说明

由来

二叉搜索树作为一种数据结构

最好的情况查找、插入和删除操作的时间复杂度都为O(logn),底数为2

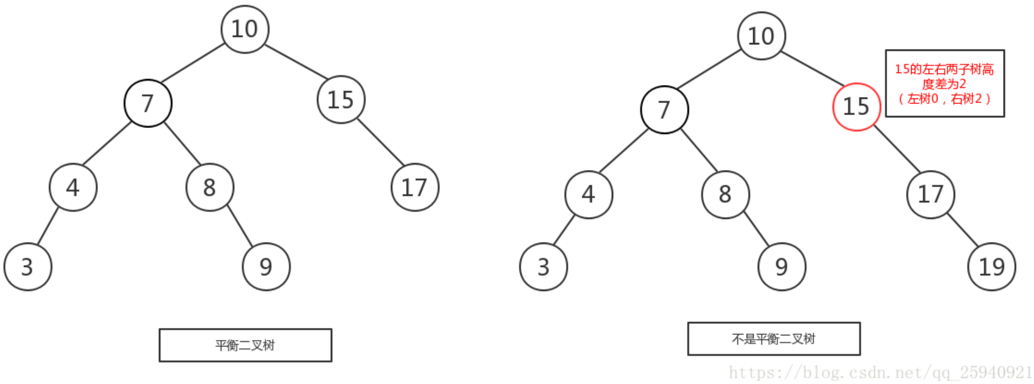
最坏的情况下，一颗斜树，插入删除时间复杂度都为)(longn)，与线性表一致

实际情况: 二叉搜索树的效率应该在O(N)和O(logN)之间，这取决于树的不平衡程度

为了解决这个问题，引出了平衡二叉树(AVL)

定义

平衡二叉树：首先是一棵二叉查找树，但是它满足一点重要的特性：每一个节点的左子树和右子树的高度差最多为1



术语

平衡因子（BF）

左子树的高度 减去 右子树的高度 的绝对值 小于等于 1 公式为：|LH - RH| <= 1

最小不平衡子树

距离插入节点最近的，并且平衡因子的绝对值大于1的节点为根 的子树，被称之为 最小不平衡子树

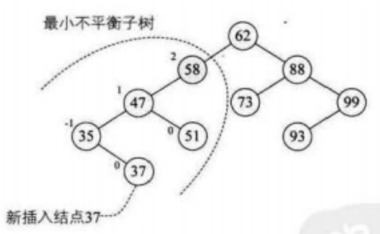


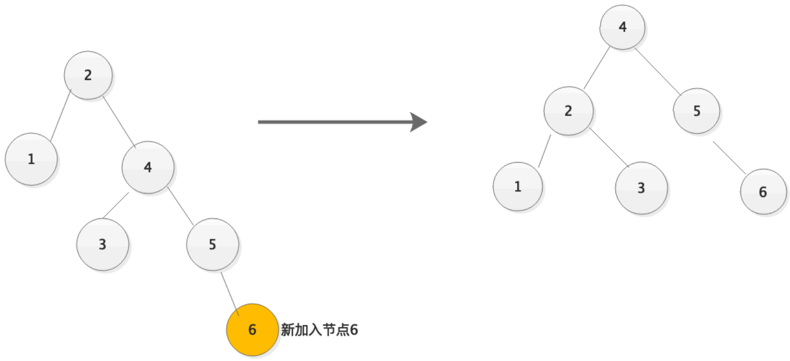
图 8-7-3

实现原理

基本思想

在构建二叉排序树的过程中，每当插入一个结点时，先检查是否因插入而破坏了树的平衡性，若是，则**找出最小不平衡子树**。在保存二叉排**各个结点之间的链接更新，进行相应的旋转，使之成为新的平衡子树**

左旋转：右子节点变成父节点，并把晋升之后多余的左子节点出让给降级节点的右子节点



左旋之前

左旋之后

右旋转：左子节点变成了父节点，并把晋升之后多余的右子节点出让给降级节点的左子节点

不管是左旋还是右旋，旋转的目的都是**将节点多的一支出让节点给另一个节点少的一支**

构建过程 - 添加节点的过程

```
{3,2,1,4,5,6,7,10,9,8}
```

构建二叉排序树

如果节点顺序构建，那么最终的二叉树排序树如图

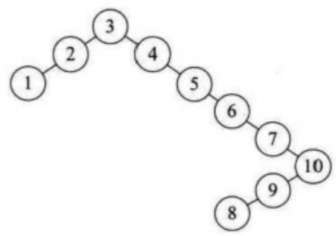


图1

虽然会符合二叉排序树的定义，但是高度达到8的二叉树，查找不好，效率不高，我们应该尽可能是二叉排序树保持平衡，比如图二

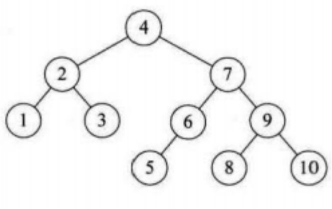


图2

下面来推导图2的生成过程

开始构建平衡二叉树AVL

1. 第一个元素3，符合平衡
2. 第二个元素2，符合平衡
3. 第三个元素1，不符合平衡，那么找到最小不平衡树 3 ，进行旋转

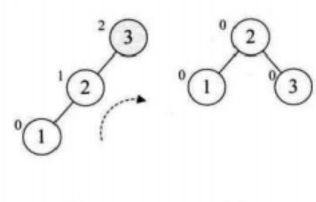


图1　　图2

```
注意：平衡因子为正数，则右转，为负数，则左转
```

4. 第四个元素4，符合平衡

图3

5. 第五个元素5，不符合平衡，找到最小不平衡树 3，进行旋转

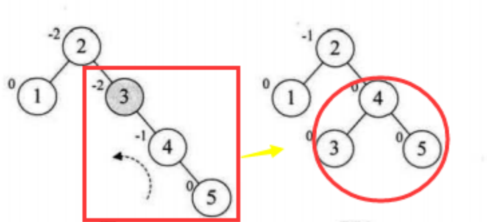
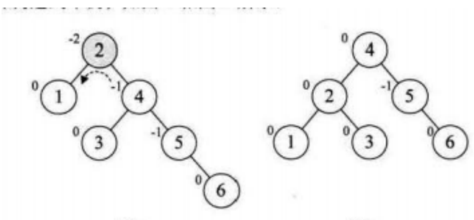


图4　　图5

6. 第六个元素6，不符合平衡，找到最小不平衡树 2，进行旋转



7. 第七个元素7，不符合平衡，找到最小不平衡树 5，进行旋转

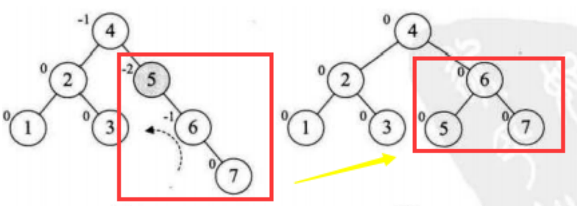
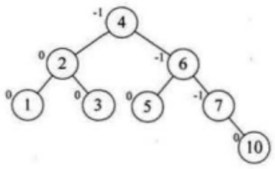


图8　　图9

图 8-7-6

8. 第八个元素10，符合平衡



9. 第九个元素9，不平衡，最小不平衡子树的节点是7

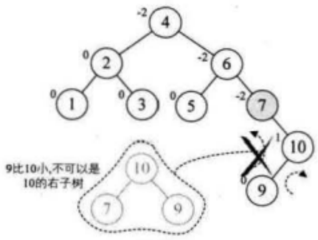


图11

注意：因为我们的结点7的BF=-2，而他的子结点10的BF是1，对于两个符号不统一的最小不平衡子树我们都应该先让其符号相同，所以将根节点的子节点10进行右转

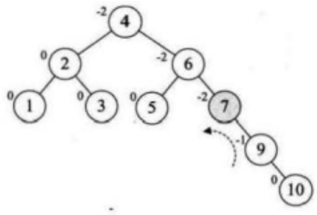


图12

图 8-7-7

然后再对整个不平衡子树按照结点7的进行左旋

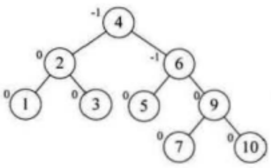


图13

10. 第十个元素8，不符合平衡，最小不平衡树的节点 6

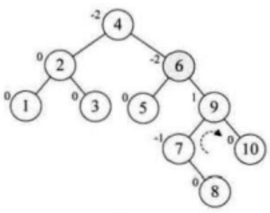


图14

我们先对最小不平衡子树的子树进行右旋转，使得其符号统一，按照结点9的BF=1

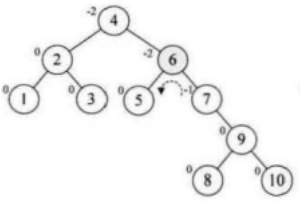


图15

使最小不平衡子树符号相同，然后我们根据结点6的BF=-2，进行左旋

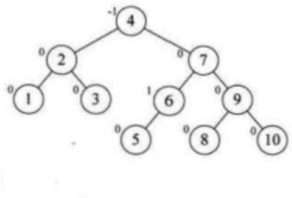


图16

旋转总结

插入的话就是以下四种情况:

- LL型：在根结点的左孩子的左子树上插入，对根结点进行右旋转
- RR型：在根结点的右孩子的右子树上插入，对根结点进行左旋转
- LR型：在根结点的左孩子的右子树上插入，先对根结点的左孩子进行左旋转，再对根结点进行右旋转
- RL型：在根结点的右孩子的左子树上插入，先对根结点的右孩子进行右旋转，再对根结点进行左旋转

注意：上述根节点，表示最小不平衡树的根节点

旋转代码实现

```
public void rotateLeft(Node h) {
    Node x = h.right; // 根结点的右孩子保存为x
    h.right = x.left; // 根结点右孩子的左孩子挂到根结点的右孩子上
    x.left = h; // 根结点挂到根结点右孩子的左孩子上
    h = x; // 根结点的右孩子代替h称为新的根结点
}

public void rotateRight(Node h) {
    Node x = h.left; // 根结点的左孩子保存为x
     h.left = x.right; // 根结点左孩子的右孩子挂到根结点的左孩子上
     x.right = h; // 根结点挂到根结点左孩子的右孩子上
     h = x; // 根结点的左孩子代替h称为新的根结点
 }
```

删除节点的处理

删除的情况也比较复杂，删除二叉树节点总结起来就两个判断：①删除的是什么类型的节点？②删除了节点之后是否导致失衡

节点的类型有三种：1.没有子节点；2.只有一个子节点；3.既有左子树又有右子树

没有子节点

当删除的节点是叶子节点，则将节点删除，然后从父节点开始，判断是否失衡，如果没有失衡，则再判断父节点的父节点是否失衡，直衡，则说此时树是平衡的；如果中间过程发现失衡，则判断属于哪种类型的失衡（左左，左右，右左，右右），然后进行调整

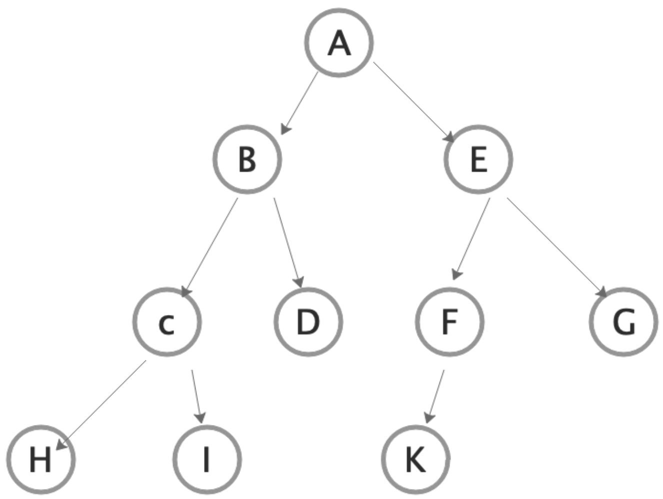
只有一个子节点

删除的节点只有左子树或只有右子树，这种情况其实就比删除叶子节点的步骤多一步，就是将节点删除，然后把仅有一支的左子树或右就一样了，从父节点开始，判断是否失衡，如果没有失衡，则再判断父节点的父节点是否失衡，直到根节点，如果中间过程发现失衡，则根

既有左子树又有右子树

删除的节点既有左子树又有右子树，这种情况又比上面这种多一步，就是找到待删除节点的左子树最大节点或者右子树最小节点，然后的节点删掉，后面的步骤也是一样，判断是否失衡，然后根据失衡类型进行调整

如何调整

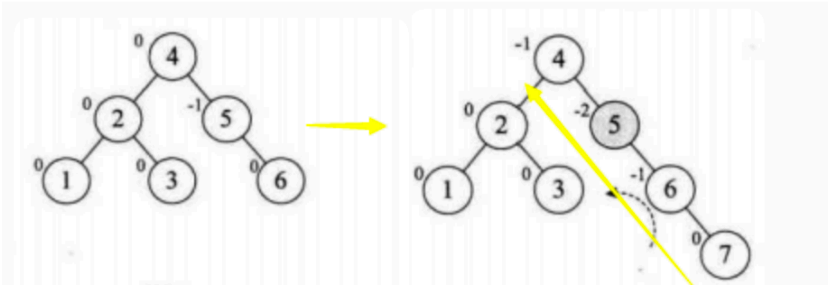


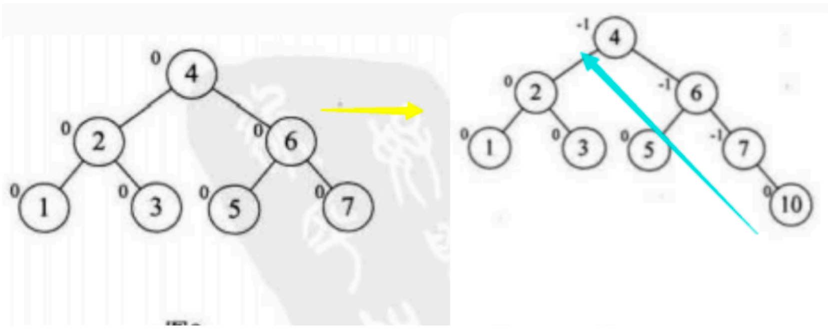
删除 K，依然保持平衡，从父节点追溯到根节点的平衡因子都需进行修改
删除 G，不平衡，E子树右旋操作： E到G，F到E，K到F 。 F 、E、A平衡因子都要修改
删除 K，F

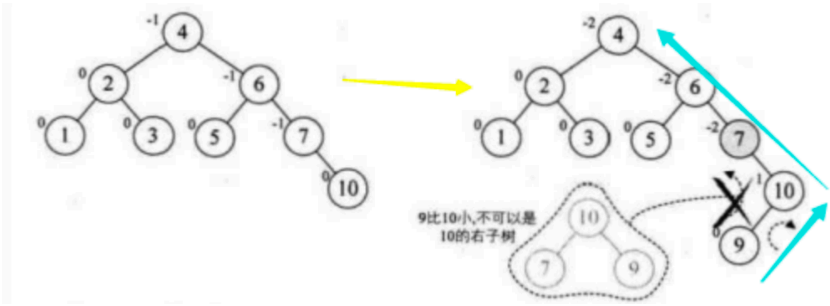
节点插入平衡因子的规律（未完成）

插入一个节点如何修改平衡因子 - 推导过程

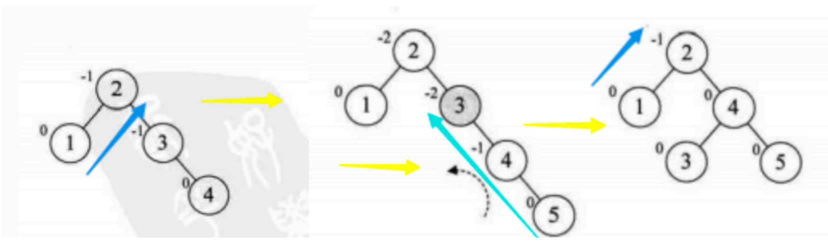
1. 插入一个节点，只会影响到该节点到根节点路径上所经过节点的BF值

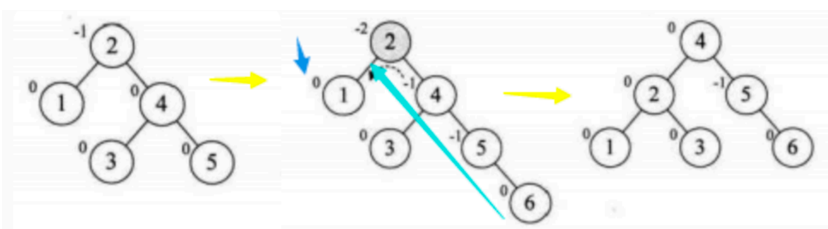


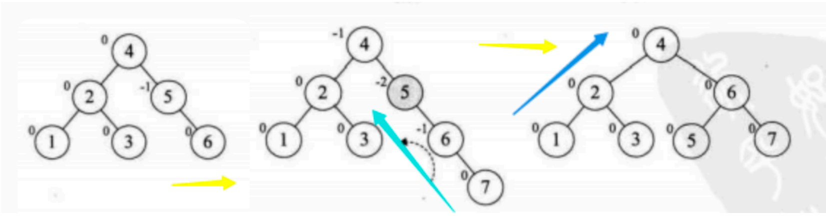


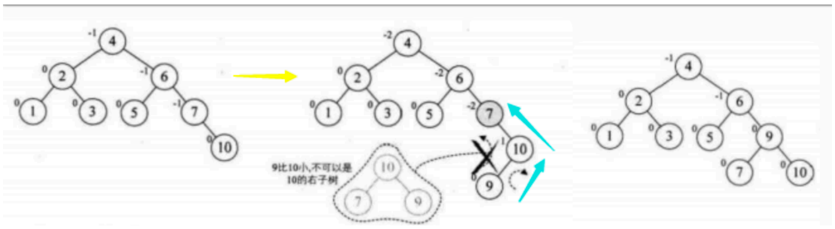


2. 插入一个新节点，那么新节点的BF值一定为0
3. 对一个最小不平衡子树做了平衡处理后，会发现只对这个最小不平衡子树的BF进行了改变，而对于这棵树中的其他结点的BF值，虽
是一样的

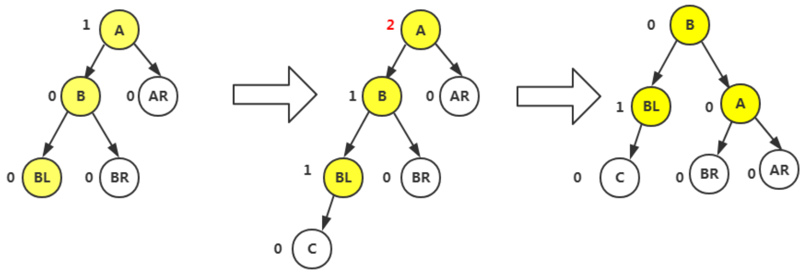




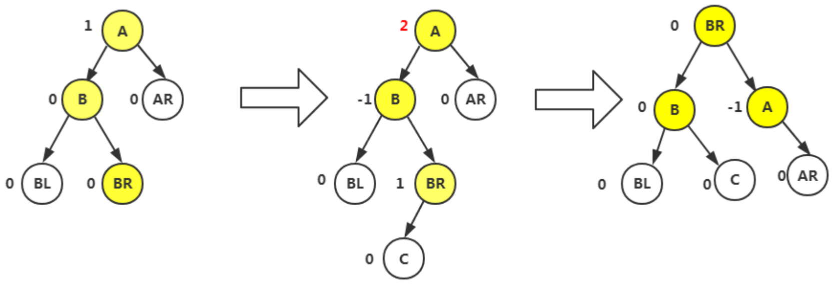




4. 除了参与旋转的三个结点，在最小不平衡子树的其他结点的BF值也不会改变



LL型



LR型

RR型和RL型同上(图略过)

5. 参与旋转的节点 BF 值变化规律