数据结构与算法（5）：AVL树

我们知道，对于一般的二叉搜索树（Binary Search Tree），其期望高度（即为一棵平衡树时）为log_2n，其各操作的时间复杂度$O(log_2n)$同时也由此而决定。但是，在某些极端的情况下（如在插入的序列是有序的时），二叉搜索树将退化成近似链或链，此时，其操作的时间复杂度将退化成线性的，即O(n)。我们可以通过随机化建立二叉搜索树来尽量的避免这种情况，但是在进行了多次的操作之后，由于在删除时，我们总是选择将待删除节点的后继代替它本身，这样就会造成总是右边的节点数目减少，以至于树向左偏沉。这同时也会造成树的平衡性受到破坏，提高它的操作的时间复杂度。于是就有了我们下边介绍的平衡二叉树。

一、平衡二叉树

平衡二叉树定义：平衡二叉树（Balanced Binary Tree）又被称为AVL树（有别于AVL算法），且具有以下性质：它是一 棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。平衡二叉树的常用算法有红黑树、AVL树等。在平衡二叉搜索树中，我们可以看到，其高度一般都良好地维持在O(log2n)，大大降低了操作的时间复杂度。

其严格定义为：
一颗空树是平衡二叉树；若T是一棵非空二叉树，其左、右子树为TL和TR，令hl和hr分别为左、右子树的深度。当且仅当

- TL、TR都是平衡二叉树；
- 并且满足公式$|hl - hr| \leq 1$时，则T是平衡二叉树

相应地，定义$hl - hr$为二叉平衡树的平衡因子（balance factor）。因此，平衡二叉树上所有结点的平衡因子可能是-1，0，1。换言之，若一颗二叉树上任一结点的平衡因子的绝对值都不大于1，则该树就是平衡二叉树。

最小二叉平衡树的节点的公式如下：$F(n) = F(n-1) + F(n-2) + 1$

这个类似于一个递归的数列，可以参考Fibonacci数列，1是根节点，$F(n-1)$是左子树的节点数量，$F(n-2)$是右子树的节点数量。

二、平衡查找树（AVL树）

2.1 AVL树的定义

平衡二叉树（AVL树，发明者的姓名缩写）：一种高度平衡的排序二叉树，其每一个节点的左子树和右子树的高度差最多等于1。平衡二叉树首先必须是一棵二叉排序树！

平衡因子（Balance Factor）：将二叉树上节点的左子树深度减去右子树深度的值。对于平衡二叉树所有包括分支节点和叶节点的平衡因子只可能是-1,0和1，只要有一个节点的因子不在这三个值之内，该二叉树就是不平衡的。

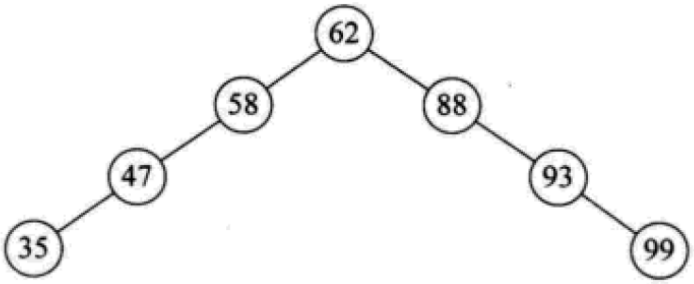


图1 平衡二叉树

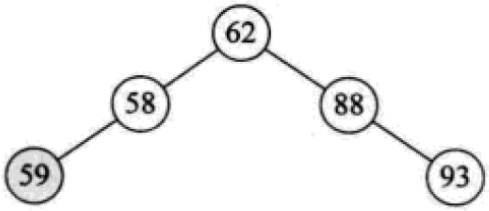


图2 不是平衡二叉树

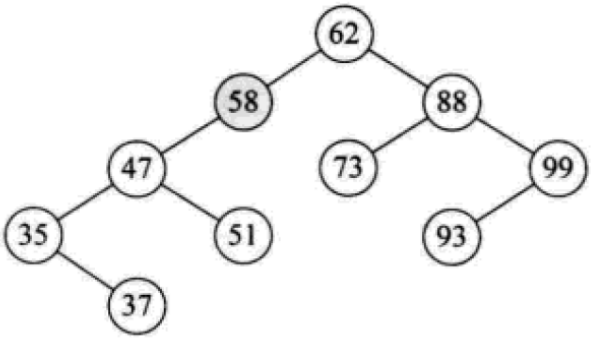


图3 不是平衡二叉树

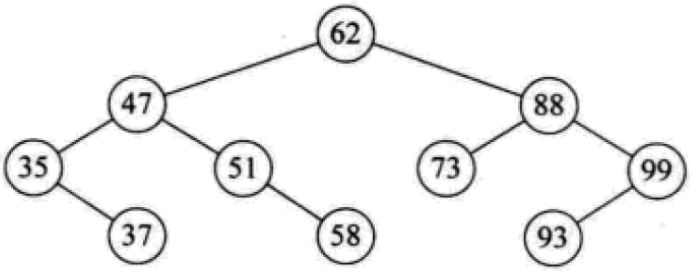


图4 平衡二叉树

最小不平衡子树：距离插入结点最近的，且平衡因子的绝对值大于1的节点为根的子树。

n个结点的AVL树最大深度约1.44log2n。查找、插入和删除在平均和最坏情况下都是O(logn)。增加和删除可能需要通过一次或多次树旋转来重新平衡这个树。这个方案很好的解决了二叉查找树退化成链表的问题，把插入，查找，删除的时间复杂度最好情况和最坏情况都维持在O(logN)。但是频繁旋转会使插入和删除牺牲掉O(logN)左右的时间，不过相对二叉查找树来说，时间上稳定了很多。

可以采用动态平衡技术保持一个平衡二叉树。构造平衡二叉树的时候，也可以采用相同的方法，默认初始时，是一个空树，插入节点时，通过动态平衡技术对二叉树进行调整。

Adeleon-Velskii和Landis提出了一个动态地保持二叉排序树平衡的方法，其基本思想是：在构造二叉排序树的过程中，每当插入一个结点时，首先检查是否因插入而破坏了树的平衡性，若是因插入结点而破坏了树的平衡性，则找出其中最小不平衡树，在保持排序树特性的前提下，调整最小不平衡子树各结点之间的连接关系，以达到新的平衡。通常这样得到的平衡二叉排序树简称为AVL树。

2.2 AVL树的自平衡操作——旋转

AVL树最关键的也是最难的一步操作就是旋转。旋转主要是为了实现AVL树在实施了插入和删除操作以后，树重新回到平衡的方法。下面我们重点研究一下AVL树的旋转。

2.2.1 不平衡的四种情况

对于一个平衡的节点，由于任意节点最多有两个儿子，因此高度不平衡时，此节点的两颗子树的高度差2。容易看出，这种不平衡出现在下面四种情况：

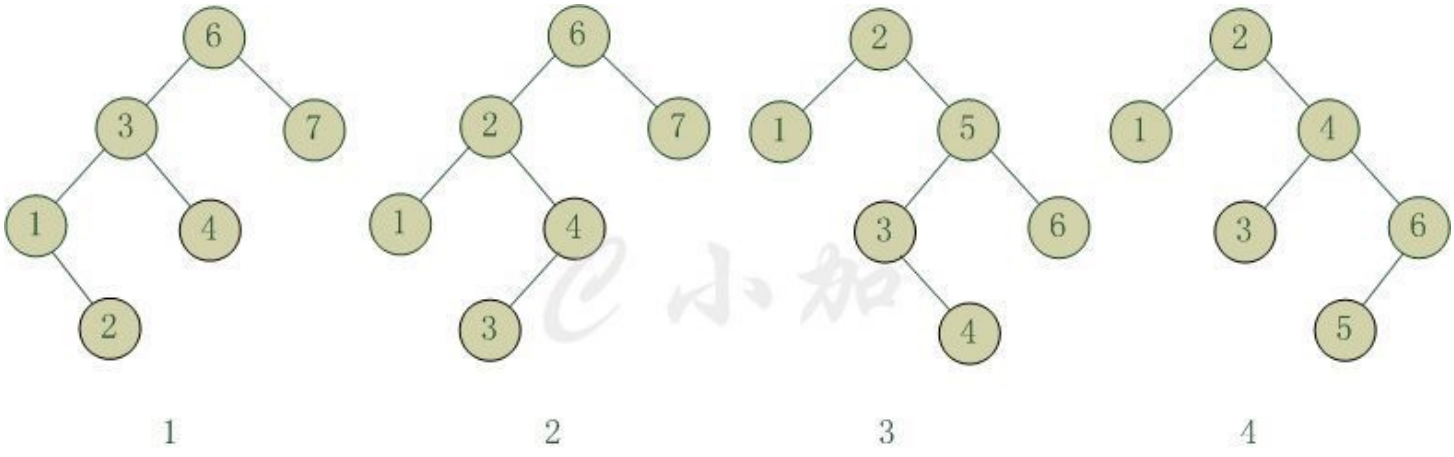


图2 四种不平衡的情况

- 1) 6节点的左子树3节点高度比右子树7节点大2，左子树3节点的左子树1节点高度大于右子树4节点，这种情况成为左左。
- 2) 6节点的左子树2节点高度比右子树7节点大2，左子树2节点的左子树1节点高度小于右子树4节点，这种情况成为左右。
- 3) 2节点的左子树1节点高度比右子树5节点小2，右子树5节点的左子树3节点高度大于右子树6节点，这种情况成为右左。
- 4) 2节点的左子树1节点高度比右子树4节点小2，右子树4节点的左子树3节点高度小于右子树6节点，这种情况成为右右。

从图2中可以可以看出，1和4两种情况是对称的，这两种情况的旋转算法是一致的，只需要经过一次旋转就可以达到目标，我们称之为单旋转。2和3两种情况也是对称的，这两种情况的旋转算法也是一致的，需要进行两次旋转，我们称之为双旋转。

2.2.2 单旋转

单旋转是针对于左左和右右这两种情况的解决方案，这两种情况是对称的，只要解决了左左这种情况，右右就很好办了。图3是左左情况的解决方案，节点k2不满足平衡特性，因为它的左子树k1比右子树Z深2层，而且k1子树中，更深的一层的是k1的左子树X子树，所以属于左左情况。

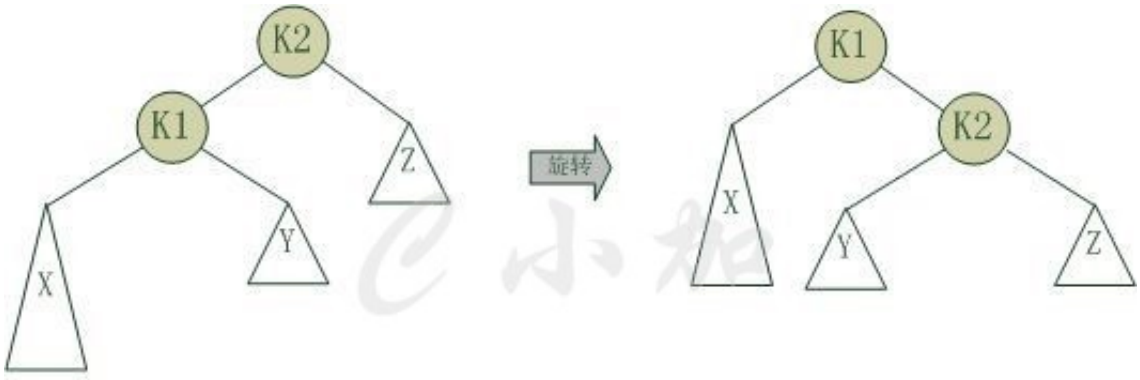


图3 左左情况下单旋转的过程

为使树恢复平衡，我们把k2变成这棵树的根节点，因为k2大于k1，把k2置于k1的右子树上，而原本在k1右子树的Y大于k1，小于k2，就把Y置于k2的左子树上，这样既满足了二叉查找树的性质，又满足了平衡二叉树的性质。

这样的操作只需要一部分指针改变，结果我们得到另外一颗二叉查找树，它是一棵AVL树，因为X向上一移动了一层，Y还停留在原来的层面上，Z向下移动了一层。整棵树的新高度和之前没有在左子树上插入的高度相同，插入操作使得X高度长高了。因此，由于这颗子树高度没有变化，所以通往根节点的路径就不需要继续旋转了。

2.2.3 双旋转

双旋转：对于左右和右左这两种情况，单旋转不能使它达到一个平衡状态，要经过两次旋转。双旋转是针对于这两种情况的解决方案，同样的，这样两种情况也是对称的，只要解决了左右这种情况，右左就很好办了。图4是左右情况的解决方案，节点k3不满足平衡特性，因为它的左子树k1比右子树Z深2层，而且k1子树中，更深的一层的是k1的右子树k2子树，所以属于左右情况。

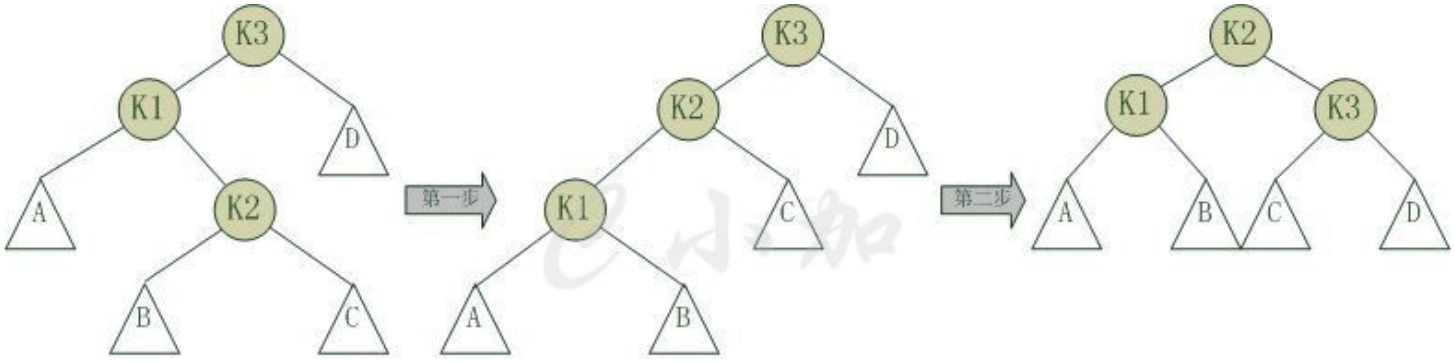
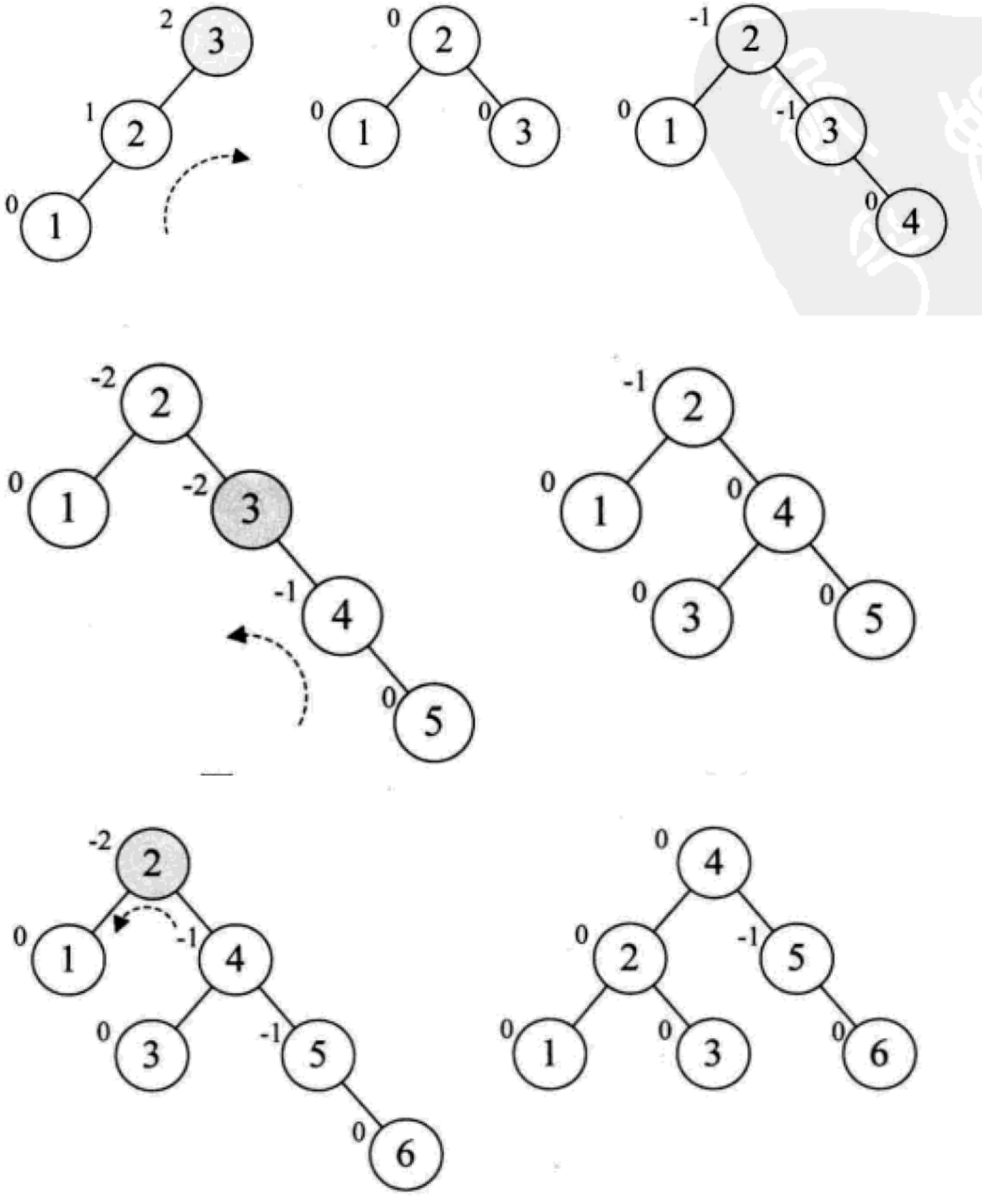


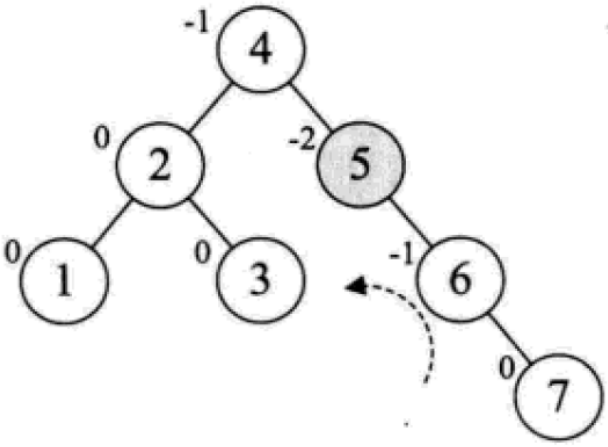
图4 左右情况下双旋转的过程

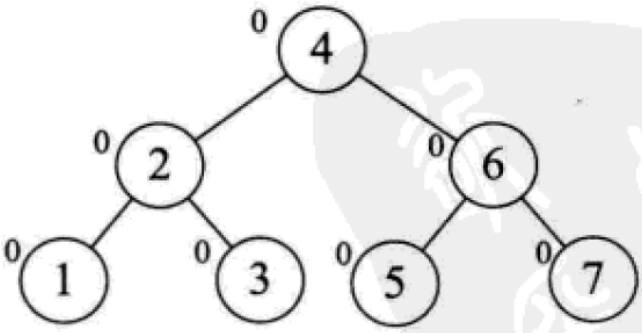
为使树恢复平衡，我们需要进行两步，第一步，把k1作为根，进行一次右右旋转，旋转之后就变成了左左情况，所以第二步再进行一次左左旋转，最后得到了一棵以k2为根的平衡二叉树树。

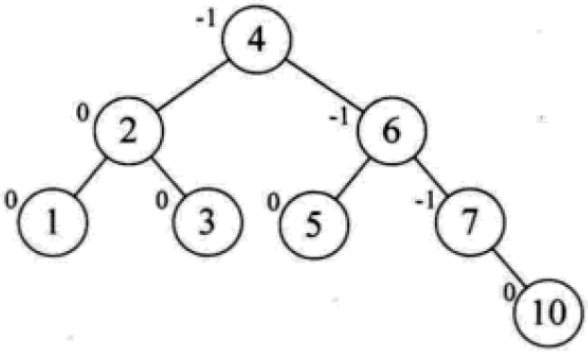
三、构建过程

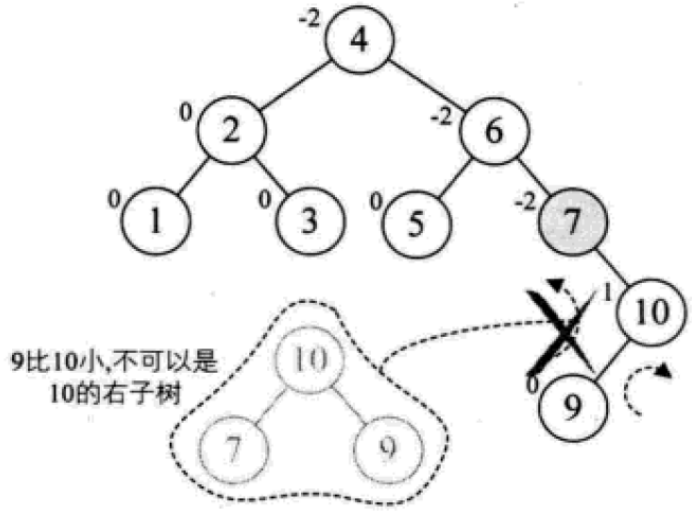
下面是由[1,2,3,4,5,6,7,10,9]构建平衡二叉树

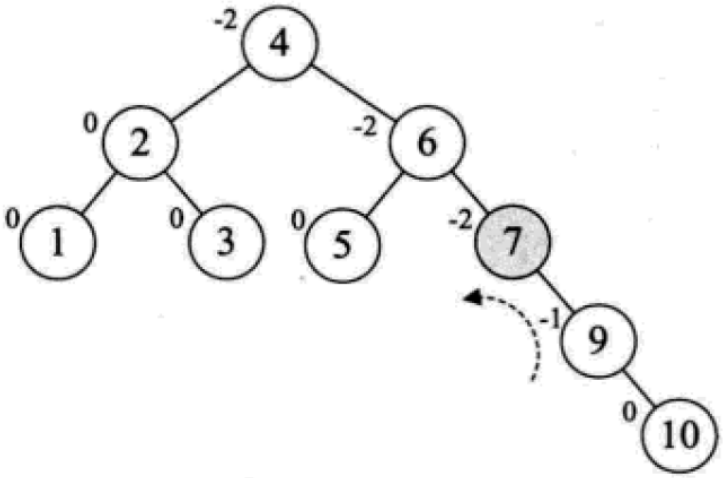



-1
4
0
2
-2
5
0
1
0
3
-1
6
0
7


0
4
0
2
0
6
0
1
0
3
0
5
0
7


-1
4
0
2
-1
6
0
1
0
3
0
5
-1
7
0
10


-2
4
0
2
-2
6
0
1
0
3
0
5
-2
7
1
10
0
9
9比10小,不可以是
10的右子树
10
7
9


-2
4
0
2
-2
6
0
1
0
3
0
5
-2
7
-1
9
0
10

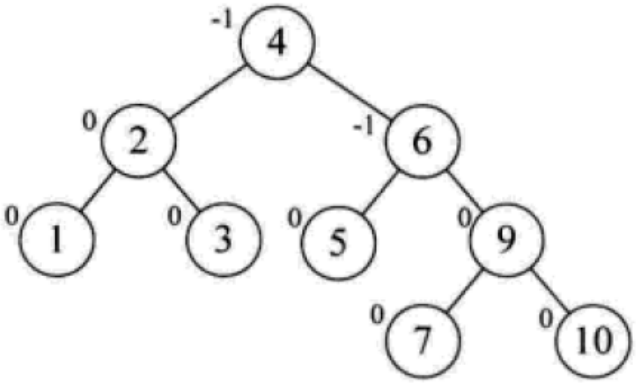


图13

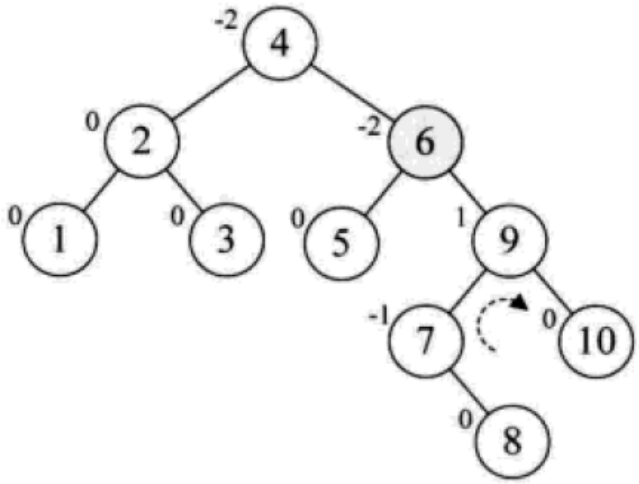


图14

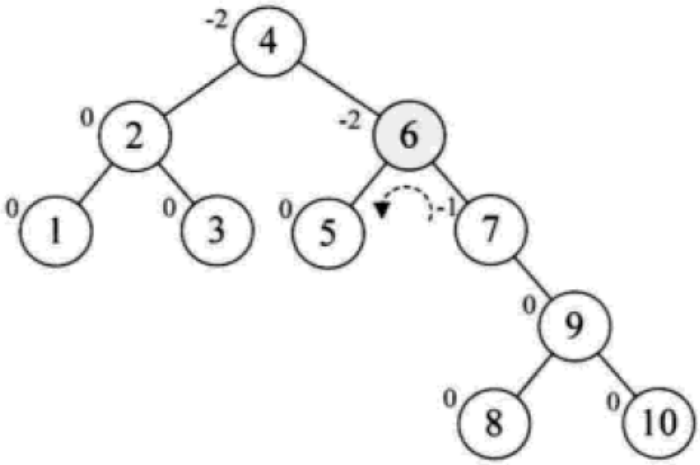


图15

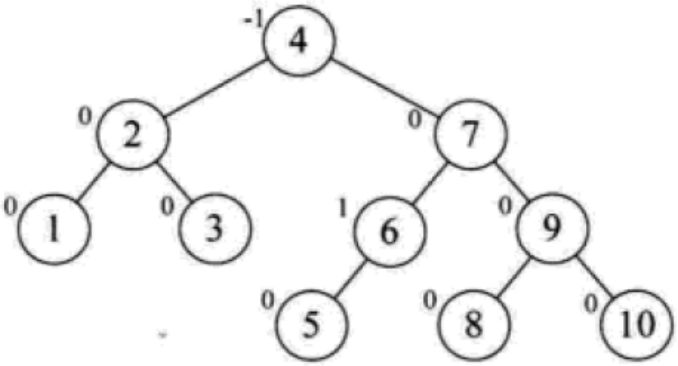


图16