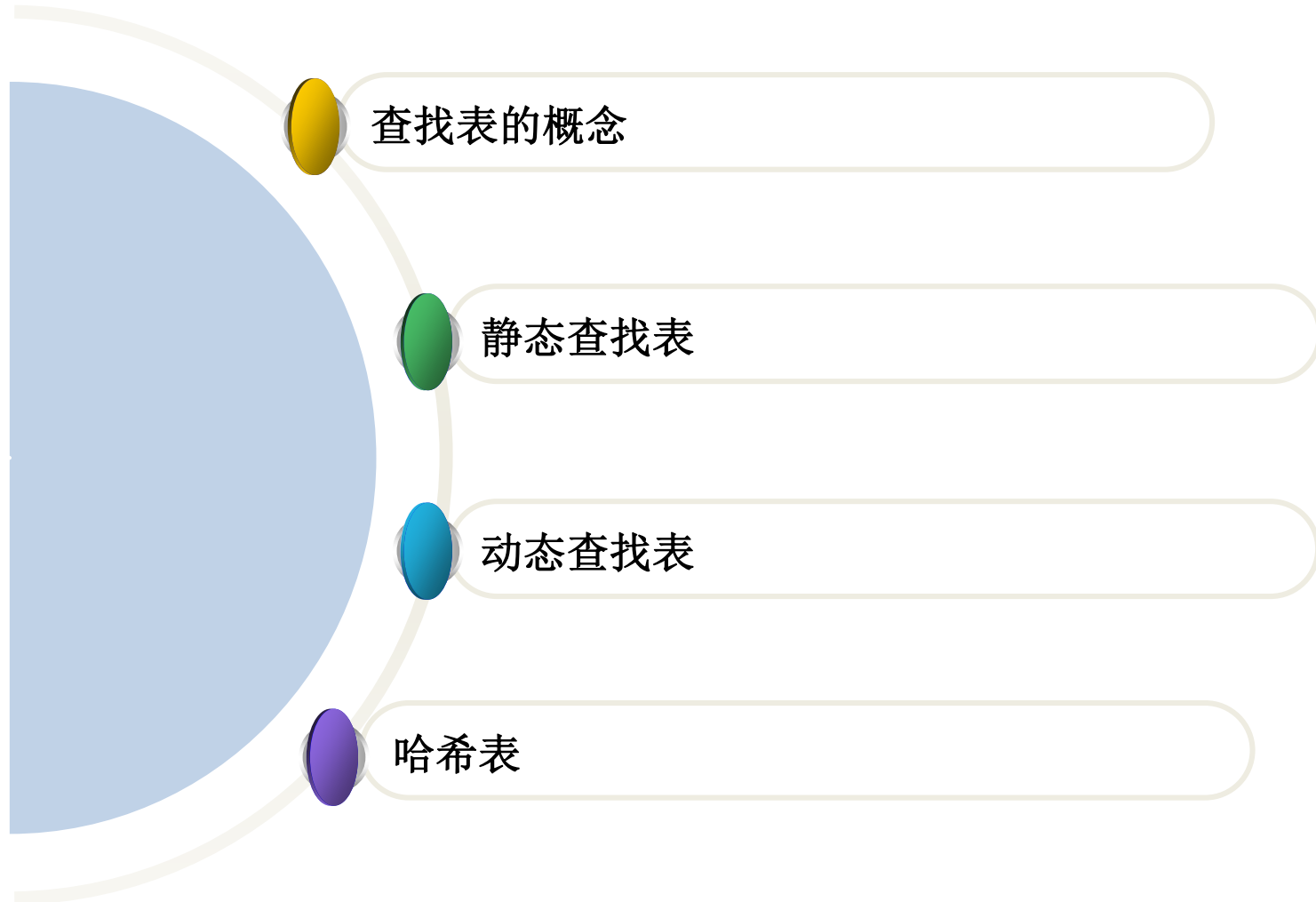


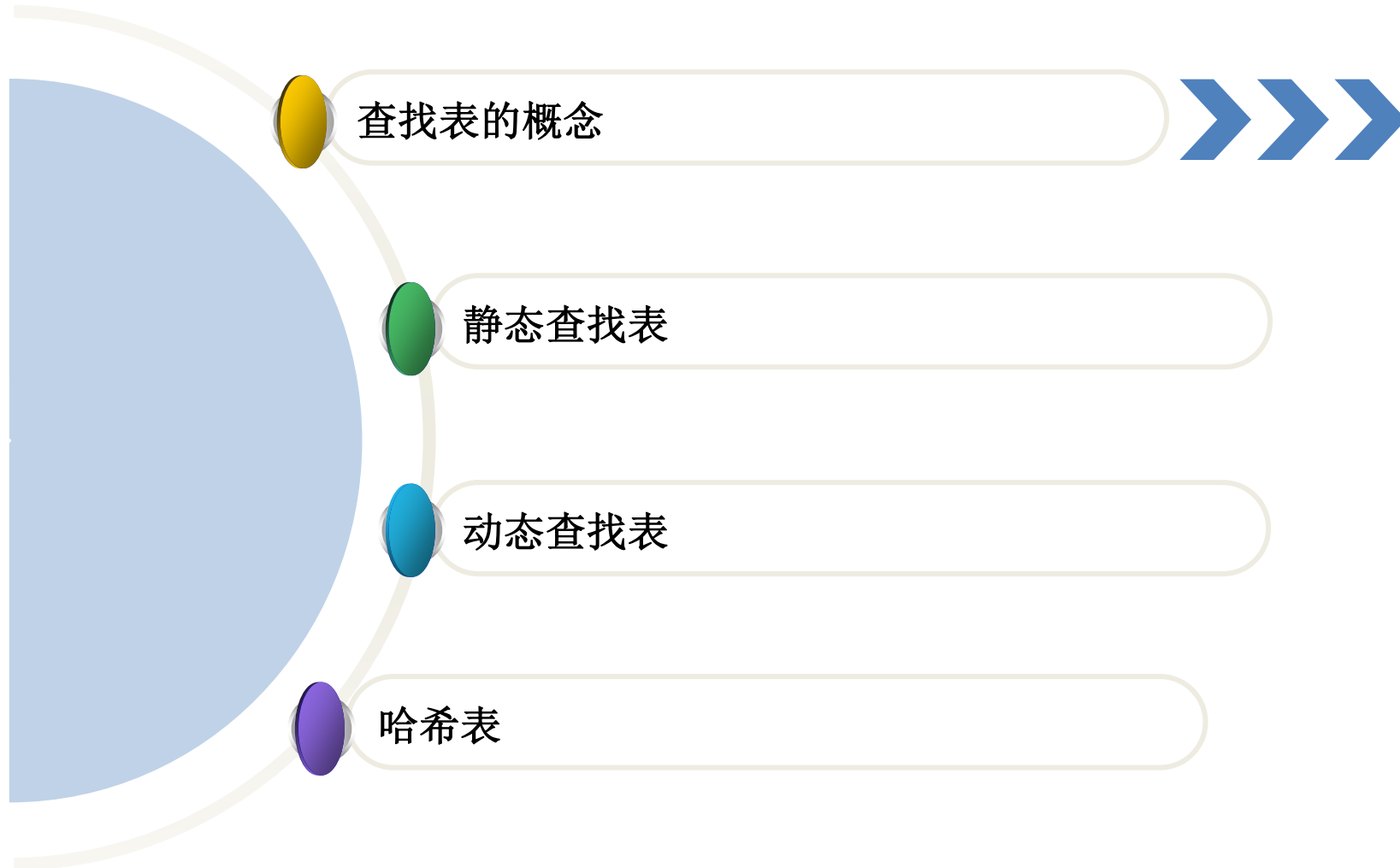
DS—第九章

查找表—Searching

# 第七章回顾

- 1、了解图的基本概念，掌握图的邻接矩阵、邻接表这两种存储结构及其构造方法；
- 2、熟练掌握图的两种遍历方法；
- 3、熟练掌握构造最小生成树的方法，并理解算法；
- 4、掌握 AOV 网的拓扑排序方法，并理解算法；
- 5、掌握求解关键路径的方法；
- 6、理解用 Dijkstra 方法求解单源点最短路径问题。



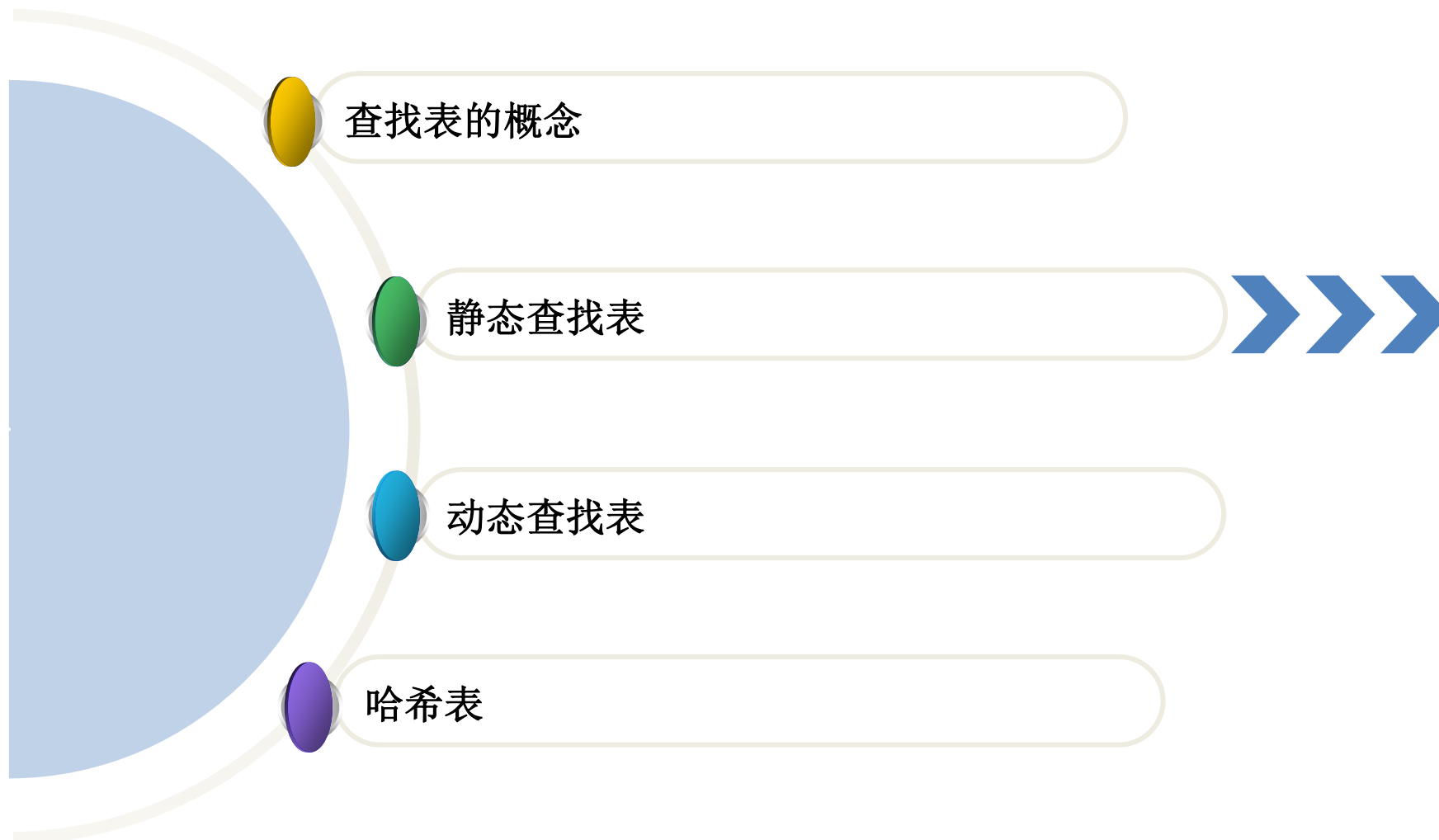


# 查找相关概念

- **查找表**：由同一类型的数据元素(或记录)构成的集合。对查找表进行的经常操作为：查找、检索、增加、删除。
- **静态查找表**：对查找表只进行前两种操作。
- **动态查找表**：不仅限于前两种操作。
- **关键字**：数据元素中某个数据项的值，用以标识一个数据元素，如果是唯一标识，则称为主关键字。
- **查找是否成功**：根据给定的值，在查找表中确定一个其关键字等于给定值的元素，如果表中存在这样元素，则称查找成功，否则，不成功。

## 查找相关概念 (续)

```
typedef float KeyType
typedef int KeyType
typedef char * KeyType
typedef struct
{
    KeyType key;
    .....
}
#define EQ(a ,b) ((a)==(b))//a,b为数值型
#define LT(a ,b) ((a) < (b))
#define LQ(a,b) ((a)<=(b))
```



# 顺序查找

顺序查找：从表的一端开始，逐个进行记录的关键字和给定值的比较。

查找过程：

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

```
typedef struct {
```

```
    ElemType * elem;
```

```
    int length; // 表长度
```

```
} SSTable;
```

} 静态查找表的顺序存储结构



# 顺序查找(续)

算法描述:

```
int Search_Seq(SSTable ST, KeyType key)
{
    ST.elem[0].key = key ;

    for (i = ST.length ; ST.elem[i].key != key ; -- i)
        ;

    if (i <= 0) break ;

    if (i > 0) return i ;

    else return 0 ;
}
```

监视哨

当 ST.length >= 1000  
时, 此改进能使进行  
一次查找所需的平均  
时间几乎减少一半。

|    |   |    |    |    |    |    |    |    |    |    |    |
|----|---|----|----|----|----|----|----|----|----|----|----|
| 0  | 1 | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 |
| 60 | 5 | 37 | 19 | 21 | 13 | 56 | 64 | 92 | 88 | 80 | 75 |

# 效率分析

平均查找长度 (ASL) 为确定记录在查找表中的位置, 需要和给定值进行比较的记录个数的期望值称为算法在查找成功时的平均查找长度。衡量查找算法好坏的依据。

对含有  $n$  个记录的表, 查找成功时:

$$ASL = \sum_{i=1}^n P_i C_i$$

找到第  $i$  个记录  
需比较的次数。

第  $i$  个记录被  
查找的概率。

$$\sum_{i=1}^n P_i = 1$$

顺序查找的平均查找长度:

$$ASL = nP_1 + (n-1)P_2 + \dots + 2P_{n-1} + P_n$$

假设每个记录的查找概率相等:  $P_i = 1/n$

则:

$$ASL_{SS} = \sum_{i=1}^n P_i C_i = \frac{1}{n} \sum_{i=1}^n (n-i+1) = \frac{n+1}{2}$$

# 效率分析(续)

若将查找不成功的情况也考虑在内

查找不成功时，关键字的比较次数总是  $n + 1$  次。

假设：1、查找成功与不成功的概率相同。

2、每个记录被查找的概率相同。

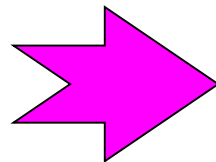
则：平均查找长度（成功与不成功的平均查找长度之和）  
为

$$ASL_{ss} = \frac{1}{2n} \sum_{i=1}^n (n - i + 1) + \frac{1}{2} (n + 1) = \frac{3(n + 1)}{4}$$

以后我们仅讨论查找成功时的平均查找长度和查找不成功时的比较次数。

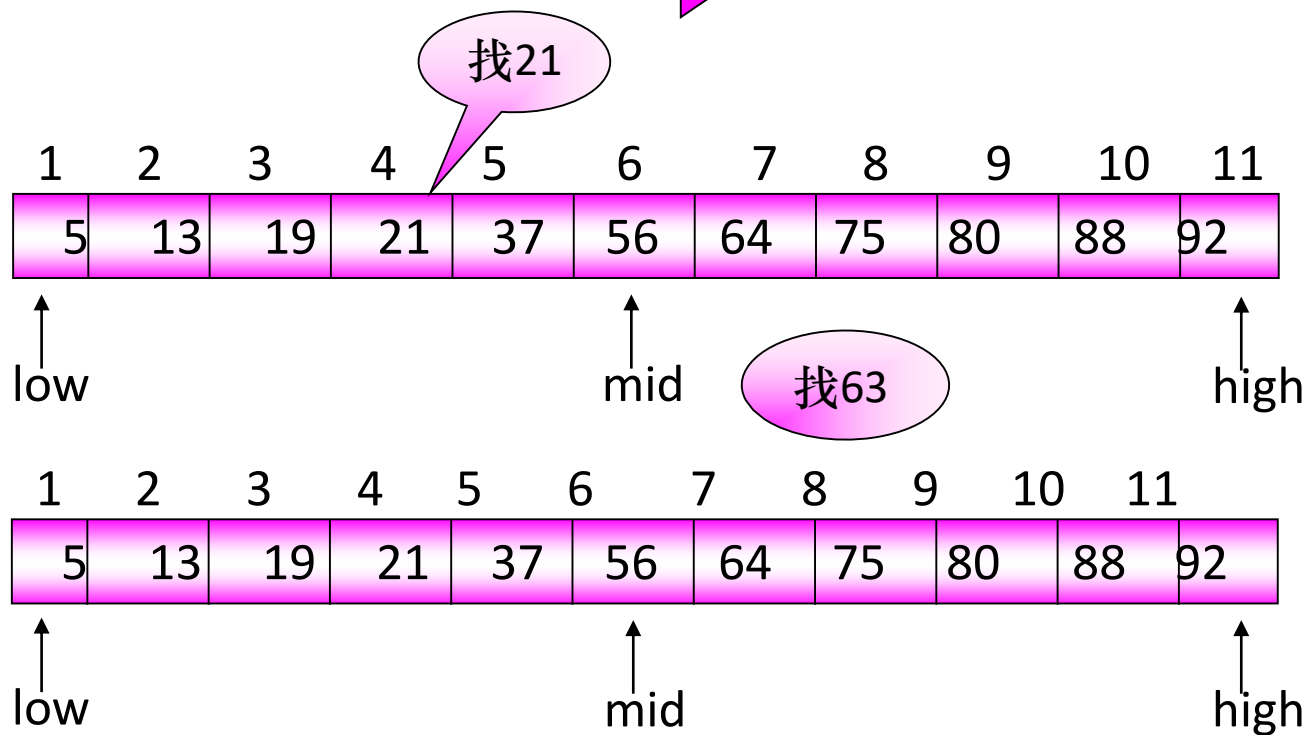
# 折半查找

## 有序表表示静态查找表



## 折半查找

### 查找过程：



High < low

## 算法描述：

```
int Search_Bin ( SSTable ST, KeyType key )
{ low = 1 ;    high = ST.length ;    // 置区间初值
  while (low <= high)
  {  mid = (low + high) / 2 ;
    if (ST.elem[mid].key = key) return mid ; // 找到待查元素
    else if (key < ST.elem[mid].key)
        high = mid - 1;    // 继续在前半区间进行查找
    else low = mid + 1;    // 继续在后半区间进行查找
  }
  return 0 ;    // 顺序表中不存在待查元素
} // Search_Bin
```

性能分析:

|       |   |    |    |    |    |    |    |    |    |    |    |
|-------|---|----|----|----|----|----|----|----|----|----|----|
| $i$   | 1 | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 |
|       | 5 | 13 | 19 | 21 | 37 | 56 | 64 | 75 | 80 | 88 | 92 |
| $C_i$ | 3 | 4  | 2  | 3  | 4  | 1  | 3  | 4  | 2  | 3  | 4  |

查找成功:

比较次数 = 路径上的结点数

比较次数 = 结点4的层数

比较次数

$\wedge$

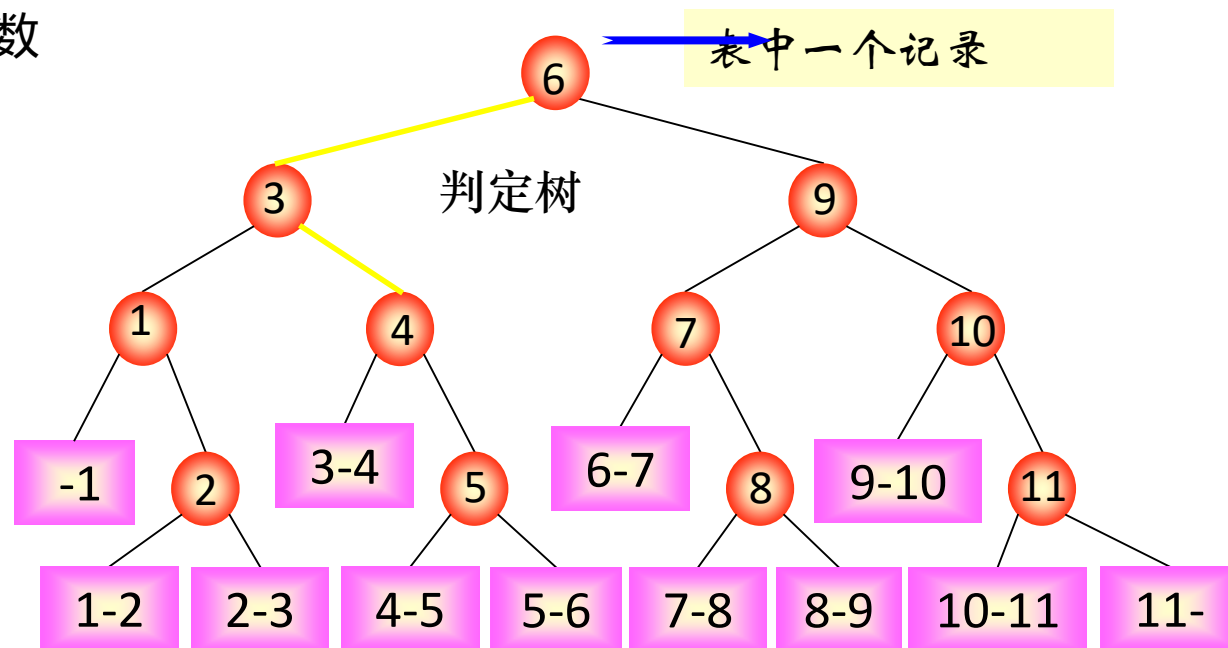
树的深度

$$\lfloor \log_2 n \rfloor + 1$$

查找不成功:

比较次数 = 路径上的内部结点数

$$\text{比较次数} \leq \lfloor \log_2 n \rfloor + 1$$



# 效率分析

平均查找长度ASL（成功时）：

设表长  $n = 2^h - 1$ ，则  $h = \log_2(n + 1)$ （此时，判定树为深度 =  $h$  的满二叉树），且表中每个记录的查找概率相等：

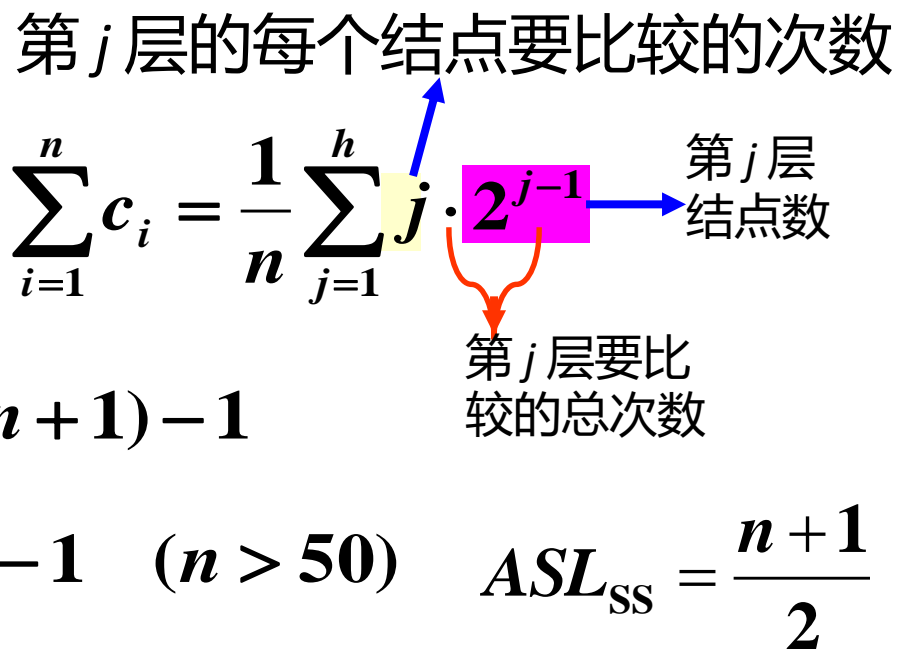
$$P_i = 1/n。$$

第  $j$  层的每个结点要比较的次数

则：

$$\begin{aligned} ASL_{bs} &= \sum_{i=1}^n p_i c_i = \frac{1}{n} \sum_{i=1}^n c_i = \frac{1}{n} \sum_{j=1}^h j \cdot 2^{j-1} \\ &= \frac{n+1}{n} \log_2(n+1) - 1 \\ &\approx \log_2(n+1) - 1 \quad (n > 50) \end{aligned}$$

ASL<sub>ss</sub> =  $\frac{n+1}{2}$



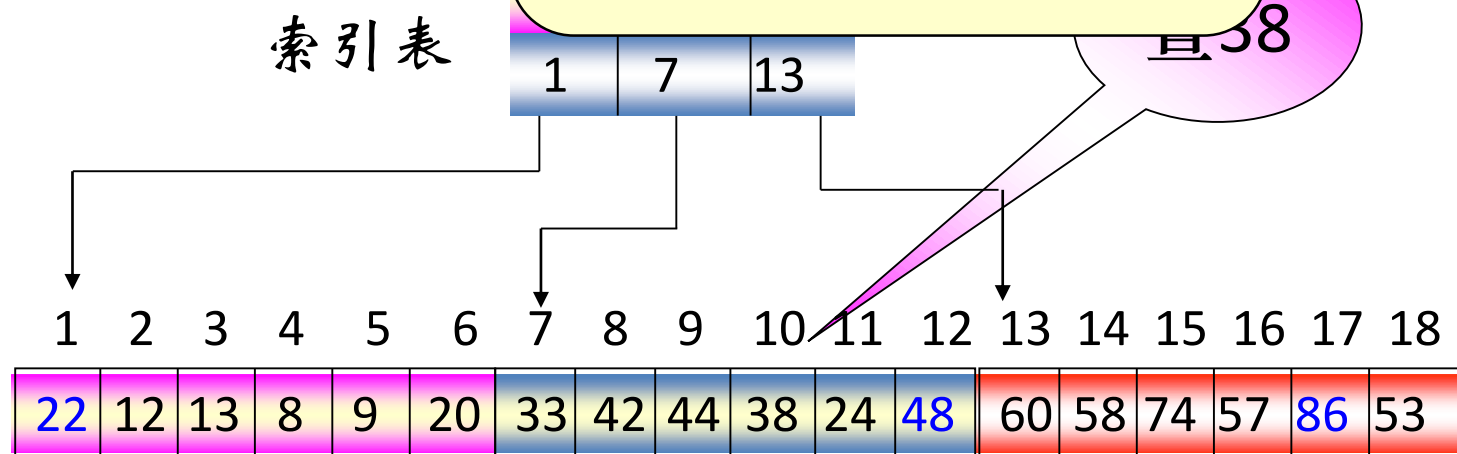
折半查找**优点**：效率比顺序查找高。

折半查找**缺点**：只适用于**有序表**，且限于**顺序存储结构**。

# 索引查找 (分块查找)

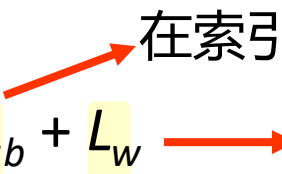
- 条件:**
- 1、将表分成几块，且表或者有序，或者**分块有序**；
  - 2、建立“索引表”（每个结点含有最大关键字域和指向本块第一个结点的指针，且按关键字有序）。

查找过程：先确定待查找记录所在块（通过索引表查找），再在块内查找。





# 性能分析

平均查找长度： $ASL_{bs} = L_b + L_w$   在索引表中查找所在块的平均查找长度  
在块中查找元素的平均查找长度

若将长度为  $n$  的表均分成  $b$  块，每块含  $s$  个记录 ( $b = \lceil n/s \rceil$ )；  
并设表中每个记录的查找概率相等，则每块查找的概率为  $1/b$ ，  
块中每个记录的查找概率为  $1/s$ 。

(1)、用顺序查找确定所在块：

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

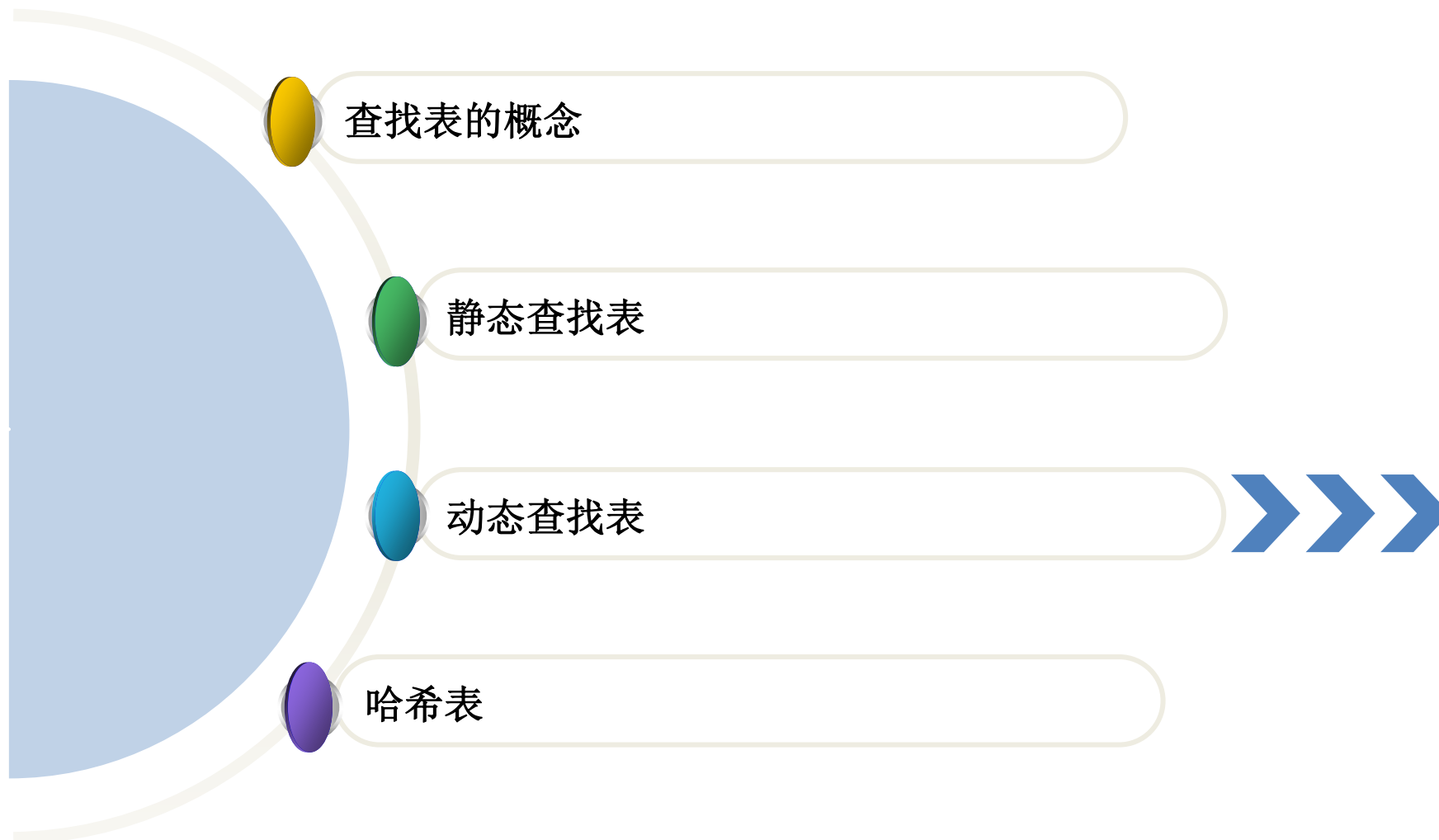
可以证明，当取  $\sqrt{n}$  时， $ASL_{bs}$  取最小值  $\sqrt{n} + 1$ 。  比顺序查好  
比折半查差

(2)、用折半查找确定所在块：

$$ASL_{bs} \approx \log_2(b+1) - 1 + \frac{s+1}{2} \approx \log_2\left(\frac{n}{s} + 1\right) + \frac{s}{2}$$

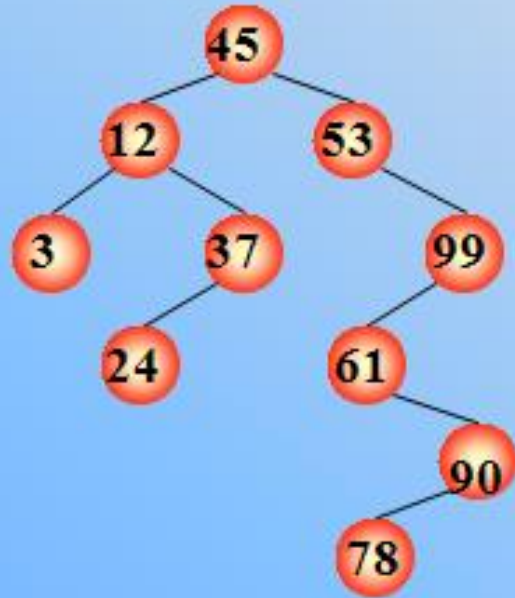
# 查找方法比较(静态查找表)

|            | 顺序查找     | 折半查找 | 分块查找     |
|------------|----------|------|----------|
| <i>ASL</i> | 最大       | 最小   | 中间       |
| 表结构        | 有序表、无序表  | 有序表  | 分块有序     |
| 存储结构       | 顺序表、线性链表 | 顺序表  | 顺序表、线性链表 |



# 二叉排序树

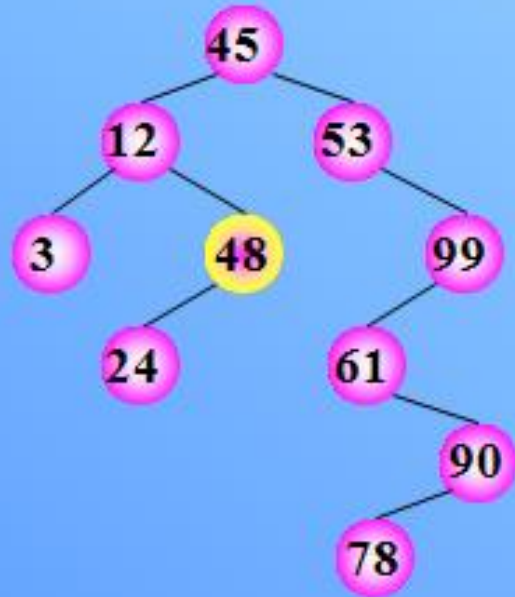
左  
子  
分



二叉排序树



的  
右  
也



非二叉排序树



# 二叉排序树的查找过程

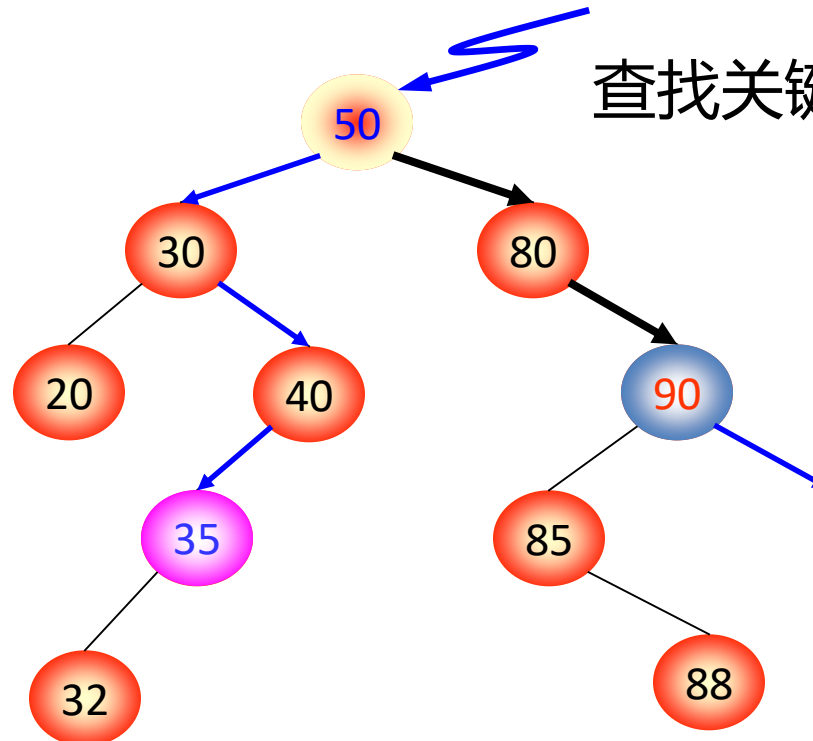
若二叉排序树为空，则查找不成功；否则

- 1) 若给定值等于根结点的关键字，则查找成功；
- 2) 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；
- 3) 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。

例：

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



查找关键字：50, 35, 90, 95

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。

——查找失败

## 算法描述(用二叉链表作二叉排序树的存储结构)

BiTree SearchBST(BiTree T, KeyType key)

{

// 在根指针 T 所指二叉排序树中递归地查找某关键字等于 key  
// 的数据元素。若查找成功，则返回指向该数据元素结点的指  
// 针，否则返回空指针。

if ((!T) || key = T-> data.key) return(T);

else if ( key < T-> data.key)

return(SearchBST (T-> lchild, key));

// 在左子树中继续查找

else return(SearchBST (T-> rchild, key));

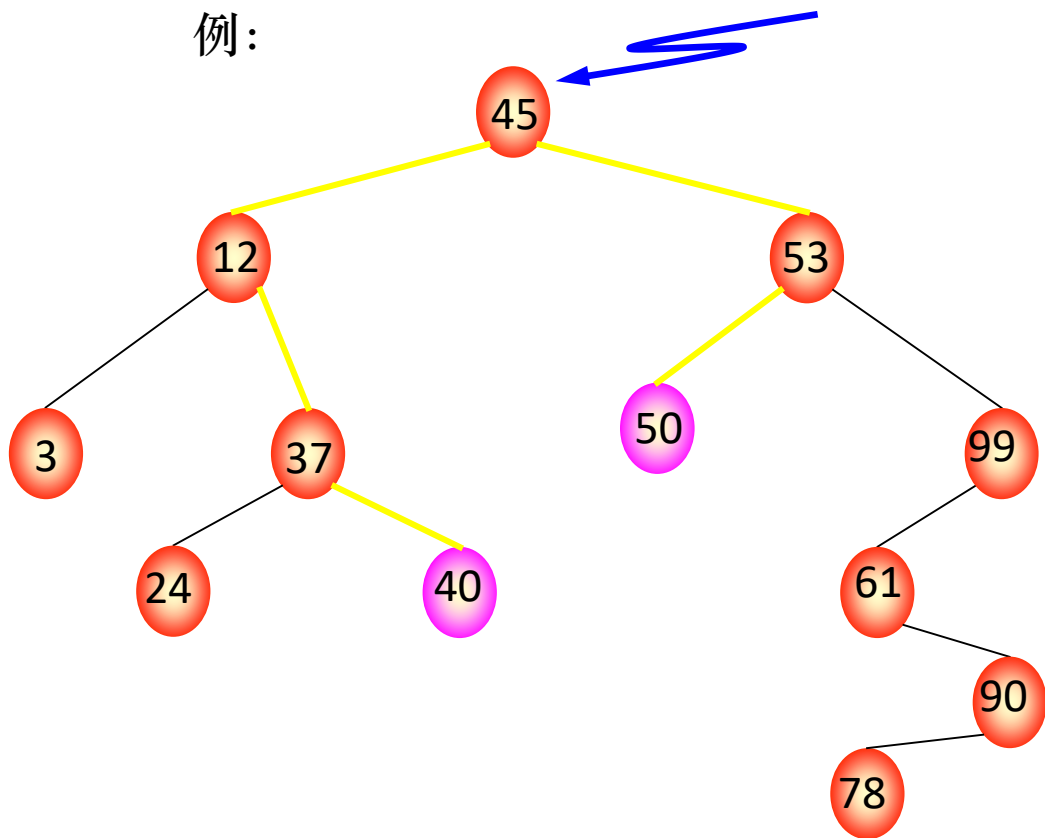
// 在右子树中继续查找

} // SearchBST

# 二叉排序树的插入

- 1)、若二叉排序树为空树，则新插入的结点为根结点；
- 2)、若二叉排序树非空，则新插入的结点必为一个新的叶子结点，并且是查找不成功时查找路径上访问的最后一个结点的左孩子或右孩子结点。

例：



插入 40， 是 37 的右孩子。

插入 50， 是 53 的左孩子。

# 查找算法（修改版）

```
Status SearchBST (BiTree T, KeyType key, BiTree f, BiTree &p )
{ // 在根指针 T 所指二叉排序树中递归地查找其关键字等于 key
  // 的数据元素，若查找成功，则指针 p 指向该数据元素结点，
  // 并返回 TRUE，否则指针 p 指向查找路径上访问的最后一个
  // 结点并返回 FALSE，指针 f 指向 T 的双亲，其初始调用值
  // 为NULL。
  if (!T) { p = f; return FALSE; } // 查找不成功
  else if ( key = T-> data.key ) { p = T; return TRUE; } // 查找成功
    else if ( key < T-> data.key )
      SearchBST (T -> lchild, key, T, p ); // 在左子树中查找
    else SearchBST (T-> rchild, key, T, p ); // 在右子树中查找
} // SearchBST
```



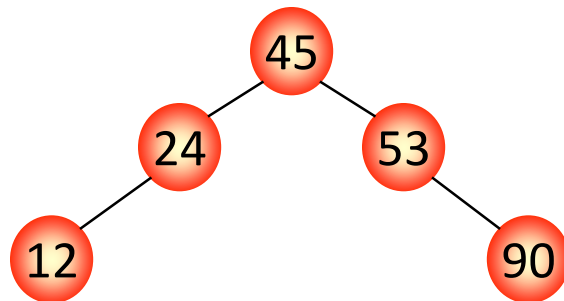
# 插入算法

```
Status Insert BST(BiTree &T, ElemType e ) {  
    // 当二叉排序树 T 中不存在关键字等于 e.key 的数据元素时 ,  
    // 插入 e 并返回 TRUE , 否则返回 FALSE  
    if (!SearchBST ( T, e.key, NULL, p )) { // 查找不成功  
        s = (BiTree) malloc (sizeof (BiTNode));  
        s -> data = e; s -> lchild = s -> rchild = NULL;  
        if ( !p ) T = s; // 插入 s 为新的根结点  
        else if ( e.key < p -> data.key ) p -> lchild = s; // 插入 s 为左孩子  
            else p -> rchild = s; // 插入 s 为右孩子  
        return TRUE;  
    }  
    else return FALSE; // 树中已有关键字相同的结点 , 不再插入  
} // Insert BST
```

## 二叉排序树生成

从空树出发，经过一系列的查找、插入操作之后，可生成一棵二叉排序树。

例：设查找的关键字序列为 {45, 24, 53, 45, 12, 24, 90}，可生成二叉排序树如下：



中序遍历二叉排序树可得到一个关键字的有序序列。

一个无序序列可通过构造二叉排序树而变成一个有序序列。构造树的过程就是对无序序列进行排序的过程。

插入的结点均为叶子结点，故无需移动其他结点。相当于在有序序列上插入记录而无需移动其他记录。

二叉排序树既有类似于折半查找的特性，又采用了链表作存储结构。

# 二叉排序树的删除

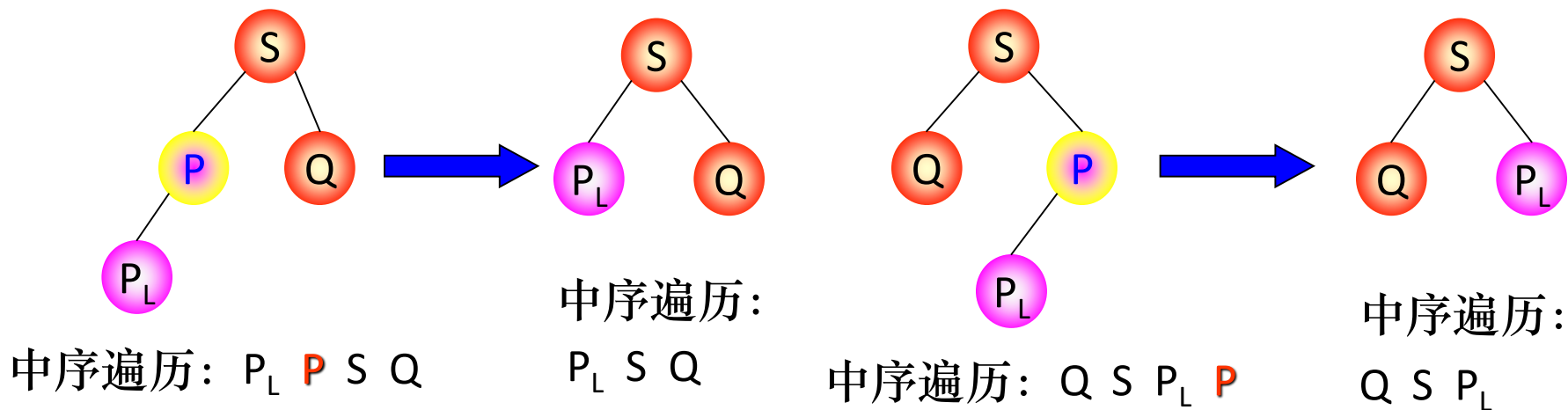
**要求：**在删除某个结点之后，仍然保持二叉排序树的特性。

删除二叉排序树中的 \*p 结点，分三种情况讨论：

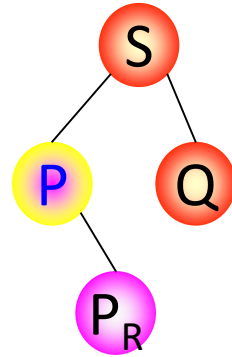
1)、\*p 为叶子结点。因删除叶子结点不破坏树的结构，故只需修改 \*p 双亲 \*f 的指针： $f \rightarrow lchild = \text{NULL}$  或  $f \rightarrow rchild = \text{NULL}$

2)、\*p 只有左子树或右子树：

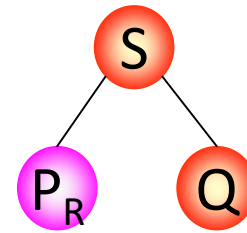
\*p 只有左子树，用 \*p 的左孩子代替 \*p；



\*p 只有右子树，用 \*p 的右孩子代替 \*p；

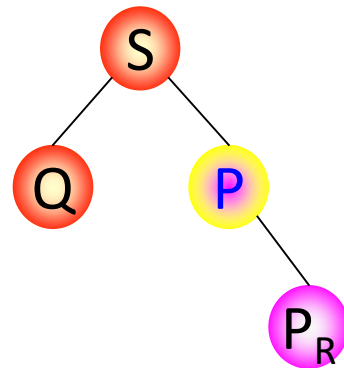


中序遍历: **P** P<sub>R</sub> S Q

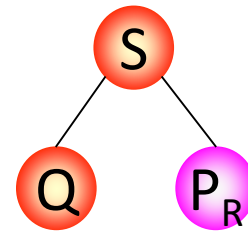


中序遍历:

P<sub>R</sub> S Q



中序遍历: Q S **P** P<sub>R</sub>



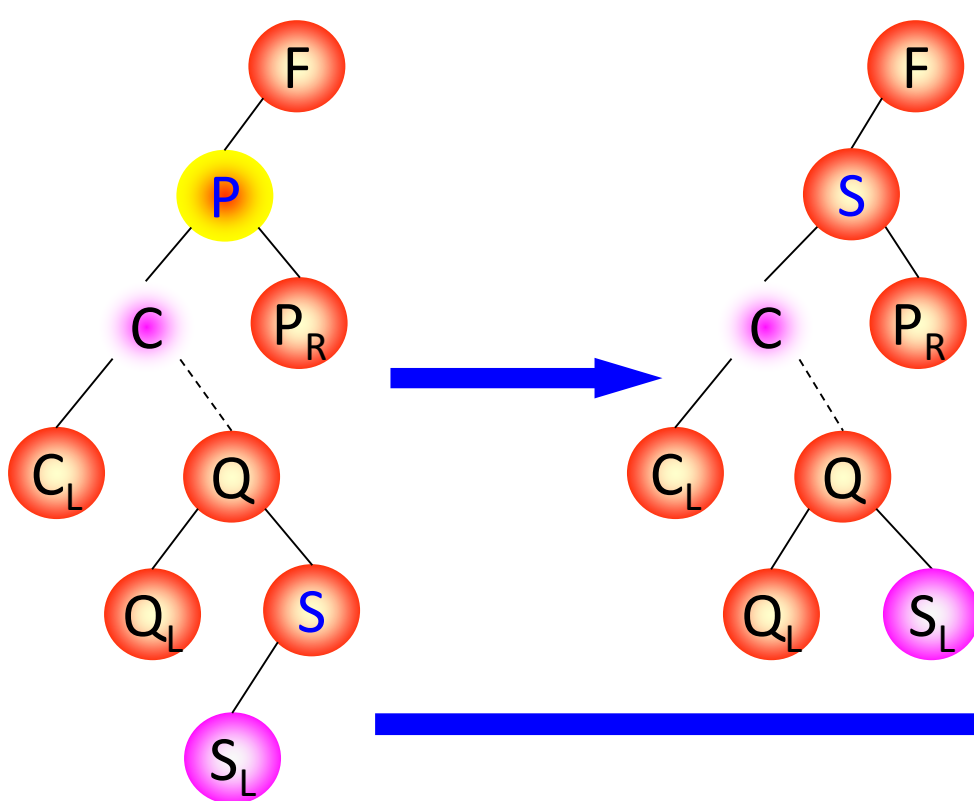
中序遍历:

Q S P<sub>R</sub>

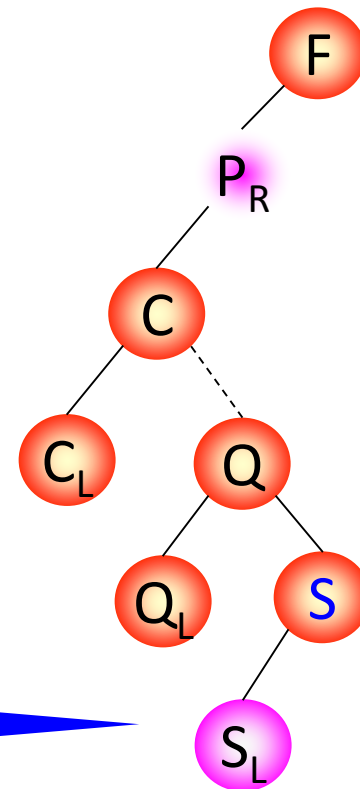
3)、\*p 左、右子树均非空：

用 \*p 的直接前驱取代 \*p。

用 \*p 的直接后继取代 \*p。

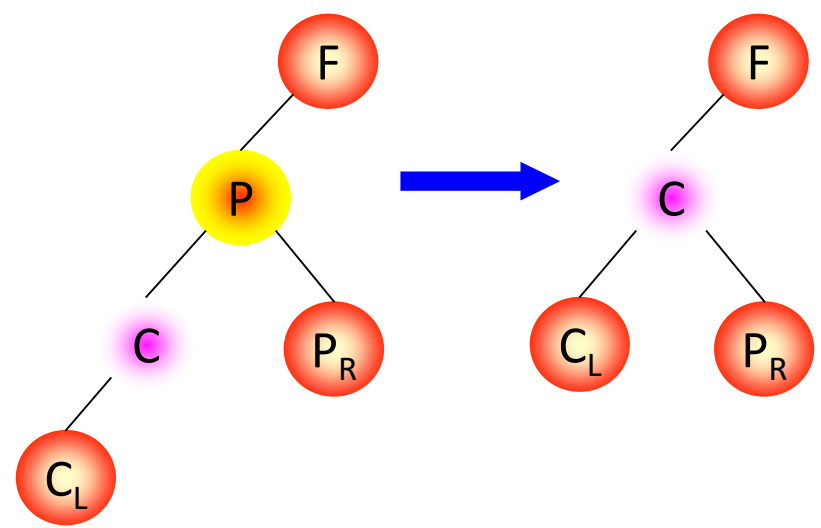


中序：C<sub>L</sub> C ... Q<sub>L</sub> Q S<sub>L</sub> S **P** P<sub>R</sub> F



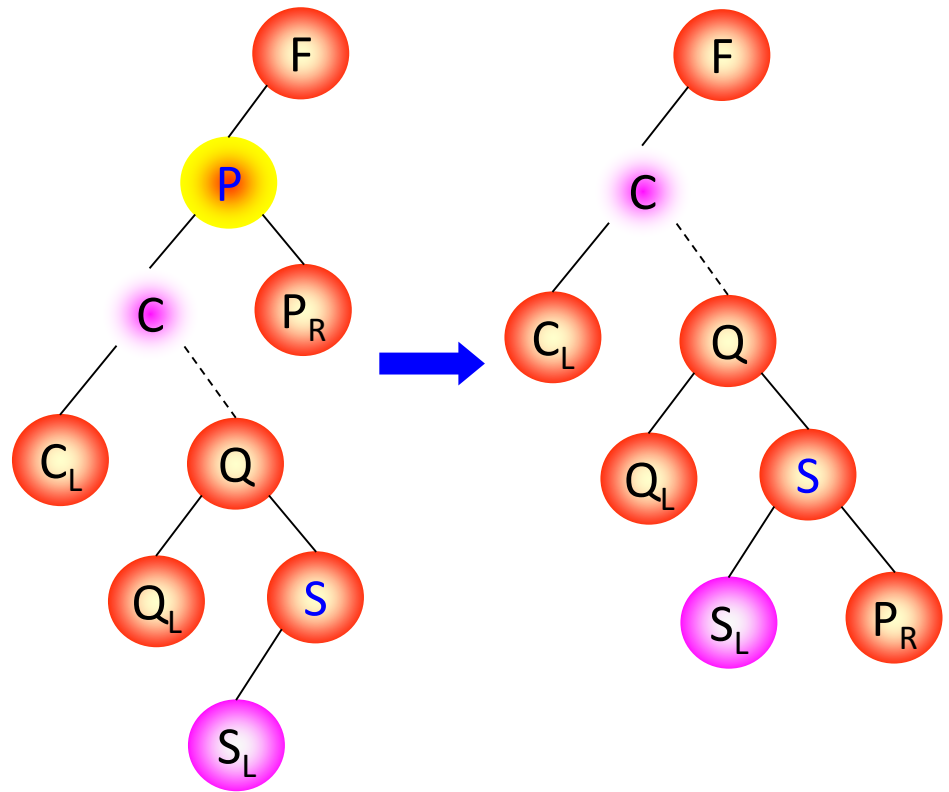
中序：C<sub>L</sub> C ... Q<sub>L</sub> Q S<sub>L</sub> S P<sub>R</sub> F

用 \*p 的左子树取代 \*p。



中序遍历:  $C_L$  C **P**  $P_R$  F

中序遍历:  $C_L$  C  $P_R$  F



中序:  $C_L$  C ...  $Q_L$  Q  $S_L$  S **P**  $P_R$  F

中序:  $C_L$  C ...  $Q_L$  Q  $S_L$  S  $P_R$  F

# 二叉排序树的查找分析

二叉排序树上查找其关键字  
等于给定值的结点的过程，  
其实就是走了一条从根到该  
结点的路径。

比较的关键字次数

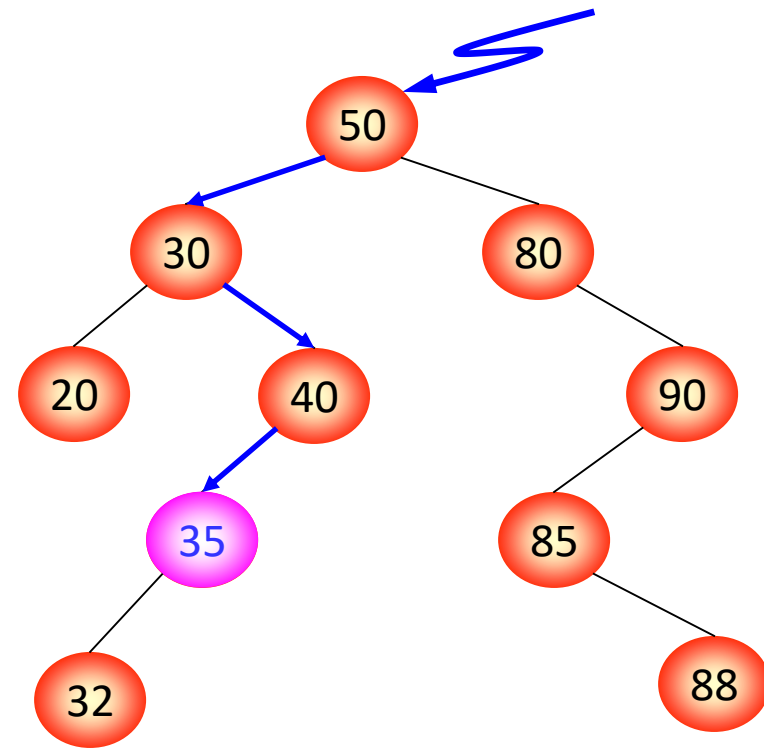
||

此结点所在层次数

最多的比较次数

||

树的深度



查找关键字：35

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

$n$ 个结点的二叉判断树是唯一的，因此，其平均查找长度也是定值。

最好情况：  $ASL = \log_2(n + 1) - 1$ ;

树的深度为： $\lfloor \log_2 n \rfloor + 1$ ;

与折半查找中的判定树相同。

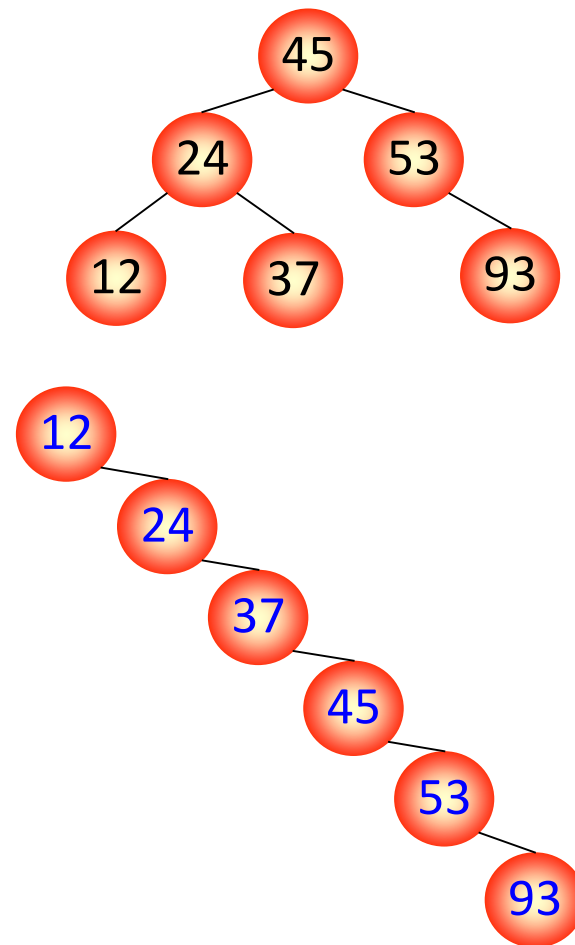
(形态比较均衡)。

最坏情况：插入的  $n$  个元素从一开始就有序，

—— 变成单支树的形态！

此时树的深度为  $n$ ;  $ASL = (n + 1) / 2$

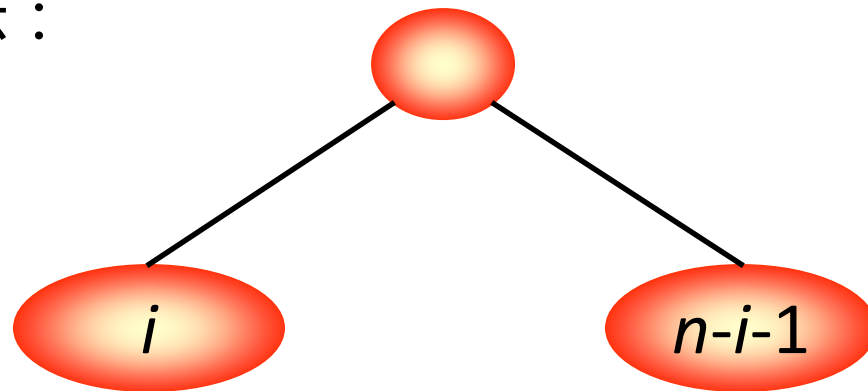
查找效率与顺序查找情况相同。





## 有 $n$ 个关键字的二叉排序树的平均查找长度

不失一般性，假设某个序列中有  $i$  个关键字小于第一个关键字，即有  $n-i-1$  个关键字大于第一个关键字，由它构造的二叉排序树如下所示：



其平均查找长度是  $n$  和  $i$  的函数： $P(n, i) (0 \leq i \leq n-1)$

$$P(n, i) = \frac{1}{n} [1 + i(p(i) + 1) + (n - i - 1)p(n - i - 1) + 1]$$

$$P(n) = \frac{1}{n} \sum_{i=0}^{n-1} p(n, i) = 1 + \frac{2}{n^2} \sum_{i=1}^{n-1} (i \times P(i))$$

## 有 $n$ 个关键字的二叉排序树的平均查找长度

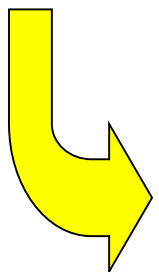
设每种树态出现概率相同，查找每个关键字也是等概率的，

则二叉排序树的  $ASL \leq 2(1 + \frac{1}{n}) \log n$

由此可见，在随机的情况下，二叉排序树的  $ASL$  和  $\log n$  是等数量级的。

**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

平衡二叉树又称 AVL 树，它是具有如下性质的二叉树：

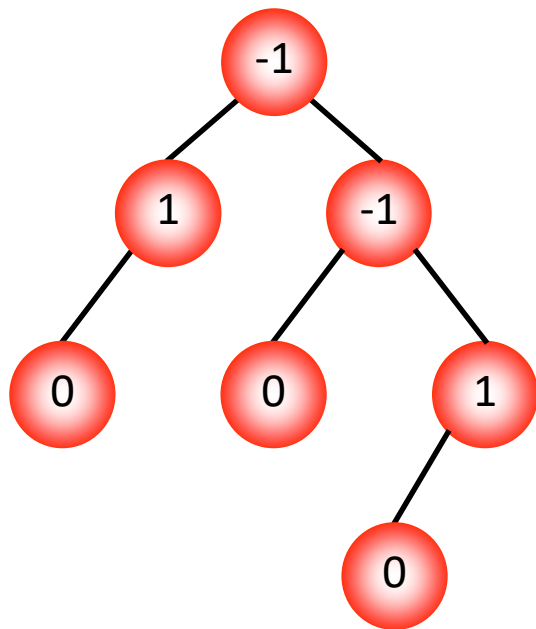
- 左、右子树是平衡二叉树；
- 所有结点的左、右子树深度之差的绝对值 $\leq 1$ 。

为了方便起见，给每个结点附加一个数字 = 该结点左子树与右子树的深度差。这个数字称为结点的平衡因子。这样，可以得到 AVL 树的其它性质（可以证明）：

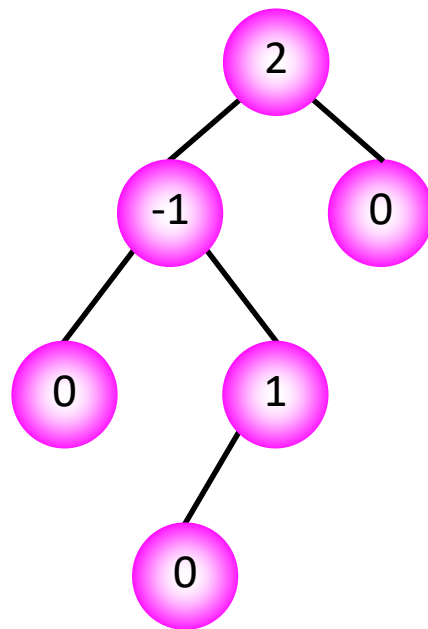
- 任一结点的平衡因子只能取： $-1$ 、 $0$  或  $1$ ；如果树中任意一个结点的平衡因子的绝对值大于  $1$ ，则这棵二叉树就失去平衡。
- 对于一棵有  $n$  个结点的 AVL 树，其深度和  $\log n$  同数量级，ASL 也和  $\log n$  同数量级。因此，在平衡二叉树上查找时间复杂度为  $O(\log n)$ 。

需要注意的是：我们以后讨论的平衡二叉树都是建立在二叉排序树基础之上的

例：判断下列二叉树是否 AVL 树？



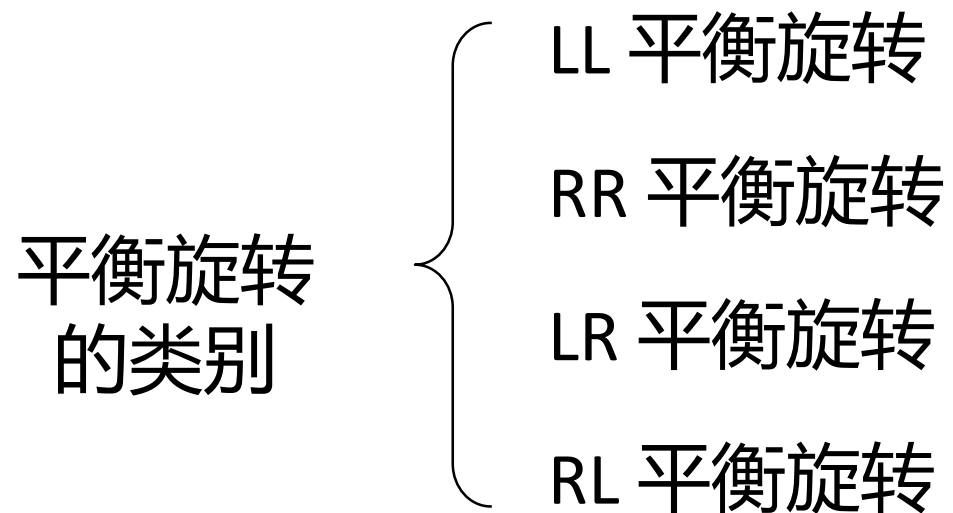
平衡二叉树



非平衡二叉树

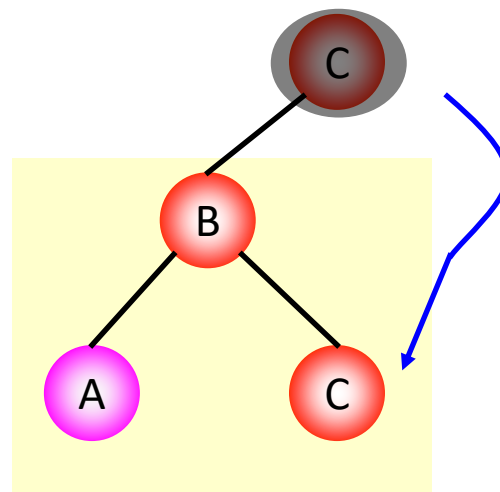
如果在一棵 AVL 树中插入一个新结点后造成失衡，则必须**重新调整树的结构**，使之恢复平衡。

我们称此调整平衡的过程为**平衡旋转**。



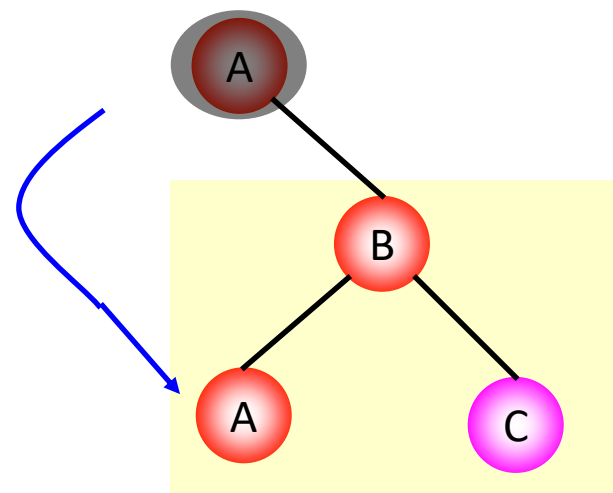
1) LL 平衡旋转:

若在 C 的左子树的左子树上插入结点，使 C 的平衡因子从 1 增加至 2，需要进行一次顺时针旋转。  
(以 B 为旋转轴)

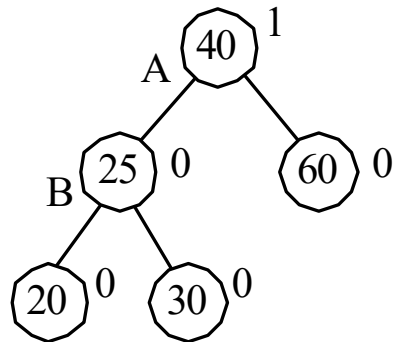


2) RR 平衡旋转:

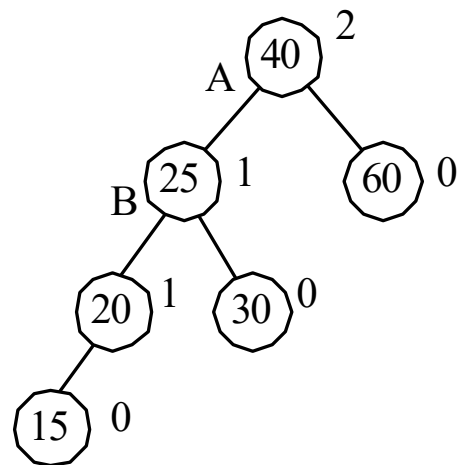
若在 A 的右子树的右子树上插入结点，使 A 的平衡因子从 -1 改变为 -2，需要进行一次逆时针旋转。  
(以 B 为旋转轴)



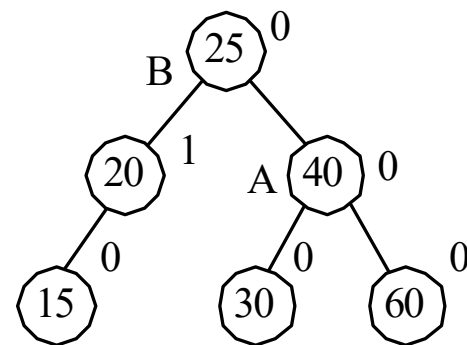
以B为轴，对A做了一次**单向右旋平衡旋转**。



(a) 一棵平衡二叉排序树

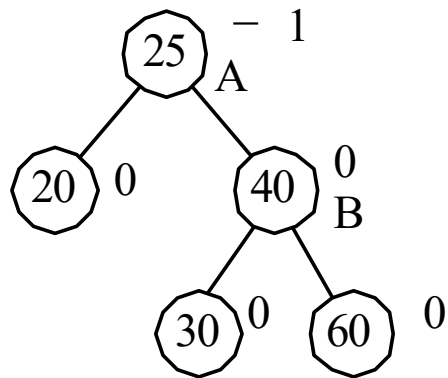


(b) 插入15后失去平衡

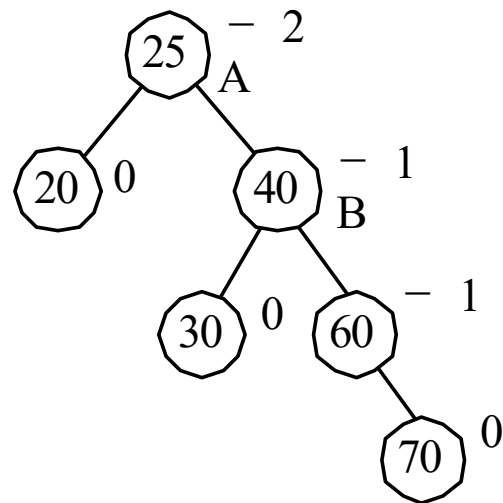


(c) 调整后的二叉排序树

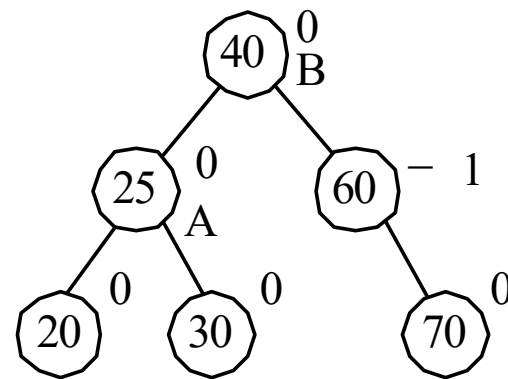
以B为轴，对A做了一次**单向左旋平衡旋转**。



(a) 一棵平衡二叉排序树



(b) 插入70后失去平衡

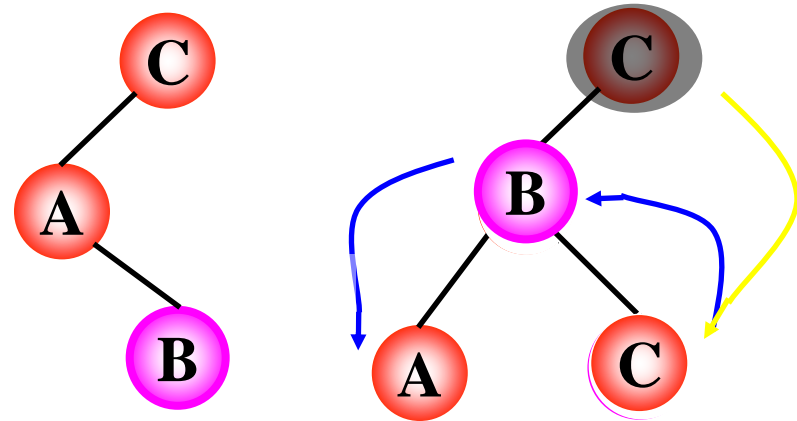


(c) 调整后的二叉排序树

### 3) LR 平衡旋转:

若在 C 的左子树的右子树上插入结点, 使 C 的平衡因子从 1 增加至 2, 需要先进行逆时针旋转, 再顺时针旋转。

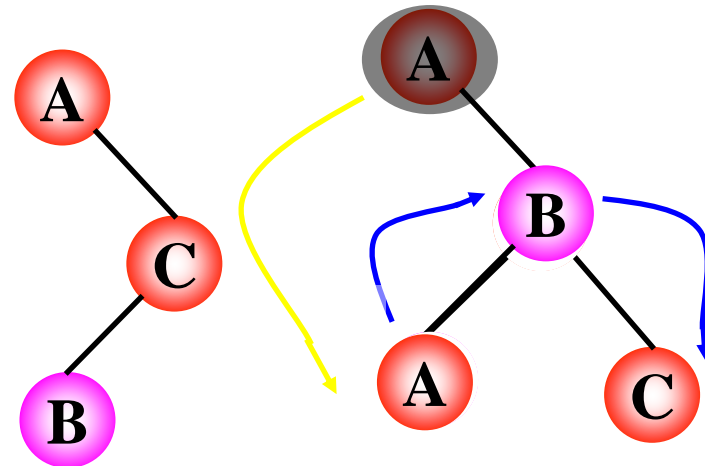
(以插入的结点 B 为旋转轴)



### 4) RL 平衡旋转:

若在 A 的右子树的左子树上插入结点, 使 A 的平衡因子从 -1 改变为 -2, 需要先进行顺时针旋转, 再逆时针旋转。

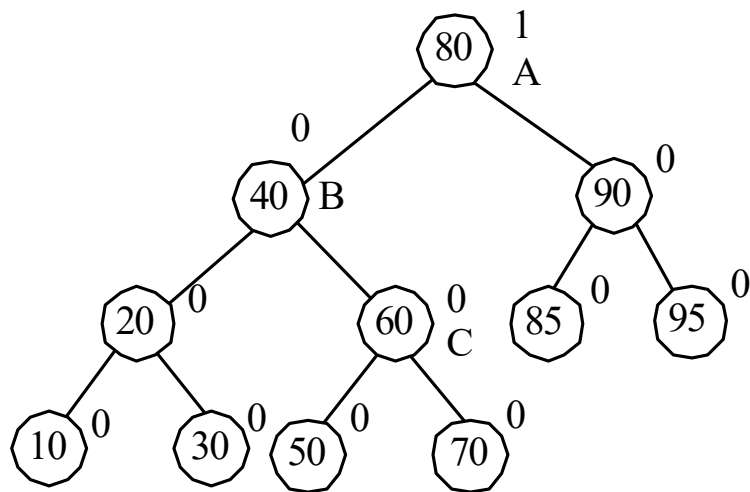
(以插入的结点 B 为旋转轴)



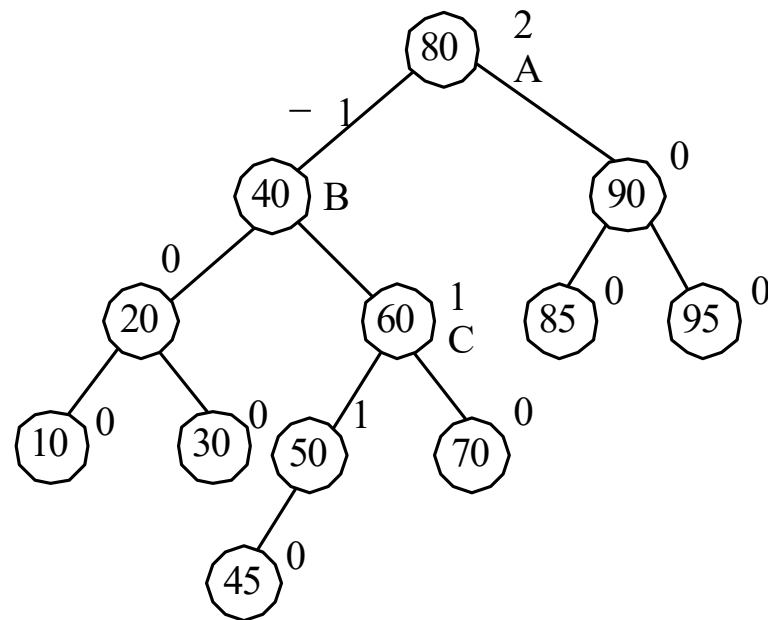
调整必须保证二叉排序树的特性不变



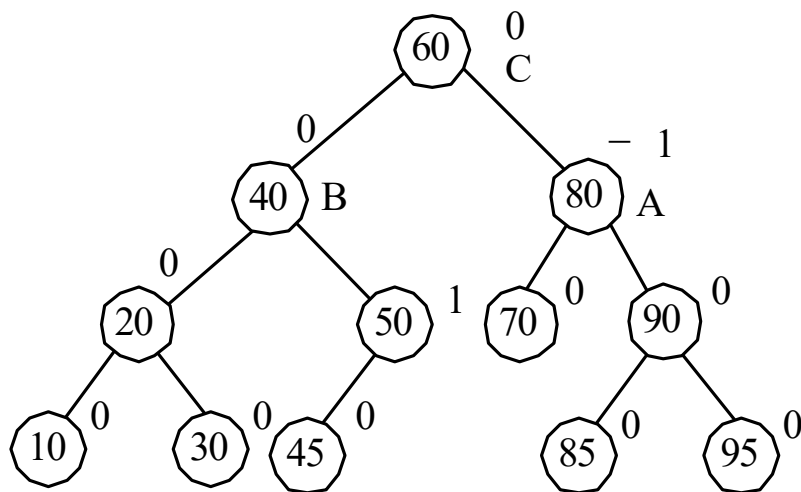
对**B**做了一次逆时针旋转，对**A**做了一次顺时针旋转。（先左后右）



(a) 一棵平衡二叉排序树

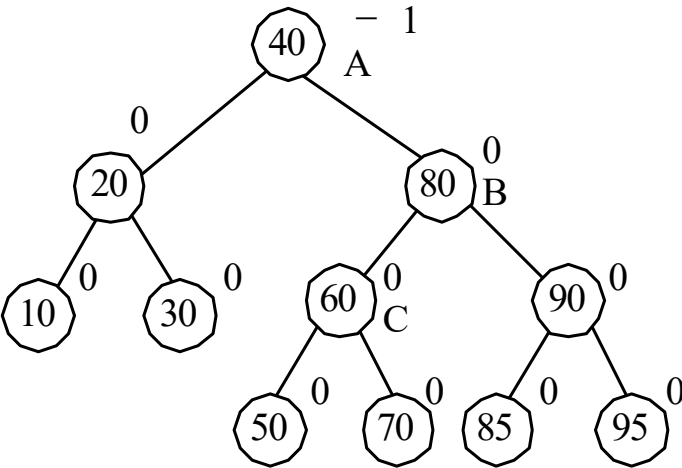


(b) 插入45后失去平衡

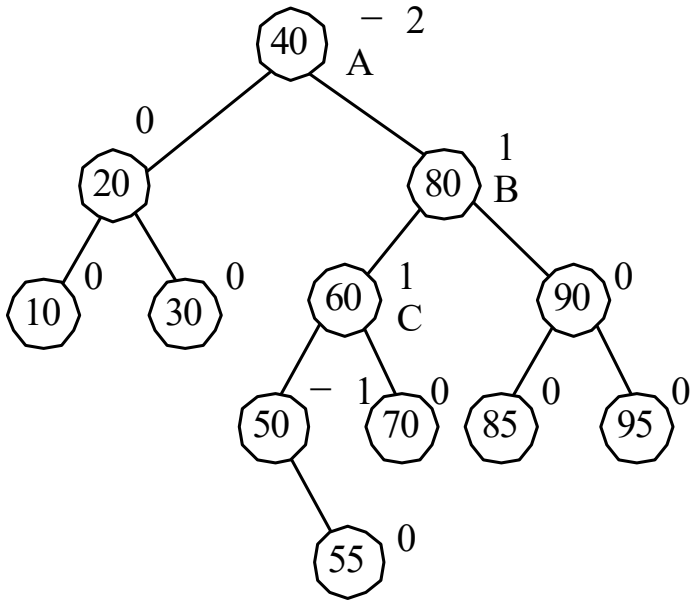


(c) 调整后的二叉排序树

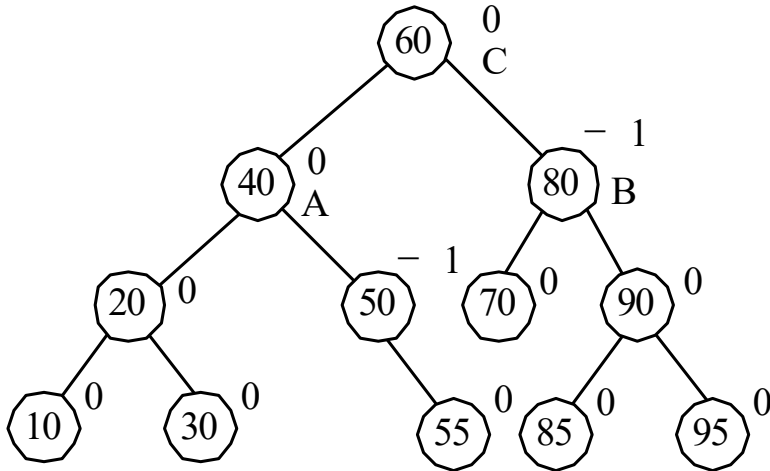
对B做了一次顺时针旋转，对A做了一次逆时针旋转。（先右后左）



(a) 一棵平衡二叉排序树

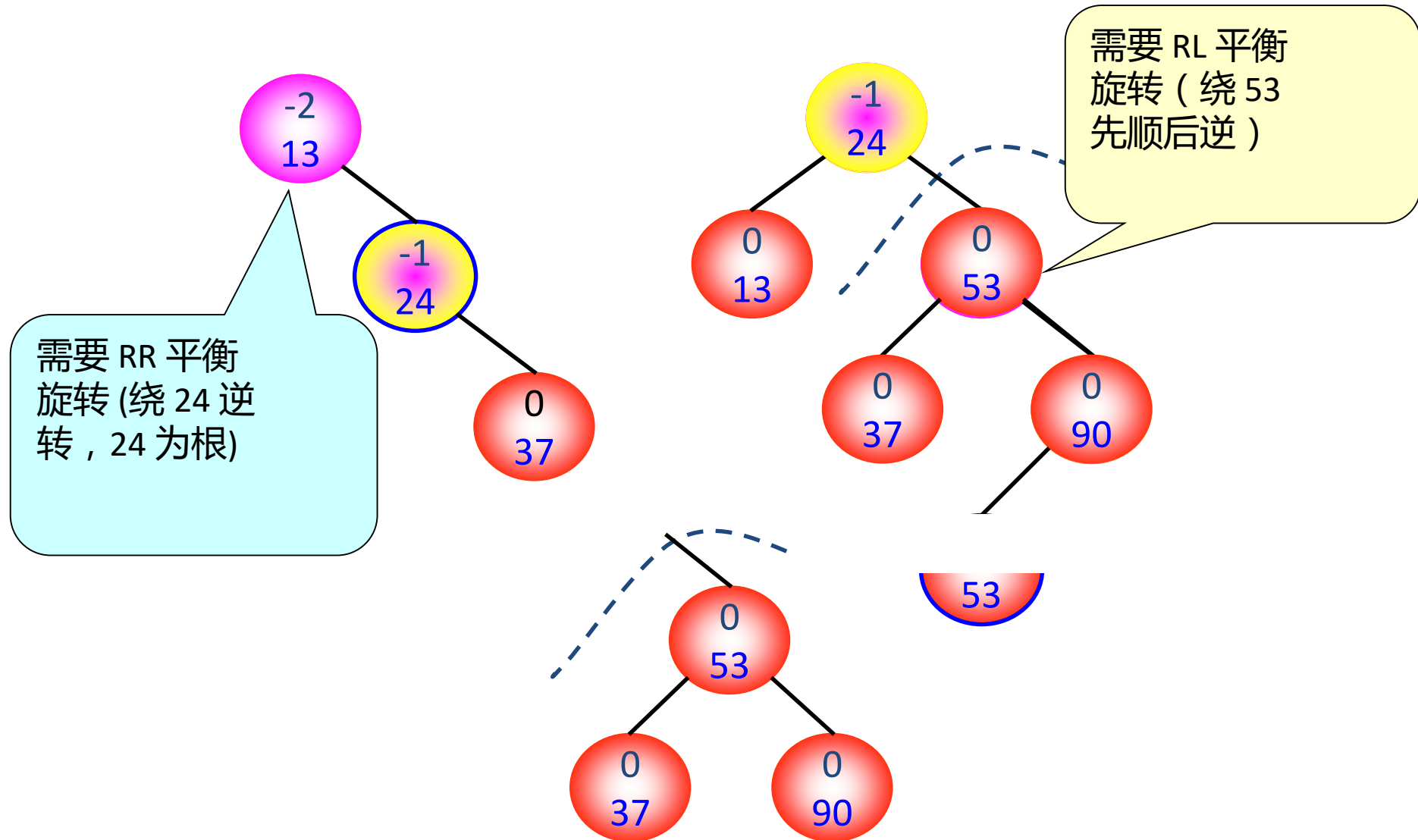


(b) 插入55后失去平衡



(c) 调整后的二叉排序树

例：请将下面序列构成一棵平衡二叉排序树：(13, 24, 37, 90, 53)

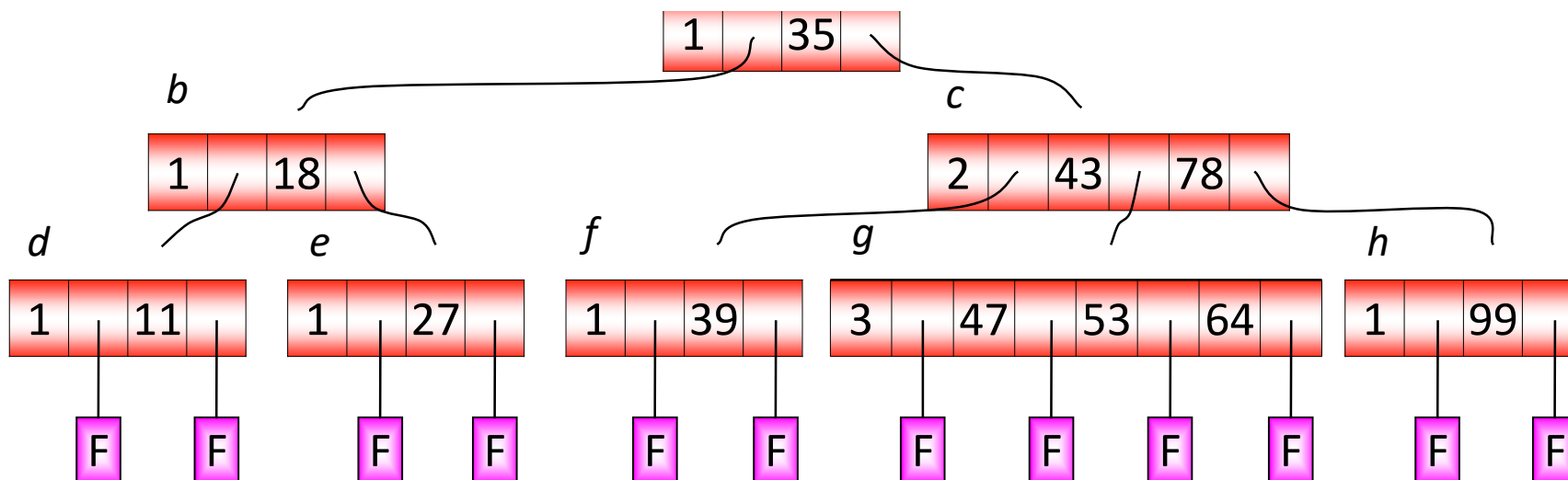


# B-树的定义

阶  $m$  可以事先任意指定，  
一旦指定后就固定不变。

一棵  $m$  阶的 B-树，或为空树或为满足下列特性的  $m$  叉树：

(5)、所有叶子结点在同一个层次上，且不含有任何信息。（可以看作是外部结点或查找失败的结点，实际上这些结点不存在，指向这些结点的指针为空）。



## B- 树特点

### 平衡

树中所有叶子结点均不带信息且在树的同一层次上；  
根结点或为叶子结点，或至少含有两棵子树；  
所有非叶子结点均含有  $n$  ( $\lceil m/2 \rceil \leq n \leq m$ ) 棵子树。

### 多路

在  $m$  阶的 B- 树上，每个非终端结点可能含有：  
 $n$  个关键字  $K_i$  ( $1 \leq i \leq n$ )  $n < m$   
 $n$  个指向记录的指针  $D_i$  ( $1 \leq i \leq n$ )  
 $n+1$  个指向子树的指针  $A_i$  ( $0 \leq i \leq n$ )

### 查找

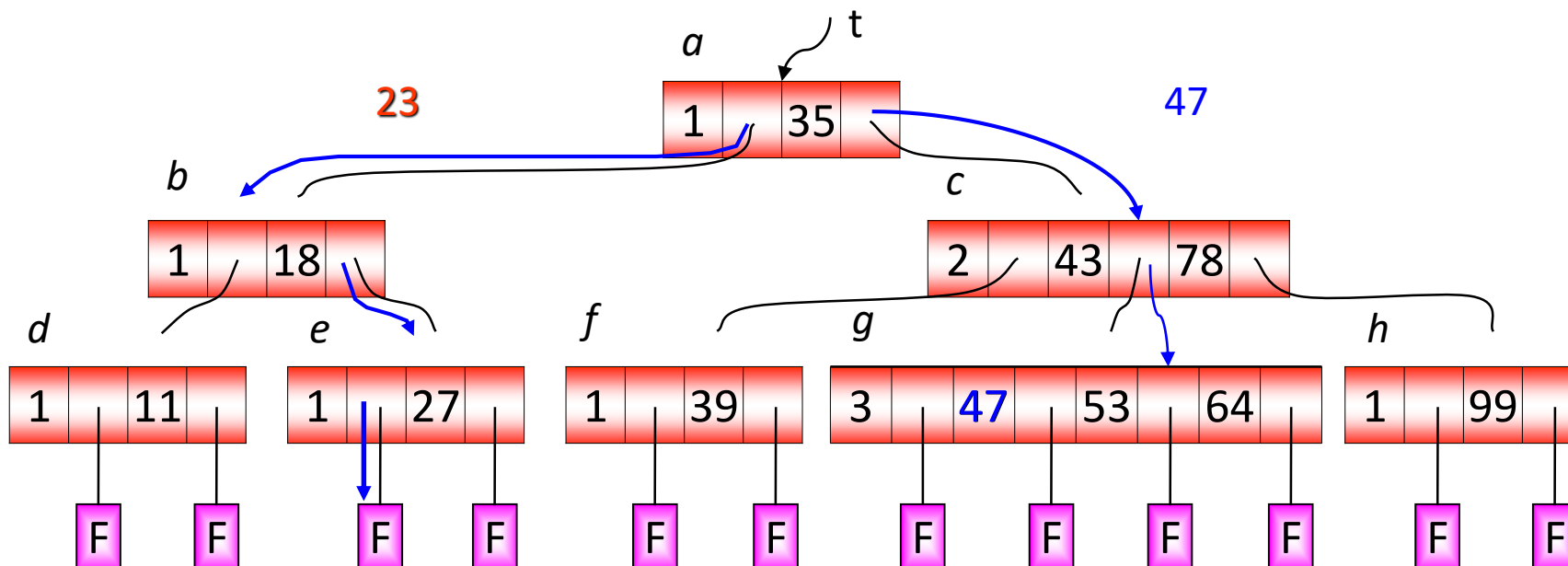
非叶子结点中的**多个关键字**均**自小至大**有序排列；  
 $A_{i-1}$ 所指子树上所有关键字均小于  $K_i$ ；  
 $A_i$ 所指子树上所有关键字均大于  $K_i$ 。

# B-树的查找

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**返回插入位置**。



## 性能分析

根据 B-树的定义，第一层至少有 1 个结点；第二层至少有 2 个结点；由于除根之外的每个非终端结点至少有  $\lceil m/2 \rceil$  棵子树，则第三层只有  $2 (\lceil m/2 \rceil)$  个结点；……；依次类推，第  $l+1$  层至少有  $2 (\lceil m/2 \rceil)^{l-1}$  个结点，而  $l+1$  层的结点为叶子结点。若  $m$  阶 B-树中具有  $n$  个关键字，则叶子结点即查找不成功的结点为  $n+1$ ，所以有：✧

$$n+1 \geq 2 * (\lceil m/2 \rceil)^{l-1}, \text{ 得出 } l \leq \log_{\lceil m/2 \rceil} \left( \frac{n+1}{2} \right) + 1, \text{ ✧}$$

也就是说，在含有  $n$  个关键字的 B-树上进行查找时，从根结点到关键字所在结点的路径上涉及的结点数不超过  $\log_{\lceil m/2 \rceil} \left( \frac{n+1}{2} \right) + 1$ 。✧

## B-的插入

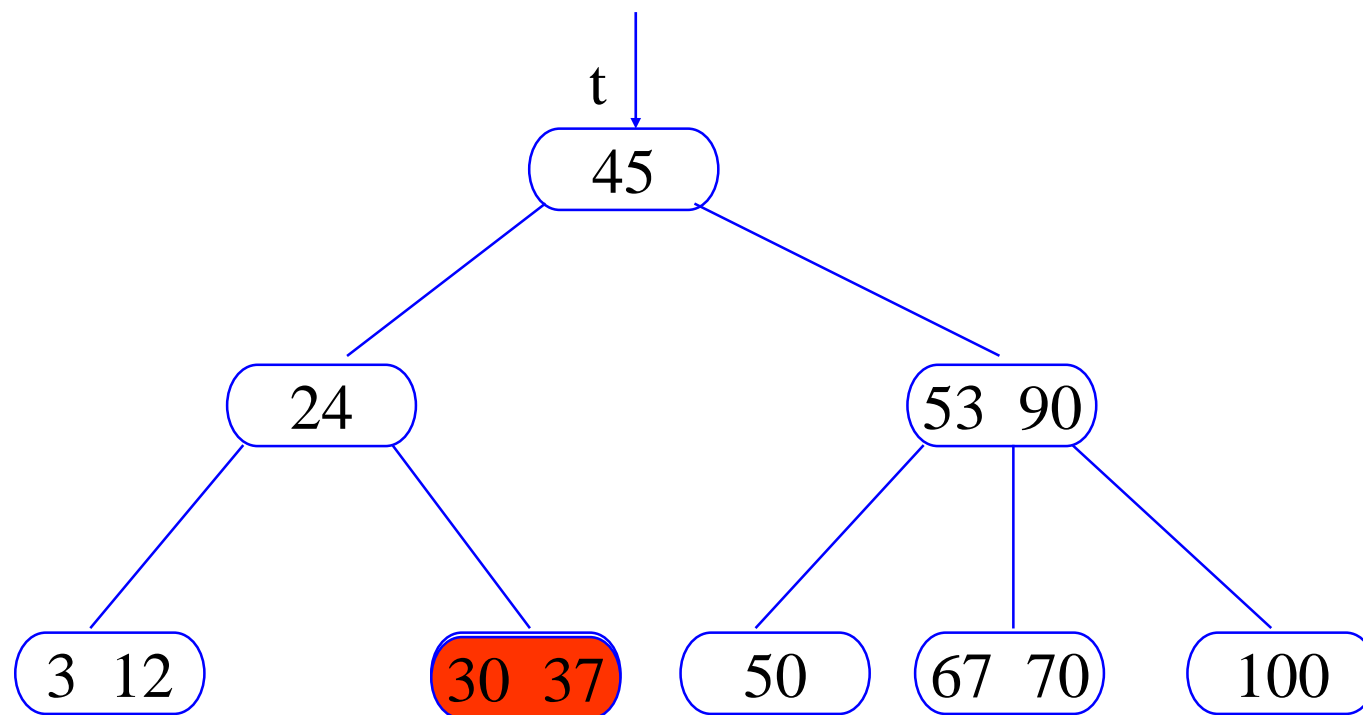
- 在B-树上插入关键字与在二叉排序树上插入结点不同，关键字的插入不是在叶结点上进行的，而是首先在最底层的某个非终端结点中添加一个关键字，若该结点的关键字个数不超过 $m-1$ ，则插入完成；否则，若该结点的关键字个数已达到 $m$ 个，这与B-树定义不符，将引起结点的“分裂”。



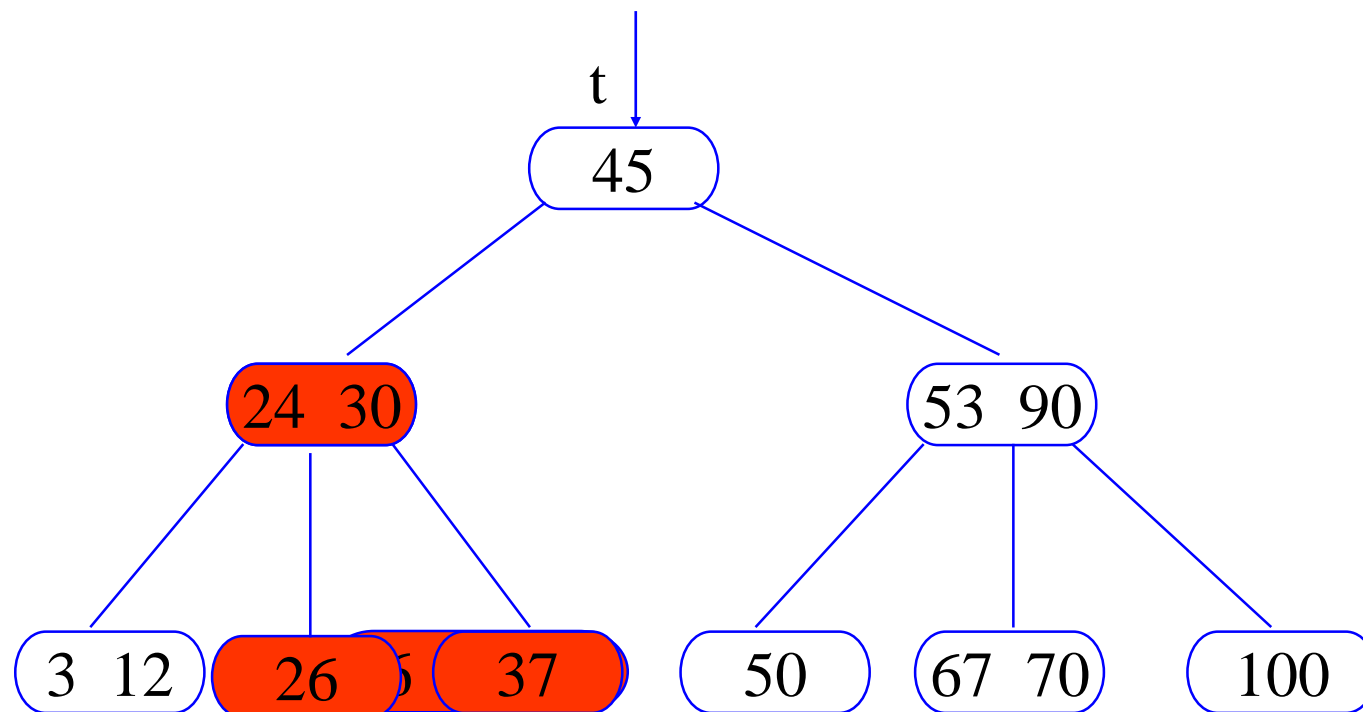


- 分裂方法为：将结点中的关键字分成三部分，使得前后两部分的关键字个数均大于等于 $\lceil m/2 \rceil - 1$ ，而中间部分只有一个关键字。前后两部分成为两个结点，而中间部分的关键字将插入到父结点中。若插入父结点而使父结点中关键字个数超过 $m-1$ ，则父结点继续分裂，直到插入某个父结点，其关键字个数小于 $m$ 。
- 例如：在下面的3阶的B-树上依次插入结点30,26,85和7的插入过程。

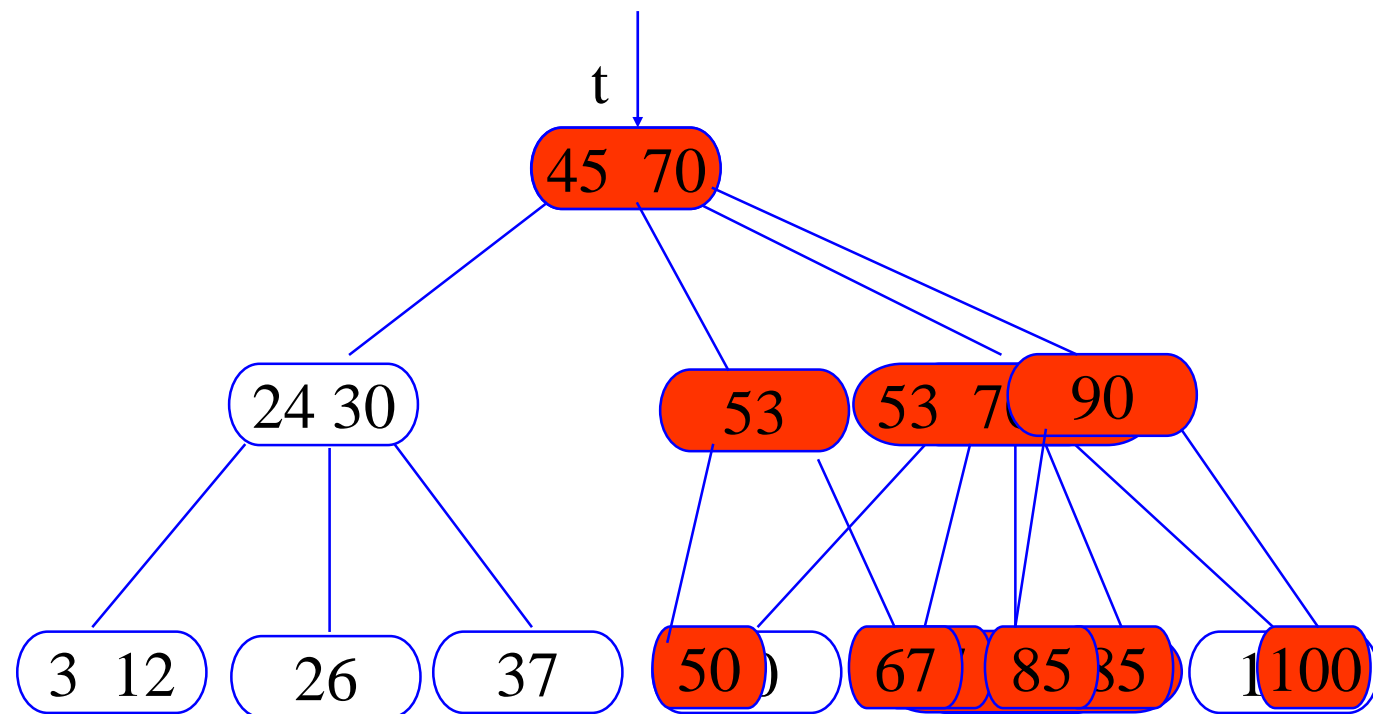
插入30



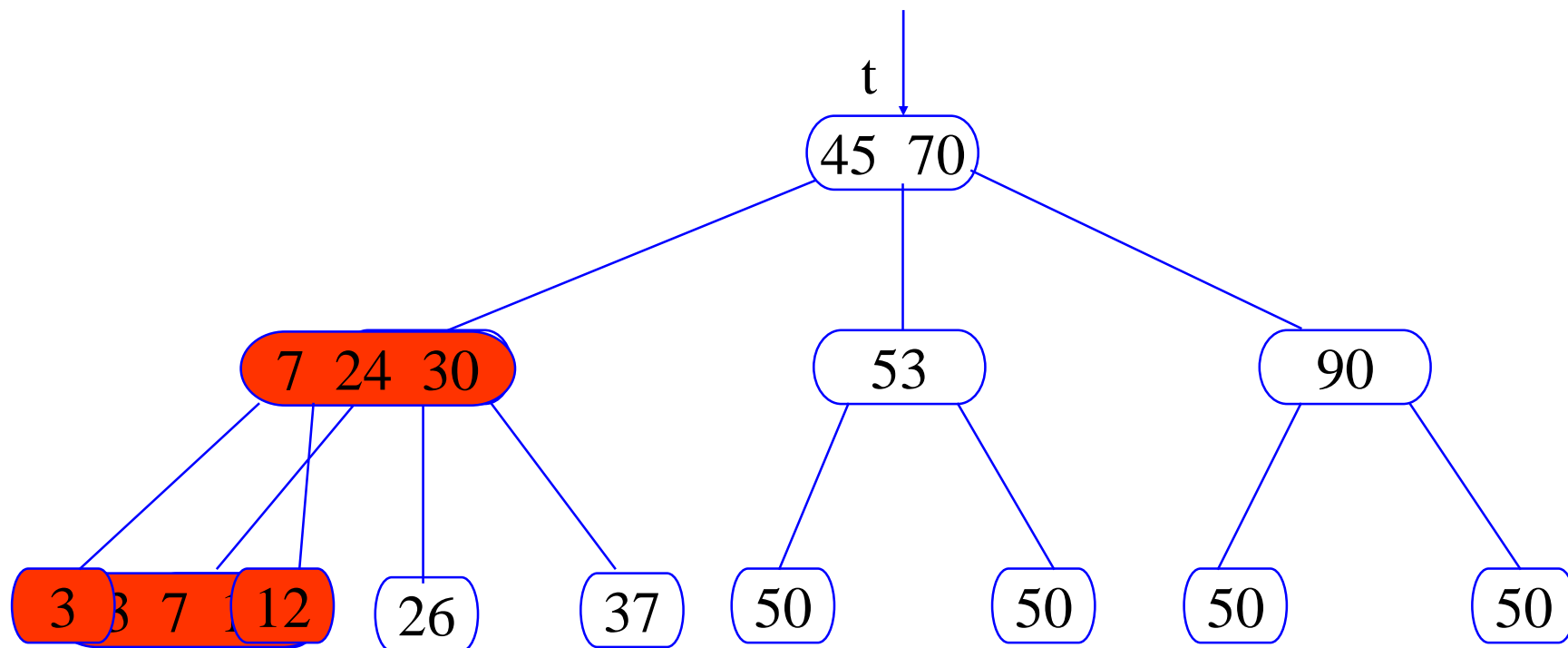
## 插入26



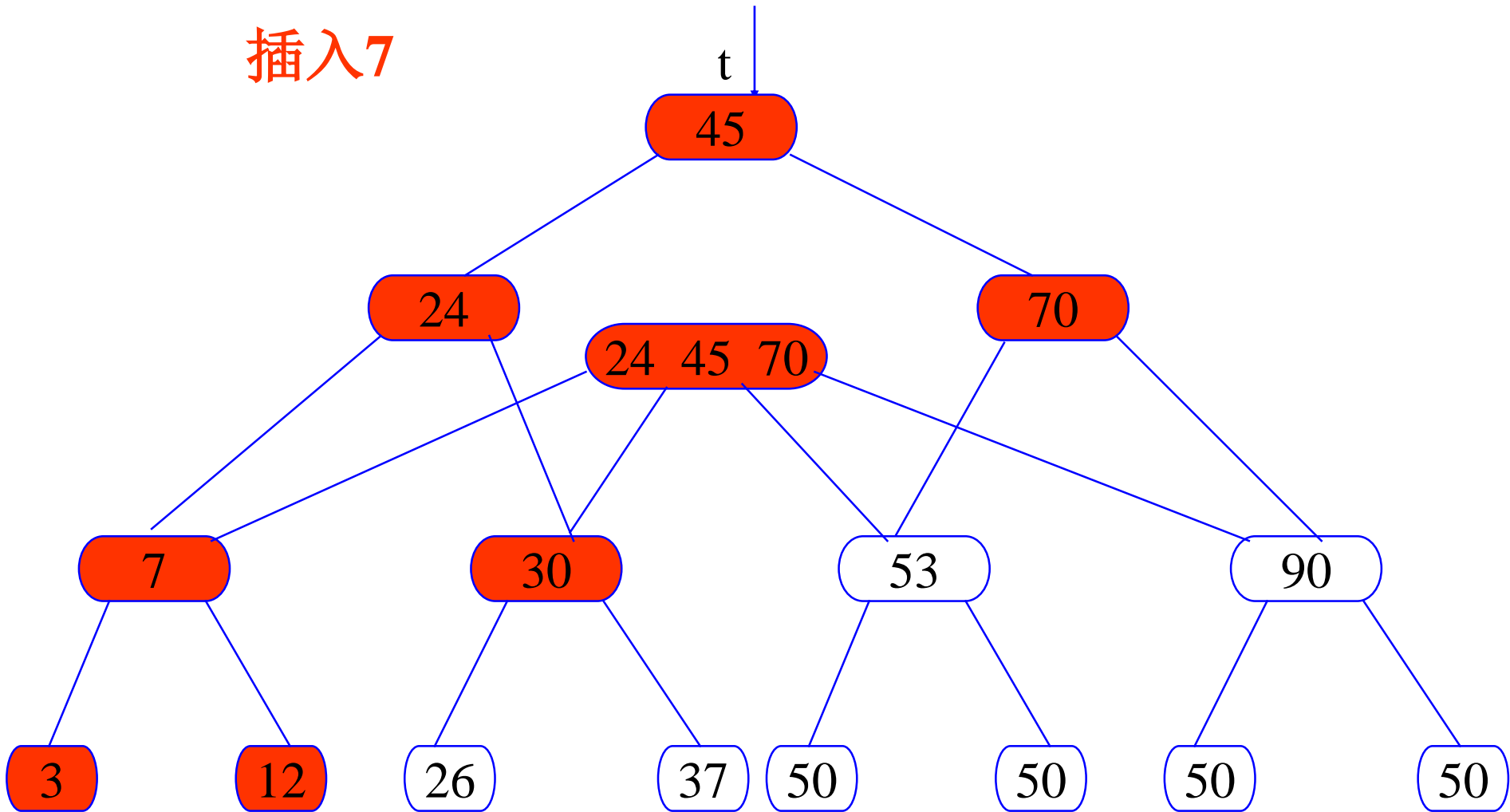
插入85



插入7



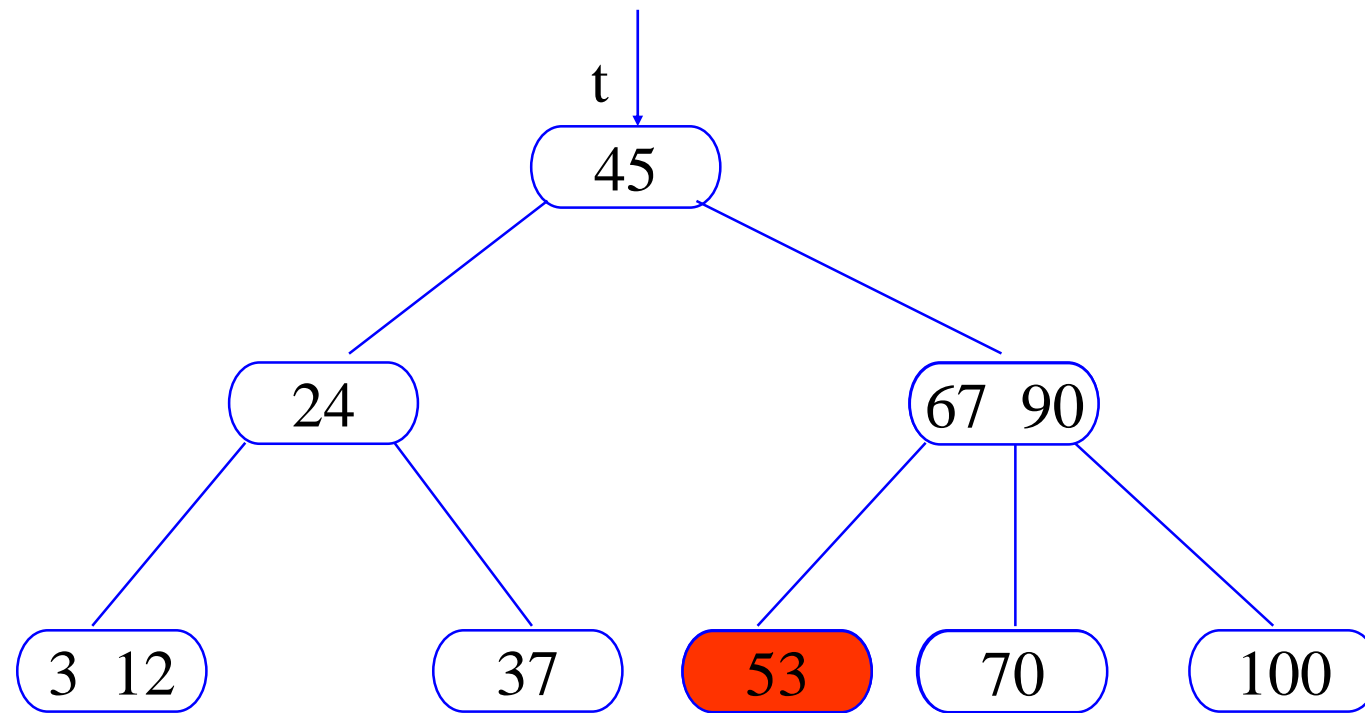
插入7



## B-树的删除

- 删除最底层的某个非终端结点中的关键字
- 如果被删关键字所在的节点中的关键字数目不小于  $\lceil m/2 \rceil$ ，则只须从该结点中删除该关键字，删除完成，否则要进行“合并”结点的操作。可以按照下列两种情况进行处理：
  - ①若右兄弟（或左兄弟）结点中的关键字数目大于  $\lceil m/2 \rceil - 1$ ，则需将兄弟结点中的最小（或最大）的关键字上移至双亲结点中，而将双亲结点中小于（或大于）且紧靠该上移关键字的关键字下移至被删关键字所在结点。例如：

## 删除关键字50



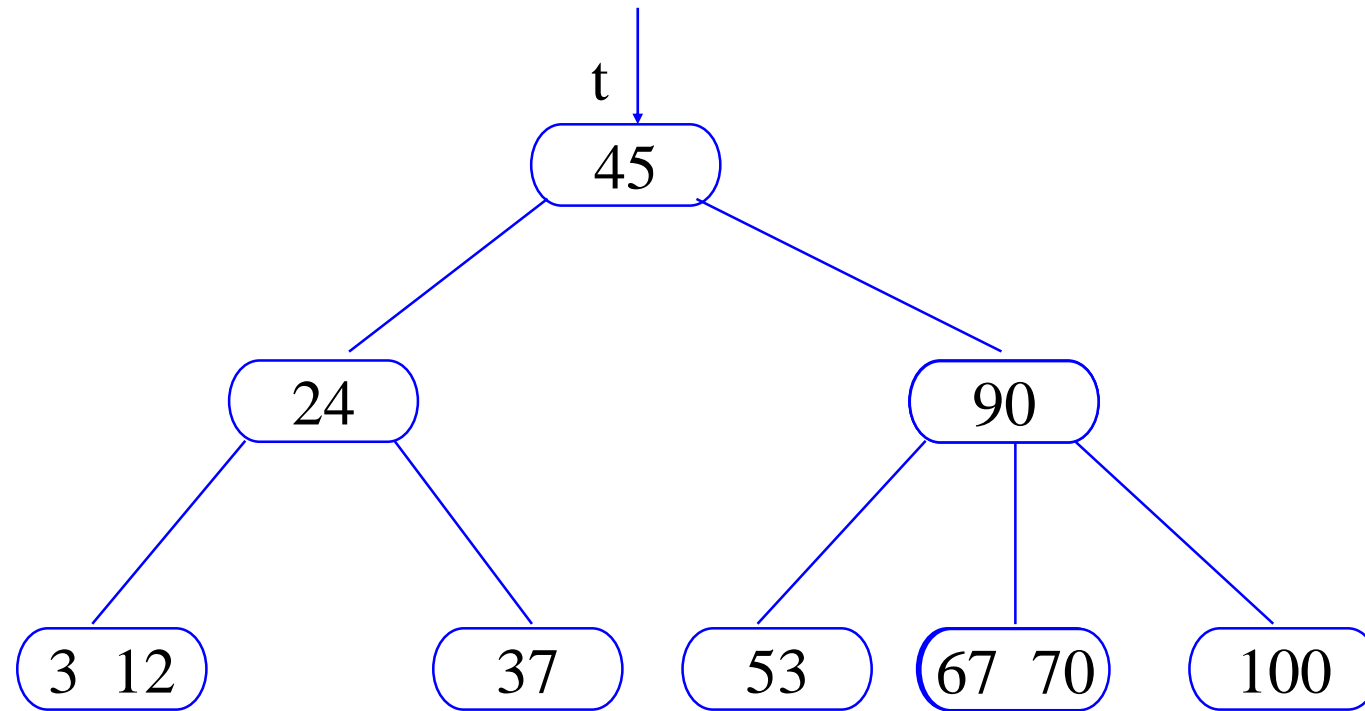
一棵3阶的B-树





②若与该结点相邻的兄弟结点中关键字数目等于  $\lceil m/2 \rceil - 1$ 。假设该结点有右兄弟，且其右兄弟结点的地址由双亲结点中的指针  $A_i$  所指，则在删除关键字之后，他所在结点中剩余的关键字和指针，加上双亲结点中的关键字  $k_i$  一起合并到  $A_i$  所指的兄弟结点中（若没有右兄弟，则合并到左兄弟中）。例如：

## 删除关键字53

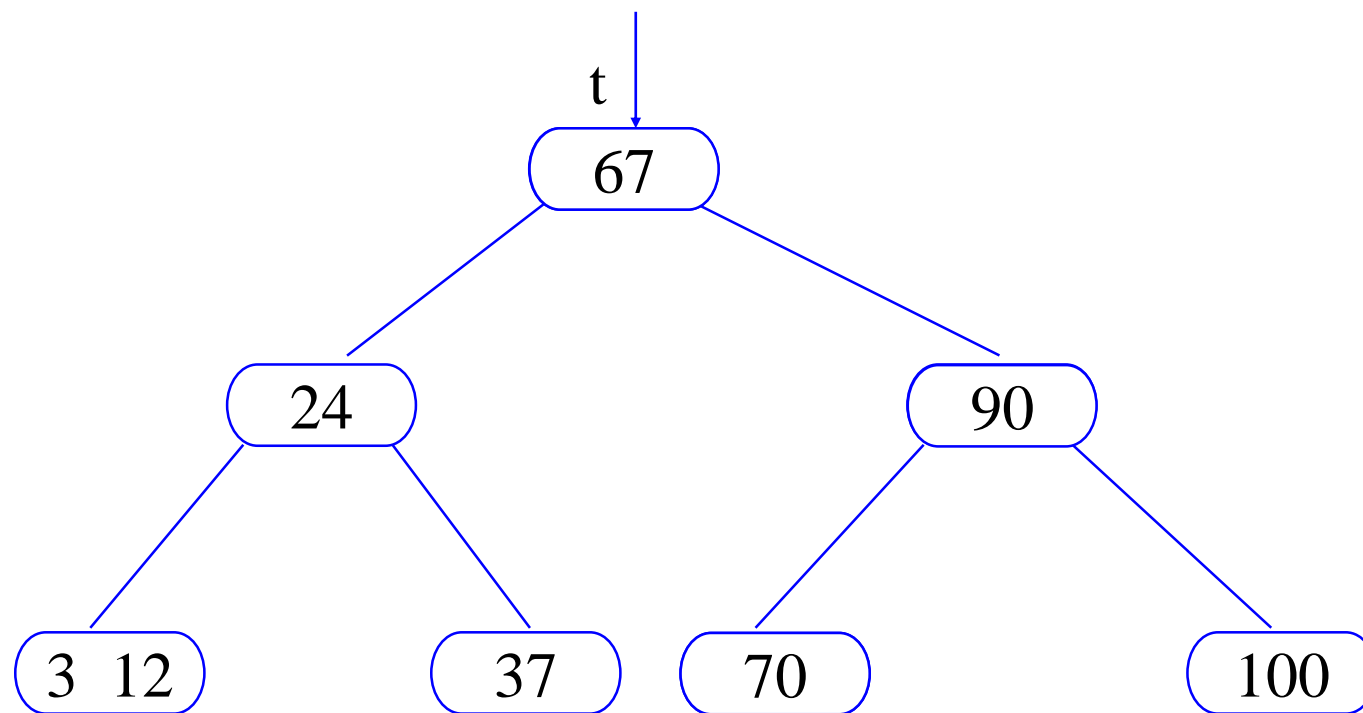


一棵3阶的B-树



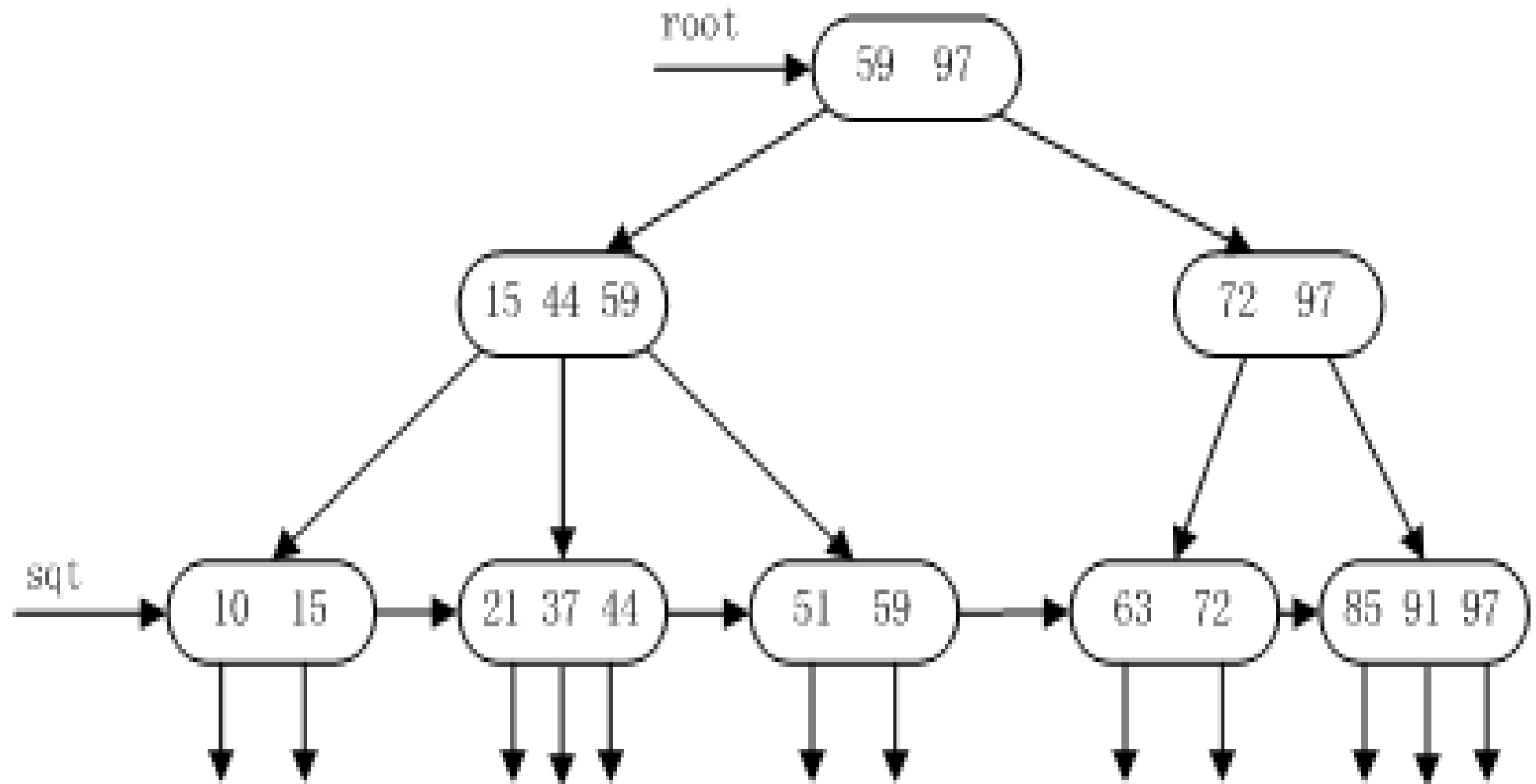
- 删除非最底层结点的关键字
- 若删除非底层结点中的关键字 $K_i$ ，则可以指针 $A_i$ 所指子树中的最小关键字 $x$ 替代 $K_i$ ，直到这个 $x$ 在最底层结点上，然后，再删除关键字 $x$ ，即转为第一种情形。例如：

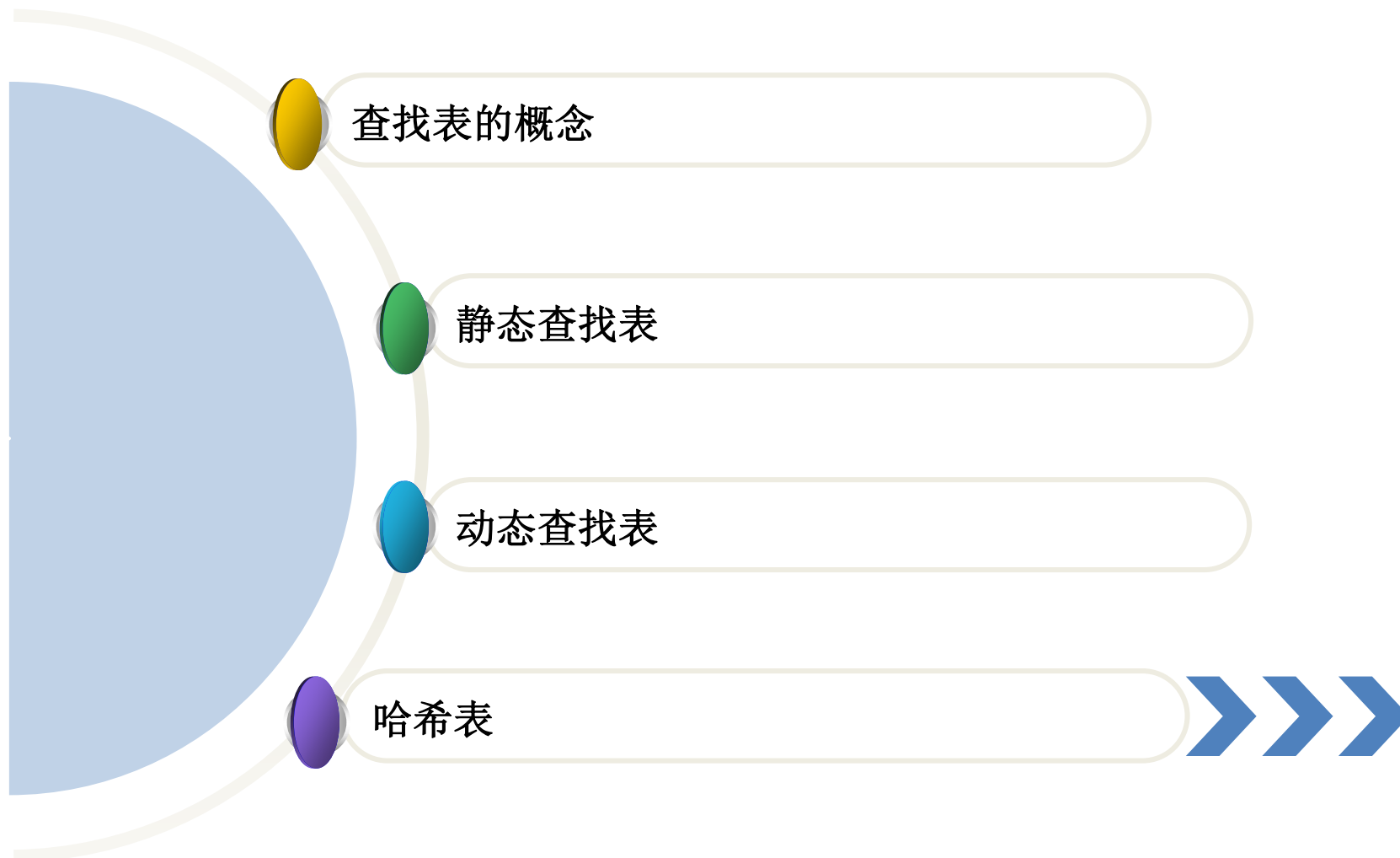
## 删除关键字45



一棵3阶的B-树

# B<sup>+</sup> 树





# 哈希表

以上讨论的表示查找表的各种结构的共同特点：记录在表中的**位置**和它的**关键字**之间不存在一个确定的关系。

**查找过程**：给定值依次和关键字集合中各关键字进行**比较**。

不同的表示方法和查找策略，其差别在于：1)、关键字和给定值进行比较的顺序(过程)不同。2)、比较的结果不同：顺序查找有两种可能——“=”与“≠”；其他查找有三种可能——“<”、“=”、“>”。

**查找的效率**取决于和给定值进行比较的关键字个数。

用这类方法表示的查找表，**其平均查找长度都不为零**。

对于频繁使用的查找表，希望  $ASL = 0$ 。

只有一个办法：预先知道所查关键字在表中的位置。

即，要求：记录在表中的位置和其关键字之间存在一种确定的关系。

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  $xx000 \sim xx999$ （前两位为年份）。

若以下标为  $000 \sim 999$  的有序顺序表表示之，则查找过程可以简单进行：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。

上例表明，记录的**关键字**与记录在表中的**存储位置**之间存在一种对应（函数）关系。若记录的关键字为  $key$ ，记录在表中的位置（称为**哈希地址**）为  $f(key)$ ，则称此函数  $f(x)$  为**哈希函数（散列函数）**。



例如：对于如下 9 个关键字：

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

设哈希函数  $f(\text{key}) = \lfloor (\text{Ord}(\text{关键字首字母}) - \text{Ord}('A') + 1) / 2 \rfloor$

0      1      2      3      4      5      6      7      8      9      10      11      12      13

|  |      |     |  |     |  |    |  |      |     |  |    |    |      |
|--|------|-----|--|-----|--|----|--|------|-----|--|----|----|------|
|  | Chen | Dai |  | Han |  | Li |  | Qian | Sun |  | Wu | Ye | Zhao |
|--|------|-----|--|-----|--|----|--|------|-----|--|----|----|------|

**问题：**若添加关键字 Zhou，会出现什么情况？

Zhou

从这个例子可见：

1) 哈希函数是一个映像，即：将关键字的集合映射到某个地址集合上。它的设置很灵活，只要使得关键字的哈希函数值都落在表长允许的范围之内即可；

2) 由于哈希函数是一个映像，因此，在一般情况下，很容易产生“冲突”现象，即： $\text{key1} \neq \text{key2}$ ，而  $f(\text{key1}) = f(\text{key2})$ 。

这种具有相同函数值的关键字称为同义词。

3) 很难找到一个不产生冲突的哈希函数。一般情况下，只能选择恰当的哈希函数，使冲突尽可能少地产生。

因此：在构造这种特殊的“查找表”时，除了需要选择一个“好”（其原则是尽可能地使任意一组关键字的哈希地址均匀地分布在整个地址空间中，即用任意关键字作为哈希函数的自变量其计算结果随机分布，以尽可能少产生冲突）的哈希函数之外；还需要找到一种“处理冲突”的方法。

# 哈希表的定义

根据设定的哈希函数 $H(\text{key})$ 和所选中的处理冲突的方法建立的**查找表**。其基本思想是：以记录的关键字为自变量，根据哈希函数，计算出对应的哈希地址，并在此存储该记录的内容。

这一映像过程称为**哈希造表**或**散列**。

当对记录进行查找时，再根据给定的关键字，用同一个哈希函数计算出给定关键字对应的存储地址（通常还会进行确认比较），随后进行访问。所以哈希表既是一种存储形式，又是一种查找方法，通常将这种查找方法称为**哈希查找**。

# 哈希函数的构造方法

对数字关键字可有下列构造方法：

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

若是非数字关键字，则需先对其进行数字化处理。

## 1. 直接定址法

哈希函数为关键字的线性函数

$$H(\text{key}) = \text{key} \quad \text{或者} \quad H(\text{key}) = a \times \text{key} + b$$

特点：地址集合的大小 = 关键字集合的大小。

不同的关键字（调整  $a$  与  $b$  的值）不发生冲突。

实际中能使用这种哈希函数的情况很少。

## 2. 数字分析法（数字选择法）

构造：取关键字的若干位或其组合作哈希地址。

适于关键字位数比哈希地址位数大，且可能出现的关键字事先知道的情况。

例：有 80 个记录，关键字为 8 位十进制数，哈希表长为 100。  
则哈希地址可取 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 |

分析：①只取 8

②只取 1

③只取 3、4

⑧只取 2、7、5

④⑤⑥⑦ 数字分布近乎随机

所以：取 ④⑤⑥⑦ 任意两位或两位与另两位的叠加作哈希地址

### 3. 平方取中法（较常用）

构造：以关键字的平方值的中间几位作为哈希地址。

求“关键字的平方值”的目的是“扩大差别”，同时平方值的中间各位又能受到整个关键字中各位的影响。

此方法适合于：关键字中的每一位都有某些数字**重复出现频率很高的**现象。

### 4. 折叠法

构造：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。

**间界叠加**：从一端沿分割界来回折叠，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。

例：关键字为：0442205864，哈希地址位数为 4。

$$\begin{array}{r} 5864 \\ 4220 \\ 04 \\ \hline 10088 \\ H(\text{key})=0088 \end{array}$$

移位叠加

$$\begin{array}{r} 5864 \\ 0224 \\ 04 \\ \hline 6092 \\ H(\text{key})=6092 \end{array}$$

间界叠加

## 5. 除留余数法（最常用）

构造：取关键字被某个不大于哈希表表长  $m$  的数  $p$  除后所得余数作哈希地址，即  $H(\text{key}) = \text{key} \bmod p$ ， $p \leq m$ 。

**特点：**简单，可与上述几种方法结合使用。  
 $p$  的选取很重要； $p$  选得不好，容易产生同义词。

$p$  应为不大于  $m$  的素数或不含 20 以下的质因子的合数。

为什么要对  $p$  加限制？

例如：

给定一组关键字：12, 39, 18, 24, 33, 21，若取  $p = 9$ ，则他们对应的哈希函数值将为：3, 3, 0, 6, 6, 3

可见，若  $p$  中含质因子 3，则所有含质因子 3 的关键字均映射到“3 的倍数”的地址上，从而增加了“冲突”的可能。

## 6. 随机数法

构造：取关键字的随机函数值作哈希地址，即：

$$H(\text{key}) = \text{Random}(\text{key})$$

其中，Random 为伪随机函数。

适于关键字长度不等的情况。



实际造表时，采用何种构造哈希函数的方法取决于建表的关键字集合的情况（包括关键字的范围和形态），总的**原则**是使产生**冲突**的可能性降到尽可能地**小**。

选取哈希函数，考虑以下因素：

- 1) 计算哈希函数所需时间
- 2) 关键字长度
- 3) 哈希表长度（哈希地址范围）
- 4) 关键字分布情况
- 5) 记录的查找频率

# 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另一个哈希地址。

## 1. 开放定址法

当发生冲突时，在冲突位置的前后附近寻找可以存放记录的空闲单元。用此法解决冲突，要产生一个探测序列，沿着此序列去寻找可以存放记录的空闲单元。最简单的探测序列产生方法是进行线性探测，即当发生冲突时，从发生冲突的存储位置的下一个存储位置开始依次顺序探测空闲单元。

为产生冲突的地址  $H(\text{key})$  求得一个探查地址序列：

$$H_0, H_1, H_2, \dots, H_s \quad 1 \leq s \leq m - 1$$

$m$  为哈希表的表长

其中： $H_i = (H(\text{key}) + d_i) \text{ MOD } m \quad i = 1, 2, \dots, s$

沿此序列逐个地址探查，直到找到一个空位置（开放的地址），  
将发生冲突的记录放到该地址中。

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

$$f(\text{key}) = \lfloor (\text{Ord}(\text{关键字首字母}) - \text{Ord}('A') + 1) / 2 \rfloor$$

对增量  $d_i$  有三种取法:

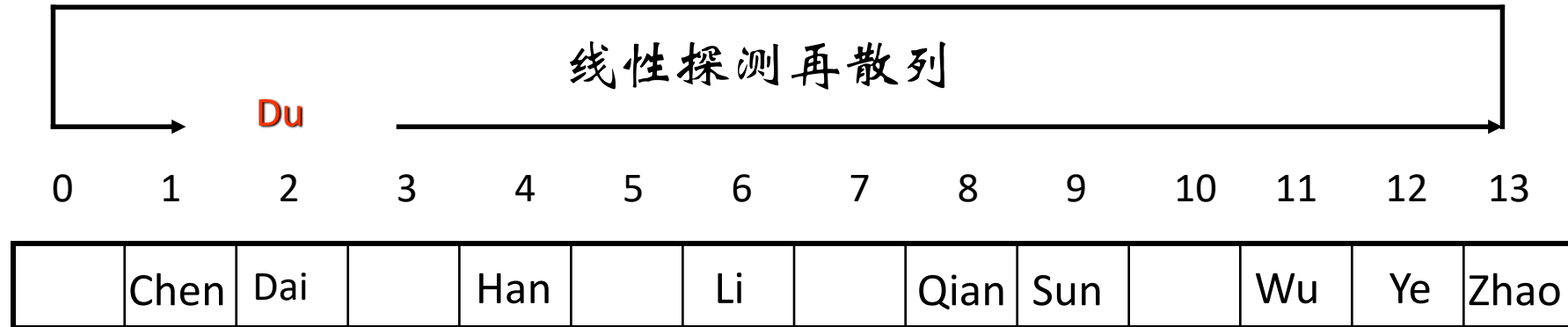
➤ 线性探测再散列:  $d_i = 1, 2, 3, \dots, m-1$

➤ 二次探测再散列 (平方探测再散列) :

$$d_i = 1^2, -1^2, 2^2, -2^2, 3^2, \dots, \pm k^2 \ (k \leq m/2)$$

➤ 伪随机探测再散列 (双散列函数探测再散列) :

$$d_i = \text{伪随机数序列}$$



例：表长为 11 的哈希表中已填有关键字为 17，60，29 的记录，  
 $H(\text{key}) = \text{key} \bmod 11$ ，现有第 4 个记录，其关键字为 38，  
按三种处理冲突的方法，将它填入表中。

| 0 | 1 | 2 | 3  | 4  | 5  | 6  | 7  | 8  | 9 | 10 |
|---|---|---|----|----|----|----|----|----|---|----|
|   |   |   | 38 | 38 | 60 | 17 | 29 | 38 |   |    |

(1)  $H(38) = 38 \bmod 11 = 5$  冲突

$H_1 = (5+1) \bmod 11 = 6$  冲突

$H_2 = (5+2) \bmod 11 = 7$  冲突

$H_3 = (5+3) \bmod 11 = 8$  不冲突

(2)  $H(38) = 38 \bmod 11 = 5$  冲突

$H_1 = (5+1^2) \bmod 11 = 6$  冲突

$H_2 = (5-1^2) \bmod 11 = 4$  不冲突

(3)  $H(38) = 38 \bmod 11 = 5$  冲突

设伪随机数序列为 9，则：

$H_1 = (5+9) \bmod 11 = 3$  不冲突

## 2. 再哈希法

方法：构造若干个哈希函数，当发生冲突时，计算另一个哈希地址，即： $H_i = RH_i(\text{key}) \quad i = 1, 2, \dots, k$   
其中： $RH_i$  —— 不同的哈希函数。

特点：不易产生“聚集”，但计算时间增加。

## 3. 溢出区法

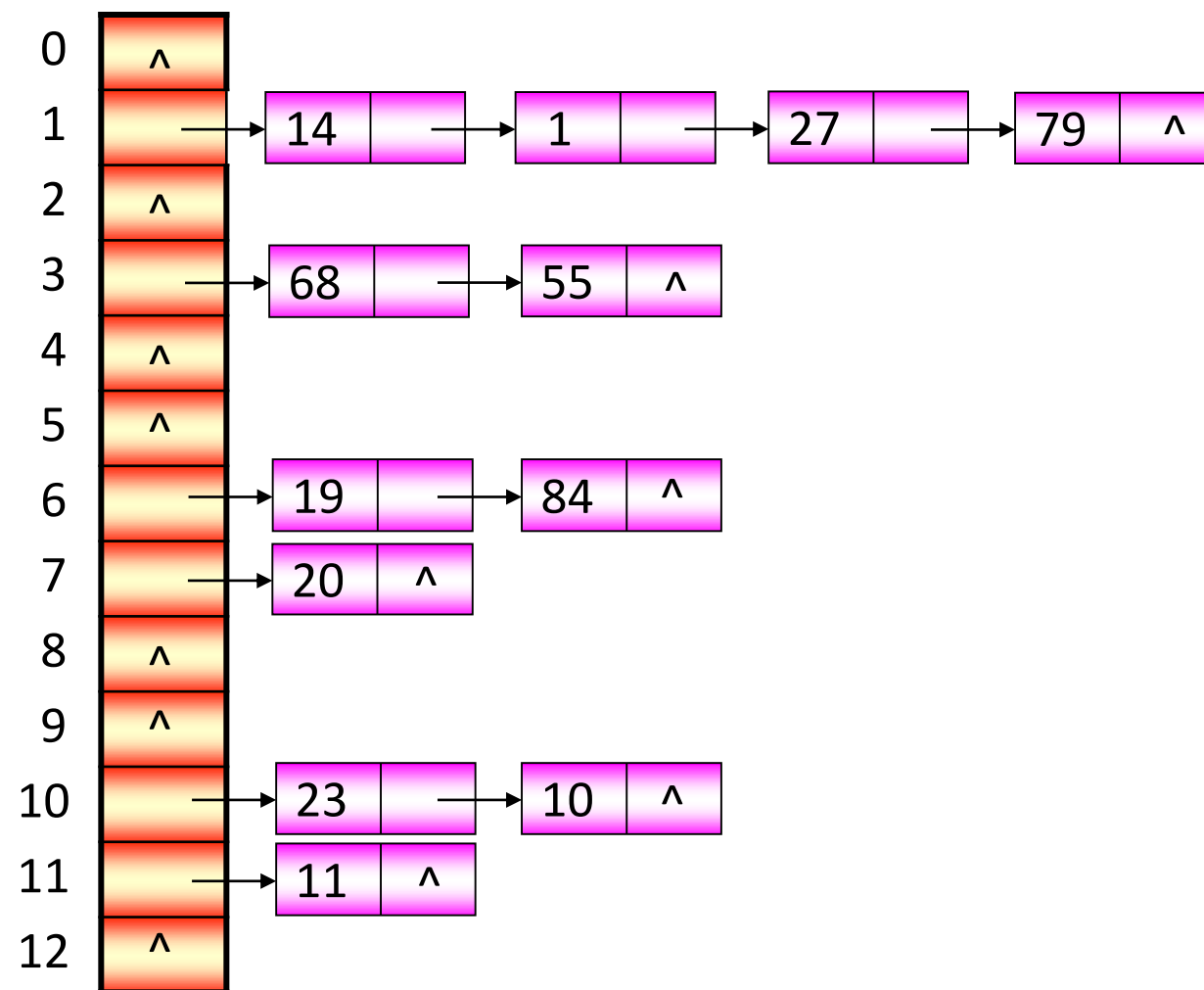
除基本的存储区外（称为基本表），另外建立一个公共溢出区（称为溢出表），当发生冲突时，记录可以存入这个公共溢出区。

## 4. 链地址法

方法：将所有关键字为同义词的记录存储在一个单链表（同义词子表）中，并用一维数组存放头指针。

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

哈希函数为： $H(\text{key}) = \text{key} \text{ MOD } 13$ ，用链地址法处理冲突。

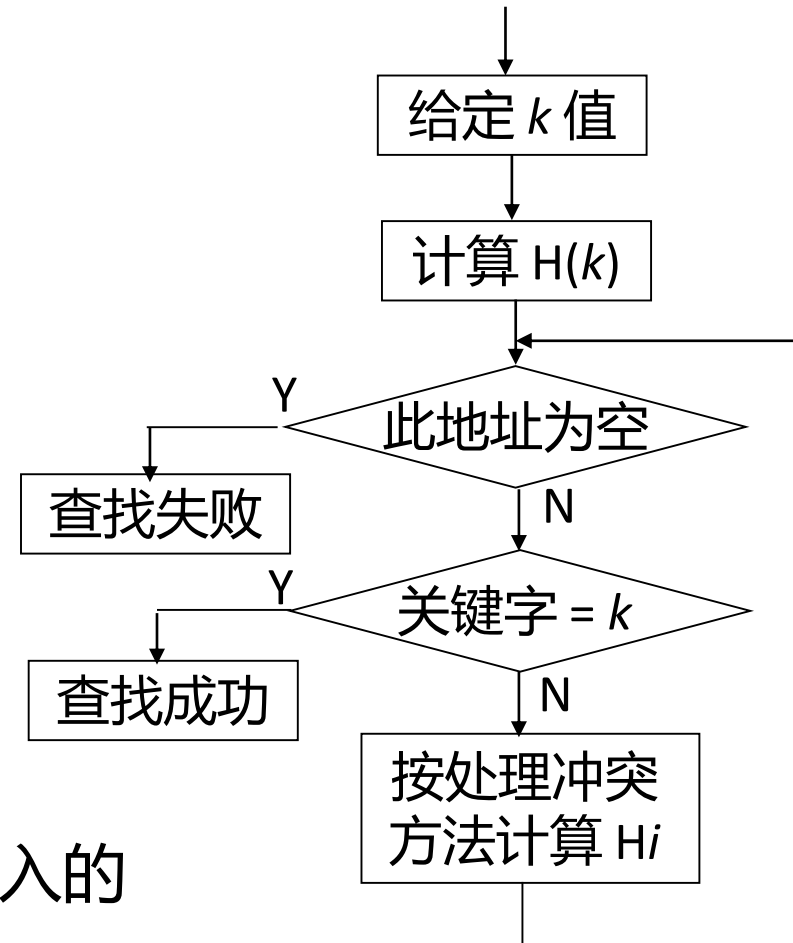


# 哈希表的查找及其分析

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：

哈希查找分析：

- 哈希查找过程仍是一个给定值与关键字进行比较的过程；
- 评价哈希查找效率仍要用ASL；
- 决定哈希表查找的ASL的因素：
  - 1) 选用的哈希函数；
  - 2) 选用的处理冲突的方法；
  - 3) 哈希表饱和的程度，**装载因子**  
 $\alpha = n/m$  值的大小（ $n$  —— 表中填入的记录数， $m$  —— 哈希表的长度）



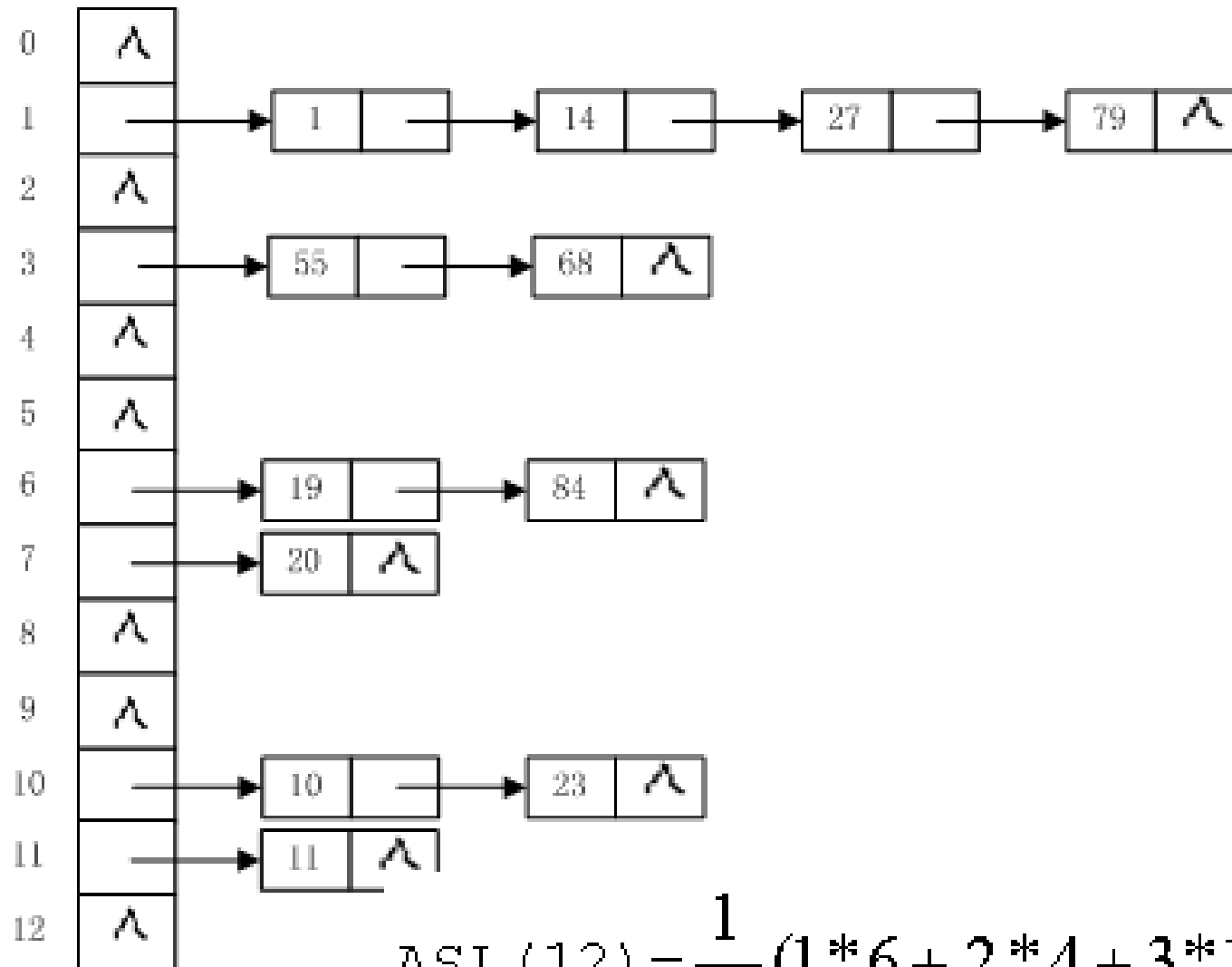


例如：一组关键字{19,14,23,1,68,20,84,27,55,11, 10,79}按照哈希函数 $\text{Hash}(\text{key})=\text{key}\%13$ 和线性探测再散列处理冲突得到的哈希表。

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

$$\text{ASL}(12) = \frac{1}{12} (1 * 6 + 2 * 1 + 3 * 3 + 4 * 1 + 9 * 1) = 2.5$$

例如：一组关键字{19,14,23,1, 68,20,84,27,55,11,10,79}按照哈希函数 $\text{Hash}(\text{key})=\text{key}\%13$ 和链地址法处理冲突得到的哈希表。



$$\text{ASL}(12) = \frac{1}{12} (1 * 6 + 2 * 4 + 3 * 1 + 4 * 1) = 1.75$$

一般情况下，可以认为选用的哈希函数是“均匀”的（即：对于关键字集中的任一关键字，经哈希函数映像到地址集中任何一个地址的概率是相等的），则在讨论 ASL 时，可以不考虑其它的因素。

因此，哈希表的 ASL 是**处理冲突方法**和**装载因子**的函数。

可以证明：**查找成功**时平均查找长度有下列结果：

线性探测再散列：
$$S_{nl} \approx \frac{1}{2} \left( 1 + \frac{1}{1-\alpha} \right)$$

随机探测再散列：
$$S_{nr} \approx -\frac{1}{\alpha} \ln(1-\alpha)$$

链地址法：
$$S_{nc} \approx 1 + \frac{\alpha}{2}$$

哈希表的平均查找长度是  $\alpha$  的函数，而不是  $n$  的函数。

——这是哈希表所特有的特点。

# 小结

- 相关概念术语
- 静态查找表
  - 1、顺序查找
  - 2、有序表查找
  - 3、索引顺序表查找
- 动态查找表
  - 1、二叉排序树
  - 2、平衡二叉树
  - 3、B-树、B+树
- 哈希表
  - 1、相关概念
  - 2、哈希函数
  - 3、处理冲突的方法
  - 4、哈希表的查找过程

**Thank You !**