第9章　查找

查找和排序是数据处理系统中最重要的两个操作；

其次是插入、删除操作；

讨论查找、排序，不可避免要涉及文件、记录、关键字等概念。

文件——查找表，是由同一类型的数据元素(记录)构成的集合

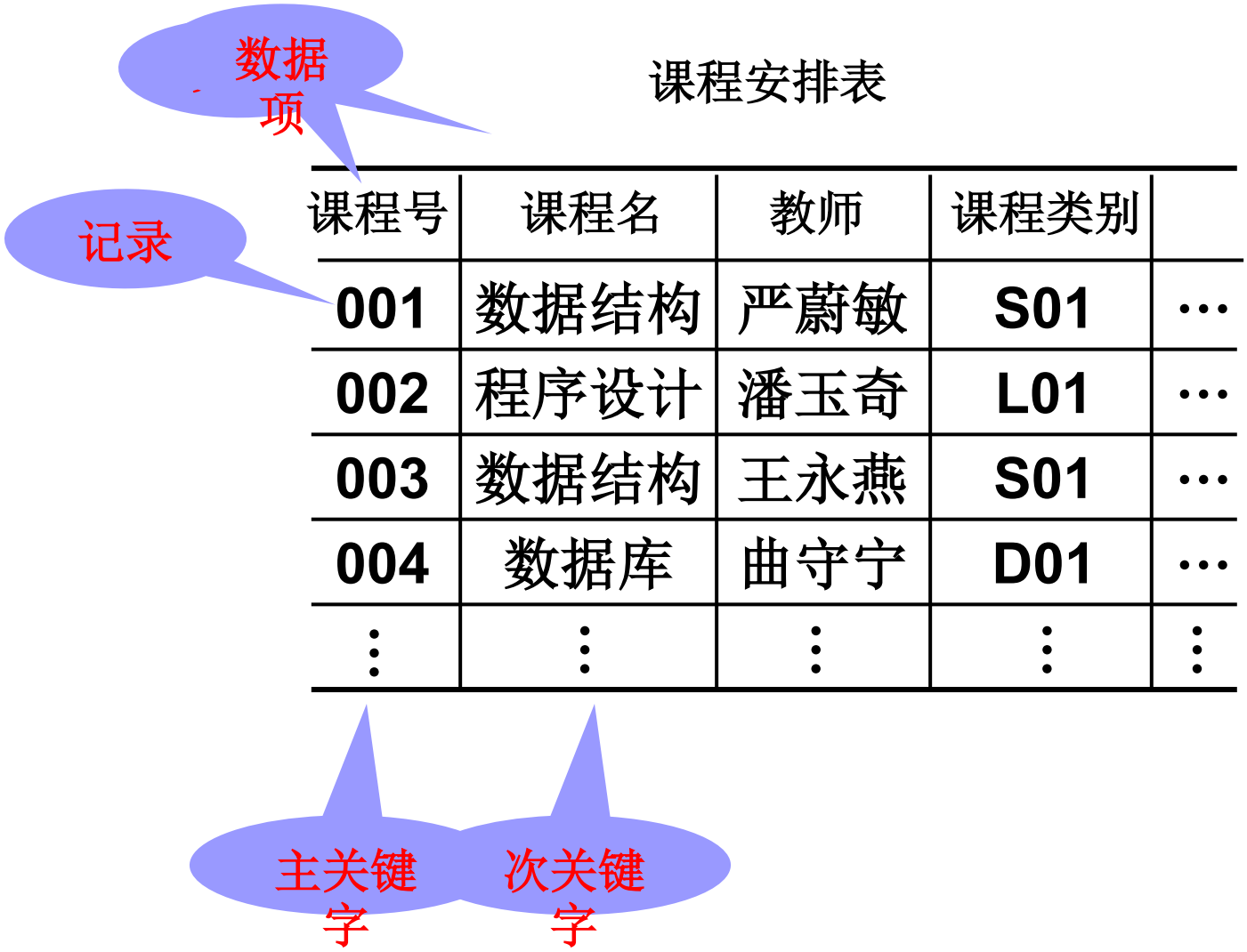
记录——构成文件的数据元素，是文件中可存取的数据的基本单位

字段——数据项，数据的最小单位

关键字——某个可以用来标识记录的数据项

主关键字——某个可以用来唯一标识记录的数据项

次关键字——可以用来识别若干记录的数据项


数据项
记录
课程安排表
课程号
课程名
教师
课程类别
001
数据结构
严蔚敏
S01
002
程序设计
潘玉奇
L01
003
数据结构
王永燕
S01
004
数据库
曲守宁
D01
主关键字
次关键字

对文件经常进行的操作有：

1) 查询某个“特定”的数据元素是否存在　　查找算法

2) 插入某个数据元素

3) 删除某个数据元素

4) 对数据元素进行排序　　排序算法

不管何种操作，都遵循一个重要的性质：

都是对主关键字操作

1. 查找的基本概念

- 查找表

  由同一类型的数据元素（记录）构成的集合。

- 查找的定义

  给定一个值**key**，在含有**n**个记录的表中找出关键字等于**key**的记录。若找到,则查找成功，返回该记录的信息或该记录在表中的位置；否则查找失败，返回相关的指示信息。

采用何种查找方法?

(1) 使用哪种数据结构来表示“表”, 即表中记录是按何种方式组织的。

(2) 表中关键字的次序。是对无序集合查找还是对有序集合查找?

- **静态查找表(Static Search Table)**：查询某个特定的元素是否在表中；检索某个特定的元素的各种属性。

- **动态查找表(Dynamic Search Table)**：若在查找的同时对表做修改运算(如插入和删除)。

2. 查找操作的性能分析

●基本操作：将记录的关键字和给定值进行比较。

●平均查找长度ASL(Average Search Length)：为确定数据元素在查找表中的位置， 需和给定值进行比较的关键字个数的期望值，称为查找算法在查找成功时的平均查找长度。

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

P_i为查找表中第i个记录的概率，C_i为找到第i个记录时，和给定值已经进行过比较的关键字个数。

9.1 静态查找表

在表的组织方式中,线性表是最简单的一种。三种在线性表上进行查找的方法：

(1) 顺序查找
(2) 折半查找（二分查找）
(3) 索引顺序表查找（分块查找）。

因为不考虑在查找的同时对表做修改,故上述三种查找操作都是在静态查找表上实现的。

1.顺序查找

顺序查找法的特点：用所给关键字与线性表中各元素的关键字逐个比较，直到成功或失败。存储结构通常为顺序结构，也可为链式结构。

例如, 在关键字序列为{3,9,1,5,8,10,6,7,2,4}的线性表查找关键字为8的元素。

开始: 3 9 1 5 8 10 6 7 2 4

第1次比较: 3 9 1 5 8 10 6 7 2 4

i=0

第2次比较: 3 9 1 5 8 10 6 7 2 4

i=1

第3次比较: 3 9 1 5 8 10 6 7 2 4

i=2

第4次比较: 3 9 1 5 8 10 6 7 2 4

i=3

第5次比较: 3 9 1 5 8 10 6 7 2 4

i=4

查找成功,返回序号4

```
typedef struct{
    ElemType *elem;
    int length;
}SSTable;

int Search_Seq(SStable ST, KeyType key)
{
   ST.elem[0].key = key;
   for( i=ST.length; !EQ(ST.elem[i].key, key); --i );
   return i;
}
```

```
for( i=ST.length; i>0&&!EQ(ST.elem[i].key, key); --i );
```

- 顺序查找的平均查找长度**ASL**：

假设表长度为**n**，那么查找到第**i**个记录时，和给定值已进行过比较的关键字个数为**n-i+1**，即C_i=n-i+1。又假设查找每个数据元素的概率相等，即P_i=1/n， 则顺序查找算法的平均查找长度为：

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

- 查找不成功时的平均查找长度：

假设查找成功和不成功的可能性相等，且每个记录的查找概率也相等， 即**Pi=1/(2n)**，则

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

2. 折半查找法 （二分法查找法）

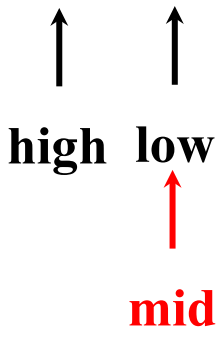
要求待查找的表必须是按关键字大小有序排列的顺序表。

折半查找的思想：将表中间位置记录的关键字与查找关键字比较，如果两者相等，则查找成功；否则利用中间位置记录将表分成前、后两个子表， 如果中间位置记录的关键字大于查找关键字，则进一步查找前一子表，否则进一步查找后一子表。重复以上过程，直到找到满足条件的记录，使查找成功，或直到子表不存在为止，此时查找不成功。

折半查找

$$mid = \lfloor (low + high) / 2 \rfloor$$

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



查找 21

mid = 6　　high = mid – 1 = 5

mid = 3　　low = mid + 1 = 4

mid = 4　　找到

查找 85

mid = 6　　low = mid + 1 = 7

mid = 9　　low = mid + 1 = 10

mid = 10　　high = mid – 1 = 9

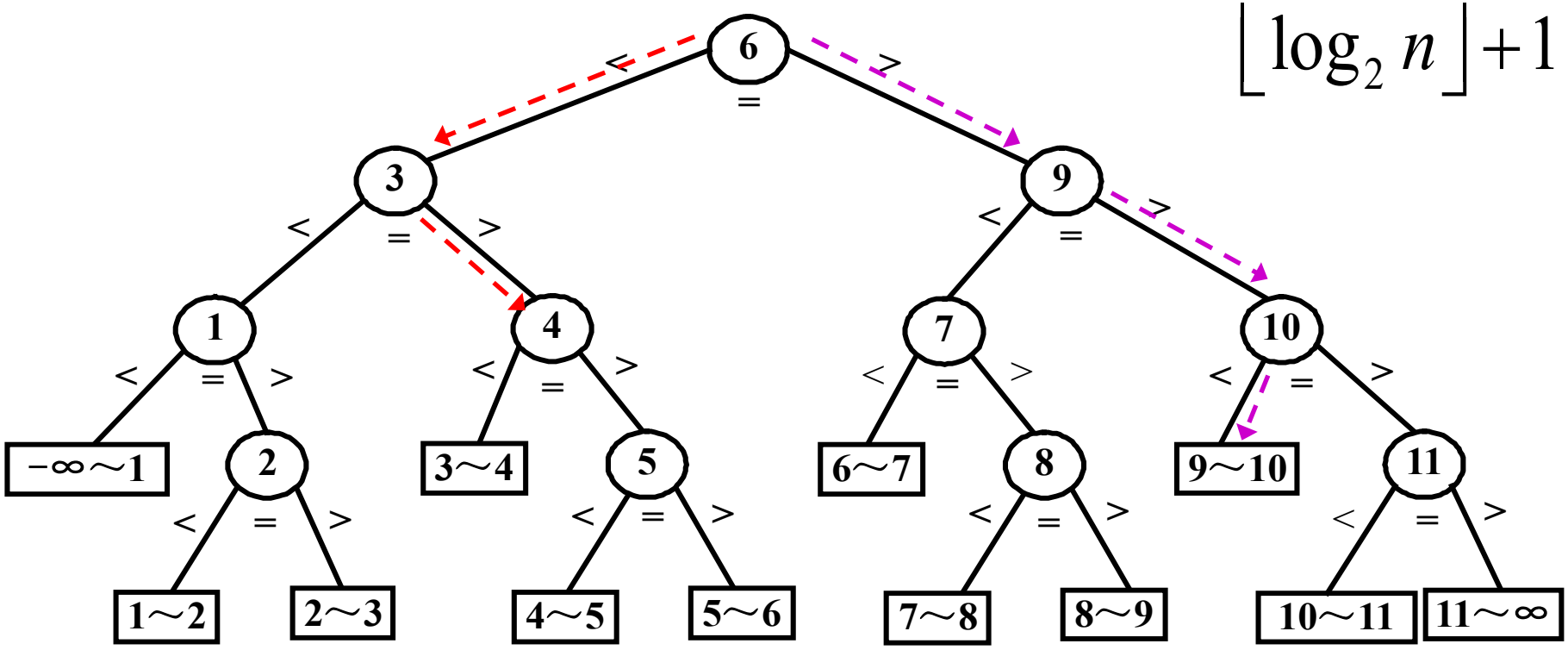
high < low　查找不成功

```
int Search_Bin (SSTable ST,  int key)
{
    low=1;  high=ST.length;
    while( low<=high ) {
        mid = (low+high)/2;
        if (ST.elem[mid].key == key)   /*查找成功返回*/
            return  mid;
        if (ST.elem[mid].key > key)
            high=mid-1;     /*继续在[low..mid-1]中查找*/
        else
            low=mid+1;     /*继续在[mid+1..high]中查找*/
    }
    return 0;
}
```

判定树(比较树)：二分查找过程可用二叉树来描述,把当前查找区间的中间位置上的记录作为根,左子表和右子表中的记录分别作为根的左子树和右子树。

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

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



●折半查找成功时的平均查找长度**ASL**

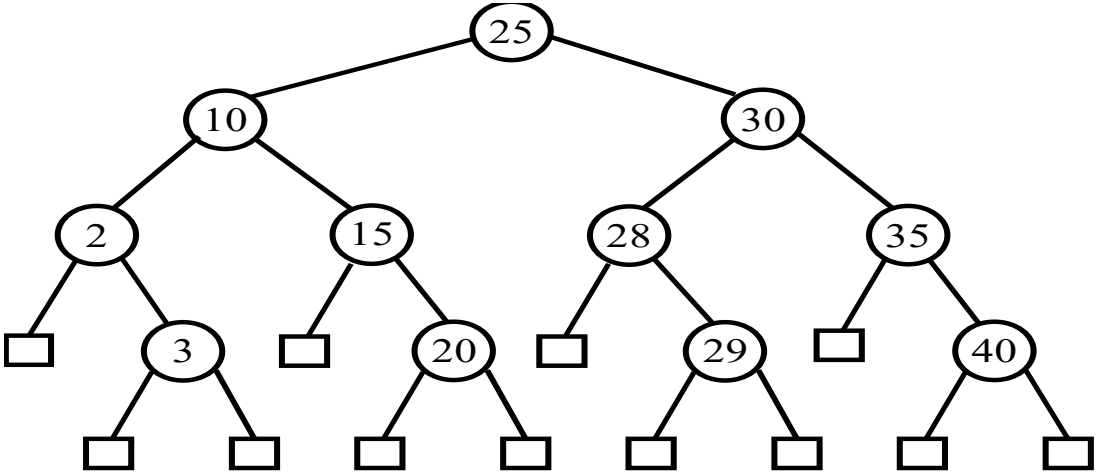
假定表的长度$n=2^h-1$，则相应判定树必为深度是h的满二叉树，$h=\log_2(n+1)$。又假设每个记录的查找概率相等， 则折半查找成功时的平均查找长度为：

$$ASL_{bs}=\sum_{i=1}^{n}P_iC_i=\frac{1}{n}\sum_{j=1}^{h}j\times 2^{j-1}=\frac{n+1}{n}\log_2(n+1)-1$$

当n较大(n>50)时，有近似结果：

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

折半查找判定树



(1)在查找成功时,会找到图中某个圆形结点,则成功时的平均查找长度:

$$ASLsucc=\frac{1\times1+2\times2+4\times3+4\times4}{11}=3$$

(2)在查找不成功时, 会找到图中某个方形结点, 则不成功时的平均查找长度:

$$ASLunsucc=\frac{4\times3+8\times4}{12}=3.67$$

3. 索引顺序查找(分块查找)

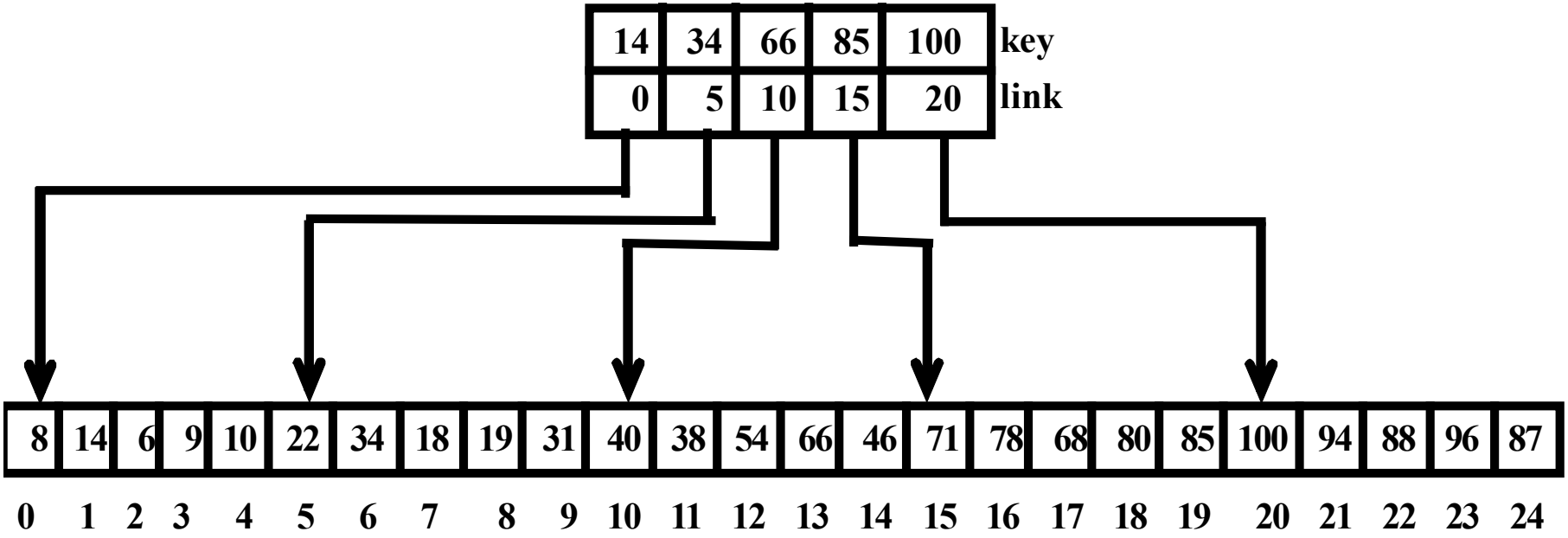
是一种性能介于顺序查找和二分查找之间的查找方法。

将表[1..n]均分为b块, 前b-1块中记录个数为s=⌈n/b⌉, 最后一块即第b块的记录数小于等于s；

每一块中的关键字不一定有序，但前一块中的最大关键字必须小于后一块中的最小关键字，即要“分块有序”。

抽取各块中的最大关键字及其起始位置构成一个索引表ID[b]。由于表R[n]是分块有序的，所以索引表是一个递增有序表。

例如,设有一个线性表,其中包含25个记录,其关键字序列为{8,14,6,9,10,22,34,18,19,31,40,38,54,66, 46,71,78,68,80,85, 100, 94,88,96,87}。假设将25个记录分为5块,每块中有5个记录,该线性表的索引存储结构如下图所示。



查找索引表的ASL为：L_B；块内进行顺序查找的ASL为L_W。

$$ASL_{bs}=L_B+L_W$$

b块，每块含s个元素。查找概率相等，则每个索引项的查找概率为1/b，块中每个元素的查找概率为1/s。

- 若用顺序查找法确定待查元素所在的块，则有

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

- 若用折半查找法确定待查元素所在的块，则有

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

- 静态查找表的三种查找方法的比较

- 顺序查找对对于表有序、无序均适用；折半查找仅适用于有序表；分块查找要求表分块后“分块有序”。
- 从表的存储结构上看，顺序查找和分块查找对于表的顺序和链式存储结构均适用，而折半查找只适用于顺序存储结构。
- 平均查找长度**ASL**而言，折半最小（$\log_2(n+1)-1$），分块次之（$\sqrt{n+1}$），顺序最大（$(n+1)/2$）。

- 作业：

  9.2　9.3　9.25

9.2 动态查找表

动态查找表的特点：表结构本身在查找过程中动态生成，即对于给定值key，若表中存在关键字等于key的记录，则查找成功，否则插入关键字等于key的记录。

二叉排序树、 平衡二叉排序树和B树等。

1. 二叉排序树BST (Binary Sort Tree) 的定义

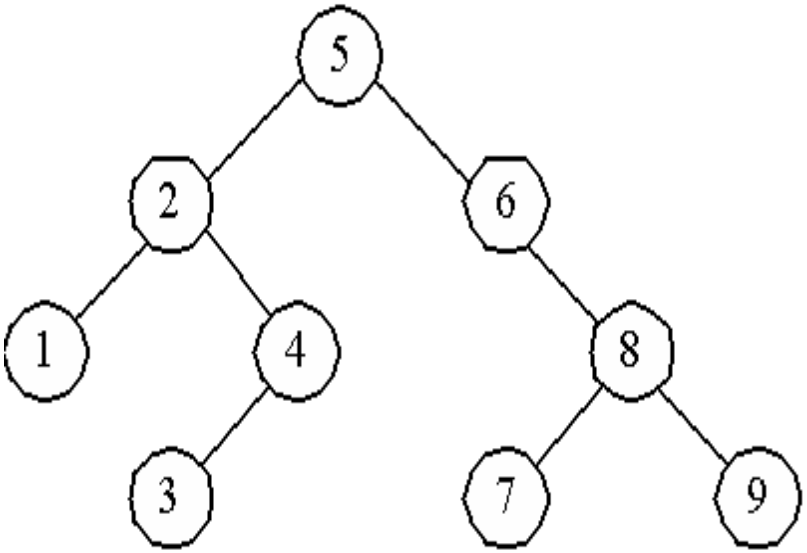
或者是一棵空树， 或者是具有如下性质的二叉树：

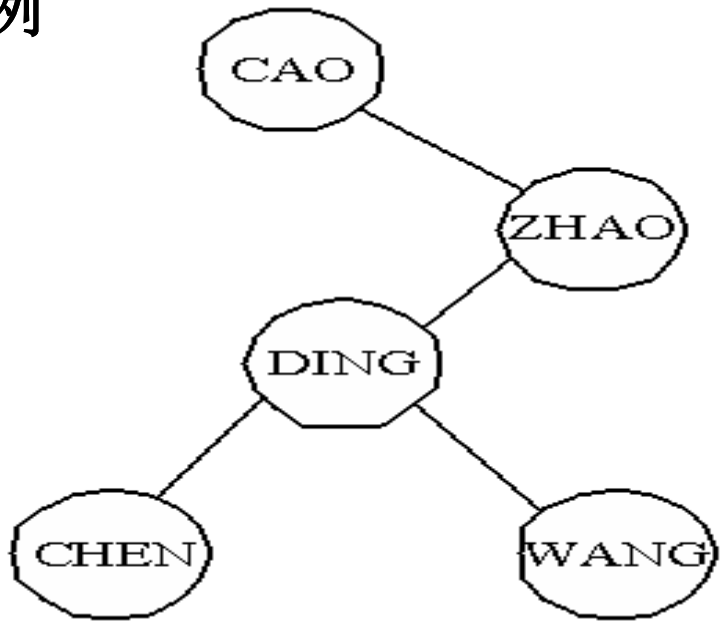
(1)若它的左子树不空， 则左子树上所有结点的值均小于根结点的值；

(2)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；

(3)它的左右子树也分别为二叉排序树。

二叉排序树示例

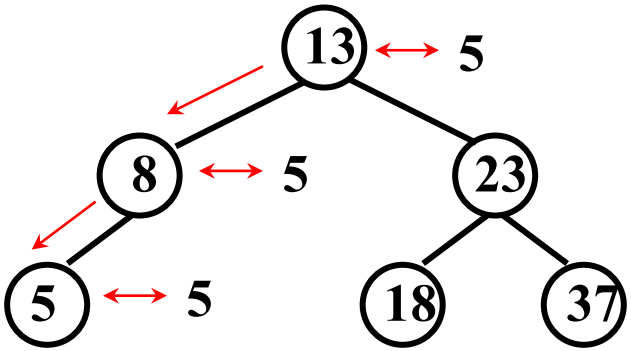





```
typedef struct BiTNode{                          /*记录类型*/
        KeyType key;                             /*关键字项*/
        InfoType data;                           /*其他数据域*/
        struct BiTNode *lchild,*rchild; /*左右孩子指针*/
} BiTNode, *BiTree;
```

查询操作:

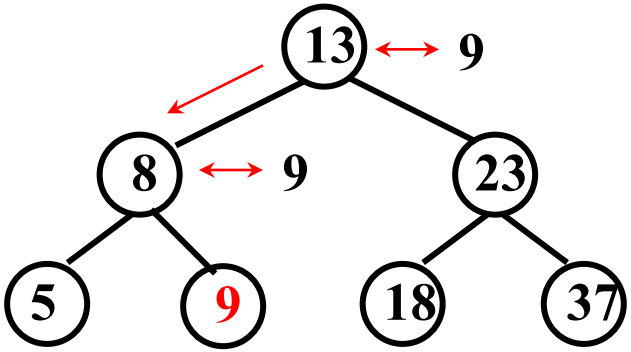
如何查找元素 5 ?



查找成功!

查询+插入操作：

如何查找元素 9？



查找不成功，执行插入。

2. 二叉排序树上的查找

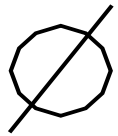
二叉排序树可看做是一个有序表，在二叉排序树上进行查找，和折半查找类似，也是一个逐步缩小查找范围的过程。

```
BiTree SearchBST(BiTree T,  KeyType key)
{
   if (!T || EQ(key, T->key))          /*递归终结条件*/
       return T;
   else if (LT(key, T->key))
       return SearchBST(T->lchild, key); /*在左子树中递归查找*/
   else
       return SearchBST(T->rchild, key); /*在右子树中递归查找*/
}
```

```
BiTree SearchBST(BiTree T,  KeyType key)
{
    p = T;
    while(①   p ){
        if(p->key == key)
         ②  return p;
        if(p->key > key)
         ③  p = p->lchild;
        else
         ④  p = p->rchild;
    }
    return NULL;
}
```

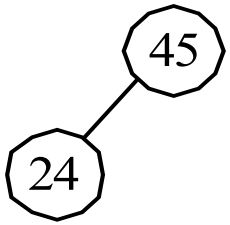
3. 二叉排序树的插入和生成

关键字序列：**{45, 24, 53, 45, 12, 28, 90}**

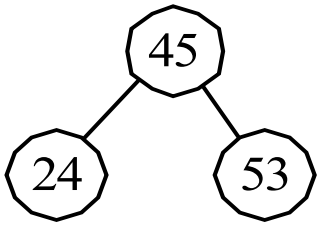


(a) 空树

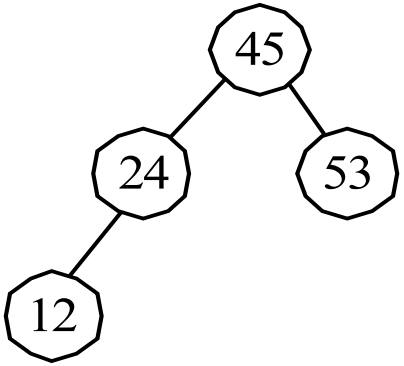
(b) 插入45



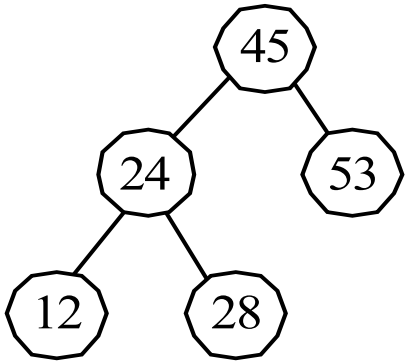
(c) 插入24



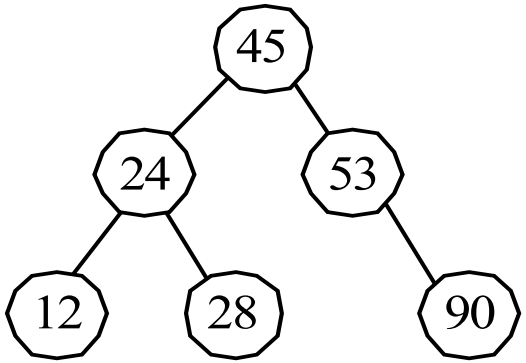
(d) 插入53



(e) 插入12



(f) 插入28



(g) 插入90

●在以T为根结点的BST中插入一个关键字为key的结点。插入成功返回1,否则返回0.

```
int InsertBST(BiTree &T, KeyType key)
{    if (T==NULL){    /*原树为空，新插入的记录为根结点*/
        T = (BiTree)malloc(sizeof(BiTNode));
        T->key=key; T->lchild=T->rchild=NULL; return 1;
     }
     else if (key == T->key) /*存在相同关键字的结点, 返回0*/
        return 0;
     else if(key < T->key)
        return InsertBST(T->lchild, key);/*插入到左子树中*/
     else
        return InsertBST(T->rchild, key);  /*插入到右子树中*/
}
```

●二叉排序树的生成, 是从一个空树开始, 每插入一个关键字, 就调用一次插入算法将它插入到当前已生成的二叉排序树中。关键字数组A[0..n-1].

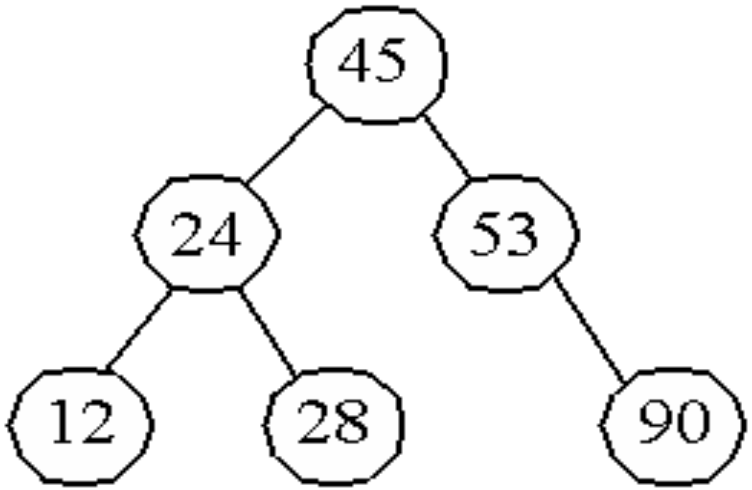
```
BiTree CreatBST(KeyType A[],  int n) /*返回树根指针*/
{       BiTree  T = NULL;    /*初始时T为空树*/
        int i=0;
        while (i<n){
            InsertBST(T, A[i]);  /*将A[i]插入二叉排序树T中*/
            i++;
        }
        return T;           /*返回建立的二叉排序树的根指针*/
}
```

(1)新插入的结点一定是一个新的叶子结点；

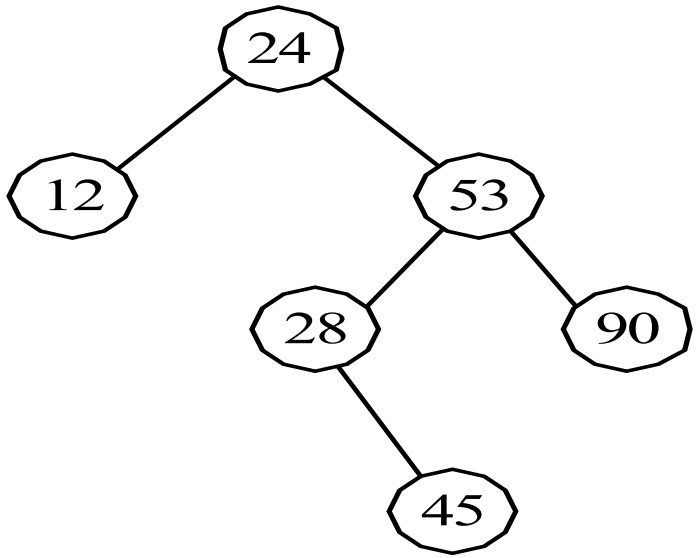
(2)中序遍历二叉排序树，得到一个关键字有序序列，树排序；

(3)二叉排序树的形态完全由一个输入序列确定；

{45, 24, 53, 45, 12, 28, 90}



{24, 53, 12, 28, 45, 90 45}



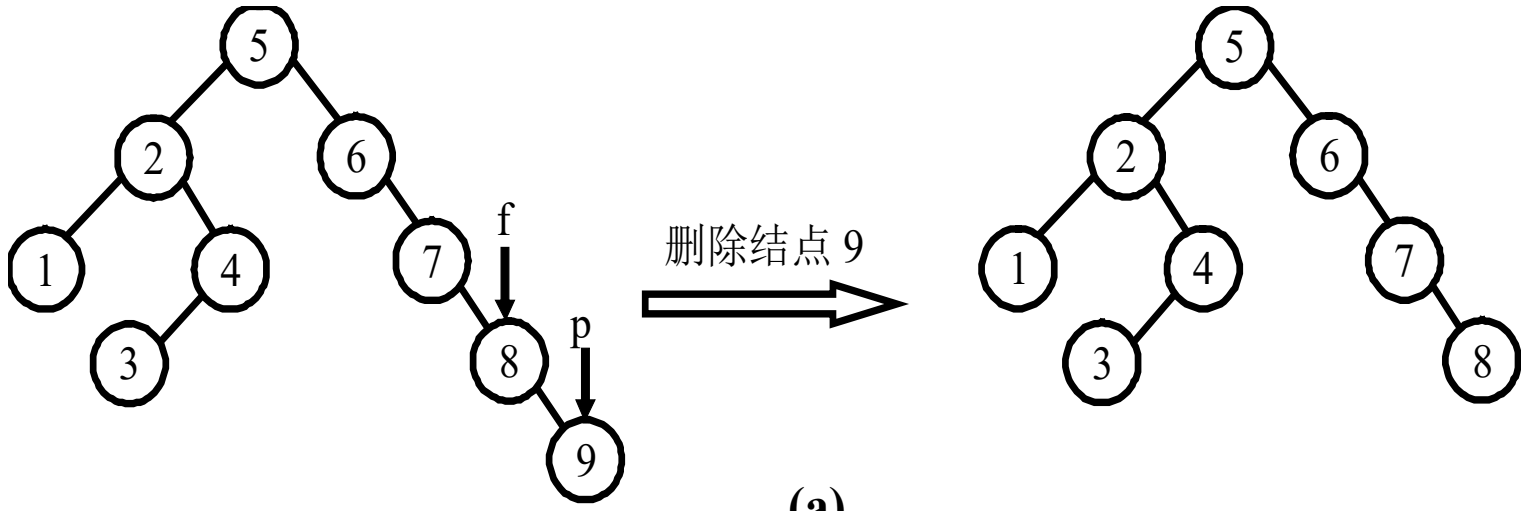
4. 二叉排序树的删除

在二叉排序树中删去一个结点相当于删去有序序列中的一个结点。假设要删除的结点为p，结点p的双亲结点为f。

下面分三种情况讨论：

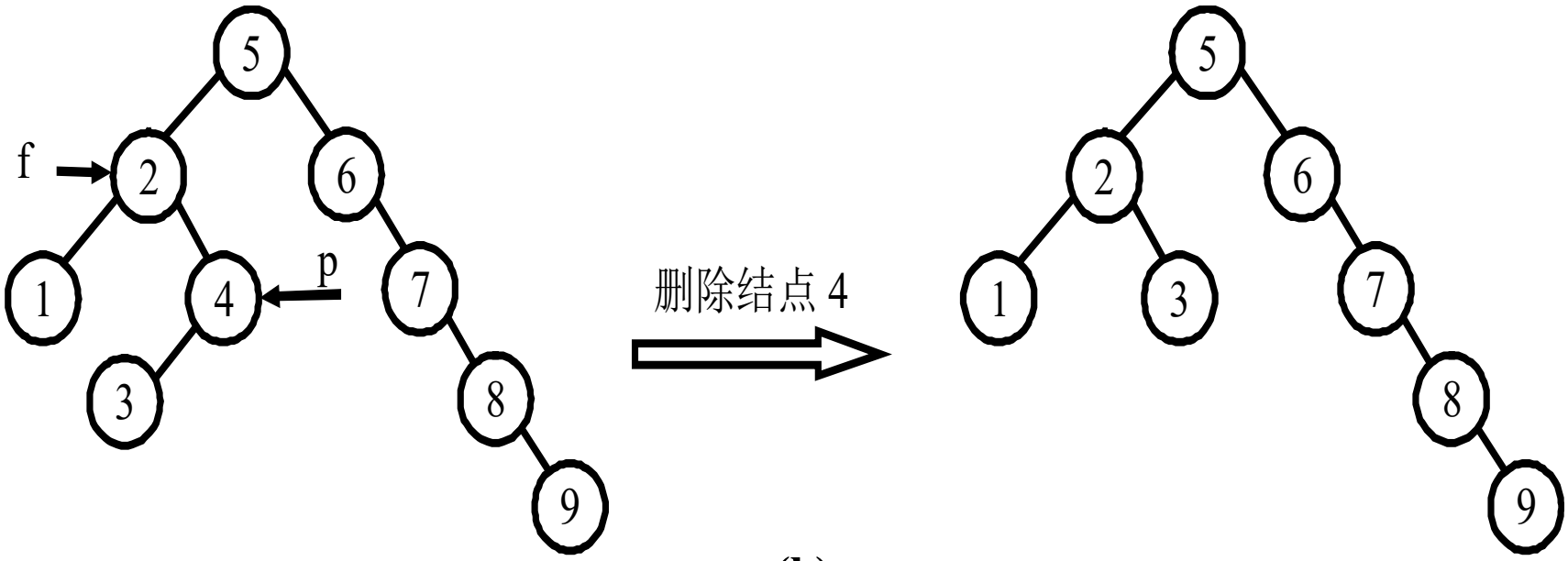
(1)若p为叶子结点，则可直接将其删除:

f->rchild=NULL(或f->lchild=NULL)； free(p);



(a)

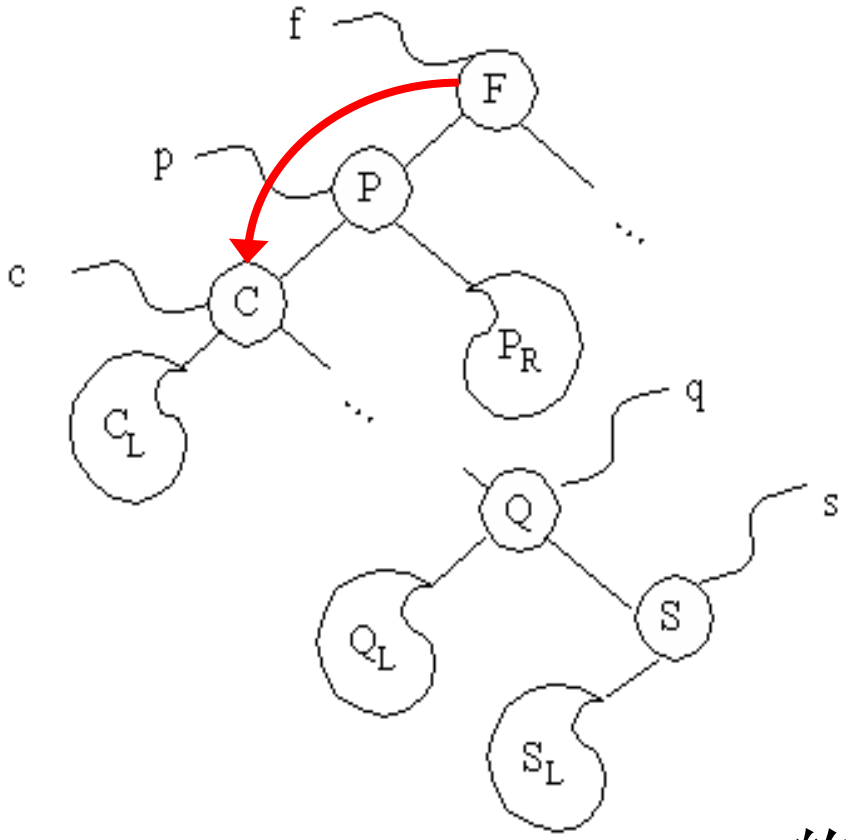
(2) 若**p**结点只有左子树，或只有右子树，则可将**p**的左子树或右子树直接改为其双亲结点**f**的左(右)子树，即：**f->rchild=p->lchild**（或**f->rchild=p->rchild**）； **free(p)**；

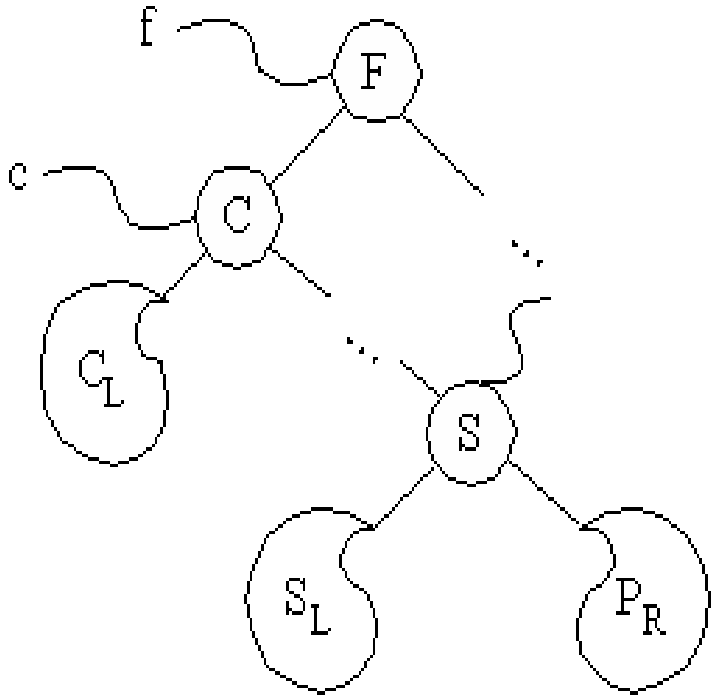


(b)

(3) 若p既有左子树，又有右子树，设p为双亲f的左孩子。 此时有两种处理方法：

方法1：将p的左子树改为f的左子树，而将p的右子树改为s的右子树。

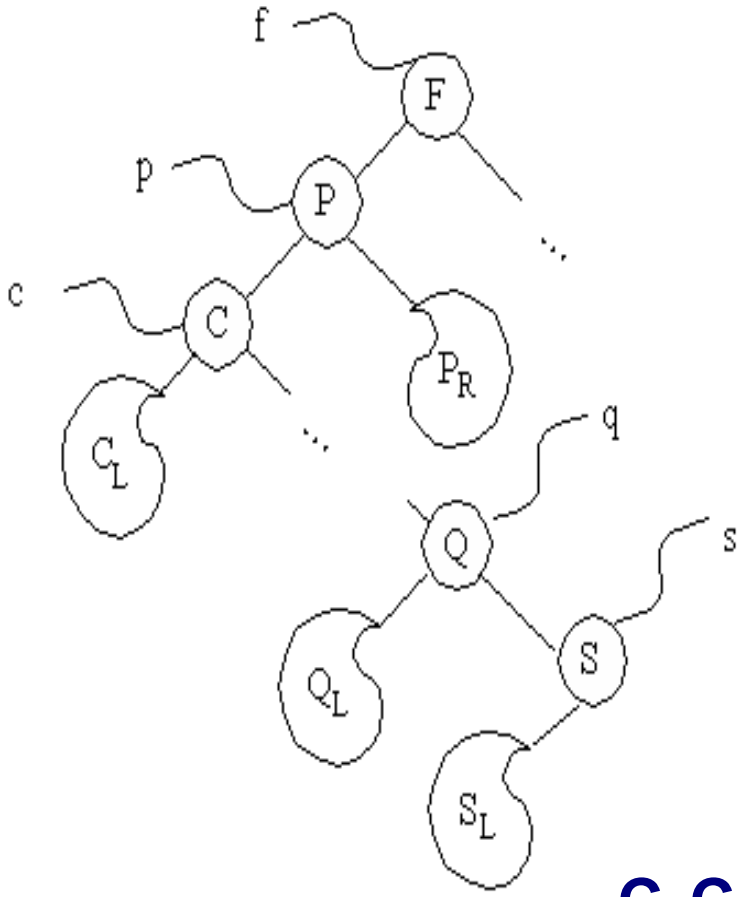


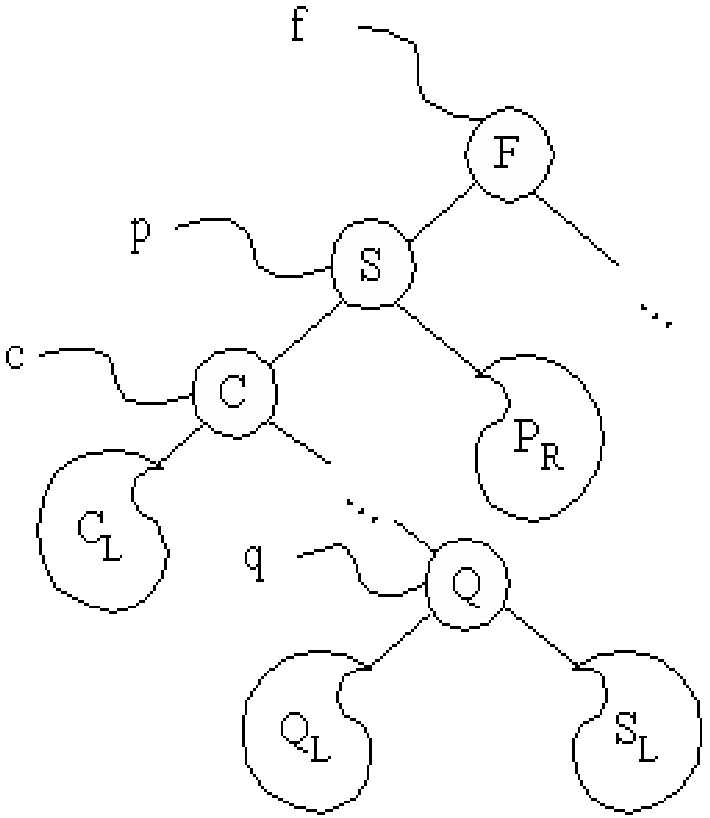


$C_LC\ldots Q_LQS_LSPP_RF$

p的直接前驱在其左子树最右下的结点

方法2：用s结点的值替代p结点的值，再将s结点删除，原s结点的左子树改为s的双亲结点q的右子树。

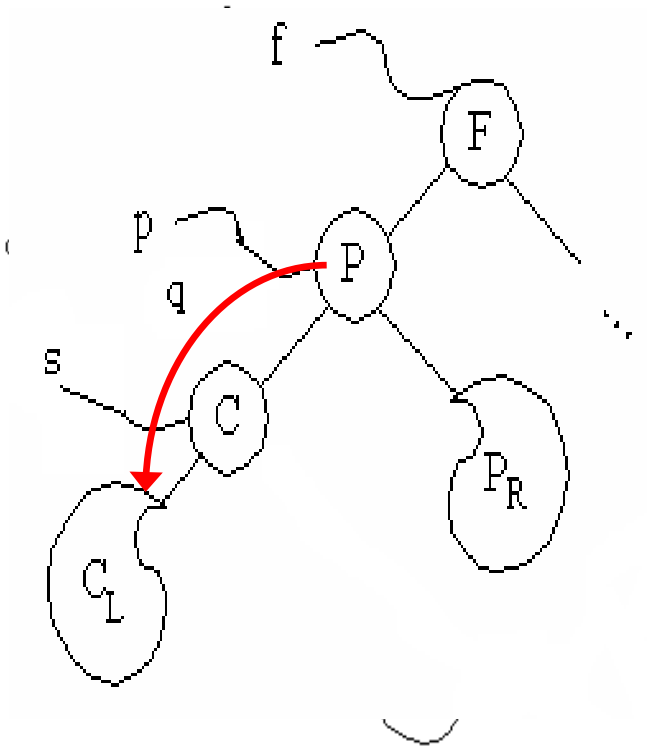




$C_LC\cdots Q_LQS_LSPP_RF$

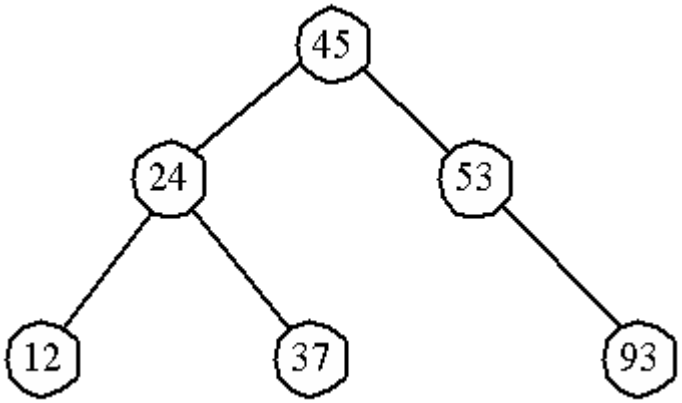
```
Status DeleteBST(BiTree &T, KeyType key)
{
    if( !T )
        return FALSE;
    else{
       if(T->key == key)
           return Delete(T);
       else if(T->key > key)
           return DeleteBST(T->lchild, key);
       else
           return DeleteBST(T->rchild, key);
         }
}
```

```
void  Delete( BiTree &p)
{
   if( !p->rchild)  {   q = p; p = p->lchild; free(q);  }//右子树空
   else if( !p->lchild) { q = p; p = p->rchild; free(q); }//左子树空
   else{//左右子树均不空
      q = p;  s = p->lchild;
      while( s->rchild)
          { q = s; s = s->rchild; }
      p->data = s->data;
      if(q!=p)    q->rchild = s->lchild;
      else q->lchild = s->lchild;
      free(s);
   }
}
```



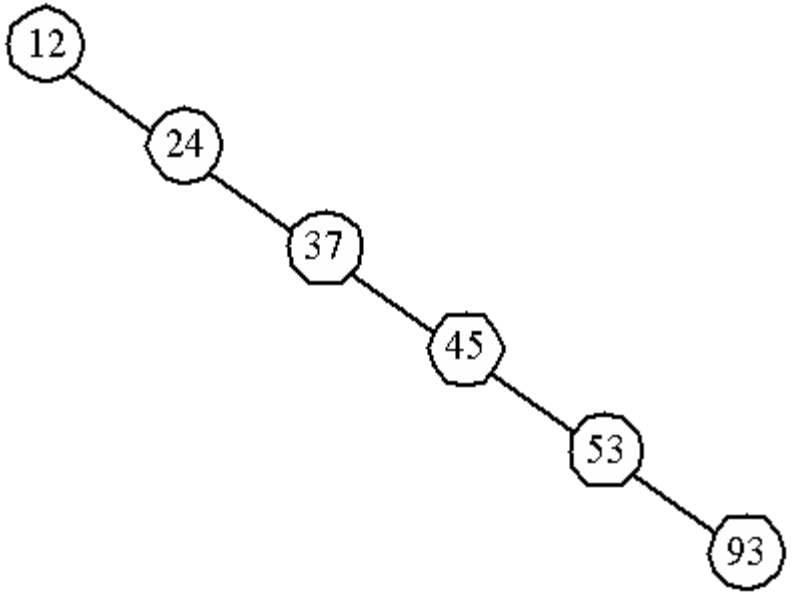
5. 二叉排序树的查找分析

45, 24, 53, 12, 37, 93



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

12, 24, 37, 45, 53, 93

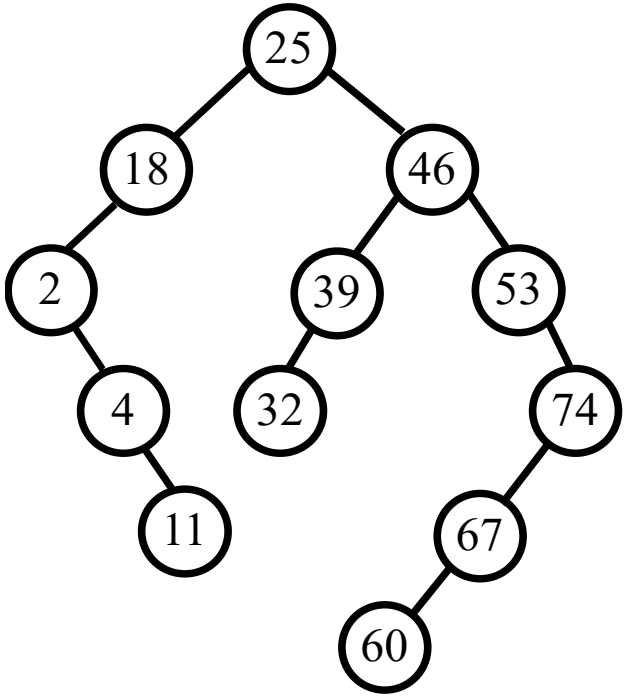


ASL=(1+2+3+4+5+6)/6

最差情况：单支树，**ASL**为**(n+1)/2**，和顺序查找相同；

最好情况：与折半查找的判定树形态相同，**ASL**和$\log_2 n$成正比。

例：已知一组关键字为{25,18,46,2,53,39,32,4,74,67,60,11}。按表中的元素顺序依次插入到一棵初始为空的二叉排序树中,画出该二叉排序树；并求在等概率的情况下查找成功时的平均查找长度。



$$ASL = \frac{1\times1+2\times2+3\times3+3\times4+2\times5+1\times6}{12} = 3.5$$

平衡二叉树　Adelson, Velskii和Landis

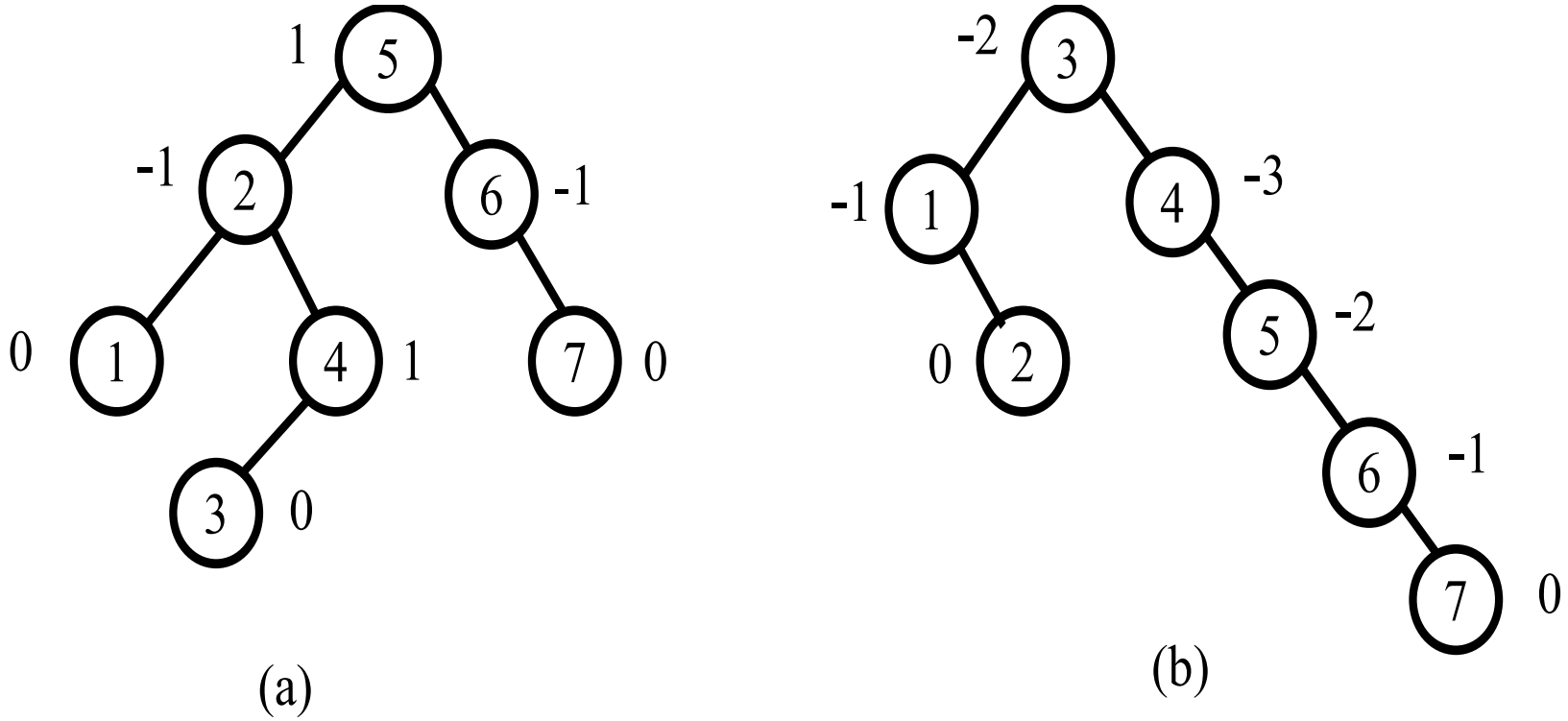
1. 平衡二叉树的定义（又称AVL树）

若一棵二叉排序树中每个结点的左、右子树的高度至多相差1,则称此二叉树为平衡二叉树。

平衡因子BF(Balance Factor)：二叉树中每个结点的平衡因子是该结点左子树的高度减去右子树的高度。

从平衡因子的角度可以说,若一棵二叉树中所有结点的平衡因子的绝对值小于或等于1,即平衡因子取值为1、0或-1,则该二叉树称为平衡二叉树。

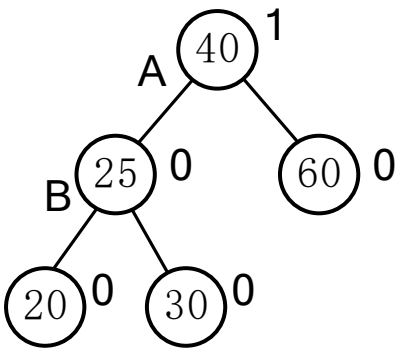
含有n个结点的平衡二叉树的平均查找长度为$O(\log_2 n)$。



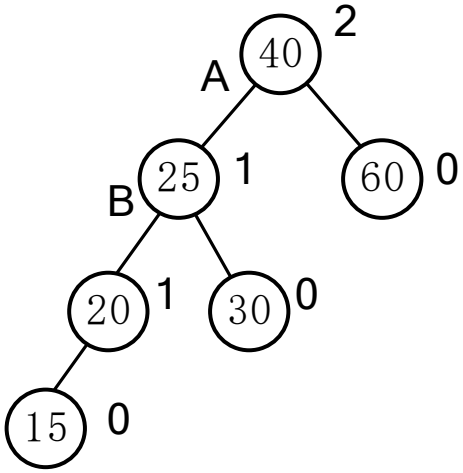
(a)平衡二叉树和(b)不平衡二叉树

最小不平衡子树：离插入结点最近,且|BF|>1的结点作为根的子树。

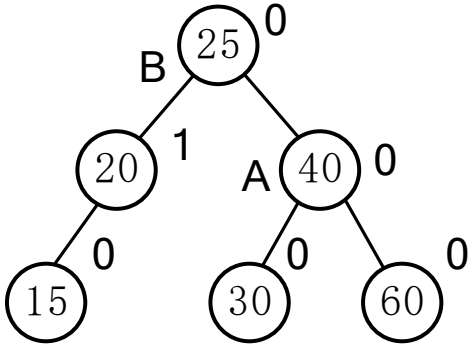
假设用A表示失去平衡的最小不平衡子树的根结点,则调整子树的操作可归纳为下四种情况。



(a) 一棵平衡二叉排序树



(b) 插入15后失去平衡

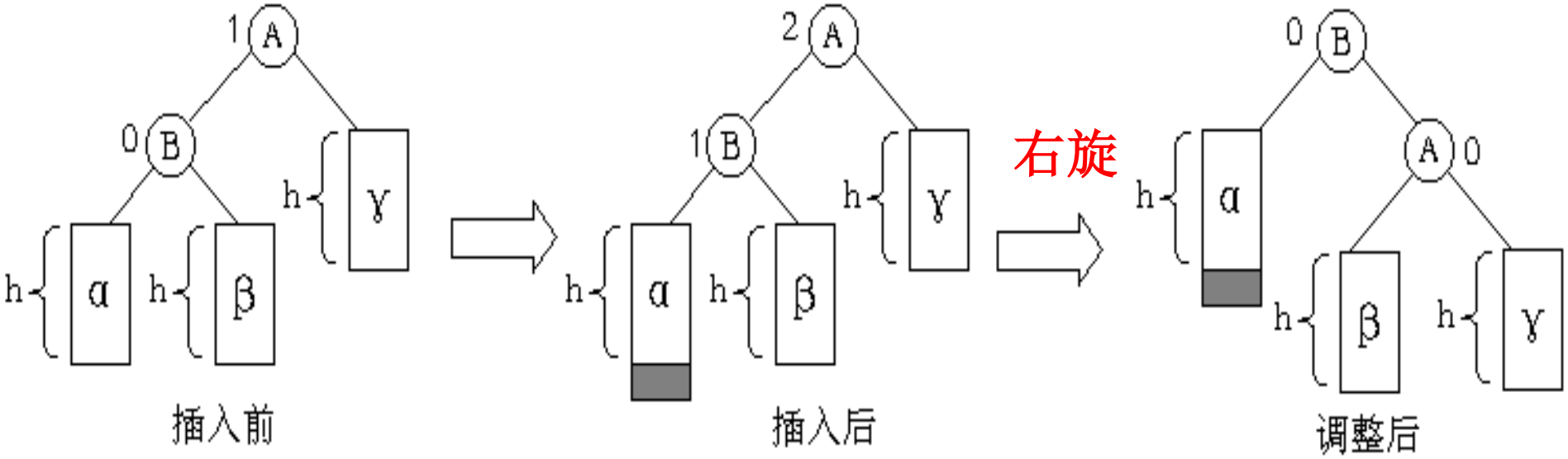


(c) 调整后的二叉排序树

LL型调整

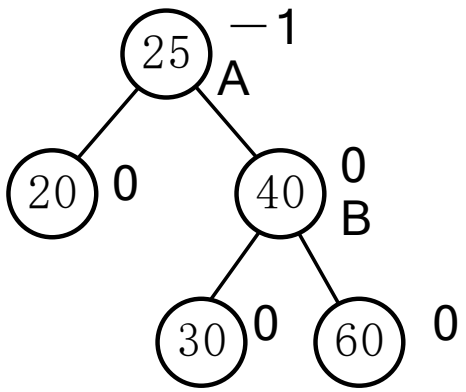
2. 最小不平衡子树的调整 （设最小不平衡子树的根结点为A）

(1) LL型调整：在A的左孩子的左子树上插入结点，使A的BF(1→2）而失去平衡

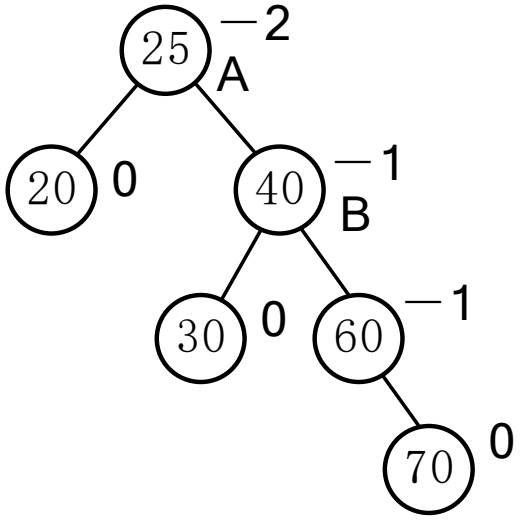


将结点B提升为新的二叉树的根，结点A连同其右子树向右下旋转，使其成为B的右子树，B的原右子树则作为A的左子树。

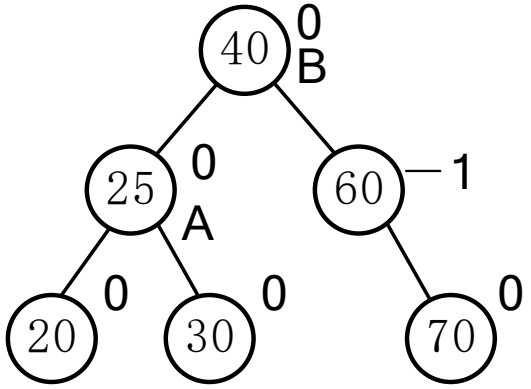
(α)B(βAγ)



(a) 一棵平衡二叉排序树



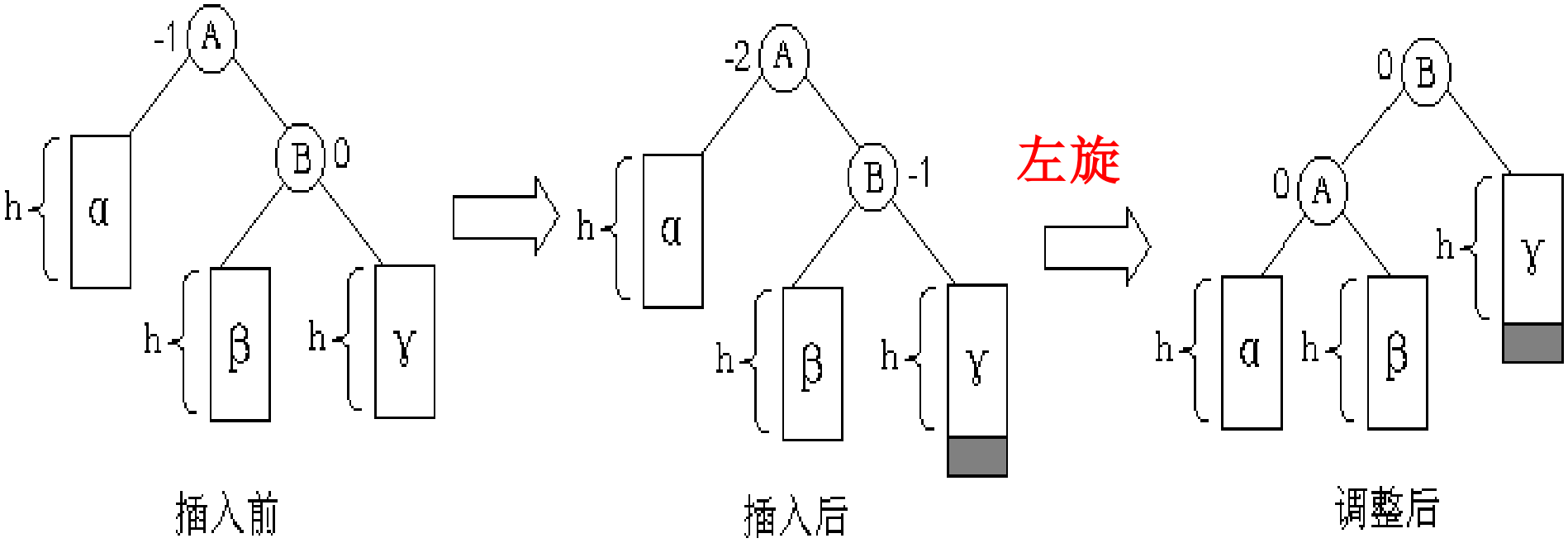
(b) 插入70后失去平衡



(c) 调整后的二叉排序树

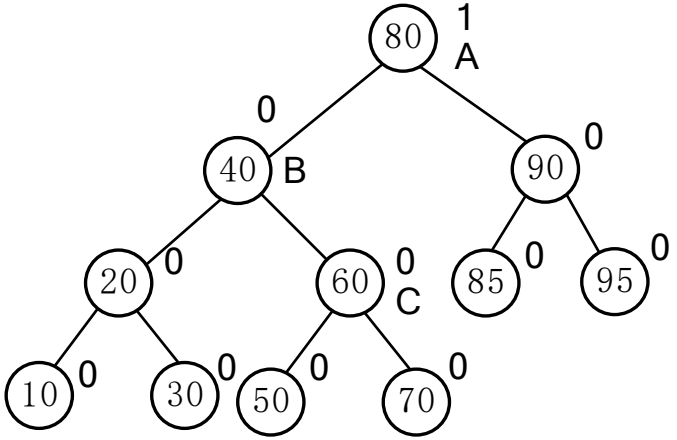
RR型调整

(2)RR型调整：在A的右孩子的右子树上插入结点，使A的BF(-1→-2）而失去平衡

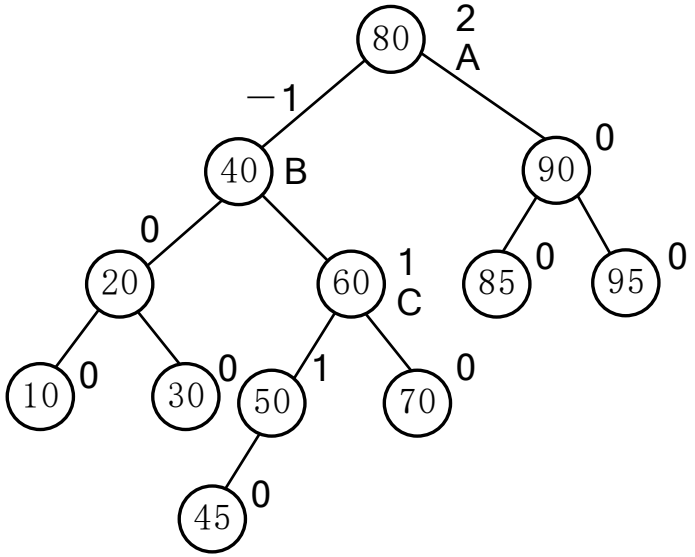


将结点B提升为新的二叉树的根，结点A连同其左子树向左下旋转，使其成为B的左子树，B的原左子树则作为A的右子树。

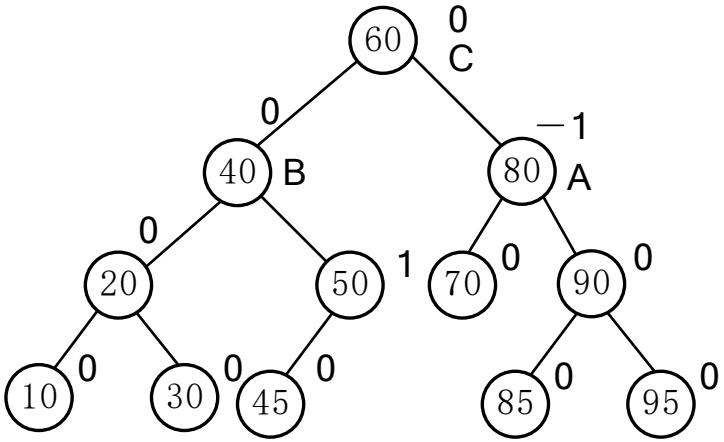
(αAβ)B(γ)



(a) 一棵平衡二叉排序树



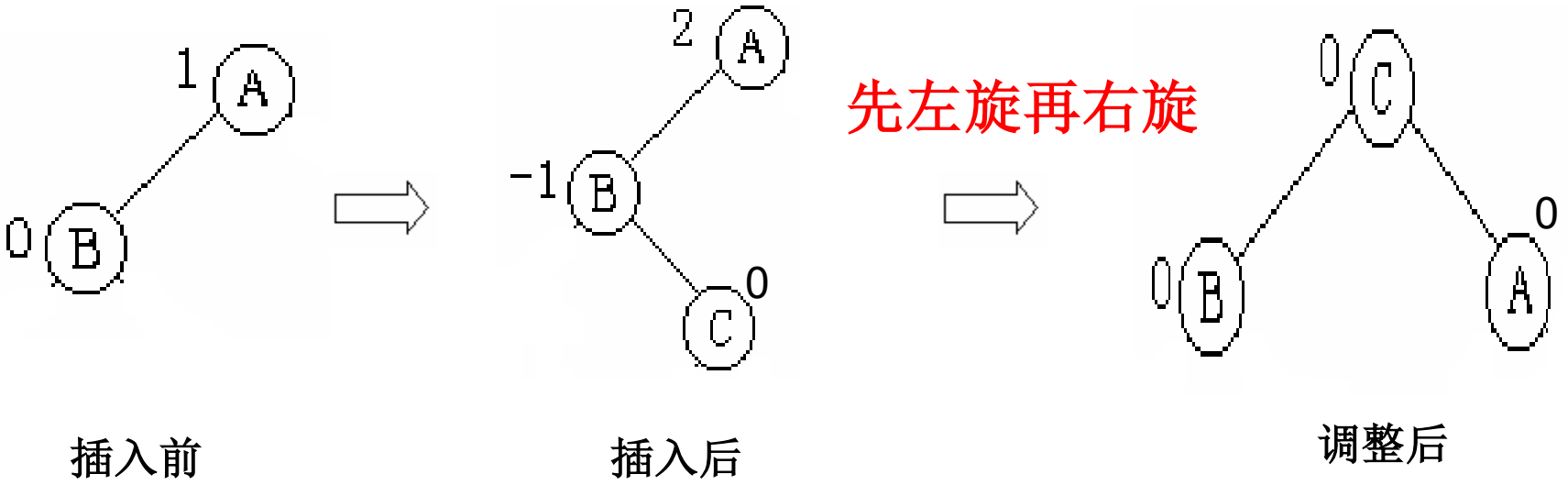
(b) 插入45后失去平衡



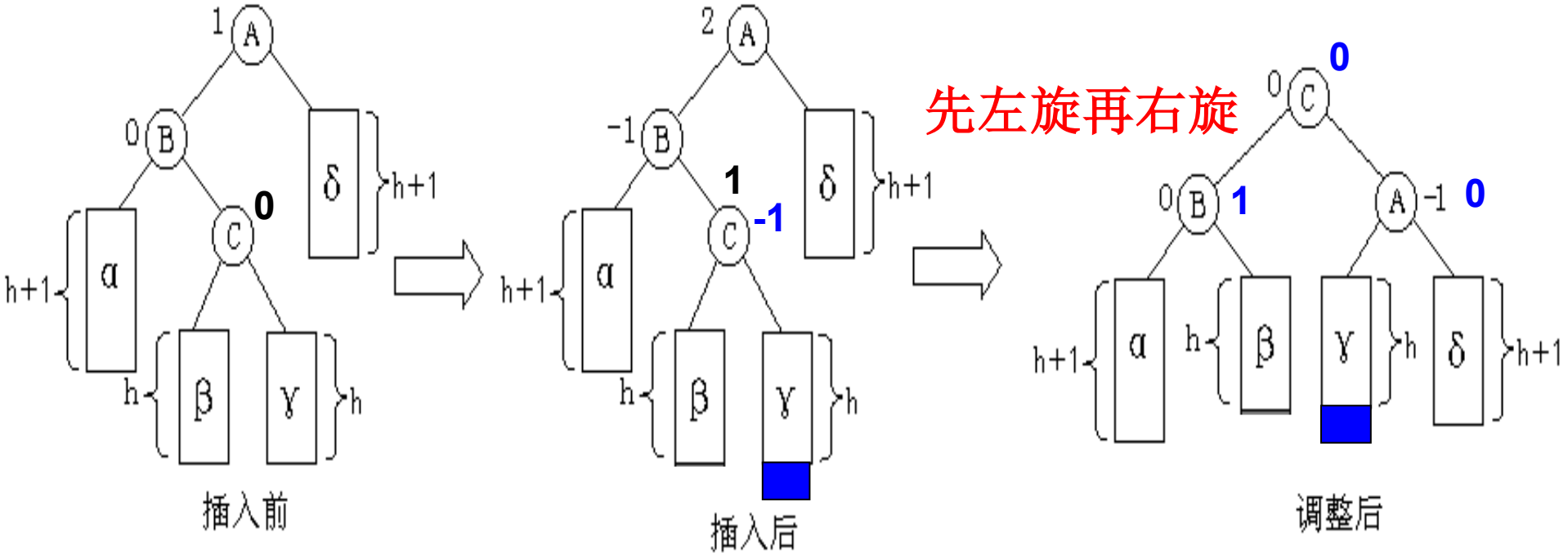
(c) 调整后的二叉排序树

LR型调整

(3)LR型调整：在A的左孩子的右子树上插入结点，使A的BF(1→2）而失去平衡

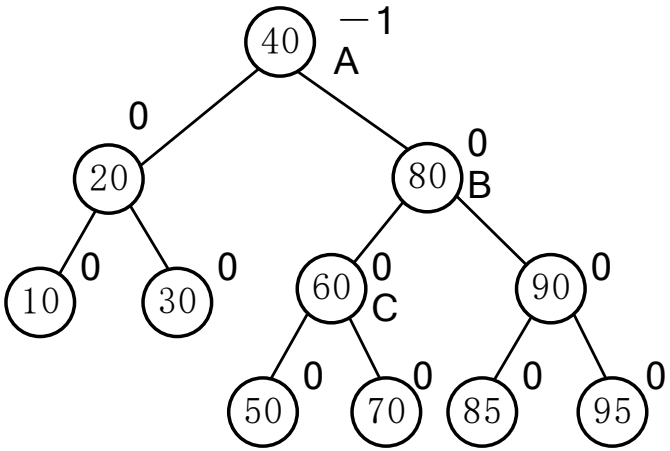


LR(0)调整

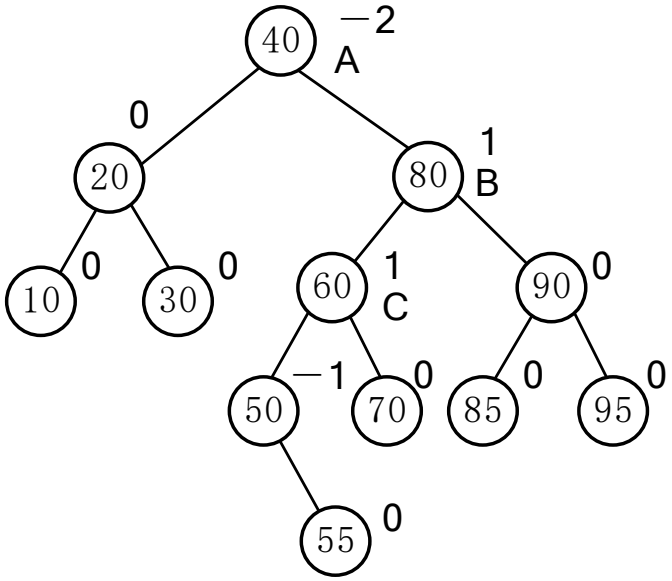


LR(L)型调整

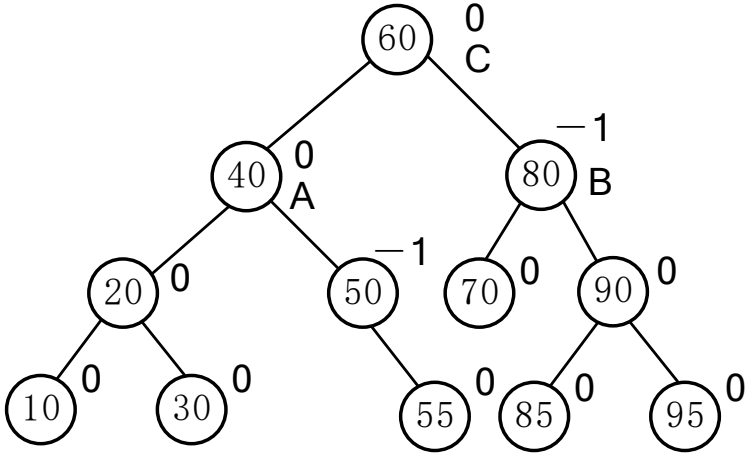
LR(R)型调整



(a) 一棵平衡二叉排序树



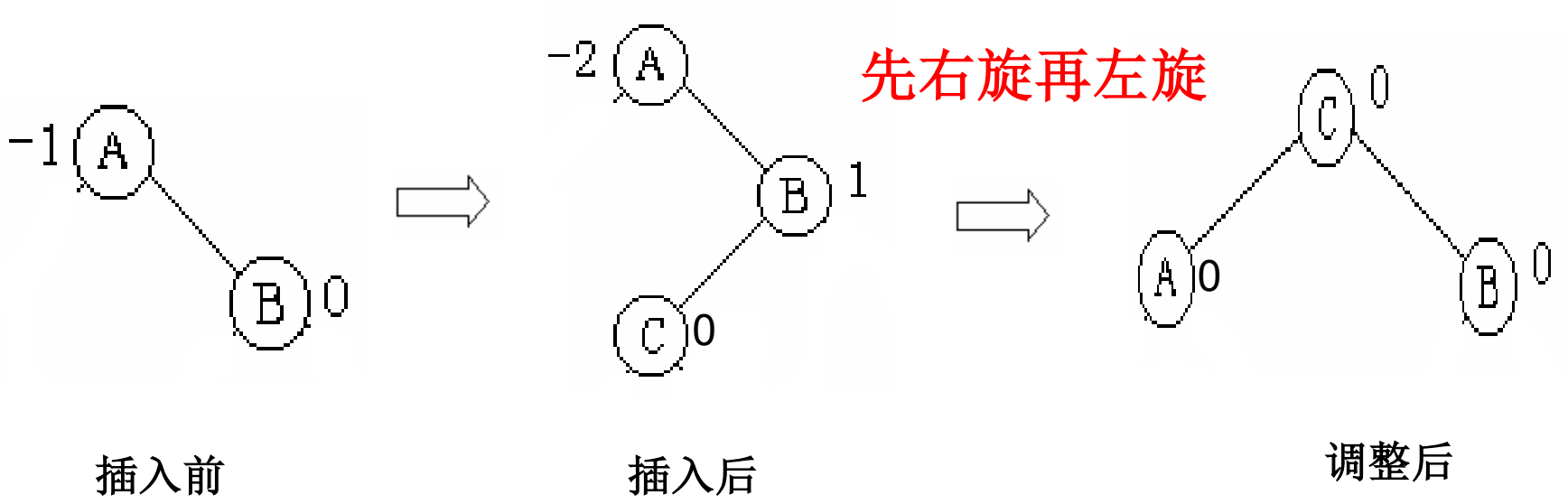
(b) 插入55后失去平衡



(c) 调整后的二叉排序树

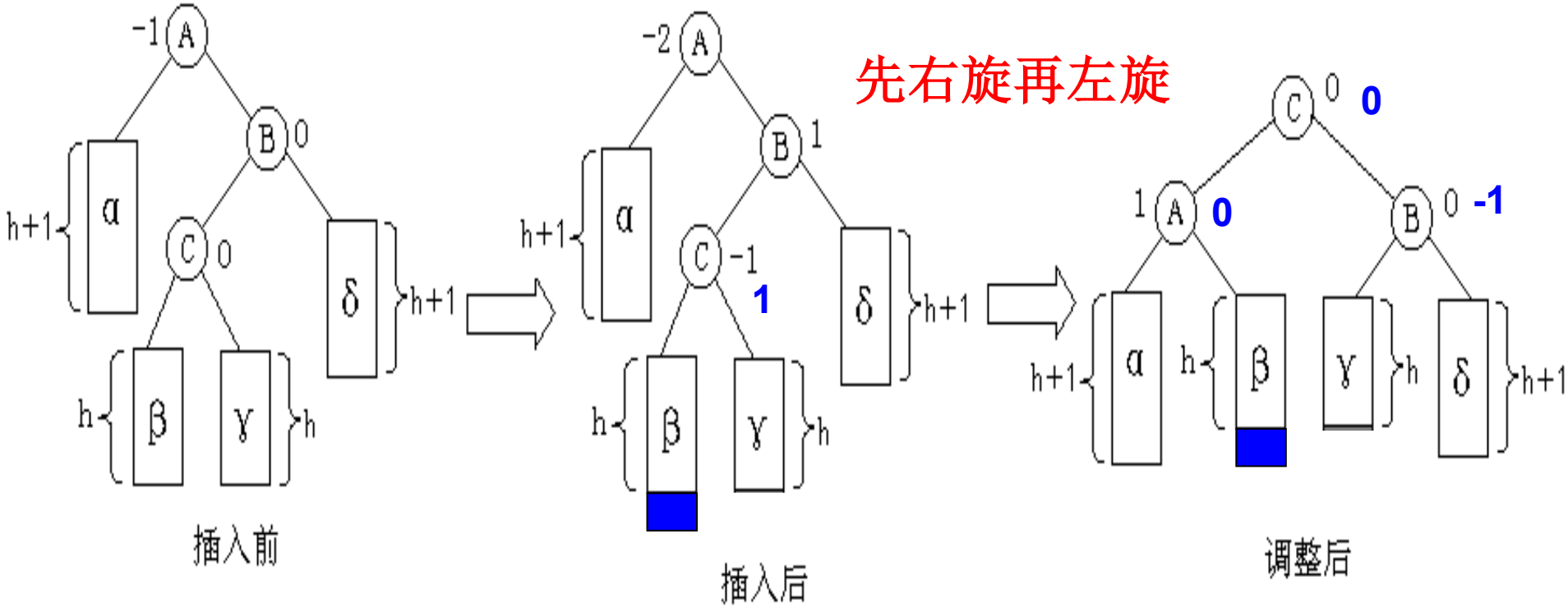
RL型调整

(4) RL型调整：在A的右孩子的左子树上插入结点，使A的BF(-1→-2）而失去平衡



插入前　　插入后　　调整后

RL (0)调整



RL (R)型调整　　**RL (L)型调整**

在一个平衡二叉排序树上插入一个新结点S时，主要包括以下三步：

(1)查找应插位置， 同时记录离插入位置最近的可能失衡结点A。

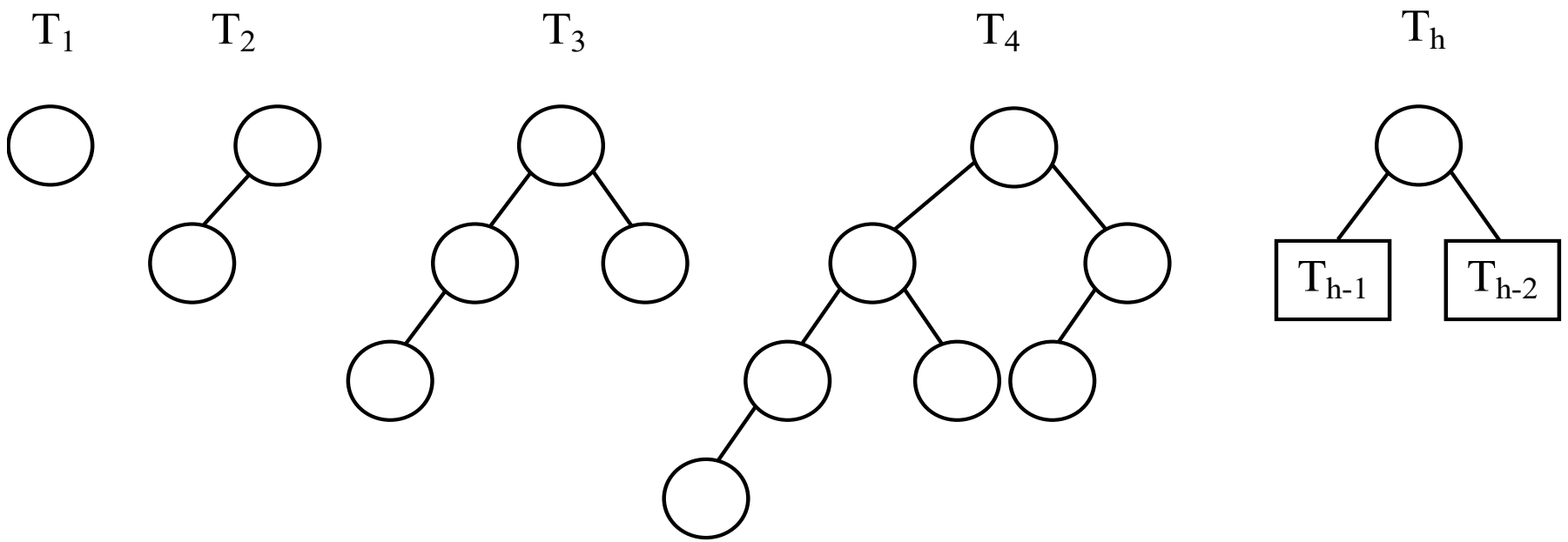
(2)插入新结点S， 并修改从A到S路径上各结点的平衡因子。

(3)根据A、 B的平衡因子， 判断是否失衡以及失衡类型，并做相应处理。

3. 平衡二叉树查找分析

平衡二叉树的高度h和结点个数n之间的关系。

T_1，T_2，T_3，…，其中，T_h（h=1，2，3，…）是高度为h且结点数尽可能少的平衡二叉树。对于每一个T_h，只要从中删去一个结点，就会失去平衡或高度不再是h（显然，这样构造的平衡二叉树在结点个数相同的平衡二叉树中具有最大高度）



设N(h)为T_h的结点数（最少结点数），有下列关系成立：

N(1)=1，N(2)=2，N(h)=N(h-1)+N(h-2)+1

当h>1时，此关系类似于定义Fibonacci数的关系：

F(1)=1，F(2)=1，F(h)=F(h-1)+F(h-2)

通过检查两个序列的前几项就可发现两者之间的对应关系：

N(h)=F(h+2)-1

由于Fibonacci数满足渐近公式：$F(h)=\frac{1}{\sqrt{5}}\varphi^{h}$

其中，$\varphi=\frac{1+\sqrt{5}}{2}$

故由此可得近似公式：$N(h)=\frac{1}{\sqrt{5}}\varphi^{h+2}-1\approx 2^{h}-1$

即：$h\approx\log_2(N(h)+1)$

所以，含有n个结点的平衡二叉树的平均查找长度为$O(\log_2 n)$。

- 作业：

  9.9

B-树

1. B-树的定义

B-树又称为多路平衡查找树，是一种组织和维护外存文件系统非常有效的数据结构。

一棵m阶B-树或者是一棵空树,或者是满足下列要求的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，则根结点至少有两棵子树；

(3)除根结点之外的所有非终端结点至少有$\lceil m/2 \rceil$棵子树；

(4)所有的非终端结点中包含下列信息数据

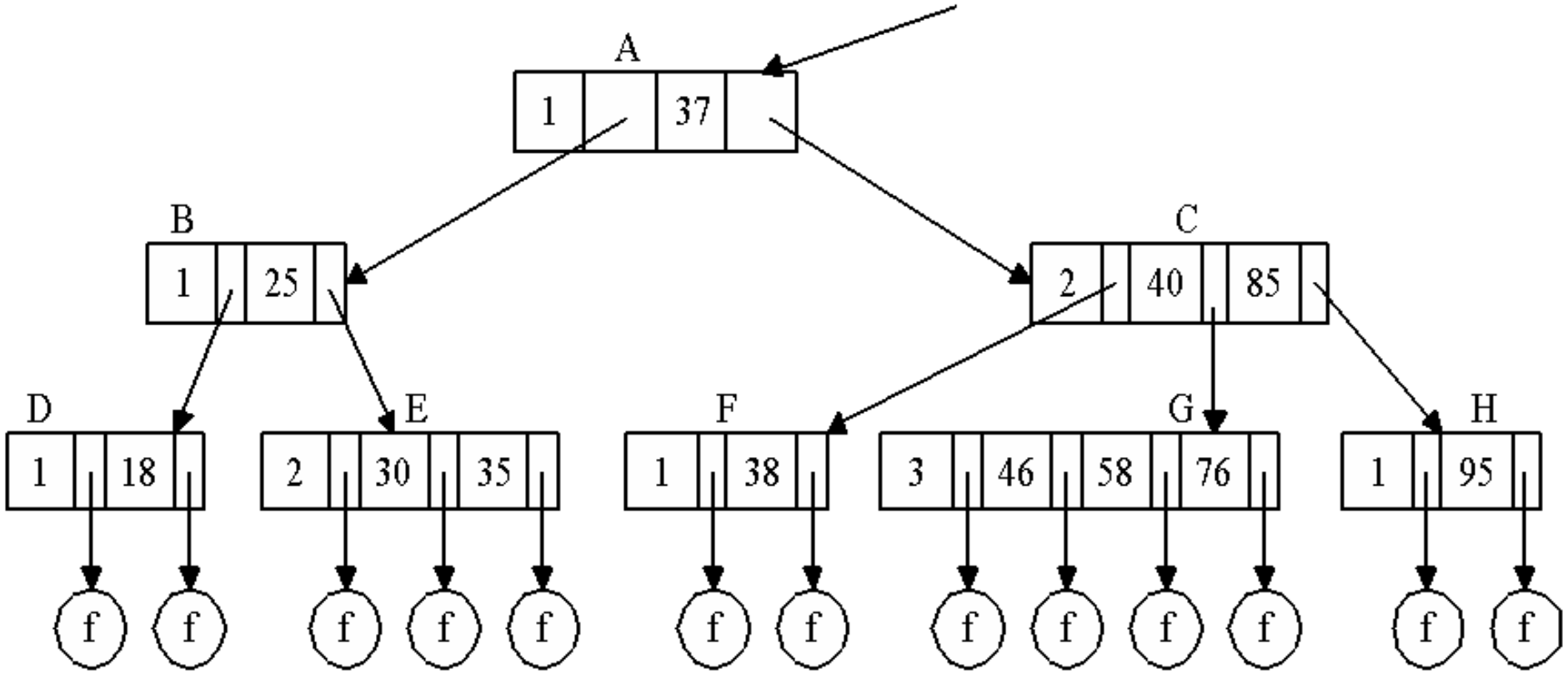
$(n, A_0, K_1, A_1, K_2, A_2, \ldots, K_n, A_n)$

n：关键字个数，除根结点外,其它所有结点的n($\lceil m/2 \rceil -1 \le n \le m-1$)；

K_i：关键字，满足$k_i < k_{i+1}$；

A_i：指向子树根结点的指针，指针A_{i-1}所指子树中所有结点关键字均小于等于K_i，A_n所指子树中所有结点的关键字均大于K_n。

(5) 所有叶子结点都在同一层上,并且不带信息（可看作外部结点或查找失败的结点）。



一棵4阶B-树

2. B-树的查找

①在树中找结点（在磁盘上进行的，内外存交换）

②在结点中找关键字（在内存中进行的）

在每个记录上确定向下查找的路径不一定是二路(即二叉)的,而是n+1路的。

将k与根结点中的key[i]进行比较：

(1) 若k=key[i],则查找成功；

(2) 若k＜key[1],则沿着指针A[0]所指的子树继续查找；

(3) 若key[i]＜k＜key[i+1],则沿着指针A[i]所指的子树继续查找；

(4) 若k＞key[n],则沿着指针A[n]所指的子树继续查找。

●B-树的查找过程是一个顺指针查找结点和在结点的关键字中进行查找交叉进行的过程。因此，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，必须加以权衡。

●在B-树上进行查找，查找成功所需的时间取决于关键字所在的层次，查找不成功所需的时间取决于树的高度。需要了解树的高度h 与树中的关键字个数 N 之间的关系。

3. B-树查找分析

在磁盘上查找的次数，即待查关键字所在结点在B-树上的层次数，是决定B-树查找效率的首要因素。

含N个关键字的m阶B-树的最大深度是多少？

3阶的B-树（2-3树）：

(1) h=2, 1≤N≤2;

(2) h=3, 3≤N≤8;

(3) h=4, 7≤N≤26;

(4) h=5, 15≤N≤80;

深度为h+1的m阶B-树所具有的最少结点数？

- 设在 m 阶B-树中，失败（叶）结点位于第 $h+1$ 层。在这棵B-树中关键字个数 N 最小能达到多少？ 从B-树的定义知,
  - 1层　1 个结点；
  - 2层　至少 2 个结点；
  - 3层　至少 $2\lceil m/2 \rceil$ 个结点；
  - 4层　至少 $2\lceil m/2 \rceil^2$ 个结点
  - 如此类推，……；
  - h 层 至少有 $2\lceil m/2 \rceil^{h-2}$ 个结点；
  - $h+1$ 层 至少有 $2\lceil m/2 \rceil^{h-1}$ 个结点，查找失败的(叶)结点。

$$N+1 \geq 2\lceil m/2 \rceil^{h-1} \rightarrow h \leq \log_{\lceil m/2 \rceil}((N+1)/2)+1$$

若B-树的阶数 $m = 199$，关键字总数 $N = 1999999$，则B-树的高度 h 不超过 $\log_{100} 1000000 + 1 = 4$

4. B-树的插入

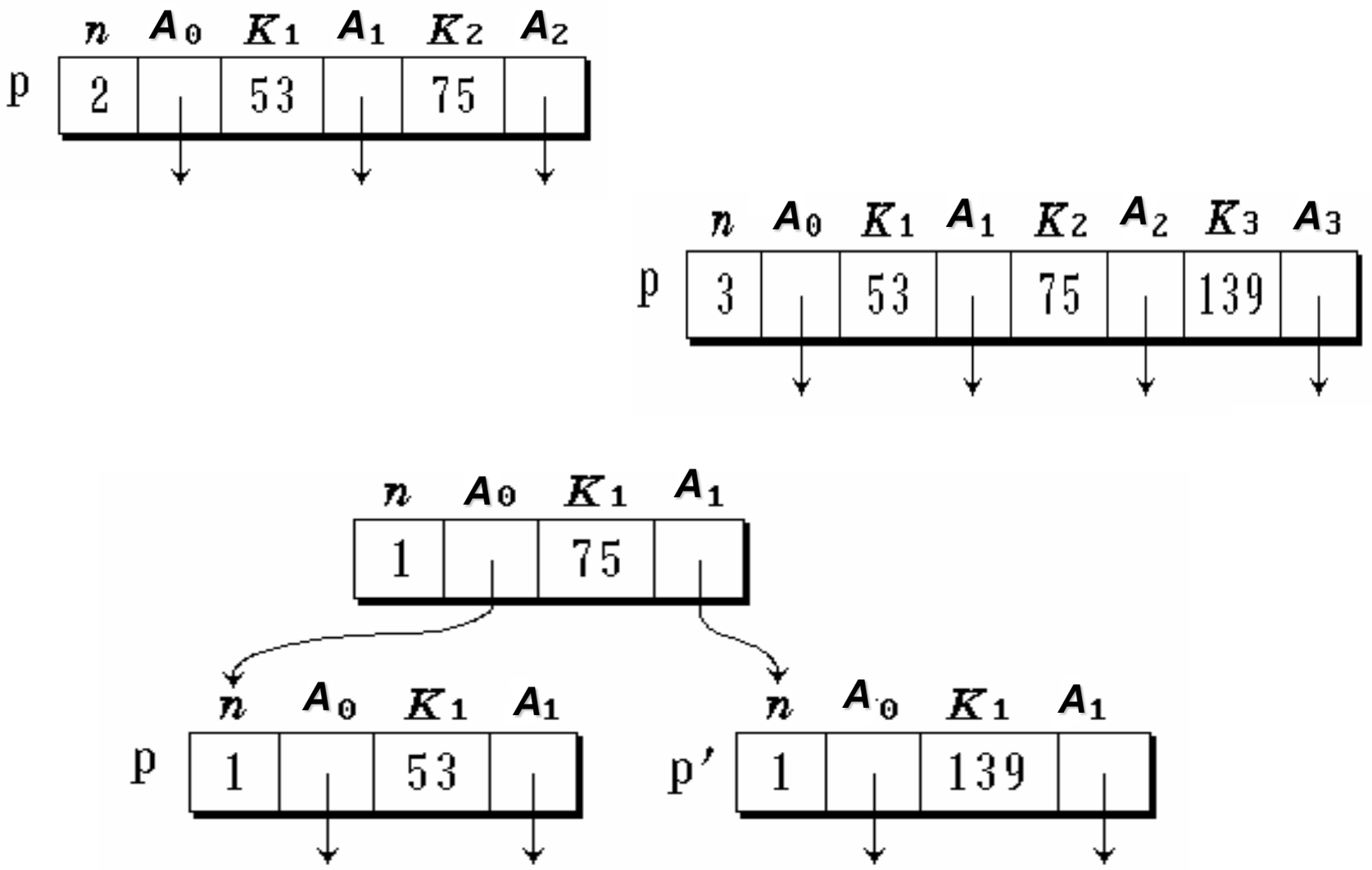
- B-树是从空树起，逐个插入关键字而生成的。
- 在B-树，每个非叶结点的关键字个数都在[$\lceil m/2 \rceil$-1, m-1]之间。
- 插入是在最低层的某个非终端结点中添加一个关键字。如果在关键字插入后结点中的关键字个数超出了上界 m-1，则结点需要“分裂”，否则可以直接插入。
- 实现结点“分裂”的原则是：

  p结点：($m, A_0, K_1, A_1, K_2, A_2, \ldots\ldots, K_m, A_m$)

p结点：($\lceil m/2 \rceil$-1, $A_0, K_1, A_1, \ldots\ldots, K_{\lceil m/2 \rceil-1}, A_{\lceil m/2 \rceil-1}$)

p’结点：(m -$\lceil m/2 \rceil, A_{\lceil m/2 \rceil}, K_{\lceil m/2 \rceil+1}, A_{\lceil m/2 \rceil+1}, \ldots\ldots, K_m, A_m$)

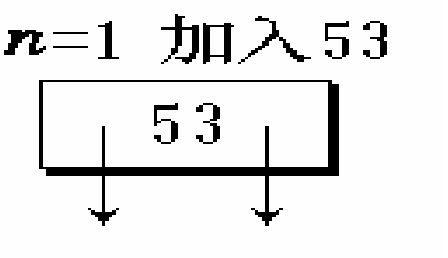
($K_{\lceil m/2 \rceil}$, p’)插入到这两个结点的双亲结点中去.

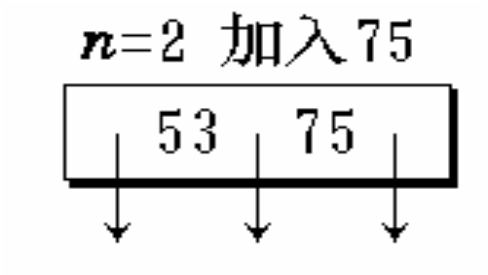

n A0 K1 A1 K2 A2
p
2
53
75
n A0 K1 A1 K2 A2 K3 A3
p
3
53
75
139
n A0 K1 A1
1
75
n A0 K1 A1
p
1
53
n A0 K1 A1
p'
1
139

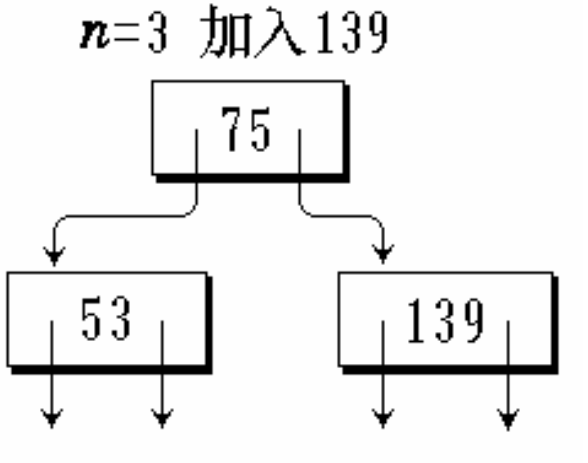
结点“分裂”的示例

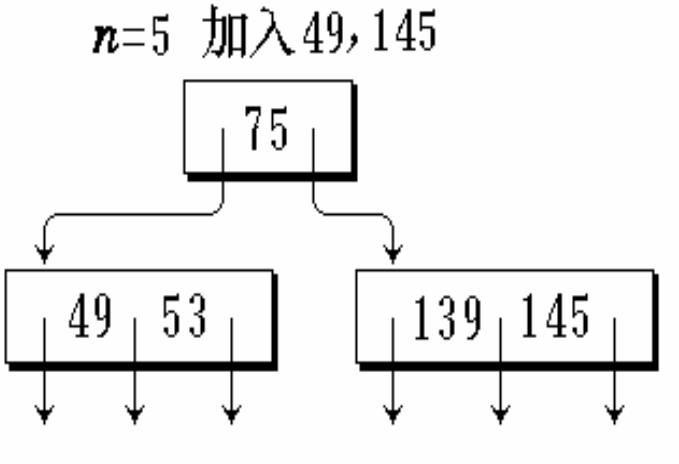
示例：从空树开始逐个加入关键字建立3阶B-树

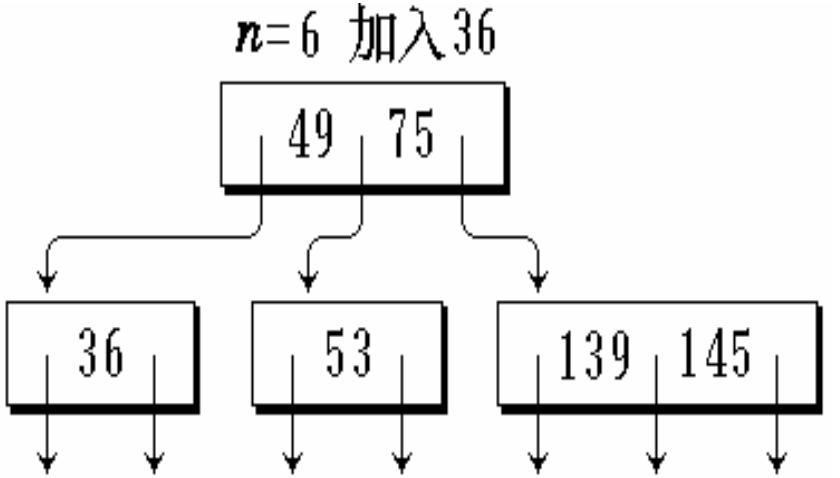
{53, 75, 139, 49, 145, 36}



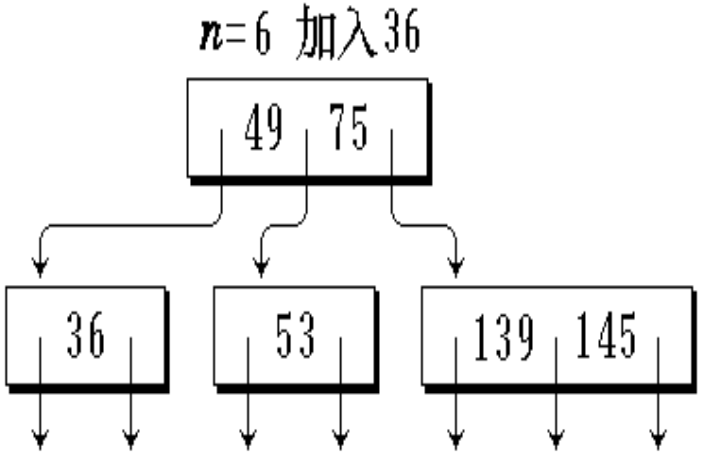


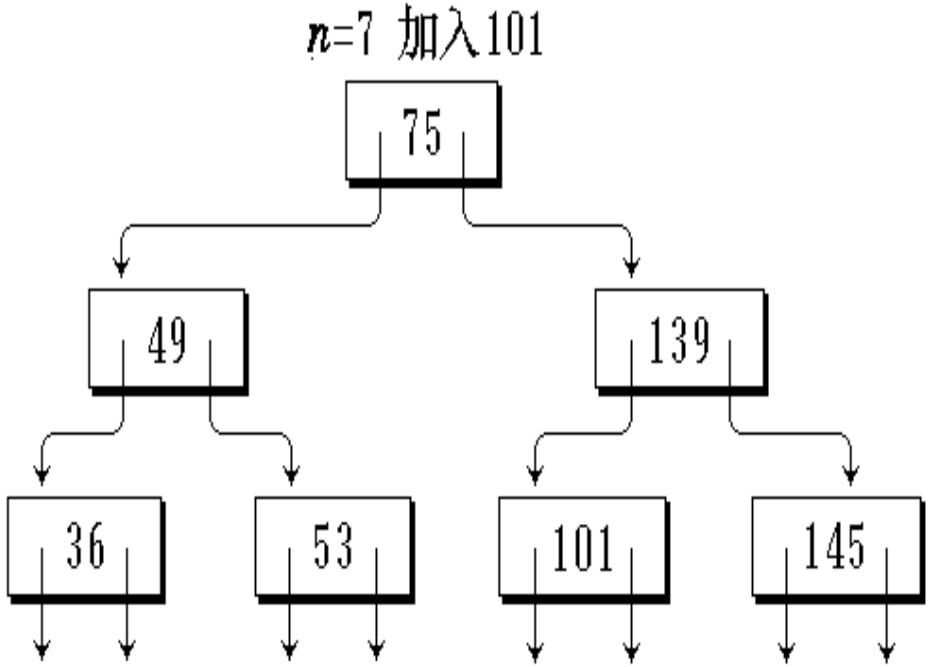






- 在插入新关键字时，需要自底向上分裂结点，最坏情况下从被插关键码所在最下层结点到根的路径上的所有结点都要分裂。

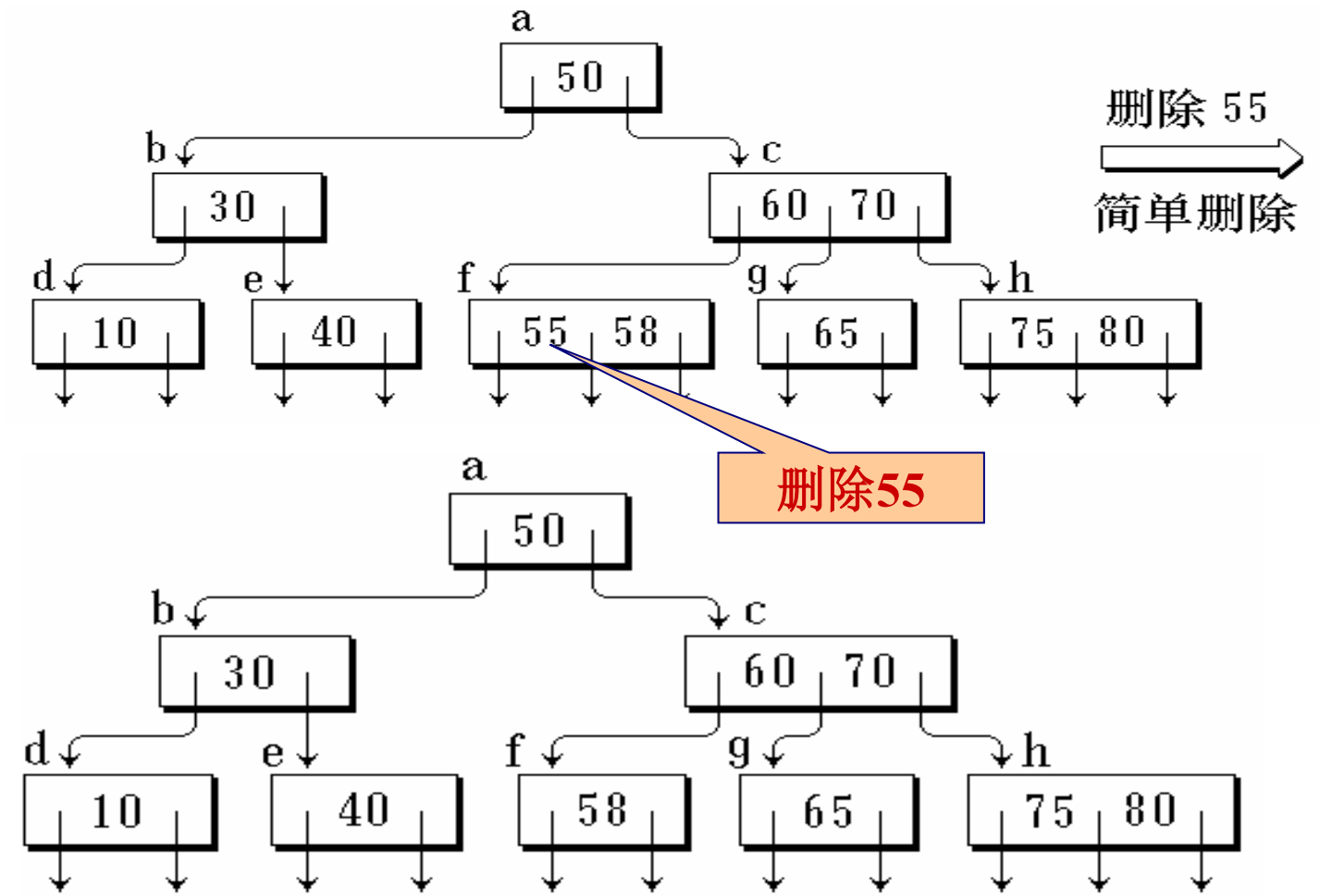




5. B-树的删除

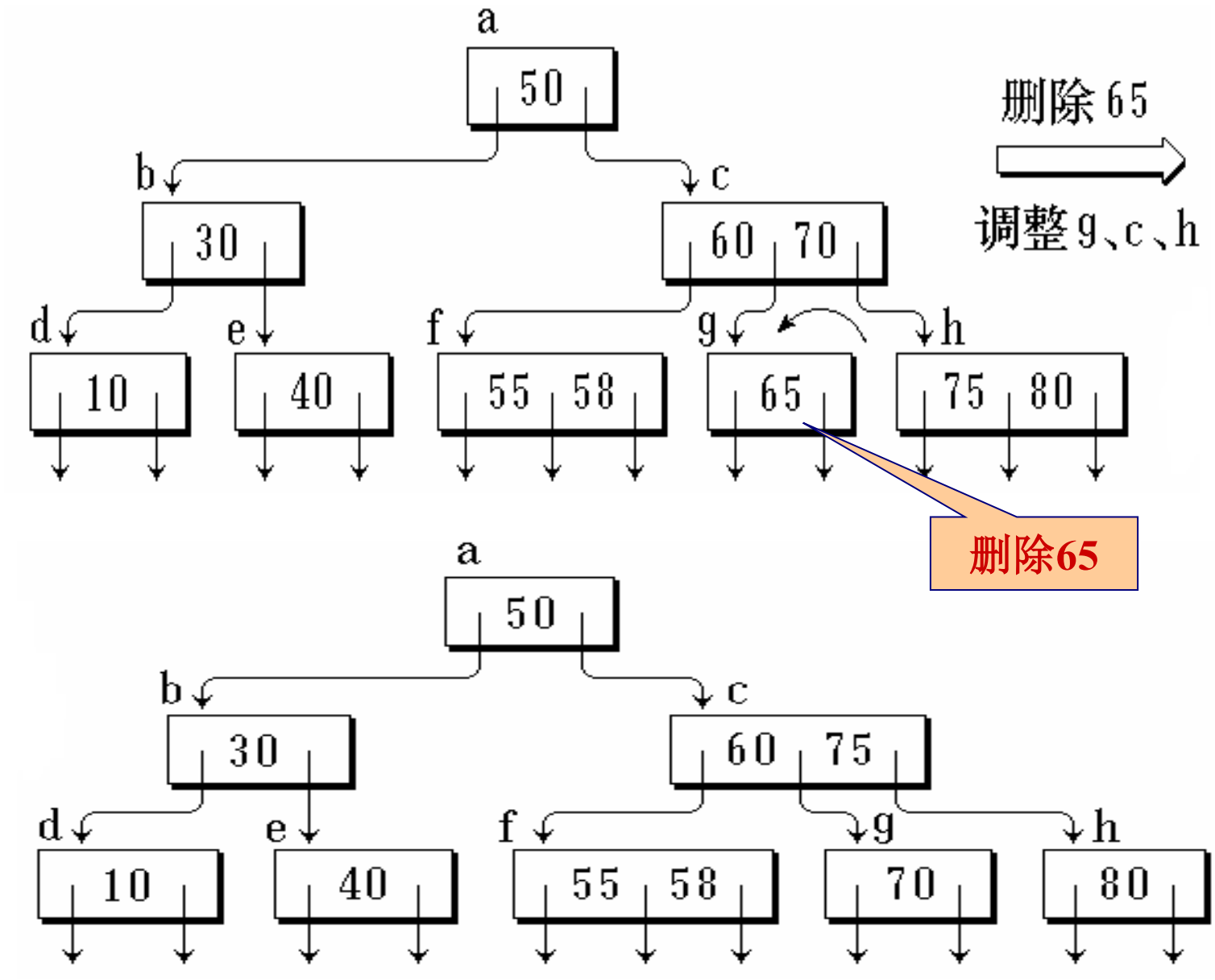
●在最下层非终端结点上的删除有 3 种情况。

被删关键字所在结点中的关键字个数 $n \geq \lceil m/2 \rceil$，则直接删去该关键字K_i和相应指针A_i，树的其它部分不变 。



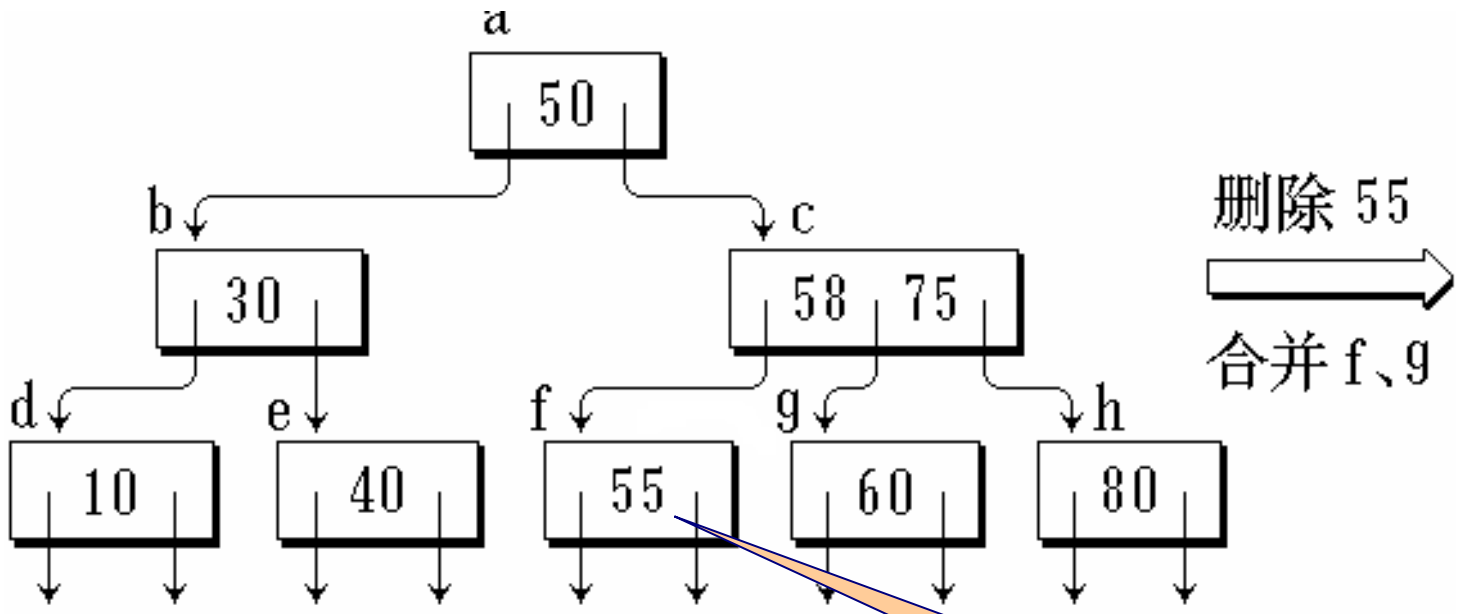
- 被删关键字所在结点中关键字个数 $n = \lceil m/2 \rceil - 1$，与该结点相邻的右兄弟 (或左兄弟) 结点的关键字个数 $n > \lceil m/2 \rceil - 1$：

  - 将双亲结点中刚刚大于 (或小于) 该被删关键码的关键字 K_i $(1 \le i \le n)$ 下移；
  - 将右兄弟 (或左兄弟) 结点中的最小 (或最大)关键码上移到双亲结点的 K_i 位置；

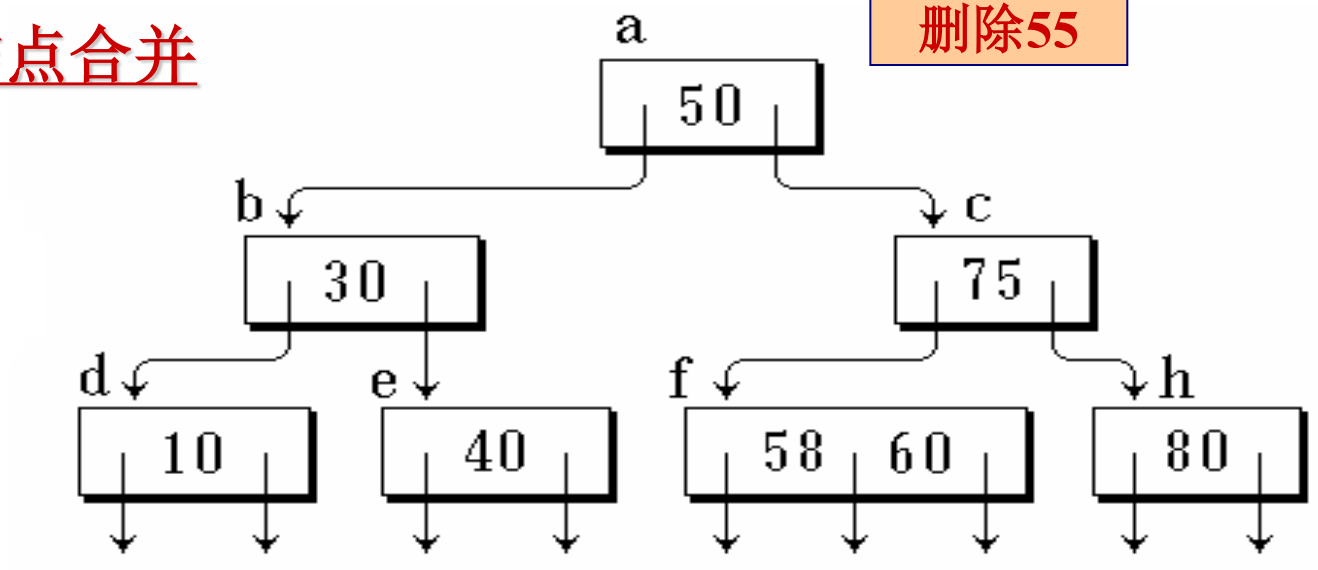

a
50
b
30
c
60 70
d
10
e
40
f
55 58
g
65
h
75 80
删除 65
调整 g、c、h
删除65
a
50
b
30
c
60 75
d
10
e
40
f
55 58
g
70
h
80

- 被删关键字所在结点中关键码个数 $n = \lceil m/2 \rceil - 1$，若这时与该结点相邻的右兄弟 (或左兄弟) 结点的关键码个数 $n = \lceil m/2 \rceil - 1$，则必须合并结点。

- 删除关键字的结点与其右 (或左)兄弟结点以及双亲结点中分割二者的关键字合并成一个结点。


a
50
b
30
c
58 75
d
10
e
40
f
55
g
60
h
80
删除 55
合并 f、g
删除55

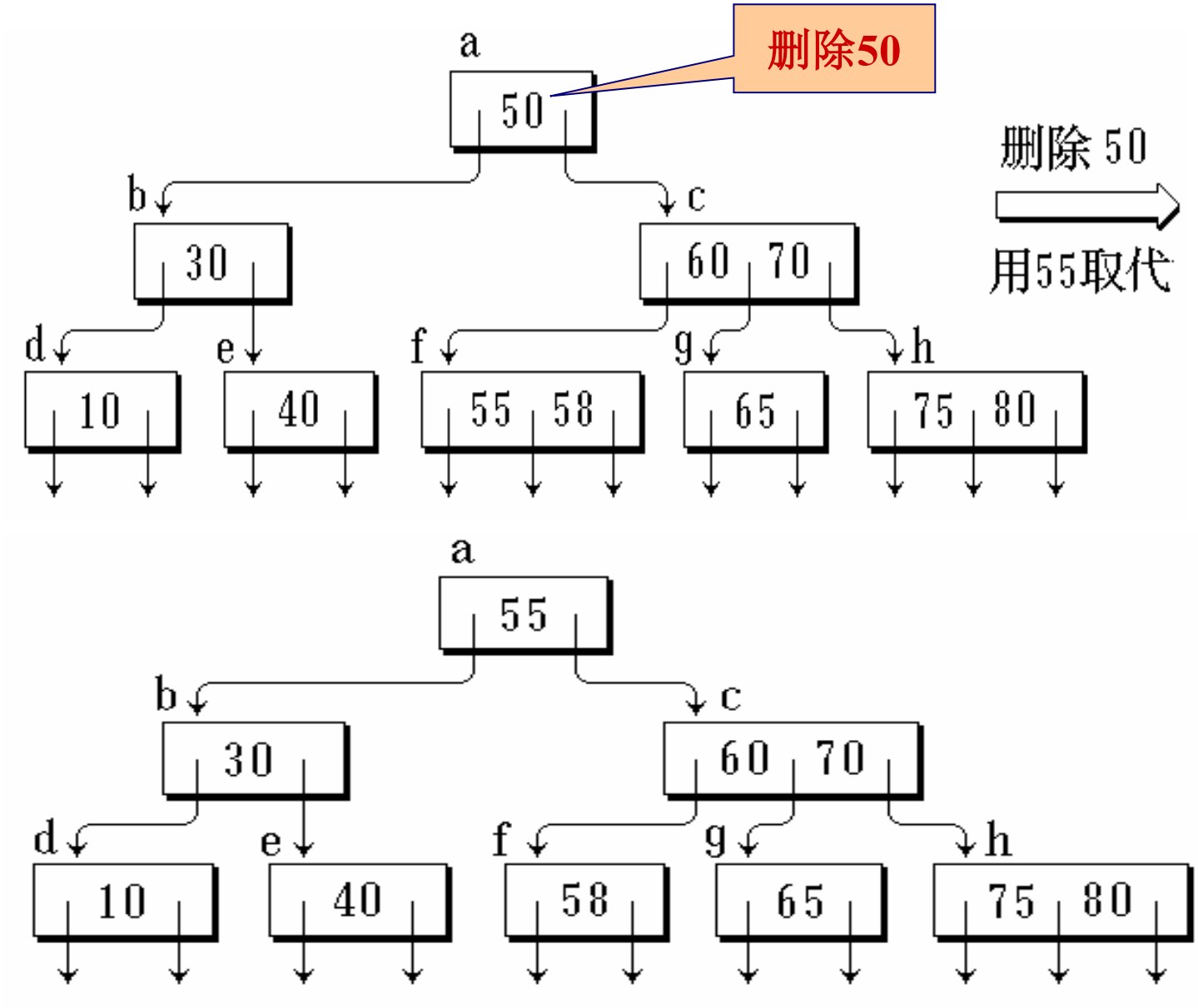
结点合并


a
50
b
30
c
75
d
10
e
40
f
58 60
h
80

●在非最下层结点上删除关键字的过程：

要删除关键字K_i，在删去该关键字后，以该结点A_i所指子树中的最小关键字**x**来代替被删关键字K_i所在的位置，然后在***x***所在的结点中删除 **x**。

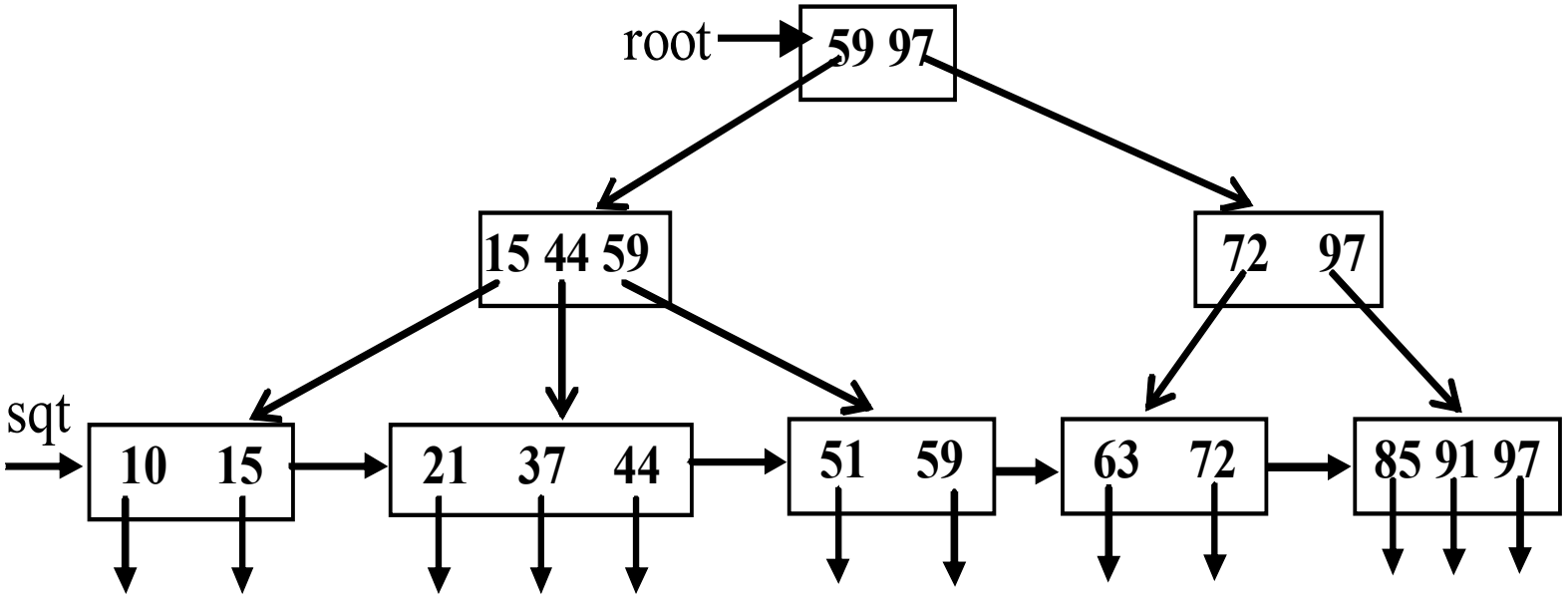
这样就把在非最下层结点上删除关键字的问题转化成了在最下层结点上删除关键字的问题。


a
50
删除50
删除 50
用55取代
b
30
c
60 70
d
10
e
40
f
55 58
g
65
h
75 80
a
55
b
30
c
60 70
d
10
e
40
f
58
g
65
h
75 80

B^+树

B^+树可以看作是B-树的一种变形，在实现文件索引结构方面比B-树使用得更普遍。

1. 一棵*m*阶B^+树与*m*阶B-的差异：
   - 有*n*棵子树的结点中含有*n*个关键字；
   - 所有的叶结点都处于同一层次上，包含了全部关键字及指向含有这些关键字记录的指针，且叶结点本身按关键字从小到大顺序链接；
   - 所有的非叶结点可以看成是索引部分，结点中含有其子树中的最大（最小）关键字。



- B^+树可进行两种查找运算：一种是循叶结点链顺序查找；另一种是从根结点开始，进行自顶向下，直至叶结点的随机查找。

9.3 哈希表

前面介绍的内容中，记录在文件中的存储地址是随机的。

	学号	姓名	年龄	
01	200305	张四	19	
02	200305	李四	20	
03	200302	张五	19	

查找某一条记录需要进行一系列的“比较”。

查找的效率依赖于比较的次数。

能否在记录的关键字和存储地址之间构造这样一种关系 f，使得关键字和存储地址一一对应？

此对应关系 f 称为哈希函数。

9.3 哈希表

1. 哈希表的基本概念

●哈希法又称散列法、杂凑法或关键字地址计算法等，相应的表称为哈希表或散列表；

●方法的基本思想：在元素的关键字Key和元素的存储位置p之间建立一个对应关系H，使得p=H(Key)，H称为哈希函数(散列函数)，是一个压缩映象，是从关键字空间到存储地址空间的一种映象。

●当查找关键字为k的元素时，利用哈希函数计算出该元素的存储位置p=H(k)，从而达到按关键字直接存取元素的目的；

例，设关键字为年龄字段，

$H(key) = key$

	年龄	学号	姓名	
01	1			
02	2			
03	3			
⋮	⋮	⋮	⋮	
56	56			
⋮	⋮	⋮	⋮	
100	100			

		⋮	
19	19	200302	张三
20	20	200301	王五
21	21	200305	李四
		⋮	

例 30个地区的各民族人口统计表

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

以编号作关键字，
构造哈希函数：H(key)=key
H(1)=1
H(2)=2

以地区别作关键字，取地区名称第一个拼音字母的序号作哈希函数：H(Beijing)=2
H(Shanghai)=19
H(Shenyang)=19

从例子可见：

哈希函数只是一种映象，所以哈希函数的设定很灵活，只要使任何关键字的哈希函数值都落在表长允许的范围之内即可。

当关键字集合很大时，关键字值不同的元素可能会映象到哈希表的同一地址上，即 $k_1 \neq k_2$，但 $H(k_1)=H(k_2)$，这种现象称为“冲突”，此时称k_1和k_2为“同义词”。实际中，冲突是不可避免的， 只能通过改进哈希函数的性能来减少冲突。

综上所述， 哈希法主要包括以下两方面的内容：

(1)如何构造哈希函数；

(2)如何处理冲突。

2. 哈希函数构造方法

①直接定址法

取关键字或关键字的某个线性函数值为哈希地址。直接定址法的哈希函数H(key)为：

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

计算简单,并且不可能有冲突发生。当关键字的分布基本连续时,可用直接定址法的哈希函数；否则,若关键字分布不连续将造成内存单元的大量浪费。

(i) 例，以年龄作为关键字

(ii) 例，统计解放后出生人口，以出生年份作为关键字

$$H(key) = key - 1949 + 1$$

地址	01	02	···	23	···	48	···
出生年份	1949	1950	···	1971	···	1996	···

②数字分析法

假设关键字集合中的每个关键字都是由 s 位数字组成 (u1, u2, …, us)，分析关键字集中的全体， 并从中提取分布均匀的若干位或它们的组合作为地址。此法适于能预先估计出全体关键字的每一位上各种数字出现的频度。

例 有80个记录，关键字为8位十进制数，哈希地址为2位十进制数

①② ③④⑤⑥ ⑦⑧

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

当无法确定关键字中哪几位分布较均匀时，取关键字平方后的中间几位作为哈希函数。

实验证明：一个数字的平方值的中间部分通常对各位数字都比较敏感。故不同关键字会以较高的概率产生不同的哈希地址。

X
2 0 0 5 2 4
2 0 0 5 0 2
0 1 2 0 0 5
0 2 2 0 0 5
0 3 2 0 0 5

X^2
40209874576
40201052004
00144120025
00484220025
01024320025

取4位
2098
2010
1441
4842
0243

④折叠法

将关键字分割成位数相同的几部分，然后去这几部分的叠加和。

两种叠加处理的方法：

移位叠加:将分割后的几部分低位对齐相加

折叠叠加:从一端沿分割界来回折送，然后对齐相加。此法适于关键字的数字位数特别多。　　**Key = 12360324711220**

	1	2	3
	6	0	3
	2	4	7
	1	1	2
+)	0	2	0
1	1	0	5

(a) 移位叠加

	1	2	3
	3	0	6
	2	4	7
	2	1	1
+)	0	2	0
	9	0	7

(b) 折叠叠加

⑤除留余数法

设定哈希函数为:

$$H(key) = key \ MOD \ p \quad (p \le m)$$

其中， m为表长

p 为不大于 m 的素数

或是

不含 20 以下的质因子

为什么要对 p 加限制?

例如:

给定一组关键字为: 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

假设哈希表长为m， p为小于等于m的最大素数， 则哈希函数为

H(key) = key % p

已知待散列元素为{18, 75, 60, 43, 54, 90, 46}, m=p=13, 则有:

H(18)=18 % 13=5　H(75)=75 % 13=10　H(60)=60 % 13=8
H(43)=43 % 13=4　H(54)=54 % 13=2　H(90)=90 % 13=12
H(46)=46 % 13=7

0	1	2	3	4	5	6	7	8	9	10	11	12
		54		43	18		46	60		75		90

装填因子α(Load Factor): α= n / m.

n: 表中关键字数； m: 表长。

⑥伪随机数法

采用一个伪随机函数作哈希函数，即H(key)=random(key)。

在实际应用中， 应根据具体情况， 灵活采用不同的方法，并用实际数据测试它的性能，以便做出正确判定。 通常应考虑以下五个因素：

- 计算哈希函数所需的时间（简单）。
- 关键字的长度。
- 哈希表的大小。
- 关键字的分布情况。
- 记录查找的频率。

3. 哈希冲突

对于不同的关键字可能得到同一哈希地址，即 **key1≠ key2** ，而 ***f*(key1) ＝ *f*(key2)** ，这种现象称为哈希冲突。

造成原因:

A. 主观设计不当

例，数字分析法中

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

冲突！

例，除留余数法中

关键字	28	35	63	77	105
哈希地址	7	14	0	14	0

$p = 21$

B. 客观存在

如何设计都不可能完全避免冲突的出现 ？

哈希地址是有限的，而记录是无限的。

解决方法:

(1) 开放定址法

(2) 再哈希法

(3) 链地址法

(4) 建立一个公共溢出区

处理冲突的方法

①开放定址法

为产生冲突的地址 **H(key)** 求得一个地址序列：

H0, H1, H2, …, Hs 1≤s≤m-1

$$H_i = (H(key) + d_i)\%m,\ i = 1, 2, ..., k\ (k \le m-1)$$

d_i: 称为增量序列

- 线性探测再散列

  d_i = 1, 2, 3, …, m-1

- 二次探测再散列

  d_i = 1^2, -1^2, 2^2, -2^2, …, k^2, $-k^2$　(k≤m/2)

- 随机探测再散列

  d_i = 伪随机数序列

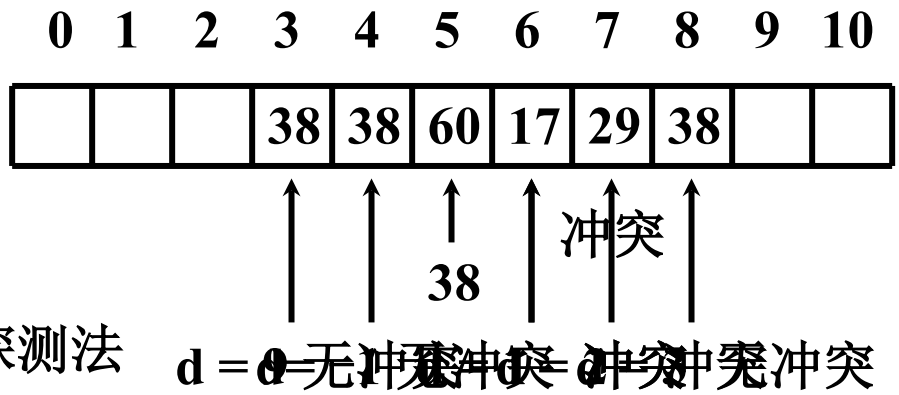
线性探测法　　$d_i = 1 , 2 , 3 , \ldots , m-1$

二次探测法　　$d_i = 1^2 , -1^2 , 2^2 , -2^2 , 3^2 , \ldots , \pm k^2$

随机探测法　　$d_i =$ 随机数

例，关键字为 (17 , 60 , 29 , 38) ， 哈希表长 11 ， H(key)=key%11

初始，



线性探测法　d = 0 无冲突　d = 1 有冲突　d = 2 有冲突　d = 3 无冲突

二次探测法

随机探测法　　不妨设第一次随机数为 9

练习：关键字序列{26, 36, 41, 38, 44, 15, 68, 12, 6, 51, 25}，除留余数法构造哈希函数，线性探测再散列处理冲突，α= 0.75。

m = ⌈*n* / α⌉ = 15，p = 13，H(key) = key%13.

地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
key	26	25	41	15	68	44	6				36		38	12	51
探查次数	1	5	1	2	2	1	1				1		1	2	3

H(26) = 0；H(36) = 10；H(41) = 2；H(38) = 12；H(44) = 5；

H(15) = 2, (2+1)%15 = 3；　　H(68) = 3, (3+1)%15 = 4；

H(12) = 12, (12+1)%15 = 13；　H(6) = 6；

H(51) = 12, (12+1)%15 = 13, (12+2)%15 = 14；

H(25) = 12, (12+1)%15 = 13, (12+2)%15 = 14, (12+3)%15 = 0, (12+4)%15 = 1.

非同义词的冲突：“二次聚集”或“堆积”

②再哈希法

同时构造多个不同的哈希函数：

$$H_i=RH_i(\ key\ ),\quad i=1, 2, \ldots , k$$

当哈希地址发生冲突时，再计算另一个哈希函数地址，直到冲突不再产生。这种方法不易产生聚集，但增加了计算时间。

③建立公共溢出区

将哈希表分为基本表和溢出表两部分，凡是与基本表发生冲突的元素一律填入溢出表。

④链地址法

将所有哈希地址为i的元素构成一个称为同义词链的单链表，并将单链表的头指针存在哈希表的第i个单元中。

H(key) = key%13

H(26) = 0; H(36) = 10;

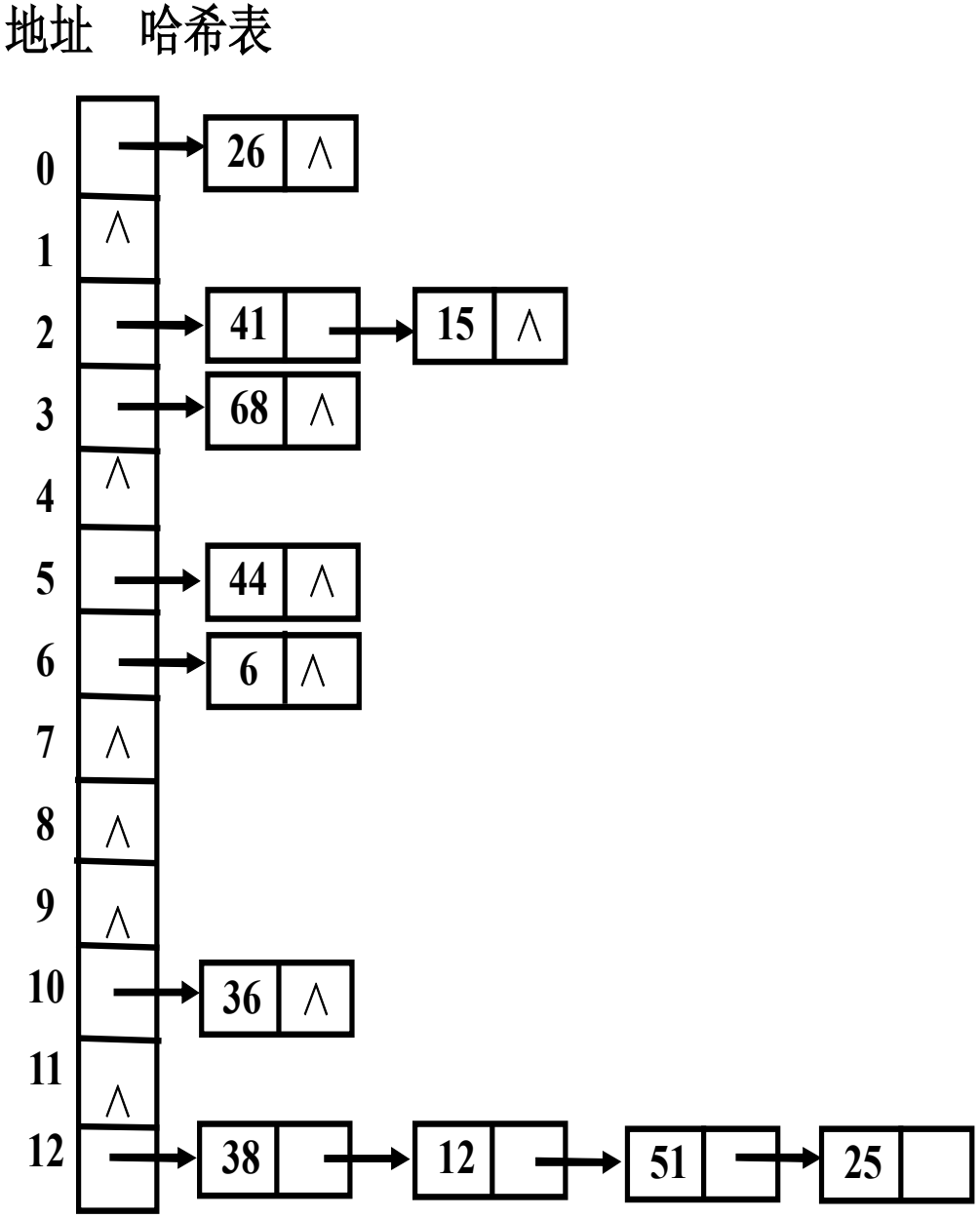
H(41) = 2; H(38) = 12;

H(44) = 5; H(15) = 2;

H(68) = 3; H(12) = 12;

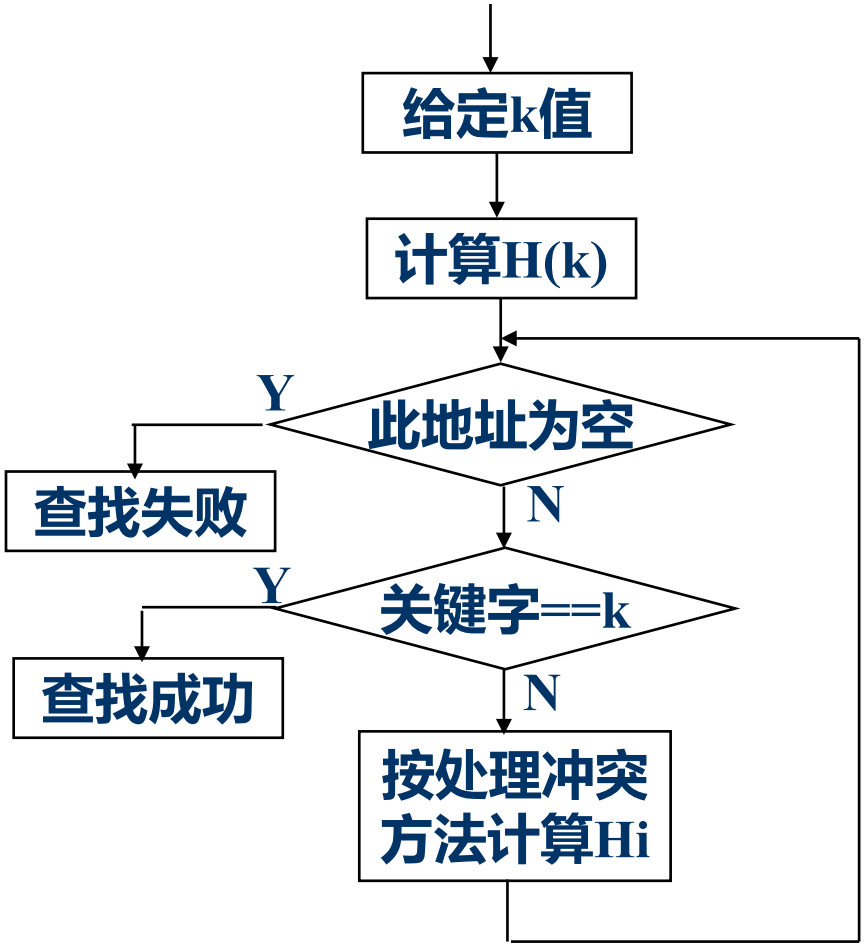
H(6) = 6; H(51) = 12;

H(25) = 12.



哈希表的查找

哈希查找过程



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。

4. 哈希表的分析

●由于冲突的存在，哈希法仍需进行关键字比较， 因此仍需用平均查找长度来评价哈希法的查找性能。

●哈希法中影响关键字比较次数的因素有三个：哈希函数、 处理冲突的方法以及哈希表的装填因子。

处理冲突的方法		平均查找长度 *ASL*	
		查找成功 *Sn*	查找不成功(插入新记录) *Un*
开放定址法	线性探测再散列法	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$
	伪随机探测法 二次探测法 再哈希法	$-\left(\frac{1}{\alpha}\right)\log_e(1-\alpha)$	$\frac{1}{1-\alpha}$
链地址法 (同义词子表法)		$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}\approx\alpha$

- 哈希表的装填因子 α 表明了表中的装满程度。越大，说明表越满，再插入新元素时发生冲突的可能性就越大。

- 哈希的查找性能，即平均查找长度依赖于哈希表的装填因子，不直接依赖于 ***n*** 或 ***m***。

- 不论表的长度有多大，总能选择一个合适的装填因子，以把平均查找长度限制在一定范围内。

关键字序列{26, 36, 41, 38, 44, 15, 68, 12, 6, 51, 25}，n=11

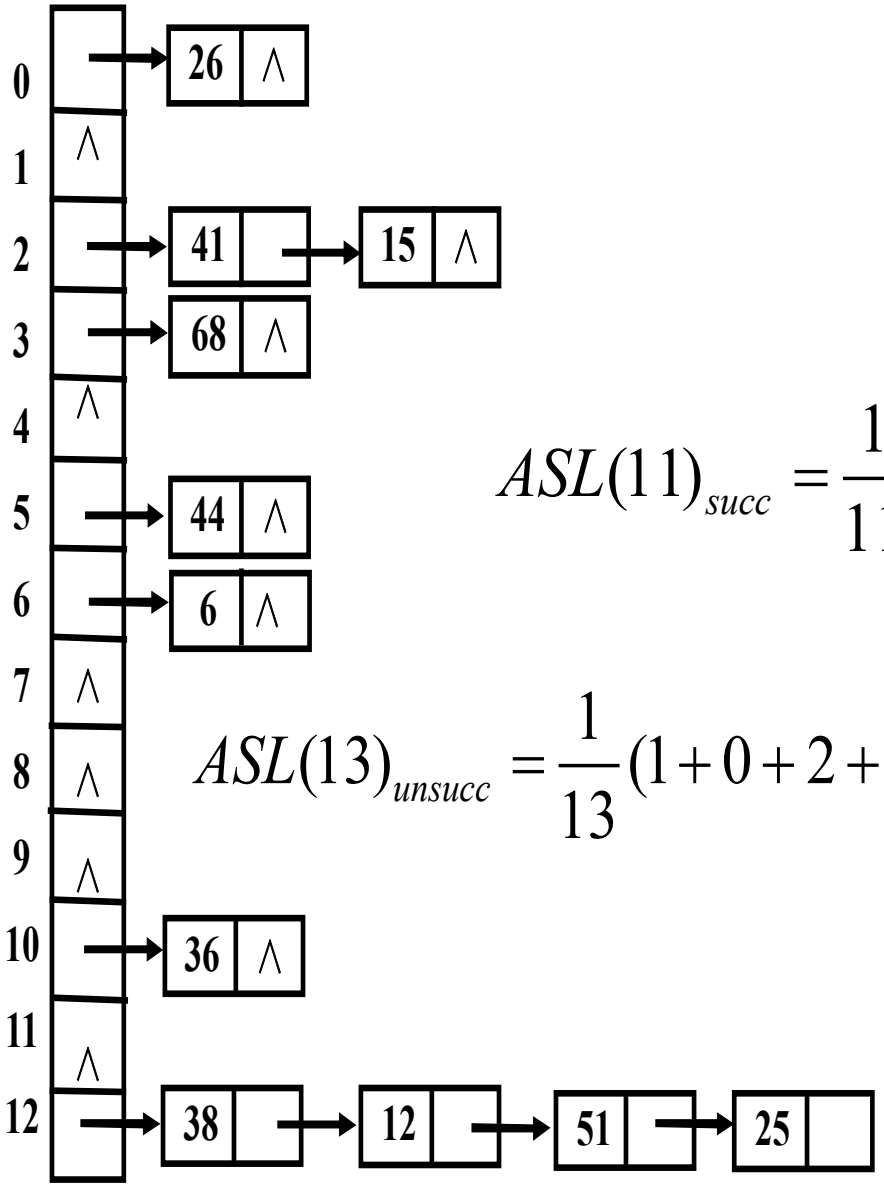
H(key) = key%13，线性探测再散列

地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
key	26	25	41	15	68	44	6				36		38	12	51
探查次数	1	5	1	2	2	1	1				1		1	2	3

$$ASL(11)_{succ} = \frac{1}{11}(1+5+1+2+2+1+1+1+1+2+3) \approx 1.82$$

$$ASL(13)_{unsucc} = \frac{1}{13}(8+7+6+5+4+3+2+1+1+1+2+1+11) \approx 4$$

地址　哈希表



$$ASL(11)_{succ} = \frac{1}{11}(1\times 7 + 2\times 2 + 3\times 1 + 4\times 1) \approx 1.64$$

$$ASL(13)_{unsucc} = \frac{1}{13}(1+0+2+1+0+1+1+0+0+0+1+0+4) \approx 0.85$$

- 作业：

  9.19　9.21