第15章

平衡搜索树

本章内容

- **15.1 AVL 树**
- *15.2 红-黑树
- *15.3 分裂树
- **15.4 B-树**

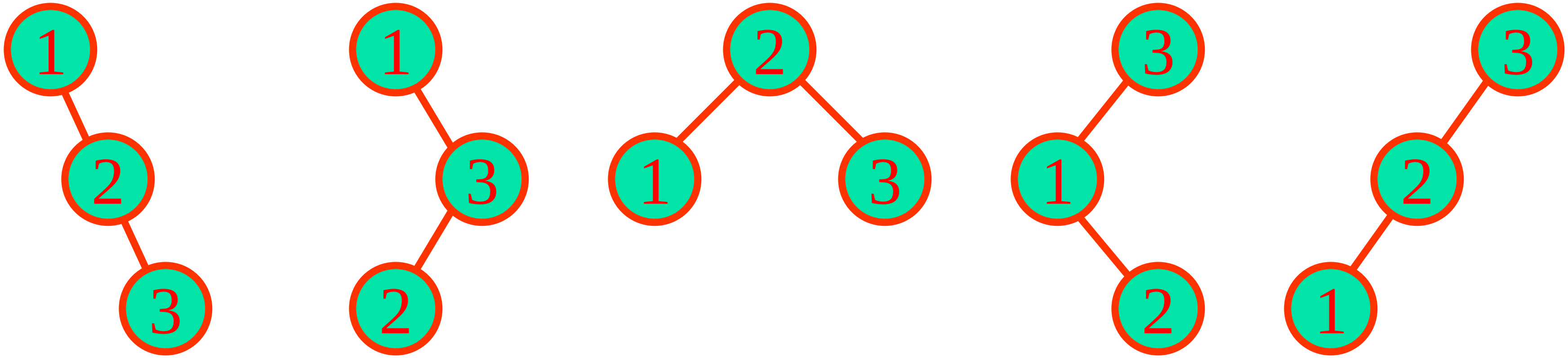
思考

同样 3 个数据{ 1, 2, 3 }，输入顺序不同，建立起来的二叉搜索树的形态也不同。这直接影响到二叉搜索树的搜索性能。

如果输入序列选得不好，会建立起一棵单支树，使得二叉搜索树的高度达到最大，这样必然会降低搜索性能。

{2, 1, 3}

{1, 2, 3}　{1, 3, 2}　{2, 3, 1}　{3, 1, 2}　{3, 2, 1}



AVL 树

- 当搜索树的高度总是O(logn)时，能够保证每个搜索树操作所占用的时间为O(logn)。
- AVL(Adelson-Velsky和Landis1962年提出)树——一种**平衡树**。

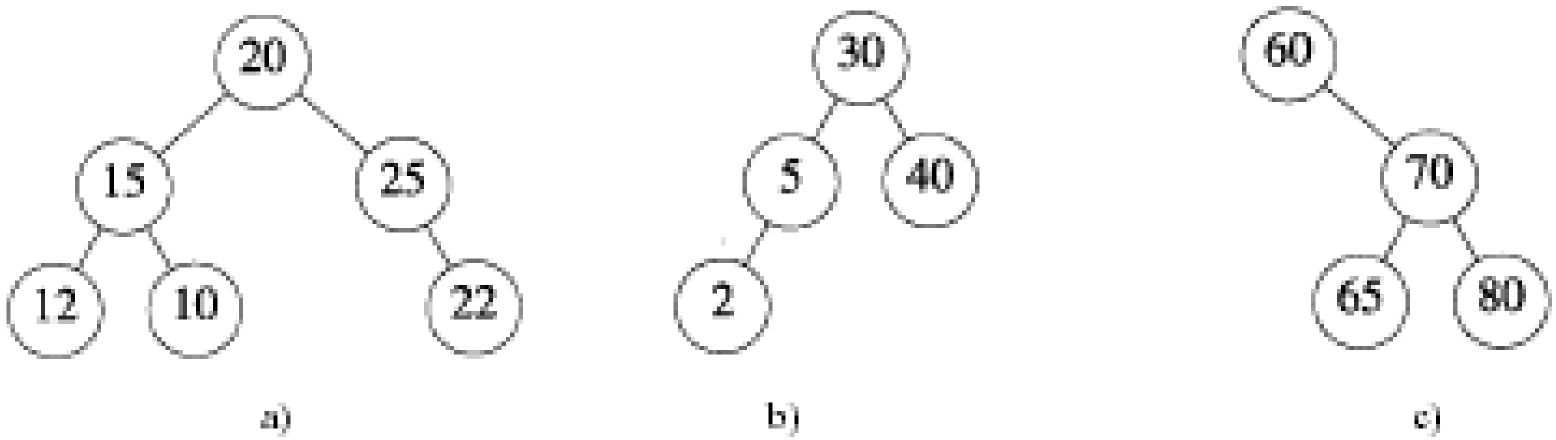
AVL树

AVL树定义：

- 空二叉树是AVL树。
- 如果T是一棵非空的二叉树， T_L 和 T_R 分别是其左子树右子树，当T满足以下条件时， T是一棵AVL树。
  1. T_L 和 T_R 是AVL树，
  2. $|h_L - h_R| \le 1$， h_L 和 h_R 分别是左子树和右子树的高度。

AVL搜索树

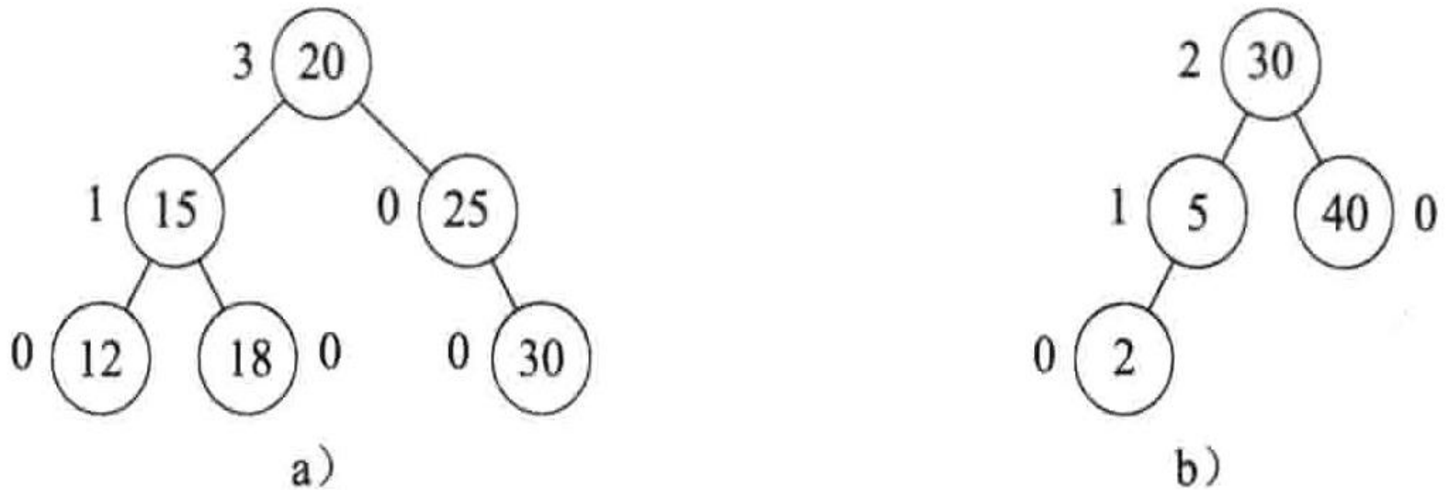
- **AVL搜索树**（平衡二叉搜索树/平衡二叉排序树）**：** 既是**二叉搜索树**，也是**AVL树**。



- AVL 树?
  - (a) 、(b)
- AVL搜索树?
  - (b)

带索引的AVL搜索树

- **带索引的AVL搜索树**既是**带索引的二叉搜索树**，也是**AVL树**。



•带索引的AVL搜索树?

–(a) 、 (b)

AVL 树的特征

1. n 个元素(节点)的AVL树的高度是O(logn)。
2. 对于每一个n(n≥0)值，都存在一棵AVL树。(否则，在插入完成后，一棵AVL树将不是AVL树，因为对当前元素数来说不存在对应的AVL树)。
3. 一棵n元素的AVL搜索树能在O(高度)= O(logn)的时间内完成搜索。
4. 将一个新元素插入到一棵n元素的AVL搜索树中，可得到一棵n+1元素的AVL树，这种插入过程可以在O(logn)时间内完成。
5. 从一棵n元素的AVL搜索树中删除一个元素，可得到一棵n-1元素的AVL树，这种删除过程可以在O(logn)时间内完成。

- 特征2可以从特征4推出。

AVL 树的高度

- 一棵有n个节点的AVL树的高度至多：
  - $1.44 \log_2 (n+2)$.

- 一棵有n个节点的AVL树的高度至少：
  - $\log_2 (n+1)$.

- N_h：高度为h的AVL树中的最小节点数。
- $N_h = N_{h-1} + N_{h-2} + 1$，$N_0 = 0$, $N_1 = 1$
- Fibonacci number(斐波那契数列):
  - $F_h = F_{h-1} + F_{h-2}$，$F_0 = 0$, $F_1 = 1$

F_h 和 N_h之间的关系

- $N_h = F_{h+2} - 1 \quad h >= 0$
- 归纳证明:
- 因为 $F_3 = F_2 + F_1 = 2F_1 + F_0 = 2$
- 所以$N_1 = 1$, h=2 成立;
- 假设 h< m 成立，证明h=m时，成立;
- $N_m = N_{m-1} + N_{m-2} + 1$
- $= (F_{m+1} - 1) + (F_m - 1) + 1$ (归纳假设）
- $= (F_{m+1} + F_m) - 1$
- $= F_{m+2} - 1$

- 按照斐波那契定理可知

$$F_h = \phi^h / \sqrt{5}, h \geq 0, 其中, \phi = (1+\sqrt{5})/2$$

$$N_h = F_{h+2} - 1, h \geq 0$$

$$N_h = \phi^{h+2} / \sqrt{5} - 1$$

$$n \geq \phi^{h+2} / \sqrt{5} - 1$$

$$h = \log_\phi(\sqrt{5}(n+1)) - 2 \approx 1.44 \log_2(n+2) = O(\log n)$$

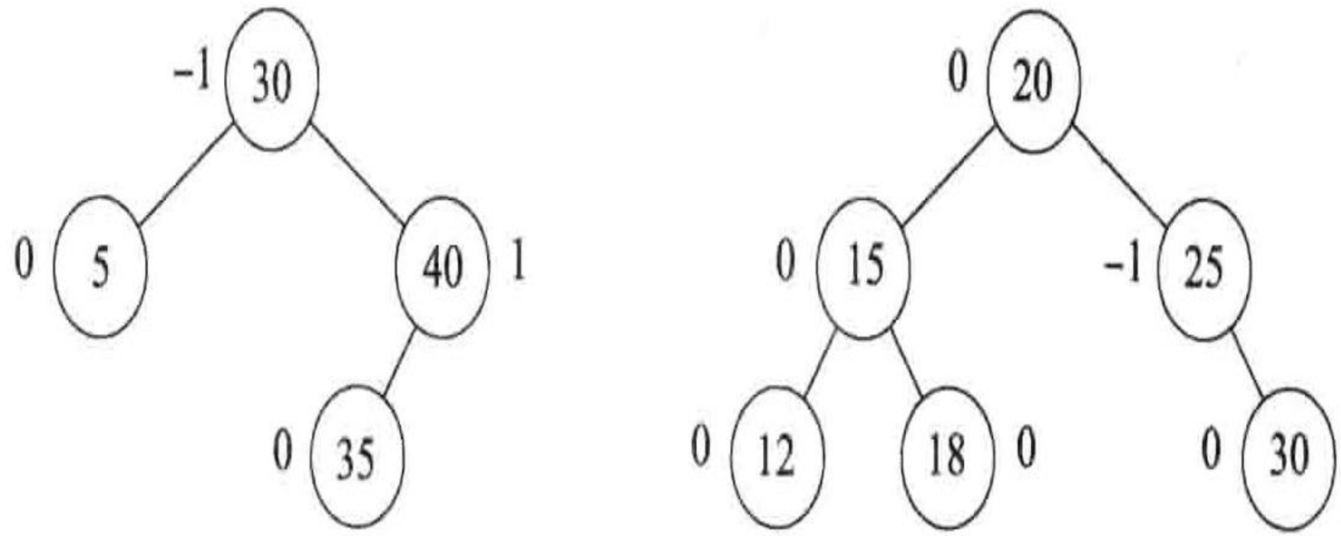
AVL树的描述

- 一般用链表方式来描述
  - 要为每个节点增加一个平衡因子

- **平衡因子(Balance Factor)**：
  - 节点x的平衡因子**bf(x)** 定义为：
    x的左子树的高度 - x的右子树的高度

  - AVL树平衡因子的可能取值为：–1, 0, 和 1.

具有平衡因子的AVL 树

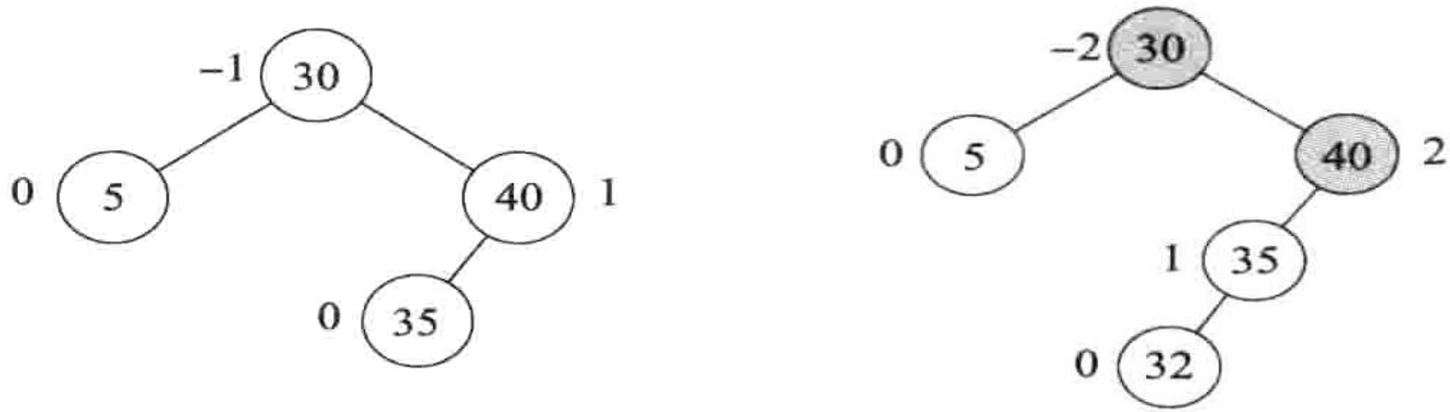


AVL搜索树的搜索

- 使用二叉搜索树的搜索
- 搜索所需时间：
  - O(logn)

AVL搜索树的插入

- 用二叉搜索树的的插入方法将元素插入到AVL搜索树中



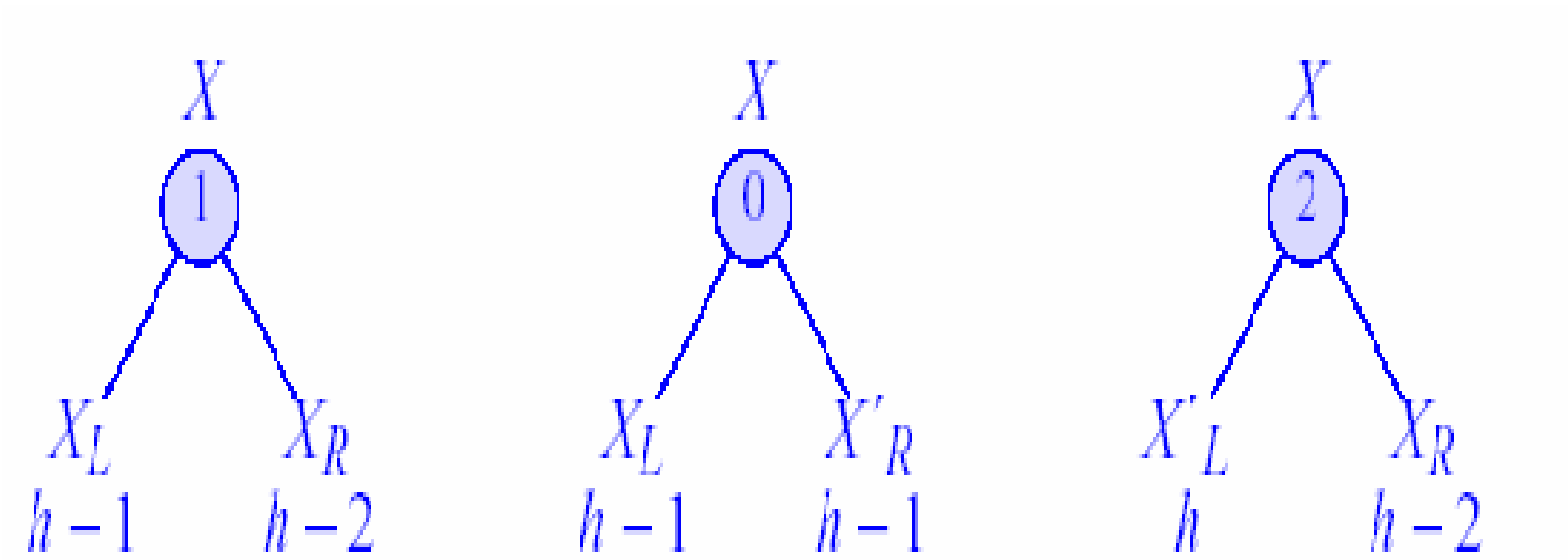
- 得到的树可能不再是AVL树(不平衡)
- 如果树成为不平衡的，我们需要进行调整来恢复树的平衡
- 如何调整？

- 由插入操作导致产生不平衡树的几种现象：

1. 不平衡树中的平衡因子的值限于-2,-1,0,1,2.
2. 平衡因子为2的节点在插入前平衡因子为1，与此类似，平衡因子为-2的，插入前为-1。
3. 在插入后，只有从根到新插入节点的路径上的节点才可能改变平衡因子。
4. 假设A是平衡因子是-2或2的距离新插入节点最近的祖先结点。那么，在插入前从A到新插入节点的路径上的所有节点的平衡因子都是0。

- A是AVL中的哪个节点？

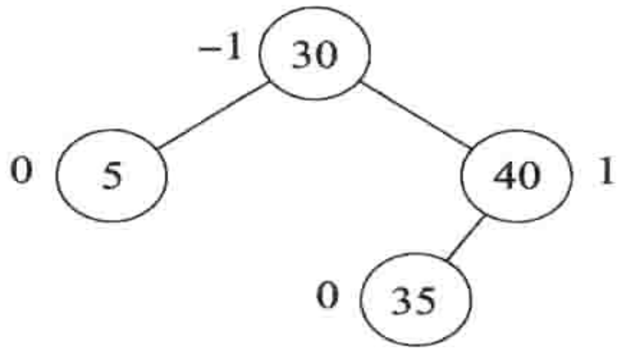
- 节点X：在插入前，我们从根节点往下移动寻找新元素的插入位置时，从根节点到新元素的路径上，最后一个遇到的平衡因子是-1或1的节点.

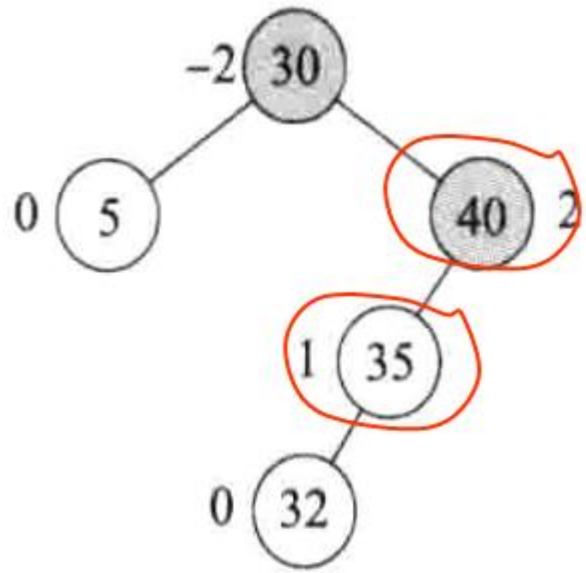


(a) 插入之前　　(b)插入到X_R中之后　　(c)插入到X_L中之后

AVL搜索树的插入

Insert(32)

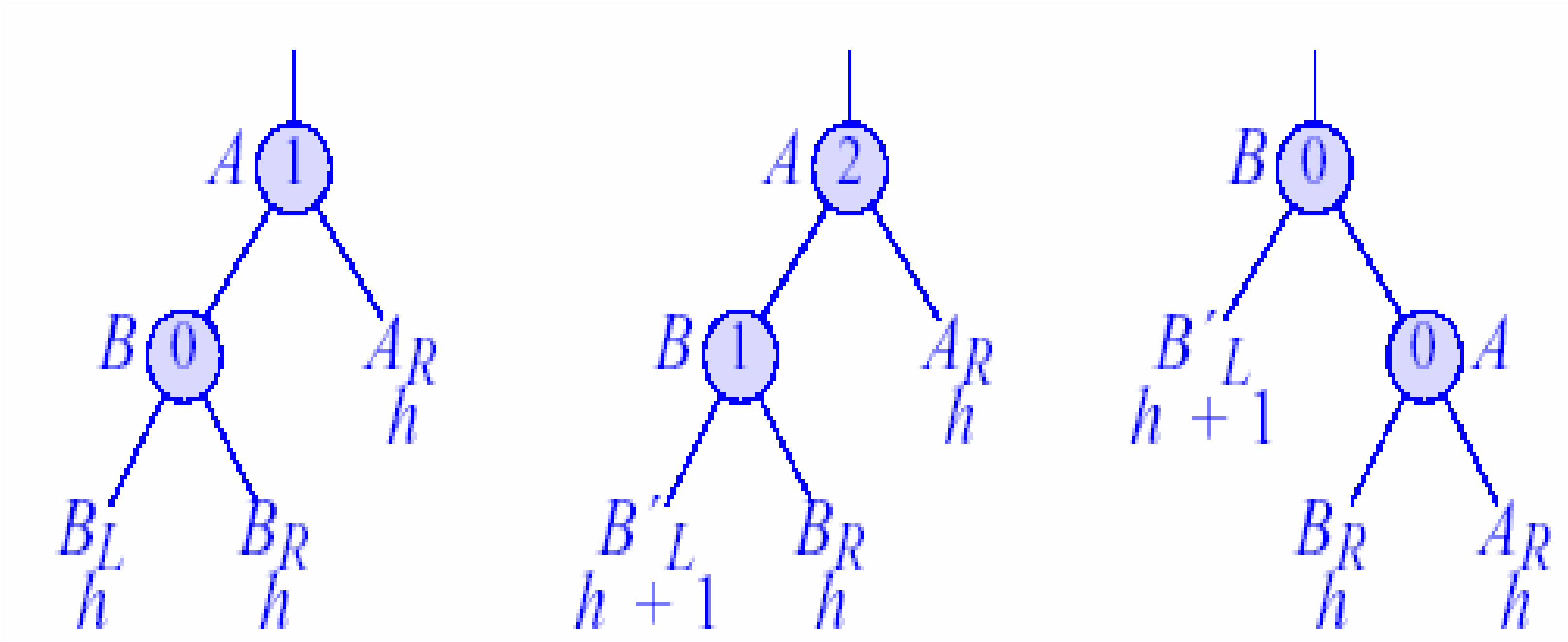




不平衡类型

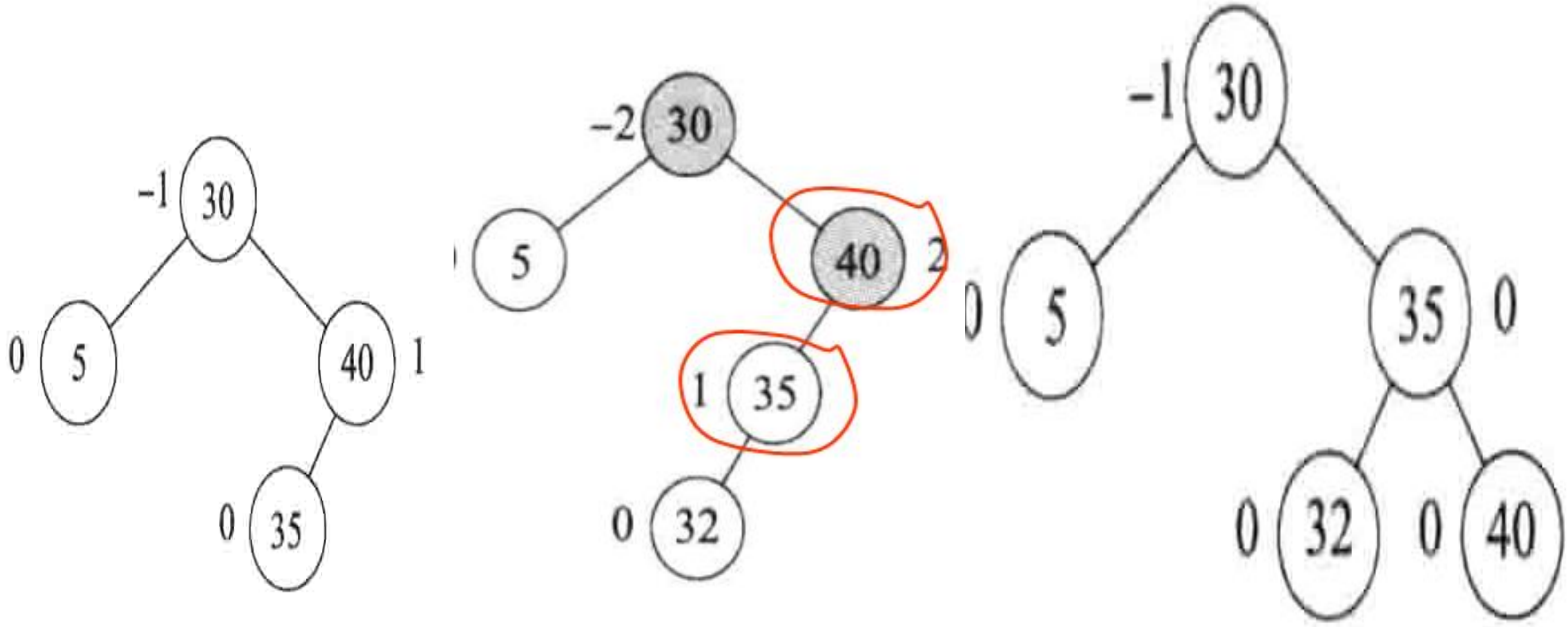
- 在插入后, A的平衡因子是-2或2， 当节点A已经被确定时，A的不平衡性：
  1. **LL**: 新插入节点在A节点的左子树的左子树中
  2. **LR**: 新插入节点在A节点的左子树的右子树中
  3. **RR**: 新插入节点在A节点的右子树的右子树中
  4. **RL**: 新插入节点在A节点的右子树的左子树中

LL旋转



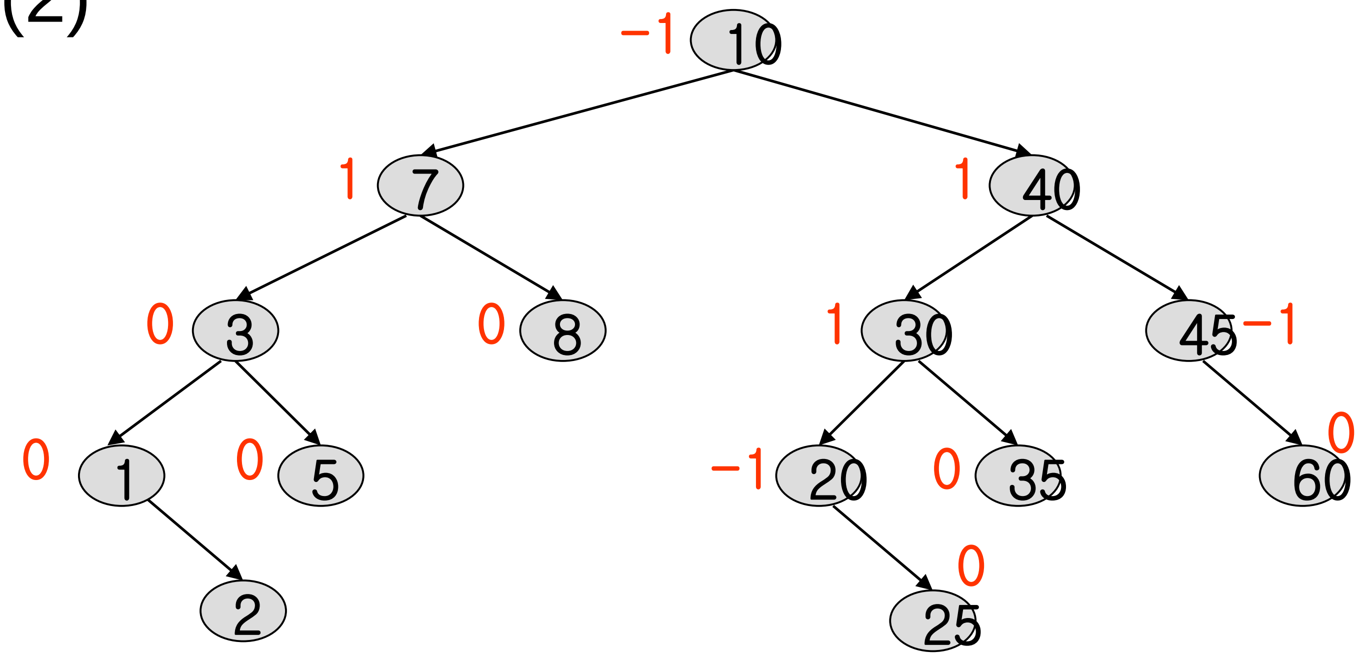
(a) 插入之前　　　(b)插入到B_L中之后　　　(c) LL旋转后

LL旋转示例



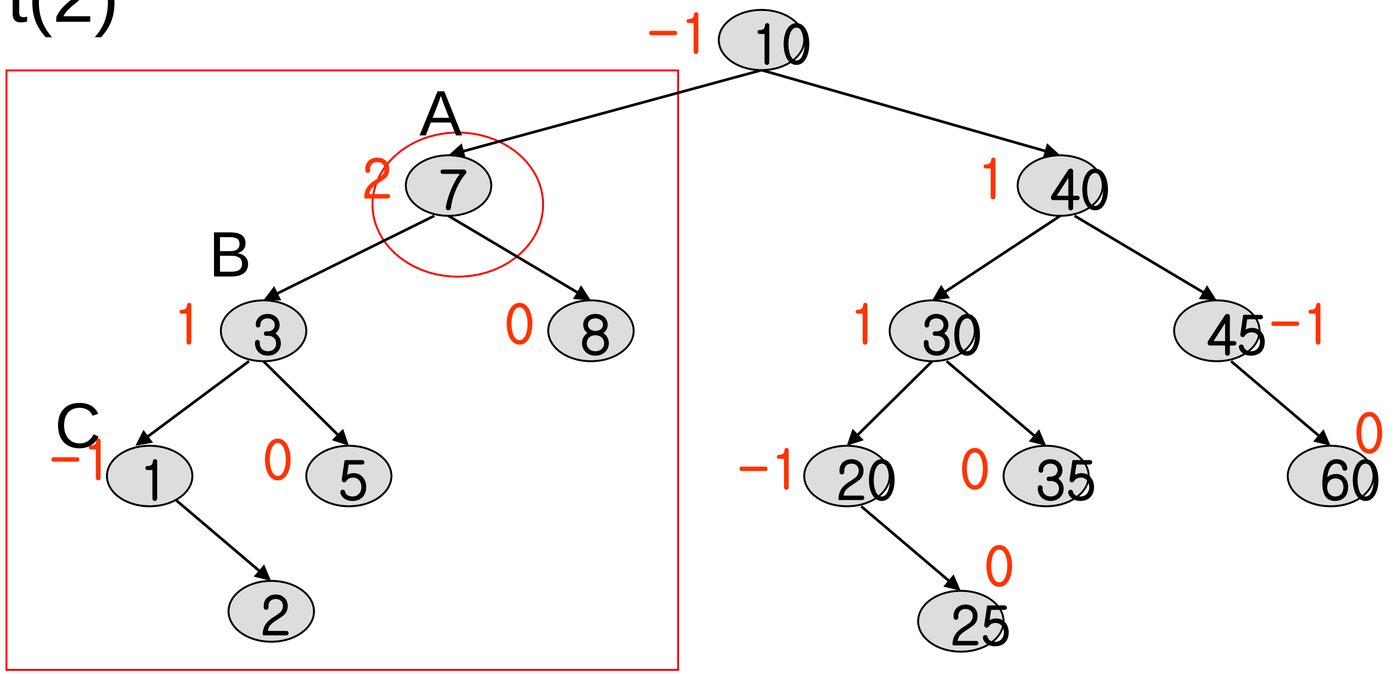
例：LL不平衡

Insert(2)



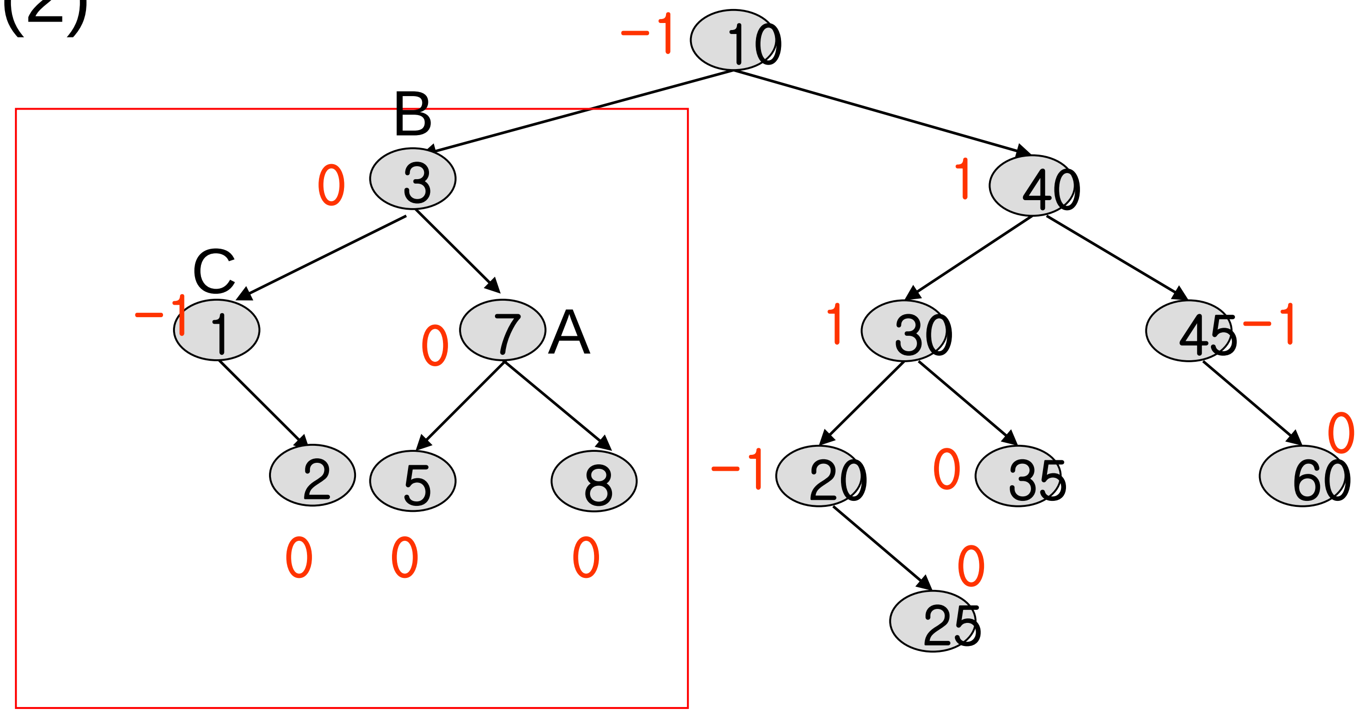
例：LL不平衡

Insert(2)

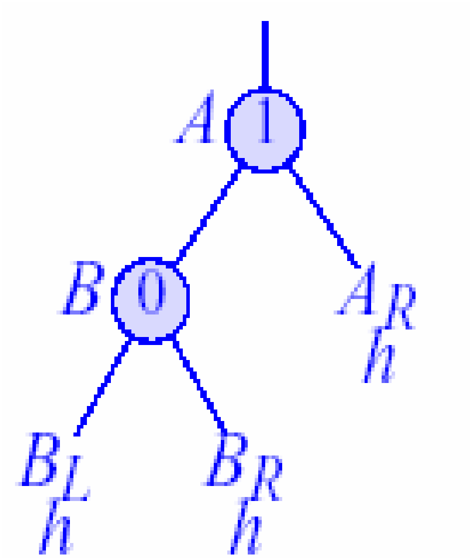


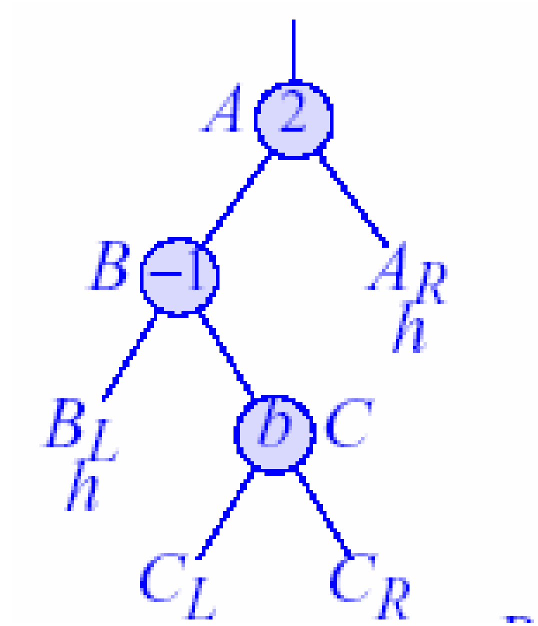
例： LL不平衡

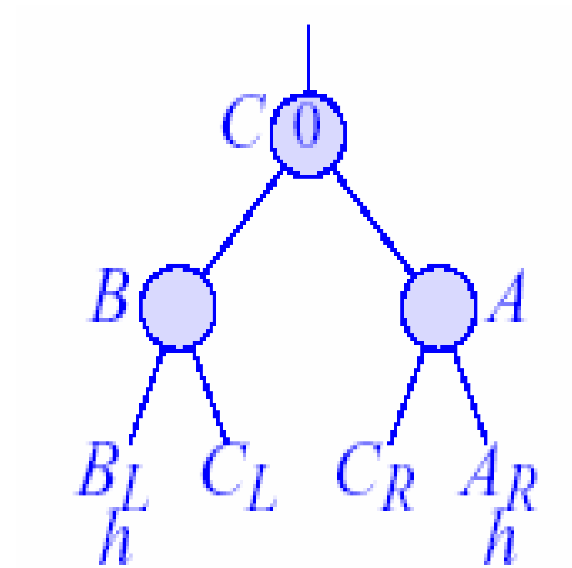
Insert(2)



LR 旋转





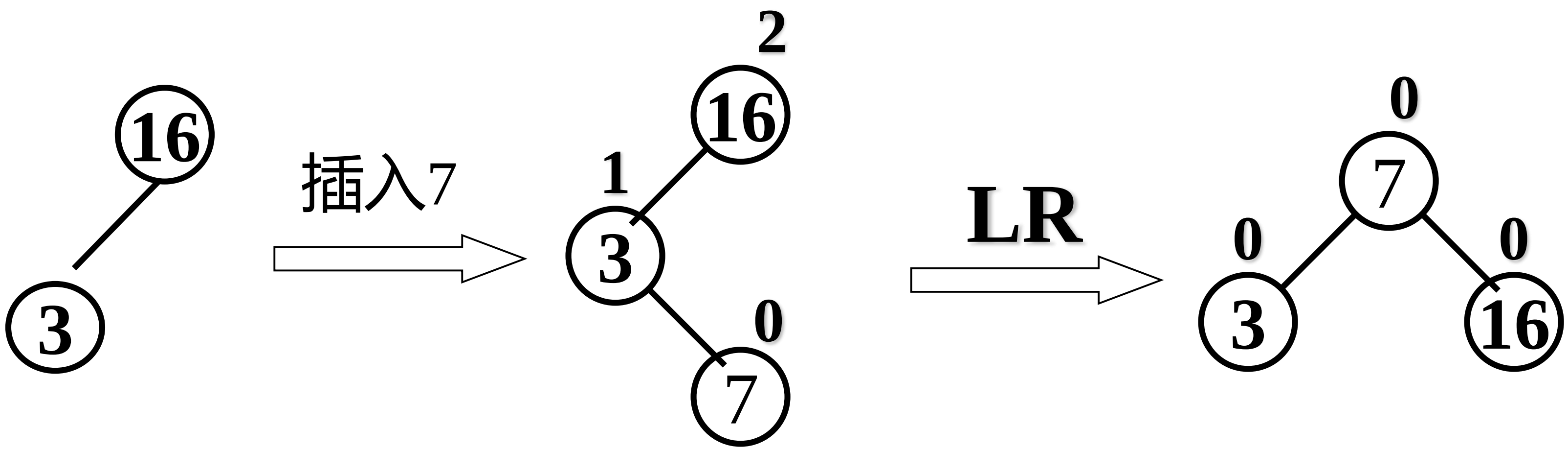


(a)插入之前

(b)插入到B_R之后

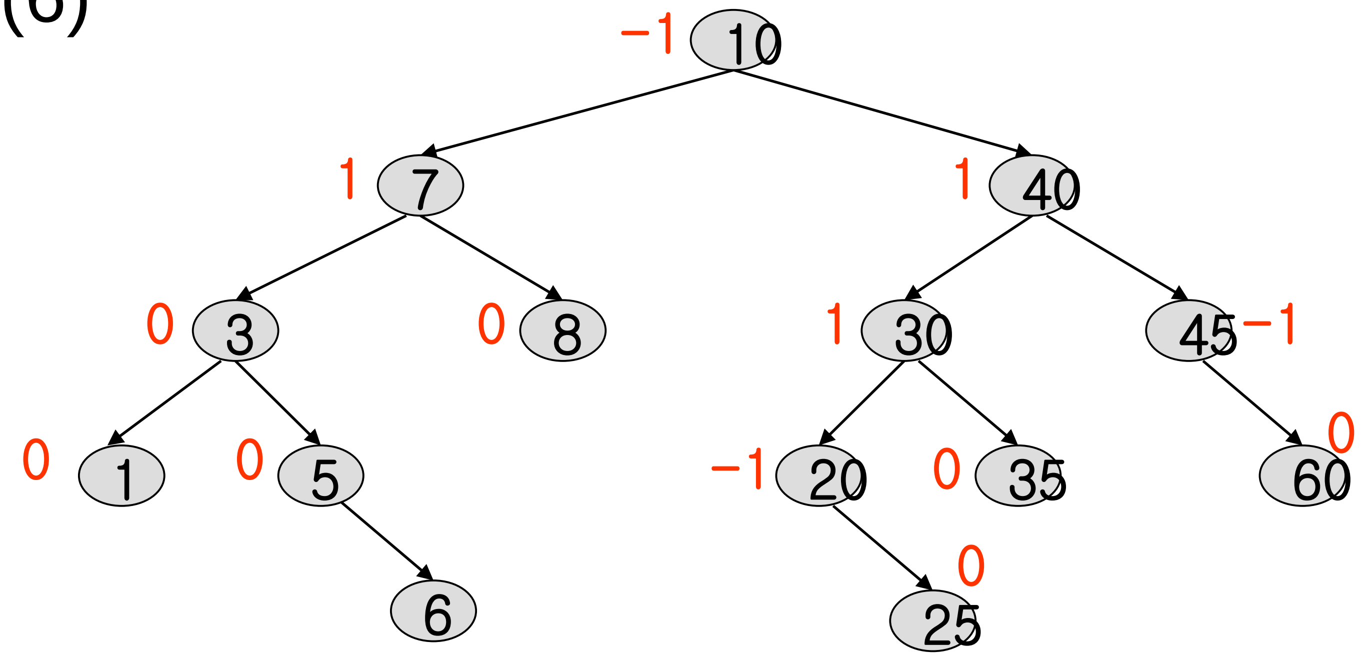
(c)LR旋转之后

例：LR旋转



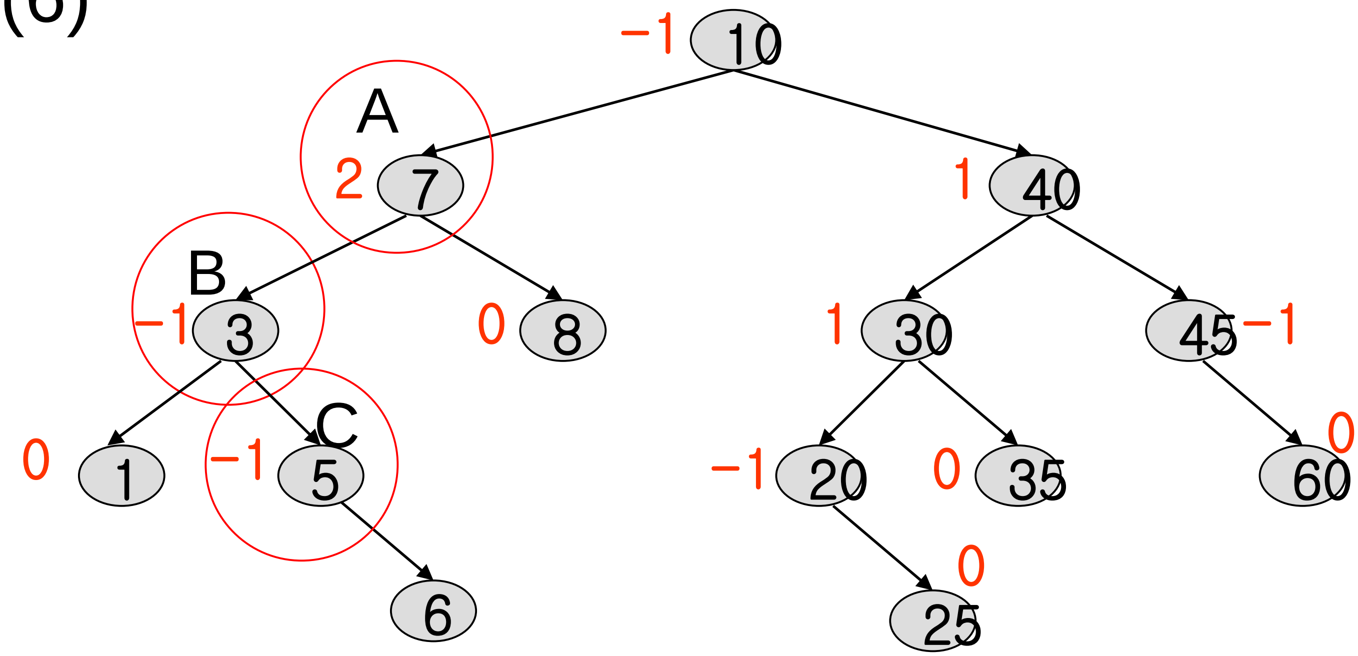
例：LR不平衡

Insert(6)



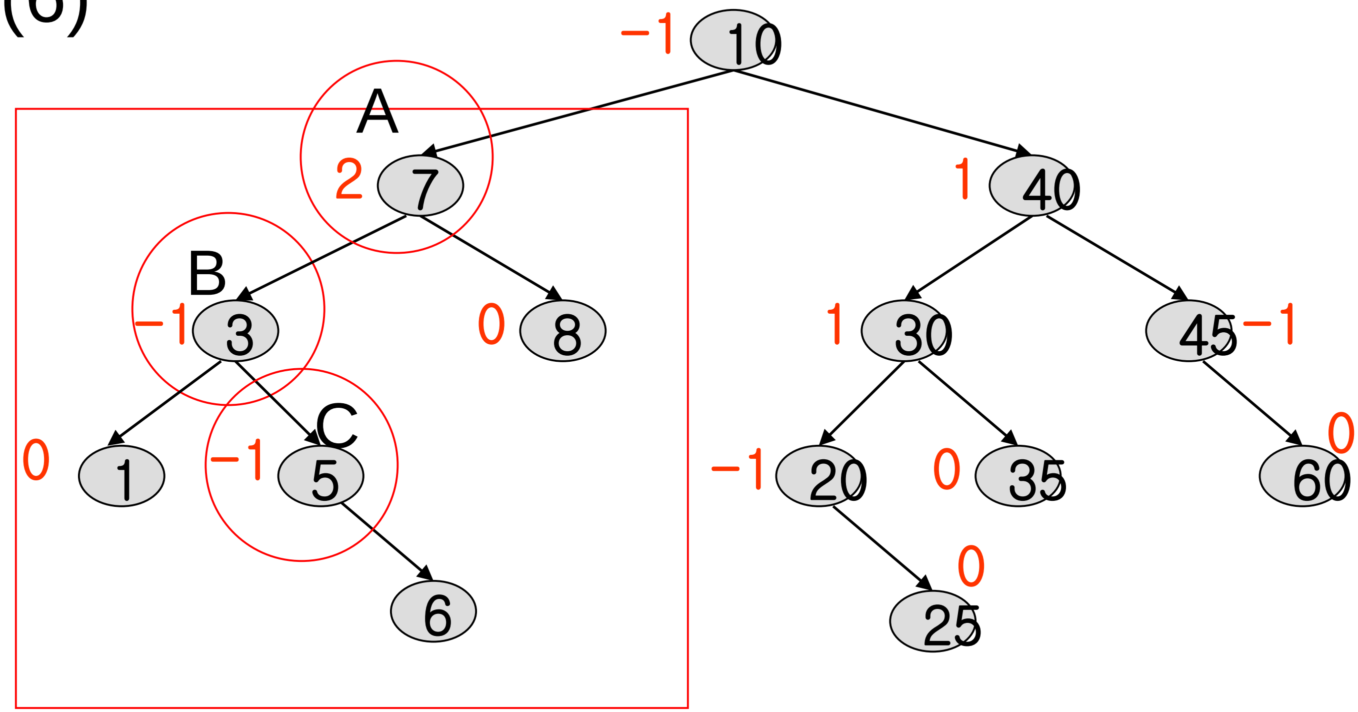
例：LR不平衡

Insert(6)



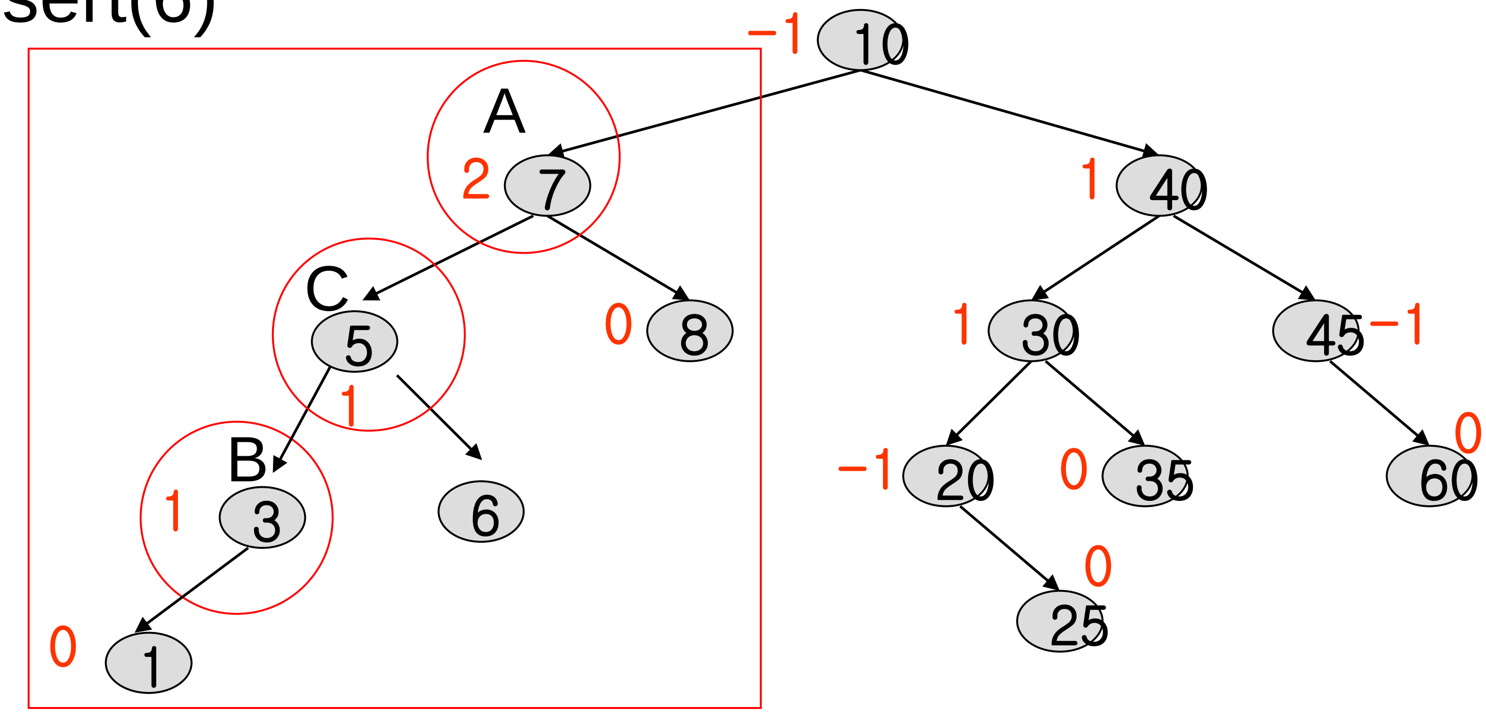
例：LR不平衡

Insert(6)



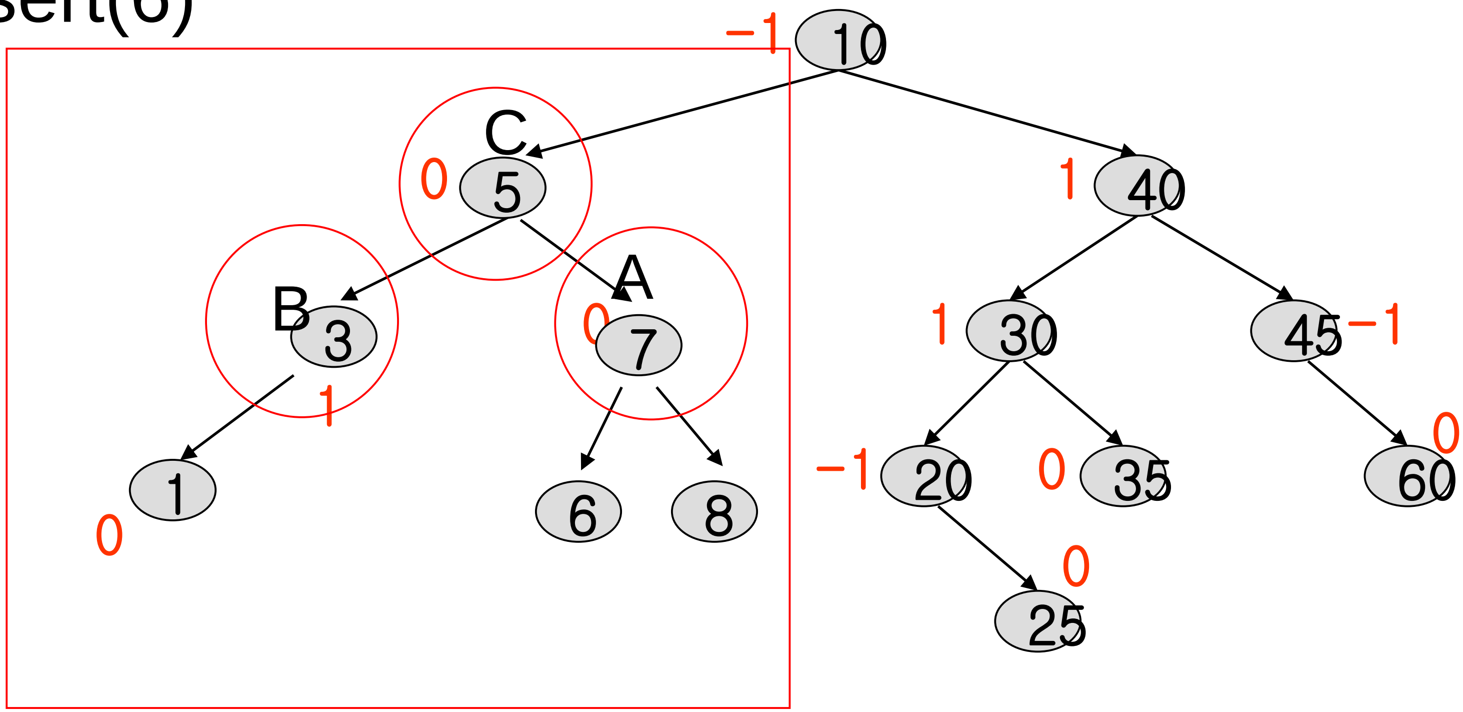
例：LR不平衡

Insert(6)

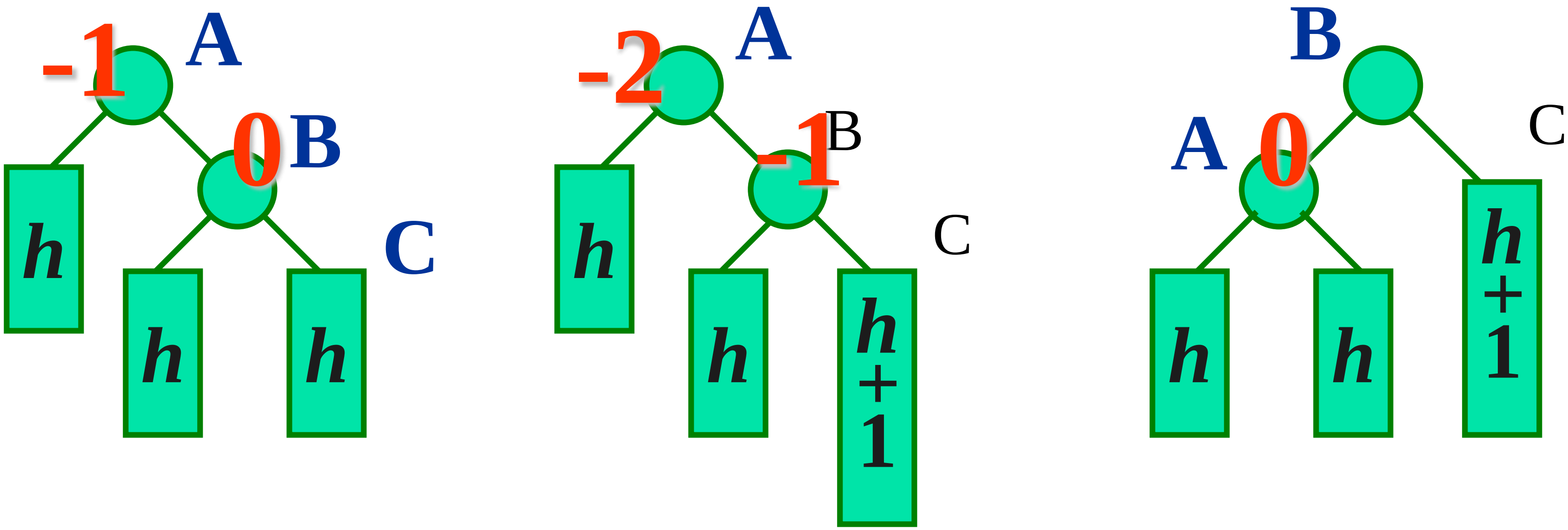


例：LR不平衡

Insert(6)



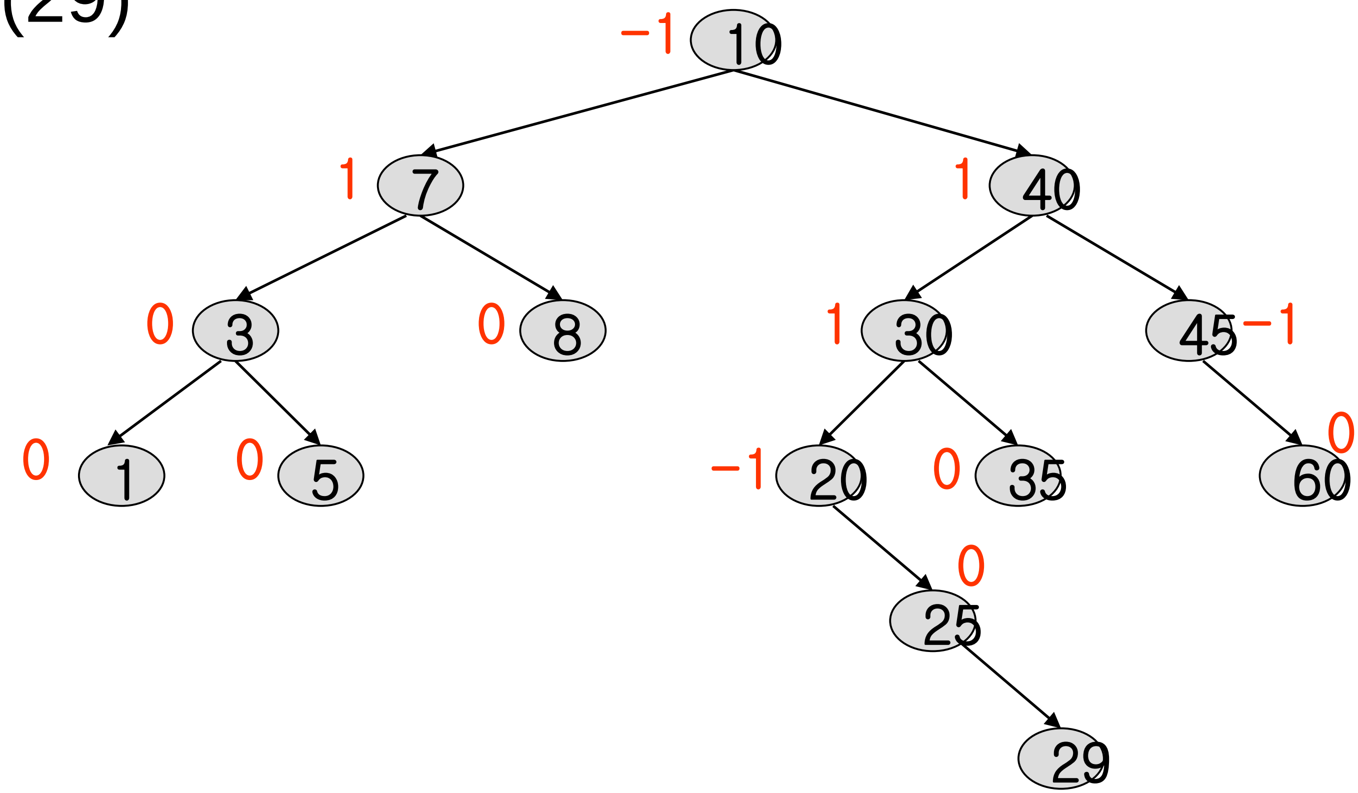
RR不平衡



(a)插入前 (b) 插入后 (c)旋转后

例： RR不平衡

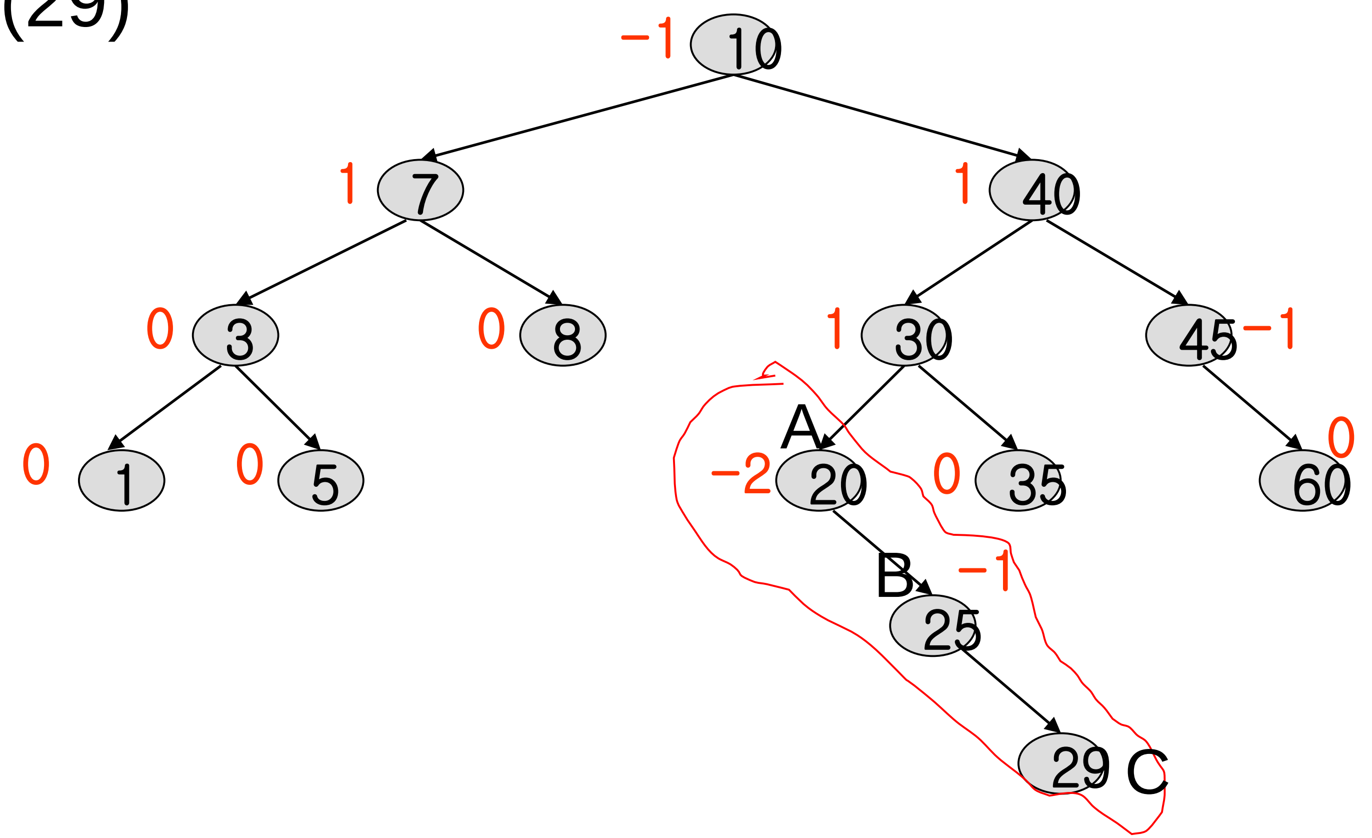
Insert(29)



•在插入之后, 是否还是 AVL搜索树(是否还平衡)?

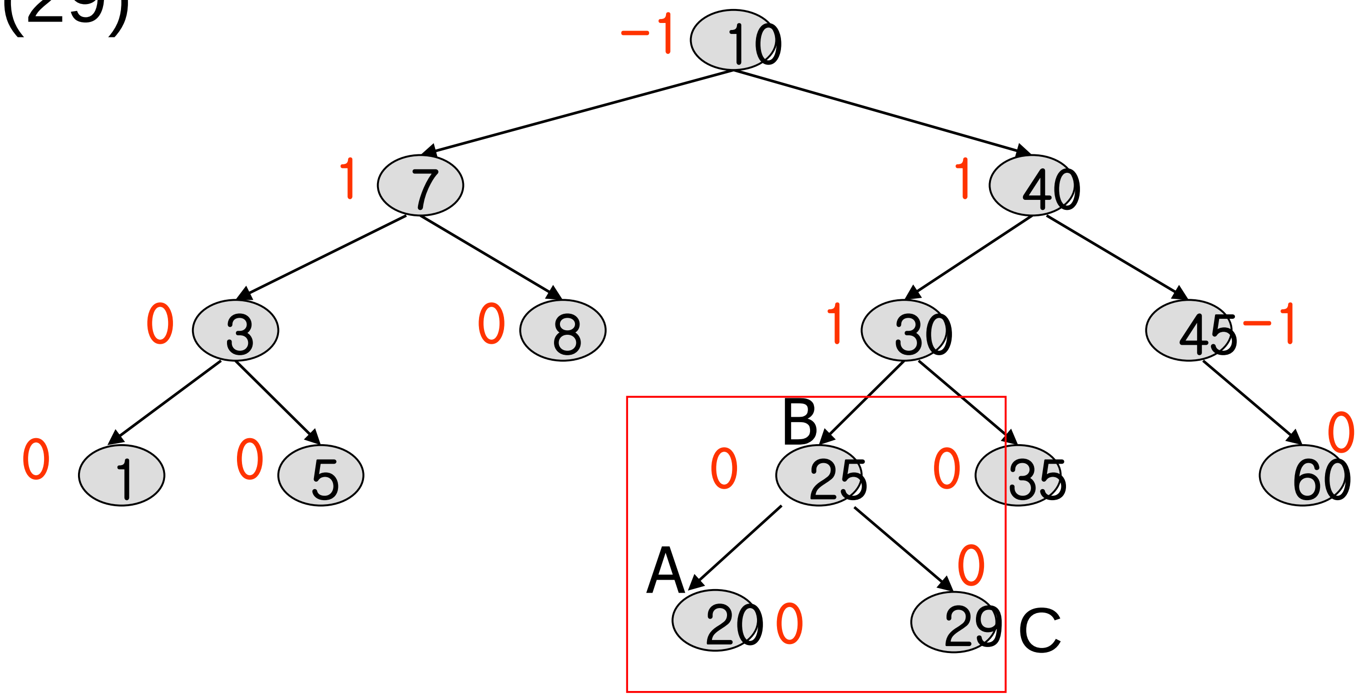
例：RR不平衡

Insert(29)

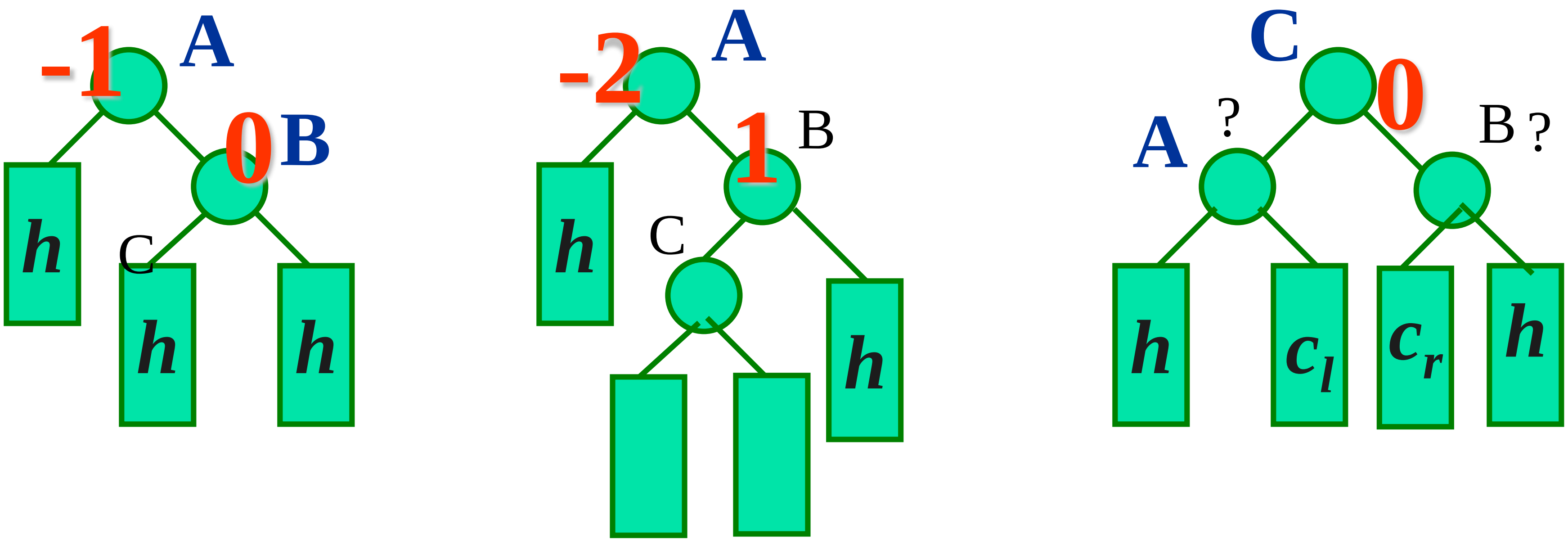


例：RR不平衡

Insert(29)



例： RL不平衡



(a)插入前 (b) 插入后 (c)旋转后

RL旋转

- 右左双旋转是左右双旋转的镜像。
- 在子树C中插入新结点，该子树高度增1。结点A的平衡因子变为-2，发生了不平衡。
- 从结点A起沿插入路径选取3个结点A、B和C，它们位于一条形如“〉”的折线上，需要进行先右后左的双旋转。
- 首先做LL旋转：以结点C为旋转轴，将结点B顺时针旋转，以C代替原来B的位置。
- 再做RR旋转：以结点C为旋转轴，将结点A反时针旋转，恢复树的平衡。

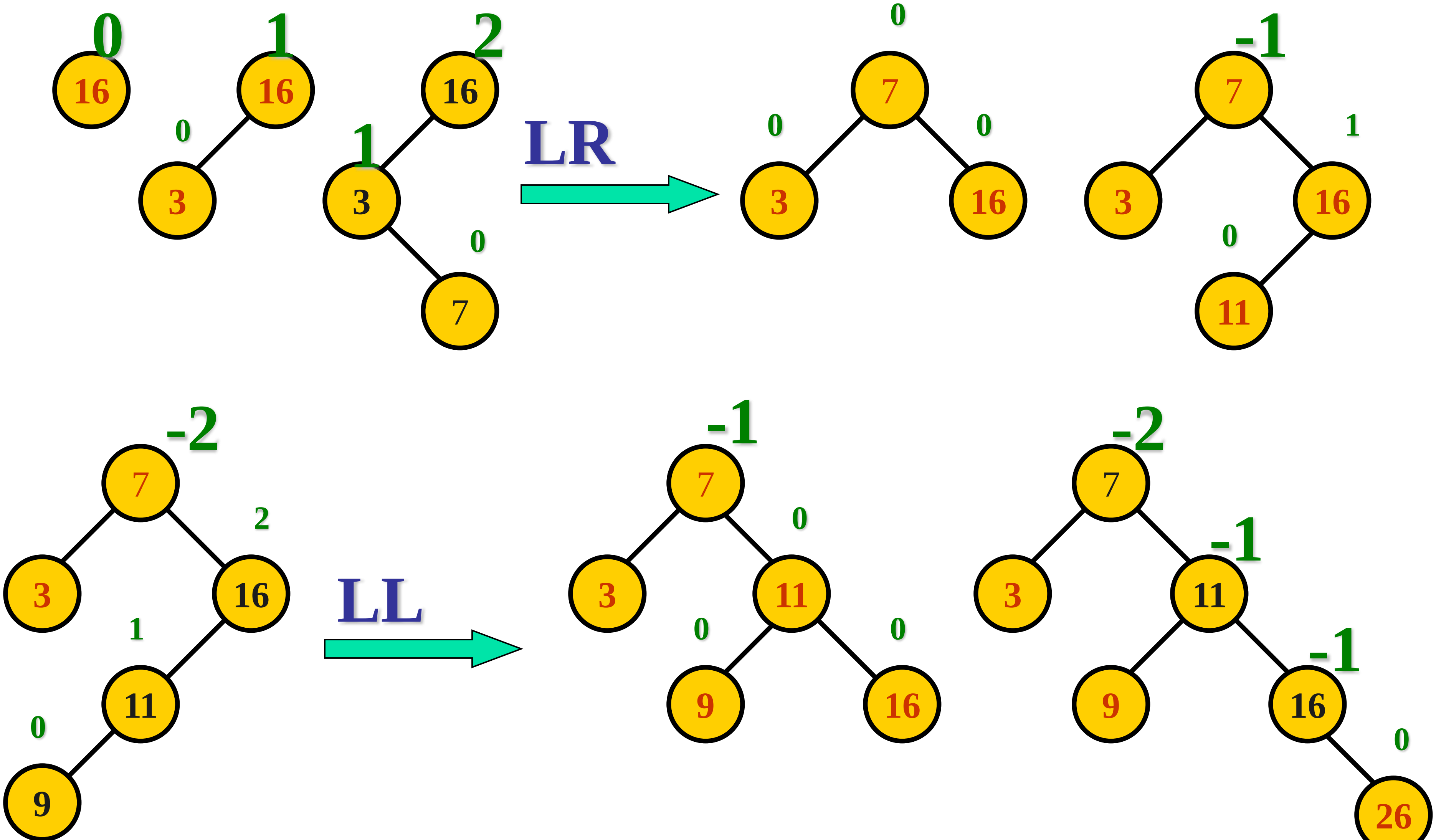
右单旋转（LL）的算法

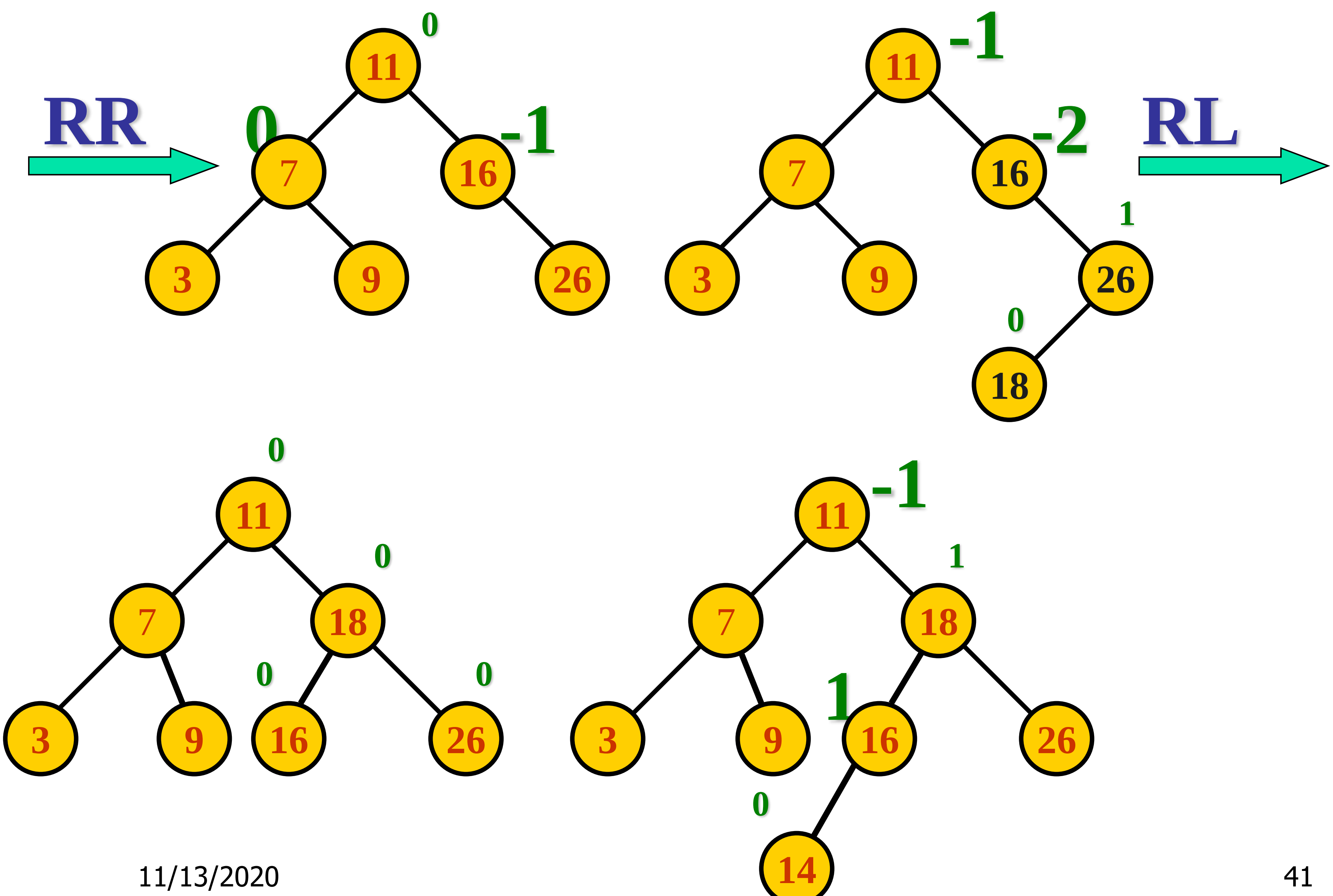
```
template <class Type>
void AVLTree<Type>::
RotateRight ( AVLNode<Type> *Tree,
      AVLNode<Type> * &NewTree) {
   NewTree = Tree→left;
   Tree→left = NewTree→right;
   NewTree→right = Tree;
}
```

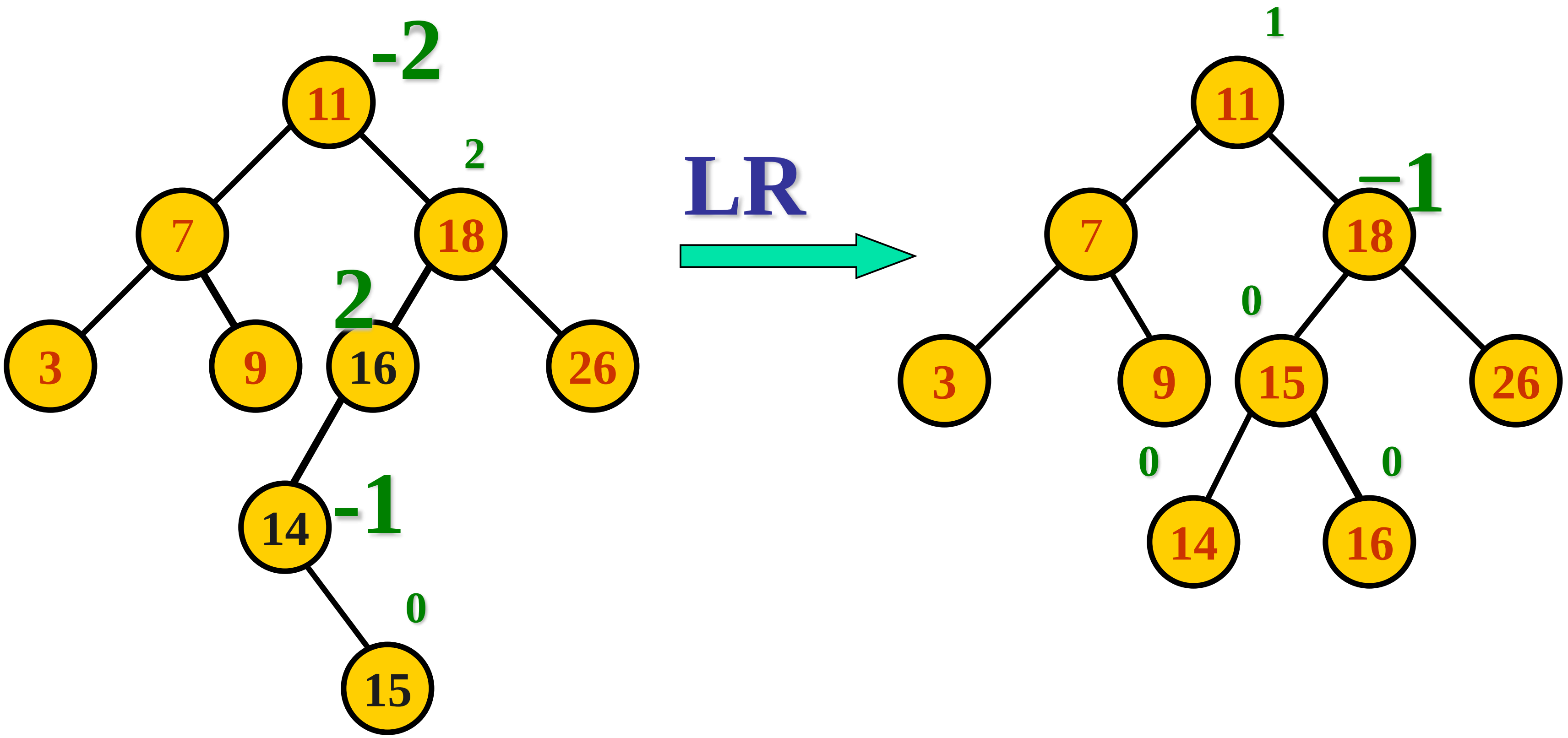
左单旋转（RR）的算法

```
template <class Type>
void AVLTree<Type> ::
RotateLeft ( AVLNode<Type> *Tree,
        AVLNode<Type> * &NewTree ) {
//左单旋转的算法
    NewTree = Tree→right;
    Tree→right = NewTree→left;
    NewTree→left = Tree;
}
```

例，输入关键码序列为 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }，插入和调整过程如下。



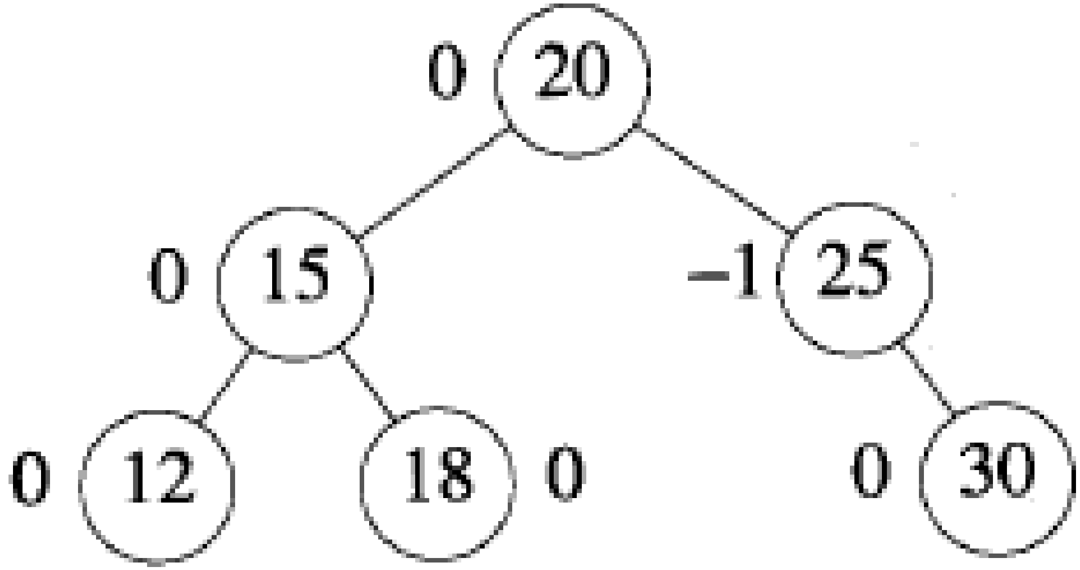

RR
RL
0
11
0
7
-1
16
3
9
26
-1
11
7
-2
16
3
9
1
26
0
18
0
11
7
0
18
3
9
0
16
0
26
-1
11
1
7
18
3
9
1
16
26
0
14



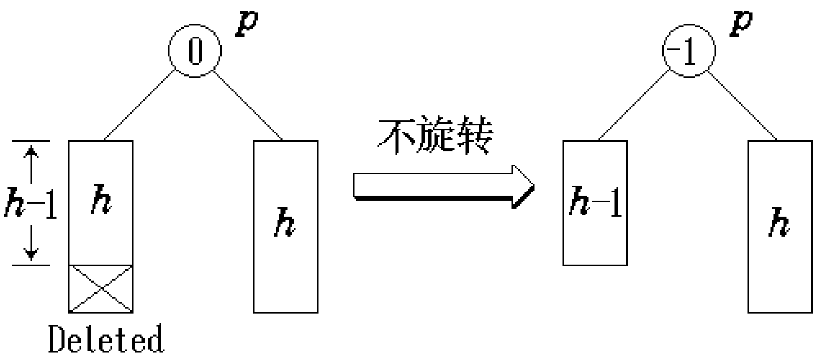
从空树开始的建树过程

AVL搜索树的删除

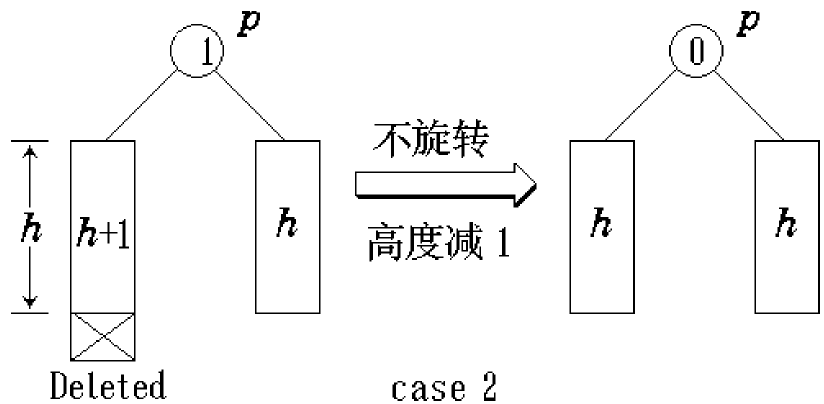
- 同普通二叉搜索树？



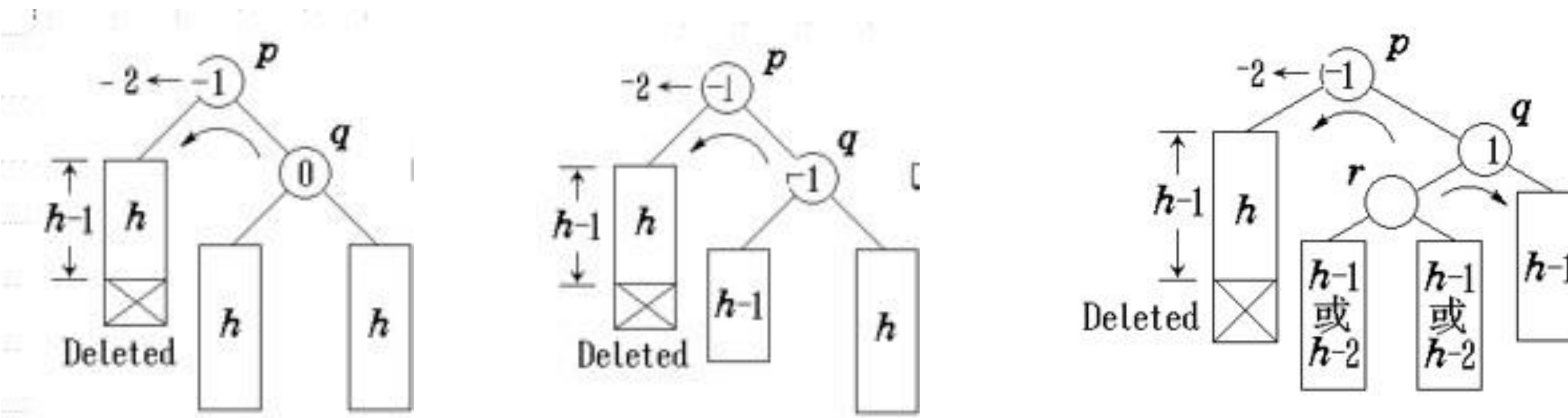
观察



case 1



case 2



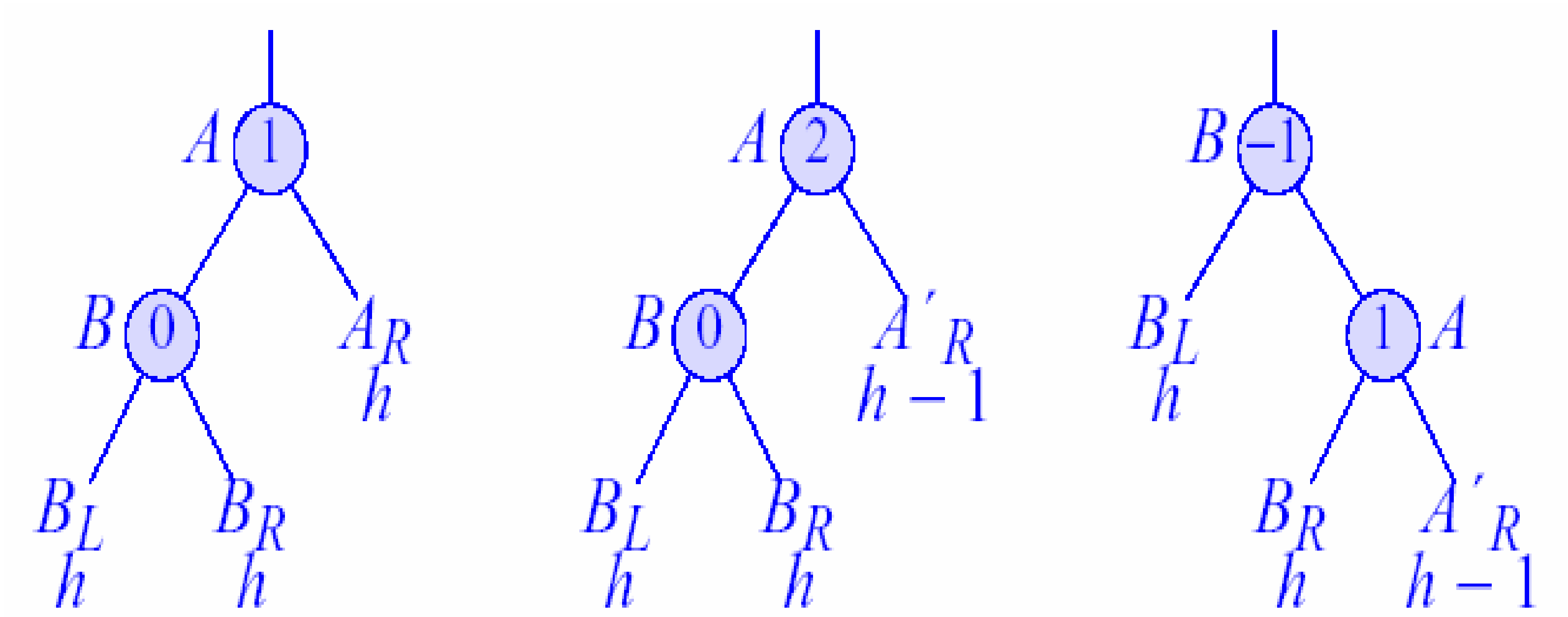
case 3

AVL搜索树的删除

- 通过执行二叉搜索树的删除，可从AVL搜索树中删除一个元素。但也会导致产生不平衡树。
- 设q:删除节点的父节点。
- 由删除操作导致产生不平衡树的几种现象:
  - 1)如果q新的平衡因子是0，那么它的高度减少了1，需要改变它的父节点(如果有的话)和其他某些祖先节点的平衡因子。
  - 2)如果q新的平衡因子是－1或1，那么它的高度与删除前相同，无需改变其祖先的平衡因子值。
  - 3)如果q新的平衡因子是－2或2，那么树在q节点是不平衡的。
- 由删除操作产生的不平衡分为六种类型：R0, R1, R-1, L0, L1, L-1。

R0 旋转

设A是从q到根节点的路径第一个平衡因子变为2或－2的节点。

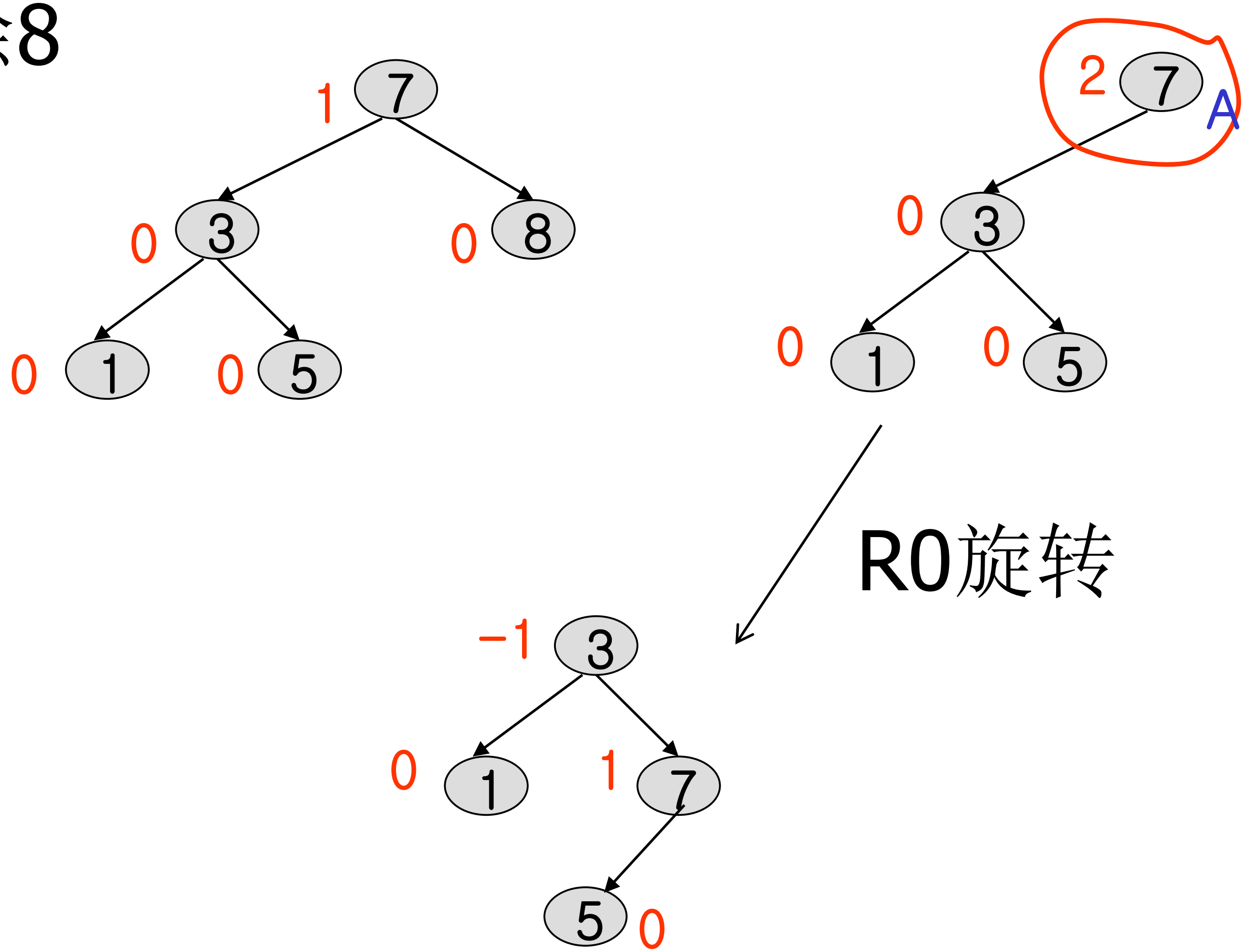


(a) 删除之前　　(b) 从A_R中删除之后　　(c) R0 旋转之后

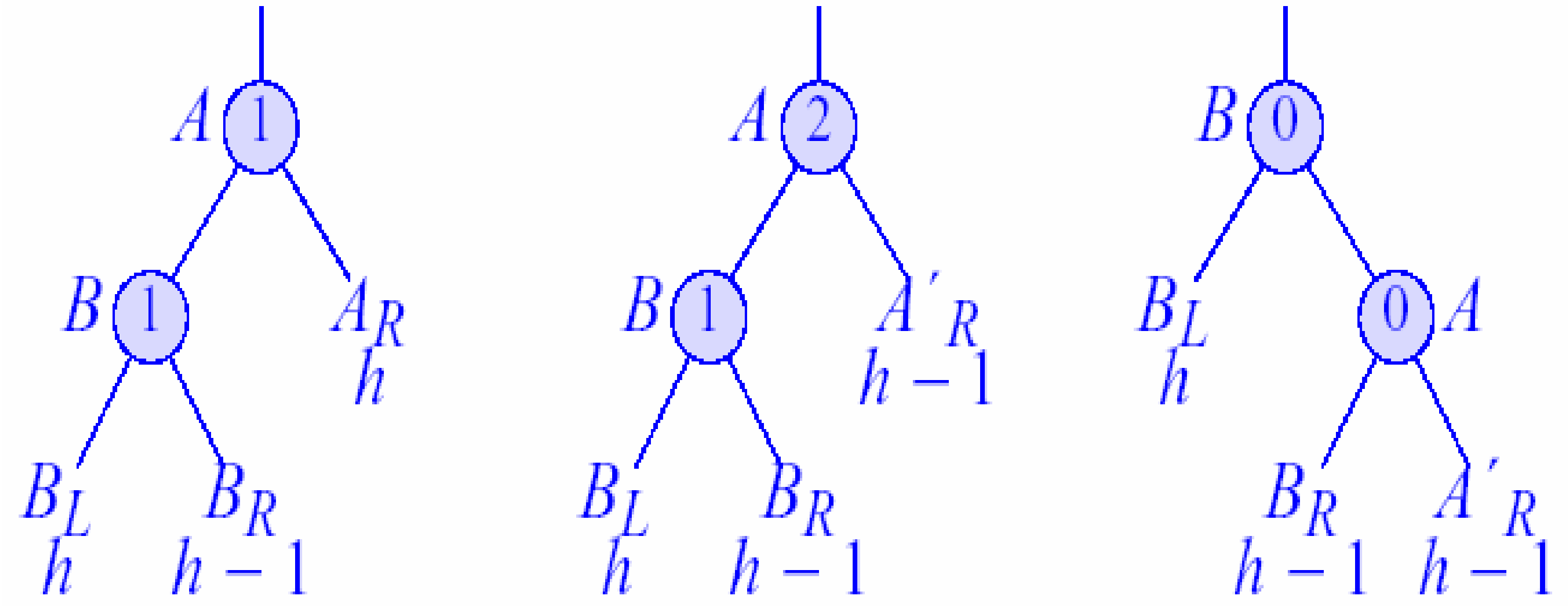
- R0 旋转:单旋转

R0 旋转例

- 删除8



R1 旋转

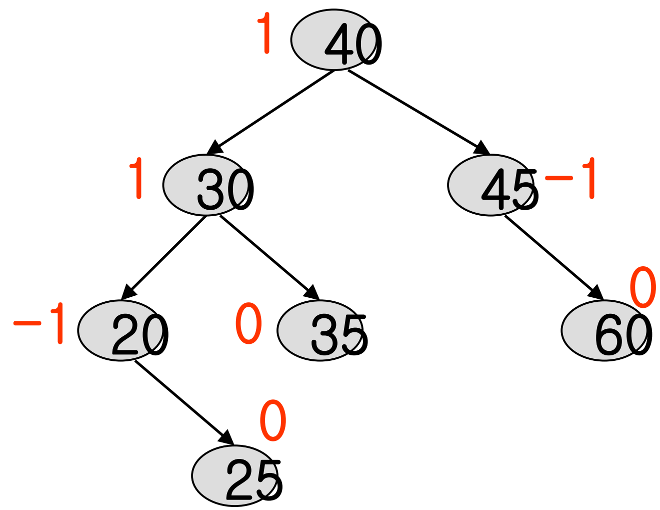


(a) 删除之前　　(b) 从A_R中删除之后　　(c) R1 旋转之后

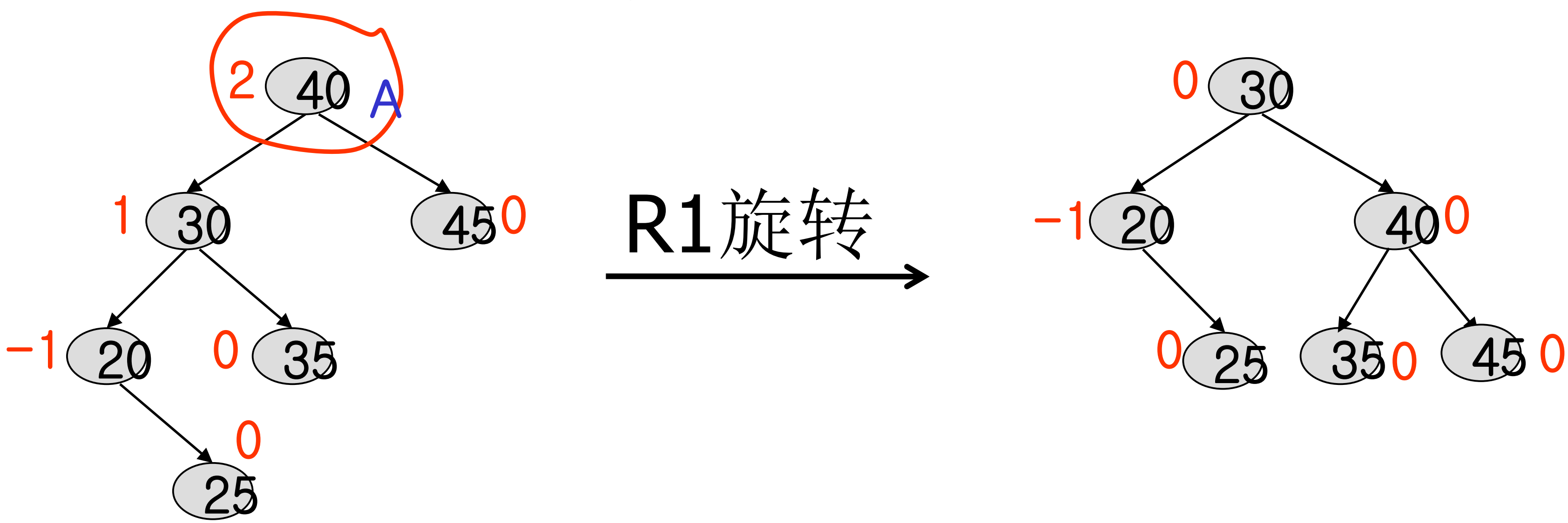
- R1 旋转:单旋转

R1 旋转例

- 删除60

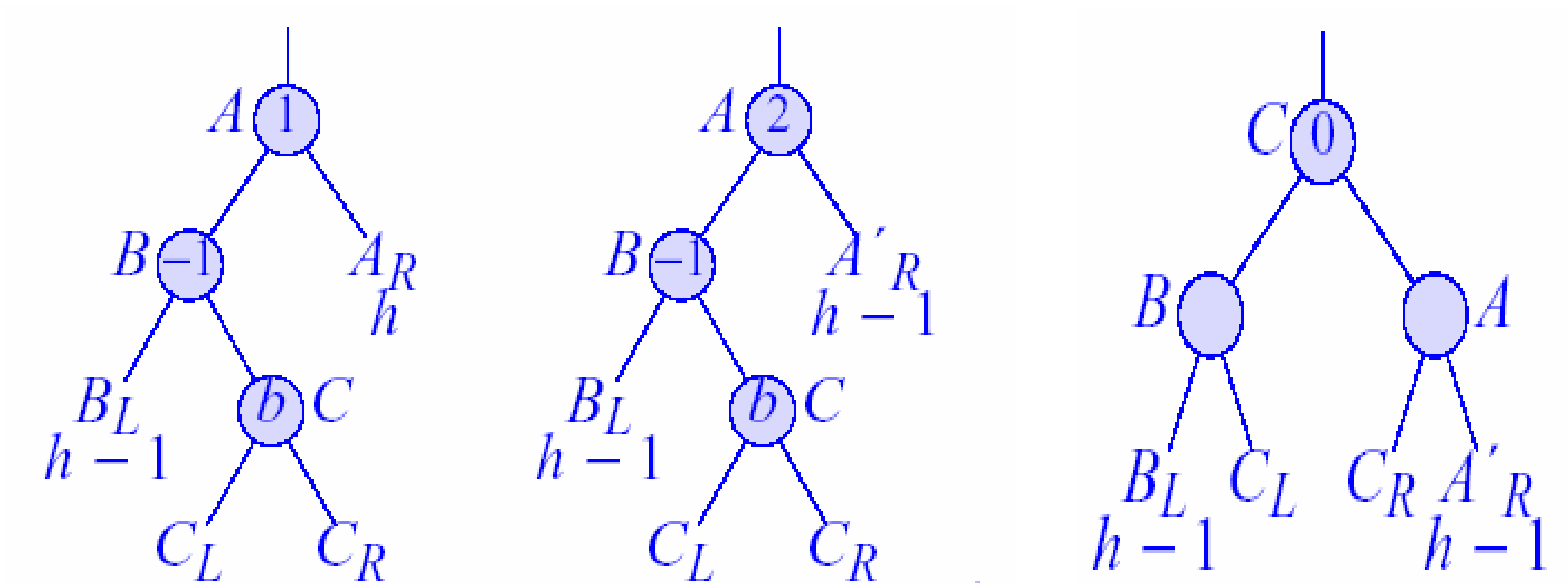


1
40
1
30
45
−1
−1
20
0
35
60
0
0
25



2
40
A
1
30
45
0
−1
20
0
35
0
25
R1旋转
0
30
−1
20
40
0
0
25
35
0
45
0

R-1 旋转

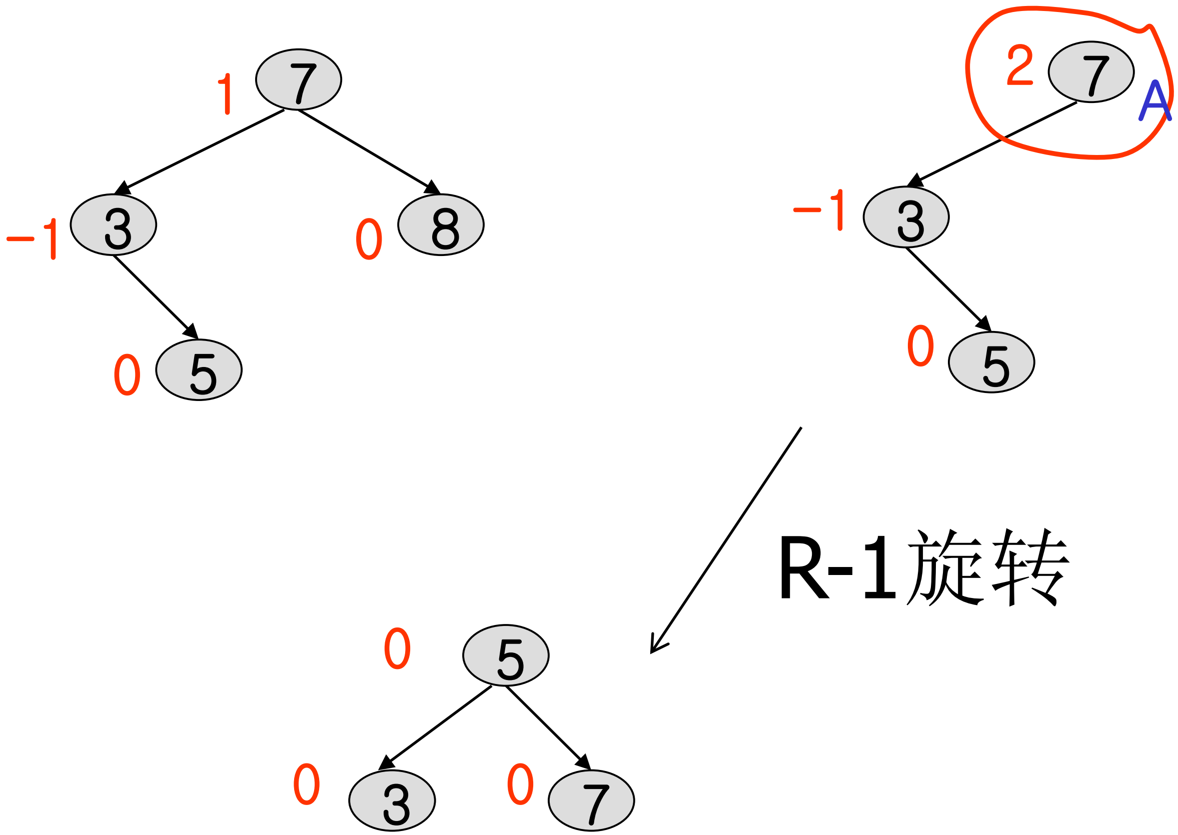


(a) 删除之前　　(b) 从A_R中删除之后　　(c) R-1 旋转之后

- R-1 旋转:双旋转

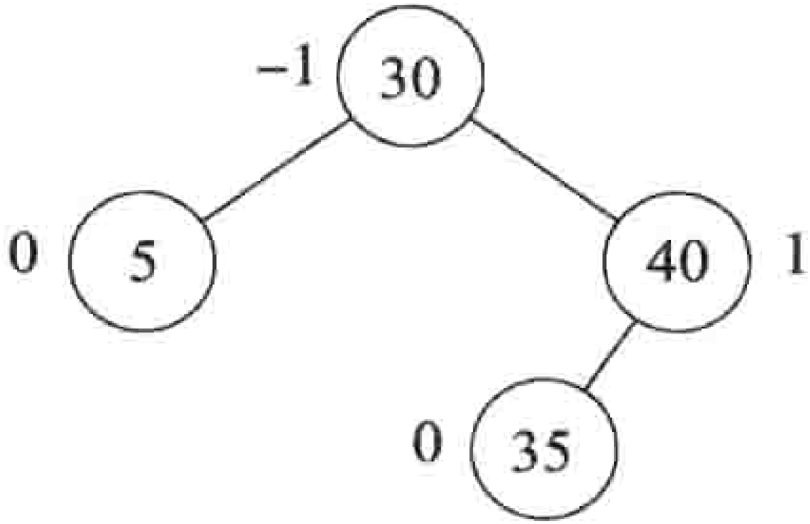
R-1 旋转例

- 删除8

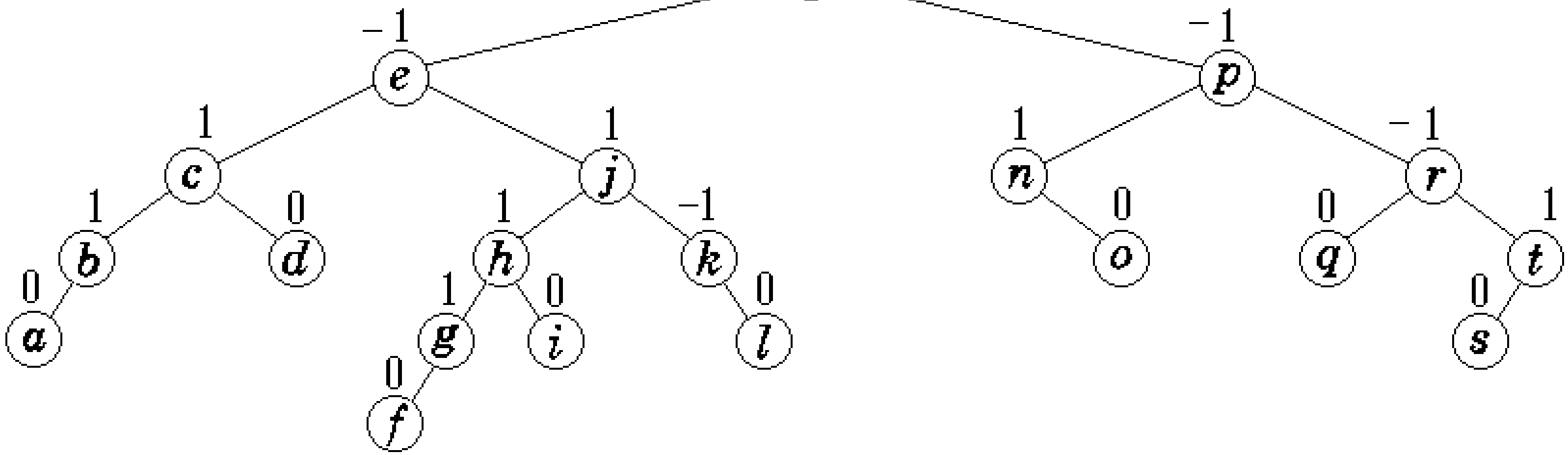


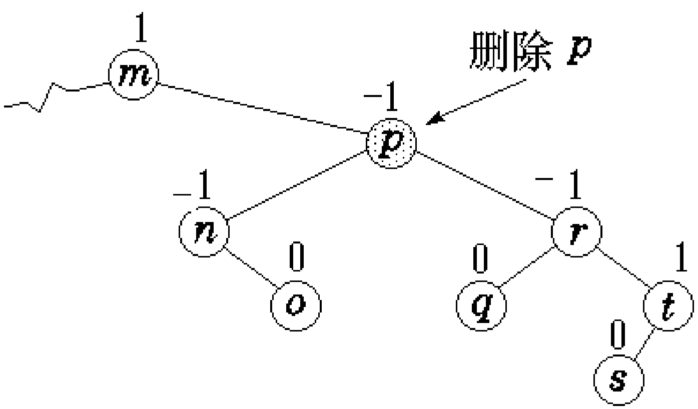
例：

- 删除5

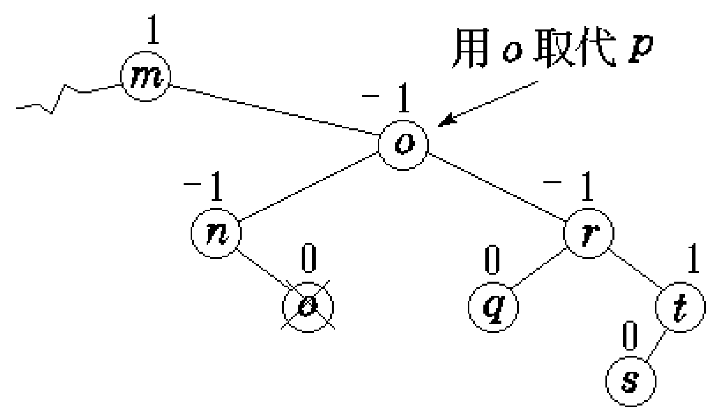


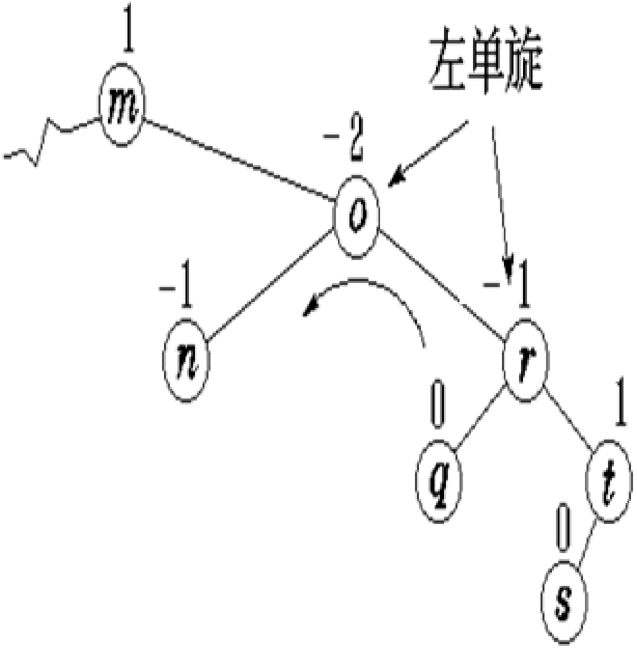
树的初始状态



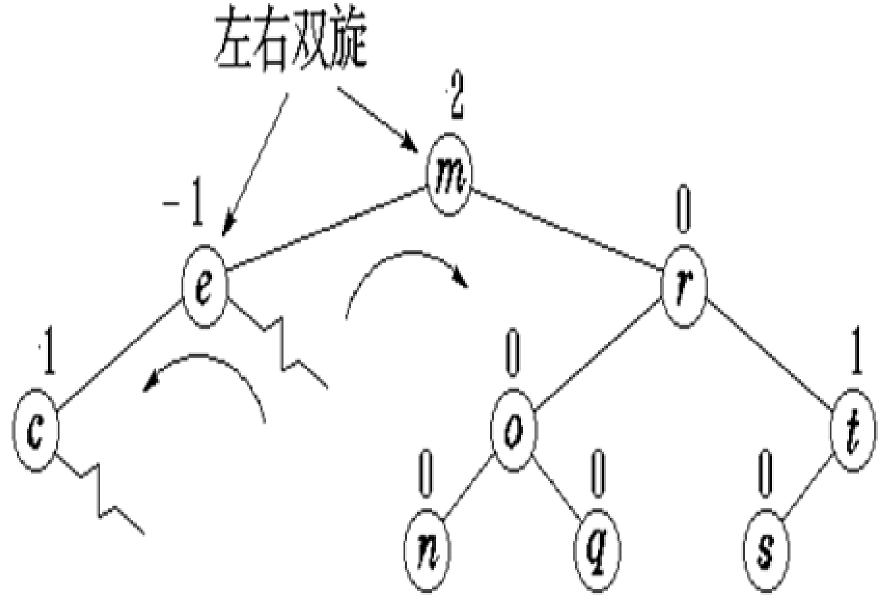


寻找 p 的中序下的直接前驱 o，用 o 取代 p，删除 o，平衡旋转

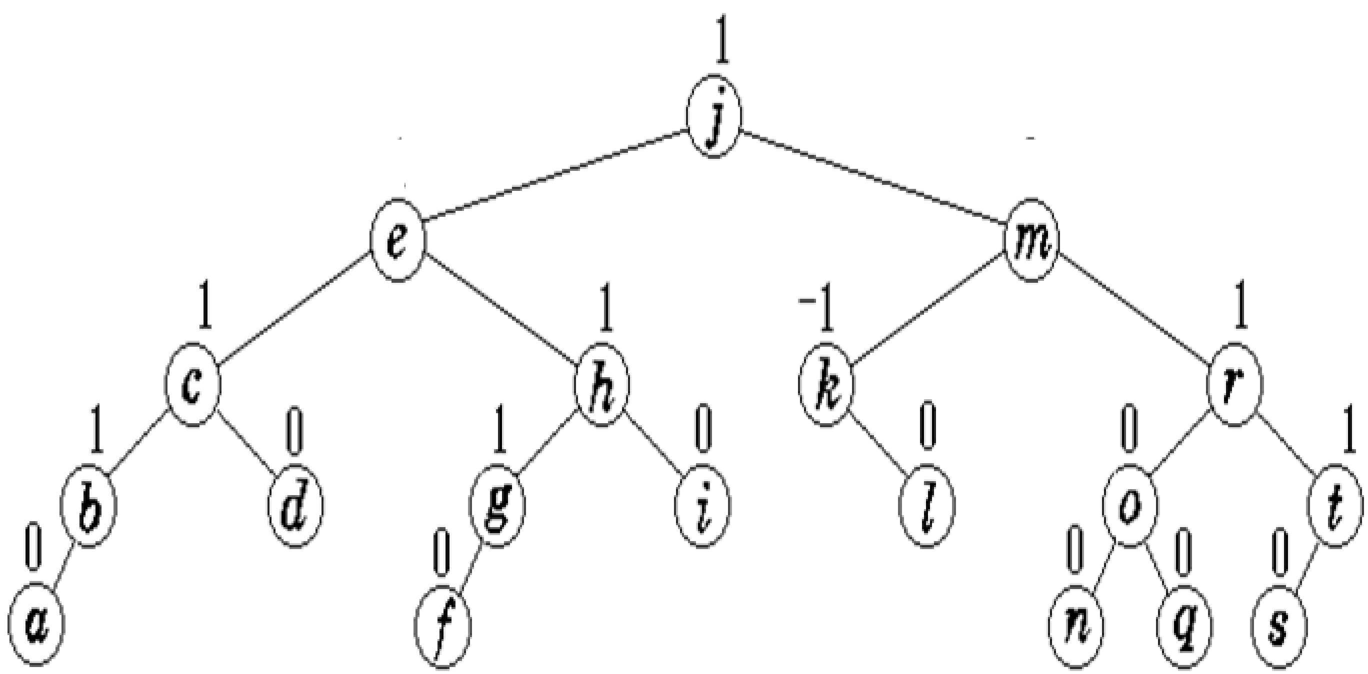




以 r 为旋转轴作左单旋，子树的高度减 1 m 发生不平衡

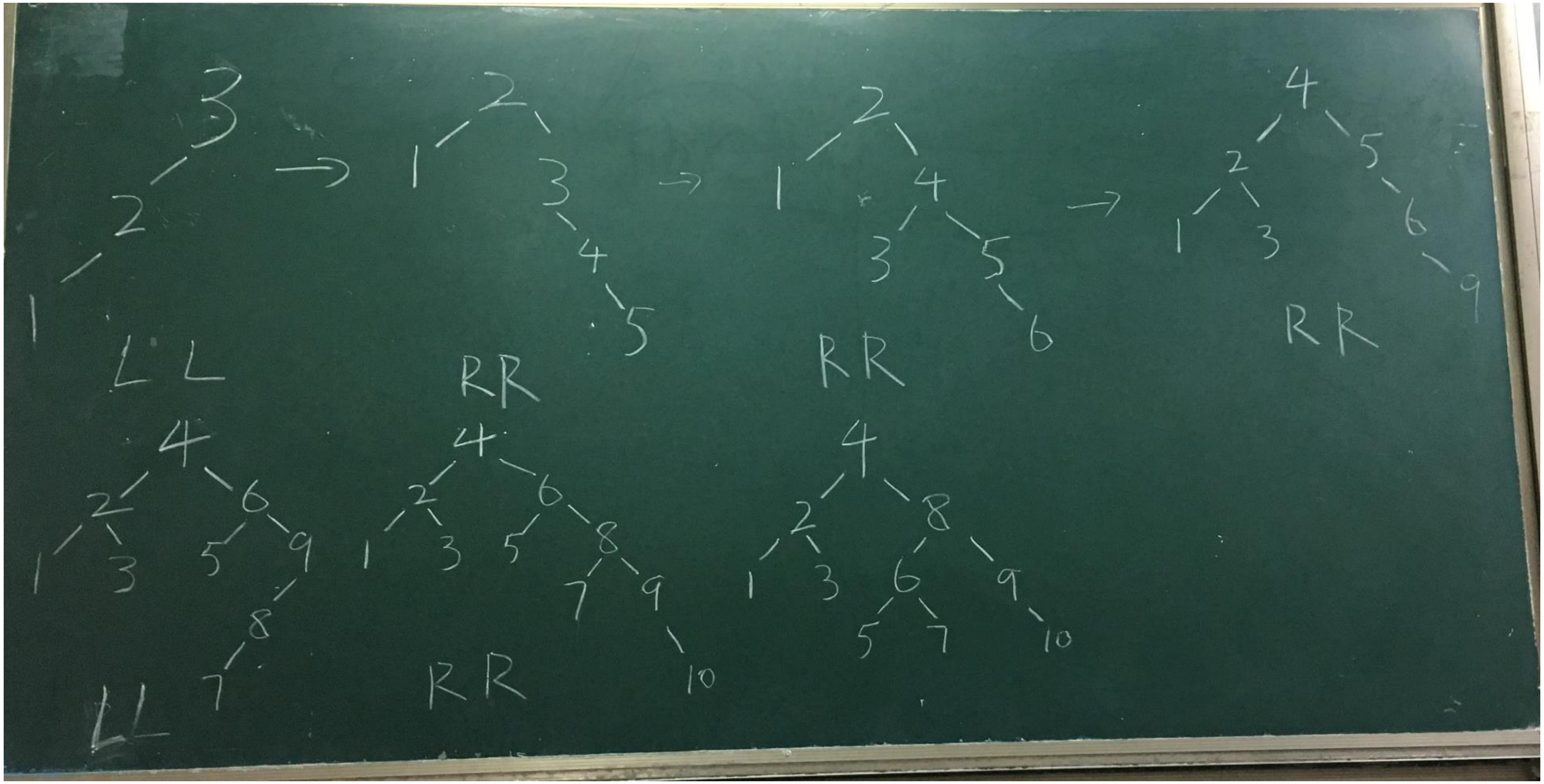


首先以 j 为旋转轴作左单旋，再以 j 为旋转轴作右单旋，让 e 成为 j 的左子女，m 成为 j 的右子女。树的高度减 1



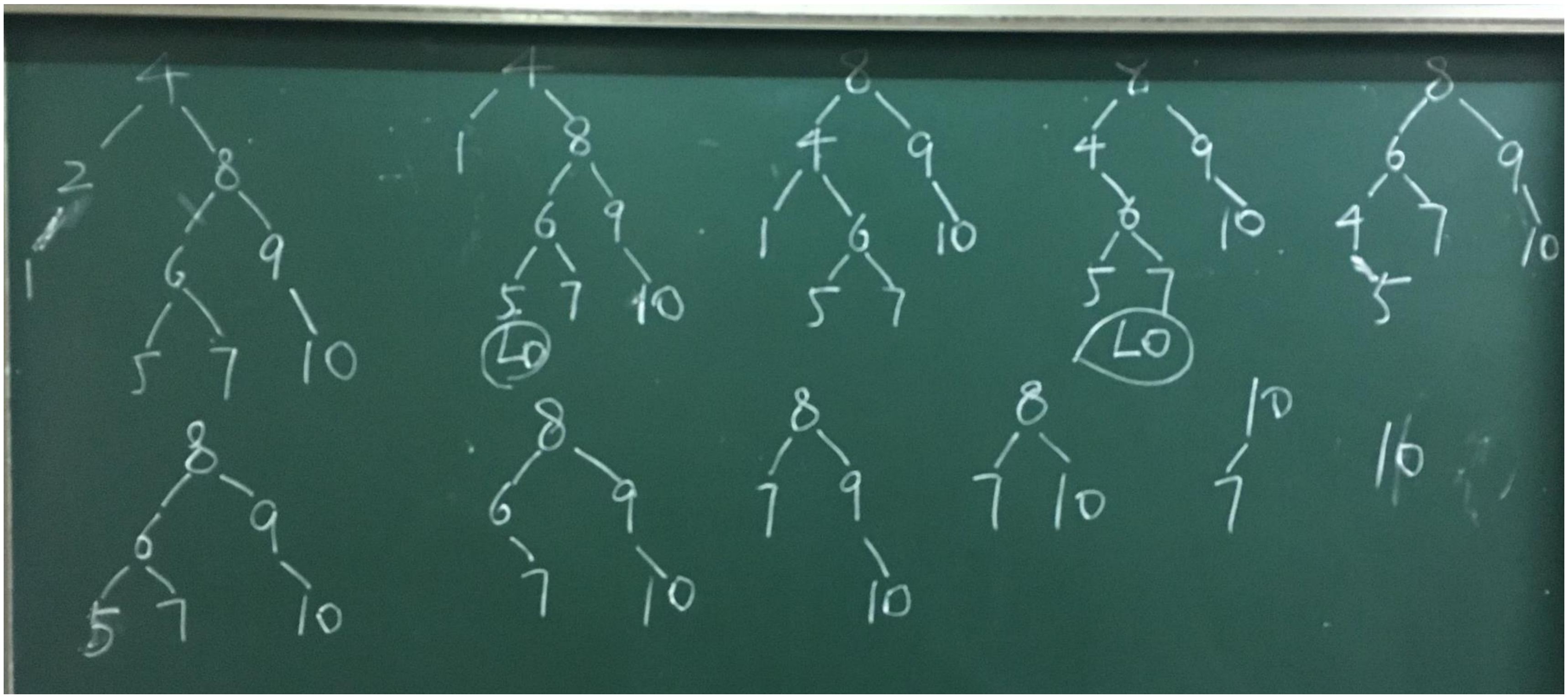
练习

按照3 2 1 4 5 6 9 8 7,10的字典顺序，构建avl搜索树


LL
RR
RR
RR
LL
RR

练习

按照3 2 1 4 5 6 9 8 7,10的字典顺序，一次删除avl搜索树的结点



4
2
1
8
6
9
5
7
10

作业

按照12个月份的字典顺序，构建二叉搜索树和avl搜索树