第9章 查找

杭州電子科技大学
HANGZHOU DIANZI UNIVERSITY

知识回顾：二叉排序树的查找

含有n个结点的二叉排序树的平均查找长度和树的形态有关

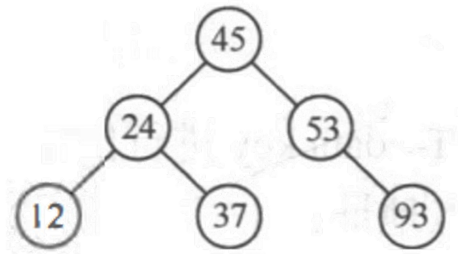
最好情况： ASL = $\log_2(n+1)-1$ (形态比较均衡)

$O(\log_2 n)$

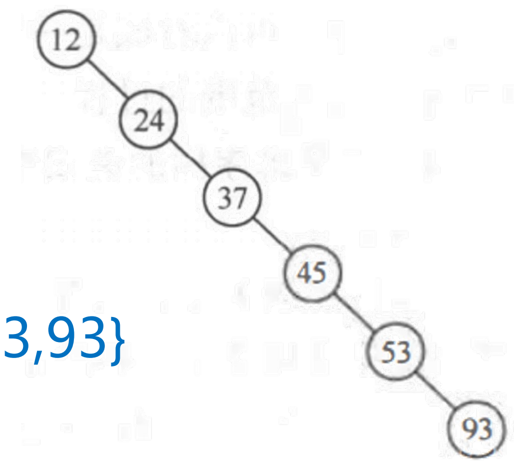
最坏情况： ASL = $(n+1) / 2$ (单支树的形态)

$O(n)$ 查找效率与顺序查找情况相同

例如： 初始序列 {45,24,53,12,37,93}



初始序列 {12,24,37,45,53,93}



二叉排序树

问题：如何提高形态不均衡的二叉排序树的查找效率？

解决办法：做“平衡化”处理，即尽量让二叉树的形状均衡！

平衡二叉树

平衡二叉树

1. 平衡二叉树的定义

平衡二叉树 (balanced binary tree)

- 又称AVL树(**A**delson-**V**elskii and **L**andis)。
- 一棵平衡二叉树或者是空树，或者是具有下列性质的二叉排序树:
  - ① 左子树与右子树的高度之差的绝对值小于等于1;
  - ② 左子树和右子树也是平衡二叉排序树。

平衡二叉树

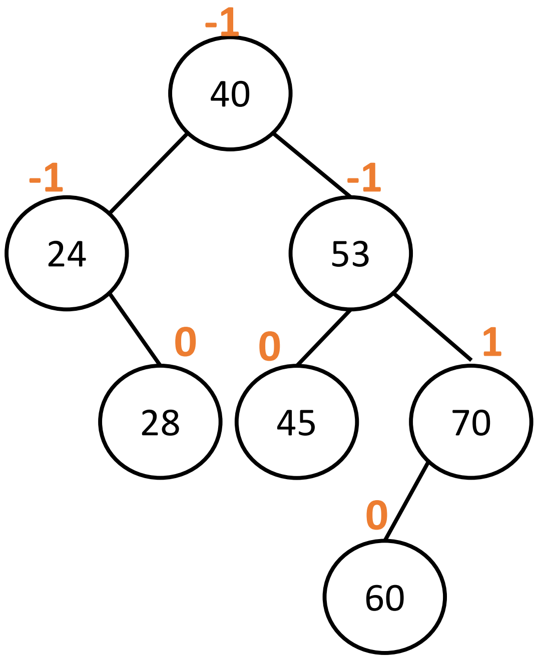
杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

为了方便起见，给每个结点附加一个数字，给出该结点左子树与右子树的高度差。这个数字称为结点的平衡因子(BF)。

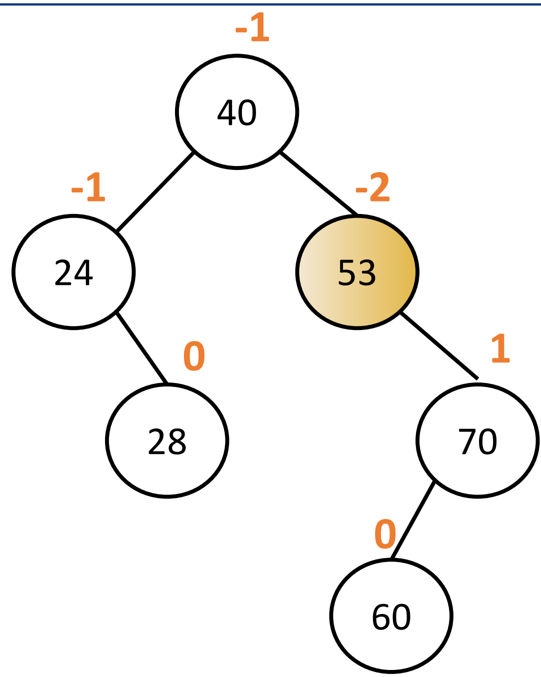
平衡因子 = 结点左子树的高度 - 结点右子树的高度

根据平衡二叉树的定义，平衡二叉树上所有结点的平衡因子只能是-1、0，或1。

平衡二叉树



√ 是平衡树



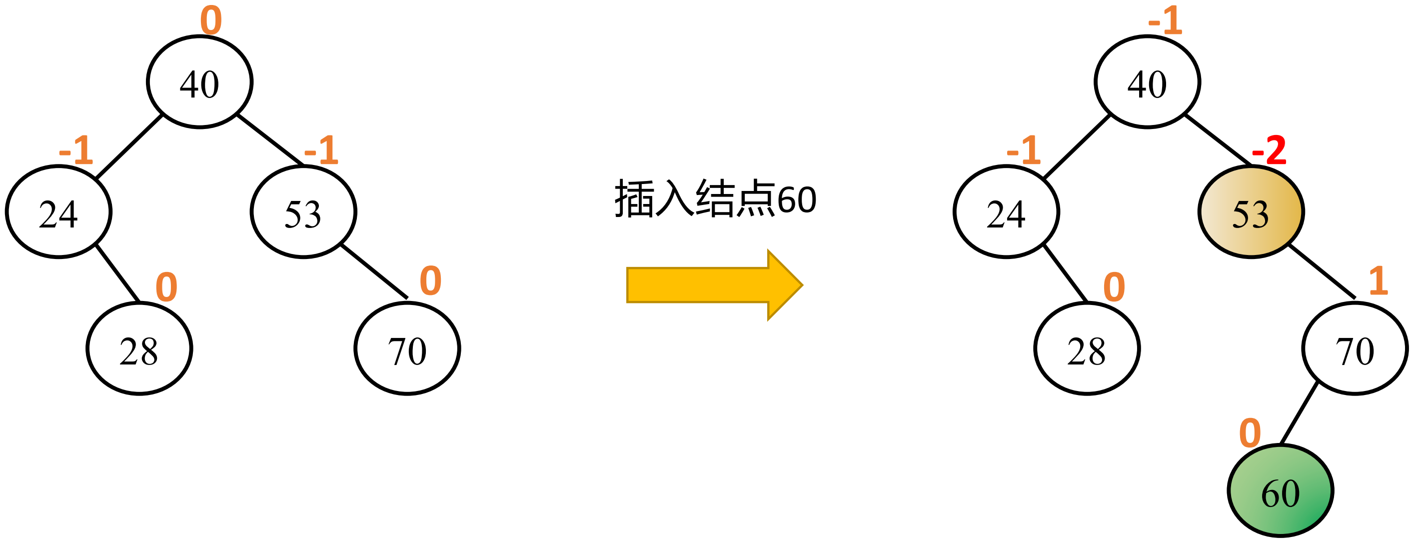
× 不是平衡树

对于一棵有n个结点的AVL树，其高度保持在$O(\log_2 n)$数量级，ASL也保持在$O(\log_2 n)$量级

失衡二叉排序树的分析与调整

杭州電子科技大学 HANGZHOU DIANZI UNIVERSITY

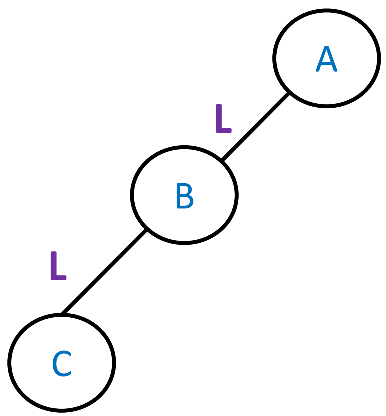
当我们在一个平衡二叉排序树上插入一个结点时，有可能导致失衡，即出现平衡因子绝对值大于1的结点，如: 2、-2。



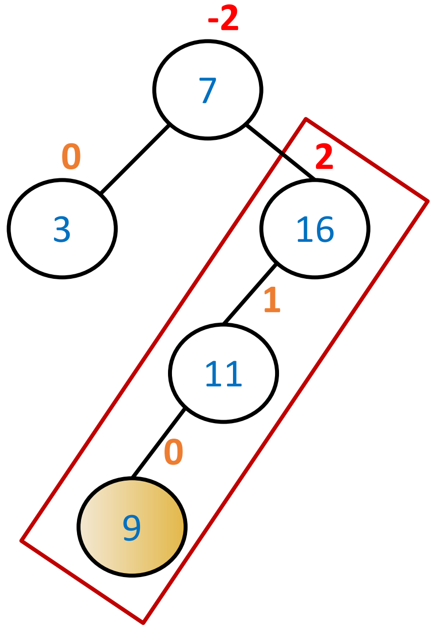
如果在一棵AVL树中插入一个新结点后造成失衡，则必须重新调整树的结构，使之恢复平衡。

失衡二叉排序树的分析与调整

平衡调整的四种类型：



插入结点9



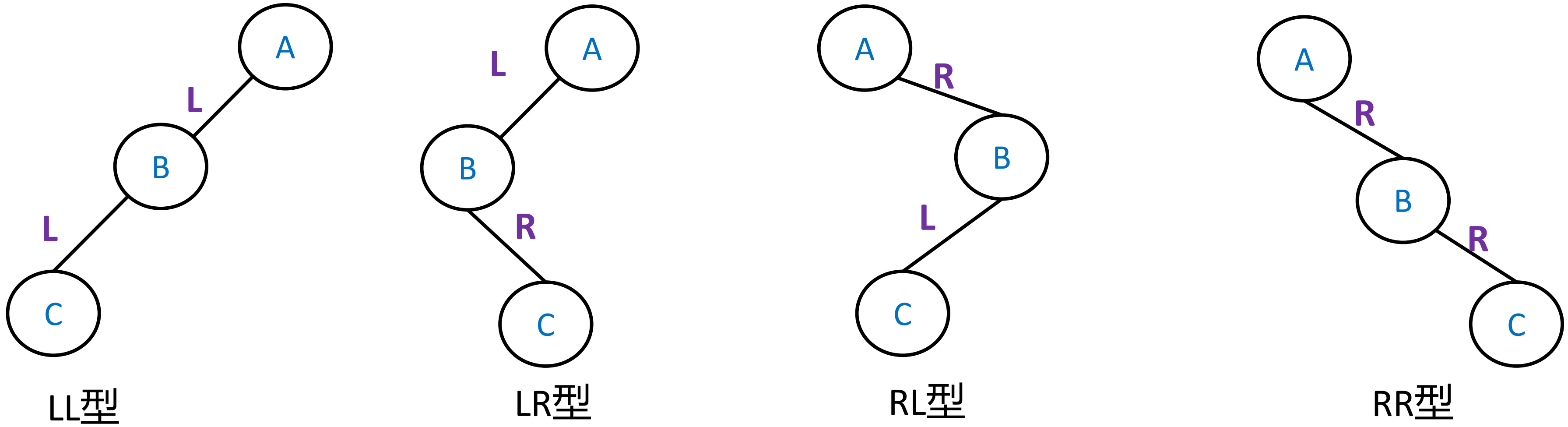
A: 失衡结点　　　不止一个失衡结点时，为最小失衡子树的根结点

B: A结点的孩子，C结点的双亲

C: 插入新结点的子树

失衡二叉排序树的分析与调整

平衡调整的四种类型：



A: 失衡结点　　不止一个失衡结点时，为最小失衡子树的根结点

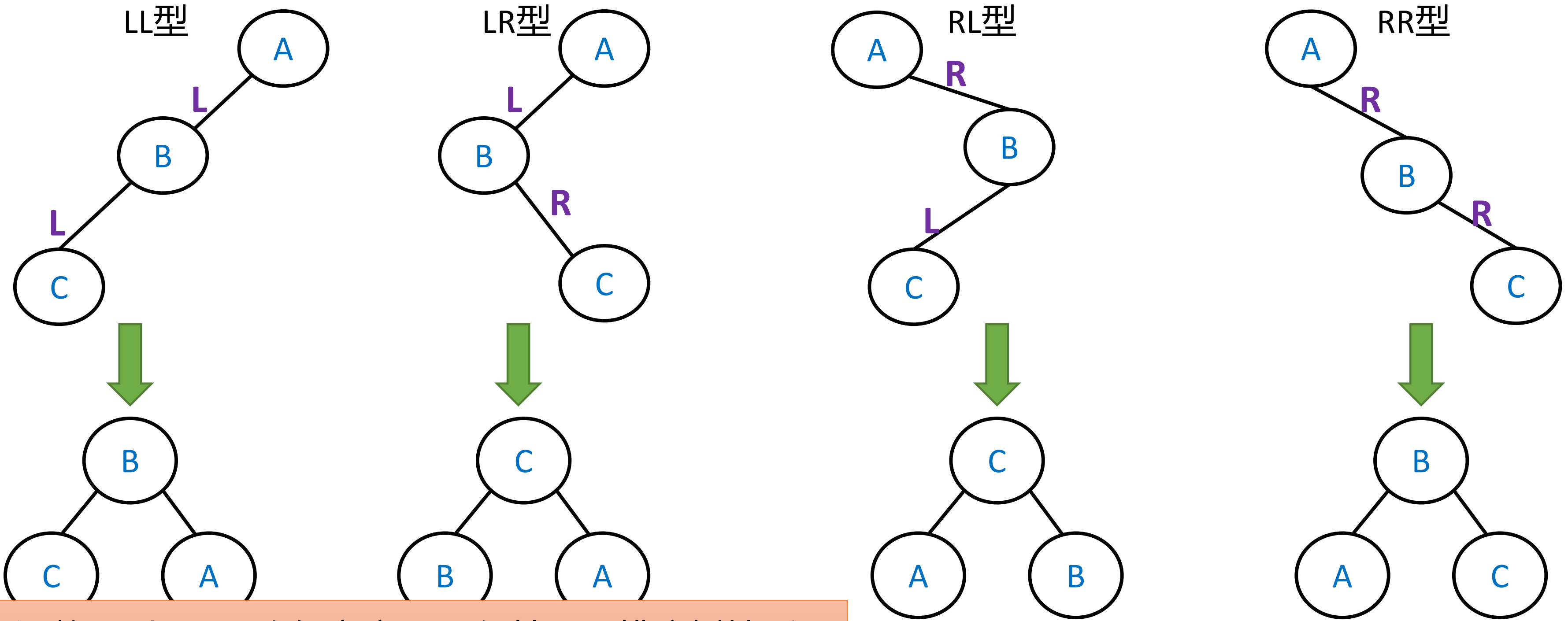
B: A结点的孩子，C结点的双亲

C: 插入新结点的子树

失衡二叉排序树的分析与调整

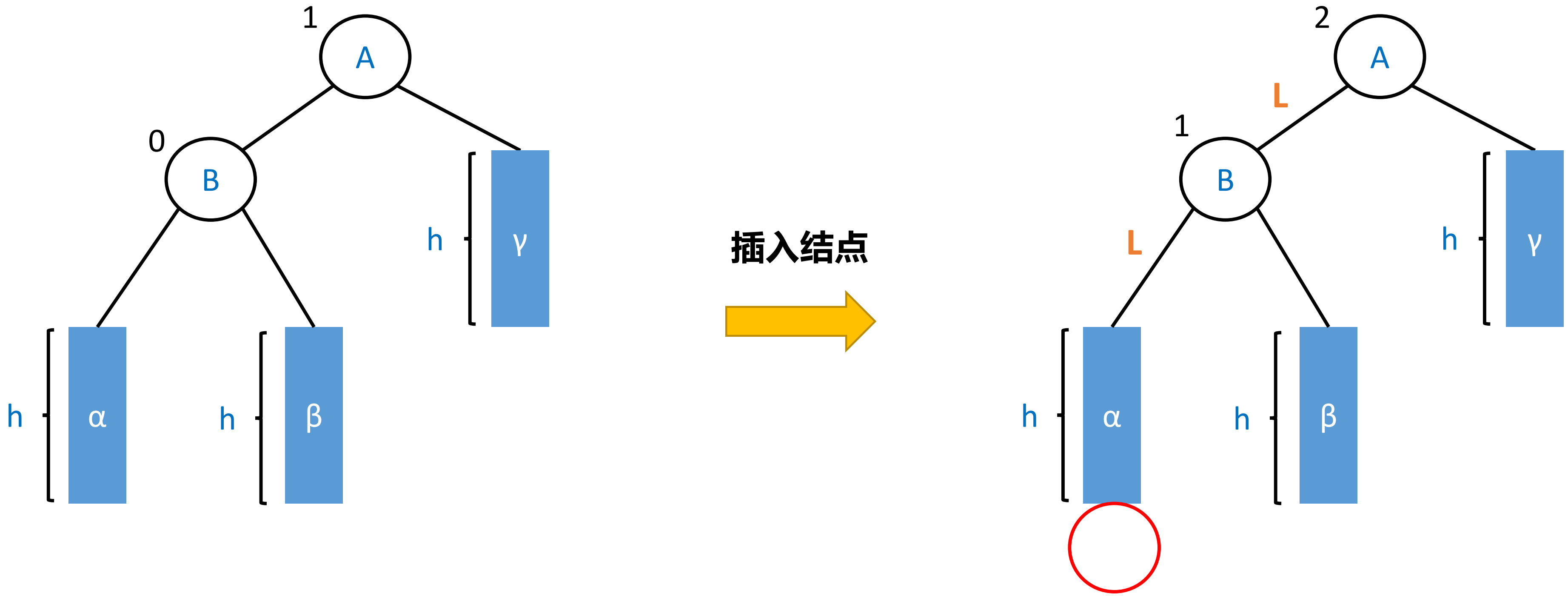
杭州电子科技大学 HANGZHOU DIANZI UNIVERSITY

平衡调整的四种类型：

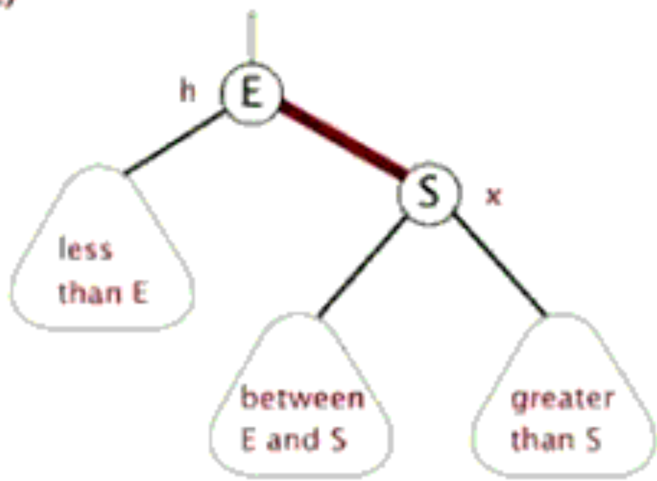


调整原则： 1）降低高度 2） 保持二叉排序树性质

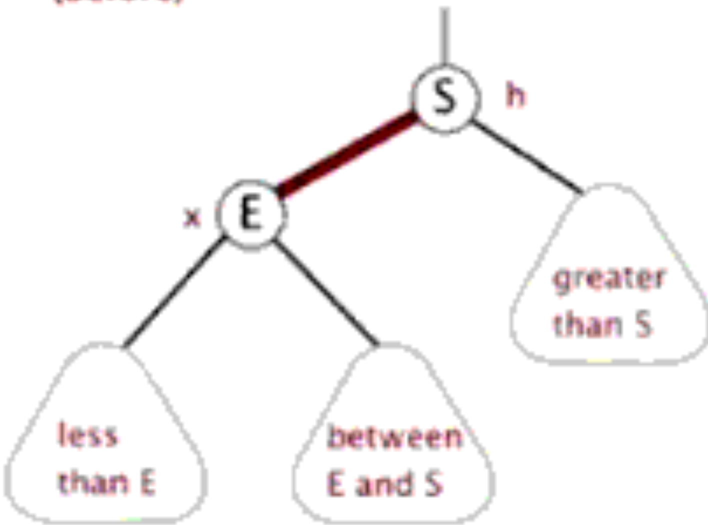
（1） LL型调整



旋转

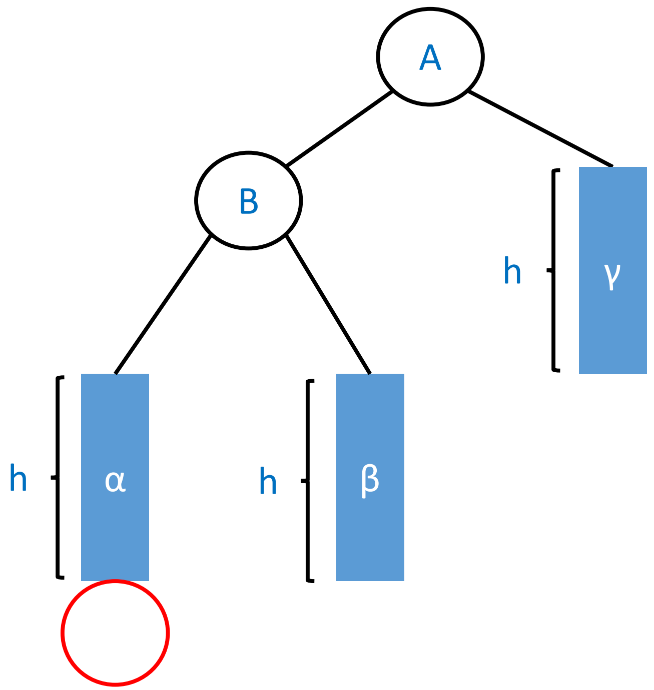

rotate E left
(before)
h
E
S
x
less
than E
between
E and S
greater
than S

左旋


rotate S right
(before)
S
h
x
E
greater
than S
less
than E
between
E and S

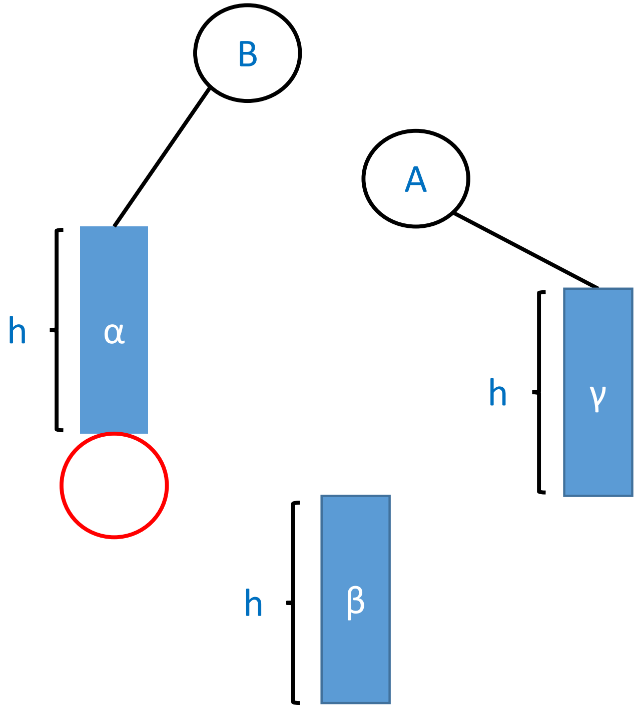
右旋

LL型调整过程



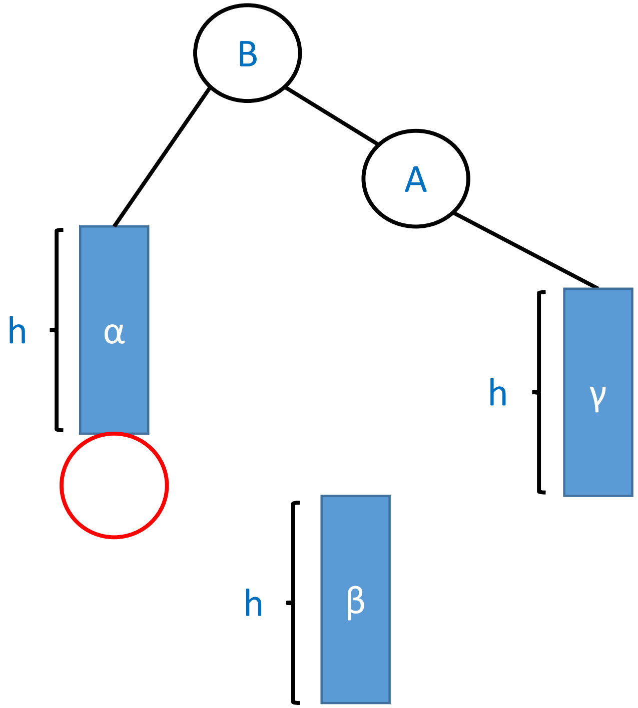
- B结点带左子树α一起上升

LL型调整过程



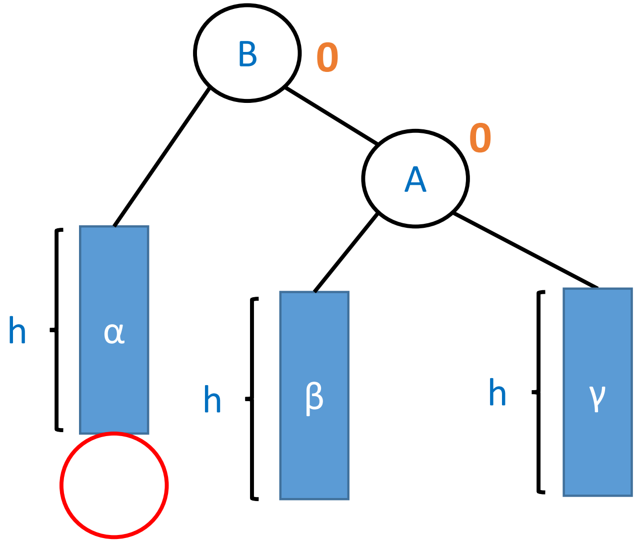
- B结点带左子树α一起上升

LL型调整过程



- B结点带左子树α一起上升
- A结点成为B的右孩子
- 原来B结点的右子树β作为A的左子树

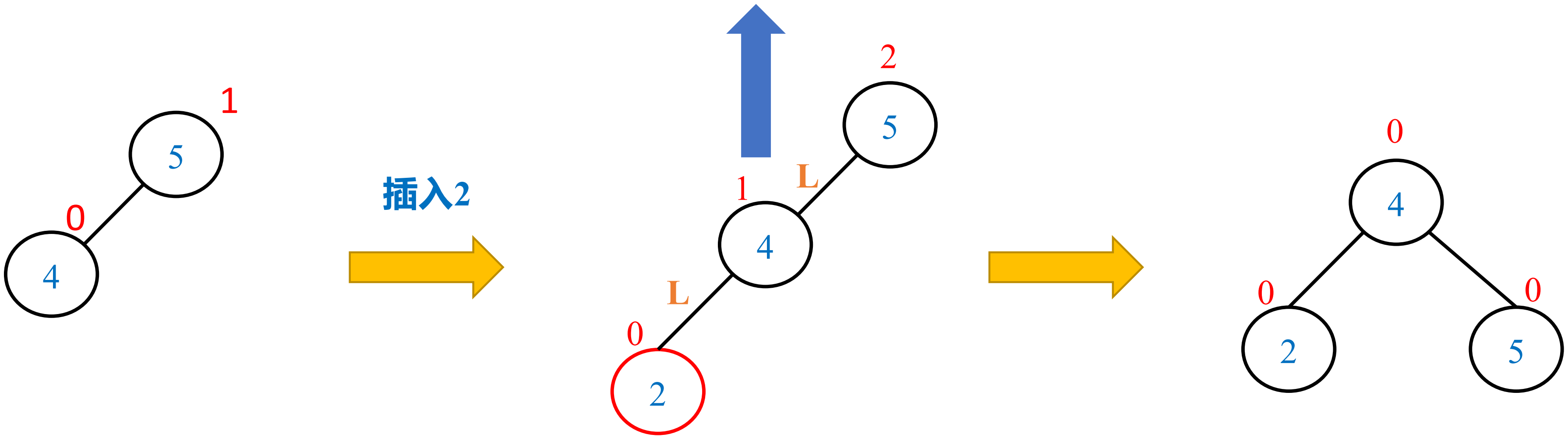
LL型调整过程



LL调整后的结果

- B结点带左子树α一起上升
- A结点成为B的右孩子
- 原来B结点的右子树β作为A的左子树

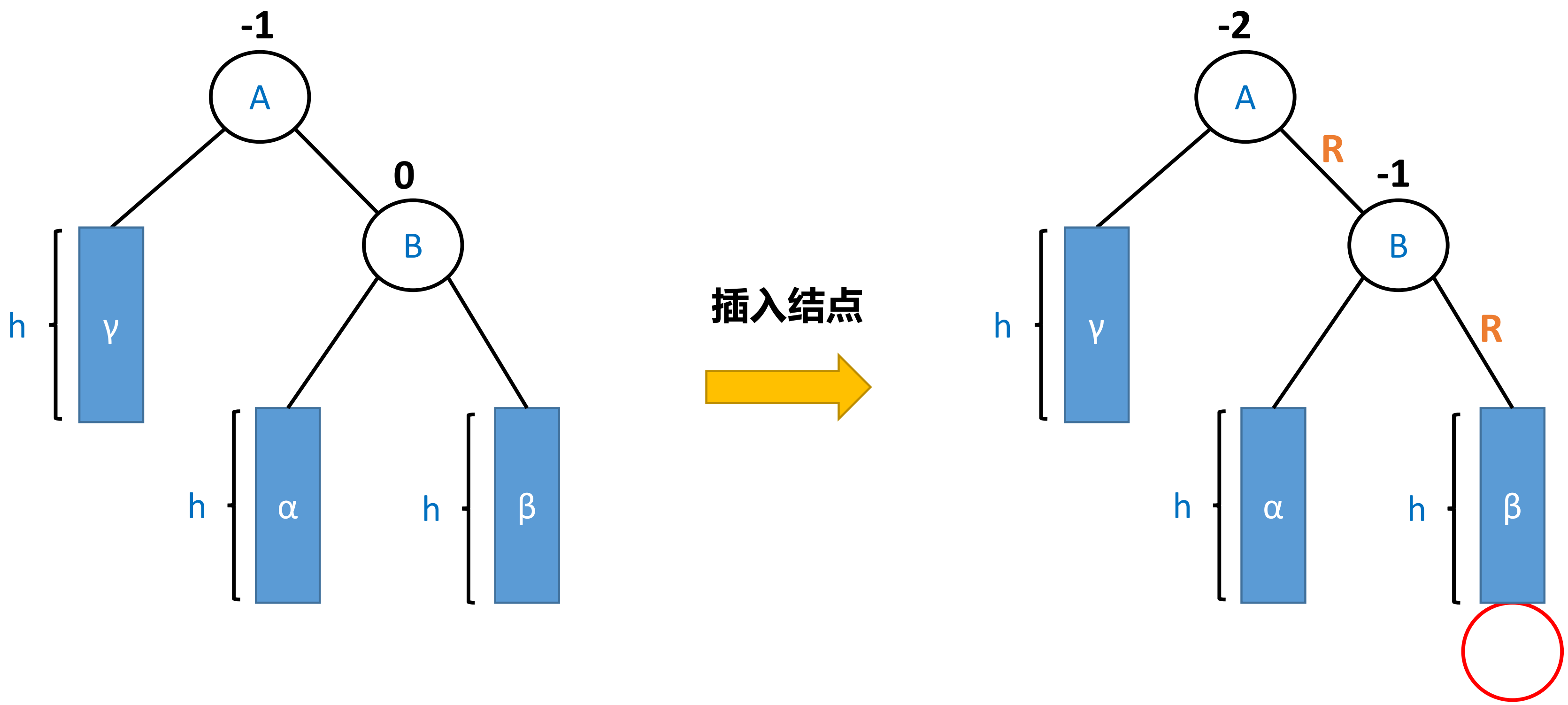
AVL树LL调整--例



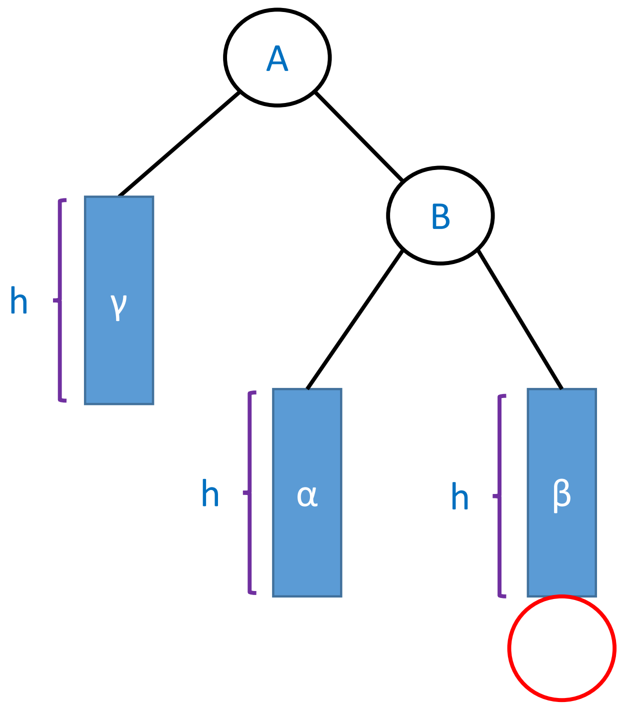
插入关键字2的结果

调整完毕

(2) RR型调整

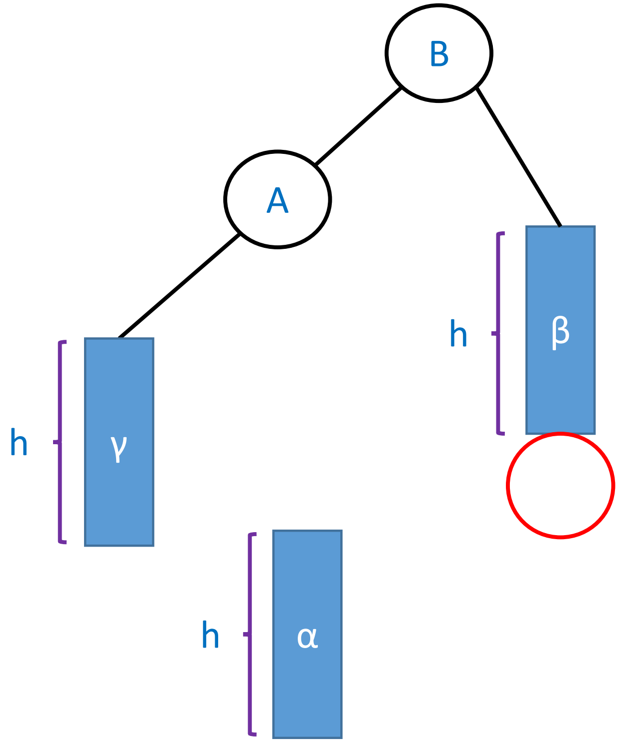


RR型调整过程



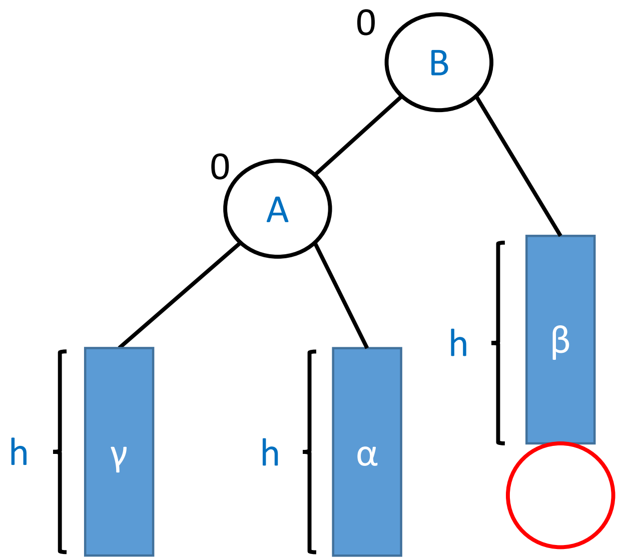
- B结点带右子树β一起上升

RR型调整过程



- B结点带右子树β一起上升
- A结点成为B的左孩子

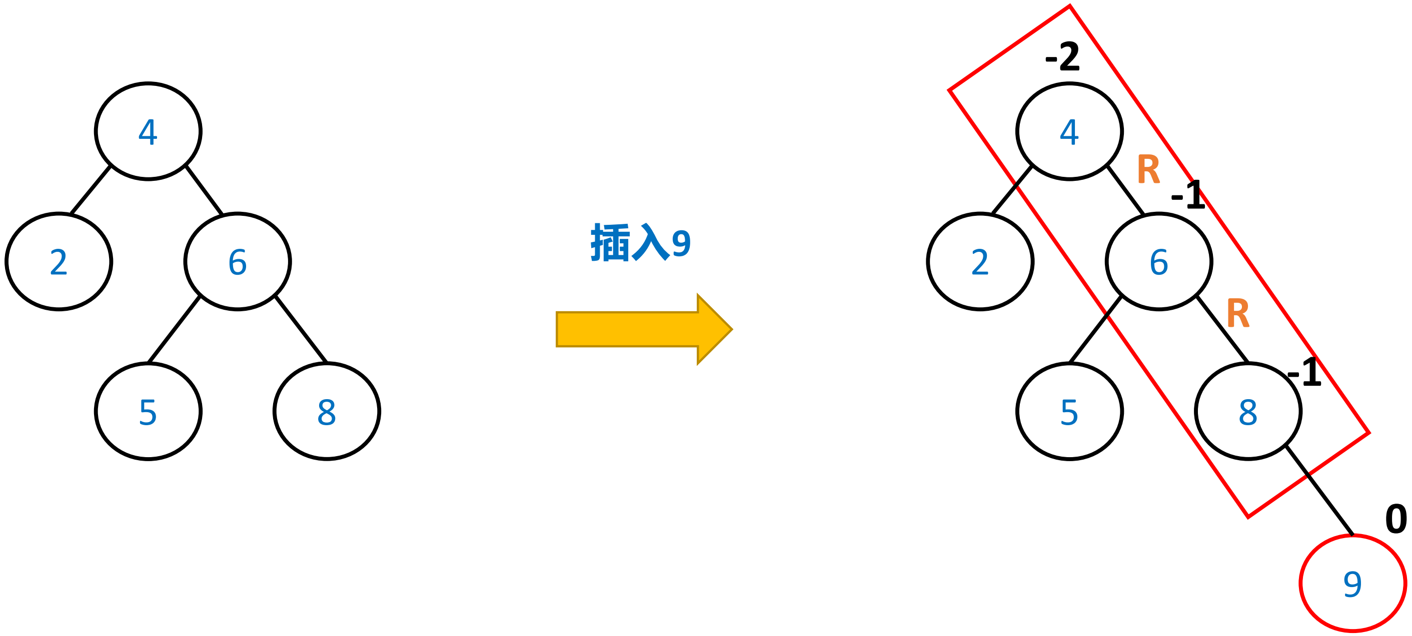
RR型调整过程



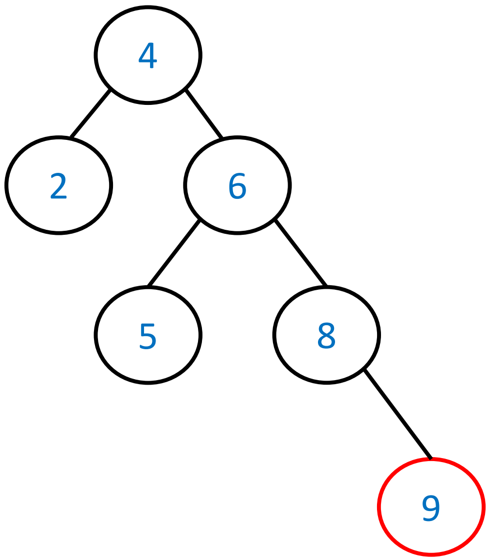
RR调整后的结果

- B结点带右子树β一起上升
- A结点成为B的左孩子
- 原来B结点的左子树α作为A的右子树

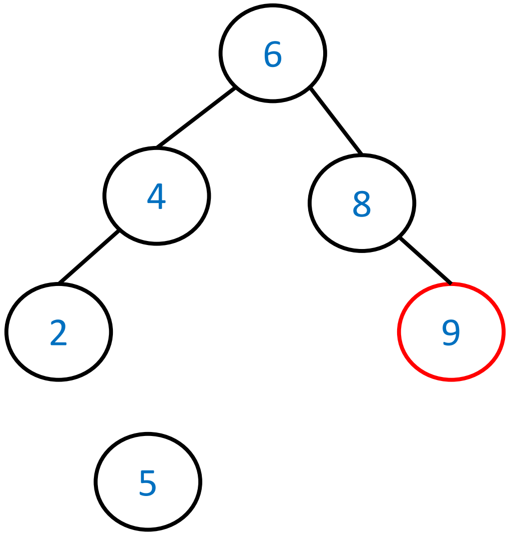
AVL树RR调整--例：



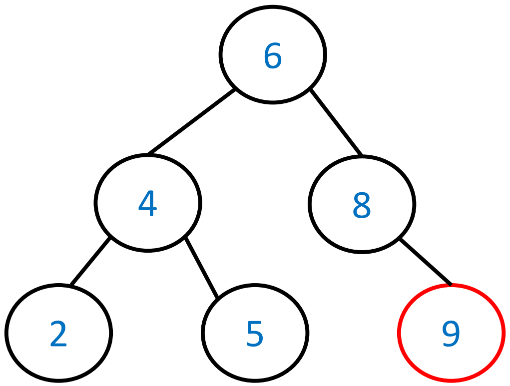
AVL树RR调整--例



AVL树RR调整--例

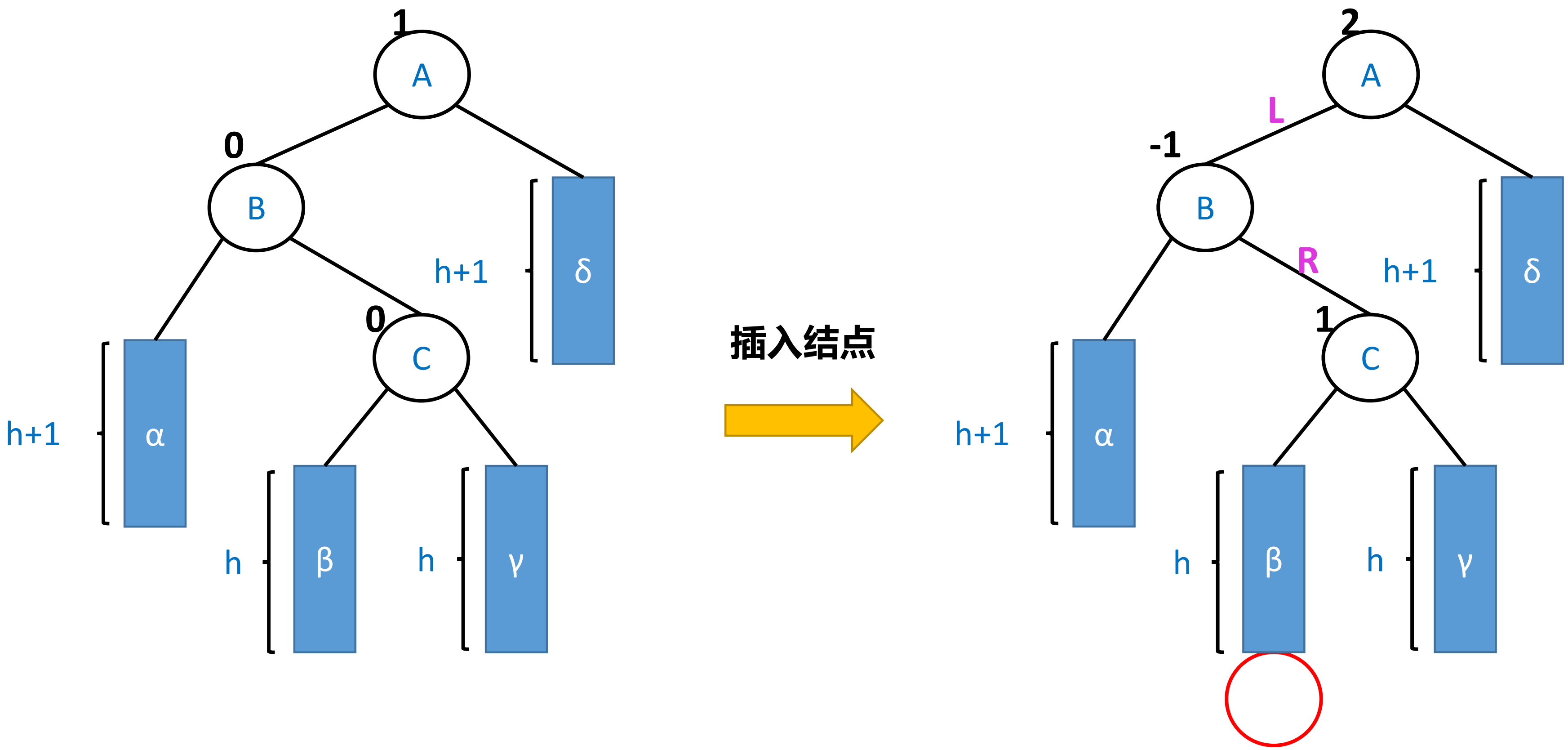


AVL树RR调整--例

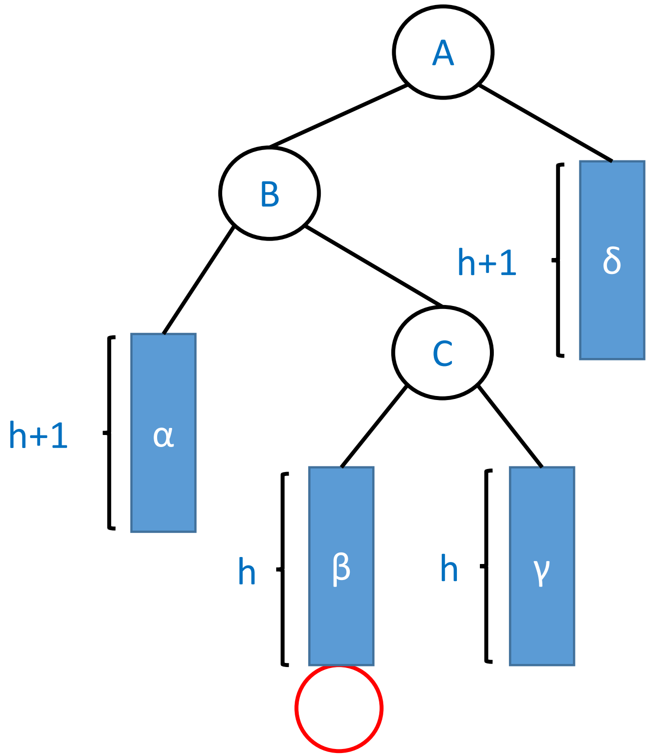


调整完毕

(3) LR型调整

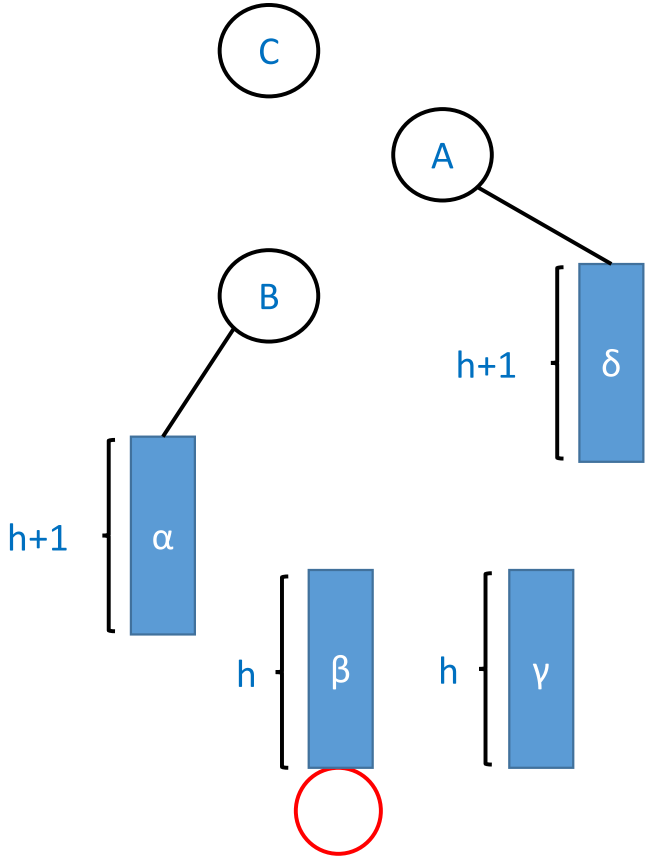


LR型调整过程



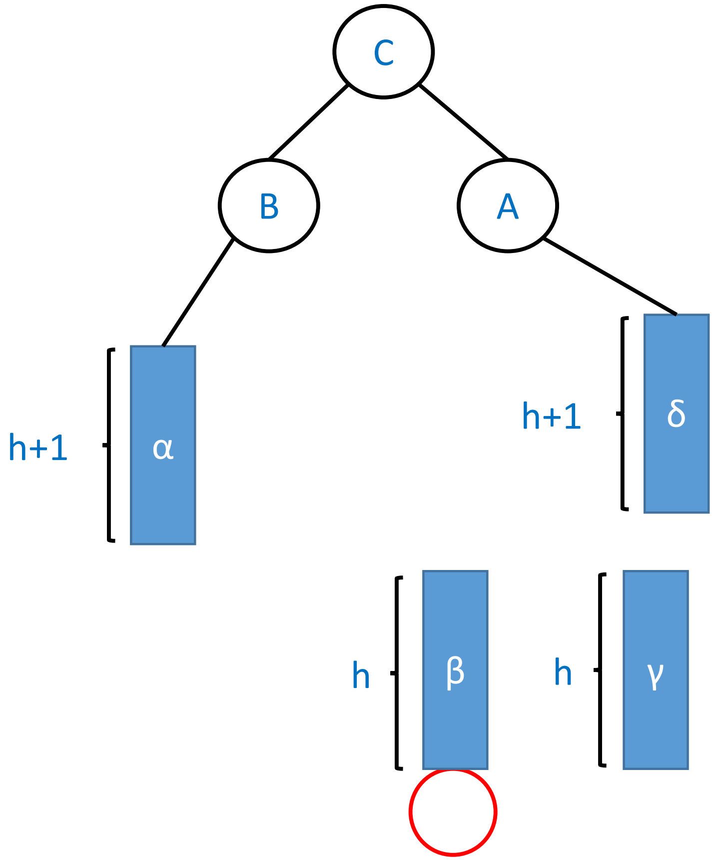
- C结点穿过A、B结点上升

LR型调整过程



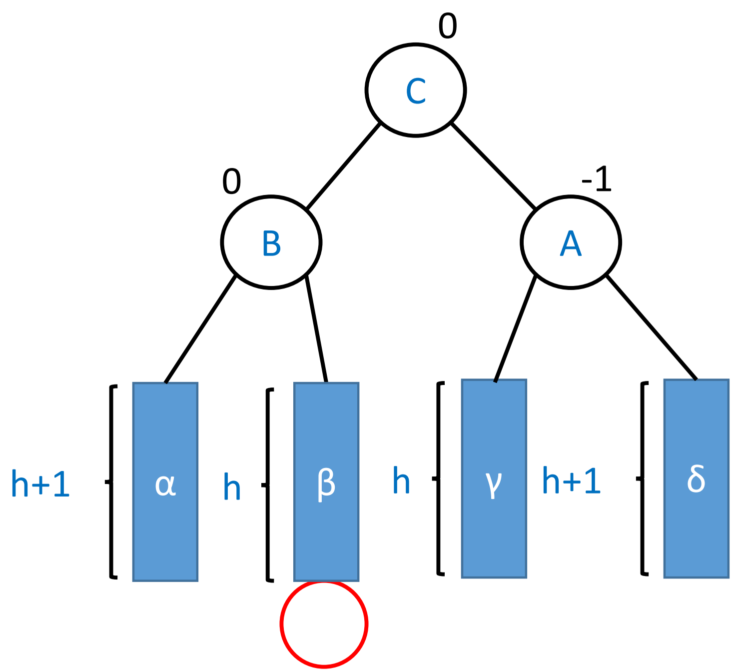
- C结点穿过A、B结点上升

LR型调整过程



- C结点穿过A、B结点上升
- B结点成为C的左孩子，A结点成为C的右孩子

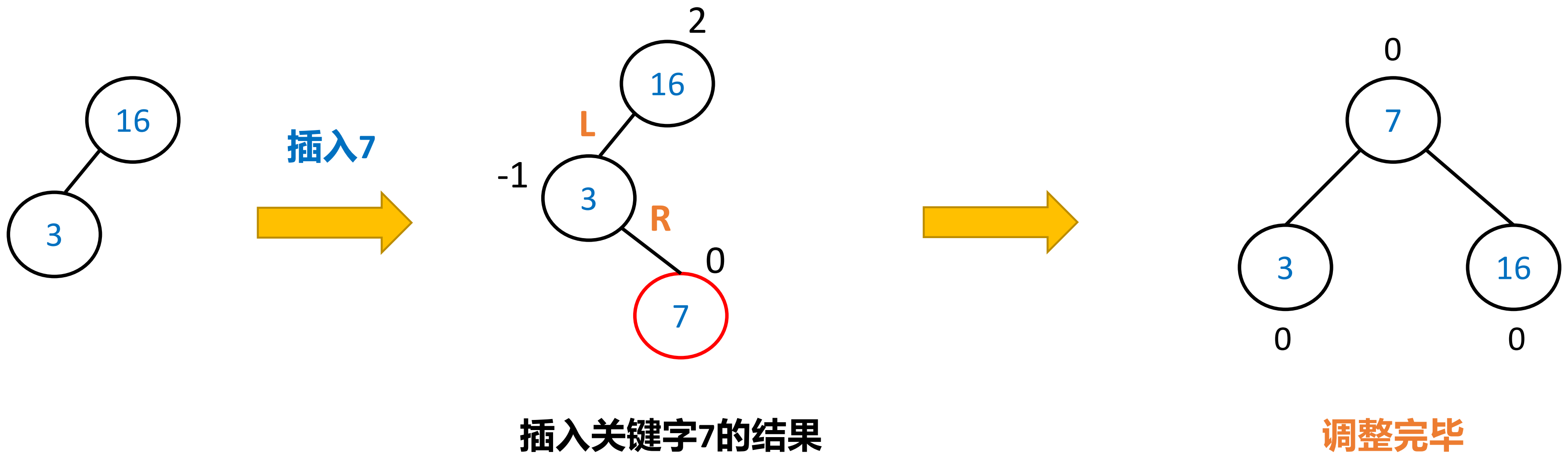
LR型调整过程



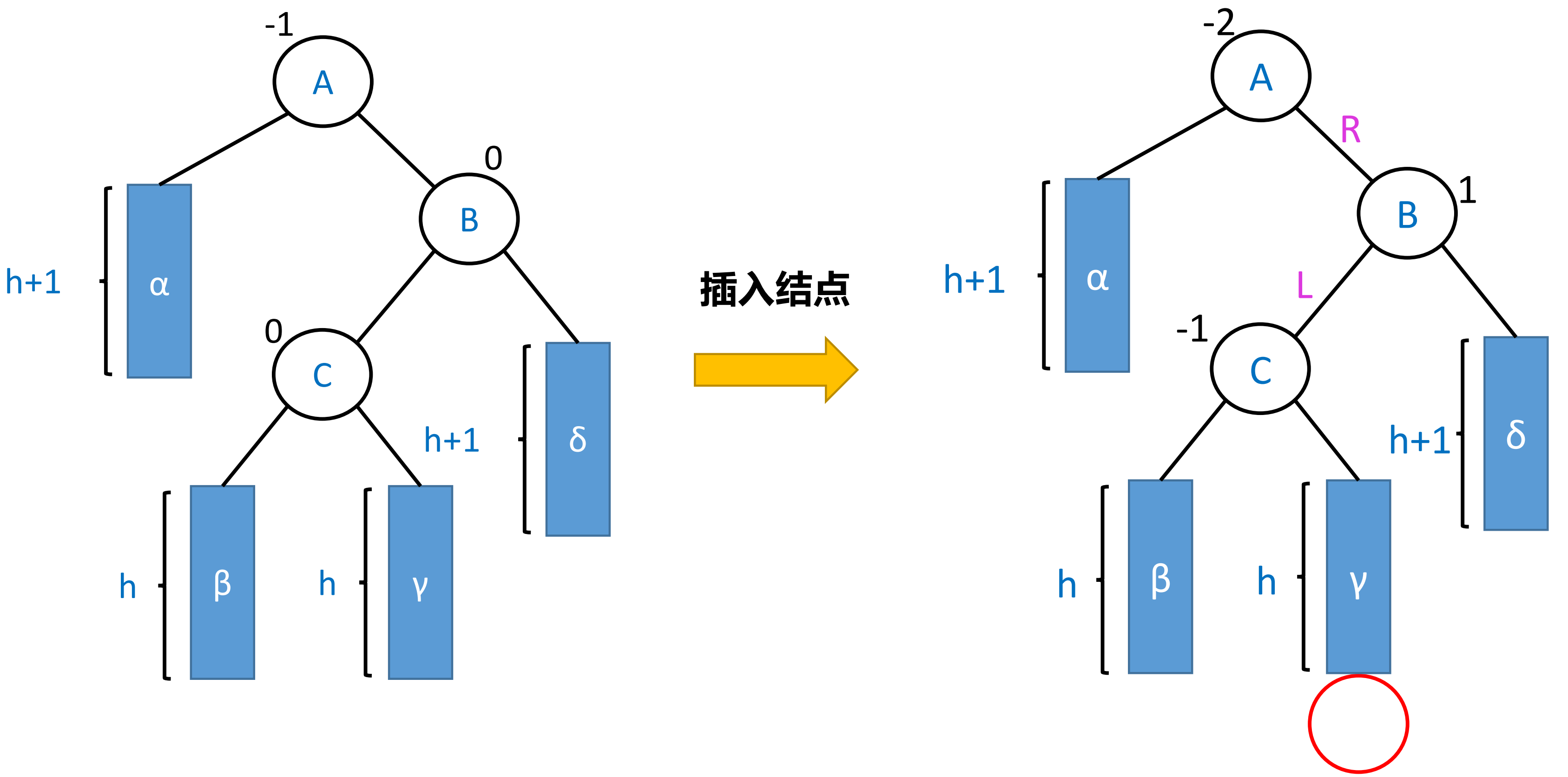
LR调整后的结果

- C结点穿过A、B结点上升
- B结点成为C的左孩子，
  A结点成为C的右孩子
- 原来C结点的左子树β作为B的右子树；
  原来C结点的右子树γ作为A的左子树

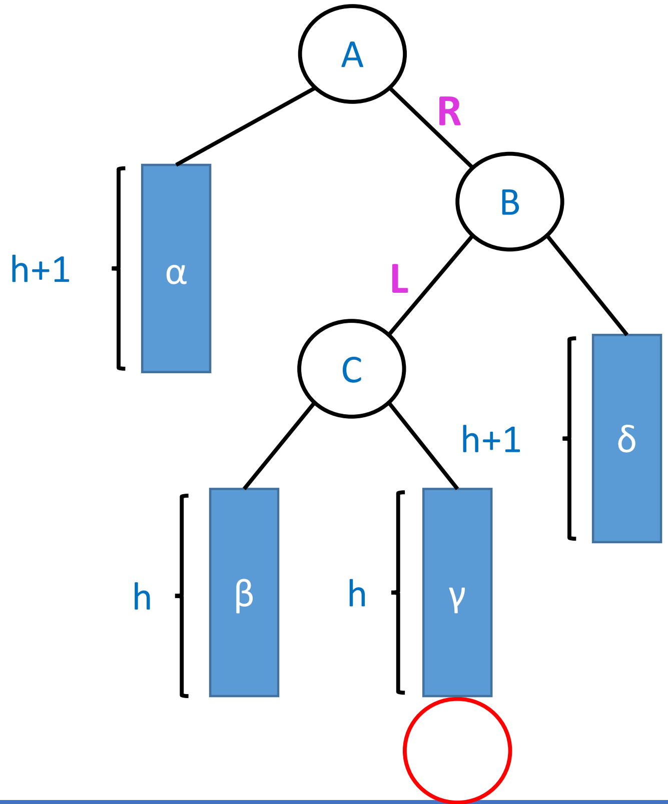
AVL树LR调整--例



（4） RL型调整

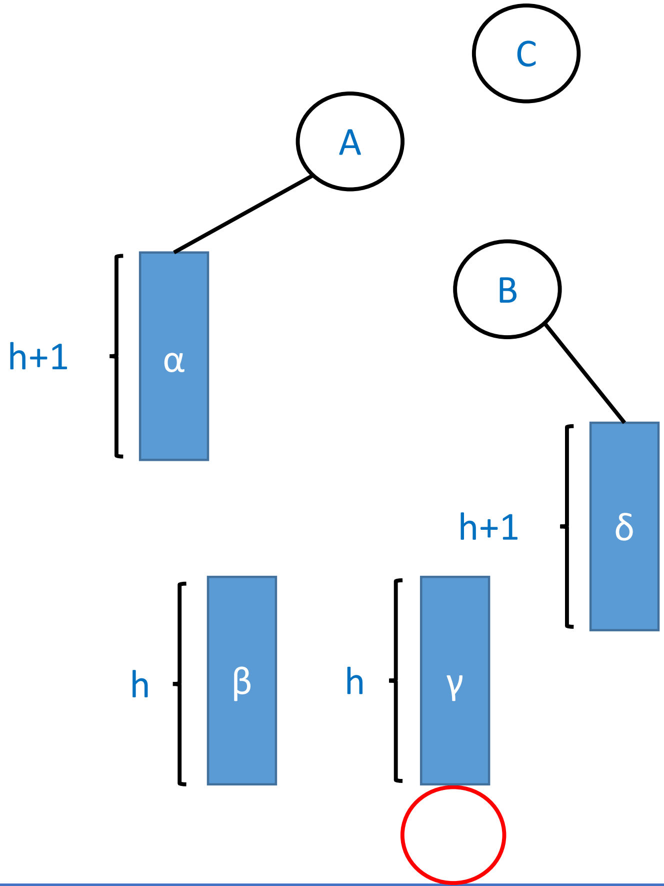


RL型调整过程



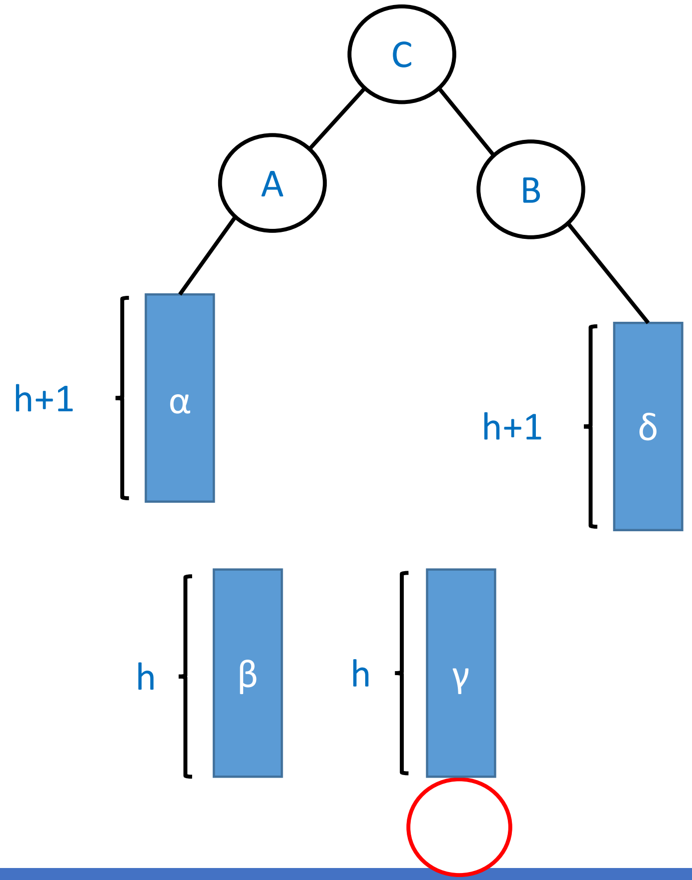
- C结点穿过A、B结点上升

RL型调整过程



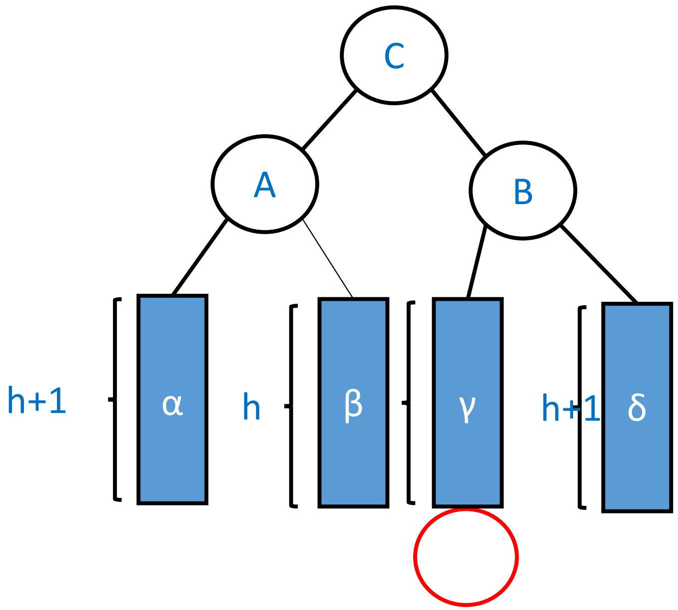
- C结点穿过A、B结点上升

RL型调整过程



- C结点穿过A、B结点上升
- A结点成为C的左孩子，
  B结点成为C的右孩子

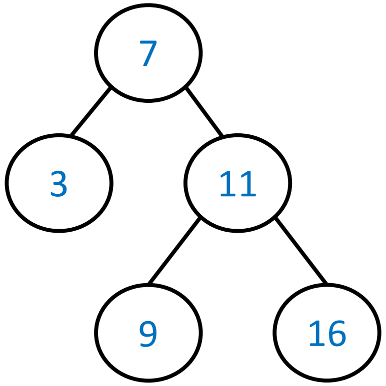
RL型调整过程



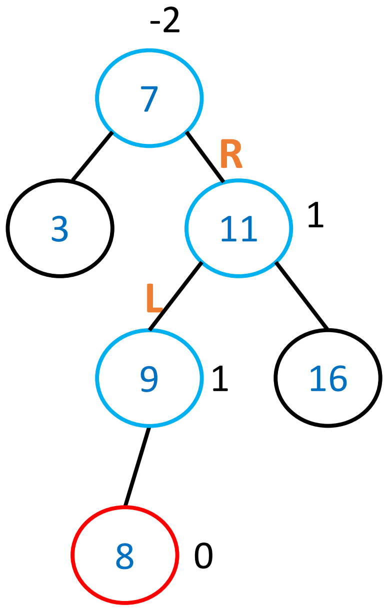
LR调整后的结果

- C结点穿过A、B结点上升
- A结点成为C的左孩子，
  B结点成为C的右孩子
- 原来C结点的左子树β作为A的右子树；
  原来C结点的右子树γ作为B的左子树

AVL树RL调整--例

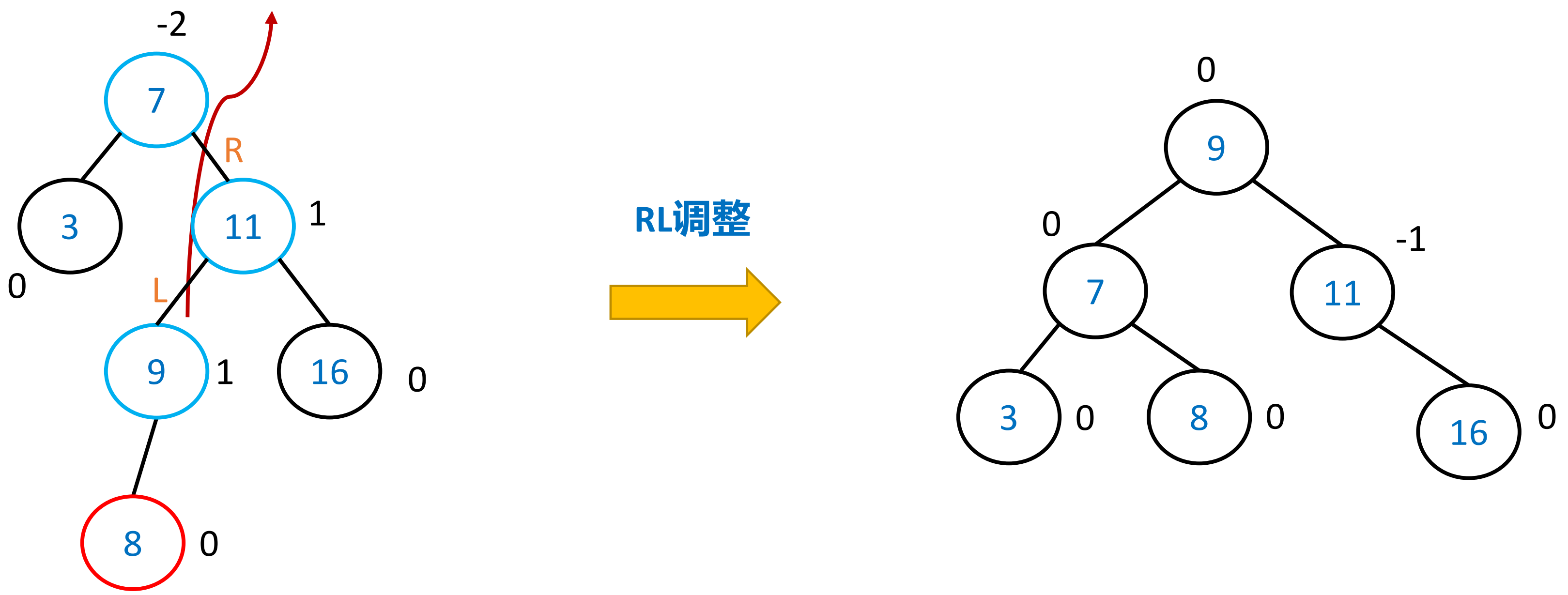


插入8



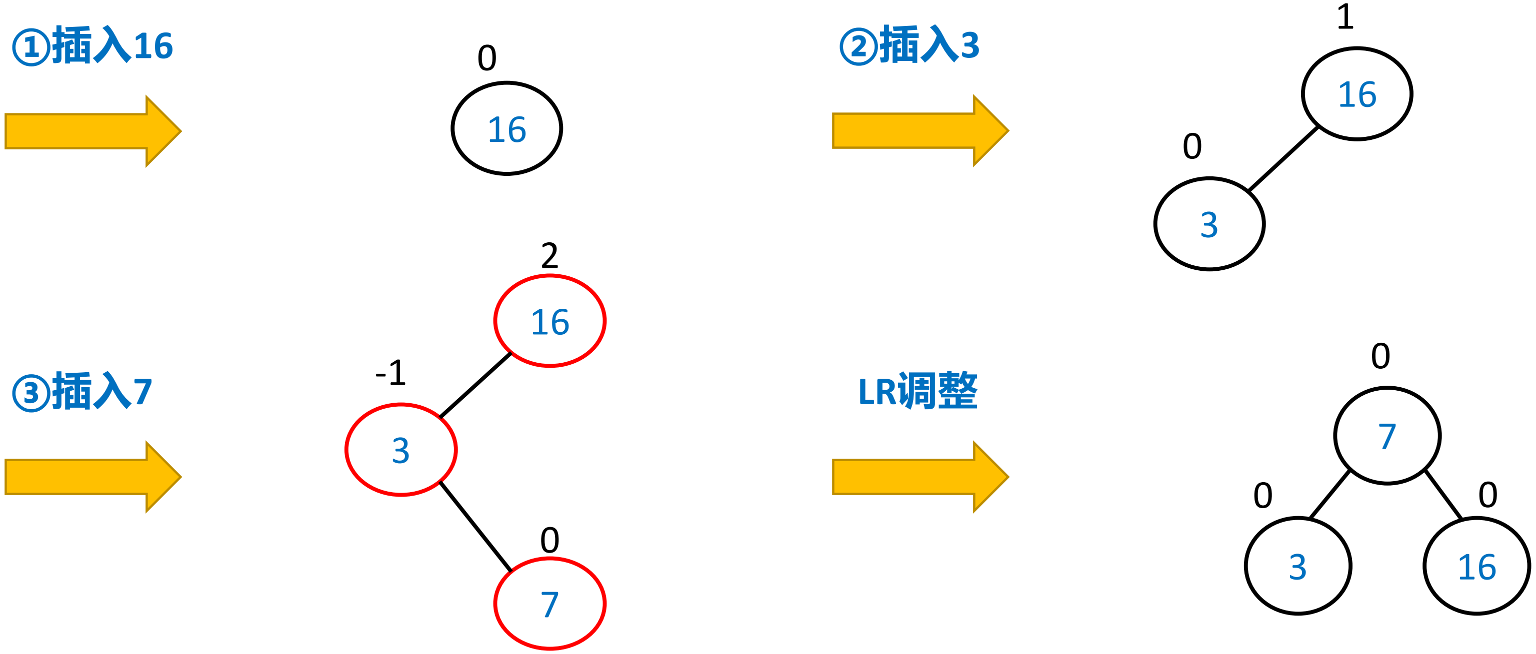
插入关键字8的结果

AVL树RL调整--例



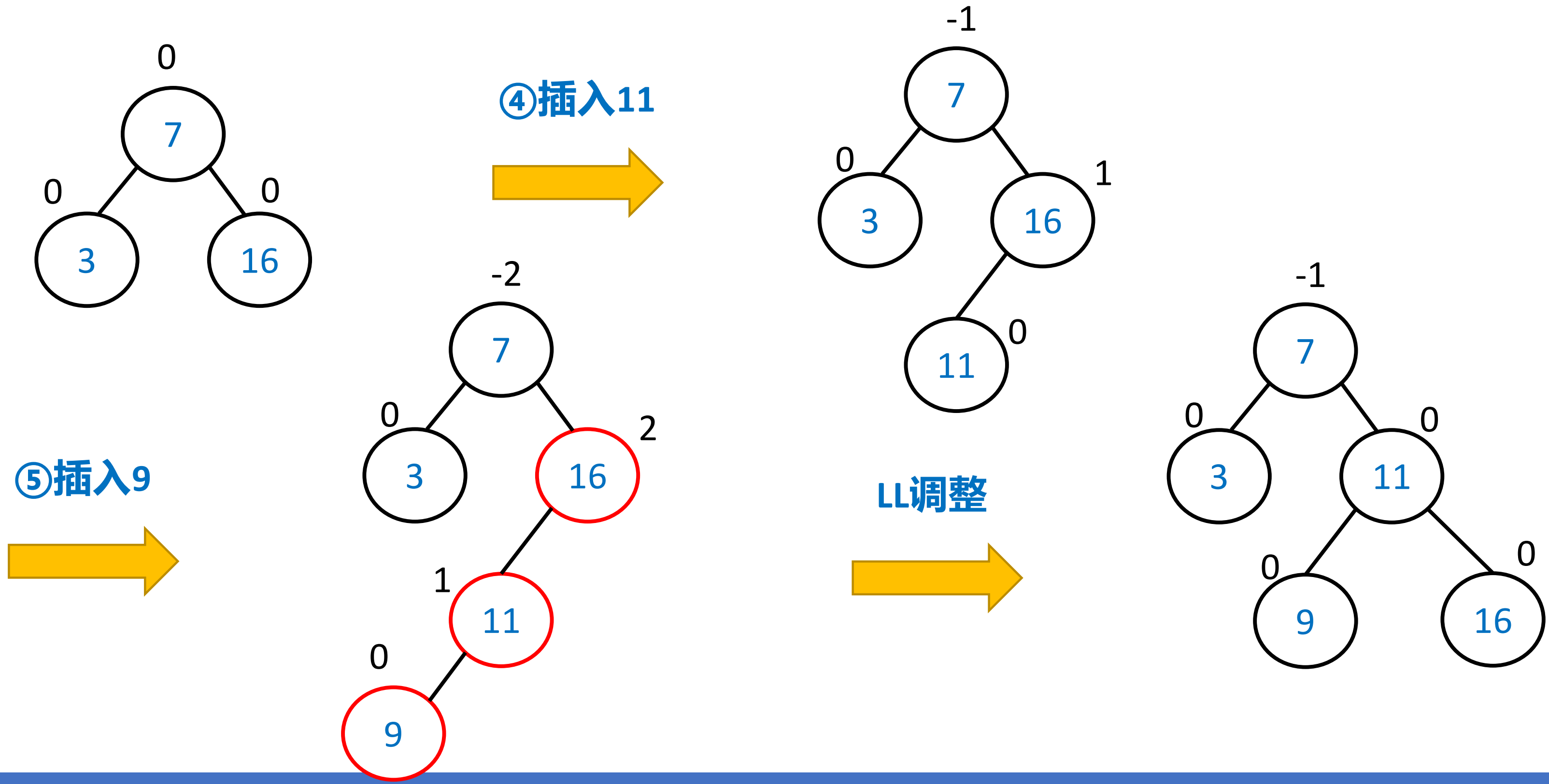
例题

输入关键字序列(16, 3, 7, 11, 9, 26, 18, 14, 15), 给出构造一棵AVL树的步骤。



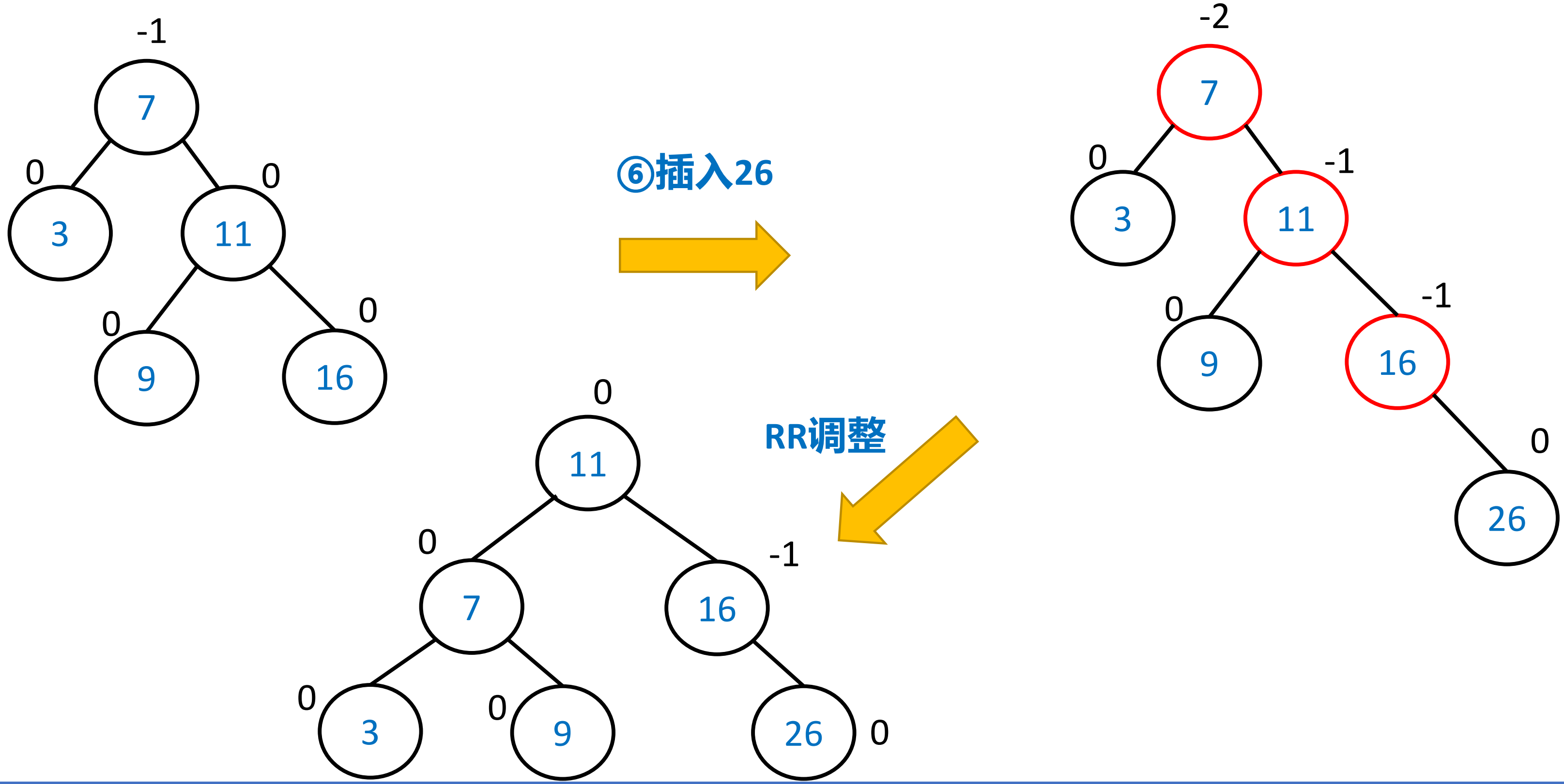
例题

输入关键字序列(16, 3, 7, 11, 9, 26, 18, 14, 15), 给出构造一棵AVL树的步骤。



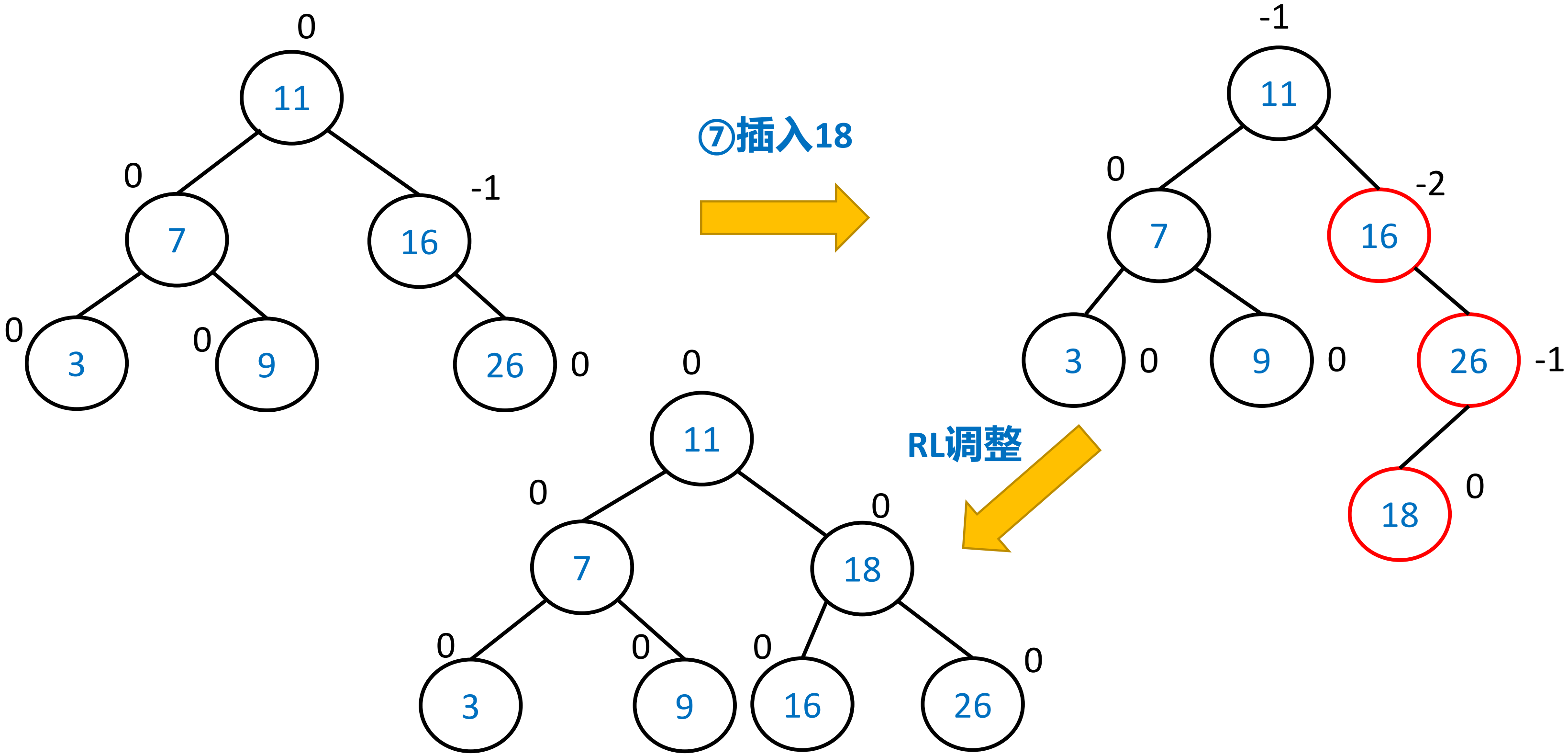
例题

输入关键字序列(16, 3, 7, 11, 9, 26, 18, 14, 15), 给出构造一棵AVL树的步骤。



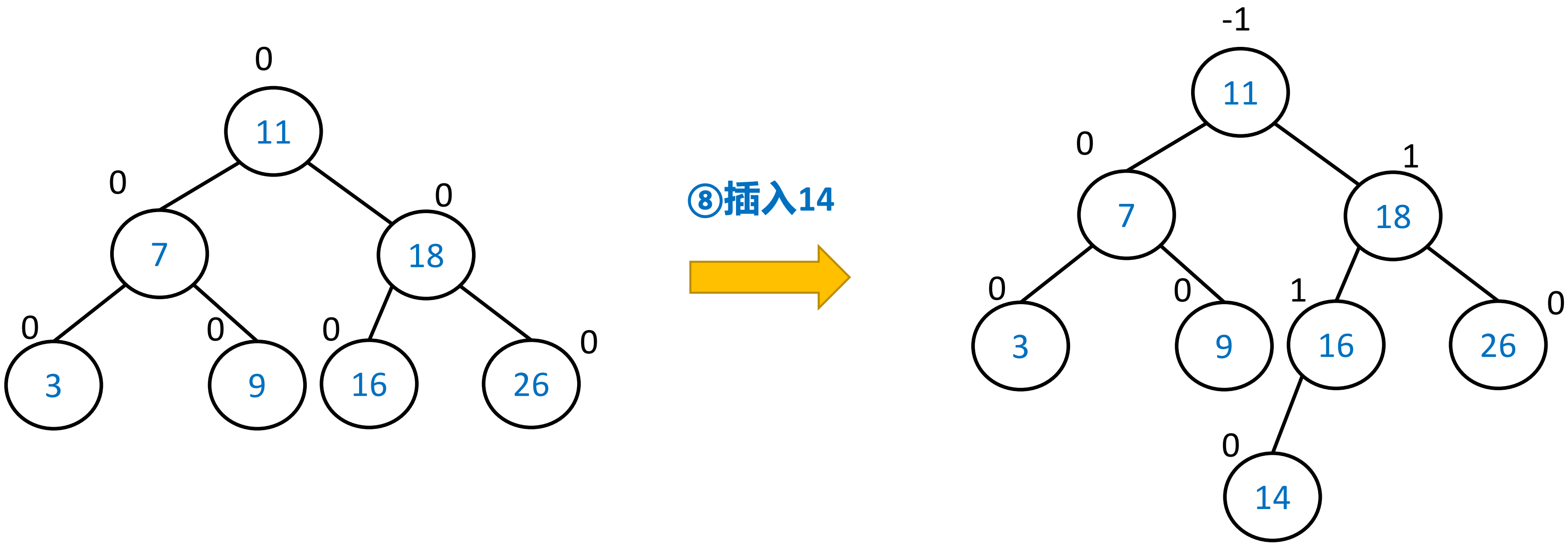
例题

输入关键字序列(16, 3, 7, 11, 9, 26, 18, 14, 15), 给出构造一棵AVL树的步骤。

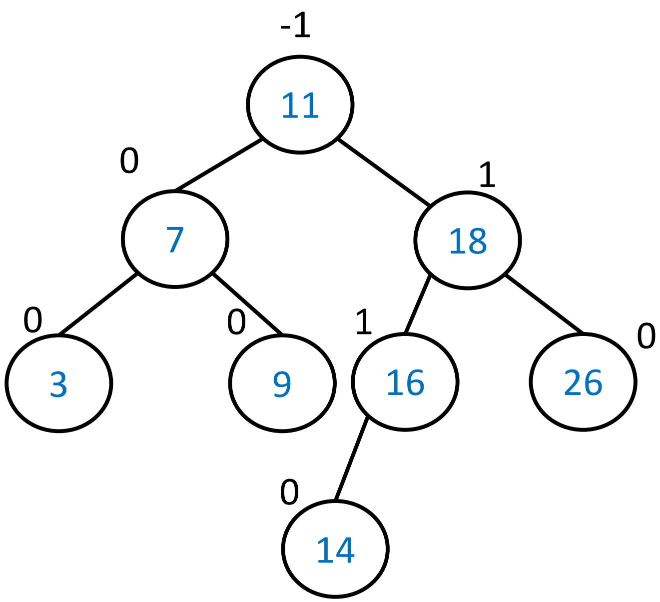


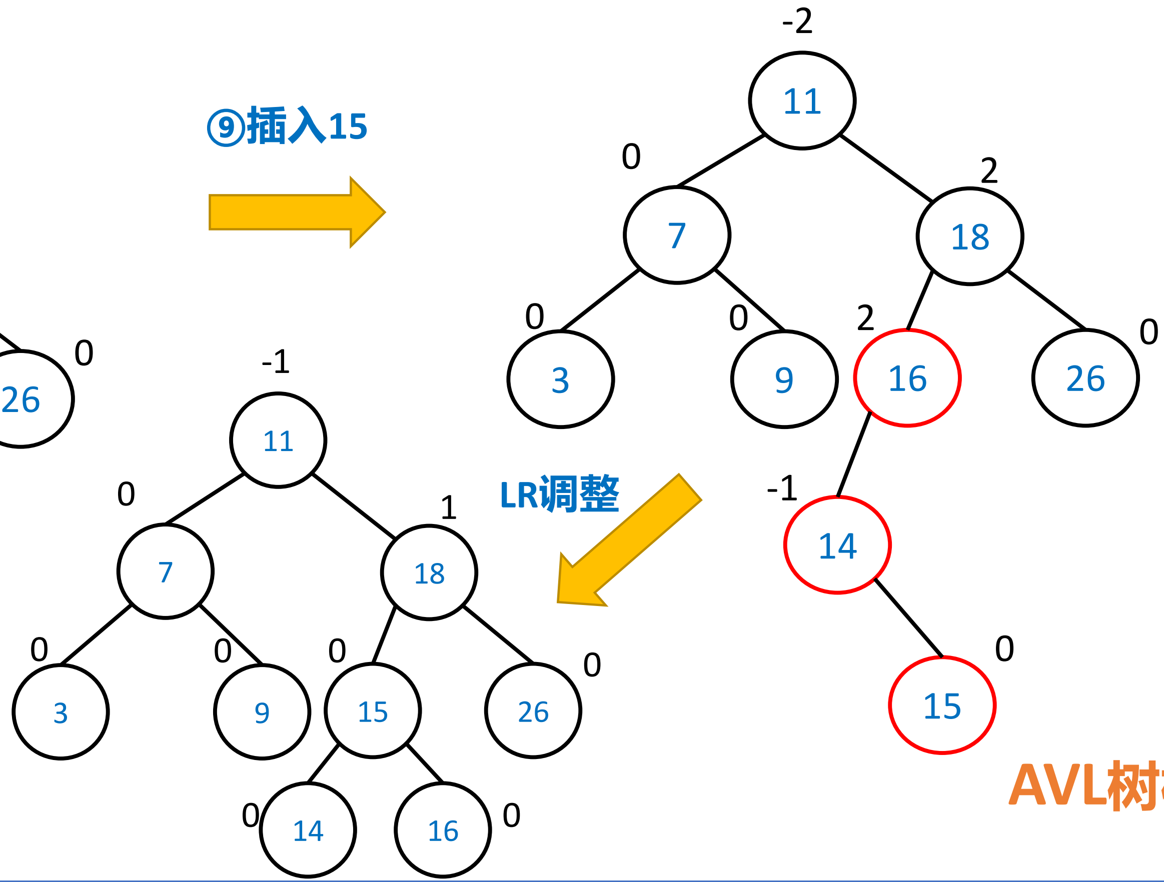
例题

输入关键字序列(16, 3, 7, 11, 9, 26, 18, 14, 15), 给出构造一棵AVL树的步骤。



例题





AVL树构造完毕

查找时间复杂度:
O(log*n*)

红黑树

杭州電子科技大学 HANGZHOU DIANZI UNIVERSITY

问题：为什么提出红黑树？

平衡二叉树 —高效→ 查找、插入和删除在平均和最坏情况下都是O(logn)

缺点：

条件严苛

插入或删除结点时需要维持平衡状态，需要对其进行旋转，旋转的次数不能预知。

→ 放弃追求完全平衡，追求大致平衡 → 红黑树 O(logn)

红黑树（Red Black Tree, RBTree） 是一种自平衡二叉排序树，与AVL树不同的是，红黑树是弱平衡二叉树，即它的左右子树高度差有可能大于1，但不超过一倍。

红黑树

红黑树同时满足以下特性：

① 结点是**红色**或者**黑色**

② 根结点是**黑色**

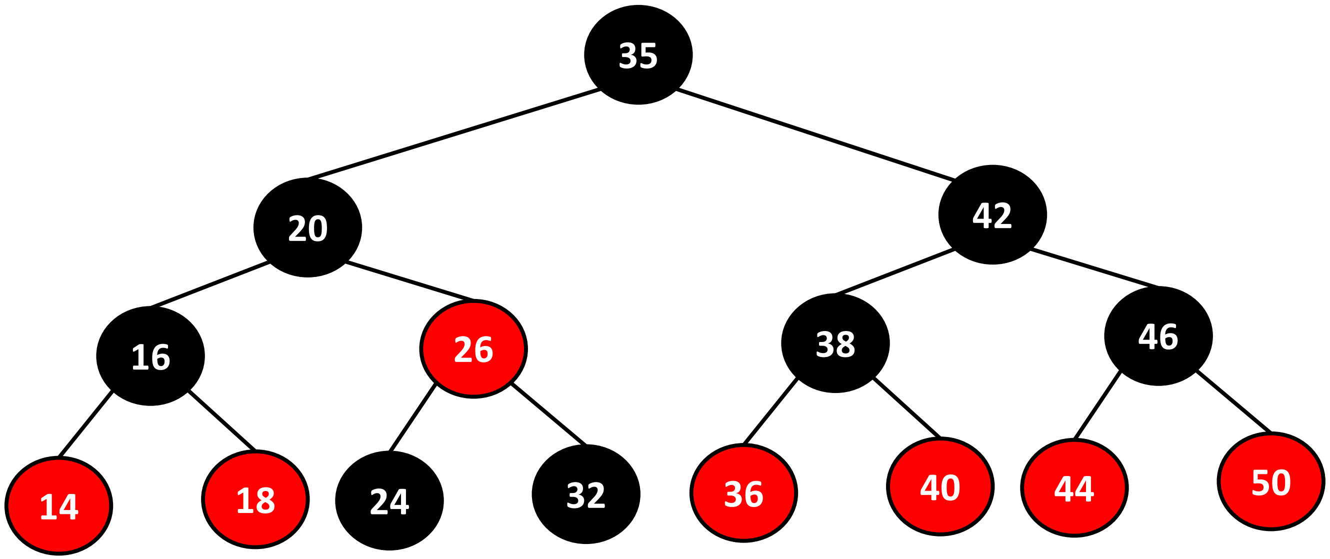
③ 叶子结点（外部结点，空结点）都是**黑色**，这里的叶子结点指的是最底层的空结点（外部结点）

④ **红色**结点的孩子结点都是**黑色**

- **红色**结点的双亲结点都是**黑色**
- 从根结点到叶子结点的所有路径上不能有 2 个连续的**红色**结点

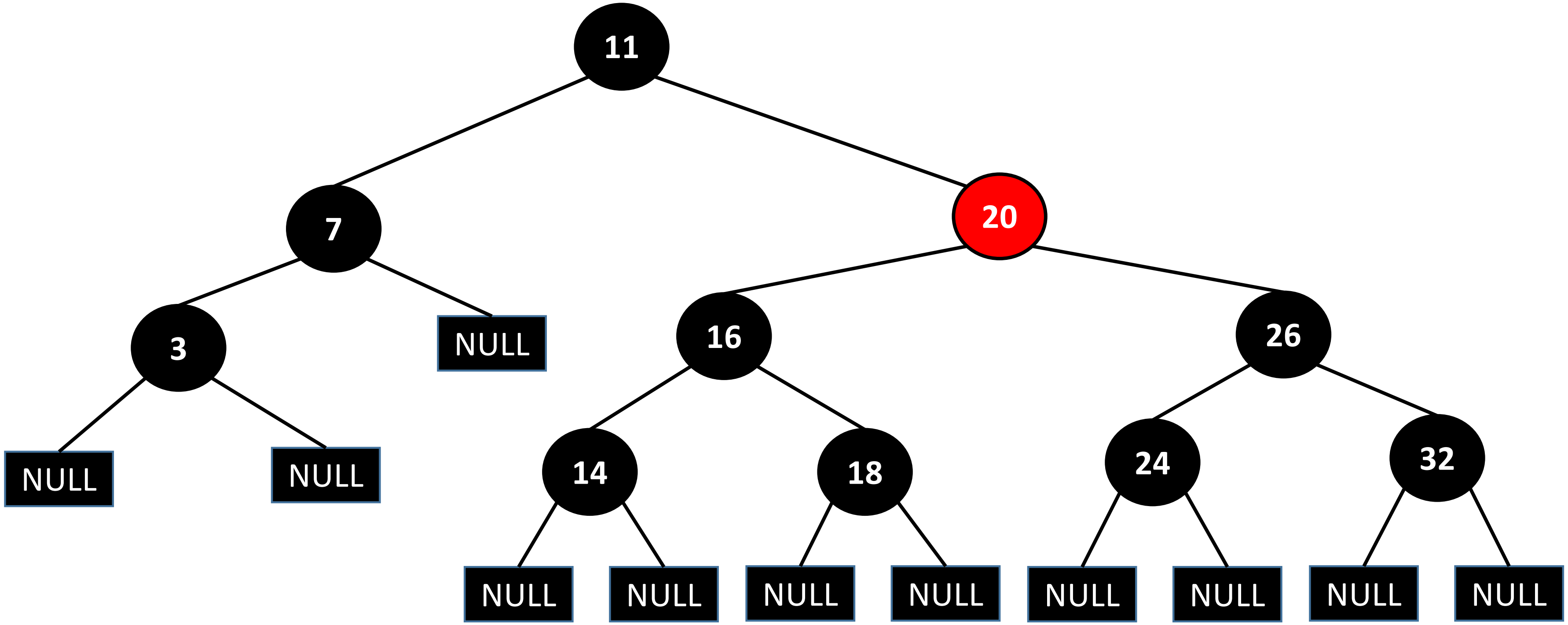
⑤ 从任一结点到叶子结点的所有路径都包含相同数目的**黑色**结点 ★

红黑树例子



黑色结点可以同时包含一个红色子结点和一个黑色子结点

红黑树例子


11
7
20
3
NULL
16
26
NULL
NULL
14
18
24
32
NULL
NULL
NULL
NULL
NULL
NULL
NULL
NULL

不是红黑树！

红黑树的插入

- 红黑树的插入包含两个步骤：

① 在树中查找插入的位置

② 插入后自平衡 → （操作1：变色；操作2：旋转）

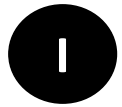
注意：插入节点应该是红色。

原因：根据红黑树特性5，红色在双亲节点（如果存在）是黑色节点时，红黑树的黑色平衡不会被破坏，所以不需要进行自平衡操作。但如果插入节点是黑色，那么插入位置所在的子树黑色节点总是多1，必须做自平衡操作。

红黑树的插入

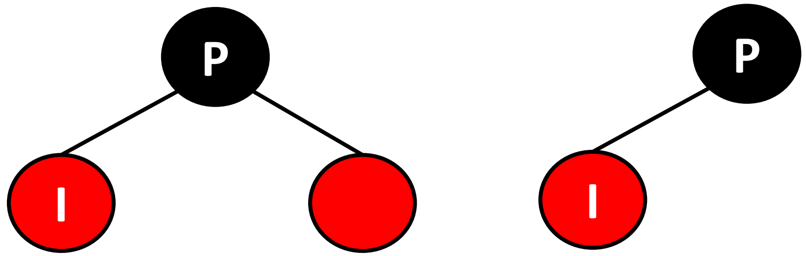
（1）红黑树为空树

步骤：① 把插入结点作为根节点； ② 把结点设置为黑色



（2）插入结点的双亲结点为黑色

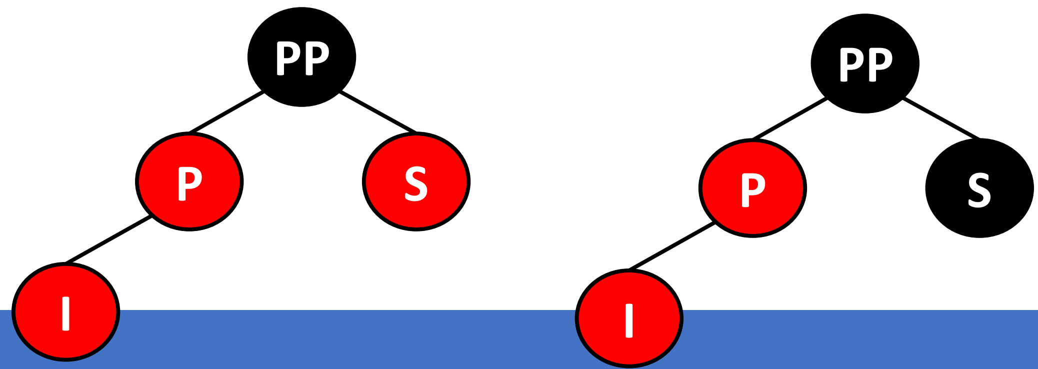
步骤：无需进行平衡操作，直接插入新结点



（3）插入结点的双亲结点为红色

A. 叔叔结点存在并且为红结点

B. 叔叔结点不存在或为黑结点

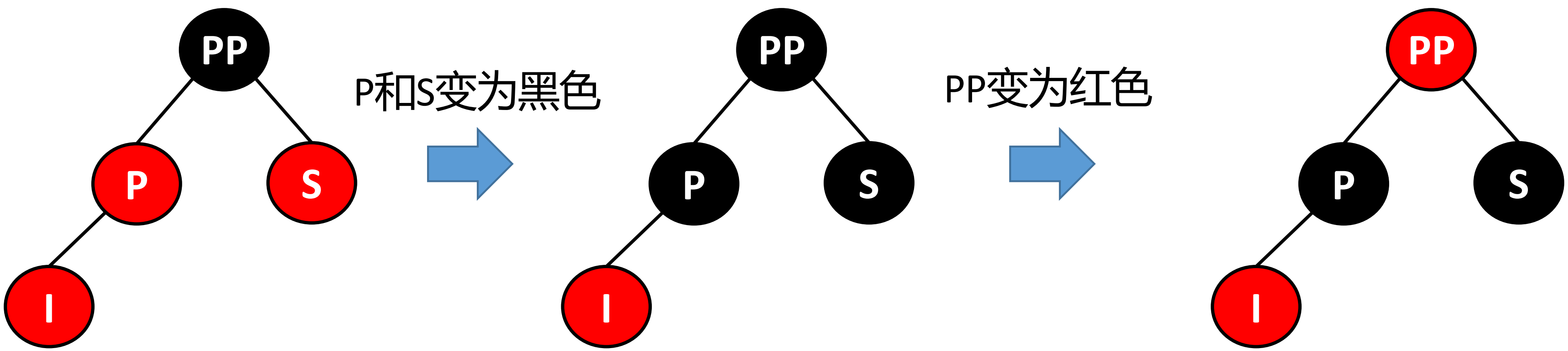


红黑树的插入

（3）插入结点的双亲结点为红色

操作：双亲、祖父、叔叔结点全变色；祖父变新结点，继续调整

A. 叔叔结点存在并且为红结点



若PP是根节点，则不变色。

若PP的双亲结点是黑色：满足红黑树特性；

若PP的双亲结点是红色：从PP结点出发，继续调整，直到红黑树平衡。

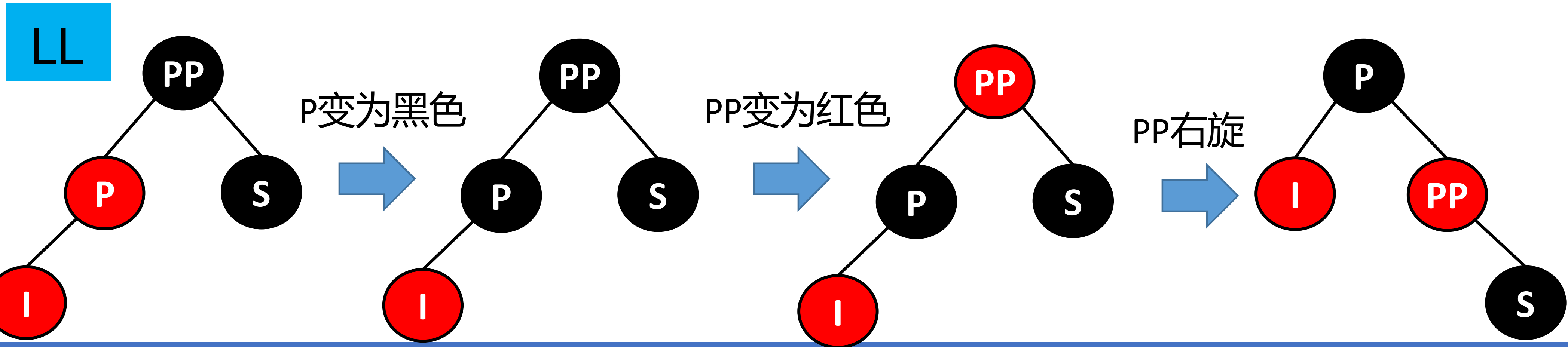
红黑树的插入

B. 叔叔结点不存在或为黑结点

B-1. 插入结点的双亲结点是祖父结点的左孩子

- B-1-1. 插入结点是双亲结点的左孩子(LL)
- B-1-2. 插入结点是双亲结点的右孩子(LR)

操作： P变黑色；PP变红色；对PP进行**右旋**



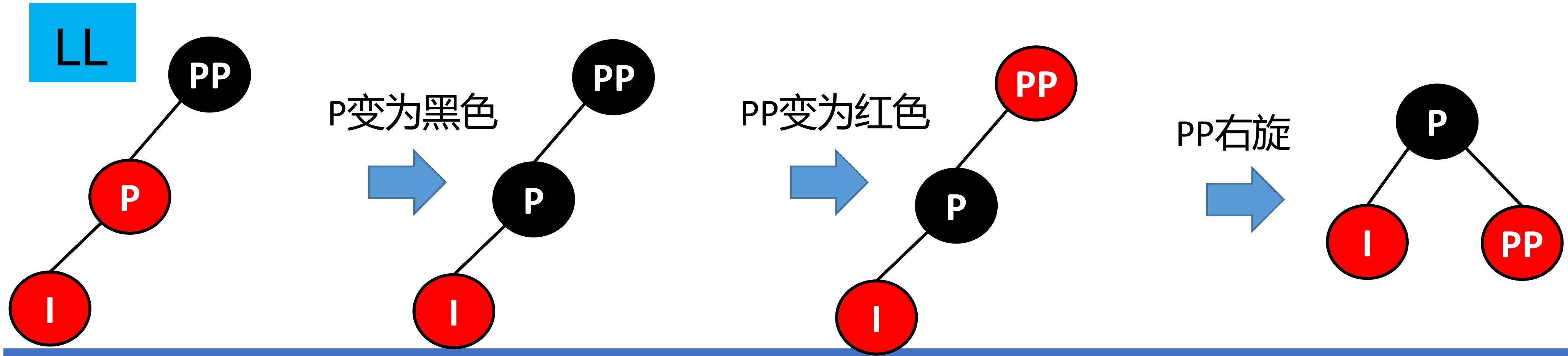
红黑树的插入

B. 叔叔结点不存在或为黑结点

B-1. 插入结点的双亲结点是祖父结点的左孩子

- B-1-1. 插入结点是双亲结点的左孩子(LL)
- B-1-2. 插入结点是双亲结点的右孩子(LR)

操作：P变黑色；PP变红色；对PP进行**右旋**



红黑树的插入

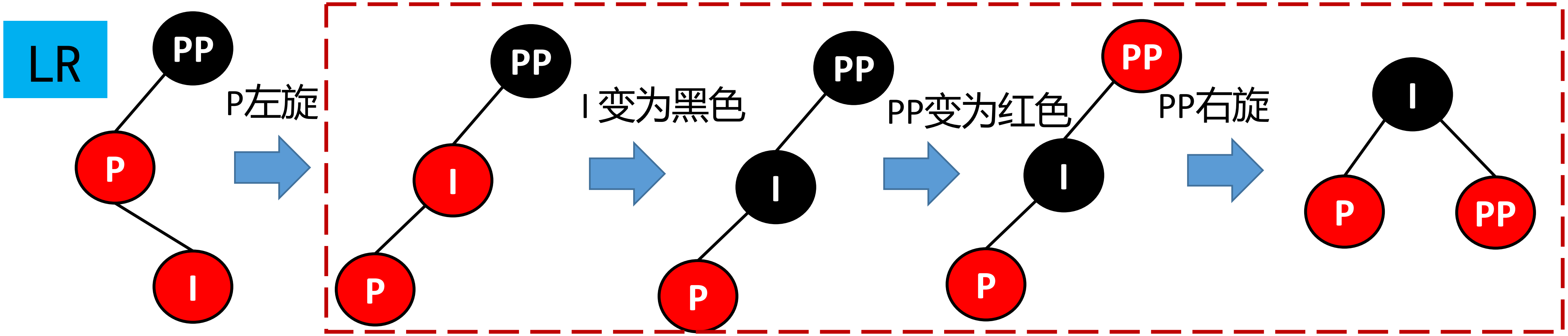
B. 叔叔结点不存在或为黑结点

B-1. 插入结点的双亲结点是祖父结点的左孩子

- B-1-1. 插入结点是双亲结点的左孩子(LL)
- B-1-2. 插入结点是双亲结点的右孩子(LR)

操作：对P结点进行左旋，变成LL

I变黑色；PP变红色；对PP进行右旋



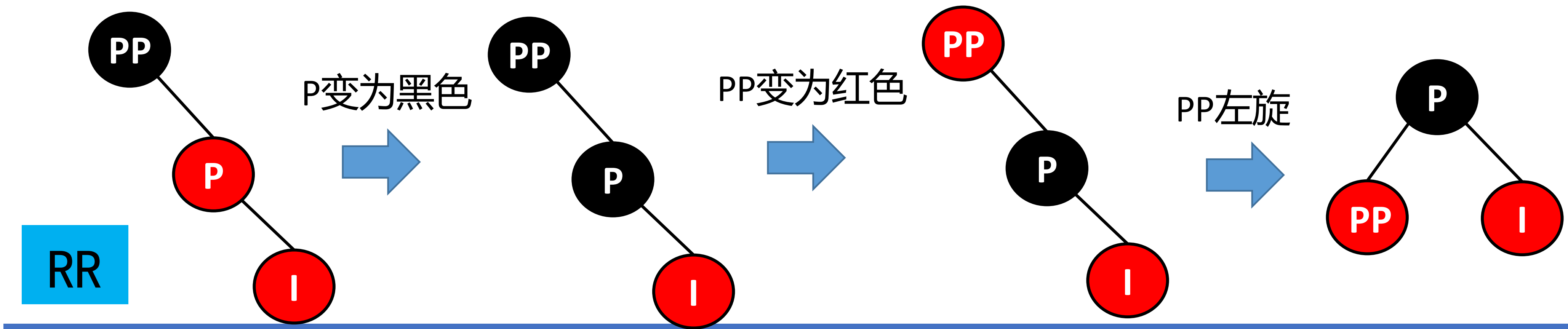
红黑树的插入

B. 叔叔结点不存在或为黑结点

B-2. 插入结点的双亲结点是祖父结点的右孩子

- B-2-1. 插入结点是双亲结点的右孩子(RR)
- B-2-2. 插入结点是双亲结点的左孩子(RL)

操作：P变黑色；PP变红色；对PP进行**左旋**



红黑树的插入

B. 叔叔结点不存在或为黑结点

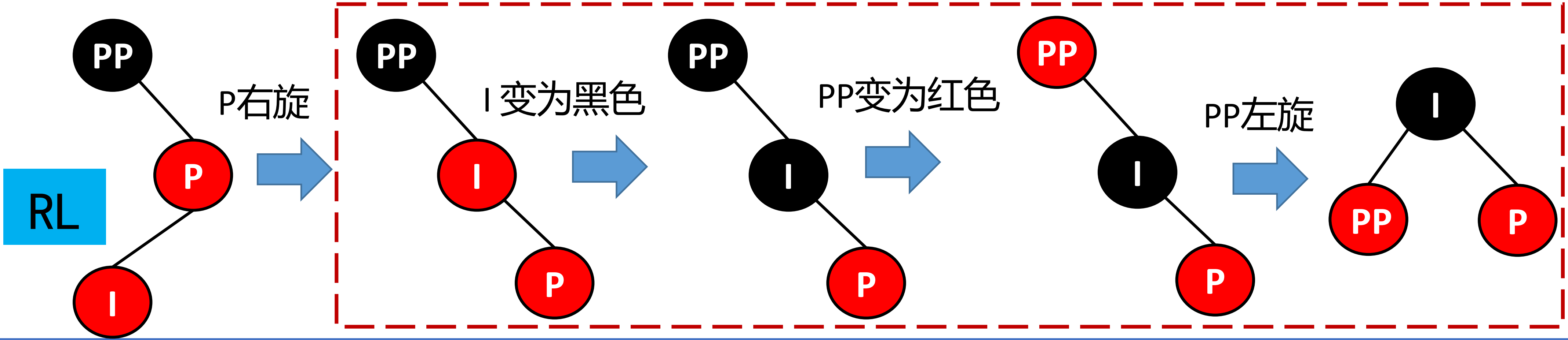
B-2. 插入结点的双亲结点是祖父结点的右孩子

B-2-1. 插入结点是双亲结点的右孩子(RR)

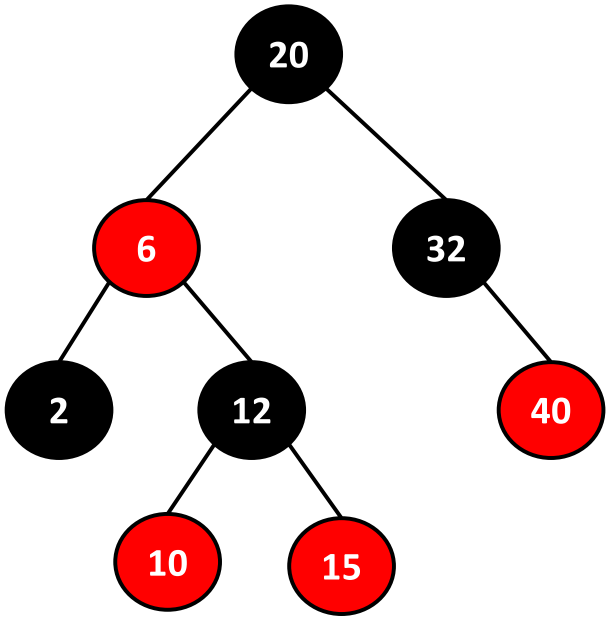
B-2-2. 插入结点是双亲结点的左孩子(RL)

操作：对P结点进行右旋，变成RR

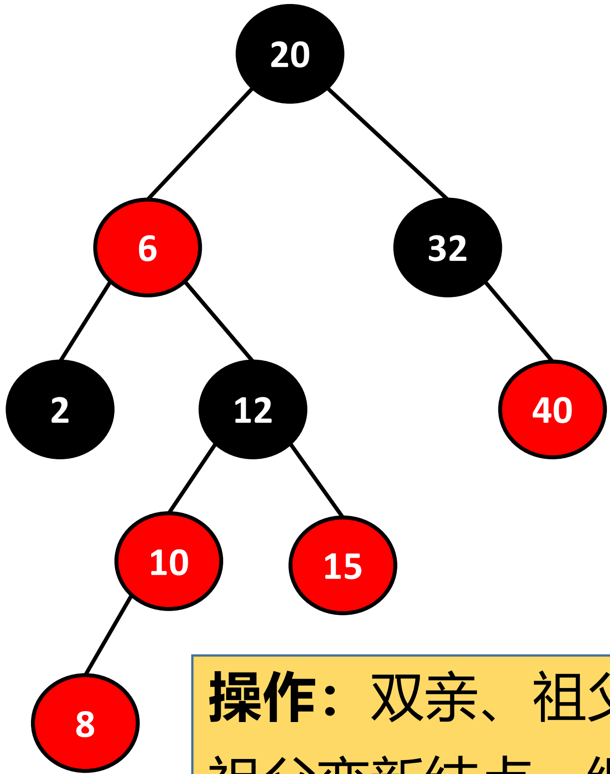
I 变黑色；PP变红色；对PP进行左旋



红黑树插入案例



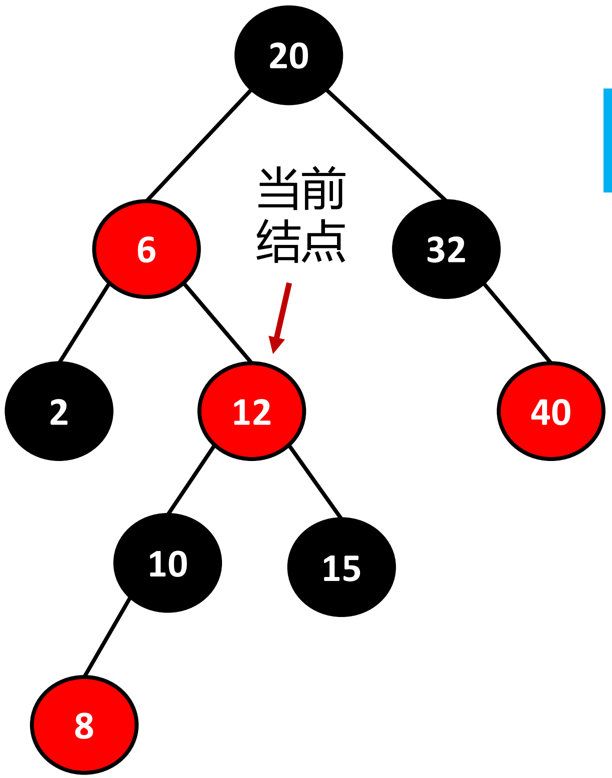
插入结点8



结点8的双亲节点是红色，
且叔叔节点也是红色

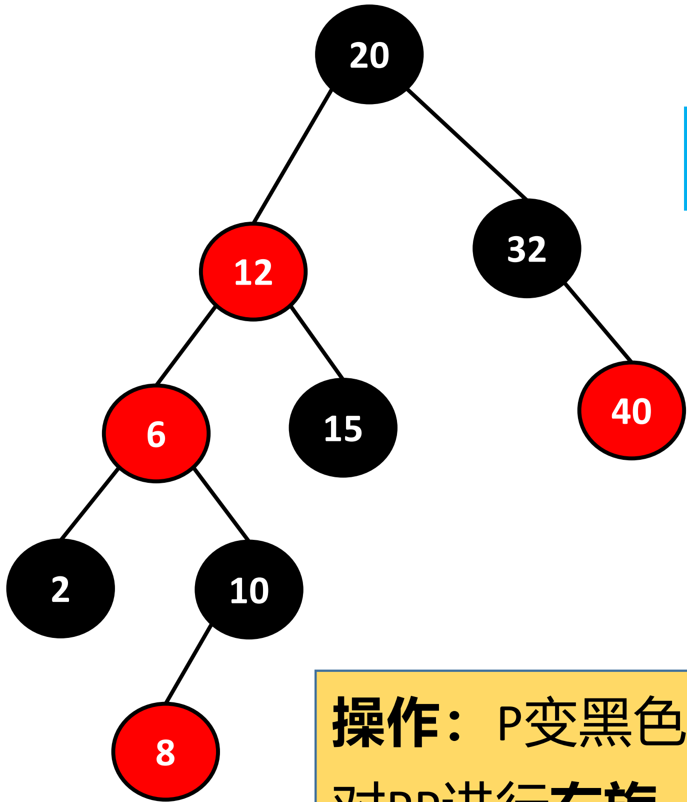
操作：双亲、祖父、叔叔结点全变色；
祖父变新结点，继续调整

红黑树插入案例



LR

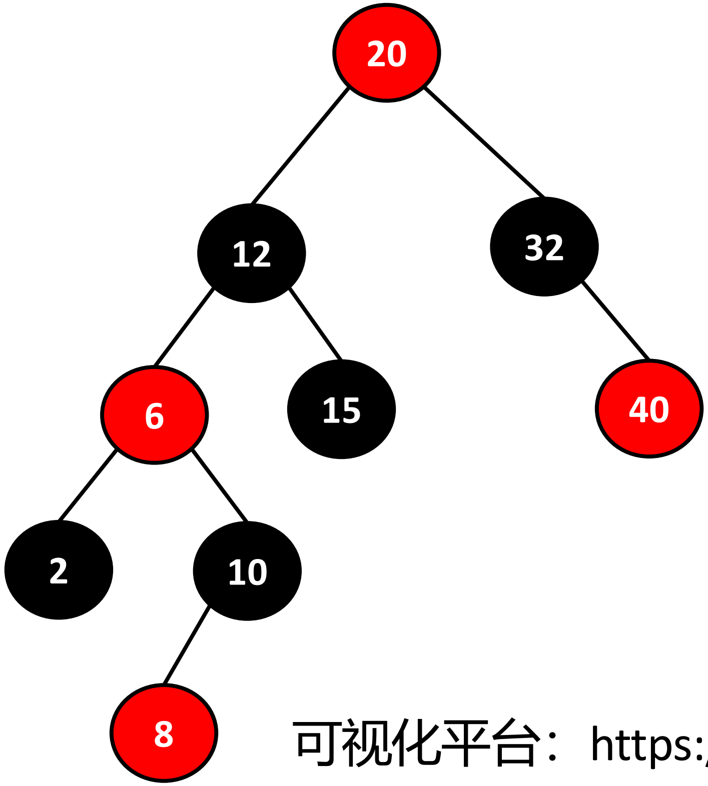
结点6左旋



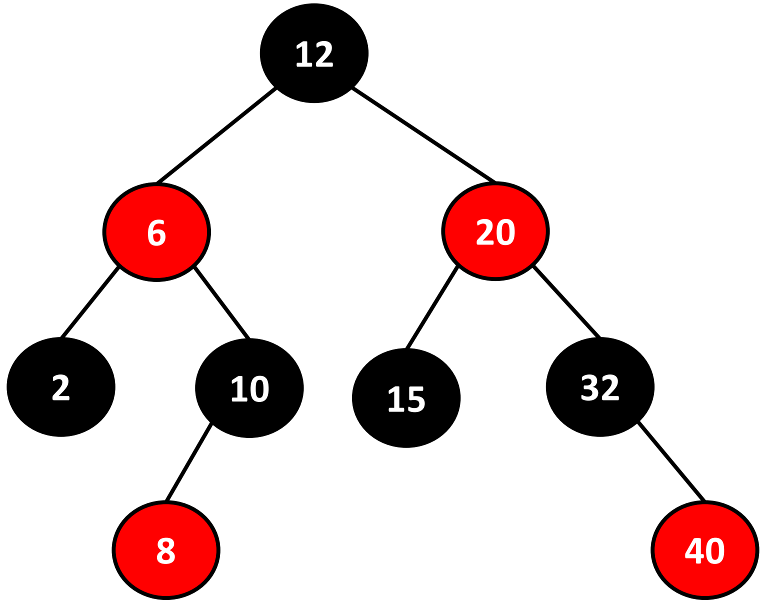
LL

操作： P变黑色； PP变红色；对PP进行**右旋**

红黑树插入案例



结点20右旋



可视化平台：https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

B-树

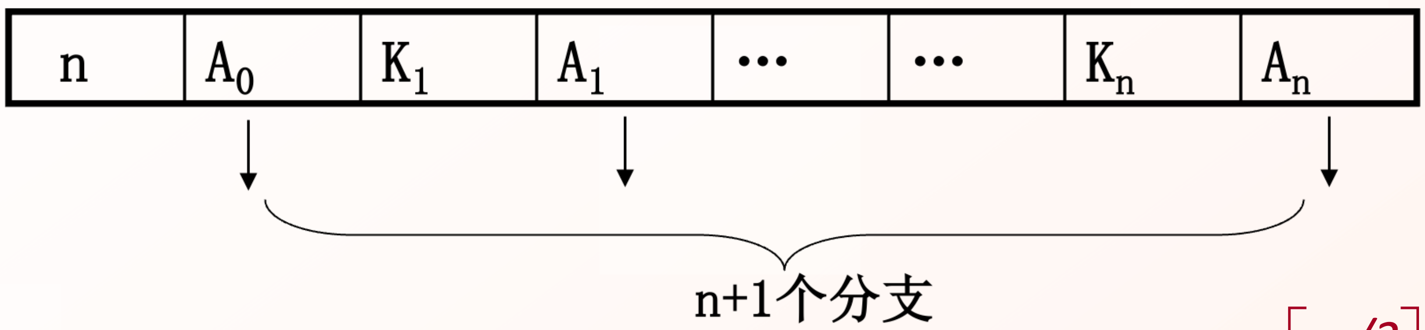
B-树（Balanced Tree） 是一种平衡的多路查找树。

*m*阶的B-树，或为空树，或为满足下列特性的m叉树：

- 树中每个结点至多有*m*棵子树。
- 若根结点不是叶子结点，则至少有两棵子树。
- 除根结点之外的所有非终端结点至少有$\lceil m/2 \rceil$棵子树。 子树数量范围: [$\lceil m/2 \rceil$, m]
- 有k个子树的非终端结点恰好包含k-1个关键字。
- 所有的叶子结点都出现在同一层次，不含任何信息。

B-树

B树包含n个关键字，n+1个指针的结点的一般形式：



$\lceil m/2 \rceil -1<=n<=m-1$

- 其中k是关键字值，k1<k2<…<kn
- Ai是指向包括ki到ki+1之间的关键字的子树的指针。

B-树

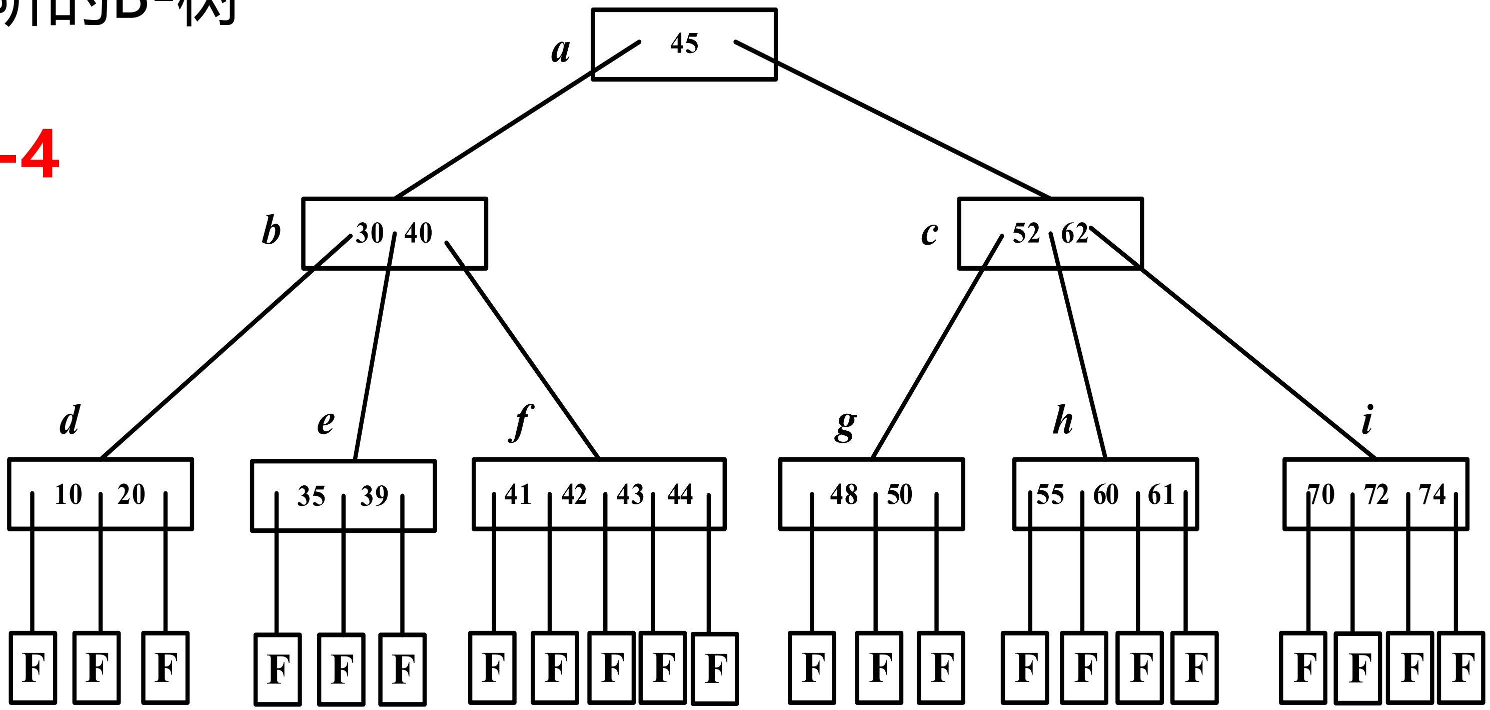
B-树的性质:

- 树高平衡，所有叶结点都在同一层。
- 关键字没有重复，父结点中的关键字是其子结点的分界。
- B-树把值接近的相关记录放在同一个磁盘页中，从而利用了访问局部性原理。
- B-树保证树中至少有一定比例的结点是满的:
  - 这样能够改进空间利用率
  - 减少检索和更新操作的磁盘读取数目

B-树

例：一棵5阶的B-树

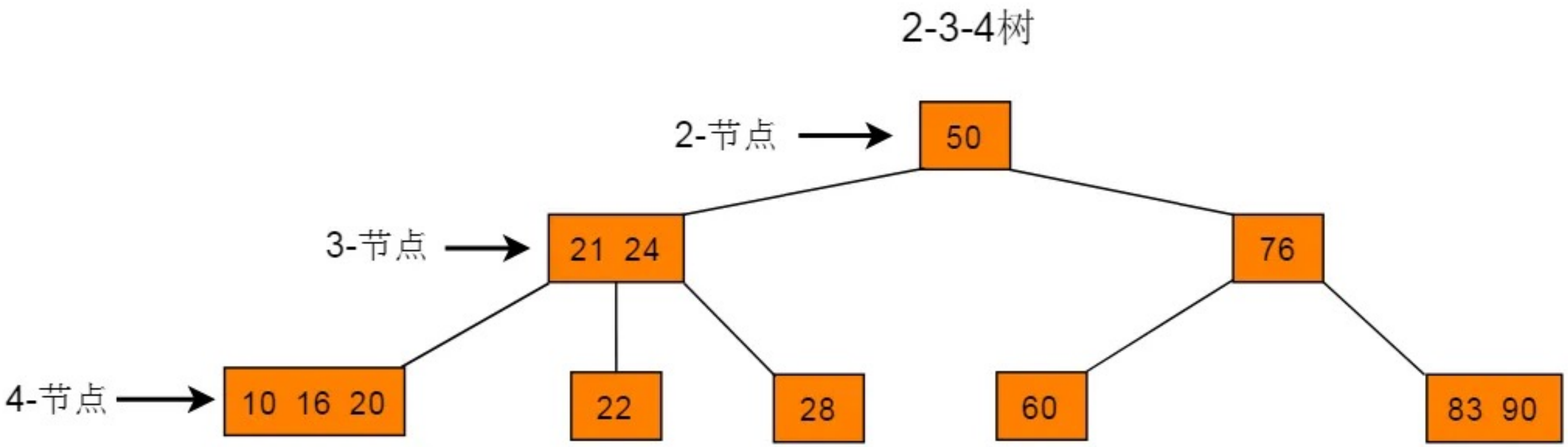
关键字：2-4
子树：3-5



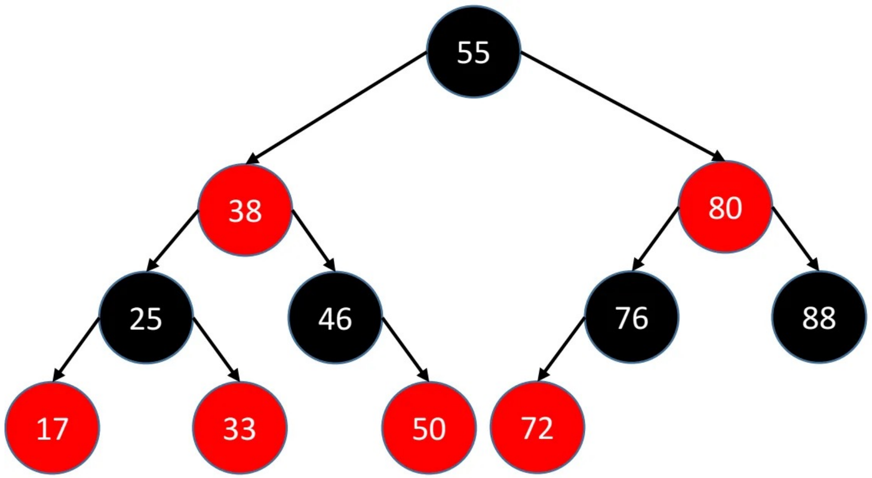
4阶B树（2-3-4树）

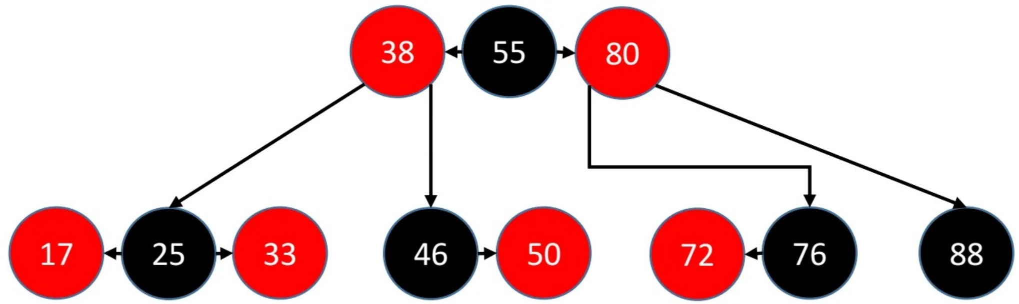
关键字：1-3
子树：2-4

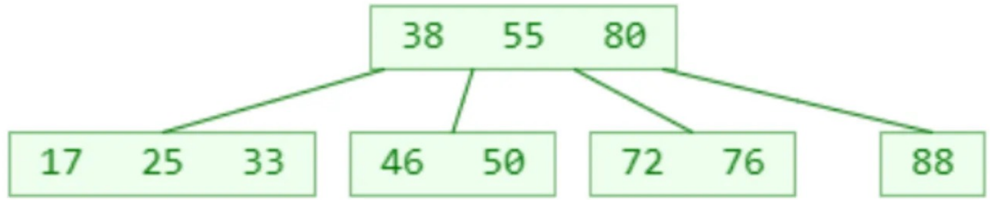
3种结点类型： 2结点、3结点、4结点



红黑树与2-3-4树等价

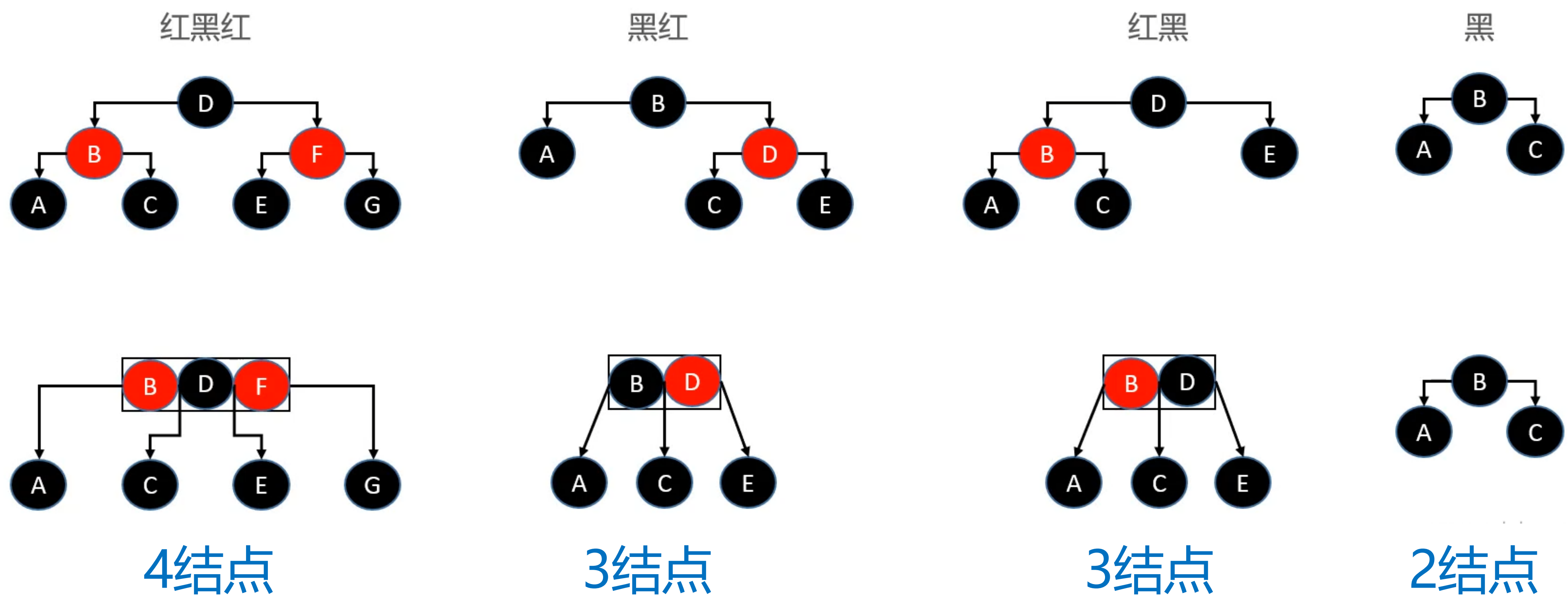






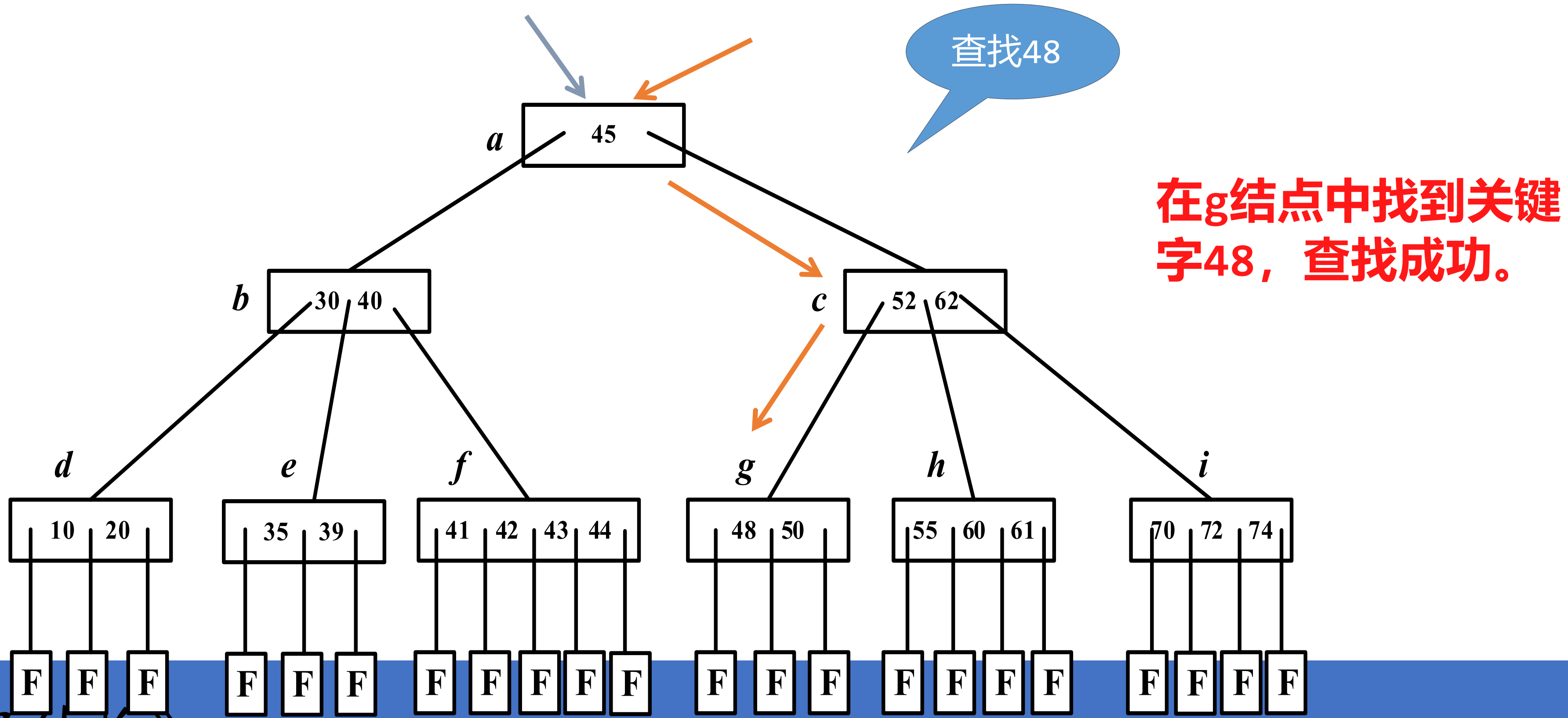
- 红黑树和2-3-4树具有等价性
- 黑色结点和它的红色子结点融合在一起，形成1个B树结点
- 红黑树的黑色结点个数和2-3-4树的结点总数相等

红黑树与2-3-4树等价

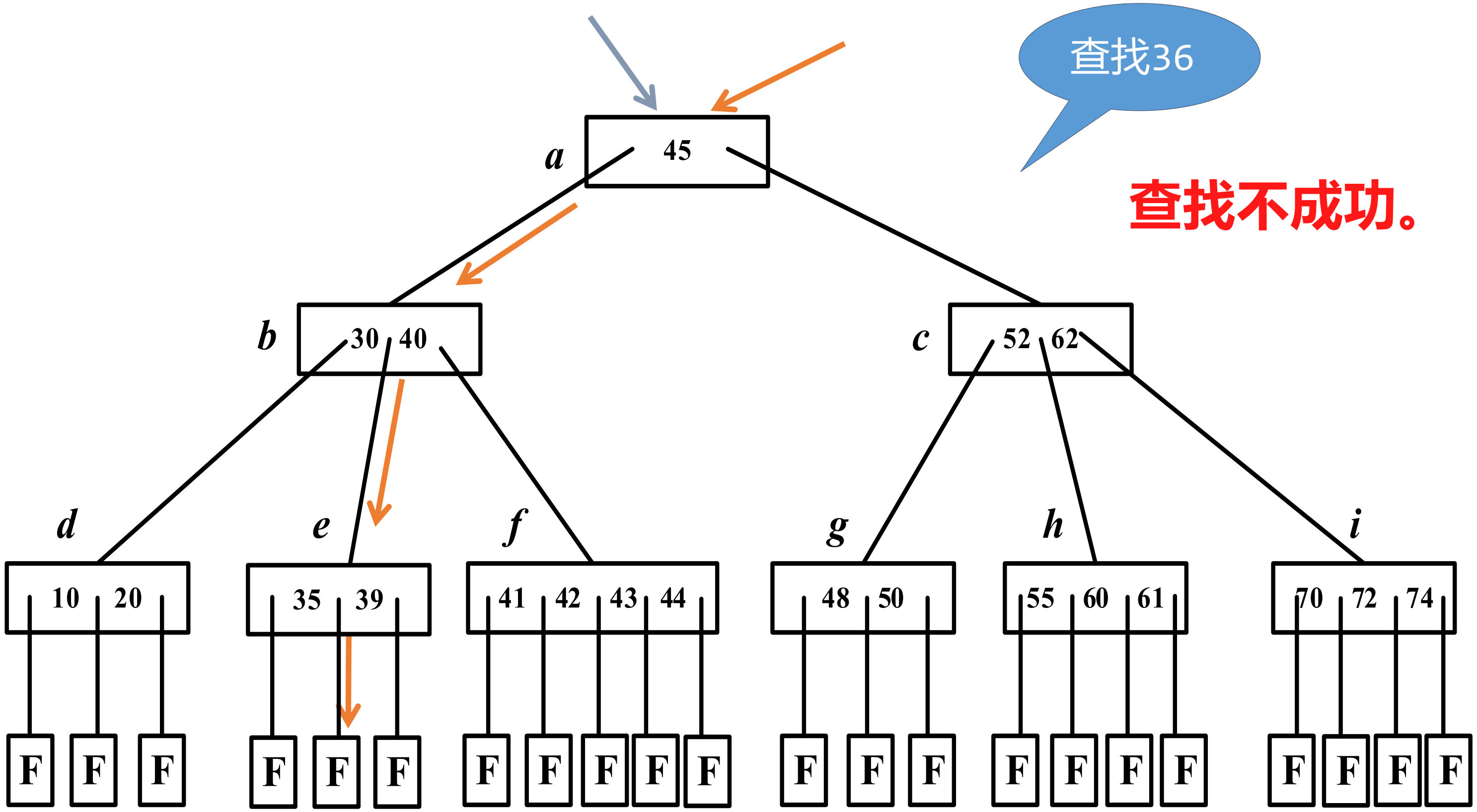


B-树的查找

从根结点出发，沿指针搜索结点和在结点内查找，两个过程交叉进行。



B-树的查找


查找36
a
45
b
30 40
c
52 62
d
10 20
e
35 39
f
41 42 43 44
g
48 50
h
55 60 61
i
70 72 74
F F F F F F F F F F F F F F F F F F F F F F F F F F F

查找不成功。

B-树的查找

在B-树上查找包含两种基本操作：

(1)在B-树上找结点；(2)在结点中找关键字。

由于B-树主要用作文件的索引，它通常存储在磁盘上，因此前一操作涉及磁盘的存取，而后一操作是在内存中进行的。在磁盘上查找的次数，即待查关键字所在结点在 B -树上的层次数，是决定B-树查找效率的主要因素。

在含 N 个关键字的 B-树上进行一次查找，需访问的结点个数不超过：$\log_{\lceil m/2 \rceil}((N+1)/2)+1$。

B-树的查找

查找时间取决于两个因素：

- 给定值的关键字所在结点的层次；
- 结点中关键字的个数。

注意：当查找到叶子结点时，查找失败

B-树的插入

注意保持B-树的性质，特别是等高和阶的限制

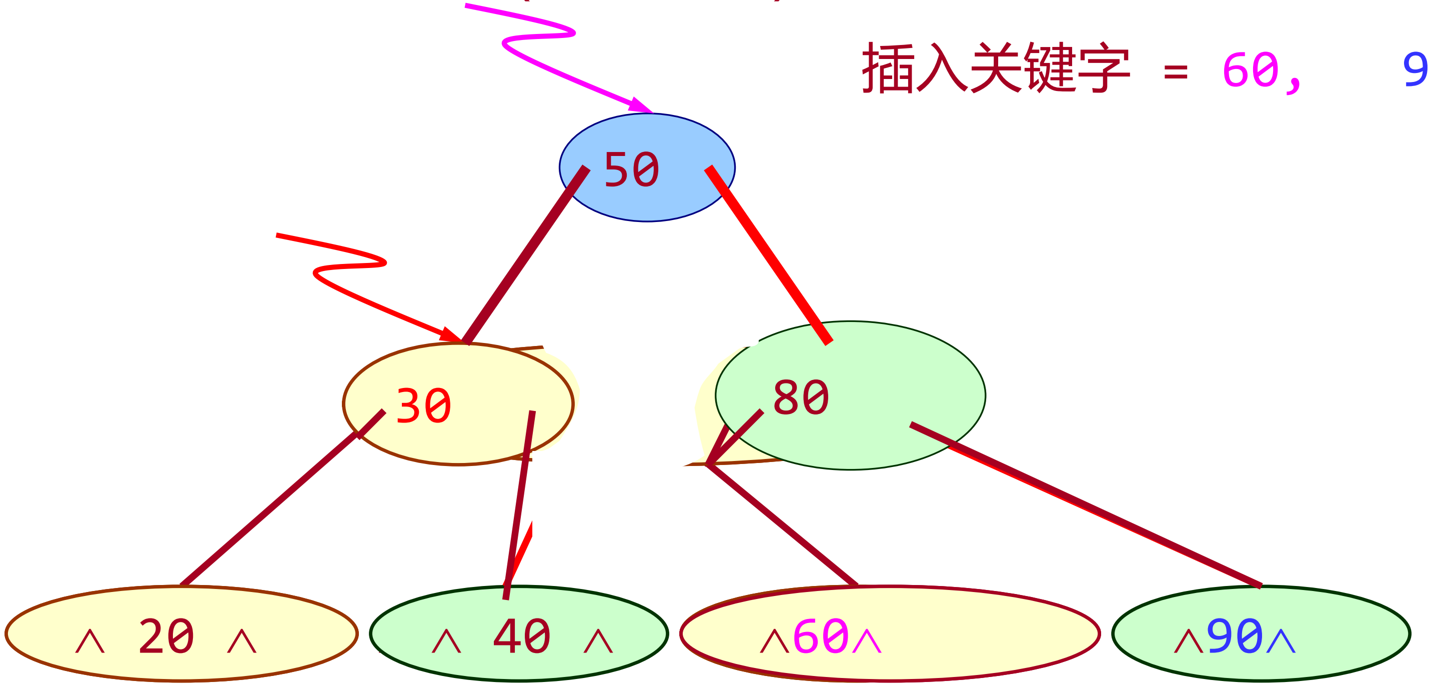
- 找到最底层非终端结点，插入
- 若溢出，则结点分裂，中间关键字同新指针插入双亲结点
- 若双亲结点也溢出，则继续分裂
- 分裂过程可能传达到根结点（则树升高一层）

中间关键字：$\lceil m/2 \rceil$

B-树的插入

例如：下列为 3 阶B-树（称为2-3树）

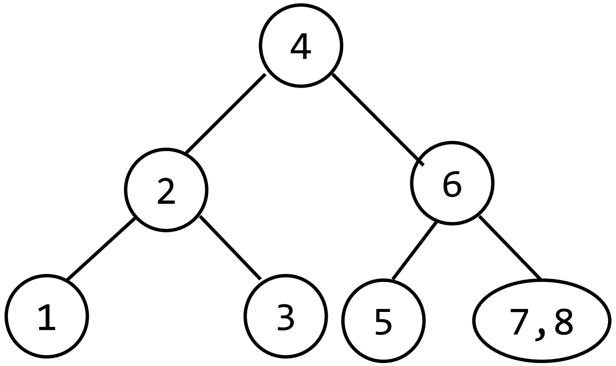
插入关键字 = 60， 90， 30



课堂练习

输入序列：{1， 2， 3， 4， 5， 6， 7， 8}

1) 创建3阶B-树；



B-树的删除

- 删除都是基于查找成功操作的。
- 若删除的关键字不是最下层的非终端节点

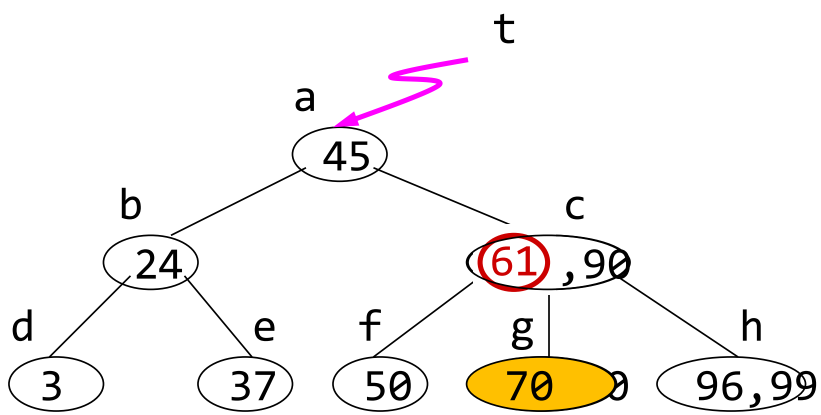
  先把此关键字与它在B树里的后继对换位置，然后再删除该关键字。
- 若删除的关键字是最下层的非终端节点

  1. 删除后关键字的个数不小于$\lceil m/2 \rceil$-1

     直接删除
  2. 删除后关键字的个数小于$\lceil m/2 \rceil$-1
     - 若兄弟结点关键字大于$\lceil m/2 \rceil$-1，从兄弟结点借（通过双亲结点）
     - 若兄弟结点关键字等于$\lceil m/2 \rceil$-1，合并

B-树的删除

删除关键字：53

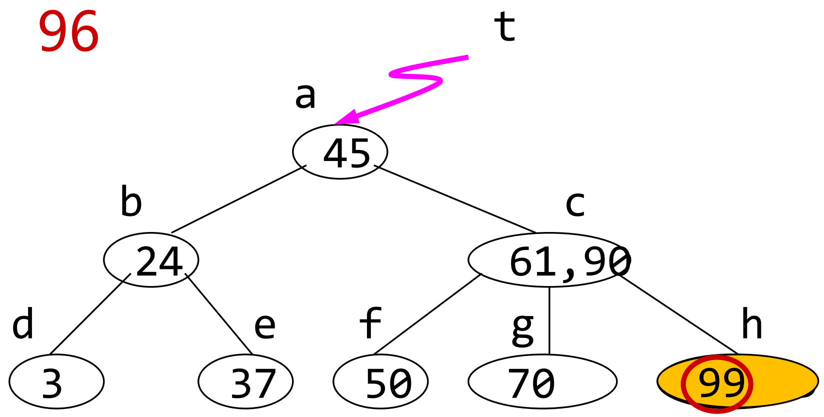


- 删除的关键字不是最下层的非终端节点

  先把此关键字与它在B树里的后继对换位置，然后再删除该关键字。

B-树的删除

删除关键字：96



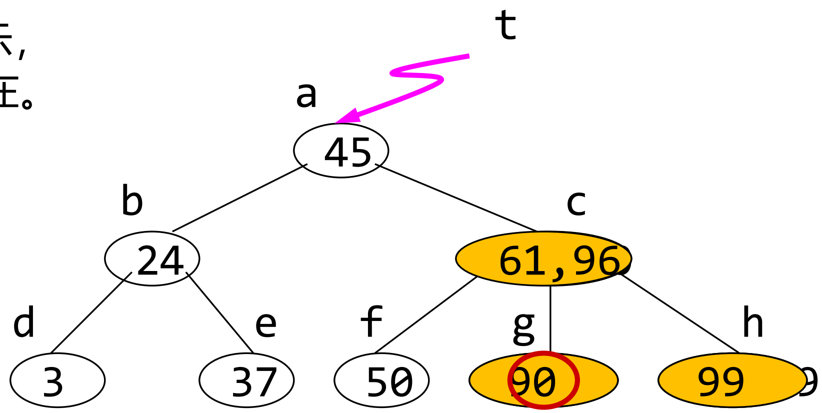
- 删除的关键字是最下层的非终端节点

  1. 删除后关键字的个数不小于⌈m/2⌉-1，直接删除

B-树的删除

删除关键字：70

为了后面例子的展示，
上一页删除的96还在。



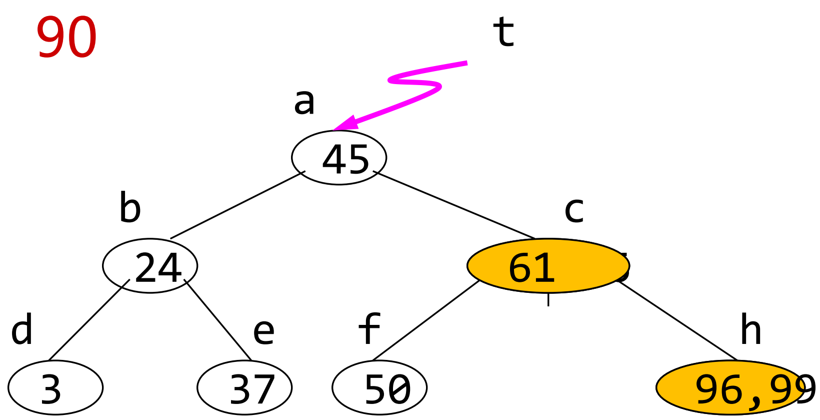
- 删除的关键字是最下层的非终端节点

  2. 删除后关键字的个数小于⌈m/2⌉-1

     若兄弟结点关键字大于⌈m/2⌉-1，从兄弟结点借（通过双亲结点）

B-树的删除

删除关键字：90



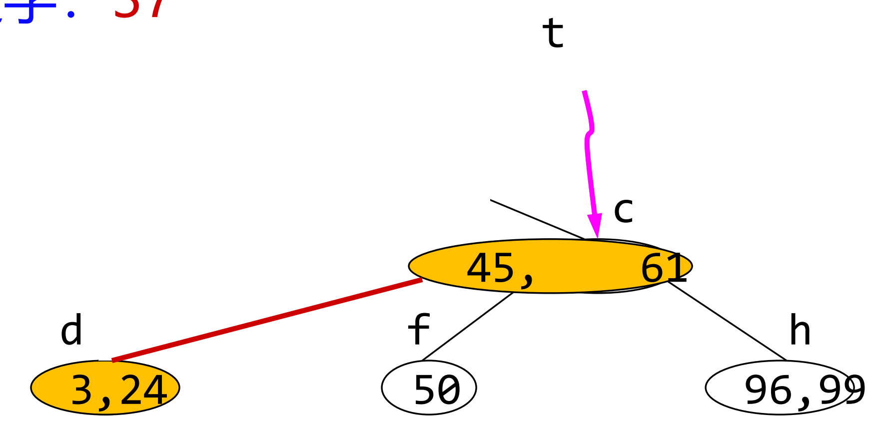
- 删除的关键字是最下层的非终端节点

  3. 删除后关键字的个数小于⌈m/2⌉-1

     若兄弟节点关键字等于⌈m/2⌉- 1，合并

B-树的删除

删除关键字：37



- 删除的关键字是最下层的非终端节点

  3. 删除后关键字的个数小于⌈m/2⌉-1

  若兄弟节点关键字等于⌈m/2⌉- 1，合并

第9章 查找

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

9.1 查找的基本概念

9.2 静态查找表 （线性表的查找）

9.3 动态查找表 （树表的查找）

9.4 哈希表

哈希表的基本概念

基本思想：记录的存储位置与关键字之间存在对应关系

对应关系——hash函数

Loc(i)=H(keyi)

hash函数

Hash：哈希

翻译为：散列、混杂、拼凑

哈希表的查找

例1

若将学生信息按如下方式存入计算机，如:

将200101181021 1的所有信息存入V[11]单元;

将20010118102 07的所有信息存入V[07]单元;

……

将20010118102 31的所有信息存入V[31]单元。

查找200101181021 6的信息，可直接访问V[16]

哈希表的查找

例2

数据元素序列(21，23，39，9，25，11)，若规定每个元素k的存储地址H(k)=k，请画出存储结构图。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
									9		11										21		23		25														39

哈希表的查找

如何查找

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
									9		11										21		23		25														39

根据哈希函数 H(key)=k

查找key=9，则访问H(9)=9号地址，若内容为9则成功;

若查不到，则返回一个特殊值，如空指针或空记录。

优点：查找效率高

缺点：空间效率低!

哈希表的若干术语

哈希方法(杂凑法)

选取某个函数，依该函数按关键字计算元素的存储位置，并按此存放;

查找时，由同一个函数对给定值k计算地址，将k与地址单元中元素关键码进行比，确定查找是否成功。

哈希函数(杂凑函数)

哈希方法中使用的转换函数

哈希表的若干术语

哈希表(杂凑表)：按上述思想构造的表

哈希函数：H(key)=k

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
									9		11										21		23		25														39

哈希表的若干术语

冲突：不同的关键码映射到同一个哈希地址

$$key1 \neq key2，但是H(key1)=H(key2)$$

例：有6个元素的关键码分别为：(21，23，39，25，9，11)。

- 选取关键码与元素位置间的函数为 H(k)= k mod 7，
- 地址编号从0-6。

有冲突！

通过哈希函数对6个元素建立哈希表：

0	1	2	3	4	5	6
21		23		39		

H(21)=21%7=0

9

25

11

H(25)=25%7=4
H(11)=11%7=4

同义词：具有相同函数值的多个关键字

哈希函数的构造方法

哈希存储

选取某个函数，依该函数按关键字计算元素的存储位置

Loc(i)=H(keyi)

冲突

不同的关键码映射到同一个哈希地址

key1≠key2，但是H(key1)=H(key2)

在哈希查找方法中，冲突是不可能避免的，只能尽可能减少。

哈希函数的构造方法

使用哈希表要解决好两个问题:

1）构造好的哈希函数

(a）所选函数尽可能简单，以便提高转换速度;

(b）所选函数对关键码计算出的地址，应让哈希地址集中且均匀分布，以减少空间浪费。

2）制定一个好的解决冲突的方案

查找时，如果从哈希函数计算出的地址中查不到关键码，则应当依据解决冲突的规则，有规律地查询其它相关单元。

哈希函数的构造方法

构造哈希函数考虑的因素

① 执行速度(即计算哈希函数所需时间);

② 关键字的长度;

③ 哈希表的大小;

④ 关键字的分布情况;

⑤ 查找频率。

哈希函数的构造方法

根据元素集合的特性构造

- 要求一：n个数据原仅占用n个地址，虽然哈希查找是以空间换时间，但仍希望哈希的**地址空间尽量小。**
- 要求二：无论用什么方法存储，目的都是尽量**均匀**地存放元素，以避免冲突。

1. 直接定址法
2. 数字分析法
3. 平方取中法
4. 折叠法
5. 除留余数法
6. 随机数法

哈希函数的构造方法

哈希函数——直接定址法

Hash(key) = a·key + b　(a、b为常数)

优点：以关键码key的某个线性函数值为哈希地址，不会产生冲突。

缺点：要占用连续地址空间，空间效率低。

例：{100，300，500，700，800，900}，

哈希函数 Hash(key)=key/100　(a=1/100, b=0)

适用情况？

0	1	2	3	4	5	6	7	8	9
	100		300		500		700	800	900

事先知道关键字，关键字集合不是很大且连续性较好。

哈希函数的构造方法

哈希函数——除留余数法

Hash(key) = key mod p　　(p是一个整数)

关键：如何选取合适的p?

技巧：设表长为m，取 p≤m且为质数（最好接近m）

例: {15，23，27，38，53，61，70}，哈希函数 Hash(key) = key mod 7

0	1	2	3	4	5	6
70	15	23	38	53	61	27

适用情况？

除留余数法是一种最简单、也是最常用的构造哈希函数的方法，并且不要求事先知道关键码的分布。

哈希函数的构造方法

哈希函数——1、数字分析法

根据关键码在各个位上的分布情况，选取分布比较均匀的若干位组成哈希地址。

例：关键码为8位十进制数，哈希地址为2位十进制数

①	②	③	④	⑤	⑥	⑦	⑧
8	1	3	4	6	5	3	2
8	1	3	7	2	2	4	2
8	1	3	8	7	4	2	2
8	1	3	0	1	3	6	7
8	1	3	2	2	8	1	7
8	1	3	3	8	9	6	7

适用情况:

能预先估计出全部关键码的每一位上各种数字出现的频度，不同的关键码集合需要重新分析。

哈希函数的构造方法

哈希函数——2、平方取中法

对关键码平方后，按哈希表大小，取中间的若干位作为哈希地址（平方后截取）。

例：哈希地址为2位，则关键码1234的哈希地址为：

$$(1234)^2 = 1522756$$

适用情况:

事先不知道关键码的分布且关键码的位数不是很大。

哈希函数的构造方法

哈希函数——3、折叠法

将关键码从左到右分割成位数相等的几部分，将这几部分叠加求和，取后几位作为哈希地址。

例：设关键码为2 5 3 4 6 3 5 8 7 0 5，哈希地址为三位。

```
  2 5 3
  4 6 3
  5 8 7
+   0 5
-------
1 3 0 8
```

```
  2 5 3
  3 6 4
  5 8 7
+   5 0
-------
1 2 5 4
```

适用情况:

关键码位数很多，事先不知道关键码的分布。

移位叠加：将分割后的几部分低位对齐相加
间界叠加：从一端沿分割界来回折送，然后对齐相加

处理冲突的方法

杭州電子科技大学
HANGZHOU DIANZI UNIVERSITY

1. 开放定址法（开地址法)
2. 链地址法 (拉链法)

1. 开放定址法（开地址法）

杭州電子科技大學 HANGZHOU DIANZI UNIVERSITY

基本思想：有冲突时就去寻找下一个空的哈希地址，只要哈希表足够大，

空的哈希地址总能找到，并将数据元素存入。

例如：　除留余数法　$H_i=(Hash(key)+d_i) \bmod m$　　d_i为增量序列

常用方法：

方法	增量序列
线性探测法	d_i为 1，2，…，m-1线性序列
二次探测法	d_i为 1^2，-1^2，2^2，-2^2，…，q^2二次序列
伪随机探测法	d_i为伪随机数序列

1. 开放定址法（开地址法）

杭州電子科技大学 HANGZHOU DIANZI UNIVERSITY

线性探测法

$H_i = (Hash(key)+d_i) \bmod m$　　($1 \le i < m$)

其中：m为哈希表长度

d_i为增量序列 1，2，…，m-1，且d_i=i

一旦冲突，就找下一个地址，直到找到空地址存入

例题

关键码集为{47，7，29，11，16，92，22，8，3}，哈希表的长度为m=11; 哈希函数为Hash(key)=key mod 11; 拟用线性探测法处理冲突。建哈希表如下:

0	1	2	3	4	5	6	7	8	9	10
11	22		47	92	16	3	7	29	8	
①	②		①	①	①	④	①	②	②	

解释:

① 47、7均是由哈希函数得到的没有冲突的哈希地址;

② Hash(29)=7，哈希地址有冲突，需寻找下一个空的哈希地址:

由H_1=(Hash(29)+1) mod 11=8，哈希地址8为空，因此将29存入。

③ 11、16、92均是由哈希函数得到的没有冲突的哈希地址;

④ 另外，22、8、3同样在哈希地址上有冲突，也是由H_1找到空的哈希地址的。

∴ 平均查找长度ASL=(1+2 + 1+1 + 1+4+1 + 2+2)/9=1.67

1. 开放定址法（开地址法）

二次探测法

关键码集为 {47，7，29，11，16，92，22，8，3}，

设:　　哈希函数为　Hash(key)=key mod 11

H_i = (Hash(key)+d_i) mod m

其中：m为哈希表长度;

d_i为增量序列 1^2，-1^2，2^2，-2^2，…，q^2，$-q^2$，其中 q≤m/2

0	1	2	3	4	5	6	7	8	9	10
11	22	3	47	92	16		7	29	8	

Hash(3)=3，哈希地址冲突，由

H_1=(Hash(3)+1^2) mod 11=4, 仍然冲突;

H_2=(Hash(3)-1^2) mod 11=2, 找到空的哈希地址，存入。

1. 开放定址法（开地址法）

杭州电子科技大学 HANGZHOU DIANZI UNIVERSITY

伪随机探测法

H_i = (Hash(key)+d_i) mod m　　（ $1 \le i < m$ ）

其中：m为哈希表长度

d_i为伪随机数

处理冲突的方法

1. 开放定址法（开地址法)

2. 链地址法 (拉链法)

2. 链地址法（拉链法）

杭州電子科技大學 HANGZHOU DIANZI UNIVERSITY

基本思想：相同哈希地址的记录链成一单链表

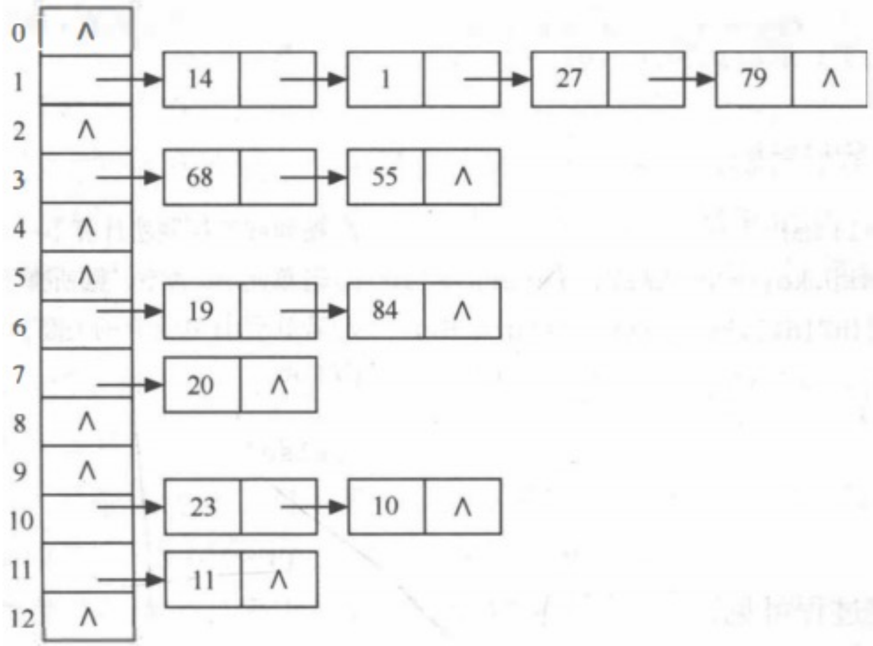
m个哈希地址就设m个单链表，然后用一个数组将m个单链表的表头指针存储起来，形成一个动态的结构。

例如：一组关键字为

{19,14,23,1,68,20,84,27,55,11,10,79}

哈希函数为

Hash(key)=key mod 13



2. 链地址法（拉链法）

杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

链地址法建立哈希表步骤

- Step1：取数据元素的关键字key，计算其哈希函数值（地址)。若该地址对应的链表为空，则将该元素插入此链表; 否则执行Step2解决冲突。

- Step2：若该地址对应的链表为不为空，则利用链表的前插法或后插法将该元素插入此链表。

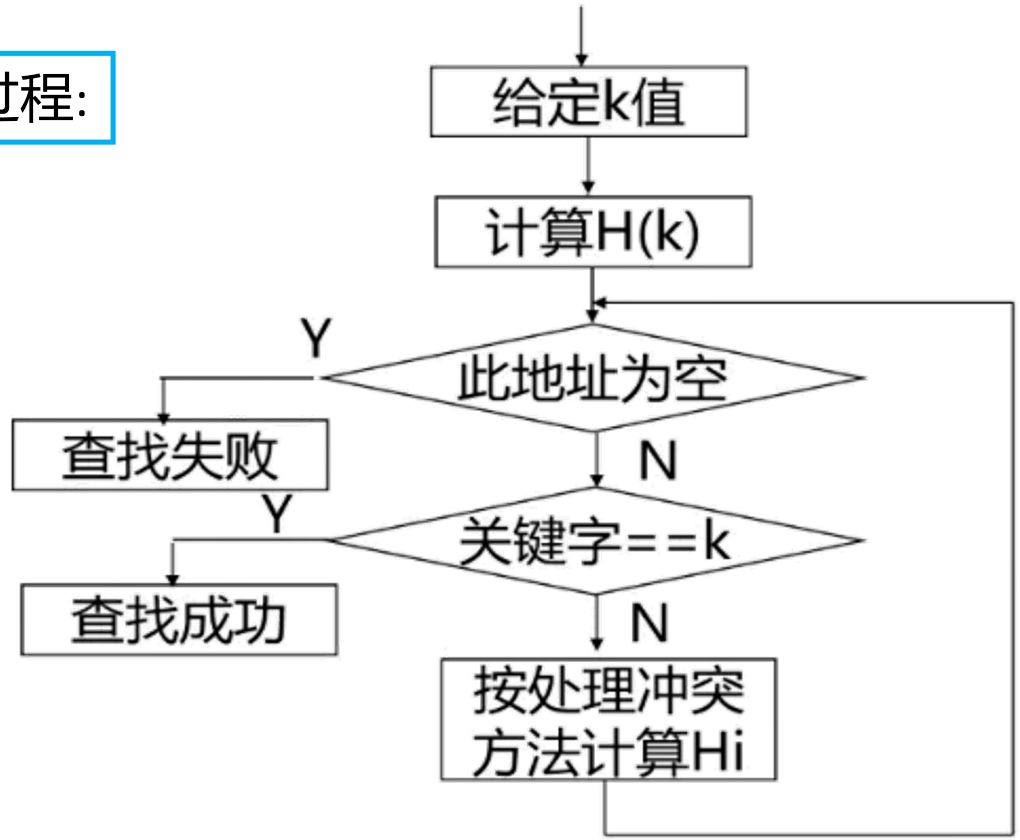
2. 链地址法（拉链法）

链地址法的优点

- 非同义词不会冲突，无“聚集”现象
- 链表上结点空间动态申请，更适合于表长不确定的情况

哈希表的查找

给定值查找值k，查找过程:



例题

已知一组关键字(19, 14, 23, 1, 68, 20, 84, 27, 55, 11, 10, 79)

哈希函数为：H(key)=key MOD 13，哈希表长为m=16,

设每个记录的查找概率相等

(1) 用线性探测再哈希处理冲突，即Hi=(H(key)+di) MOD m

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	14	1	68	27	55	19	20	84	79	23	11	10			
	1	2	1	4	3	1	1	3	9	1	1	3			

H(19)=6

H(14)=1

H(23)=10

H(1)=1 冲突，H(1)=(1+1) MOD 16=2

H(68)=3

H(20)=7

H(84)=6 冲突，H(84)=(6+1) MOD 16=7

冲突，H(84)=(6+2) MOD 16=8

H(27)=1 冲突，H(27)=(1+1) MOD 16=2

冲突，H(27)=(1+2) MOD 16=3

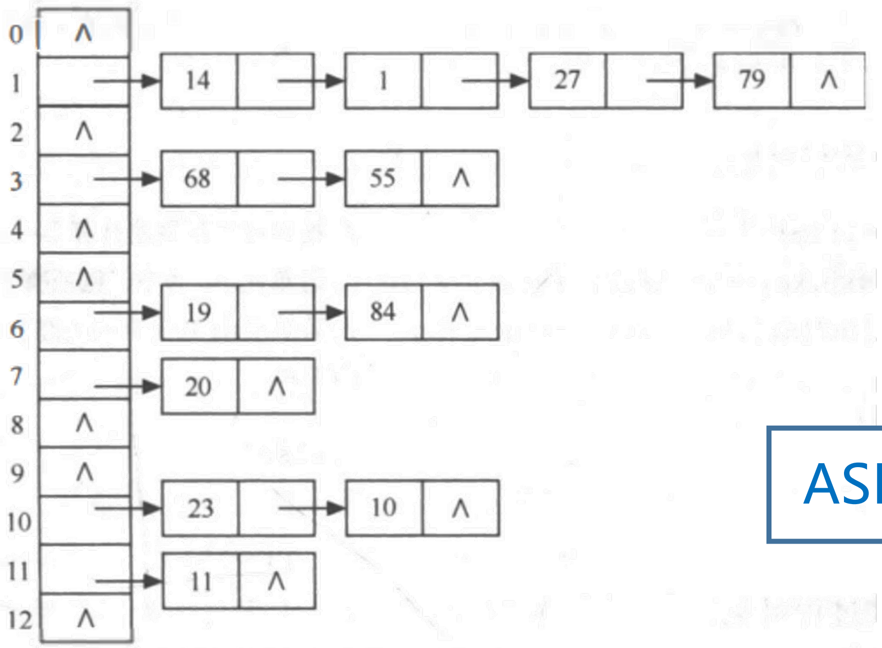
冲突，H(27)=(1+3) MOD 16=4

ASL=(1*6+2+3*3+4+9)/12=2.5

例题

（2）用链地址法处理冲突

关键字为{19, 14, 23, 1, 68, 20, 84, 27, 55, 11, 10, 79}



ASL = (1*6+2*4+3+4)/12 = 1.75

思考

对于关键字集(19, 14, 23, 1, 68, 20, 84, 27, 55, 11, 10, 79)，n=12

无序表查找ASL?　　$ASL = (12+1)/2 = 6.5$

有序表折半查找ASL?　　$ASL = \log_2(12+1)-1 = 2.7$

那么，哈希表上查找ASL?

O(1)?

哈希表的查找效率分析

使用平均查找长度ASL来衡量查找算法，ASL取决于

- 哈希函数
- 处理冲突的方法
- 哈希表的装填因子α

$$\alpha = \frac{\text{表中填入的记录数}}{\text{哈希表的长度}}$$

α越大，表中记录数越多，说明表装得越满，发生冲突的可能性就越大，查找时比较次数就越多。

哈希表的查找效率分析

杭州電子科技大学 HANGZHOU DIANZI UNIVERSITY

ASL与装填因子α有关！既不是严格的O(1)，也不是O(n)

$$ASL \approx 1 + \frac{\alpha}{2} \qquad \text{(拉链法)}$$

$$ASL \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \qquad \text{(线性探测法)}$$

$$ASL \approx -\frac{1}{\alpha}\ln(1-\alpha) \qquad \text{(随机探测法)}$$

几点结论

- 哈希表技术具有很好的平均性能，优于一些传统的技术
- 链地址法优于开地址法
- 除留余数法作哈希函数优于其它类型函数