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 第六章二叉搜索树

97

> 二叉搜索树之区别于普通的二叉树在于：
> A. 任意节点均不大于其右子树中的节点，不小于其左子树中的节点
> B. 任意节点均不大于其右孩子，不小于其左孩子
> C. 除了根节点外所有节点均不大于其父亲
> D. 关键码可以比较

Solution 1. 正确答案是 *A*。

详细分析：

二叉搜索树（*Binary Search Tree, BST*）是一种特殊的二叉树，它具有以下性质（对于树中任意节点 *N*）：

- *N* 的左子树中所有节点的值（关键码）均小于或等于 *N* 的值。
- *N* 的右子树中所有节点的值（关键码）均大于或等于 *N* 的值。
- *N* 的左、右子树也分别为二叉搜索树。

（注意：关于等号的处理，不同定义可能略有差异，例如不允许重复值，或者重复值只存在于一侧。但核心思想是基于比较的有序性。）

我们来分析各个选项：

- ***A.* 任意节点均不大于其右子树中的节点，不小于其左子树中的节点。**
  - “任意节点均不大于其右子树中的节点”意味着：节点的值 ≤ 其右子树中任意节点的值。
  - “任意节点均不小于其左子树中的节点”意味着：节点的值 ≥ 其左子树中任意节点的值。

  这可以概括为：左子树所有节点的值 ≤ 当前节点的值 ≤ 右子树所有节点的值。这正是二叉搜索树的核心定义。
- ***B.* 任意节点均不大于其右孩子，不小于其左孩子。**这个条件只约束了节点与其直接孩子之间的关系，而没有约束与整个子树的关系。一个满足此条件的树不一定是二叉搜索树。例如：

```
    5
   / \
  3   8
       \
        4  （节点8的值 > 节点4的值，但4应在5的左子树或8的左子树）
```

  在这个例子中，$5 \geq 3$ 且 $5 \leq 8$。但整个树不是 *BST*，因为 *4* 在 *8* 的右子树中，而 $4 < 8$ *(*且 $4 < 5$*)*。
- ***C.* 除了根节点外所有节点均不大于其父亲。**这个描述（节点值 ≤ 父节点值）更像是堆（特别是最大堆）的性质，而不是二叉搜索树的性质。在二叉搜索树中，右孩子的值通常大于其父节点的值。
- ***D.* 关键码可以比较。**虽然二叉搜索树要求节点中的关键码是可以比较的，但这只是一个前提条件，而不是二叉搜索树区别于普通二叉树的结构性特征。一个普通的二叉树也可以存储可比较的关键码，但它不一定满足二叉搜索树的有序性。

因此，选项 *A* 最准确地描述了二叉搜索树区别于普通二叉树的根本特性。

98

> 二叉搜索树的何种遍历序列是递增的?
> A. 先序
> B. 中序
> C. 后序
> D. 层次

Solution 2. 正确答案是 *B*。

详细分析:

二叉搜索树（*Binary Search Tree, BST*）具有以下关键性质：对于树中的任意节点 *N*:

- *N* 的左子树中所有节点的值均小于（或等于，取决于具体定义）*N* 的值。
- *N* 的右子树中所有节点的值均大于（或等于）*N* 的值。
- *N* 的左、右子树也分别为二叉搜索树。

我们来分析各种遍历方式:

- ***A. 先序遍历 (根-左-右)***：首先访问根节点，然后是左子树，然后是右子树。由于根节点的值介于左子树和右子树的值之间，所以先序遍历序列通常不是递增的。例如，对于 *BST*:

```
      4
    /   \
   2     6
  / \   / \
 1   3 5   7
```

  先序遍历是：*4, 2, 1, 3, 6, 5, 7*。这不是递增的。

- ***B. 中序遍历 (左-根-右)***：首先递归地遍历左子树，然后访问根节点，最后递归地遍历右子树。根据 *BST* 的性质，左子树中的所有值都小于根节点的值，而右子树中的所有值都大于根节点的值。因此，中序遍历一个 *BST* 会得到一个按升序排列的节点值序列。对于上面的例子，中序遍历是：*1, 2, 3, 4, 5, 6, 7*。这是递增的。**这是正确的。**
- ***C. 后序遍历 (左-右-根)***：首先递归地遍历左子树，然后是右子树，最后访问根节点。根节点是最后访问的，所以这通常不是递增的。对于上面的例子，后序遍历是：*1, 3, 2, 5, 7, 6, 4*。这不是递增的。
- ***D. 层次遍历 (按层，从左到右)***：节点按层级顺序访问，同一层级从左到右。这通常不是递增的。对于上面的例子，层次遍历是：*4, 2, 6, 1, 3, 5, 7*。这不是递增的。

因此，二叉搜索树的中序遍历序列是递增的。

99

> 在含 n 个节点的 BST 中进行查找的最坏时间复杂度为:
> A. O(1)
> B. $O(\log_2(n))$
> C. O(n)
> D. $O(n\log_2(n))$

Solution 3. 正确答案是 C。

详细分析：

二叉搜索树（*Binary Search Tree, BST*）的查找操作依赖于树的高度。

- **查找过程：**从根节点开始，将目标值与当前节点的值进行比较。
  - 如果目标值等于当前节点的值，则查找成功。
  - 如果目标值小于当前节点的值，则在当前节点的左子树中继续查找。
  - 如果目标值大于当前节点的值，则在当前节点的右子树中继续查找。

  这个过程一直持续到找到目标节点或者到达一个空子树（表示目标值不存在于树中）。
- **时间复杂度：**查找操作所花费的时间与从根节点到目标节点（或查找路径的末端）的路径长度成正比。这条路径的长度最多等于树的高度。
- **最好情况：**如果二叉搜索树是平衡的（例如 *AVL* 树或红黑树），其高度大约为 $O(\log_2 n)$。在这种情况下，查找的时间复杂度是 $O(\log_2 n)$。
- **最坏情况：**如果二叉搜索树是极度不平衡的，例如退化成一个链表（所有节点都只有左孩子或都只有右孩子），那么树的高度将是 $n-1$，即 $O(n)$。在这种情况下，查找操作可能需要遍历树中的所有 n 个节点。例如，如果节点按顺序 *1, 2, 3, ..., n* 插入，*BST* 会变成一个只有右孩子的链：

```
1
 \
  2
   \
    3
     ...
      \
       n
```

  查找节点 n *(*或一个大于 n 的值*)* 将需要访问所有 n 个节点。
- **平均情况：**对于随机插入的节点形成的 *BST*，平均查找时间复杂度也是 $O(\log_2 n)$，但最坏情况仍然是 $O(n)$。

题目问的是“最坏时间复杂度”，因此我们考虑树退化成链表的情况。

分析选项：

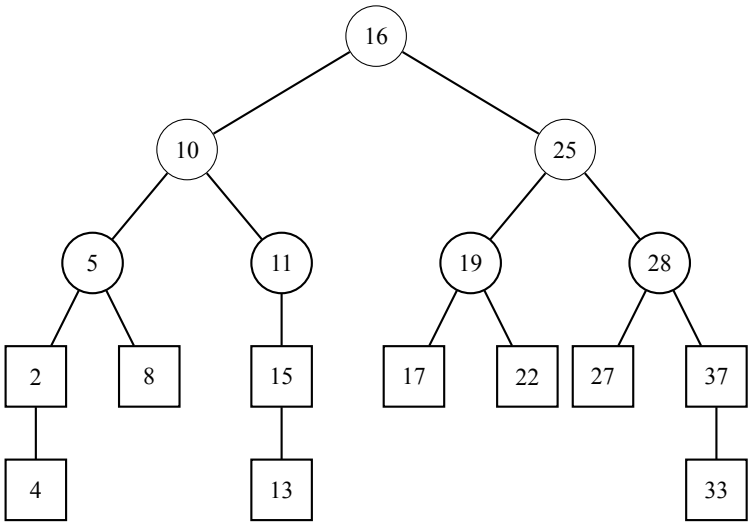
- ***A. O(1)***：只有当要查找的元素恰好是根节点时，才可能是 *O(1)*，这不是普遍的最坏情况。
- ***B. O(log$_2$(n))***：这是平衡 *BST* 的查找时间复杂度，不是最坏情况。
- ***C. O(n)***：这是当 *BST* 退化成链表时的查找时间复杂度，是最坏情况。
- ***D. O(nlog$_2$(n))***：这个复杂度通常与基于比较的排序算法相关，不直接对应 *BST* 的单次查找操作。

因此，在含 *n* 个节点的 *BST* 中进行查找的最坏时间复杂度为 $O(n)$。

100

> 在以下二叉查找树中查找关键码 13，经过的节点依次为：
>
> A. 16, 11, 13
>
> B. 16, 10, 5, 8, 13

C. 16, 10, 11, 15, 13
D. 以上二叉树根本不是二叉查找树



Solution 4. 正确答案是 *C*。

101

对 BST 进行插入操作，对待插入的目标元素 e 进行查找后，若查找失败，_hot 指向的节点为:
A. 待插入的节点
B. 被插入后的父亲
C. 被插入后的左孩子
D. 根节点

Solution 5. 正确答案是 *B*。

详细分析:

在二叉搜索树（*BST*）的插入操作中，通常会首先执行一个查找操作来确定待插入元素 *'e'* 的位置。这个查找过程会从根节点开始，根据 `e` 与当前节点值的比较，向左或向右子树深入。变量 `_hot` *(*或者类似名称的变量，如 `parent` 或 `p`*)* 通常用来追踪当前查找路径上节点的父节点。

具体步骤如下:

(1) 初始化一个指针 `current` 指向根节点，`_hot` 指向 *null (*或者一个哨兵*)*。

(2) 当 `current` 不为 *null* 时，循环:

- ***(1)*** 将 `_hot` 更新为 `current`。
- ***(2)*** 如果 `e` 等于 `current` 的值，则查找成功（元素已存在，通常不插入或根据策略处理重复）。
- ***(3)*** 如果 `e` 小于 `current` 的值，则 `current` 移动到其左孩子。
- ***(4)*** 如果 `e` 大于 `current` 的值，则 `current` 移动到其右孩子。

(3) 当循环结束时，如果 `current` 为 *null*，则表示查找失败。此时，`_hot` 指向的是查找路径上最后一个非空节点，也就是新节点 `e` 应该被插入为其孩子的位置的父节点。

因此，若查找失败，`_hot` 指向的节点将成为新插入节点 `e` 的父亲节点。

分析选项:

- ***A.* 待插入的节点：** $\_hot$ 指向的是树中已存在的节点，而不是尚未创建的待插入节点。
- ***B.* 被插入后的父亲：** 这是正确的。$\_hot$ 是查找路径上最后一个访问的非空节点，新节点将作为 $\_hot$ 的左孩子或右孩子插入。
- ***C.* 被插入后的左孩子：** 新插入的节点是 $\_hot$ 的孩子，而不是 $\_hot$ 是新节点的左孩子。
- ***D.* 根节点：** 只有当树为空（此时 $\_hot$ 可能为 *null*，新节点成为根）或者新节点直接插入为根节点的子节点时，$\_hot$ 才可能是根节点。一般情况下，$\_hot$ 是查找路径上更深层的节点。

所以，$\_hot$ 指向的是被插入节点 e 的父亲节点。

102

> 当欲删除的节点 v 在 BST 中的度为 2 时，实际被删除的节点为：
> A. v 在中序遍历下的直接前驱
> B. v 在先序遍历下的直接后继
> C. v 的右子树中左侧分支的最后一个节点
> D. v 的父亲

Solution 6. 正确答案是 *C (*或者 *A*，取决于具体实现策略，但 *C* 是常见选择*)*。

详细分析：

当在二叉搜索树（*BST*）中删除一个度为 *2* 的节点 *'v'*（即 *'v'* 同时拥有左孩子和右孩子）时，节点 *'v'* 本身并不会直接从树结构中移除。替代地，采用以下步骤：

(1) 找到一个替代节点：这个替代节点通常是 *'v'* 在中序遍历下的直接前驱（即 *'v'* 左子树中的最大节点）或直接后继（即 *'v'* 右子树中的最小节点）。

(2) 用替代节点的值替换 *'v'* 的值。

(3) 然后，删除原来的替代节点。这个替代节点由于其特性（作为子树中的最大或最小元素），其度数最多为 *1*（即它最多只有一个孩子，或者没有孩子），因此删除它会比较简单，可以归约为删除度为 *0* 或度为 *1* 节点的情况。

所以，"实际被删除的节点"指的是这个被用来替换 *'v'* 的值的、原来的前驱或后继节点。

现在分析选项：

- ***A. v* 在中序遍历下的直接前驱：** 这是指 *'v'* 左子树中的最大节点。如果选择用前驱来替换 *'v'*，那么这个前驱节点就是实际被从其原位置删除的节点。这是一个有效的策略。
- ***B. v* 在先序遍历下的直接后继：** 这不是标准 *BST* 删除操作中用来替换度为 *2* 节点的方法。
- ***C. v* 的右子树中左侧分支的最后一个节点：** 这描述的是 *'v'* 在中序遍历下的直接后继，即 *'v'* 右子树中的最小节点（从右孩子开始，一直向左走到尽头）。如果选择用后继来替换 *'v'*，那么这个后继节点就是实际被从其原位置删除的节点。这同样是一个有效的、且非常常见的策略。
- ***D. v* 的父亲：** *'v'* 的父亲与此删除过程无关。

由于选项 *A* 和 *C* 都描述了在标准删除策略中可能被"实际删除"的节点，我们需要判断题目更倾向于哪一个。在许多教材和实现中，选择中序后继（即右子树中的最小节点）作为替代节点是一个非常普遍的做法。

如果必须选择一个，并且考虑到选项 *C* 对该节点的描述（"*v* 的右子树中左侧分支的最后一个节点"）非常具体地指出了如何找到中序后继，这通常是教学中强调的一个方法。

因此，虽然使用中序前驱（选项 *A*）也是一种完全正确的策略，但选项 *C* 描述的中序后继是更常被引用的方法之一。题目问"实际被删除的节点为："，暗示了一个确定的答案，这通常指向一个标准算法步骤。

假设题目倾向于使用中序后继的策略，则选项 C 是正确的。

103

> 两个等价的平衡二叉搜索树有相同的:
> A. 先序遍历序列
> B. 中序遍历序列
> C. 后序遍历序列
> D. 层次遍历序列

Solution 7. 正确答案是 B。

详细分析:

“等价的”平衡二叉搜索树通常指它们包含相同的节点集合（即相同的关键码集合）。“平衡二叉搜索树”是指满足二叉搜索树性质，并且通过特定算法（如 *AVL* 树、红黑树的旋转操作）保持树高在 $O(\log n)$ 数量级的树。

我们来分析各种遍历序列:

- ***A. 先序遍历序列 (根-左-右)***: 先序遍历序列与树的具体结构（尤其是根节点和子树的排列）密切相关。即使两棵平衡二叉搜索树包含相同的节点集，它们的具体结构（由于平衡调整的方式或历史操作可能不同）也可能不同，从而导致不同的先序遍历序列。例如，对于节点集 *{1, 2, 3}*，一棵平衡树可能是根为 *2*，左孩子 *1*，右孩子 *3*；另一棵（如果允许不同的平衡策略或由不同操作序列形成）可能有不同的根或结构，导致不同的先序序列。
- ***B. 中序遍历序列 (左-根-右)***: 对于任何二叉搜索树（无论是否平衡），其中序遍历序列都会得到一个按关键码升序排列的节点序列。如果两棵平衡二叉搜索树是“等价的”（即包含相同的关键码集合），那么它们的中序遍历序列必然是相同的，因为这个序列就是这些关键码的唯一排序结果。**这是正确的。**
- ***C. 后序遍历序列 (左-右-根)***: 与先序遍历类似，后序遍历序列也依赖于树的具体结构。不同的平衡结构会导致不同的后序遍历序列。
- ***D. 层次遍历序列***: 层次遍历序列（按层，从左到右）同样高度依赖于树的结构。不同的平衡结构（例如，同一层上的节点顺序或哪些节点在哪些层）会导致不同的层次遍历序列。

因此，两个包含相同节点集合的平衡二叉搜索树，它们的中序遍历序列一定是相同的，因为该序列代表了这些节点关键码的有序排列。其他遍历序列则会因树的具体平衡结构不同而可能不同。

104

> 在 AVL 树中刚插入一个节点后失衡节点个数最多为
> A. O(1)
> B. O(loglogn)
> C. O(logn)
> D. O(n)

Solution 8. 正确答案是 C。

详细分析：

该问题询问的是在 *AVL* 树中插入一个新节点之后、进行任何旋转恢复平衡操作 * 之前 *，树中最多可能有多少个节点的平衡因子变为 *+2* 或*-2*（即“失衡”）。

***(1)* 影响路径：** 插入一个新节点只会影响从该新节点的父节点到根节点这条路径上各个节点的平衡因子。路径外的其他节点的平衡因子不会改变。

***(2)* 失衡条件：** 一个节点会变得失衡，当且仅当它的平衡因子原本是 *+1* 或*-1*，并且新节点被插入到了其较高的那个子树中，导致该子树的高度增加了 *1*。

***(3)* 最坏情况分析：** 我们可以构造一种情况，使得路径上的多个节点同时满足失衡条件。考虑一棵 *AVL* 树，其中从根节点到某个叶节点路径上的所有节点，其平衡因子都是*-1*（或 *+1*）。例如，一条全由左孩子组成的路径，且路径上每个节点的平衡因子都是*-1*。

```
        p_0 (bf=-1)
       /
     p_1 (bf=-1)
    /
  p_2 (bf=-1)
 /
...
```

当一个新的节点被插入到这条路径最末端节点的左子树时：

- 最末端的节点（比如 p_m）的左子树高度增加 *1*，其平衡因子从*-1* 变为*-2*，导致 p_m 失衡。
- 由于 p_m 的高度增加了 *1*，这会导致其父节点 p_{m-1} 的左子树高度增加 *1*。因为 p_{m-1} 的平衡因子原本也是*-1*，所以它现在也变为*-2* 而失衡。
- 这个效应会沿着路径一直向上传播。路径上的每一个节点的平衡因子都会从*-1* 变为*-2*，从而全部失衡。

***(4)* 复杂度计算：** 这条路径的最大长度等于 *AVL* 树的高度。对于一个包含 *n* 个节点的 *AVL* 树，其高度为 $O(\log n)$。因此，在最坏情况下，失衡节点的个数可以与树的高度成正比。

所以，在插入一个节点后，失衡节点的个数最多为 $O(\log n)$。

（注：虽然可能会有 $O(\log n)$ 个节点失衡，但 *AVL* 树的一个重要特性是，只需要在最靠近插入点的那个失衡节点上执行一次旋转（单旋或双旋），就可以使整棵树恢复平衡。）

105

> 在 AVL 树中刚删除一个节点后失衡节点个数最多为
> A. O(1)
> B. O(loglogn)
> C. O(logn)
> D. O(n)

Solution 9. 正确答案是 *C*。

详细分析：

在 *AVL* 树中，删除一个节点比插入一个节点要复杂。删除操作可能导致树的高度降低，这种高度变化会从被删除节点的父节点开始，沿着到根节点的路径向上传播。

***(1)* 影响路径：**与插入操作类似，删除一个节点只会影响从物理删除点到根节点的路径上各个节点的平衡因子。

***(2)* 失衡与传播：**与插入操作不同，删除操作后的平衡恢复过程可能不会在第一次旋转后就结束。

- 当一个节点的平衡因子从*-1* 变为 *0*（因其左子树高度降低）或从 *+1* 变为 *0*（因其右子树高度降低）时，该节点本身是平衡的，但其所在子树的总高度降低了 *1*。
- 这个高度降低会继续向上传播，可能导致其父节点、祖父节点等更高层的节点失衡。
- 当这个高度降低的传播最终导致某个祖先节点的平衡因子从 *+1* 变为 *+2* 或从*-1* 变为*-2* 时，该祖先节点就失衡了。

***(3)* 最坏情况：**最坏的情况是，高度降低的效应一直传播到根节点。在向上传播的过程中，路径上的每一个节点都可能需要进行检查，并可能需要旋转来恢复平衡。因为在删除后，对一个失衡节点进行旋转，并不能保证其子树的高度恢复到删除前的状态，所以其父节点仍然可能失衡。这个过程需要沿着整条路径一直检查到根。

***(4)* 复杂度计算：**需要检查和可能需要恢复平衡的节点路径，其最大长度等于 *AVL* 树的高度。对于一个包含 n 个节点的 *AVL* 树，其高度为 $O(\log n)$。因此，在删除操作后，沿着路径向上的整个过程中，可能会遇到多个需要调整的失衡节点。

因此，在删除一个节点后，需要考虑进行平衡调整的节点数量（即可能失衡的节点数量）最多可以达到路径的长度，即 $O(\log n)$。

106

> AVL 树中插入节点引发失衡，经旋转调整后重新平衡，此时包含节点 g,p,v 的子树高度
>
> A. 减小 1
>
> B. 不变
>
> C. 增加 1
>
> D. 有可能不变也有可能增加 1

Solution 10. 正确答案是 *B*。

详细分析：

这个问题询问的是，在因插入导致失衡并进行旋转调整后，失衡子树的高度与它在插入操作 * 之前 * 的高度相比，发生了什么变化。

***(1)* 插入前的状态：**假设失衡发生在以节点 *'g'* 为根的子树。在插入新节点之前，*'g'* 是平衡的，我们设其子树高度为 H。为了在插入后会失衡，*'g'* 的平衡因子必须是 *+1* 或 *-1*，表示它的一棵子树比另一棵高 *1*。

***(2)* 插入后、旋转前的状态：**新节点被插入到 *'g'* 的那棵"更高"的子树中，导致该子树的高度增加了 *1*。这使得 *'g'* 的两棵子树高度差变为 *2*，*'g'* 变得失衡。此时，以 *'g'* 为根的子树的总高度也增加了 *1*，变为 $H+1$。

***(3)* 旋转调整后的状态：**对以 *'g'* 为根的失衡子树进行旋转调整（无论是单旋转还是双旋转）。*AVL* 树插入操作后旋转调整的一个关键特性是：**旋转会使该子树的高度恢复到插入操作之前的原始高度**。也就是说，经过旋转后，新的根节点（可能是原来的 *'p'* 或 *'v'*）所领导的子树，其高度会变回 H。

结论：

- 子树在插入前的原始高度为 H。
- 子树在旋转调整后的高度也是 H。

因此，与插入操作之前相比，经过“插入-失衡-旋转”这一完整过程后，子树的高度保持不变。

107

> 经过 3+4 重构后的 AVL 树 _____ 不变。A. 先序遍历序列 B. 中序遍历序列 C. 后序遍历序列 D. 层次遍历序列

Solution 11. 正确答案是 *B*。

详细分析：

“*3+4* 重构”是描述 *AVL* 树（以及其他平衡二叉搜索树）旋转操作的一种通用模型。它涉及到三个节点（失衡的祖父节点 *‘g‘*、其较高的孩子 *‘p‘*、以及 *‘p‘*的较高的孩子 *‘v‘*）和附属于它们的四个可能的子树。

(1) **重构的目的：**旋转或重构操作的根本目的是在不破坏二叉搜索树（*BST*）基本性质的前提下，恢复树的平衡。

(2) ***BST* 的基本性质：**二叉搜索树最重要的性质是，对它进行**中序遍历**会得到一个所有节点按其关键码升序排列的序列。

(3) **重构过程：**“*3+4* 重构”通过将 *‘g‘, ‘p‘, ‘v‘*这三个节点按照它们关键码的大小（$a < b < c$）重新排列，使得关键码为 *‘b‘*的节点成为新的子树根，*‘a‘*成为其左孩子，*‘c‘*成为其右孩子，并重新链接四个子树。这个过程本质上就是一次或两次旋转。

(4) **不变性分析：**

- **中序遍历序列：**由于重构操作必须保持 *BST* 的性质，而 *BST* 的中序遍历结果是唯一的（即所有元素的排序结果），因此，重构前后子树的中序遍历序列是完全相同的。
- **先序、后序、层次遍历序列：**这三种遍历方式都与树的具体拓扑结构（谁是根、谁是谁的孩子）密切相关。重构操作改变了 *‘g‘, ‘p‘, ‘v‘*之间的父子关系，从而改变了树的结构。因此，先序、后序和层次遍历序列几乎总是会发生改变。

结论：“*3+4* 重构”是一种保持中序遍历序列不变的结构调整。因此，经过重构后，*AVL* 树的中序遍历序列保持不变。

108

> AVL 树中删除节点引发失衡，经旋转调整后重新平衡，此时包含节点 g,p,v 的子树高度 A. 减小 1 B. 不变 C. 增加 1 D. 有可能不变也有可能减小 1

Solution 12. 正确答案是 *D*。

详细分析：

在 *AVL* 树中，删除操作比插入操作更为复杂，其对高度的影响也不同。当删除一个节点导致以 *‘g‘* 为根的子树失衡时，我们需要对 *‘g‘* 进行旋转调整。调整后子树的高度变化取决于 *‘g‘* 的较高子树（设其根为 *‘p‘*）的平衡因子。

(1) **失衡的起因：**节点 *‘g‘* 失衡，是因为它的某一个子树（比如左子树）的高度没有变化，而另一个子树（右子树）因为删除了节点，高度减小了 *1*。这使得 *‘g‘* 的平衡因子从*-1* 变为*-2*（或从 *+1* 变为 *+2*）。

(2) **旋转调整后的高度变化：**对 *‘g‘* 进行旋转调整后，新的子树高度与 *‘p‘* 节点的平衡因子有关：

- **情况一：如果 *'p'* 的平衡因子与 *'g'* 的失衡方向相同** *(*例如，*'g'* 是左高，*'p'* 也是左高，即 *LL* 情况*)*，或者相反 *(LR* 情况*)*。在这种情况下，旋转后，整个子树的高度会**减小 *1***。由于高度减小，这个变化会继续向上传播，可能导致更高层的祖先节点也失衡。
- **情况二：如果 *'p'* 的平衡因子为 *0***。在这种情况下，旋转后，整个子树的高度会恢复到删除操作之前的原始高度，即高度**不变**。由于高度没有变化，平衡调整到此结束，不会再影响更高层的祖先节点。

结论：与插入操作不同（插入后的旋转总是使子树高度恢复到插入前的高度），删除操作后的旋转调整，可能会使子树高度减小 *1*，也可能使子树高度保持不变。

因此，选项 *D "*有可能不变也有可能减小 *1"*是最准确的描述。

109

> 包含节点 {1,2,3,4} 的不同二叉搜索树有多少棵?

Solution 13. 答案是 ***14*** 棵。

详细分析：

这个问题询问的是，给定 n 个不同的节点，可以构成多少种结构不同的二叉搜索树（*BST*）。这是一个经典的组合数学问题，其解由第 n 个卡特兰数（*Catalan Number*）给出。

卡特兰数的计算公式为：$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$

对于本题，我们有 $n = 4$ 个节点。因此，我们需要计算第 *4* 个卡特兰数 C_4。

$C_4 = \frac{1}{4+1}\binom{2\times4}{4} = \frac{1}{5}\binom{8}{4}$

首先计算组合数 $\binom{8}{4}$：$\binom{8}{4} = \frac{8!}{4!(8-4)!} = \frac{8!}{4!4!} = \frac{8\times7\times6\times5}{4\times3\times2\times1} = \frac{1680}{24} = 70$

然后代入卡特兰数公式：$C_4 = \frac{1}{5} \times 70 = 14$

另一种解释（递归方法）：令 C_n 为有 n 个节点的 *BST* 的数量。

- $C_0 = 1$ *(*空树*)*
- $C_1 = 1$
- $C_2 = 2$
- $C_3 = 5$

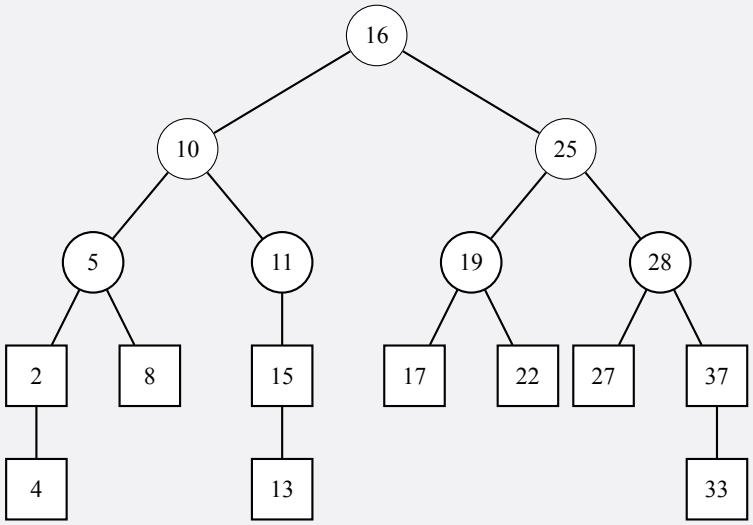
我们可以通过选择不同的根节点来构建有 *4* 个节点的 *BST*:

- ***以 1 为根：****左子树有 0 个节点，右子树有 3 个节点 ({2,3,4})*。数量 = $C_0 \times C_3 = 1 \times 5 = 5$ 棵。
- ***以 2 为根：****左子树有 1 个节点 ({1})，右子树有 2 个节点 ({3,4})*。数量 = $C_1 \times C_2 = 1 \times 2 = 2$ 棵。
- ***以 3 为根：****左子树有 2 个节点 ({1,2})，右子树有 1 个节点 ({4})*。数量 = $C_2 \times C_1 = 2 \times 1 = 2$ 棵。
- ***以 4 为根：****左子树有 3 个节点 ({1,2,3})，右子树有 0 个节点*。数量 = $C_3 \times C_0 = 5 \times 1 = 5$ 棵。

总数量 = $5 + 2 + 2 + 5 = 14$ 棵。

110

> 在以下二叉搜索树中查找元素 14，第 3 个和 14 发生比较的元素为:



Solution 14. 答案是 ***11***。

详细分析： 在二叉搜索树中查找元素，我们遵循以下规则：

- 如果目标值小于当前节点的值，则移动到左孩子。
- 如果目标值大于当前节点的值，则移动到右孩子。
- 如果目标值等于当前节点的值，则查找成功。

我们来追踪查找元素 *14* 的过程：

(1)* 第 *1* 次比较：** 从根节点 ***16 开始。

- 比较 *14* 和 *16*。
- 因为 $14 < 16$，所以向左移动到节点 *10*。

(2)* 第 *2* 次比较：** 当前节点为 ***10。

- 比较 *14* 和 *10*。
- 因为 $14 > 10$，所以向右移动到节点 *11*。

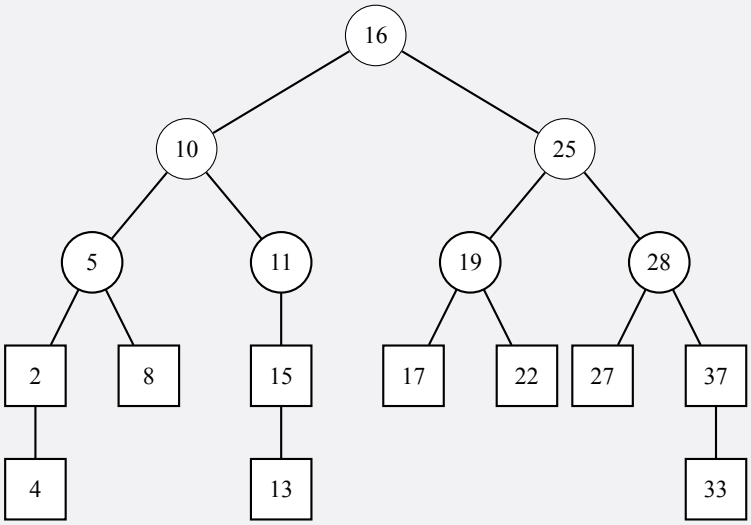
(3)* 第 *3* 次比较：** 当前节点为 ***11。

- 比较 *14* 和 *11*。
- 因为 $14 > 11$，所以继续向右移动。

根据查找路径，第 *3* 个与 *14* 发生比较的元素是 ***11***。

111

> 欲在以下二叉搜索树中删除节点 16，可行的方案是：



A. 将 16 摘除后令 25 为 10 的右子，从而 10 是新的根节点
B. 以 16 为轴进行一次 zig 操作使之不再是根节点，再直接摘除 16
C. 将节点 16 和节点 33 的关键码互换，再摘除新的节点 16
D. 将节点 16 和节点 15 的关键码互换，摘除新的节点 16 并令 13 为 11 的右子

Solution 15. 正确答案是 *D*。

详细分析:

要删除的节点 *16* 是根节点，并且它有两个孩子（度为 *2*）。在二叉搜索树（*BST*）中删除一个度为 *2* 的节点，标准算法如下:

(1) 在该节点的左子树中找到最大的元素（即中序遍历的前驱），或者在其右子树中找到最小的元素（即中序遍历的后继）。

(2) 用找到的这个元素的值替换要删除的节点的值。

(3) 然后，从原位置删除那个被用来替换的节点。由于这个被替换的节点（前驱或后继）最多只有一个孩子，所以删除它会很简单。

我们来分析两种可行的策略:

- **策略 *1*: 使用中序前驱**
  - ***(1)*** 节点 *16* 的中序前驱是其左子树中最大的节点。从左孩子 *10* 开始，一直向右走到底，找到节点 ***15***。
  - ***(2)*** 将节点 *16* 的值替换为 *15*。
  - ***(3)*** 现在问题变为删除原位置的节点 *15*。这个节点（现在值为 *16*）有一个左孩子 *13*。
  - ***(4)*** 要删除它，我们将其父节点 *11* 的右孩子指针指向它的孩子 *13*。即“令 *13* 为 *11* 的右子”。

  这个过程与选项 *D* 描述的完全一致:“将节点 *16* 和节点 *15* 的关键码互换，摘除新的节点 *16* 并令 *13* 为 *11* 的右子”。
- **策略 *2*: 使用中序后继**
  - ***(1)*** 节点 *16* 的中序后继是其右子树中最小的节点。从右孩子 *25* 开始，一直向左走到底，找到节点 ***17***。
  - ***(2)*** 将节点 *16* 的值替换为 *17*。
  - ***(3)*** 现在问题变为删除原位置的节点 *17*。这个节点是叶子节点，可以直接删除。

现在评估所有选项:

- *A:* 这种重构方式会破坏 *BST* 的性质，例如 *19* 会成为 *10* 的后代，但 *19>10*，这不符合规则。

- *B: Zig* 操作是伸展树（*Splay Tree*）中的概念，不是标准 *BST* 的删除方法。
- *C:* 节点 *33* 既不是 *16* 的前驱也不是后继，用它来替换会破坏 *BST* 的有序性。
- *D:* 准确描述了使用中序前驱（节点 *15*）来删除节点 *16* 的标准过程。

因此，选项 *D* 是正确的。

112

> 二叉搜索树的高度 h 和节点个数 n 满足关系 A. h=O(1) B. h=O(log(n)) C. h=O(n) D. h=O(nlog(n))

Solution 16. 正确答案是 *C*。

详细分析:

二叉搜索树（*BST*）的高度 *'h'* 与其节点个数 *'n'* 之间的关系取决于树的平衡程度。我们需要考虑所有可能的情况，特别是最坏情况。

- **最好情况（完全平衡的 *BST*）:** 当二叉搜索树接近完全平衡时，其结构类似于一个完全二叉树。在这种情况下，树的高度 *'h'* 大约是 $\log_2(n)$。因此，在最好情况下，$h = O(\log n)$。
- **最坏情况（退化的 *BST*）:** 当插入的节点序列是完全有序（升序或降序）时，二叉搜索树会退化成一个线性链表。例如，依次插入 *1, 2, 3, ..., n*，会形成一个只有右孩子的链。

```
1
 \
  2
   \
    3
     ...
      \
       n
```

在这种情况下，树的高度为 $n - 1$。因此，在最坏情况下，$h = O(n)$。

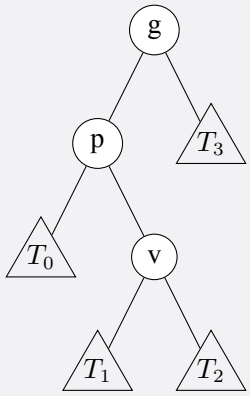
题目询问的是二叉搜索树的一般关系，这必须包含所有情况，尤其是定义其复杂度上界的 ** 最坏情况 **。因此，二叉搜索树的高度 *'h'* 的上界是 $O(n)$。

分析选项:

- *A. h=O(1):* 不正确，高度随节点数增长。
- *B. h=O(log(n)):* 这是平衡二叉搜索树（如 *AVL* 树）的高度，是 *BST* 的最好情况，但不是所有 *BST* 都满足。
- *C. h=O(n):* 这是 *BST* 高度的最坏情况，是正确描述其复杂度上界的选项。
- *D. h=O(nlog(n)):* 不正确，这个复杂度与树的高度无关。

113

> 对以下子树进行 3+4 重构，得到的子树为:



Solution 17. 详细分析:

*"3+4 重构"*是处理 *AVL* 树失衡的一种统一方法。它涉及三个关键节点和四个子树。

***(1)* 识别*"3+4"*:**

- **三个节点:** 从最低的失衡节点开始，向上追溯三代：失衡节点 *'g' (grandparent)*，其较高的孩子 *'p' (parent)*，以及 *'p'* 的较高的孩子 *'v' (vulnerable)*。
- **四个子树:**这三个节点将整棵子树分割成四个部分,我们按中序遍历的顺序列出它们:T_0, T_1, T_2, T_3。

根据题目给出的图，我们可以识别出:

- 三个节点是: *'g', 'p', 'v'*。
- 四个子树是: *'p'*的左子树 T_0，*'v'*的左子树 T_1，*'v'*的右子树 T_2，以及 *'g'*的右子树 T_3。

(2)* 确定中序次序:** 对这三个节点进行中序排序。根据二叉搜索树的性质，中序遍历序列为: $...T_0...$ ***p $...T_1...$ ***v*** $...T_2...$ ***g*** $...T_3...$ 因此，这三个节点的有序序列是 *'p', 'v', 'g'*。

***(3)* 进行重构:**

- 将有序序列中的中间节点 *'v'* 提升为新的根。
- 将序列中的第一个节点 *'p'* 作为新根 *'v'* 的左孩子。
- 将序列中的第三个节点 *'g'* 作为新根 *'v'* 的右孩子。
- 将四个子树 T_0, T_1, T_2, T_3 按照中序次序重新链接到 *'p'* 和 *'g'* 上。
  - T_0 链接为 *'p'* 的左孩子。
  - T_1 链接为 *'p'* 的右孩子。
  - T_2 链接为 *'g'* 的左孩子。
  - T_3 链接为 *'g'* 的右孩子。

重构后的子树结构为: