第7章 查找

查找表是由同一类型的数据元素(或记录)构成的集合

什么是查找表

- 查找表是由同一类型的数据元素(或记录)构成的集合

- 对查找表经常进行的操作：
  - 1）查询某个“特定的”数据元素是否在查找表中；
  - 2）检索某个“特定的”数据元素的各种属性；
  - 3）在查找表中插入一个数据元素；
  - 4）从查找表中删去某个数据元素。

查找(Search)的概念

- 查找就是在数据集合中寻找满足某种条件的数据元素。
  - 条件关键字等于给定值
- 查找的结果通常有两种可能：
  - 查找成功，即找到满足条件的数据元素。这时，作为结果，可报告该元素在结构中的位置，还可给出该元素中的具体信息。
  - 查找不成功，即查找失败。作为结果，应报告一些信息，如失败标志、位置等。
- 通常称用于查找的数据集合为查找表，它是由同一数据类型的元素（或记录）组成。

什么是关键字？

- 关键字：是数据元素（或记录）中某个数据项的值，用以标识（识别）一个数据元素（或记录）。
- 主关键字：此关键字可以识别唯一的一个记录。
- 次关键字：此关键字能识别若干记录。

学号	姓名	专业	年龄
20030001	王洪	计算机	17
20030002	李文	计算机	18
20030003	谢军	计算机	18
20030004	张辉	信息工程	20
20030005	李文	信息工程	19

什么是查找表

- 查找表的分类：

- 静态查找表：仅作查询和检索操作的查找表。
- 动态查找表：可以对查找表进行插入删除操作：
  - 将“查询”结果为“不在查找表中”的数据元素插入到查找表中；
  - 从查找表中删除其“查询”结果为“在查找表中”的数据元素。

查找方法评价

- 度量查找算法的效率
- 查找算法的基本操作：比较
- 平均查找长度－ASL (Average Search Length)：
  - ¶ 为了确定记录在表中的位置, 在查找过程中关键码的平均比较次数。
- 设查找成功的总概率为1：$\sum_{i=1}^{n} p_i = 1$

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

  - n 为表长
  - p_i：查找第 i 个记录的概率
  - c_i：查找第 i 个记录所需的比较次数

本章内容

- 7.1 简单查找方法
- 7.2 二叉查找树
- 7.3 AVL树
- 7.4 B树
- 7.5 散列表及其查找

7.1 简单查找方法

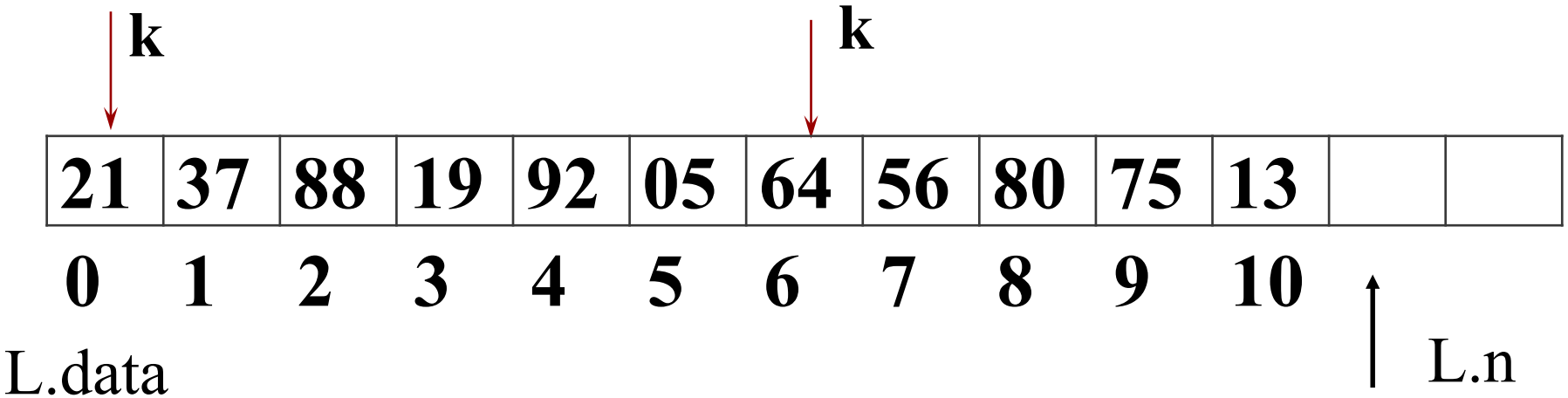
- 7.1.1 顺序查找法
- 以顺序表或线性链表表示静态查找表

```
typedef int DataType;                  //查找数据的类型
typedef struct {                       //查找表定义
   DataType data[maxSize];             //数据存储数组
   int n;                              //数组当前长度
} SeqList, OrderedList;
```

顺序静态查找表结构定义

7.1.1 顺序查找法

- 顺序表的查找过程：从前向后查找



假设给定值 e=64,

要求 L.data[k] = e, 问: k = ?

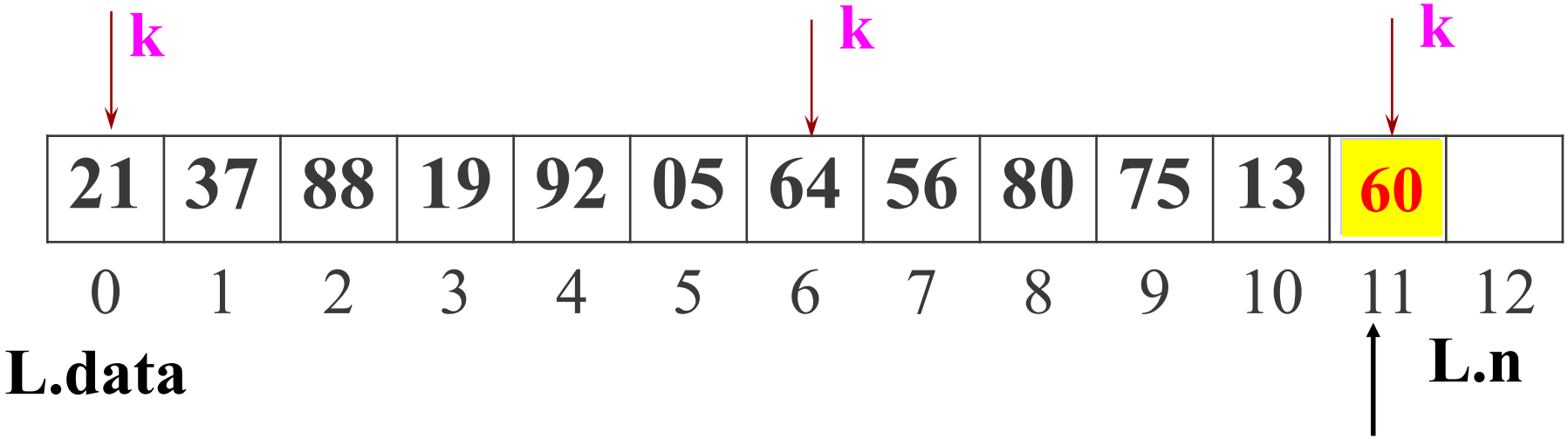
e=60?

7.1.1 顺序查找法

```
int SeqSearch ( SeqList& L, DataType x ) {
//在数据表L.data[0]..L.data[n-1] 中顺序查找关键码
//值与给定值 x 相等的数据元素
        //从前向后顺序查找
        for( i = 0; i < L.n && L.data[i]!= x; i++);
        if (i >= n) return -1;
        else return i;
}//SeqSearch
```

两次比较

7.1.1 顺序查找法



假设给定值 e=64,

要求 L.data[k] = e, 问: k = ?

e=60?

设置“监视哨”的顺序查找算法

```
int SeqSearch ( SeqList& L, DataType x ) {
//在数据表L.data[0]..L.data[n-1] 中顺序查找关键码
//值与给定值 x 相等的数据元素
//L.data[n] 作为控制搜索自动结束的“监视哨”使用
    L.data[L.n] = x;   //将x设置为监视哨
    for( i = 0; L.data[i]!= x; i++);//从前向后顺序查找
    if (i >= n) return -1;
    else return i;
};//SeqSearch
```

一次比较

顺序查找的时间性能分析

- 在等概率查找情况下，元素成功时的查找概率为：1/n
- 设查找第 i 个元素的概率为 p_i，查找到第 i 个元素所需比较次数为 c_i，则查找成功的平均查找长度为：

$$ASL_{succ} = \sum_{i=0}^{n-1} p_i \cdot c_i . \quad (\sum_{i=0}^{n-1} p_i = 1)$$

- 在顺序查找并设置“监视哨”情形，$c_i = i+1$

$$ASL_{succ} = \sum_{i=0}^{n-1} \frac{1}{n}(i+1) = \frac{1}{n}\ \frac{n(n+1)}{2} = \frac{n+1}{2}.$$

顺序查找的时间性能分析

- 考虑查找不成功的情况
  - 算法的ASL= 成功时ASL + 不成功时的ASL
- 对于顺序表并设置“监视哨”情形
  - 查找不成功的比较次数均为n+1
  - 假设查找成功和不成功的可能性相同，对每个元素的查找概率也相同则
  - $ASL_{ss} = \frac{1}{2n+1}\sum_{i=0}^{n-1}(i+1) + \frac{n(n+1)}{2n+1}$

- 在不等概率查找的情况下
  - ASL_{ss} 在$P_1 \geq P_2 \geq \cdots \geq P_{n-1} \geq P_n$ 时，平均查找长度取极小值

顺序查找的特点

- 顺序查找表的查找算法的优缺点：
  - 算法简单；
  - 无排序要求；
  - 存储结构：顺序、链式；
  - 平均查找长度$ASL_{SS} = (n+1)/2$ ；
  - 平均查找长度较大，不适用于表长较大的查找表。
- 若查找概率无法事先测定，则查找过程采取的改进办法是，在每次查找之后，将刚刚查找到的记录直接移至表尾的位置上。

L1=(45, 61, 12, 3, 37, 24, 90, 53, 98, 78）

基于有序顺序表的顺序查找

- 有序顺序表限定表中的元素按照其关键码值从小到大或从大到小依次排列。
- 在有序顺序表中做顺序查找时，若查找不成功，不必检测到表尾才停止，只要发现有比它的关键码值大的即可停止查找。

05	13	19	21	37	56	64	75	80	88	92		
0	1	2	3	4	5	6	7	8	9	10	11	12

基于有序顺序表的顺序查找

```
int OrderSeqSearch1 ( OrderedList& L, DataType x ){
//在有序数据表L.data[0..n-1] 中顺序查找关键码
//值为 x 的数据元素

    for ( i = 0; i < L.n && L.data[i] < x; i++ );//顺序比较
    if ( i < L.n && L.data[i] == x ) return i;
                                  //查找成功，返回 x 所在位置
    else return -1;               //查找不成功, 返回-1

}//OrderSeqSearch1
```

基于有序顺序表的顺序查找

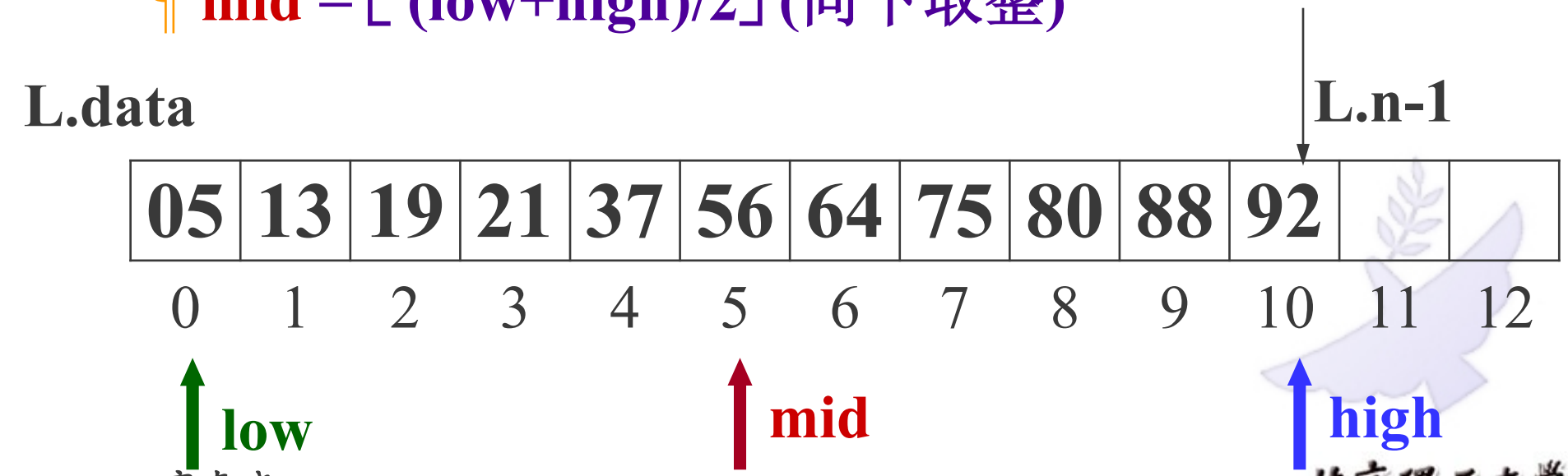
- 性能分析
- 查找成功的平均查找长度: 与前面的算法相同
- 查找不成功的平均查找长度为(等概率条件下):

$$ASL_{unsucc} = \sum_{i=0}^{n} p_i c_i = \frac{1}{n+1}(1 + 2 + \cdots + n + n)$$

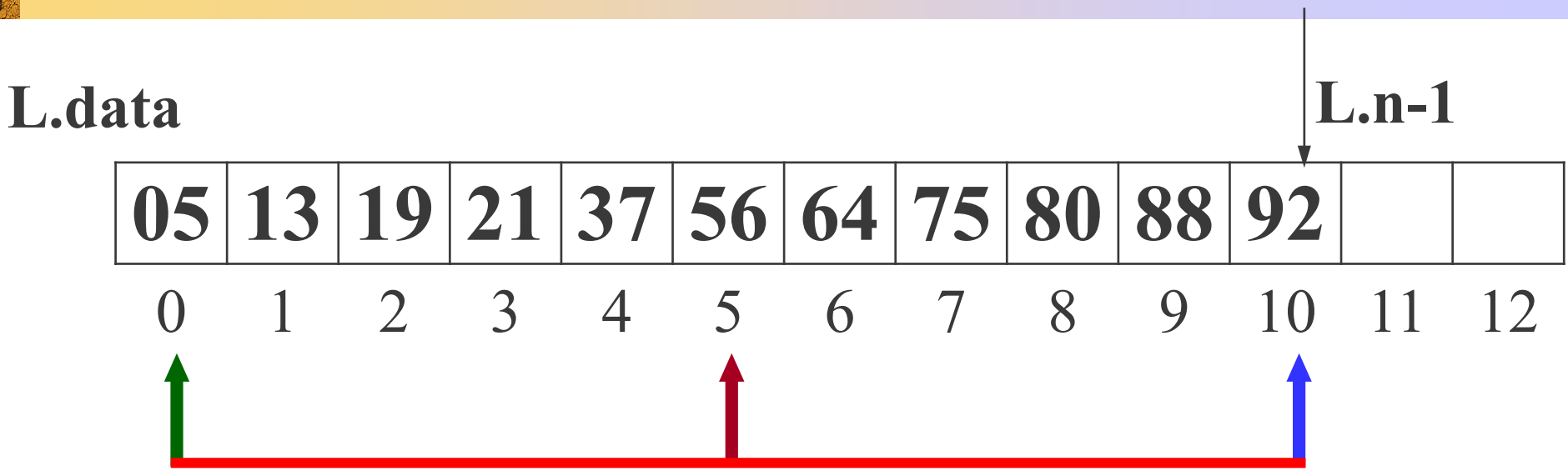
$$= \frac{n}{2} + \frac{n}{n+1}$$

7.1.2 折半查找

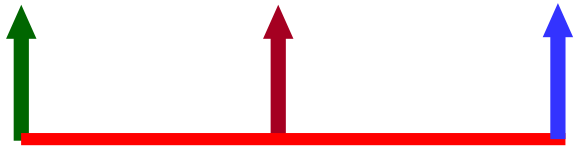
- 若以有序表表示静态查找表, 则查找过程可以基于“折半”进行。
- 定义：
  - low 指示查找区间的下界
  - high 指示查找区间的上界
  - mid =⌊(low+high)/2⌋(向下取整)



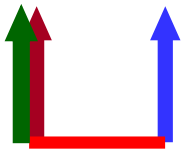
key=64 的查找过程



64>56　low=mid+1 , mid=8



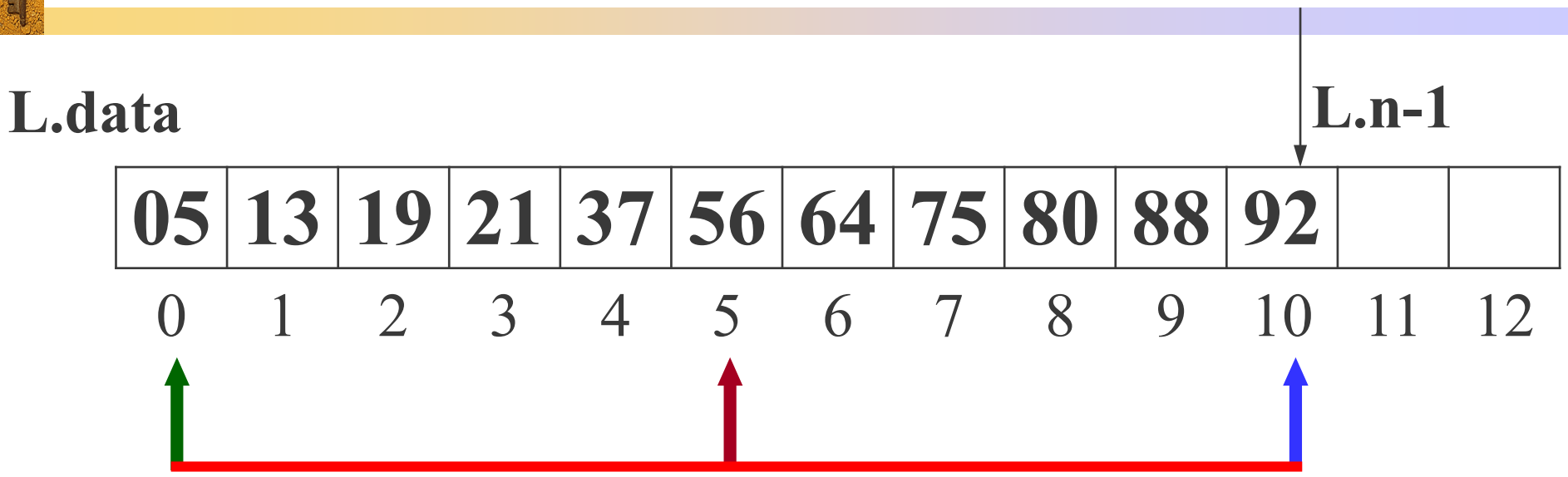
64<80　high=mid-1, mid=6



64= 64

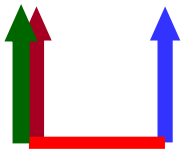
成功！

key=60 的查找过程

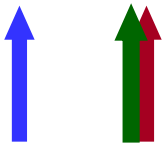


60>56　low=mid+1, mid=8

60<80　high=mid-1 , mid=6



60<64　high=mid-1



low>high 失败！

low指示插入位置！

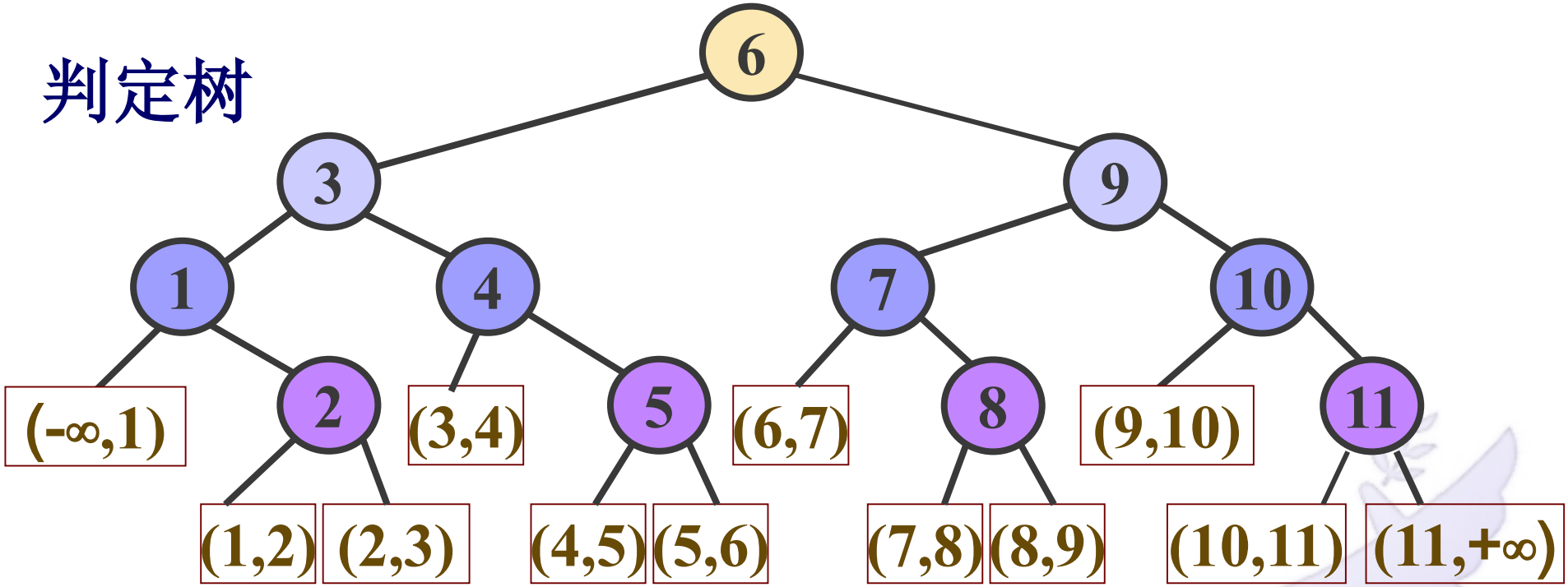
7.1.2 折半查找

```
int BinSearch ( OrderedList& L, DataType x, int& is ) {
//is 在查找失败时表示要插入的位置,成功时返回x位置
    int low = 0, high = L.n-1, mid;
    while ( low<= high ) {
        mid = ( low + right ) / 2;          is = mid;
        if ( x == L.data[mid] ) return mid;    //查找成功
        else if ( x > L.data[mid] )
                low = mid+1;     //右缩查找区间
        else high = mid-1;   //左缩查找区间
    }//while
    is = low ;  return -1;                          //查找失败
}
```

分析折半查找的平均查找长度

i	1	2	3	4	5	6	7	8	9	10	11
Ci	3	4	2	3	4	1	3	4	2	3	4
key	05	13	19	21	37	56	64	75	80	88	92

判定树



具有n个结点的判定树的深度为$\lceil \log_2(n+1) \rceil$

分析折半查找的平均查找长度

- 一般情况下, 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:$h = \lceil \log_2(n+1) \rceil$。
- 设判定树为满二叉树，则
- $n = 2^h - 1,\ h = \log_2(n+1)$

$$ASL_{succ} = \sum_{i=1}^{n} p_i\ c_i = \frac{1}{n}\sum_{i=1}^{n} c_i = \frac{1}{n}\sum_{j=1}^{h} j \cdot 2^{j-1}$$

$$= \frac{1}{n}((h-1)\times 2^h + 1) = \frac{1}{n}((n+1)\log_2(n+1) - n)$$

$$= \frac{n+1}{n}\log_2(n+1) - 1 \approx \log_2(n+1) - 1 \quad (n>50)$$

$$1*2^0 + 2*2^1 + 3*2^2 + \cdots + (h-1)*2^{h-2} + h*2^{h-1})$$

$$\sum_{j=1}^{h} j \cdot 2^{j-1} = (h-1)2^h + 1$$

折半查找的特点

1. 要求元素按关键字有序
2. 存储结构：顺序
3. 平均查找长度 $ASL_{bs}= \log_2(n+1)-1 \ (n>50)$

L=(3, 12, 24, 37, 45, 53, 61, 78, 90, 98)

其它顺序查找方法

- 斐波那契查找
- 斐波那契数列
  - F(0)=0, F(1)=1,
  - F(k)=F(k-1)+F(k-2) (k≥2)

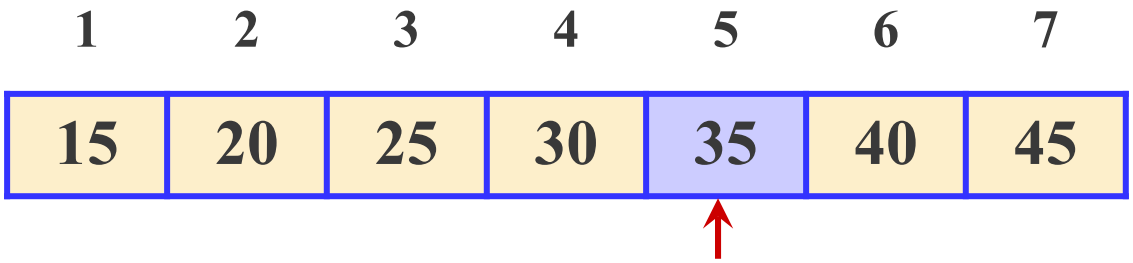
1	2	3	4	5	6	7
15	20	25	30	35	40	45

- 确定查找区间的原则是：若表长 n = F(k)−1，则
  - 中间点为 F(k−1)。
  - 左侧子表的长度为F(k−1)−1
  - 右侧子表的长度为F(k−2) −1。

F(0)	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)	F(7)	F(8)	…
0	1	1	2	3	5	8	13	21	…

斐波那契查找

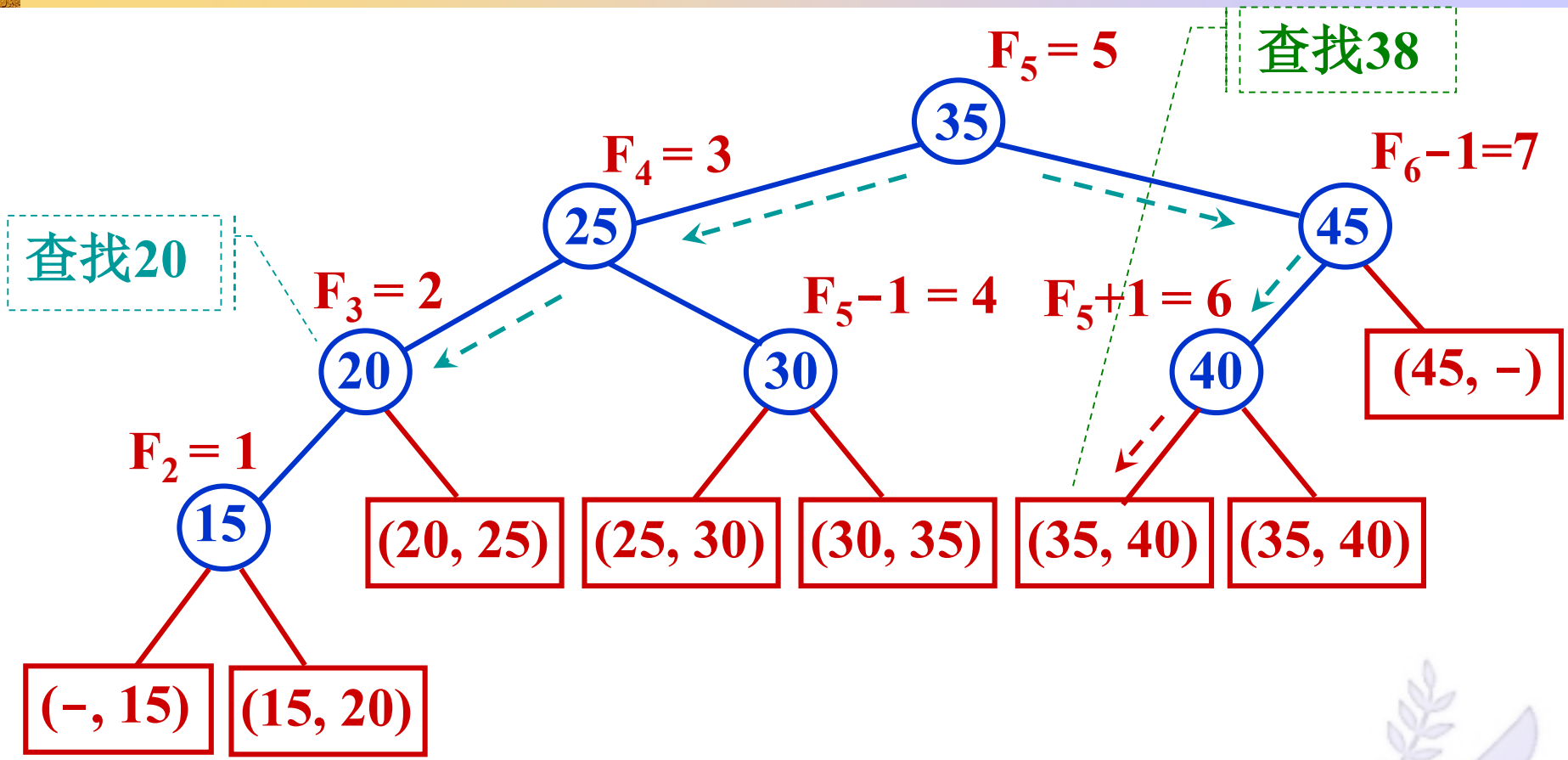
- n = 7 = F(6)−1
- m=F(5)



查找算法描述如下：
若查找区间存在，则
　　若给定值 x = data[m]，查找成功，返回m；
　　若给定值 x < data[m]，到左子区间中查找；
　　若给定值 x > data[m]，到右子区间中查找；
若查找区间不存在，则查找失败。

F(0)	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)	F(7)	F(8)	…
0	1	1	2	3	5	8	13	21	…

斐波那契查找的判定树



- 斐波那契查找的特点：
  - 平均性能比折半查找好；但是最坏性能比之差；

其它顺序查找方法

- 插值查找
  - ¶ 分割点取序列中的位于取值范围中间的位置;

- Key＝6➔ mid=?

$$mid = \frac{6-2}{20-2}(11-1+1) = 2$$

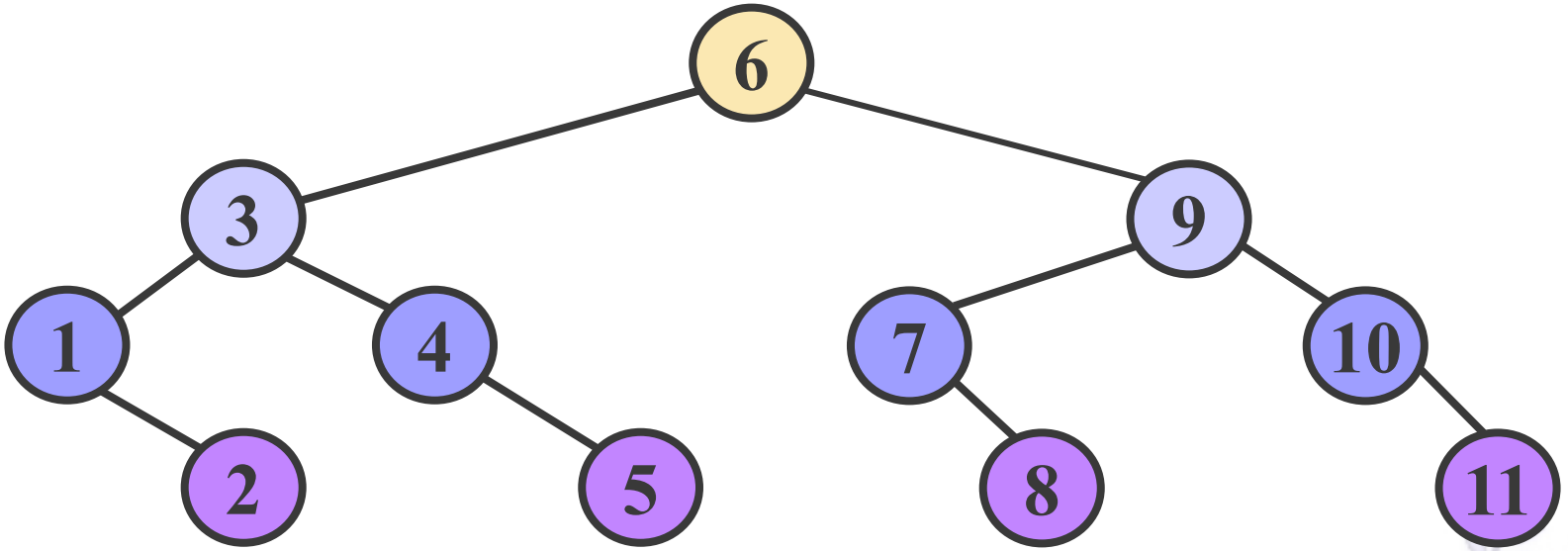
$$mid = \frac{key - \min}{\max - \min}(location(\max) - location(\min) + 1)$$

特点：适合于关键字均匀分布的情况；
在上述情况下，平均性能比折半查找好；

i	1	2	3	4	5	6	7	8	9	10	11
key	2	4	6	8	10	12	14	15	17	19	20

静态搜索树

- 1）静态最优搜索树（Static Optimal Search Tree）
- 回顾：折半查找的平均查找长度：$\log_2(n+1)-1$
- 实际查找长度：取决于树的构造

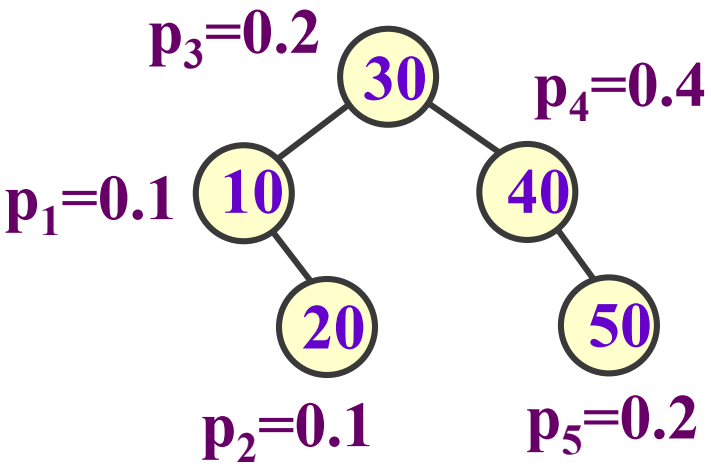


i	1	2	3	4	5	6	7	8	9	10	11
Ci	3	4	2	3	4	1	3	4	2	3	4
key	05	13	19	21	37	56	64	75	80	88	92

静态搜索树

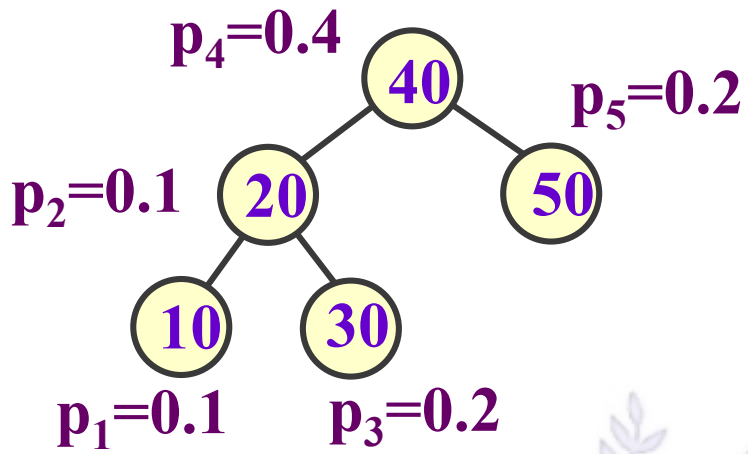
- 假设已知关键字的查找概率，则最优的搜索树是其带权内路径长度之和PH最小的二叉树。

$$PH = \sum_{i=1}^{n} w_i h_i \qquad w_i = cP_i$$



折半查找的判定树1

ASL_{succ}= 0.2*1+(0.1+0.4)*2 +(0.1+0.2)*3 = 2.1



折半查找的判定树2

ASL_{succ}= 0.4*1+(0.1+0.2)*2 +(0.1+0.2)*3 = 1.9

静态最优搜索树

- 设有序顺序表中关键码为 $k_1, k_2, \ldots, k_n$, 它们的查找概率分别为 $p_1, p_2, \ldots, p_n$, 构成查找树后，它们在树中的层次为 $l_1, l_2, \ldots, l_n$。
- 基于该查找树的查找算法的平均查找长度为

$$ASL_{succ} = \sum_{i=0}^{n-1} p_i \cdot l_i$$

- 使得 ASL_{succ} 达到最小的静态查找树为静态最优二叉排序树。然而，求最优查找树的算法效率较低，达 $O(n^3)$, 可求次优查找树，其时间代价减低为 $O(n\log_2 n)$。

静态次优搜索树

- 静态次优搜索树（Nearly Optimal Search Tree）
- 构造方法：
- 已知所有记录按照其关键字大小排序

$$\{ k_l, k_{l+1}, \ldots, k_i, \ldots, k_h \}$$

- 1）在上述序列中取出第i个记录作为根结点, 使得下列值最小

$$\Delta P_i = \left| \sum_{j=i+1}^{m} w_j - \sum_{j=l}^{i-1} w_j \right|$$

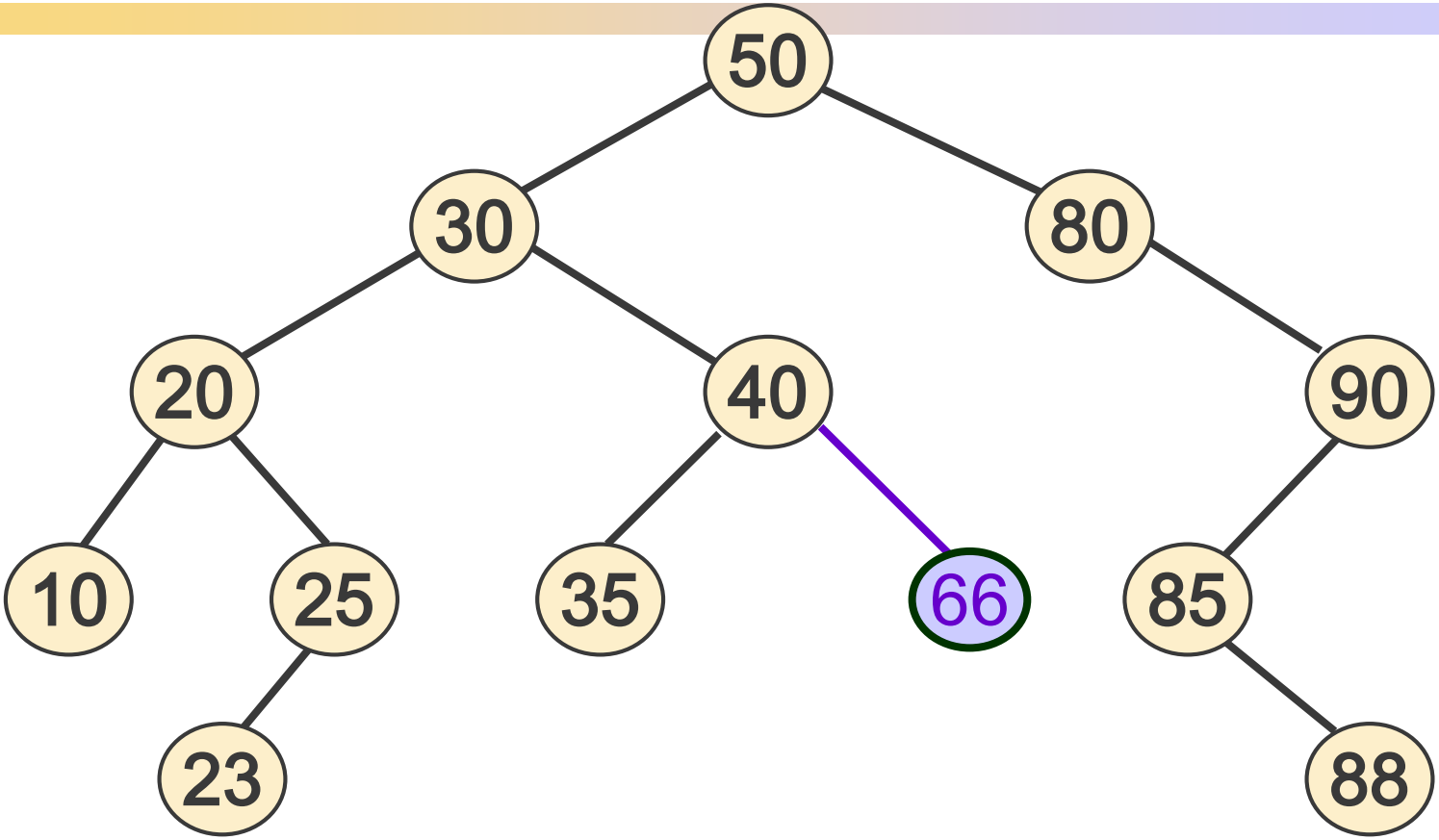
- 2）按照上述步骤分别递归的构造左子树和右子树

7.2 二叉查找树

- 定义：二叉查找树或者是一棵空树；或者是具有如下特性的二叉树
  - ① 每个结点都有一个作为查找依据的关键码(key)，所有结点的关键码互不相同。
  - ② 若它的左子树不空, 则左子树上所有结点的值均小于根结点的值；
  - ③ 若它的右子树不空, 则右子树上所有结点的值均大于根结点的值；
  - ④ 它的左、右子树也都分别是二叉查找树

左子树小于根结点
右子树大于根结点

7.2 二叉查找树



中序遍历：10-20-23-25-30-35-40-50-80-85-88-90

图中加入结点66后, 不再是二叉查找树！

中序遍历：10-20-23-25-30-35-40-66-50-80-85-88-90

二叉查找树的存储结构:二叉链表

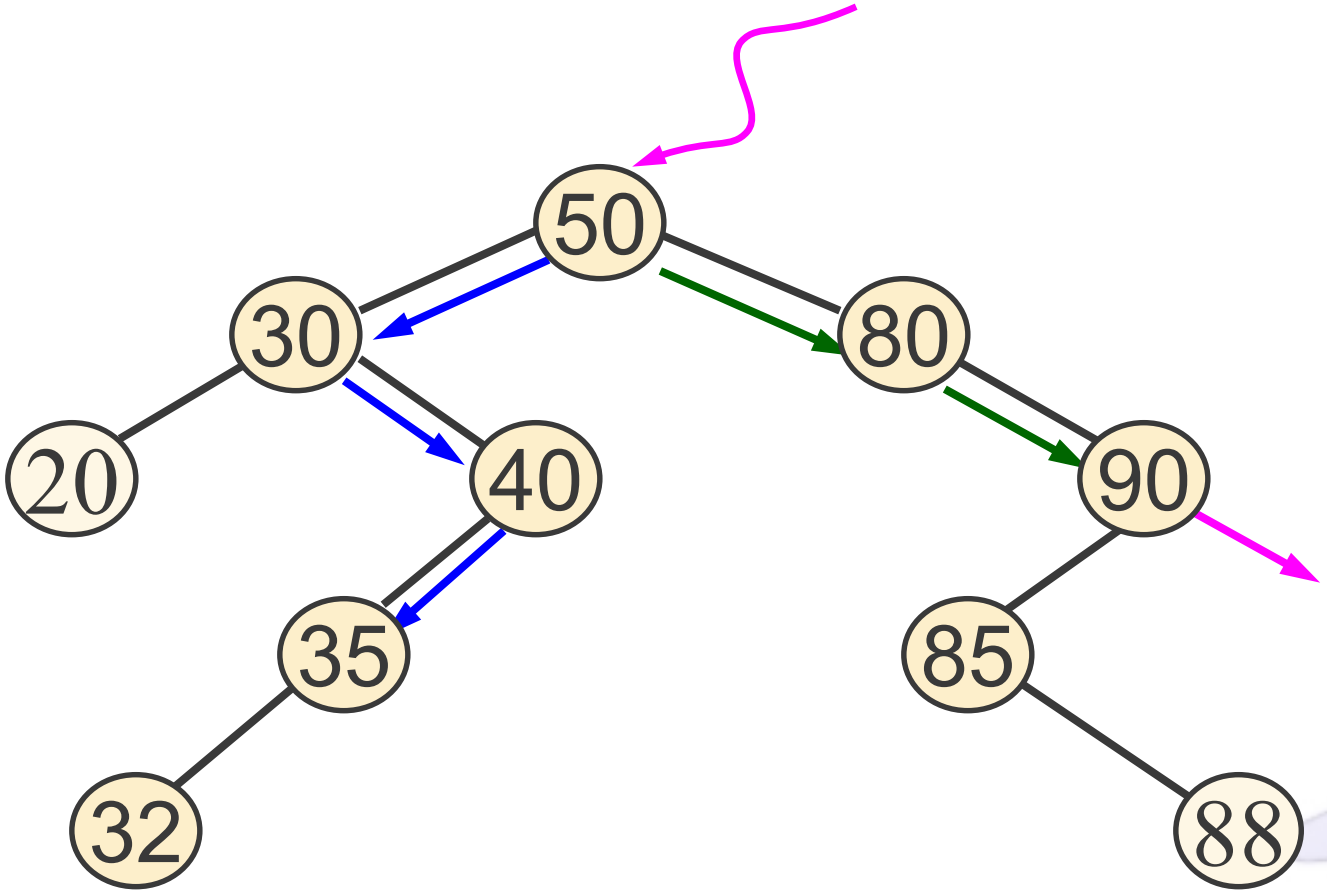
```
typedef int TElemType;     //结点关键码数据类型

typedef struct tnode {
    TElemType data;                 //结点值
    struct tnode *lchild, *rchild;//左、右子女指针
} BSTNode, *BSTree;
```

二叉查找树的查找算法

- 递归算法：
- 若二叉查找树为空, 则查找不成功；
- 否则,
  - 若给定值等于根结点的关键字, 则查找成功；
  - 若给定值小于根结点的关键字, 则继续在左子树上进行查找；
  - 若给定值大于根结点的关键字, 则继续在右子树上进行查找。
- 总之：是在根指针T所指二叉查找树中递归地查找关键字等于key的记录

二叉查找树的查找算法

关键字 **50　35　90　95**



二叉查找树的查找算法

- 在查找过程中, 生成了一条查找路径
  - 1）从根结点出发, 沿着左分支或右分支逐层向下直至关键字等于给定值的结点;

    ——查找成功
  - 2）从根结点出发, 沿着左分支或右分支逐层向下直至指针指向空树为止。

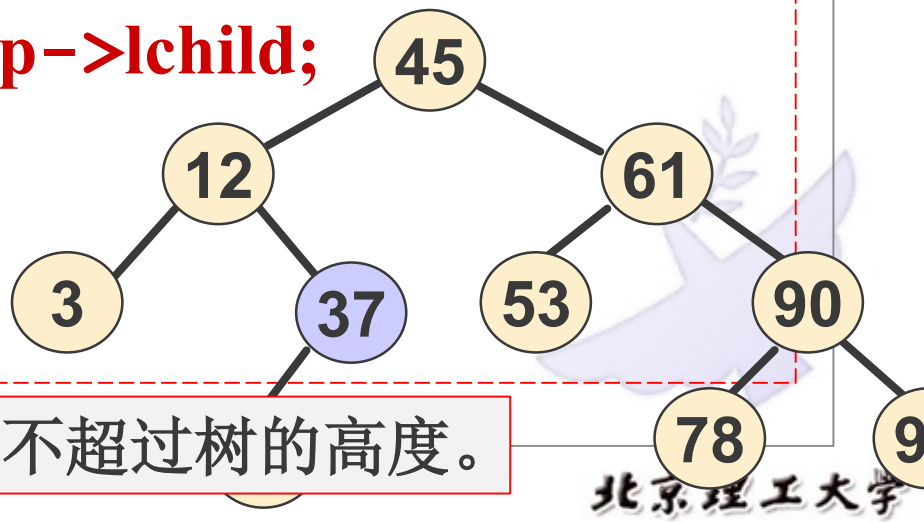
    ——查找不成功

2—二叉查找树的查找算法

```
BSTree Search(BSTree T,  KeyType x)
{ //二叉查找树用二叉链表存储。
  //若查找成功, 则返回该记录结点的指针
  //否则返回空指针
    if( T== NULL) return NULL;
    if( x = T->data ) return T;
      else if ( x < T->data )
                return(Search(T->lchild,  x));
      else  return(Search(T->rchild,  x))
}//Search
```

2—二叉查找树的查找算法

```
BSTNode *Search ( BSTree BT, TElemType x,
    BSTNode *& father ) {
//成功时函数返回找到结点地址，father 是其双亲结点.
//不成功时函数返回空，father指向插入结点的父结点
   BSTNode *p = BT;  father  = NULL;
   while ( p != NULL && p->data != x ) {
         father  = p;     //向下层继续查找
         if ( x < p->data ) p = p->lchild;
         else p = p->rchild;
   }
   return p;
} //S
```

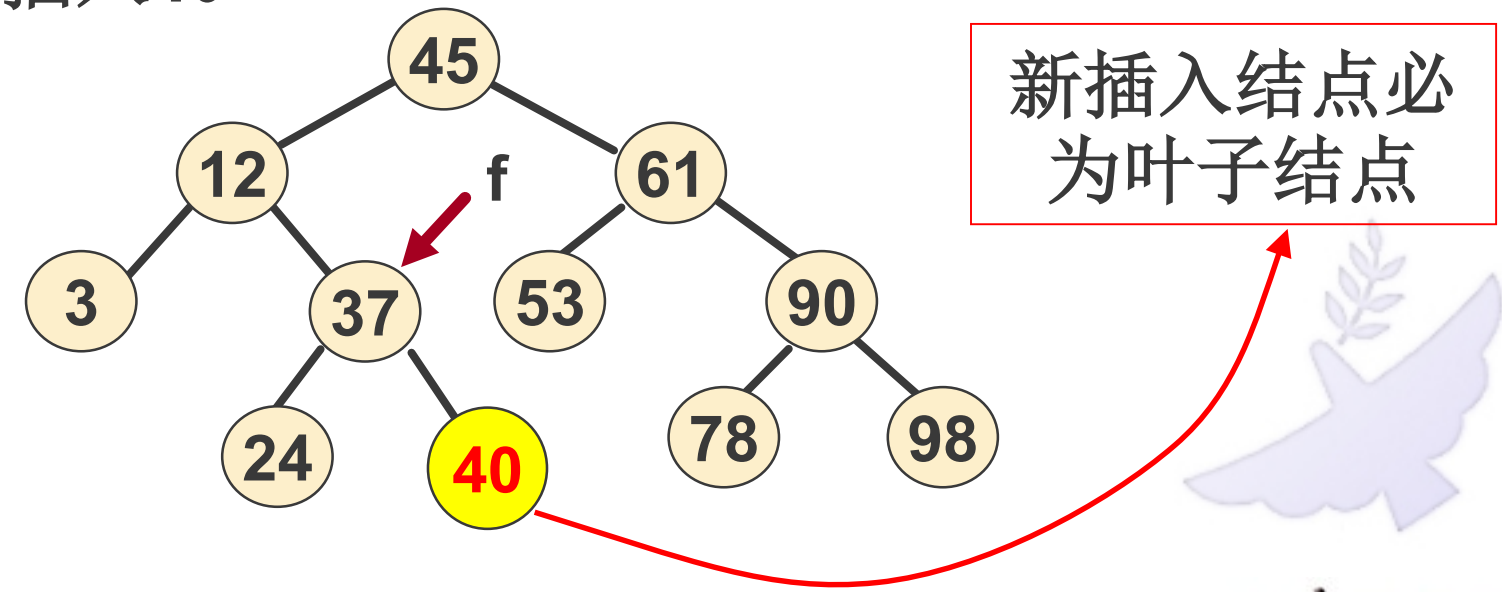


查找的关键码比较次数最多不超过树的高度。

3—二叉查找树的插入算法

- 二叉查找树的特点：是一种动态树表，树的结构通常不是一次生成的，而是在查找过程中，当树中不存在关键字等于给定值的结点时再进行插入。
- 动态查找表：“插入”操作在查找不成功时才进行；

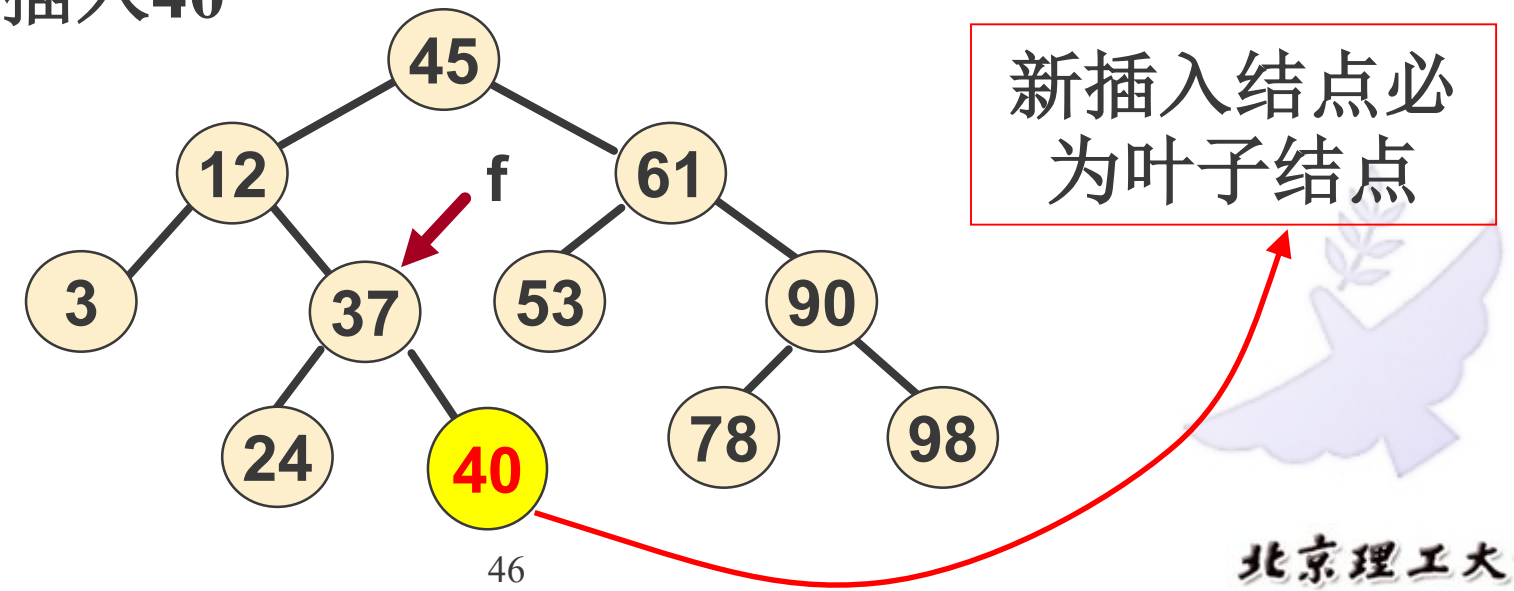
例如：插入40



3—二叉查找树的插入算法

- 插入算法：
  - (1) 若二叉树为空，则首先单独生成根结点；
  - (2) 执行查找算法，找出被插入结点的双亲结点；
  - (3) 判断被插入结点是其双亲结点的左孩子结点还是右孩子结点，将被插入结点作为叶子结点插入；

例如：插入40



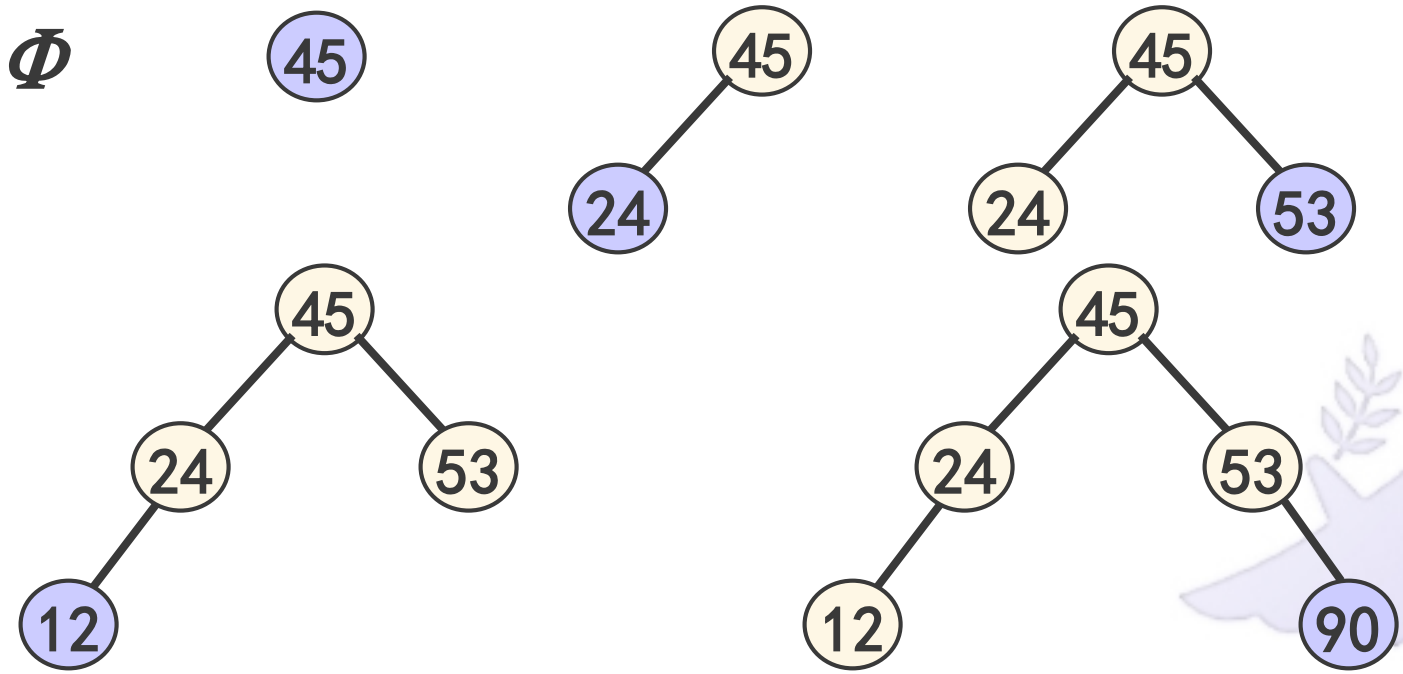
3—二叉查找树的插入算法

```
Status InsertBST ( BSTree  &T, TElemType x ){
  f = NULL;
 if ( ! Search( T, x, f ) ) // 查找
 {  //查找不成功。f指向访问路径上最后一个结点
        s = (BSTree) malloc( sizeof(BSTNode) );
        if ( !s ) {prinf("内存分配失败！ \n"); exit(1);}
        s->data=x;   s->lchild = s->rchild=NULL; //是叶子结点
        if ( !f ) T= s;        // T为空，被插结点为根结点
        else if ( x < f->data) )  f->lchild = s;
         else   f->rchild = s;
        return TRUE;       // 插入成功
  }
 else return FALSE; // 树中已有x，不需要再插入
} // InsertBST
```

例：在一棵空树中，查找如下的关键字序列

{ 45, 24, 53, 45, 12, 24, 90 }

生成的二叉查找树如下：



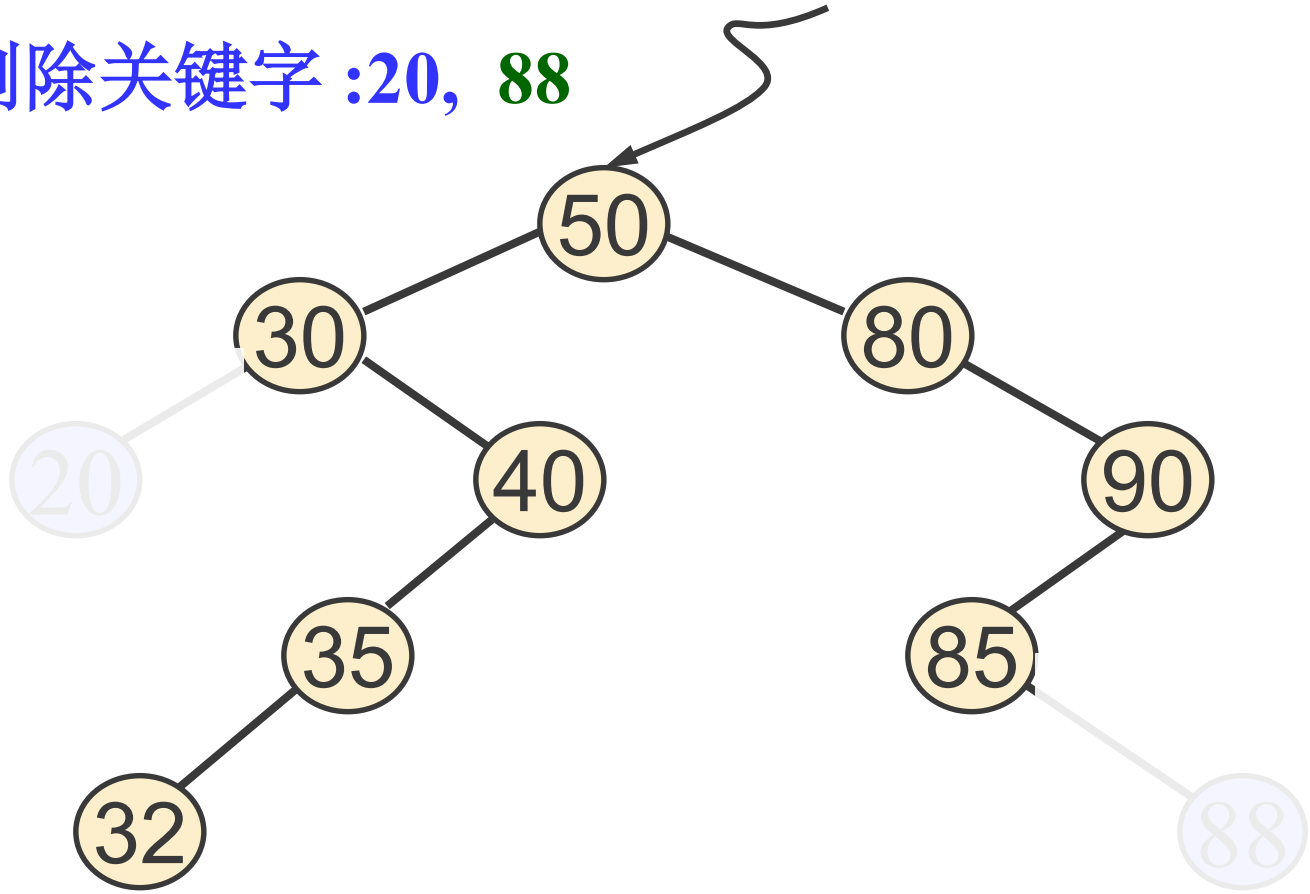
4-二叉查找树的删除算法

- 和插入相反, 删除在查找成功之后进行, 并且要求在删除二叉查找树上某个结点之后, 仍然保持二叉查找树的特性。

- 可分三种情况讨论：
  - 1）被删除的结点是叶子；
  - 2）被删除的结点只有左子树或者只有右子树；
  - 3）被删除的结点既有左子树, 也有右子树。

1）被删除的结点是叶子

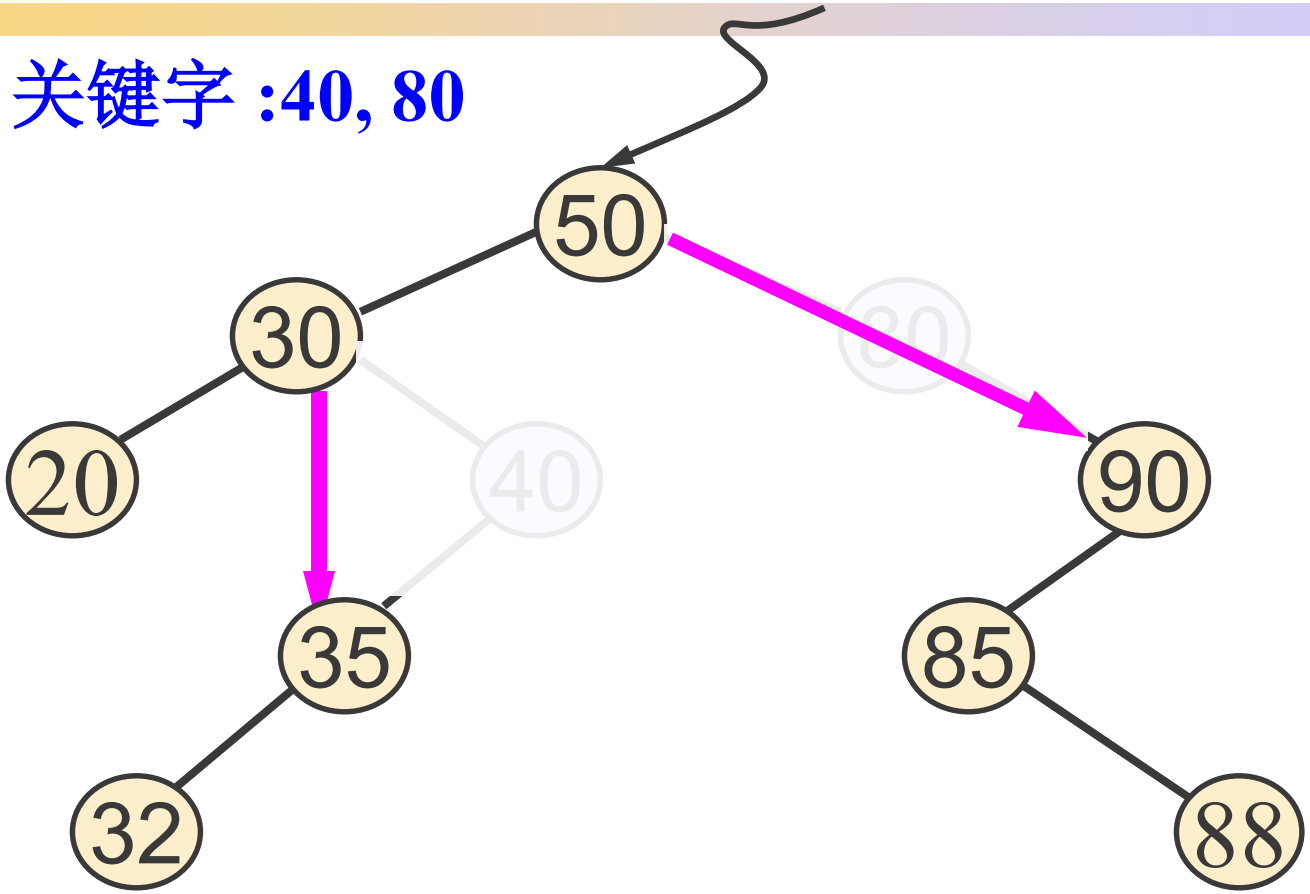
例：删除关键字 :20, 88



方法：其双亲结点中相应指针域的值改为“空”

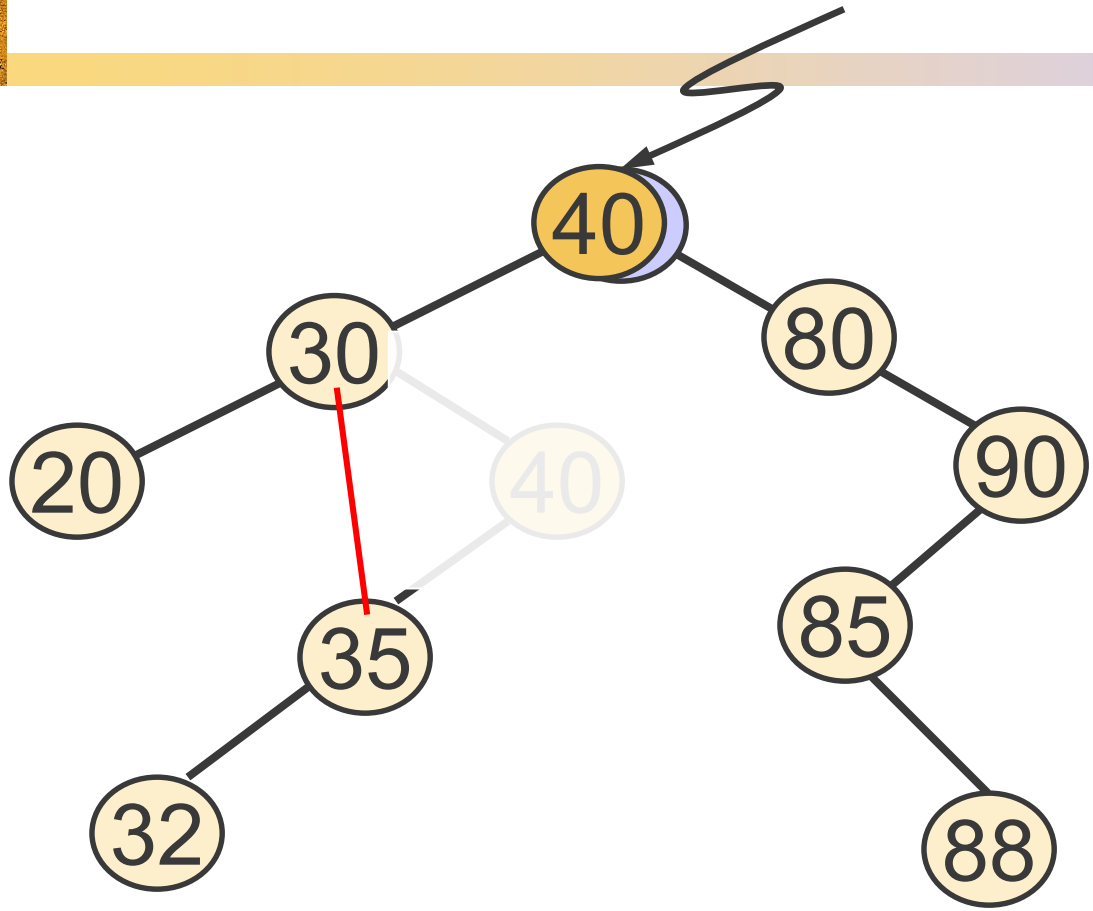
2）被删除的结点只有左子树或者只有右子树；

例如：关键字 :40, 80



方法：其双亲结点的相应指针域的值改为“指向被删除结点的左子树或右子树”。

3）被删除的结点既有左子树, 也有右子树



被删关键字 = 50

前驱结点：左子树的最右下的结点！

中序遍历：20-30-32-35-40-**50**-80-85-88-90

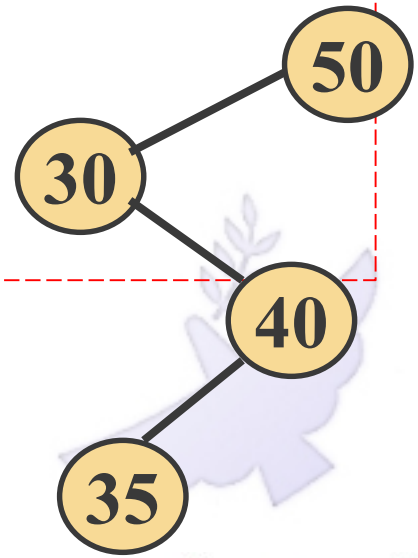
方法：以其前驱替（或后继）代之, 然后再删除该结点

4-二叉查找树的删除算法

```
bool Remove ( BSTree& t, TElemType x ) {
//在*t 为根的二叉查找树中删除关键码为x的结点
    BSTNode *s, *p, *f =NULL;
    p = Search ( t, x, f ) ;
    if ( p == NULL )    return false; //查找失败不删除

     //分情况处理
     //被删结点的左、右子树都不为空
     //被删结点的左或右子树为空

}  // Remove
```

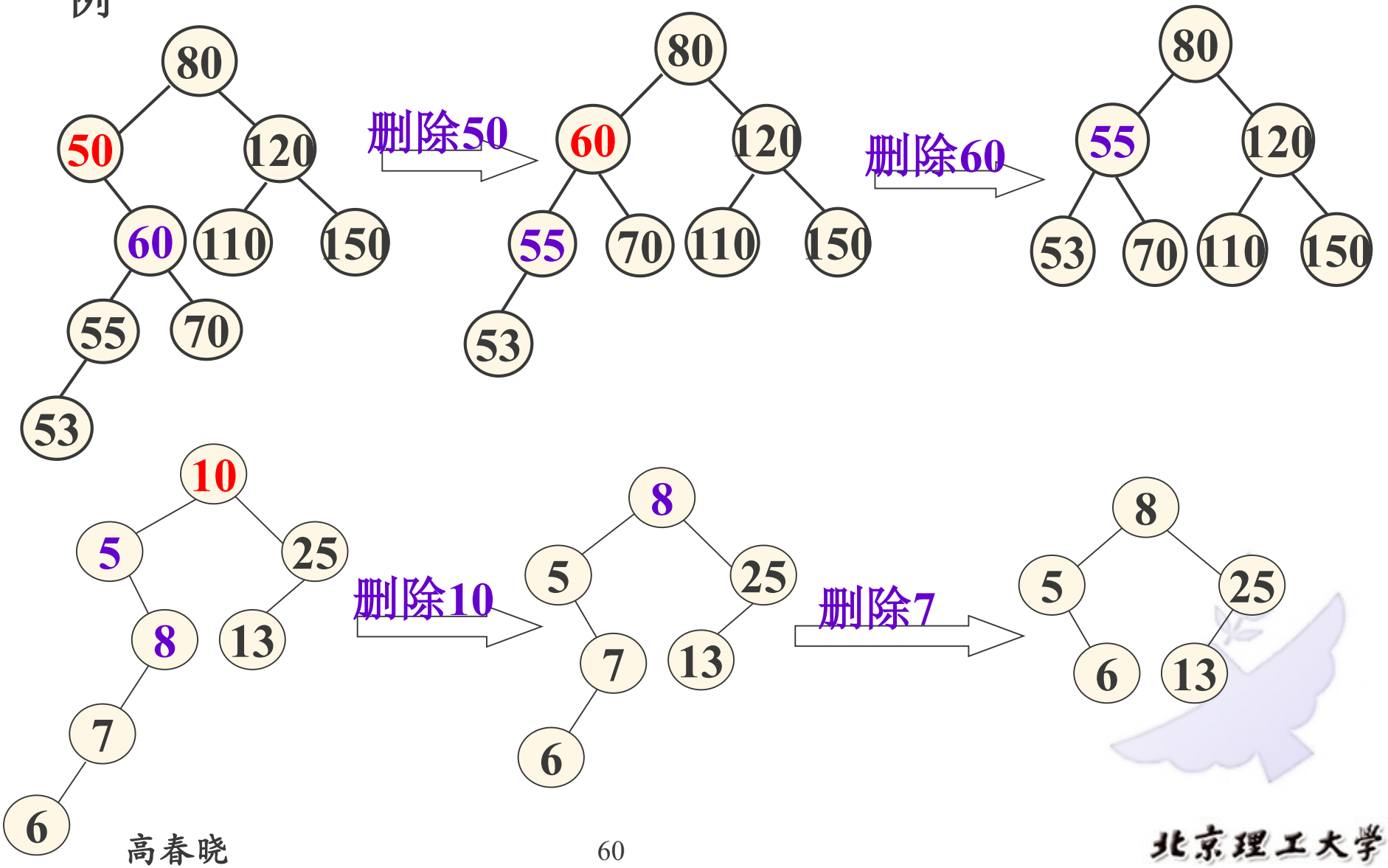
```
//被删结点的左、右子树都不为空
  if ( p->lchild != NULL && p->rchild != NULL ) {
      //找 p 的中序前驱s，f指向s的父节点
      s = p->lchild;  f = p;
      while ( s->rchild != NULL ) {
            f = s;  s = s->rchild;
      }
      p->data = s->data;  //将*s的数据传给*p
      p = s;    // p指向真正要删除的结点
 } // if
```

```
//被删结点的右子树为空，或者左子树为空，或都为空
// s记录p的唯一子树，若p为叶子结点则s为空
    if ( p->lchild != NULL ) s = p->lchild;
    else s = p->rchild;
    if ( p == t ) t = s;              //被删结点为根结点
    else if ( p->data < f->data )
            f->lchild = s;            //p是f的左子树
    else f->rchild = s;  //p是f的右子树
    free ( p );                       //释放被删结点
    return true;                      //删除成功
}
```



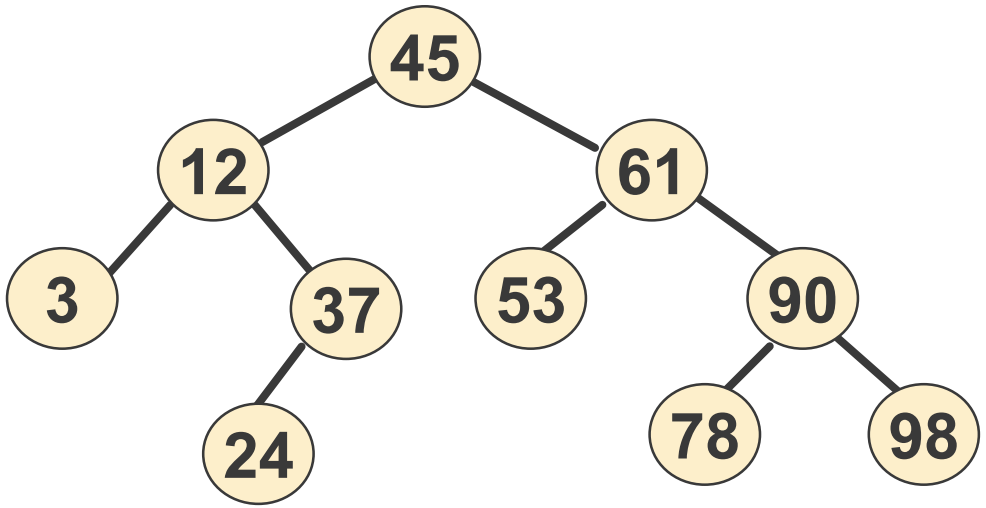
◆ 二叉查找树的删除操作

例

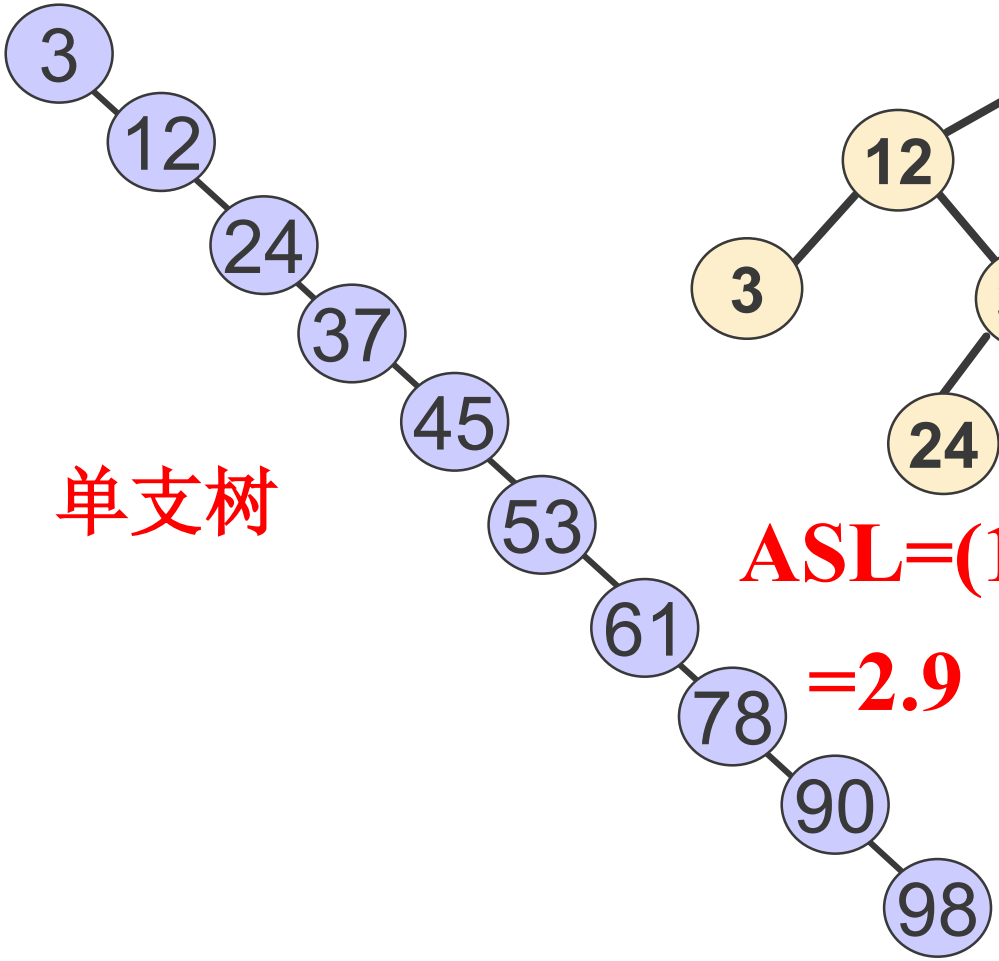


二叉查找树查找性能的分析

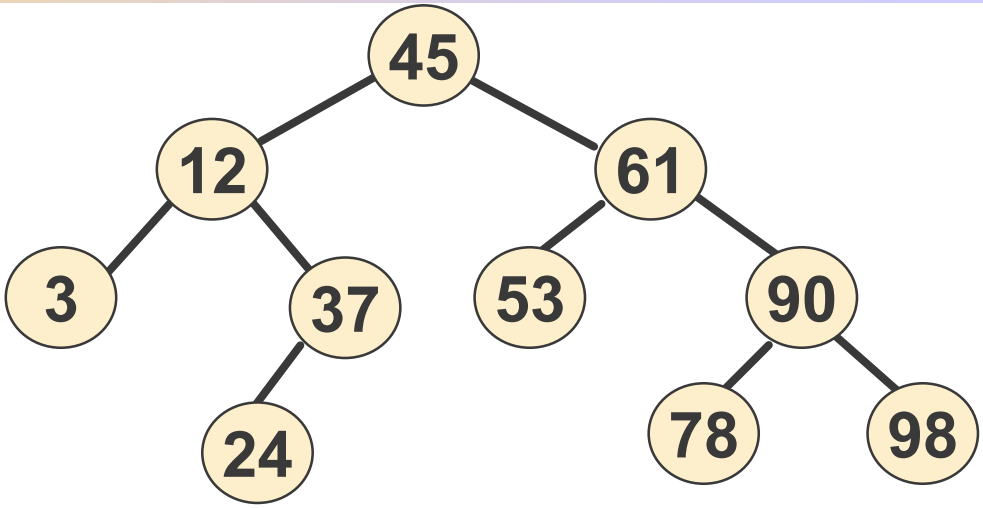
- 对于每一棵特定的二叉查找树, 均可按照平均查找长度的定义来求它的 ASL 值.
- 由值相同的 n 个关键字, 其关键码有 $n!$ 种不同排列, 可构成不同二叉查找树有 $\frac{1}{n+1}C_{2n}^{n}$



查找表：{3, 12, 24, 37, 45, 53, 61, 78, 90, 98}



单支树



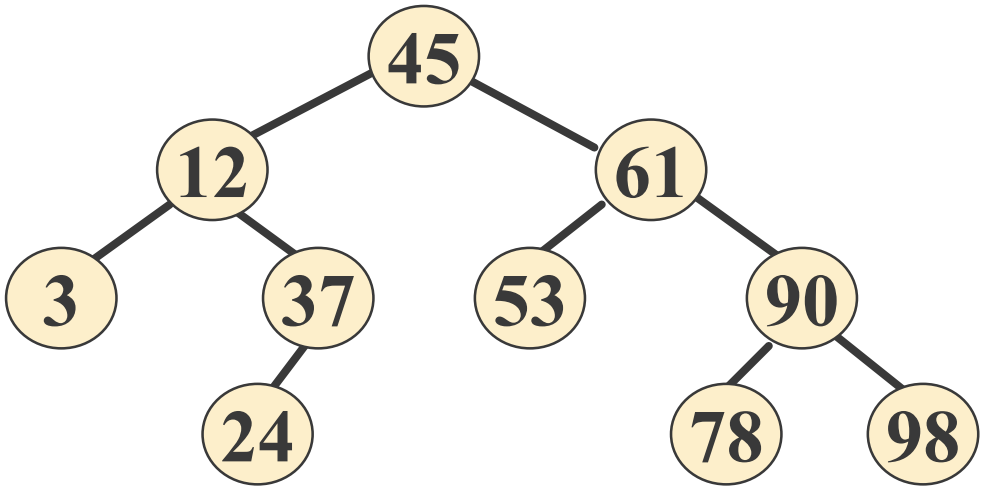
ASL=(1+2×2+3×4+4×3)/10

=2.9

ASL=(1+2+3+4+5+6+7+8+9+10)/10=5.5

二叉查找树查找的特点

- 要求查找表按二叉查找树的形式组织
- 二叉查找树的查找长度与树的形态有关（与树的高度有关）
- 在随机情况下查找、插入、删除的时间复杂度为O($\log_2 n$)；



中序序列：3 12 24 37 45 53 61 78 90 98

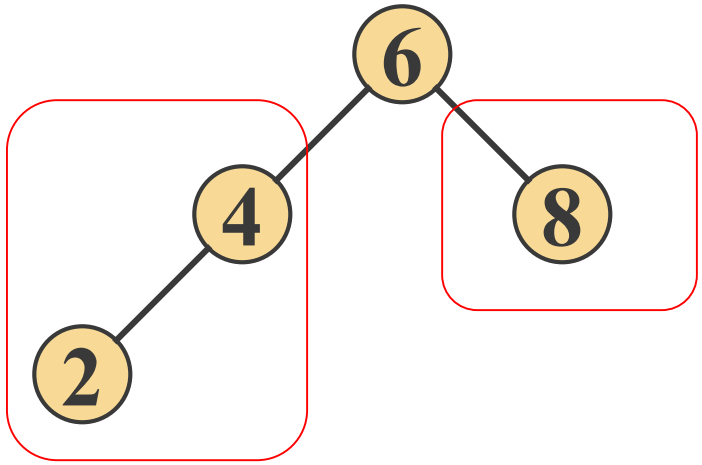
二叉查找树查找的特点

- 二叉查找树的缺点
  - 没有对树的深度进行控制。
- 二叉查找树的适用范围
  - 用于组织规模较小的、内存中可以容纳的数据。对于数据量较大必须存放在外存中的数据，则无法快速处理。
- 在构造二叉查找树的过程中进行“平衡化”处理，成为平衡二叉树（AVL树）。
- 平衡二叉树：左子树和右子树的深度之差的绝对值不超过预定值。

AVL树在1962年由Adelson-Velskii和Landis提出。

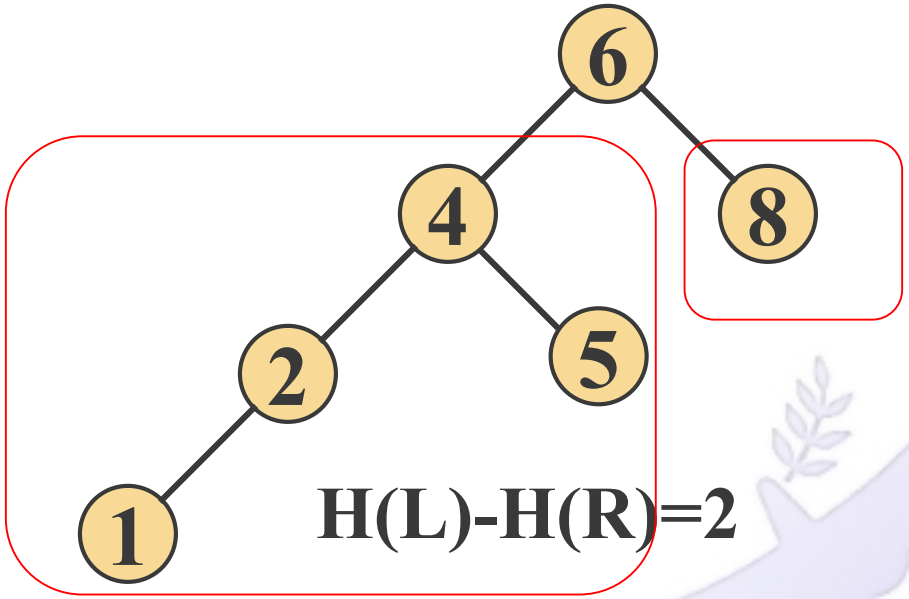
7.3 平衡二叉树

- 定义：平衡二叉树是二叉查找树的另一种形式, 其特点为：
- 树中每个结点的左、右子树深度之差的绝对值不大于1：| H(L)-H(R) | <=1



H(L)-H(R)=1

平衡二叉树

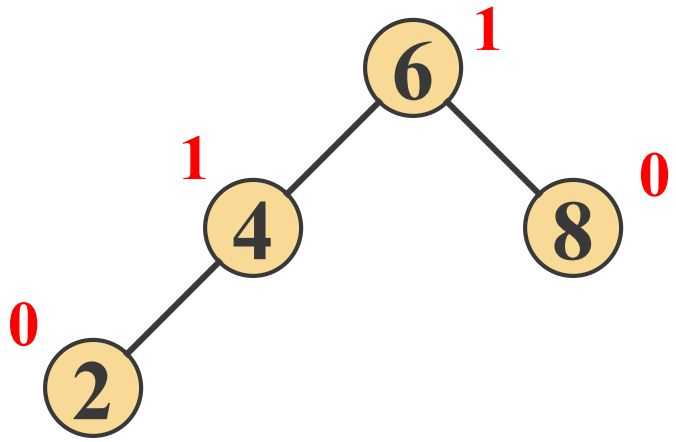


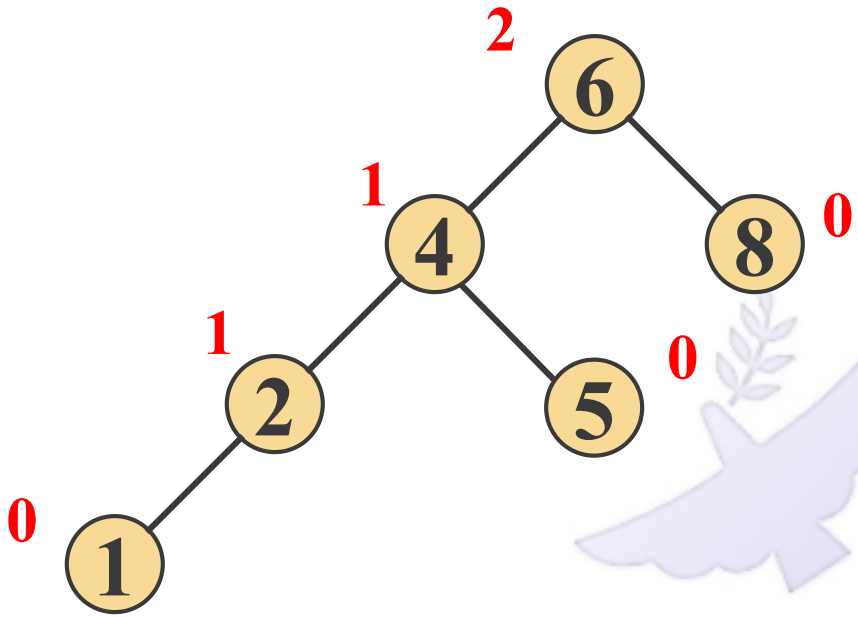
H(L)-H(R)=2

非平衡二叉树

结点的平衡因子 balance factor

- 结点的平衡因子 bf（balance factor）| H(L)-H(R) |
- AVL树任一结点平衡因子只能取 -1, 0, 1。
- 如果一个结点的平衡因子的绝对值大于 1，则这棵二叉查找树就失去了平衡，不再是AVL树。

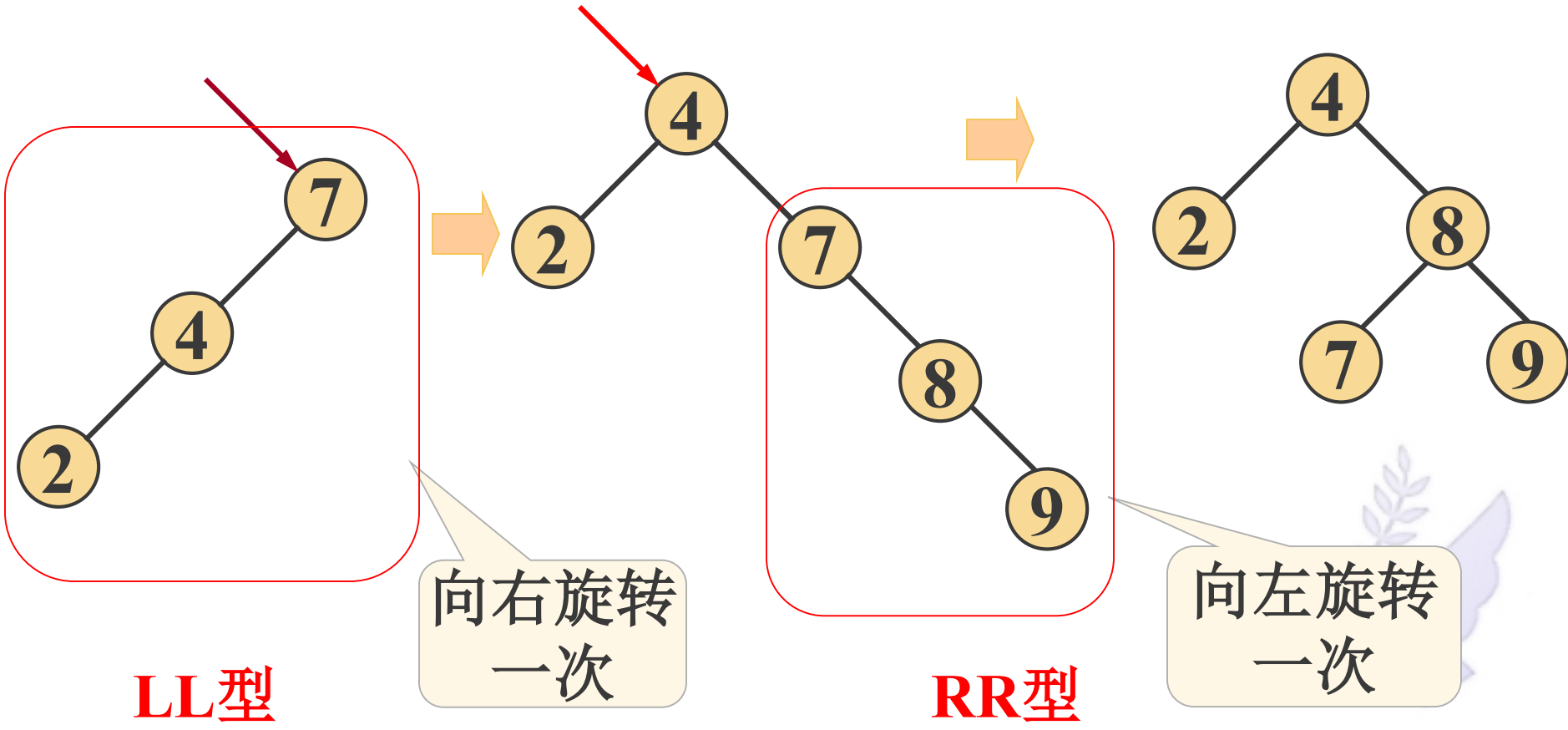




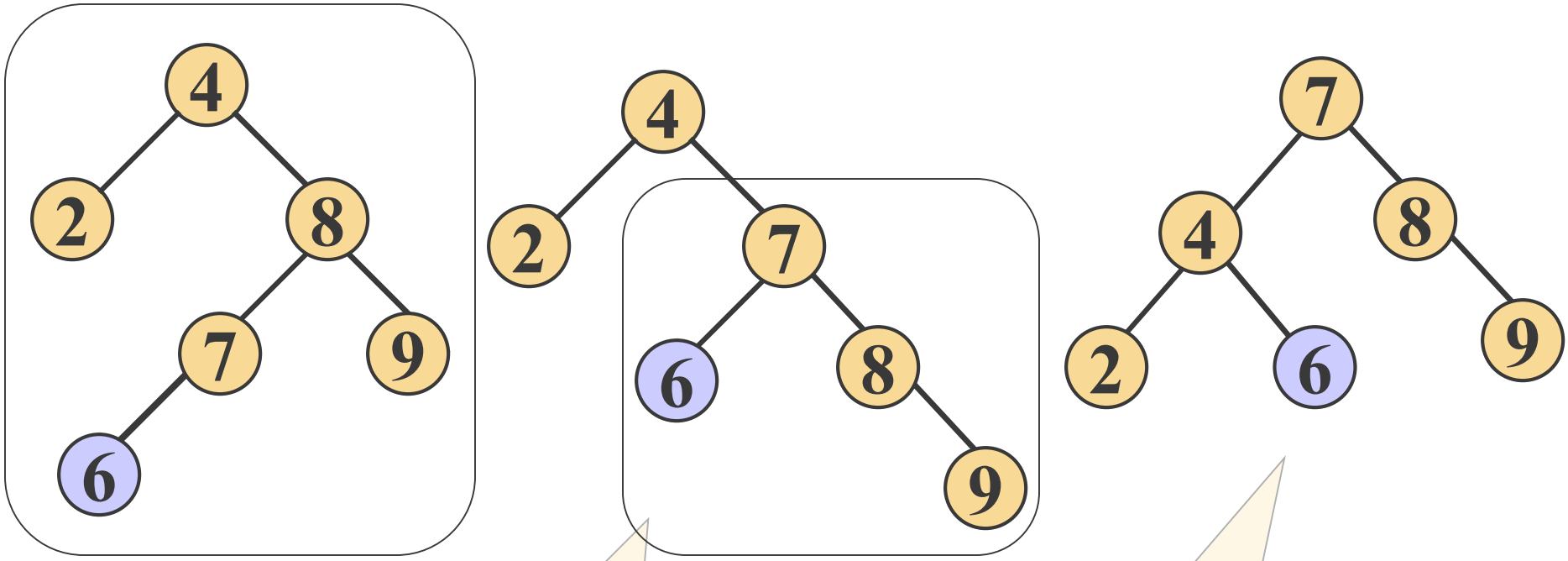
构造平衡二叉树

- 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7，4，2，8，9，6



例如:依次插入的关键字为7，4，2，8，9，6



RL型

右子树先向右旋转

整棵树再向左旋转

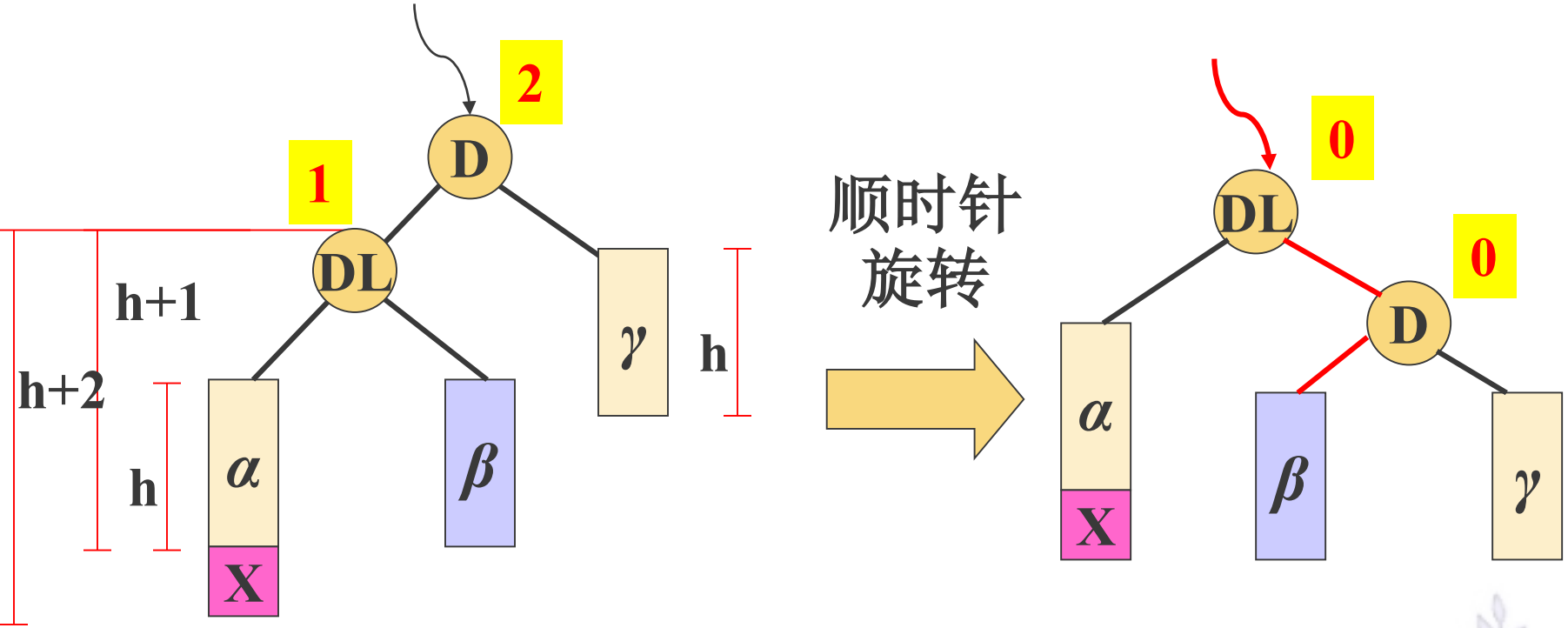
二叉查找树的平衡旋转

- 在插入一个新结点后，从插入位置沿通向根的路径回溯，检查各结点的平衡因子。
- 如果在某一结点发现高度不平衡，停止回溯。
- 从发生不平衡的结点起，沿刚才回溯的路径取直接下两层的结点。判断旋转操作的类型。

- **LL型平衡旋转（单向旋转）**
- **RR型平衡旋转（单向旋转）**
- **LR型平衡旋转（双向旋转）**
- **RL型平衡旋转（双向旋转）**

LL型平衡旋转

以D为根的子树不平衡：左子树的左子树造成的

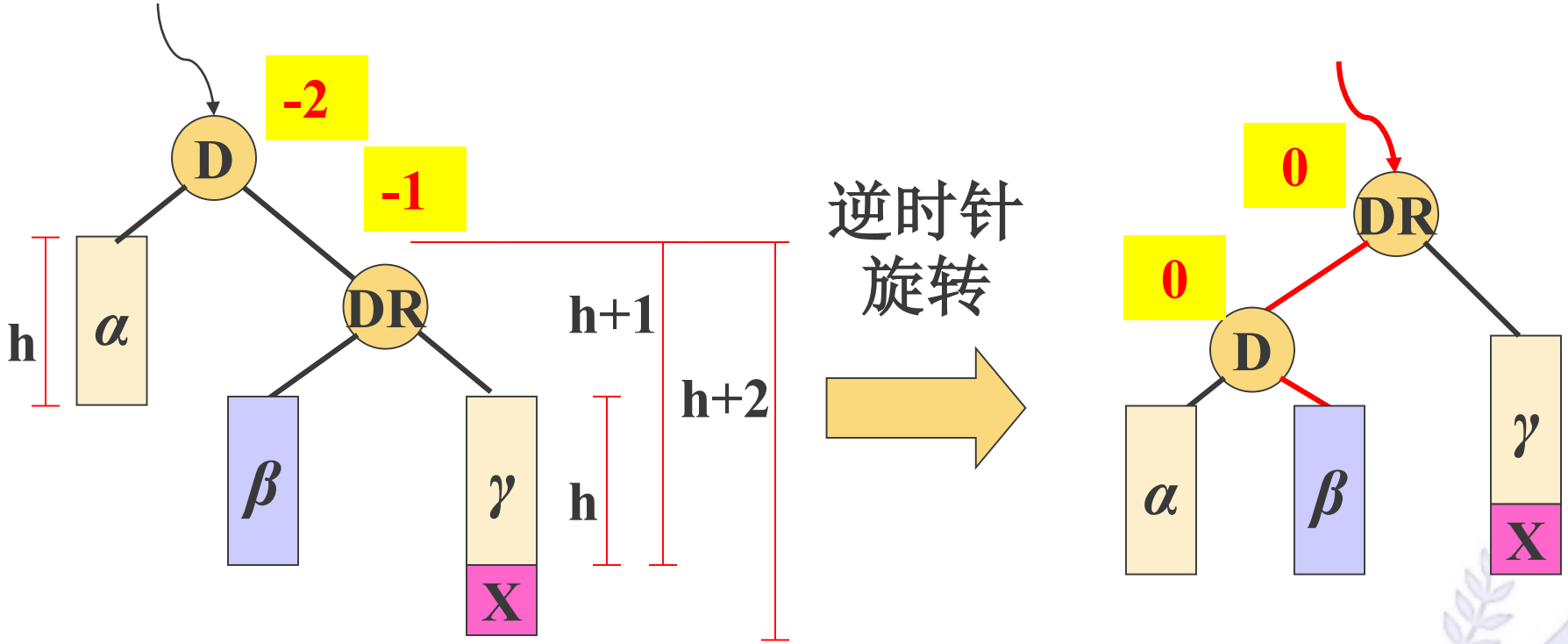


H(L) – H(R) = 2, 不平衡！

LL型：整棵树向右旋转一次

RR型平衡旋转

以D为根的子树不平衡：右子树的右子树造成的

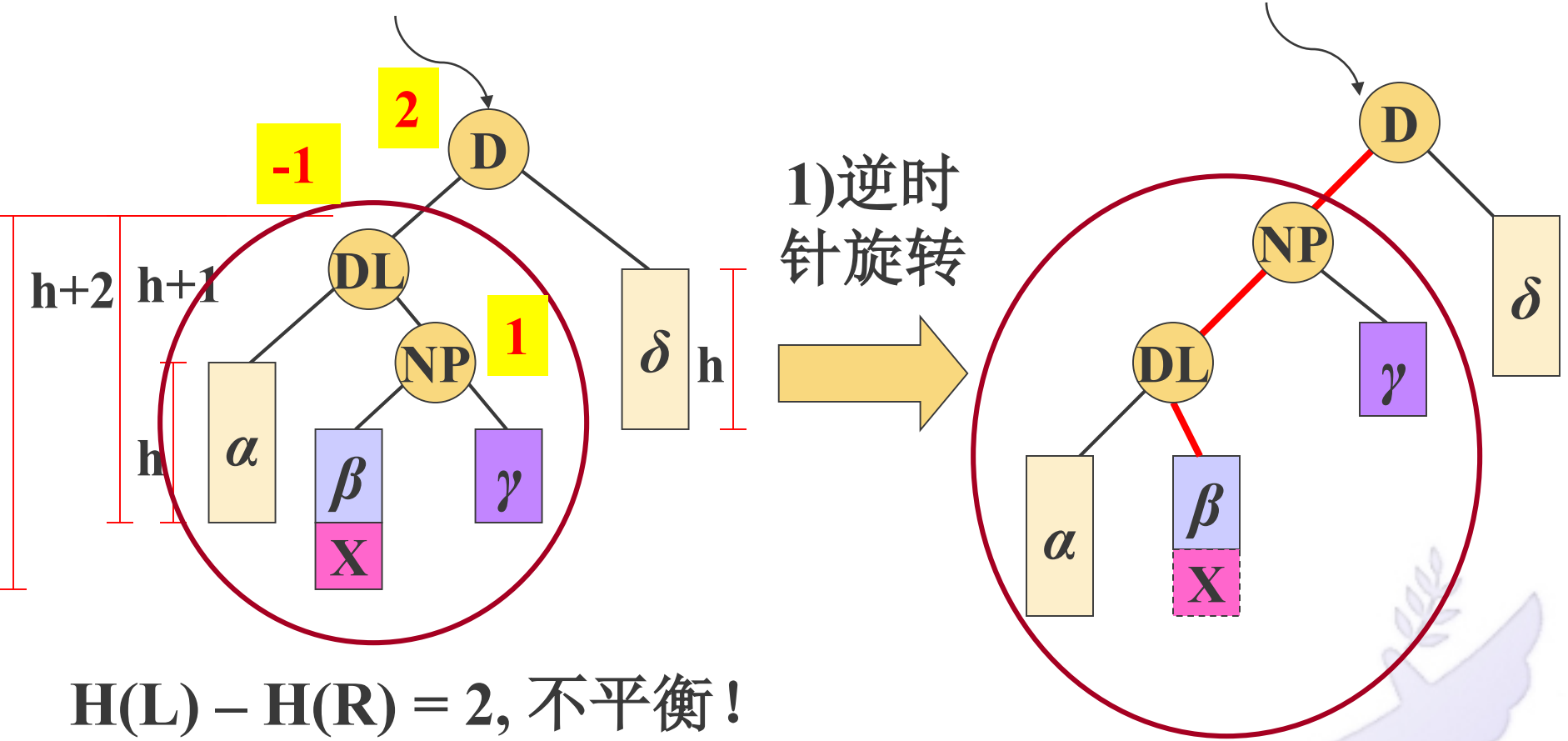


H(L) – H(R) = -2, 不平衡！

RR型：整棵树向左旋转一次

LR型平衡旋转

以D为根的子树不平衡：左子树的右子树造成的

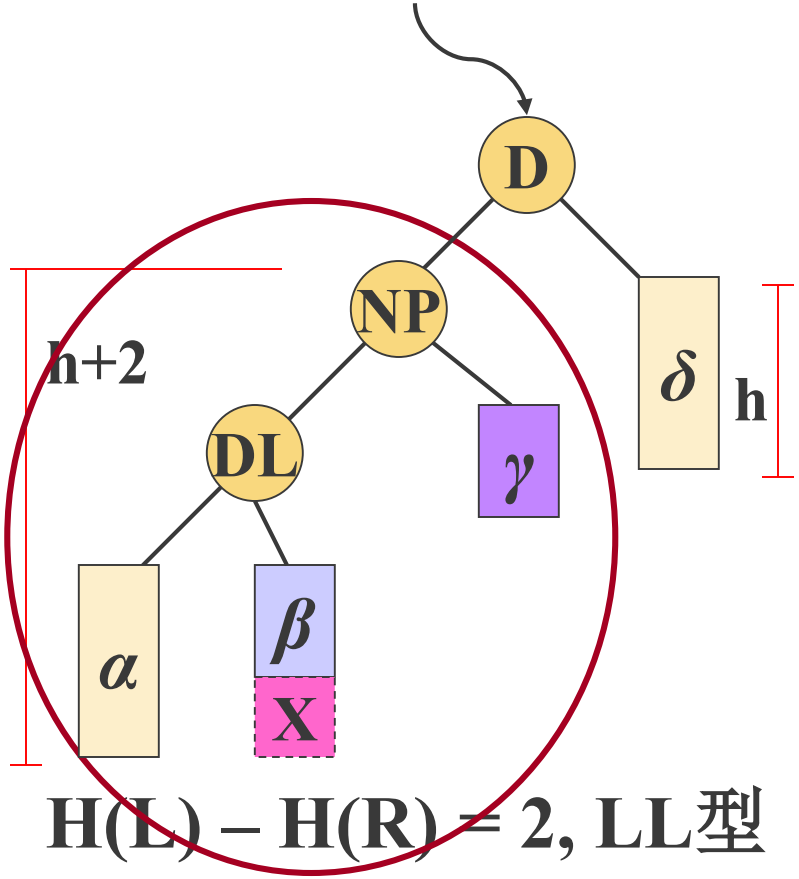


H(L) – H(R) = 2, 不平衡！

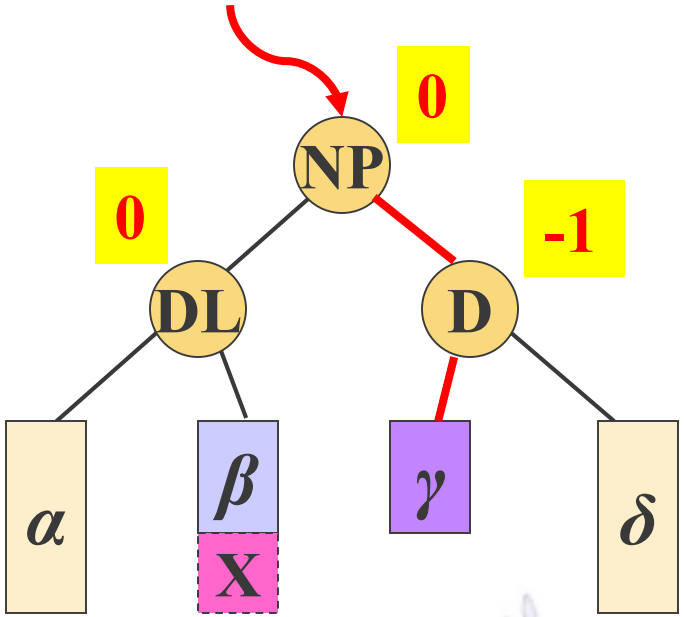
LR型：先向左旋转, 再向右旋转

LR型平衡旋转

以D为根的子树不平衡：左子树的右子树造成的



2)顺时针旋转

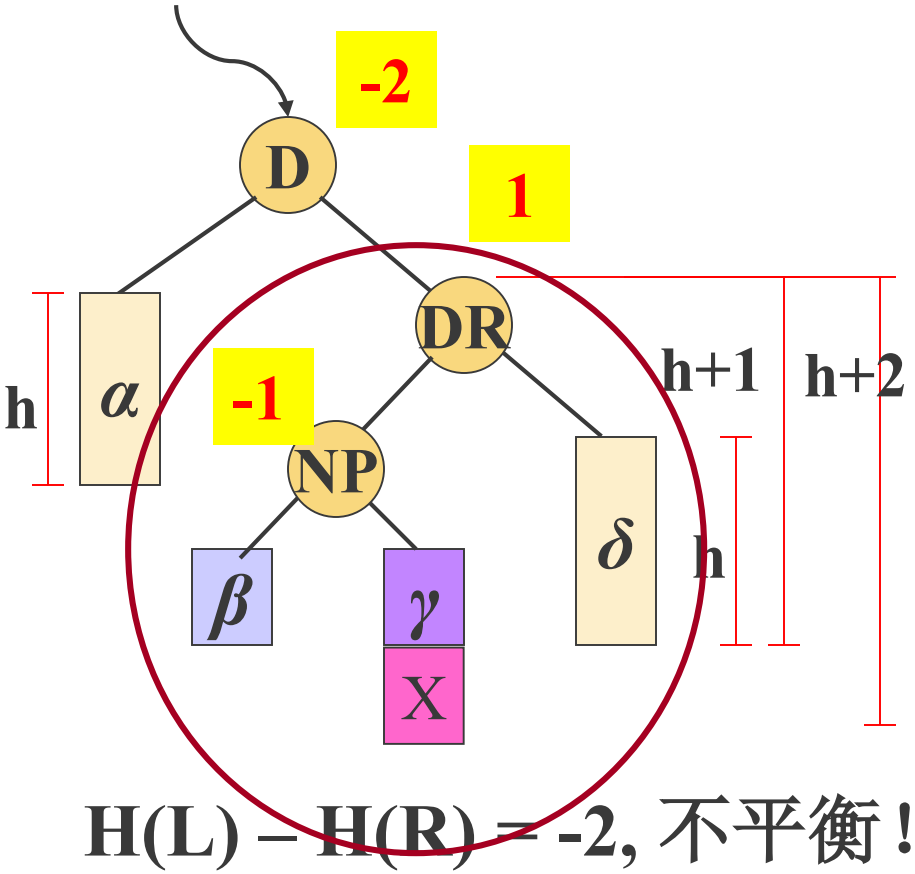


H(L) – H(R) = 2, LL型

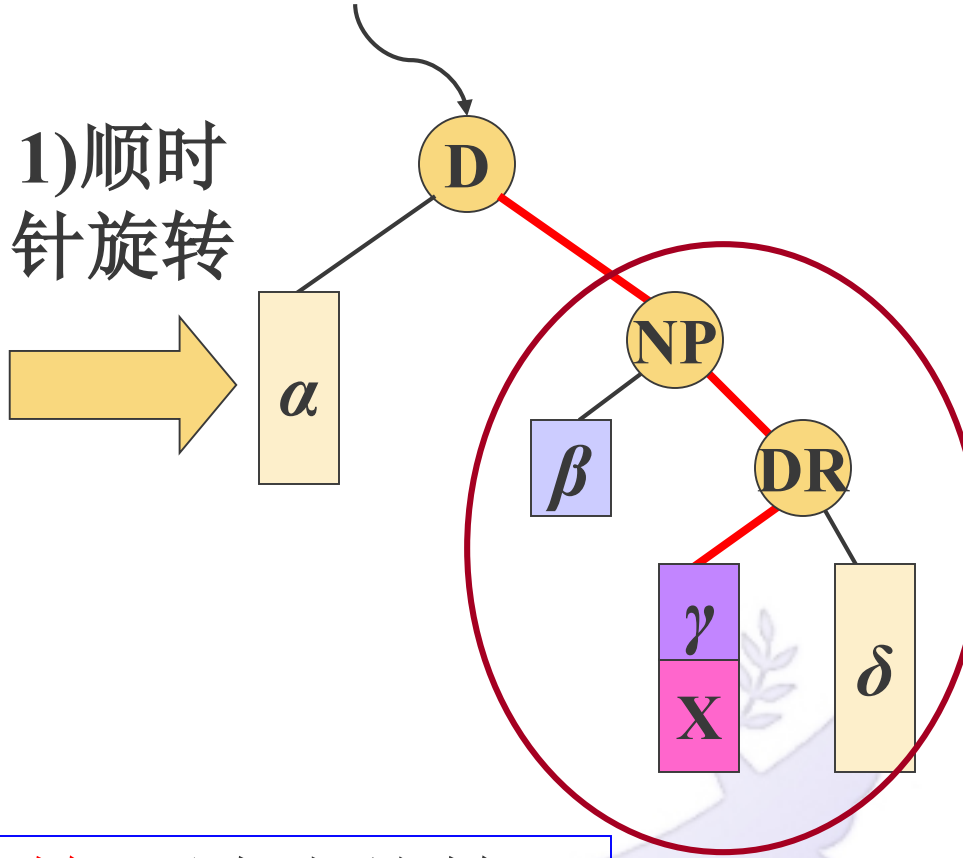
LR型：先向左旋转, 再向右旋转

RL型平衡旋转

以D为根的子树不平衡：右子树的左子树造成的



1)顺时针旋转

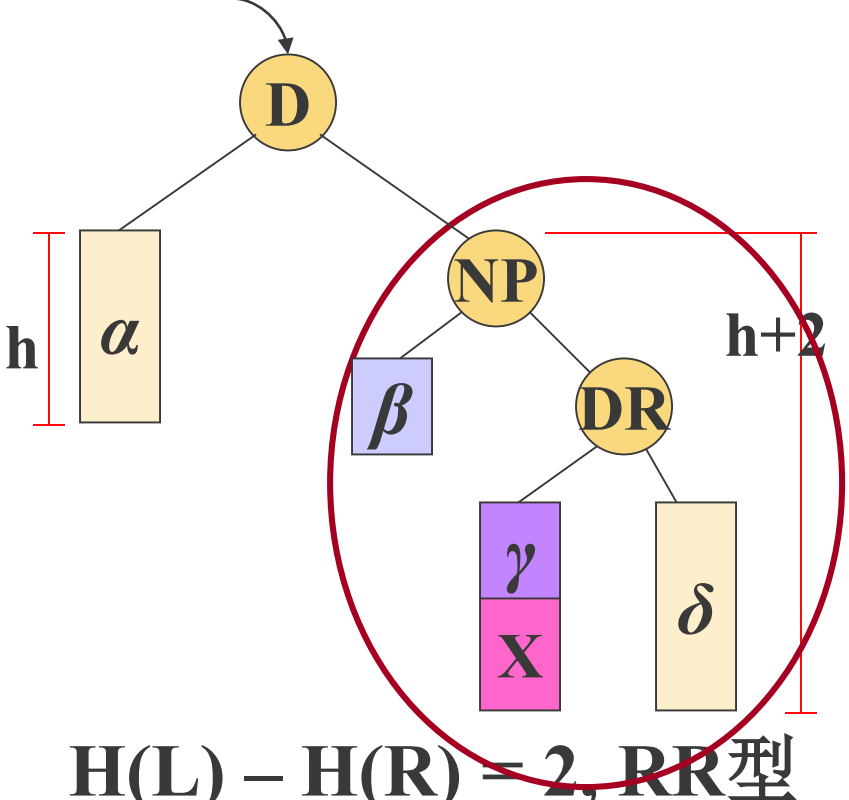


H(L) – H(R) = -2, 不平衡！

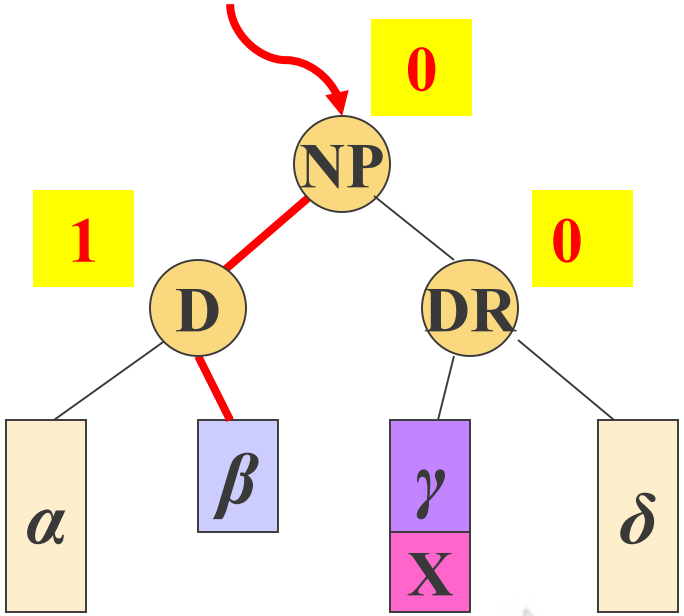
RL型先向右旋转, 再向左旋转

RL型平衡旋转

以D为根的子树不平衡：右子树的左子树造成的



2)逆时针旋转



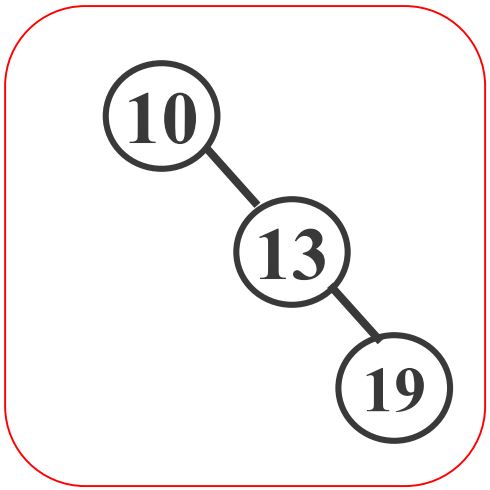
H(L) – H(R) = 2, RR型

RL型先向右旋转, 再向左旋转

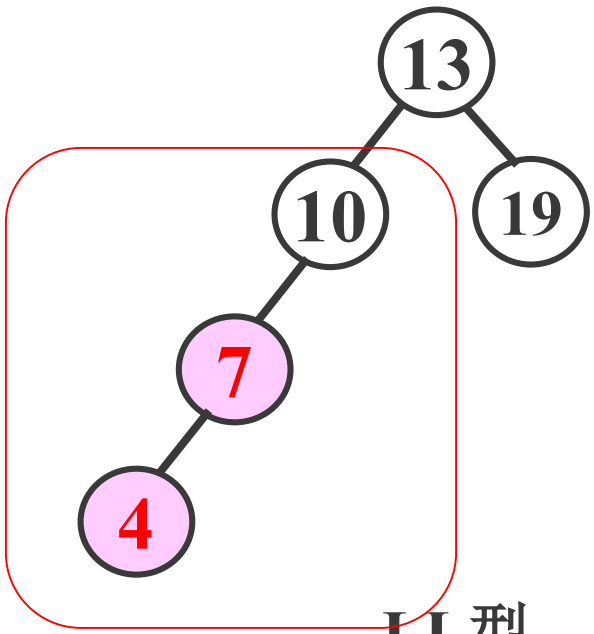
建立AVL树

- 例题:按照如下顺序建立平衡二叉树:
- 10, 13, 19, 7, 4, 8, 15, 24, 33, 21

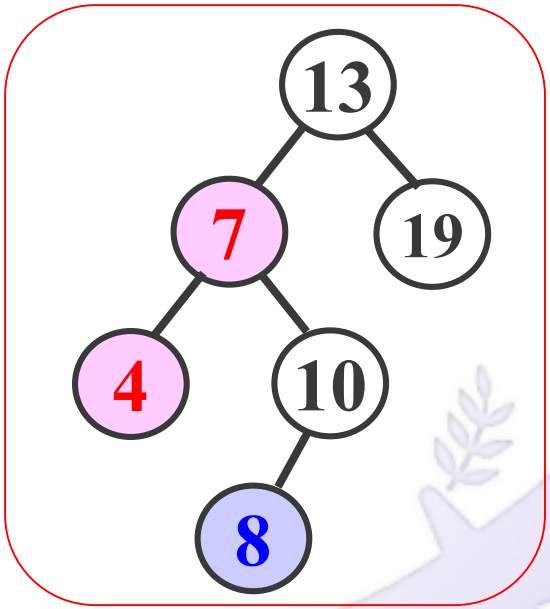
插入10、13、19、7、4、8



RR型



LL型

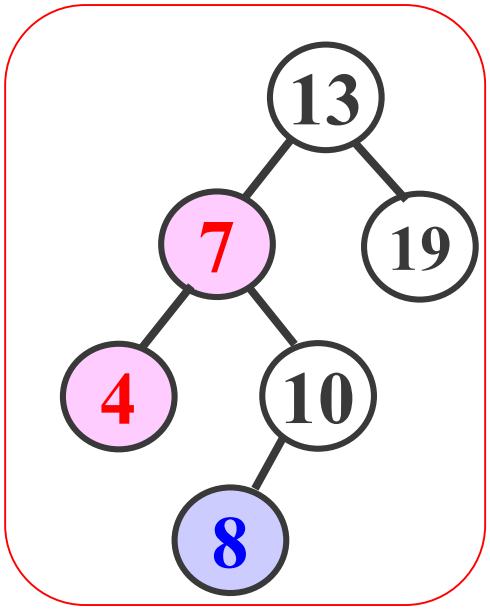


LR型

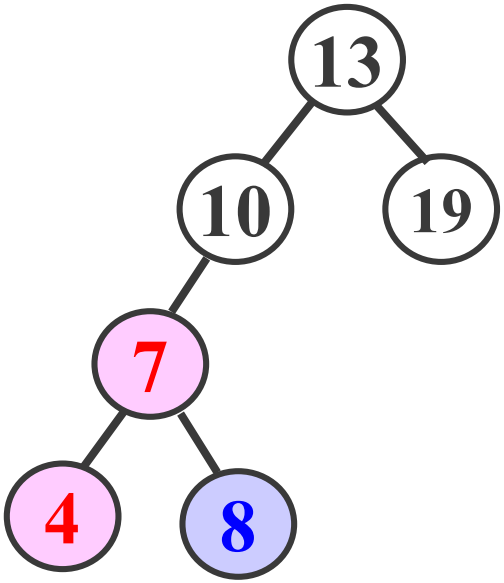
建立AVL树

- 例题:按照如下顺序建立平衡二叉树:
- 10, 13, 19, 7, 4, 8, 15, 24, 33, 21

插入10、13、19、7、4、8、15

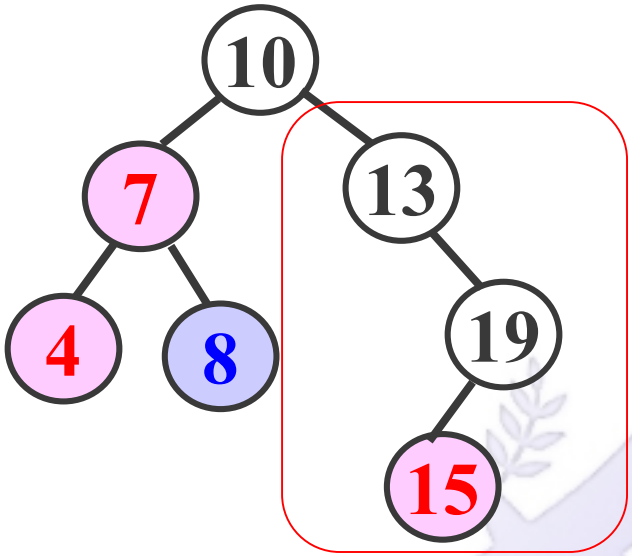


LR型



先左旋

左子树左旋



再右旋

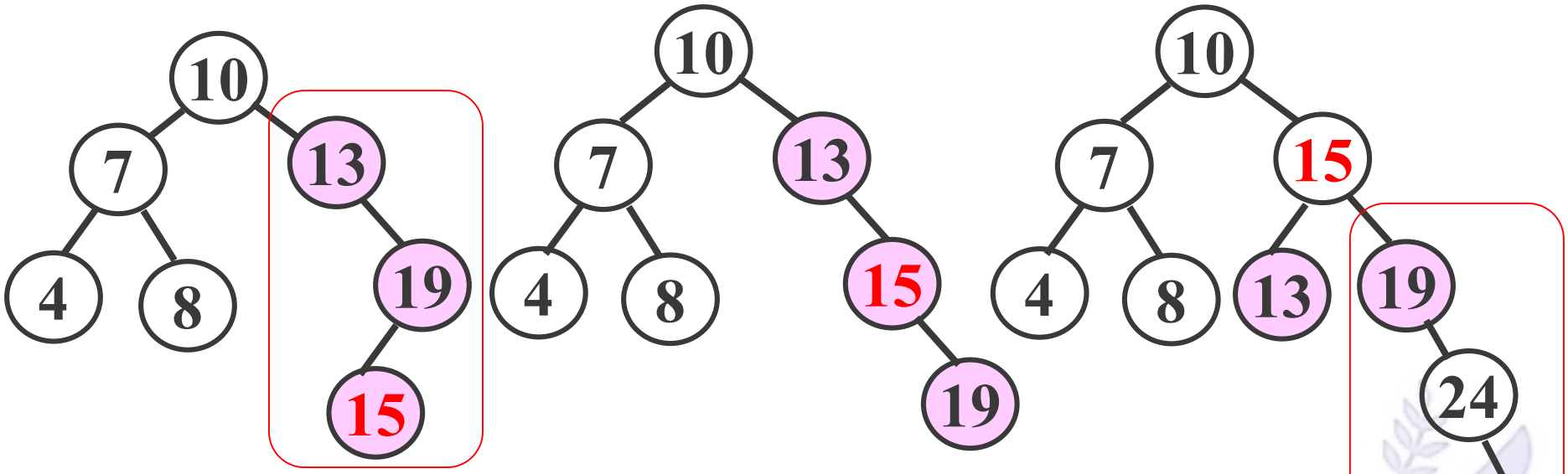
整体右旋

RL型

建立AVL树

- 例题:按照如下顺序建立平衡二叉树:
- 10, 13, 19, 7, 4, 8, 15, 24, 33, 21

插入10、13、19、7、4、8、15、24、33

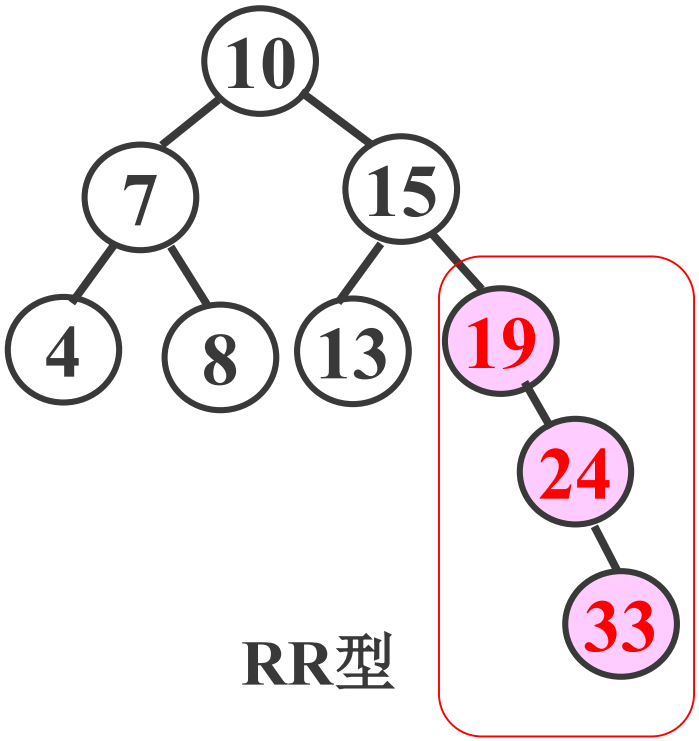


RL型　　先右旋 右子树右旋　　再左旋 整体左旋　　RR型

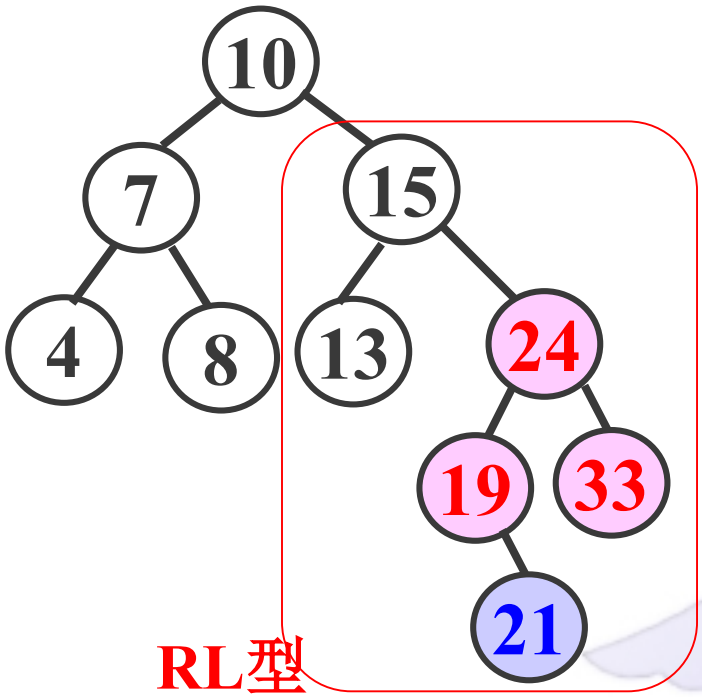
建立AVL树

- 例题:按照如下顺序建立平衡二叉树:
- 10, 13, 19, 7, 4, 8, 15, 24, 33, 21

插入10、13、19、7、4、8、15、24、33、21



RR型

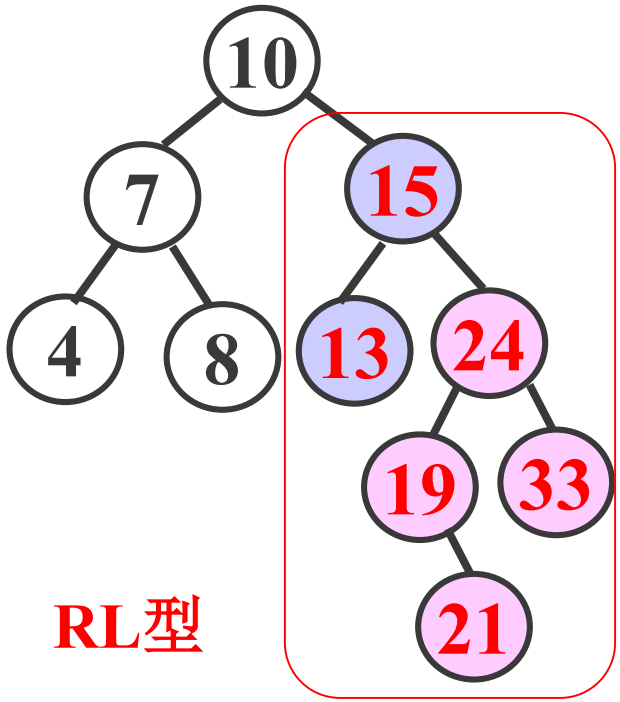


RL型

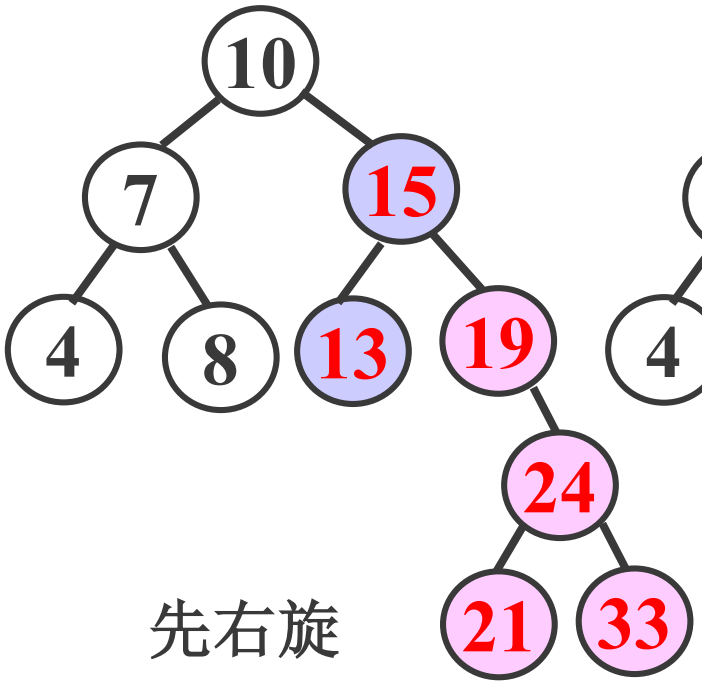
建立AVL树

- 例题:按照如下顺序建立平衡二叉树:
- 10, 13, 19, 7, 4, 8, 15, 24, 33, 21

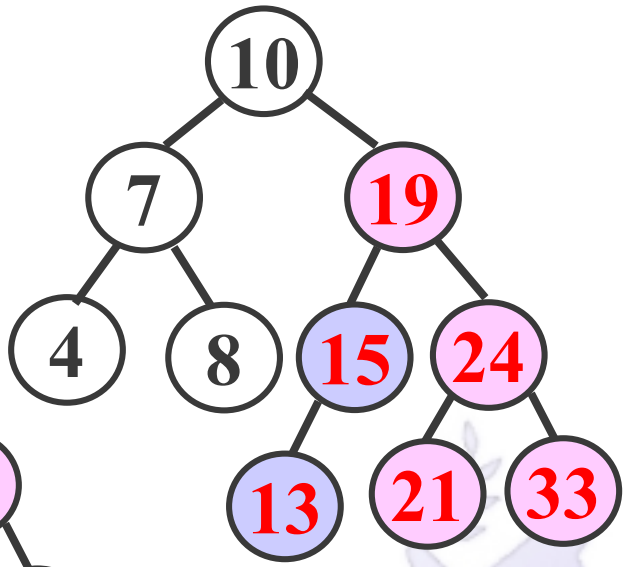
插入10、13、19、7、4、8、15、24、33、21



RL型



先右旋
右子树右旋

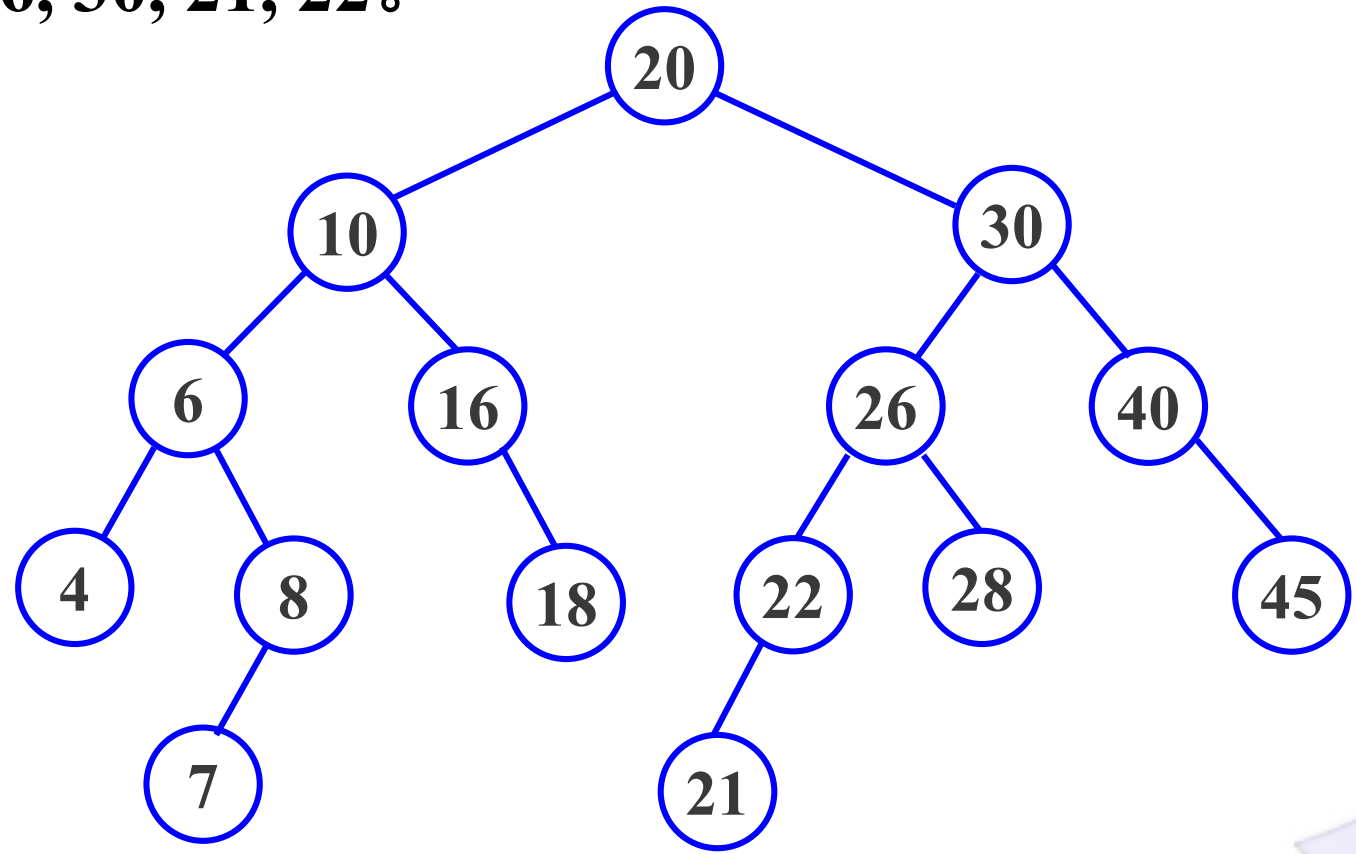


再左旋
整体左旋

AVL树的删除

例：按照如下顺序删除平衡二叉树中的结点：

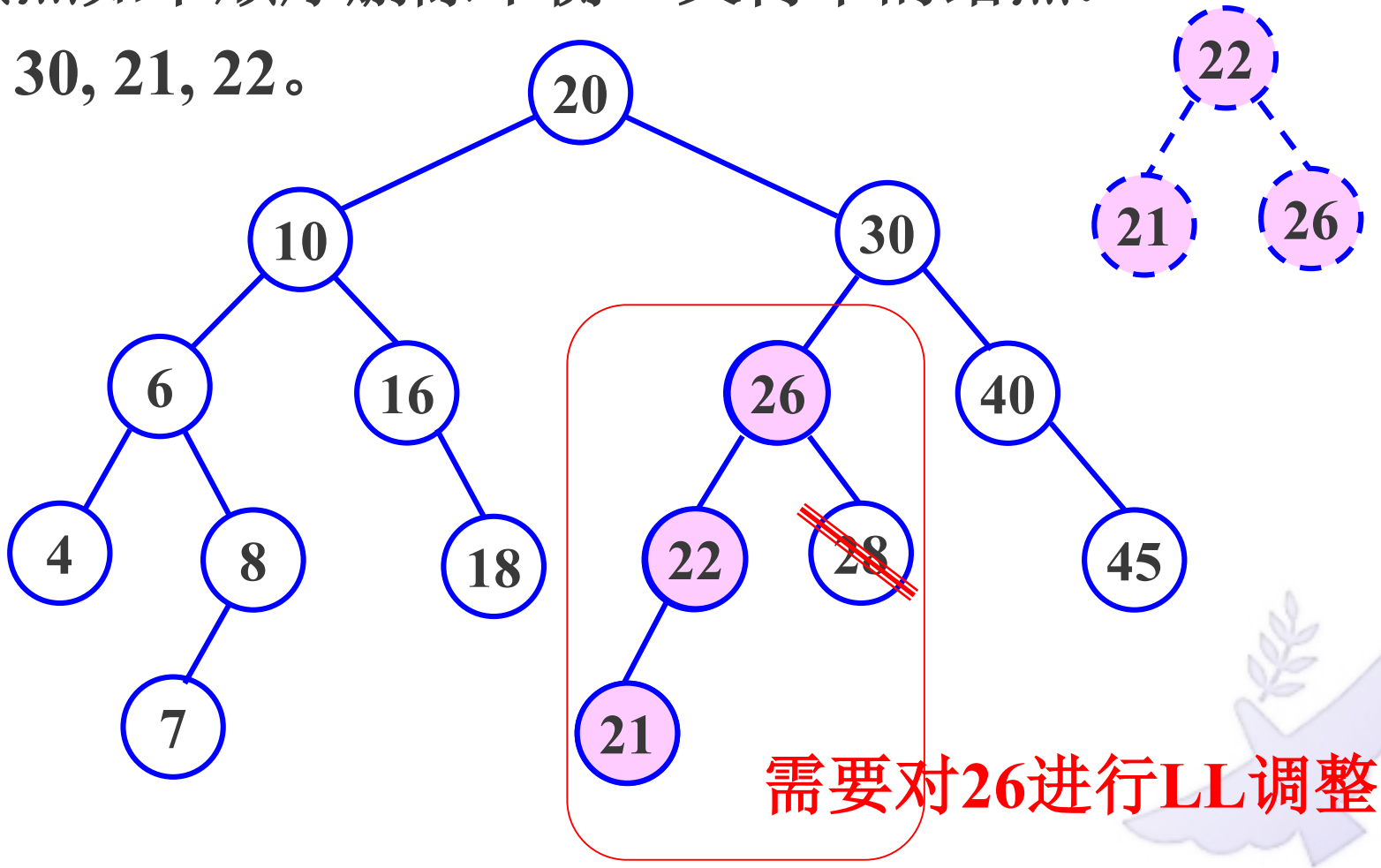
28, 16, 30, 21, 22。



AVL树的删除

例：按照如下顺序删除平衡二叉树中的结点：

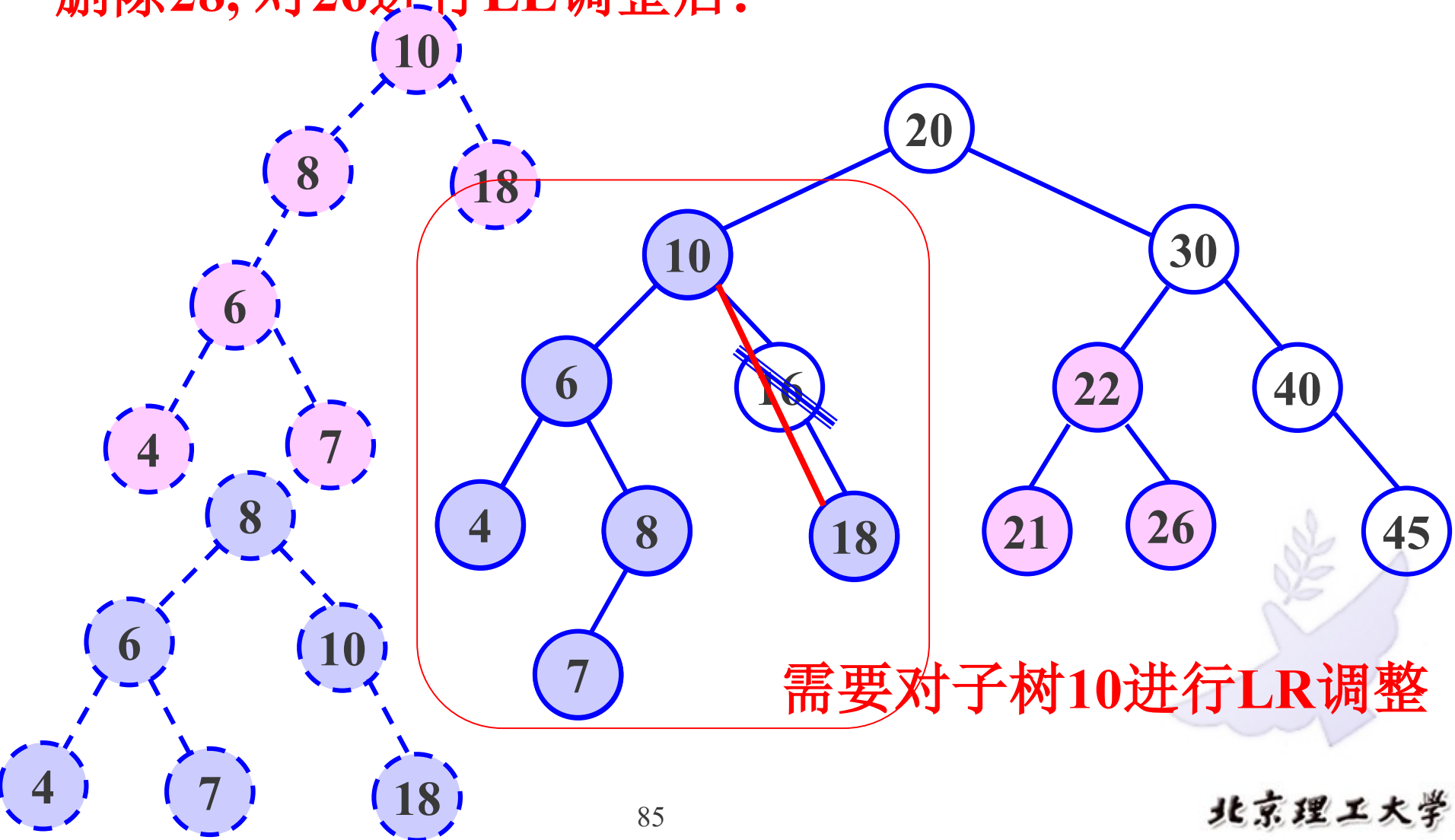
28, 16, 30, 21, 22。



例：按照如下顺序删除平衡二叉树中的结点：

28, 16, 30, 21, 22。

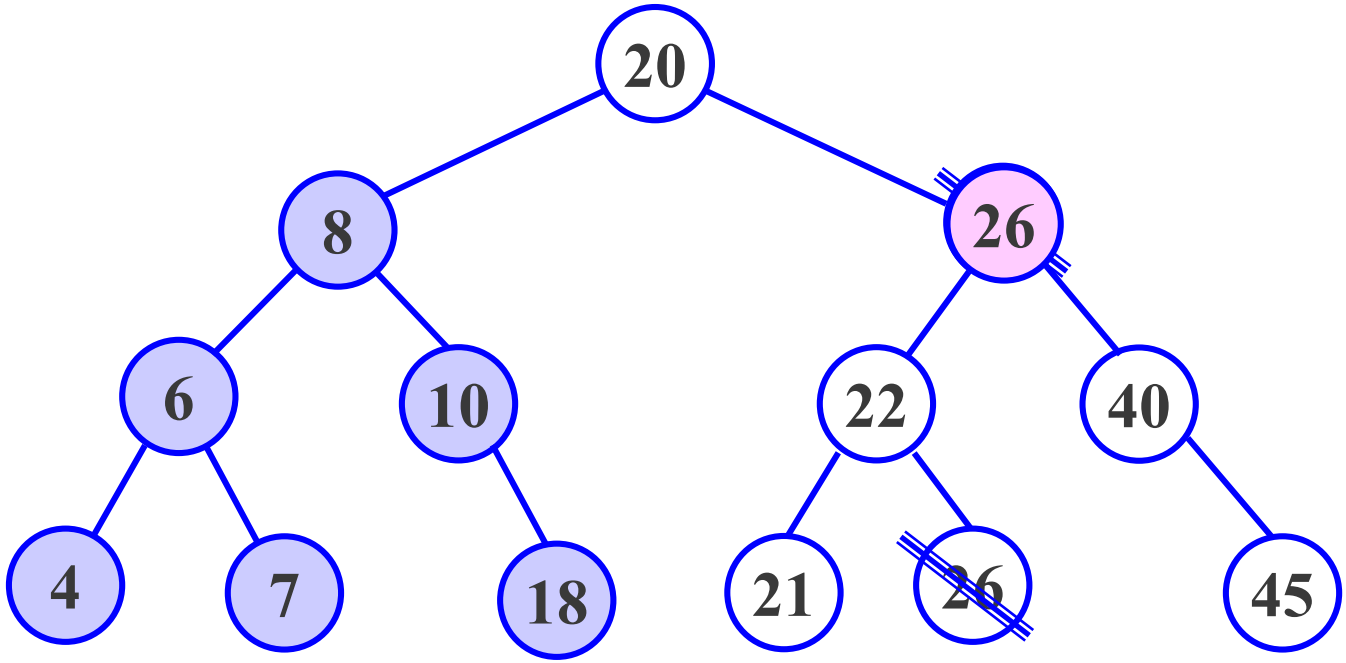
删除28, 对26进行LL调整后：



例：按照如下顺序删除平衡二叉树中的结点：

28, 16, 30, 21, 22。

删除16, 对10进行LR调整后：



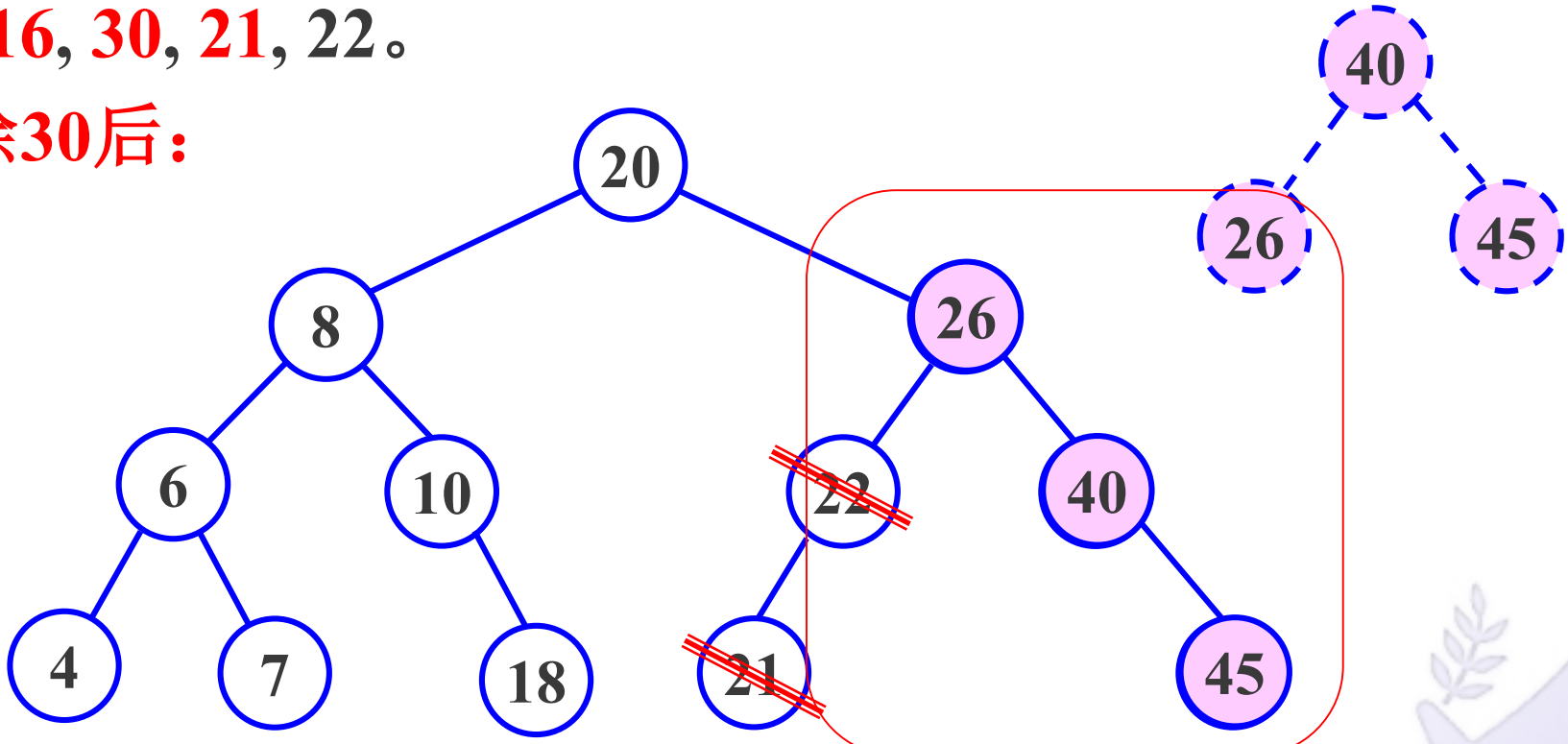
用30的中序前驱26替换30；

并将原来的26结点删除

例：按照如下顺序删除平衡二叉树中的结点：

28, 16, 30, 21, 22。

删除30后：

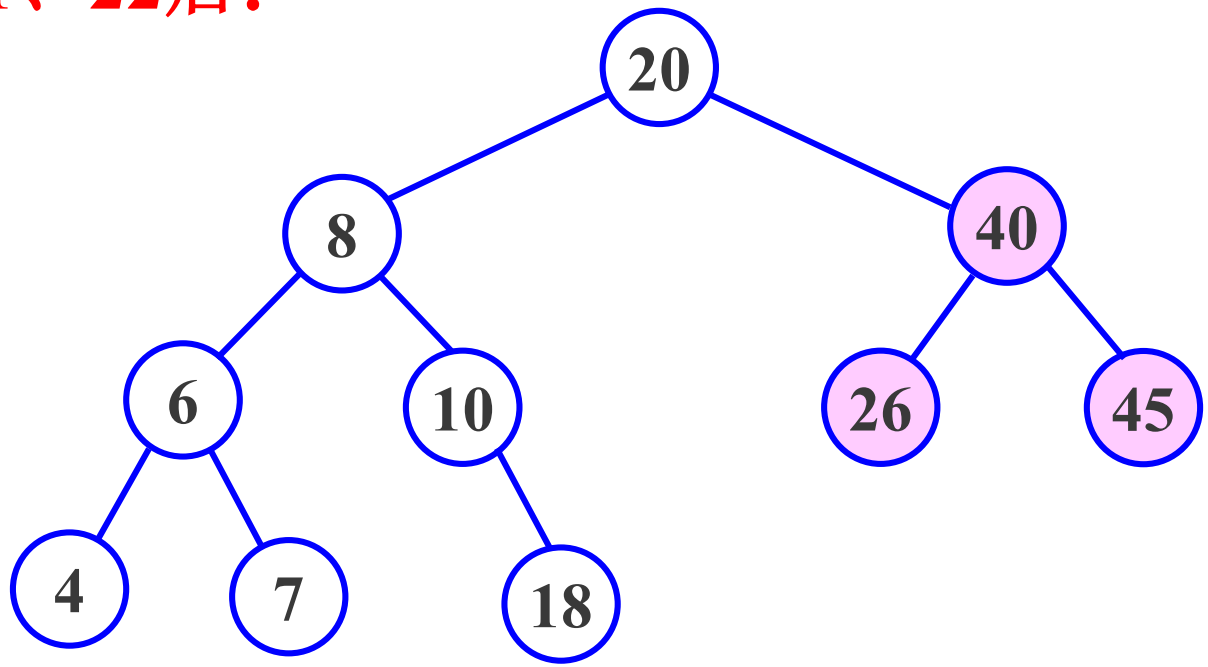


需要对26进行RR调整

例：按照如下顺序删除平衡二叉树中的结点：

28, 16, 30, 21, 22。

删除21、22后：

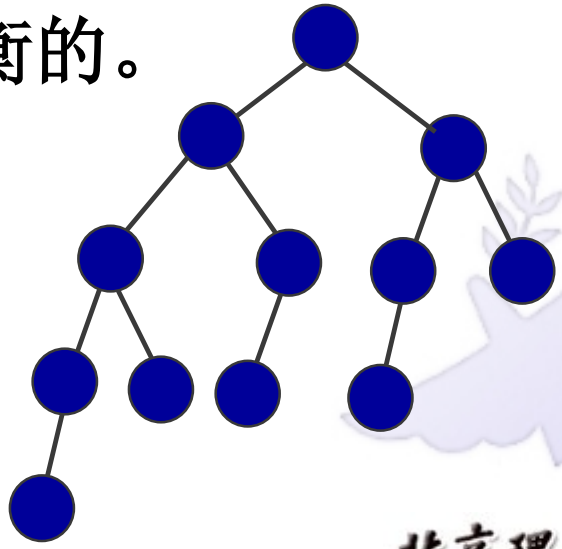


平衡二叉树查找分析

- 在平衡二叉树上查找的算法与排序二叉树相同
- 查找过程中的比较次数不超过树的深度
- 所以在平衡二叉树上查找的时间复杂度为$O(\log_2 n)$

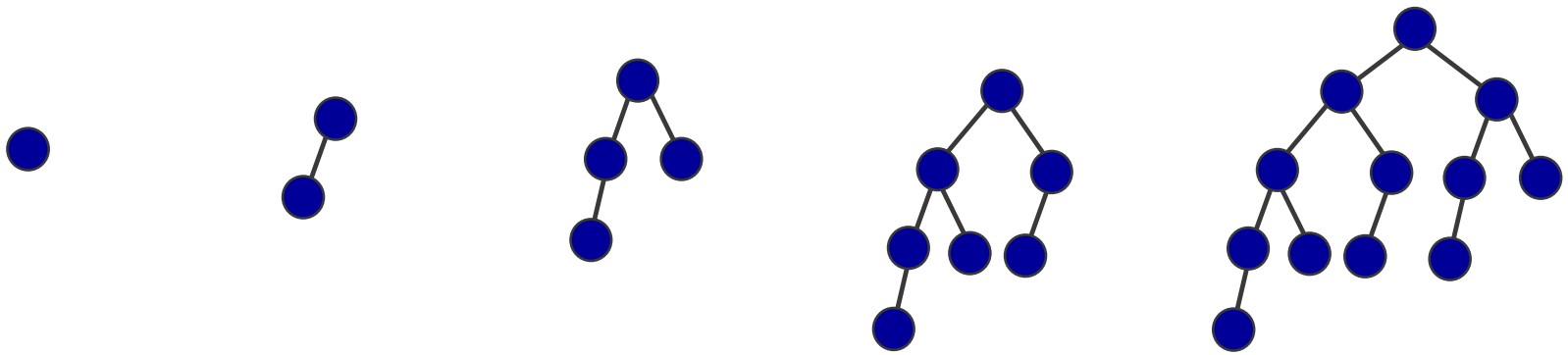
AVL树的高度

- 设在新结点插入前AVL树的高度为h，结点个数为n，则插入一个新结点，其时间代价是O(h)。
- 对于有n个结点的AVL树来说，h多大？
- 高度为 h 的AVL树最多结点数为$2^h - 1$，即满二叉树情形。
- 设 N_h 是高度为 h 的AVL树的最少结点个数。
- 左右子树高度相差1的树是平衡的。



AVL树的高度

- 例：具有 5 层结点的AVL树至少有多少个结点？
- 解：左右子树高度相差1的树是平衡的。



$N_1=1$　　$N_2=2$　　$N_3=4$　　$N_4=7$　　$N_5=13$

- $N_1 = 1$，$N_2 = N_1+N_0+1 = 2$
- $N_3 = N_2+N_1+1 = 4$，$N_4 = N_3+N_2+1 = 7$
- ➔$N_h = N_{h-1} + N_{h-2} +1$, $h > 1$

AVL树的高度

- ALV结点数：
  - $N_0 = 0$，$N_1 = 1$，$N_2 = N_1+N_0+1 = 2$，
  - $N_h = N_{h-1} + N_{h-2} + 1$
- 斐波那契数列
  - $F_0 = 0$，$F_1 = 1$，$F_2 = F_1+F_0 = 1$，$F_3 = F_2+F_1 = 2$，
  - $F_h = F_{h-1}+F_{h-2}$
- 可得 $N_h = F_{h+2} - 1$。

$$F_{h+2} = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{h+2} - \left(\frac{1-\sqrt{5}}{2}\right)^{h+2}\right) > \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+2} - 1$$

$$N_h > \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+2} - 2 \qquad \Phi = \frac{1+\sqrt{5}}{2}$$

$$\Phi^{h+2} < \sqrt{5}(N_h + 2) \quad h + 2 < log_{\Phi}\sqrt{5} + log_{\Phi}(N_h + 2)$$

$$log_{\Phi}X = log_2 X / log_2 \Phi \qquad log_2 \Phi = 0.694$$

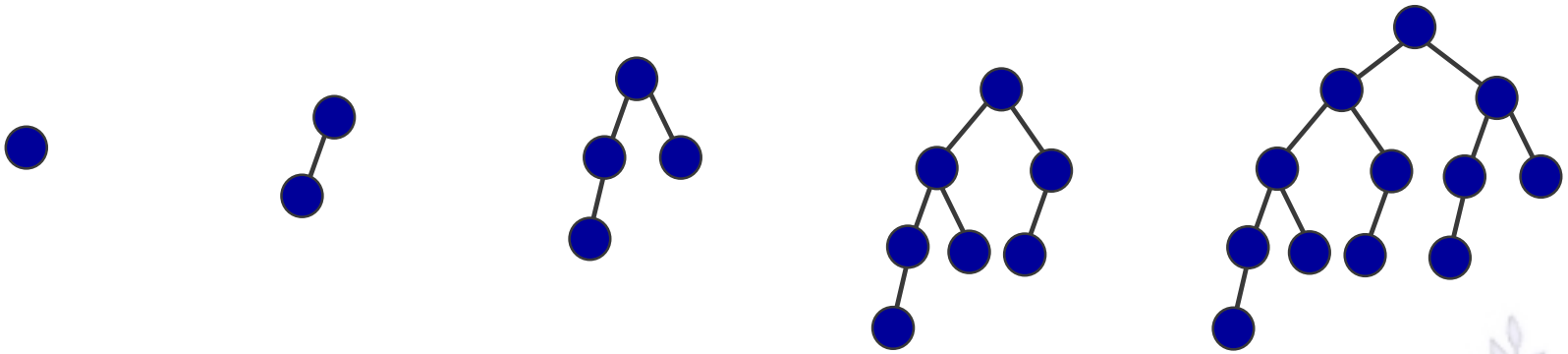
$$h + 2 < \frac{log_2\sqrt{5}}{log_2\Phi} + \frac{log_2(N_h + 2)}{log_2\Phi}$$

$$= 1.6723 + 1.4404 \times log_2(N_h + 2)$$

$$h_{max} = \lceil 1.44 log_2(n + 1) - 0.327 \rceil$$

AVL树的高度

- 因此，若AVL树有n个结点
  - ¶ 最小高度为$\lceil \log_2(n+1) \rceil$。
  - ¶ 最大高度不超过$\lceil 1.44log_2(n+1) - 0.327 \rceil$。



N_1=1　　N_2=2　　N_3=4　　N_4=7　　N_5=13

7.4 B树

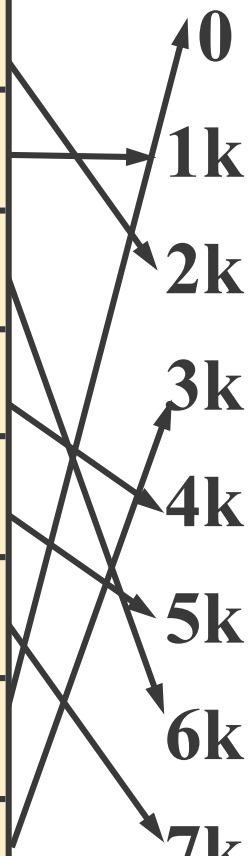
- **B-树： Balanced Tree**
- **B-树在Rudolf Bayer, Edward M. McCreight(1970)的论文《Organization and Maintenance of Large Ordered Indices》中提出的。**
- **B-树: 平衡多路查找树**

wikipedia：
http://en.wikipedia.org/wiki/B-tree

7.4.1 索引顺序表与分块查找

索引表

key	addr
03	2k
08	1k
17	6k
24	4k
47	5k
51	7k
83	0
95	3k



数据表

职工号	姓名	性别	职务	婚否	
83	林达	女	教师	已婚	…
08	陈洱	男	教师	已婚	…
03	张珊	男	教务员	已婚	…
95	李斯	女	实验员	未婚	…
24	何武	男	教师	已婚	…
47	王璐	男	教师	已婚	…
17	刘淇	男	实验员	未婚	…
51	岳跋	女	教师	未婚	…

7.4.1 索引顺序表与分块查找

- 当数据元素个数很多，*n* 很大时，可采用索引方法来实现存储和查找。
- 若数据不能一次读入内存，那么无论是顺序查找或折半查找, 都需要多次读取外存记录。
- **线性索引 (Linear Index List)**
- 查询时需要从外存中把索引表读入内存，经过查找索引后确定了数据元素的存储地址，再经过 1 次读取元素操作就可以完成查找。只需 2 次读盘即可。

- **采用索引结构可以加速查找速度**

7.4.1 索引顺序表与分块查找

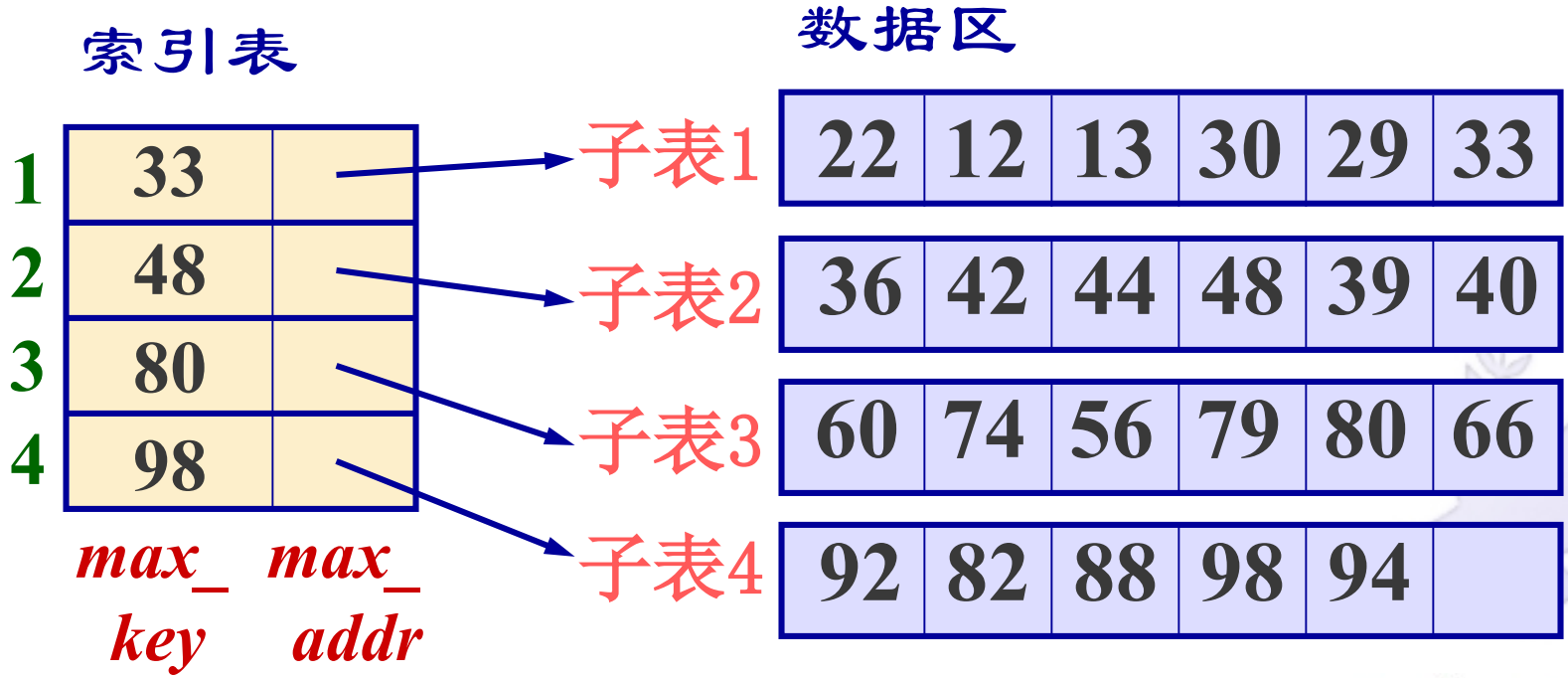
- 索引可分为 2 种：
- 1. 稠密索引：一个索引项对应数据表中一个元素。当元素在外存中按加入顺序存放而不是按关键码值有序存放时必须采用稠密索引，这时的索引结构叫做索引非顺序结构。
- 2. 稀疏索引：当元素在外存中有序存放时，可以把所有 n 个元素分为 b 个子表存放，一个索引项对应数据表中一组元素（子表）。第 i个索引项是第 i 个子表的索引项，$i = 0, 1, \ldots, n-1$。这种索引结构叫做索引顺序结构。

7.4.1 索引顺序表与分块查找

- 对索引顺序结构进行查找时，采用分块查找：
- 1. 先在索引表 *ID* 中查找给定值 *K*, 确定满足

$$ID[i\text{-}1].max\_key < K \le ID[i].max\_key$$

- 2. 再在第*i*个子表中按给定值查找要求的元素.



7.4.1 索引顺序表与分块查找

- 索引顺序查找的查找成功时的平均查找长度

$$ASL_{IndexSeq} = ASL_{Index} + ASL_{SubList}$$

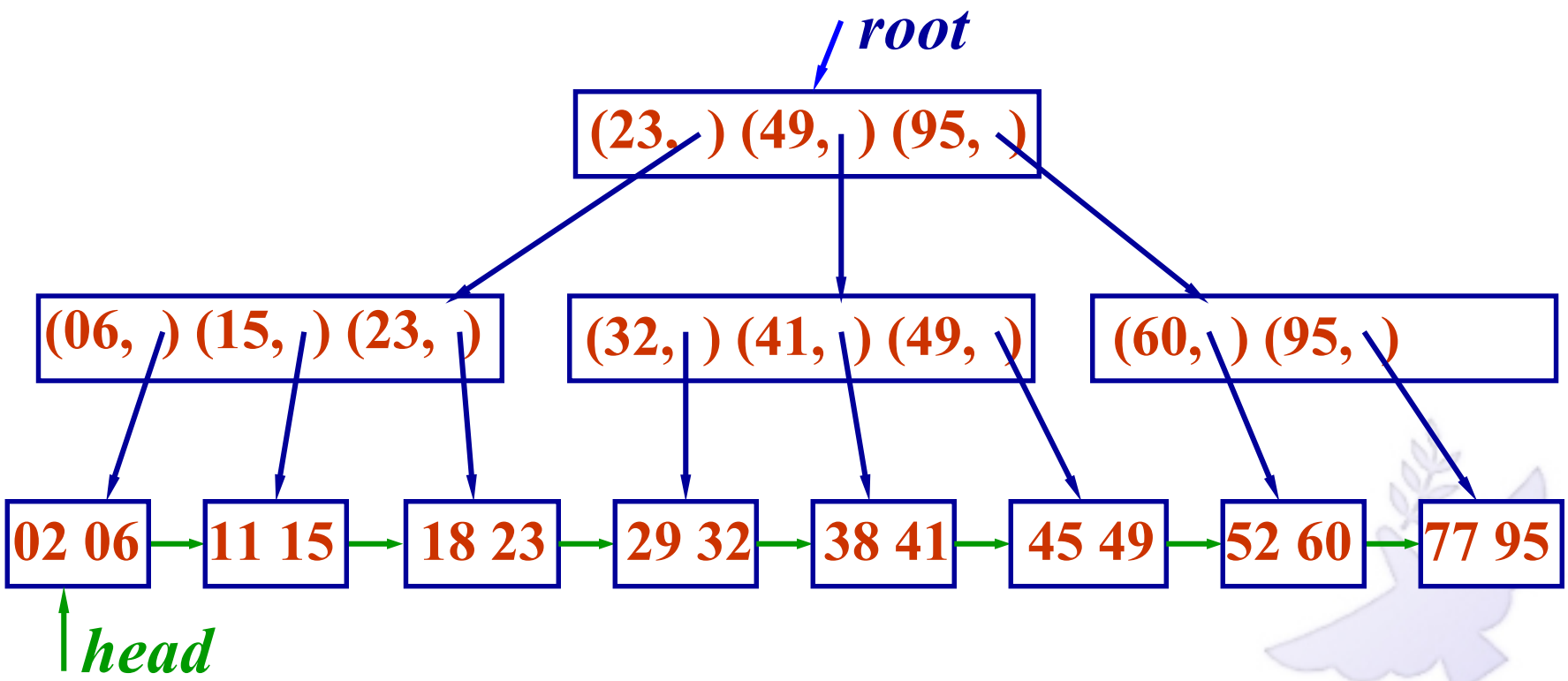
- ASL_{Index} 是在索引表中查找子表位置的平均查找长度.
- $ASL_{SubList}$ 是在子表内查找元素位置的查找成功的平均查找长度。

7.4.2 多级索引与m 叉查找树

- 当数据元素数目特别大，索引表本身很大，在内存中放不下，需要分批多次读取外存才能把索引表查找一遍。
- 此时, 可以建立索引的索引（二级索引）。二级索引中一个索引项对应一个索引块，登记该索引块的最大关键码及该索引块的存储地址。
- 如果二级索引在内存中也放不下，需要分为许多块多次从外存读入。可以建立二级索引的索引（三级索引）。
- 这时，访问外存次数等于读入索引次数再加上 1 次读取元素。

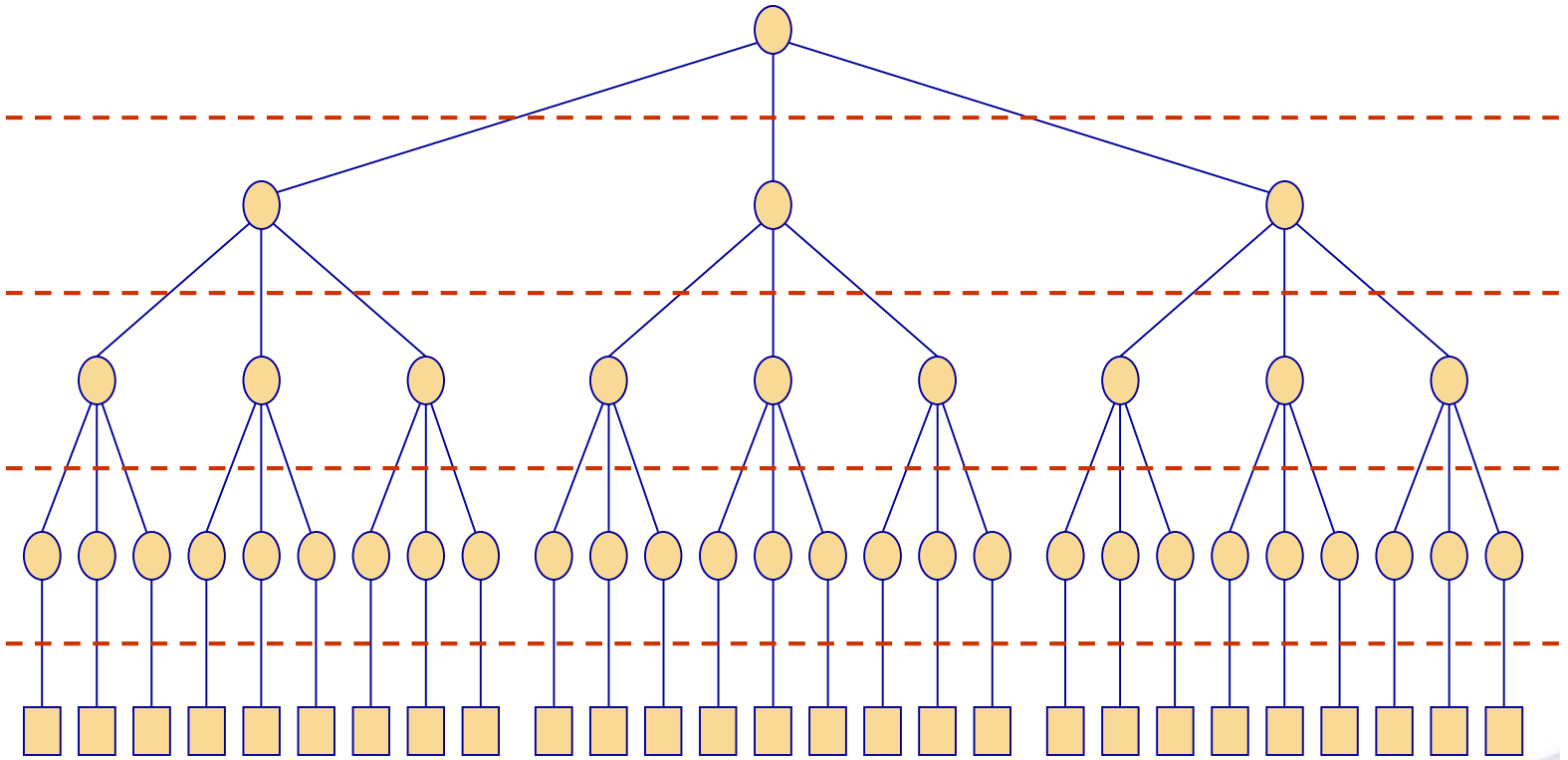
7.4.2 多级索引与m 叉查找树

- 树中每一个分支结点表示一个索引块，每个索引项给出各子树结点的最大关键码和结点地址。



7.4.2 多级索引与m 叉查找树

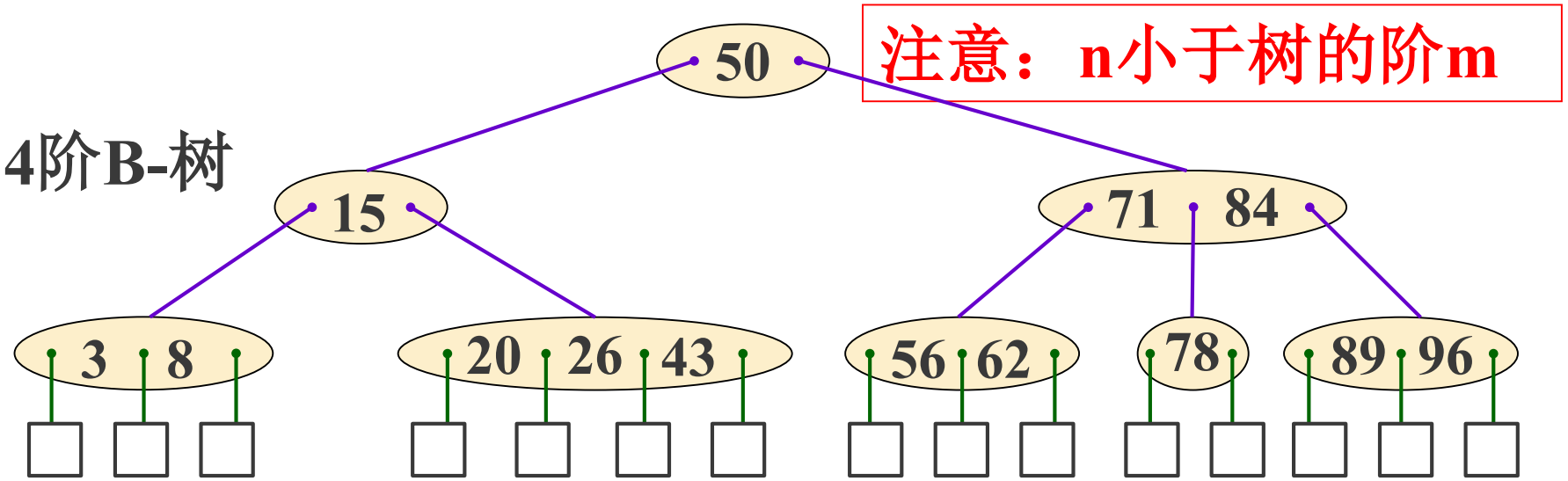
- m 叉查找树：树的叶结点中各索引项给出在数据表中存放的元素的关键码和存放地址



多级索引结构形成 *m* 路查找树

7.4.3 B - 树

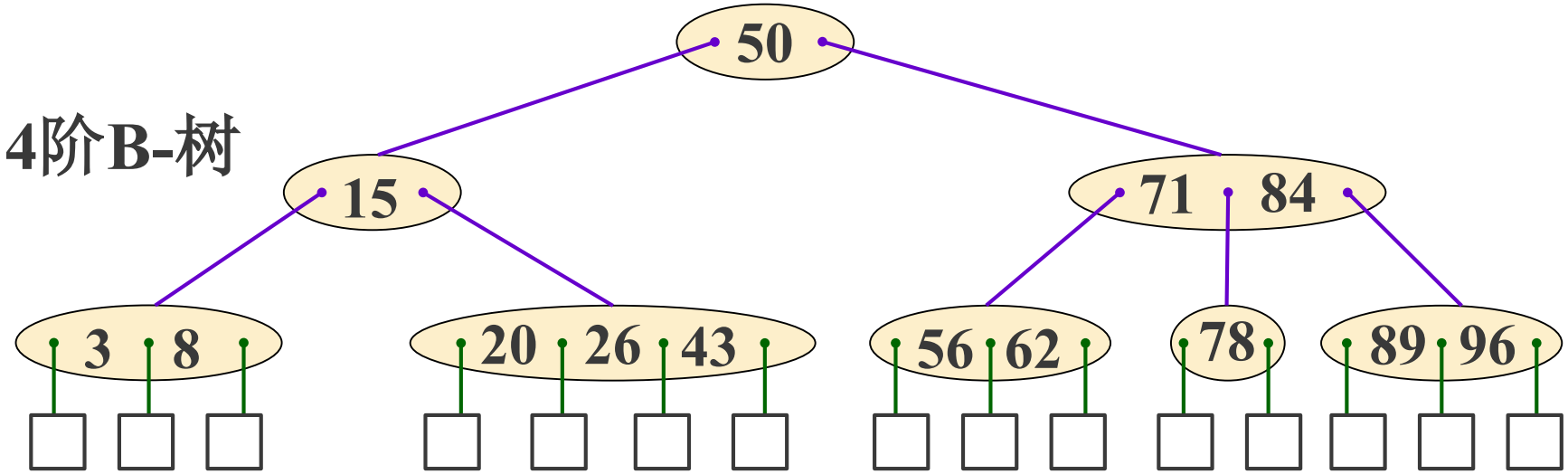
- **B-树的定义：B-树是一种高度平衡的m叉查找树：**



- 结点结构$(n, P_0, K_1(D_1), P_1, K_2(D_2), P_2, \ldots, K_n(D_n), P_n)$
- 性质1：每个结点有 n 个关键字； n 个指向记录的指针 Di；

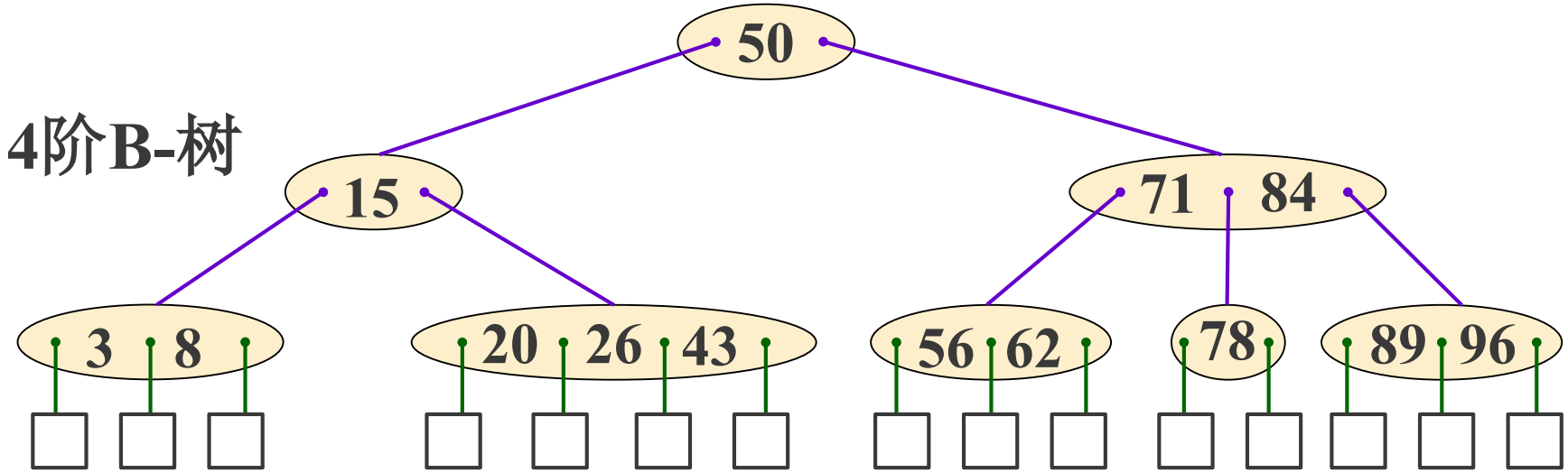
n+1 个指向子树的指针 Pi；

B-树的定义



- 性质2：
  - 每个结点中的多个关键字均自小至大有序排列, 即：$K_1 < K_2 < \ldots < K_n$ ；
  - 指针P_{i-1} 所指子树上所有关键字均小于K_i ；
  - 指针P_i 所指子树上所有关键字均大于K_i ；

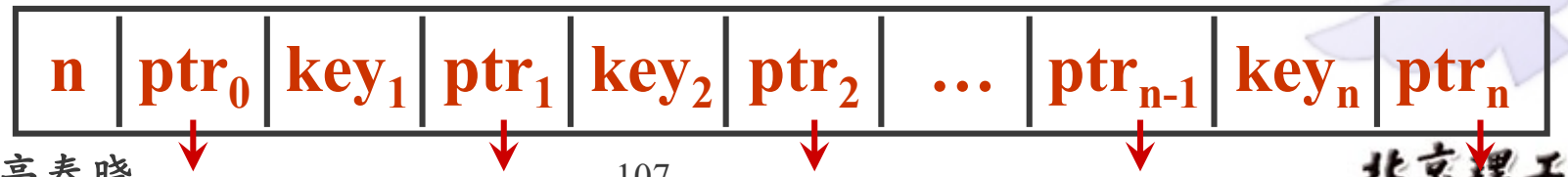
B-树的定义



- 性质3：
  - 树中所有叶子结点（失败结点）均在树中的同一层次上；
  - 根结点或为空, 或至少含有两棵子树；
  - 其余所有结点均至少含有⌈m/2⌉棵子树, 至多含有 m 棵子树；

B 树的定义和 B 树结点的定义

```
#define m 4                          //B树的阶数
typedef int KeyType;                 //关键码类型
typedef struct node {                //B树结点定义
   int n;                            //结点内关键码个数
   struct node *parent;              //双亲指针
   KeyType key[m+1];                 //关键码数组, key[0]不用
   struct node *ptr[m+1];            //子树指针数组
   int *recptr[m+1];                 //指向数据区记录的指针
} BTNode, *BTree;                    //B树的定义
```



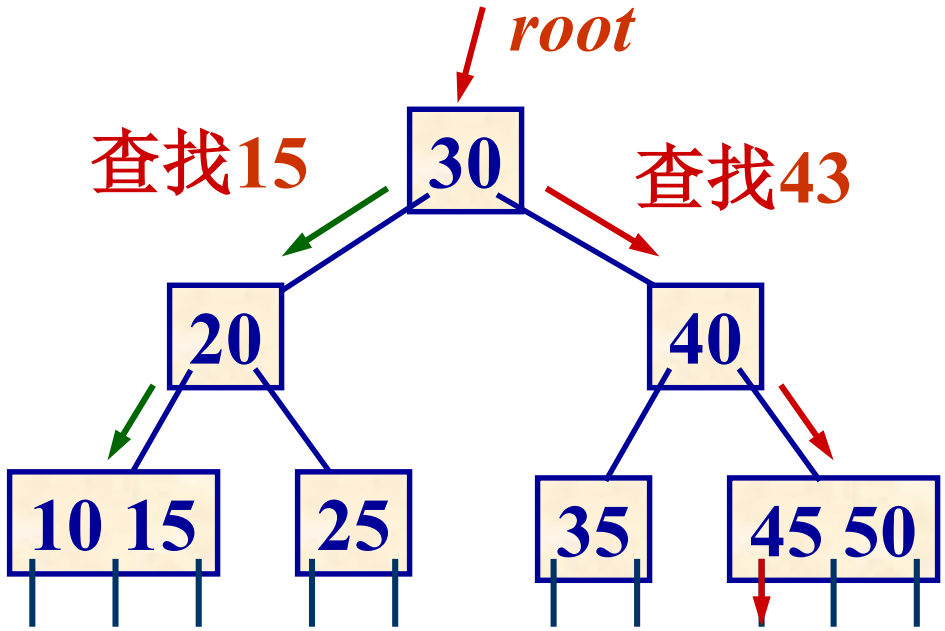
B-树的查找过程

- 基本过程
  - 从根结点出发
  - 沿指针搜索结点和在结点内进行顺序（或折半）查找 两个过程交叉进行。
  - 若查找成功, 则返回指向被查关键字所在结点的指针和关键字在结点中的位置；
  - 若查找不成功, 则可以返回插入位置。

- 在查找不成功之后, 可以进行插入。

B-树的查找过程

- 例：3阶B树：2-3树
- B 树的查找时间与 B 树的阶数 *m* 和B 树的高度 *h* 直接有关。



结点至少⌈m/2⌉棵子树至多有m棵子树

m=3, ⌈m/2⌉-1=1；至少1个关键字, 最多2个关键字。

B-树查找性能分析

- 在B-树中进行查找时, 其查找时间主要花费在搜索结点（访问外存）上, 即主要取决于B-树的深度*h*。
- 考虑最坏情况：含*N*个关键字的 m 阶 B-树可能达到的最大深度 *h* ？
- 分析：
  - 设在 *m* 阶B-树中，失败结点位于第 *h*+1层，即叶子结点。
  - 若树中关键码有 *N* 个, 则失败结点数为 *N*+1。
  - 这是因为失败发生在 $K_i < x < K_{i+1}$, $0 \le i \le N$，设$K_0 = -\infty$，$K_{N+1} = +\infty$。

5）B-树查找性能分析

- 1）最坏情况下m 阶B-树的结点数：
  - 第一层：1；
  - 第二层：2
  - 第三层：$2(\lceil m/2 \rceil)$ ……
  - 第h层：$2(\lceil m/2 \rceil)^{h-2}$
- 2）第h+1层失败结点为N+1个（即第l层的指针数）
  - $N+1 >= 2(\lceil m/2 \rceil)^{h-1}$ → $h <= \log_{\lceil m/2 \rceil}((N+1)/2)+1$
- 所以：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
  - $\log_{\lceil m/2 \rceil}((N+1)/2)+1$

5）B-树查找性能分析

- 所以：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

  ¶ $\log_{\lceil m/2 \rceil}((N+1)/2)+1$

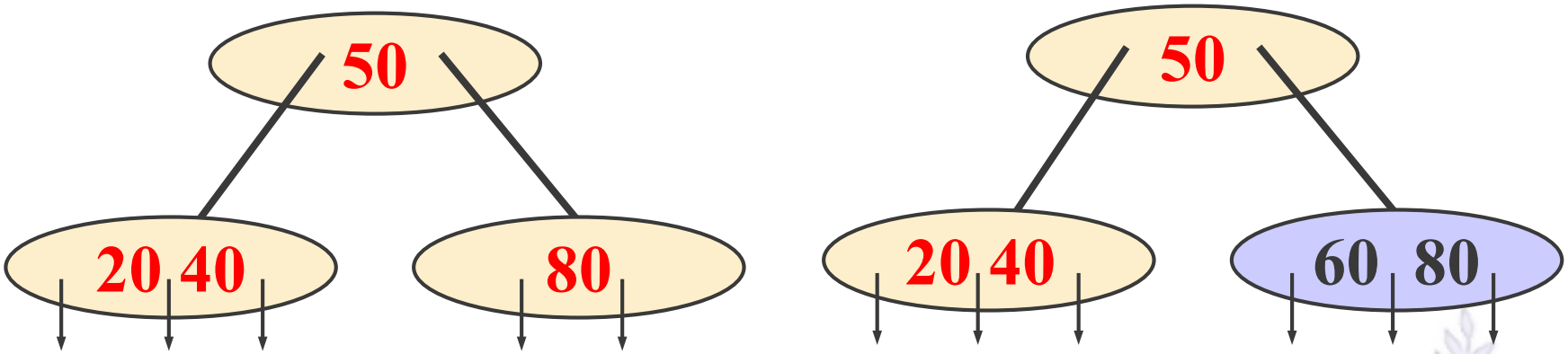
- 示例：若B-树的阶数 $m = 199$，关键码总数 $N = 1999999$，则B-树的高度 h 不超过

  $\log_{100} 1000000 +1= 4$

B-树的插入过程

- 关键字插入的位置必定在叶子结点, 分三种情况
- 情况1：3阶B－树, 插入60

m=3, ⌈*m*/2⌉-1=1；至少1个关键字, 最多2个关键字。



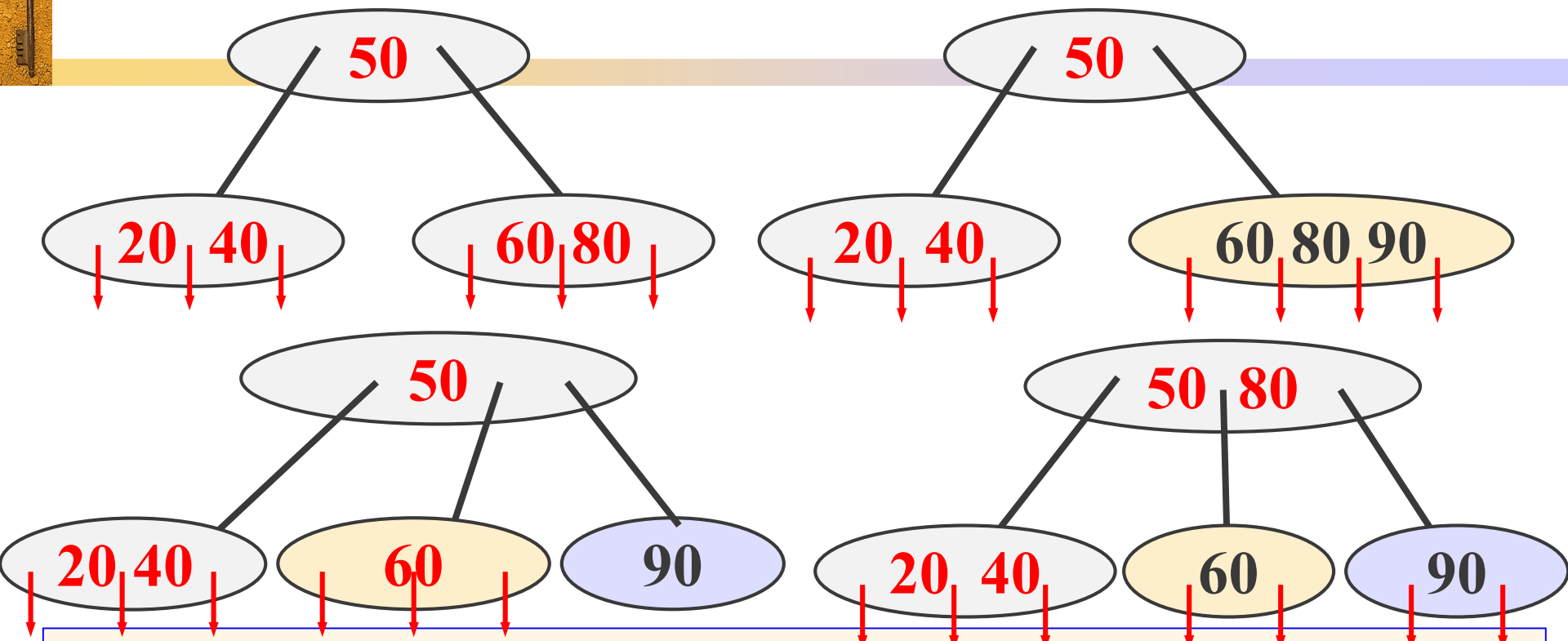
情况1——

特点：插入后, 该结点的关键字个数n<m

方法：不修改指针

◆ 情况2： 3阶B－树, 再插入90

m=3, 1-2个关键字



情况2——特点：插入后结点 n=m

方法：结点分裂, 令 $s = \lceil m/2 \rceil$,

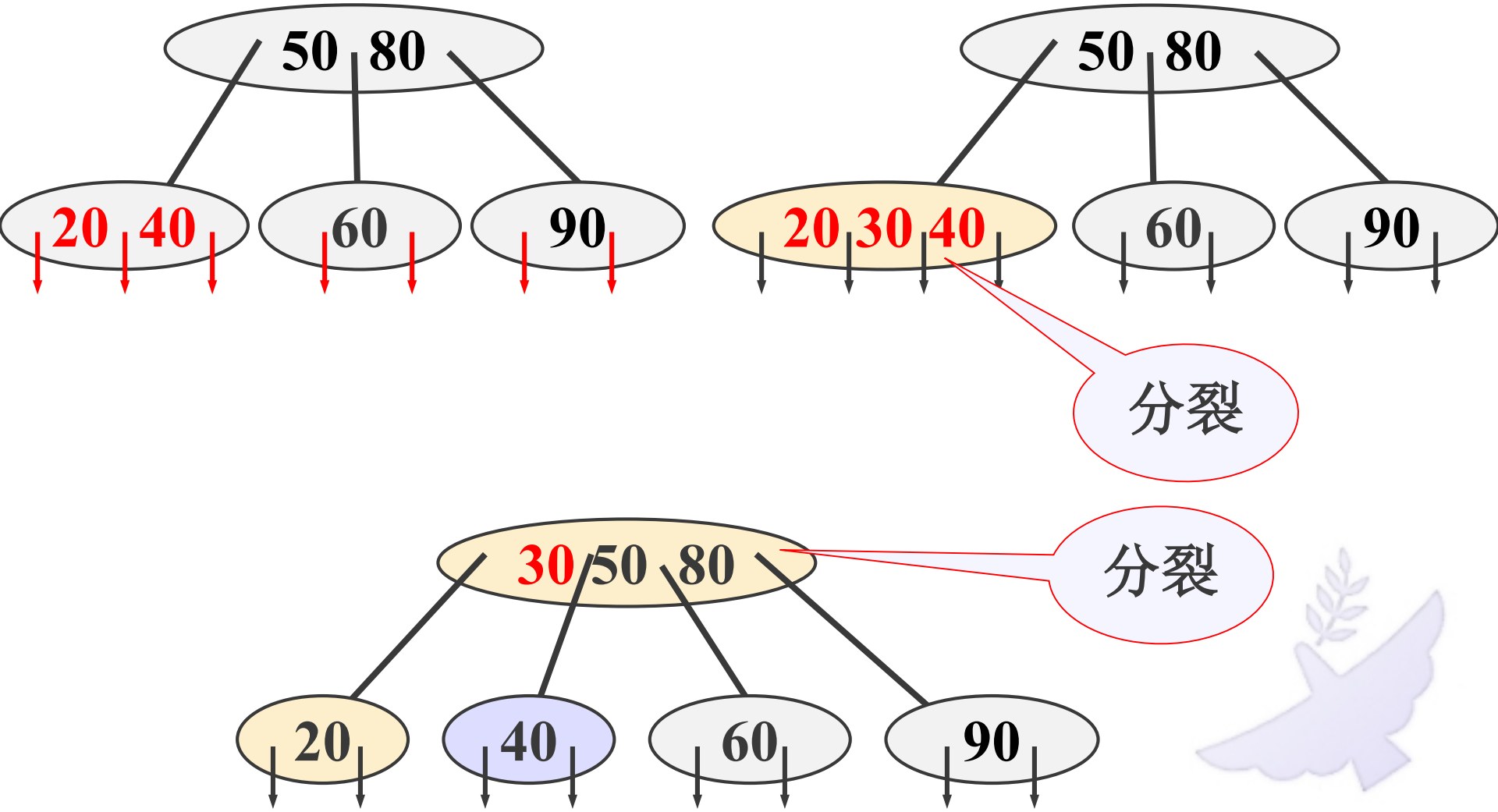
原结点中保留：（$P_0, K_1, \cdots\cdots, K_{s-1}, P_{s-1}$）；

建新结点：（$P_s, K_{s+1}, \cdots\cdots, K_n, P_n$）；

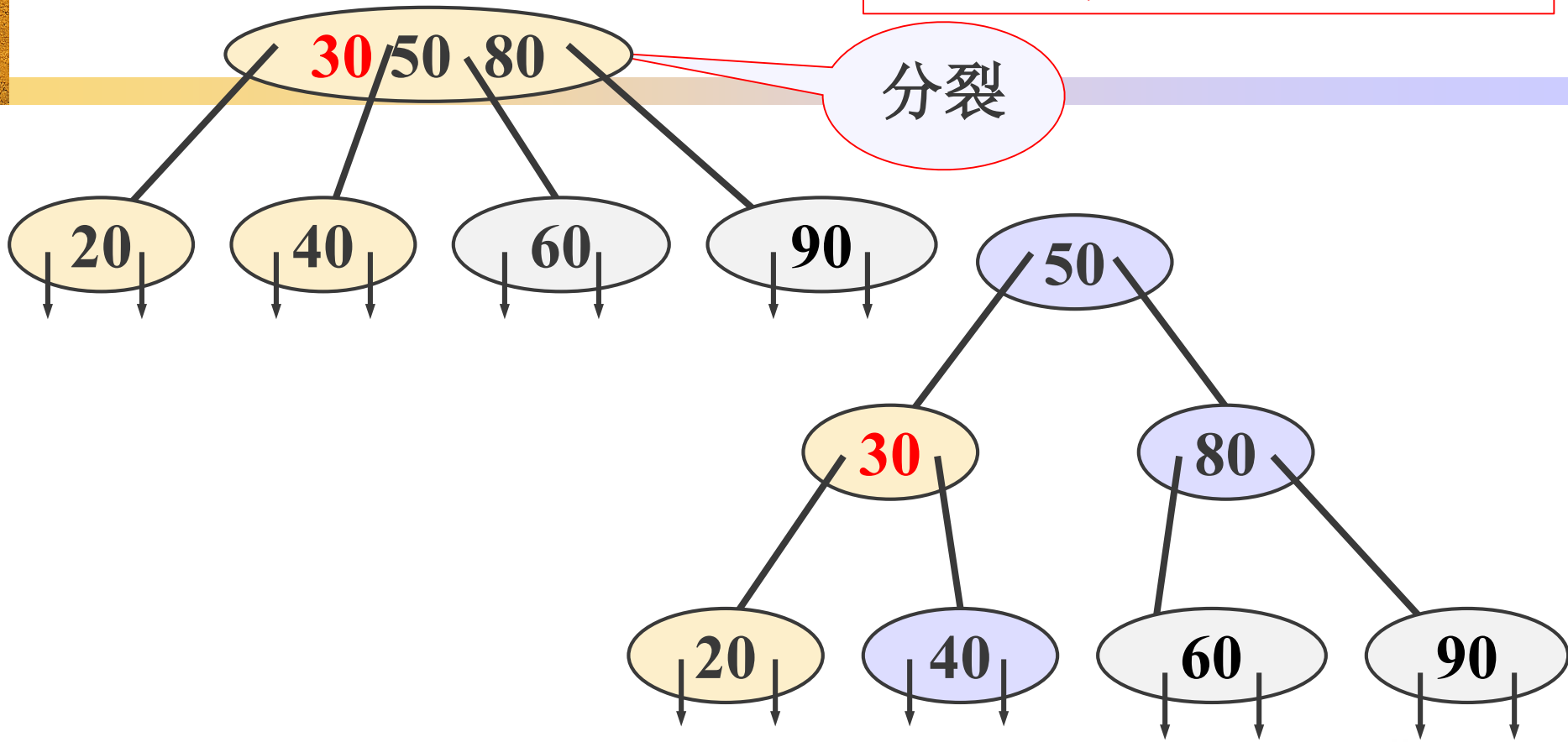
将（Ks, p）插入双亲结点；

◆ 情况3： 3阶B－树, 再插入30

m=3, 1-2个关键字



m=3, 1-2个关键字



- 情况3——特点：结点分裂后，没有双亲
- 方法：新建双亲结点

B-树的插入过程分析

- 在插入新关键码时，首先要查找插入位置，也可能需要自底向上分裂结点
  - 若设B 树的高度为*h*, 那么查找过程中需要进行*h* 次读盘。
  - 最坏情况下从被插关键码所在叶结点到根的路径上的所有结点都要分裂。
    - 分裂非根结点时需要写出2个结点
    - 分裂根结点时需要写出3个结点
- 假设从根到叶结点的路径上所有结点都可以存放在内存中。高度为 h 的B树共需 3h+1 次磁盘访问。
- 平均情况下大约 h+1 次磁盘访问

B-树上的删除过程

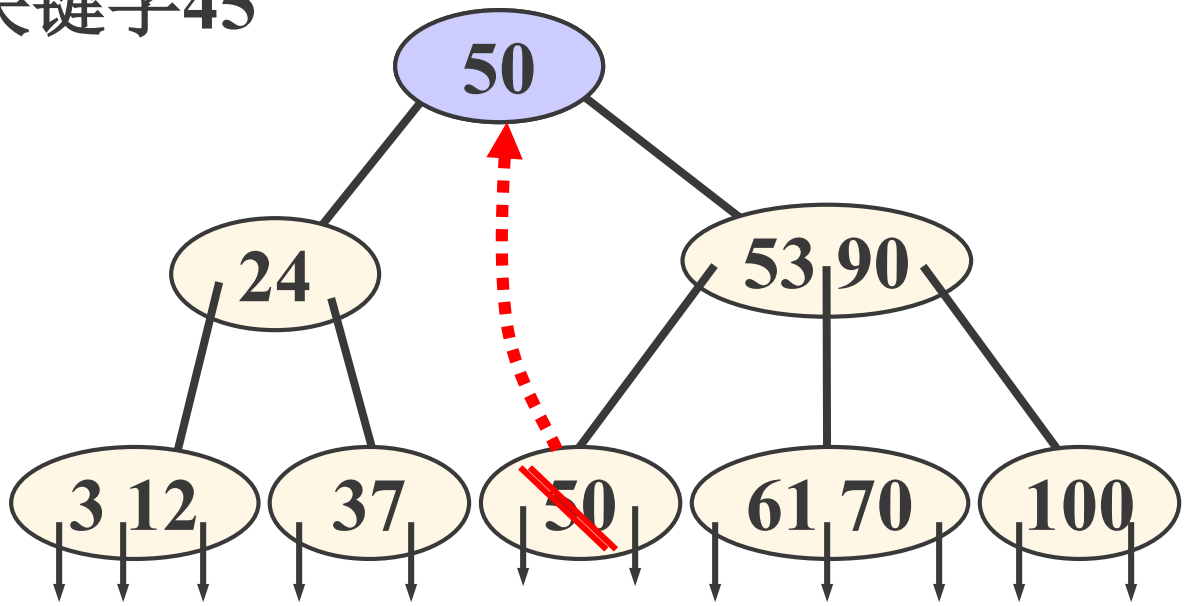
- 首先必须找到待删关键字所在结点；
- 要求删除之后, 结点关键字个数不能小于⌈m/2⌉- 1
- 否则, 要从先其左(或右)兄弟结点“借调”关键字
- 若其左和右兄弟结点均无关键字可借(结点中只有最少量的关键字), 则必须进行结点“合并”。

⌈m/2⌉—— m 棵子树
⌈m/2⌉-1 —— m-1 个关键字

B-树上的删除过程

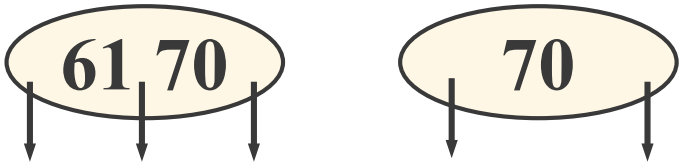
例如：m=3, $\lceil m/2 \rceil - 1 = 1$；至少1个关键字, 2个子结点, 最多2个关键字。

删除关键字45



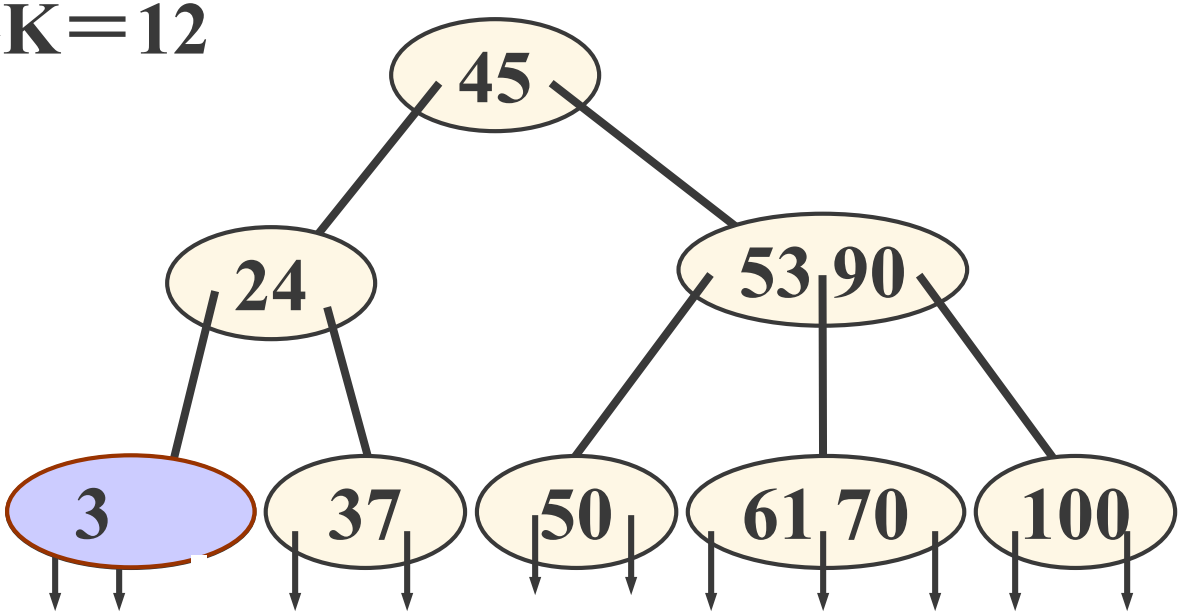
- 特点：非叶子结点中的关键字
- 方法：向前驱/后继借关键字, 然后删除前驱/后继中借走的关键字

- 情况1：
- 特点：被删关键码所在叶结点同时又是根结点且删除前该结点中关键码个数 n ≥ 2，
- 方法：直接删去该关键码并将修改后的结点写回磁盘。



B-树上的删除过程

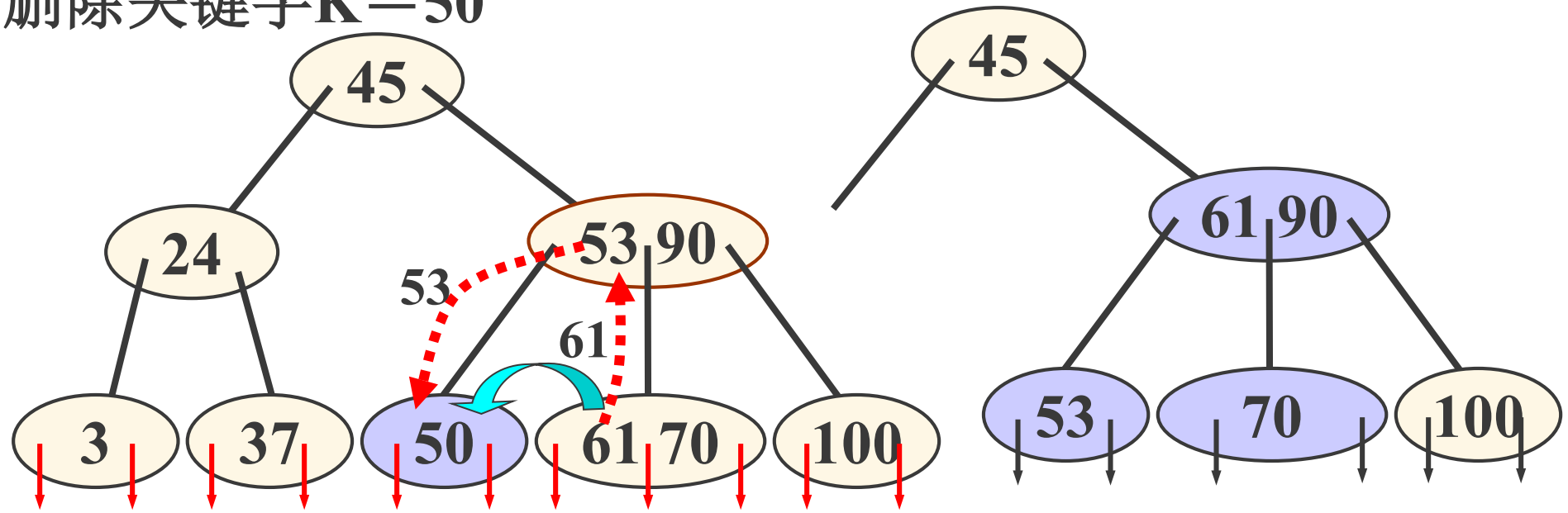
删除关键字K＝12



- 情况2——
- 特点：K所在结点Dk的关键字数目不小于⌈m/2⌉-1
- 方法：从Dk中删除关键字K及其对应指针

B-树上的删除过程

删除关键字K＝50

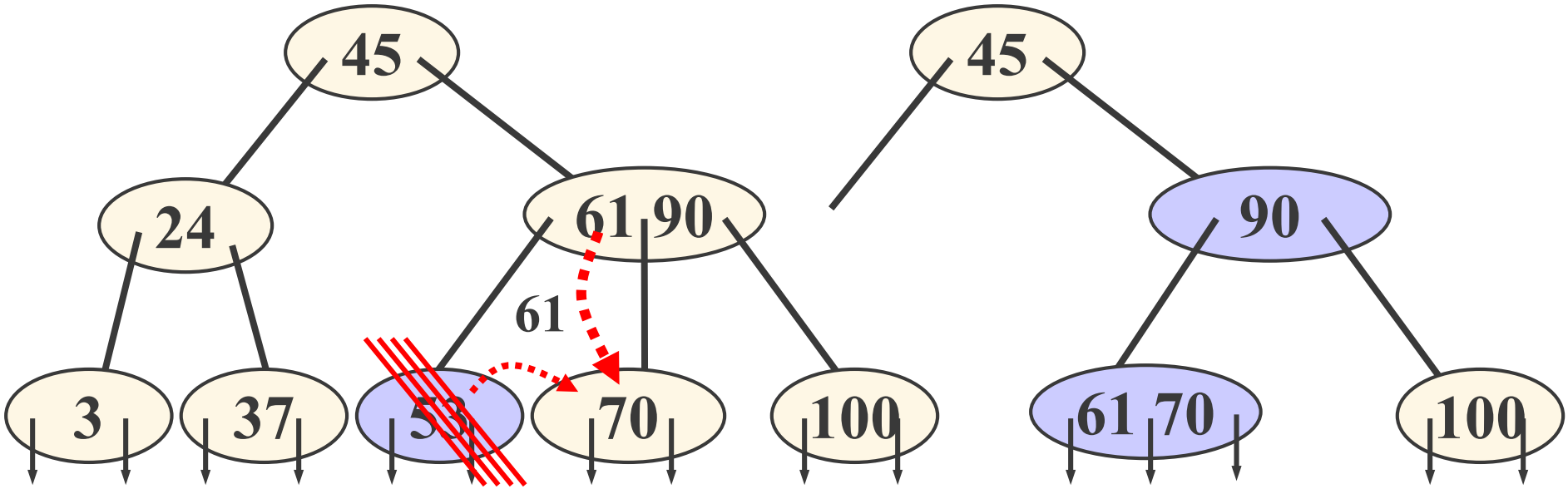


- 情况3——
- 特点：K所在结点Dk的关键字数目等于⌈m/2⌉－1, 而其相邻右兄弟Dkr（左兄弟Dkl）关键字数目大于⌈m/2⌉－1

- 方法：将Dkr(Dkl)中最小（最大）的关键字上移到父结点中, 而将父结点中小于（大于）且紧邻该上移关键字的关键字下移到Dk中。

B-树上的删除过程

删除关键字K=53

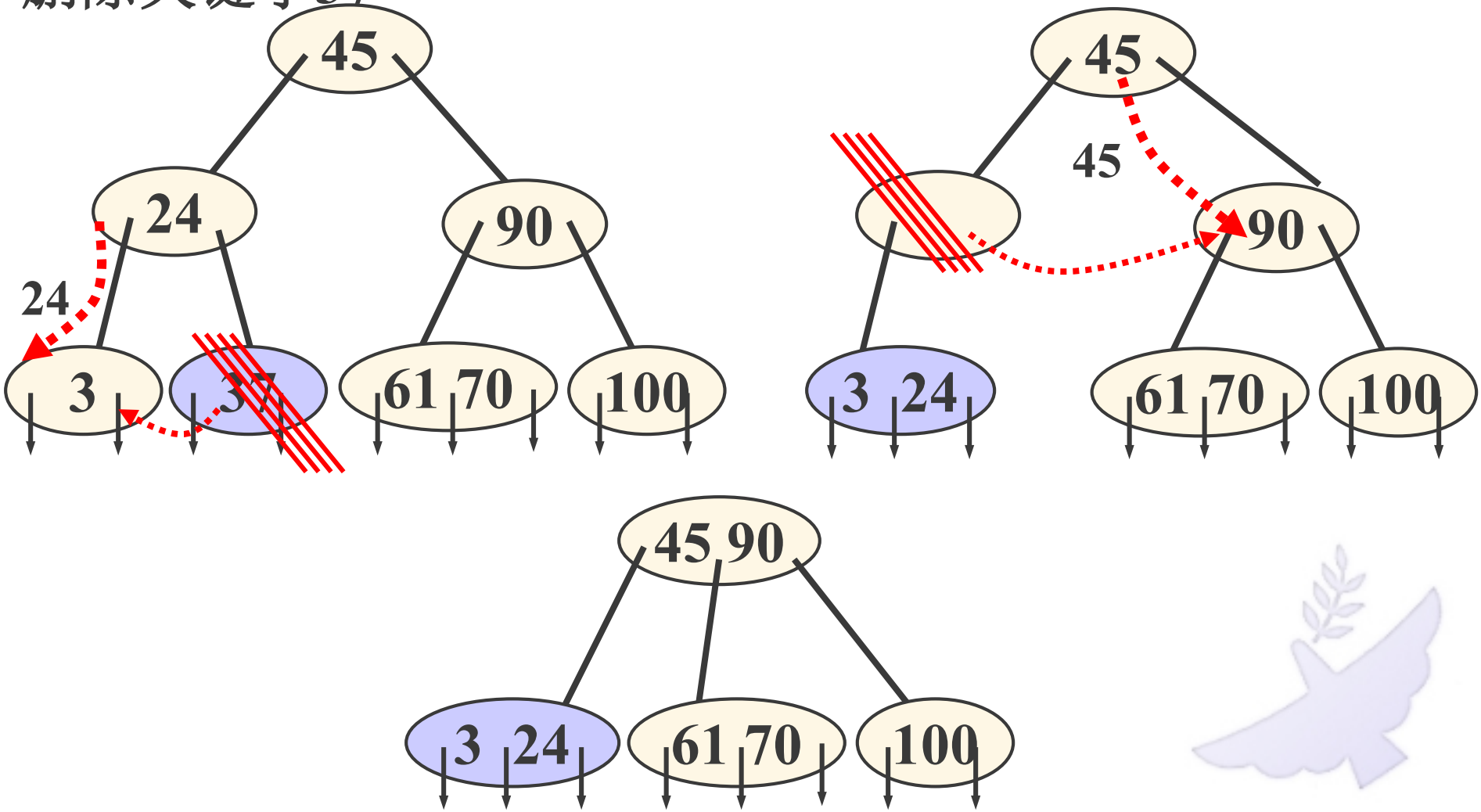


- 情况4——特点： K所在结点Dk和其相邻右兄弟Dkr和左兄弟Dkl的关键字数目都小于⌈m/2⌉−1

- 方法：删除关键字K, DK剩余的关键字及父结点中与之对应的关键字一起加入到Dkr（Dkl）中。
- 最后删除结点DK

B-树上的删除过程

删除关键字37

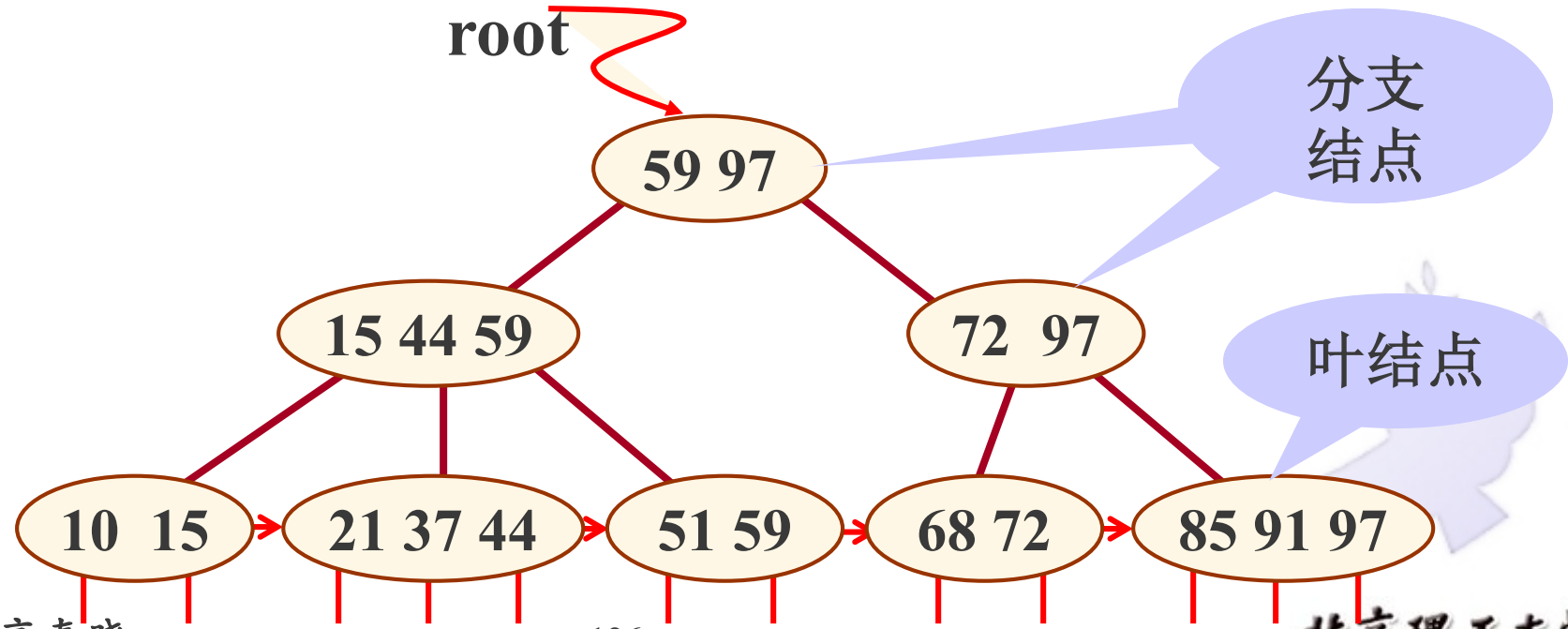


B-树的删除过程分析

- 在删除关键码时，首先要查找插入位置，也可能需要自底向上合并结点
  - ¶ 若设B 树的高度为h, 查找过程中需要进行h 次读盘。
  - ¶ 最坏情况下从被删除关键码所在叶结点到根的路径上的所有结点都要合并。
    - ▸ 合并过程中需要读入h-1个兄弟结点
    - ▸ 然后把合并后的h-1个结点写回磁盘
    - ▸ 释放1个根结点和h-1个兄弟结点
- 假设从根到叶结点的路径上所有结点都可以存放在内存中。高度为 h 的B树共需 3h-2 次磁盘访问；释放h个结点。

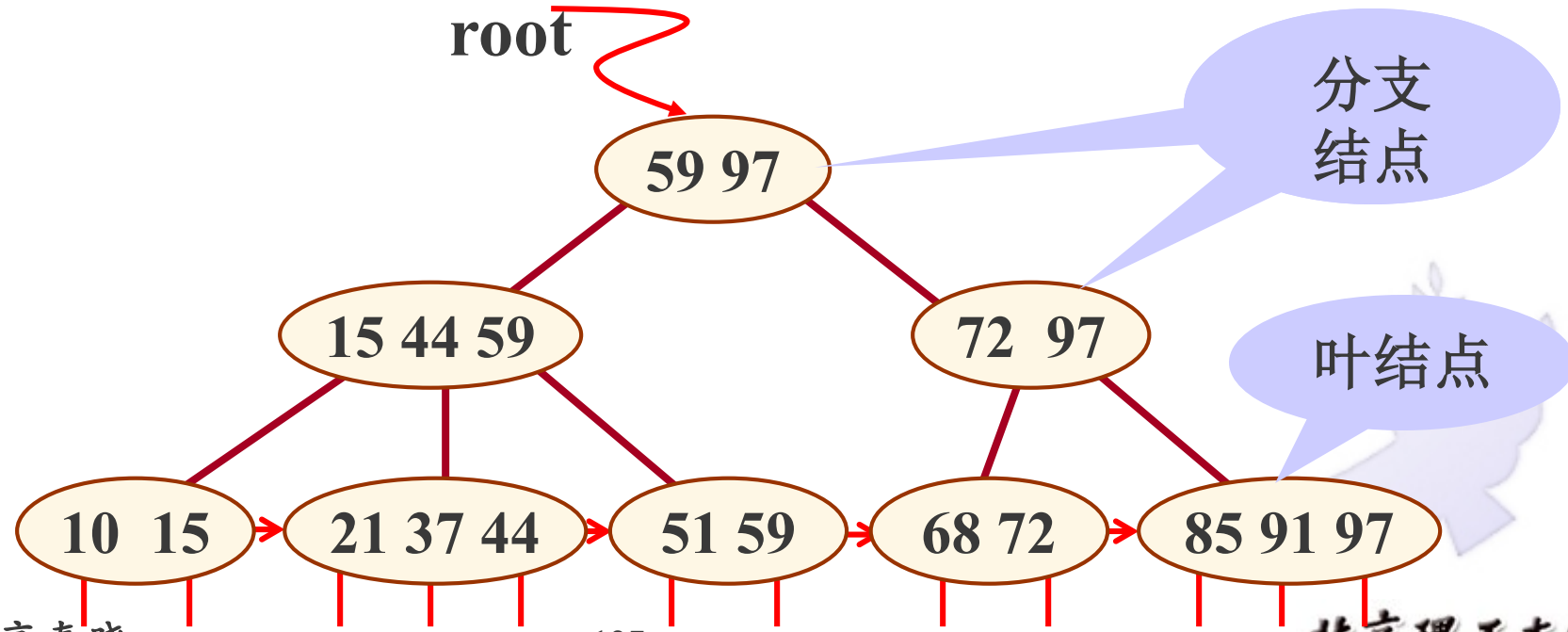
6) B+树

- B+树：B-树的一种变形
- 定义：一棵m阶的B+树是满足下列特征的m叉树：
- (1) 树中每个结点至多有m棵子树；
- (2) 若根结点不是叶子结点，则至少有两棵子树；
- (3) 除根之外的所有非终端结点至少有⌈*m*/2⌉棵子树；



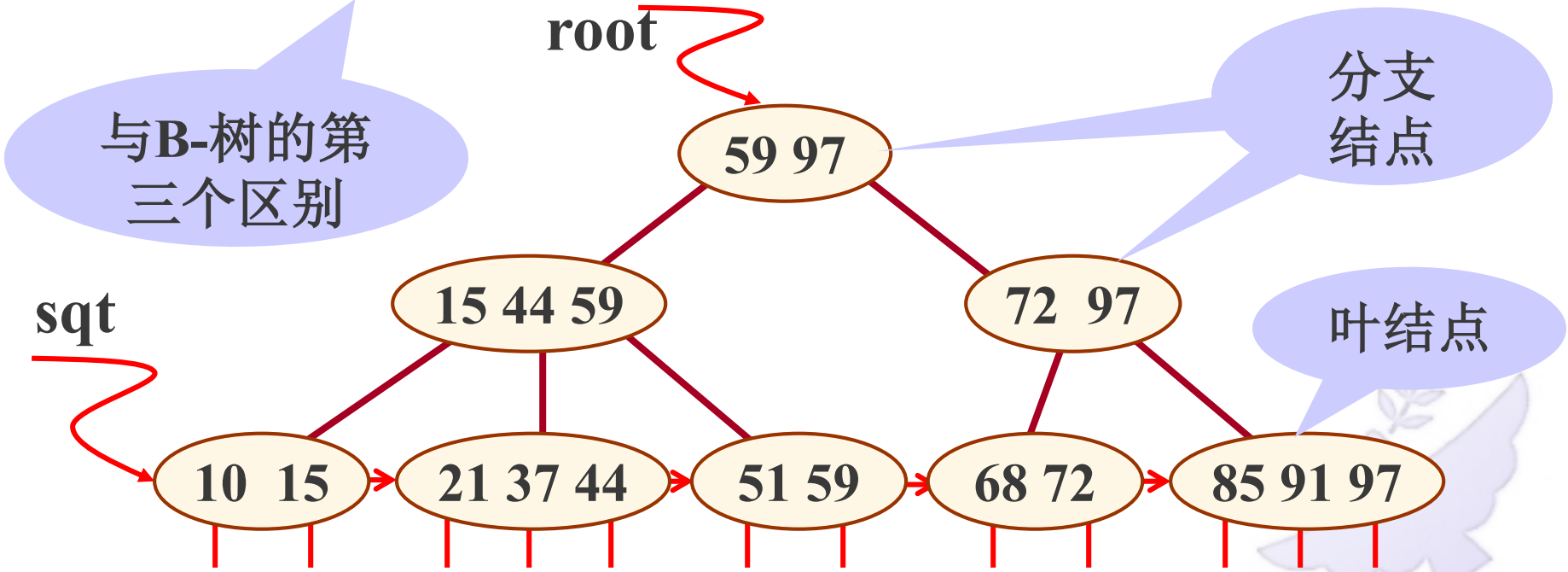
与B-树的第一个区别

- (4) 有 n 棵子树的结点有 n 个关键字, 每个结点包含：(n, k_1, a_1, k_2, a_2, ……, k_n, a_n)，其中,
  - k_i 为关键字, 且 $k_i<k_{i+1}$;
  - a_i 为指向子树根结点的指针, 且指针 a_i 所指子树中所有结点的关键字均小于等于 k_i。



与B-树的第二个区别

- (5) 所有叶结点均在同一层。叶结点按关键字大小顺序链接。可将每个叶结点看成一个基本索引块（直接指向数据文件）。
- (6) 分支结点中仅包含它的各个结点中最大（或最小）关键字的分界值及子结点的指针。所有分支结点可看成是索引的索引。



B+树的查找：两种方式

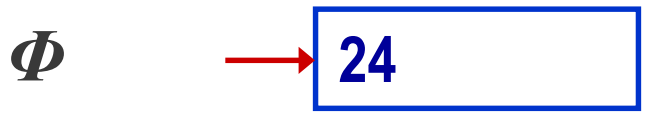
- B+树有两个头指针：
  - 一是指向B+树的根结点；
  - 另一是指向关键字码最小的叶结点，所有叶结点链成线形表，则可以直接从最小关键字开始顺序检索。
- 当从B+树根结点开始随机查找时，检索方法与B-树相似，但若在分支结点中的关键字与检索关键字相等时，检索并不停止，要继续查找到叶结点为止。

与B-树的第四个区别

B+ 树的插入

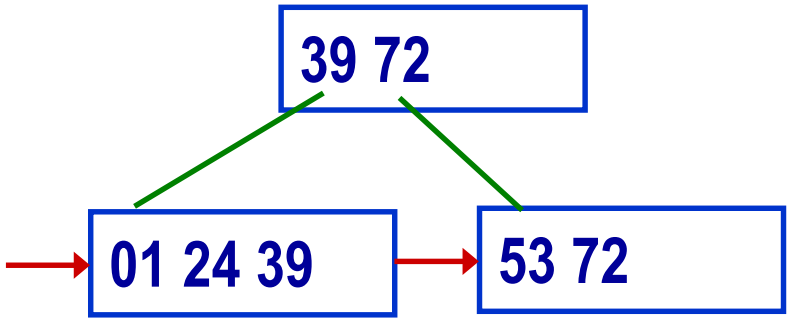
- B+ 树的插入仅在叶结点上进行。每插入一个（关键码–指针）索引项后都要判断结点中的索引项个数是否超出范围 m。
- 当插入后叶结点中的关键码个数 $n > m$ 时，需要将叶结点分裂为两个结点：它们包含的关键码个数分别为 $\lceil (m+1)/2 \rceil$ 和 $\lfloor (m+1)/2 \rfloor$。并且它们的双亲结点中应同时包含这两个结点的最大关键码和结点地址。

- 例如，在空的4阶B+树中的依次插入：
- 24, 72, 01, 39, 53, 63, 90, 15, 88, 10, 44, 68, 74

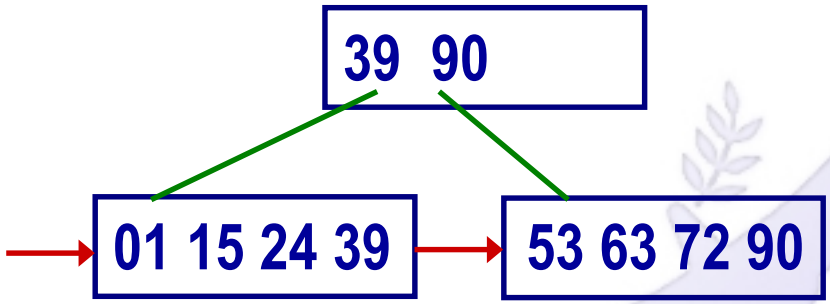


插入24

插入72、01、39

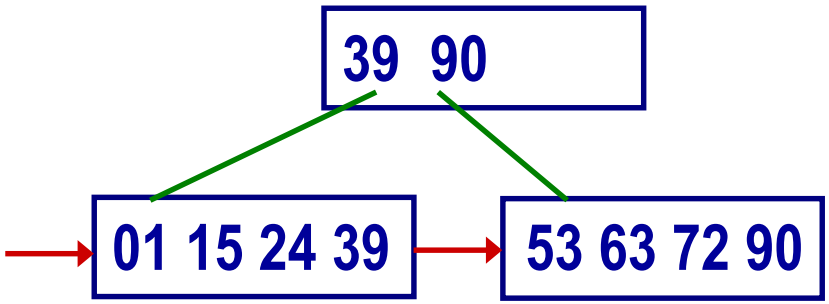


插入53

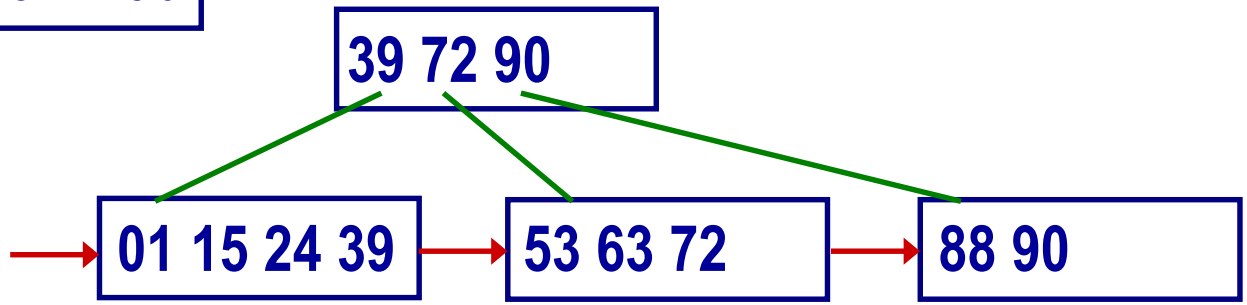


插入63、90、15

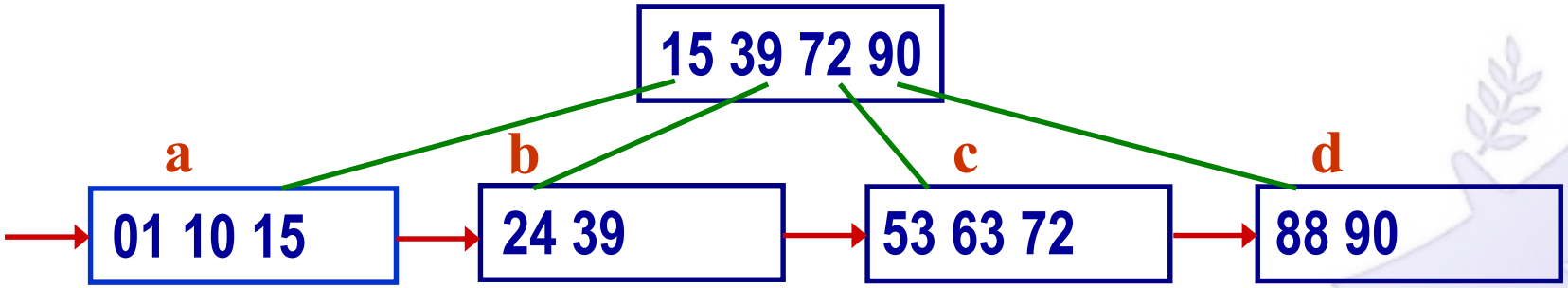
24, 72, 01, 39, 53, 63, 90, 15, 88, 10, 44, 68, 74



插入63、90、15

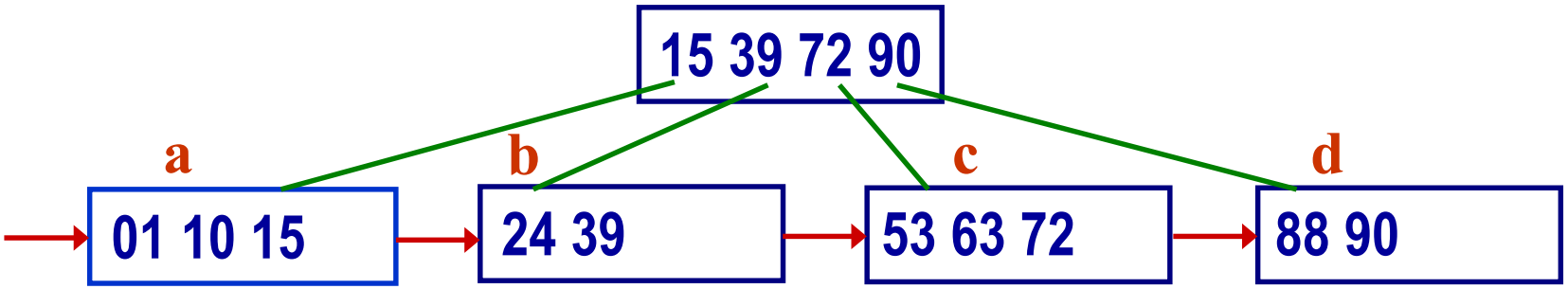


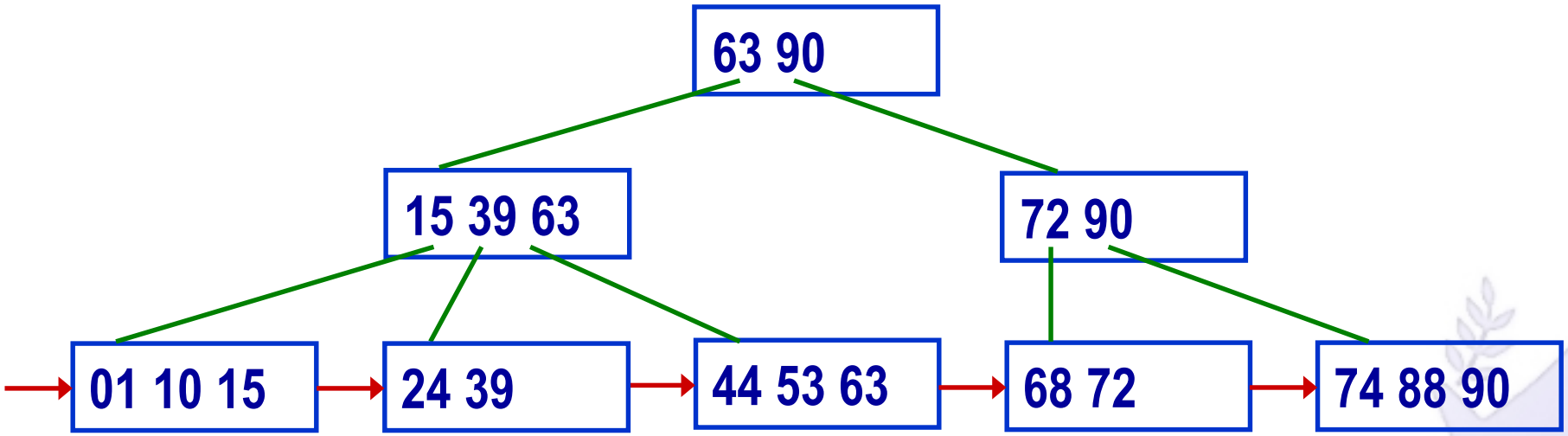
插入88



插入10

24, 72, 01, 39, 53, 63, 90, 15, 88, 10, 44, 68, 74



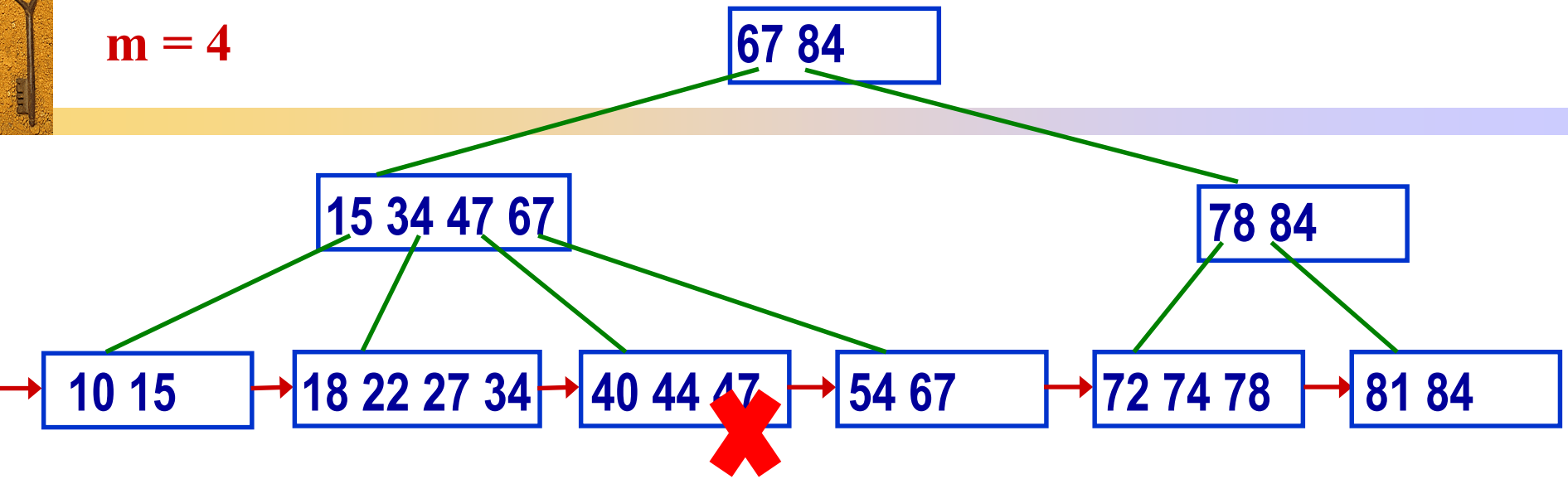


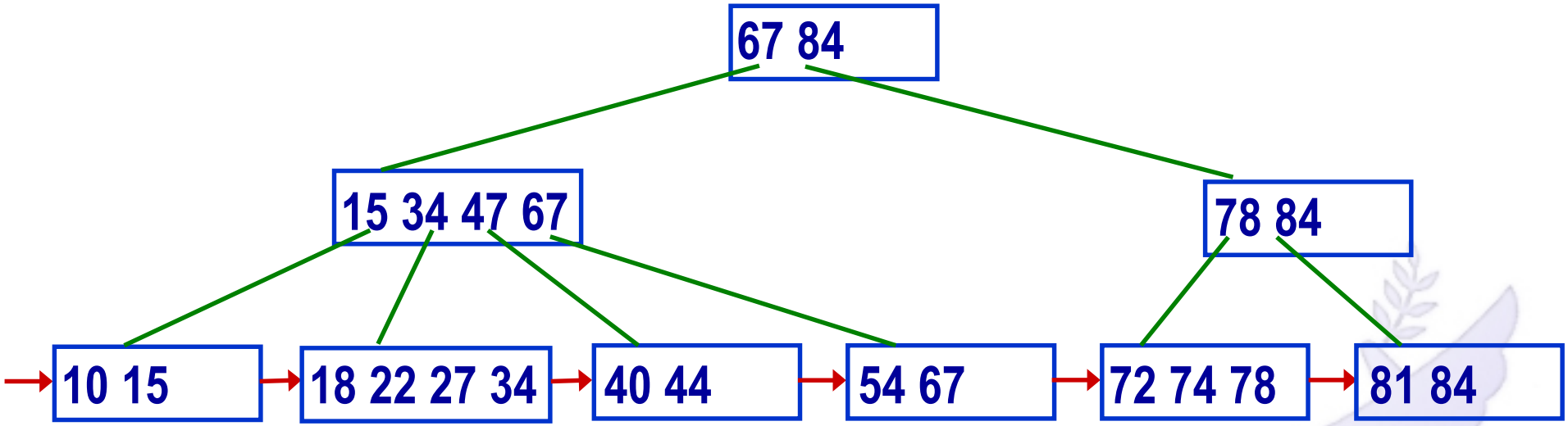
插入44, 68, 74

B+树的删除

- B+ 树的删除仅在叶结点上进行。
- 情况1：当在叶结点上删除一个（关键码-指针）索引项后，结点中的索引项个数仍然不少于⌈m/2⌉，这属于简单删除，其上层索引可以不改变。
- 如果删除结点的最大关键码，但因在其上层的副本只起了一个引导搜索的“分界关键码”的作用，所以即使树中已经删除了关键码，但上层的副本仍然可以保留
- 情况2：如果在叶结点中删除后，结点中索引项个数小于⌈m/2⌉，必须做结点的调整或合并工作。

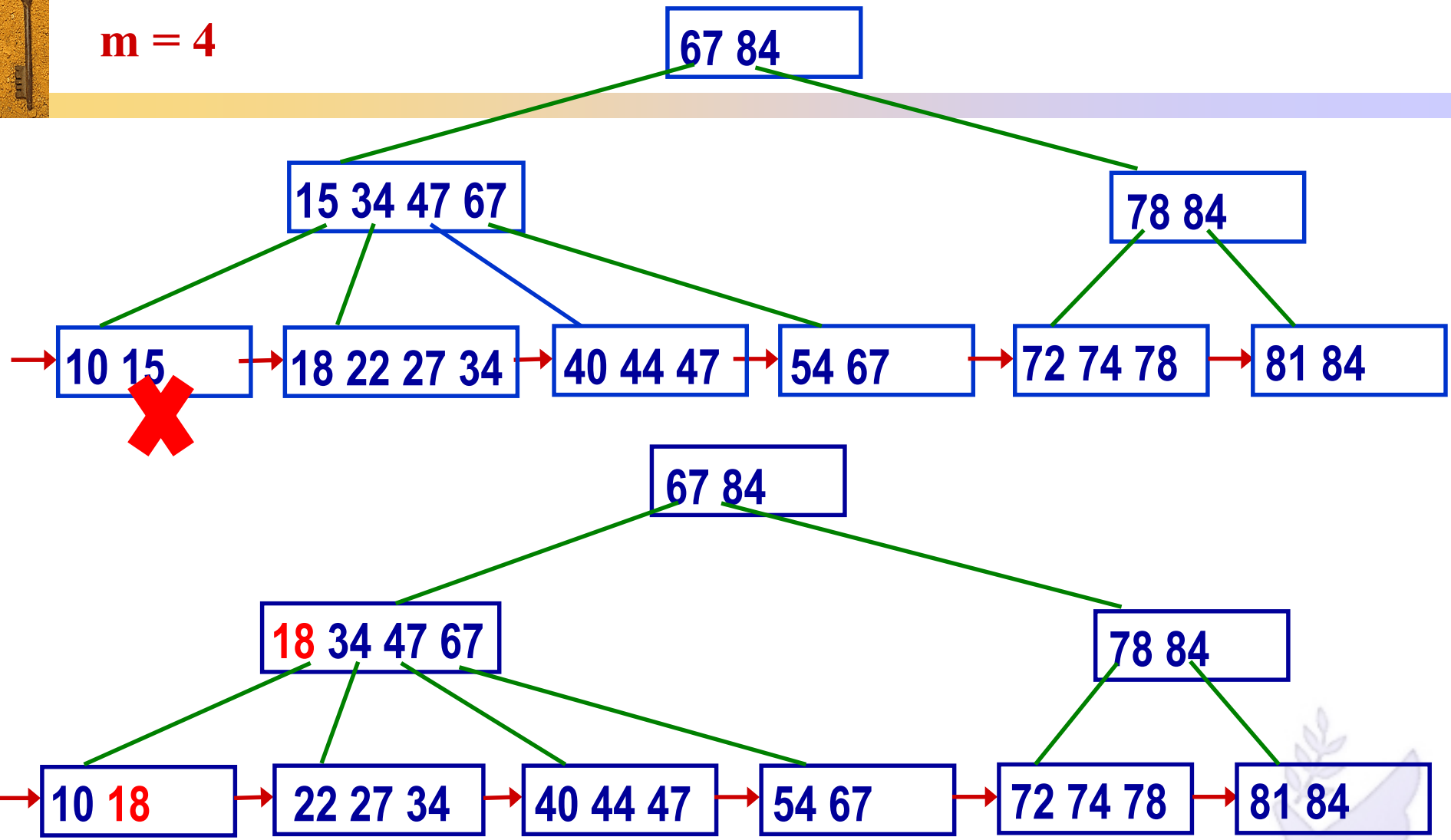
m = 4



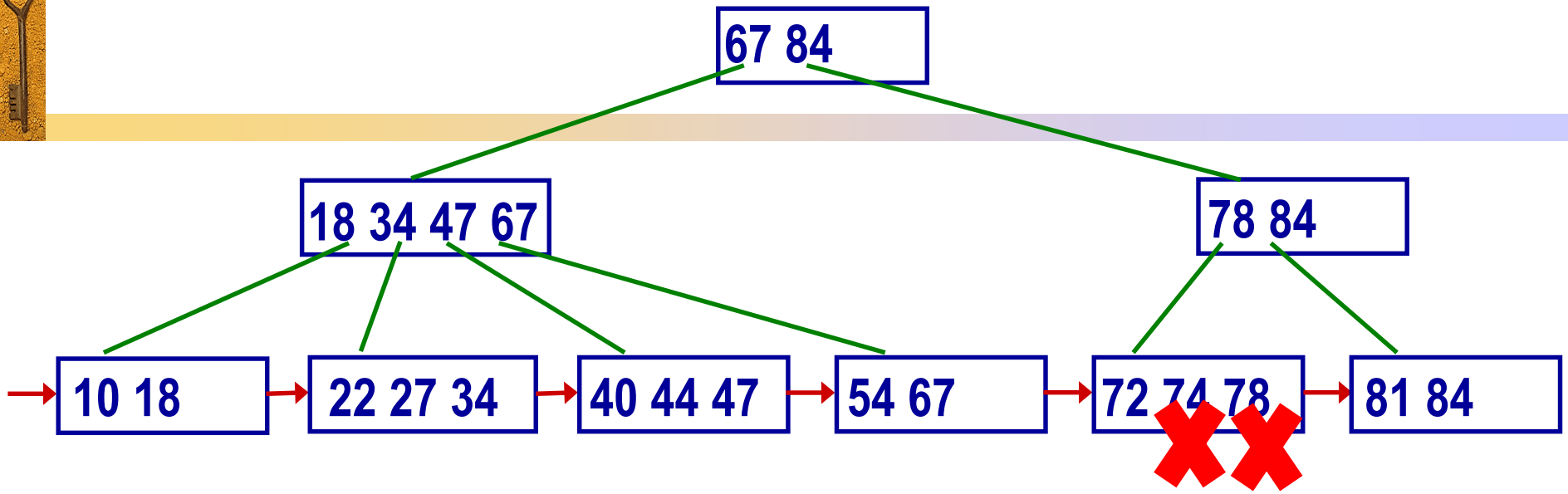


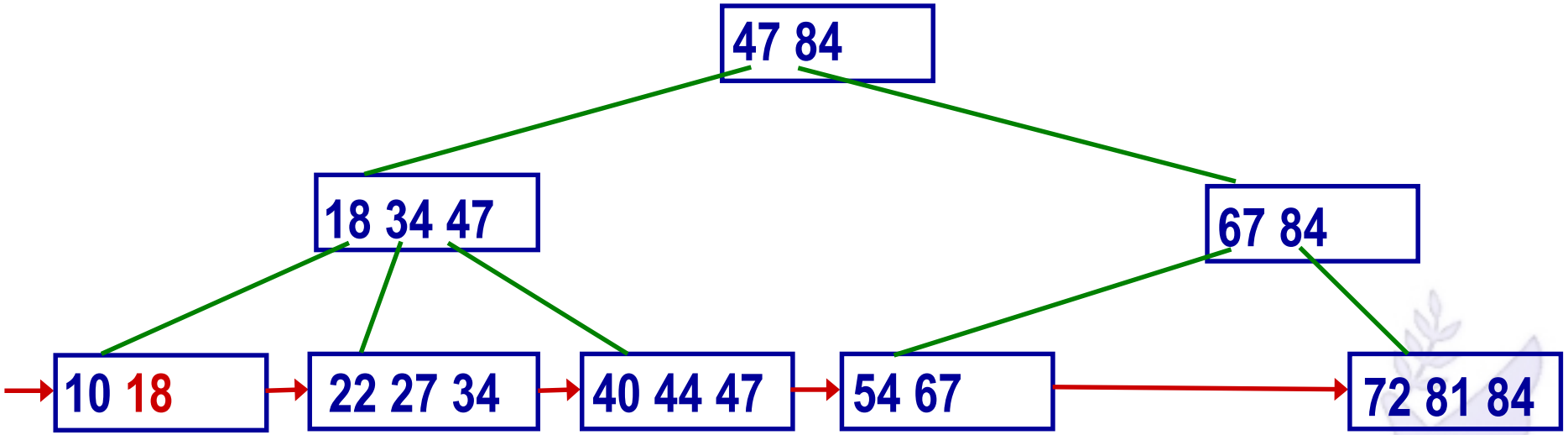
简单删除关键码47, 上层索引可以不改

m = 4



删除关键码15, 调整上层索引改变

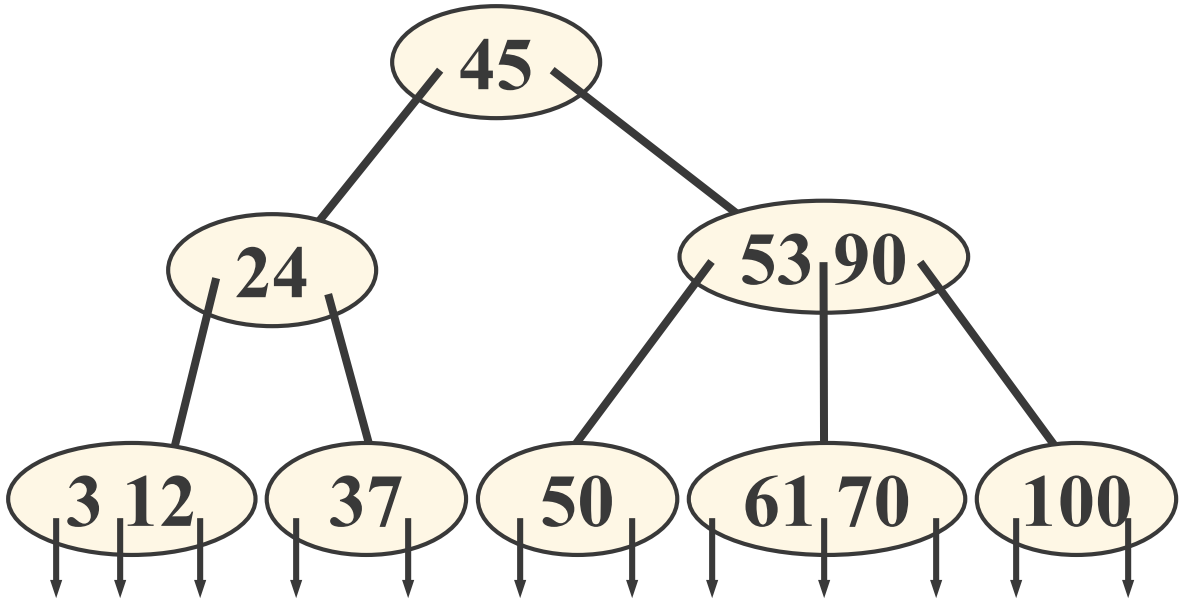




删除关键码74, 78, 结点合并

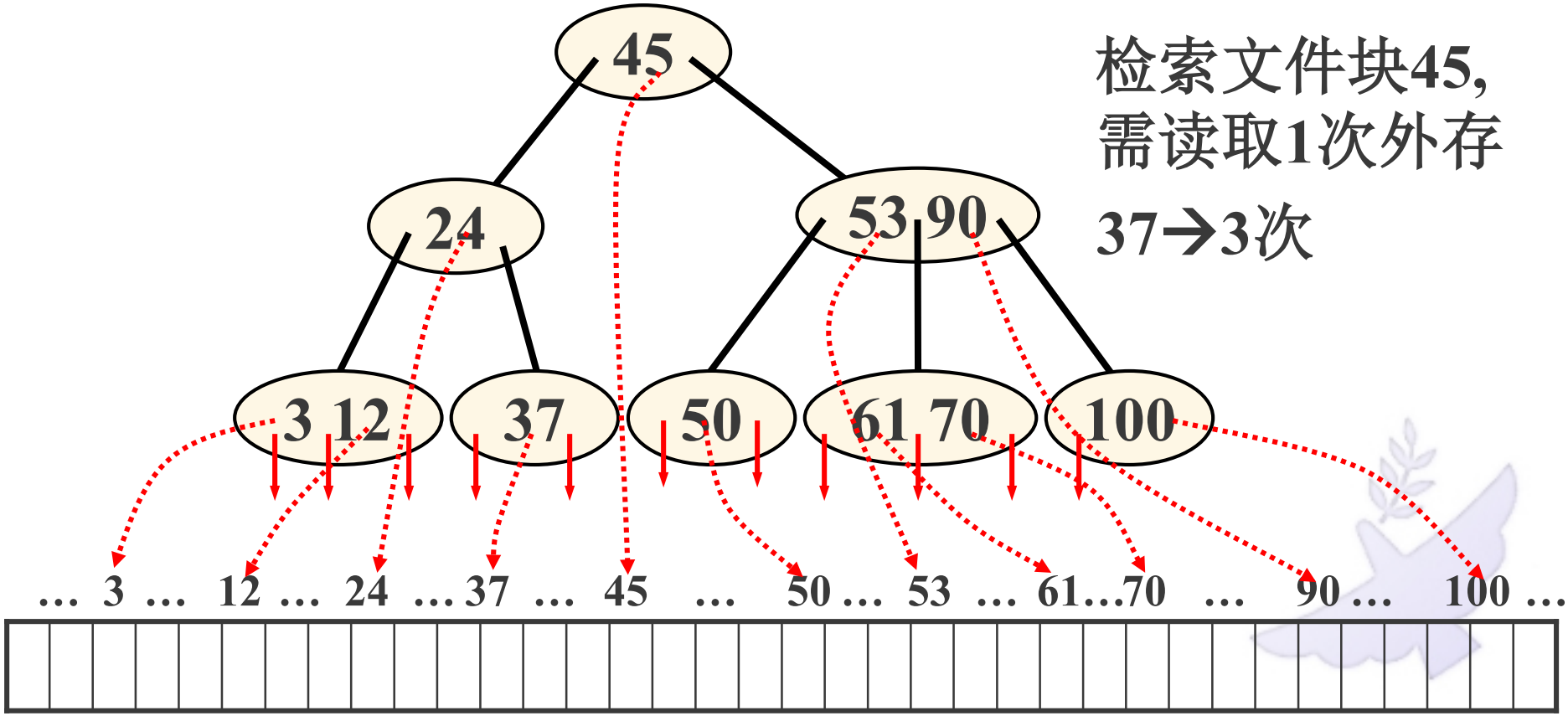
B-树B+树在索引文件中的应用

- B-树在索引文件中的应用
- 1、B-树存储在外存中
- 2、B-树的每个结点存放在外存的一个页块上（因此B树的阶数一般取得较大）。



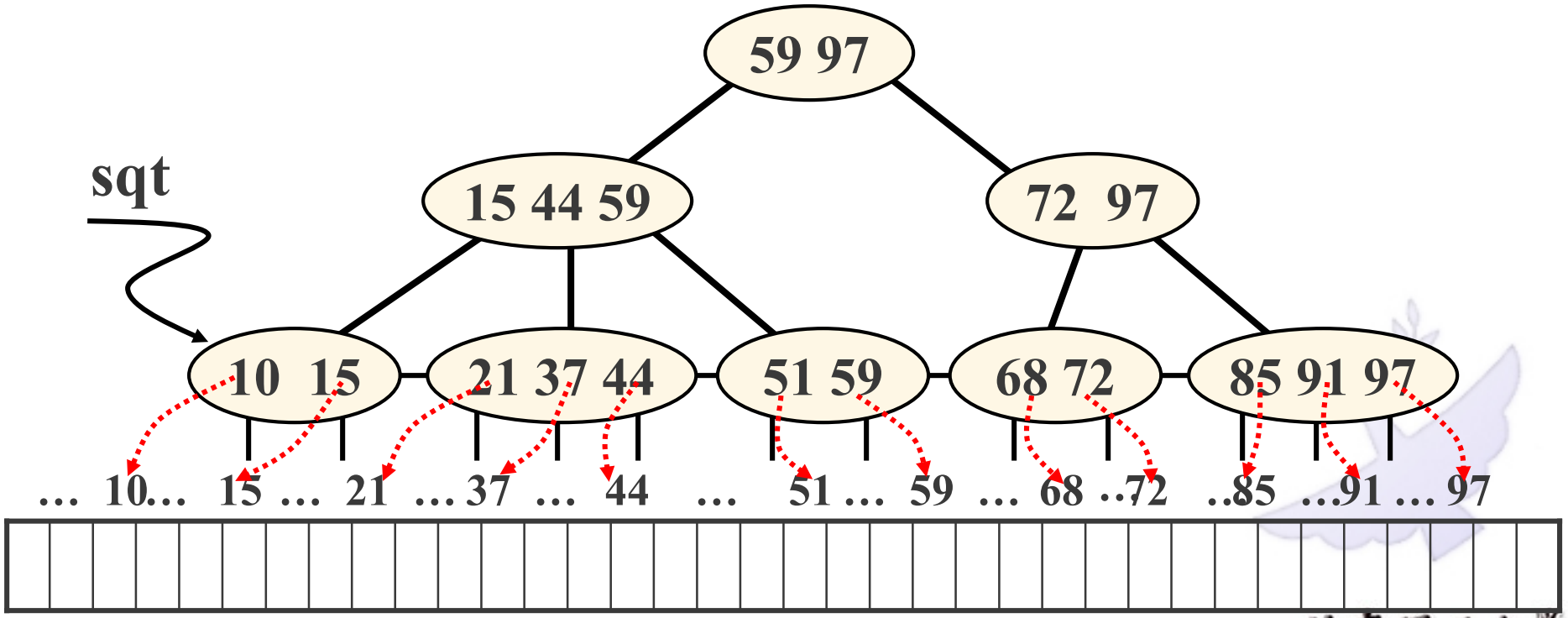
B-树在索引文件中的应用

- 3、B树中任何结点内的一个关键字实际上是一个索引项, 由一个关键字 k 和一个指针 q 组成二元组（k, q）。q是指向主文件页块（或主文件记录）的指针



B+树在索引文件中的应用

- 1、B+树的每个结点存放在外存的一个页块上（因此B+树的阶数一般都比B树大）。
- 2、B+树的树叶层是主文件的稀疏索引，整个B+树构成多级索引。索引项就是B+树中的一个关键字和它对应的指针所构成的二元组。



- B+树比B~树更适合实际应用中操作系统的文件索引
- 1、B+树的磁盘读写代价更低
  - B+树结点小, 一个结点可容纳更多的关键字
  - 因此, 访问外存的次数少；
- 2、B+树的查询效率更加稳定。
  - B+树任何关键字的查找必须走一条从根结点到叶子结点的路；

Oracle 索引

◆索引的原理

- 在TOPIC列上建立索引，Oracle对全表进行一次搜索，将每条记录的TOPIC值按升序排列，然后构建索引条目，即（TOPIC值，ROWID值），存储到索引段中。

TOPIC	ROWID
APPLE	AAAHagAABAAAMZKAAF
BOOK	AAAHagAABAAAMZKAAA
CUP	AAAHagAABAAAMZKAAD
HAT	AAAHagAABAAAMZKAAE
PEN	AAAHagAABAAAMZKAAB
TEE	AAAHagAABAAAMZKAAC
WINE	AAAHagAABAAAMZKAAG

Oracle 索引

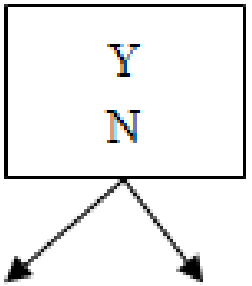
◆索引的类型

- Oracle支持多种类型的索引，可以按列的多少、索引值是否唯一和索引数据的组织形式对索引进行分类
  - ¶ 1. 单列索引和复合索引
  - ¶ 2. B+树索引
  - ¶ 3. 位图索引
  - ¶ 4. 函数索引
- B+树索引是Oracle数据库中最常用的一种索引。
- 当使用CREATE INDEX语句创建索引时，默认创建的索引就是B+树索引。

位图索引

- 针对基数很小的列建立位图索引
- 基数：指某个列可能拥有的不重复值的个数
- 性别、婚姻状况、政治面貌等只具有几个固定值的字段

ID	TOPIC	ISLOOK	ROWID
T0001	BOOK	Y	AAAHagAABAAAMZKAAA
T0203	PEN	Y	AAAHagAABAAAMZKAAB
T1437	TEE	N	
T1682	CUP	Y	
T2735	HAT	N	
T3412	APPLE	N	
T4724	WINE	Y	



ISLOOK 列可能的取值

Y	N	ROWID
		AAAHagAABAAAMZKAAA
		AAAHagAABAAAMZKAAB
		AAAHagAABAAAMZKAAC
		AAAHagAABAAAMZKAAD
		AAAHagAABAAAMZKAAE
		AAAHagAABAAAMZKAAF
		AAAHagAABAAAMZKAAG

ISLOOK 列上创建位图索引的结构

函数索引

- 当需要经常访问一些函数或表达式时，可以将其存储在索引中。
- 函数索引既可以使用B树索引，也可以使用位图索引，可以根据函数或表达式的结果的基数大小来进行选择，当函数或表达式的结果不确定时采用B树索引，当函数或表达式的结果是固定的几个值时采用位图索引。

ID	TOPIC	ISLOOK	ROWID
T0001	Book	Y	AAAHagAABAAAMZKAAA
T0203	Pen	Y	AAAHagAABAAAMZKAAB
T1437	Tee	N	AAAHagAABAAAMZKAAC
T1682	Cup	Y	AAAHagAABAAAMZKAAD
T2735	Hat	N	AAAHagAABAAAMZKAAE
T3412	Apple	N	AAAHagAABAAAMZKAAF
T4724	Wine	Y	AAAHagAABAAAMZKAAG

- SELECT * FROM SALES WHERE TOPIC='TEE';
- 将没有结果。现在忽略大小写，将代码修改如下：
- SELECT * FROM SALES WHERE UPPER(TOPIC)='TEE';
- 这样可以查到相应的结果。
- 由于不是直接查询TOPIC列，所以，即使在TOPIC列上创建了索引也无法使用。
- 使用函数索引，创建函数索引的代码如下：
- CREATE INDEX funidx_upper_topic ON SALES(UPPER(TOPIC));

◆管理索引的原则

◆1．小表不需要建立索引。

◆2．对于大表而言，如果经常查询的记录数目少于表中总记录数目的15%时，可以创建索引。这个比例并不绝对，它与全表扫描速度成反比。

◆3．对于大部分列值不重复的列可建立索引。

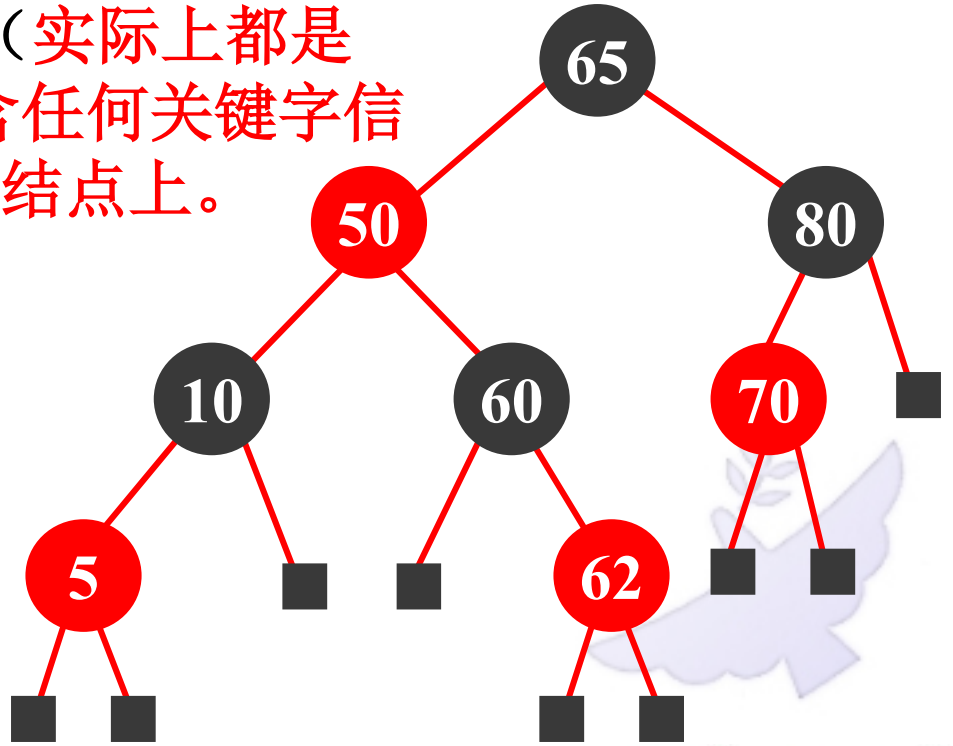
◆4．对于基数大的列，适合建立B树索引，而对于基数小的列适合建立位图索引。

◆5．对于列中有许多空值，但经常查询所有的非空值记录的列，应该建立索引。

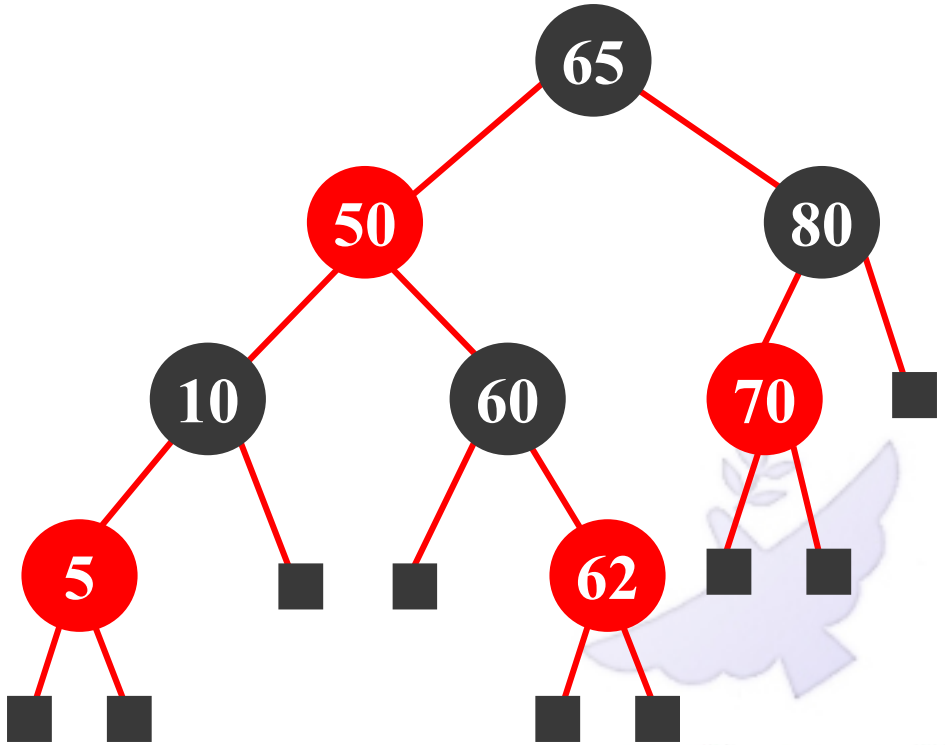
- 6．LONG和LONG RAW列不能创建索引。
- 7．经常进行连接查询的列上应该创建索引。
- 8．在使用CREATE INDEX语句创建查询时，将最常查询的列放在其他列前面。
- 9．维护索引需要开销，特别时对表进行插入和删除操作时，因此要限制表中索引的数量。对于主要用于读的表，则索引多就有好处，但是，一个表如果经常被更改，则索引应少点。
- 10．在表中插入数据后创建索引。如果在装载数据之前创建了索引，那么当插入每行时，Oracle都必须更改每个索引。

7.5 其它搜索树

- 1）红黑树(red-black tree) 是一棵满足下述性质的二叉查找树：
- 1. 每个结点要么是红色, 要么是黑色。
- 2. 根结点是黑色的。
- 3. 所有叶子结点都是黑色的（实际上都是Null指针）。叶子结点不包含任何关键字信息, 所有查询关键字都在非终结点上。



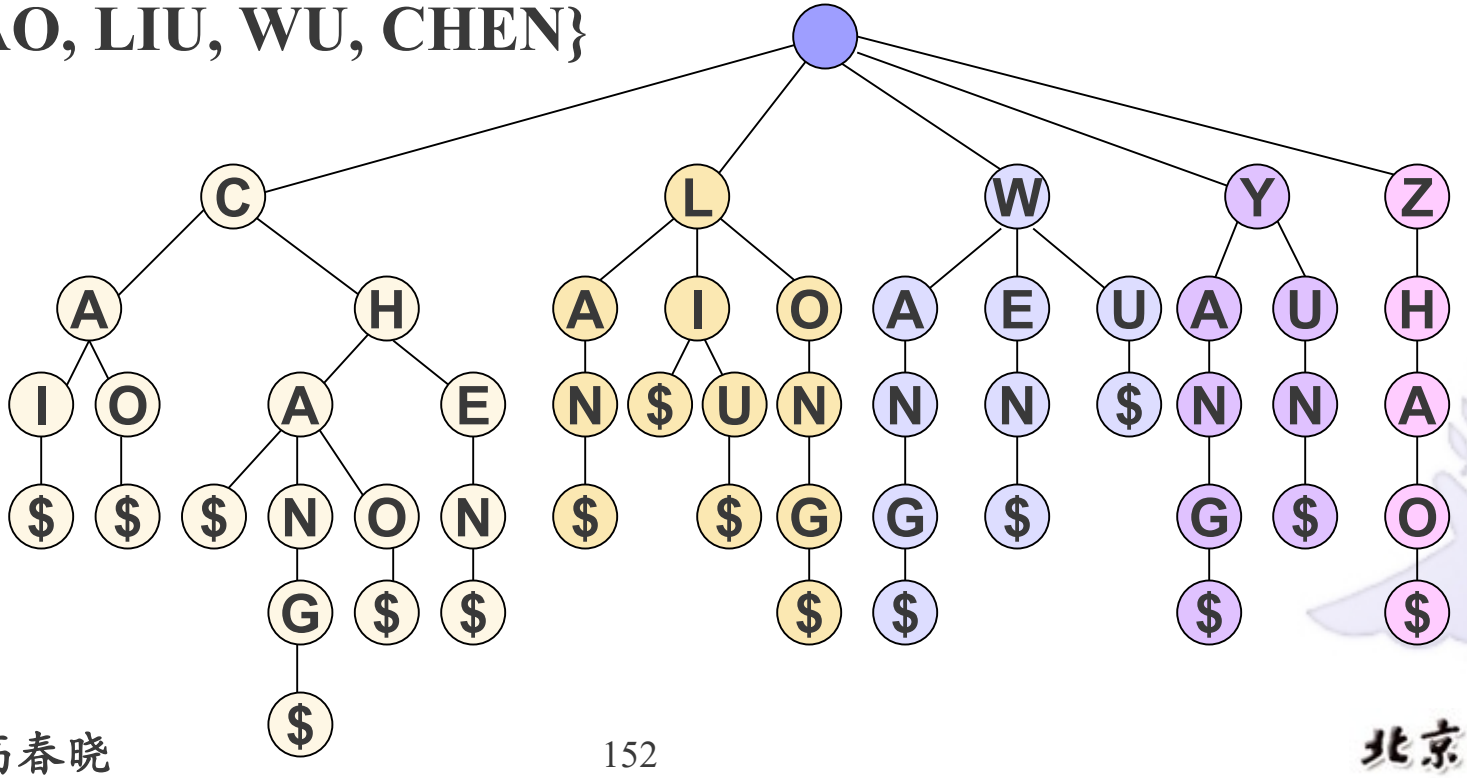
- **4. 每个红色结点的两个子结点必须是黑色的。换句话说：从每个叶子到根的所有路径上不能有两个连续的红色结点 .**
- **5. 从任一结点到其每个叶子的所有路径都包含相同数目的黑色结点。**



2）键树

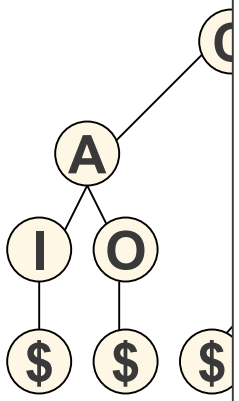
- 2）键树, 又称数字搜索树（Digital Search Tree）
  - ¶ 是一棵度大于等于2 的树
  - ¶ 树中每个结点不是包含关键字, 而含有组成关键字的符号。

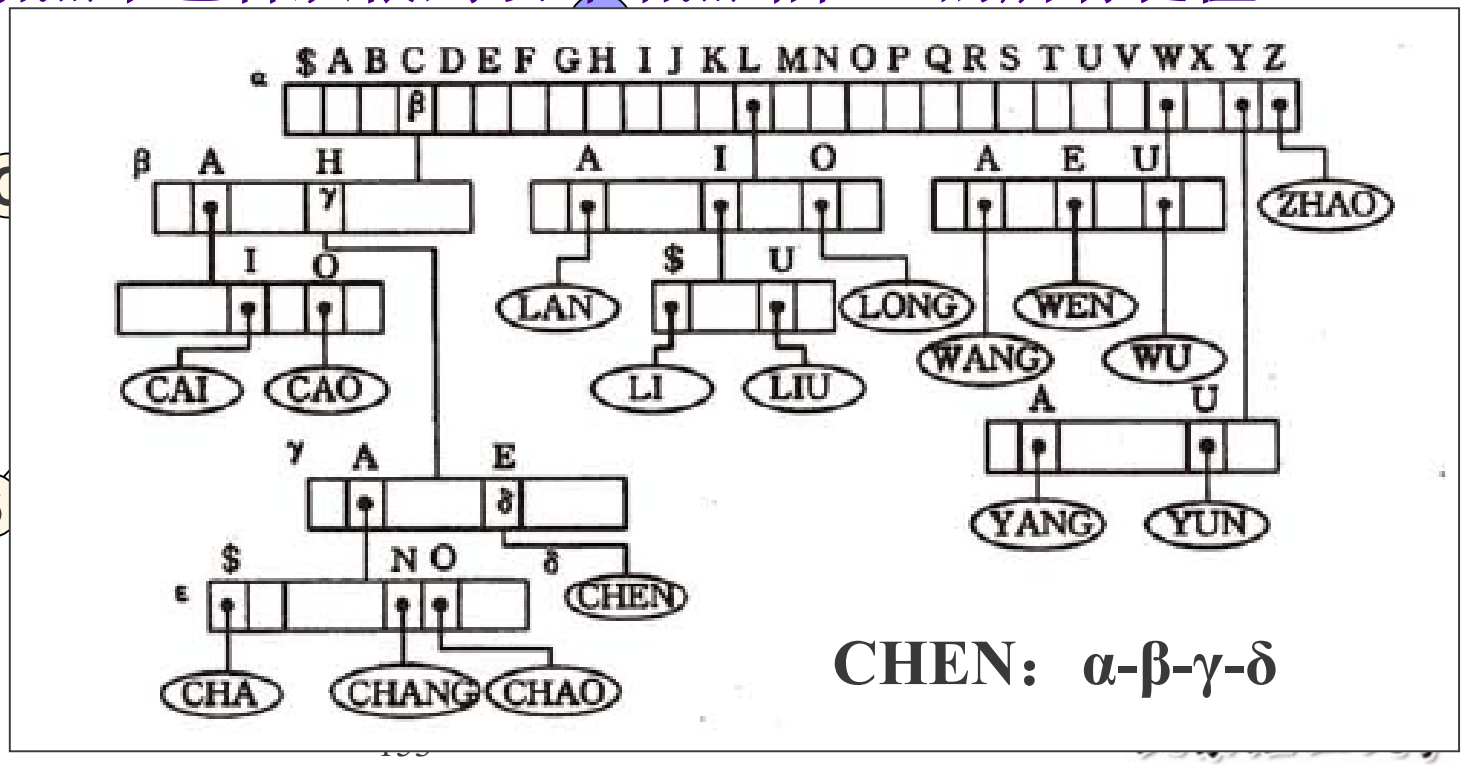
例：将集合进行逐层分割, 得到下列键树{CAI, CAO, LI, LAN, CHA, CHANG, WEN, CHAO, YUN, YANG, LONG, WANG, ZHAO, LIU, WU, CHEN}



2）键树

- 键树的存储方式
- A）树的孩子兄弟链表
- B）树的多重链表（Trie 树）
  - 每个结点都含有d个指针域
  - 每个叶结点中包含从根到该叶结点路径上的所有键值





7.6 散列表及其查找

- 前两节讨论的查找表的各种结构之共同特点：
  1. 记录在表中的位置和它的关键字之间不存在确定关系；
  2. 查找过程为给定值依次和各个关键字比较；
  3. 查找的效率取决于和给定值进行比较的关键字个数。
- 对于频繁使用的查找表,最理想的情况：
  - 根据关键码值，直接找到记录的存储地址
  - 预先知道所查关键字在表中的位置
  - 即, 要求记录的位置和其关键字之间存在确定的关系

- 例如：
  - 为每年招收的 1000 名新生建立一张查找表
  - 其关键字为学号
  - 其值的范围为 xx000 ~ xx999 (前两位为年份)。
- 若以下标为000 ~ 999 的顺序表表示。
- 查找过程：取给定值（学号）的后三位, 不需要经过比较便可直接从顺序表中找到待查关键字。

7.6.1 什么是散列函数

- 如果关键字与记录在表中的存储位置之间建立一个函数关系： $H(key)$
- $H(key)$是关键字为 key 的记录在表中的位置
- 通常称函数$H(key)$ 为散列函数。

- 例如：对于如下 9 个关键字

{Zhao, Qian, Sun, Li, Wu, Chen, Han, Ye, Dai}

- 设 *f*(key) =[Ord(第一个字母) - Ord('A')+1]/2

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	Chen	Dai		Han		Li		Qian	Sun		Wu	Ye	Zhao

添加关键字 Zhou ？

找另一个散列函数？

什么是散列函数

- 1) 散列函数是一个映象, 即：
  - ¶ 将关键字的集合映射到某个地址集合上；
- 2) 散列函数是一个压缩映象, 因此一般情况下, 很容易产生“冲突”现象, 即：
  - ¶ key1< > key2, 而 $H(key1) = H(key2)$。
- 3) 需要找到一种“处理冲突” 的方法

什么是散列表

- **散列表：**根据设定的散列函数 H(key) 和所选中的处理冲突的方法, 将一组关键字映象到一个有限的、地址连续的地址集 (区间) 上, 并以关键字在地址集中的“象”作为相应记录在表中的存储位置, 如此构造所得的查找表称之为“散列表”。
- **使用散列表需要解决的问题：**
  - （1）构造散列函数
  - （2）解决冲突方法

7.6.2 构造散列函数的方法

- 对数字的关键字可有下列构造方法：
  - ¶ 1. 直接定址法 → H(key)为关键字的线性函数
  - ¶ 2. 数字分析法 → 取关键字的若干位组成散列地址
  - ¶ 3. 平方取中法 → 取关键字平方后的中间几位为散列地址
  - ¶ 4. 折叠法 → 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作为散列地址。
  - ¶ 5. 除留余数法 → H(key) = key MOD p
p一般取小于表长(m)的最小素数
或者不包含小于20的质因数的合数
  - ¶ 6. 随机数法

- 若是非数字关键字，则需先对其进行数字化处理

7.6.3 处理冲突的方法

- “处理冲突” 的实际含义是：
  - ¶ 为产生冲突的地址寻找下一个散列地址。
- 例如：关键字集合

k	19	01	23	14	55	68	11	82	36
H	8	1	1	3	0	2	0	5	3

- 设定H(key) = key MOD 11 (表长=11)

- 装填因子 α=记录数/表长　　上例中α = 9/11

1) 开放定址法

- 为产生冲突的地址 H(key) 求得一个地址序列：
  - $H_0, H_1, H_2, \ldots, H_s$　　$1 \le s \le m-1$
  - 其中：$H_0 = H(key)$
  - $Hi = (H(key) + d_i) \text{ MOD } m$　　$i=1, 2, \ldots, s$
  - d_i称为增量
- 对增量 *di* 有三种取法：

1) 开放定址法

- 对增量 *di* 有三种取法：
- 1)线性探测再散列
  - ¶ $d_i = c \times i$
  - ¶ 最简单的情况 $c=1$, $d_i=1, 2, 3, 4, \ldots\ldots$
- 2) 平方探测再散列
  - ¶ $d_i = 1^2, -1^2, 2^2, -2^2, 3^2, -3^2, \ldots, \pm k^2,$
- 3) 随机探测再散列
  - ¶ d_i 是一组伪随机数列 或者
  - ¶ $d_i = i \times H_2(key)$ (又称双散列函数探测)

k	19	01	23	14	55	68	11	82	36
H	8	1	1	3	0	2	0	5	3

•设定H(key) = （key + d_i ）MOD 11 (表长=11)

1）若采用线性探测再散列处理冲突: $di = i$

0	1	2	3	4	5	6	7	8	9	10	
55	01	23	14	68	11	82	36	19			
1	1	2	1	3	6	2	5	1			成功
10	9	8	7	6	5	4	3	2	1	1	不成功

成功 ASL=(1+1+2+1+3+6+2+5+1)/9 = 22/9 = 2.44

不成功 ASL=(10+9+8+7+6+5+4+3+2+1+1)/11 = 56/11 = 5.09

查找不成功时的ASL = 总查找次数 / 有效表长

k	19	01	23	14	55	68	11	82	36
H	8	1	1	3	0	2	0	5	3

•设定H(key) = （$H_0 + d_i$ ）MOD 11 (表长=11)

2）若采用平方探测再散列处理冲突： $d_i = 1^2, -1^2, 2^2, -2^2, 3^2, -3^2, \ldots, \pm k^2$, (地址对11求余)

0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	36	82	68		19		11
1	1	2	1	2	1	4		1		2

ASL=(1+1+2+1+2+1+4+1+2)/9 = 15/9 = 1.67

3）再散列法

- Hi = (H(*key*) + d_i) MOD m　　*i=1，2，…，s*
  - ¶ $d_i = i \times H_2(key)$（双散列函数探测）
  - ¶ H_2(key) 是另一个散列函数，它的函数值应和 m 互为素数。
- 或者Hi = RHi(*key*)
  - ¶ 其中Rhi均是不同的散列函数

k	19	01	23	14	55	68	11	82	36
H	8	1	1	3	0	2	0	5	3

- 设定H(key) = （key + d_i ）MOD 11 (表长=11)
- 冲突：01－23, 55－11, 14－36

当 m=11时，可设：$H_2(key) = (3*key) \%10 + 1$

当冲突时 $d_i = i \times H_2(key)$

0	1	2	3	4	5	6	7	8	9	10
23	01	68	14	11	82	55		19		36
2	1	1	1	2	1	2		1		3

$H_2($ $+ 1 = 9$

$H(11) = (11+4) \% 11 = 4$ $H(36) = (36+2*9) \% 11 = 10$

ASL=(2+1+1+1+2+1+2+1+3)/9 = 15/9 = 1.67

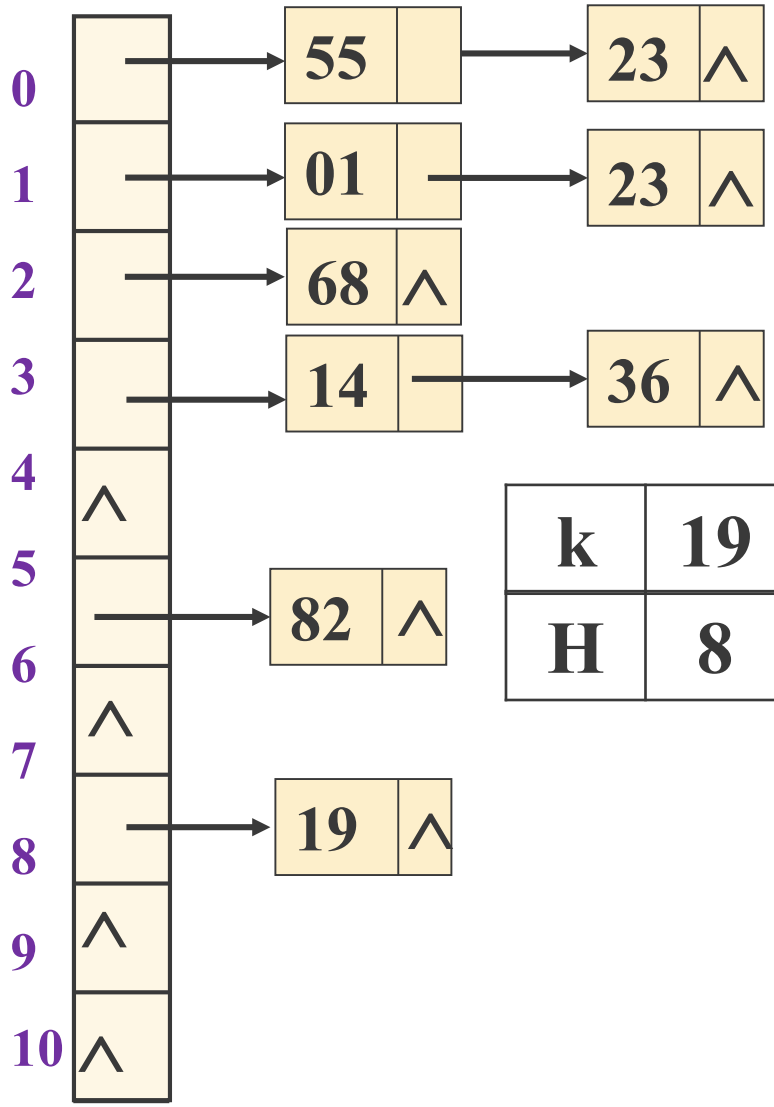
散列表的结构定义

```
enum KindofState{ Active, Blank, Deleted};
typedef int KeyType;    //结点关键码数据类型
typedef struct  {
    KeyType key;                //关键字
} HElemType;
typedef struct  {
    int divisor;                //散列函数的除数
    int n, m;                   //当前已用地址数和散列表大小
    HElemType *data;            //散列表存储数组
    KindofState *state;         //状态数组
    int *count;                 //探查次数数组
} HashTable;
```

散列表的查找过程

- 查找过程和造表过程一致。假设采用开放定址处理冲突, 则查找过程为:
- 对于给定值 K, 计算散列地址 i = H(K)
- 若 r[i] = NULL 则查找不成功
- 若 r[i].key = K 则查找成功
- 否则 “求下一地址 Hi”, 直至
- r[Hi] = NULL (查找不成功)
- 或 r[Hi].key = K (查找成功) 为止。

3） 链地址法



- 将所有散列地址相同的记录都链接在同一链表中。

H(key)=key MOD 11

k	19	01	23	14	55	68	11	82	36
H	8	1	1	3	0	2	0	5	3

ASL=(6×1+2×2)/9=10/9

3）公共溢出区

- 假设散列函数的值域为[0, m-1]，则设向量 HashTable [0…m-1] 为基本表；
- 另设立向量OverTable[0…v]为溢出表。
- 一旦发生溢出，都填入溢出表

§7.3.5 散列表查找性能分析

- 与散列表查找性能相关因素：
  - 散列函数（一般假设散列函数是均匀的）
  - 处理冲突方法
  - 装载因子
- 若散列函数是均匀的，则散列表平均查找长度不依赖于散列函数。
- 若处理冲突方法相同，则散列表平均查找长度依赖于装载因子α 。

•装填因子 α=记录数/表长

§7.3.5 散列表查找性能分析

- 散列表查找成功的ASL
- 线性探测再散列

$$S_{nl} \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$$

- 随机探测、二次探测再散列和再散列

$$S_{nr} \approx -\frac{1}{\alpha}\ln(1-\alpha)$$

- 链地址法

$$S_{ne} \approx 1+\frac{\alpha}{2}$$

•装填因子 α=记录数/表长

§7.3.5 散列表查找性能分析

- 散列表查找不成功的ASL
- 线性探测再散列

$$U_{nl} \approx \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$$

- 随机探测再散列

$$U_{nr} \approx \frac{1}{1-\alpha}$$

- 链地址法

$$U_{ne} \approx \alpha + e^{-\alpha}$$

•装填因子 α=记录数/表长

本章学习要点

- 1. 熟练掌握顺序表和有序表的查找方法及其平均查找长度的计算方法。
- 2. 熟练掌握二叉查找树和平衡二叉树的构造和查找方法。
- 3. 理解B- 树的特点以及它们的建树和查找的过程。
- 4. 熟练掌握散列表的构造方法, 深刻理解散列表与其它结构的表的实质性的差别。

种类	具体实例
二叉树	二叉查找树、Van Emde Boas树、笛卡尔树、Top树、T-树
平衡二叉树	红黑树、平衡二叉查找树、AA树、伸展树、替罪羊树、Treap
B+树	B+树、B*树、UB+树、2-3树、（a,b）树，舞蹈树、H树、B^X-树
Tries	后缀树、基树、Ternary Search 树
二叉区分树	Quad树、OC树、KD-树、VP-树
非二叉树	指数树、聚合树、区间树、PQ树、SPQR树
图形树	R-树、X-树、段树
其他树	堆散列树、手指树、度量树、覆盖树、BK树、双链树、最小期望树

END