第14-15章　搜索树(一)(H)

计算机学院

主要内容

- 二叉搜索树
  - 定义
  - 搜索
  - 插入
  - 删除
- AVL树

BST定义

- **定义：二叉搜索树**（binary search tree）是一棵二叉树，可能为空，如果非空的话，应满足以下特征：

1) 每个元素有一个关键值，并且没有任意两个元素有相同的关键值；因此，所有的关键值都是唯一的

BST定义

2) 根节点左子树的关键值（如果有的话）小于根节点的关键值

3) 根节点右子树的关键值（如果有的话）大于根节点的关键值

4) 根节点的左右子树也都是二叉搜索树

另一种定义

- 目标：高效的有序输出
- 一棵二叉树，如果其中序遍历的结果，得到的是关键值按升序排列的列表，则它是二叉搜索树

别名

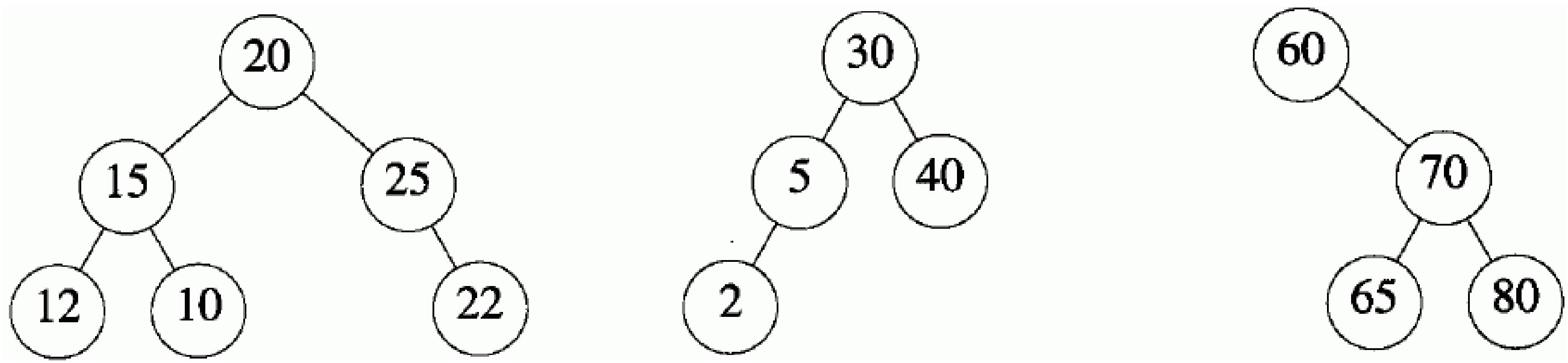
- 二叉搜索树

=二叉排序树

=二叉查找树

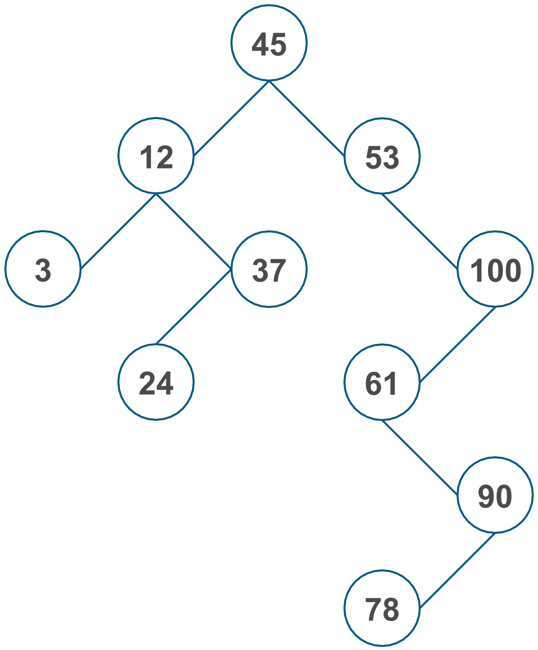
=BST（Binary Search Tree）

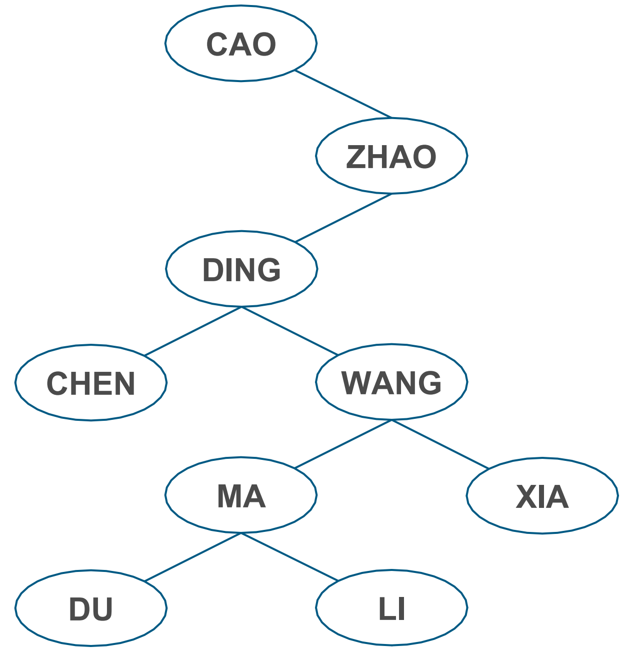
例



不是　　是　　是

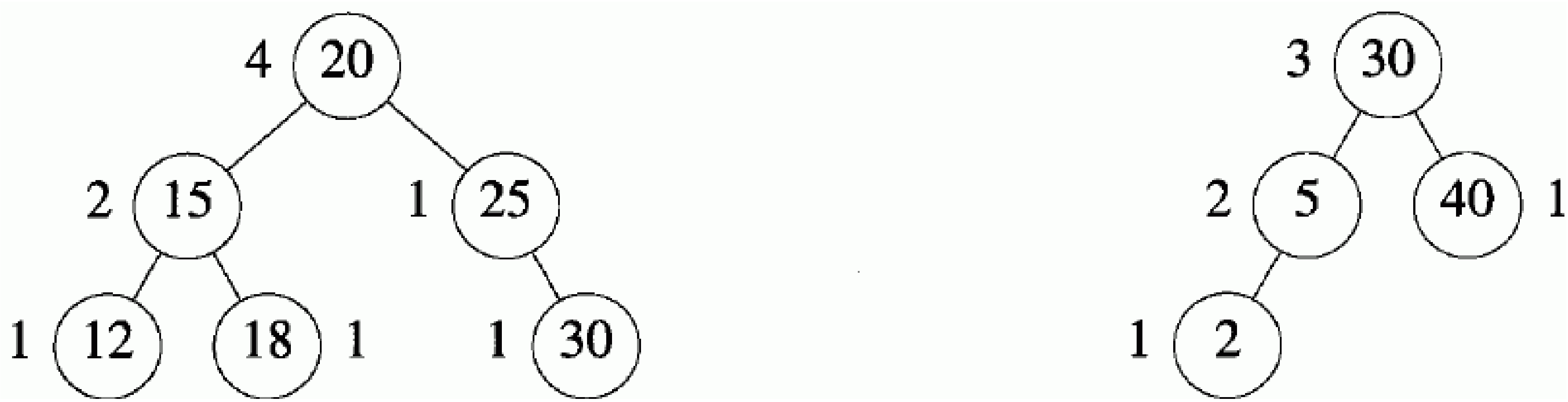
例





索引二叉搜索树

- 每个节点记录一个索引值：
  左子树大小+1——　　节点在子树中的排名！
- 如何通过索引得到“求第k元”的高效算法



抽象数据类型

抽象数据类型*BSTree*{

实例

二叉树，每一个节点中有一个元素，该元素有一个关键值域；所有元素的关键值各不相同；任何节点左子树的关键值小于该节点的关键值；任何节点右子树的关键值大于该节点的关键值。

操作

Create()：创建一个空的二叉搜索树

Search(k, e)：将关键值为*k*的元素返回到*e*中；如果操作失败则返回false，否则返回true

Insert(e)：将元素*e*插入到搜索树中

Delete(k, e)：删除关键值为*k*的元素并且将其返回到*e*中

Ascend()：按照关键值的升序排列输出所有元素

}

索引二叉搜索树

抽象数据类型*IndexedBSTree*{

实例

除每个节点有一个LeftSize域以外，其他与BSTree相同

操作

Create()：产生一个空的带索引的二叉搜索树

Search(k, e)：将关键值为*k*的元素返回到*e*中；如果操作失败返回*false*，否则返回true

IndexSearch(k, e)：将第*k*个元素返回到*e*中

Insert(e)：将元素*e*插入到搜索树

Delete(k, e)：删除关键值为*k*的元素并且将其返回到*e*中

IndexDelete(k, e)：删除第*k*个元素并将其返回到*e*中

Ascend()：按照关键值的升序排列输出所有元素

}

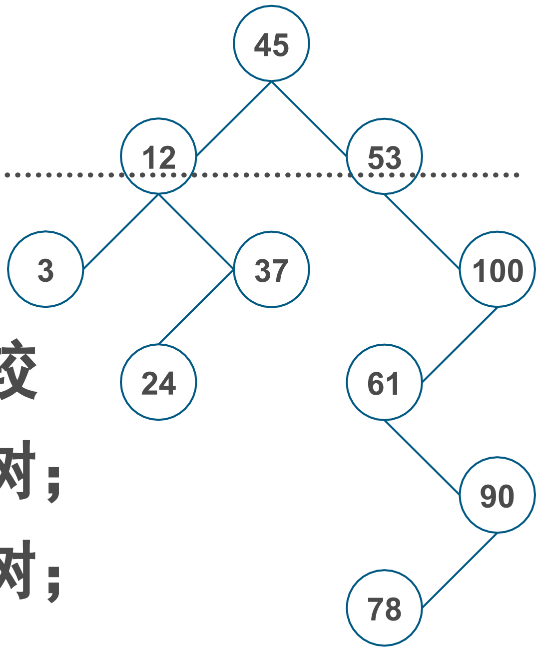
类BSTree

由二叉树类派生

```
template<class E, class K>
class BSTree : public BinaryTree<E> {
  public:
    bool Search(const K& k, E& e) const;
    BSTree<E,K>& Insert(const E& e);
    BSTree<E,K>& Delete(const K& k, E& e);
    void Ascend() {InOutput();}
};
```

升序输出：中序遍历即可

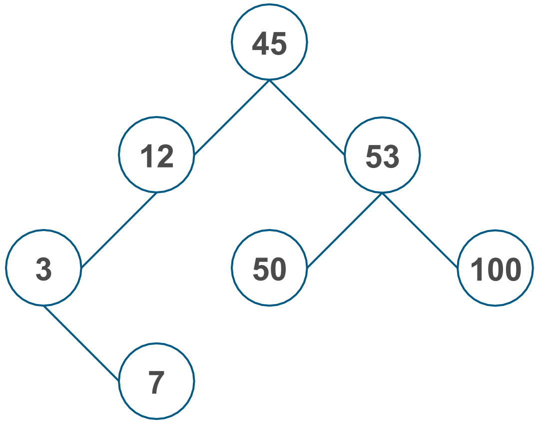
BST搜索

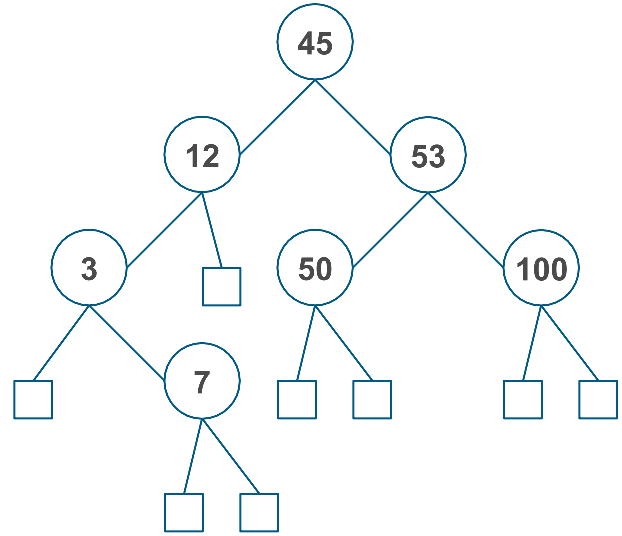


- 算法
  - 从根节点开始将key与节点值比较
  - 如果key小于节点值，进入左子树；
  - 如果key大于节点值，进入右子树；
  - 直到找到或子树为空停止。

- 示例
  - 找100
  - 找40

平均查找长度





$ASL_{成功}$=（1*1+2*2+3*3+4*1）/7=18/7

到达所有元素所需的长度之和/元素个数

$ASL_{不成功}$=（2*1+3*5+4*2）/8=25/8

到达所有空节点所需长度之和/空节点个数

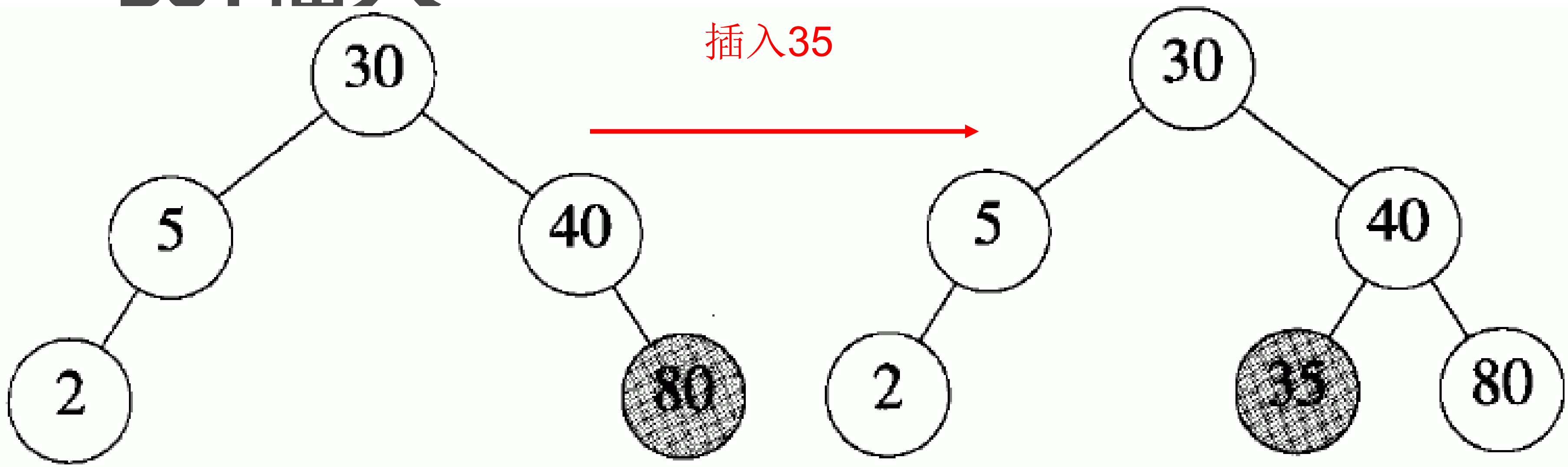
搜索函数

```
template<class E, class K>
bool BSTree<E,K>::Search(const K& k, E &e) const
{
  BinaryTreeNode<E> *p = root;
  while (p) // examine p->data
    if (k < p->data) p = p->LeftChild;
    else if (k > p->data) p = p->RightChild;
        else {// found element
            e = p->data;
            return true;}
  return false;
}
```

非常简单，从根节点开始，
“小左大右”即可，直至
找到关键字或到达空节点

思考：索引二叉搜索树中
如何求第k元？

BST插入



- 需先进行搜索操作
  - 若成功，表明有重复关键字，插入失败
  - 若失败，搜索结束的位置即为插入位置

请思考：35为什么没被插在80的左孩子位置？
请绘制：在上述BST中查找3和35的查找路径？

关于BST插入的结论

- 新插入的节点一定是一个**叶子节点**
- 并且是查找不成功时查找路径上访问的最后一个节点的左孩子或右孩子节点

插入操作

```
template<class E, class K>
BSTree<E,K>& BSTree<E,K>::Insert(const E& e)
{// Insert e if not duplicate.
  BinaryTreeNode<E> *p = root,  // search pointer
                    *pp = 0;    // parent of p
  // find place to insert
  while (p) {// examine p->data
    pp = p;
    // move p to a child
    if (e < p->data) p = p->LeftChild;
    else if (e > p->data) p = p->RightChild;
        else throw BadInput(); // duplicate
    }
```

插入操作（续）

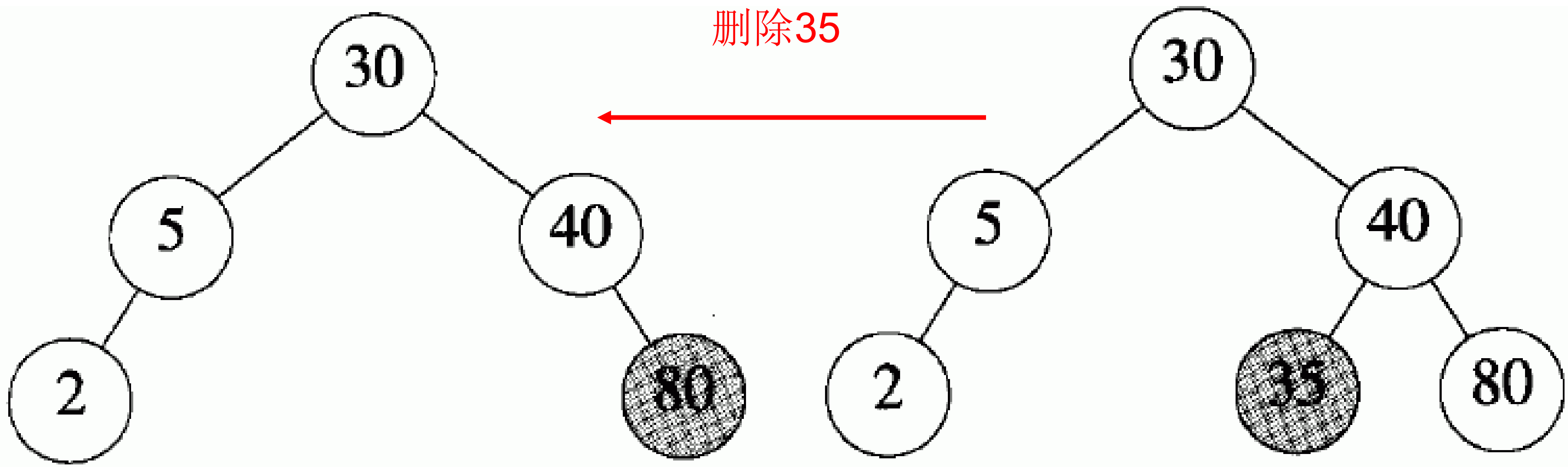
```
// get a node for e and attach to pp
BinaryTreeNode<E> *r = new BinaryTreeNode<E> (e);
if (root) {// tree not empty
  if (e < pp->data) pp->LeftChild = r;
  else pp->RightChild = r;}
else // insertion into empty tree
    root = r;

return *this;
}
```

由一系列插入生成BST

- 从空树出发，由序列{45, 24, 53, 45, 12, 24, 90}构造BST
- 上述过程本质上是将无序序列转变为有序序列的过程

BST删除

1. **删除叶节点**
   - 直接丢弃——父节点指向它的指针置为0

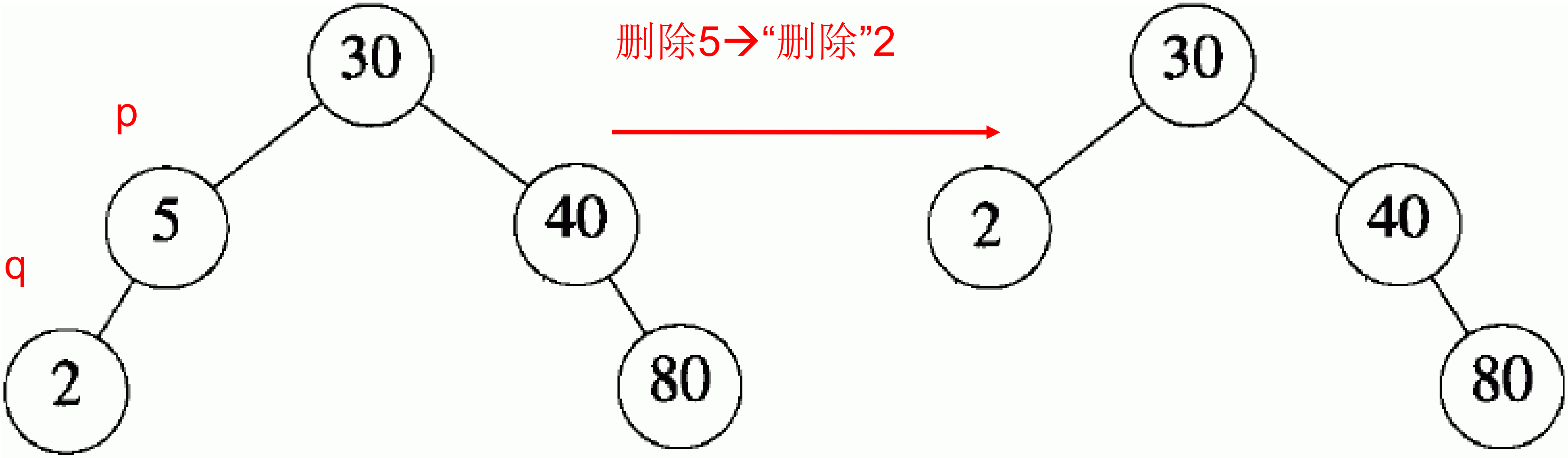


删除算法（续）

- 不是叶节点：如何保持二叉搜索树特性？——删除节点p，p的子树内部调整

2. p有且只有一个非空子树t，其根为q
   - 丢弃p，以q取代p的位置
     - p是整个二叉搜索树的根——q作为新的根
     - 否则——p的父节点的“指向p的指针”变为指向q

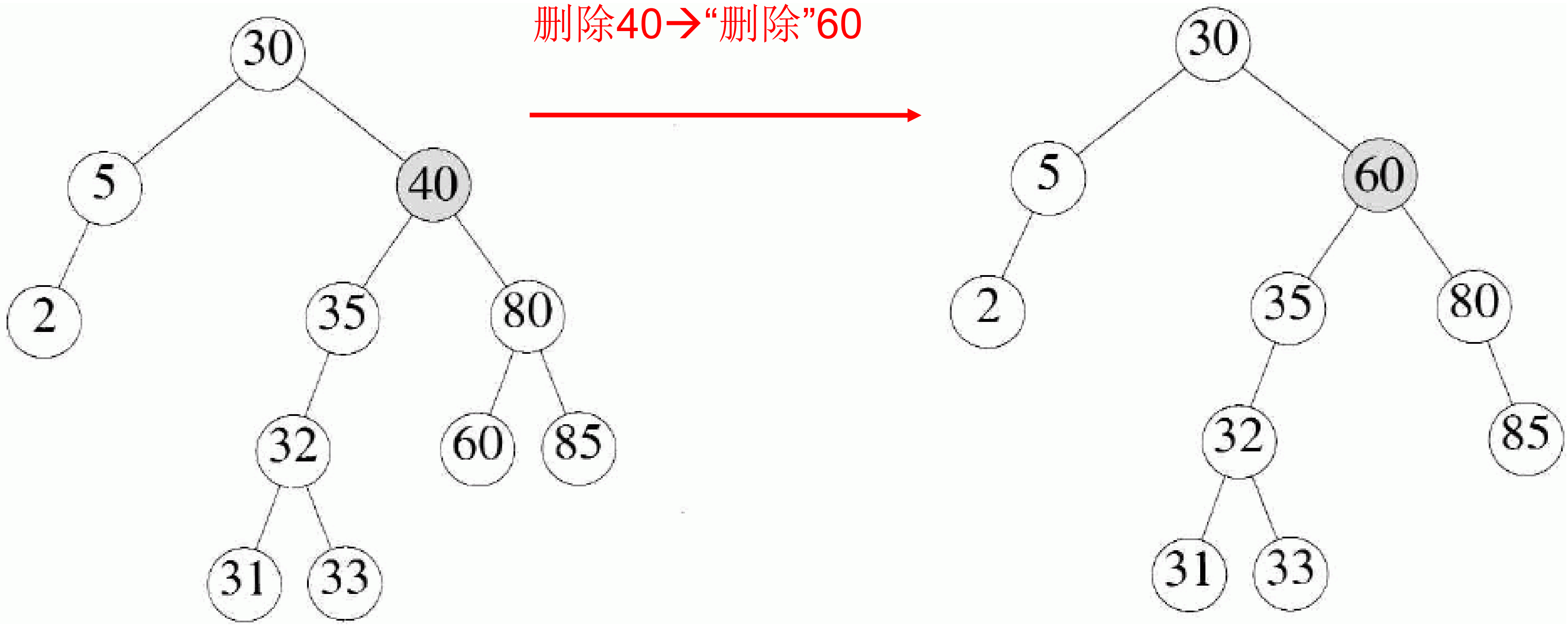
删除算法（续）



删除算法（续）

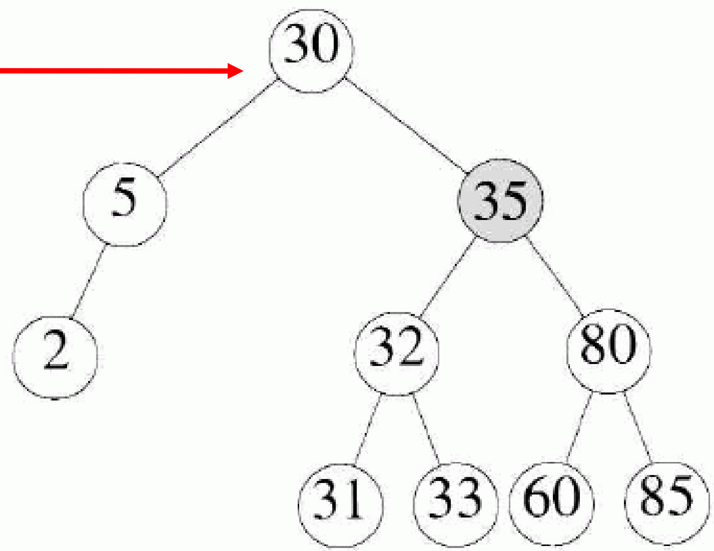
3. p的两个子树都不空——转换为情况2（或1）
   - p与某个节点q（满足情况2或1）交换
   - 删除p→删除q
   - q的选择准则？——替代p成为子树的根后，应当保持二叉搜索树特性
   - 左子树的最右节点或右子树的最左节点，排名恰与p相邻——之前或之后

删除算法（续）

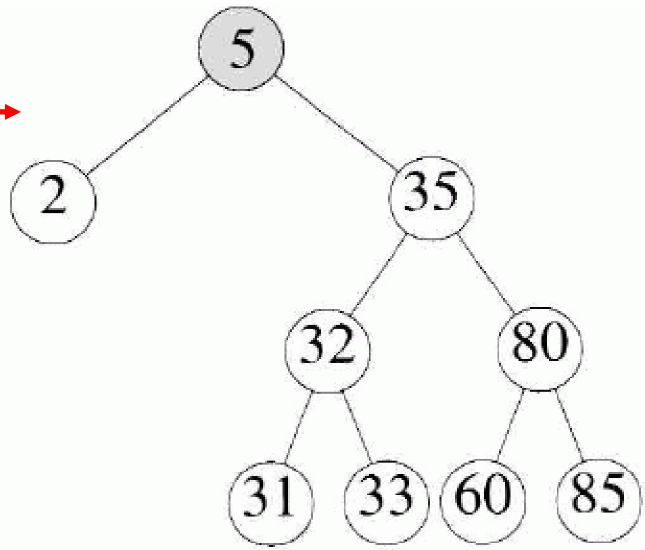


删除算法（续）

删除40的另一种方法



删除30→“删除”5

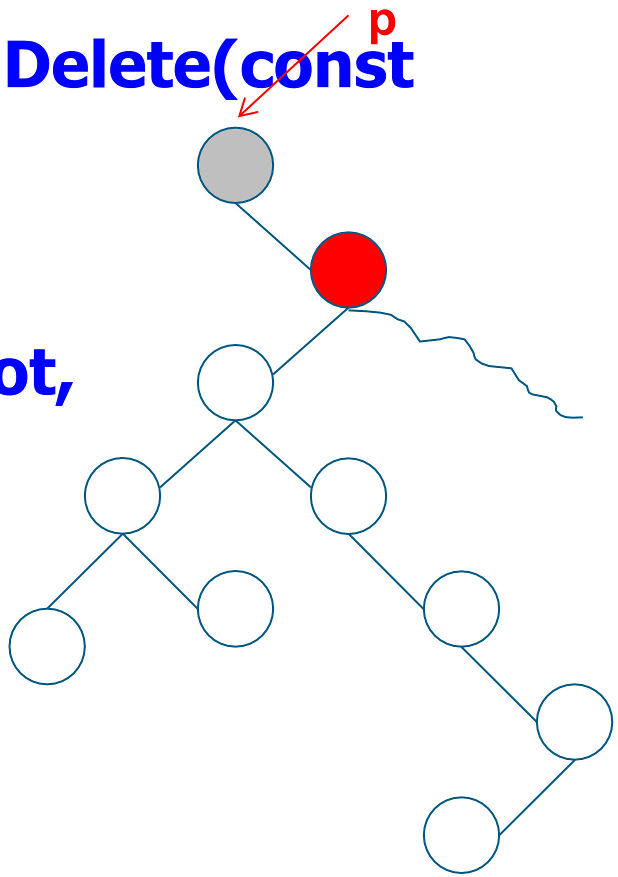


关于删除算法的结论

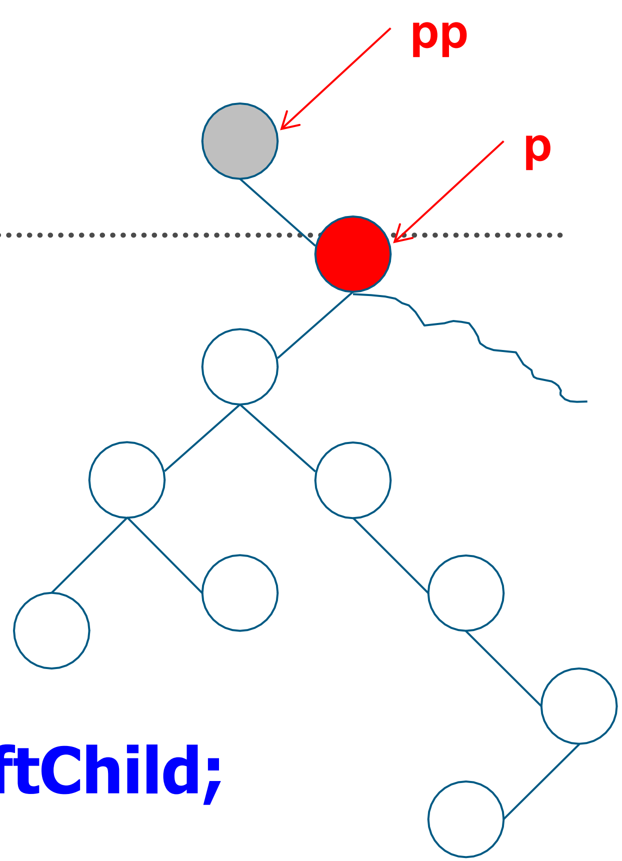
- 情况1、2无需讨论
- 情况3的算法思路其实是
  - 假定被删节点是p
  - 找到BST中p的直接前驱或直接后继替代它
  - 一般惯例是选用直接前驱，即**被删节点左子树的最右节点**

删除算法实现

```
template<class E, class K>
BSTree<E,K>& BSTree<E,K>::Delete(const
  K& k, E& e)
{
  BinaryTreeNode<E> *p = root,
                    *pp = 0;
```

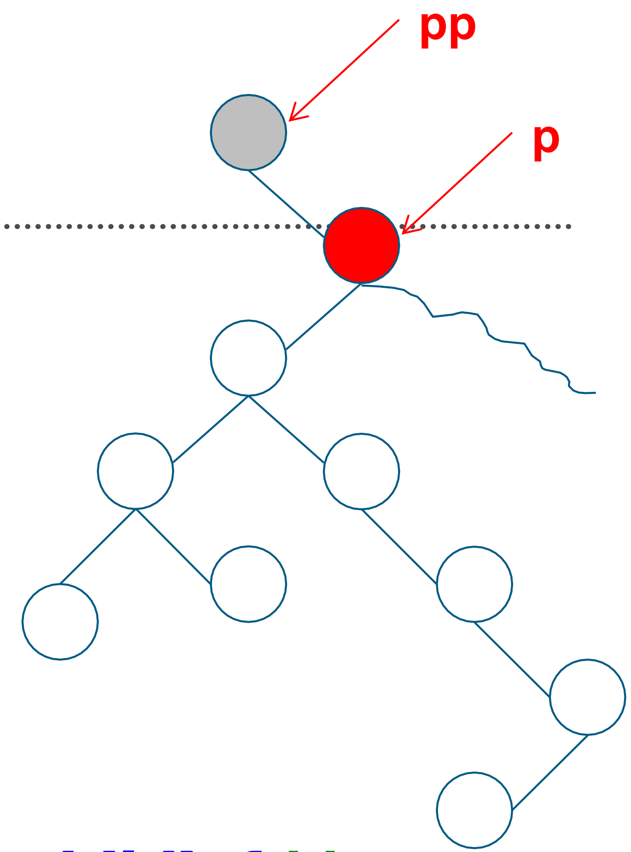


删除算法实现（续）




```
//先搜索要删除的节点p
while (p && p->data != k)
{
   pp = p;
   if (k < p->data) p = p->LeftChild;
   else p = p->RightChild;
}
if (!p) throw BadInput(); // no element with
key k
```

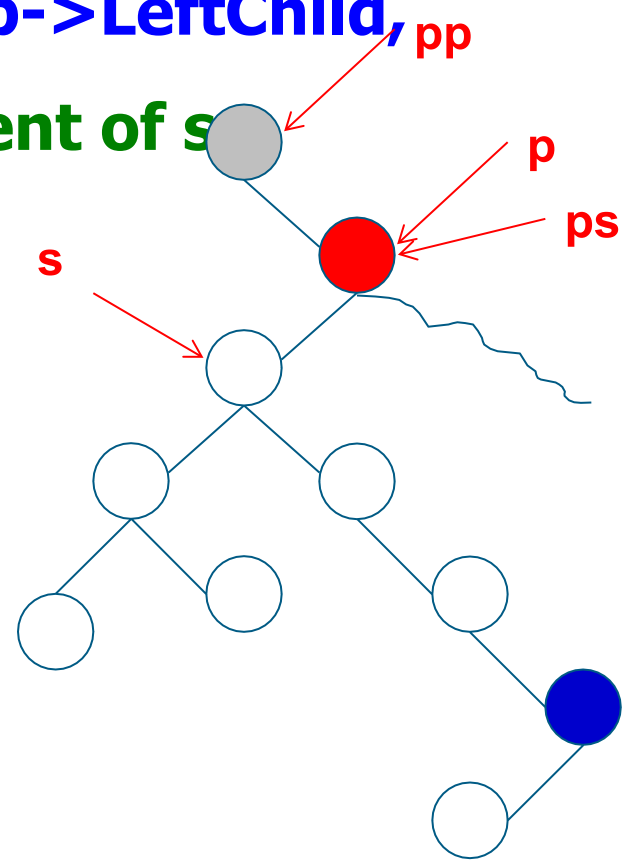
删除算法实现（续）




```
e = p->data;

//最坏情况，
//两个子树均不空，
// 转换为简单情况
if (p->LeftChild && p->RightChild) {// two children
   // convert to zero or one child case
   // find largest element in left subtree of p
```

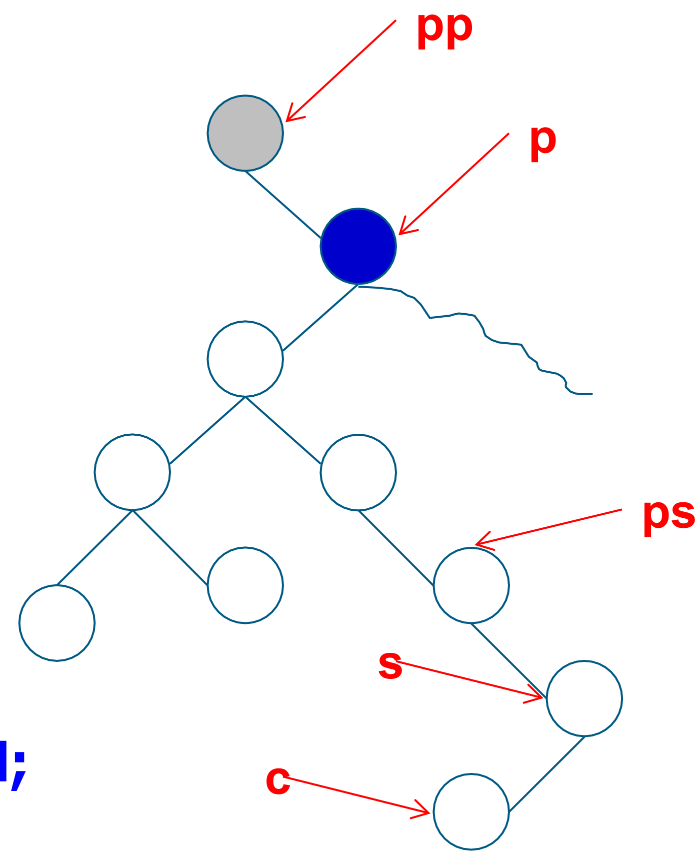
删除算法实现（续）

```
// 搜索左子树最右（大）节点
BinaryTreeNode<E> *s = p->LeftChild,
                *ps = p;  // parent of s
while (s->RightChild) {
   ps = s;
   s = s->RightChild;}
```



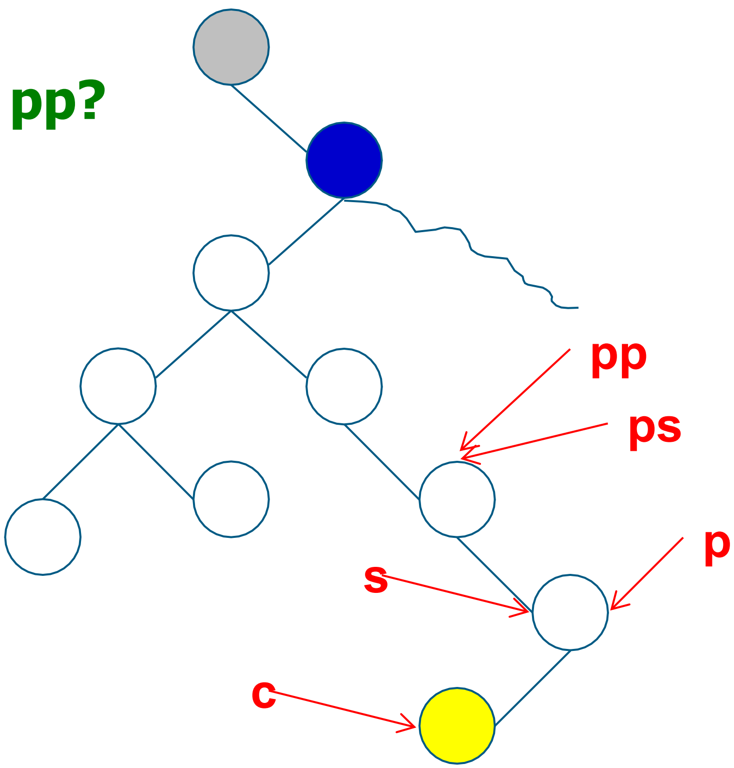
删除算法实现（续）

```
// 交换p和其左子树的最右节点
p->data = s->data;
p = s;
pp = ps;}

// p has at most one child
// save child pointer in c
BinaryTreeNode<E> *c;
if (p->LeftChild) c = p->LeftChild;
else c = p->RightChild;
```

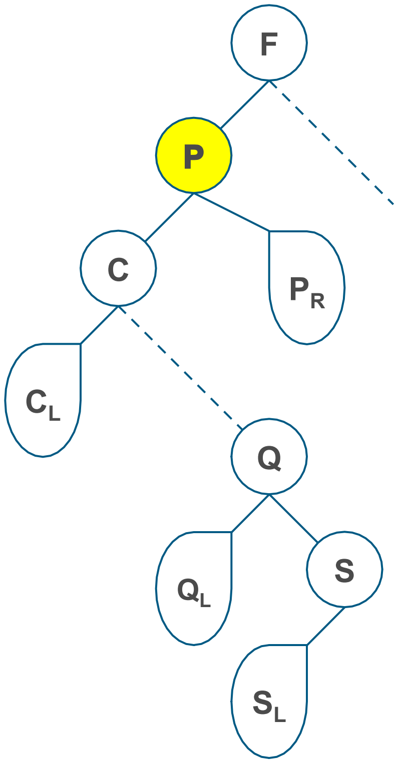


删除算法实现（续）

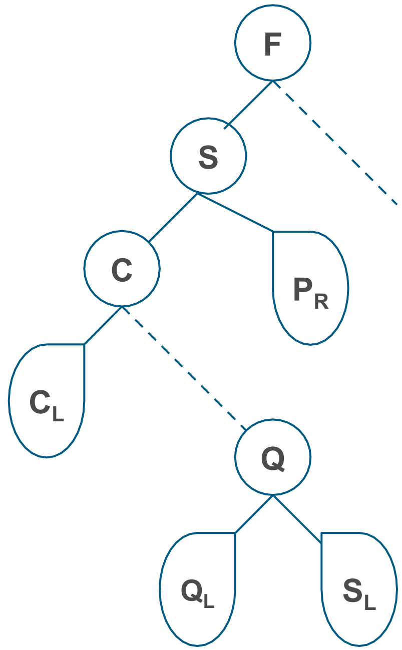
```
// delete p
if (p == root) root = c;
else {// is p left or right child of pp?
    if (p == pp->LeftChild)
        pp->LeftChild = c;
    else pp->RightChild = c;}
delete p;

return *this;
}
```



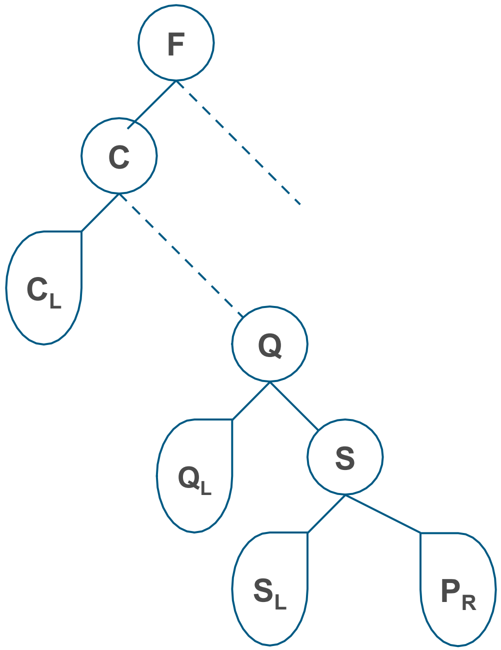
补充：另一种删除的思路



初始BST



思路一结果



思路二结果

两种思路哪种更好？

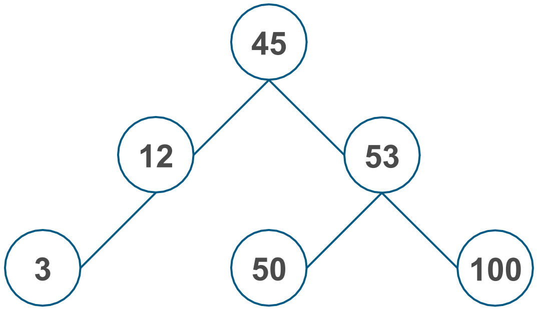
复杂性分析

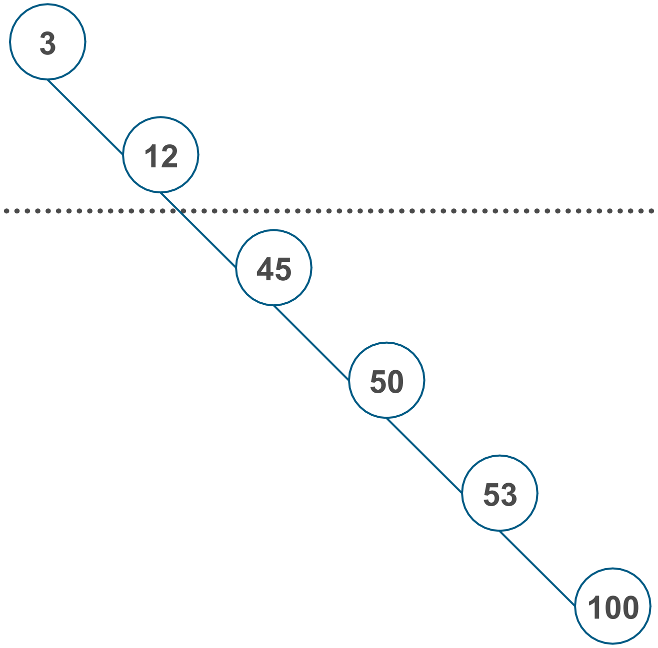
- 搜索、插入、删除的复杂性为O(h)
- 二叉搜索树的高度h，最坏情况为n
- 可证明，若搜索、插入、删除是随机的，平均情况为O(logn)
- 而升序输出为O(n)

主要内容

- 二叉搜索树
- AVL树
  - 定义
  - 搜索
  - 插入
  - 删除

动机





ASL=14/6

ASL=21/6

直观感受：BST越扁平越好！

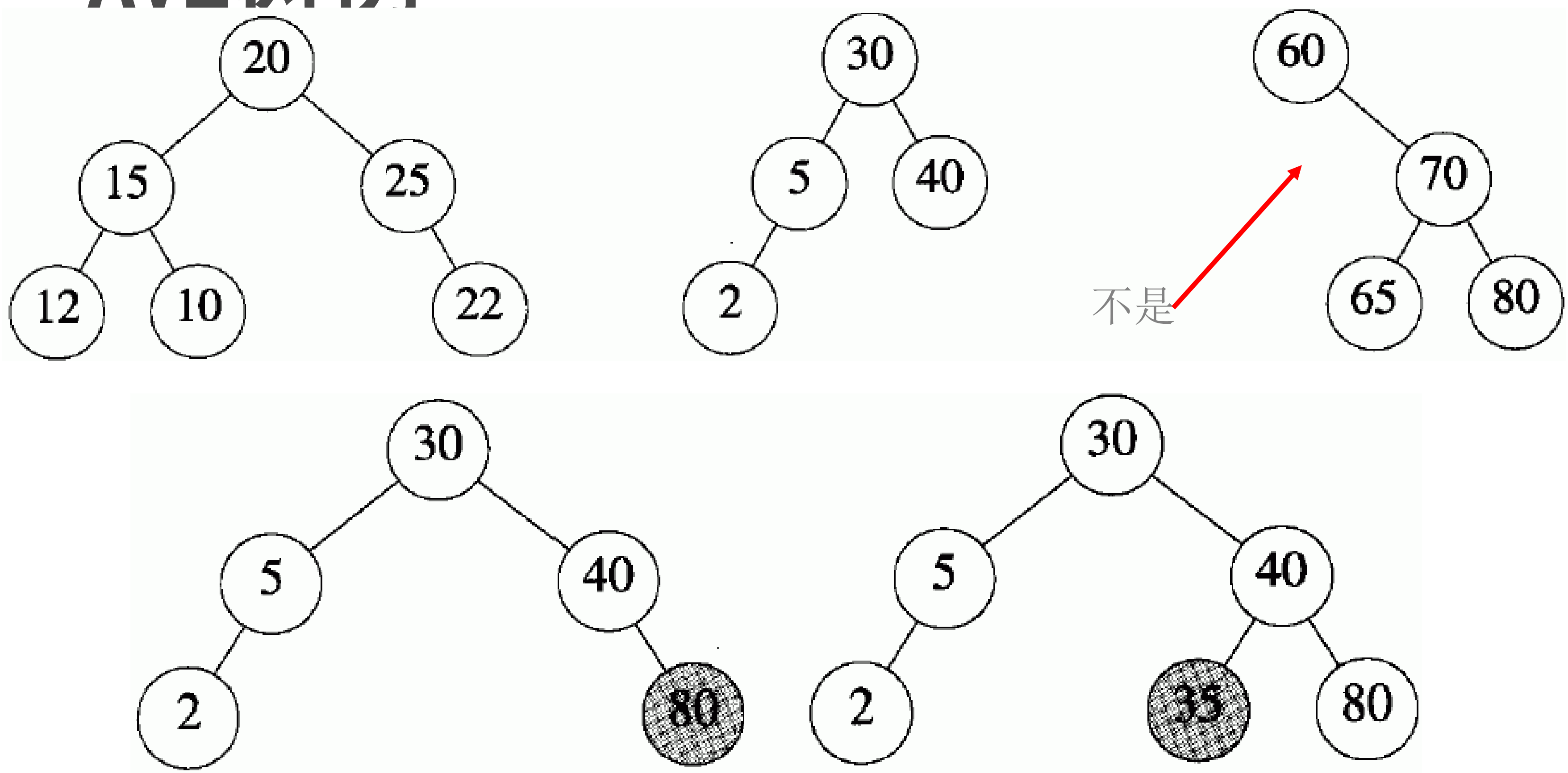
AVL树

- 如何提高二叉搜索树的最坏情况？
- 树高最坏情况保持在O(logn)
- AVL树——一种平衡树
- 1962年，Adelson-Velskii和Landis

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分别是其左子树和右子树，
  那么满足以下条件，T是一棵AVL树：
  1) T_L和T_R是AVL树
  2) $|h_L-h_R|\leqslant 1$，h_L和h_R分别是左子树和右子树的高度
- AVL搜索树

AVL树例



AVL树的特性

1. n个元素（节点）的AVL树的高度是O(logn)
2. 对于每一个n（$n\geq 0$）值，都存在一棵AVL树（保证任何时刻，插入操作都是可完成的）
3. 一棵n元素的AVL搜索树能在O(高度)=O(log$n$)的时间内完成搜索。

AVL树的特性

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$O(\log n)$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$O(\log n)$时间内完成

AVL树的高度

- 高度——复杂性
- 显然，平均情况不会低于随机二叉搜索树，O(logn)
- 最坏情况呢，如果也是O(n)，就失去改进的意义了
- n个节点的AVL树的高度最高是多少?

问题变换

- 高度为h的AVL树的最少节点数
  - F_h表示高度为h的节点数最少的AVL树
  - 左、右子树也是AVL树，高度一个为h-1，另一个为h-1或h-2
  - F_h节点数最少→子树中分别为F_{h-1}和F_{h-2}
  - $|F_h|=|F_{h-1}|+|F_{h-2}|+1$

AVL树的高度（续）

- $|F_h|+1=|F_{h-1}|+1+|F_{h-2}|+1$：菲波那契数列！

$$|F_h|+1=\frac{1}{\sqrt{5}}\left[\frac{1+\sqrt{5}}{2}\right]^{h+2}$$

- $h \approx 1.44\log_2|F_h| = 1.44\log_2 n = O(\log n)$

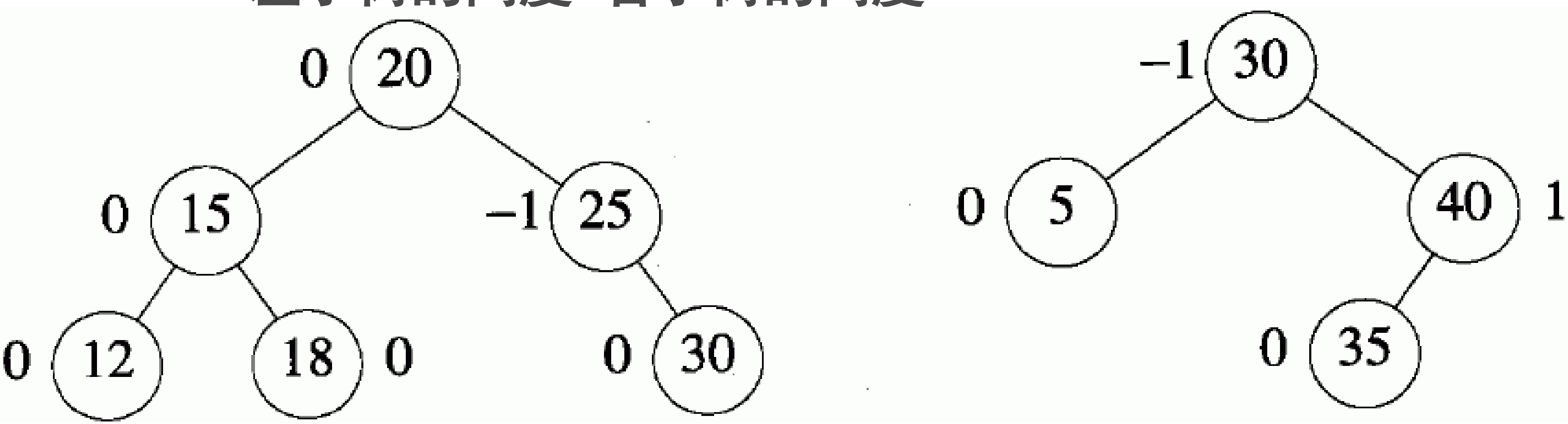
关于AVL树的一组值

- 高度为h的AVL树最少有几个节点？

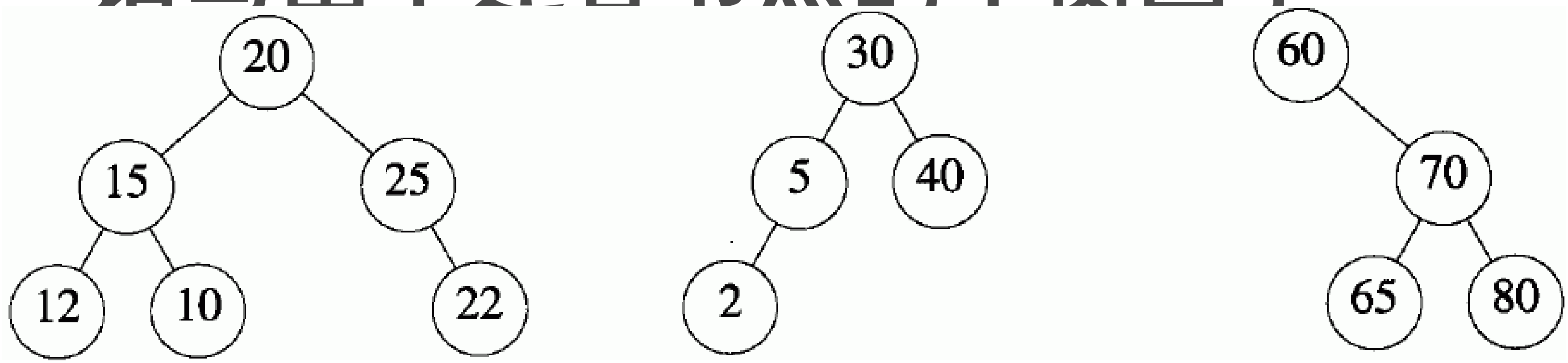
h	1	2	3	4	5	6	7	……
MinNum	**1**	**1+1**	**1+2+1**	**2+4+1**	**4+7+1**	**7+12+1**	**12+20+1**	……
	1	2	4	7	12	20	33	……

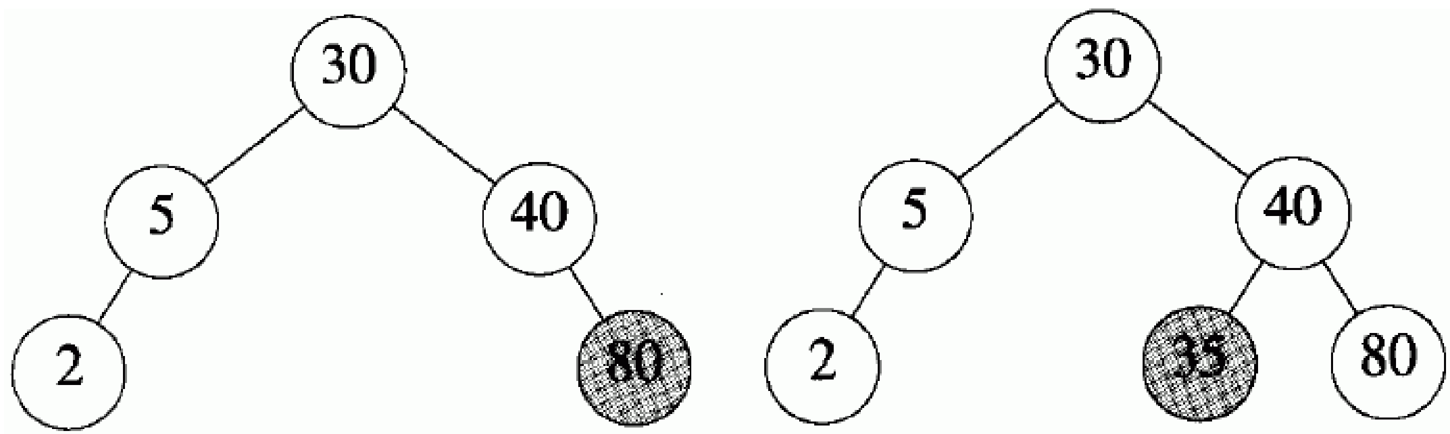
AVL树的描述

- 与BSTree类相似，增加平衡因子域bf
  - 左子树的高度-右子树的高度



请写出下述各节点的平衡因子



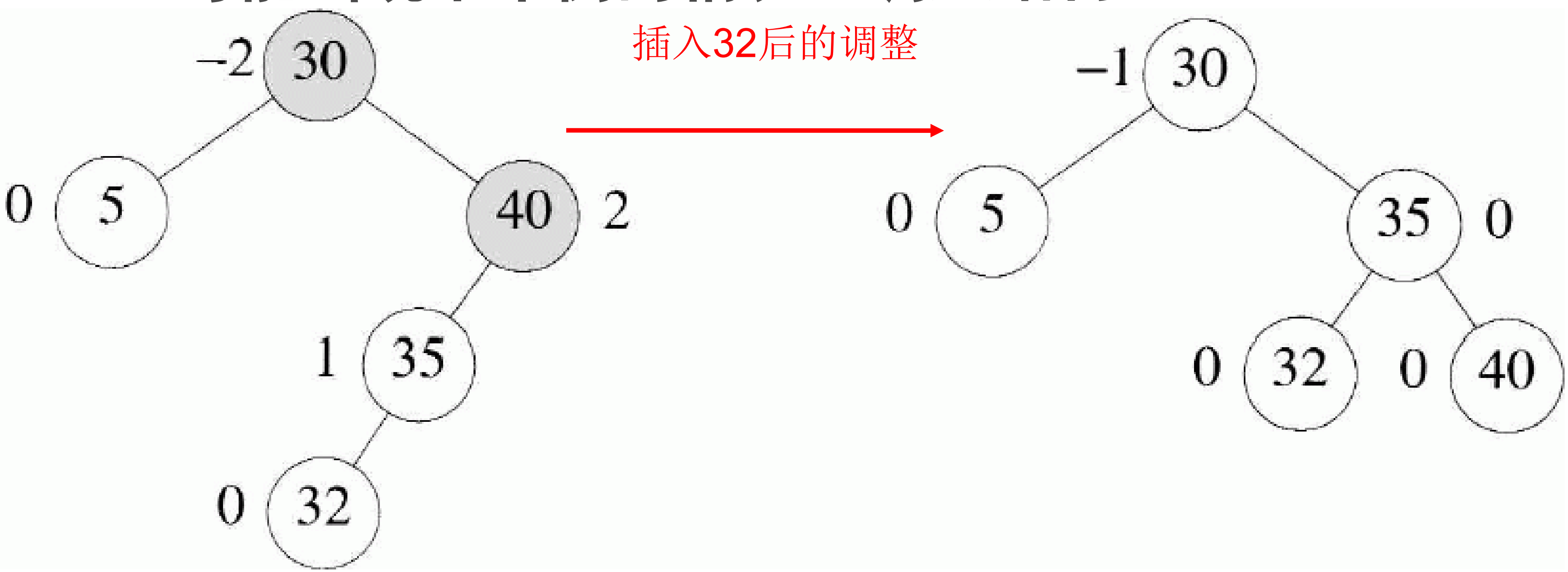


AVL搜索

- 与一般的二叉搜索树一致
  - 从根节点开始将key与节点值比较
  - 如果key小于节点值，进入左子树；
  - 如果key大于节点值，进入右子树；
  - 直到找到或子树为空停止。

AVL插入

- 首先利用二叉搜索树的插入算法
- 可能出现不平衡的情况→调整结构



插入导致的不平衡树的特性

1） 不平衡树中的平衡因子的值限于-2，-1，0，1和2

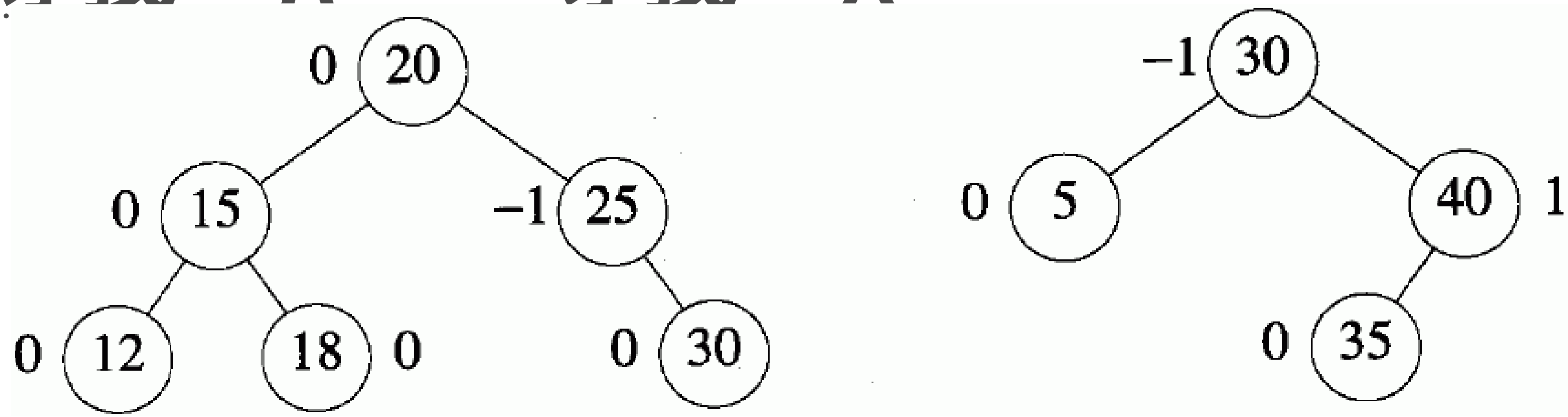
2） 平衡因子值为2的节点，在插入前其平衡因子为1
类似的，平衡因子值为-2的，插入前为-1

插入导致的不平衡树的特性

3) 只有从根到新插入节点路径上的节点，其平衡因子才会在插入操作后发生改变

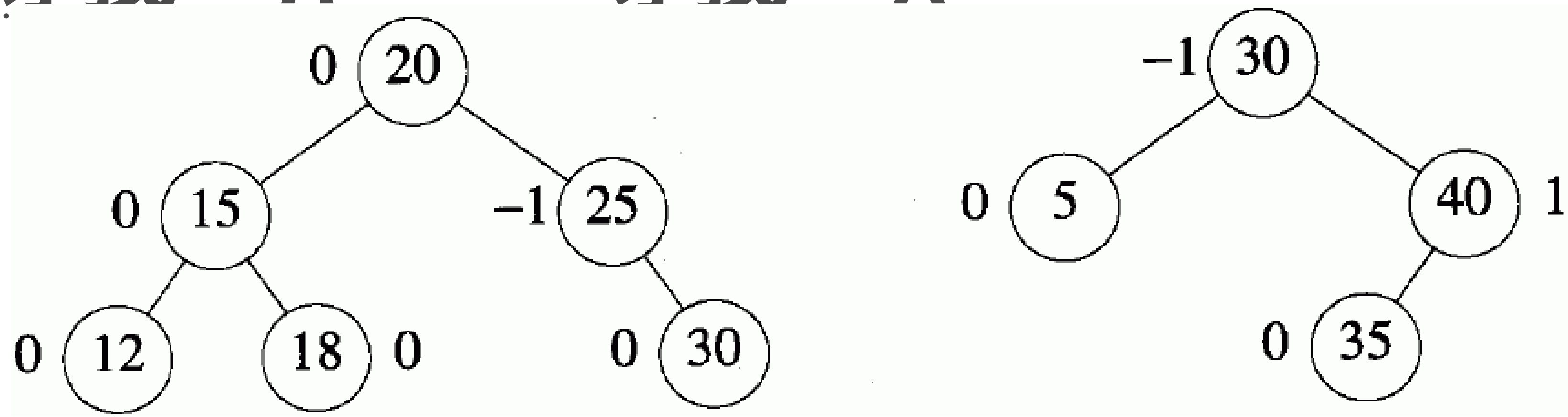
4) 假设*A*是离新插入节点最近的，平衡因子为-2或2的祖先节点，则在插入前，从*A*到新插入节点的路径上，所有节点的平衡因子都是0

寻找“A”—寻找“X”



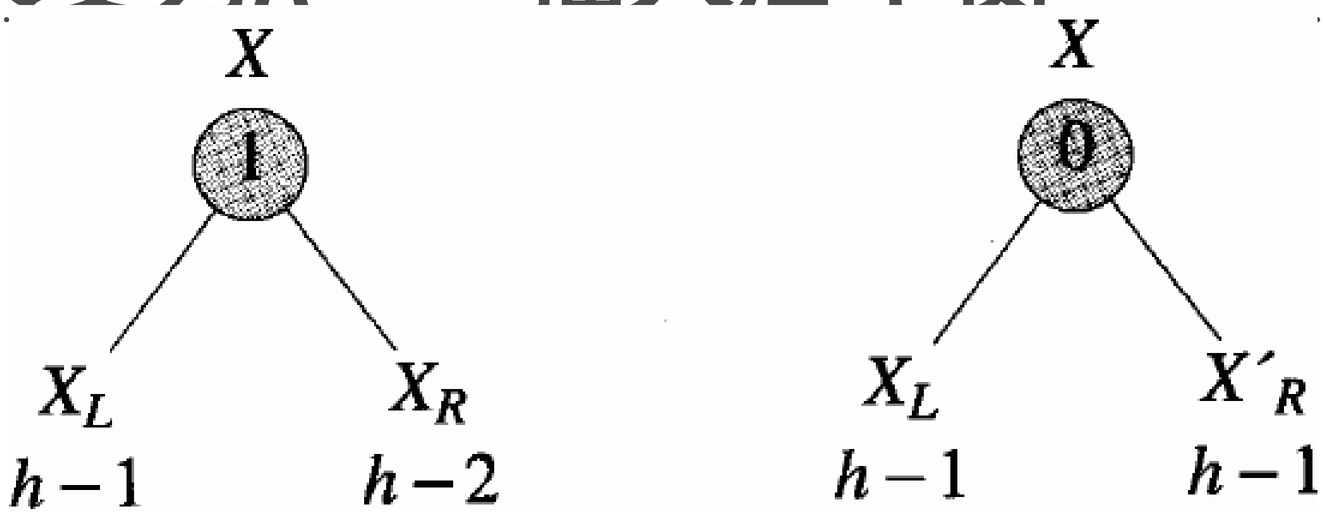
- 插入操作前，bf(A)必然为1或-1
- X——这样的节点中的“最后”一个
- 32插入右图，X——40

寻找“A” —寻找“X”



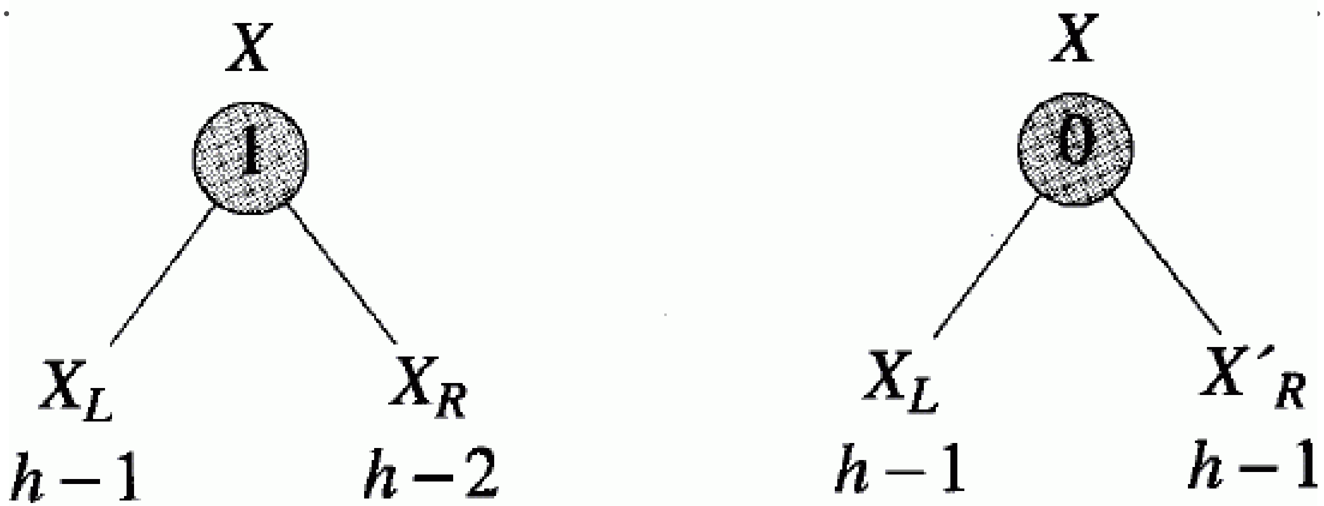
- 22、28、50插入左图，X——25
- 10、14、16、19插入左图，X——不存在
- X不存在——bf值全为0，插入不会导致不平衡

X未变为A——插入后平衡



- X的后代节点bf值全为0
  ➔插入后子树高度必然发生改变
  ➔X的bf值必然发生改变，变化为±1

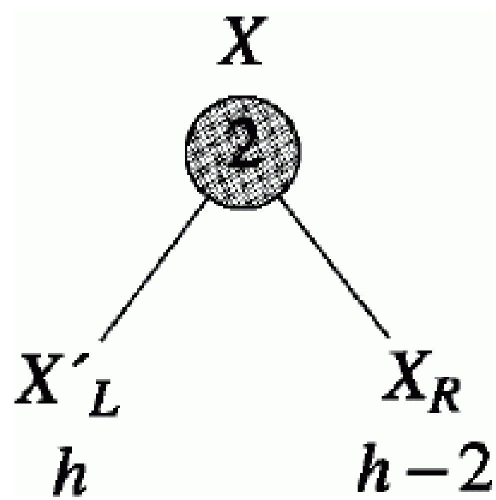
X未变为A——插入后平衡



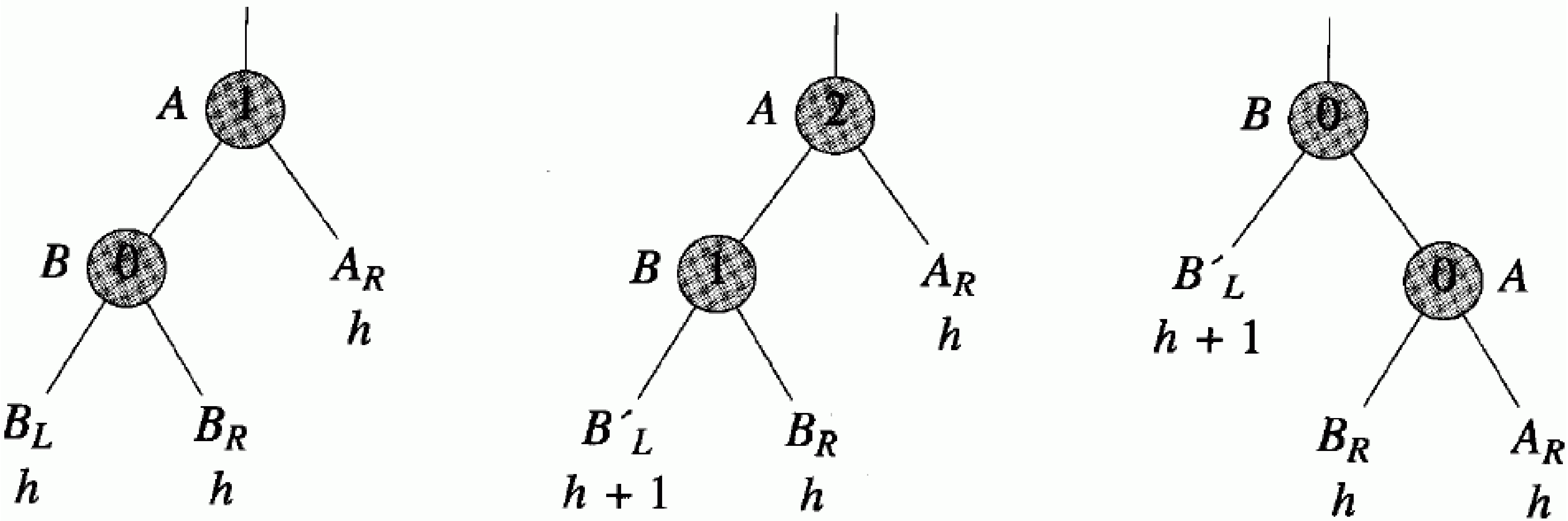
- ➔仍旧平衡的唯一可能——插入后bf(X)=0
- ➔插入前后以X为根的子树的高度未改变
- ➔新元素必然插入原来较矮的那棵子树

X变为A→找到了A

- bf值由-1变为-2（或1变为2）
- →新节点插入了较高的那棵子树
  - L型不平衡：左子树较高，新节点插入左子树
    - LL：新节点插入左子树的左子树
    - LR：插入左子树的右子树
  - R型不平衡：RL、RR

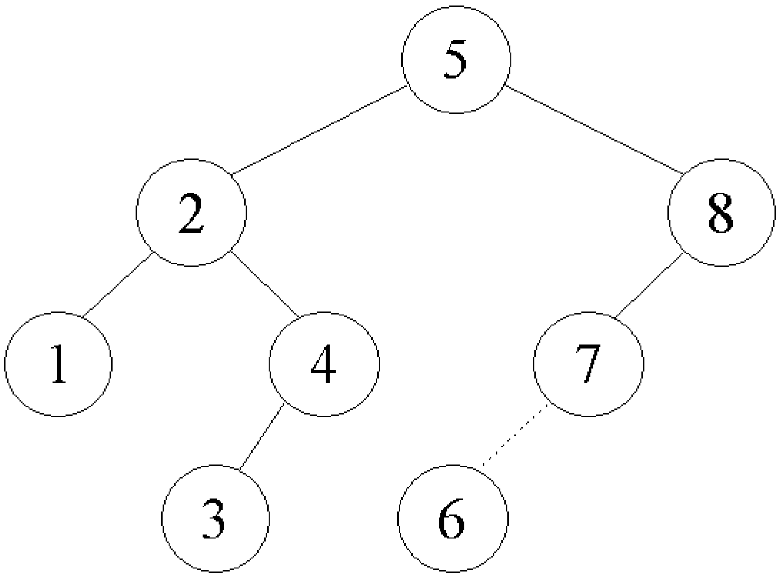


LL型不平衡及其调整方法

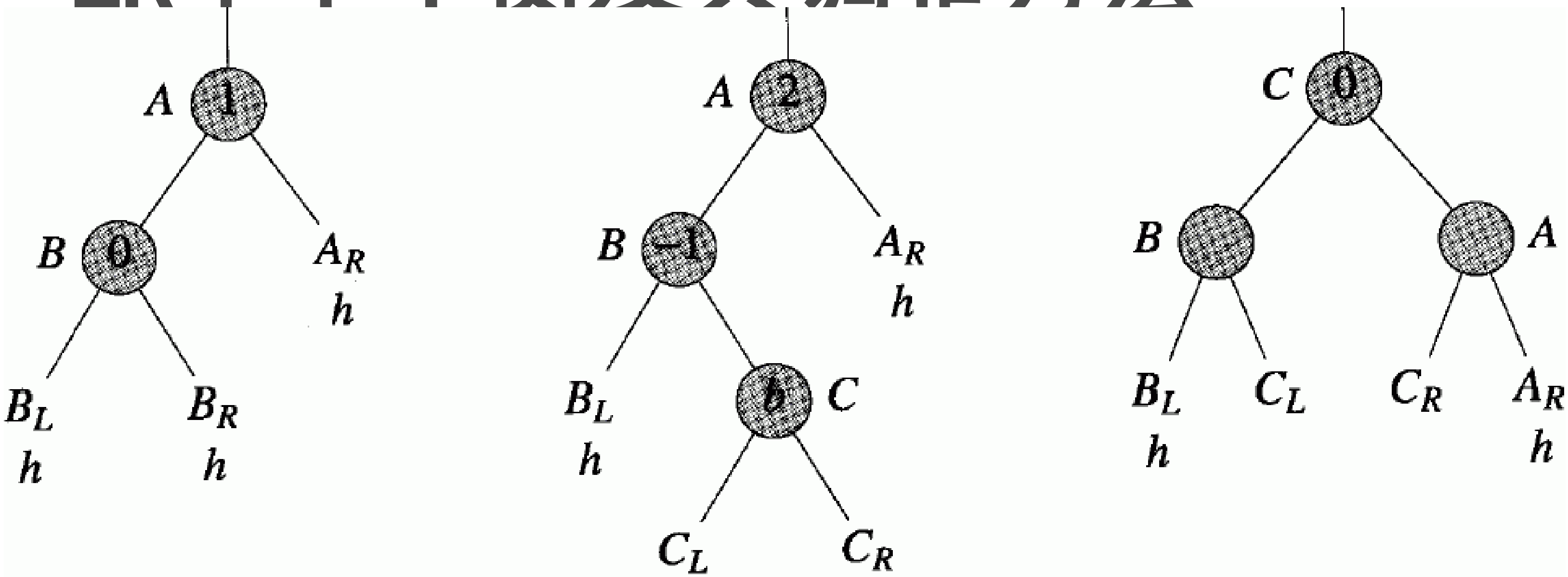


- 旋转后保持搜索树特性
  - $B'_L < B < B_R < A < A_R$
- 单旋转：RR——对称的

单旋转例

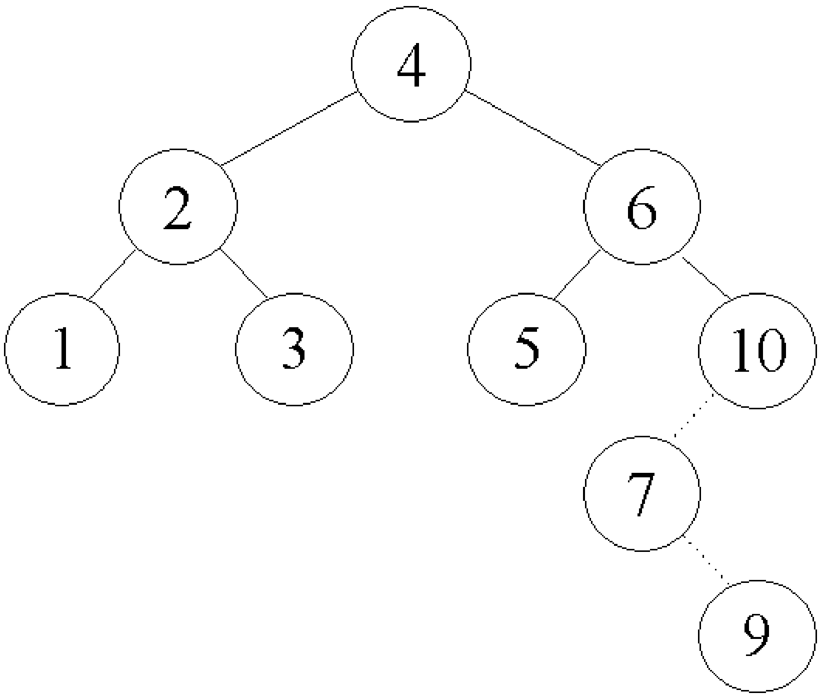


LR型不平衡及其调整方法

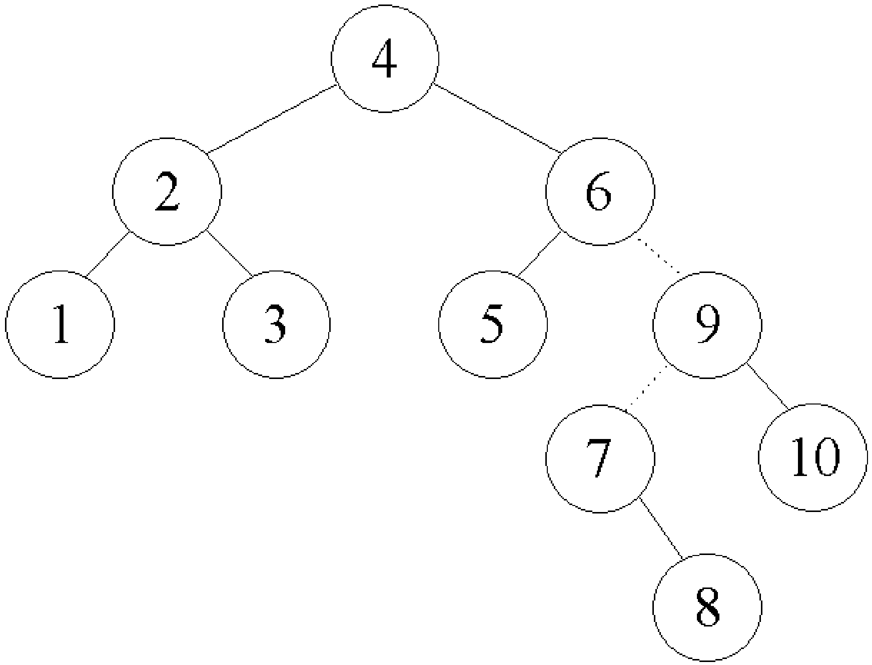


- b=0（C_L高度h，C_R高度h）➔bf(B)=bf(A)=0
  b=1（C_L高度h，C_R高度h-1）➔bf(B)=0，bf(A)=-1
  b=-1（C_L高度h-1，C_R高度h）➔bf(B)=1，bf(A)=0
- **RL——对称**

双旋转例



双旋转例



AVL树插入算法

1) 从根节点开始搜索，确定插入位置
   - 同时寻找最后的平衡因子为-1或1的节点，记为*A*
   - 若找到相同关键字的元素，插入失败
2) 若没有这样的节点*A*——插入后平衡
   - 从根节点重新遍历一次，修改平衡因子，然后终止

AVL树插入算法

3) 若*bf*(A)=1且新节点插入A的右子树，或*bf*(A)=-1且新节点插入A的左子树
➔A 的新平衡因子是0
   – 修改从A 到新节点路径中节点的平衡因子，然后终止

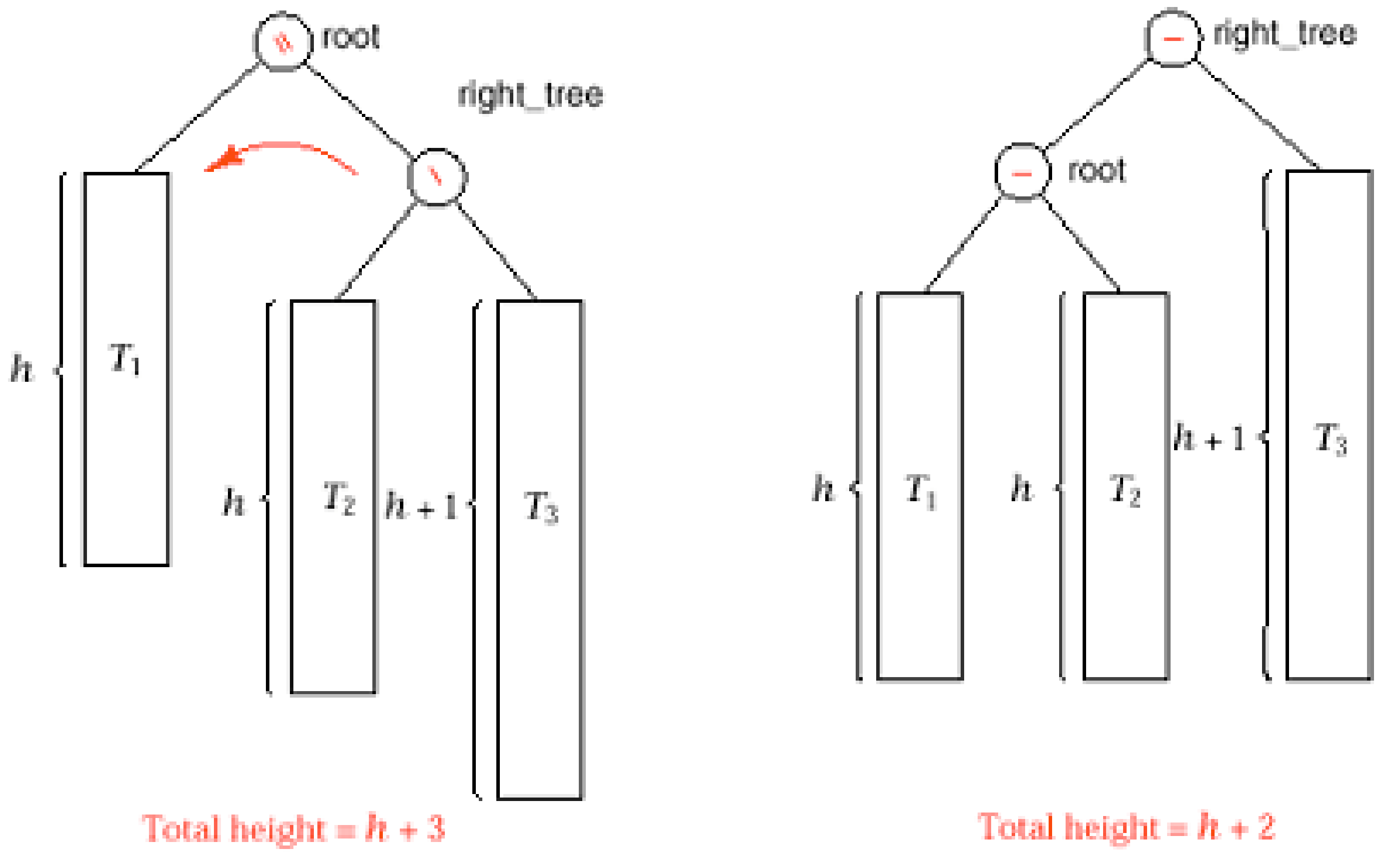
4) 不平衡情况
   – 确定不平衡类型，并执行相应的旋转
   – 在从新子树根节点至新插入节点路径中，根据旋转需要修改相应的平衡因子

AVL插入小结

- 不平衡共分四种情况
  - LL与RR对称：一次旋转
  - LR与RL对称：两次旋转

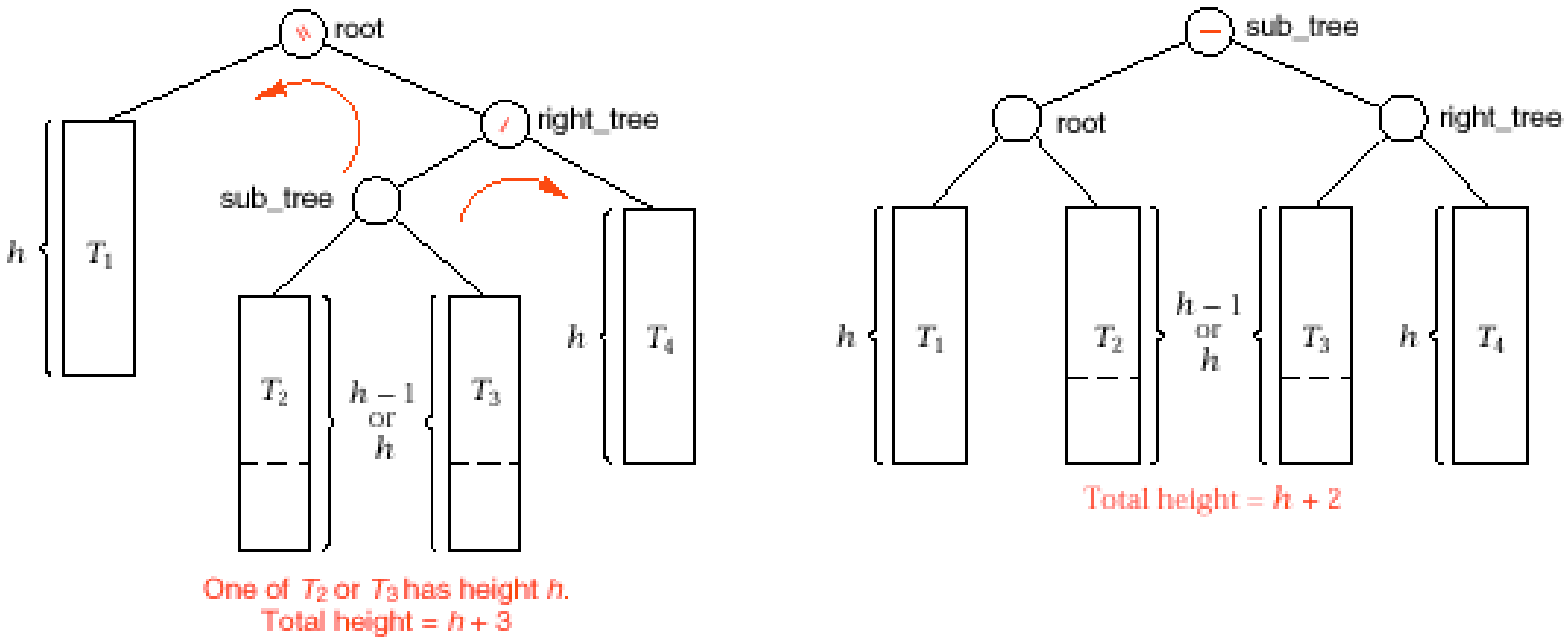
- 关键是找到距离插入点最近的**不平衡节点**！

一次旋转示例（RR）



root
right_tree
h
T_1
h
T_2
$h+1$
T_3
Total height = $h+3$
right_tree
root
h
T_1
h
T_2
$h+1$
T_3
Total height = $h+2$

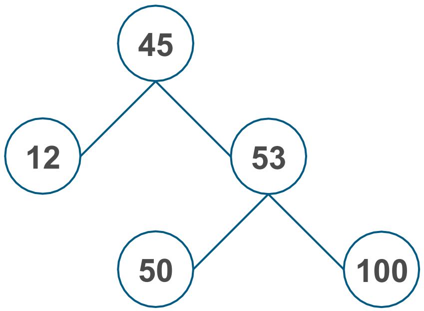
两次旋转示例（RL）



root
right_tree
sub_tree
h
T_1
T_2
T_3
T_4
h − 1
or
h
One of T_2 or T_3 has height h.
Total height = $h + 3$
sub_tree
root
right_tree
Total height = $h + 2$

小练习

- 请将元素52插入到如下AVL树中，生成新的AVL树

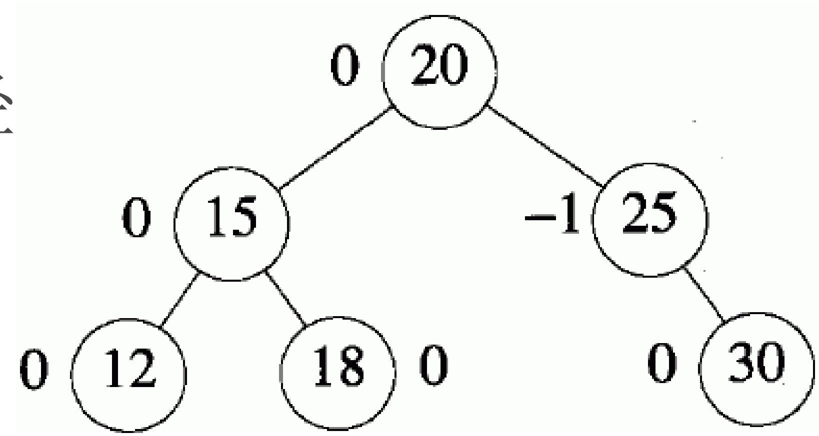


练习

- 设有一个关键字序列 {55, 31, 11, 37, 46, 73, 63, 2, 7}
  - 从空树开始构造平衡二叉树，画出每加入一个新节点时二叉树的形态。若发生不平衡，指明需做的旋转类型及旋转结果
  - 计算最终平衡二叉树在等概率下的查找成功平均查找长度和查找不成功平均查找长度

AVL删除

- 首先执行二叉搜索树的删除，如不平衡则调整
- 从删除节点的父节点开始到根节点，对路径上的每个节点q，根据新的平衡因子，判断子树高度变化，调整方法，以及是否继续调整过程



根据删除后q的平衡因子进行调整

1） 0：子树平衡，无需调整，但其高度减少了1，需继续向上，改变祖先节点平衡因子，继续修正过程

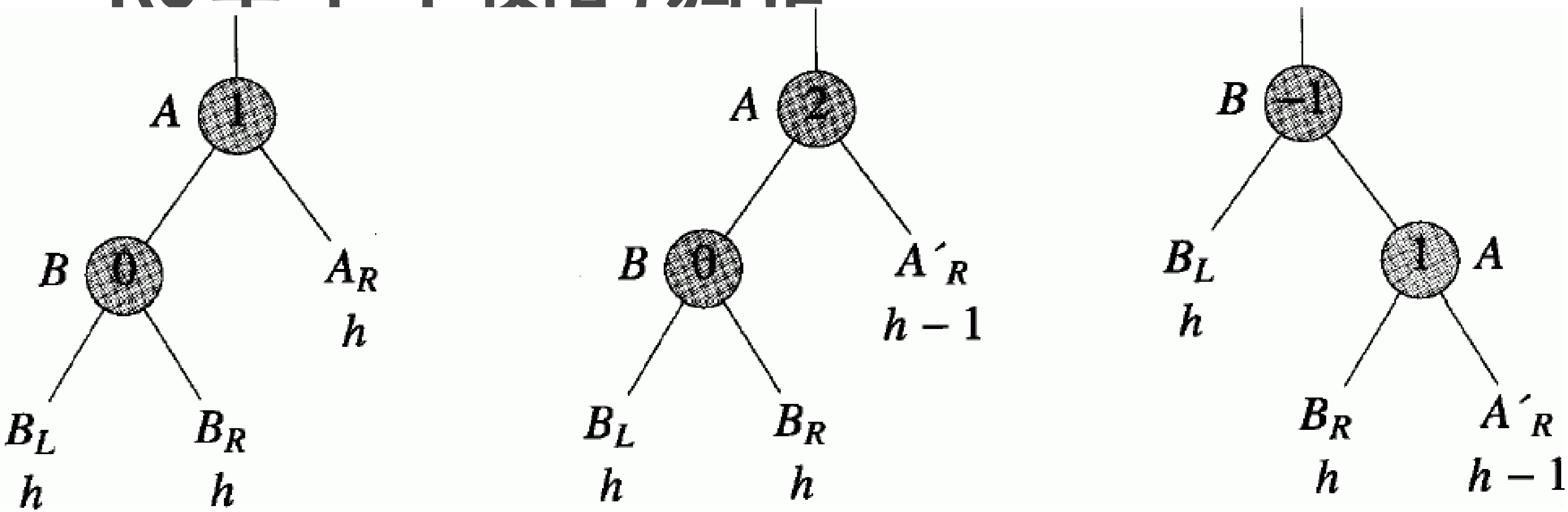
2） 1或-1：子树平衡，且高度未变，修正过程终止

3） 2或-2：子树不平衡，需调整，根据调整结果判断是否继续向上修正过程

不平衡子树的调整方法

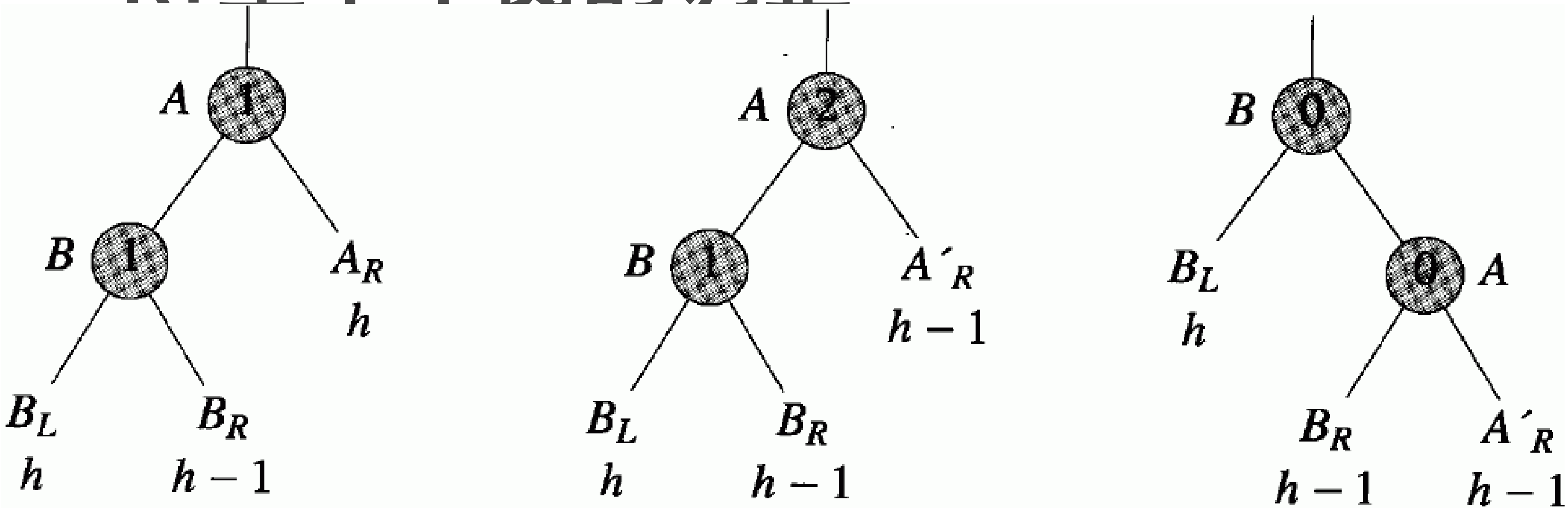
- 不失一般性，假定删除发生在q的右子树，R型不平衡
  - 删除前bf(A)=1，删除后bf(A)=2
  - 令q的左子树为p
    - bf(p)=0，R0型不平衡
    - bf(p)=1，R1型不平衡
    - bf(p)=-1，R-1型不平衡
- 删除发生在左子树的情况类似

R0型不平衡的调整



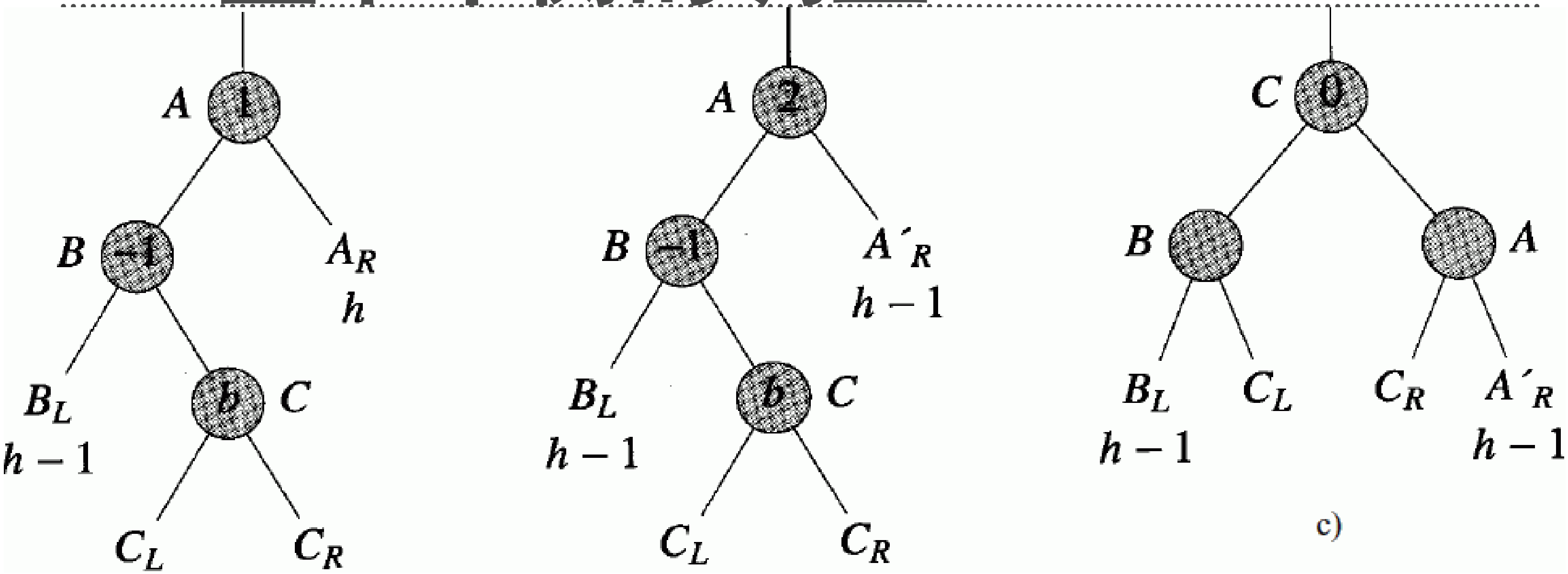
- 高度未变，祖先平衡因子不需调整
- 与LL旋转类似

R1型不平衡的调整



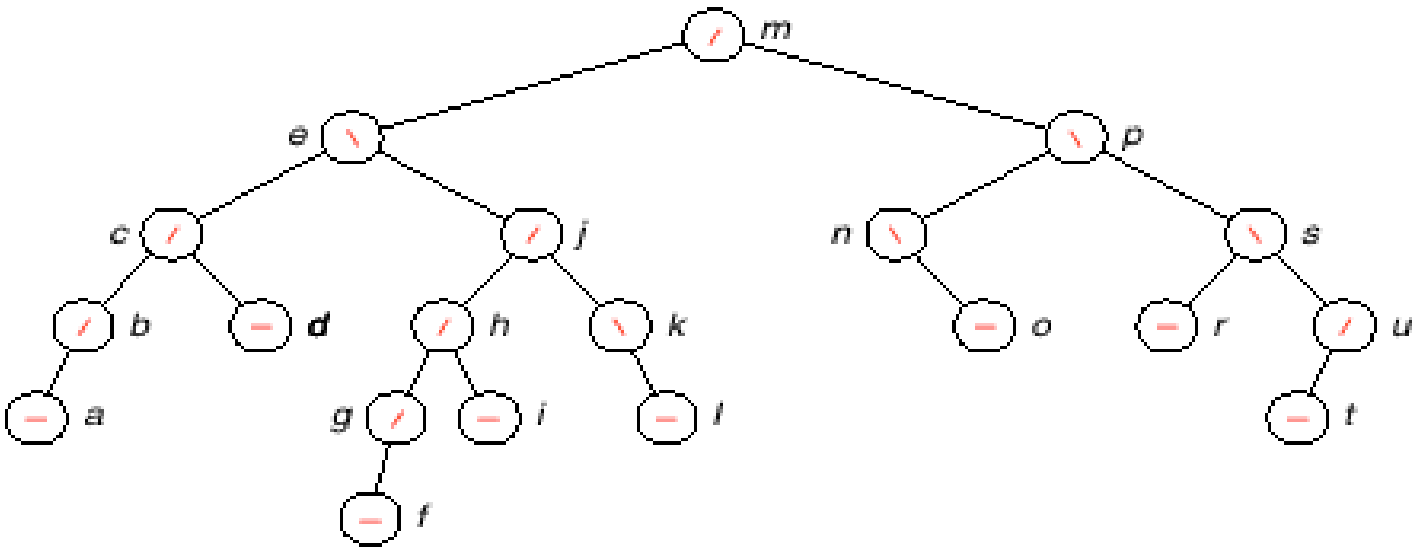
- 高度减少1，继续修正过程
- 与LL旋转类似

R-1型不平衡的调整

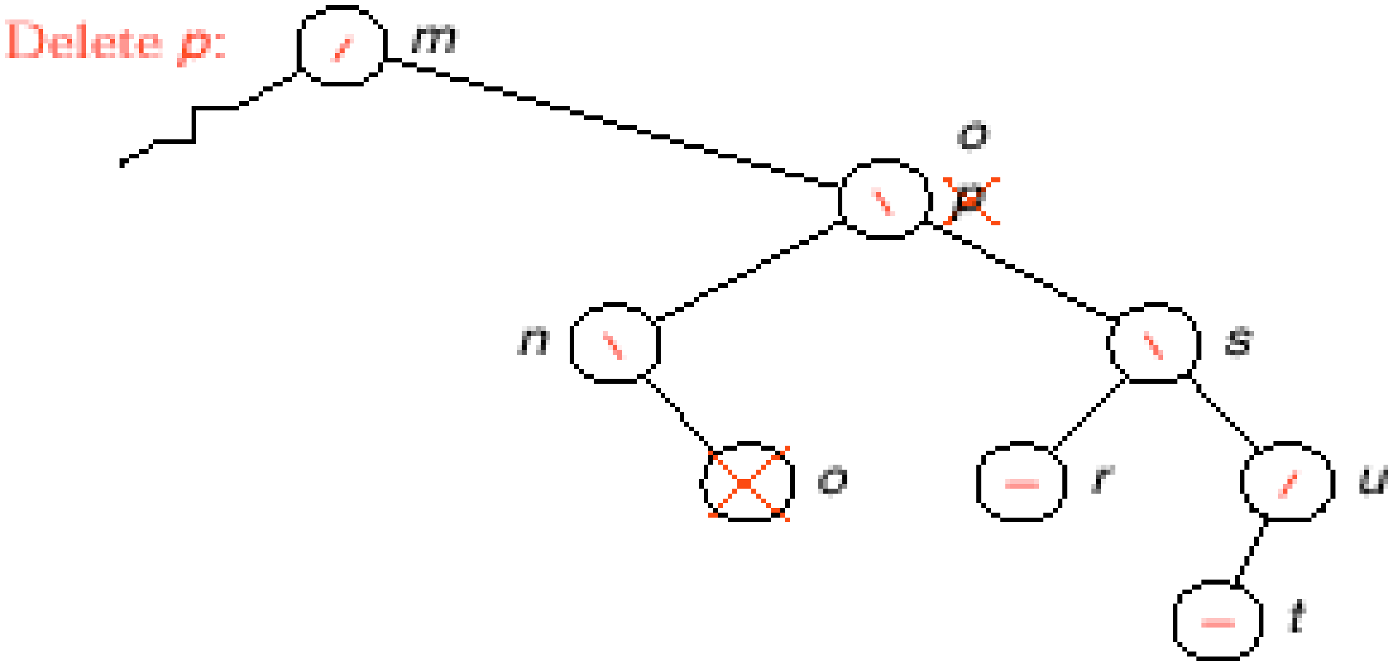


- b=0（C_L高度h-1，C_R高度h-1）➔bf(B)=bf(A)=0
  b=1（C_L高度h-1，C_R高度h-2）➔bf(B)=0，bf(A)=-1
  b=-1（C_L高度h-2，C_R高度h-1）➔bf(B)=1，bf(A)=0
- 高度减少1，需继续修正过程
- 与LR旋转类似

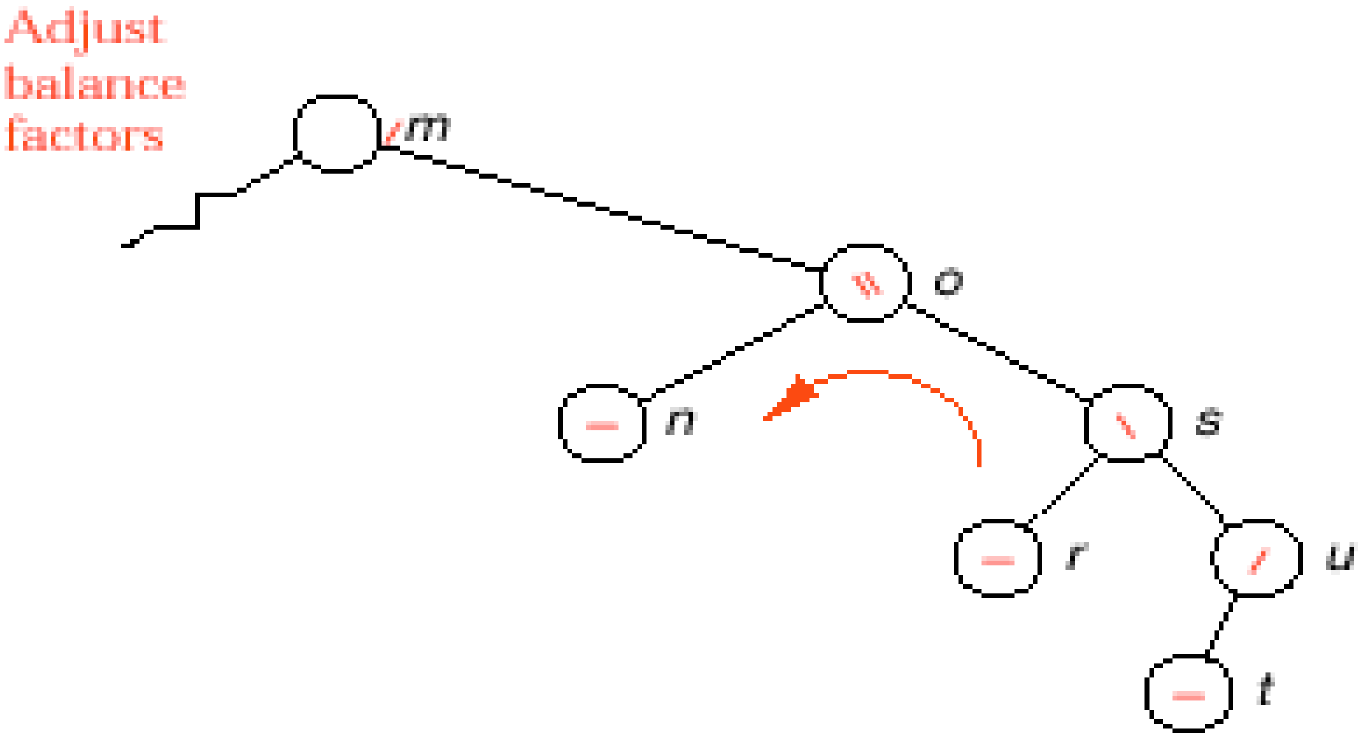
例题：初始状态



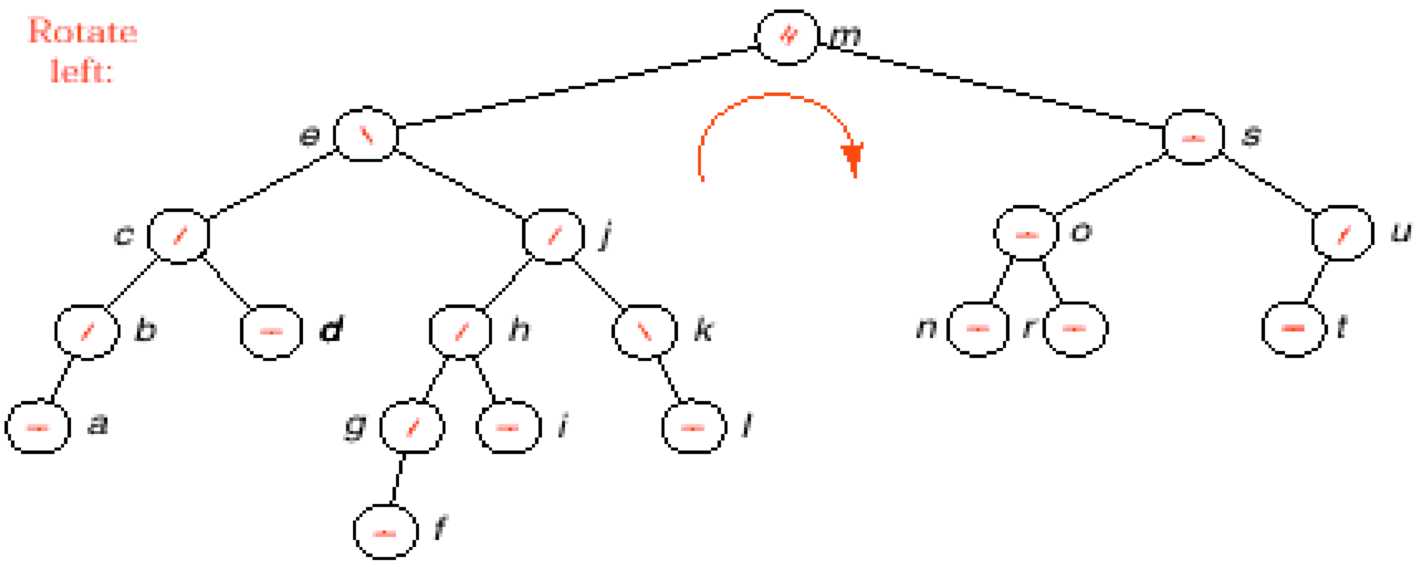
删除p



判断平衡性

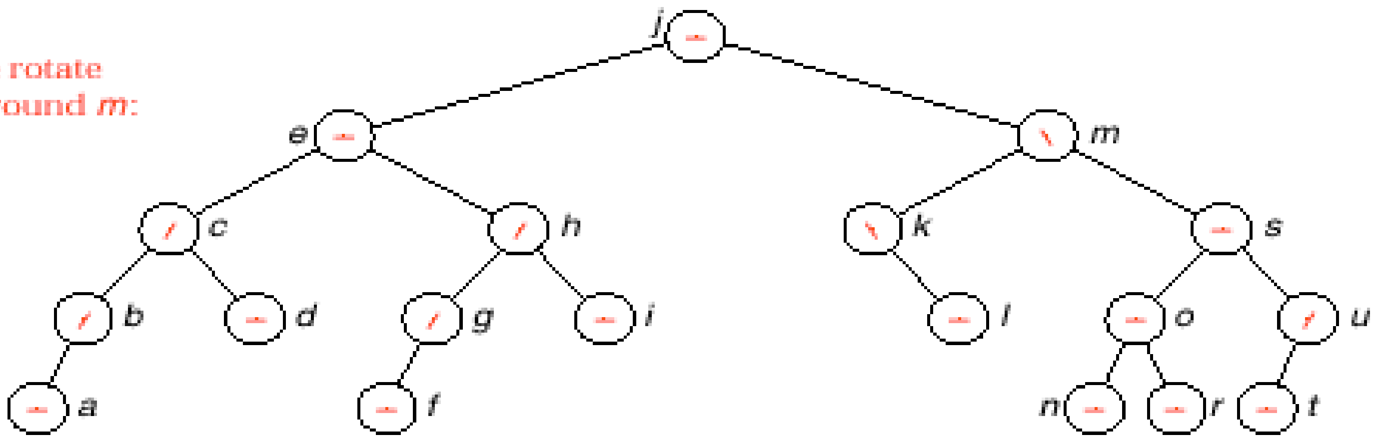


上一级仍然不平衡



最终结果

Double rotate
right around *m*:

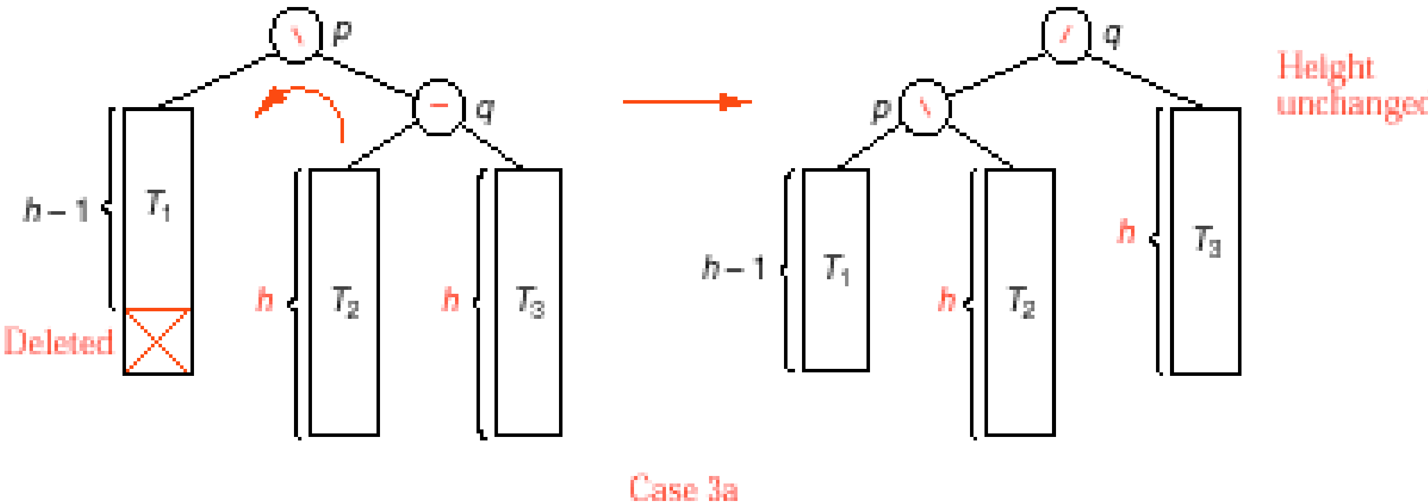


AVL删除小结

- 不平衡共分三种情况
  - 一次旋转即停止
  - 一次旋转不平衡
  - 两次旋转不平衡

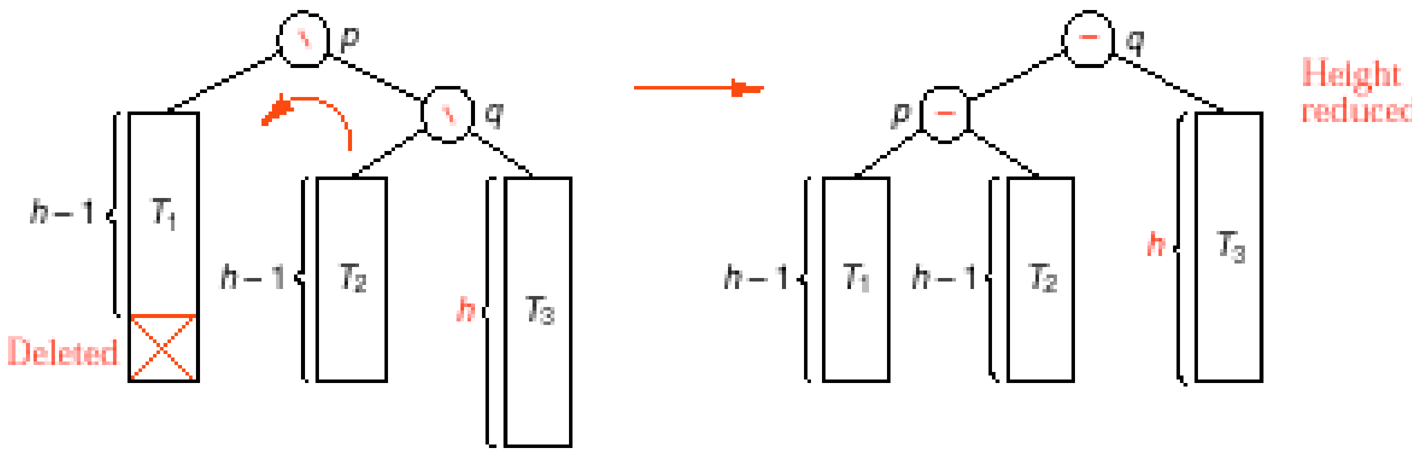
- 关键是考察不平衡节点的**另一棵子树**！

一次旋转即停止



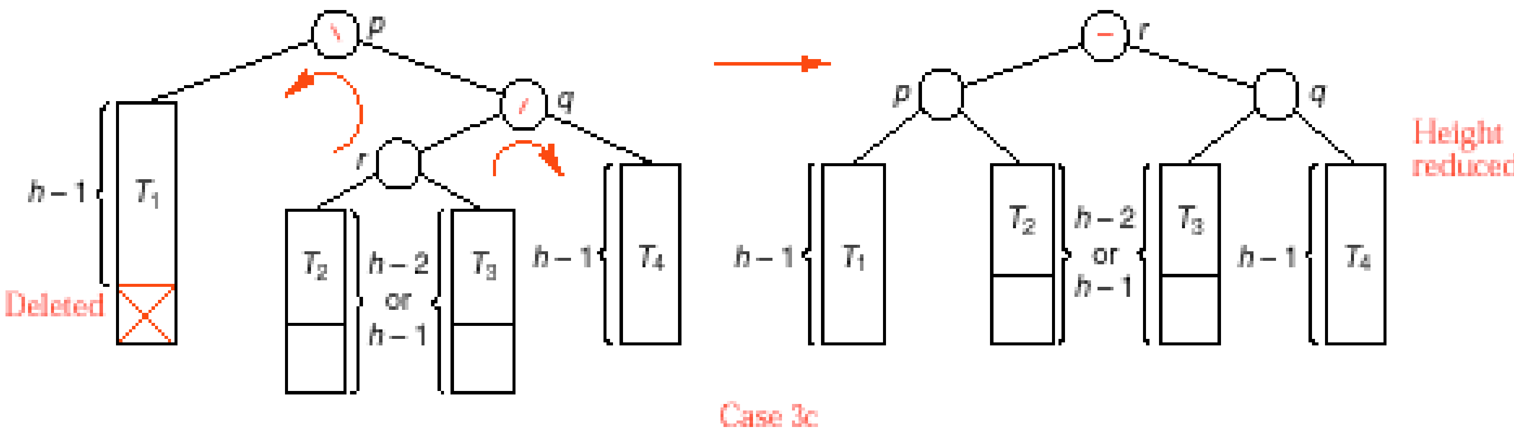
p
q
h − 1
T_1
Deleted
h
T_2
h
T_3
q
p
Height
unchanged
h − 1
T_1
h
T_2
h
T_3
Case 3a

一次旋转不平衡

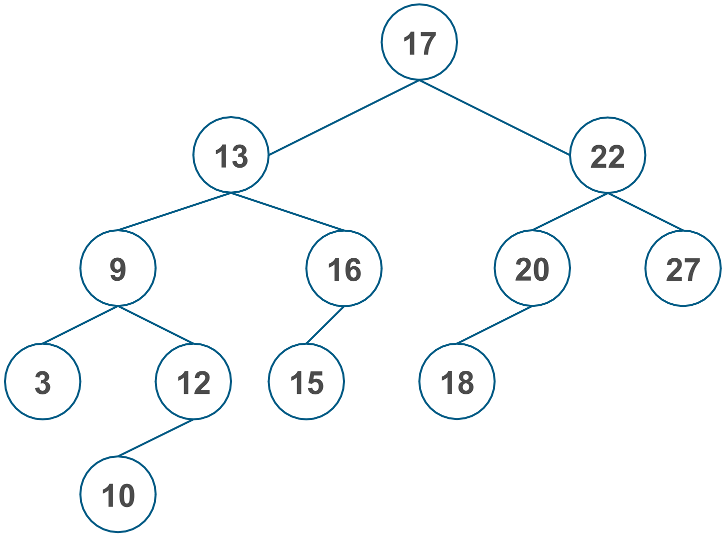


Case 3b

两次旋转不平衡



练习



- **在如图AVL树中依次删除22, 3, 10, 9，画出每步处理后树的形态**

本章结束