结构。

核心问题：冲突 (Collision)

两个不同的键 $k_1 \neq k_2$ 经过哈希函数计算后，映射到同一个数组索引 $h(k_1) = h(k_2)$。

常用哈希函数的设计与选择

哈希函数的设计目标

- 快速计算：哈希函数应能在常数时间内计算完成。
- 均匀分布：不同的键应尽可能均匀地分布在哈希表的各个槽位上。
- 最小冲突：设计良好的哈希函数应尽量减少不同键映射到同一索引的概率。

1. 直接散列函数
2. 数字分析法
3. 平方取中法
4. 折叠法
5. 除留余数法
6. 随机数法

1. 直接散列函数

- **基本思想:** 直接取关键字或关键字的某个线性函数值作为散列地址。
- **公式:**

$$H(\text{key}) = \text{key} \quad \text{或} \quad H(\text{key}) = a \cdot \text{key} + b$$

- **应用场景:** 关键字范围小且分布连续。
- **特点:**
  - 优点: 实现简单，若一一对应则无冲突。
  - 缺点: 关键字范围大时，空间浪费严重。

2. 数字分析法

- **基本思想:** 分析关键字各位数字的分布，选取分布均匀的若干位作为散列地址。
- **工作原理:** 剔除所有关键字都相同的位（冗余位）和分布规律性差的位。
- **应用场景:** 关键字位数较长且事先知道（例如：学号、身份证号）。
- **特点:**
  - 优点: 针对特定集合效果极好，能有效减少冲突。
  - 缺点: 通用性差，需要事先进行详细的统计分析。

3. 平方取中法

- **基本思想:** 取关键字的平方值，然后从中间截取所需的位数作为散列地址。
- **步骤:**
  1. 计算 key 的平方值 $(\text{key})^2$。
  2. 从 $(\text{key})^2$ 中间抽取 k 位数字作为 $H(\text{key})$。
- **应用场景:** 关键字位数不太大且不规则的情况。
- **特点:**
  - 优点: 地址与关键字的每位数字都有关系，分布较均匀。
  - 缺点: 计算复杂度略高。

4. 折叠法

- **基本思想:** 将关键字分割成若干部分，然后将这些部分相加（或进行其他位运算）得到散列地址。
- **两种方式:**
  1. **位移折叠:** 各部分简单相加。
  2. **边界折叠:** 相邻两部分边界反转后相加。
- **应用场景:** 关键字位数很多（例如超过 8 位），且所有位都重要。
- **特点:**
  - 优点: 保留了关键字所有位的信息，适用于长关键字。

5. 除留余数法 (最常用)

- **基本思想:** 取关键字被某个数 P 除后的余数作为散列地址。
- **公式:**

$$H(\text{key}) = \text{key} \pmod{P}$$

  其中 $P \leq M$ (M 为散列表长度)。
- **应用场景:** 最常用、最简单且有效的构造方法。
- **关键选择:**
  - 必须选择一个好的 P 值。
  - P 应选择为一个不接近 2^k 或 10^k 的素数。
- **特点:**
  - 优点: 计算简单，效率高，应用广泛。
  - 缺点: 对 P 的选择高度敏感，选不好冲突率极高。

6. 随机数法

- **基本思想:** 通过一个确定的随机函数，将关键字转为一个随机数作为散列地址。
- **公式:**

$$H(\text{key}) = \text{random}(\text{key})$$

- **应用场景:** 关键字长度不等或数值分布规律性不强。
- **特点:**
  - 优点: 具有随机性，能避免特定分布导致的冲突。
  - 要求: 必须保证同一关键字始终产生相同随机数。

散列函数总结

- **核心原则:** 散列函数的优劣在于其散列地址的均匀分布。
- **常用方法:** 工业界最常用的是除留余数法（配合优秀的 P 值选择）和其他方法的组合变体。
- **后续思考:**
  - 如何评估一个散列函数的冲突率？
  - 不同的冲突处理方法（如链地址法、开放地址法）对散列函数的要求有何不同？

查找基本流程

核心目标

实现平均 $O(1)$ 的查找时间。

1. **计算哈希地址 (`Index = H(key)`):**
   - 使用哈希函数计算键值在散列表中的初始槽位 Index。
2. **判断冲突并查找:**
   - **无冲突:** 槽位为空或 Key 匹配，查找成功 ($O(1)$)。
   - **有冲突:** 槽位空占用 Key 失配，必须根据建表时的**冲突解决策略**进行后续查找。

常用冲突处理方法

1. 开放地址法
2. 再散列法
3. 链地址法
4. 公共溢出区法

1. 开放地址法（Open Addressing）

- **基本思想:** 当发生冲突时，按照一定的探查序列寻找下一个空槽位进行存储。
- **探查方式:**
  1. 线性探查（Linear Probing）
  2. 二次探查（Quadratic Probing）
  3. 双重散列（Double Hashing）
- **优缺点:**
  - 优点: 实现简单，无需额外存储空间。
  - 缺点: 容易产生聚集现象，影响性能。

a. 线性探查（Linear Probing）

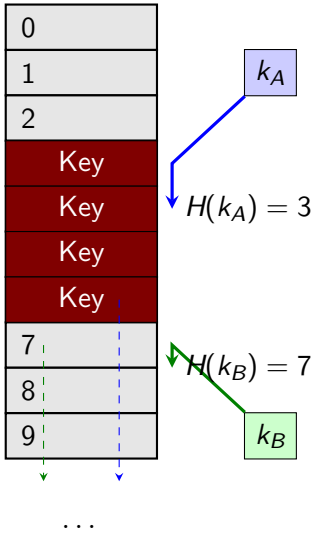
- **基本思想:** 当发生冲突时，依次检查下一个槽位，直到找到空槽位为止。
- **探查公式:**

$$H_i(\text{key}) = (H(\text{key}) + i) \mod M$$

其中 $i = 0, 1, 2, \ldots$

- **优缺点:**
  - 优点: 实现简单，探查过程线性。
  - 缺点: 容易产生**主聚集**现象，影响查找效率。

线性探测（及其主聚集现象）示意图



→ 任何哈希到连续块中或其附近的键，都将沿着此线性路径探测，从而使该块变长。

b. 二次探查（Quadratic Probing）

- **基本思想:** 当发生冲突时，按照二次方的间隔进行探查，减少聚集现象。
- **探查公式:**

$$H_i(\text{key}) = (H(\text{key}) + i^2) \mod M$$

其中 $i = 0, 1, 2, \ldots$

- **优缺点:**
  - 优点: 减少主聚集现象，提高查找效率。
  - 缺点: 可能无法探查到所有槽位，需谨慎选择表大小 M。

c. 双重散列（Double Hashing）

- **基本思想:** 使用两个不同的哈希函数，第一次哈希用于初始地址，第二次哈希用于计算探查间隔。
- **探查公式:**

$$H_i(\text{key}) = (H_1(\text{key}) + i \cdot H_2(\text{key})) \mod M$$

其中 $i = 0, 1, 2, \ldots$

- **优缺点:**
  - 优点: 大大减少聚集现象，提高查找效率。
  - 缺点: 需要设计两个良好的哈希函数，增加实现复杂度。

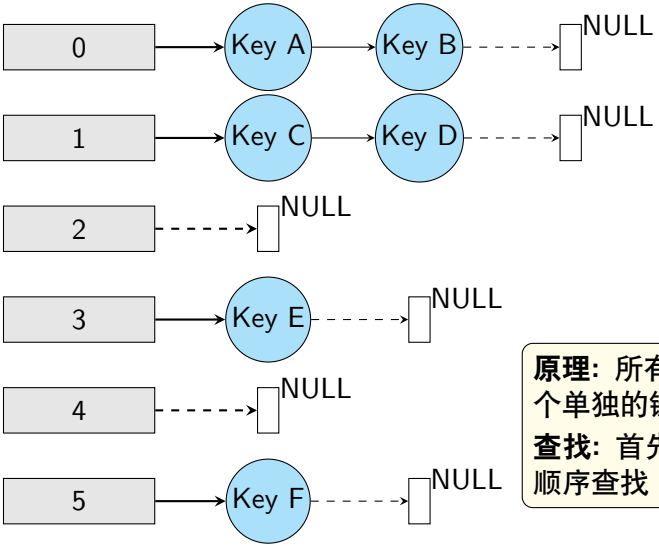
2. 再散列法（Rehashing）

- **基本思想:** 当哈希表达到一定负载因子时，创建一个更大的哈希表，并将所有元素重新哈希到新表中。
- **步骤:**
  1. 创建一个新的、更大的哈希表。
  2. 使用原哈希函数或新的哈希函数，将旧表中的所有元素重新插入新表。
- **优缺点:**
  - 优点: 有效降低负载因子，减少冲突，提高性能。
  - 缺点: 重新哈希过程开销较大，可能导致性能波动。

3. 链地址法（Separate Chaining）

- **基本思想:** 每个槽位存储一个链表（或其他数据结构），所有映射到同一槽位的元素都存储在该链表中。
- **操作流程:**
  1. 计算哈希值，定位槽位。
  2. 在对应槽位的链表中查找、插入或删除元素。
- **优缺点:**
  - 优点: 简单易实现，冲突处理灵活，适应性强。
  - 缺点: 链表过长时，查找效率降低至 $O(n)$；缓存不友好（局部性低。

链地址法 (Separate Chaining) 示意图



原理: 所有哈希到同一地址的元素，都存储在一个单独的链表中。

查找: 首先 $O(1)$ 访问槽位，然后沿链表 $O(L)$ 顺序查找（L 为链表长度）。

4. 公共溢出区法（Common Overflow Area）

- **基本思想:** 设立一个公共的溢出区，当发生冲突时，将冲突的元素存储在溢出区中。
- **操作流程:**
  1. 计算哈希值，定位槽位。
  2. 如果槽位已被占用，将元素插入溢出区。
  3. 查找时，先检查槽位，再检查溢出区。
- **优缺点:**
  - 优点: 简化冲突处理，易于实现。
  - 缺点: 溢出区可能成为性能瓶颈，查找效率降低。

std::unordered_set 背后的哈希实现

- **哈希函数:** 默认使用标准库提供的哈希函数，支持自定义哈希函数。
- **冲突处理:** 采用链地址法，每个槽位存储一个链表或其他容器。
- **负载因子管理:** 当负载因子超过阈值时，自动进行再散列，扩展哈希表大小。
- **性能特点:**
  - 平均情况下，插入、删除和查找操作均为 $O(1)$,
  - 但在最坏情况下可能退化为 $O(n)$.

负载因子 (α)

负载因子 (α)

定义： $\alpha = \frac{n}{m}$　(n: 元素数, m: 槽位数)

- 衡量散列表拥挤程度的核心指标。
- α 越大，冲突越多，效率越低。
- 链地址法中 α 可 > 1。
- 理想值通常 < 0.75。

散列表的性能依赖于 α

负载因子直接影响查找、插入和删除操作的平均时间复杂度。

链地址法性能分析

平均查找所需探测次数 C_{avg}

- **失败查找 (C_{miss}):** $1 + \alpha$
- **成功查找 (C_{hit}):** $1 + \frac{\alpha}{2}$

结论: 性能与 α 呈线性关系，增长缓慢且稳定。

证明.

假设平均链表长度为 α，则:

- **失败查找:** 需检查平均 α 个元素 $+ 1$ 次哈希计算。
- **成功查找:** 平均需检查 $\frac{\alpha}{2}$ 个元素 $+ 1$ 次哈希计算。

□

开放地址法与聚集

开放地址法的性能对 α 极其敏感

线性探测 (聚集严重)

- **失败:** $C_{\text{miss}} \approx \frac{1}{(1-\alpha)^2}$
- **成功:** $C_{\text{hit}} \approx \frac{1}{1-\alpha}$

双重散列 (接近理论最优)

- **失败:** $C_{\text{miss}} \approx \frac{1}{1-\alpha}$
- **成功:** $C_{\text{hit}} \approx \frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$

性能敏感度

线性探测: 当 $\alpha \to 1$ 时，$C_{\text{avg}} \to \infty$，性能迅速恶化。

消除聚集

双重散列能有效消除聚集，因此性能远优于线性探测。

部分语言散列表实现对比

- 链地址法阵营
  - **C++ std::unordered_map**：经典链地址法，实现简单，删除操作稳定。
  - **Java HashMap**：结合链地址和红黑树，长链优化为 $O(\log n)$ 查找。
  - **Go map**：带小型 Bucket 结构，结合两种优势，保证局部缓存命中率。
- 开放地址法阵营
  - **Python dict/set**：伪随机探测，内存紧凑，缓存友好，性能极高。
  - **Rust HashMap**：Robin Hood 变体，高性能，缓存局部性强。
  - **JavaScript Map**：V8 引擎通常为混合/开放地址，极致优化存取速度。

本节内容概览

总结

静态查找方法

- 顺序查找
- 二分查找
- 索引查找

二叉查找树

- 查找
- 插入
- 删除

AVL 树

- 平衡因子
- 旋转操作
- 插入
- 删除

散列查找方法

- 散列函数
- 冲突处理
- 性能分析

欢迎提问！

你的疑惑，我的动力