数据结构与算法期末复习
Homework

尹超

中国科学院大学，北京 100049

Carter Yin

University of Chinese Academy of Sciences, Beijing 100049, China

2024.8 – 2025.1

序言

本文为笔者数据结构与算法的期末复习笔记。
望老师批评指正。

目录

题目 第七章高级搜索树

114

> 伸展树利用了实际问题中数据访问的何种特点：A. 大量性 B. 局部性 C. 整体性 D. 多样性

Solution 1. 正确答案是 *B*。

详细分析：

伸展树（*Splay Tree*）是一种自适应的二叉搜索树，它的核心思想是：每当一个节点被访问（查找、插入、删除）后，通过一系列的旋转操作（称为“伸展”或“*splaying*”），将该节点移动到树的根部。

这种设计的理论基础是 ** 访问的局部性原理（*Principle of Locality*）**，该原理在计算机科学中非常普遍，它包含两个方面：

(1) **时间局部性（*Temporal Locality*）：**如果一个数据项被访问，那么它在不久的将来很可能再次被访问。

(2) **空间局部性（*Spatial Locality*）：**如果一个数据项被访问，那么与它相邻的数据项也很可能在不久的将来被访问。

伸展树正是利用了时间局部性。通过将被访问的节点移动到根部，下一次对同一个节点的访问将变得非常快（*O(1)* 时间）。同时，这也使得与该节点在数值上相近的节点（即它在树中的邻近节点）也被带到了离根更近的位置，从而在一定程度上也利用了空间局部性。

因此，伸展树的性能优势在数据访问呈现出明显的局部性时最为显著，而在均匀随机访问的情况下，其性能与普通平衡树相当。

115

> 伸展树每次访问过某节点后都会把该节点：A. 删除 B. 上移一层 C. 移动到根节点 D. 再次访问该节点

Solution 2. 正确答案是 *C*。

详细分析：

伸展树（*Splay Tree*）是一种自适应的二叉搜索树。它的核心和定义性特征是 ** 伸展（*Splaying*）** 操作。

这个操作规定，每当树中的一个节点被访问（无论是通过查找、插入还是删除操作），该节点都会通过一系列特定的旋转（*zig, zig-zig, zig-zag*）被移动到树的根部。

- ***A.* 删除：**访问不等于删除。删除是伸展树的一种操作，但访问本身的目的不是删除。
- ***B.* 上移一层：**这不准确。节点不是仅仅上移一层，而是通过一系列旋转操作，最终被移动到树的根节点位置，这通常需要移动多层。
- ***C.* 移动到根节点：**这是正确的。将最近访问的节点移动到根节点是伸展树的根本操作，也是其名称“伸展”的由来。
- ***D.* 再次访问该节点：**移动节点是访问操作之后的一个结构调整步骤，而不是再次进行访问。

因此，伸展树每次访问过某节点后，都会通过伸展操作将该节点移动到根节点。

116

> Tarjan 提出的伸展算法每几层一起伸展？A. 1 B. 2 C. 3 D. 4

Solution 3. 正确答案是 *B*。

详细分析：

伸展树（*Splay Tree*）的伸展（*splaying*）操作是通过一系列旋转将目标节点移动到根部。这个过程是迭代进行的，每次迭代都将目标节点向上移动。

伸展操作的核心步骤分为三种情况，这取决于目标节点 *'x'*、其父节点 *'p'* 和其祖父节点 *'g'* 的相对位置：

(1) Zig (单旋): 当 *'p'* 是树的根节点时，执行一次单旋转，将 *'x'* 提升为新的根。这个是伸展过程的最后一步。

(2) Zig-Zig (双单旋): 当 *'p'* 不是根，且 *'x'* 和 *'p'* 都是左孩子（或都是右孩子）时，先对 *'g'* 进行旋转，再对 *'p'* 进行旋转。这个操作涉及 *'x', 'p', 'g'* 三个节点。

(3) Zig-Zag (双旋): 当 *'p'* 不是根，且 *'x'* 和 *'p'* 一个是左孩子、另一个是右孩子时，对 *'x'* 进行两次旋转。这个操作也涉及 *'x', 'p', 'g'* 三个节点。

在主要的迭代过程中（即当目标节点 *'x'* 的祖父节点 *'g'* 存在时），伸展算法总是考虑 *'x', 'p', 'g'* 这三代节点，即 ** 一次处理两层 ** 父子关系（*'g'* 到 *'p'* 是一层，*'p'* 到 *'x'* 是另一层）。通过 *Zig-Zig* 或 *Zig-Zag* 操作，*'x'* 会被提升两层（或更多）。

因此，伸展算法是每 *2* 层一起进行伸展的。

117

> 在一棵退化成单链的伸展树中访问其最深的节点，经过伸展后树高大约为原先的：A. 三分之一 B. 一半 C. 不变 D. 两倍

Solution 4. 正确答案是 *B*。

详细分析：

***(1)* 初始状态：** 一棵退化成单链的伸展树，其结构就是一个链表。如果它有 *'n'* 个节点，其高度为 $h_{old} = n - 1$。

***(2)* 操作：** 访问（并伸展）其最深的节点。这个节点就是链表的末端节点。

***(3)* 伸展过程：** 当对链表末端的节点进行伸展操作时，由于该节点、其父节点、其祖父节点等都排列在一条直线上，伸展过程会连续执行一系列的 *Zig-Zig* 操作。

***(4)* 结果状态：** 经过一系列的 *Zig-Zig* 操作后，原来的链表会被重构。新的树会以被访问的节点为根，而原来的其他节点会大致均匀地分布在根节点的左、右两侧（如果原链表是单向的，则分布在一侧），形成一个相对平衡得多的结构。

具体来说， 原来链上的节点会交替地成为新树中左、右分支上的节点。这使得新树的高度大大降低。

***(5)* 高度变化：** 可以证明，当对一个有 *'n'* 个节点的链表的末端节点进行伸展后，新树的高度大约为 $h_{new} \approx \frac{n}{2}$。

***(6)* 比例计算：** 新旧高度的比值为：$\frac{h_{new}}{h_{old}} \approx \frac{n/2}{n-1}$ 当 *'n'* 足够大时，这个比值趋近于 $\frac{1}{2}$。

因此，经过伸展后，树的高度大约变为原来的一半。

118

> 按照 Tarjan 的伸展算法，单次伸展操作的分摊复杂度为：
> A. O(1)
> B. O(logn)
> C. O(n)
> D. O(nlogn)

Solution 5. 正确答案是 B。

详细分析：

Tarjan 证明了伸展树（*Splay Tree*）在进行一系列操作时，**单次伸展操作的分摊（均摊）时间复杂度为** $O(\log n)$。也就是说，虽然某一次操作的最坏复杂度可能达到 $O(n)$，但在一系列 m 次操作中，平均每次操作的实际代价不会超过 $O(\log n)$。

这是伸展树的一个重要性质，使其在实际应用中具有很好的性能保证。

119

> 双层伸展策略优于逐层伸展策略的关键在于：
> A. zig-zig/zag-zag
> B. zig-zag/zag-zig
> C. zig-zig/zig-zag
> D. 以上均是

Solution 6. 正确答案是 A。

详细分析：

双层伸展策略（即 *Splay Tree* 的伸展操作）与逐层伸展（每次只将目标节点上移一层）相比，效率更高的关键在于“*zig-zig*”和“*zag-zag*”这两种情形。

- ***zig-zig/zag-zag*：** 当目标节点和其父节点同为左（或同为右）孩子时，连续两次旋转可以让目标节点一次上升两层。这种批量提升大大加快了节点到根的速度，是双层伸展策略的核心优势。
- ***zig-zag/zag-zig*：** 这种情况虽然也能提升节点，但提升效率不如 *zig-zig/zag-zag*。

因此，双层伸展策略优于逐层伸展策略的关键在于 *zig-zig/zag-zag* 操作。

120

> 伸展树采用双层伸展策略，即可避免最坏情况的发生 ()

Solution 7. 这个陈述是 **错误的**。

详细分析：

双层伸展策略并不能完全避免最坏情况的发生。具体来说：

***(1)* 单次操作的最坏情况：**即使采用双层伸展策略，单次伸展操作在最坏情况下仍然可能需要 $O(n)$ 的时间。例如，当要访问的节点位于一条退化成链表的伸展树的末端时，将该节点伸展到根仍然需要经过所有其他节点，时间复杂度为 $O(n)$。

***(2)* 双层伸展的优势：**双层伸展策略的真正优势在于：

- 它保证了 **分摊（均摊）时间复杂度** 为 $O(\log n)$。
- 它避免了某些连续最坏情况的累积效应。
- 在一系列操作中，虽然个别操作可能很慢，但总体性能是有保证的。

***(3)* 与逐层伸展的对比：**如果采用逐层伸展（每次只上移一层），那么对链状树的连续访问可能导致持续的 $O(n)$ 复杂度，而双层伸展能够”打破”这种持续的最坏情况。

结论：双层伸展策略的核心作用是保证分摊复杂度为 $O(\log n)$，而不是完全避免最坏情况的发生。单次操作的最坏情况仍然存在，但不会持续累积。

121

> 所访问的节点及其父亲都是右孩子，则双层伸展要执行的操作是：
> A. zig-zag
> B. zag-zig
> C. zig-zig
> D. zag-zag

Solution 8. 正确答案是 D。

详细分析：

在伸展树的双层伸展策略中，操作类型取决于目标节点和其父节点相对于各自父节点的位置关系。

- ***zig*：**表示节点是其父节点的左孩子
- ***zag*：**表示节点是其父节点的右孩子

设目标节点为 x，其父节点为 p，其祖父节点为 g。

根据题目描述：

- x 是 p 的右孩子 → 这是一个*”zag”*
- p 是 g 的右孩子 → 这也是一个*”zag”*

当目标节点和其父节点都位于相同的方向（都是右孩子或都是左孩子）时，执行的是同向的双旋转操作。

在这种情况下，由于两个关系都是”右孩子”（*zag*），所以执行的操作是 ***zag-zag***。

这个操作包括：

(1) 首先对祖父节点 g 执行左旋转（*zag*），使 p 上升

(2) 然后对 p 执行左旋转（*zag*），使 x 上升到 g 的原位置

122

> 规模为 n 的伸展树中若所访问的节点只有 k 个，经过足够长时间的访问序列后，访问的分摊复杂度为：
> A. O(lg k)
> B. O(k lg n)

> C. O(n lg k)
> D. O(lg n)

Solution 9. 正确答案是 A。

详细分析：

这是伸展树的一个重要性质，体现了它的自适应特性。

***(1)* 基本情况：** 在一个有 n 个节点的伸展树中，如果我们只访问其中的 k 个节点（$k \leq n$），那么经过足够长时间的访问后，这 k 个经常被访问的节点会因为伸展操作逐渐聚集到树的上层，形成一个相对平衡的子树。

***(2)* 自适应重构：** 由于伸展树的自适应性质，经常被访问的节点会通过伸展操作不断向根部移动。经过足够长的时间后，这 k 个节点会在树的上层形成一个高度大约为 $O(\log k)$ 的子结构。

***(3)* 分摊复杂度分析：** 当访问模式稳定后（即主要访问这 k 个节点），每次访问的分摊复杂度主要取决于这 k 个活跃节点形成的子树的高度，而不是整个树的规模 n。

***(4)* 理论依据：** 可以证明，在伸展树中，如果访问集中在 k 个节点上，那么经过足够多次访问后，这些节点的访问成本的分摊复杂度为 $O(\log k)$。这是伸展树局部性优化的一个重要体现。

因此，尽管整个树有 n 个节点，但由于访问的局部性，分摊复杂度降低到了 $O(\log k)$，其中 k 是实际被访问的节点数量。

123

> 在任意一颗伸展树中，按节点值的大小以升序依次访问完所有节点，最后树变为一条单链，且仅含左孩子。

Solution 10. 这个陈述是 **正确的**。

详细分析：

***(1)* 升序访问过程：** 当我们按节点值的升序依次访问伸展树中的所有节点时，访问顺序就是该树的中序遍历序列。设节点值为 $v_1 < v_2 < v_3 < \ldots < v_n$。

***(2)* 每次访问的伸展效果：**

- 访问 v_1（最小节点）：v_1 被伸展到根。由于 v_1 是最小的，它没有左子树，所有其他节点都在它的右子树中。
- 访问 v_2（第二小节点）：v_2 被伸展到根。此时 v_1 成为 v_2 的左孩子，而 $v_3, v_4, \ldots, v_n$ 都在 v_2 的右子树中。
- 访问 v_3：v_3 被伸展到根。此时 v_1, v_2 都在 v_3 的左子树中，且形成一个左链。
- 依此类推...

***(3)* 最终结构：** 当访问到最后一个节点 v_n（最大节点）时：

- v_n 被伸展到根位置
- 所有之前访问过的节点 $v_1, v_2, \ldots, v_{n-1}$ 都位于 v_n 的左子树中
- 由于访问的升序特性，这些节点会排列成一条从 v_n 开始向左延伸的链
- v_n 没有右孩子（因为它是最大的）

***(4)* 为什么是左链：** 在最终状态下，根节点是 v_n，它的左子树包含所有其他节点。由于伸展操作的特性和访问顺序，这些节点会排列成一条链，其中每个节点只有左孩子，没有右孩子。

结论：该陈述是正确的。按升序访问伸展树的所有节点后，树确实会退化成一条单链，且该链仅由左孩子连接构成。

124

> B-trees are 树结构上的特点是：
> A. 矮胖，每个节点至多两个孩子
> B. 矮胖，每个节点可有多于两个孩子
> C. 瘦高，每个节点至多两个孩子
> D. 瘦高，每个节点可有多于两个孩子

Solution 11. 正确答案是 *B*。

详细分析：

B 树（*B-tree*）是一种自平衡的多路搜索树，专门为外存储器（如磁盘）设计。其主要特点包括：

***(1)* 多路分支：** *B* 树的每个节点可以有多个孩子，而不是像二叉树那样最多只有两个孩子。一个 *m* 阶的 *B* 树，每个节点最多可以有 *m* 个孩子。

***(2)* 矮胖的结构：**

- **矮：** *B* 树通过增加每个节点的分支数来减少树的高度。相比于二叉树，*B* 树在存储相同数量的数据时，高度显著降低。
- **胖：** 每个节点包含多个关键字和多个孩子指针，节点”更宽”。

***(3)* 设计目的：** *B* 树的矮胖设计是为了适应外存储器的特性：

- 减少磁盘 *I/O* 次数（因为树高较低）
- 每次磁盘访问读取更多数据（一个节点包含多个关键字）
- 提高外存储器上搜索、插入、删除操作的效率

***(4)* 与其他选项的对比：**

- *A* 错误：*B* 树不是每个节点至多两个孩子
- *C* 错误：*B* 树不是瘦高的结构
- *D* 错误：*B* 树不是瘦高的，虽然每个节点确实可以有多于两个孩子

因此，*B* 树的结构特点是”矮胖，每个节点可有多于两个孩子”。

125

> B 树的层数少有助于：
> A. 减少 I/O 次数
> B. 减少每次 I/O 的时间
> C. 降低查找的渐进时间复杂度
> D. 节省存储空间

Solution 12. 正确答案是 *A*。

详细分析：

B 树设计的核心目标是优化外存储器（如磁盘）上的数据访问性能。层数少的直接好处是减少了访问所需的 *I/O* 操作次数：

(1) A. 减少 I/O 次数（正确）:

- 在 *B* 树中进行查找、插入或删除操作时，需要从根节点开始，逐层向下访问节点。
- 每次访问一个节点都需要进行一次磁盘 *I/O* 操作。
- 层数少意味着从根到叶子的路径短，因此需要的磁盘 *I/O* 次数少。
- 这是 *B* 树相对于二叉搜索树的主要优势，也是其设计的核心目的。

(2) B. 减少每次 I/O 的时间（错误）: 每次 *I/O* 的时间主要取决于磁盘的物理特性（寻道时间、旋转延迟等）和传输的数据量，而不是树的层数。

(3) C. 降低查找的渐进时间复杂度（错误）:

- *B* 树的查找时间复杂度仍然是 $O(\log n)$，与平衡二叉搜索树相同。
- 虽然常数因子可能更好（由于 *I/O* 次数减少），但渐进复杂度没有改变。

(4) D. 节省存储空间（错误）:

- *B* 树的节点通常比二叉树节点包含更多的关键字和指针。
- 虽然可能减少了一些指针的总数，但不是其主要设计目标。
- 层数少并不直接导致存储空间的节省。

结论： *B* 树层数少的最重要作用是减少磁盘 *I/O* 操作的次数，这是 *B* 树在外存储器环境中性能优越的根本原因。

126

> 4 阶 B 树中每个节点的分支数为:
> A. 1 4
> B. 2 5
> C. 2 4
> D. 3 5

Solution 13. 正确答案是 C。

详细分析:

对于 *m* 阶 *B* 树，其定义包含以下约束条件:

(1) 关键字数量:

- 每个节点最多包含 $m-1$ 个关键字
- 除根节点外，每个节点至少包含 $\lceil \frac{m}{2} \rceil - 1$ 个关键字

(2) 分支数（孩子数）:

- 每个节点最多有 m 个孩子
- 除根节点外，每个内部节点至少有 $\lceil \frac{m}{2} \rceil$ 个孩子
- 叶子节点没有孩子（分支数为 *0*）

(3) 根节点的特殊情况:

- 如果根节点不是叶子节点，则它至少有 *2* 个孩子
- 如果根节点是叶子节点，则它没有孩子

对于 *4* 阶 *B* 树（$m = 4$）:

- $\lceil \frac{4}{2} \rceil = 2$
- 每个节点最多有 *4* 个孩子
- 除根节点外，每个内部节点至少有 *2* 个孩子

- 根节点如果不是叶子，至少有 *2* 个孩子
- 叶子节点没有孩子

因此，*4* 阶 *B* 树中：

- 叶子节点：*0* 个分支
- 内部节点（包括根）：*2 4* 个分支

但是，题目问的是”每个节点的分支数”，通常在讨论 *B* 树分支数时，我们主要关注内部节点（因为叶子节点显然没有分支）。

所以答案是 ***2 4***。

127

> 在存储了 n 个元素的 4 阶 B 树中查找，单个节点进行一次查找的时间复杂度为：
> A. O(1)
> B. O(lg n)
> C. O(n)
> D. O(n lg n)

Solution 14. 正确答案是 *A*。

详细分析：

题目问的是”单个节点进行一次查找”的时间复杂度，这里需要理解清楚什么是”单个节点的查找”。

(1) 4 阶 B 树的节点特性：

- 每个节点最多包含 $m-1=3$ 个关键字
- 每个节点最少包含 $\lceil \frac{m}{2} \rceil - 1 = 1$ 个关键字（除根节点外）

(2) 单个节点内的查找： 在一个 *B* 树节点内部查找特定关键字时：

- 节点内的关键字是有序排列的
- 可以使用线性搜索或二分搜索
- 由于 *4* 阶 *B* 树的每个节点最多只有 *3* 个关键字，这是一个常数
- 无论使用哪种搜索方法，在常数大小的有序数组中查找的时间复杂度都是 $O(1)$

(3) 关键理解：

- 题目强调的是”单个节点”的查找，不是整个 *B* 树的查找
- *4* 阶 *B* 树中每个节点包含的关键字数量是有界常数（最多 *3* 个）
- 在固定大小的数据结构中进行查找，时间复杂度是常数时间

(4) 与其他选项的区别：

- *B. O(lg n)*：这是整个 *B* 树查找的复杂度，不是单个节点内的查找
- *C. O(n)*：这远超过了单个节点的大小
- *D. O(n lg n)*：这通常是排序算法的复杂度，与此无关

结论： 在 *4* 阶 *B* 树的单个节点内进行查找的时间复杂度是 $O(1)$，因为每个节点包含的关键字数量是有界常数。

128

> B 树查找算法若最终失败，返回值为：
> A. None
> B. NULL
> C. 指向最后一个所查找节点的指针
> D. 指向根节点的指针

Solution 15. 正确答案是 *B*。

详细分析：

在 *B* 树的查找算法中，当查找失败时，标准的返回值是 *NULL*（或其等价形式）。

***(1)* 查找失败的情况：**

- 查找过程从根节点开始，逐层向下搜索
- 在每个节点内部搜索目标关键字
- 如果没有找到，根据关键字大小关系选择合适的子树继续搜索
- 最终到达叶子节点仍未找到目标关键字时，查找失败

***(2)* 标准返回值：**

- **查找成功：**返回指向包含目标关键字的节点的指针
- **查找失败：**返回 *NULL* 指针，表示未找到目标关键字

***(3)* 为什么是 *NULL*：**

- *NULL* 是编程中表示”无效指针”或”空值”的标准方式
- 它明确表示查找操作没有找到有效的结果
- 这是大多数搜索算法的通用约定
- 便于调用者判断查找是否成功

***(4)* 其他选项分析：**

- *A. None*：这是 *Python* 等语言中的空值表示，但在讨论数据结构时，通常使用 *NULL*
- *C.* 指向最后一个所查找节点的指针：这不是标准做法，且会造成混淆
- *D.* 指向根节点的指针：这没有意义，不能表示查找失败

结论： *B* 树查找算法失败时的标准返回值是 *NULL*，这是数据结构中表示查找失败的通用约定。

129

> 若 B 树的阶 m=128，则它的高度大致是对应的 BBST 的：
> A. 1/5
> B. 1/6
> C. 1/7
> D. 1/8

Solution 16. 正确答案是 *B*。

详细分析：

***(1)* *BBST* 的高度：**一个平衡二叉搜索树（*Balanced Binary Search Tree, BBST*）的高度 h_{BBST} 约等于 $\log_2(n)$，其中 n 是节点总数。对数的底是 *2*，因为每个节点最多有两个分支。

(2) B 树的高度： 一个 m 阶 B 树的高度 h_B 约等于 $\log_d(n)$，其中 d 是每个节点的平均分支数。对于 m 阶 B 树，除根节点外，每个内部节点的分支数至少为 $\lceil \frac{m}{2} \rceil$。为了估算，我们可以取 $d \approx m/2$。

(3) 计算比例： 我们要求解的比例是 $\frac{h_B}{h_{BBST}}$。$\frac{h_B}{h_{BBST}} \approx \frac{\log_d(n)}{\log_2(n)}$

(4) 使用对数换底公式： 根据换底公式 $\log_a(b) = \frac{\log_c(b)}{\log_c(a)}$，我们可以将 $\log_d(n)$ 转换为以 *2* 为底的对数：$\log_d(n) = \frac{\log_2(n)}{\log_2(d)}$

代入比例公式中：$\frac{h_B}{h_{BBST}} \approx \frac{\frac{\log_2(n)}{\log_2(d)}}{\log_2(n)} = \frac{1}{\log_2(d)}$

(5) 代入数值： 题目中 $m = 128$，所以我们取 $d \approx m/2 = 128/2 = 64$。比例 $\approx \frac{1}{\log_2(64)}$

计算 $\log_2(64)$：因为 $2^6 = 64$，所以 $\log_2(64) = 6$。

(6) 最终结果： 比例 $\approx \frac{1}{6}$。

因此，一个 *128* 阶 *B* 树的高度大约是对应的平衡二叉搜索树高度的 *1/6*。

130

> B 树的上溢是指：
> A. 删除关键码后违反了 B 树的性质
> B. 插入新的关键码后违反了 B 树的性质
> C. 分裂后违反了 B 树的性质
> D. B 树的高度过高

Solution 17. 正确答案是 *B*。

详细分析：

(1) B 树的性质： 一个 *m* 阶 *B* 树的每个节点最多只能包含 $m-1$ 个关键字。

(2) 上溢（Overflow）的定义： 当向一个已经包含 $m-1$ 个关键字的节点（即已满节点）插入一个新的关键字时，该节点的关键字数量会临时变为 m。这个状态违反了 *B* 树关于节点最大关键字数量的规定，这种情况就称为“上溢”。

(3) 上溢的处理： 上溢是通过“分裂”(*split*) 操作来解决的。将上溢的节点分裂成两个新节点，并将中间的关键字提升到父节点中。

(4) 分析选项：

- ***A. 删除关键码后违反了 B 树的性质：*** 这不是上溢的定义。删除可能导致下溢（*underflow*），而不是上溢。
- ***B. 插入新的关键码后违反了 B 树的性质：*** 这是正确的。上溢是插入操作的直接后果，具体表现为节点关键字数量超过上限。
- ***C. 分裂后违反了 B 树的性质：*** 分裂是解决上溢的方法，其目的是恢复 *B* 树的性质，而不是违反它。
- ***D. B 树的高度过高：*** 上溢是单个节点的局部状态，与树的整体高度没有直接关系。*B* 树的设计本身就是为了保持较低的高度。

结论： *B* 树的上溢特指在插入操作中，某个节点的关键字数量超过了允许的最大值，从而违反了 *B* 树的性质。

131

> B 树高度的增加一定伴随着:
> A. 每个节点所存放的关键码数量增加
> B. 每个节点所存放的关键码数量减少
> C. 分裂到根
> D. 分裂到叶

Solution 18. 正确答案是 C。

详细分析:

B 树的高度增长是一个相对不频繁的事件，它只在一种特定情况下发生。

(1) **插入与上溢:** 当向 B 树中插入一个新关键字时，可能会导致某个叶子节点发生“上溢”(关键字数量超过上限)。

(2) **分裂的传播:** 为了解决上溢，该节点会进行“分裂”。分裂操作会将一个中间关键字提升到父节点，并可能导致父节点也发生上溢，从而引发父节点的分裂。这个分裂过程可以像链式反应一样，沿着父节点链一直向上传播。

(3) **高度增加的唯一条件:** B 树的高度 **唯一** 增加的时刻是当这个分裂的链式反应一直传播到根节点，并导致根节点也发生上溢时。此时，根节点会分裂成两个新的节点，同时会创建一个全新的、只包含一个关键字（从旧根节点提升上来的中间关键字）的根节点。这个新根节点的创建使得树的整体高度增加了 1。

(4) **分析选项:**

- A 和 B: 分裂操作会重新分配关键字，而不是单纯地增加或减少所有节点的关键字数量。
- ***C.* 分裂到根:** 这是正确的。只有当分裂操作一直传播到根节点，导致根节点本身也分裂时，树的高度才会增加。
- D: 分裂是从叶子节点开始向上进行的，而不是分裂到叶子。

结论: B 树高度的增加必然是由于根节点发生了分裂，而根节点的分裂是由一系列从下至上的分裂操作传播而来的。

132

> B 树的下溢发生于:
> A. B 树高度减少
> B. 插入关键码后违反了 B 树的性质
> C. 删除关键码后违反了 B 树的性质
> D. 在叶节点插入关键码

Solution 19. 正确答案是 C。

详细分析:

(1) ***B* 树的性质:** 一个 m 阶 B 树中，除了根节点，每个节点必须至少有 $\lceil m/2 \rceil - 1$ 个关键字。

(2) **下溢（*Underflow*）的定义:** 当从一个恰好包含最小数量关键字的节点中删除一个关键字时，该节点的关键字数量会降到规定的下限以下。这个状态违反了 B 树关于节点最小关键字数量的规定，这种情况就称为“下溢”。

(3) **下溢的处理：**下溢通常通过向兄弟节点“借用”关键字（旋转）或与兄弟节点“合并”来解决。

(4) **分析选项：**

- *A. B* 树高度减少：*B* 树高度减少是处理下溢的一种可能结果（当根节点的两个孩子合并，导致旧根被删除时），但它不是下溢本身。
- *B.* 插入关键码后违反了 *B* 树的性质：这是“上溢”（*Overflow*），指插入后节点的关键字数量超过了规定的上限。
- ***C.* 删除关键码后违反了 *B* 树的性质：**这是正确的。下溢是删除操作的直接后果，具体表现为节点关键字数量少于下限。
- *D.* 在叶节点插入关键码：这是插入操作，可能导致上溢，而不是下溢。

结论： *B* 树的下溢特指在删除操作中，某个节点的关键字数量少于了允许的最小值，从而违反了 *B* 树的性质。

133

> B 树高度的减少只会发生于
> A. 根节点的两个孩子合并
> B. 根节点被删除
> C. 根节点发生旋转
> D. 根节点有多个关键码

Solution 20. 正确答案是 *A*。

详细分析：

B 树的高度减少是一个相对不频繁的事件，它只在一种特定情况下发生。

(1) **删除与下溢：**当从 *B* 树中删除一个关键字时，可能会导致某个节点发生“下溢”（关键字数量少于上限）。

(2) **下溢的解决：**下溢通常通过向兄弟节点“借用”关键字（旋转）或与兄弟节点“合并”来解决。旋转不改变树的高度。合并会导致父节点失去一个关键字和一个孩子指针。

(3) **合并的传播：**如果父节点因为失去一个关键字而发生下溢，这个合并过程就会向上传播。

(4) **高度减少的唯一条件：***B* 树的高度 **唯一** 减少的时刻是当这个合并的连锁反应一直传播到根节点时。具体来说：

- 此时的根节点必须只包含一个关键字（因此它只有两个孩子）。
- 当它的这两个孩子因为下溢而需要合并时，它们会与根节点中的那个关键字一起，形成一个全新的、单独的节点。
- 这个新合并的节点将成为树的全新根节点。
- 原来的根节点因为其唯一的关键字和所有的孩子都已并入新节点而变得多余，从而被删除。
- 这个过程使得树的层数减少了 *1*，即高度减小了 *1*。

(5) **分析选项：**

- ***A.* 根节点的两个孩子合并：**这是正确的。这是导致旧根被移除、树高减 *1* 的直接和根本原因。
- *B.* 根节点被删除：这是 *A* 选项的结果，而不是原因。根节点之所以被删除，是因为它的孩子们合并了。
- *C.* 根节点发生旋转：旋转是解决下溢的一种方式，但它不改变树的高度。

- *D.* 根节点有多个关键码：如果根节点有多个关键字，它就会有三个或更多的孩子。即使其中两个孩子合并，根节点依然存在，树的高度不会减少。

结论： *B* 树高度的减少必然是由于根节点发生了分裂，而根节点的分裂是由一系列从下至上的分裂操作传播而来的。

134

> 我们分别学习了 B-树的插入和删除，请判断正误：
> 如果在不发生上溢和下溢的情况下，那么单次删除和插入操作的时间花费大致相同。

Solution 21. 这个陈述是 **错误的**。

详细分析：

即使在不发生上溢和下溢的情况下，*B* 树的删除操作通常比插入操作更复杂，时间花费也更多。

***(1)* 插入操作（无上溢）：**

- 从根节点开始，沿着合适的路径向下搜索到叶节点
- 在叶节点中找到合适的位置插入新关键字
- 保持节点内关键字的有序性
- 操作相对简单直接

***(2)* 删除操作（无下溢）的复杂性：** 删除操作需要考虑三种不同的情况：

- **情况 *1*：删除叶节点中的关键字** 这种情况相对简单，类似于插入操作。
- **情况 *2*：删除内部节点中的关键字** 这种情况更复杂：
  - 不能直接删除内部节点的关键字
  - 需要找到该关键字的前驱或后继
  - 用前驱/后继替换要删除的关键字
  - 然后删除原前驱/后继位置的关键字
- **情况 *3*：需要向兄弟节点借用或进行调整** 即使不发生下溢，有时也需要进行内部调整以维持 *B* 树的平衡性质。

***(3)* 时间复杂度对比：**

- **插入（无上溢）：** 主要是搜索路径 + 简单插入，时间复杂度为 $O(\log n)$
- **删除（无下溢）：** 搜索路径 + 可能的前驱/后继查找 + 替换操作，时间复杂度也是 $O(\log n)$，但常数因子更大

(4) 即使渐进复杂度相同，删除操作在实际执行中通常需要更多的步骤和更复杂的逻辑判断。

结论： 虽然两者的渐进时间复杂度都是 $O(\log n)$，但删除操作的实际时间开销通常大于插入操作，因为删除涉及更多的情况判断和可能的额外操作步骤。

135

> 伸展树虽然单次操作的最坏时间复杂度比较大，但是可以利用存储器的层次结构降低 I/O 的次数

Solution 22. 这个陈述是 **错误的**。

详细分析：

(1) 关于第一部分:“伸展树虽然单次操作的最坏时间复杂度比较大”

- 这一部分是**正确**的。伸展树的单次操作（查找、插入、删除）在最坏情况下，例如访问一个链状树的末端节点，其时间复杂度的确是 $O(n)$。

(2) 关于第二部分:“但是可以利用存储器的层次结构降低 I/O 的次数”

- 这一部分是**错误**的。
- 专门为利用存储器层次结构（特别是主存-外存结构）来降低 *I/O* 次数而设计的数据结构是 ***B* 树（*B-tree*）**，而不是伸展树。
- ***B* 树**通过其“矮胖”的结构（每个节点存储大量关键字，分支数多）来最小化树的高度。由于每次从外存（如磁盘）读取数据通常是按块（*block/page*）进行的，*B* 树的节点大小通常被设计为与磁盘块大小相匹配。这样，一次 *I/O* 操作就可以将大量相关关键字读入内存，从而大大减少了访问外存的次数。
- **伸展树**是一种二叉搜索树，它的优势在于利用了访问模式的“局部性原理”。通过将最近访问的节点移动到根部，使得后续对该节点或其附近节点的访问变得更快。这种优化主要体现在**内存内部**的缓存效率上，它能获得很好的**分摊（均摊）性能**，但其结构（瘦高）并不适合减少对外存的 *I/O* 操作。在需要频繁进行磁盘 *I/O* 的场景下，伸展树的性能远不如 *B* 树。

结论: 该陈述将伸展树的特性与 *B* 树的特性混淆了。伸展树优化的是分摊时间复杂度，而 *B* 树优化的是 *I/O* 次数。因此，整个陈述是错误的。

136

> 伸展树相较于 AVL 树的缺点是它实现起来较为复杂

Solution 23. 这个陈述是 **错误的**。

详细分析:

通常情况下，伸展树（*Splay Tree*）被认为比 *AVL* 树的实现**更简单**。

***(1) AVL* 树的实现复杂性:**

- **存储开销:** 每个节点需要额外存储一个平衡因子（或子树高度），增加了节点的复杂性。
- **维护平衡因子:** 在每次插入或删除后，需要从修改点一路向上更新路径上所有祖先节点的平衡因子。
- **复杂的旋转逻辑:** 需要准确判断失衡的四种情况（*LL, RR, LR, RL*），并执行相应的单旋转或双旋转。删除操作后的平衡恢复逻辑尤其复杂，可能需要多次旋转才能恢复整棵树的平衡。

***(2)* 伸展树的实现简单性:**

- **无额外存储:** 节点结构与普通二叉搜索树相同，无需存储平衡因子或高度。
- **统一的伸展操作:** 所有的操作（查找、插入、删除）都依赖于一个核心的、统一的“伸展”（*splay*）操作。程序员只需要正确实现这一个核心函数即可。
- **简化的逻辑:** 伸展操作的逻辑是固定的（循环执行 *zig, zig-zig, zig-zag*），不需要像 *AVL* 树那样根据不同的失衡情况进行复杂的判断。

结论: 伸展树的优点之一恰恰是其实现相对简单。它的缺点主要在于单次操作的最坏时间复杂度为 $O(n)$（尽管分摊复杂度很好），以及其树形结构会因访问模式而不断剧烈变化，这在某些需要稳定结构的场景下可能不适用。而“实现复杂”通常被认为是 *AVL* 树的缺点。因此，原陈述是错误的。

137

> 伸展树插入操作的分摊复杂度比 AVL 树大

Solution 24. 这个陈述是 **错误的**。

详细分析:

我们需要比较两种数据结构在插入操作上的 ** 分摊（均摊）复杂度 **。

***(1) AVL* 树的复杂度:**

- *AVL* 树是一种严格的平衡二叉搜索树。它的高度始终保持在 $O(\log n)$。
- 一次插入操作包括：查找插入位置（$O(\log n)$），插入节点，然后从插入点向上回溯更新平衡因子并可能执行一次旋转（$O(\log n)$）。
- 因此，*AVL* 树的**单次操作最坏时间复杂度**就是 $O(\log n)$。
- 一个操作的均摊复杂度不会超过其单次操作的最坏复杂度，所以 *AVL* 树插入操作的**分摊复杂度也是** $O(\log n)$。

***(2)* 伸展树的复杂度:**

- 伸展树的单次操作最坏时间复杂度是 $O(n)$。
- 然而，伸展树最重要的理论成果就是其**分摊复杂度**。通过势能法分析可以证明，对伸展树进行任何操作（包括插入）的**分摊复杂度为** $O(\log n)$。
- 这意味着，虽然偶尔有一次操作可能很慢（$O(n)$），但在一系列连续的操作中，平均每次操作的成本是 $O(\log n)$。

结论: 伸展树插入操作的分摊复杂度为 $O(\log n)$，*AVL* 树插入操作的分摊复杂度也为 $O(\log n)$。两者在分摊复杂度上是**相同**的。

因此，原陈述“伸展树插入操作的分摊复杂度比 *AVL* 树大”是错误的。

138

> 伸展树单次查找操作的最坏时间复杂度比 AVL 树大

Solution 25. 这个陈述是 **正确的**。

详细分析:

我们需要比较两种数据结构在**单次查找操作**的**最坏时间复杂度**。

***(1) AVL* 树的复杂度:**

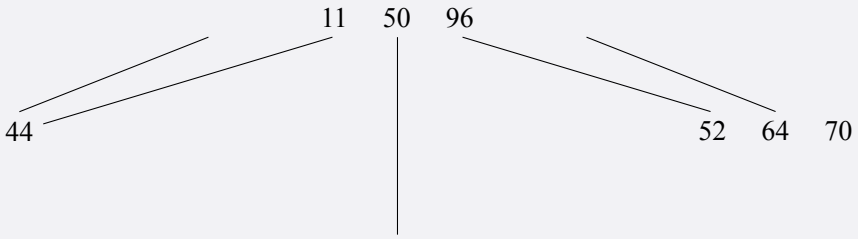
- *AVL* 树通过旋转操作严格保证了树的高度始终为 $O(\log n)$，其中 n 是节点的数量。
- 查找操作最多需要从根遍历到最深的叶子，其路径长度不会超过树的高度。
- 因此，*AVL* 树单次查找操作的**最坏时间复杂度是** $O(\log n)$。这个性能是有保证的。

***(2)* 伸展树的复杂度:**

- 伸展树不保证严格的平衡。在最坏的情况下，伸展树可以退化成一个线性链表。
- 如果树退化成一个链表，并且我们要查找的元素位于链表的末端，那么查找操作需要遍历整条链表，即访问 n 个节点。
- 因此，伸展树单次查找操作的**最坏时间复杂度是** $O(n)$。
- （尽管其分摊复杂度是 $O(\log n)$，但这不影响单次最坏情况的分析）。

结论：伸展树的单次查找最坏时间复杂度为 $O(n)$，而 *AVL* 树的单次查找最坏时间复杂度为 $O(\log n)$。由于 $O(n)$ 在渐进意义上大于 $O(\log n)$，所以该陈述是正确的。

139

> 下图是 (3,6)-树中刚删除某节点后的情形，可以看出发生了下溢，调整后的结果为：
>
> 
>

Solution 26. 详细分析：

***(1)* 分析 *B* 树性质：**

- 题目给出的是一个 *(3,6)*-树。这通常指 *B* 树的阶 $m = 6$。
- 对于一个 $m = 6$ 阶的 *B* 树，除根节点外，每个节点必须至少有 $\lceil m/2 \rceil - 1 = \lceil 6/2 \rceil - 1 = 3 - 1 = 2$ 个关键字。

***(2)* 识别下溢：**

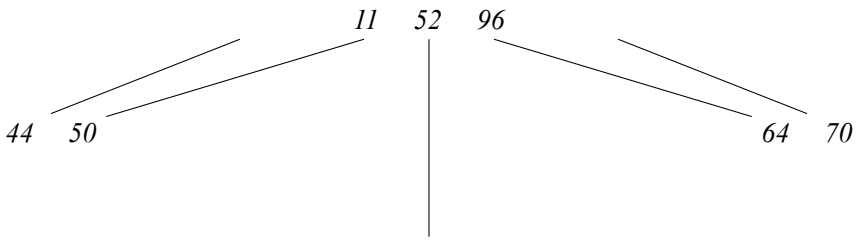
- 左侧节点只包含一个关键字 *44*，这少于了规定的最少 *2* 个关键字，因此发生了“下溢”。
- 其右兄弟节点包含 *52, 64, 70*，共 *3* 个关键字，多于最少数量，因此可以向它“借用”一个关键字。

***(3)* 确定调整策略（旋转）：** 当一个节点下溢，而其相邻的兄弟节点有多余的关键字时，优先采用“旋转”（*borrowing*）操作进行调整。

- **步骤 *1*：**将父节点中分隔下溢节点和其富裕兄弟的关键字（即 *50*）移动到下溢节点中。下溢节点变为 *44, 50*。
- **步骤 *2*：**将富裕兄弟节点中的最小关键字（即 *52*）提升到父节点，以填补 *50* 留下的空位。
- **步骤 *3*：**富裕兄弟节点失去其最小关键字 *52*，变为 *64, 70*。

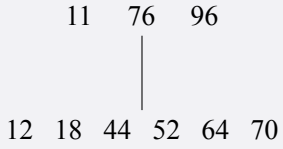
经过这个旋转操作，所有节点都重新满足了 *B* 树的性质。

调整后的结果为：



140

下图是 (3,6)-树中刚插入节点 52 后的情形，可以看出发生了上溢，分裂后的结果为：



Solution 27. 详细分析：

***(1)* 分析 *B* 树性质：**

- 题目给出的是一个 *(3,6)*-树，这通常指 B 树的阶 $m = 6$。
- 对于一个 $m = 6$ 阶的 B 树，每个节点最多只能包含 $m - 1 = 5$ 个关键字。

***(2)* 识别上溢：**

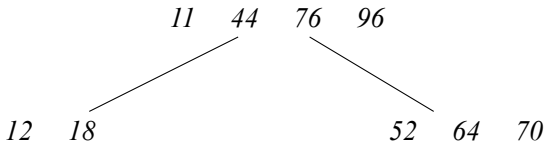
- 子节点包含 *12, 18, 44, 52, 64, 70*，共 *6* 个关键字，这超过了规定的最多 *5* 个关键字，因此发生了“上溢”。

***(3)* 确定调整策略（分裂）：** 当一个节点发生上溢时，必须进行“分裂”（*split*）操作。

- ***步骤 1*：** 从上溢的节点中选取中间的关键字。对于 *12, 18, 44, 52, 64, 70*，中间的关键字是 ***44***。
- ***步骤 2*：** 将这个中间关键字 ***44*** 提升（*promote*）到其父节点中，并按序插入。父节点原来的关键字是 *11, 76, 96*，插入 *44* 后变为 *11, 44, 76, 96*。
- ***步骤 3*：** 将原上溢节点中剩余的关键字分裂成两个新的节点：
  - 小于 *44* 的关键字形成一个新的左孩子节点：*12, 18*。
  - 大于 *44* 的关键字形成一个新的右孩子节点：*52, 64, 70*。
- ***步骤 4*：** 将这两个新节点链接到父节点，分别位于提升上来的关键字 *44* 的两侧。

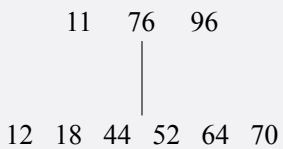
经过这个分裂操作，所有节点都重新满足了 B 树的性质。

分裂后的结果为：



140 (版本二)

下图是 (3,6)-树中刚插入节点 52 后的情形，可以看出发生了上溢，分裂后的结果为（以 52 为中间关键字）：



Solution 28. 详细分析：

***(1)* 分析 *B* 树性质：**

- 题目给出的是一个 *(3,6)*-树，即 *B* 树的阶 $m = 6$。
- 每个节点最多只能包含 $m - 1 = 5$ 个关键字。

***(2)* 识别上溢：**

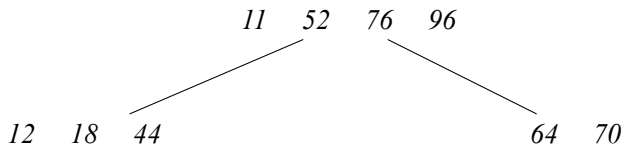
- 子节点包含 *12, 18, 44, 52, 64, 70*，共 *6* 个关键字，超过了最多 *5* 个的限制，发生“上溢”。

(3)* 确定调整策略（分裂）：** 当节点上溢时，必须进行“分裂”（*split*）操作。对于一个有偶数个关键字的节点，中间关键字的选择可以有两种（例如，对于 *6* 个关键字，可以选择第 *3* 个或第 *4* 个）。这里我们按照要求，选择 ***52 作为提升的中间关键字。

- **步骤 *1*：** 从上溢的节点中选取中间的关键字 ***52***。
- **步骤 *2*：** 将这个中间关键字 ***52*** 提升（*promote*）到其父节点中，并按序插入。父节点原来的关键字是 *11, 76, 96*，插入 *52* 后变为 *11, 52, 76, 96*。
- **步骤 *3*：** 将原上溢节点中剩余的关键字分裂成两个新的节点：
  - 小于 *52* 的关键字形成一个新的左孩子节点：*12, 18, 44*。
  - 大于 *52* 的关键字形成一个新的右孩子节点：*64, 70*。
- **步骤 *4*：** 将这两个新节点链接到父节点，分别位于提升上来的关键字 *52* 的两侧。

经过这个分裂操作，所有节点都重新满足了 *B* 树的性质。

分裂后的结果为：



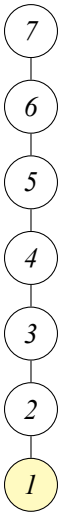
141

> 在以下伸展树中插入节点 1 并经过双层伸展后的结果是：
>
> 7
> |
> 6
> |
> 5
> |
> 4
> |
> 3
> |
> 2

Solution 29. 详细分析：

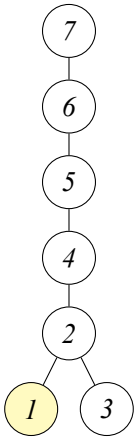
第 0 步：插入节点 1

首先，按照二叉搜索树的规则，将节点 *1* 插入为节点 *2* 的左孩子。此时，目标节点 ***1*** 位于树的最深处。

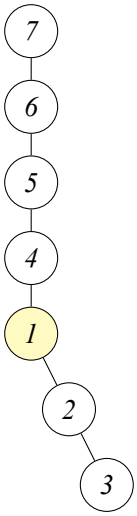


第 1 轮伸展：Zig-Zig (x=1, p=2, g=3)

1a. 第一次右旋 (围绕 g=3):

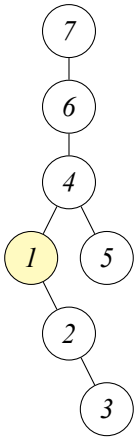


1b. 第二次右旋 (围绕 p=2):

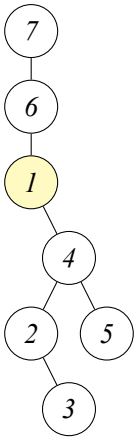


第 2 轮伸展：Zig-Zig (x=1, p=4, g=5)

2a. 第三次右旋 (围绕 g=5):

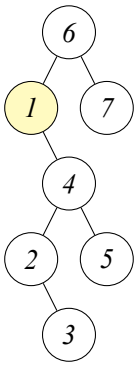


2b. 第四次右旋 (围绕 p=4):

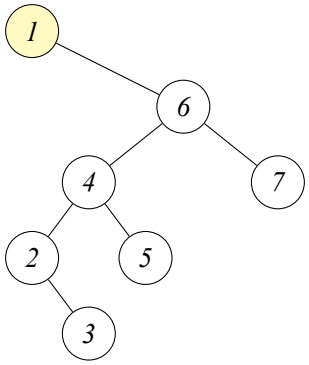


第 3 轮伸展：Zig-Zig (x=1, p=6, g=7)

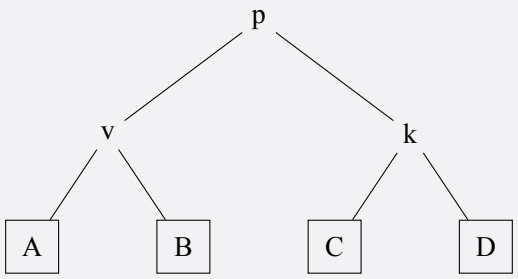
3a. 第五次右旋 (围绕 g=7):



3b. 第六次右旋 (围绕 p=6) - 最终结果:



142（拓展）

> 下图是伸展树，其中节点 v 刚被访问过，双层伸展后的结果是：
>
> 
>

Solution 30. 详细分析：

***(1)* 分析伸展情况：**

- 被访问的节点是 ***v***。
- 节点 ***v*** 的父节点是 ***p***。
- 父节点 ***p*** 是树的根节点，因此节点 ***v*** **没有祖父节点**。

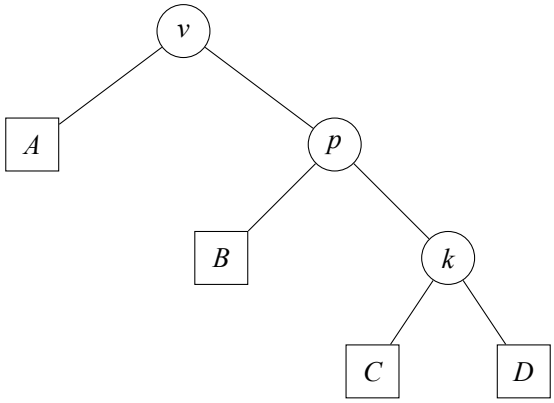
***(2)* 确定伸展操作：**

- “双层伸展”（*zig-zig* 或 *zig-zag*）只在被访问节点**存在祖父节点**时执行。
- 当被访问节点的父节点就是根节点时，执行的是最简单的“单层伸展”，即 ***zig*** 操作。
- 在本例中，节点 ***v*** 是其父节点 ***p*** 的**左孩子**。因此，需要对父节点 ***p*** 执行一次**右旋转**。

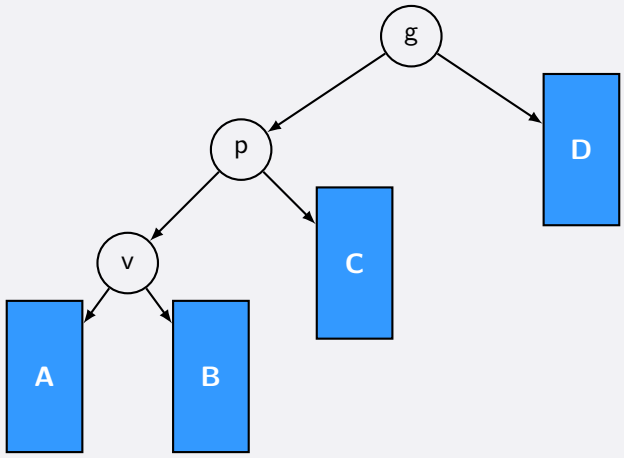
***(3)* 执行右旋转（围绕 *p*）：**

- 节点 ***v*** 上升，成为新的根。
- 节点 ***p*** 下降，成为 ***v*** 的右孩子。
- 节点 ***v*** 原来的右子树（即子树 ***B***）成为 ***p*** 的新左孩子。
- 节点 ***v*** 原来的左子树（即子树 ***A***）保持不变，仍是 ***v*** 的左孩子。
- 节点 ***p*** 原来的右子树（即节点 ***k*** 和其子树）保持不变，仍是 ***p*** 的右孩子。

伸展后的结果为：



142

> 下图是伸展树，其中节点 v 刚被访问过，双层伸展后的结果是：
>
> 
>

Solution 31. 详细分析：

***(1)* 分析伸展情况：**

- 被访问的节点（目标节点）是 ***v***。
- ***v*** 的父节点是 ***p***。
- ***v*** 的祖父节点是 ***g***。

***(2)* 确定伸展操作：**

- 节点 ***v*** 是其父节点 ***p*** 的 ***左孩子*** *(zig)*。
- 节点 ***p*** 是其父节点 ***g*** 的 ***左孩子*** *(zig)*。
- 由于两个关系的方向相同（都是左孩子），因此需要执行 ***zig-zig*** 操作。

***(3)* 执行 *Zig-Zig* 操作：** *Zig-zig* 操作包含两次同向的旋转。

- ***第一次旋转：*** 对祖父节点 ***g*** 进行一次右旋转。这会使 ***p*** 上升，***g*** 下降成为 ***p*** 的右孩子。
- ***第二次旋转：*** 对（新的）父节点 ***p*** 进行一次右旋转。这会使 ***v*** 上升，成为新的根。

***(4)* 重构子树链接：** 在两次旋转后，为了保持二叉搜索树的性质，四个子树 *A, B, C, D* 会被重新链接：

- 新的根是 ***v***。
- ***v*** 的左子树仍然是 ***A***。
- ***v*** 的右孩子是 ***p***。

- ***p*** 的左孩子是 ***v*** 原来的右子树 ***B***。
- ***p*** 的右孩子是 ***g***。
- ***g*** 的左孩子是 ***p*** 原来的右子树 ***C***。
- ***g*** 的右孩子是 ***g*** 原来的右子树 ***D***。

伸展后的结果为：