计算机领域本科教育教学改革试点
工作计划（“101计划”）研究成果

数据结构

俞勇、张铭、陈越、韩文弢

上海交通大学、北京大学、浙江大学、清华大学

第 11 章 查找

11.3 AVL树

林劼
电子科技大学

提纲

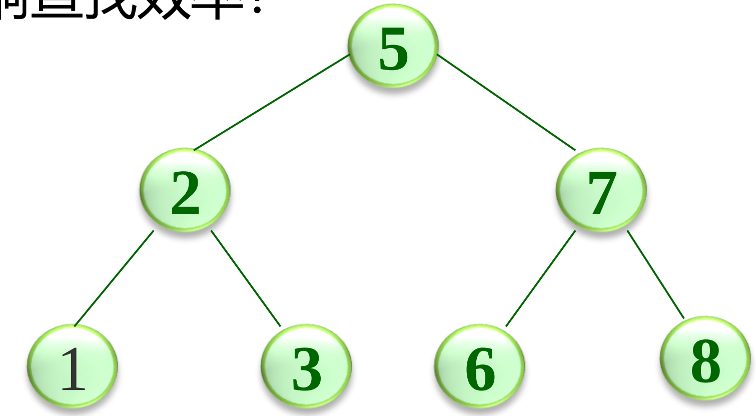
11.3.1　AVL树

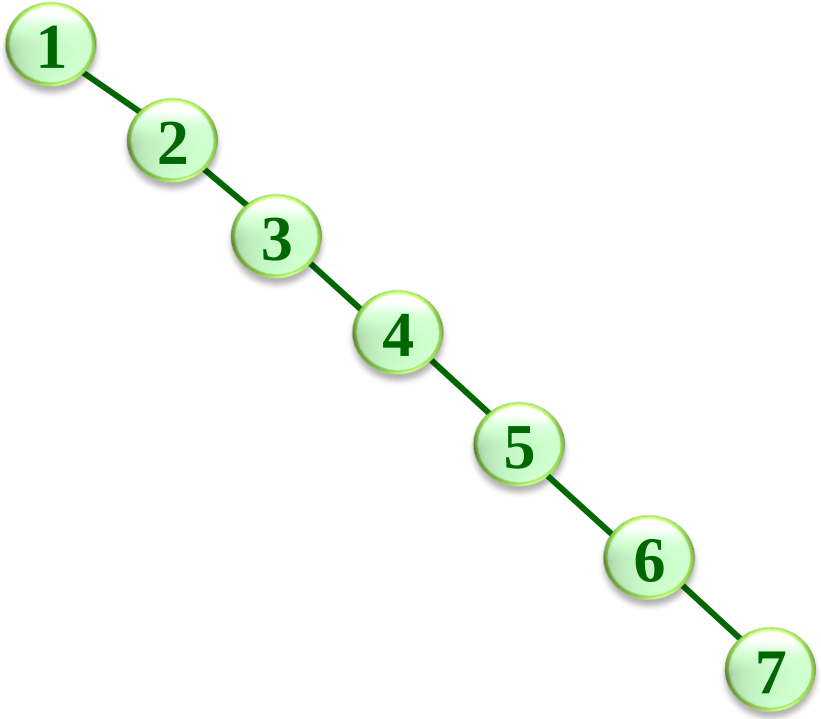
11.3.2　AVL树旋转

11.3.3　作业

11.3 AVL树

二叉查找树BST： 结构上的“不平衡”，可能严重影响查找效率！





ASL =（1+2+2+3+3+3+3）/ 7
= 2.4

完全二叉树： 树的高度最低，ASL最小！

ASL =（1+2+3+4+5+6+7）/ 7
= 4

使BST始终保持完全二叉树的形状难度大，因此可以降低对树结构平衡性的要求，比如AVL树等

11.3 AVL树

AVL树定义

AVL树或者是一棵空树，或者是具有下列性质的二叉树：

它的左、右子树都是平衡二叉树（AVL树），

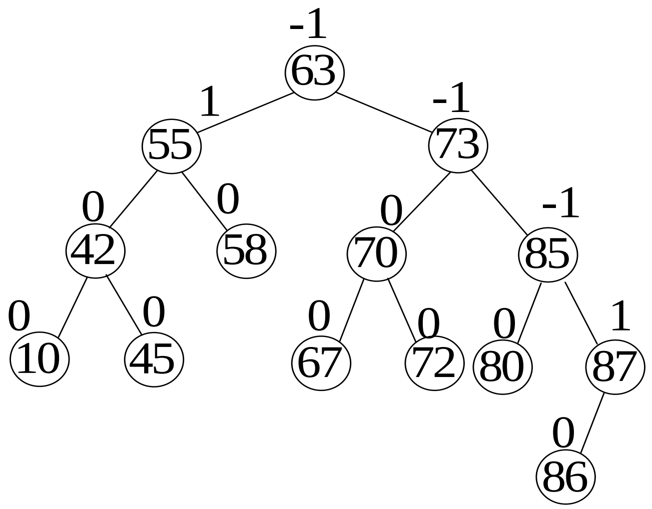
并且左、右子树的深度之差不超过1。

- AVL树由苏联数学家G. M. Adelson-Velsky和Evgenii Landis发明
- AVL树是计算机科学中最早被发明的自平衡二叉（查找）树（其它如红黑树、2-3树、B+树等)
- 在AVL树中，任一结点的左右子树的高度差不超过1，因此它也被称为高度平衡树
- 增加和删除元素时，可以通过一次或多次旋转操作，实现重新平衡
- 查找、插入和删除在最坏情况下的时间复杂度都是 O(log(n))

11.3 AVL树

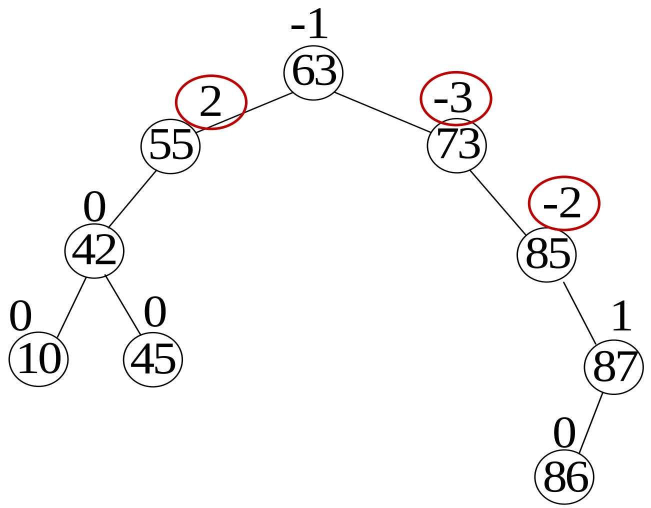
AVL树示例

平衡因子：左子树高度 - 右子树高度



(a) 平衡二叉树

AVL树



(b) 非平衡二叉树

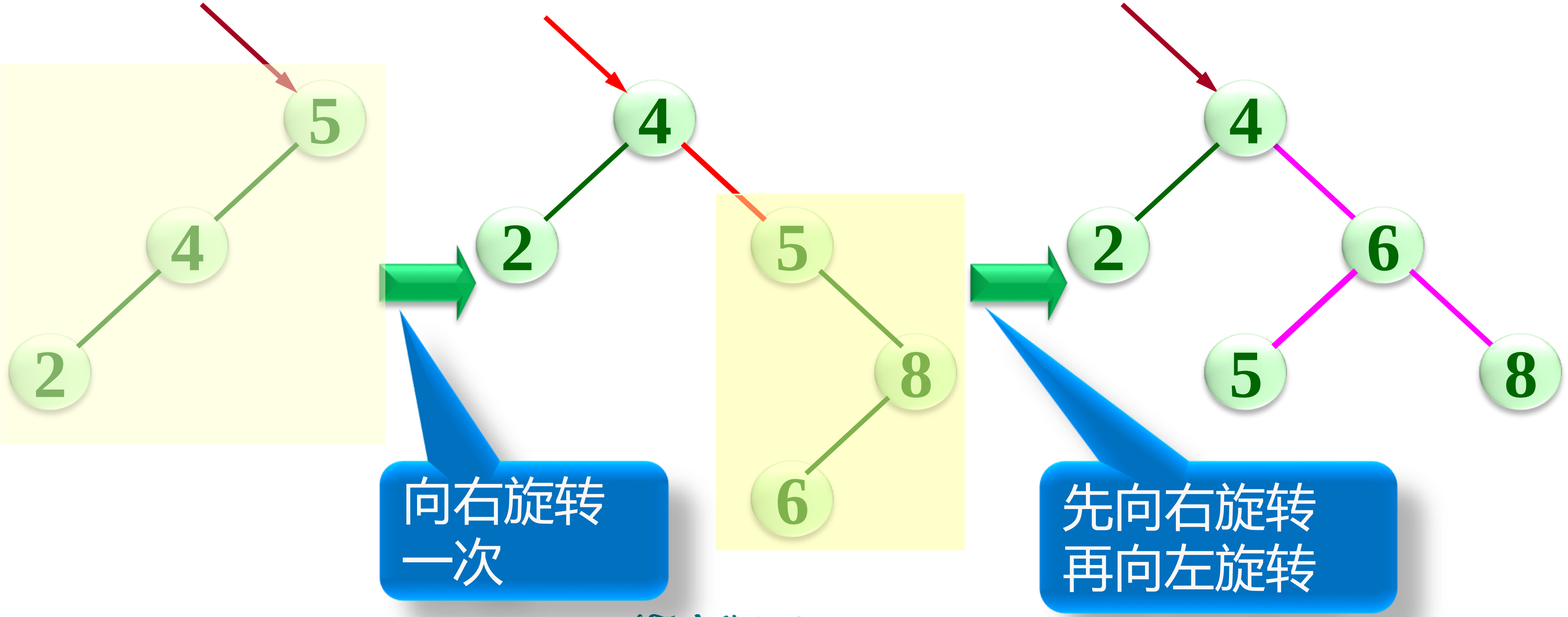
非AVL树

11.3 AVL树

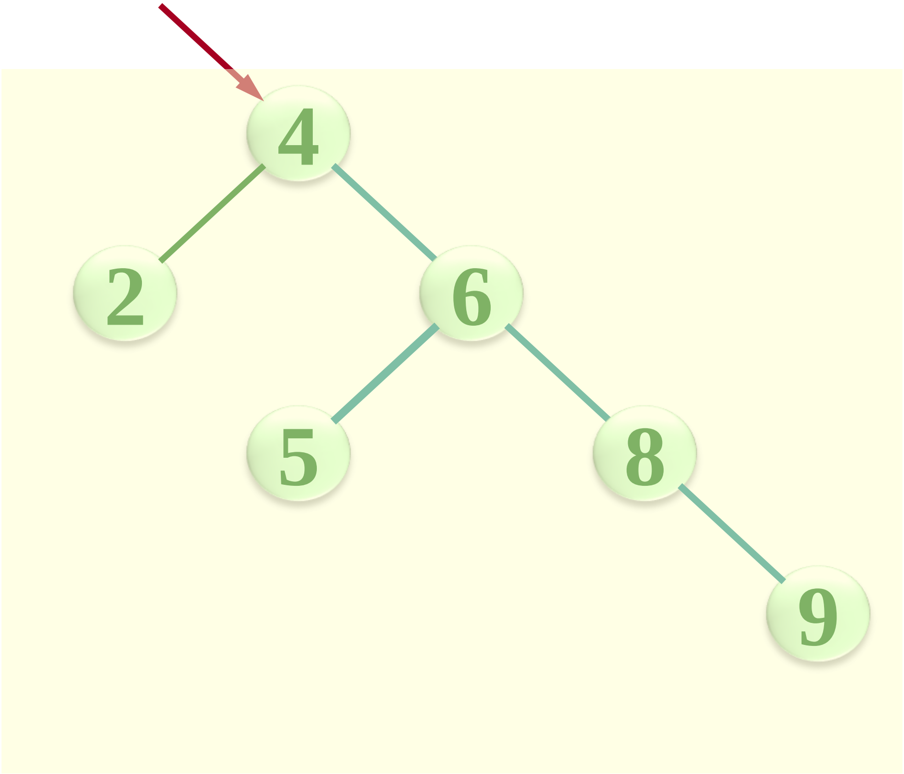
构造平衡二叉树的方法

在插入过程中，采用平衡旋转技术。

例如:依次插入的关键字为5, 4, 2, 8, 6, 9

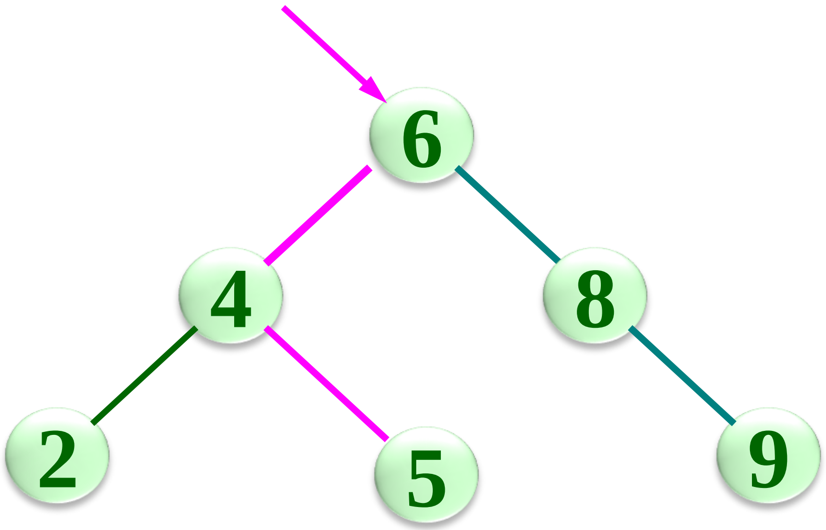


11.3 AVL树



继续插入关键字 9

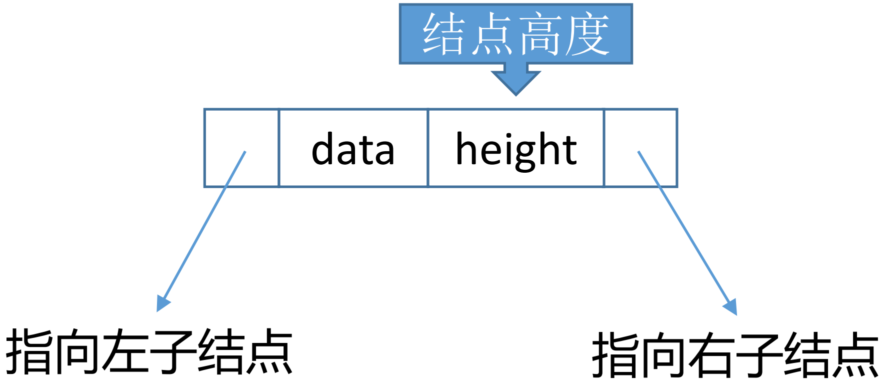
向左旋转一次



11.3 AVL树

AVL树关键数据结构：结点

左指针域	数据域	高度域	右指针域
left	*data*	*height*	*right*



二叉链表

O(1)

算法：GetHeight(tree)
输入：AVL树tree
输出：AVL树（根结点）的高度

```
if tree = NIL then
|  return 0   //空树高度为0
end
return tree.height   //直接返回高度域的值
```

O(1)

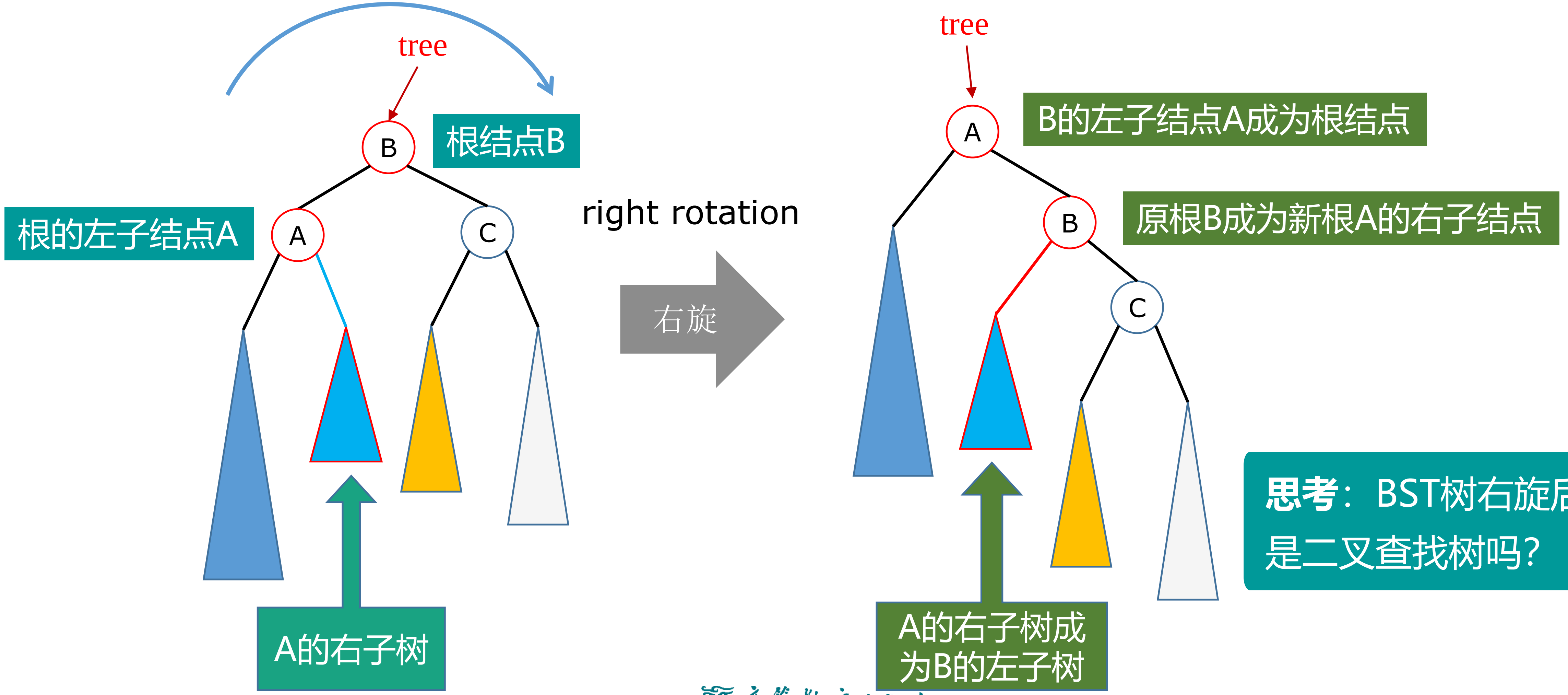
算法：UpdateHeight(tree)
输入：AVL树tree
输出：重新计算AVL树（根结点）的高度

```
if tree ≠ NIL then
|  left_ht ← GetHeight(tree.left)      //左子树高度
|  right_ht ← GetHeight(tree.right)  //右子树高度
|  tree.height ← Max(left_ht, right_ht) + 1
end                       //更新结点高度
```

基本操作

11.3 AVL树

AVL树关键操作1：右旋转 right rotation

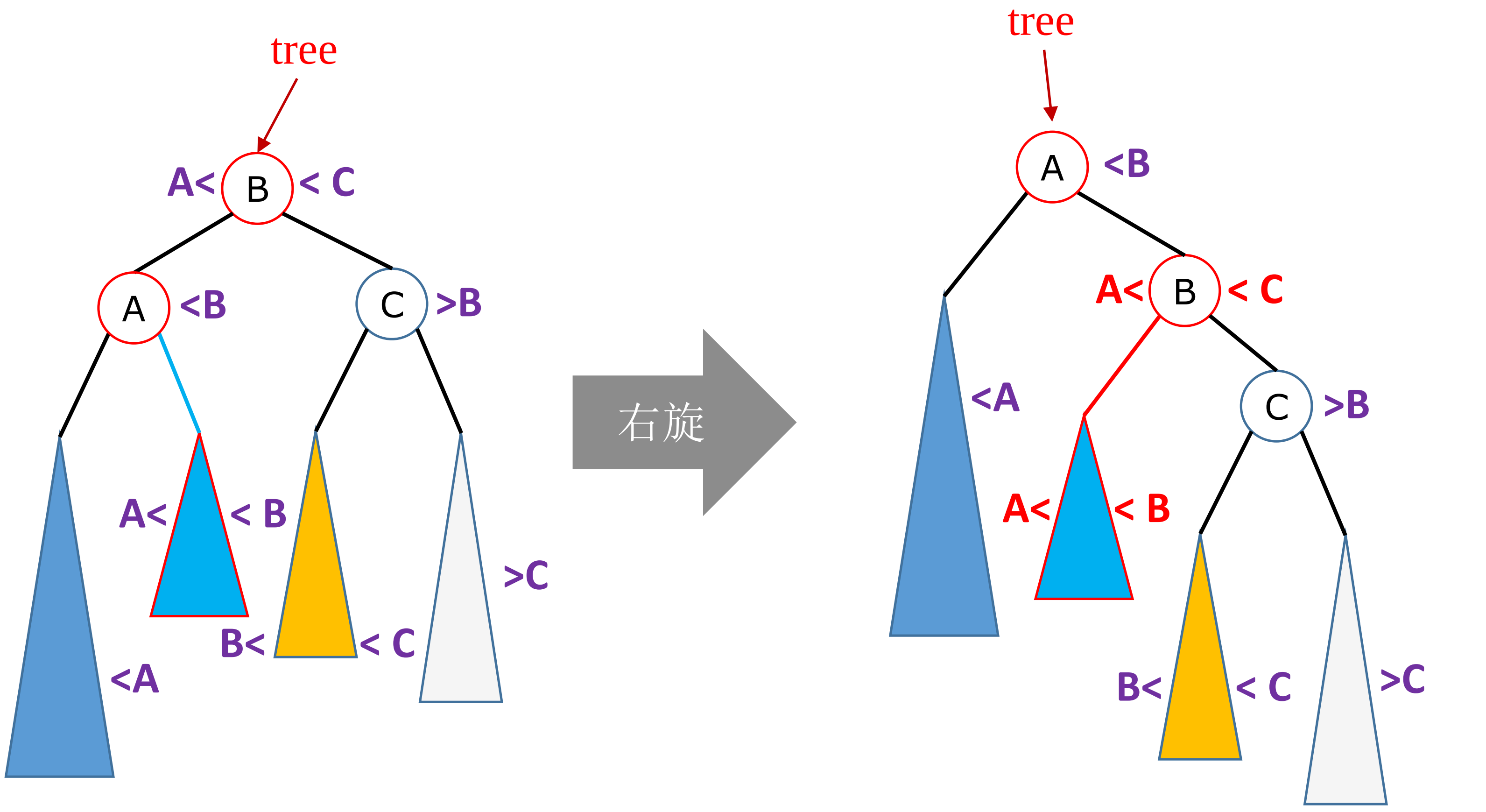


思考：BST树右旋后还是二叉查找树吗？

11.3 AVL树

AVL树关键操作1：右旋转 right rotation

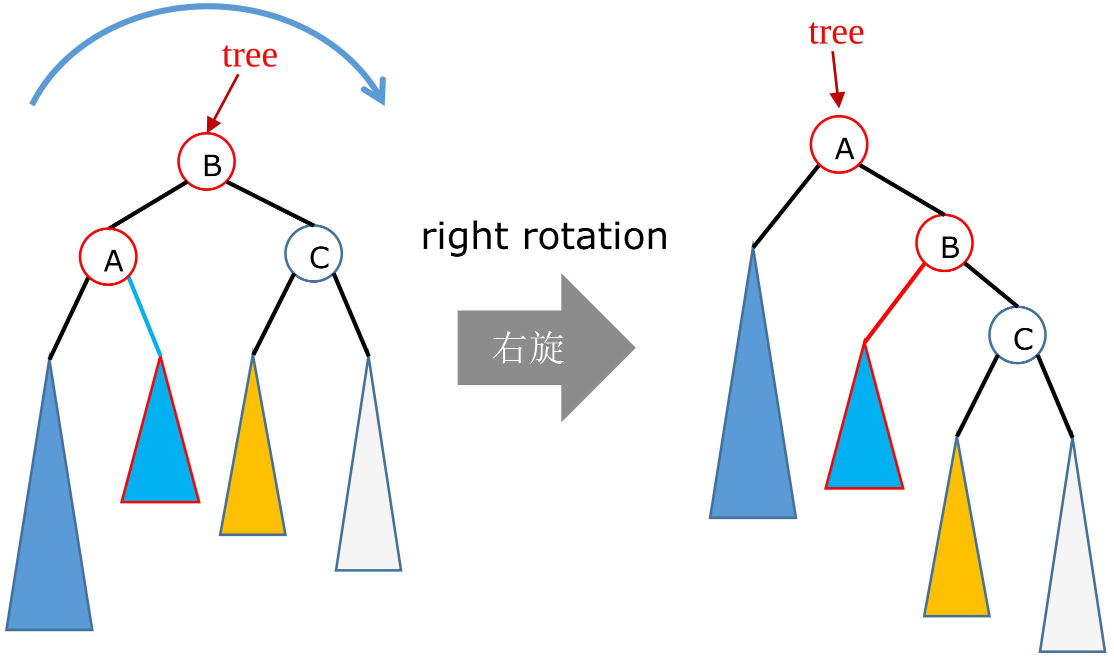
思考：BST树右旋后还是二叉查找树吗？



BST！

11.3 AVL树

AVL树关键操作1：右旋转 right rotation



算法：RightRotate(tree)
输入：AVL树tree (tree ≠ NIL 且 tree.left ≠ NIL)
输出：右旋AVL树，返回根结点

1. node ← tree.left //指向根的左子结点
2. tree.left ← node.right
3. UpdateHeight(tree) //更新tree高度（左子树有变化）
4. node.right ← tree
5. UpdateHeight(node) //更新node高度（右子树有变）
6. **return** node //返回新根结点

时间复杂度：O(1)

思考：右旋使结点A的深度减1，而结点B和C的深度加1。那么对各结点的高度有何影响？

多选题 1分

设置

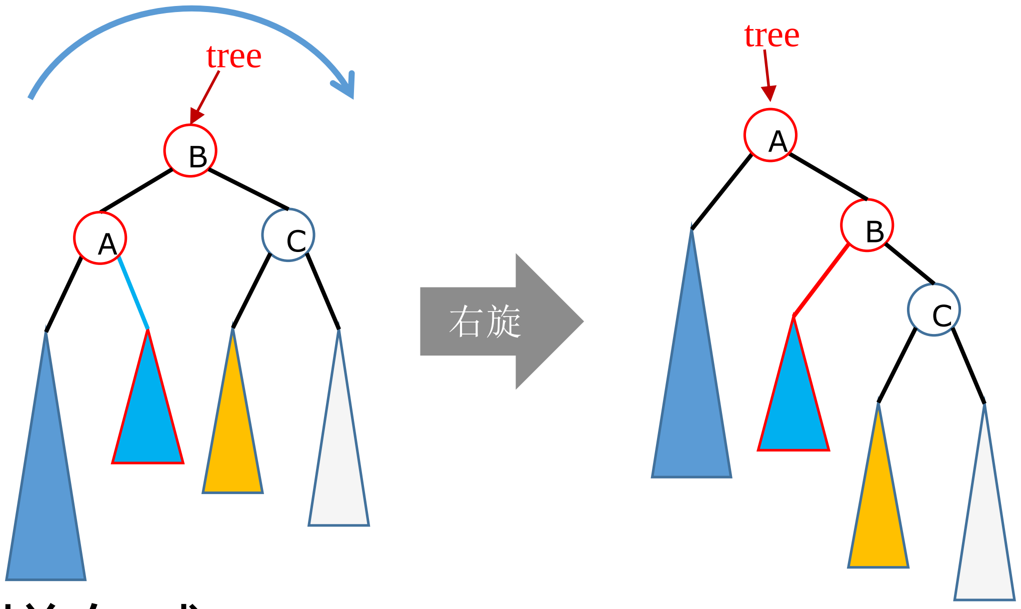
右旋过后，结点A\B\C的高度会有哪些变化？选择正确的描述。

A A的高度只会增不会减

B B的高度只会减不会增

C C的高度一定不变

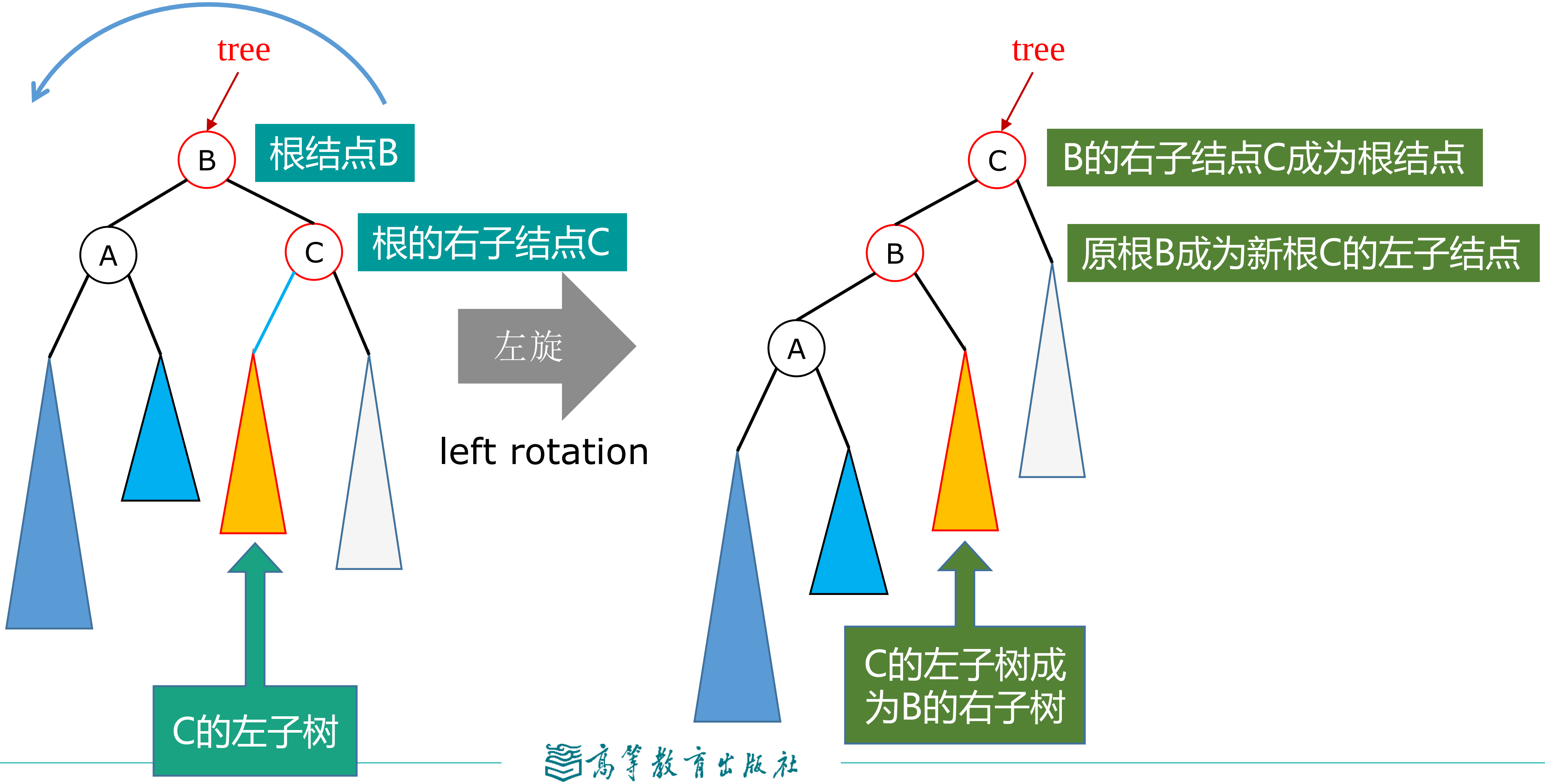
D 除了C不变之外，A和B可以有增有减



提交

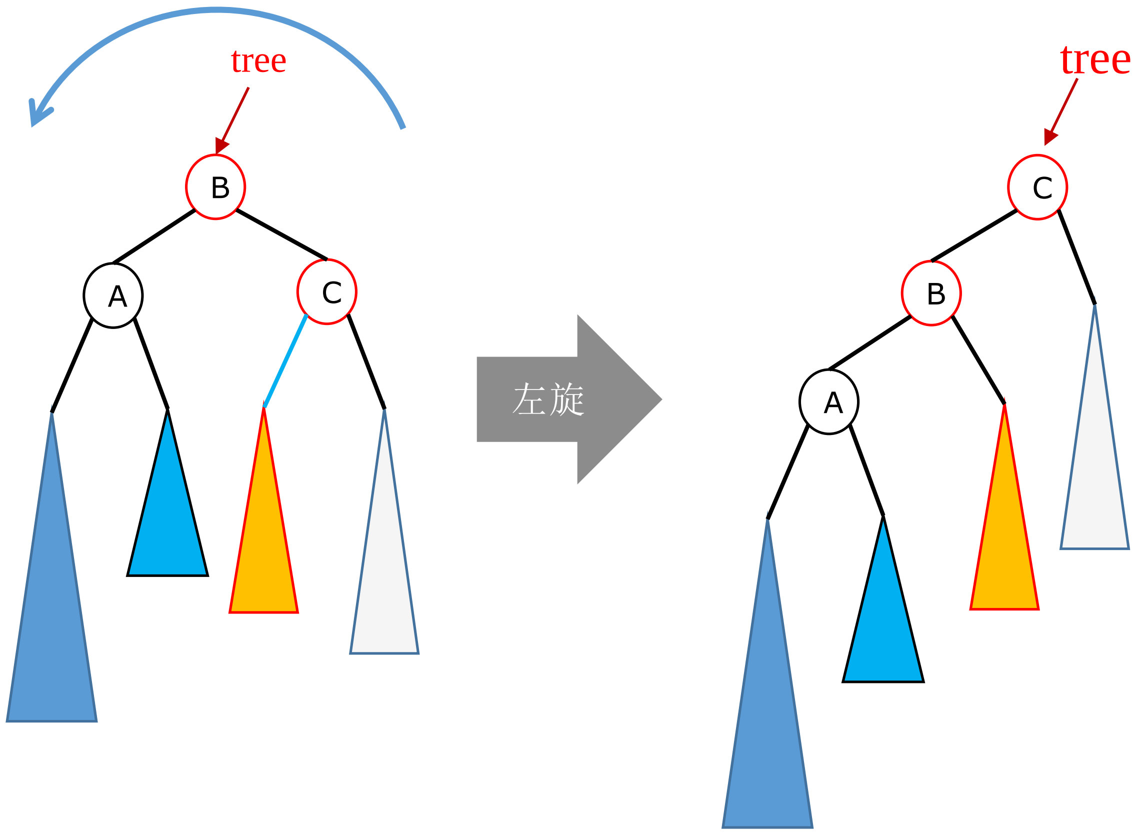
11.3 AVL树

AVL树关键操作2：左旋转 left rotation



11.3 AVL树

AVL树关键操作2：左旋转 left rotation



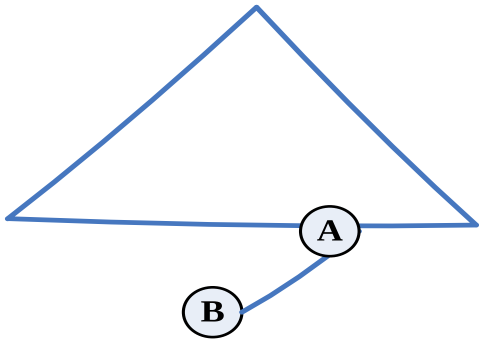
算法：LeftRotate(tree)
输入：AVL树tree (tree ≠ NIL 且 tree.left ≠ NIL)
输出：左旋AVL树，返回根结点

1. node ← tree.right //指向根的右子结点
2. tree.right ← node.left
3. UpdateHeight(tree) //更新tree高度（右子树有变化）
4. node.left ← tree
5. UpdateHeight(node) //更新node高度（左子树有变）
6. **return** node //返回新根结点

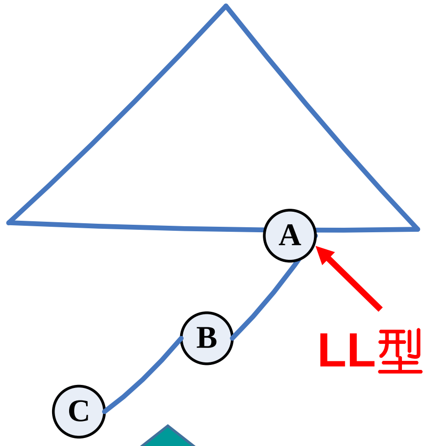
11.3 AVL树

失衡状况的分类：

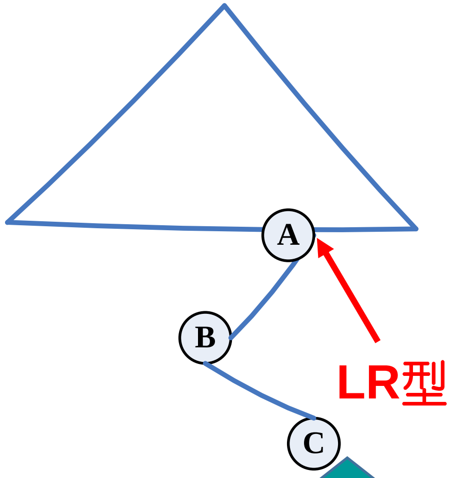
(1)一颗平衡的二叉树



(2)插入结点C后的情况



- 结点的左子树与右子树的高度差等于2
- 以左子结点为根的树中：左子树不低于右子树

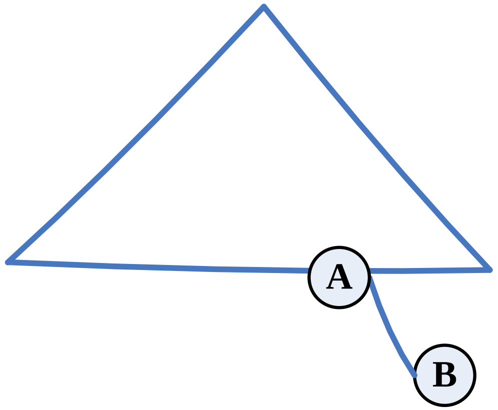


- 结点的左子树与右子树的高度差等于2
- 以左子结点为根的树中：左子树低于右子树

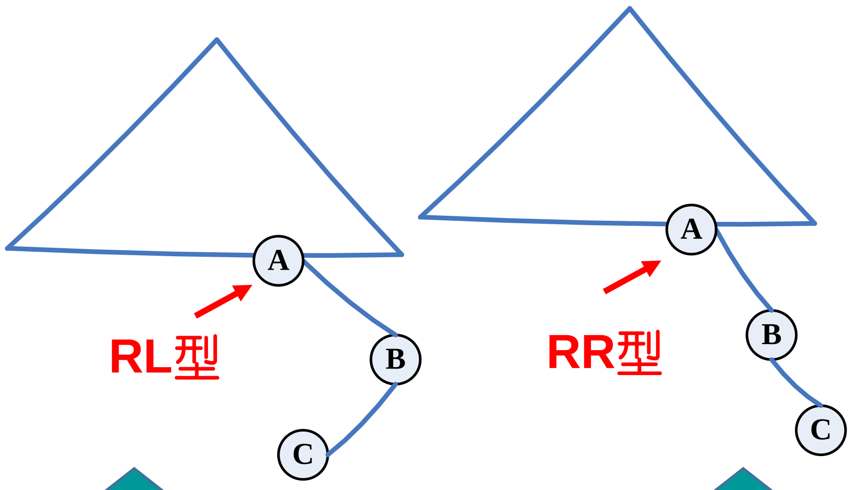
11.3 AVL树

失衡状况的分类

(1)一颗平衡的二叉树



(2)插入结点C后的情况

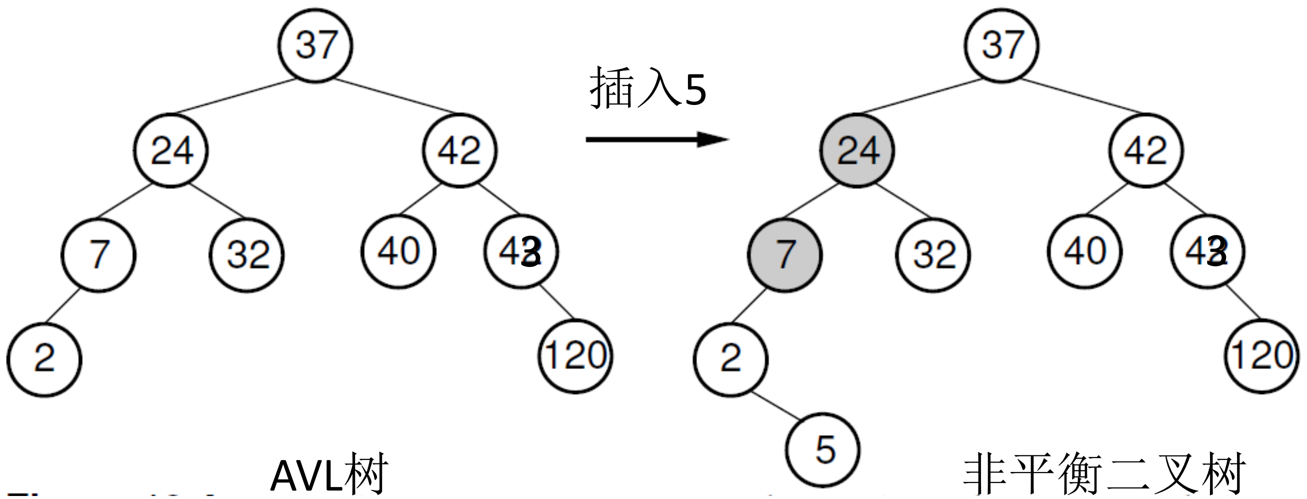


- 结点的右子树与左子树的高度差等于2
- 以右子结点为根的树中：左子树比右子树高

- 结点的右子树与左子树的高度差等于2
- 以右子结点为根的树中：右子树不低于左子树

11.3 AVL树

失衡状况的示例



- 以结点7为根的子树不平衡，属于LR类型
- 以结点24为根的子树不平衡，属于LL类型

重要性质：对AVL树插入新结点，则失衡结点（子树）只出现在从根到新插入结点的路径上！

例如：从新结点5到根结点37的路径含结点5,2,7,24,37

失衡的调整从新插入结点开始由下至上（递归）调整其祖先结点，直到根结点为止！

11.3 AVL树

失衡调整旋转平衡处理

(1)单向右旋（LL）

(2)单向左旋（RR）

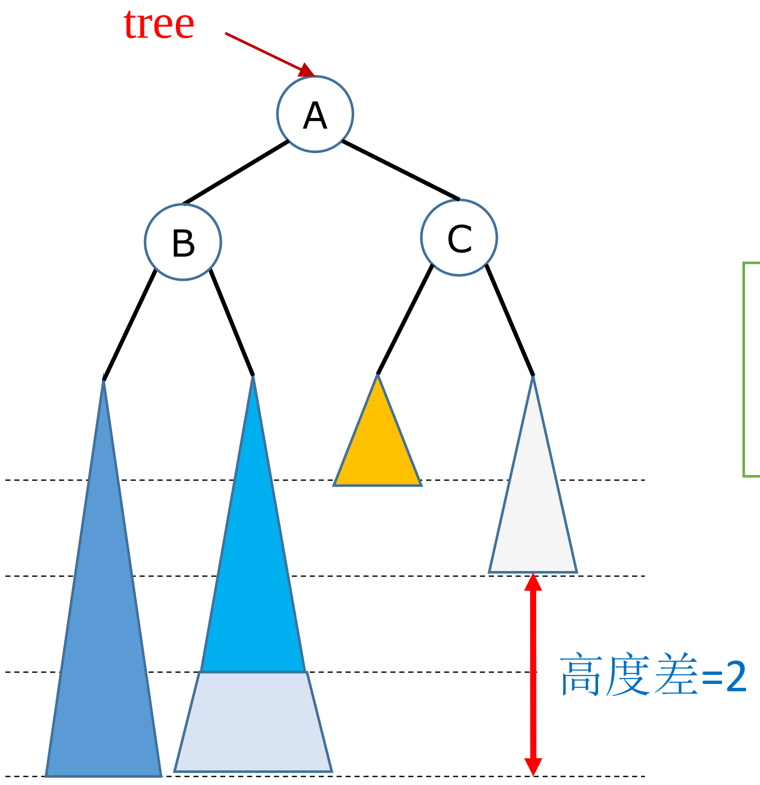
(3)先左后右旋转（LR）

(4)先右后左旋转（RL）

11.3 AVL树

失衡调整旋转平衡处理

(1)单向右旋（LL）



平衡处理的前提条件：失衡结点的所有子孙结点均已平衡，各自满足AVL树条件

➢ 失衡结点属于LL型的判断方法：

GetHeight(tree.left) – GetHeight(tree.right) = 2

且

GetHeight(tree.left.left) ≥ GetHeight(tree.left.right)

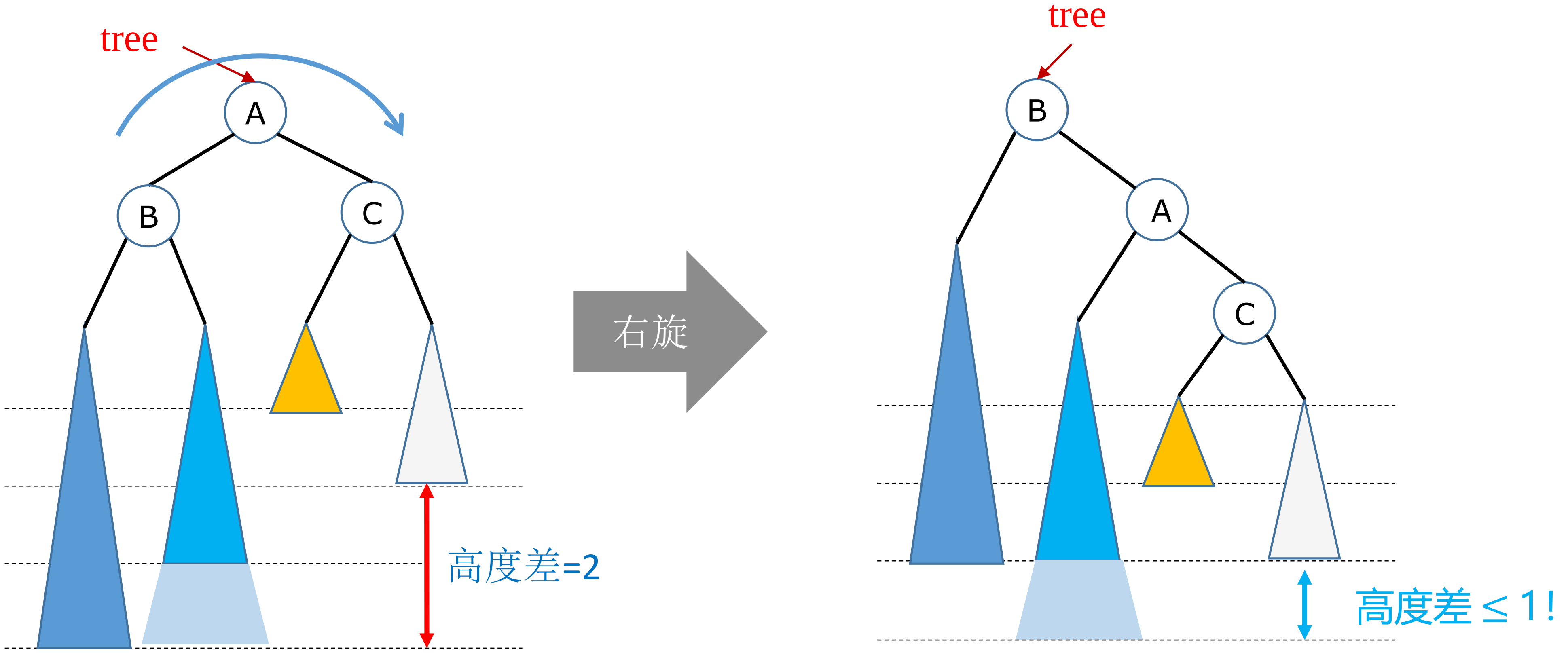
➢ 调整方法：右旋失衡结点一次！

tree ← RightRotate(tree)

11.3 AVL树

失衡调整旋转平衡处理

(1)单向右旋（LL)

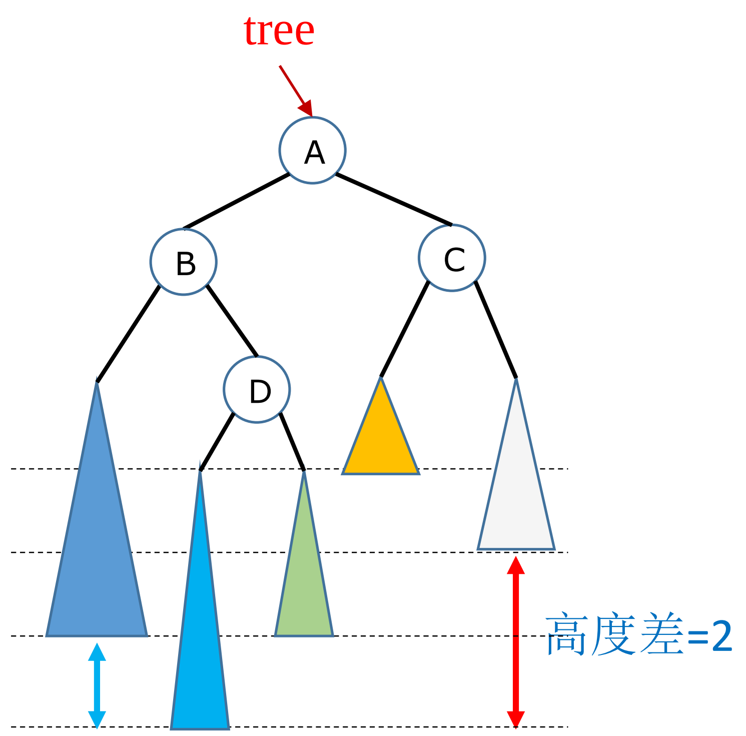


非平衡二叉树　　AVL树

11.3 AVL树

失衡调整旋转平衡处理

(3)先左后右旋转（LR）



前提条件：失衡结点的所有子孙结点均已平衡，全部满足AVL树条件

- 失衡结点属于LR型的判断方法：

GetHeight(tree.left) – GetHeight(tree.right) = 2

且

GetHeight(tree.left.left) < GetHeight(tree.left.right)

- 调整方法：先左旋左子结点，再右旋结点

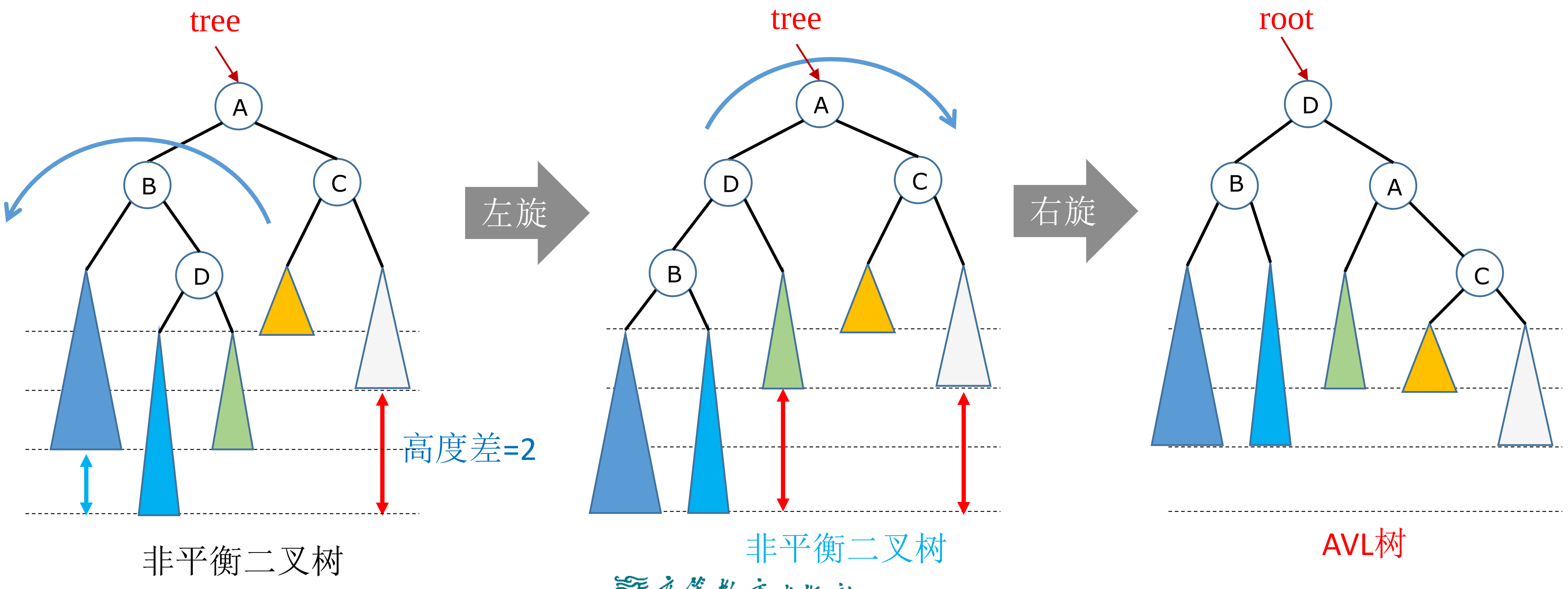
(1) tree.left ← LeftRotate(tree.left)

(2) tree ← RightRotate(tree)

11.3 AVL树

失衡调整旋转平衡处理

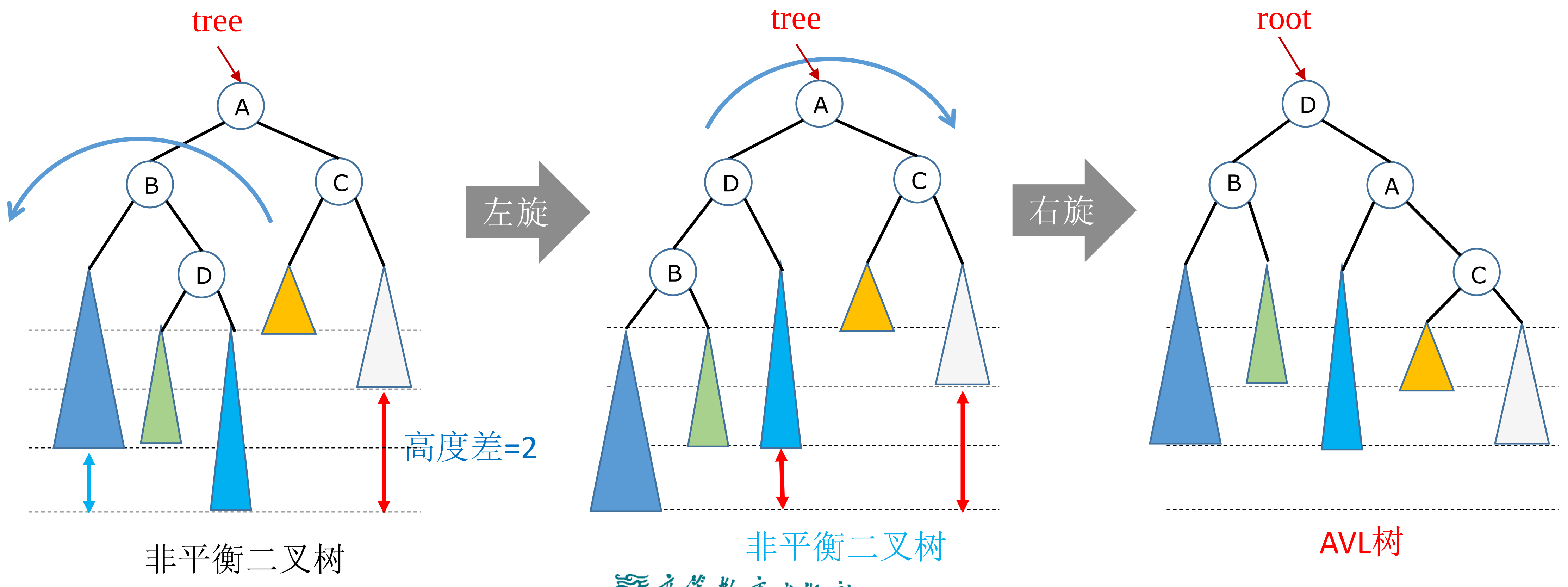
(3)先左后右旋转（LR）



11.3 AVL树

失衡调整旋转平衡处理

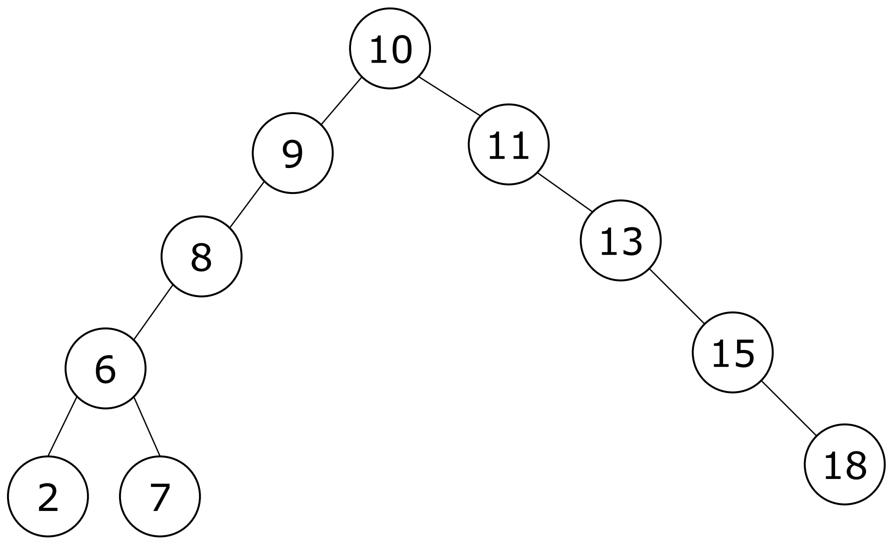
(3)先左后右旋转（LR）



11.3 AVL树

示例

Insert 10，11，13，15，18，9，8，6，7，2 to BST



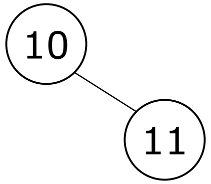
非平衡二叉树

11.3 AVL树

示例

Insert 10，11，13，15，18，9，8，6，7，2 to AVL

insert 10,11



平衡二叉树

11.3 AVL树

示例

Insert 10，11，13，15，18，9，8，6，7，2 to AVL

insert 13

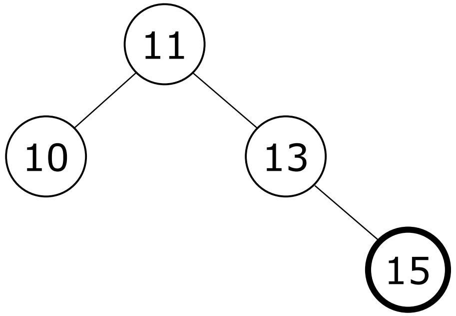
非平衡二叉树　　　　平衡二叉树

11.3 AVL树

示例

Insert 10，11，13，15，18，9，8，6，7，2 to AVL

insert 15



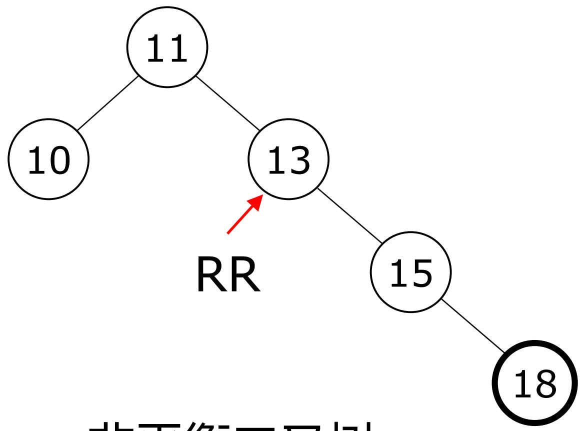
平衡二叉树

11.3 AVL树

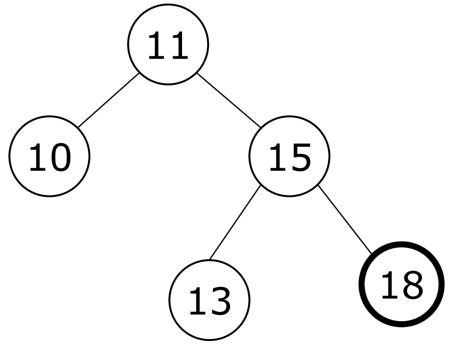
示例

Insert 10，11，13，15，18，9，8，6，7，2 to AVL

insert 18



非平衡二叉树



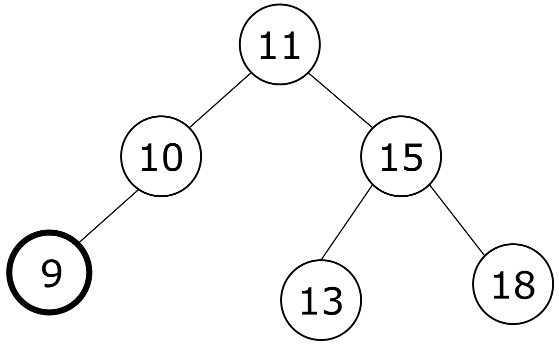
平衡二叉树

11.3 AVL树

示例

Insert 10，11，13，15，18，9，8，6，7，2 to AVL

insert 9



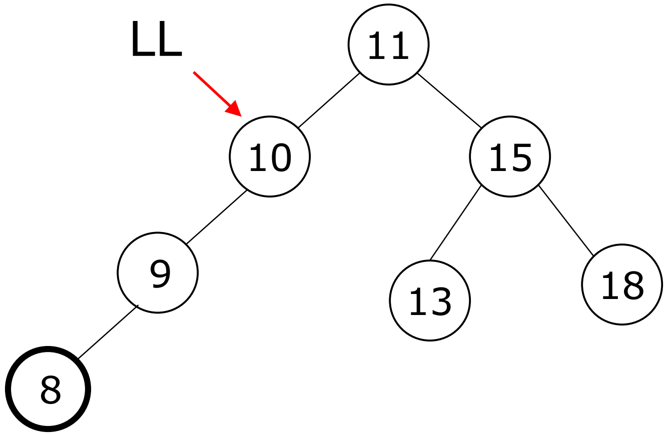
平衡二叉树

11.3 AVL树

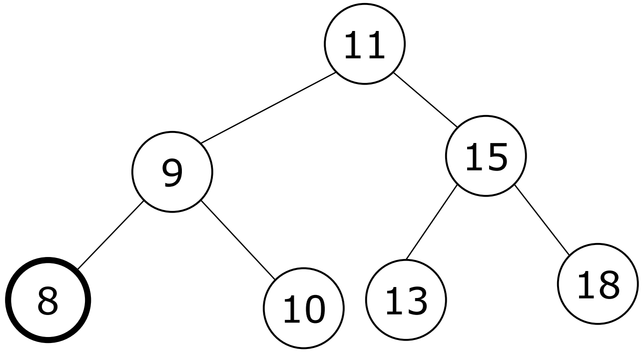
示例

Insert 10，11，13，15，18，9，8，6，7，2 to AVL

insert 8



非平衡二叉树



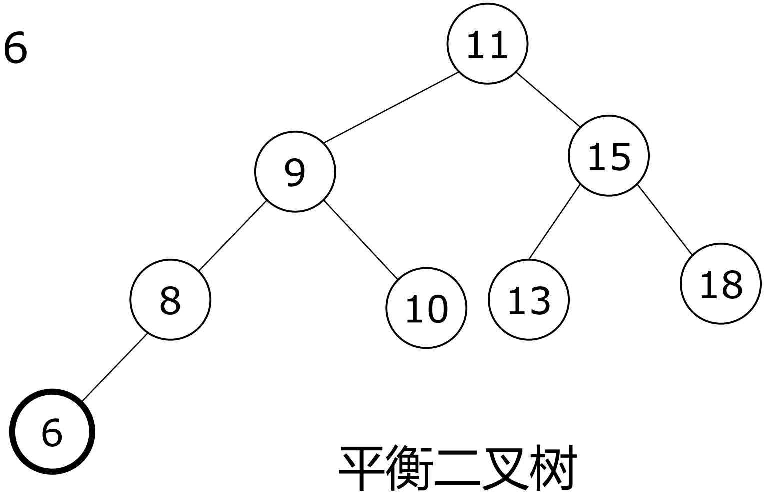
平衡二叉树

11.3 AVL树

示例

Insert 10，11，13，15，18，9，8，6，7，2 to AVL

insert 6



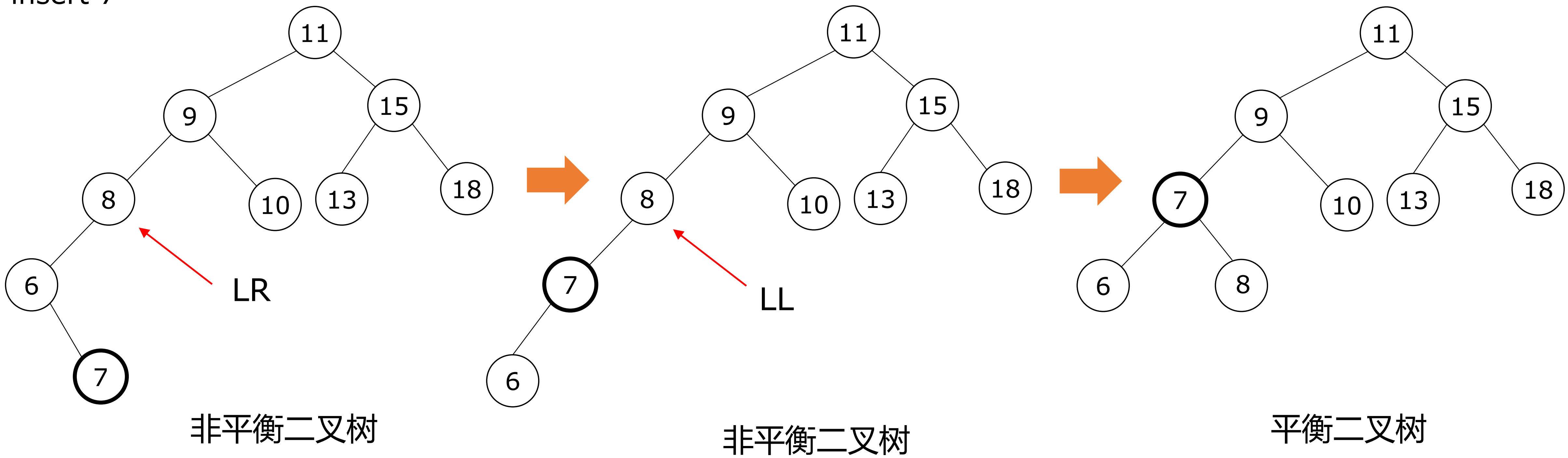
平衡二叉树

11.3 AVL树

示例

Insert 10，11，13，15，18，9，8，6，7，2 to AVL

insert 7

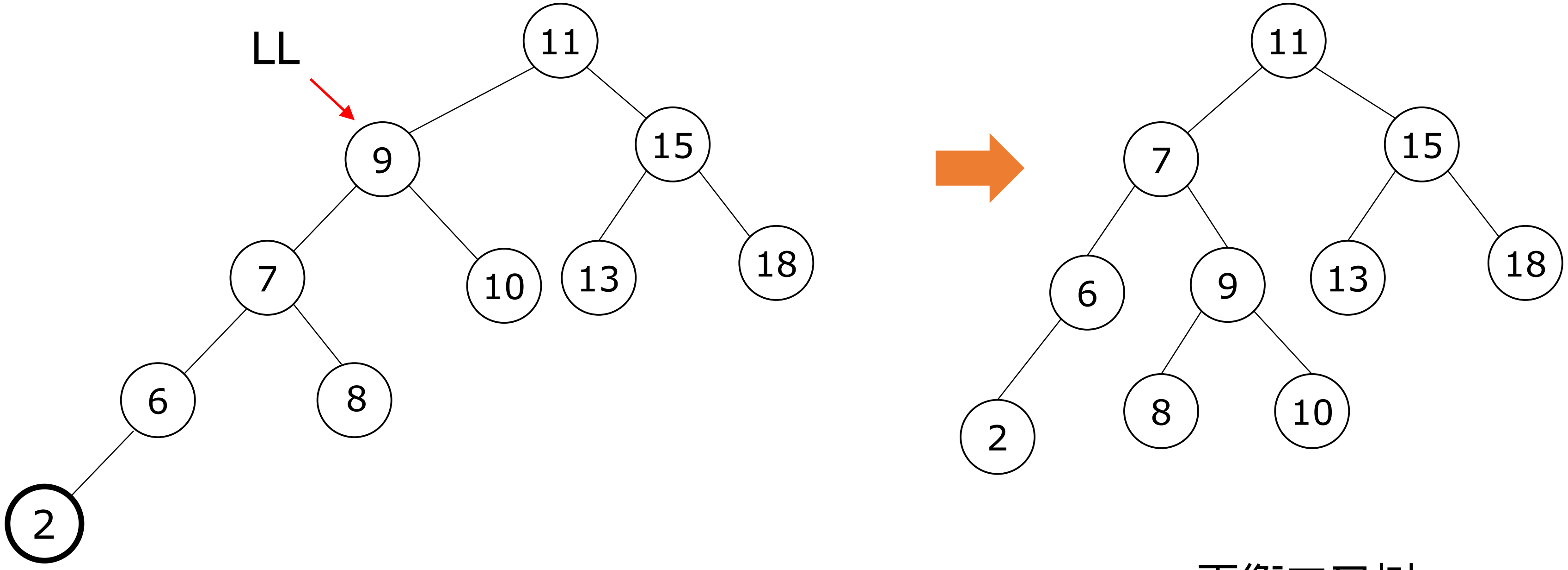


11.3 AVL树

示例

Insert 10，11，13，15，18，9，8，6，7，2 to AVL

insert 2



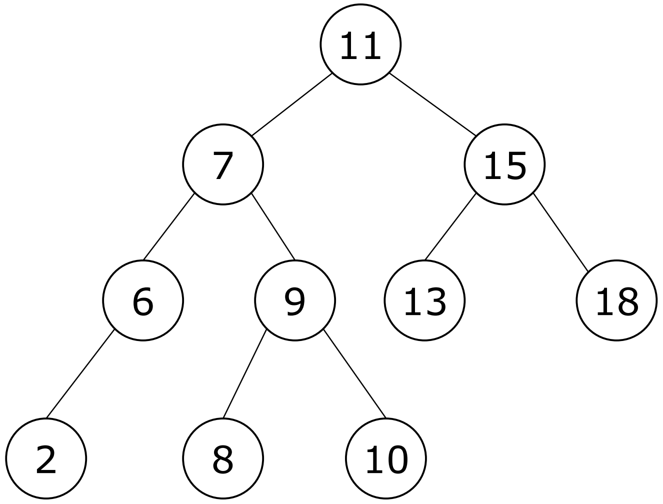
非平衡二叉树

平衡二叉树

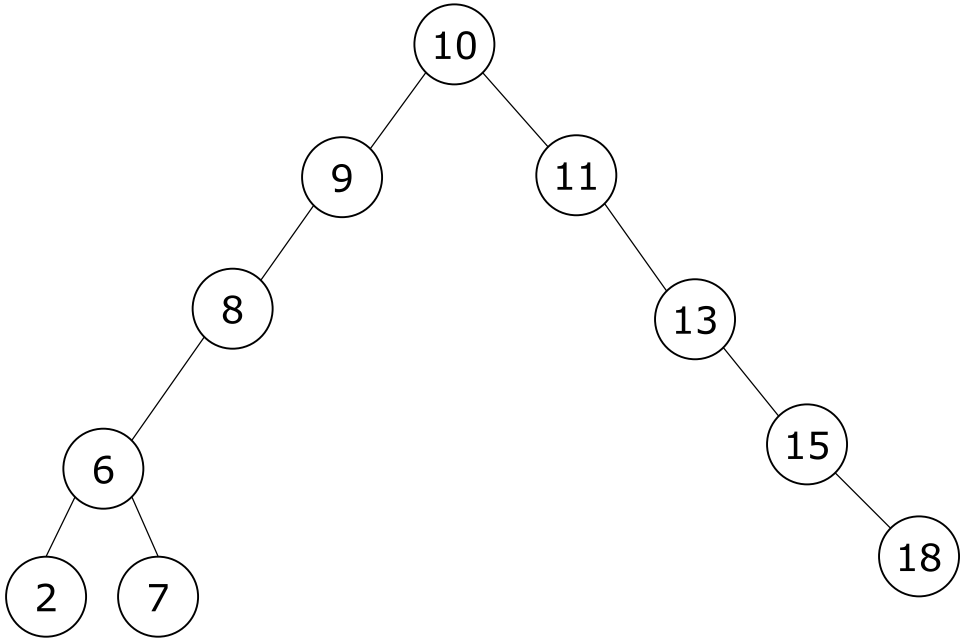
11.3 AVL树

示例

Insert 10，11，13，15，18，9，8，6，7，2 to AVL



AVL



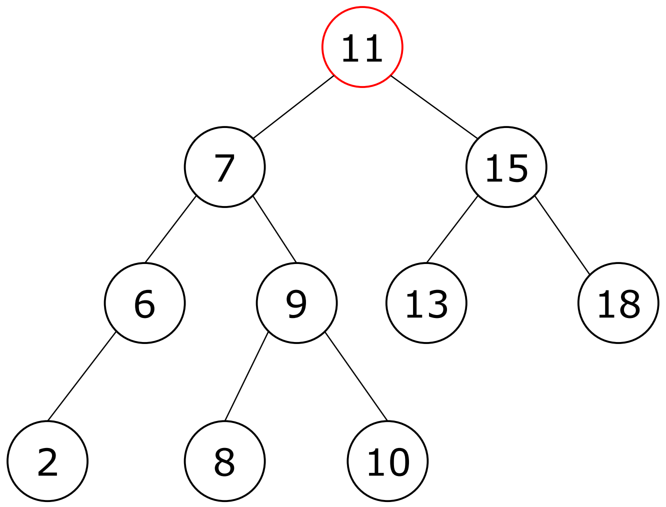
BST

11.3 AVL树

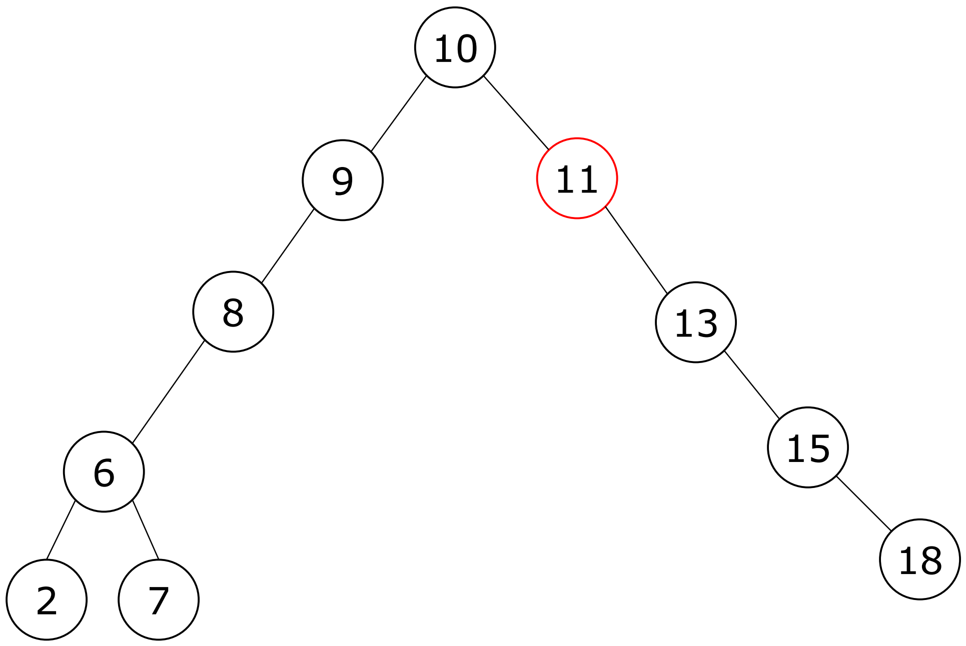
示例

Insert 10，11，13，15，18，9，8，6，7，2 to AVL

Delete 11



AVL

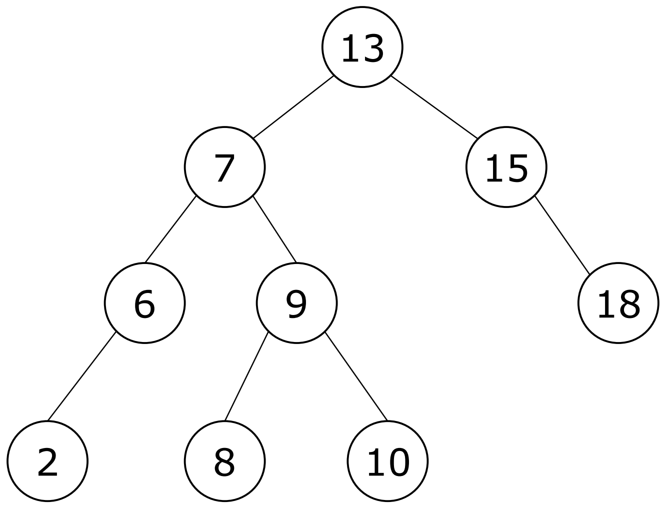


BST

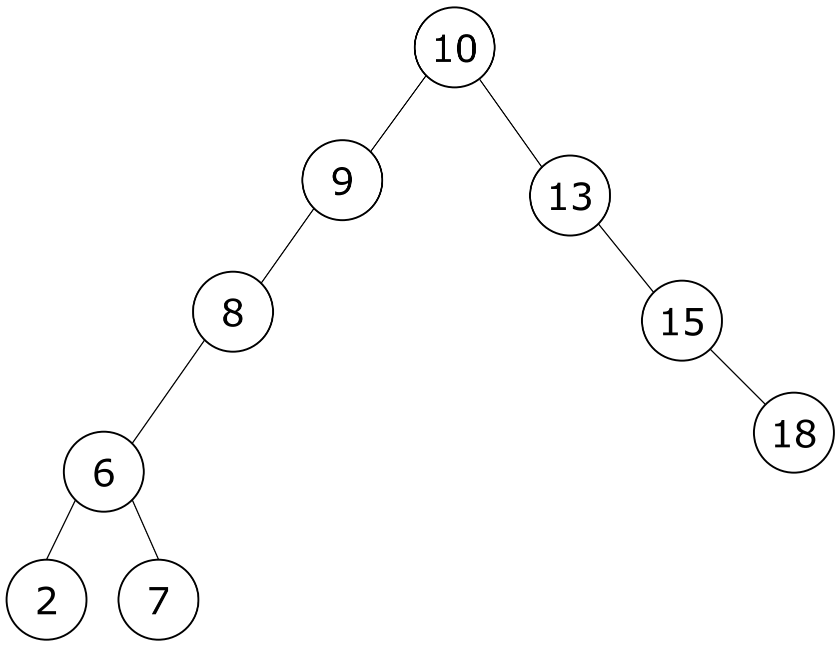
11.3 AVL树

示例

Insert 10，11，13，15，18，9，8，6，7，2 to AVL



AVL



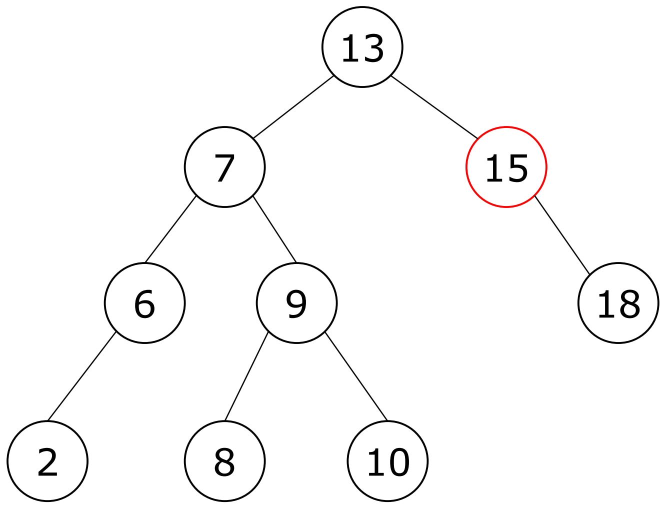
BST

11.3 AVL树

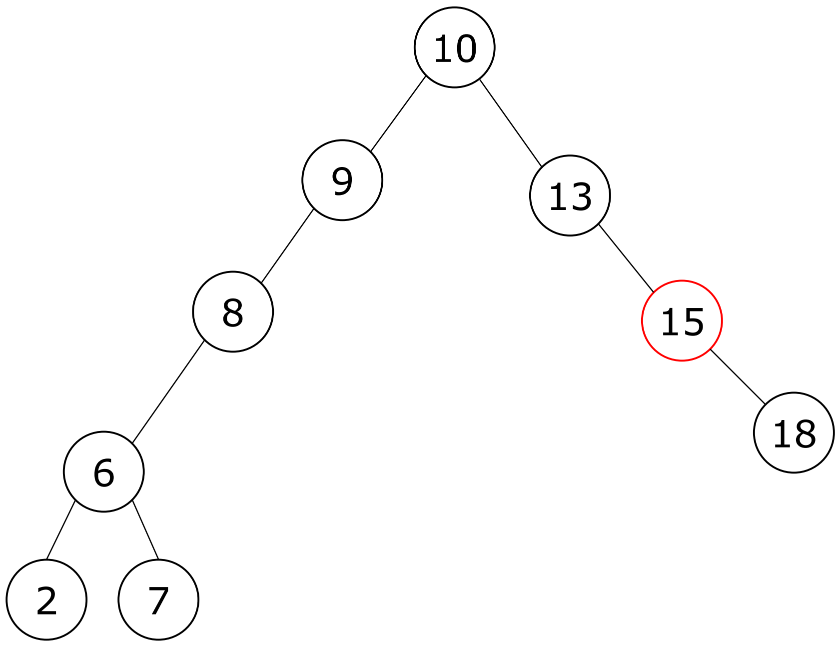
示例

Insert 10，11，13，15，18，9，8，6，7，2 to AVL

Delete 15



AVL



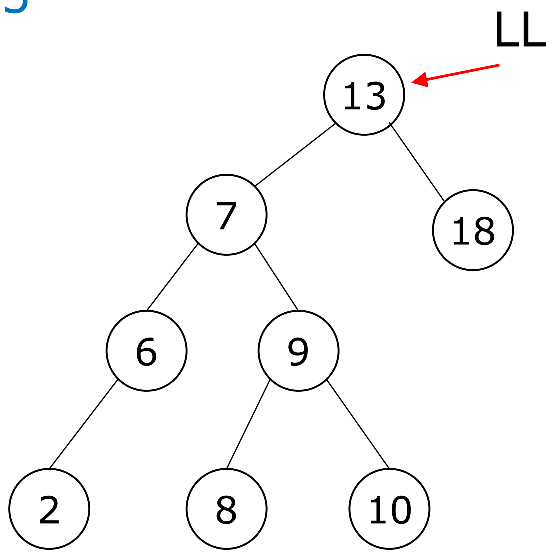
BST

11.3 AVL树

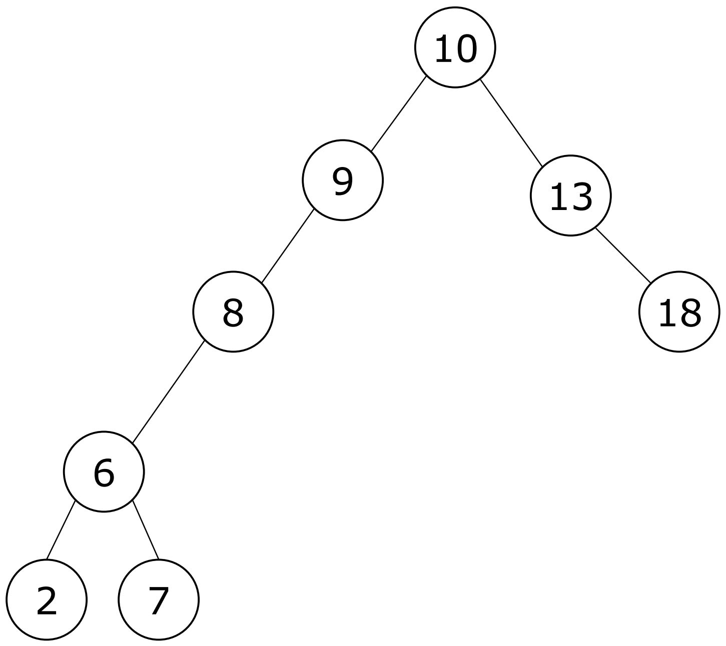
示例

Insert 10，11，13，15，18，9，8，6，7，2 to AVL

Delete 15



根结点失衡，需要调整

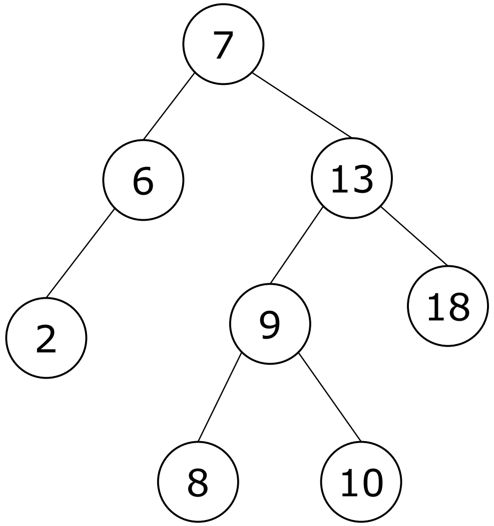


BST

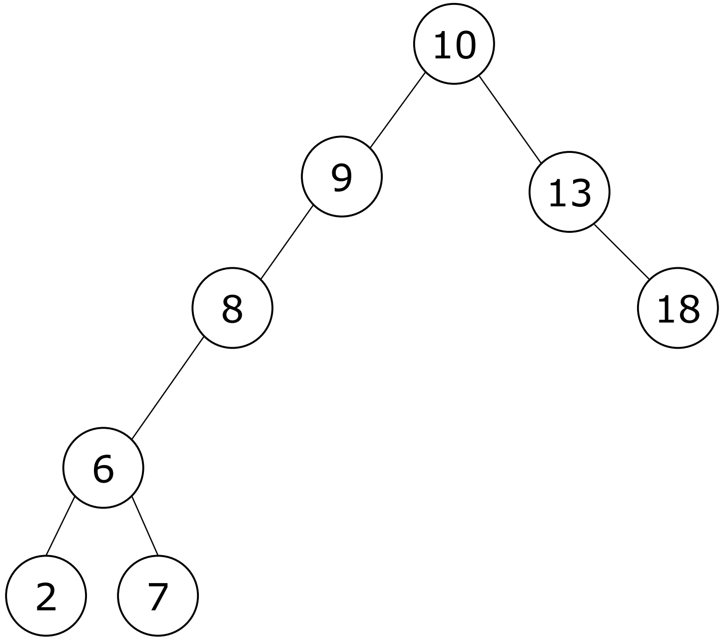
11.3 AVL树

示例

Insert 10，11，13，15，18，9，8，6，7，2 to AVL



AVL



BST

11.3 AVL树

平衡化算法

- 对AVL树的插入与删除操作可以直接调用BST的算法Insert(tree, key)和Removal(tree, key)。
- 在上述算法中，当生成新的叶结点或删除已有结点后，可以对新结点或删除结点的父结点（非空），调用右边的算法，对树进行调整，维护AVL树的平衡性。

- 时间复杂度: O(log(n))

算法：Balancing(tree, node)
输入：AVL树tree, 结点node是新插入结点或被删除结点的父结点（非空）
输出：调整后的AVL树

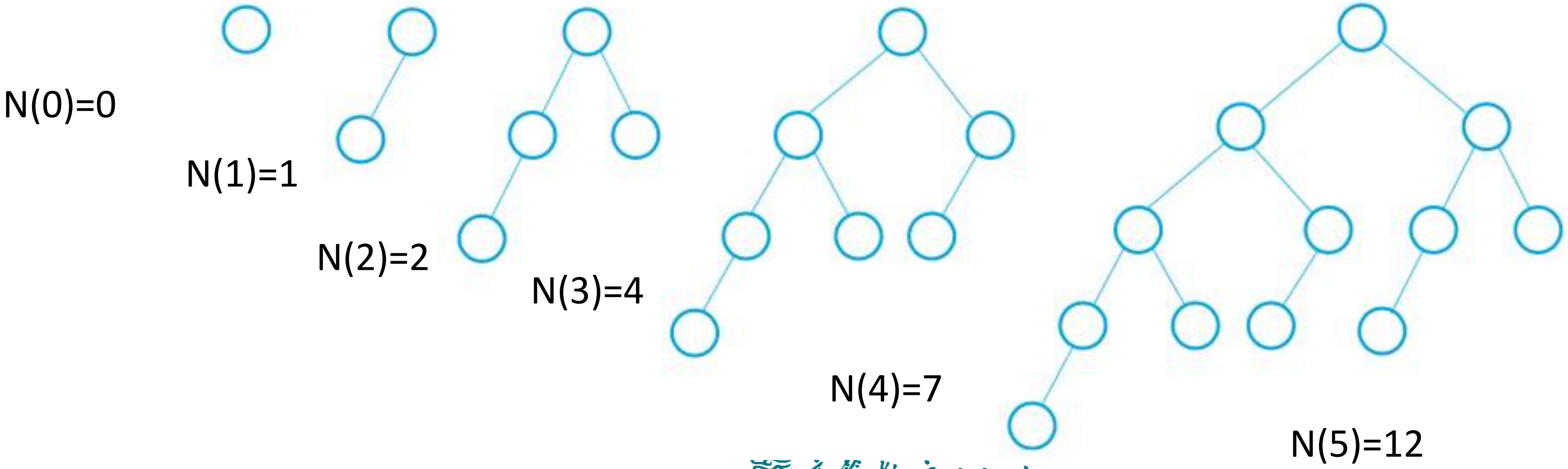
```
if tree ≠ node then      //二分查找node
|  if tree.data < node.data then
|  |  tree.left ← Balancing(tree.left, node)      //递归查找左子树
|  else
|  |  tree.right ← Balancing(tree.right, node)  //递归查找右子树
|  end                                           //保存根到node的路径
end              //tree = node, 从结点node开始从下至上依次调整
UpdateHeight(tree)    //首先更新结点高度，因其子树可能调整
           //
if GetHeight(tree.left) – GetHeight(tree.right) = 2 then
|  if GetHeight(tree.left) < GetHeight(tree.right) then    //LR
|  |  tree.left ← LeftRotate(tree.left)           //LR → LL
|  end
|  tree ← RightRotate(tree)        //调整LL
|            //
else if GetHeight(tree.right) – GetHeight(tree.left) = 2 then
|  if GetHeight(tree.right) < GetHeight(tree.left) then     //RL
|  |  tree.right ← RightRotate(tree.right)      //RL → RR
|  end
|  tree ← LeftRotate(tree)          //调整RR
end
return tree        //返回调整好的AVL树
```

11.3 AVL树

*经典问题：树高上界

问题：含n个结点的AVL树，其最大高度是多少？或者问，高度为d的AVL树，其中至少含多少个结点？(n, d ≥ 0)

用N(d) 表示高度为d的AVL树所含结点的最小数目



N(10000)= ？

11.3 AVL树

*经典问题：树高上界

问题描述：高度为d的AVL树，其中至少含多少个结点？(n, d > 0)

用N(d) 表示高度为d的AVL树所含结点的最小数目

分析1：高度为d且结点数最少的AVL树，其左右子树也是AVL树， 并且这两个子树的结点数也一定最少（？）。设左右子树的高度为 d_l 和 d_r，因此

$$N(d) = N(d_l) + N(d_r) + 1$$

分析2：根据树与子树在高度上的关系：$d = \mathrm{Max}(d_l, d_r) + 1$，那么对高度为$Min(d_l, d_r)$的（较矮）子树，其高度应该是多少？（同时满足AVL树以及结点数最少的条件）

$$Min(d_l, d_r) = \mathrm{Max}(d_l, d_r) - 1$$

11.3 AVL树

*经典问题：树高上界

问题描述：高度为d的AVL树，其中至少含多少个结点？(n, d > 0)

用N(d) 表示高度为d的AVL树所含结点的最小数目

$$N(d) = N(d_l) + N(d_r) + 1$$

$$d = \text{Max}(d_l, d_r) + 1$$

$$Min(d_l, d_r) = \text{Max}(d_l, d_r) - 1$$

$$N(d) = N(d-1) + N(d-2) + 1$$

N(0) = 0

N(1) = 1

- 时间复杂度：O(d)

**思考：如何算N(10^9)?

11.3 AVL树

*经典问题：树高上界

问题描述：高度为d的AVL树，其中至少含多少个结点？(n, d > 0)

思考：如何算N(10^9)?

N(d) = N(d-1) + N(d-2) + 1

转换为矩阵式

$$\begin{pmatrix} N(d) \\ N(d-1) \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} N(d-1) \\ N(d-2) \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}^2 \begin{pmatrix} N(d-2) \\ N(d-3) \\ 1 \end{pmatrix}$$

$$= \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}^{d-1} \begin{pmatrix} N(1) \\ N(0) \\ 1 \end{pmatrix}$$

求矩阵的幂，可二分！

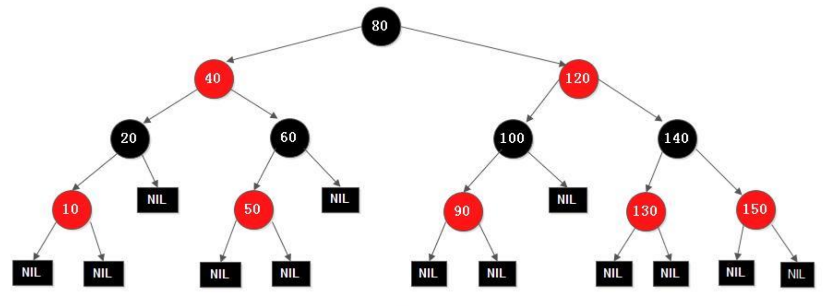
- 时间复杂度：O(log(d))

11.3 AVL树

平衡树的种类与实际应用

- **AVL树**：最早、最基本的平衡二叉树，应用于Windows的进程地址空间管理

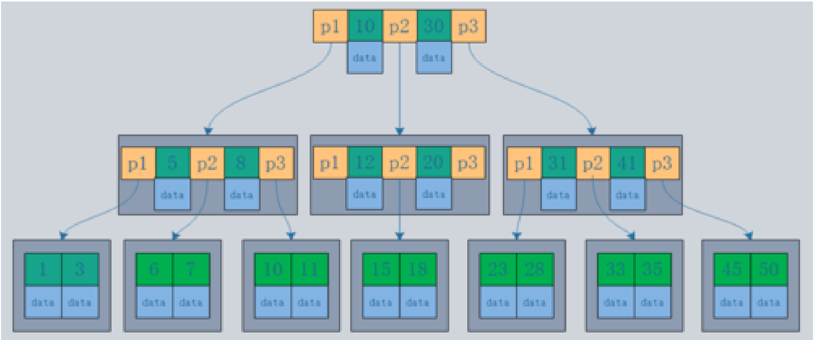
- **红黑树**：平衡二叉树，广泛应用于C++的STL（如map、set），JAVA的HashMap



红黑树

- **B树**：平衡多叉树，主要为数据库（如MySQL)和文件系统提供树形索引

- **B+树**：B树的变形，主要用于磁盘和存储系统，例如MySQL引擎 InnoDB 使用B+树作为索引的数据结构



11.3.3 作业

1、给定关键词输入序列{CAP,AQU,PIS,ARI,TAU,GEM,CAN,LTB,VIR,LEO,SCO}，假定关键词比较按英文字典顺序，请画出从一棵空的平衡树开始，依上述顺序（从左到右）输入关键词，用平衡树的查找和插入算法生成一棵平衡树的过程，并说明生成过程中采用了何种旋转方式进行平衡调整。

谢谢观看