数据结构与算法

主讲教师：刘峤

第9章 查找

第8章 内容提要

8.1 线性表上的查找

8.2 树上的查找

8.3 哈希表（散列表）

查找的基本概念

- 什么是查找表?
  - 查找表是由同类型的数据元素(或记录)构成的集合
- 查找表的基本操作
  - 查询：查询某个数据元素是否在查找表中
  - 检索：检索某个数据元素的各种属性
  - 插入：在查找表中插入一个数据元素
  - 删除：从查找表中删除某个数据元素

查找的基本概念

- 查找操作通常是依据数据元素的某个数据项进行
  - 这个数据项通常是数据的关键字
- 关键字
  - 是数据元素中某个数据项的值，用以标识数据元素
  - 若关键字能唯一标识一个数据元素，称为主关键字
  - 若关键字能标识若干个数据元素，则称为次关键字

查找的基本概念

- 查找操作的定义
  - 根据给定的某个值，在查找表中确定一个其关键字等于给定值的数据元素，这个操作称为查找操作
  - 若查找表中存在这样一个记录，则称“查找成功”
    - 返回查找结果：数据元素的信息
    - 或返回该数据元素在查找表中的位置
  - 否则称“查找不成功”

查找的基本概念

- 查找算法的评价方法
  - 查找算法的复杂程度
    - 时间复杂度：查找速度
    - 空间复杂度：算法占用多少存储空间
  - 平均查找长度：ASL（Average Search Length）
    - 为确定记录在表中的位置
    - 需和给定值进行比较的关键字的个数的期望值

查找的基本概念

- 平均查找长度：ASL（Average Search Length）
  - 查找过程中与给定值进行比较的关键字的个数的期望值
  - 对含有n个记录的表：

$$ASL = \sum_{i=1}^{n} p_i c_i$$

  - p_i 表示用户希望查找表中第 i 个元素的概率
  - c_i 表示从表中找出第 i 个元素所需进行的比较次数

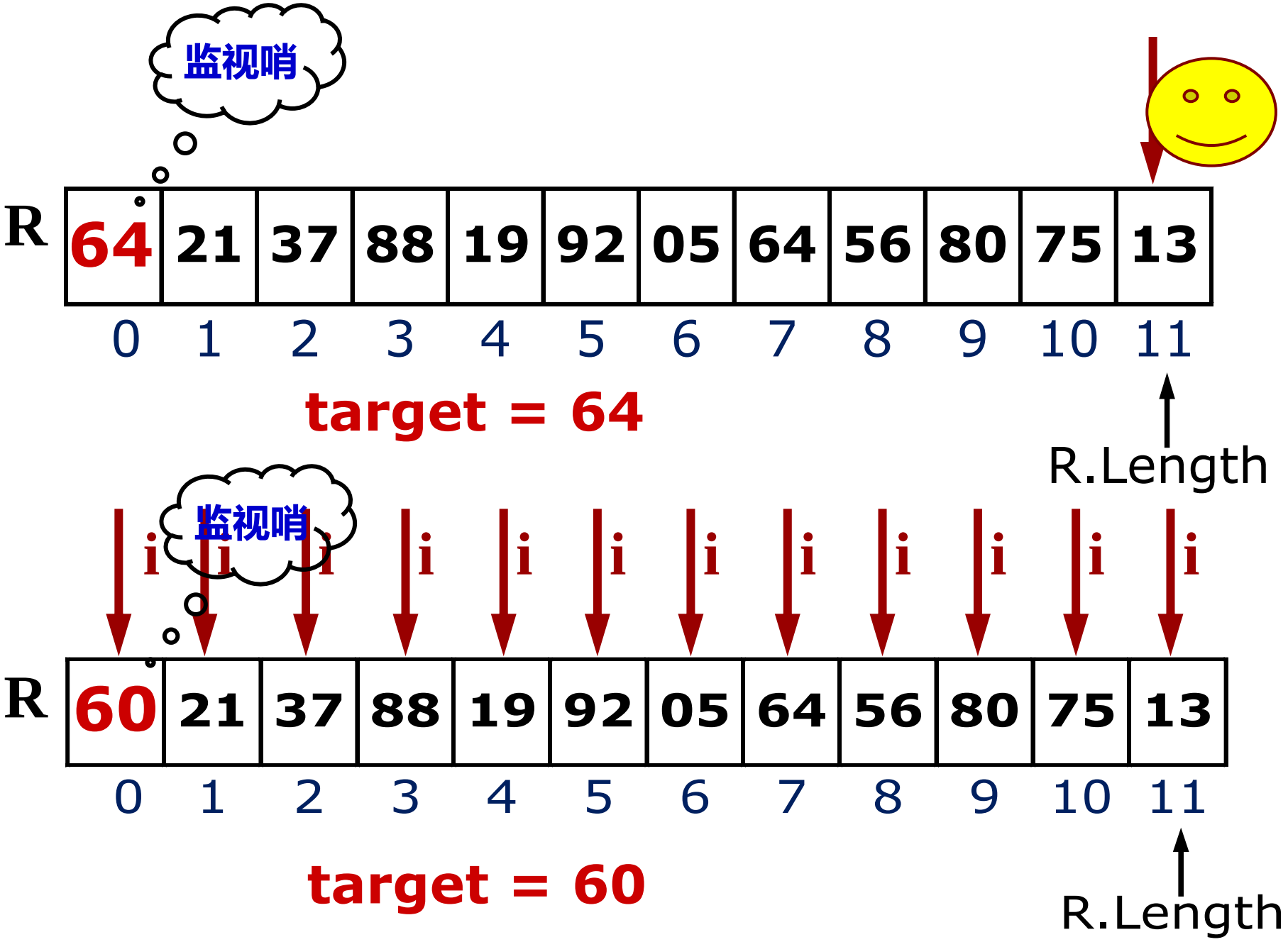
1. 线性表上的查找

- 顺序查找算法
- 折半查找算法
- 索引查找算法

顺序表：顺序查找

顺序查找

- 顺序查找算法的基本思想
  - 从顺序表中的指定位置开始，沿某个方向进行顺序查找
  - 查找过程中将表中记录的关键字与给定值相比较
    - 若某个记录的关键字和给定值相等，则**查找成功**
    - 若找完整个顺序表，都没有与给定关键字值相等的记录
      - 则**查找失败**：可判定表中没有满足查找条件的记录


监视哨
R
64 21 37 88 19 92 05 64 56 80 75 13
0 1 2 3 4 5 6 7 8 9 10 11
target = 64
R.Length
监视哨
i i i i i i i i i i i i
R
60 21 37 88 19 92 05 64 56 80 75 13
0 1 2 3 4 5 6 7 8 9 10 11
target = 60
R.Length

顺序查找

```
int sequential_search(int R[], int n, int tar) {
    int i = n;   R[0] = tar;
    while (R[i] != tar)   i--;
    return i;
}
```

- 顺序查找算法
  - 对R[0]位置的关键字赋值target，目的在于免去查找过程中每一次都要检测整个顺序表是否查找完毕
  - R[0]起到了一个监视哨的作用，在数据量较大或需要反复查找元素时，能一定程度地减少查找的时间

顺序查找算法性能分析

- 空间复杂度：**O(1)**
  - 需要一个辅助存储单元空间R[0]
- 时间复杂度：（基本运算：关键字比较）
  - 最好情况：**O(1)**
    - 第一次比较就成功找到所需数据
  - 最坏情况：**O(n)**
    - 所查找的记录不在顺序表中，这时需要和整个顺序表的记录进行比较，比较的次数为n+1
  - 平均情况：需要和顺序表中大约一半的记录进行比较，即比较次数为n/2，因而时间复杂度为：**O(n)**

顺序查找算法性能分析

- 顺序表上执行顺序查找的平均查找长度
  - 对含有n个记录的表：$ASL = \sum_{i=1}^{n} p_i c_i$
    - p_i 表示用户希望查找表中第 i 个元素的概率
    - c_i 表示从表中找出第 i 个元素所需进行的比较次数
  - 对于顺序表有：$c_i = n - i + 1$
  - 等概率情况下：$p_i = 1 / n$

$$ASL = \frac{1}{n}\sum_{i=1}^{n}(n-i+1) = \frac{n+1}{2}$$

  - 不等概率情况下：
    - ASL 在 $P_n \geq P_{n-1} \geq \cdots \geq P_2 \geq P_1$ 时取极小值

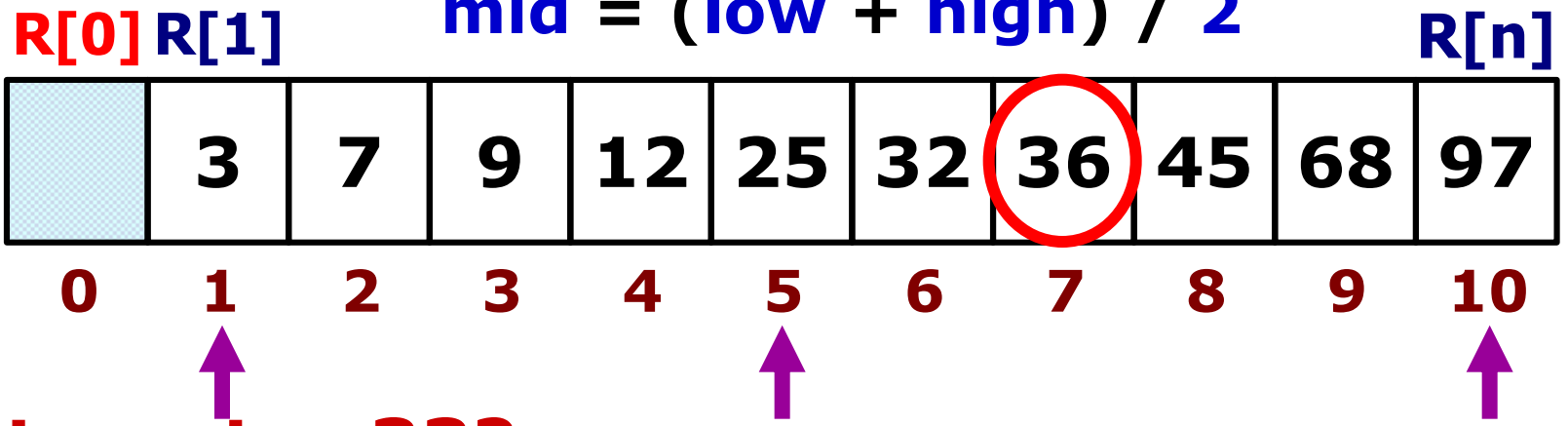
有序表：折半查找

有序表上的查找：折半查找

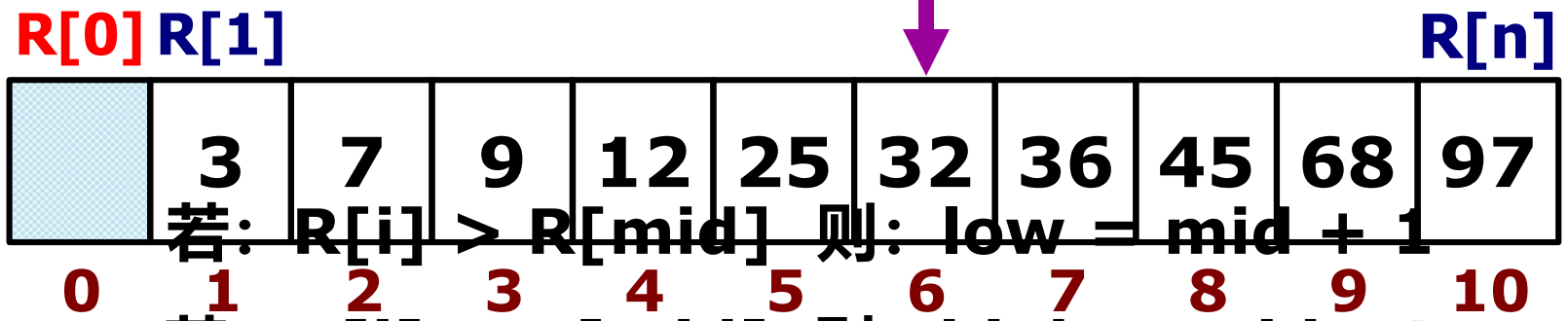
- 有序表（顺序表中的记录按关键字的取值有序）
  - 若：R[i]≤R[i+1]（或 R[i]≥R[i+1]）　(i = 1, 2, ..., n-1)
  - 则称这种形式的顺序表为：有序表
- 折半查找算法的基本思想
  - 将待查关键字与有序表**中间位置**的记录进行比较
  - 若二者相等，查找成功，返回记录在表中的位置
  - 若待查关键字小于中间记录，则只可能在表的前半部分
  - 若待查关键字大于中间记录，则只可能在表的后半部分
  - 因此经过一次比较，就将查找范围缩小一半，这样一直进行下去直至找到目标记录，或判定记录不在查找表中

例如: target = 36 的查找过程如下:

mid = (low + high) / 2



target = 33?



若: R[i] > R[mid] 则: low = mid + 1

若: R[i] ≤ R[mid] 则: high = mid - 1

若: low > high 则: 结束查找，查找不成功

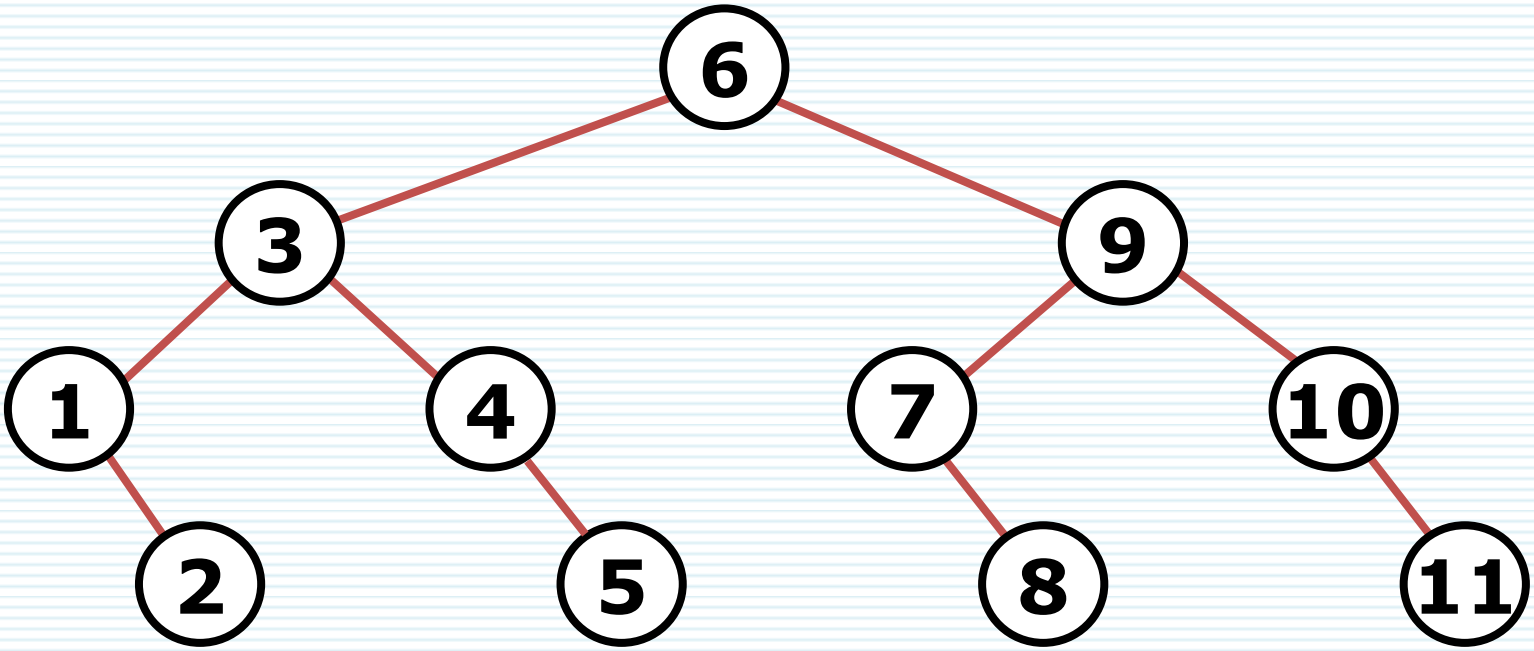
折半查找

- 算法流程
  - 设顺序表R长度为n+1，k为给定关键字（查找目标）
    - low和high分别指向待查元素所在区间的上界和下界
    - mid指向该区间的中点
  - 初始化条件：令low=1，high=n，mid=(low+high)/2
  - 将k与mid指向的记录进行关键字比较
    - 若k==R[mid]，查找成功，程序返回
    - 若k<R[mid]，则：high=mid-1
    - 若k>R[mid]，则：low=mid+1
  - 重复上述操作，直至low>high时结束，表示查找失败

折半查找算法

```
int binary_search(int R[], int tar, int n){
    int low = 1, high = n;
    while( low <= high ) {
        mid = (low+high)/2;
        if(tar == R[mid])
            return mid;
        else if ( tar > R[mid]) // 设数组从小到大有序
            low = mid+1;
        else high = mid-1;
    }
    return 0;  // 查找失败
}
```

折半查找算法性能分析



- 判定树：描述查找过程的二叉树叫判定树
- 有n个结点的判定树的深度为： $\log_2 n+1$

i	1	2	3	4	5	6	7	8	9	10	11
C_i	3	4	2	3	4	1	3	4	2	3	4

折半查找算法性能分析

- 表长为 n 的折半查找的判定树的深度
  - 和含有 n 个结点的完全二叉树的深度相同
- 折半查找法在查找过程中进行的比较次数
  - 最多不超过其判定树的深度
- 折半查找的ASL（设查找概率相等：$p_i = 1 / n$）
  - 以深度为h的满二叉树为例（即表长： $n=2^h-1$ ）

$$ASL = \frac{1}{n}\sum_{i=1}^{n} C_i = \frac{1}{n}\left[\sum_{k=1}^{h} k \times 2^{k-1}\right] = \frac{n+1}{n}\log_2(n+1) - 1$$

  - 当n值较大时（例如n>50），可近似认为：

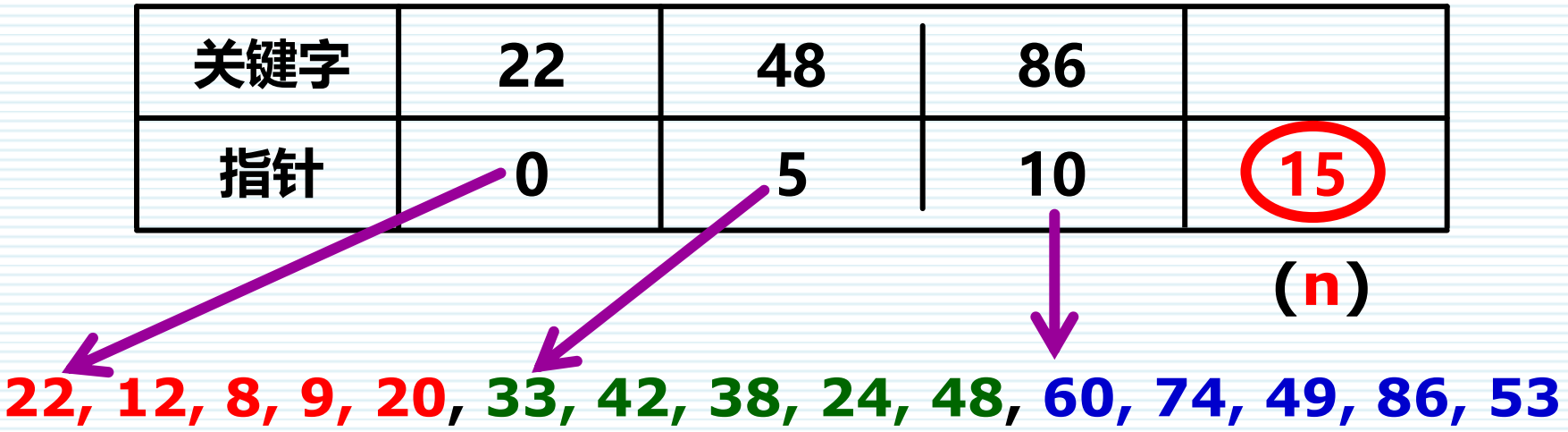
$$ASL_{\text{binary search}} \approx \log_2(n+1) - 1$$

索引表：索引查找

索引表的基本概念

- 使用索引能够快速地定位查找范围
  - 例如：书的目录就是一种索引
  - 计算机中对数据的存储和处理也可以采用索引
    - 当数据量太大以至于内存装不下时
    - 可以对数据建立“索引”
    - 当需要查询时，根据索引将所需要的数据块读入内存
    - 这样只需对读入的部分数据进行查询，提高查找效率

索引表的查找示例



- 给定查找表如下

  22, 12, 8, 9, 20, 33, 42, 38, 24, 48, 60, 74, 49, 86, 53

- 首先对其分块

  (22, 12, 8, 9, 20), (33, 42, 38, 24, 48), (60, 74, 49, 86, 53)

- 建立索引表：索引表项的构成：关键字+指针

索引表的构建

R =[22, 12, 8, 9, 20,
33, 42, 38, 24, 48,
60, 74, 49, 86, 53]

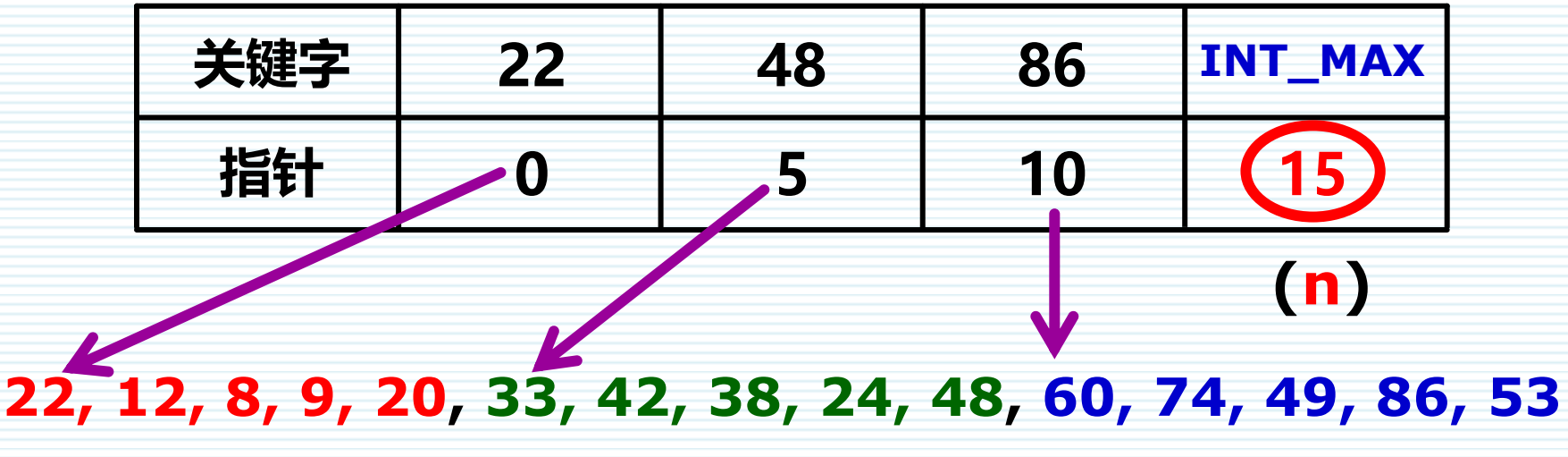
```
typedef struct{
        int max;
        int start;
}TBlock;
```

1. 分块：将查找表中数据按关键字分成若干块：R_1, …, R_m
   - 要求：使得“分块有序”；　(k=1, 2, …, m-1)
   - 即：第 R_k 块中所有关键字 < R_{k+1} 块中所有关键字
2. 建立索引项：对每一个块建立一个索引项
   - 每个索引项包含两项内容：
     - 关键字项：记载该块中最大关键字值
     - 指针项：　记载该块第一个记录在表中位置
3. 所有索引项组成索引表

索引表的查找

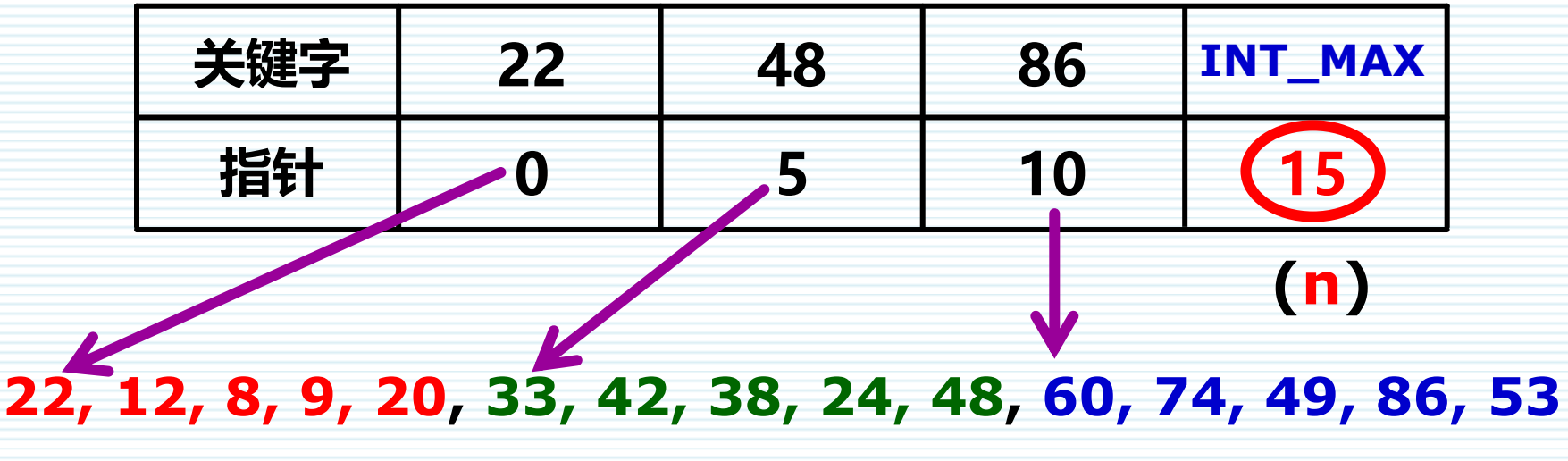
- 索引表的查找操作分两步进行
  - 在索引表上查找
    - 借助索引表确定待查记录所在区间
    - 由于索引表有序：可采用二分查找法
  - 在查找表上查找
    - 在查找表的某个区间内进行记录查找
- 索引表的查找算法流程
  - 将表分块（块内无序，块间有序）并建立索引表
  - 查找时首先确定待查记录所在块
  - 然后再在目标块内进行顺序查找

索引表的查找示例



- 查找关键字： **k = 38** **思考：怎样确定关键字所在数据块?**
  - 块间查找： **(22 < k < 48)**
    - **k = 38** 的数据应在第2块中查找
  - 块内查找： **思考：块内查找的起止点?**
    - 从 **R[5]** 开始, 直到 **R[i]==k** 或 **i > 9** 为止
    - 由于**R[10]==38**，查找成功

索引表的查找示例



- 查找关键字： **k = 50** **思考：怎样确定关键字所在数据块?**
  - 块间查找： **(48 < k < 86)**
    - **k = 50** 的数据应在第3块中查找
  - 块内查找： **思考：块内查找的起止点?**
    - 从 **R[10]** 开始, 直到 **R[14]** 为止
    - 由于整个块内均未出现匹配，查找失败

索引表的顺序查找算法

```
// 参数B为索引表，m为索引表长减一（索引块个数）
int block_search(int R[], TBlock B[], int m, int tar){
    int idx, k = 0;
    while( k < m && tar > B[k].max) ++k;   // 块间查找
    if( k == m ) return -1;   // 块间查找失败
    else{
        idx = B[k].start;  // 从块内第一个元素开始查找
        while( (R[idx] != tar) && ( idx < B[k+1].start) )
            idx++;            // 在块内顺序查找
        if( R[idx] == tar ) return idx; // 查找成功
        else return -1; // 查找失败
    }
}
```

分块查找算法评价

- 平均查找长度：$ASL_{bs} = L_b + L_w$
  - L_b：查找索引表（确定所在块）的平均查找长度s
  - L_w：在块中查找元素的平均查找长度
- 计算方法
  - 若将表长为n的表平均分成b块，每块含s个记录
  - 并设表中每个记录的查找概率相等，则：
    - 顺序查找索引表：

$$\mathrm{ASL}_{bs} = \frac{1}{b}\sum_{i=1}^{b} i + \frac{1}{s}\sum_{j=1}^{s} j = \frac{b+1}{2} + \frac{s+1}{2} = \frac{1}{2}\left(\frac{n}{s} + s\right) + 1$$

    - 折半查找索引表：

$$\mathrm{ASL}_{bs} \approx \log_2\left(\frac{n}{s} + 1\right) + \frac{s}{2}$$

三类查找方法比较

	顺序查找	折半查找	分块查找
ASL	最大	最小	两者之间
表结构要求	有序表 无序表	有序表	分块有序表
可用存储结构	顺序存储结构 线性链表	顺序存储结构	顺序存储结构 线性链表

2. 树上的查找

- 二叉排序树
- 平衡树
- B+树（自学）

二叉排序树

(Binary Search Tree)

二叉排序树

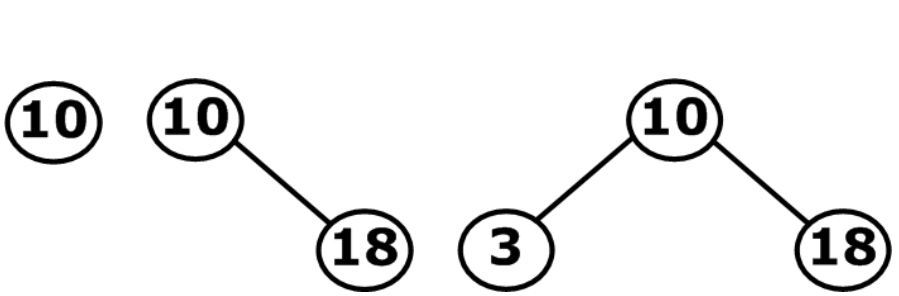
- 二叉排序树（Binary Search Tree）定义
  - 空树或具有下列性质的二叉树称为二叉排序树
  - 若根节点的左子树非空
    - 则左子树上所有结点的值均小于根结点的值
  - 若根节点的右子树非空
    - 则右子树上所有结点的值均大于或等于根结点的值
  - 左、右子树也分别为二叉排序树（递归定义！）
- 二叉排序树的性质
  - 按中序遍历该树所得到的中序序列是一个递增有序序列

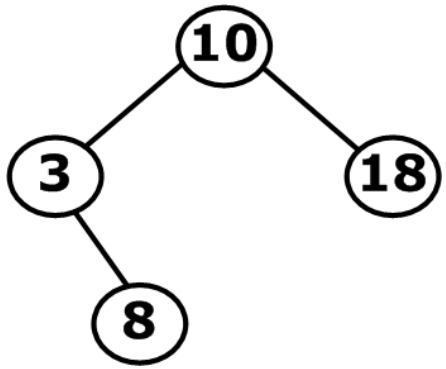
二叉排序树

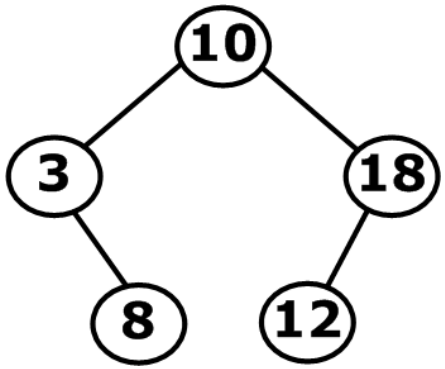
- 二叉排序树的结点插入操作
  - 若二叉排序树为空，则插入结点应为新的根结点
  - 否则根据排序树的性质继续在其左、右子树上查找
    - 直至某个结点的左子树或右子树为空为止
    - 则插入结点应为该结点的左孩子或右孩子
- 生成二叉排序树
  - 从空树出发，将给定序列中的元素逐一插入到BST中
  - 经过一系列查找、插入操作之后，可生成一棵二叉排序树

二叉排序树的结点插入操作（生成二叉排序树）

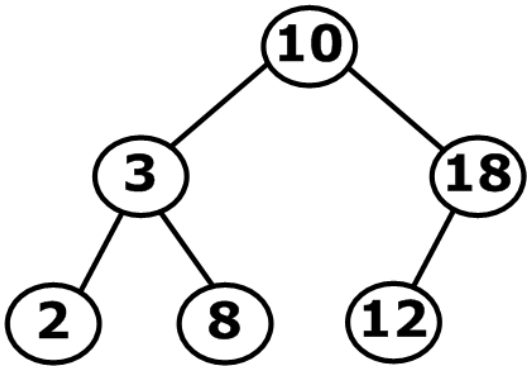
例：{ 10，18，3，8，12，2，7，3 }

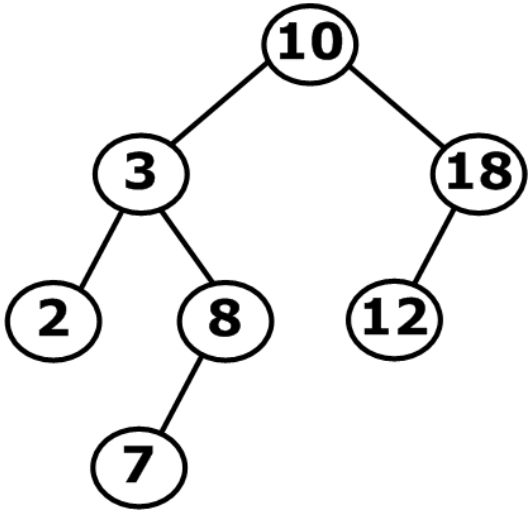


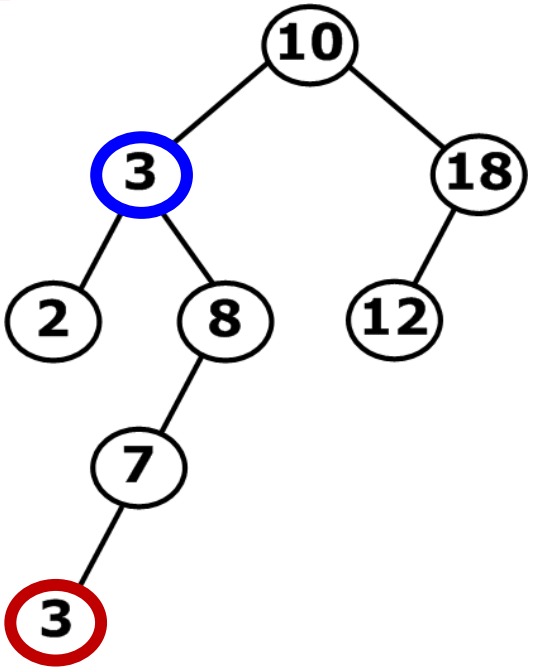




中序遍历二叉排序树可得到一个有序序列！







中序遍历：2 3 3 7 8 10 12 18

二叉排序树的存储结构

- 二叉排序树的存储结构：二叉链表

```
Typedf struct node{
        ElemType key;
        node *lchild;
        node *rchild;
}TNode, *PNode;
```

二叉排序树： 查找指定结点

```
PNode bst_search (PNode p,  ElemType e) {
    while( p ){
        if (e == p->key) return p;
        else if (e < p->key)
            p = p->lchild;
        else p = p->rchild;
    }
    return p;
}
```

二叉排序树： 查找结点e的插入位置

```
PNode bst_search (PNode p,  ElemType e) {
    Pnode q;  // q指向结点e在bst中的父节点
    while( p ){
        if (e < p->key) {
            q = p; p = p->lchild;
        }
        else {
            q = p; p = p->rchild;
        }
     }
     return q;
}
```

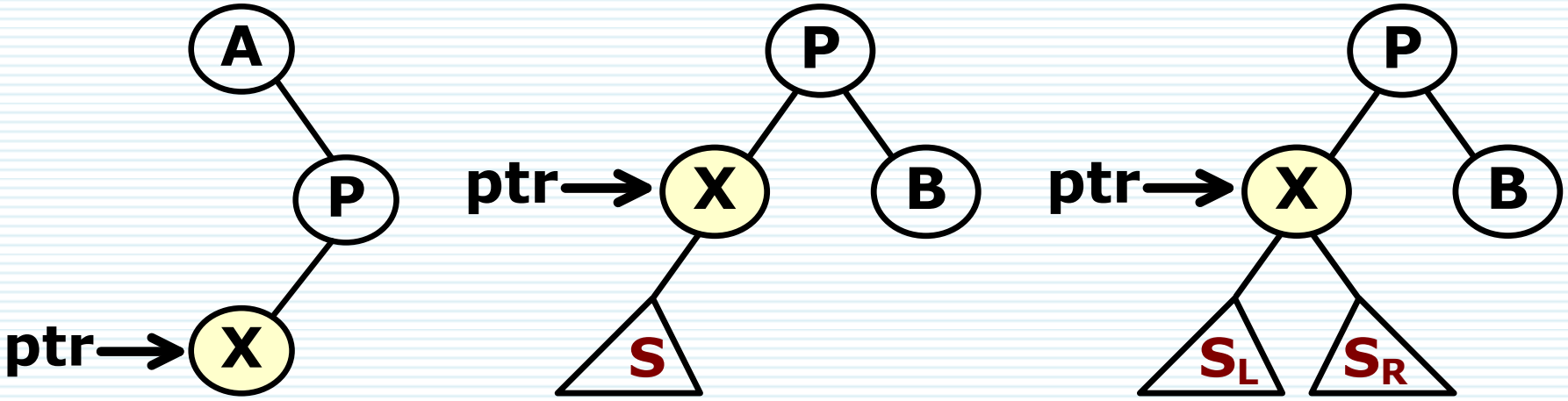
二叉排序树：插入给定结点

```
void bst_insert (PNode *pp,  ElemType e) {
    PNode pnode, p, q;  p = *pp;
    pnode = (PNode)malloc(sizeof(TNode));
    pnode->key = e;
    pnode->lchild = pnode->rchild = NULL;
    if( !p ){ *pp = pnode;  return;} // 空树
    q = bst_search(p, e);  // q 指向 e 在bst中的父节点
    if(e < q->key )
        q->lchild = pnode;
    else
        q->rchild = pnode;
    return;
}
```

生成二叉排序树

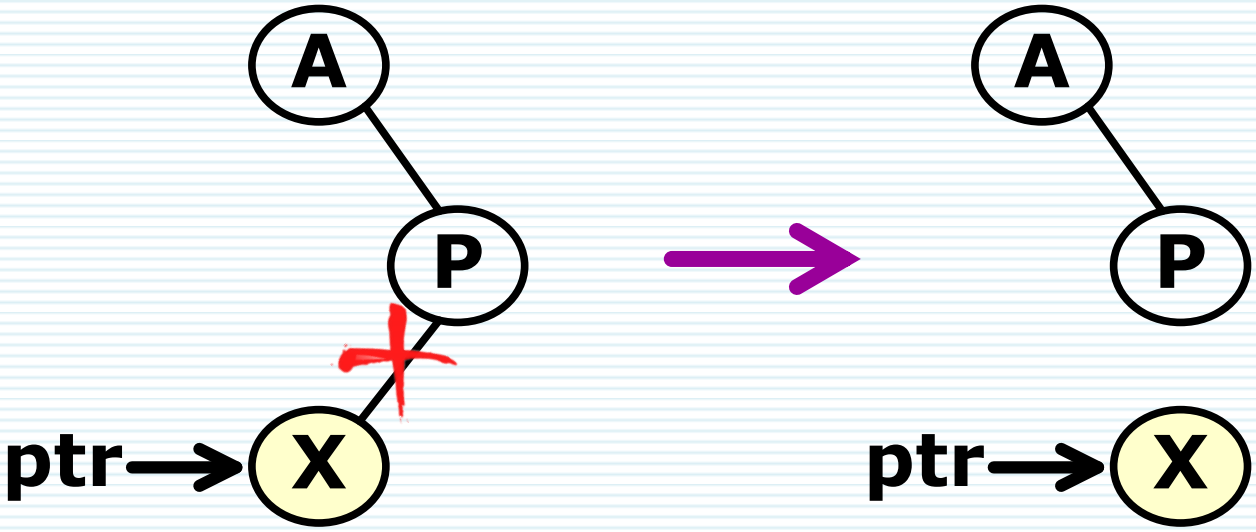
```
void bst_create (ElemType *R, int n) {
    int i = 0;
    PNode * p = NULL;
    for(i = 0; i < n; i++){
        bst_insert(p, R[i]);
    }
    return p;
}
```

二叉排序树的结点删除操作



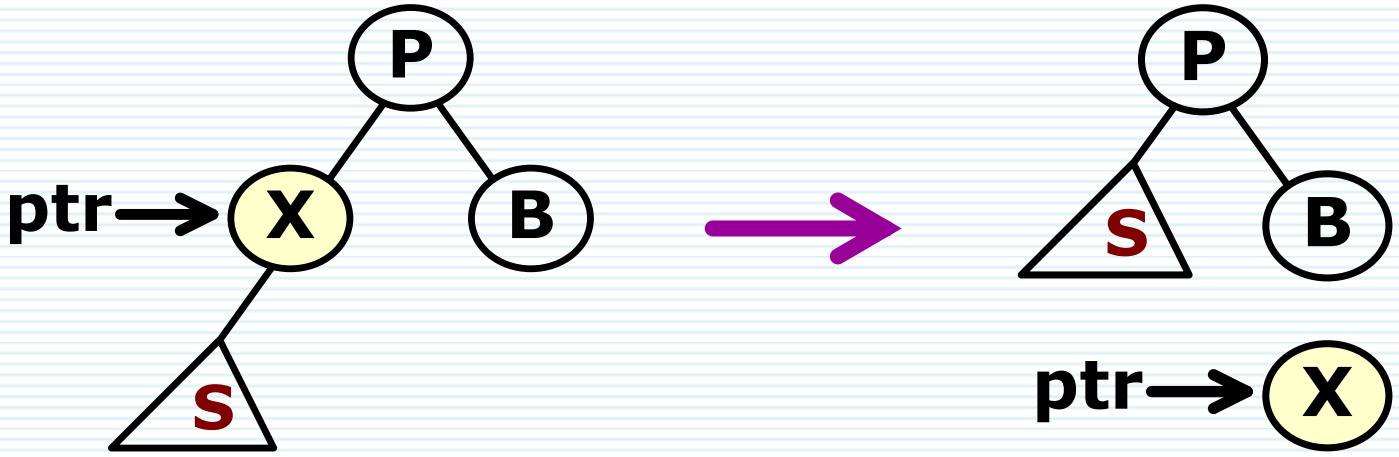
- 要删除二叉排序树中指针ptr指向的结点，可能遇到哪几种情况？
  - (*ptr)为叶结点
  - (*ptr)为单分支结点：只有左子树或只有右子树
  - (*ptr)为双分支结点：左、右子树均非空

二叉排序树的结点删除操作



- 若：(*ptr)为叶结点
  - 只需修改(*ptr)的双亲(*par)的孩子指针
  - 本例中： (*ptr)为结点X，(*par)为结点P
    - par->lchild = NULL 或 par->rchild = NULL

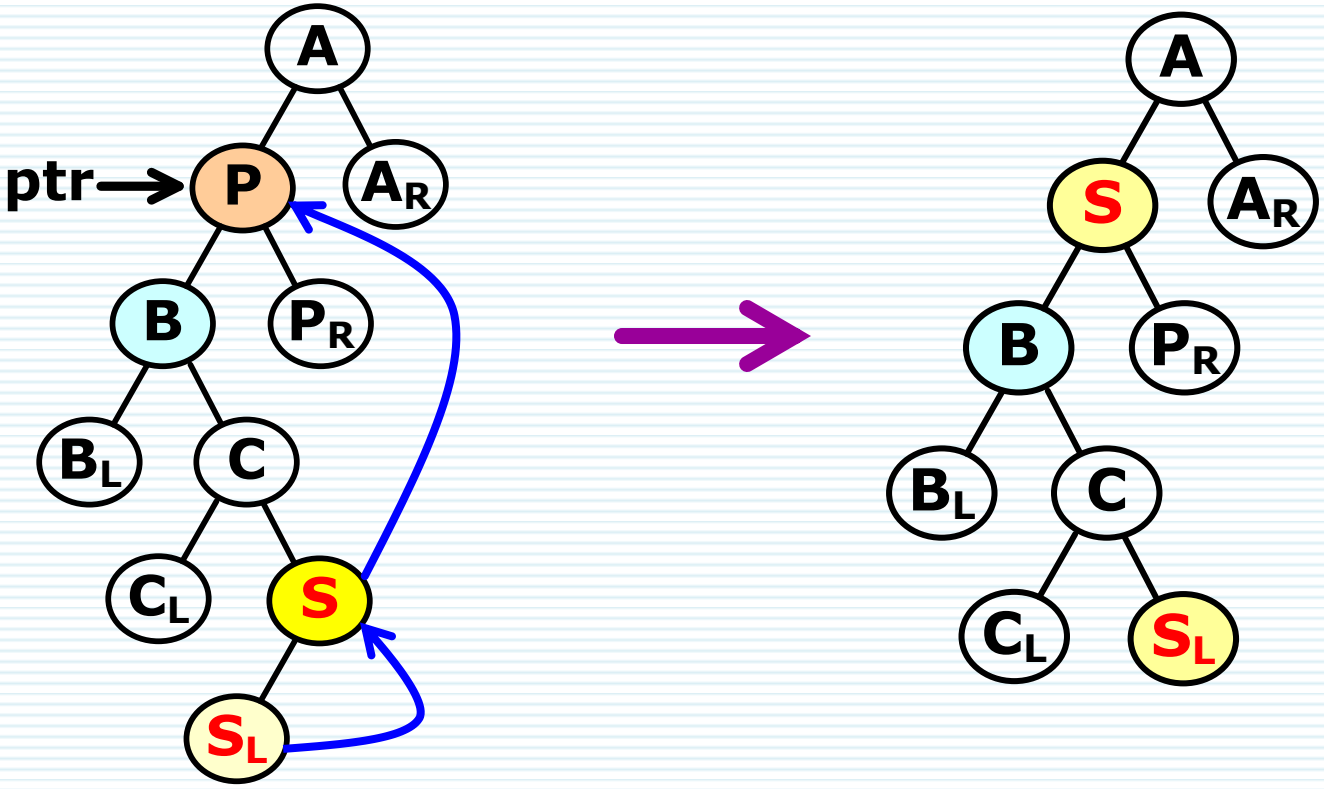
二叉排序树的结点删除操作



中序遍历：S X P B → S P B

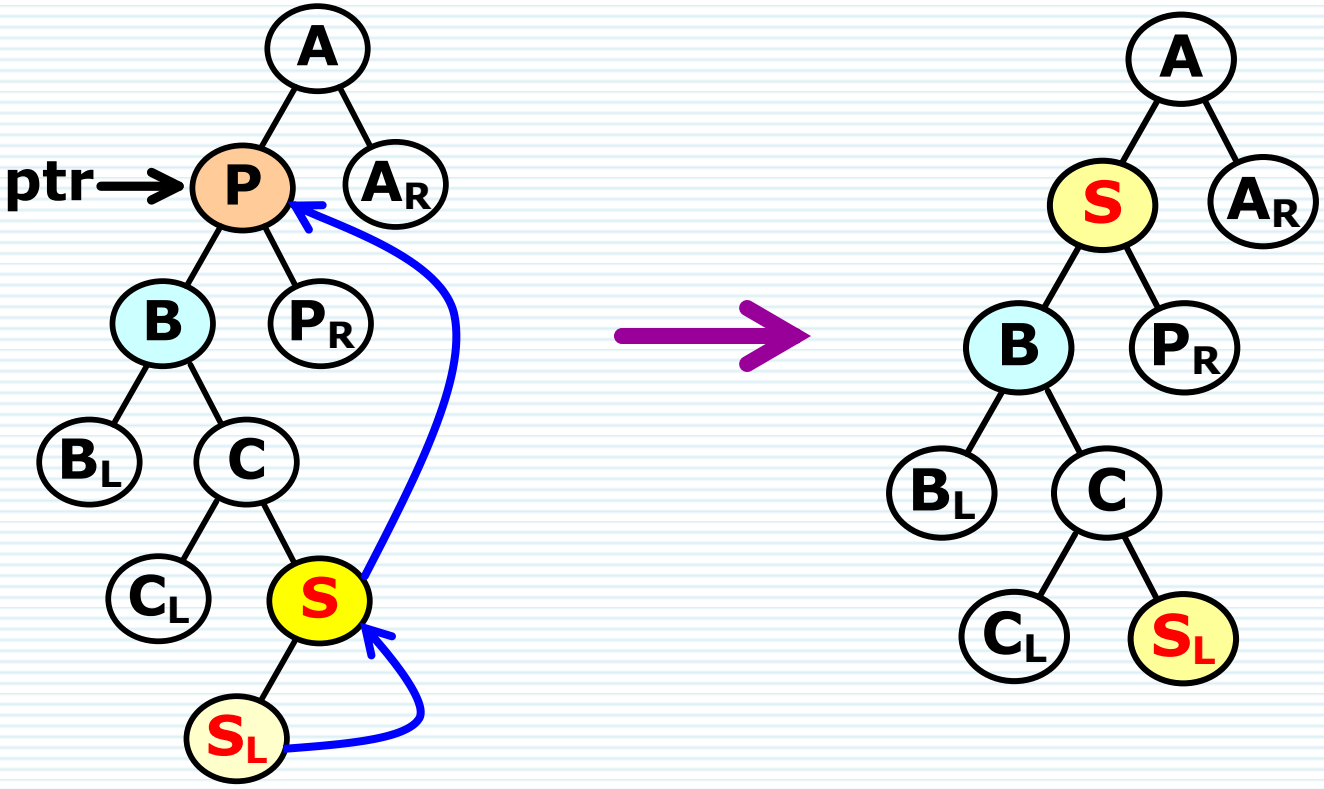
- 若：(*ptr)为单分支结点：只有左子树或右子树
  - 若(*ptr)只有左子树，用(*ptr)的左孩子代替(*ptr)
  - 若(*ptr)只有右子树，用(*ptr)的右孩子代替(*ptr)

二叉排序树的结点删除操作



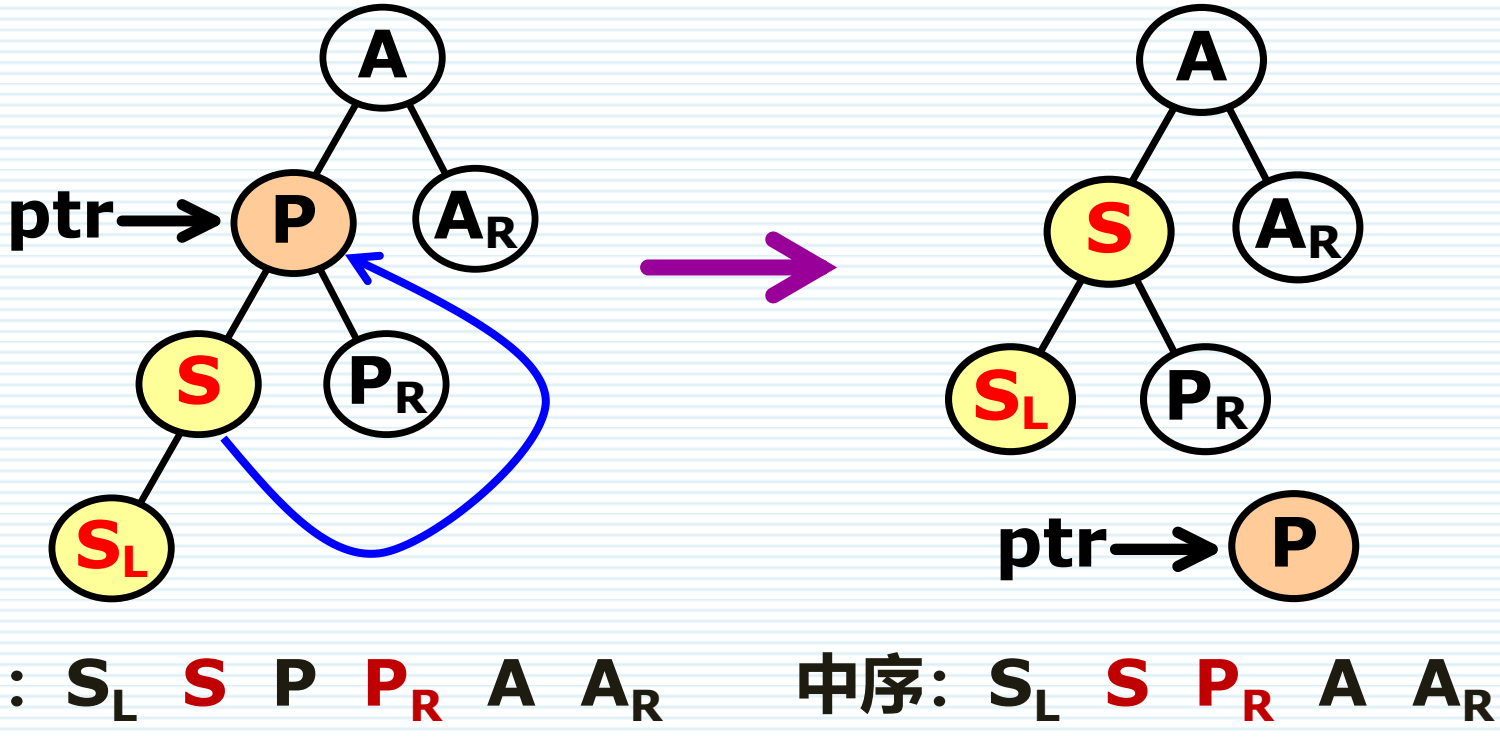
- 若：(*ptr)为2分支结点（左、右子树均非空）
  - 沿**ptr->lchild**的右子树向下查找**(*ptr)**的前驱结点**S**
    - 查找依据：**S->rchild==NULL**
  - 以**S->lchild**取代**S**，以**S**取代（***ptr**）

二叉排序树的结点删除操作



- 中序遍历（左）：B_L B C_L C S_L S P P_R A A_R
- 中序遍历（右）：B_L B C_L C S_L S P_R A A_R
- 问题：如果 **ptr->lchild->rchild == NULL** 怎么办？

二叉排序树的结点删除操作



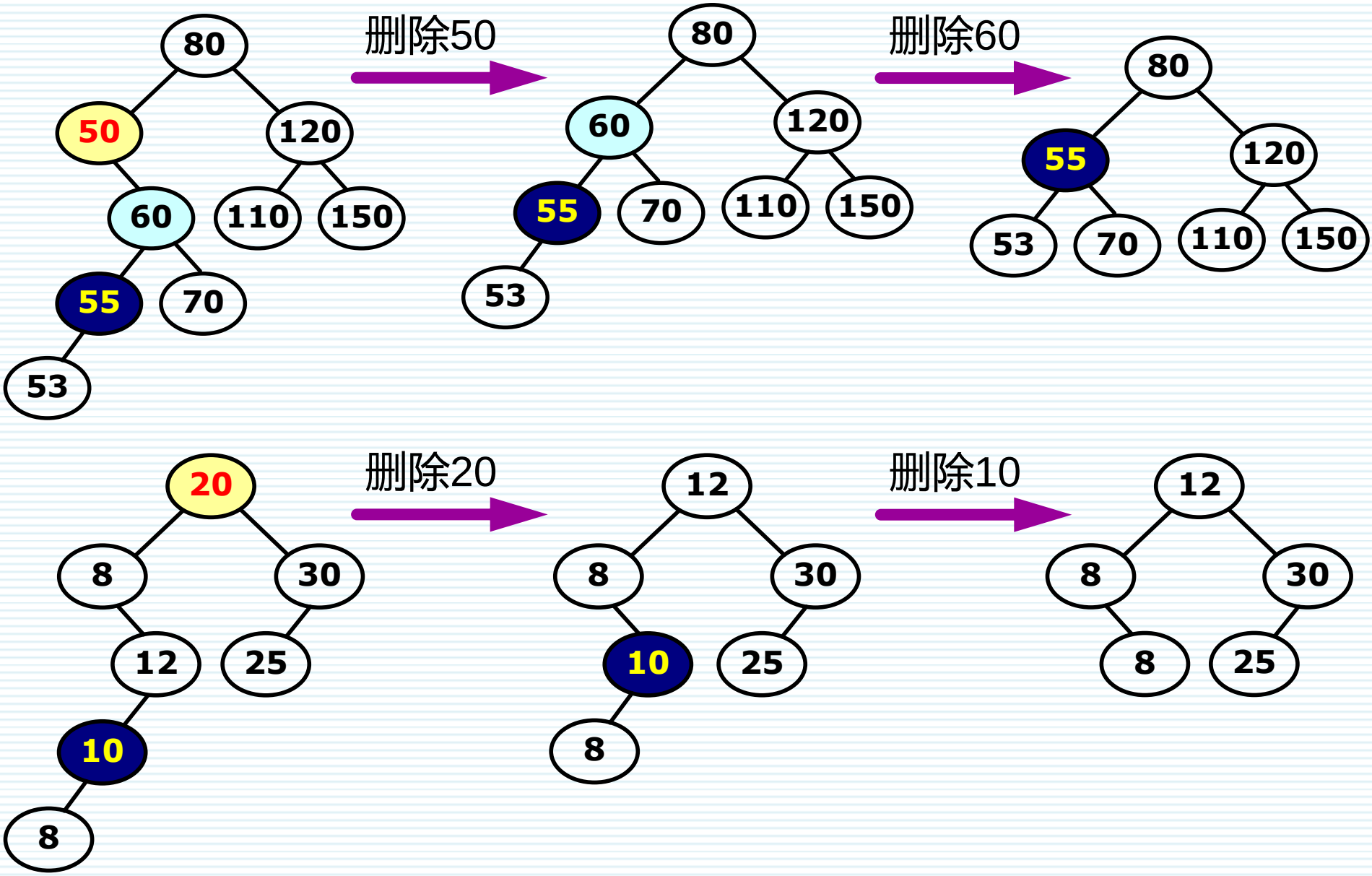
中序：S_L S P P_R A A_R　　中序：S_L S P_R A A_R

- 若：(*ptr)为2分支结点（左、右子树均非空）
  - 且：ptr->lchild->rchild == NULL
  - 以 **ptr->lchild** 取代 **(*ptr)** 即可

小结：二叉排序树的结点删除操作

- 要删除二叉排序树中的(*ptr)结点
  - 若(*ptr)为叶结点：只需修改(*ptr)双亲的孩子指针为空
    - par->lchild=NULL (或 par->rchild=NULL)
  - 若(*ptr)只有左子树或右子树
    - 若(*ptr)只有左子树，用(*ptr)的左孩子代替(*ptr)
    - 若(*ptr)只有右子树，用(*ptr)的右孩子代替(*ptr)
  - 若(*ptr)的左、右子树均非空
    - 若ptr->lchild的右子树为空：以ptr->lchild取代(*ptr)
    - 否则沿ptr->lchild的右子树向下查找(*ptr)的前驱S
      - 以S->lchild 取代 S，以前驱结点 S 取代(*ptr)
- 最后释放掉(*ptr)结点所占空间：**free(ptr);**

二叉排序树的结点删除操作



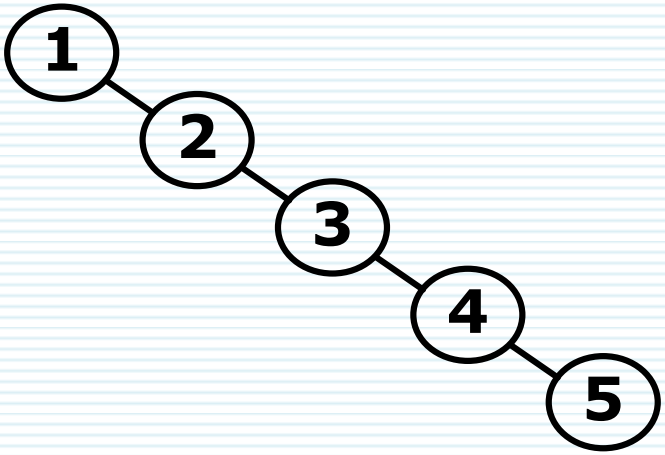
二叉排序树的查找性能分析

- 平均查找长度（Average Search Length，ASL）
  - 为确定记录在查找表中的位置，需和给定值进行比较的关键字个数的期望值称为查找算法在查找成功时的平均查找长度
  - 对于含有n个数据元素的查找表，查找成功的平均查找长度为：

$$\mathrm{ASL} = \sum_{i=1}^{n} P_i C_i \quad (i=1,2,3,\ldots,n)$$

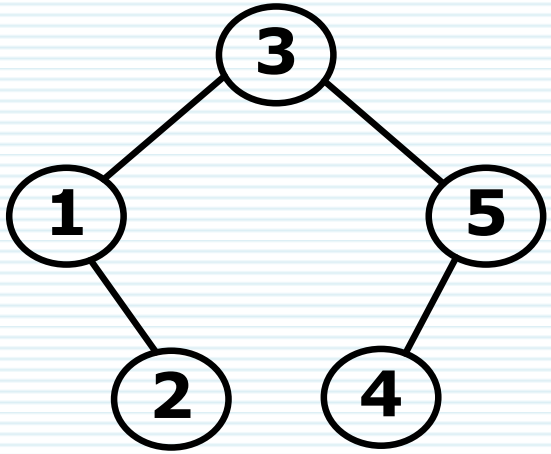
  - 其中：P_i 为查找表中第 i 个数据元素的概率
  - C_i 为找到第 i 个数据元素时已经比较过的次数
  - 在查找表中查找不到待查元素，但是找到待查元素应该在表中存在的位置的平均查找次数称为查找不成功时的平均查找长度
  - 对于任意一棵特定的二叉排序树均可按照定义求得ASL值

二叉排序树的查找性能分析



解决方案

平衡二叉树



- n 个关键字的不同排列可构造出不同形态的多棵二叉排序树
  - 由于查找顺序不同，其平均查找长度值可能差别很大
  - 由关键字序列 **1，2，3，4，5** 构造的二叉排序树
    - ASL = (1+2+3+4+5) / 5 = **3** (等概率查找)
  - 由关键字序列 **3，1，2，5，4** 构造的二叉排序树
    - ASL = (1+2+3+2+3) / 5 = **2.2** (等概率查找)

平衡二叉树（自学）

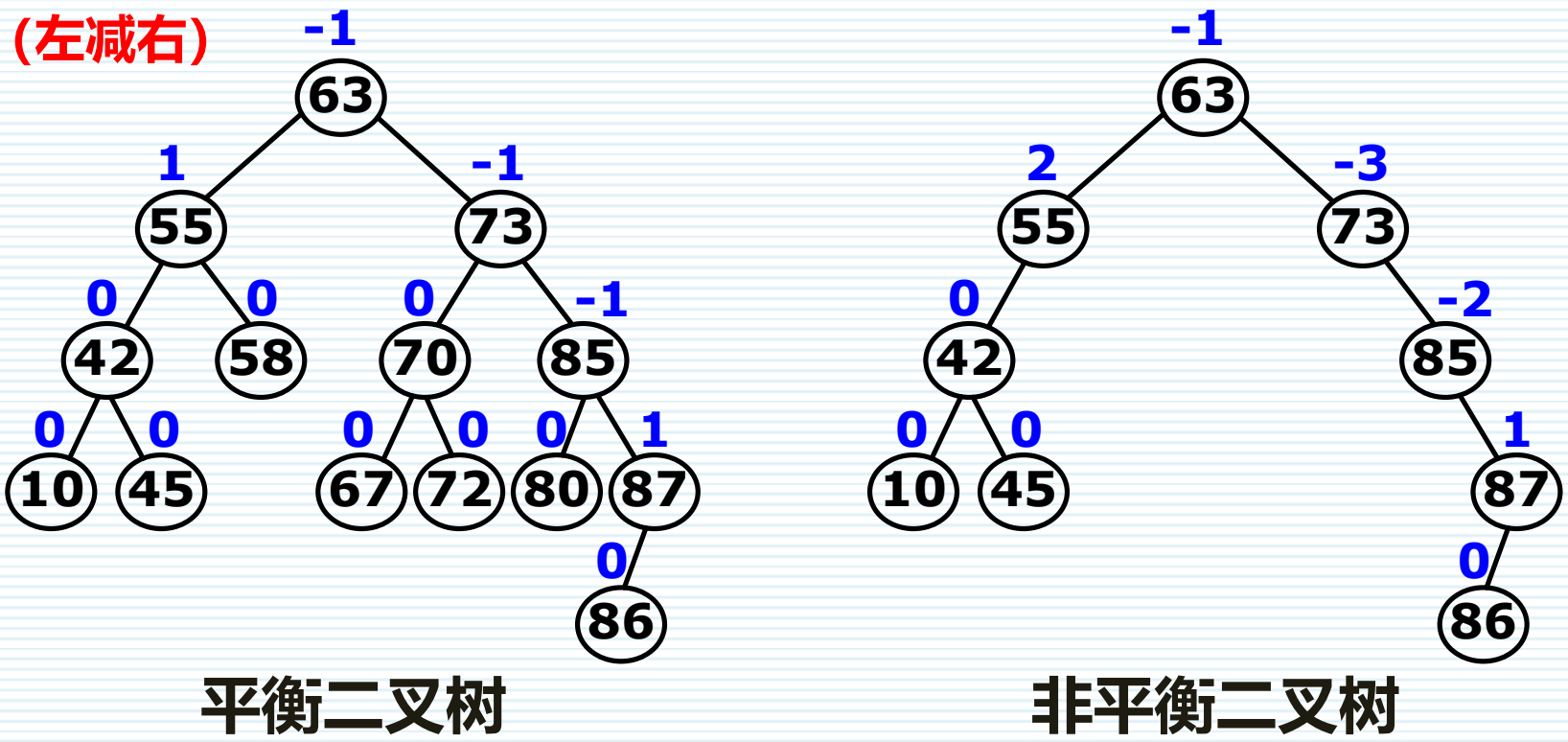
平衡二叉树（AVL树）

- 平衡二叉树：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
  - 并且左、右子树的深度之差不超过1

- 背景知识：**AVL树（Adelson-Velskii and Landis）**

G. Adelson-Velskii; E. M. Landis (1962). "**An algorithm for the organization of information**". Proceedings of the USSR Academy of Sciences 146: 263–266. (Russian) English translation by Myron J. Ricci in Soviet Math. Doklady, 3:1259–1263, 1962.

平衡二叉树（AVL树）



平衡二叉树　　**非平衡二叉树**

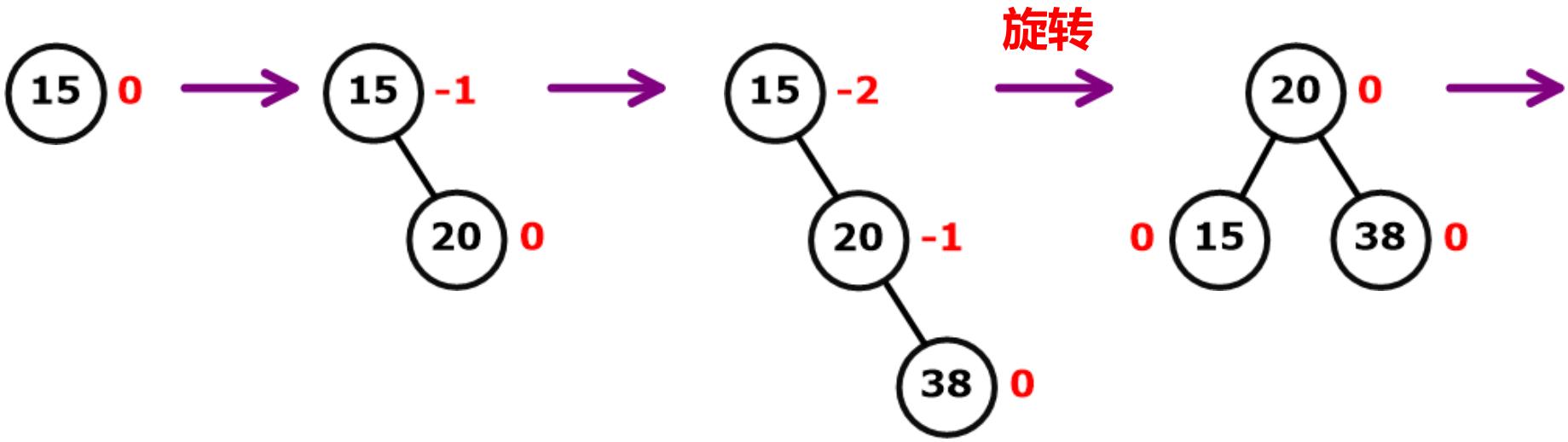
- 平衡二叉树左、右子树的深度之差不超过1
- 结点的平衡因子(balance factor)
  - 结点的**左**子树的深度减去**右**子树的深度的值

平衡二叉树的存储结构

- 平衡二叉树的存储结构：二叉链表

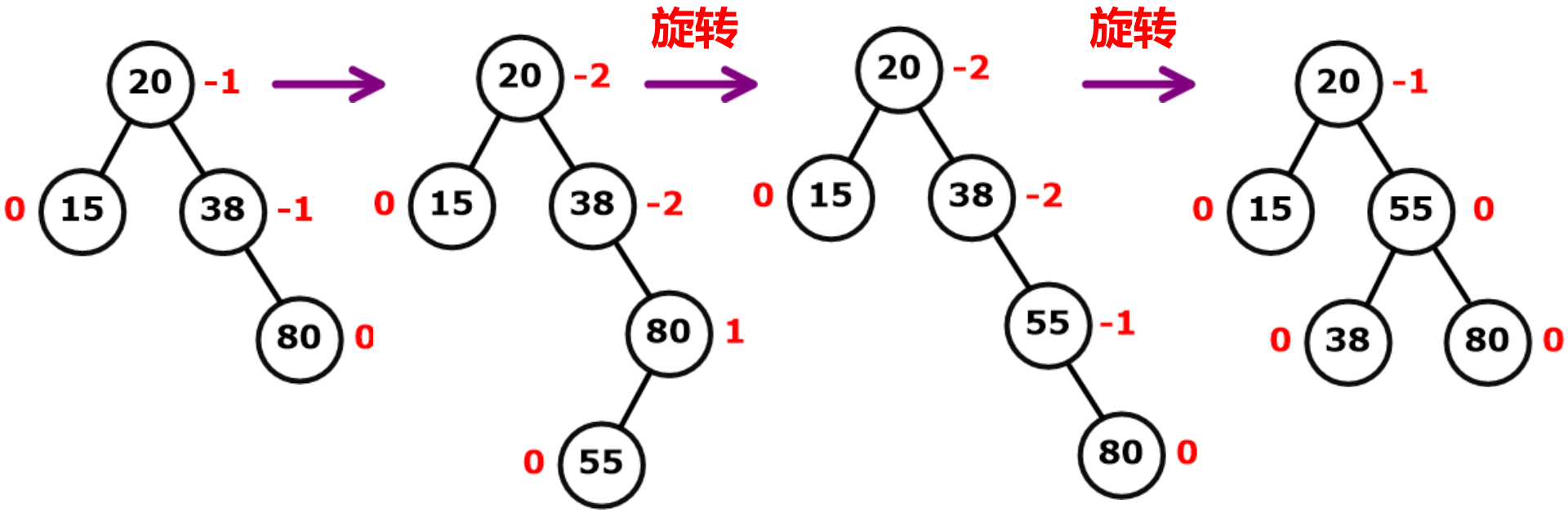
```
Typedf struct node{
        int bf;    // balance factor
        ElemType key;
        node *lchild;
        node *rchild;
}AVLNode;
```

旋转操作



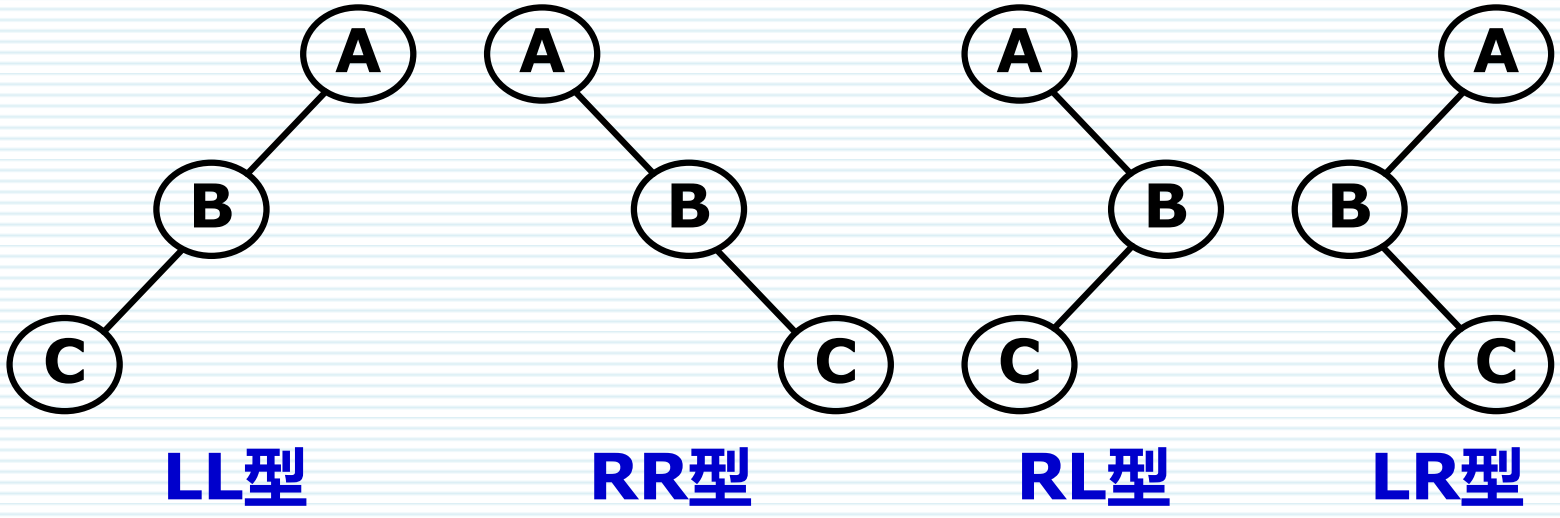
- 旋转（rotation）
  - 对AVL树进行简单修改以保证其平衡的操作
  - 例如：用序列 {15,20,38,80,55} 构建AVL树

旋转操作



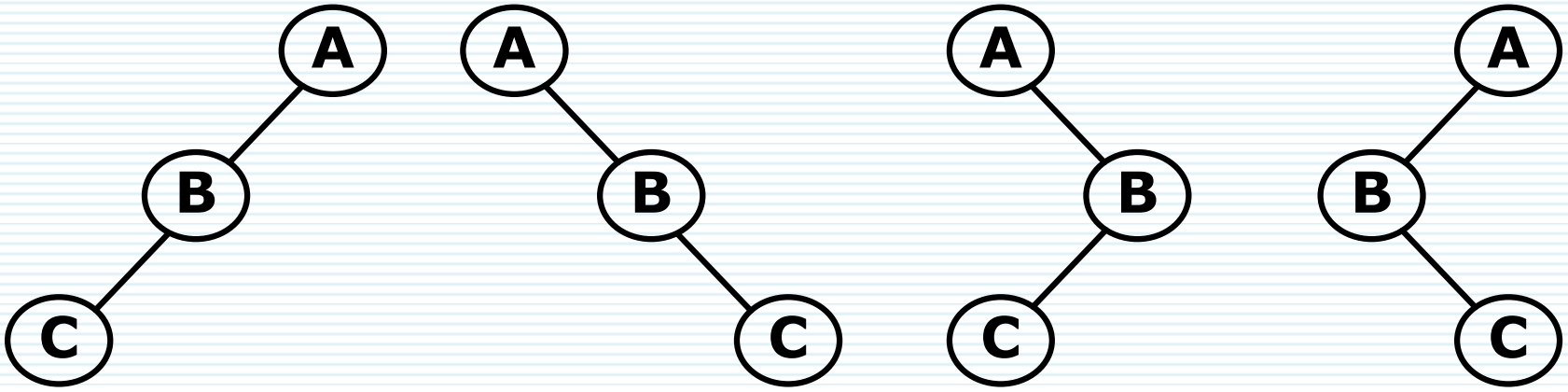
- 旋转（rotation）
  - 对AVL树进行简单修改以保证其平衡的操作
  - 例如：用序列 {15,20,38,80,55} 构建AVL树

平衡二叉树的失衡调整



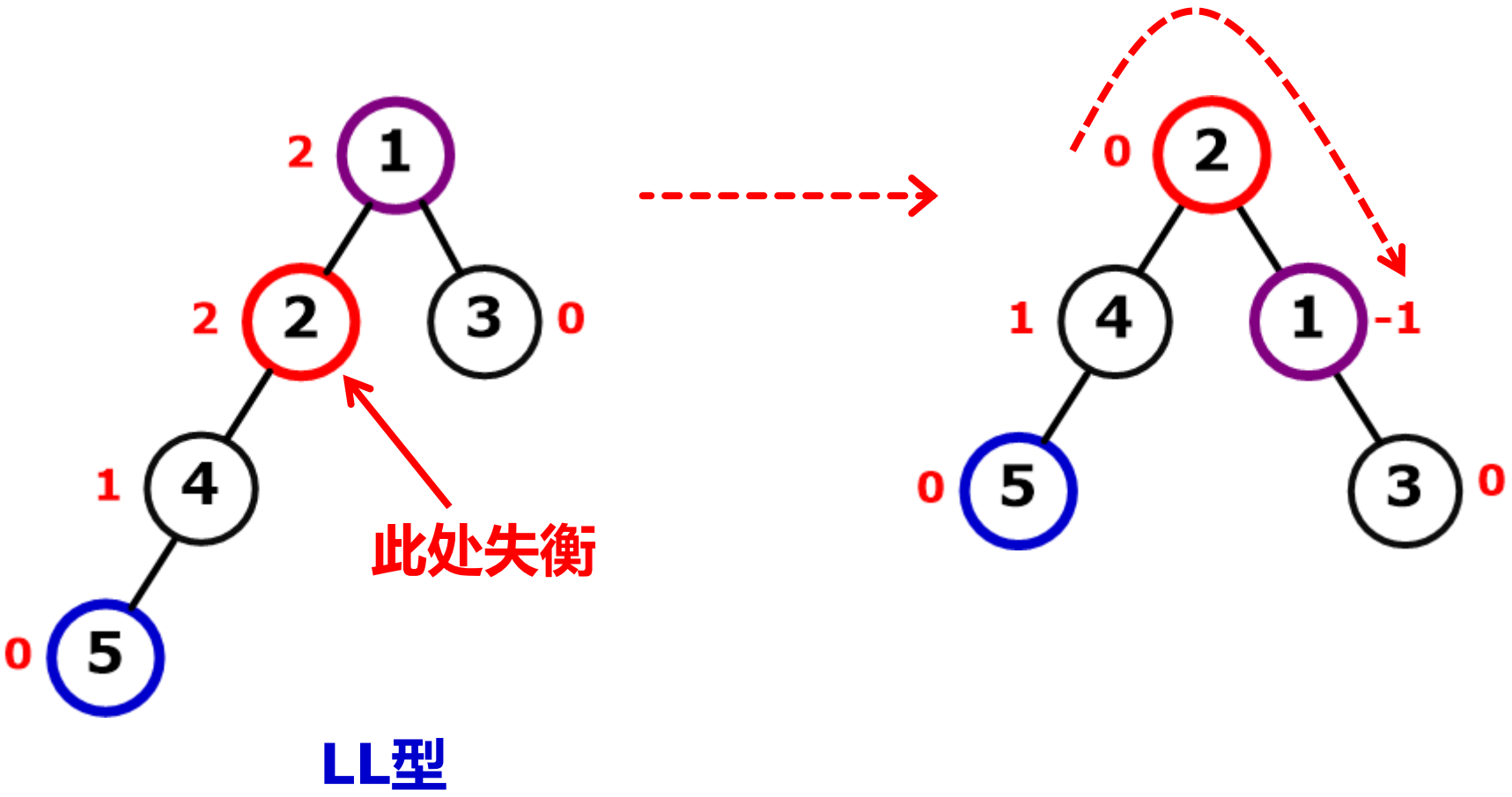
- 导致结点失衡的情况
  - LL型：对a的左孩子的左子树进行一次插入操作
  - LR型：对a的左孩子的右子树进行一次插入操作
  - RL型：对a的右孩子的左子树进行一次插入操作
  - RR型：对a的右孩子的右子树进行一次插入操作

平衡二叉树的失衡调整

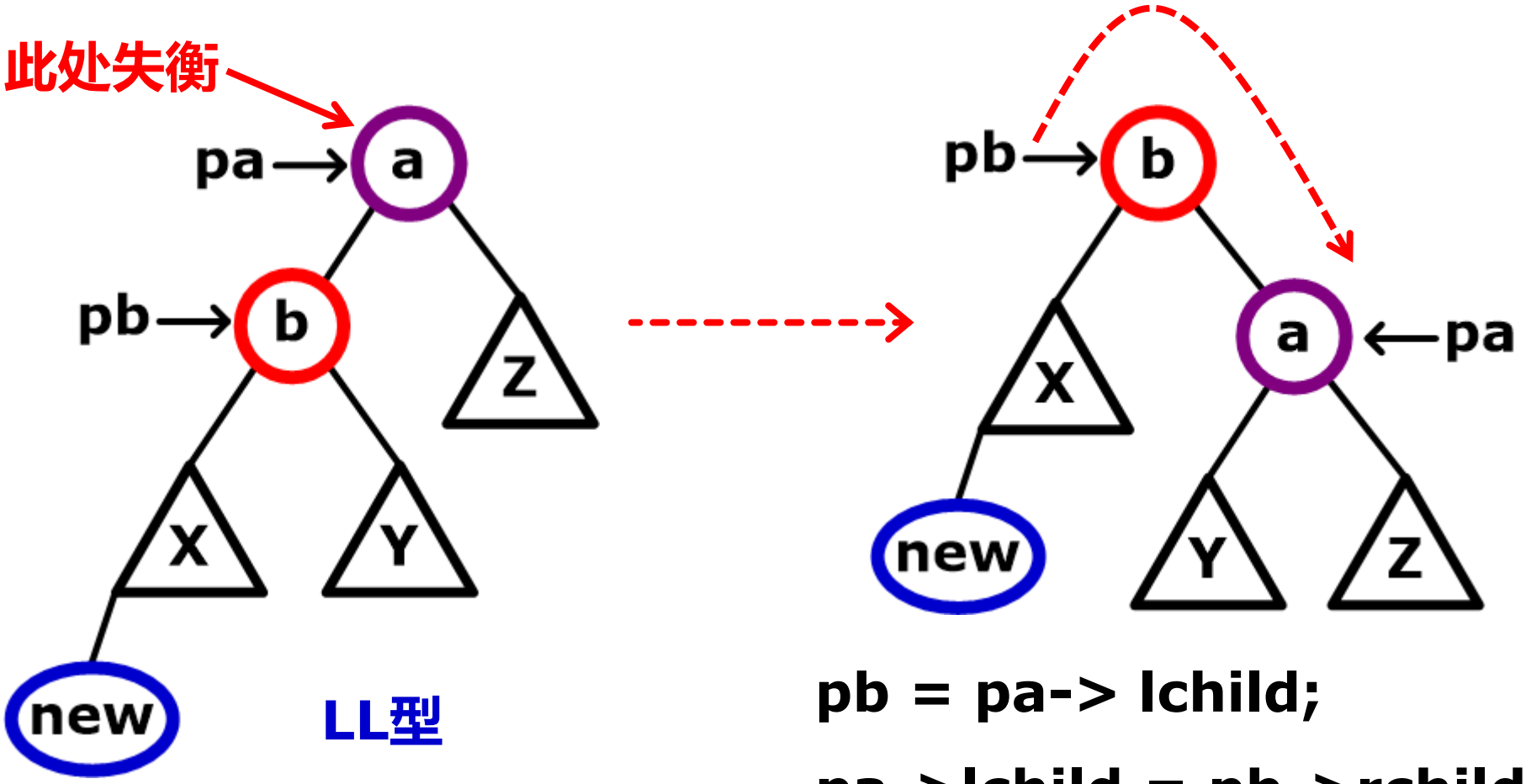


- 两类旋转平衡化处理方法
  - 单旋转：一次旋转达到平衡
    - 单向右旋（**LL**）和 单向左旋（**RR**）
  - 双旋转：两次旋转达到平衡
    - 先左后右（**LR**）和 先右后左（**RL**）

单旋转：LL型右旋（顺时针旋转）

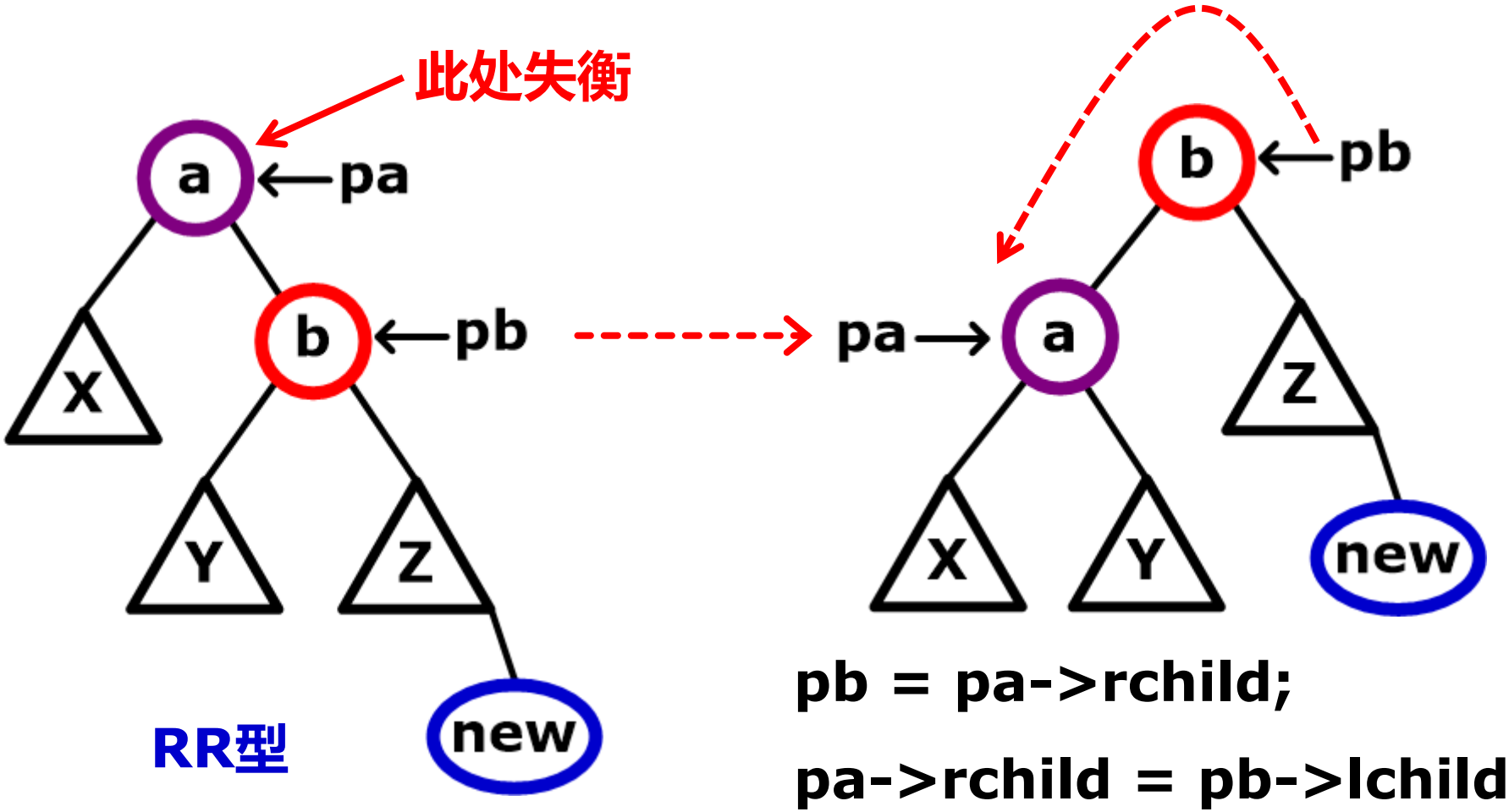


单旋转：LL型右旋（顺时针旋转）



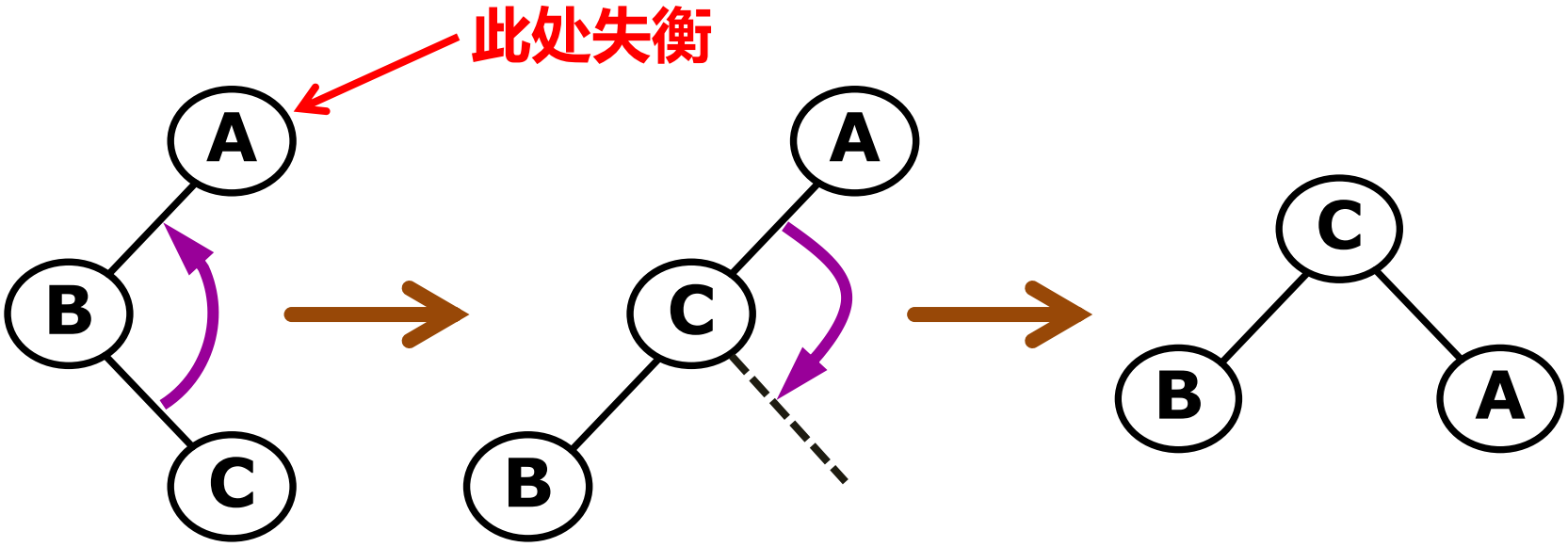

```
pb = pa-> lchild;
pa->lchild = pb->rchild;
pb->rchild = pa;
pa->bf = 0; pb->bf = 0;
```

单旋转：RR型左旋（逆时针旋转）




```
pb = pa->rchild;
pa->rchild = pb->lchild;
pb->lchild = pa;
pa->bf = 0; pb->bf = 0;
```

双旋转：LR型（先左旋，再右旋）



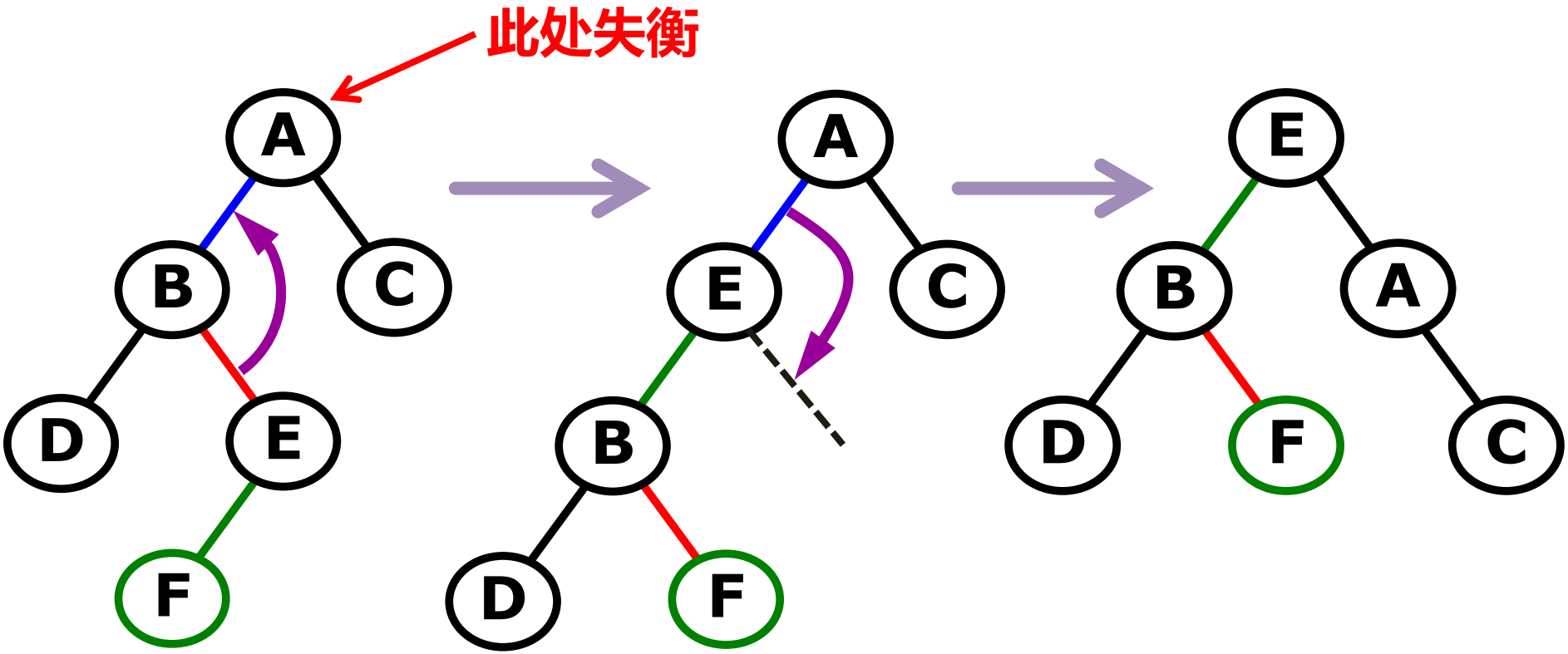
pb->rchild = pc->lchild;

pc->lchild = pb;

pa->lchild = pc->rchild;

pc->rchild = pa;

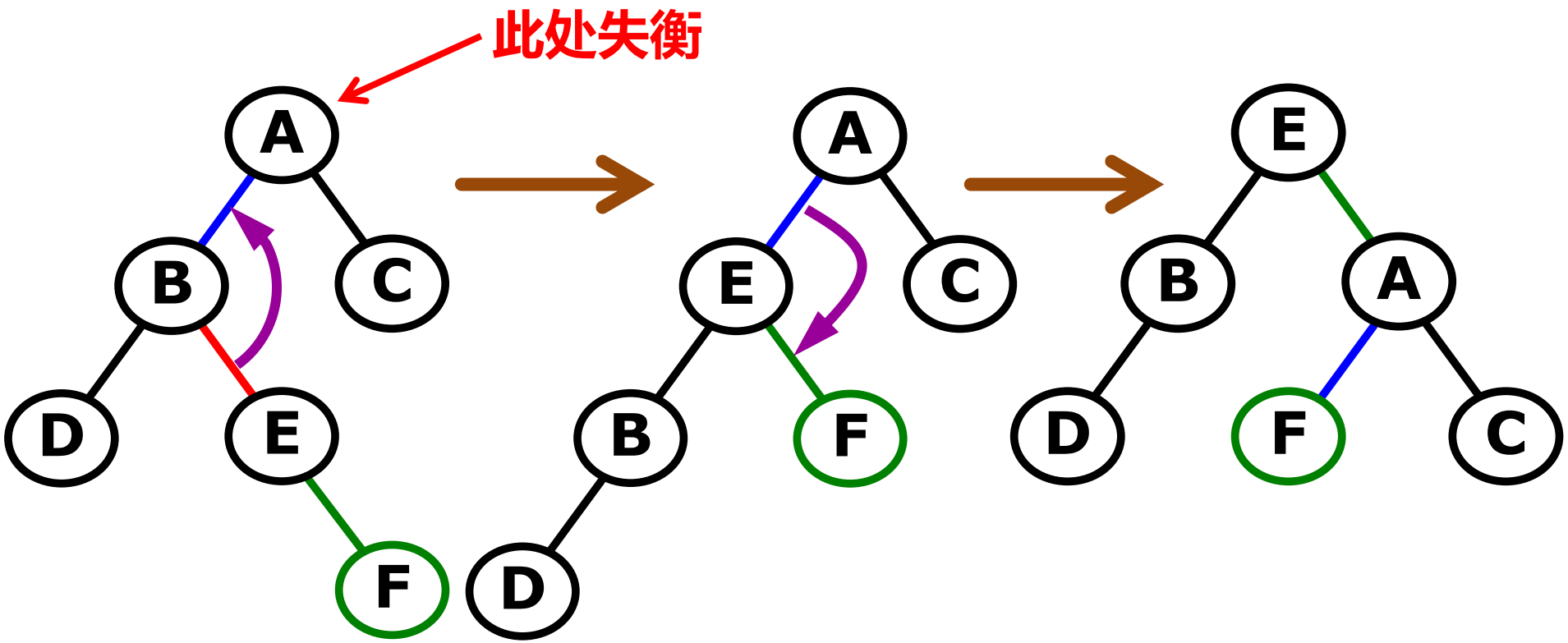
双旋转：LR型（先左旋，再右旋）




```
pb->rchild = pe->lchild;
pe->lchild = pb;
```

```
pa->lchild = pe->rchild;
pe->rchild = pa;
```

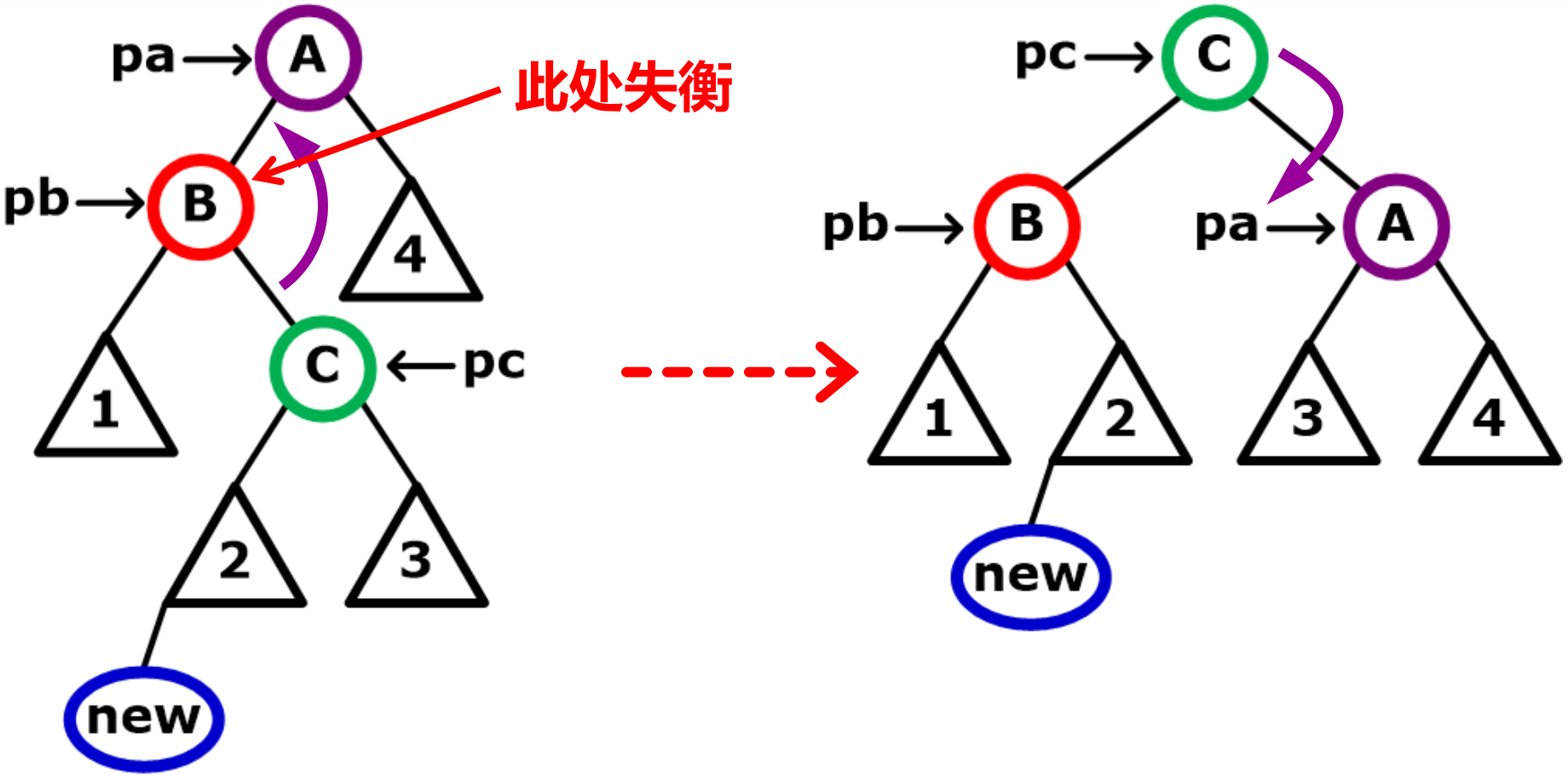
双旋转：LR型（先左旋，再右旋）




```
pb->rchild = pe->lchild;
pe->lchild = pb;
```

```
pa->lchild = pe->rchild;
pe->rchild = pa;
```

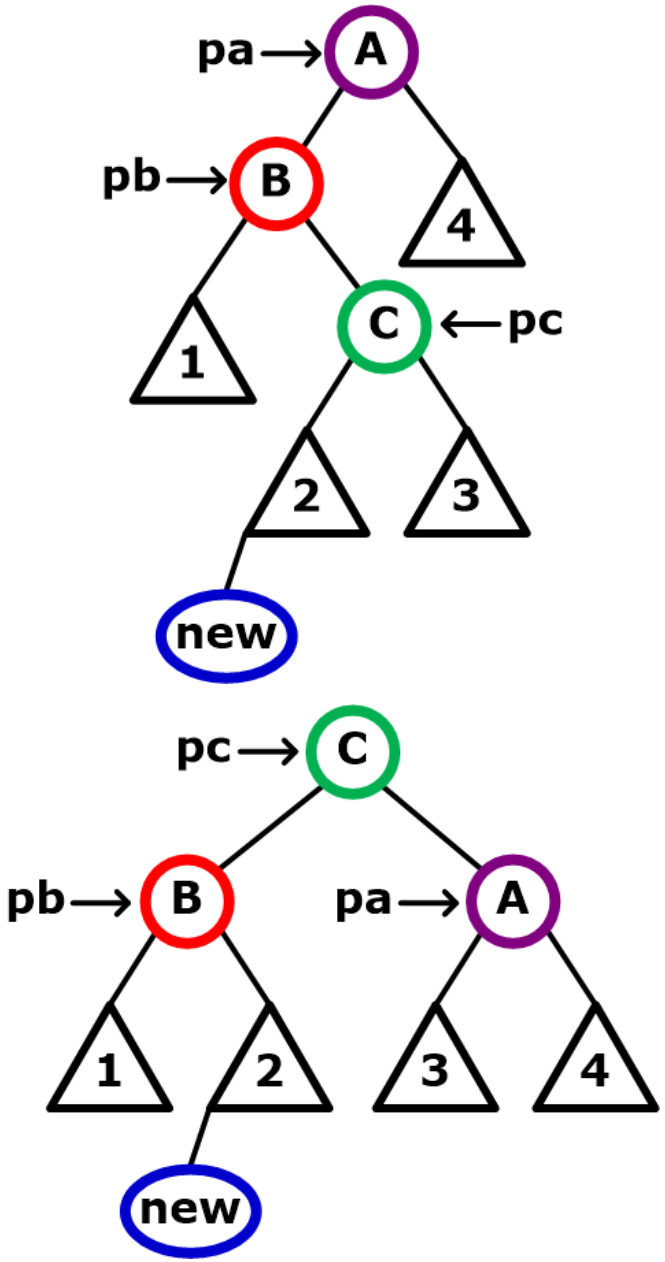
双旋转：LR型（先左旋，再右旋）




```
pb->rchild = pc->lchild;
pc->lchild = pb;
```

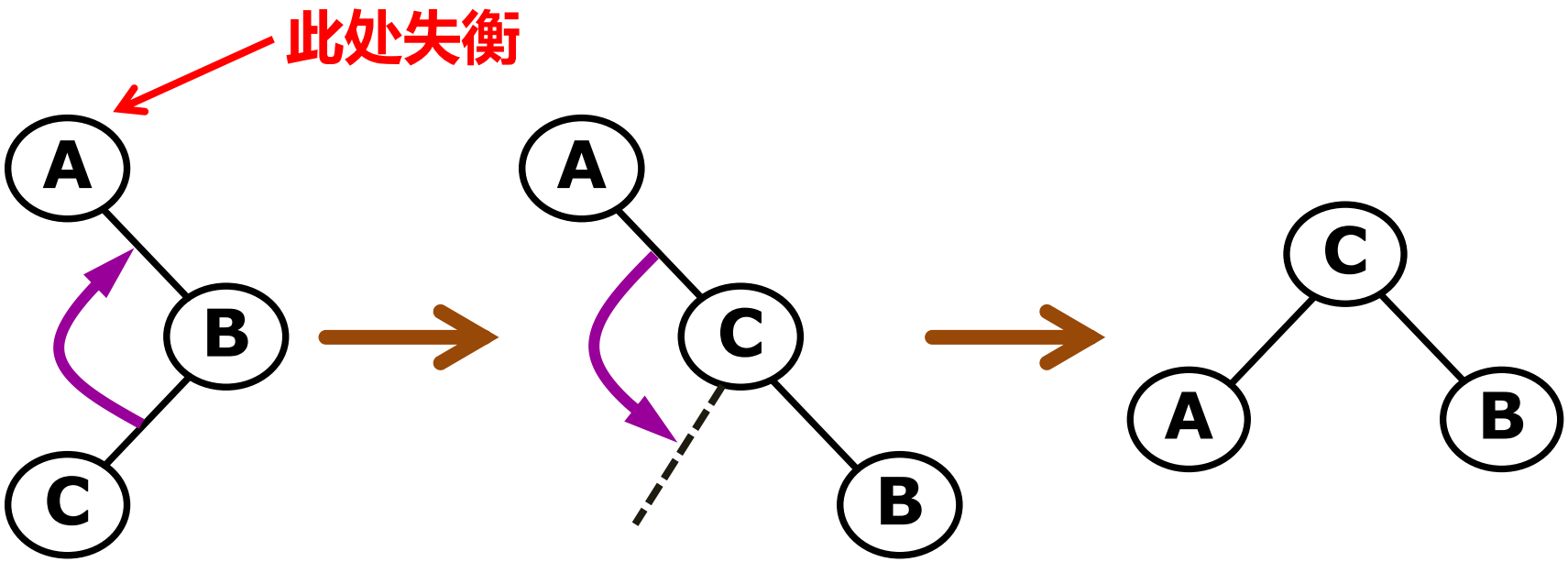
```
pa->lchild = pc->rchild;
pc->rchild = pa;
```

双旋转：LR型（先左旋，再右旋）




```
pb = pa->lchild;
pc = pb->rchild;
pb->rchild = pc->lchild;
pa->lchild = pc->rchild;
pc->lchild = pb;
pc->rchild = pa;
switch(pc->bf){
   case  1: pa->bf = -1;
            pb->bf = 0; break;
   case -1: pa->bf = 0;
            pb->bf = -1; break;
   case  0: pa->bf = 0;
            pb->bf = 0; break;
}
```

双旋转：RL型（先右旋，再左旋）



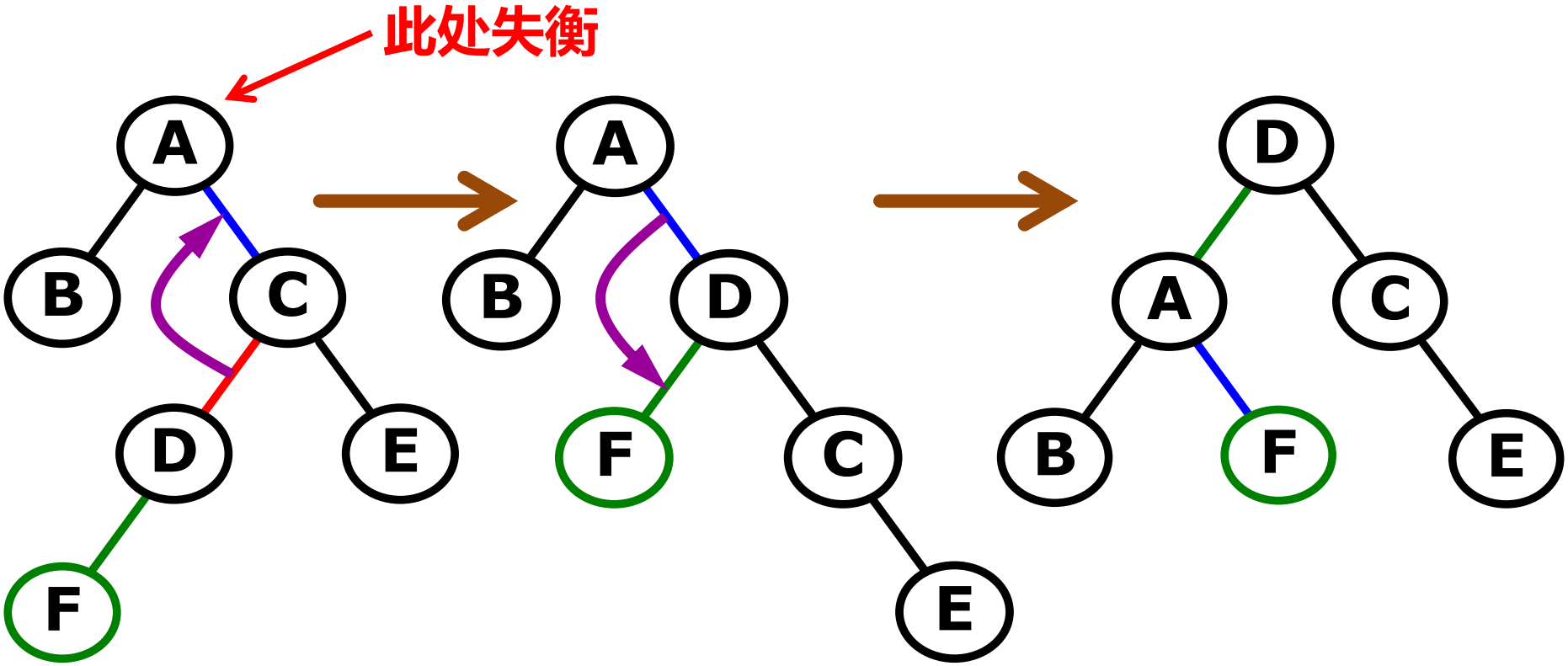
pb->lchild = pc->rchild;

pc->rchild = pb;

pa->rchild = pc->lchild;

pc->lchild = pa;

双旋转：RL型（先右旋，再左旋）



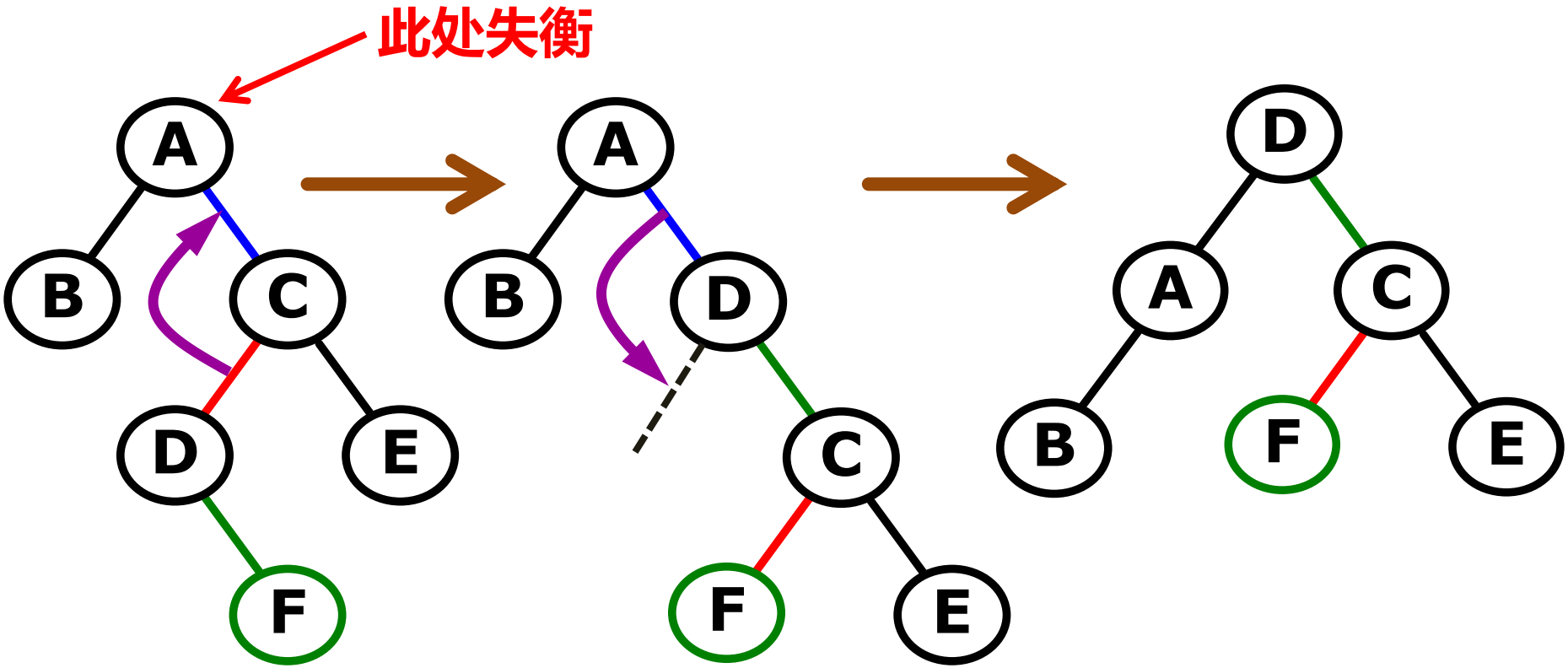
pc->lchild = pd->rchild;

pd->rchild = pc;

pa->rchild = pd->lchild;

pd->lchild = pa;

双旋转：RL型（先右旋，再左旋）



pc->lchild = pd->rchild;

pd->rchild = pc;

pa->rchild = pc->lchild;

pc->lchild = pa;

课堂练习

- 给定序列{**7, 6, 2, 8, 4, 9, 5**}
  - 构造二叉排序树
  - 给出删除关键字8后的二叉排序树
  - 构造AVL

3. 哈希表（散列表）

- 哈希表概述
- 哈希函数的构造
- 冲突的解决方法
- 哈希表的查找分析

哈希表概述

哈希表

- 哈希（Hashing）
  - 设：待查找表中的关键字集合为K
  - 使用函数 h 将 K 映射到表 T[0..M-1] 的下标上
    - 这样h(Ki)就是相应结点 Ki 的存储地址
  - 函数h：K→{0, 1, …, M-1}
    - 称为哈希函数(Hash function)或者散列函数
  - 表T[0..M-1]称为哈希表(Hash table)
  - 通过哈希函数将关键字存储到哈希表中的过程称为哈希
    - 查找使用直接寻址法（不通过关键字比较）

哈希表

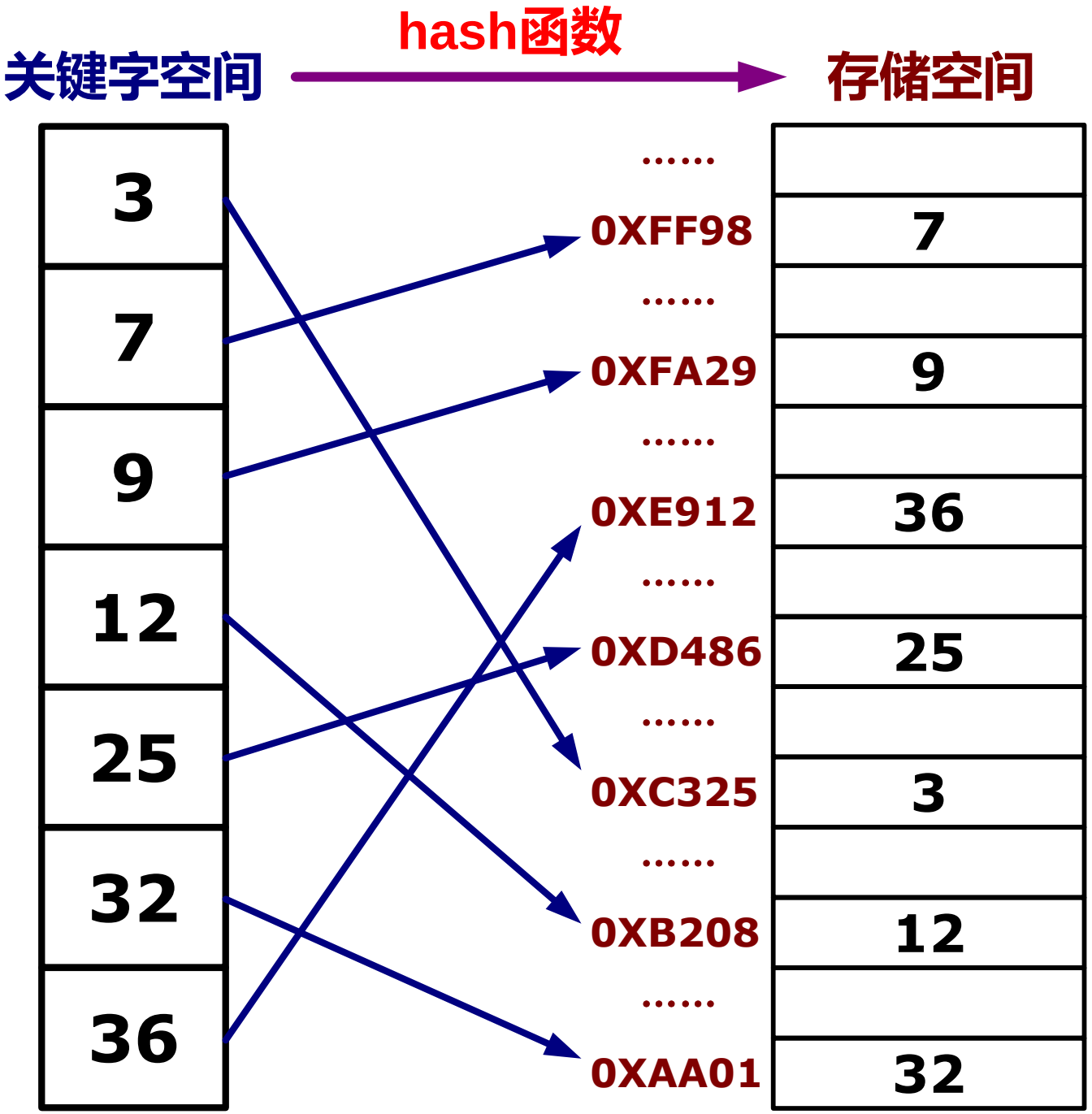
- 例如：为每年招收的 1000 名新生建立一张查找表
  - 关键字为学号
  - 取值范围：xxxx000 ~ xxxx999 （前四位为年份）
- 若将学生记录分别存放在下标为000 ~ 999的查找表中
  - 则对于给定学号的查找过程为：
    - 取给定学号的后三位（作为下标）
    - 根据下标可直接从查找表中找到给定学生的记录
- 特点：查询时不需要经过关键字比较

哈希表

- 在记录的存储地址和其关键字间建立一个确定的映射关系
  - 查找时可根据映射关系直接求得待查记录的存储地址
  - 不需经过关键字比较，即可直接访问待查记录
  - 这种映射关系，称为哈希函数（hash function）
- 哈希表（散列表）
  - 借助哈希函数对记录的关键字进行计算得到地址值
  - 根据计算得到的地址值，将记录放入表中该地址处
  - 这样构成的表称为哈希表
- 哈希查找（散列查找）
  - 利用哈希函数进行查找的过程称为哈希查找

哈希函数

- 哈希函数的定义
  - 在记录的关键字与记录的存储地址之间
  - 建立的某种映射（1:1对应）关系称为哈希函数
- 映射：关键字空间 → 存储地址空间
  - 表示：**Addr(R_i) = Hash(key_i)**
    - **R_i** 是表中的一个元素
    - **key_i** 是 **R_i** 的关键字


关键字空间
hash函数
存储空间
3
7
9
12
25
32
36
……
0XFF98
……
0XFA29
……
0XE912
……
0XD486
……
0XC325
……
0XB208
……
0XAA01
7
9
36
25
3
12
32

例：30个地区的各民族人口统计表

编号	地区	总人口	汉族	回族 ……
1	北京			
2	上海			
⋮	⋮			

以编号为关键字构造哈希函数：H(key) = key

有：H(1)=1　H(2)=2　……

以地区为关键字，取地区名称拼音首字母的序号作为哈希函数

有：H(Beijing)=2　H(Shanghai)=19　H(Shenyang)=19

哈希函数

- 哈希函数的本质只是一种映象
  - 所以哈希函数的设定很灵活，只要使得：
  - 任意关键字的哈希函数值都落在表长允许的范围之内即可
- 哈希函数的同义词
  - 具有相同哈希值的两个不同的关键字称为该函数的同义词
- 哈希函数的冲突
  - **key1 ≠ key2，但：Hash(key1) = Hash(key2)**
- 哈希函数通常是一种压缩映象：所以冲突不可避免
  - 冲突发生后，应该有处理冲突的方法
- 设计良好的哈希函数：（1）计算简单（2）冲突少

哈希函数的构造

哈希函数的构造方法

- 常见的哈希函数构造方法
  - 直接哈希函数法
  - 数字分析法
  - 平方取中法
  - 折叠法
  - 除留余数法
  - 随机数法

实践中常用

1：直接定址法

- 构造方法：首先对关键字进行分析
  - 取关键字或关键字的某个线性函数作为哈希地址
  - 即：**Hash(key) = a·key+b** （a, b为常数）
    - 最简单的情况：**Hash(key) = key**
- 直接定址法的特点
  - 地址集合与关键字集合大小相等
  - 若关键字彼此不冲突，则哈希地址不会发生冲突
  - 实际应用中很少使用这种形式的哈希函数
  - 适于关键字事先可预知的情况

1：直接定址法

哈希地址	01	……	22	……
年份	1949	……	1970	……
人数		……		

- 例如：有一个解放后出生人口调查表，每个记录包含：年份、人数等数据项，其中年份为关键字
- 则哈希函数可取为：
  - **Hash (key) = key + (-1948)**
  - 这样就可以方便地存储和查找1948年后任一年的记录

2：数据分析法

- 构造方法：在关键字比较长和关键字的形式已知情况下
  - 选取其中随机性好的若干位（或将其组合）作为哈希地址
- 使用前提：关键字取值范围比哈希地址取值范围大
  - 设有 n 个 d 位数的关键字，由 r 个不同的符号组成
    - 这 r 个符号在关键字各位出现的频率不一定相同
    - 可能在某些位上均匀分布
      - 即：每个符号出现的次数都接近于 n / r 次
    - 而在另一些位上分布不均匀
  - 则可选择其中分布均匀的s位作为哈希地址（s < d）
    - 即：H（key）=“key中数字均匀分布的s位”

例：有80个记录，关键字为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
8	1	3	6	8	5	3	7
8	1	4	1	9	3	5	5
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

n=80, d=8, r=10, s=2

分析：① 只含数字8;

② 只含数字1;

③ 只含数字3、4;

⑧ 只含数字2、7、5;

④⑤⑥⑦数字分布近乎随机

所以：取④⑤⑥⑦任意两位（或其中两位与另两位的叠加）作哈希地址

例如：可以取第4、6两位组成的2位十进制数作为每个数据的哈希地址，则图中列出的关键字的哈希地址分别为：

45，72，84，03，28，39，51，65，13

3：平方取中法

- 构造方法
  - 取关键字平方值的中间几位作为哈希地址
  - 即：**Hash（key）= “key² 的中间几位”**
  - 其中：所取的位数由哈希表的大小确定
- 求关键字的平方值的目的是
  - “扩大差别”和“贡献均衡”
  - 关键字的各位都对平方值的中间几位有所贡献
  - Hash值中应该包含有各位的信息
- 适于不知道全部关键字情况

3：平方取中法

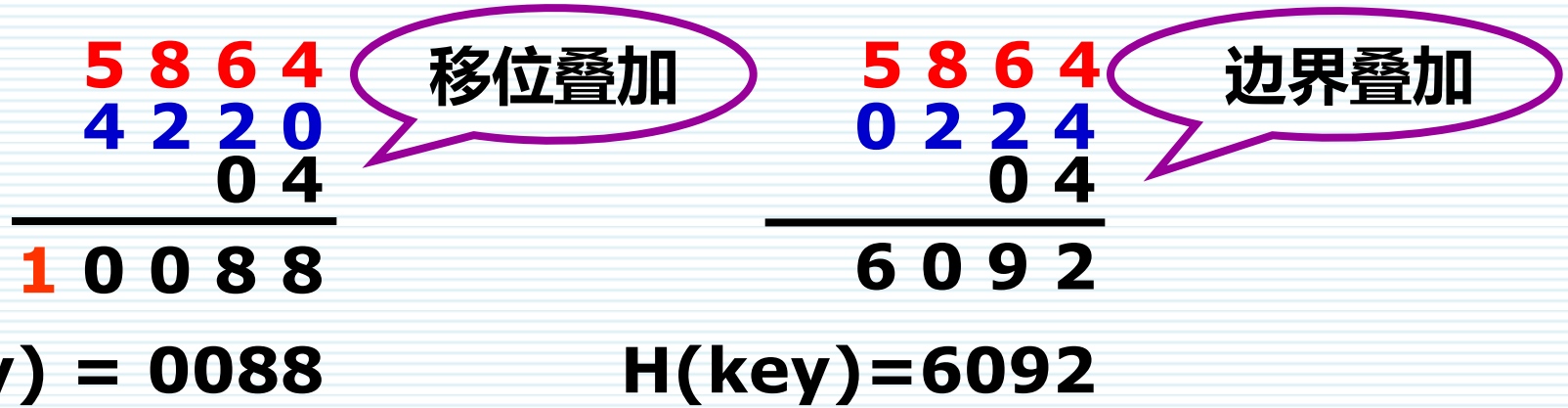
- 为BASIC源程序中的标识符建立一个哈希表
  - 假设BASIC语言中允许的标识符为一个字母
  - 或一个字母加一个数字
- 取标识符在计算机中的八进制数为它的关键字
  - 假设哈希表长度为：$2^9 = 512$
  - 可以取关键字平方后的中间9位二进制数作为哈希地址
- 如下表所示（中间9位二进制数即3位八进制数）

3：平方取中法

数据	关键字	(关键字)^2	哈希地址 (2^0~2^9)
A	0100	0010000	010
I	1100	1210000	210
J	1200	1440000	440
I0	1160	1370400	370
P1	2061	4310541	310
P2	2062	4314704	314
Q1	2161	4734741	734
Q2	2162	4741304	741
Q3	2163	4745651	745

4：折叠法

例：关键字为：0442205864　　哈希地址位数为4



- 构造方法
  - 将关键字分割成位数相同的几部分
  - 然后取这几部分的叠加和（舍去进位）做哈希地址
  - 移位叠加：将分割后的几部分低位对齐相加
  - 边界叠加：从一端沿分割界来回折送，然后对齐相加
- 适于关键字位数很多，且每一位上数字分布大致均匀情况

5：除留余数法

例如：给定关键字序列 {121,123,136,214,286,334}

取hash函数为： Hash(key) = key mod 7

得到相应的hash函数值：{2, 4, 3, 4, 6, 5}

- 关键字的哈希地址采用如下方式计算得到
  - 关键字被某个不大于哈希表表长m的数p除后所得余数
  - 即：**Hash(key) = key mod p** （p ≤ m）
- 特点：简单、常用，可与上述几种方法结合使用
  - p的选取很重要（p选的不好容易产生同义词）
    - 通常选择一个不大于m的素数作为p

6：随机数法

- 取关键字的随机函数值作哈希地址
  - 即：**Hash(key) = rand(key)**
- 适用于关键字长度不等的情况

- 实际工作中需根据不同的情况选用不同的哈希函数
- 通常需要考虑的因素有
  - 计算哈希函数所需时间
  - 关键字长度和关键字分布情况
  - 哈希表长度（哈希地址范围）
  - 记录的查找频率

冲突的解决方法

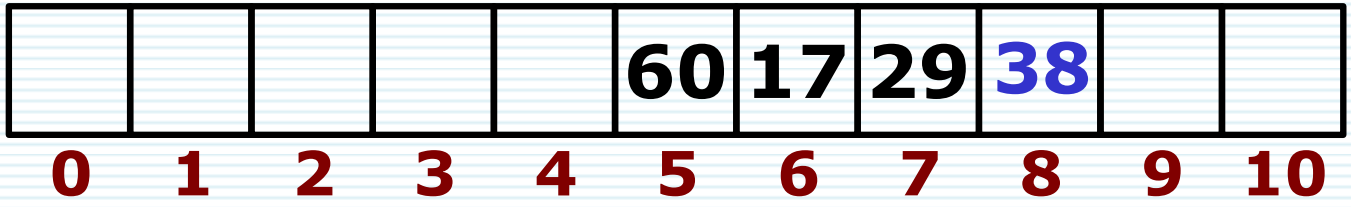
冲突的解决方法

- 冲突
  - 经由关键字计算得到的Hash地址上已存有其他记录
- 冲突处理
  - 为出现地址冲突的关键字寻找下一个哈希地址
  - 设计良好的哈希函数可以减少冲突，但很难避免冲突
- 常见的冲突处理方法
  - 开放定址法
  - 双散列法（再哈希法）
  - 链地址法

1：开放定址法

- 当冲突发生时，使用某种算法在哈希表中形成一个探测序列
- 沿该序列逐个进行地址探测直到找到一个空位置（开放的地址）
- 将发生冲突的记录存放到该地址中
- 即：$\mathbf{H_i = (Hash(key) + d_i) \bmod m}$ $(i=1, 2, \ldots, k \le m-1)$
- 根据增量序列 d_i 的选择方法进一步分为
  - 线性探测法：$\mathbf{d_i = 1, 2, 3, \ldots, m-1}$
  - 平方探测法：$\mathbf{d_i = 1^2, -1^2, 2^2, -2^2, 3^2, \ldots\ldots \pm k^2 \ (k \le m^{1/2})}$

线性探测法



H(38)=38 mod 11=5　冲突

H1=(5+1) mod 11=6　冲突

H2=(5+2) mod 11=7　冲突

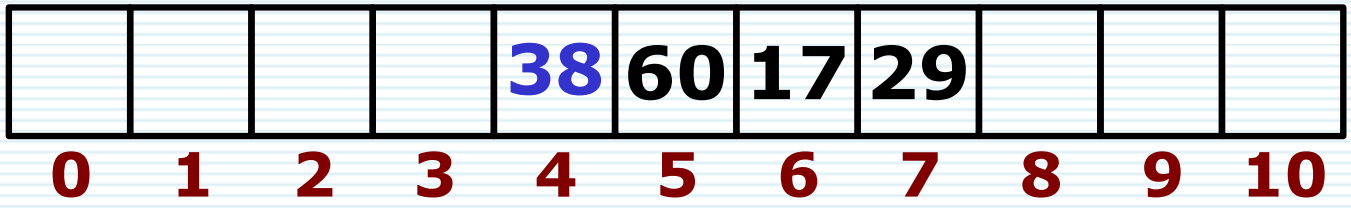
H3=(5+3) mod 11=8　不冲突

- 例：表长为11的哈希表中已填入三条记录
  - 表中记录的关键字分别为：17，60，29
  - 已知哈希函数为：**Hash(key) = key mod 11**
  - 请将第4个关键字**38**填入表中（冲突采用**线性探测法**处理）
- 线性探测法：d_i**=1, 2, 3, …, m-1**

线性探测法

- 采用线性探测法时算法终止于3种情况
  - 若当前探测的单元为空
    - 元素查找操作：查找失败
    - 元素插入操作：将key写入其中，插入成功
  - 若当前探测的单元中含有key
    - 元素查找操作：查找成功
    - 元素插入操作：当前插入失败（继续探测）
  - 若探测到T[d-1]时仍未发现空单元，也未找到key
    - 则无论是查找还是插入均意味着失败（此时表满）

平方探测法



H(38)=38 mod 11=5 冲突

H1=(5+1²) mod 11=6 冲突

H2=(5-1²) mod 11=4 不冲突

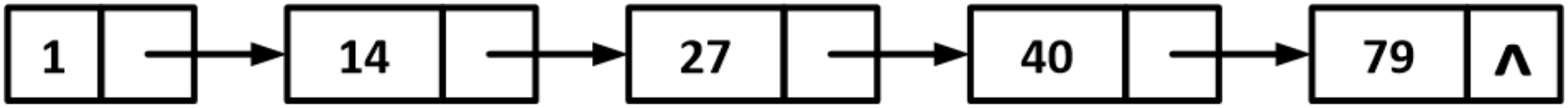
- 例：表长为11的哈希表中已填入三条记录
  - 表中记录的关键字分别为：17，60，29
  - 已知哈希函数为：**Hash(key) = key mod 11**
  - 请将第4个关键字**38**填入表中（冲突采用**平方探测法**处理）
- 平方探测法：**di = 1²,-1²,2²,-2²,3²,……±k²**

2：双散列法

- 思路：以关键字的散列值作为增量进行地址探测
  - 设有两个哈希函数分别为 h_1 和 h_2
  - 当冲突发生时选择 $d_i = i \times h_2(key)$
  - 即：当地址 $i = h_1(key)$ 发生冲突时，探测的地址序列为
    - $i+h_2(key)$, $i+2h_2(key)$, $i+3h_2(key)$, ...
- 特点
  - 这种方法不会产生数据聚集
  - 但计算开销较大

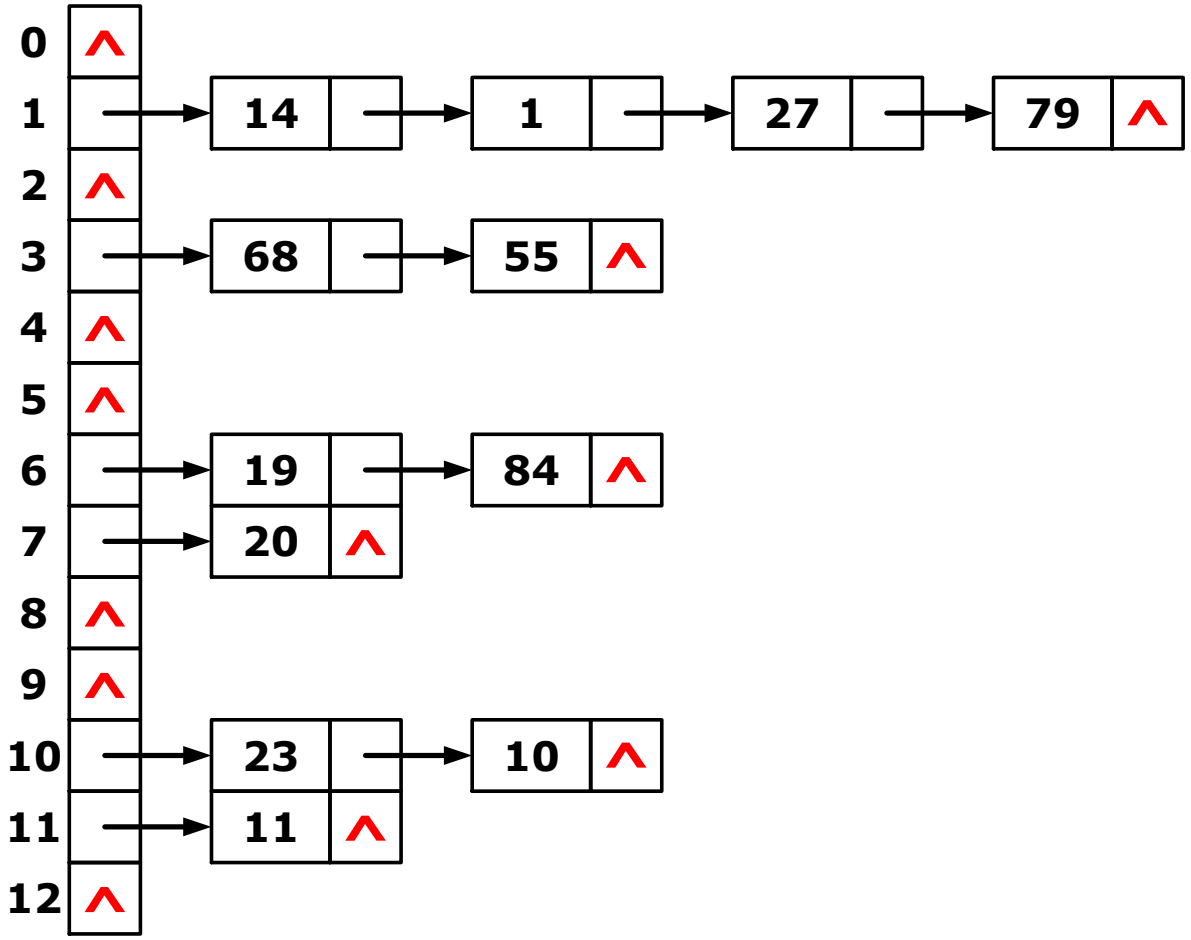
3：链地址法

设：哈希函数 Hash(key)=key mod 13



- 链地址法
  - 将哈希表的每一个地址空间定义为一个单链表头指针
    - 哈希表用一维数组表示（数组元素为表头指针）
  - 当发生冲突时，将所有哈希值相同的关键字链接到
    - 以该哈希地址为表头指针的链表中
- 优点：不会产生溢出（因为链表会在新加入关键字后扩展）

3：链地址法



- 例：给定一组关键字(19,14,23,1,68,20,84,27,55,11,10,79)
- 设：哈希函数H(key)=key mod 13，用链地址法处理冲突

3：链地址法

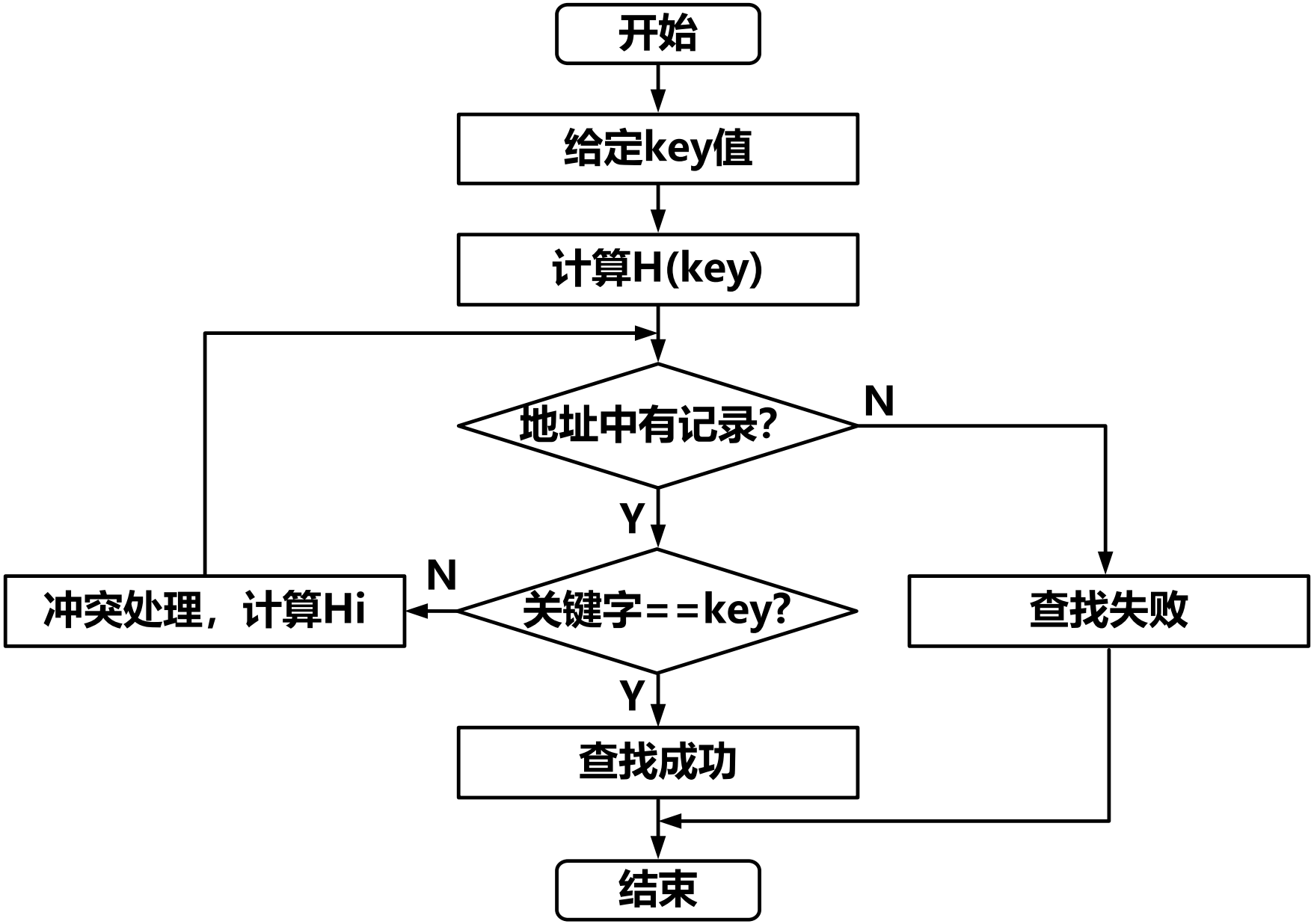
- 链地址法的优点
  - 冲突处理简单
    - 不会像开放定址法那样发生关键字堆积现象
    - 即哈希值不同的关键字之间不会发生冲突
  - 不会产生溢出（因为链表可扩展）
  - 适用于无法确定哈希表长度的情况
  - 删除结点的操作易于实现

哈希表的查找分析

哈希表的查找算法

- 在哈希表上查找的过程和构造哈希表的过程是一致的
- 算法流程
  - 给定Key值，根据构造表时所用的哈希函数求哈希地址
  - 若此位置无记录，则查找不成功
  - 否则比较关键字，若和给定的关键字相等则查找成功
  - 否则根据构造哈希表时设定的冲突处理方法计算“下一个可能的存储地址”，流程返回步骤(2)继续执行查找

哈希表的查找流程



哈希表的查找性能分析

装填因子：**α = 表中填入的记录数 / 哈希表长度**

- 在哈希表上查找的过程是一个给定值与关键字进行比较的过程
  - 因此要评价哈希查找算法的效率仍需对ASL进行分析
- 哈希查找过程中与给定值进行比较的关键字个数取决于
  - 选用的哈希函数和冲突处理方法
  - 装填因子α：哈希表饱和的程度
    - 线性探测法：$\mathrm{ASL} \approx \frac{1}{2}(1+\frac{1}{1-\alpha})$
    - 链地址法：$\mathrm{ASL} \approx 1+\frac{\alpha}{2}$


UESTC
1956