第7章

搜索树

[7-1] 试证明，一棵二叉树是二叉搜索树，当且仅当其中序遍历序列单调非降。

【解答】

考查二叉搜索树中的任一节点r。按照中序遍历的约定，r左（右）子树中的节点（若存在）均应先于（后于）r接受访问。

按照二叉搜索树的定义，r左（右）子树中的节点（若存在）均不大于（不小于）r，故中序遍历序列必然在r处单调非降；反之亦然。

鉴于以上所取r的任意性，题中命题应在二叉搜索树中处处成立。

由此题亦可看出，二叉搜索树的定义不能更改为“任意节点r的左（右）孩子（若存在）均不大于（不小于）r”——相当于将原定义中的“左（右）后代”，替换为“左（右）孩子”。为强化印象，读者不妨构造一个符合这一“定义”，但却不是二叉搜索树的具体实例。

[7-2] 试证明，由一组共 n 个互异节点组成的二叉搜索树，总共有(2n)!/n!/(n + 1)!棵。

【解答】

我们将n个互异节点所能组成二叉搜索树的总数，记作T(n)。

由上题结论，尽管由同一组节点组成的二叉搜索树不尽相同，但它们的中序遍历序列却必然相同，不妨记作：

$$\boxed{x_0 \quad x_1 \quad x_2 \quad \ldots \quad x_{k-1}} \quad x_k \quad \boxed{x_{k+1} \quad x_{k+2} \quad \ldots \quad x_{n-1}}$$

根据所取树根节点的不同，所有搜索树可以分为n类。如上所示，对于其中以x_k为根者而言，左、右子树必然分别由$\{ x_0, x_1, x_2, \ldots, x_{k-1} \}$和$\{ x_{k+1}, x_{k+2}, \ldots, x_{n-1} \}$组成。

如此，可得边界条件和递推式如下：

$$T(0) = T(1) = 1$$

$$T(n) = \sum_{k=0}^{n-1} T(k) \cdot T(n-k-1)$$

这是典型的Catalan数式递推关系，解之即得题中结论。

[7-3] 试证明，含 n 个节点的二叉树的最小高度为$\lfloor \log_2 n \rfloor$——这也是由 n 个节点组成的完全二叉树高。

【解答】

实际上不难证明，若高度为h的二叉树共含n个节点，则必有：

$$n \leq 2^{h+1} - 1$$

这里的等号成立，当且仅当是满树。于是有：

$$h \geq \log_2(n + 1) - 1$$

$$h \geq \lceil \log_2(n + 1) \rceil - 1 = \lfloor \log_2 n \rfloor$$

[7-4] 与其它算法类似，searchIn()算法的递归版（教材 186 页代码 7.3）也存在效率低下的问题。试将该算法改写为迭代形式。请注意保持出口时返回值和 hot 的语义。

【解答】

只要注意到该算法的递归形式接近于尾递归，即可实现其迭代版如代码x7.1所示。

```
#define EQUAL(e, v)  (!(v) || (e) == (v)->data) //节点v（或假想的通配哨兵）的关键码等于e
template <typename T> //在以v为根的（AVL、SPLAY、rbTree等）BST子树中查找关键码e
static BinNodePosi(T) & searchIn ( BinNodePosi(T) & v, const T& e, BinNodePosi(T) & hot ) {
   if ( EQUAL ( e, v ) ) return v; hot = v; //退化情况：在子树根节点v处命中
   while ( 1 ) { //一般地，反复不断地
      BinNodePosi(T) & c = ( e < hot->data ) ? hot->lc : hot->rc; //确定深入方向
      if ( EQUAL ( e, c ) ) return c; hot = c; //命中返回，或者深入一层
   } //hot始终指向最后一个失败节点
} //返回时，返回值指向命中节点（或假想的通配哨兵），hot指向其父亲（退化时为初始值NULL）
```

代码x7.1 二叉搜索树searchIn()算法的迭代实现

不难验证，该迭代版出口时返回值和hot的语义，与递归版完全一致。

[7-5] 试证明，采用 BST::insert()算法（教材 188 页代码 7.5），在二叉搜索树中插入节点 v 之后

a） 除 v 的历代祖先以外，其余节点的高度无需更新；

【解答】

我们知道，节点的高度仅取决于其后代——更确切地，等于该节点与其最深后代之间的距离。因此在插入节点v之后，节点a的高度可能发生变化（增加），当且仅当v是a的后代，或反过来等价地，a是v的祖先。

b） 祖先高度不会降低，但至多加一；

【解答】

插入节点v之后，所有节点的后代集不致缩小。而正如前述，高度取决于后代深度的最大值，故不致下降。

另一方面，假定节点a的高度由h增加至h'。若将v的父节点记作p，则a到p的距离不大于a在此之前的高度h，于是必有：

$$h' \leq |ap| + 1 \leq h + 1$$

c） 一旦某个祖先高度不变，更高的祖先也必然高度不变。

【解答】

对于任意节点p，若将其左、右孩子分别记作l和r（可能是空），则必有：

$$height(p) = 1 + max(height(l), height(r))$$

在插入节点v之后，在l和r之间，至多其一可能会（作为v的祖先而）有所变化。一旦该节点的高度不变，p以及更高层祖先（如果存在的话）的高度亦保持不变。

[7-6] 试证明，采用 BST::remove()算法（教材 190 页代码 7.6）从二叉搜索树中删除节点，若实际被删除的节点为 x，则此后：

a） 除 x 的历代祖先以外，其余节点的高度无需更新；

【解答】

同样地，节点的高度仅取决于其后代——更确切地，等于该节点与其最深后代之间的距离。因此在删除节点x之后，节点a的高度可能发生变化（下降），当且仅当x是a的后代，或反过来等价地，a是x的祖先。

b） 祖先高度不会增加，但至多减一；

【解答】

假设在删除节点x之后，祖先节点a的高度由h变化为h'。现在，我们假想式地将x重新插回树中，于是自然地，a的高度应该从h'恢复至h。由[7-5]题的结论b)，必有：

$$h \le h' + 1$$

亦即：

$$h' \ge h - 1$$

c） 一旦某个祖先高度不变，更高的祖先也必然高度不变。

【解答】

反证，假设在删除节点x之后，祖先节点的高度会间隔地下降和不变。

仿照上一问的思路，假想着将x重新插回树中。于是，所有节点的高度均应复原，而祖先节点的高度则必然会间隔地上升和不变。这与[7-5]题的结论c)相悖。

[7-7] 利用以上事实，进一步改进 updateHeightAbove()方法，提高效率。

【解答】

在逐层上行依次更新祖先高度的过程中，一旦某一祖先的高度不变，便可随即终止。

当然，就最坏情况而言，依然必须更新至树根节点。

[7-8] a） 试按照随机生成和随机组成两种方式，分别进行实际测试，并统计出二叉搜索树的平均高度；

b） 你得到的统计结果，与 7.3.1 节所给的结论是否相符？

【解答】

请读者按照教材中对这两种方式的定义，以及相关的介绍，独立完成编码、调试和实测任务，并根据统计结果给出结论和分析。

[7-9] BinTree::removeAt()算法（教材 190 页代码 7.7）的执行过程中，当目标节点同时拥有左、右孩子时，总是固定地选取直接后继与之交换。于是，从二叉搜索树的整个生命期来看，左子树将越来越倾向于高于右子树，从而加剧整体的不平衡性。

一种简捷且行之有效的改进策略是，除直接后继外还同时考虑直接前驱，并在二者之间随机选取。

a） 试基于习题[5-14]扩展的 pred()接口，实现这一策略；

【解答】

针对这一问题，实现随机选取的一种简明方法是：

调用rand()取（伪）随机数，根据其奇偶，相应地调用succ()或pred()接口

从理论上讲，如此可以保证各有50%的概率使用直接后继或直接前驱，从而在很大程度上消除题中指出的“天然”不均衡性。

BinTree::removeAt()算法的其余部分，无需任何修改。

b） 通过实测统计采用新策略之后的平均树高，并与原策略做一对比。

【解答】

请读者按照以上介绍，独立完成编码、调试和实测任务，并根据统计结果给出结论和分析。

[7-10] 为使二叉搜索树结构支持多个相等数据项的并存，需要增加一个 BST::searchAll(e)接口，以查找出与指定目标 e 相等的所有节点（如果的确存在）。

a） 试在 BST 模板类（教材 185 页代码 7.2）的基础上，扩充接口 BST::searchAll(e)。

要求该接口的时间复杂度不超过 $\mathcal{O}$(k + h)，其中 h 为二叉搜索树的高度，k 为命中节点的总数；

【解答】

从后面第8.4.1节所介绍范围查询的角度来看，从二叉搜索树中找出所有数值等于e的节点，完全等效于针对区间(e - ε, e + ε)的范围查找，其中ε为某一足够小的正数。

因此，自然可以套用第8.4.1节所给的算法框架：针对e - ε和e + ε各做一次查找，并确定查找路径终点的最低公共祖先；在从公共祖先通往这两个终点的路径上，自上而下地根据各层的分支方向，相应地忽略整个分支，或者将整个分支悉数报告出来。

整个算法所拣出的分支，在每一层不超过两个，故总共不会超过$\mathcal{O}$(h)个。借助（任何一种常规的）遍历算法，都可在线性时间内枚举出每个分支中的所有节点；而对所有分支的遍历，累计耗时亦不过$\mathcal{O}$(k)。

需要特别说明的是，这里既不便于也不需要显式地确定ε的具体数值。实际上，我们只需要对比较器做适当的调整：针对e - ε（e + ε）的查找过程，与针对e的查找过程基本相同，只是在遇到数值为e的节点时，统一约定向左（右）侧深入。

b） 同时，改进原有的 BST::search(e)接口，使之总是返回最早插入的节点 e——即先进先出。

【解答】

在中序遍历序列中，所有数值为e的雷同节点，必然依次紧邻地构成一个区间。为实现“先进先出”的规范，需要进一步地要求它们在此区间内按插入次序排列。

为此可以统一约定：在BST::insert(e)内的查找定位过程中，凡遇到数值相同的节点，均优先向右侧深入；而在BST::search(e)的查找过程中，凡遇到数值相同的节点，均向左侧深入。

当然，将以上约定的左、右次序颠倒过来，亦同样可行。

[7-11] 考查包含 n 个互异节点的二叉搜索树。

试证明，无论树的具体形态如何，BST::search()必然恰有 n 种成功情况和 n + 1 种失败情况。

【解答】

通过对树高做数学归纳，不难证明。请读者独立完成这一任务。

[7-12] 试证明，在高度为 h 的 AVL 树中，任一叶节点的深度均不小于⌈h/2⌉。

【解答】

对树高h做数学归纳。作为归纳基，h = 1时的情况显然。假设以上命题对高度小于h的AVL树均成立，以下考查高度为h的AVL树。

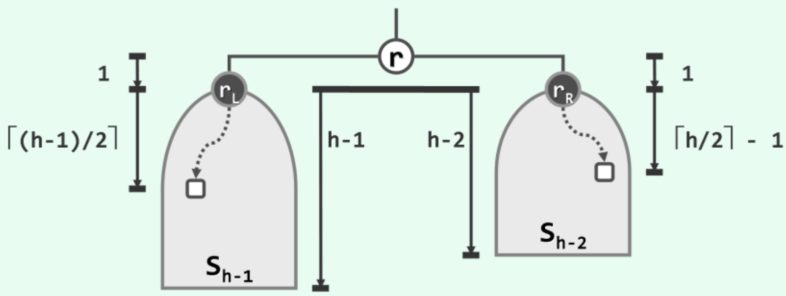
根据AVL树的性质，如图x7.1所示，此时左、右子树的高度至多为h - 1，至少为h - 2。

由归纳假设，在高度为h - 1的子树内部，叶节点深度不小于：

$$\lceil(h - 1)/2\rceil \geq \lceil h/2\rceil - 1$$

而在高度为h - 2的子树内部，叶节点深度也不小于：

$$\lceil h/2\rceil - 1$$



图x7.1 AVL树中最浅的叶节点

因此在全树中，任何叶节点深度都不致小于：

$$1 + (\lceil h/2\rceil - 1) = \lceil h/2\rceil$$

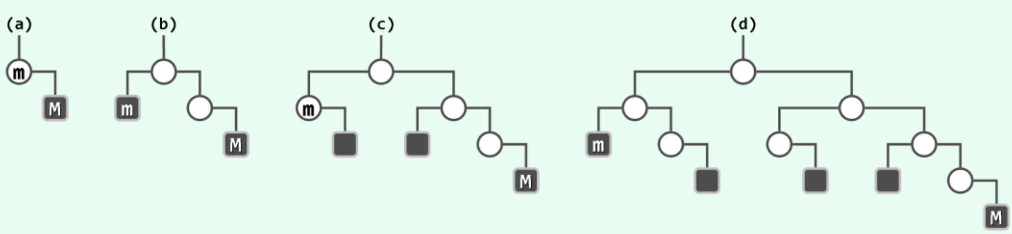
[7-13] 试证明：

a） 按照二叉搜索树的基本算法在 AVL 树中引入一个节点后，失衡的节点可能多达Ω(logn)个；

【解答】

首先，引入一类特殊的AVL树，它们符合以下条件：其中每个内部节点的左子树，都比右子树在高度上少一。这也就是所谓的Fib-AVL树（Fibonaccian AVL tree）。

如图x7.2(a~d)所示，即为高度分别为1、2、3和4的Fib-AVL树。通过数学归纳法不难证明，此类AVL树的高度若为h，则其规模必然是fib(h + 3) - 1，故此得名。实际上，Fib-AVL树也是在高度固定的前提下，节点总数最少的AVL树。



图x7.2 Fib-AVL树

考查其中数值最大（中序遍历序列中最靠后）的节点M。该节点共计h个祖先，而且它们的平衡因子均为-1。现在，假设需要将一个词条插入其中，而且该词条大于节点M。

按照二叉搜索树的插入算法，必然会相应地在节点M之下，新建一个右孩子x。此时，节点M所有祖先的平衡因子都会更新为-2，从而出现失衡现象。失衡节点的总数为：

$$h = fib^{-1}(n + 1) - 3 = \log_\Phi n = \mathcal{O}(\log n)$$

其中，

$$\Phi = (\sqrt{5} + 1) / 2 = 1.618$$

b） 按照二叉搜索树的基本算法从 AVL 树中摘除一个节点后，失衡的节点至多 1 个。

【解答】

请注意，节点的失衡与否，取决于其左、右子树高度之差。因此反过来，只要子树的高度不变，则节点不可能失衡。

在删除节点之后自底而上逐层核对平衡因子的过程中，一旦遇到一个失衡节点v，则被删除节点必然来自v原本更低的一棵子树，而v的高度必然由其另一更高的子树确定，故v的高度必然保持不变。由以上分析结论，除了v本身，其祖先节点必然不可能失衡。

[7-14] 按照教材第 7.3.4 节的定义和描述，实现节点旋转调整算法 zig()和 zag()。

【解答】

请读者对照教材193页图7.11，以及193页图7.12，独立完成编码和调试任务。

[7-15] 试证明：

a） 规模为 n 的任何二叉搜索树，经过不超过 n - 1 次旋转调整，都可等价变换为仅含左分支的二叉搜索树，即最左侧通路（leftmost path）；

【解答】

可以设计一个具体的算法，以完成这一等价变换。为此，需要回顾迭代式先序遍历算法的版本#2（教材127页代码5.14），并对该算法的流程略作改动。

具体地如教材127页图5.18所示，考查二叉搜索树的最左侧通路。从该通路的末端节点L_d开始，我们将逐步迭代地延长该路径，直至不能继续延长。每次迭代，无非两种情况：

其一，若L_k的右子树为空，则可令L_k上移一层，转至其父节点。

其二，若L_k的右孩子R_k存在，则可以L_k为轴，做一次zag旋转调整。如此，R_k将（作为L_k的父亲）纳入最左侧通路中。

不难看出，整个迭代过程的不变性为：

1）当前节点L_k来自最左侧通路
2）L_k的左子树（由不大于L_k的所有节点组成）已不含任何右向分支

另外，整个迭代过程也满足如下单调性：

最左侧通路的长度，严格单调地增加

故该算法必然终止，且最终所得的二叉搜索树不再含有任何右向分支。

以上思路，可具体实现如代码x7.2所示。

```
//通过zag旋转调整，将子树x拉伸成最左侧通路
template <typename T> void stretchByZag ( BinNodePosi(T) & x ) {
   int h = 0;
   BinNodePosi(T) p = x; while ( p->rc ) p = p->rc; //最大节点，必是子树最终的根
   while ( x->lc ) x = x->lc; x->height = h++; //转至初始最左侧通路的末端
   for ( ; x != p; x = x->parent, x->height = h++ ) { //若x右子树已空，则上升一层
      while ( x->rc ) //否则，反复地
         x->zag(); //以x为轴做zag旋转
   } //直到抵达子树的根
}
```

代码x7.2 将任意一棵二叉搜索树等价变换为单分支列表

可见，每做一次zag旋转调整，总有一个节点归入最左侧通路中，后者的长度也同时加一。最坏情况下，除原根节点外，其余节点均各自对应于一次旋转，累计不过n - 1次。

通过进一步的观察不难看出：

任一节点需要通过一次旋转归入最左侧通路，当且仅当它最初不在最左侧通路上

故若原最左侧通路的长度为s，则上述算法所做的旋转调整，恰好共计n - s - 1次。

其中特别地，s = 0（根节点的左子树为空），当且仅当需做n - 1次旋转——这也是最坏情况的充要条件。

b） 规模为 n 的任何两棵等价二叉搜索树，至多经过 2n - 2 次旋转调整，即可彼此转换。

【解答】

既然每棵二叉搜索树经过至多n - 1次旋转调整，总能等价变换为最左侧通路，故反之亦然。因此，对于任何两棵二叉搜索树，都可按照上述方法，经至多n - 1次旋转调整，先将其一等价变换为最左侧通路；然后同理，可再经至多n - 1次旋转调整，从最左侧通路等价变换至另一棵二叉搜索树。

[7-16] 为使 AVL 树结构支持多个相等数据项的并存，需要增加一个 AVL::searchAll(e)接口，以查找出与指定目标 e 相等的所有节点（如果的确存在）。

a） 试在如 194 页代码 7.8 所示 AVL 模板类的基础上扩充接口 AVL::searchAll(e)，要求其时间复杂度不得超过 $\mathcal{O}$(k + logn)，其中 n 为 AVL 树的规模，k 为命中节点的总数；

【解答】

原理及方法均与习题[7-10]完全相同。

性能方面，通过遍历枚举所有命中子树中的节点，仍可以在线性的$\mathcal{O}$(k)时间内完成；因为AVL树可以保持适度平衡，故所涉及的查找可以更快完成，累计耗时不超过$\mathcal{O}$(logn)。

b） 同时，改进原有的 AVL::search(e)接口，使之总是返回最早插入的节点 e——即先进先出。

【解答】

原理及方法均与习题[7-10]基本相同。

需要强调的是，尽管在插入或删除操作的过程中，可能会做旋转以重新平衡，但因这些都属于等价变换，（包括雷同节点在内的）所有节点的中序遍历序列始终保持不变，每一组雷同节点都始终依照插入次序排列。

[7-17] 试证明，对于任意大的正整数 n，都存在一棵规模为 n 的 AVL 树，从中删除某一特定节点之后，的确需要做$\Omega(\log n)$次旋转，方能使全树恢复平衡。

【解答】

首先，考查习题[7-13]所引入的Fib-AVL树。

如150页的图x7.2(a~d)所示，若从该树中删除最小的节点（亦即中序遍历序列中的首节点）m，则首先会导致m的父节点p失衡。在树高h为奇数时，m虽不是叶节点，但按照二叉搜索树的删除算法，在实际摘除m之前，必然已经将m与其直接后继（此时亦即其右孩子）交换，从而等效于删除其右孩子。

不难验证，在父节点p恢复平衡之后，其高度必然减一，从而造成m祖父节点g的失衡。同样地，尽管节点g可以恢复平衡，但其高度必然减一，从而造成更高层祖先的失衡。这种现象，可以一直传播至树根。

仿照习题[7-12]的分析方法不难证明，在高度为h的Fib-AVL树中，节点m的深度为$\lfloor h/2 \rfloor$。因此，上述重平衡过程所涉及的节点旋转次数应不少于：

$$
\begin{aligned}
\lfloor h/2 \rfloor &= \lfloor (fib^{-1}(n + 1) - 3)/2 \rfloor \\
&= \log_{\Phi} n / 2 \\
&= \Omega(\log n)
\end{aligned}
$$

其中，

$$\Phi = (\sqrt{5} + 1) / 2 = 1.618$$

实际上，只需对以上Fib-AVL树的结构做进一步的调整，完全可以使得每个节点的重平衡都属于双旋形式，从而使得总体的旋转次数加倍至：

$$\lfloor h/2 \rfloor \cdot 2 \approx h$$

当然，从渐进的角度看，以上结论并未有实质的改进。

请读者参照以上思路，独立给出具体的调整方法。

尽管以上方法仅适用于规模为$n = fib(h + 3) - 1$的AVL树，但其原理及方法并不难推广至一般性的n。

[7-18] D. E. Knuth[3]曾指出，AVL::remove()操作尽管在最坏情况下需做Ω(logn)次旋转，但平均而言仅需做 0.21 次。试通过实验统计，验证这一结论。

【解答】

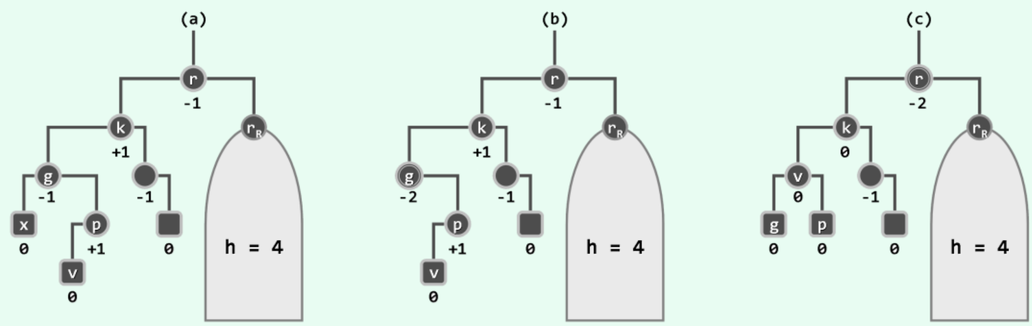
请读者独立完成测试和统计，并结合实测结果给出结论及分析。

[7-19] 设在从 AVL 树中摘除一个节点之后，刚刚通过调整使 g(x)重新恢复了平衡。此时，若发现 g(x)原先的父节点依然平衡，则是否可以不必继续检查其更高层的祖先，并随即停止上溯？也就是说，此时在更高层是否依然可能有失衡的祖先？若是，请说明理由；否则，试举一反例。

【解答】

实际上，此时若停止上溯，则有可能会遗漏更高层的失衡祖先节点——AVL树节点删除操作的这一性质，与节点插入操作完全不同。

考查如图x7.3(a)所示的实例，只需注意逐一核对各节点的平衡因子，不难验证这的确是一棵AVL树，且高度为5。其中，左子树高度为3，右子树高度为4，但鉴于其具体结构组成无所谓，故未予详细绘出。



图x7.3 从AVL-树中删除节点之后，需要重平衡的祖先未必相邻

现在，若从中删除节点x，则首先按照二叉搜索树的算法，将其直接摘除。此时应如图(b)所示，全树唯一的失衡节点只有g。于是接下来按照AVL-树的重平衡算法，经双旋调整即可恢复这一局部的平衡。

此时，考查g原先的父节点k。如图(c)所示，尽管节点k的平衡因子由+1降至0，却依然不失平衡。然而，自底而上的调整过程不能就此终止。我们注意到，此时节点k的高度已由3降至2，于是对于更高层的祖先节点r而言，平衡因子由-1进一步降至-2，从而导致失衡。

由上可见，仅仅通过平衡性，并不足以确定可否及时终止自底而上的重平衡过程。然而，并非没有办法实现这种优化。实际上，只要转而通过核对重平衡后节点的高度，即可及时判定是否可以立即终止上溯过程。请读者按照这一提示和思路，独立给出改进的方法。

由此反观AVL-树的插入操作，之所以能够在首次重平衡之后随即终止上溯，原因在于此时不仅局部子树的平衡性能够恢复，而且局部子树的高度亦必然同时恢复。

[7-20] 试证明，按递增次序将 2^{h+1} - 1 个关键码插入初始为空的 AVL 树中，必然得到高度为 h 的满树。

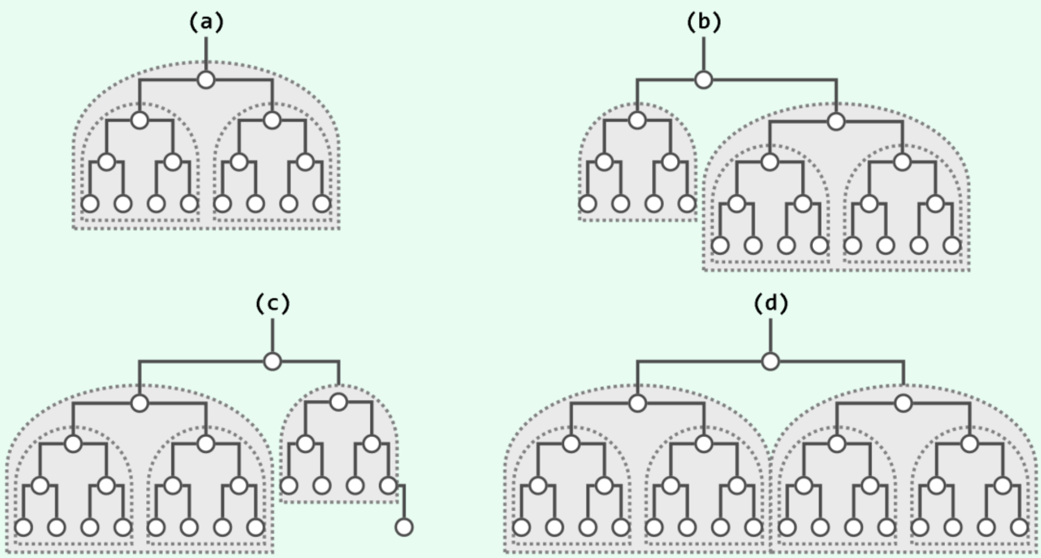
【解答】

首先，考察AVL的右侧分支。对照AVL树的重平衡算法不难发现，在这样的插入过程中，该分支上沿途上各节点v始终满足以下不变性：

1) v的左子树必为满树；
2) height(rc(v)) - 1 ≤ height(lc(v)) ≤ height(rc(v))

实际上，在这一系列的插入操作过程中出现的每一次失衡，都可以通过zag单旋予以修复。如教材196页图7.15(a)所示，若T_0、T_1和T_2都是满树，则旋转之后应如图(b)所示，节点g与T_0和T_1必然也构成一棵（增高一层的）满树。

为更加细致地展示这一演变过程并证明以上结论，以下不妨对树高做数学归纳。作为归纳基，以上命题自然对高度为0（单节点）的AVL树成立。假设以上命题对高度不超过h的AVL树均成立，现考查高度为h + 1的情况。



图x7.4 将31个关键码按单调次序插入，必然得到一棵高度为4的满树

如图x7.4所示，我们不妨将关键码[0, 2^{h+2} - 1)的插入过程，分为四个阶段：

a） 首先插入关键码[0, 2^{h+1} - 1)

由归纳假设，应得到一棵高度为h的满树。

以h = 3为例，在将关键码[0, 15)依次插入初始为空的AVL树后，应如图(a)所示，得到一棵高度为3、规模为15的满树。

b） 继续插入关键码[2^{h+1} - 1, $3\cdot2^h$ - 1)

这一阶段的插入对树根的左子树没有影响，其效果等同于将这些关键码单调地插入右子树。因此亦由归纳假设，右子树必然成为一棵高度为h的满树。

继续以上实例。在接下来依次插入关键码[15, 22)之后，该AVL树应如图(b)所示，根节点的左子树与右子树分别是一棵高度为2和3的满树。

c） 再插入关键码$[3\cdot2^h - 1]$

如此，必将引起树根节点的失衡，并在以根为轴做zag单旋之后恢复平衡。此后，根节点的左子树是高度为h的满树；右子树高度亦为h，但最底层只有一个关键码——新插入的$[3\cdot2^h - 1]$。

仍然继续上例。在接下来再插入关键码[23]之后，该AVL树应如图(c)所示，根节点的左子树是一棵高度为3的满树；右子树高度亦为3，但最底层仅有一个关键码[23]。

d） 最后，插入关键码$[3\cdot2^h,\ 2^{h+2} - 1)$

同样地，这些关键码的插入并不影响树根的左子树，其效果等同于将这些关键码单调地插入右子树。故由归纳假设，右子树必然成为一棵高度为h的满树。至此，整体得到一棵高度为h + 1的满树。

仍然继续上例。在接下来再插入关键码[24, 32)之后，该AVL树应如图(d)所示，根节点的左子树和右子树都是高度为3的满树，整体构成一棵高度为4的满树。