数据结构
第九章 查找

主讲：陈锦秀

厦门大学信息学院计算机系

何谓查找表 ？

- 查找表是由同一类型的数据元素(或记录)构成的集合。
- 由于“集合”中的数据元素之间存在着松散的关系，因此查找表是一种应用灵便的结构。

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。

查找表可分为两类:

- 静态查找表
  - 仅作查询和检索操作的查找表。
- 动态查找表
  - 在查找过程中同时插入查找表中不存在的数据元素；或者，从查找表中删除已存在的某个数据元素。

关键字和查找

- 关键字（Key）是数据元素（或记录）中某个数据项的值，用它可以标识一个数据元素（或记录）。
  - 若此关键字可以惟一地标识一个记录，则称此关键字为主关键字。
  - 反之，称用以识别若干记录的关键字为次关键字。
- 查找（Searching）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记录或数据元素。
  - 若表中存在这样的一个记录，则称“查找成功”，此时查找的结果为该记录的信息，或指示该记录在查找表中的位置；
  - 若表中不存在关键字等于给定值的记录，则称“查找不成功”，此时可给出一个“空”记录或“空”指针。

如何进行查找？

- 查找的方法取决于查找表的结构。
- 由于查找表中的数据元素之间不存在明显的组织规律，因此不便于查找。
- 为了提高查找的效率， 需要在查找表中的元素之间人为地附加某种确定的关系，换句话说， 用另外一种结构来表示查找表。

关键字

- 典型的关键字类型说明：

```
typedef float KeyType;
typedef int KeyType;
typedef char * KeyType;
```

- 数据元素可以定义为：

```
typedef struct{
      KeyType key;
      …
} ElemType;
```

- 对两个关键字的比较约定为如下的宏定义：

对于数值类型

```
#define EQ(a, b) ((a)==(b))
#define LT(a, b) ((a)<(b))
#define LQ(a, b) ((a)<=(b))
```

对于字符串类型

```
#define EQ(a, b) (!strcmp((a),(b)))
#define LT(a, b) (strcmp((a),(b))<0)
#define LQ(a, b) (strcmp((a),(b))<=0)
```

第九章 查找

- 折半查找
- 斐波那契查找
- 插值查找

9.1 静态查找表

9.1.1 顺序查找表

9.1.2 有序查找表

9.1.3 静态查找树表

9.1.4 索引顺序表

- 针对有序表，不等概率查找
- 构造次优查找树

9.2 动态查找表

9.2.1 二叉排序树

9.2.2 平衡二叉树

9.2.3 B-树和B+树

9.3 哈希查找表

9.1 静态查找表

假设静态查找表的顺序存储结构为：

```
typedef struct {
    ElemType *elem;
        // 数据元素存储空间基址，建表时
        // 按实际长度分配，0号单元留空
    int length;   // 表的长度
} SSTable;
```

数据元素类型的定义为:

```
typedef struct {
    keyType key;        // 关键字域
        … …             // 其它属性域
} ElemType , TElemType ;
```

9.1.1 顺序查找表

- 应用范围：顺序表或线性链表表示的*静态查找表*
- 表内元素之间无序。
- 顺序查找表的实现：

设置哨兵的好处：
在顺序表中总可以找到待查结点。否则，必须将判断条件 i >= 0 加进 for 语句。

```
int  Search_Seq( Sstable  ST,  KeyType key )
 {   ST.elem[0]. key  = key;  // 哨兵
   for ( i = ST.length ; ! EQ(ST.elem[i]. key, key ) ; - - i )
                    //从后往前查找
   return i; //  返回 0，查找失败，否则，找到 key  所在的
              //   数组元素的下标地址
 }  // Search_Seq
```

9.1.1 顺序查找表

例1: 查找 key = 8 的结点所在的数组元素的下标。

	i	i	i		i	i	i	i
key								
数组 ST.elem	8	100	10	....	0	7	1	3
	0	1	2		n-3	n-2	n-1	n

查找失败，则 i = 0;

9.1.1 顺序查找表

例2: 查找 key = 8 的结点所在的数组元素的下标。

	key					i	i	i
数组 ST.elem	8	100	10	……………….	0	8	1	3
	0	1	2		n-3	n-2	n-1	n

查找成功，则 i 是key 值为 8 的结点所在的数组元素的下标，即 i = n-2;

9.1.1 顺序查找表

分析顺序查找的时间性能：

- 定义：　查找算法的平均查找长度 (Average Search Length)：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

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

其中：n 为表长，P_i 为查找表中第i个记录的概率，

且 $\sum_{i=1}^{n} P_i = 1$ ，C_i为找到该记录时，曾和给定值比较过的关键字的个数。

9.1.1 顺序查找表

- 对顺序表而言，$C_i = n\text{-}i\text{+}1$

则：$ASL = nP_1 + (n\text{-}1)P_2 + \ldots + 2P_{n\text{-}1} + P_n$

- 查找成功情况下，设每个结点的查找概率相等，即：$P_i = \dfrac{1}{n}$

- 顺序表查找的平均查找长度为：

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

9.1.1 顺序查找表

- 如果考虑查找不成功，则要求一般情况下（包括成功、不成功两种情况）的平均查找长度**ASL**
- 设成功与不成功两种情况可能性相等，各为**1/2**。成功有**n**种可能，不成功可以分为**n+1**种可能。每种可能的查找概率也相等。

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

9.1.1 顺序查找表

- 有时候各个记录的查找概率并不相等，为了提高查找效率，把表中记录按查找概率由小至大重新排列。则在**不等概率查找**的情况下，ASL_{ss}在

$$P_n \geqslant P_{n-1} \geqslant \cdots \geqslant P_2 \geqslant P_1$$

时**取极小值。**

- 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。**

9.1.2 有序查找表——折半查找

- 上述顺序查找表的查找算法简单，但平均查找长度较大，特别不适用于表长较大的查找表。
- 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。

——折半查找（或二分查找法）

- 应用范围：顺序表，表内元素之间有序。不可直接用于线性链表。

9.1.2 有序查找表——折半查找

例如: 查找 **key = 9** 的结点所在的数组元素的下标地址。

- 查找成功的情况：数组 **ST.elem** 如下图所示有序
- 数组 **ST.elem** ：

  递增序 **ST.elem[i]. Key <= ST.elem[i+1]. Key;　i= 1,2,……n-1**
- 查找范围 ： **low**(低下标)**= 1; high**(高下标)**= 7** （初始时为最大下标 **n** ）；
- 比较对象：中点元素，其下标地址为 **mid =** $\lfloor$（**low+high**）**/ 2**$\rfloor$ **=4**

mid=4 但 key=9 < 10, 向左

key

	4	8	9	10	11	13	19
0	1	2	3	4	5	6	7

low=1　　　　high=7

9.1.2 有序查找表——折半查找

- 查找成功的情况：数组 ST.elem 如下图所示有序
- 查找范围 ：low（低下标）= 1; high（高下标）= 3 ;
  low（低下标）= 3; high（高下标）= 3 ;
- 比较对象：中点元素，其下标地址为 mid = ⌊(low+high）/ 2⌋ =2
  中点元素，其下标地址为 mid = ⌊(low+high）/ 2⌋ =3

9.1.2 有序查找表——折半查找

例如: 查找 key = 5 的结点所在的数组元素的下标地址。

- 查找不成功的情况：数组 ST.elem 如下图所示有序
- 数组 ST.elem ：

  递增序 ST.elem[i]. Key <= ST.elem[i+1]. Key;   i= 1,2,……n-1
- 查找范围 ： low(低下标)= 1; high(高下标)= 7 （初始时为最大下标 n ）;
- 比较对象：中点元素，其下标地址为 mid = $\lfloor$（low+high）/ 2$\rfloor$ =4

mid=4 但 key=5<10, 向左

key

	4	8	9	10	11	13	19
0	1	2	3	4	5	6	7

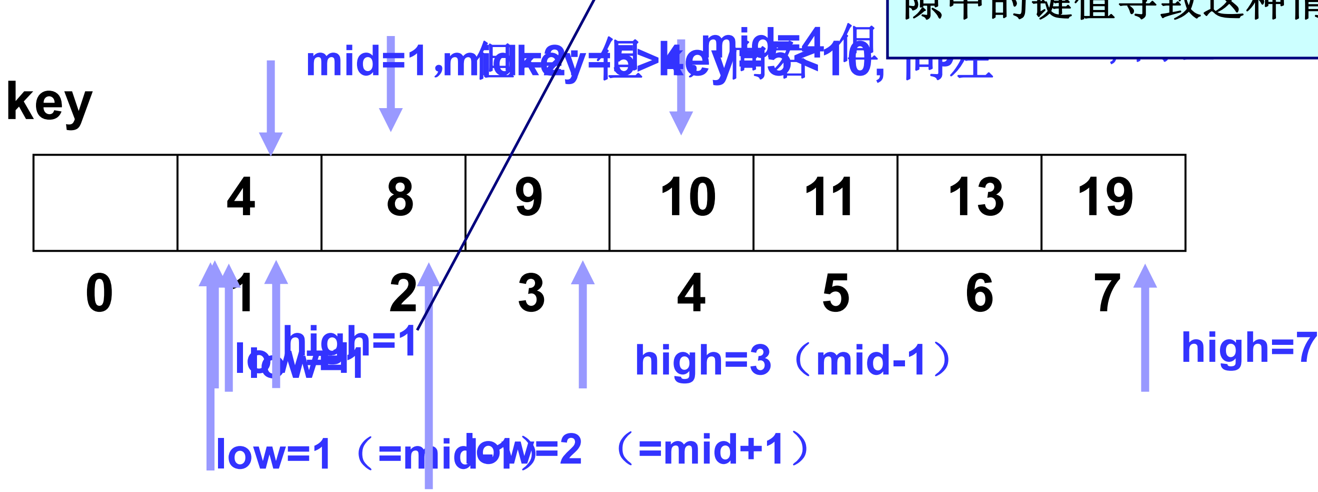
low=1

high=7

9.1.2 有序查找表——折半查找

- 查找不成功的情况：数组 ST.elem 如下图所示有序
- 查找范围 ： low（低下标）= 1; high（高下标）= 3 ;
  low（低下标）= 1; high（高下标）= 1 ;
- 比较对象：中点元素，其下标地址为 mid = ⌊（low+high）/ 2⌋ =2
  中点元素，其下标地址为 mid = ⌊（low+high）/ 2⌋ =1

失败条件：low > high; 处于间隙中的键值导致这种情况！



例如：key=64 的查找过程如下：

ST.length

ST.elem

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

low high

mid

low 指示查找区间的下界

high 指示查找区间的上界

mid = (low+high)/2

表示：**typedef struct** { ElemType * elem; **int** length; } SStable; **//** **length = n**

```
int Search_Bin ( SSTable ST, KeyType key ) {
// 在有序表中查找关键字之值为 key 的结点，找到返回该结点在
表中的下标地址，否则返回 0
  low = 1;  high = ST.length;      // 置区间初值
  while (low <= high) {
    mid = (low + high) / 2;
    if （EQ (key , ST.elem[mid].key) )
      return  mid;        // 找到待查元素
    else  if ( LT (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
```

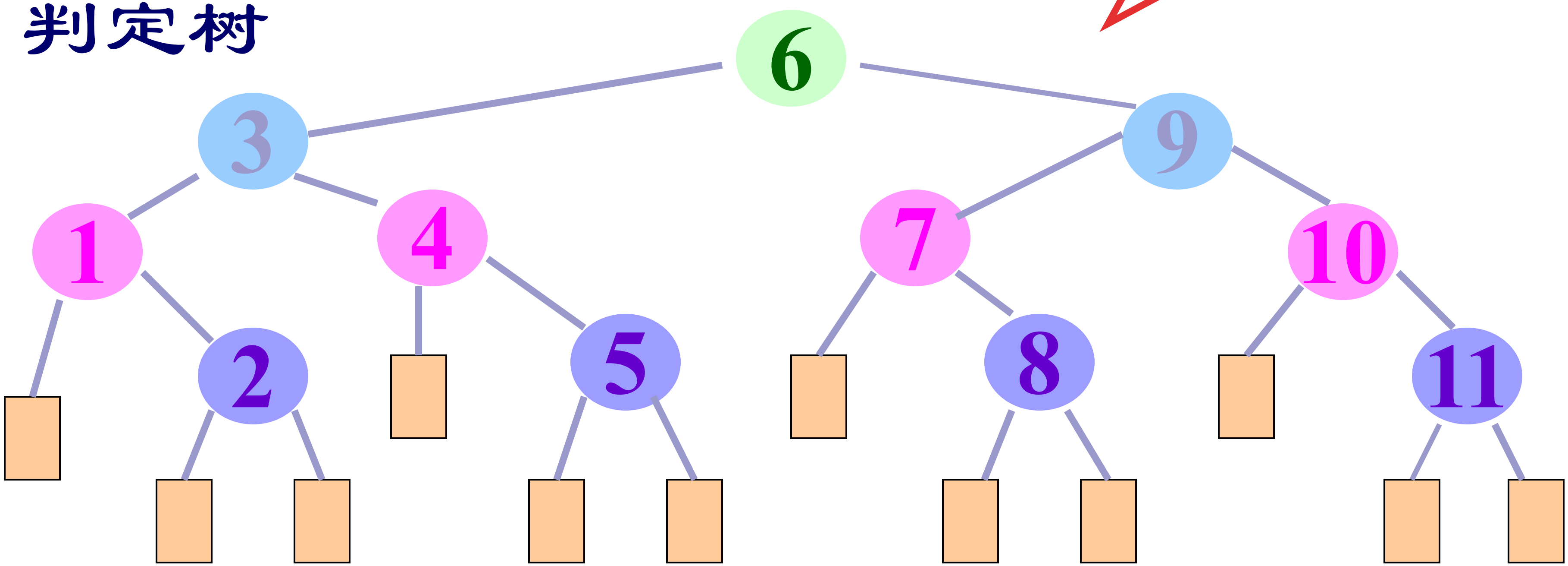
分析折半查找的平均查找长度：

先看一个具体的情况，假设：n=

i	1	2	3	4	5	6	7
Ci	3	4	2	3	4	1	3

找到有序表中任一记录的过程就是走了一条从根结点到与该记录相应的结点的路径，和给定值进行比较的关键字个数恰为该结点在判定树上的层次数。

判定树



9.1.2 有序查找表——折半查找

- 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。
- 具有n个结点的判定树的深度为$\lfloor \log_2(n) \rfloor + 1$
- 折半查找法在查找成功时进行比较的关键字个数最多不超过树的深度$\lfloor \log_2(n) \rfloor + 1$。
  - 找到有序表中任一记录的过程就是走了一条从根结点到与该记录相应的结点的路径
- 查找不成功时进行比较的关键字个数最多也不超过树的深度$\lfloor \log_2(n) \rfloor + 1$。
  - 走了一条从根结点到与外部结点的路径，和给定值进行比较的关键字个数等于该路径上内部结点个数。

9.1.2 有序查找表——折半查找

- 假定有序表的长度为$n=2^h-1$（反之 $h=\log_2(n+1)$），

则描述折半查找的判定树是深度为h的满二叉树。

  - 树中层次为1的结点有1个， 树中层次为2的结点有2个， ……
  - 树中层次为h的结点有2^{h-1}个

- 假设查找概率相等（$P_i=\dfrac{1}{n}$）则平均情况下查找成功时：

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

在n>50时，可得近似结果

$$ASL_{bs}\approx\log_2(n+1)-1$$

9.1.2 有序查找表——折半查找

- 平均情况分析（在成功、非成功查找两种的情况下）：
- 为了简单起见，设结点个数为 $n = 2^t - 1$ 。
- 这样，成功查找的情况共有n种情况，非成功查找的情况共有n+1情况。
- 设每种情况出现的概率相同，即都为 1/(2n+1) 。

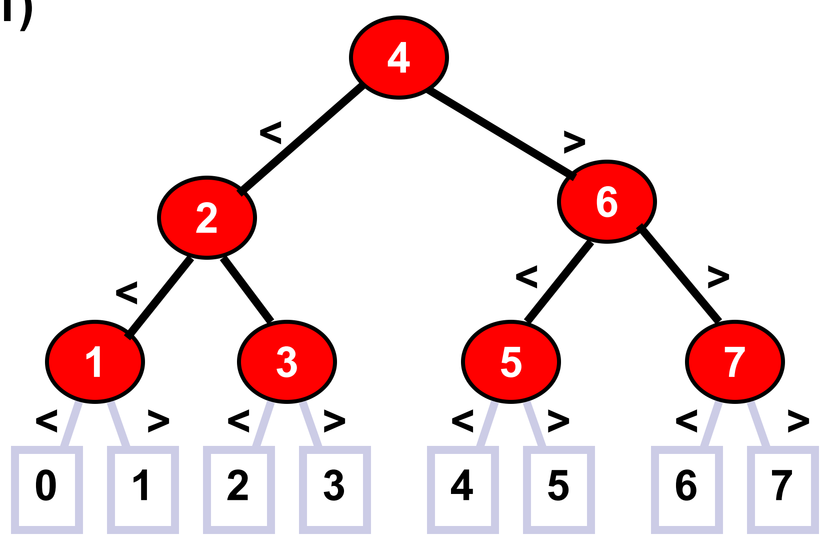
$$ASL = \sum_{i=1}^{t} (i \times 2^{i-1} + t \times (n+1)) / (2n+1)$$

$$= O（\log n）$$

e.g: 当 t = 3 时的例子：

成功：最多经过 t=3 次比较

失败：都必须经过 t = 3 次比较



9.1.2 有序查找表——Fibonacci查找

- 斐波那契查找是根据斐波那契序列的特点对表进行分割的。

  Fibonacci 数定义： $F_0 = 0$

  $F_1 = 1$

  $F_i = F_{i-1} + F_{i-2}$

  如：0,1,1,2,3,5,8,13,21,34,55,89,144,233

- **实现:** 假设开始时表中记录个数 $n = F_u-1$，查找键值为 key 的结点

  首先比较 key $\overset{?}{=}$ ST.elem[F_{u-1}]

  1. 如果 key = ST.elem[F_{u-}
  2. 如果 key < ST.elem[F_{u-1}]

     则继续在自ST.elem[1]至ST.e
     进行查找；

  3. 如果 key > ST.elem[F_{u-1}].key ，

     则继续在自ST.elem[[F_{u-1}+1]至ST.elem[F_u-1].key 的子表中进行查找；

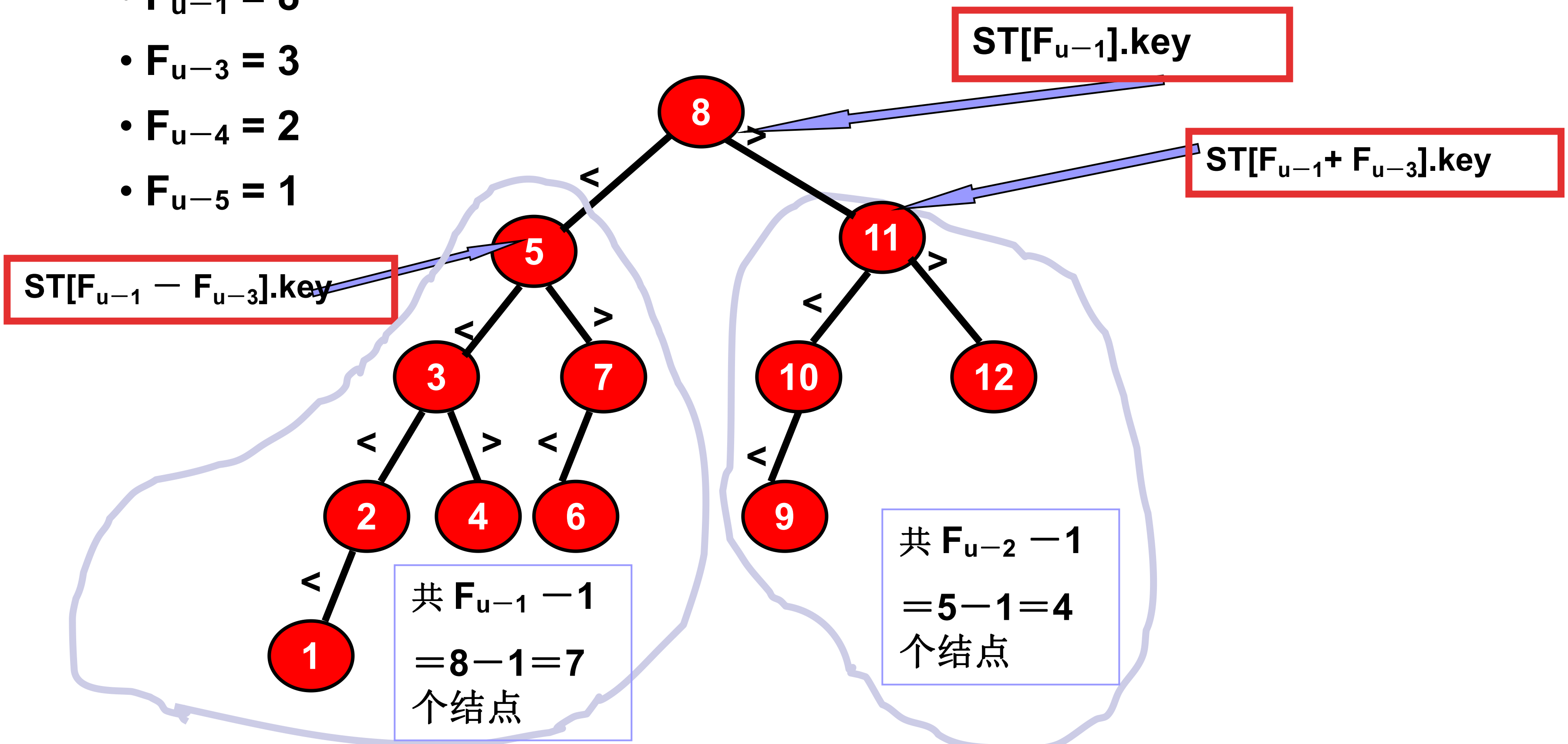
下一个比较key 和
ST.elem[F_{u-1} - F_{u-3}].key

下一个比较key 和
ST.elem[F_{u-1} + F_{u-3}].key

9.1.2 有序查找表——Fibonacci查找

例子： $n = F_7 - 1 = 13 - 1 = 12$个结点的查找过程，这时$u=7$。

- $F_{u-1} = 8$
- $F_{u-3} = 3$
- $F_{u-4} = 2$
- $F_{u-5} = 1$



优点：只用 ＋、－法，不用除法。平均查找速度比折半查找更快。O（$\log_2 n$）级。

缺点：最坏情况下比二分查找法差。必须先给出 Fibonacci 数。

9.1.2 有序查找表——插值查找

- 插值查找是根据给定值key来确定进行比较的关键字ST.elem[i].key的查找方法。
  - 除中点下标的选择和二分查找不同外，其余类似。
- 实现： 设mid为中点的下标。low为具有最小关键字值结点的下标，high为具有最大关键字值结点的下标。

$$mid = \frac{key - ST.elem[low].key}{ST.elem[high].key - ST.elem[low].key}(high - low + 1)$$

- 举例：如果在10份试卷里，查找成绩为90分的记录。那么

mid=(10-1+1)*(90-0)/(100-0)=9

- 插值查找基于low点和high点的关键值，考虑待查关键值的可能位置。适用于关键字平均分布的表，在这种情况下，对表长较大的顺序表，其平均性能比折半查找好。

9.1.3 静态树表的查找

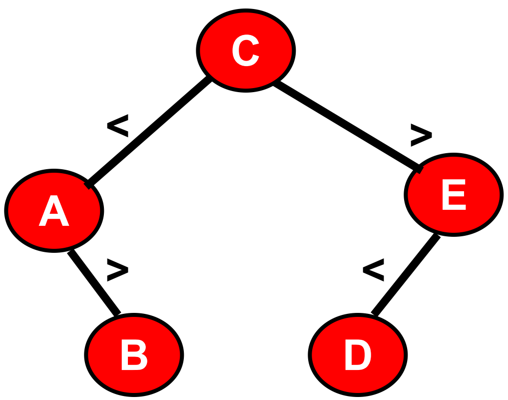
- 在不等概率查找的情况下，折半查找不是有序表最好的查找方法。
- 静态树表的查找

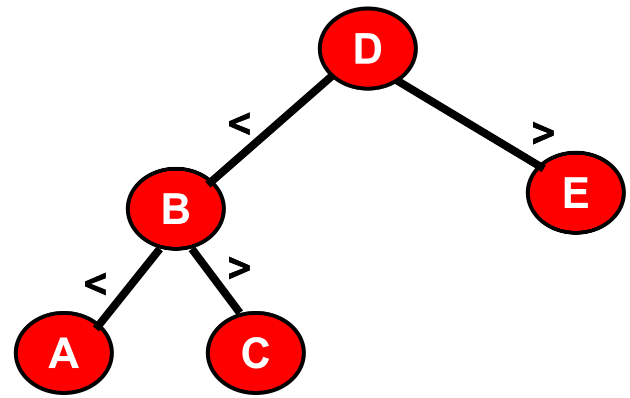
  ——应用范围：有序表、查找概率不等。

9.1.3 静态树表的查找

- 举例：已知 A、B、C、D、E；查找概率分别为：0.1、0.2、0.1、0.4、0.2

  求成功查找时的平均查找长度？





平均查找长度＝

1×0.1+2×0.1+ 2×0.2+ 3×0.2+ 3×0.4

＝2.5

平均查找长度＝

1×0.4+2×0.2+ 2×0.2+ 3×0.1+ 3×0.1

＝1.8

• 例子说明查找概率大者越接近根结点，那么代价之和将越小。

9.1.3 静态树表的查找

- 如果只考虑查找成功的情况，则使查找性能最佳的判定树是其带权内路径长度之和PH 值最小的二叉树，称为静态最优查找树：

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

  - 其中，n为二叉树上结点的个数（即有序表的长度）；h_i为第i个结点在二叉树上的层次数；结点的权$w_i=\alpha p_i$，p_i为结点的查找概率，α为某个常量。

- 构造最优查找树花费的时间代价较高

——构造次优查找树：构造近似最优查找树的有效算法。

9.1.3 静态树表的查找

- 介绍一种次优二叉树的构造方法——选择二叉树的根结点，使下式达最小：

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

- 然后分别对子序列$\{r_l, r_{l+1}, \cdots, r_{i-1}\}$，$\{r_{i+1}, r_{i+2}, \cdots, r_h\}$构造两棵次优查找树，并分别设为根结点$r_i$的左子树和右子树。
- 为便于计算，引入累计权值和：$sw_i = \sum_{j=l}^{i} w_j$
- 并设 $w_{l-1} = 0$ 和 $sw_{l-1} = 0$，则推导可得

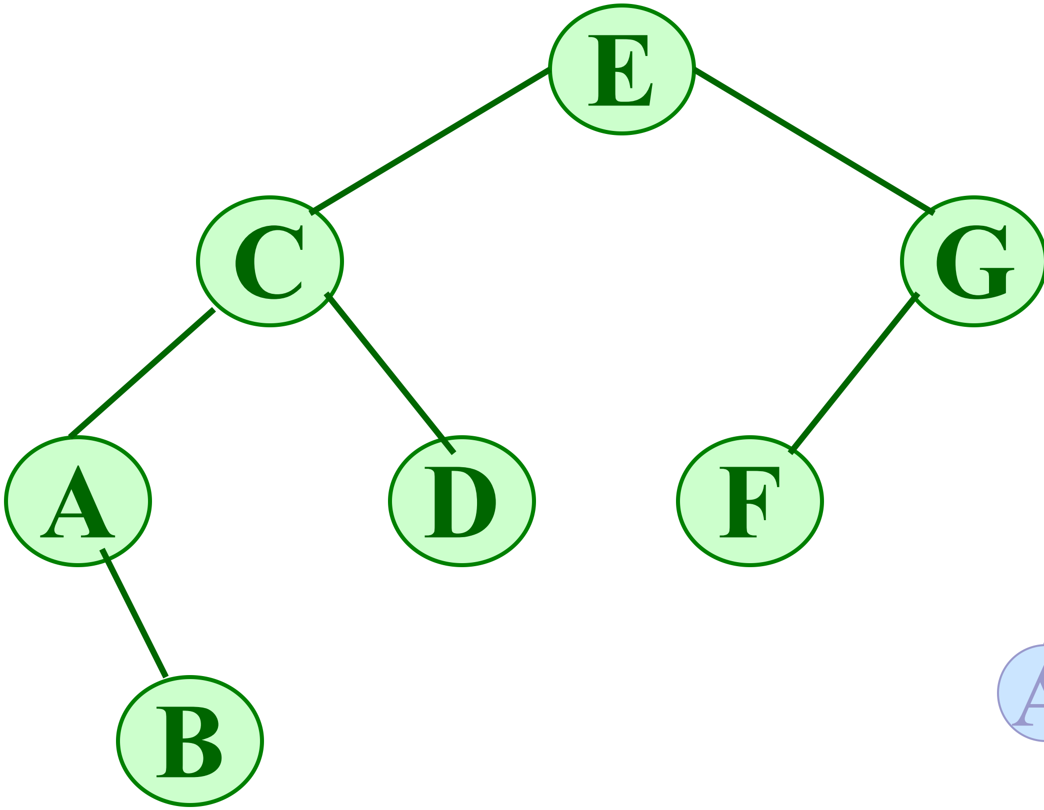
$$\Delta P_i = \left|(sw_h - sw_i) - (sw_{i-1} - sw_{l-1})\right| = \left|(sw_h + sw_{l-1}) - sw_i - sw_{i-1}\right|$$

例如：

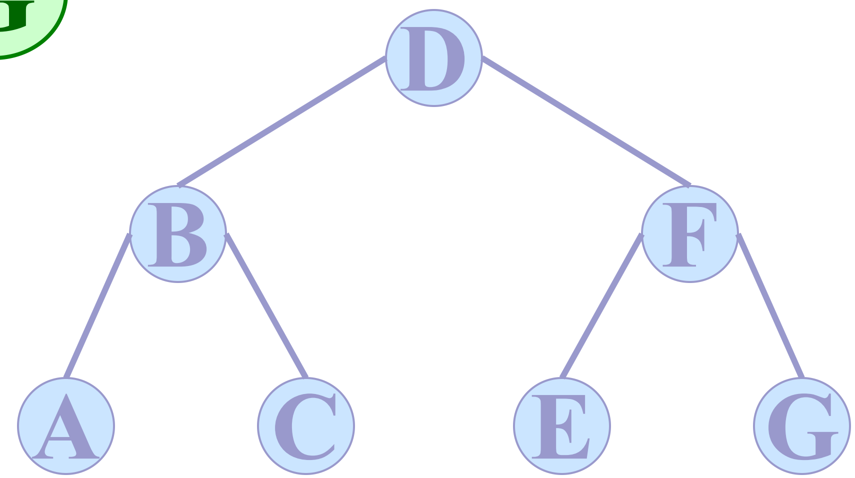
l h

j	0	1	2	3	4	5	6	7
w_j	0	2	1	5	3	4	3	5
sw_j	0	2	3	8	11	15	18	23
Δp_j		21	18	12	4	3	10	18
		9	6	0	8		5	3
		1	2					
key		A	B	C	D	E	F	G

所得次优二叉树如下所示:



和折半查找相比较



查找比较“总次数”

$= 3\times 2+4\times 1+2\times 5+3\times 3$

$+1\times 4+3\times 3+2\times 5 = 52$

查找比较“总次数”

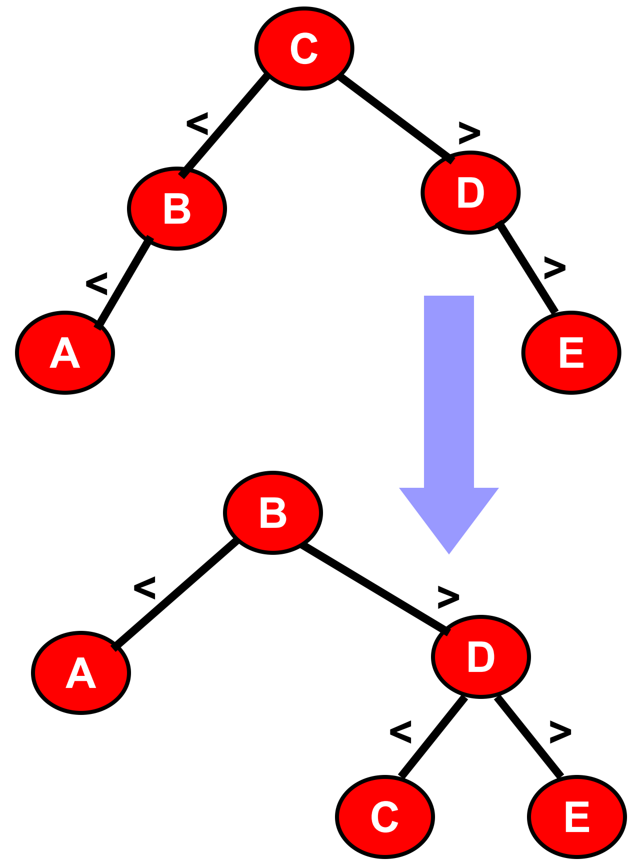
$= 3\times 2+2\times 1+3\times 5+1\times 3$

$+3\times 4+2\times 3+3\times 5 = 59$

9.1.3 静态树表的查找

A、B、C、D、E

1 30 2 29 3



- 由于在构造次优查找树的过程中，没有考察单个关键字的相应权值，则有可能出现被选为根的关键字的权值比与它相邻的关键字的权值小。

——此时应作适当调整：选取邻近的权值较大的关键字作次优查找树的根结点。（如图）

- 构造次优查找树两原则：

1. 两边权值尽可能相等；

2. 根结点或子树的根结点应尽量大。

构造次优二叉树的算法：

```
Status SecondOptimal(BiTree &T, ElemType R[],
                     float sw[], int low, int high)
{// 由有序表R[low..high]及其累计权值表sw递归构造次优查找树T。
 选择最小的ΔPi值
 if (!(T = (BiTree)malloc(sizeof(BiTNode))))
    return ERROR;
 T->data = R[i];                 // 生成结点
 if (i==low)  T->lchild = NULL;  // 左子树空
 else  SecondOptimal(T->lchild, R, sw, low, i-1);    // 构造左子树
 if (i==high)  T->rchild = NULL;  // 右子树空
 else  SecondOptimal(T->rchild, R, sw, i+1, high);   // 构造右子树
 return OK;
} // SecondOptimal
```

次优查找树采用二叉链表的存储结构

```
Status CreateSOSTre(SOSTree &T, SSTable ST) {
  // 由有序表 ST 构造一棵次优查找树 T
  // ST 的数据元素含有权域 weight
  if (ST.length = 0)  T = NULL;
  else {
    FindSW(sw, ST);
               // 按照有序表 ST 中各数据元素
             // 的 weight 值求累计权值表
    SecondOpiamal(T, ST.elem, sw, 1, ST.length);
  }
  return OK;
} // CreatSOSTree
```

9.1.4 索引顺序表的查找

- 应用范围：分块表示的顺序表。
  - 块内元素之间无序、有序皆可。
  - 块间元素有序。
- 索引顺序表的查找过程：

  1）由索引确定记录所在区间；

  2）在顺序表的某个区间内进行查找。
- 可见， 索引顺序查找的过程也是一个“缩小区间”的查找过程。
- 注意：索引可以根据查找表的特点来构造。

9.1.4 索引顺序表的查找

例如: 查找 key = 47 的结点索引。

- 查索引表，确定在第三块
- 在第三块内进行顺序查找

块最大关键字有序

索引表			
块最大关键字	8	36	60
块起始地址	1	4	7

索引	1	2	3	4	5	6	7	8	9
顺序表	3	8	7	36	35	21	40	60	47

块内关键字无序（也可有序）

索引顺序查找的平均查找长度＝

查找“索引”的平均查找长度

＋查找“顺序表”的平均查找长度

9.1.4 索引顺序表的查找

1、索引表、顺序表皆顺序查找：

$$ASL=Lb+Lw = \sum_{j=1}^{b} (j/b) + \sum_{i=1}^{s} (i/s)$$

$$= (b+1)/2 + (s+1)/2 = (n/s+s)/2+1$$

当 s =sqr(n) 时，ASL极小

$$ASL= sqr(n) + 1$$

2、索引表二分查找、顺序表顺序查找：

$$ASL \approx \log2(n/s + 1) + s/2$$

注意：b: 块数， s: 块内元素数，b = (n/s)（上整数）

第九章 查找

9.1 静态查找表

9.1.1 顺序查找表

9.1.2 有序查找表

- 折半查找
- 斐波那契查找
- 插值查找

9.1.3 静态查找树表

- 针对有序表，不等概率查找
- 构造次优查找树

9.1.4 索引顺序表

9.2 动态查找表

9.2.1 二叉排序树

- 分割式查找法
- 插入算法
- 查找分析
- 删除算法

9.2.2 平衡二叉树

- 定义
- 如何构造
- 查找分析

9.2.3 B-树和B+树

9.3 哈希查找表

- 定义
- 查找、插入、删除
- 查找分析

9.2 动态查找表

- 特点：用于频繁进行插入、删除、查找的所谓动态查找表。
- 综合上一节讨论的几种查找表的特性：

	查找	插入	删除
无序顺序表	O(n)	O(1)	O(n)
无序线性链表	O(n)	O(1)	O(1)
有序顺序表	O(logn)	O(n)	O(n)
有序线性链表	O(n)	O(1)	O(1)
静态查找树表	O(logn)	O(nlogn)	O(nlogn)

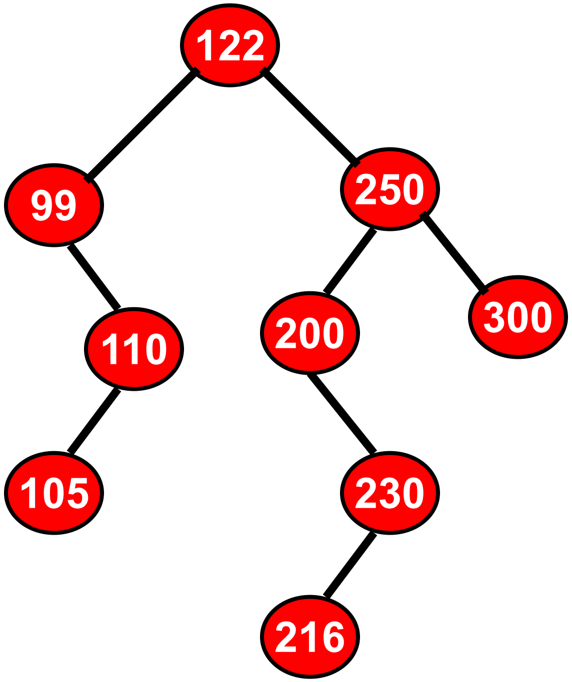
可得如下结论:

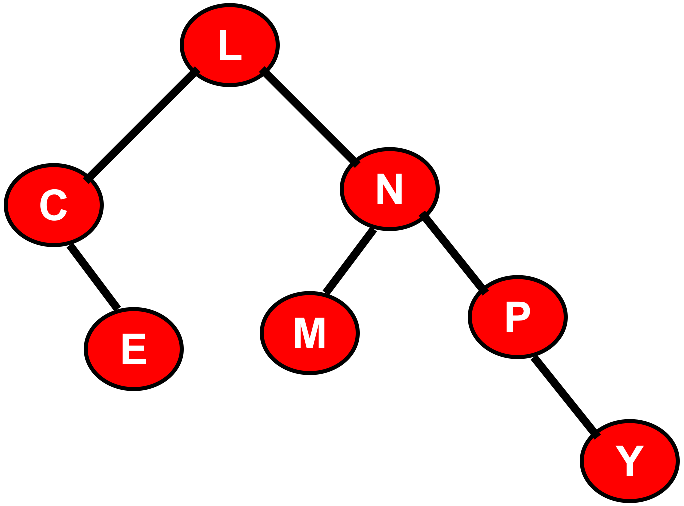
1）从查找性能看，最好情况能达O(logn)，此时要求表有序；

2）从插入和删除的性能看，最好情况能达O(1)，此时要求存储结构是链表。

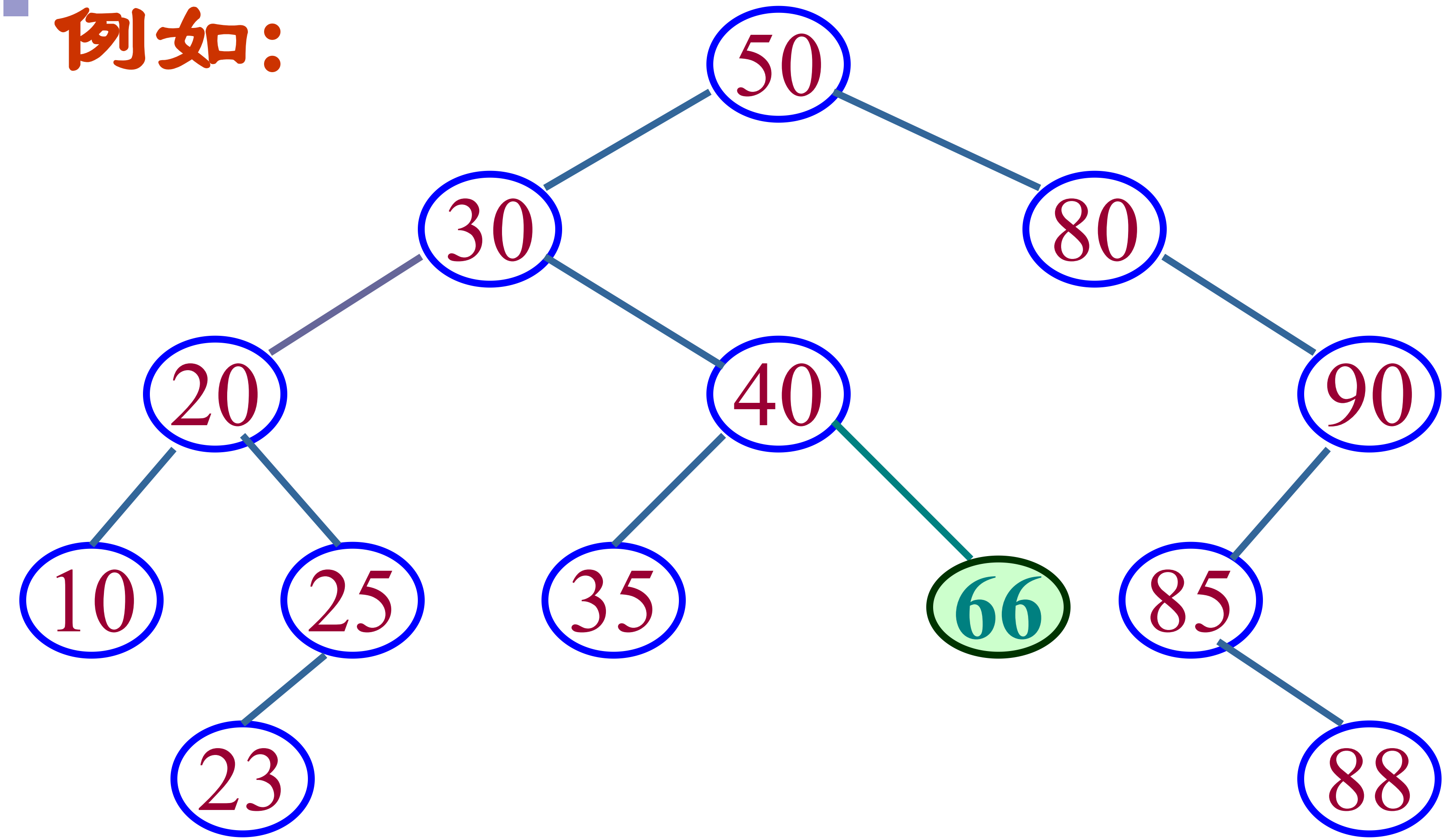
9.2.1 二叉排序树

- 二叉排序树定义：或者是一棵空树；或者是具有如下特性的二叉树：
  - 根的左子树若非空，则左子树上的所有结点的关键字值均小于根结点的值。
  - 根的右子树若非空，则右子树上的所有结点的关键字值均大于根结点的值。
  - 根结点的左右子树同样是二叉排序树。





例如：



不是二叉排序树。

9.2.1 二叉排序树

通常，取二叉链表作为二叉排序树的存储结构

```
typedef struct BiTNode { // 结点结构
    TElemType    data;
    struct BiTNode  *lchild, *rchild;
                                  // 左右孩子指针
} BiTNode, *BiTree;
```

9.2.1 二叉排序树——分割式查找法

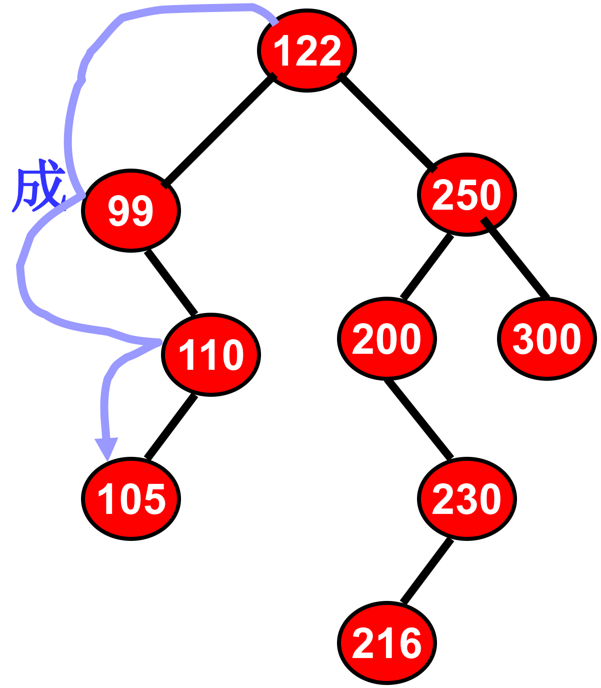
1、分割式查找法查找步骤：

若二叉排序树为空，则查找不成功；否则，

（1）若查找的关键字等于根结点的关键字值，成功。

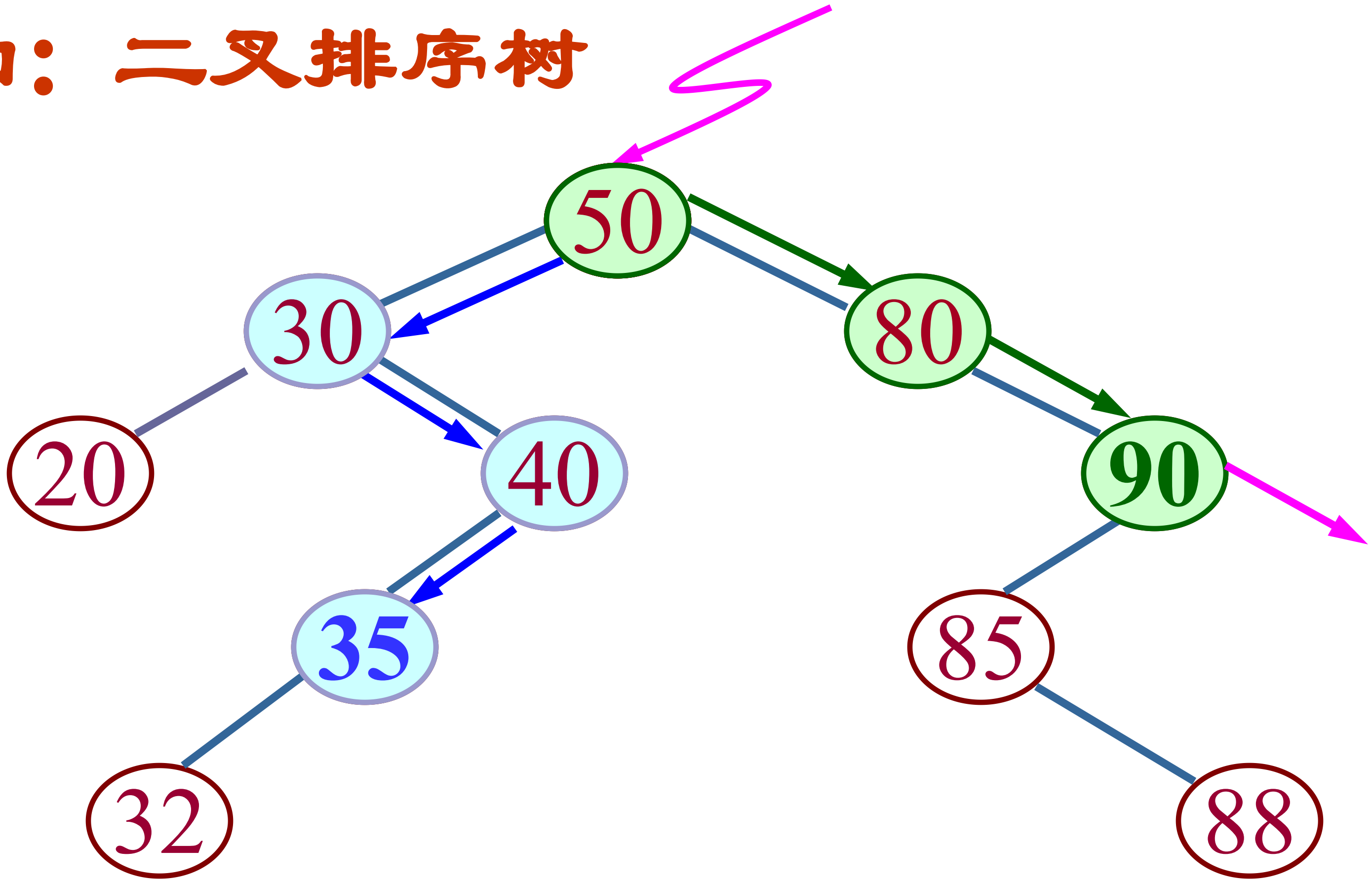
（2）若小于根结点的关键字值，查其左子树。

（3）若大于根结点的关键字值，查其右子树。



在左右子树上的操作类似。比如查找105。

例如：二叉排序树



查找关键字

== 50 , 35 , 90 , 95 ,

9.2.1 二叉排序树——分割式查找法

从上述查找过程可见，

在查找过程中，生成了一条**查找路径**:

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

或者

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

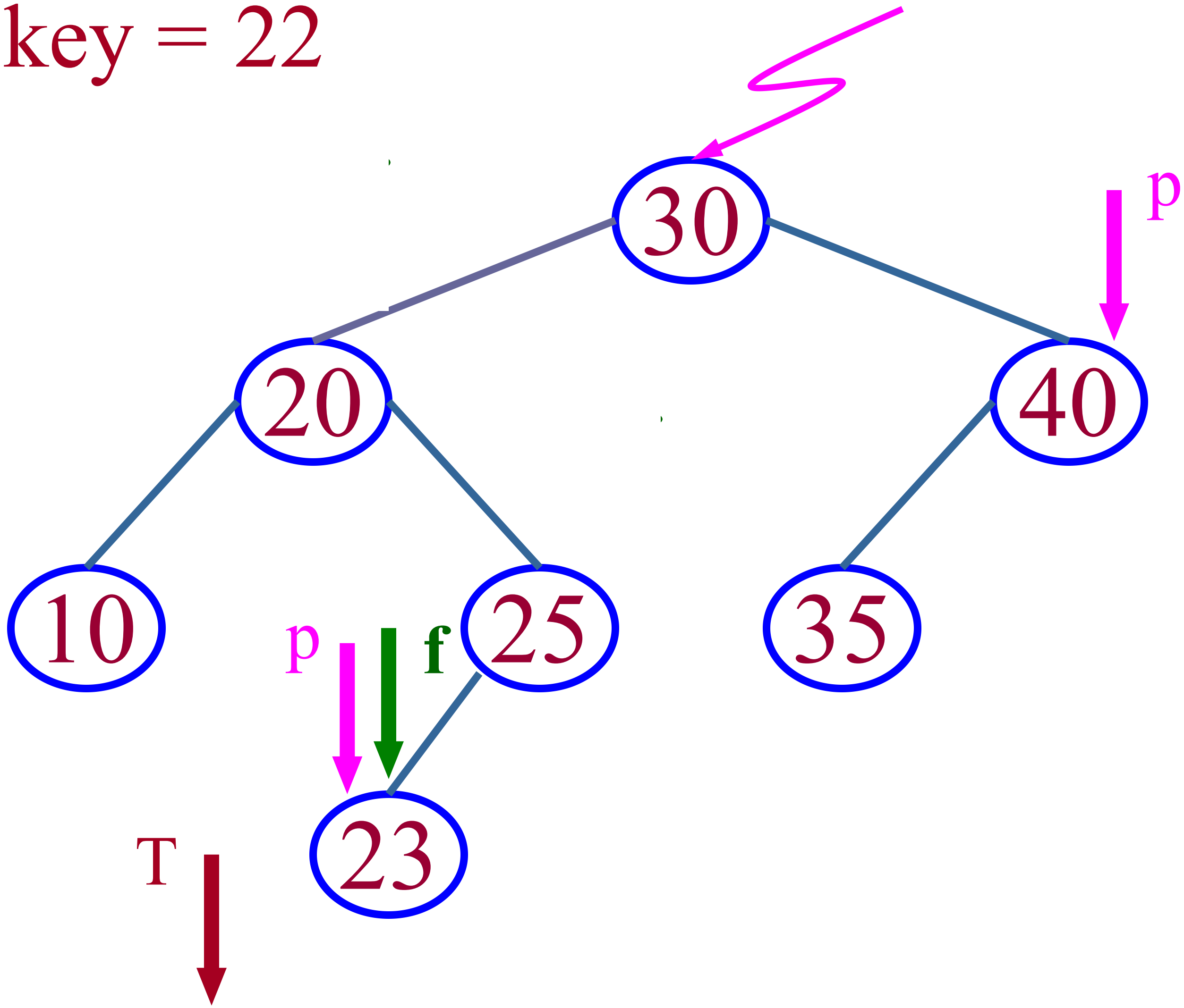
```
Status SearchBST (BiTree T, KeyType key, BiTree f, BiTree &p )
{ //在根指针T所指二叉排序树中递归地查找其关键字等于key的数据元素，
若查找成功，则返回指针p指向该数据元素的结点，并返回TRUE; 否则表明
查找不成功，返回指针p指向查找路径上访问的最后一个结点，并返回
FALSE，指针f指向当前访问的结点的双亲，其初始调用值为NULL

if (!T)
   { p = f;  return FALSE; } // 查找不成功
else  if ( EQ(key, T->data.key) )
          { p = T;  return TRUE; } // 查找成功
     else  if ( LT(key, T->data.key) )
          SearchBST (T->lchild, key, T, p ); // 在左子树中继续查找
            else  SearchBST (T->rchild, key, T, p );
                                      // 在右子树中继续查找

} // SearchBST
```

9.2.1 二叉排序树——分割式查找法

设 key = 22

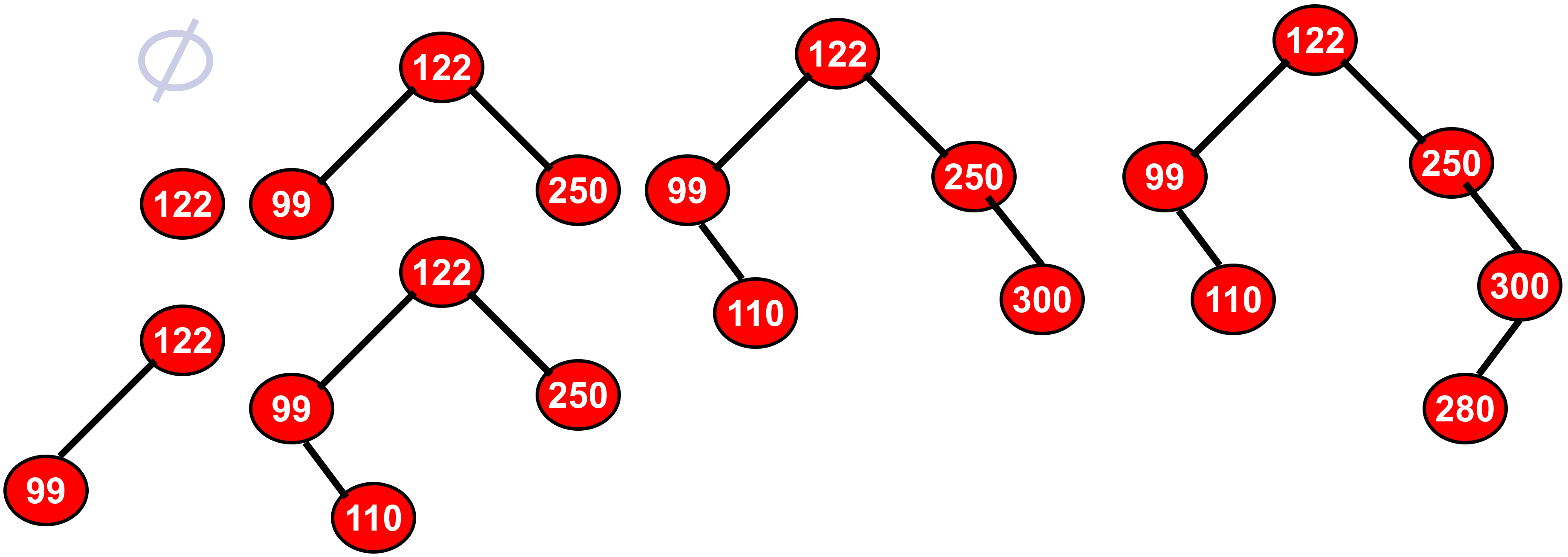


9.2.1 二叉排序树——插入算法

- 根据动态查找表的定义，“插入”操作在查找不成功时才进行；
  - 若二叉排序树为空树，则新插入的结点为新的根结点；否则，新插入的结点必为一个新的叶子结点，其插入位置由查找过程得到。
- 插入算法步骤：
  - 首先执行查找算法，找出被插结点的父亲结点。
  - 判断被插结点是其父亲结点的左、右儿子。将被插结点作为叶子结点插入。
  - 若二叉树为空。则首先单独生成根结点。
- 注意：新插入的结点总是叶子结点。

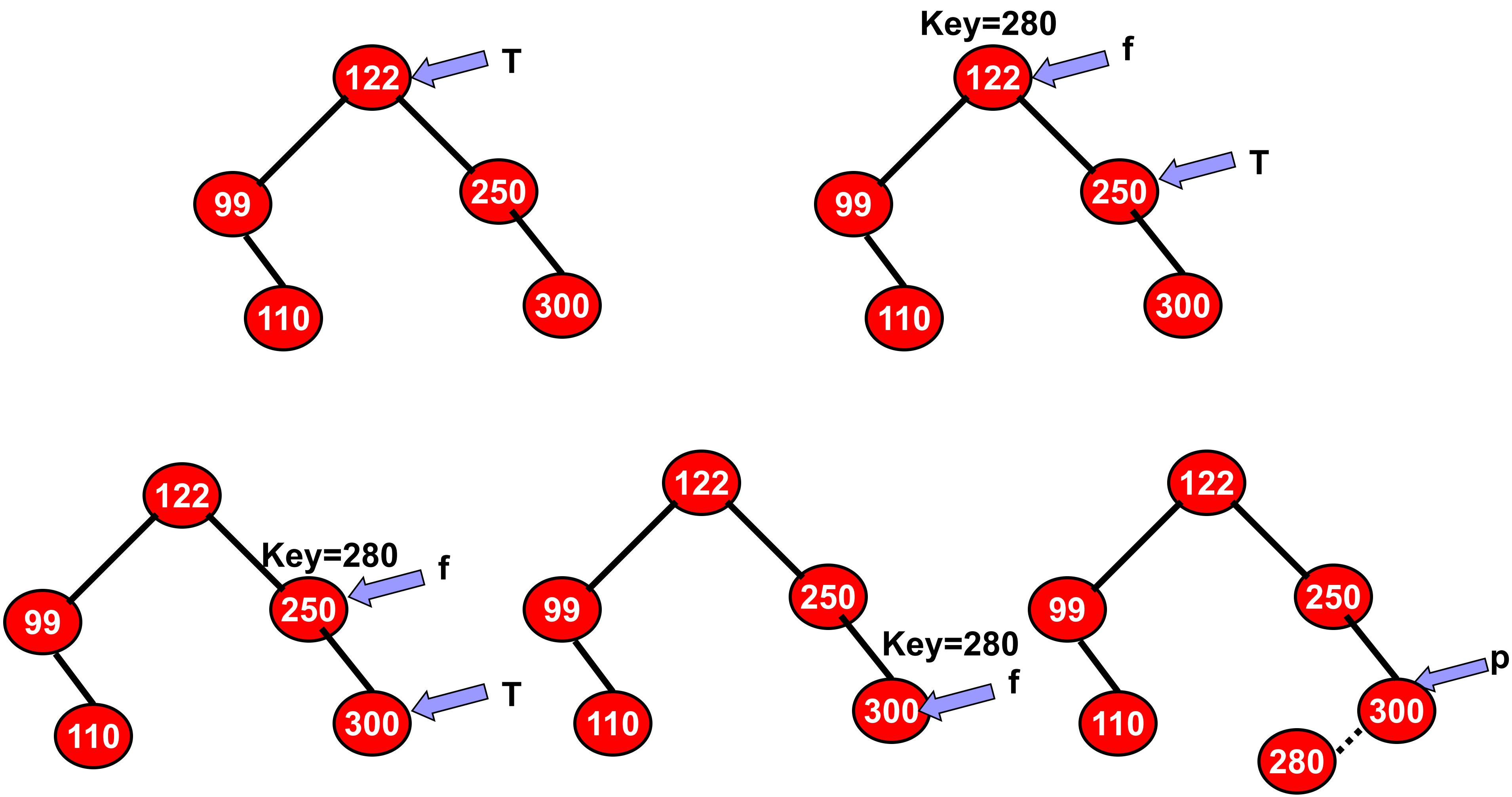
9.2.1 二叉排序树——插入算法

例子：将数的序列：**122、99、250、110、300、280** 作为二叉排序树的结点的关键字值，生成二叉排序树。



9.2.1 二叉排序树——插入算法

- 执行实例：插入值为 **280** 的结点



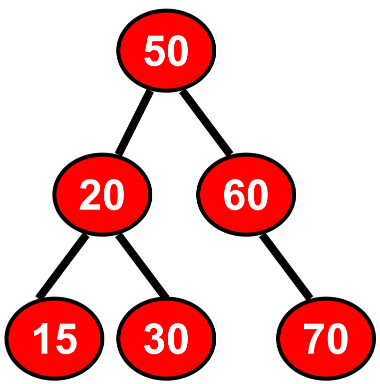
```
Status Insert BST(BiTree &T, ElemType e )
{ // 当二叉排序树中不存在关键字等于 e.key 的数据元素时，插入元素值为 e 的结点，并返回 TRUE; 否则，不进行插入并返回FALSE
   if (!SearchBST ( T, e.key, NULL, p ))
  {    s = (BiTree) malloc (sizeof (BiTNode));     // 为新结点分配空间
       s->data = e;
       s->lchild = s->rchild = NULL;
       if ( !p ) T = s;     // 插入 s 为新的根结点
      else   if ( LT(e.key, p->data.key) )        // 小于关系
                  p->lchild = s;       // 插入 *s 为 *p 的左孩子
              else  p->rchild = s;     // 插入 *s 为 *p 的右孩子
      return TRUE;     // 插入成功
  }
  else return FALSE;
} // Insert BST
```

9.2.1 二叉排序树——查找分析

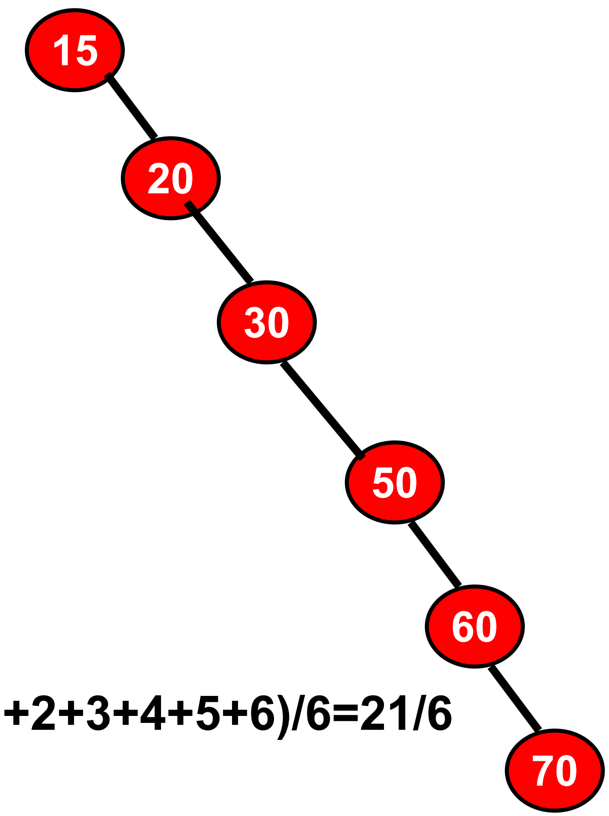
对于每一棵特定的二叉排序树，均可按照平均查找长度的定义来求它的 ASL 值，显然，由值相同的 n 个关键字，构造所得的不同形态的各棵二叉排序树的平均查找长度的值不同，甚至可能差别很大。

9.2.1 二叉排序树——查找分析

- 平均情况分析（在成功查找的情况下）
- 例如: 下述两种情况下的成功的平均查找长度 **ASL**



ASL=(1+2+2+3+3+3)/6=14/6



ASL=(1+2+3+4+5+6)/6=21/6

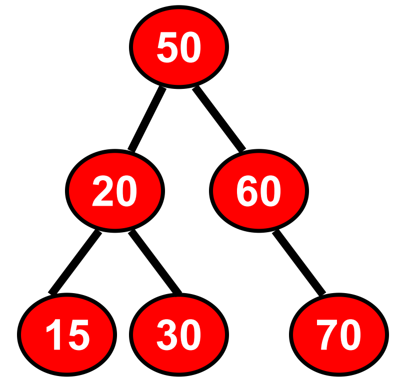
折半查找长度为n的表的判定树是唯一的，而含有n个结点的二叉排序树却不唯一。

9.2.1 二叉排序树——查找分析

- 含有n个结点的二叉排序树的平均查找长度和树的形态有关。
  - 最差的情况：当先后插入的关键字有序时，构成的二叉排序树蜕变为单支树，树的深度为n，其平均查找长度为(n+1)/2（和顺序查找相同）；
  - 最好的情况：二叉排序树的形态和折半查找的判定树相同，其平均查找长度和$\log_2 n$成正比。

——那么，平均性能如何呢？

9.2.1 二叉排序树——查找分析



- 平均情况分析（在成功查找的情况下）
- 设P(n)表示n个结点的二叉排序树的平均查找长度。
  - 二叉排序树有多种可能，它的左子树可能有不同的结点数。
  - 设 P(n, i)为它的左子树的结点个数为i 时的平均查找长度。
- 例子: 右图的结点个数为 n = 6 且 i = 3; 则

$$P(n,i)= P(6, 3) = [\ 1+ (\ P(3) + 1) * 3 + (\ P(2) + 1) * 2\ ] / 6$$

$$= [\ 1+ (\ 17/9 + 1) * 3 + (\ 3/2 + 1) * 2\ ] / 6$$

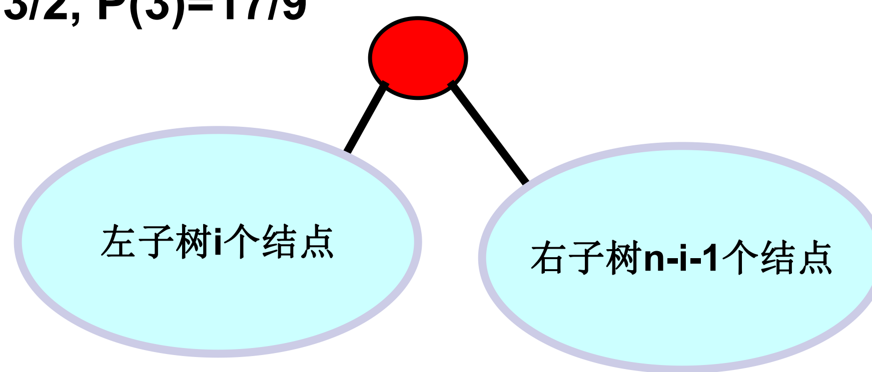
已知：P(0)=0, P(1)=1, P(2)＝(1+2)/ 2 = 3/2, P(3)=17/9

这样可以计算出P(6, 3),

类似地可以计算P(6, 0), P(6, 1), P(6, 2),

P(6, 3), P(6, 4), P(6, 5)。

则P(6)等于以上可能值的平均值。



9.2.1 二叉排序树——查找分析

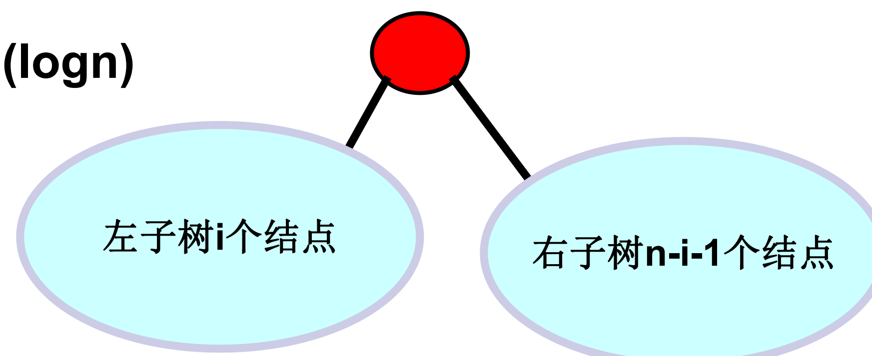
- 平均情况分析（在成功查找的情况下）
- 设P(n)表示n个结点的二叉排序树的
  - 二叉排序树有多种可能，它的左子树
  - 设 P(n, i)为它的左子树的结点个数为i 时的 查找长度。
- 在一般情况下，

$$P(n,i) = [\,1+ (\,\underline{P(i) + 1}\,) * i + (\,\underline{P(n-i-1) + 1}\,) * (n-i-1)\,] / n$$

$$P(n) = \sum^{n-1} P(n,i) / n = O(\log n)$$

P(n-i-1) + 1为查找右子树中每个关键字时所用比较次数的平均值

P(i) + 1为查找左子树中每个关键字时所用比较次数的平均值



9.2.1 二叉排序树——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排序树上某个结点之后，仍然保持二叉排序树的特性。
- 可分三种情况讨论：

（1）被删除的结点是叶子;

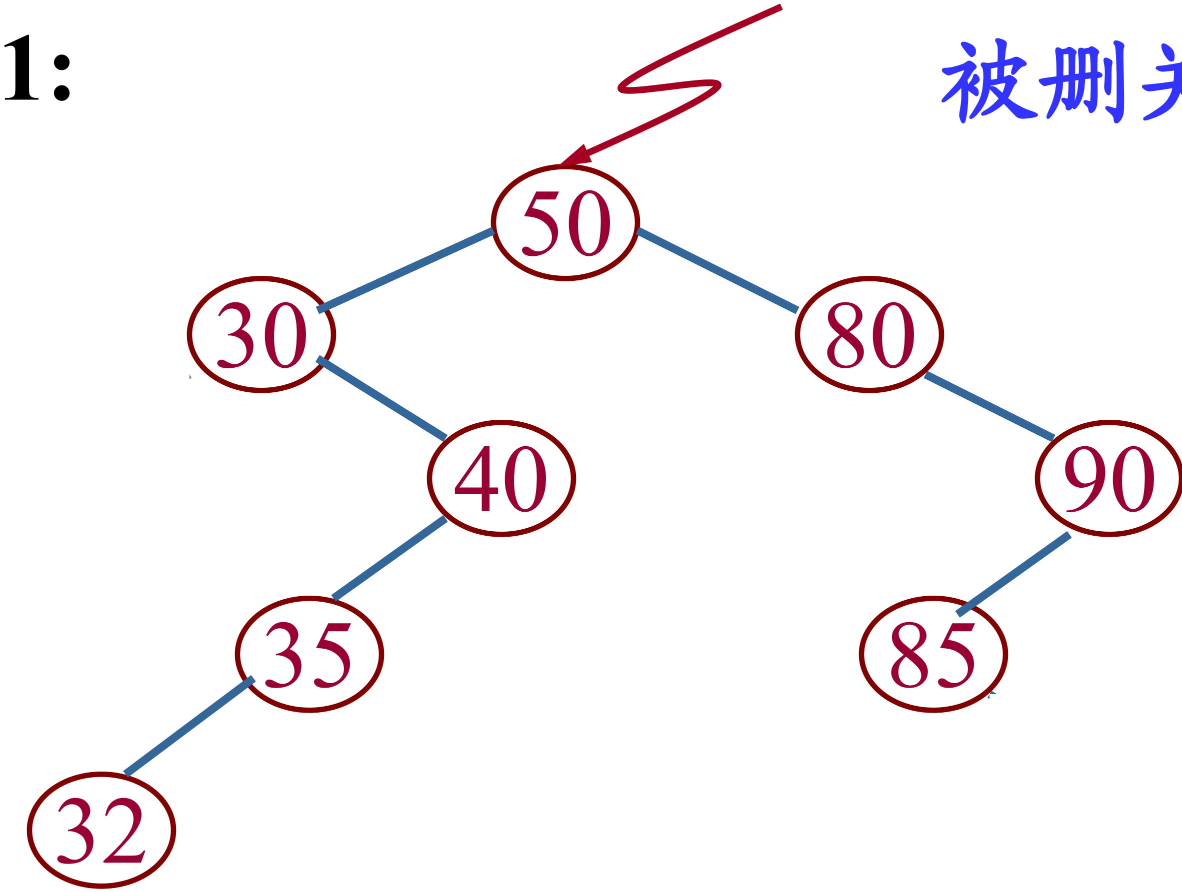
（2）被删除的结点只有左子树或者只有右子树;

（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结点

例1:

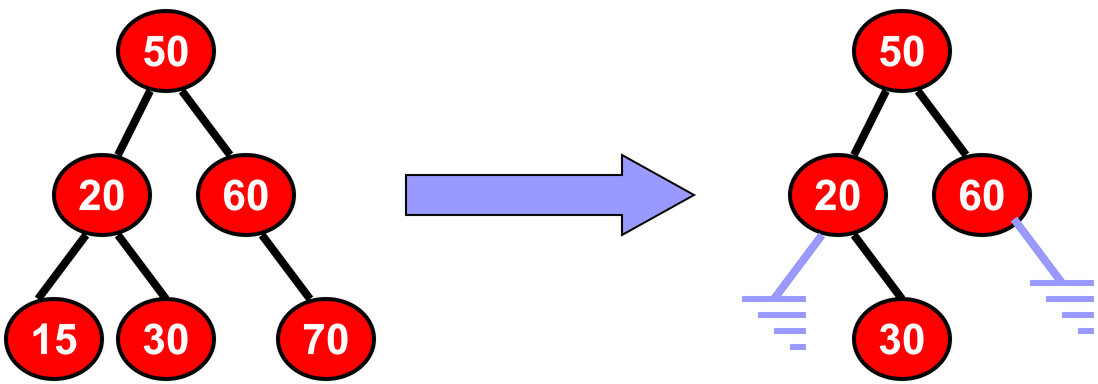
被删关键字 = 88



- 直接删除，其双亲结点中相应指针域的值改为“空”

（1）被删除的结点是叶子结点

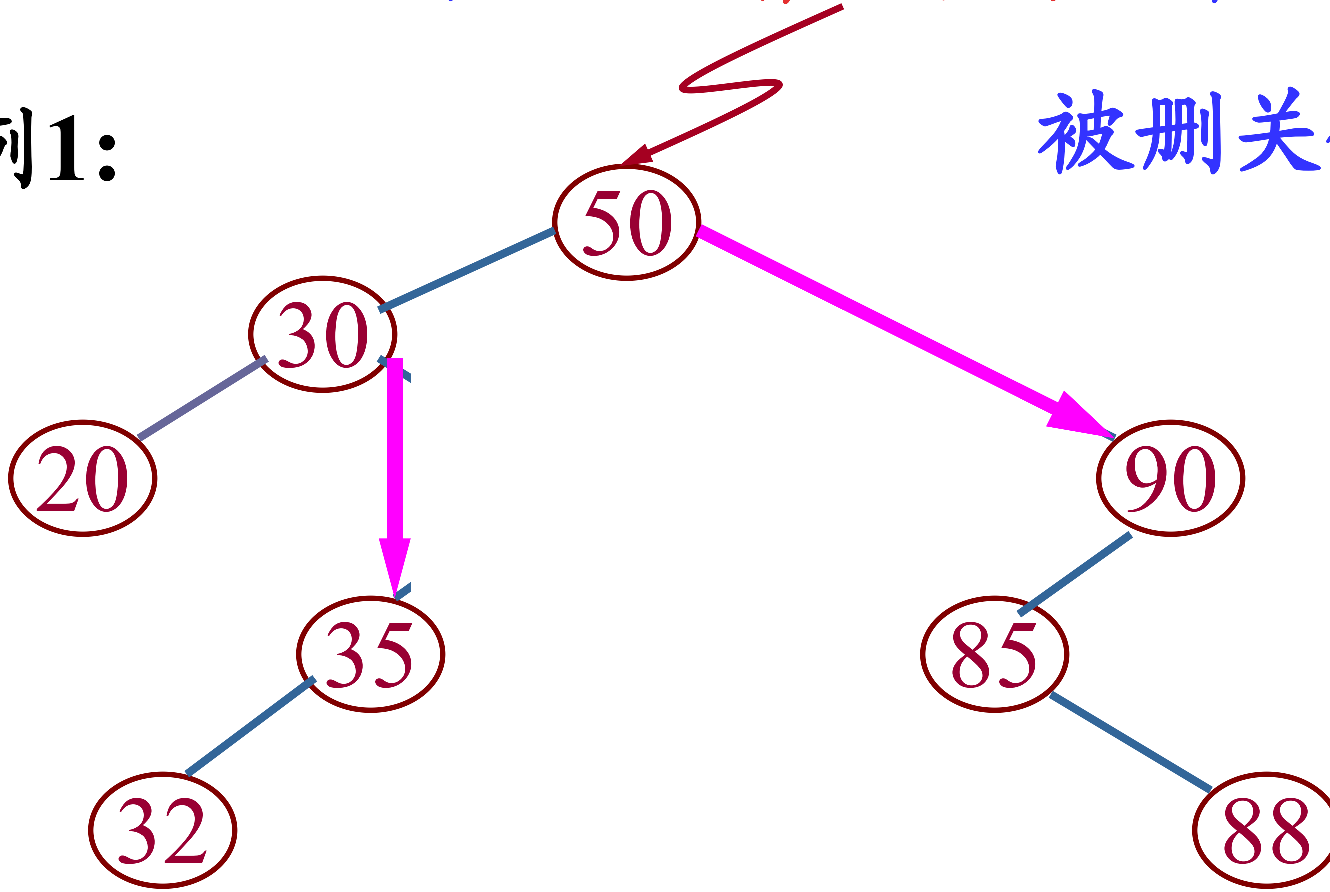
- 例2：删除数据域为 **15**、**70** 的结点。



（2）被删除的结点只有左子树或者只有右子树

例1:

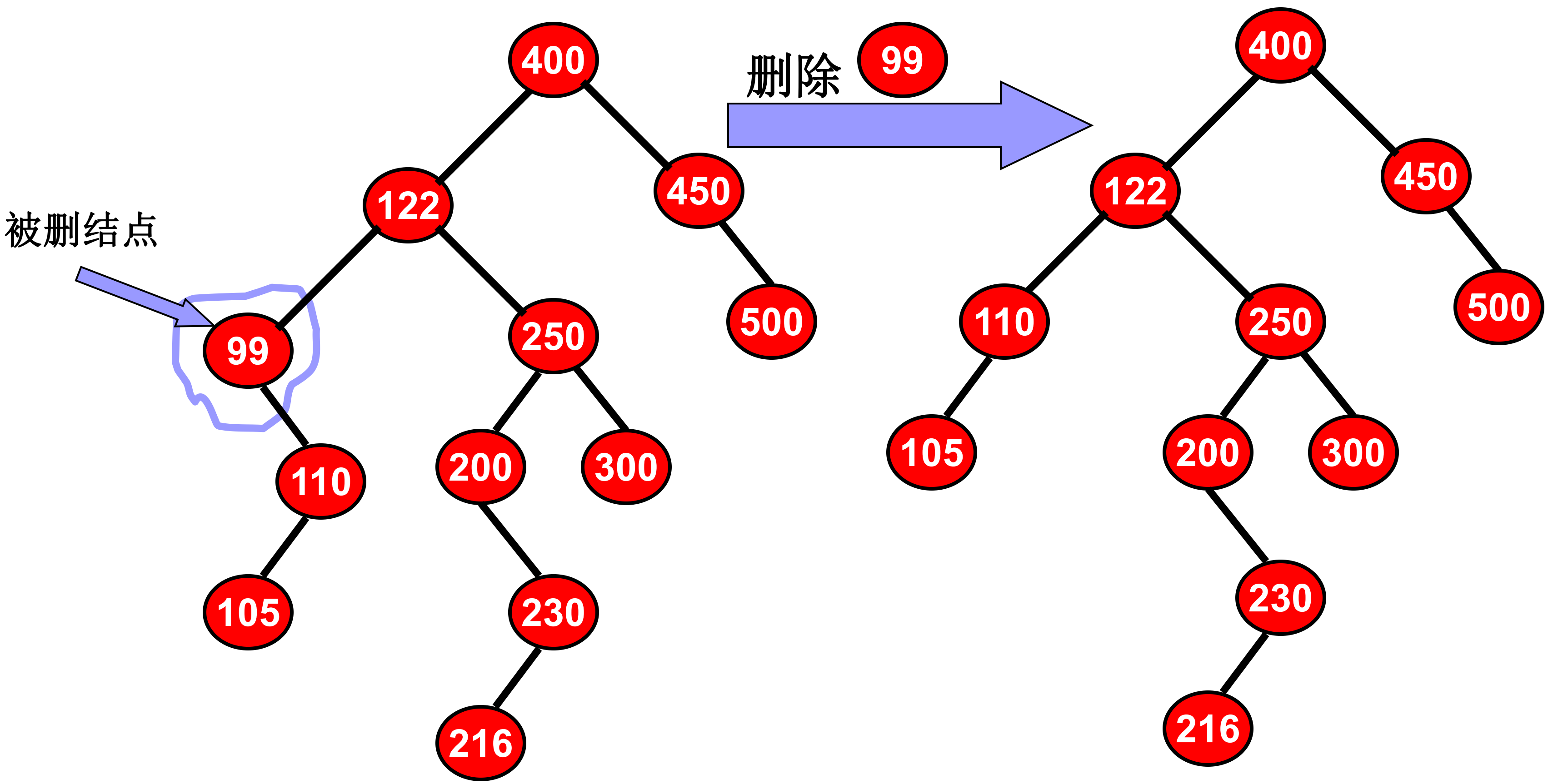
被删关键字 = 80



■ 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”。

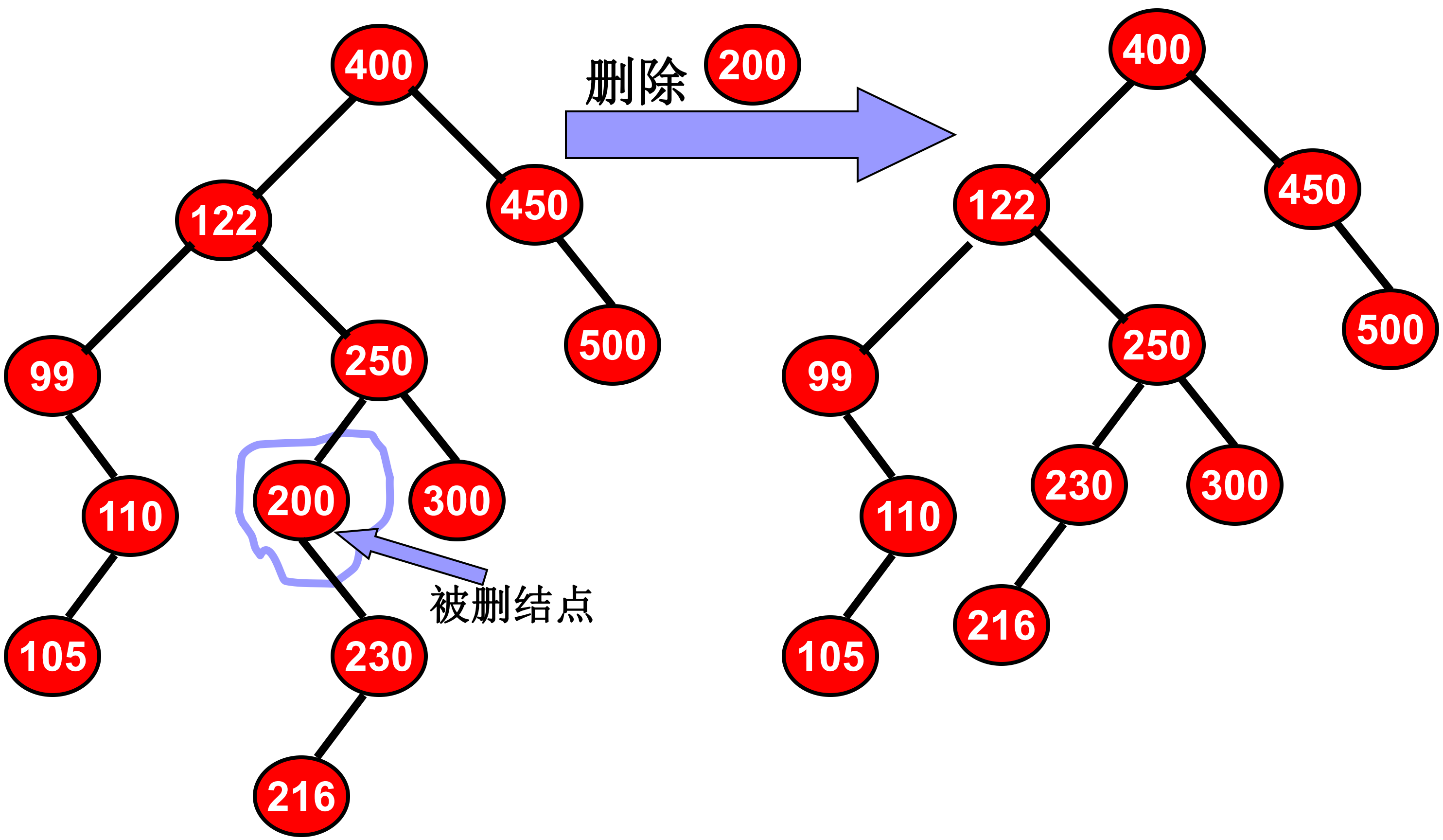
（2）被删除的结点只有左子树或者只有右子树

例2：删除结点的数据域为 99 、200 的结点。



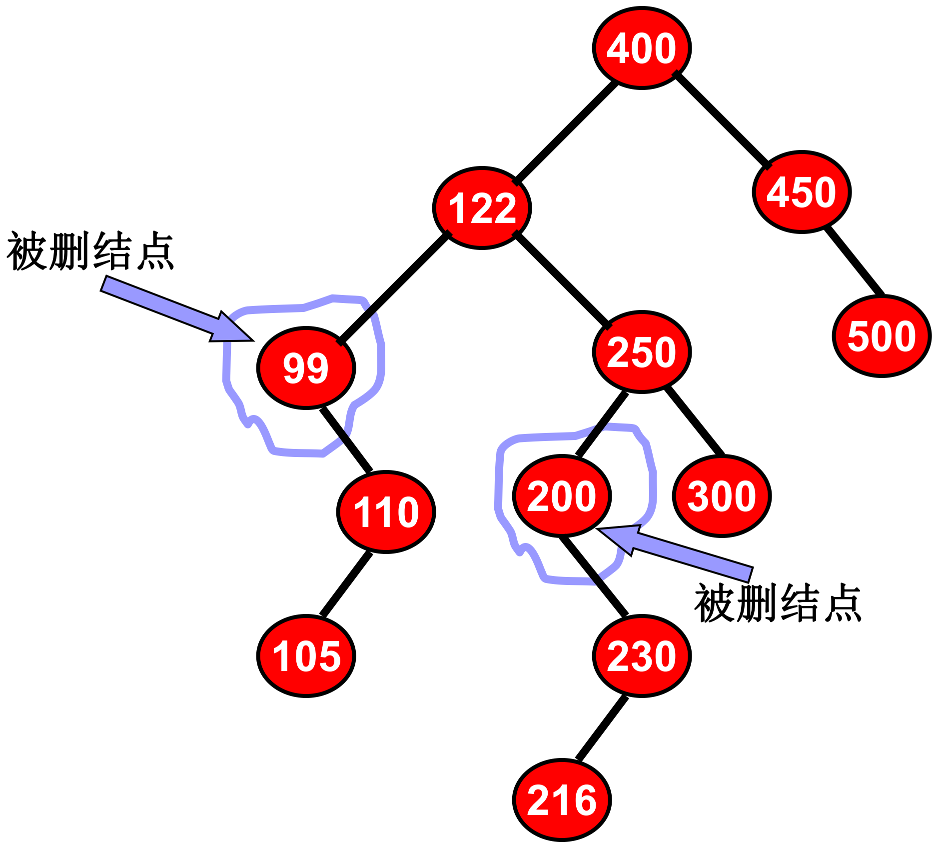
（2）被删除的结点只有左子树或者只有右子树

例2：删除结点的数据域为 99 、200 的结点。



（2）被删除的结点只有左子树或者只有右子树

例2：删除结点的数据域为 99 、200 的结点。



结论：

- 将被删结点的另一儿子作为它的父亲结点的儿子，究竟是作为左儿子还是右儿子依原替身结点和其父亲结点的关系而定。
- 释放被删结点的空间。

（3）被删除的结点既有左子树，也有右子树

- 通常的做法：选取“替身”取代被删结点。
- 谁有资格充当该替身?
- 要点：维持二叉分类树的特性不变。在中序周游中紧靠着被删结点的结点才有资格作为“替身”。

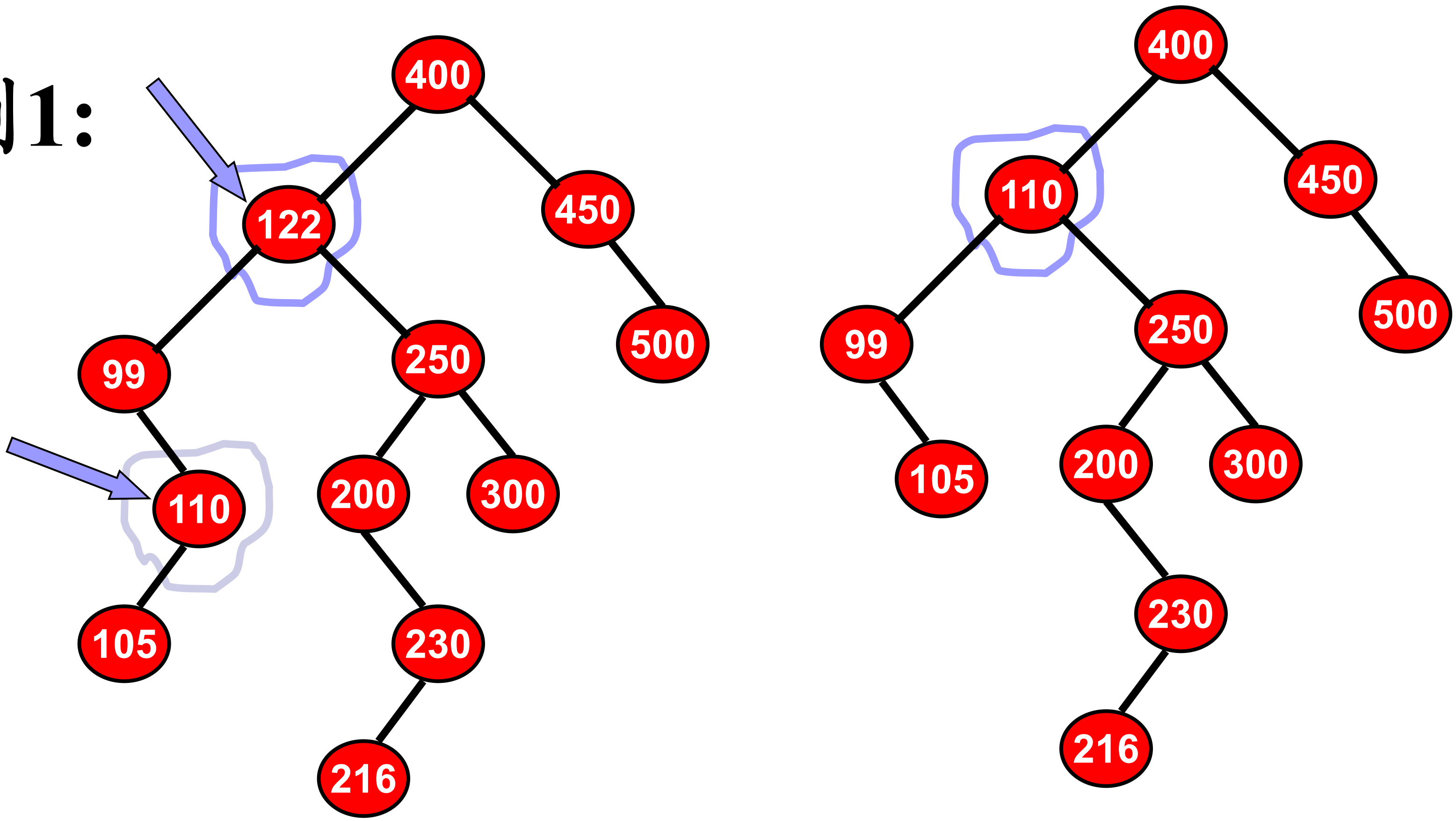
（1）左子树中最大的结点(被删结点的左子树中的最右的结点，其右儿子指针域为空)

或

（2）右子树中最小的结点(被删结点的右子树中的最左的结点，其左儿子指针域为空)。

（3）被删除的结点既有左子树，也有右子树

例1:

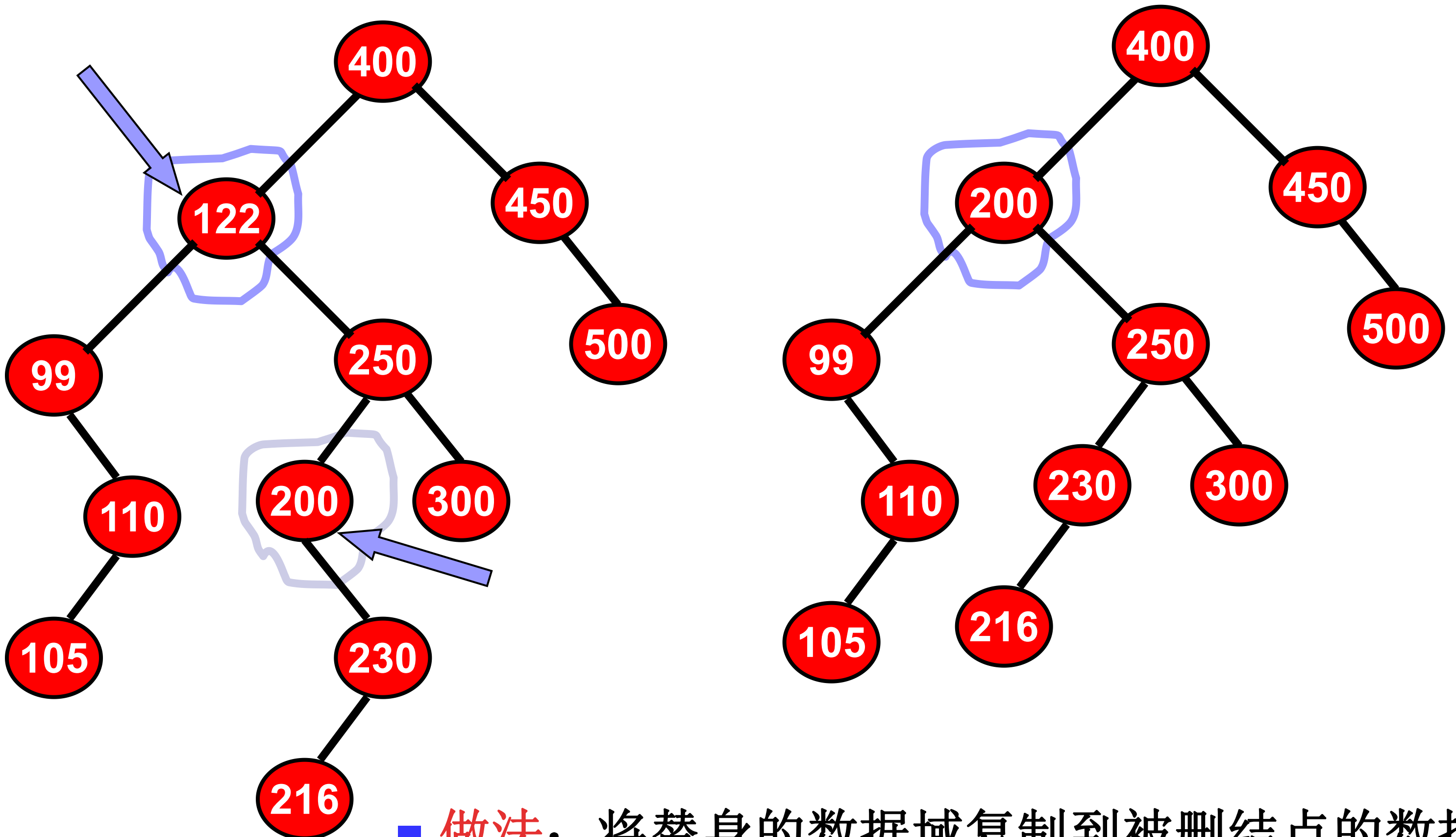


- 做法：将替身的数据域复制到被删结点的数据域。

  将结点110的左儿子作为110的父结点99 的右儿子。

（3）被删除的结点既有左子树，也有右子树

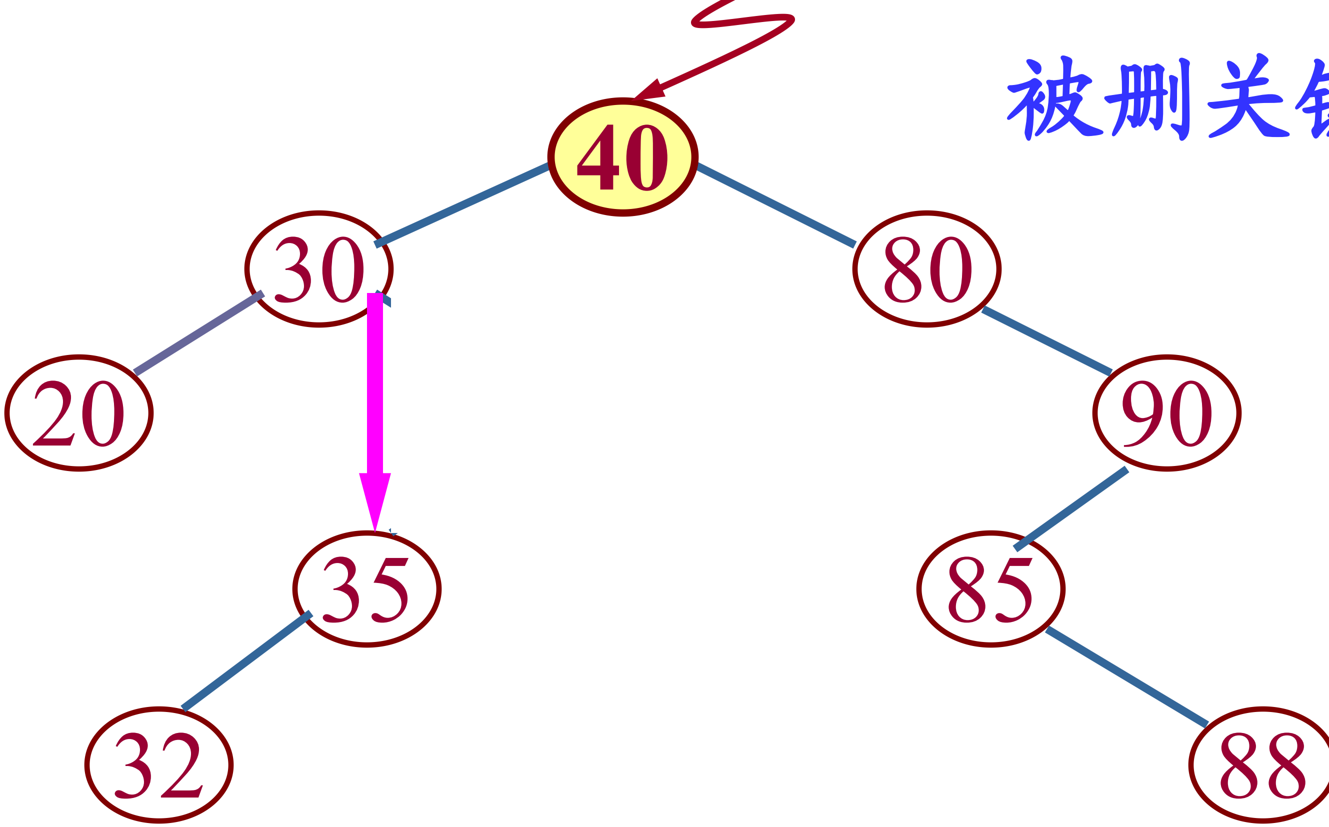
例2:

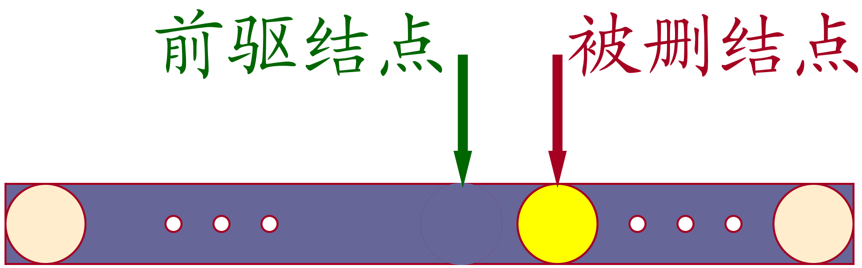


- 做法：将替身的数据域复制到被删结点的数据域。将结点200的右儿子作为 200的父结点的左儿子。
- 注意：结点200左儿子必为空，结点 200的父结点为250

（3）被删除的结点既有左子树，也有右子树

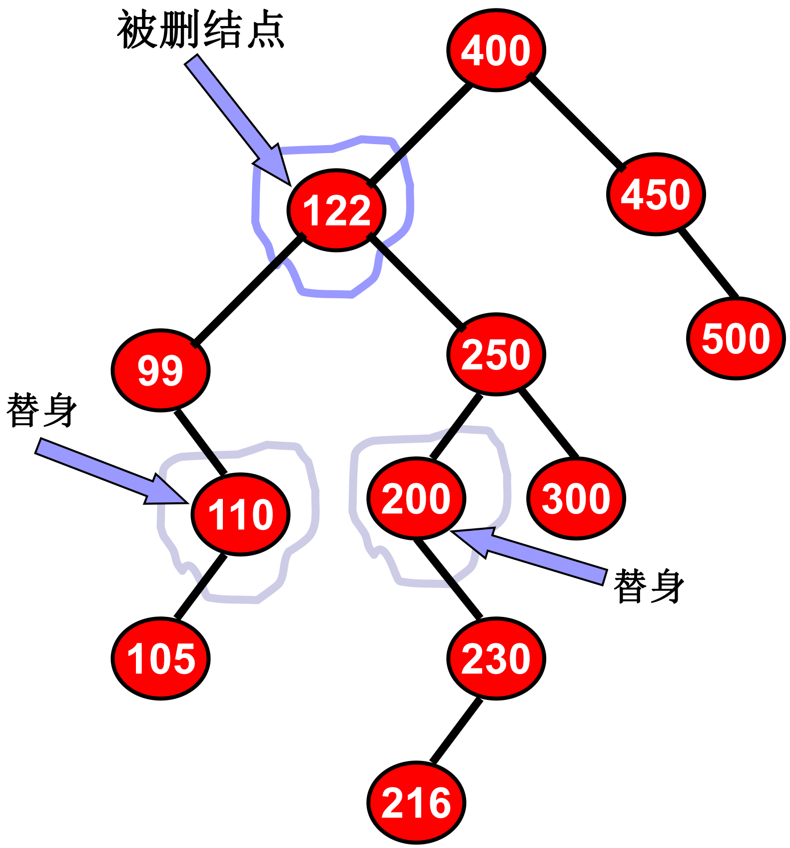
被删关键字＝50





以其前驱替代之，然后再删除该前驱结点

（3）被删除的结点既有左子树，也有右子树

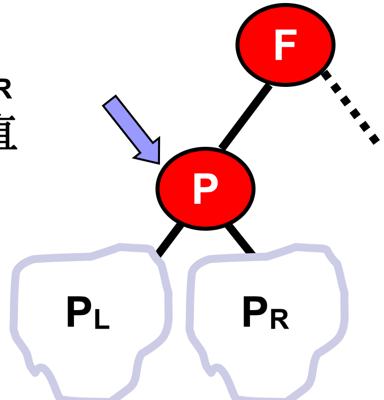


结论：

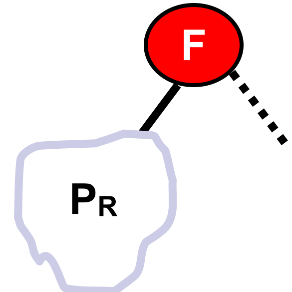
- 先将替身的数据域复制到被删结点；
- 将原替身的另一儿子作为它的父亲结点的儿子，究竟是作为左儿子还是右儿子依原替身结点和其父亲结点的关系而定。
- 释放原替身结点的空间。

9.2.1 二叉排序树——删除算法

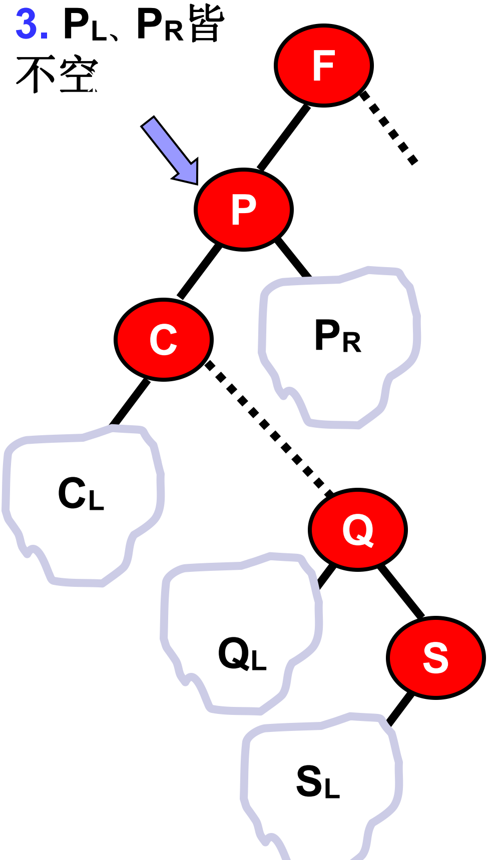
1. P_L、P_R 皆空， 直接删除 。



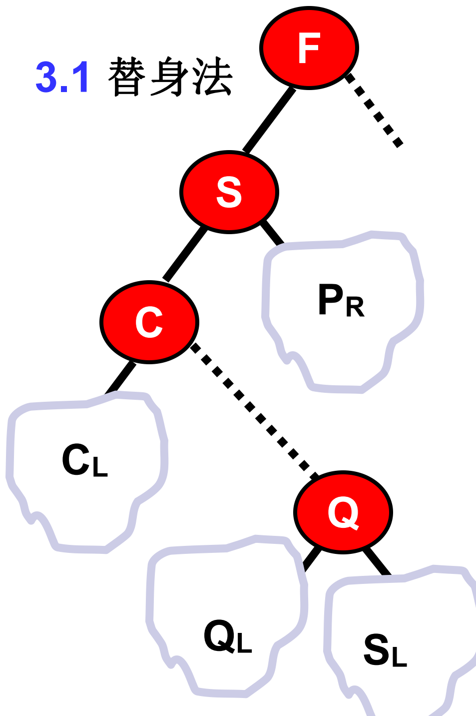
2. PL或PR为空。 比如P_L为空，，则直接接上P_R 。



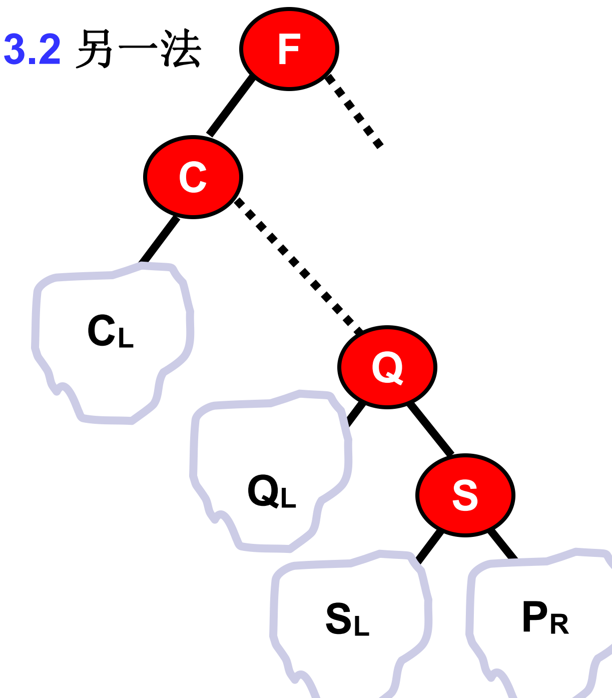
3. P_L、P_R皆不空



3.1 替身法



3.2 另一法



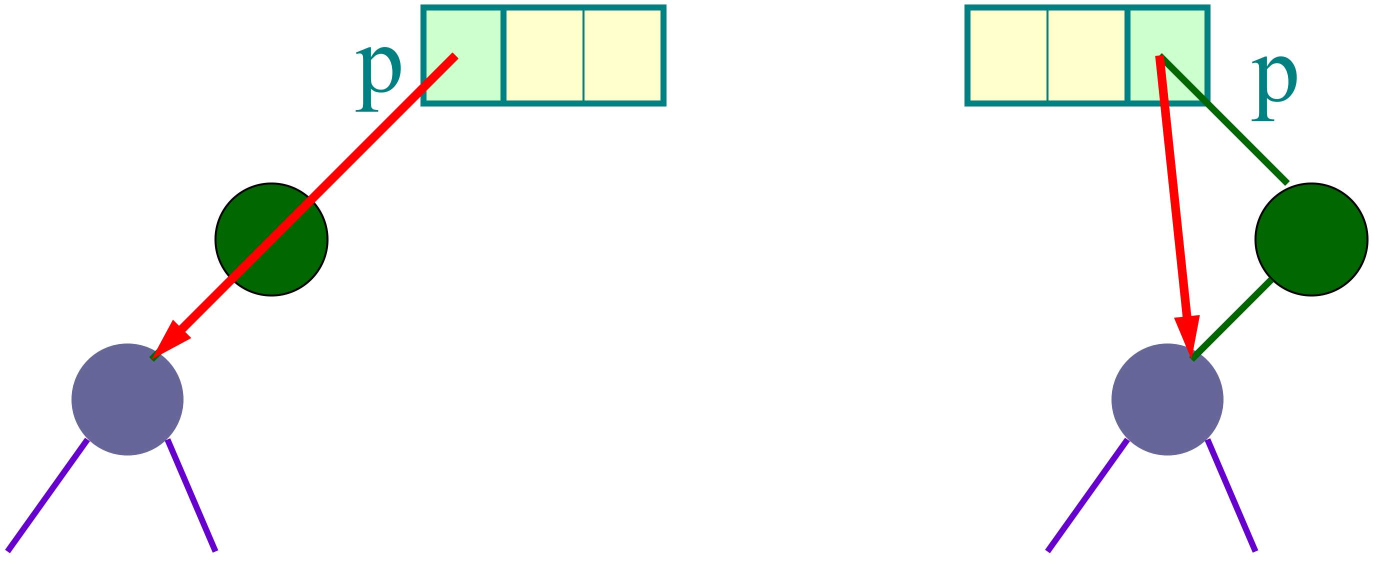
9.2.1 二叉排序树——删除算法

```
Status DeleteBST ( BiTree &T，KeyType key )
// 若二叉排序树 T 中存在关键字为 key 的结点时，则删除该结点，
// 并返回 TRUE；否则返回 FALSE。
 { if （( !T） return FALSE;
                    // 二叉分类树 T 中不存在关键字为 key 的结点
   else if ( EQ( key, T ->data. key ) ) return Delete (T);
                              // 存在关键字为 key 的结点，进行删除
        else if ( LT( key , T ->data. key ) ) return DeleteBST ( T -> lchild, key );
                              // 继续在左子树中进行查找
             else return DeleteBST ( T -> rchild, key );
                              // 继续在右子树中进行查找
   return TRUE;
 } // DeleteBST
```

```
Status Delete ( BiTree &p )
// 在二叉排序树中删除地址为 p 的结点，并重接它的左子树或右子树，
// 保持二叉排序树的性质不变。
{ if  ( ! p-> rchild )  { q = p; p = p->lchild ; free(q); }    //右子树为空
  else if ( ! p-> lchild ) { q = p; p = p->rchild ; free(q); }   //左子树为空
        else    //左右子树均不空，使用替身法，找到左子树的最右结点
         { q = p; s = p-> lchild;
           while ( s->rchild ) { q = s; s = s->rchild; }   // s 指向被删结点的前驱
           p->data = s->data;
           if ( q != p ) q->rchild = s->lchild;   // 重接*q的左子树
           else q->lchild = s->lchild;       // 重接*q的左子树
           delete s;       }
  return TRUE;
 } // Delete
```

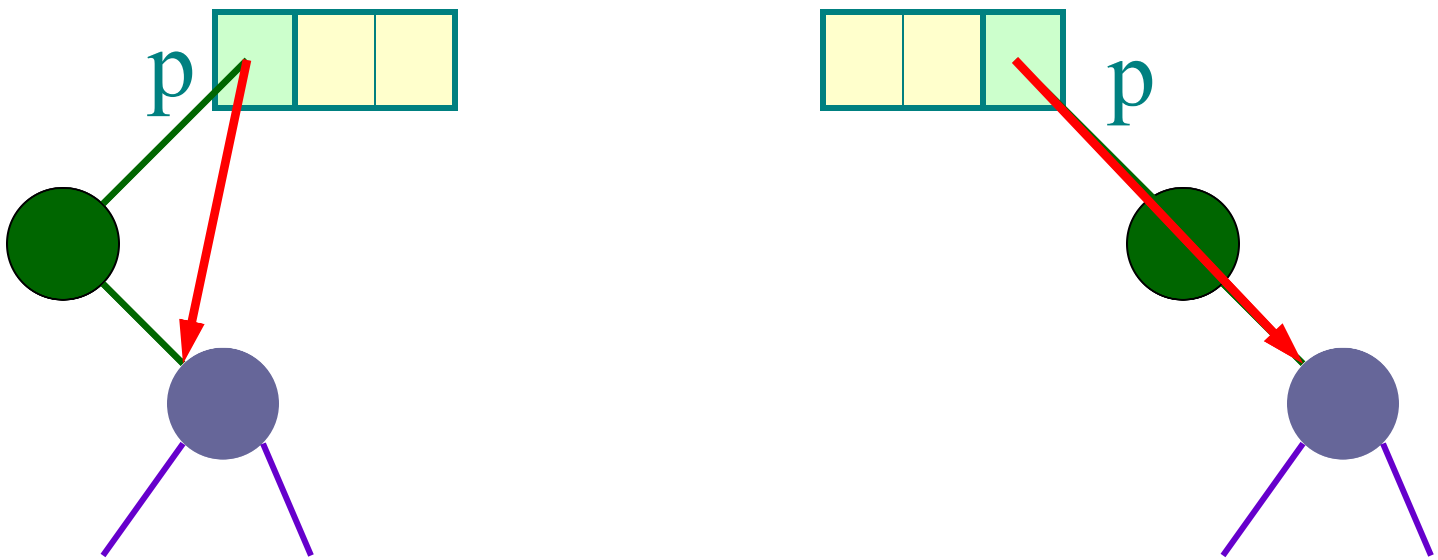
// 右子树为空树则只需重接它的左子树

```
q = p;  p = p->lchild;  free(q);
```



// 左子树为空树只需重接它的右子树

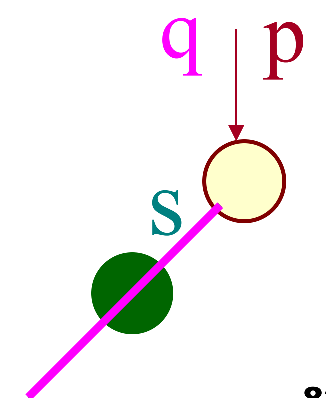
q = p; p = p->rchild; free(q);



// 左右子树均不空

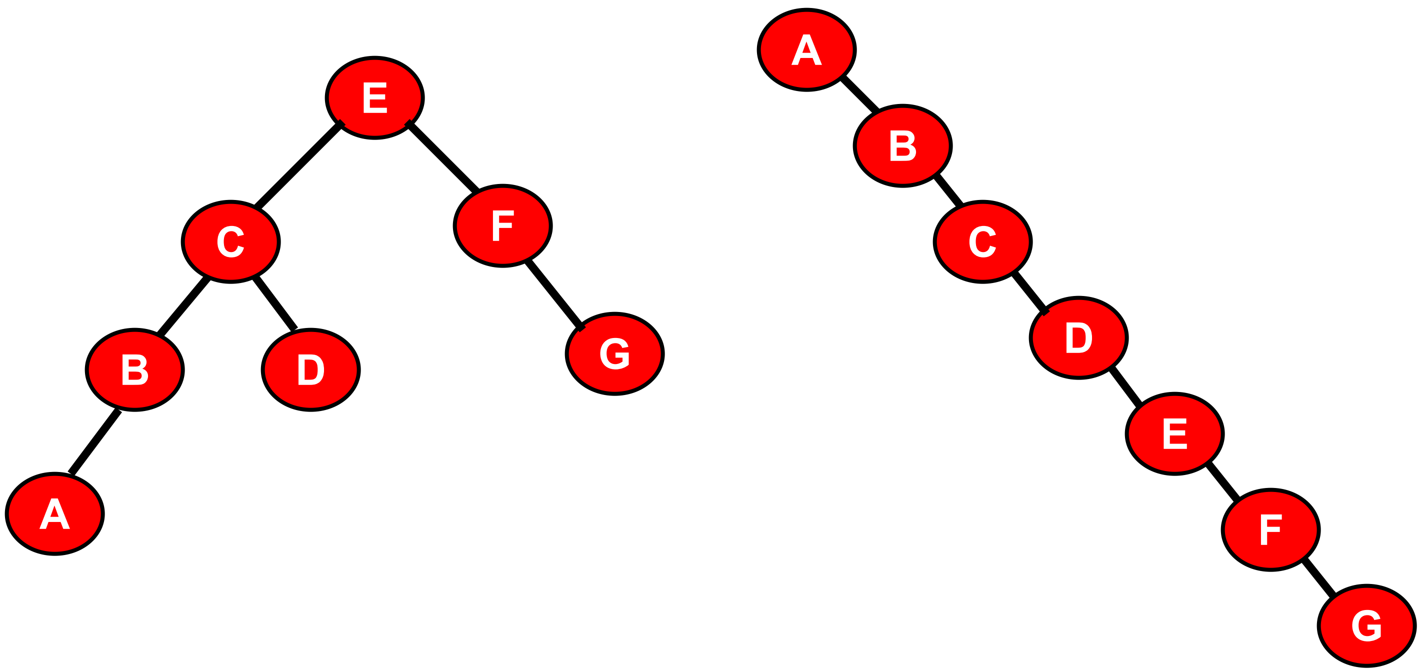
```
q = p;  s = p->lchild;
while (!s->rchild) { q = s;  s = s->rchild; }
                // s 指向被删结点的前驱

p->data = s->data;
if (q != p )  q->rchild = s->lchild;
else  q->lchild = s->lchild;
                // 重接*q的左子树
free(s);
```



9.2.2 平衡二叉树(AVL树)

- 起因：提高查找速度，避免最坏情况出现。如右图情况的出现。
- 虽然丰满树的树型最好，但构造困难。常使用平衡树。



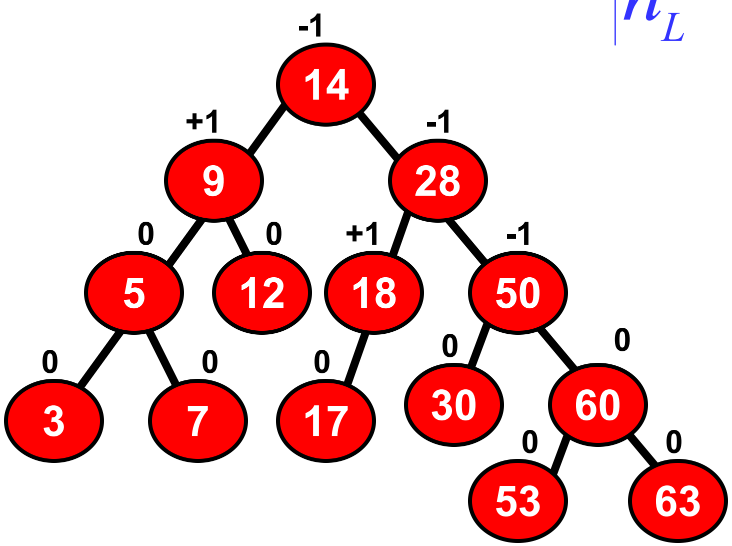
9.2.2 平衡二叉树(AVL树)

- 何谓“二叉平衡树”？
- 如何构造“二叉平衡树”
- 二叉平衡树的查找性能分析

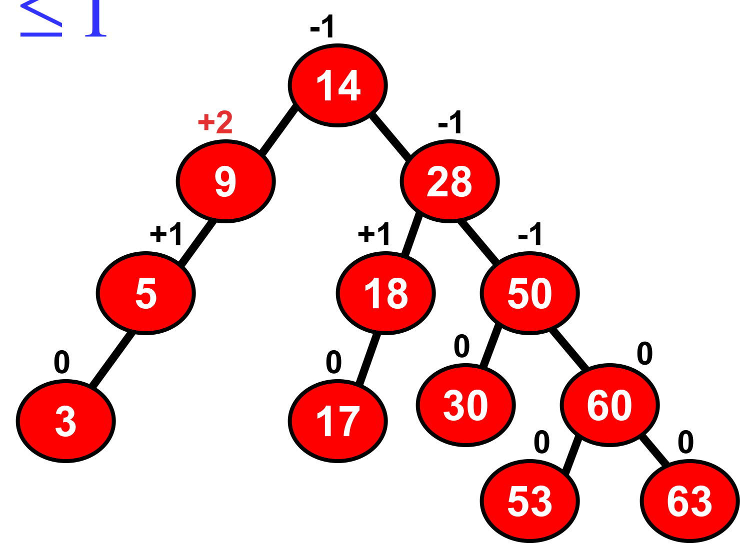
9.2.2 平衡二叉树(AVL树)——定义

- 平衡因子（平衡度）：结点的左子树的高度 － 右子树的高度。
- 平衡二叉树：每个结点的平衡因子都为 ＋1、－1、0 的二叉树。或者说每个结点的左右子树的高度最多差1 的二叉树。

$$|h_L - h_R| \le 1$$



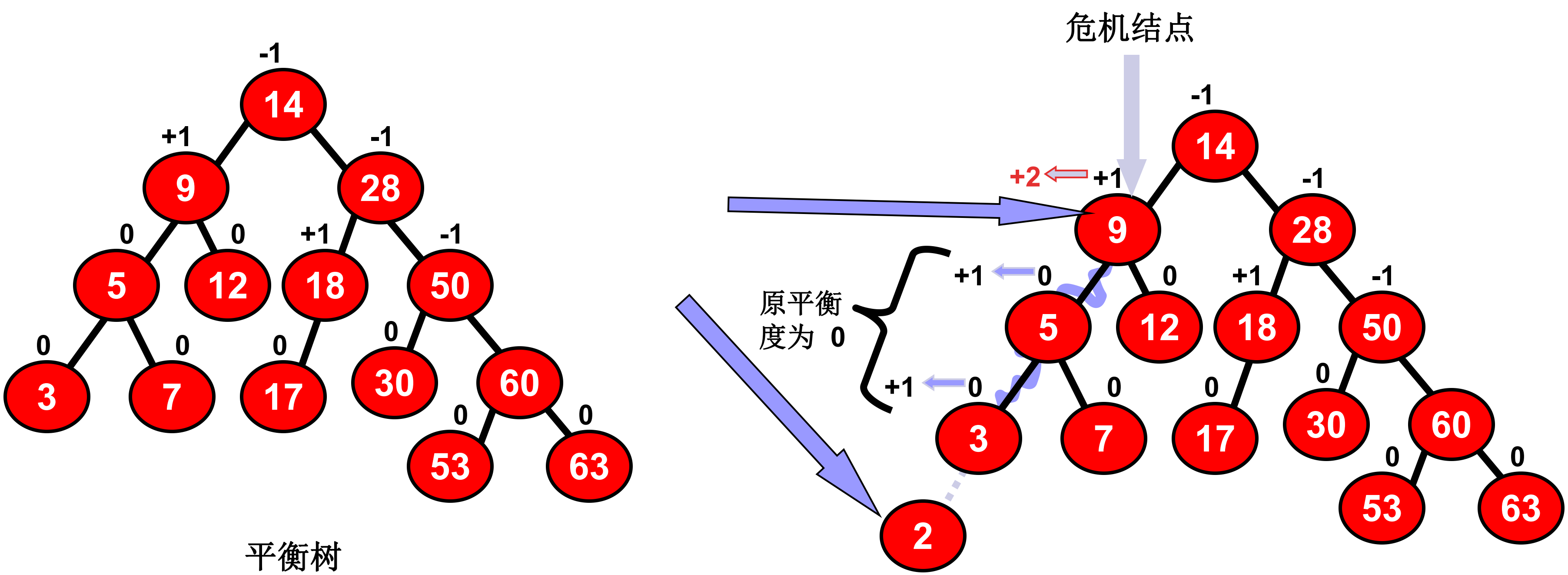
是平衡树



不是平衡树

9.2.2 平衡二叉树(AVL树)——如何构造?

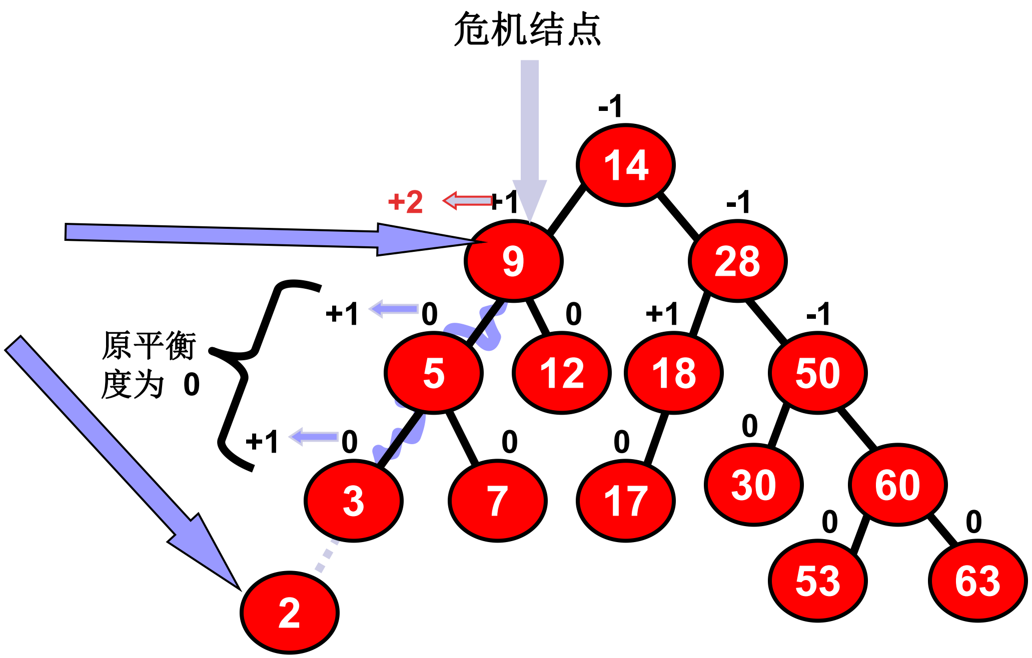
- 平衡树的 Adelson 算法的本质特点：插入之后仍应保持平衡二叉树的性质不变。
- 以插入为例：在左图所示的平衡树中插入数据域为 2 的结点。



如何用最简单、最有效的办法保持平衡二叉树的性质不变?

9.2.2 平衡二叉树(AVL树)——如何构造?

- 如何用最简单、最有效的办法保持平衡二叉树的性质不变?

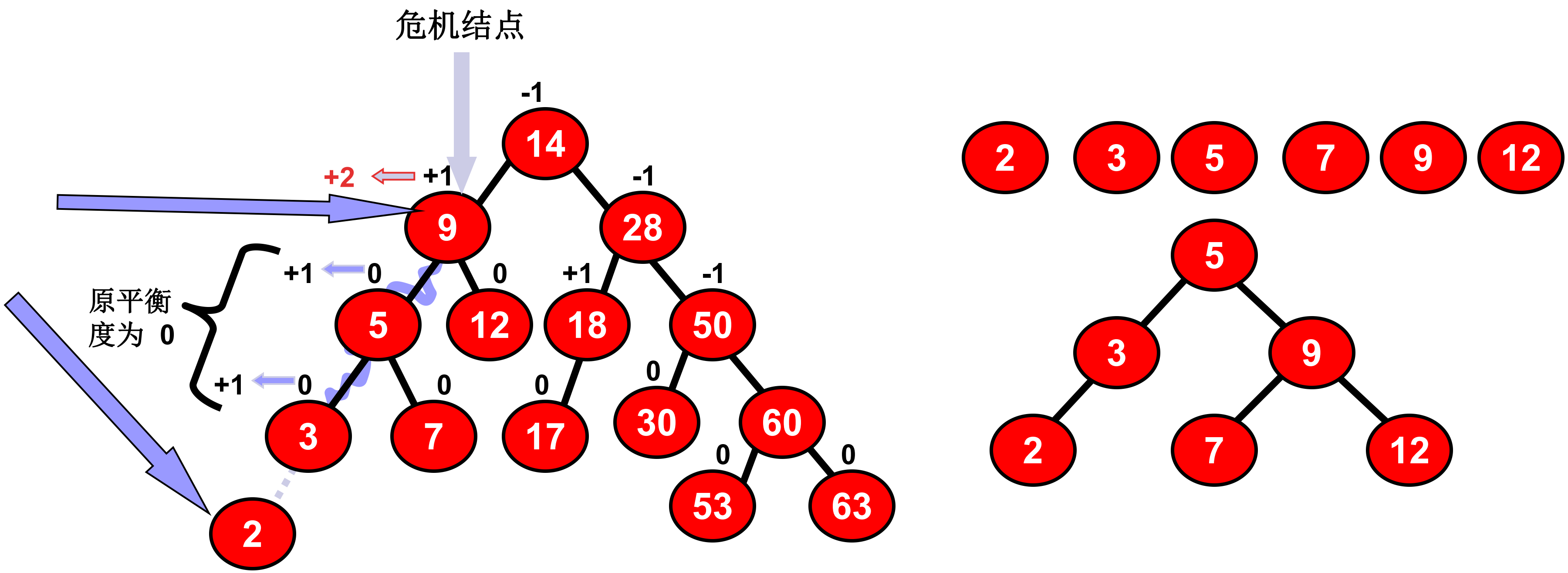


解决方案:

- 不涉及到危机结点的父亲结点，即以危机结点为根的子树的高度应保持与新结点插入前一样。
- 新结点插入后，找到第一个平衡度不为0的结点（如左图结点 9）即可。新插入的结点和第一个平衡度不为 0 的结点（也可能是危机结点）之间的结点，其平衡度皆为 0。
- 在调整中，仅调整那些在平衡度变化的路径上的结点（如：3，5，9），其它结点不予调整。
- 仍应保持分类二叉树的性质不变。

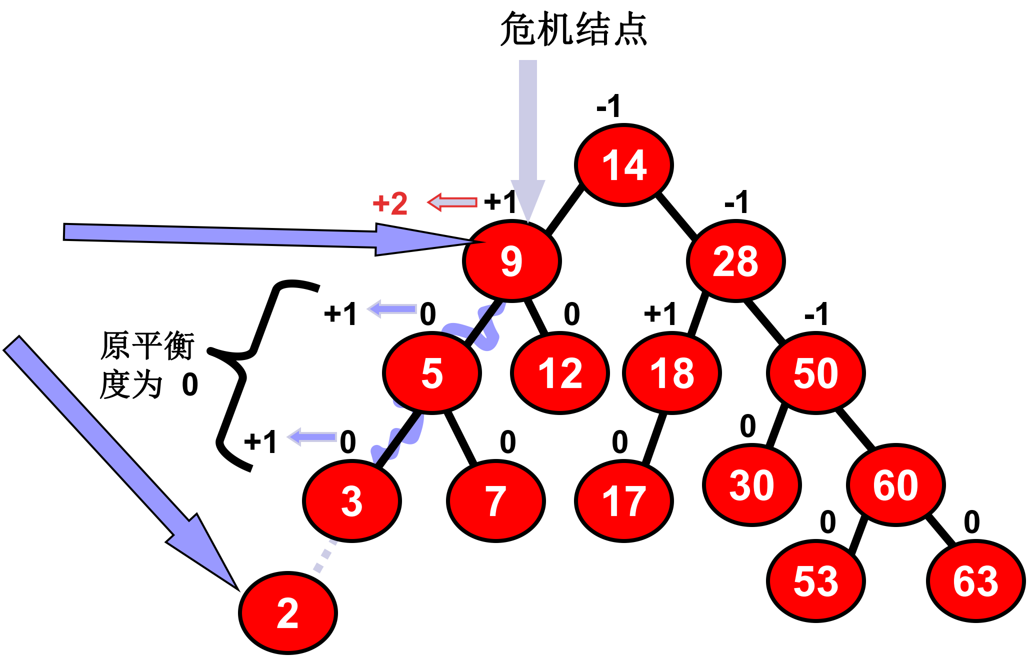
9.2.2 平衡二叉树(AVL树)——如何构造?

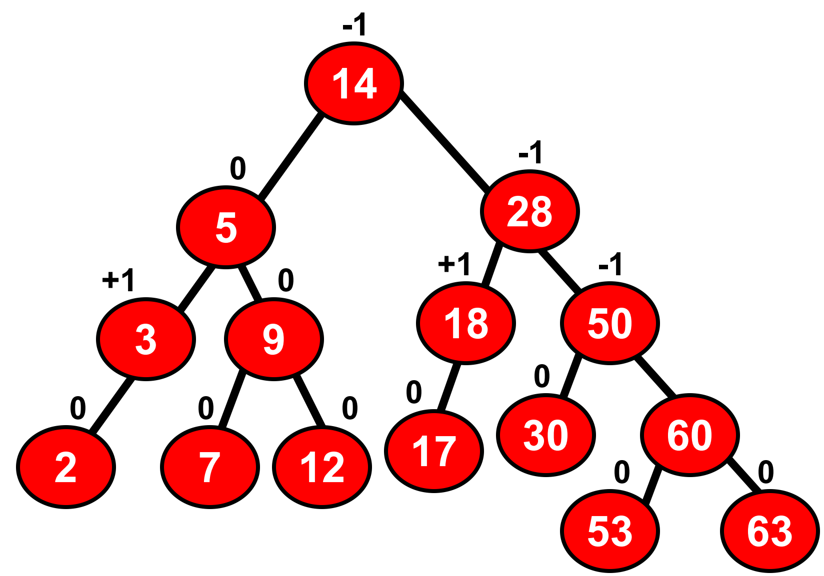
- 如何用最简单、最有效的办法保持平衡二叉树的性质不变?



9.2.2 平衡二叉树(AVL树)——如何构造?

- 关键：将导致出现危机结点的情况全部分析清除，就可以使得平衡二叉树的性质保持不变！！

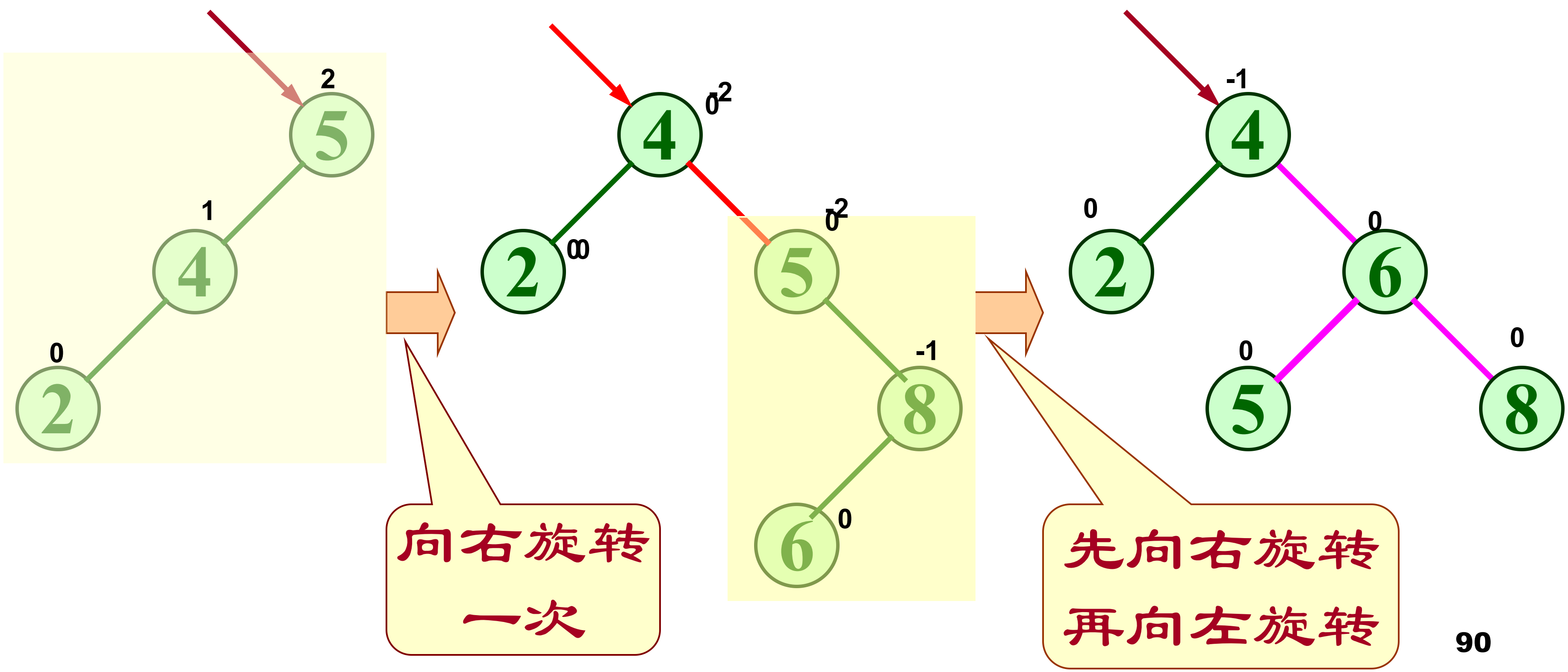


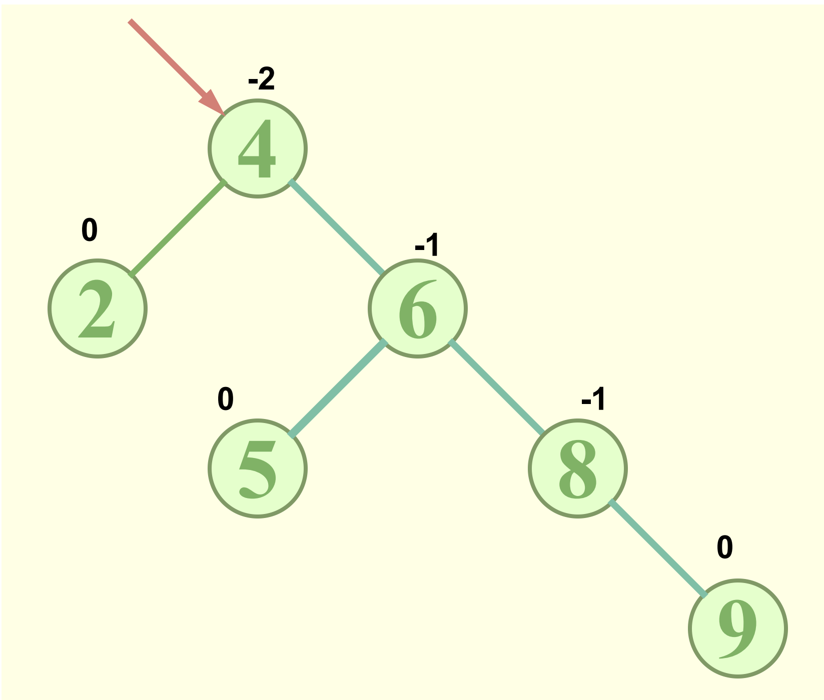


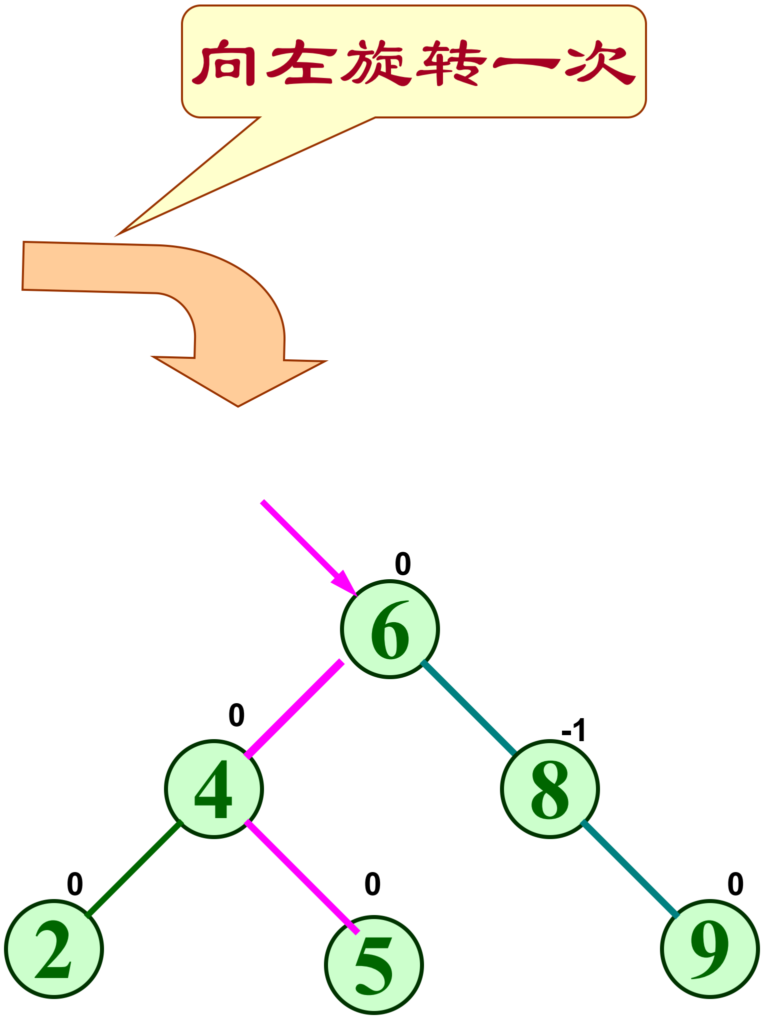
构造二叉平衡（查找）树的方法是:

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5, 4, 2, 8, 6, 9





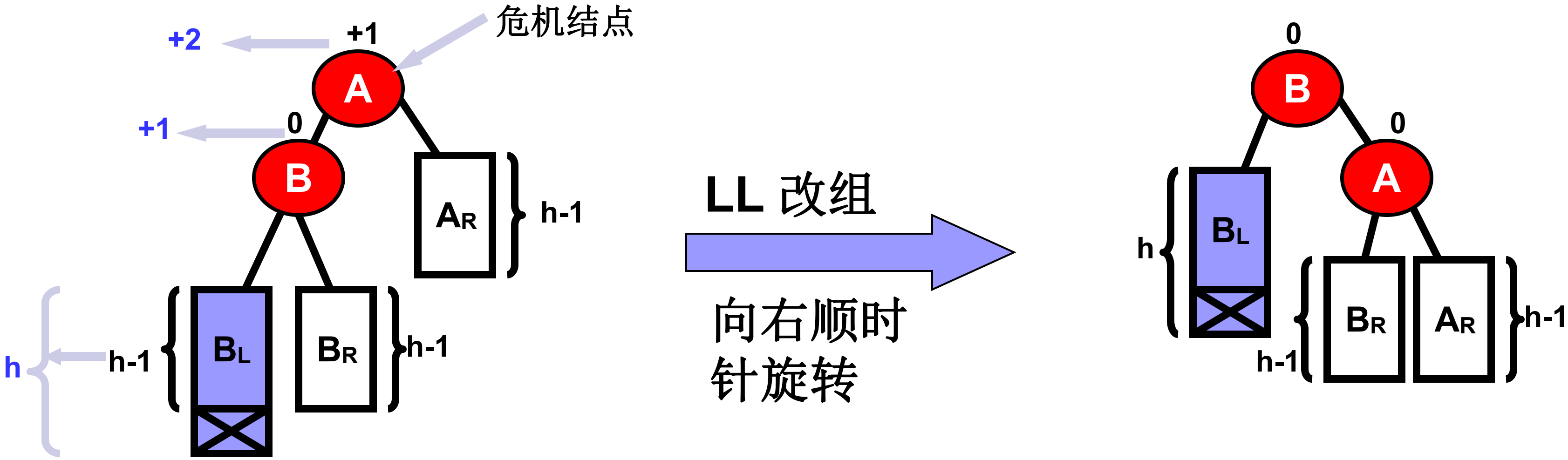


继续插入关键字 9

9.2.2 平衡二叉树(AVL树)——如何构造?

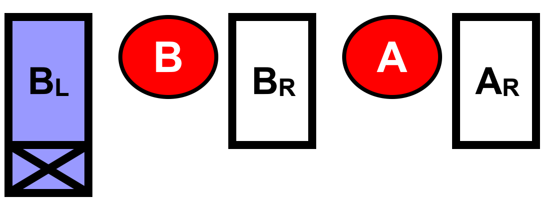
- 左改组（新插入结点出现在危机结点的左子树上进行的调整）的情况分析：

1、LL 情况：（LL：表示新插入结点在危机结点的左子树根结点的左子树上）



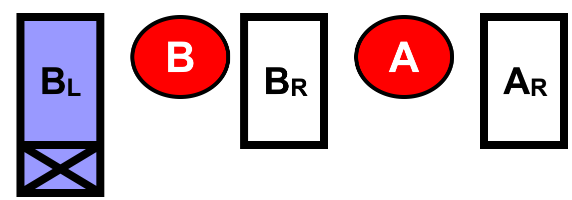
改组前：高度为 h + 1

中序序列：



改组后：高度为 h + 1

中序序列：



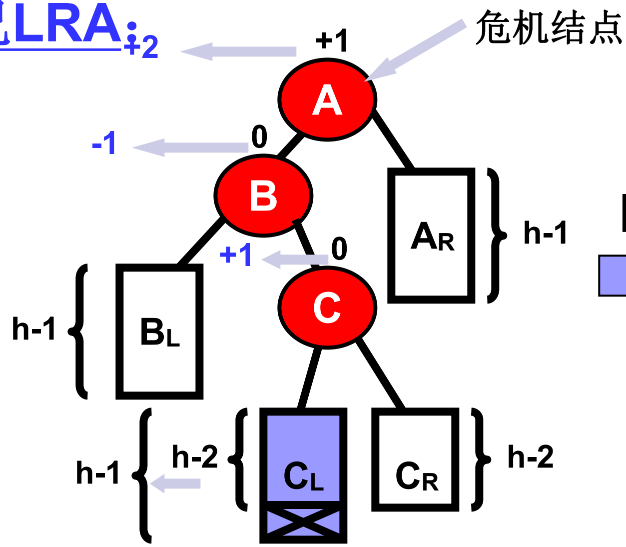
注意：改组后B、A平衡度为 0

9.2.2 平衡二叉树(AVL树)——如何构造?

- 左改组（新插入结点出现在危机结点的左子树上进行的调整）的情况分析：

2、LR 情况：（LR：表示新插入结点在危机结点的左子树根结点的右子树上）

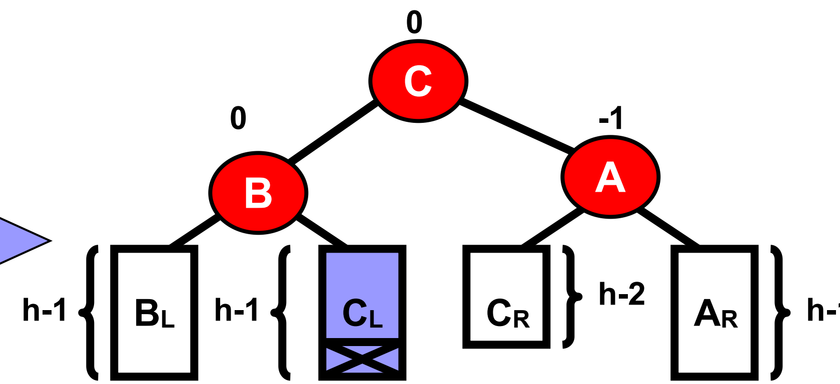
情况LRA:



LRA 改组

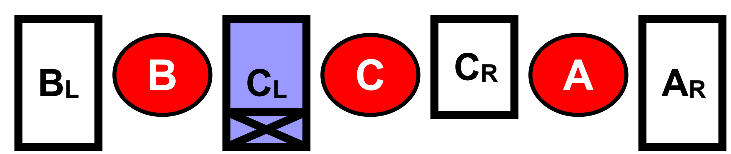
进行两次旋转（先左后右）

改组后：



高度为 **h + 1**

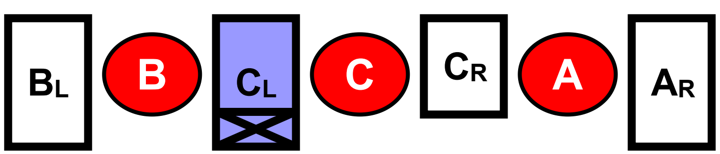
中序序列：



改组前：

高度为 **h + 1**

中序序列：

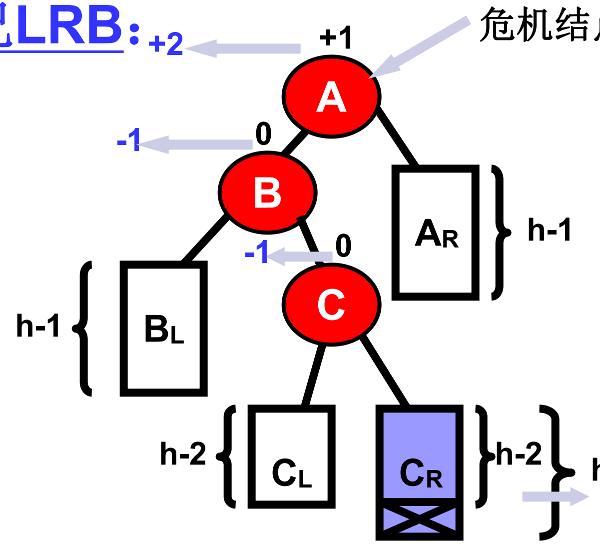


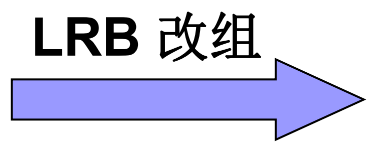
注意：改组后 **B、C、A**平衡度为 **0,0,-1**

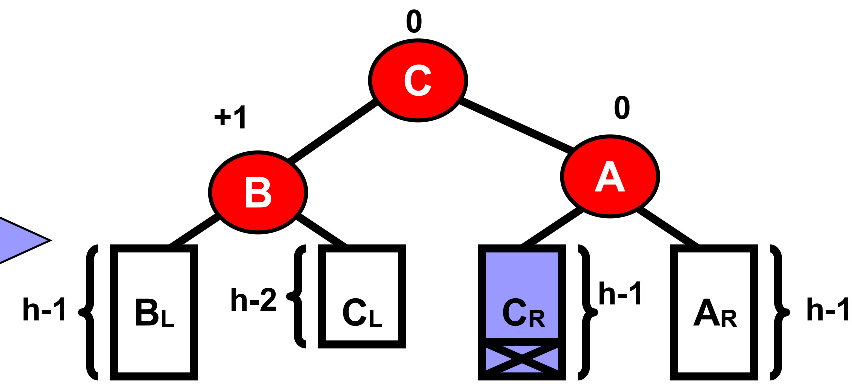
- 左改组（新插入结点出现在危机结点的左子树上进行的调整）的情况分析：

2、LR 情况：（LR：表示新插入结点在危机结点的左子树根结点的右子树上）

情况LRB：



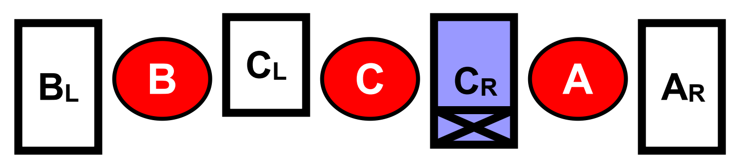




进行两次旋转（先左后右）改组后：

高度为 h + 1

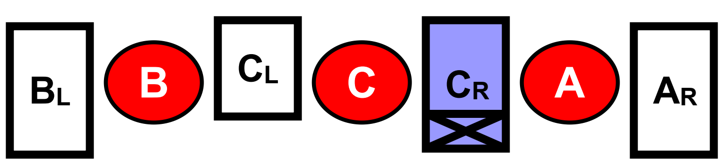
中序序列：



改组前：

高度为 h + 1

中序序列：

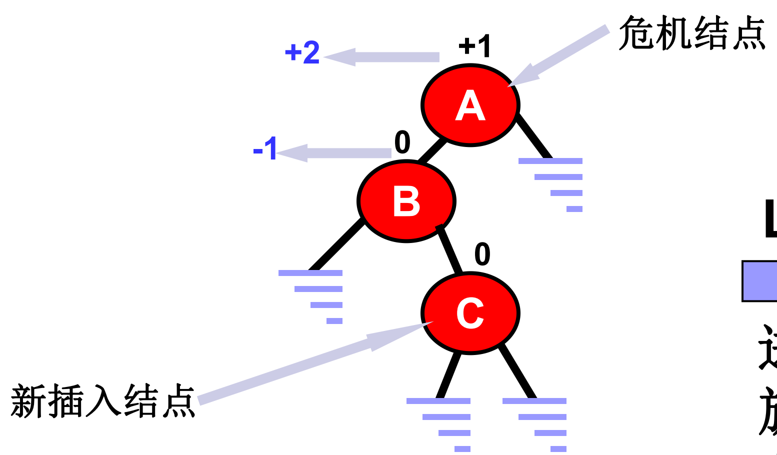


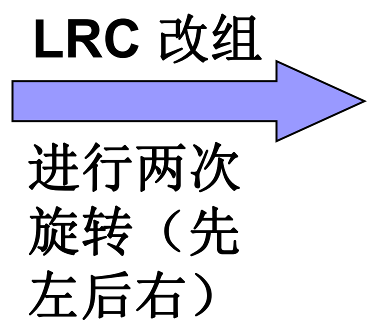
注意：改组后B、c、A 平衡度为 +1,0,0

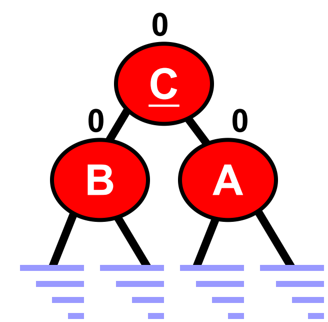
- 左改组（新插入结点出现在危机结点的左子树上进行的调整）的情况分析：

2、LR 情况：（**LR**：表示新插入结点在危机结点的左子树根结点的右子树上）

情况LRC：





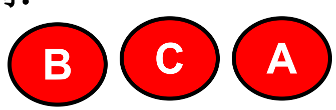


注意：改组后**B**、**C**、**A**平衡度为 **0,0,0**

改组前：

高度为 **2**

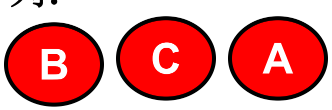
中序序列：



改组后：

高度为 **2**

中序序列：



9.2.2 平衡二叉树(AVL树)——如何构造?

- 左改组（新插入结点出现在危机结点的左子树上进行的调整）的四种情况分析：
  - 如果B的平衡度为＋1 则为 **LL型**改组；
  - 否则为 **LR型**改组：

  若C的平衡度为＋1、－1 、0；则分别为 **LRA**、**LRB**、**LRC**型改组。

9.2.2 平衡二叉树(AVL树)——如何构造?

- 右改组（新插入结点出现在危机结点的右子树上进行的调整）的情况分析：

  1、RR 情况：（RR：表示新插入结点在危机结点的右子树根结点的右子树上）

  处理图形和 LL 镜像相似

  2、RL 情况：（RL：表示新插入结点在危机结点的右子树根结点的左子树上）

  A、处理图形和 LRA 镜像相似

  B、处理图形和 LRB 镜像相似

  C、处理图形和 LRC 镜像相似

9.2.2 平衡二叉树——查找分析

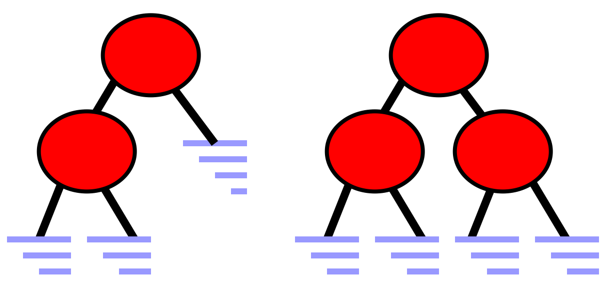
- 定理：具有N个结点的平衡树，高度h满足：

$$\log_2(N+1) \le h \le \log_\alpha(\sqrt{5}(N+1)) - 2$$

$$\text{其中}\ \alpha = (1+\sqrt{5})/2$$

- 证明：

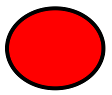
1、高度为h的二叉树最多有2^h-1个结点，平衡树的结点个数不会超过2^h-1个。即： $N \le 2^h - 1$； $N + 1 \le 2^h$ ； 所以： $\log_2(N+1) \le h$



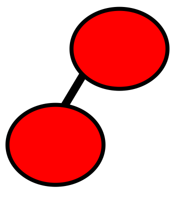
如：高度 2 的平衡树，丰满树结点个数最多为 $2^2 - 1$ 个。

9.2.2 平衡二叉树——查找分析

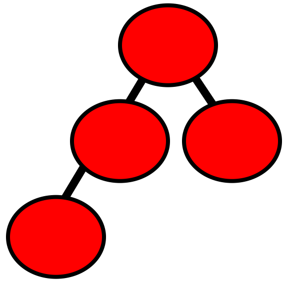
2、为了证第二步，构造一系列平衡树， T_1、 T_2、 T_3 、……. T_h; 这种树的高度分别为 1、2、3、……h。且是具有高度为1、2、3、……h的平衡树中结点个数最少的二叉树。



T_1 高度 h = 1 结点个 数最少

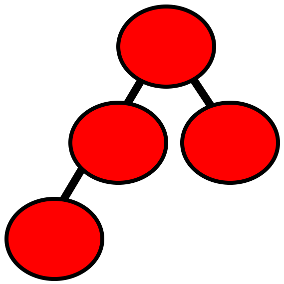


T_2 高度 h = 2 结点个数最少

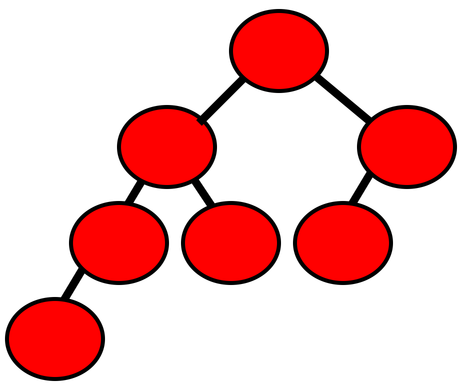


T_3 高度 h = 3 结点个数最少

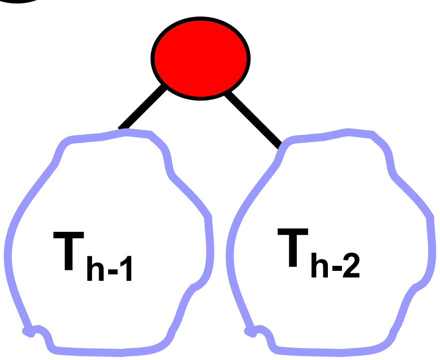
9.2.2 平衡二叉树——查找分析



T_3 高度 h = 3 结点个数最少



T_4 高度 h = 4 结点个数最少



T_h 高度 h 结点个数最少的平衡树

左子树为 T_{h-1} 右子树为T_{h-2}

9.2.2 平衡二叉树——查找分析

- 证明：2、设 $t(h)$ 是高度 h 的平衡树 T_h 的最少结点个数，可以得出：

$t(1) = 1;$

$t(2) = 2;$

$t(3) = 4;$

$t(4) = 7;$

$\vdots$

$t(h) = t(h-1) + t(h-2) + 1 \quad \text{for } h >= 3$

该数的序列为 1、2、4、7、12、20、33、54、88……

9.2.2 平衡二叉树——查找分析

- 定理：具有 N 个结点的平衡树，高度 h 满足：

$$\log_2(N+1) \le h \le \log_\alpha(\sqrt{5}(N+1)) - 2$$

$$其中 \alpha = (1+\sqrt{5})/2$$

- 证明：

2、 该数的序列为 1、2、4、7、12、20、33、54、88 ……

而 Fibonacci 数列为：0、1、1、2、3、5、8、13、21、34、55、89……

利用归纳法容易得证：*t(h) = f(h+2)* - 1；于是转化为求 Fibonacci 数的问题。

- 由于：$f(h+2) \approx \dfrac{\alpha^{h+2}}{\sqrt{5}}$;这里 $\alpha = (1+\sqrt{5})/2$

- 所以：$N \approx \dfrac{\alpha^{h+2}}{\sqrt{5}} - 1$

- 所以，具有 N 个结点的平衡树的最大深度为 $\log_\alpha(\sqrt{5}(N+1)) - 2$

9.2.2 平衡二叉树——查找分析

- 由此推得，深度为***h***的二叉平衡树中所含结点的最小值：

$$N_h \approx \frac{\alpha^{h+2}}{\sqrt{5}} - 1$$

- 反之，含有***n***个结点的二叉平衡树能达到的最大深度：

$$h_n = \log_\alpha(\sqrt{5}(N+1)) - 2$$

- 因此，在二叉平衡树上进行查找时，查找过程中和给定值进行比较的关键字的个数和 ***log(n)*** 相当。

9.2.3 B-树

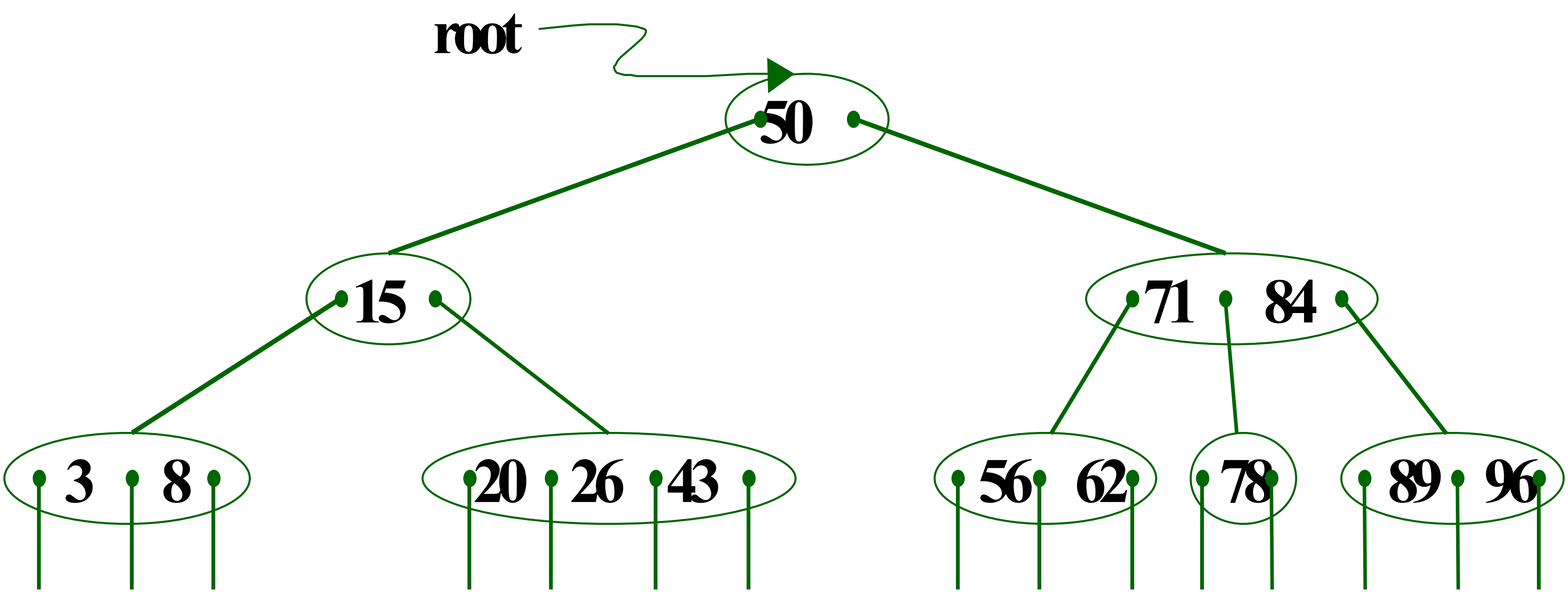
- 为什么采用B-树：
  - 大量数据存放在外存中，通常存放在硬盘中。由于是海量数据，不可能一次调入内存。因此，要多次访问外存。但硬盘的驱动受机械运动的制约，速度慢。所以，主要矛盾变为减少访外次数。
  - 例: 用二叉树组织文件，当文件的记录个数为 **100,000**时，要找到给定关键字的记录，需访问外存**17**次（**log100,000**）,太长了！
  - 在 **1970** 年由 **R bayer** 和 **E macreight** 提出用**B-**树作为索引组织文件。提高访问速度、减少时间。

9.2.3 B-树

1. 定义
2. 查找过程
3. 插入操作
4. 删除操作
5. 查找性能的分析

9.2.3 B-树 ——定义

B-树是一种 **平衡** 的 **多路** **查找** 树:



9.2.3 B-树 ——定义

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（$1\le i\le n$） $n<m$

n 个指向记录的指针 D_i（$1\le i\le n$）

$n+1$ 个指向子树的指针 A_i（$0\le i\le n$）

—— 多叉树的特性

B-树结构的C语言描述如下：

```
#define m   3
typedef struct BTNode {
    int  keynum;          // 结点中关键字个数，结点大小
    struct BTNode  *parent;
                          // 指向双亲结点的指针
    KeyType   key[m+1]; // 关键字（0号单元不用）
    struct BTNode  *ptr[m+1]; // 子树指针向量
    Record        *recptr[m+1];  // 记录指针向量
} BTNode, *BTree; // B-树结点和B-树的类型
```

9.2.3 B-树 ——定义

- 非叶结点中的多个关键字均自小至大有序排列，即：$K_1 < K_2 < \ldots < K_n$；
- 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；

—— 查找树的特性

9.2.3 B-树 ——定义

- 树中所有叶子结点均不带信息，且在树中的同一层次上;
- 根结点或为叶子结点，或至少含有两棵子树;
- 其余所有非叶结点均至少含有$\lceil m/2 \rceil$棵子树，至多含有m棵子树;

——— 平衡树的特性

9.2.3 B-树 ——定义

- m 阶 B_ 树满足或空，或：

A、根结点要么是叶子，要么至少有两个儿子

B、除根结点和叶子结点之外，每个结点的儿子个数为： $m/2 \leq s \leq m$

C、有 s 个儿子的非叶结点具有 $n = s - 1$ 个关键字，即 $s = n + 1$

这些结点的数据信息为：

（n，A0，K1，R1，A1，K2，R2，A2，……… Kn，Rn，An）

这里：n: 关键字的个数

A0：<K1 的结点的地址（指在该 B- 树中）

K1：关键字

R1：关键字 = K1 的数据记录在硬盘中的地址

A1：> K1 且 < K2 的结点的地址（指在该 B- 树中）

余类推 ………

An：> Kn 的结点的地址（指在该 B- 树中）

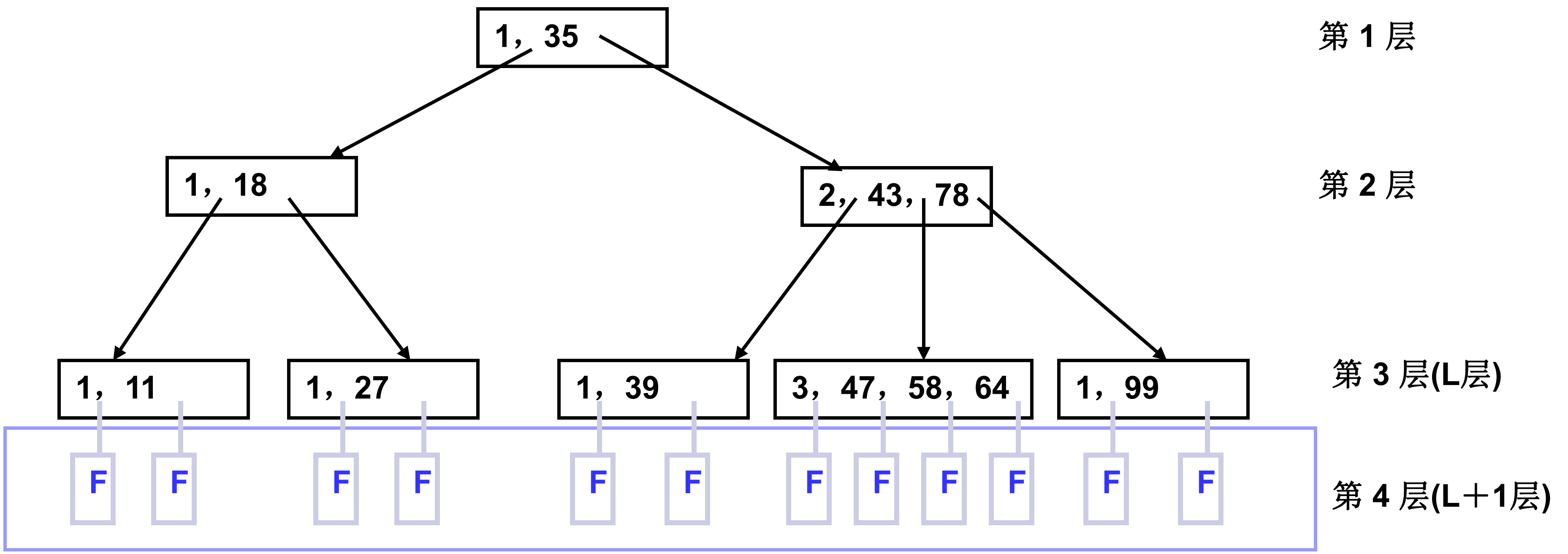
注意：$K1 \leq K2 \leq \ldots\ldots \leq Kn$

D、所有的叶子结点都出现在同一层上，不带信息（可认为外部结点或失败结点）。

9.2.3 B-树 ——定义

- 例如：**m = 4 阶 B- 树。**

除根结点和叶子结点之外，每个结点的儿子个数至少为 $\lceil m/2 \rceil$= **2** 个；结点的关键字个数至少为 **1** 。该 **B-**树的深度为 **4**。叶子结点都在第 **4** 层上。



注意到：叶子结点数＝N＋1 （N为关键字总数）

9.2.3 B-树 —— 程

在外存进行的查找操作

在内存进行的查找操作

- 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。

9.2.3 B-树 ——查找过程

- 假设返回的是如下所述结构的记录：

```
typedef struct {
  BTNode  *pt;     // 指向找到的结点的指针
  int  i;          // 1..m，在结点中的关键字序号
  int  tag;        // 标志查找成功(=1)或失败(=0)
} Result;          // 在B树的查找结果类型
```

9.2.3 B-树 ——查找过程

```
Result SearchBTree(BTree T, KeyType K) {
 // 在m 阶的B-树T中查找关键字K, 返回查找结果 (pt, i, tag)。若查找成功，则特征值 tag=1, 指针pt所指结点中第i个关键字等于K；否则特征值tag=0, 等于K的关键字应插入在指针pt 所指结点中第i个关键字和第i+1个关键字之间
      … …
} // SearchBTree
```

9.2.3 B-树 ——查找过程

```
//初始化，p指向待查结点，q指向p的双亲
p=T;  q=NULL;  found=FALSE;  i=0;
 while (p && !found) {
   n=p->keynum;   i=Search(p, K);  // 在p->key[1..keynum]
                        //中查找 i，p->key[i]≤K<p->key[i+1]
   if (i>0 && p->key[i]==K)  found=TRUE;
   else { q=p;   p=p->ptr[i]; } // q 指示 p 的双亲
 }
 if (found)  return (p,i,1);      // 查找成功
 else return (q,i,0);             // 查找不成功
```

9.2.3 B-树 ——查找过程

在B-树上找结点

- B- 树的查找代价分析：代价主要在于访问外存的时间。
  - 查找过程类似于二叉树的查找。如查找关键字为KEY的记录。

从根开始查找，如果 K_i = KEY 则查找成功，R_i 为关键字为 KEY 的记录的地址。

若 K_i < KEY < K_{i+1}; 查找 A_i 指向的结点

若 KEY < K_1; 查找 A_0 指向的结点

若 KEY > K_n; 查找 A_n指向的结点

若 找到叶子，则查找失败。

- 注意：每次查找将去掉 （s-1）个分支，比二分查找快得多。

9.2.3 B-树 ——查找过程

- 现考虑最坏的情况，设关键字的总数为N，求 m阶 B-树的最大层次L。

层次	结点数（至少）
1	1
2	2
3	$2(\lceil m/2\rceil)$
4	$2(\lceil m/2\rceil)^2$
L	$2(\lceil m/2\rceil)^{L-2}$
L+1	$2(\lceil m/2\rceil)^{L-1}$ （L+1层为叶子结点，即查找不成功的结点为N＋1）

因为除根之外的每个非终端结点至少有 $\lceil m/2\rceil$ 棵子树。

所以，$N+1 \geq 2*\lceil m/2\rceil^{L-1}$

故：$L \leq \log_{\lceil m/2\rceil}((N+1)/2)+1$

这就是说，在含有N的关键字的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$\log_{\lceil m/2\rceil}((N+1)/2)+1$。

9.2.3 B-树 ——查找过程

这就是说，在含有N的关键字的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$\log_{\lceil m/2 \rceil}((N+1)/2)+1$。

- 例：设 N ＝1000,000 且 m＝256， 则 L ≤ 3；最多 3 次访问外存可找到所有的记录。

9.2.3 B-树 ——插入操作

- 在查找不成功之后，需进行插入。显然，关键字插入的位置必定在最下层的非叶结点，找到插入位置，将关键字和其它信息按序插入，有下列几种情况：

1）插入后，该结点的关键字个数n<m，不修改指针;

2）插入后，该结点的关键字个数 n=m，则需进行“结点分裂”，令 s = $\lceil m/2 \rceil$，将（K_s，p）插入双亲结点

在原结点中保留

（A_0，K_1，…… ，K_{s-1}，A_{s-1}）；

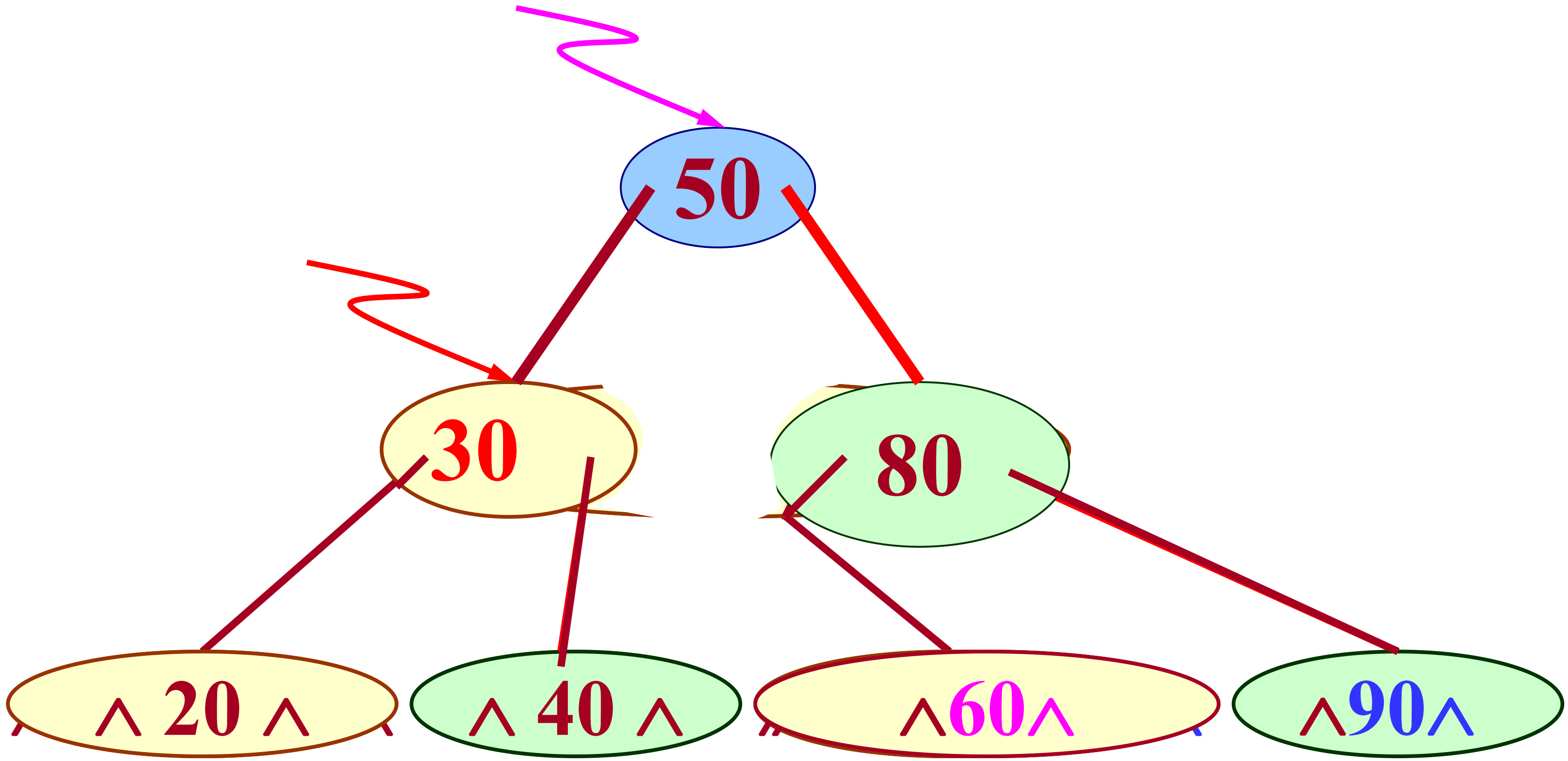
建新结点

（A_s，K_{s+1}，…… ，K_n，A_n）；

3）若双亲为空，则建新的根结点。

若分裂一直进行到根结点，树可能长高一层。

例如：下列为3阶B-树

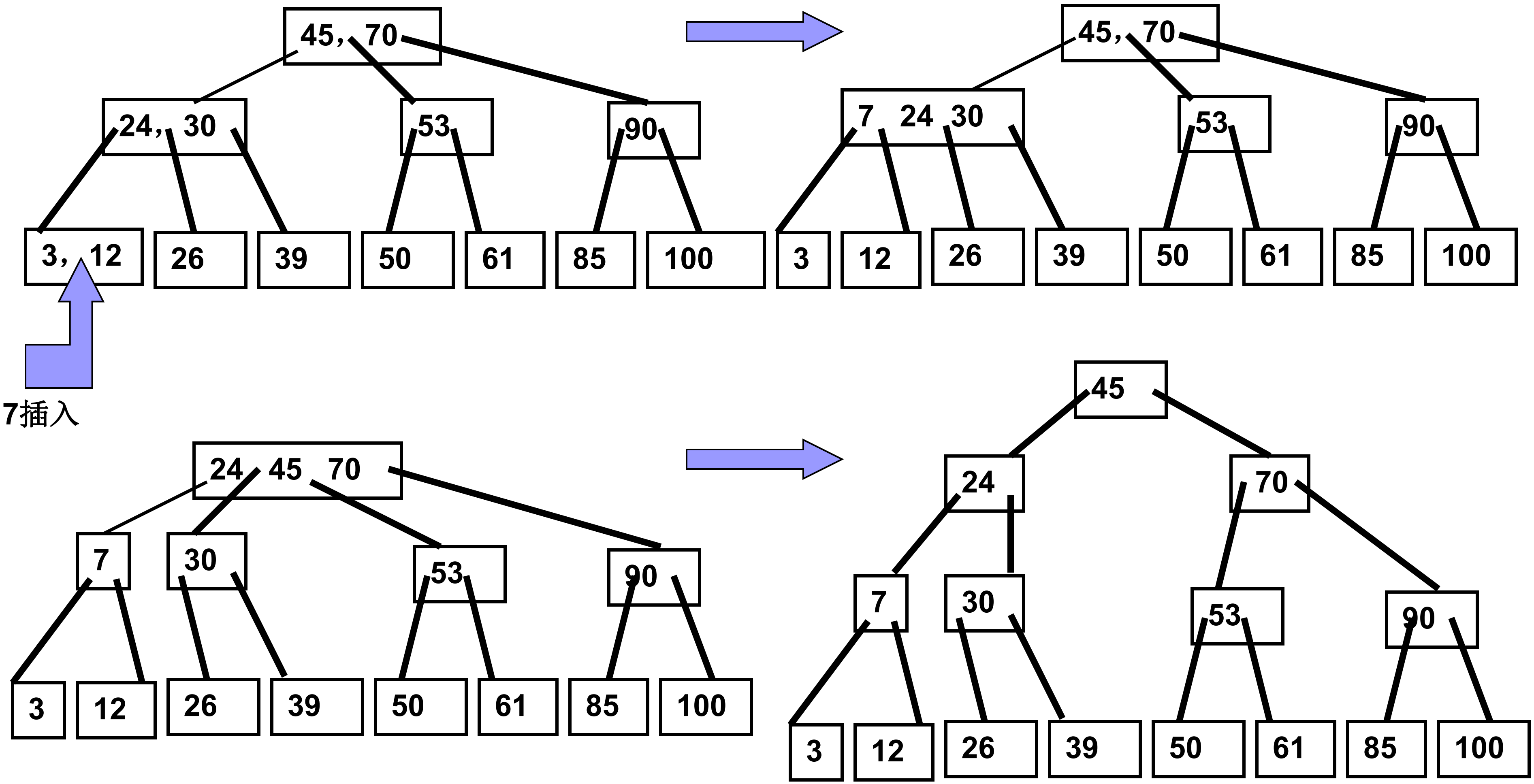


插入关键字 = 60, 90, 30,

9.2.3 B-树 ——插入操作

例如：3 阶 B-树的插入操作。m=3, $\lceil m/2 \rceil$;子树的数目为2或者3，故又称2－3树。

结点内的关键字数目至少1个，至多2个。



9.2.3 B-树 ——删除操作

- 和插入的考虑相反，首先必须找到待删关键字所在结点，并且要求删除之后，结点中关键字的个数不能小于$\lceil m/2 \rceil -1$，否则，要从其左(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)，则必须进行结点的“合并”。

9.2.3 B-树 ——删除操作

- **B-树删除操作步骤：**

  1、查找具有给定键值的关键字 K_i

  2、如果在第 L 层（注意：第 L+1 层为叶子结点），转 4 。

  3、否则，则首先生成“替身”。用它的右子树中的最左面的结点的关键字值，即 处于第 L 层上的最小关键字值取代。然后，删除第 L 层上的该关键字。

  4、从第 L 层开始进行删除。

9.2.3 B-树 ——删除操作

- **4、从第 L 层开始进行删除。**

A、不动：若删除关键字值的那个结点的关键字的个数仍处于 $\lceil m/2 \rceil -1$ 和 $m-1$ 之间。则结束。

B、借：若删除关键字值的那个结点的关键字的个数原为 $\lceil m/2 \rceil -1$ 。而它们的左或右邻居结点的关键字的个数 $> \lceil m/2 \rceil -1$；则借一个关键字过来。处理结束。

C、并：若该结点的左或右邻居结点的关键字的个数为 $\lceil m/2 \rceil -1$；则执行合并结点的操作。

如结点原为：（ …………. $(K_i, A_i), (K_{i+1}, A_{i+1})$, …………. ）

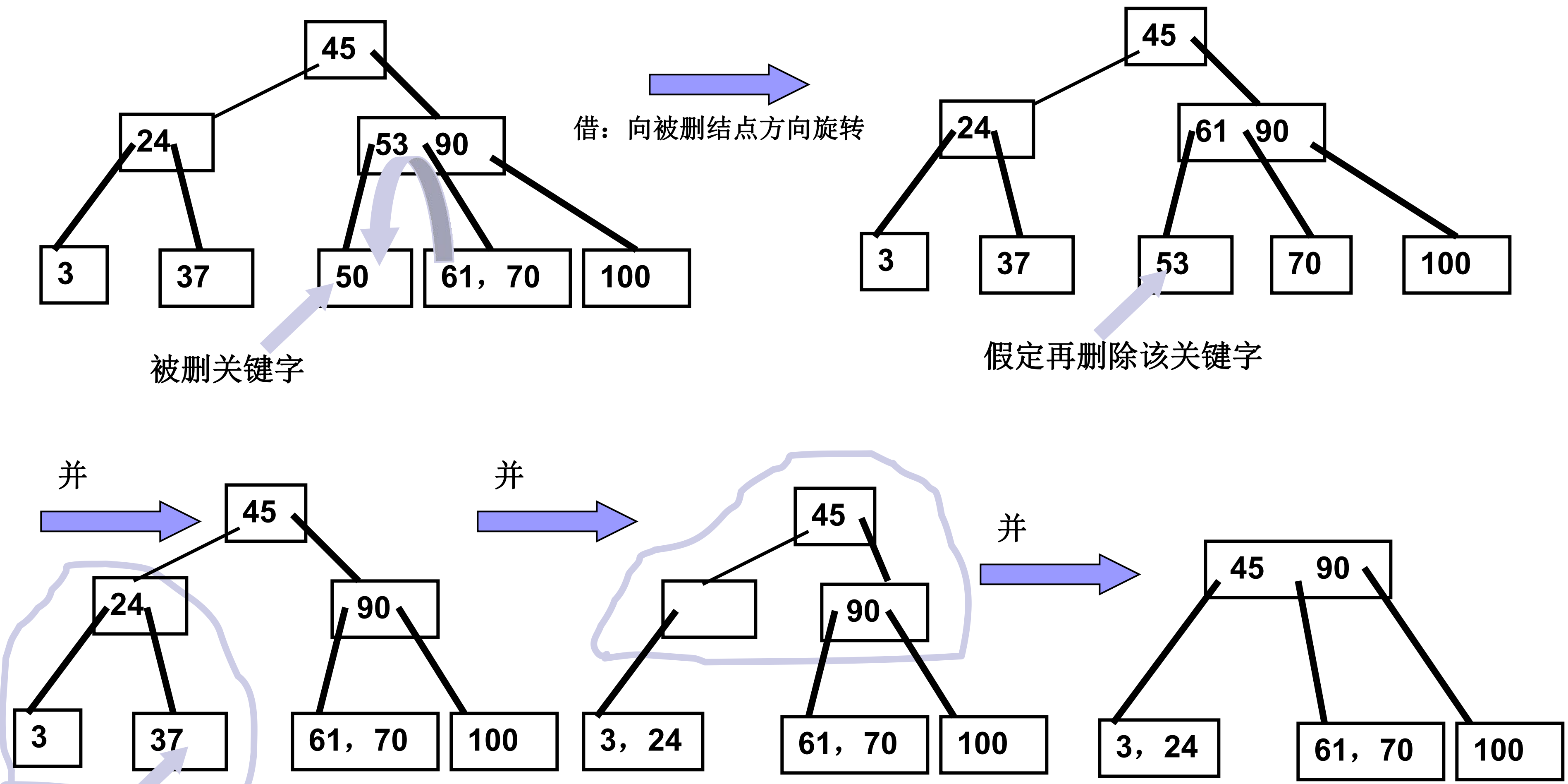
（ A_0', (K_1', A_1') ……… ）

（ A_0'', (K_1'', A_1'') ……… ）

…… K_1^L ……… 第 L 层：找到了被删结点的替身。

9.2.3 B-树 ——删除操作

例如：3 阶 B- 树的删除操作。m=3，m/2 - 1 = 1; 至少 1 个关键字，二个儿子结点。



假定再删除该关键字

并：和父结点的一个关键字、及邻居合并。有可能进行到根，使**B-**树的深度降低一层！

9.2.4 B+树 ——是B-树的一种变型

m阶B+树和m阶B-树的结构差异在于：

- 有**n**棵子树的结点中含有**n**个关键字。
- 所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接构成一个有序链表，其头指针指向含最小关键字的结点。
- 所有的非终端结点可以看成是索引部分，结点中仅含有其子树（根结点）中的最大（或最小）关键字。
- 所有叶子结点都处在同一层次上，每个叶子结点中关键字的个数均介于$\lceil m/2 \rceil$和m之间。

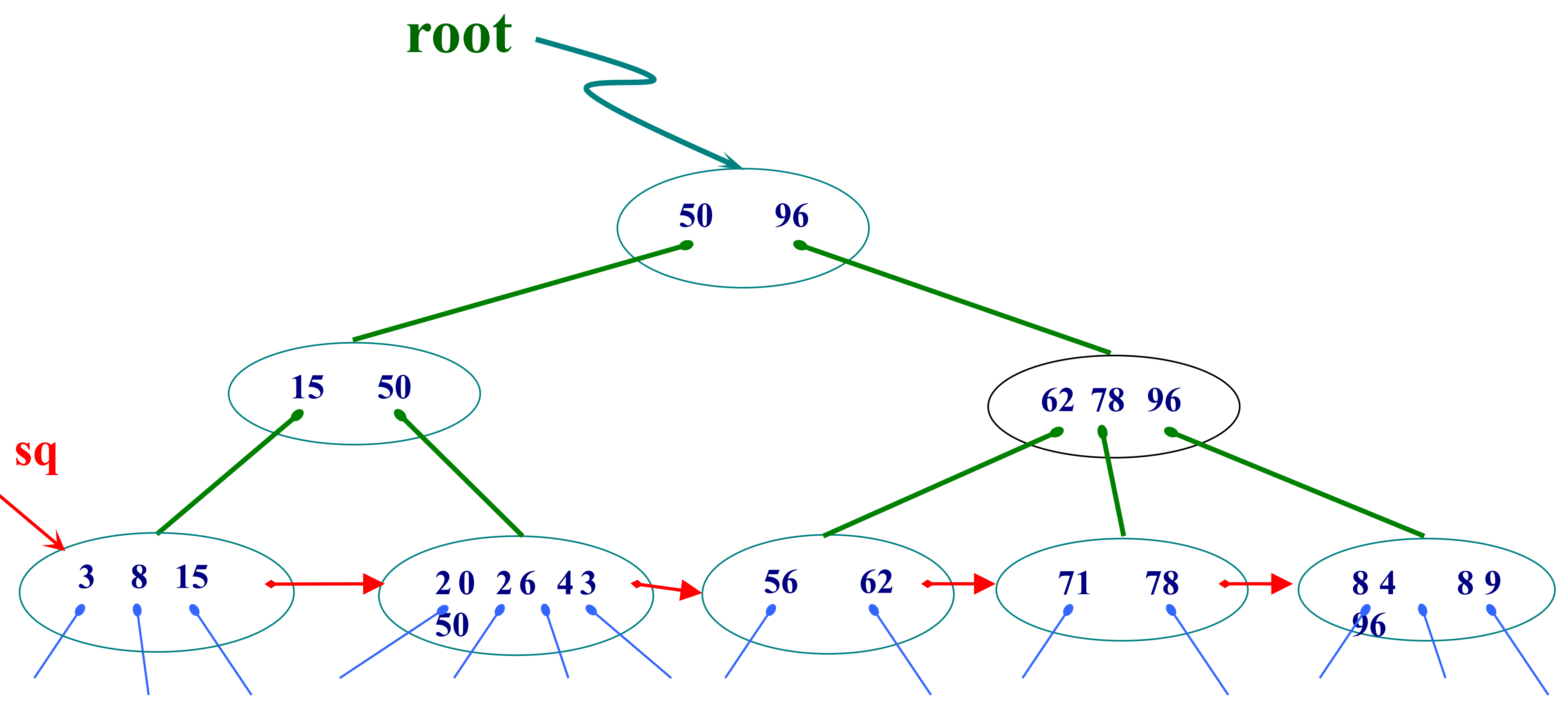
9.2.4 B+树——查找

- 例如图9.18所示为一棵3阶的B+树。通常在B+树上有两个头指针，一个指向根结点，另一个指向关键字最小的叶子结点。
- 因此，可以对B+树进行两种查找运算：一种是从最小关键字起顺序查找，另一种是从根结点开始，进行随机查找。
- 随机查找：不管查找成功与否，每次查找都是走了一条从根到叶子结点的路径。

——因为在查找时，若非终端结点上的关键字等于给定值，并不终止，而是继续向下直到叶子结点。

9.2.4 B+树——插入与删除

- 插入与删除：**B+**树的插入（删除）仅在叶子结点进行，所引起的分裂（合并）与**B-**树类似。



root
50 96
15 50
62 78 96
sq
3 8 15
2 0 2 6 4 3 50
56 62
71 78
8 4 8 9 96

第九章 查找

9.1 静态查找表

9.1.1 顺序查找表

9.1.2 有序查找表

- 折半查找
- 斐波那契查找
- 插值查找

9.1.3 静态查找树表

- 针对有序表，不等概率查找
- 构造次优查找树

9.1.4 索引顺序表

9.2 动态查找表

9.2.1 二叉排序树

- 分割式查找法
- 插入算法
- 查找分析
- 删除算法

9.2.2 平衡二叉树

- 定义
- 如何构造
- 查找分析

9.2.3 B-树和B+树

9.3 哈希查找表

- 定义
- 查找、插入、删除
- 查找分析

9.3 哈希查找表

9.3.1 哈希表是什么？

9.3.2 哈希函数的构造方法

9.3.3 处理冲突的方法

9.3.4 哈希表的查找

9.3.1 哈希表是什么？

- 以上两节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，查找的过程为给定值依次和关键字集合中各个关键字进行比较，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。

9.3.1 哈希表是什么？

- 不同的表示方法，其差别仅在于：关键字和给定值进行比较的顺序不同。
- 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：记录在表中位置和其关键字之间存在一种确定的关系。

9.3.1 哈希表是什么？

- **哈希表特点**：不用比较的办法，直接根据所求结点的关键字值 KEY 找到这个结点。 追求更快的速度 O(1)，优于任何其它的查找算法。
- **定义**：设 M 存区由 m 个单元构成，它的第一个单元的地址为 0。
  - 设表具有n 个结点 $a_1, a_2, a_3, \ldots\ldots\ldots\ldots a_n$;
  - 这些结点相应的关键字值分别为： $k_1, k_2, k_3, \ldots\ldots\ldots k_n$。
  - 又设 H 函数是一个确定的函数，它能将关键字值 k_i 映射为 M 存区的地址：即，

    $H(k_i) \longrightarrow 0 \sim m-1$ （注意：是一个确定的地址）

    该地址就是结点 a_i 的存放地址，也称哈希地址。

    H 函数通常称之为哈希（hashing）函数，

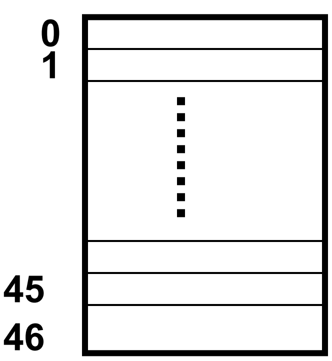
    而 M 存区称之为 hashing表。
  - 负载系数（或称装填因子）：$\alpha = n/m$

9.3.1 哈希表是什么？

例1：对于31个自然数，进行哈希。哈希表可以设置为47个单元，可能的哈希函数可以是：f(key)=key MOD 47。则 $\alpha=n/m=31/47$

注意：哈希函数的设计要考虑所有可能的关键字。比如上例，哈希函数的设计就要以自然数为定义域，任给一个自然数，都可以得到它的相应哈希地址。

哈希表



例2：对于如下 9 个关键字

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

设哈希函数 f(key) =

$\lfloor$(Ord(第一个字母) -Ord('A')+1)/2$\rfloor$

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

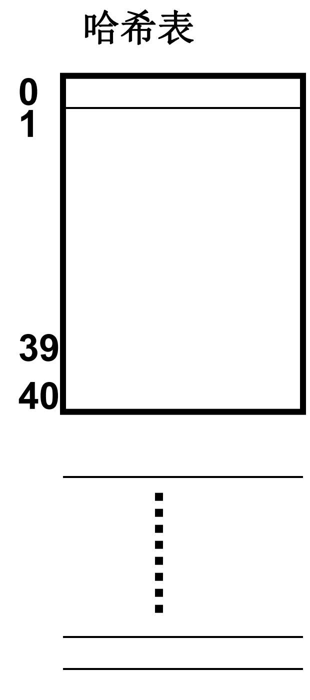
问题：若添加关键字 Zhou，怎么办？

能否找到另一个哈希函数？

9.3.1 哈希表是什么？

- 冲突：对不同的关键字可能得到同一哈希地址，即key1≠key2，而f(key1)=f(key2)，这种现象称冲突。Key1和key2称为同义词。

- 例：将 31 个常用的英文单词(A、AND、……YOU)，映射到 M 存区。设 m = 41，映射是等可能性的。则可能的映射种数为 41^{31}，不冲突的分布为 P_{41}^{31}；不冲突的可能性为 1/10,000,000



- 简短的结论：选取好的 hashing 函数非常困难，不冲突的可能性非常小。只能选择恰当的哈希函数，使冲突尽可能少地产生。
- 减少冲突的方法：好的 hashing 函数；减少负载系数

9.3.2 哈希函数的构造方法

对数字的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

$H(key) = key$ 或者

$H(key) = a \times key + b$

此法仅适合于:

地址集合的大小 == 关键字集合的大小

2. 数字分析法

- 假设关键字是以r为基的数，并且哈希表中可能出现的关键字都是事先知道的，则可取关键字的若干数位组成哈希地址。
  - 例：对于以下21个关键字求哈希函数（假设表长为30）：

    2005022001

    2005022022

    2005022043

    …

    2005022379

    2005022880
  - 观察以上关键字，可以发现只有后三位有变化，其他位不变化，则可以取后三位之和作为哈希函数。

3. 平方取中法

- 以关键字的平方值的中间几位作为存储地址。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度很高的现象。

e.g: $(4731)^2$ ＝ 223 82 361；选取 82（在 m ＝100 情况下）。

4. 折叠法

- 将关键字分割成若干部分，然后取它们的叠加和为哈希地址。有两种叠加处理的方法：移位叠加和间界叠加。
- 此方法适合于：关键字的数字位数特别多。

- 移位叠加法和间界叠加法：

  key = 381，412，975

  选取 768 或 570 作为散列地址（在 m ＝1000 情况下）。

$$\begin{array}{r} 381 \\ 412 \\ +\quad 975 \\ \hline 1\ 768 \end{array} \qquad\qquad \begin{array}{r} 975 \\ 214 \\ +\quad 381 \\ \hline 1\ 570 \end{array}$$

5. 除留余数法

定哈希函数为：

H(key) = key MOD p 或 H(key) = key MOD p +c

其中， $p \le m$（表长）并且

p 应为不大于 m 的素数

或是不含20 以下的质因子

余数总在 0 ~ p-1 之间

为什么要对 p 加限制选为素数?

例如:

设选 p 为偶数，当key 值都为奇数：则 H(key) = key MOD p ，哈希函数值为奇数，哈希表中一半单元被浪费掉。

设选 p 为 95 ，当key 值都为 5 的倍数：则 H(key) = key MOD p ，哈希函数值为：0、5、10、15、…… 90，哈希表中4/5 的单元被浪费掉。

6.随机数法

设定哈希函数为:

$$H(key) = Random(key)$$

其中，Random 为伪随机函数

通常，此方法用于对长度不等的关键字构造哈希函数。

9.3.2 哈希函数的构造方法

- 实际工作中需视不同的情况采用不同的哈希函数。通常，考虑的因素有：
  - （1）计算哈希函数所需时间；
  - （2）关键字的长度；
  - （3）哈希表的大小；
  - （4）关键字的分布情况；
  - （5）记录的查找频率。
- 总的原则是使产生冲突的可能性降到尽可能地小。

9.3.3 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**哈希地址。

1. 开放定址法
2. 链地址法
3. 公共溢出区法

9.3.3 处理冲突的方法——开放定址法

为产生冲突的地址 H(key) 求得一个**地址序列**：

$$\mathbf{H_0, H_1, H_2, \ldots,\ \ H_s} \qquad 1 \le s \le m-1$$

其中：$\mathbf{H_0 = H(key)}$

$$\mathbf{H_i = (\ H(key) + \mathit{d_i}\ )\ MOD\ m}$$

$$i=1, 2, \ldots, s$$

对增量 d_i 有三种取法:

1) 线性探测再散列$d_i=c\times i$，一般选和m互质的数做为步长，最简单的情况$c=1$；

2) 二次探测再散列$d_i=1^2, -1^2, 2^2, -2^2, \ldots,$

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），即用另一个哈希函数作为步长进行探测，找到下一个空单元

注意：增量 d_i 应具有“完备性”

即：产生的 H_i 均不相同，且所产生的s个 H_i 值能覆盖哈希表中所有地址。则要求：

※ 线性探测可以保证做到，只要哈希表未填满，总能找到一个不发生冲突的地址。

※ 二次探测时的表长 m 必为形如 $4j+3$ 的素数（如: 7, 11, 19, 23, … 等）；

※ 随机探测时的 m 和 d_i 没有公因子。

9.3.3 处理冲突的方法——开放定址法

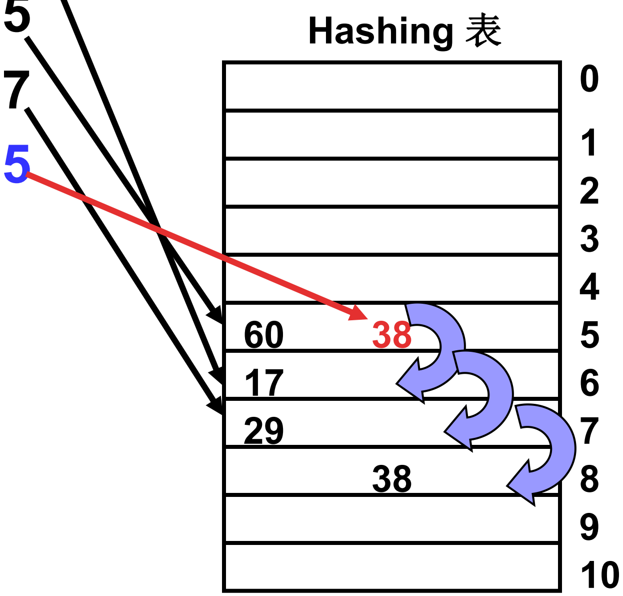
- 例1: 假定采用的 hashing 函数为：H(key) = key MOD 11 。假定的关键字序列为：17、60、29、38 …… ；它们的散列过程为：

H(17) = 6

H(60) = 5

H(29) = 7

H(38) = 5



当散列 38 时发生冲突，
同 60 争夺第 5 个单元。

解决办法：
线性探测下一个空单元

步长：1

$H(key) = (key+d_i) \text{ MOD } 11$

其中： d_i 为 1、2……10

注意：可取其它步长，如 3

- 冲突：
  - 初级冲突：不同关键字值的结点得到同一个散列地址。
  - 二次聚集：和不同散列地址的结点争夺同一个单元。（5、6、7地址的记录都将争8）
  - 结果：冲突加剧，最坏时可能达到 O（n）级代价。

例2：关键字集合

{ 19, 01, 23, 14, 55, 68, 11, 82, 36 }

设定哈希函数 H(key) = key **MOD** 11 (表长=11)

若采用**线性探测再散列**处理冲突：

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		

若采用**二次探测再散列**处理冲突：

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

若采用**随机探测再散列**处理冲突:

$H_2(key)$ 是另设定的一个哈希函数，它的函数值应和 m **互为素数**。若 m 为素数，则 $H_2(key)$ 可以是 1 至 m-1 之间的**任意数**；若 m 为2的幂次，则$H_2(key)$ 应是 1 至 m-1 之间的**任意奇数**。

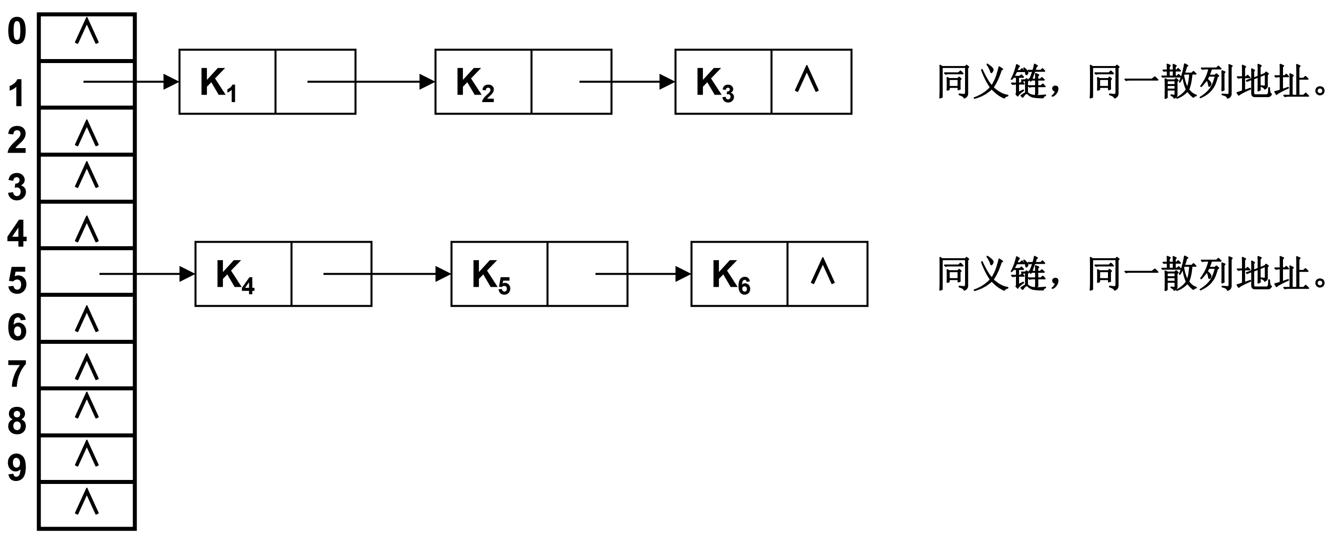
例如， 当 m=11时，

可设 $H_2(key)=(3\ key)\ MOD\ 10+1$

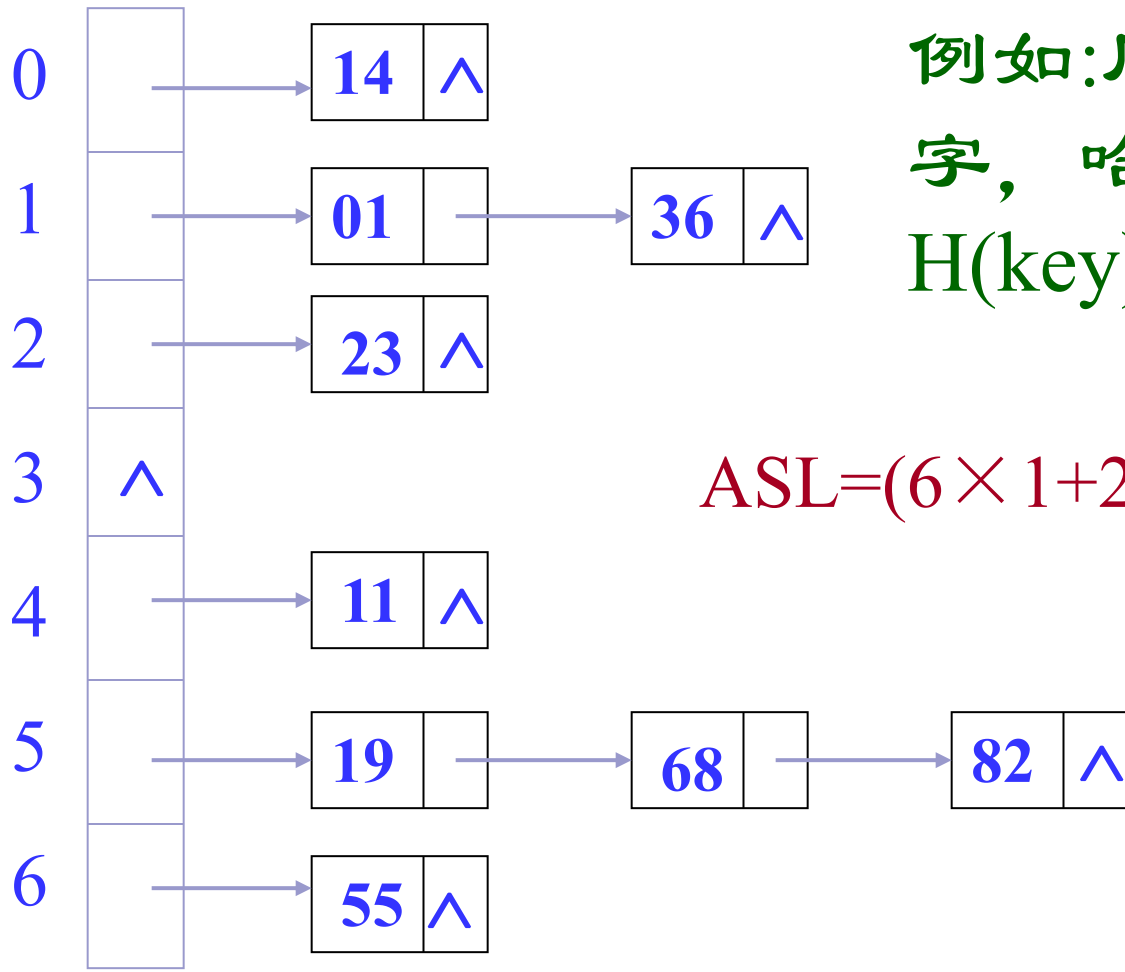
0	1	2	3	4	5	6	7	8	9	10
23	01	68	14	11	82	55		19		36
2	1	1	1	2	1	2		1		3

9.3.3 处理冲突的方法——链地址法

- 链地址法：—— 将具有同一散列地址的结点保存于 M 存区的各自的链表之中。
  - 假设某哈希函数产生的哈希地址在区间[0, m-1]上，则设立一个指针型向量Chain ChainHash[m]; 其每个分量的初始状态都是空指针。
  - 凡哈希地址为 i 的记录都插入到头指针为ChainHash[i]的链表中。在链表中的插入位置可以在表头或表尾；也可以在中间，以保持同义词在同一线性链表中按关键字有序。



9.3.3 处理冲突的方法——链地址法



例如:同前例的关键字，哈希函数为

H(key)=key MOD 7

$ASL=(6\times 1+2\times 2+3)/9=13/9$

9.3.3 处理冲突的方法——公共溢出区法

- 公共溢出区法：
  - M 存区只存放一个记录，发生冲突的记录都存放在公共溢出区内。
  - 假设哈希函数的值域为[0, m-1]，则设向量HashTable[0..m-1]为基本表，每个分量存放一个记录，另设立向量OverTable[0..v]为溢出表。
  - 所有关键字和基本表中关键字为同义词的记录，不管它们由哈希函数得到的地址是什么，一旦发生冲突，都填入溢出表。

9.3.4 哈希表的查找

- 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，计算哈希地址 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。

9.3.4 哈希表的查找

- 已知一组关键字（**19,14,23,1,68,20,84,27,55,11,10,79**），按照哈希函数**H(key)=key MOD 13**和线性探测处理冲突构造所得哈希表**a.elem[0..15]**如下图：

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

查找成功的例子：查找**K＝1**，**H(1)=1**

可以求得平均查找次数为：**ASL=(1*6+2+3*3+4+9)/12=2.5**

9.3.4 哈希表的查找

- 从查找过程得知，哈希表查找的平均查找长度实际上并不等于零。
- 虽然哈希表在关键字与记录的存储位置建立了直接映像，但由于“冲突”的产生，使得哈希表的查 给定值和关键字进行比较的过程。
- 因此仍需以平均查找长度作为衡量哈希 。
- 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**α=n/m** 值的大小（**n**—记录数，**m**—表的长度）

> 一般情况下，可以认为选用的哈希函数是“均匀”的，则在讨论**ASL**时，可以不考虑它的因素

可以证明：查找成功时有下列结果：

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

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

9.3.4 哈希表的查找

- 以随机探测为例进行分析推导：

- 分析长度为m的哈希表中装填有n个记录时查找不成功的平均查找长度。这个问题相当于要求在这张表中填入第n+1个记录时所需作的比较次数的期望值。

- 设哈希函数是均匀的、处理冲突后产生的地址也是均匀的。

- 查找不成功的平均查找长度：

$$U_n = \sum_{i=1}^{n+1} i \times q_i$$

其中：

q_i = 经过 i 次探测确定某结点不在哈希表中的概率。

= 第1到第 i-1 次探测由于冲突而失败的概率，及第 i 次探测找到空单元的概率。

$$= \frac{n}{m} \times \frac{n-1}{m-1} \times \frac{n-2}{m-2} \times \ldots\ldots \frac{n-(i-2)}{m-(i-2)} \times \left[1 - \frac{n-(i-1)}{m-(i-1)}\right]$$

其中： $1 \le i \le n+1$

可推出：$U_n = \sum_{i=1}^{n+1} i \times q_i = \frac{1}{1-\alpha}$ $\qquad \alpha = \frac{n}{m}$

9.3.4 哈希表的查找

- 查找成功的平均查找长度:

- 成功地找到某一个key所需要的探测次数恰等于将这个 key 插入到散列表中所需要的探测次数。

- 所以，n个key查找成功时的总查找长度＝ 将 n 个key 插入到散列表中所需要的探测次数

＝ 表空时进行插入时的探测次数 （对应于U_0 ）

＋ 表中有一个结点时进行插入时的探测次数 （对应于U_1）

＋ 表中有二个结点时进行插入时的探测次数 （对应于U_2 ）

＋ 表中有三个结点时进行插入时的探测次数 （对应于U_3）

⋮

＋ 表中有 n-1 个结点时进行插入时的探测次数（对应于U_{n-1} ）

所以：

$$s_n = (1/n)\sum_{i=0}^{n-1} U_i = \frac{m}{n}\int_0^{\alpha} \frac{1}{1-x}dx = \frac{-1}{\alpha}\ln(1-\alpha)$$

9.3.4 哈希表的查找

从以上结果可见:

哈希表的平均查找长度是 α 的函数，而不是 n 的函数。

这说明，用哈希表构造查找表时，可以选择一个适当的装填因子 α ，使得平均查找长度限定在某个范围内。

—— 这是哈希表所特有的特点。

本章要点

1. 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. 静态查找树的构造方法和查找算法，理解静态查找树和折半查找的关系。
3. 熟练掌握二叉排序树的构造和查找方法。
4. 理解B-树、B+树和建树的特点以及它们的建树和查找的过程。
5. 熟练掌握哈希表的构造方法，深刻理解哈希表与其它结构的表的实质性的差别。
6. 掌握按定义计算各种查找方法在等概率情况下查找成功时的平均查找长度。

课后作业：

- 推荐作业
- 9.1， 9.3， 9.29
- 9.11, 9.35, 9.38
- 9.13, 9.14, 9.19, 9.22, 9.24,9.40