# 第9章　查找

- 静态查找表
- 二叉排序树
- 平衡二叉树（*AVL*树）
- 小结
- B树
- 哈希表

# 静态查找表

## 查找(Search)的概念

- 查找：就是在数据集合中寻找满足某种条件的数据对象。
- 查找表：是由同一类型的数据元素(或记录)组成的数据集合。
- 查找的结果通常有两种可能：
  - 查找成功，即找到满足条件的数据对象。
  - 查找不成功，或查找失败。作为结果，报告一些信息，如失败标志、失败位置等。

■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。

■主关键字：可唯一地标识一个数据元素的关键字。

■次关键字：用以识别若干记录的关键字。

使用基于主关键字的查找，查找结果应是唯一的。

■静态查找表（p214）——没有元素的变化

■动态查找表（p214）——发生元素的变化

# 9.1 静态查找表

- 衡量一个查找算法的时间效率的标准是：在查找过程中关键字的**平均比较次数或平均读写磁盘次数**(只适合于外部查找)，这个标准也称为**平均查找长度**ASL(Average Search Length)，通常它是查找结构中**对象总数n**或文件结构中**物理块总数n的函数**。
- 另外，衡量一个查找算法还要考虑算法所需要的**存储量**和算法的**复杂性**等问题。
- 在静态查找表中，数据对象存放于数组中，利用数组元素的下标作为数据对象的存放地址。查找算法根据给定值x，在数组中进行查找。直到找到x在数组中的**存放位置**或可确定在数组中**找不到x为止**。

# 9.1.1顺序表的查找(Sequential Search)

■顺序查找，又称线性查找，主要用于在线性结构中进行查找。

■存储结构：

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

■查找过程：从表中最后一个元素开始，顺序用各元素的关键字与给定值 x 进行比较。若找到与其值相等的元素，则查找成功，给出该元素在表中的位置；否则，若直到第一个记录仍未找到关键字与 x 相等的对象，则查找失败。

# 算法9.1 （p217）

```
Search_Seq(SSTable ST, KeyType key){
//顺序查找的算法，0号元素为监视哨
 int i;
 ST.elem[0].key=key;
 for (i=ST.length; !EQ(ST.elem[i].key,key);- -i);
 return i;
}
```

# 顺序查找的平均查找长度

设查找第 i 个元素的概率为 $p_i$，查找到第 i 个元素所需比较次数为 $c_i$，则查找成功的平均查找长度：

$$ASL_{succ} = \sum_{i=1}^{n} p_i \cdot c_i \qquad (\sum_{i=1}^{n} p_i = 1)$$

在顺序查找情形，$c_i = n-i+1$，　$i = 1, \ldots, n$，
因此：

$$ASL_{succ} = \sum_{i=1}^{n} p_i \cdot (n-i+1)$$

在等概率情形，$p_i = 1/n$，　$i = 0, 1, \ldots, n-1$。

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

# 9.1.2 有序表的查找

■**折半查找**：先求位于查找区间正中的对象的下标mid，用其关键字与给定值x比较：

- Element[mid].getKey( ) = x，查找成功；
- Element[mid].getKey( ) > x，把查找区间缩小到表的**前半部分**，再继续进行对分查找；
- Element[mid].getKey( ) < x，把查找区间缩小到表的**后半部分**，再继续进行对分查找。

■每比较一次，查找区间缩小一半。如果查找区间已缩小到一个对象，仍未找到想要查找的对象，则查找失败。

# 9.1.2 有序表的查找

■折半查找：

（1）mid= ⌊(low+high)/2⌋

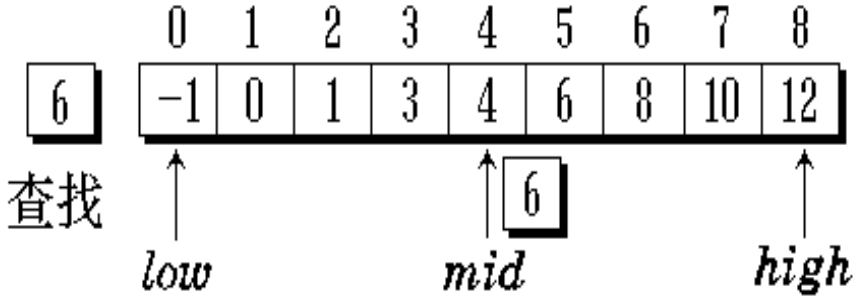
（2）比较 ST.elem[mid].key = = key?

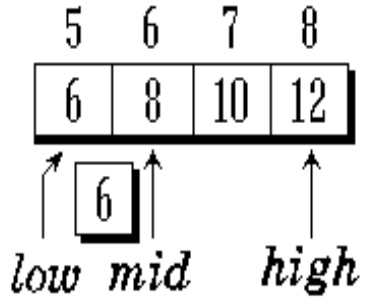
如果ST.elem[mid].key = = key，则查找成功，返回mid值

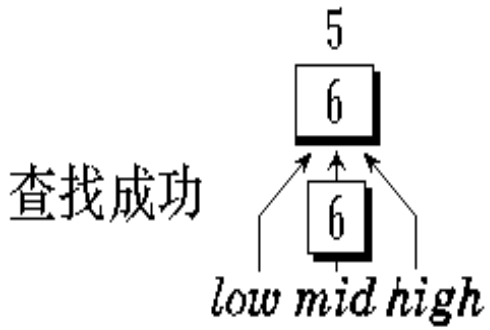
如果ST.elem[mid].key > key，则置high=mid-1

如果ST.elem[mid].key < key，则置low=mid+1

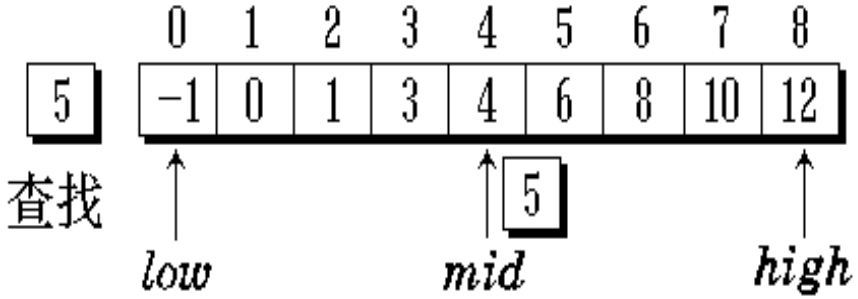
（3）重复计算mid 以及比较ST.elem[mid].key与 key，当low>high时，表明查找不成功，查找结束。

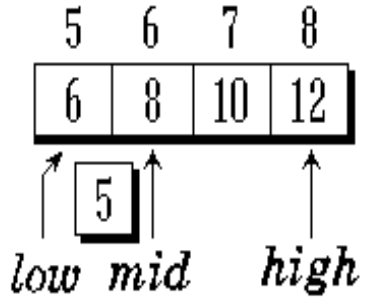


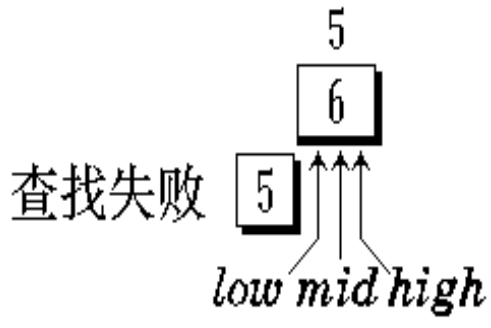




**查找成功的例子**



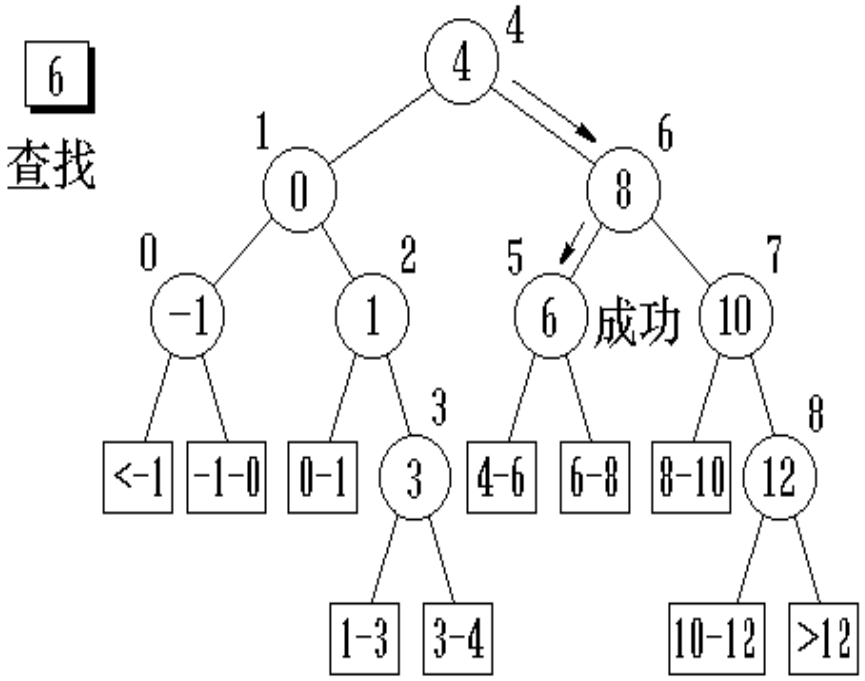




**查找失败的例子**

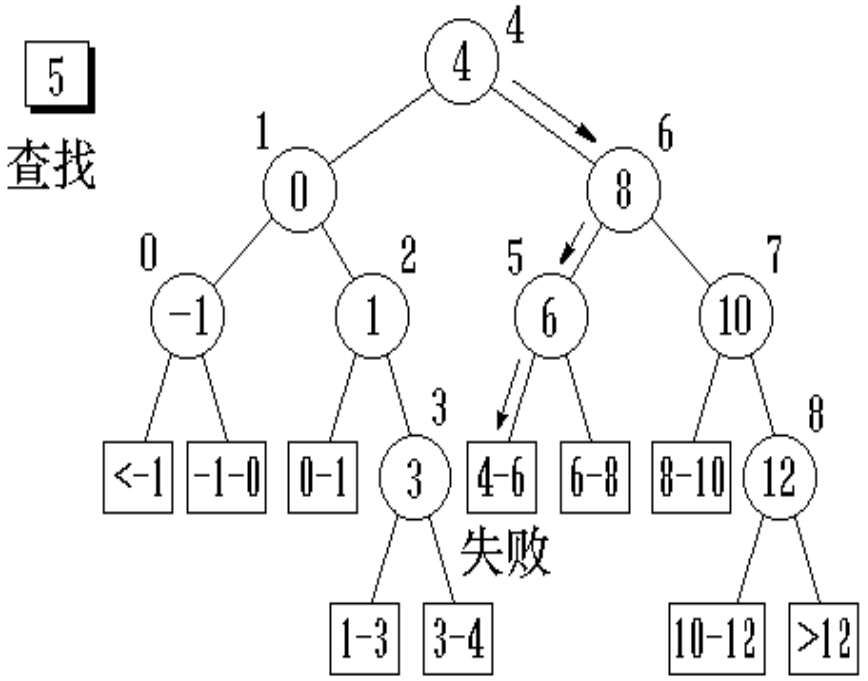
# 算法9.2(p220)

```
int Search_Bin(SSTable ST,KeyType key)
//折半查找 （二分查找）
 { int low,high,mid;
   low = 1; high=ST.length;
   while (low <= high)
    { mid = (low+high)/2;
      if (EQ(key,ST.elem[mid].key)) return mid;
      else if (LT(key,ST.elem[mid].key))
                high=mid-1;
           else low=mid+1;
     }
  return 0;
 }
```

# 从有序表构造出的二叉查找树(判定树)




查找成功的情形




查找不成功的情形

- 若设 $n = 2^h-1$，则描述二分查找的二叉查找树是高度为 $h$ 的满二叉树。$2^h = n+1$，$h = \log_2(n+1)$。
- 第1层结点有1个，查找第1层结点要比较1次；第2层结点有2个，查找第2层结点要比较2次；…，

# 9.1.2 索引顺序表的查找——分块查找

1）分块有序（升序或降序）

——第i块中的最大（小）值小（大）于第i+1块中的最（大）小值

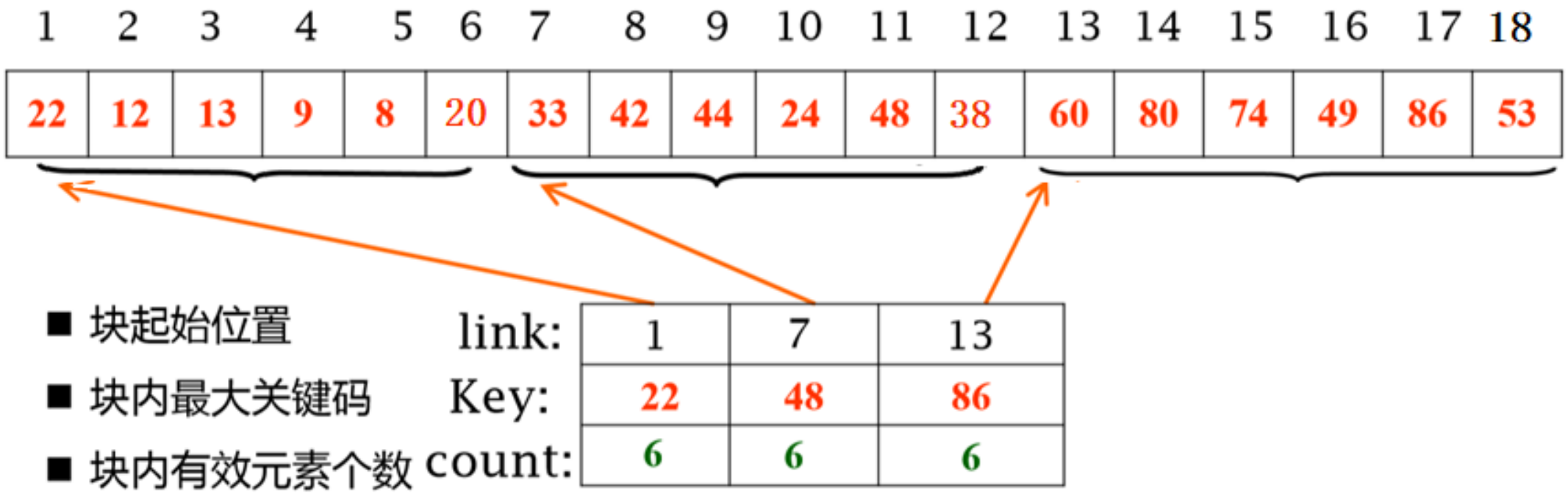
2）查找

（1）先查索引表——折半查找

（2）再查顺序表——顺序查找

**块间有序，块内无序**

```
typedef  struct
   { int key;}  Elemtype;
typedef  struct {
      Elemtype *elem;
      int length;
      } SSTable;
# define BLOCK_NUM  3
```




```
Typedef  struct {
      int Maxkey;
      int next;  } BLKNode,IT[BLOCK_Num];
```

```
Typedef  struct {
     int Maxkey;
     int next;  } BLKNode,IT[BLOCK_Num];
int Search_Bin(SSTable ST,int key) //折半查找
  {int i,p,low,high,mid;
   printf(“Create index table,Input Maxkey& next\n”)
   for(i=1;i<=BLOCK_NUM;i++)
   scanf(“%d,%d”,&IT[i].Maxkey,&IT[i].next)
   if(key>IT[BLOCK_NUM].Maxkey)  return(0);
   low = 1; high=BLOCK_NUM;
   while (low < =high)
     { mid = (low+high)/2;
       if (IT[mid].Maxkey>=key)) high=mid-1;
       else low=mid+1; }
```

```
  i=IT[low].next;
  ST.elem[0].key = key;
  If(low!=BLOCK_NUM)  p=IT[low+1].next;
  else  p=ST.length+1;
  while(ST.elem[i%p].key != key )  i++ ;
  return( i%p );
}
```

性能分析：

ASL（bls）=Lb+Lw=(b+1/)2 +(s+1)/2

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

b ——— 分块的个数  b=n/s

s ——— 每块中的记录个数

n ——— 记录的总个数

Lb——— 确定所在的块

Lw——— 块内查找

# 9.2 动态查找表

表结构本身是在查找过程中动态生成的。

基本操作：

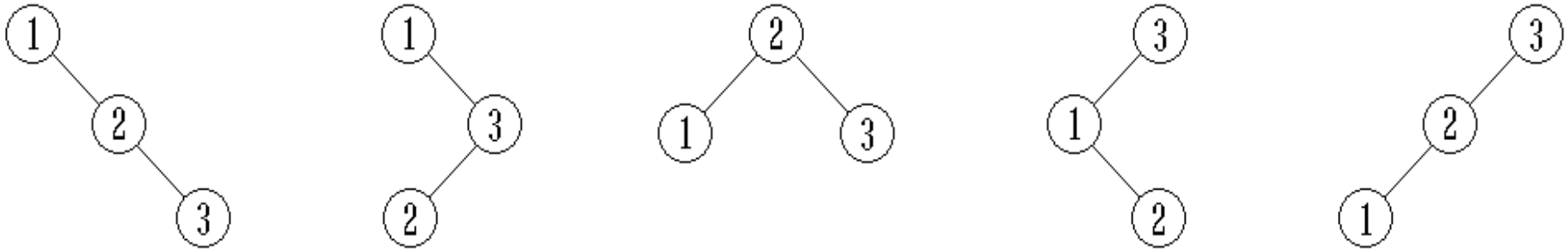
```
InitDSTable(&DT);  //构造一个空的动态查找表DT
DestroyDSTable(&DT);  //销毁表
SearchDSTable(DT,key);  //查找关键字为key的数据元素
InsertDSTable(&DT,e);
DeleteDSTable(&DT,key);
TraverseDSTable(DT,visit());  //遍历查找表
```

# 9.2.1二叉排序树（Binary Sort Tree）

## 定义

二叉排序树（二叉查找树）或者是一棵空树，或者是具有下列性质的二叉树：

- 每个结点都有一个作为查找依据的关键字(key)，所有结点的关键字互不相同。
- 左子树(若非空)上所有结点的关键字都小于根结点的关键字。
- 右子树(若非空)上所有结点的关键字都大于根结点的关键字。
- 左子树和右子树也是二叉排序树。




# 几个二叉排序树的例子

如果对一棵二叉排序树进行中序遍历，可以按从小到大的顺序，将各结点关键字排列起来。

# 二叉排序树上的构建

## ——如何构造二叉排序树

**1）构造过程：**

从空树出发，依次插入$R_1$～ $R_n$各数据值：

（1）如果二叉排序树是空树，则插入结点就是二叉排序树的根结点；

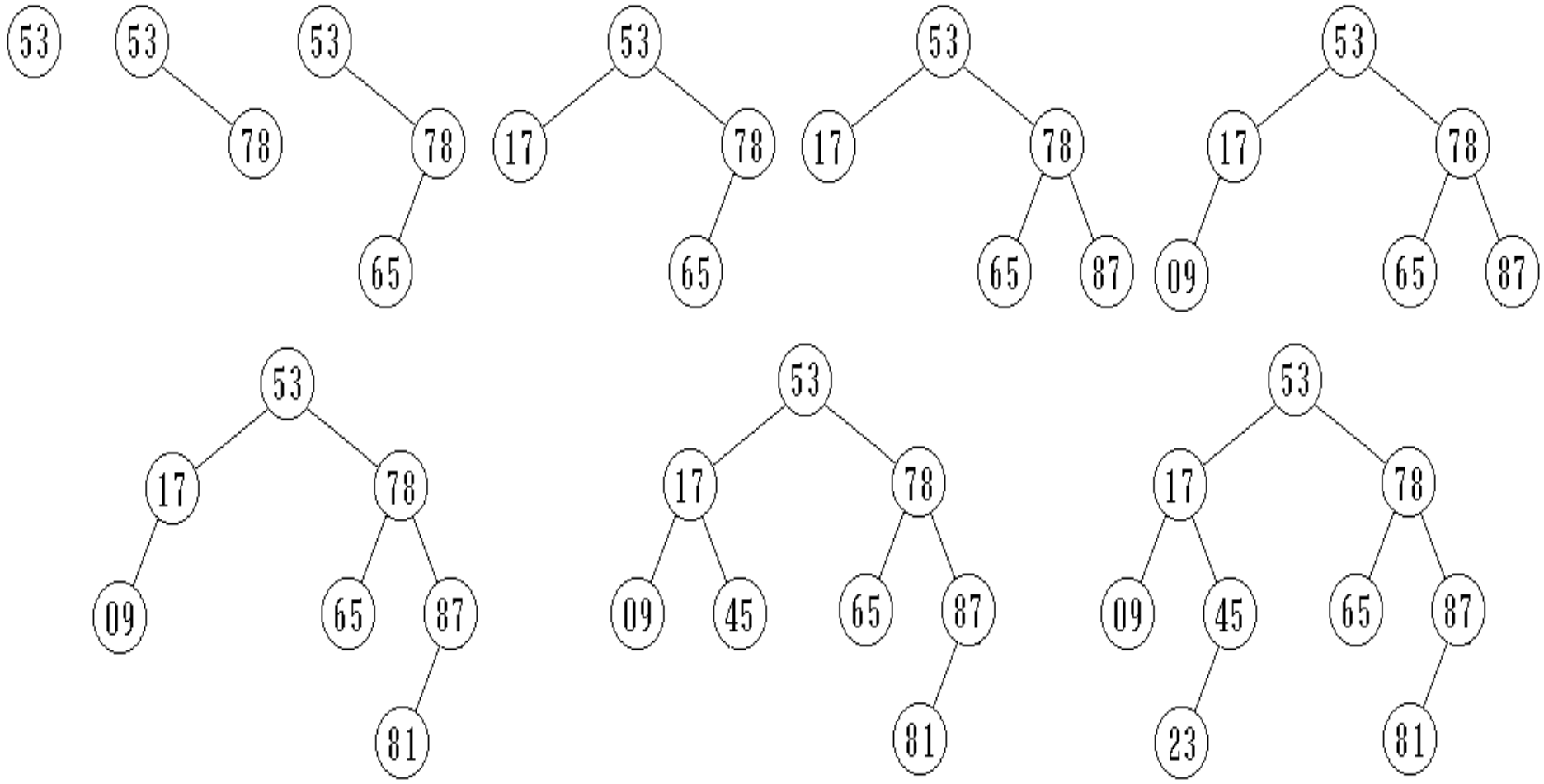
（2）如果二叉排序树是非空的，则插入值与根结点比较，若小于根结点的值，就插入到左子树中去；否则插入到右子树中。

示例：

{ 45，24，53，12，22，90 }

# 输入数据，建立二叉排序树的过程

输入数据序列 { 53, 78, 65, 17, 87, 09, 81, 45, 23 }



## 2）二叉排序树的数据类型描述

```
typedef  struct { int key; }  ElemType;
typedef  struct  BiTNode{
                ElemType  data;
                struct  BiTNode  *lchild,*rchild;
                } BiTNode, * BiTree;
```

## 3）二叉排序树上插入结点的算法

（1）在二叉排序树上插入一个结点的算法；
（2）依次输入一组数据元素，并插入到二叉排序树中的算法

# （1）将指针S所指的结点插入到根结点指针为T的二叉排序树中的算法

```
void Insert_BST( BiTree &T, BiTree S )
 { BiTree  p, q;
   if(!T) T=S;
   else { p=T;
          while ( p )
               { q = p;
                 if(S->data.key < p->data.key)p=p->lchild;
                 else p=p->rchild;
               }
          if(S->data.key < q->dat.key) q->lchild = S;
          else q->rchild = S;
        }
   return;
 }
```

## （2）输入一组数据元素的序列，构造二叉排序树的算法

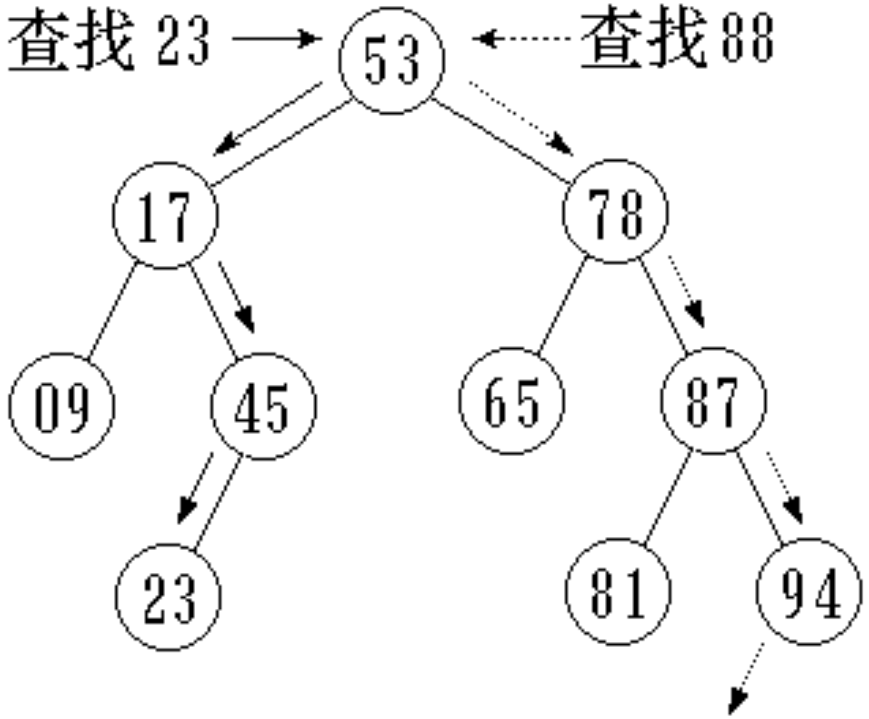
```
void Creat BST( BiTree &T )
  { int x; BiTree S; T=NULL;
    while ( scanf (“%d”,&x), x!=0 )
        { S = (BiTNode *) malloc(sizeof(BitNode));
          S->data.key = x;
          S->lchild = NULL;
          S->rchild = NULL;
          Insert_BST( T,S );
        }
    return;
  }
```

# 4）二叉排序树上的查找（动态查找）

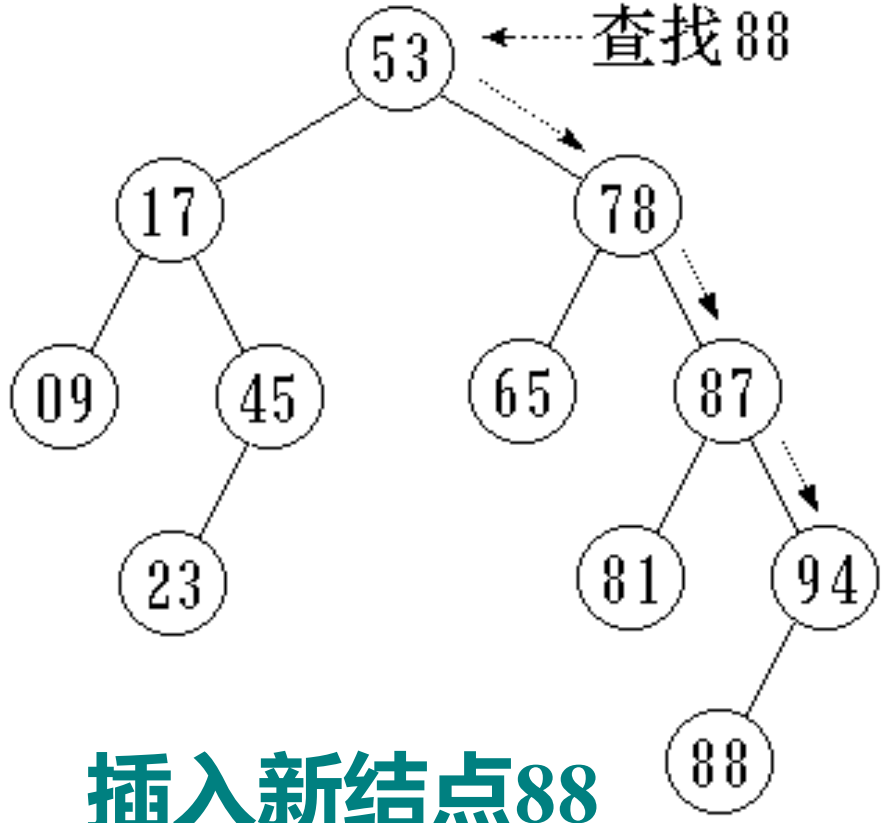
```
int  Searh_BST( BiTree &T, int key )
 { BiTree  p, q, S;
           p = T;
   while ( p )
        if ( p->data.key == key )  return(1);
        else if ( p->data.key > key) { q=p; p=p->lchild; }
             else {q=p; p=p->rchild; }
   S = (BiTNode *) malloc(sizeof(BitNode));
   S->data.key = key;  S->lchild=S->rchild=NULL;
   if (!T) T=S;
   else  if ( q->data.key > key ) q->lchild=S;
         else q->rchild=S;
   return(0);
 }
```

说明: (1) 正常情况下，返回0是未找到，但实际上已经插入;

(2) 也可以用return返回一个自己定义的标志值。




**二叉排序树的查找**




**插入新结点88**

每次结点的插入，都要从根结点出发查找插入位置，然后把新结点作为叶结点插入。

# 二叉排序树上的查找

## ■二叉排序树上的查找分析 P231

- ■在二叉排序上查找其关键字等于给定值的结点的过程，恰是走了一条从根结点到该结点的路径的过程
- ■和给定值比较的关键字个数等于路径长度加1（或结点所在的层次数）
- ■和折半查找类似，与给定值比较的关键字个数不超过树的高度
- ■含有n个结点的二叉排序的平均查找长度和树的形态有关——如P231图9.10所示
- ■最好的情况是，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

## 5）二叉排序树的删除

❖ **要求：删除一个结点后，仍然保持二叉排序树的有序性**

**删除结点的算法：**

（1）确定被删除结点：

A. 有没有被删除的结点；

B. 若有，则确定被删除的结点是**根结点**还是**一般结点**

（2）如果被删除结点是**根结点**，则调用**删除根结点的算法**；

（3）如果被删除结点是**一般结点**，则调用**删除一般结点的算法**。

# 删除二叉排序树的根结点的算法

（1）根结点有左右子树的情况下，选择根结点的左子树中的最大结点为新的根结点； /或者是右子树中的最小结点为新的根结点；

** 最大（小）结点——中序遍历

（2）如果根结点没有左子树，则以右子树的根结点作为新的根结点；

（3）如果根结点没有右子树，则以左子树的根结点作为新的根结点。

# 删除二叉排序树中一般结点的算法

（1）若被删除结点P有左、右子树，则按照中序遍历找其左子树中的最大结点，以此最大结点的值代替P结点的值，然后删除此最大结点（如同删除根结点）；

（2）若被删除结点P没有左子树，则把结点P的右子树的全部挂接在P的双亲上，且位置是P在其双亲中原来的位置；

（3）若被删除结点P没有右子树，则把结点P的左子树的全部挂接在P的双亲上，且位置是P在其双亲中原来的位置。

## 删除二叉排序树中叶子结点的算法

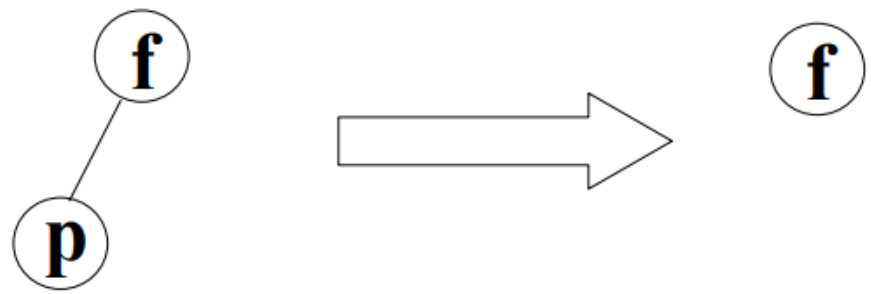
只需将其双亲结点指向它的指针清零，再释放它即可。

# 删除结点

假设在二叉排序树上被删结点为*P，其双亲结点为*f，且不失一般性，可设P是f的左孩子。

1) 若*p 结点为叶子结点，即$P_L$和$P_R$均为空树。由于删去叶子结点不破坏整棵树的结构，则只需修改其双亲结结点的指针即可：

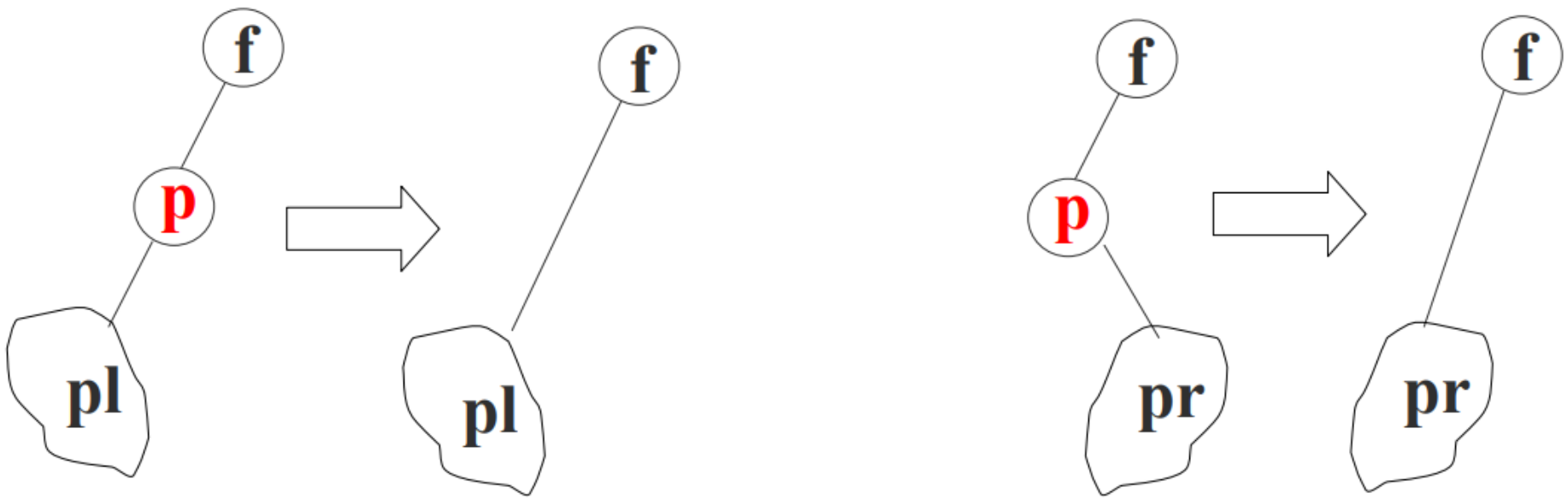
f→lchild=NIL。



# 删除结点

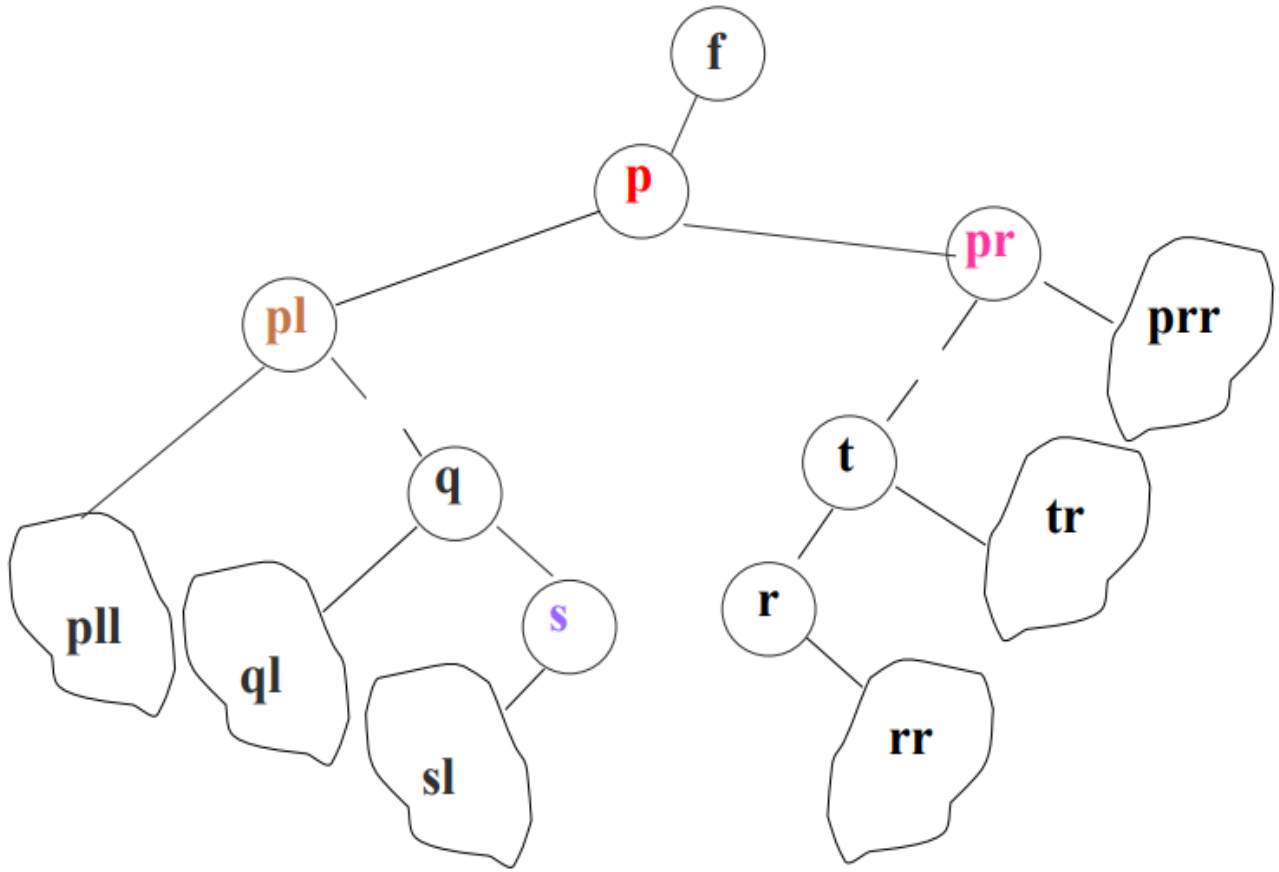
2) 若*p结点只有左子树$P_L$或者只有右子树$P_R$，此时只要令$P_L$或$P_R$直接成为f 的左子树即可:

f→lchild=p→lchild 或 f→lchild=p→rchild。



# 删除结点

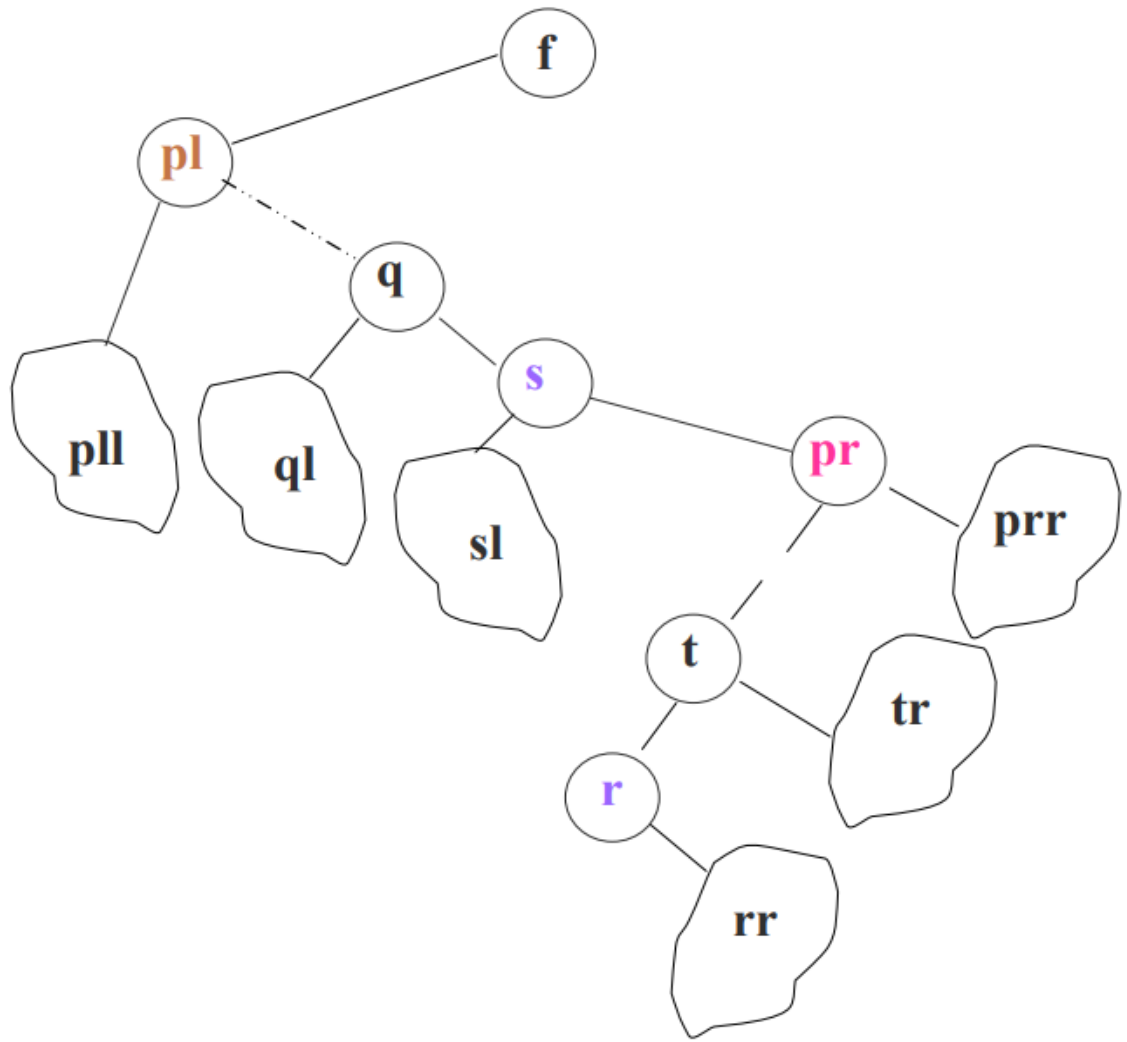
**3) 若*P结点的左子树和右子树均不空。可以有两种做法：**



# 删除结点

方法一：

令*p 的左子树(右子树）为*f 的左子树，而*p 的右子树为*s 的右子树（左子树为*r 的左子树）；

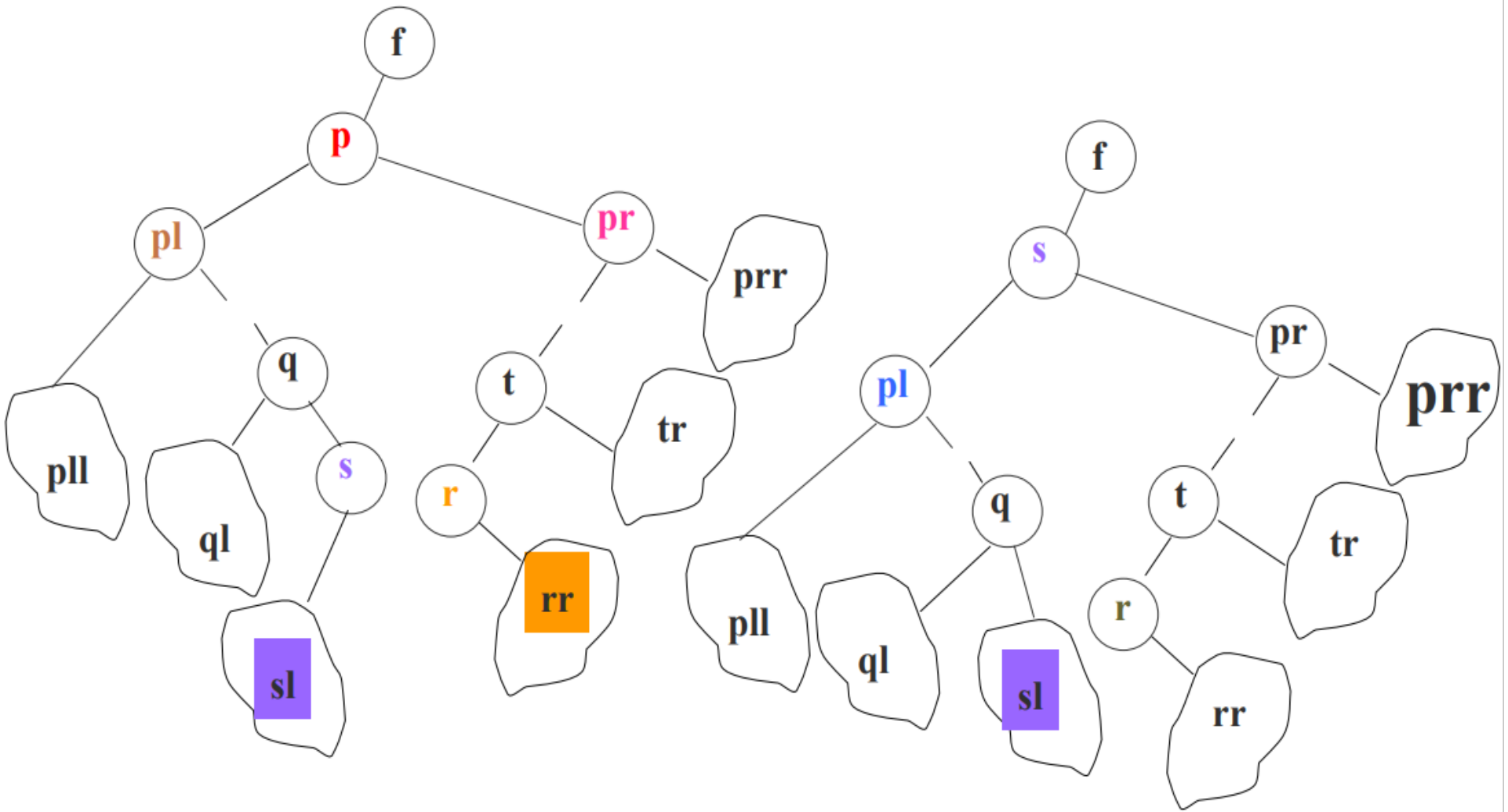


# 删除结点

方法二：

令*p 的直接前驱 (或直接后继) 的左子树 (或右子树) 成为直接前驱 (或直接后继)双亲的右子树 (或左子树)，而直接前驱 (或直接后继)替代 *p 。

# 删除结点



# 二叉排序树的删除递归算法（算法9.7)

```
int DeleteBST(BiTree &T,KeyType key)
 {
   if (!T) return FALSE;
  else {
        if (EQ(key,T->data.key)) {return Delete(T);}
      else  if (LT(key,T->data.key))
                  return DeleteBST(T->lchild,key);
            else return DeleteBST(T->rchild,key);
       }
 }
```

```
int Delete(BiTree &p){//算法9.8
   BiTree q,s;
   if (!p->rchild){
      q=p;  p=p->lchild;  free(q);}
   else if (!p->lchild){
         q=p; p=q->rchild; free(q);}
         else{
                  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;
                   delete s;}
     return TRUE;}
```

# 删除二叉排序树中结点的非递归算法

——寻找被删除的结点

```
int Delete_BST( BiTree &T,int key)
{BiTree p, f ;
 p=T; f=NULL;
 while(p)
    { if(p->data.key == key){ delNode ( T, p,f ) ; return(1) ;}
      else if (p->data.key > key) { f=p; p=p->lchild; }
           else  { f=p; p=p->rchild; }
    }
return(0)
 }
```

# 删除二叉排序树中结点的算法 ——删除找到的结点

```
void delNode ( BiTree &T, BiTree p, BiTree f )
{ BiTree s, q ;
  int tag ;  tag=0;
  if (!p->lchild) s=p->rchild; //左孩子不存在
  else if (!p->rchild) s=p->lchild; //右孩子不存在
       else{  q=p; s=p->lchild;
              while(s->rchild)  { q=s; s=s->rchild; }
              p->data=s->data;
              if (q==p) q->lchild=s->lchild;
              else q->rchild=s->lchild;
              free(s);
              tag=1;  } //左右孩子都存在
  if (!tag){  if ( !f ) T=s;
              else if ( f->lchild==p ) f->lchild=s;
                   else f->rchild=s;
              free(p);
              return;
            }
}
```

# 二叉排序树的删除

- 在二叉排序树中删除一个结点时，必须将因删除结点而断开的二叉链表重新链接起来，同时确保二叉排序树的性质不会失去。

- 为保证在执行删除后，树的查找性能不至于降低，还需要防止重新链接后树的高度增加。
  - 删除叶结点，只需将其双亲结点指向它的指针清零，再释放它即可。
  - 被删结点缺右子树，可以拿它的左子女结点顶替它的位置，再释放它。
  - 被删结点缺左子树，可以拿它的右子女结点顶替它的位置，再释放它。

# 平衡二叉树

- 平衡二叉树（AVL树） P233
  - 或者是一棵空树
  - 或者是具有下列性质的二叉树：
    - 它的左子树和右子树都是平衡二叉树，且左子树和右子树的深度之差的绝对值不超过1
- 结点的平衡因子
  - 该结点的左子树的深度减去它的右子树的深度
  - 平衡因子的取值：-1，0，1

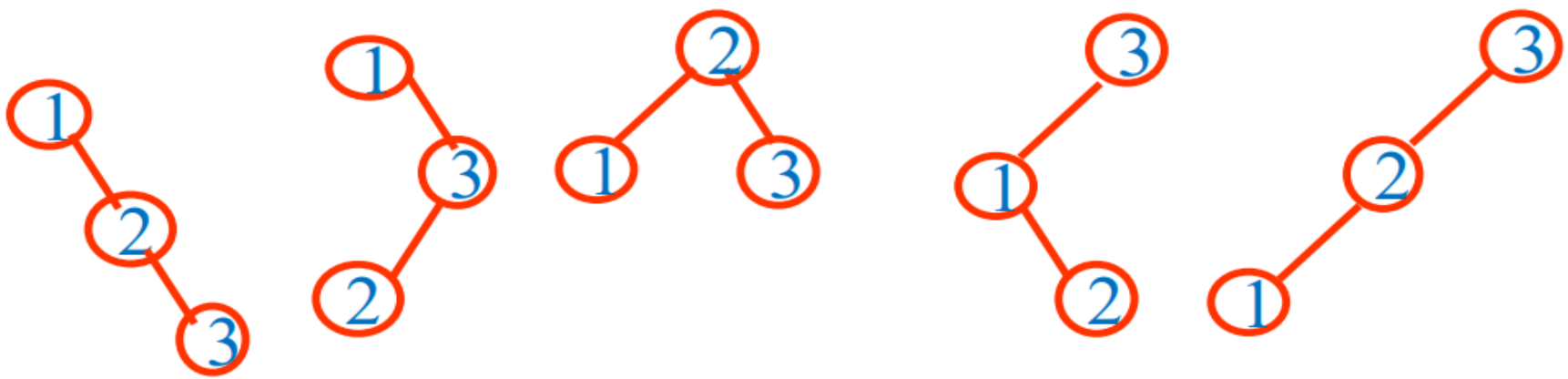
# 二叉排序树查找分析

同样 3 个数据{ 1, 2, 3 }，输入顺序不同，建立起来的二叉排序树的形态也不同。这直接影响到二叉排序树的查找性能。

如果输入序列选得不好，会建立起一棵单支树，使得二叉排序树的高度达到最大，这样必然会降低查找性能。

{2, 1, 3}

{1, 2, 3}　{1, 3, 2}　{2, 3, 1}　{3, 1, 2}　{3, 2, 1}



# 二叉排序树查找分析

最佳情形：平均查找长度$O(\log_2 n)$；

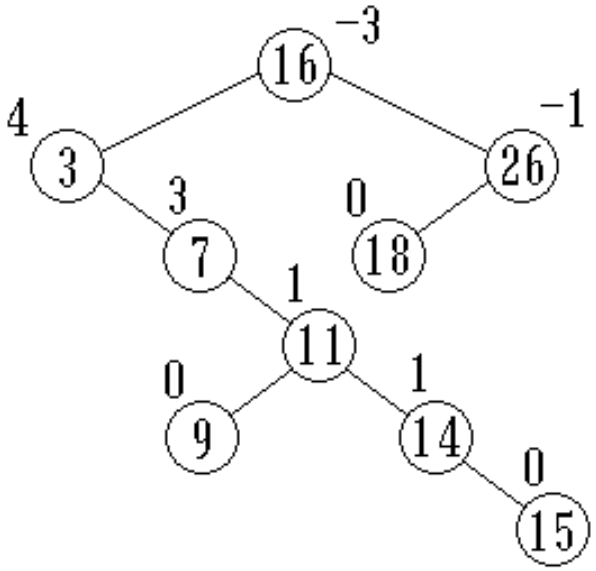
最坏情形：平均查找长度$O(n)$；

平均情形：$\leq 1 + 4\log_2 n$

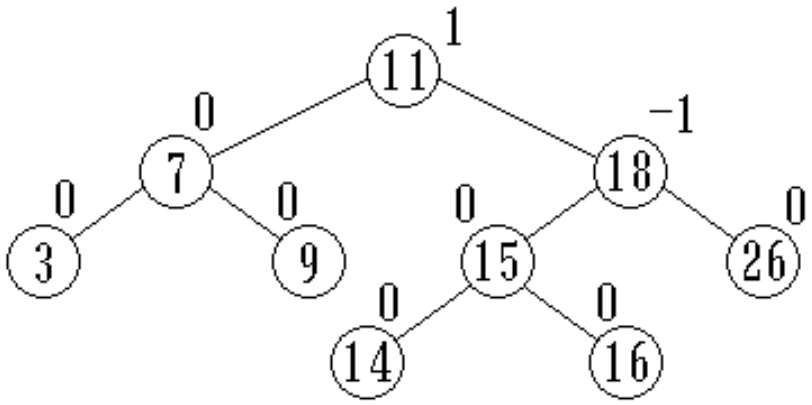
# AVL树 ——高度平衡的二叉查找树

## AVL树的定义

一棵AVL树或者是空树，或者是具有下列性质的二叉查找树：它的左子树和右子树都是AVL树，且左子树和右子树的高度之差的绝对值不超过1。




高度不平衡的二叉排序树




高度平衡的二叉查找树

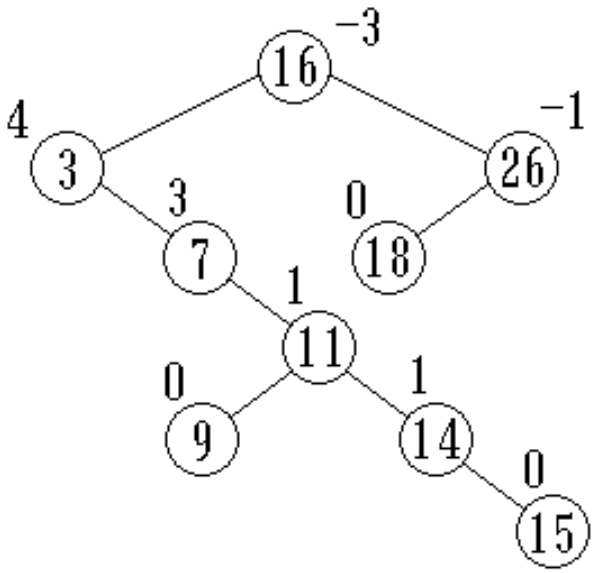
# 结点的平衡因子（Balance Factor）

- 每个结点附加一个数字，给出该结点左子树的高度减去右子树的高度所得的高度差。这个数字即为结点的平衡因子（Balance Factor）。
- 根据AVL树的定义，任一结点的平衡因子只能取 -1，0和 1。
- 如果一个结点的平衡因子的绝对值大于1，则这棵二叉查找树就失去了平衡，不再是AVL树。
- 如果一棵二叉查找树是高度平衡的，它就成为 AVL树。如果它有 n 个结点，其高度可保持在$O(\log_2 n)$，平均查找长度也可保持在$O(\log_2 n)$。

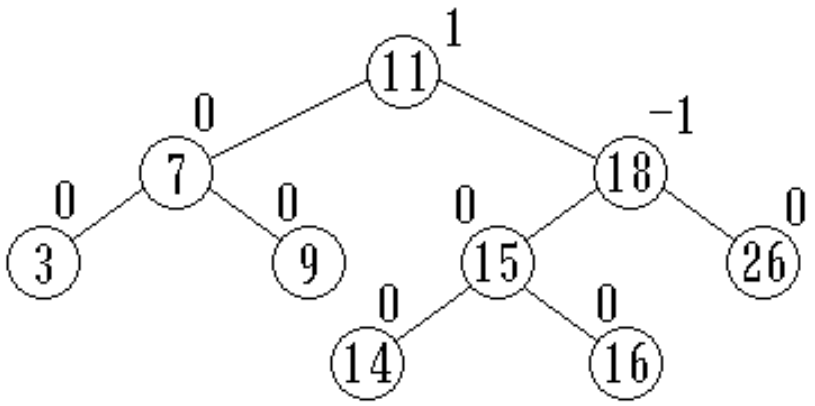
# AVL树 ——高度平衡的二叉查找树

## AVL树的定义

一棵AVL树或者是空树，或者是具有下列性质的二叉查找树：它的左子树和右子树都是AVL树，且左子树和右子树的高度之差的绝对值不超过1。




高度不平衡的二叉排序树




高度平衡的二叉查找树

# 结点的平衡因子（Balance Factor）

- 每个结点附加一个数字，给出该结点左子树的高度减去右子树的高度所得的高度差。这个数字即为结点的平衡因子（Balance Factor）。
- 根据AVL树的定义，任一结点的平衡因子只能取 -1，0 和 1。
- 如果一个结点的平衡因子的绝对值大于1，则这棵二叉查找树就失去了平衡，不再是AVL树。
- 如果一棵二叉查找树是高度平衡的，它就成为 AVL树。如果它有 n 个结点，其高度可保持在$O(\log_2 n)$，平均查找长度也可保持在$O(\log_2 n)$。

# AVL树的类型定义

```
typedef struct BSTNode {
        ElemType data;
        struct BSTNode *lchild, *rchild;
        int bf;
} BSTNode,*BSTree;
```

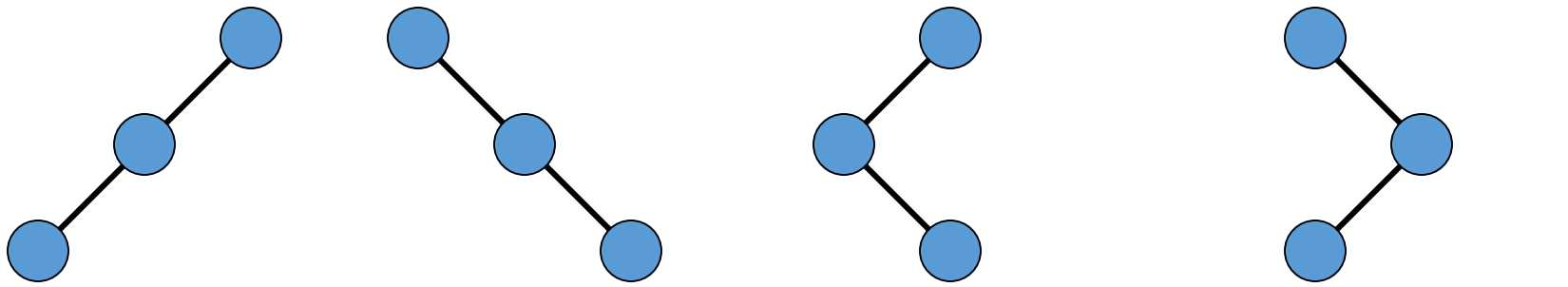
**基本操作：**

```
void R_Rotate(BSTree &p);
void L_Rotate(BSTree &p);
Status InsertAVL(BSTree &T,ElemType
e,Boolean &taller);
void LeftBalance(BSTree &T);
void RightBalance(BSTree &T);
```

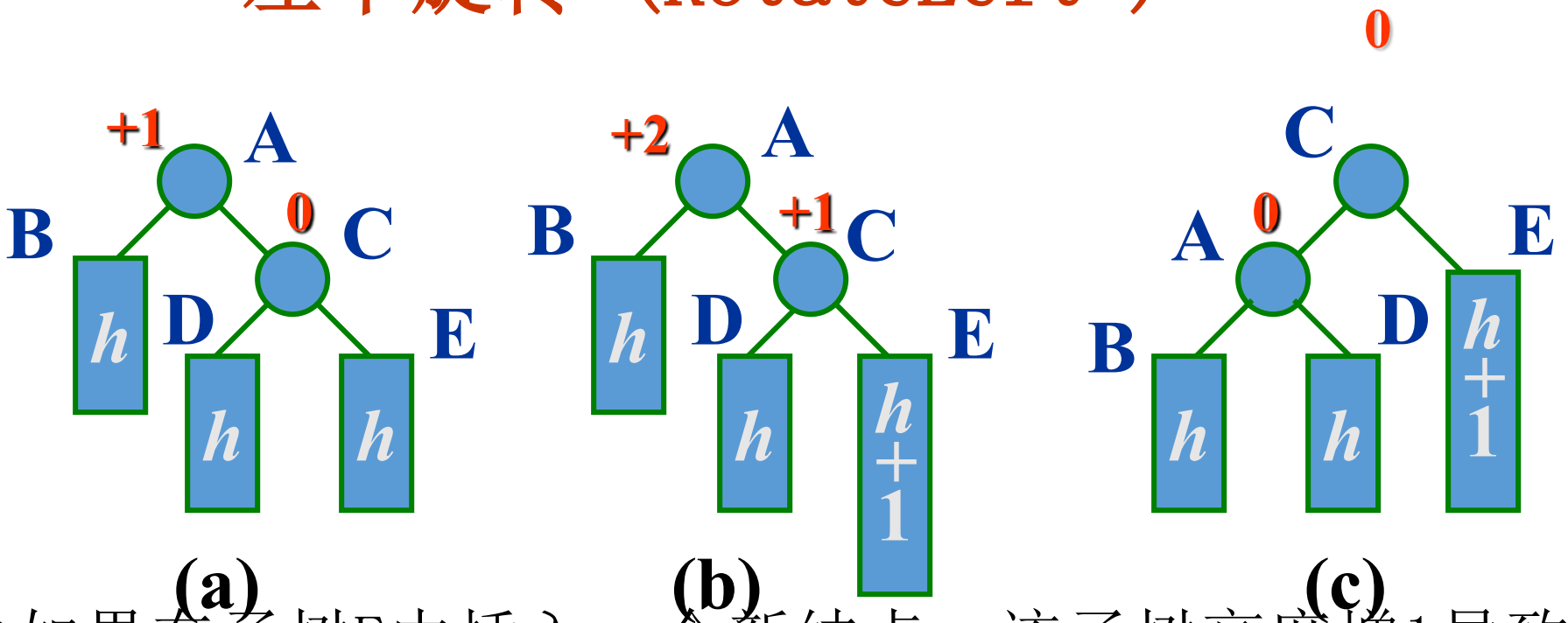
# 平衡化旋转

- 如果在一棵平衡的二叉查找树中插入一个新结点，造成了不平衡。此时必须调整树的结构，使之平衡化。
- 平衡化旋转有两类：
  - 单旋转（左旋和右旋）
  - 双旋转（左平衡和右平衡）
- 每插入一个新结点时，AVL树中相关结点的平衡状态会发生改变。因此，在插入一个新结点后，需要从插入位置沿通向根的路径回溯，检查各结点的平衡因子(左、右子树的高度差)。
- 如果在某一结点发现高度不平衡，停止回溯。

- 从发生不平衡的结点起，沿刚才回溯的路径取直接下两层的结点。
- 如果这三个结点处于一条直线上，则采用单旋转进行平衡化。单旋转可按其方向分为左单旋转和右单旋转，其中一个是另一个的镜像，其方向与不平衡的形状相关。
- 如果这三个结点处于一条折线上，则采用双旋转进行平衡化。双旋转分为先左后右和先右后左两类。



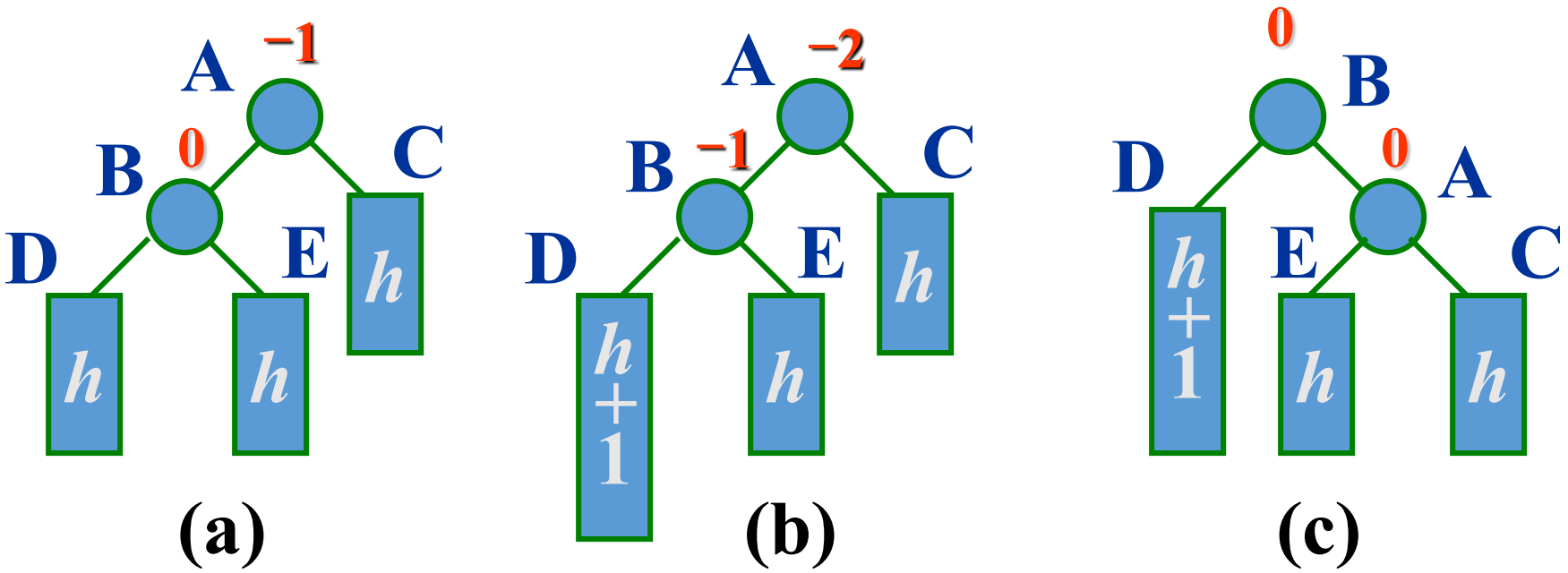
# 左单旋转（RotateLeft ）




- 如果在子树E中插入一个新结点，该子树高度增1导致结点A的平衡因子变成+2，出现不平衡。
- 沿插入路径检查三个结点A、C和E。它们处于一条方向为“\”的直线上，需要做左单旋转。
- 以结点C为旋转轴，让结点A反时针旋转。

```
void L_Rotate(BSTree &p){
//左单旋转的算法(p236 算法9.10)
        BSTree rc;
        rc=p->rchild;
        p->rchild=rc->lchild;
        rc->lchild=p; p=rc;
}
```

# 右单旋转（RotateRight ）

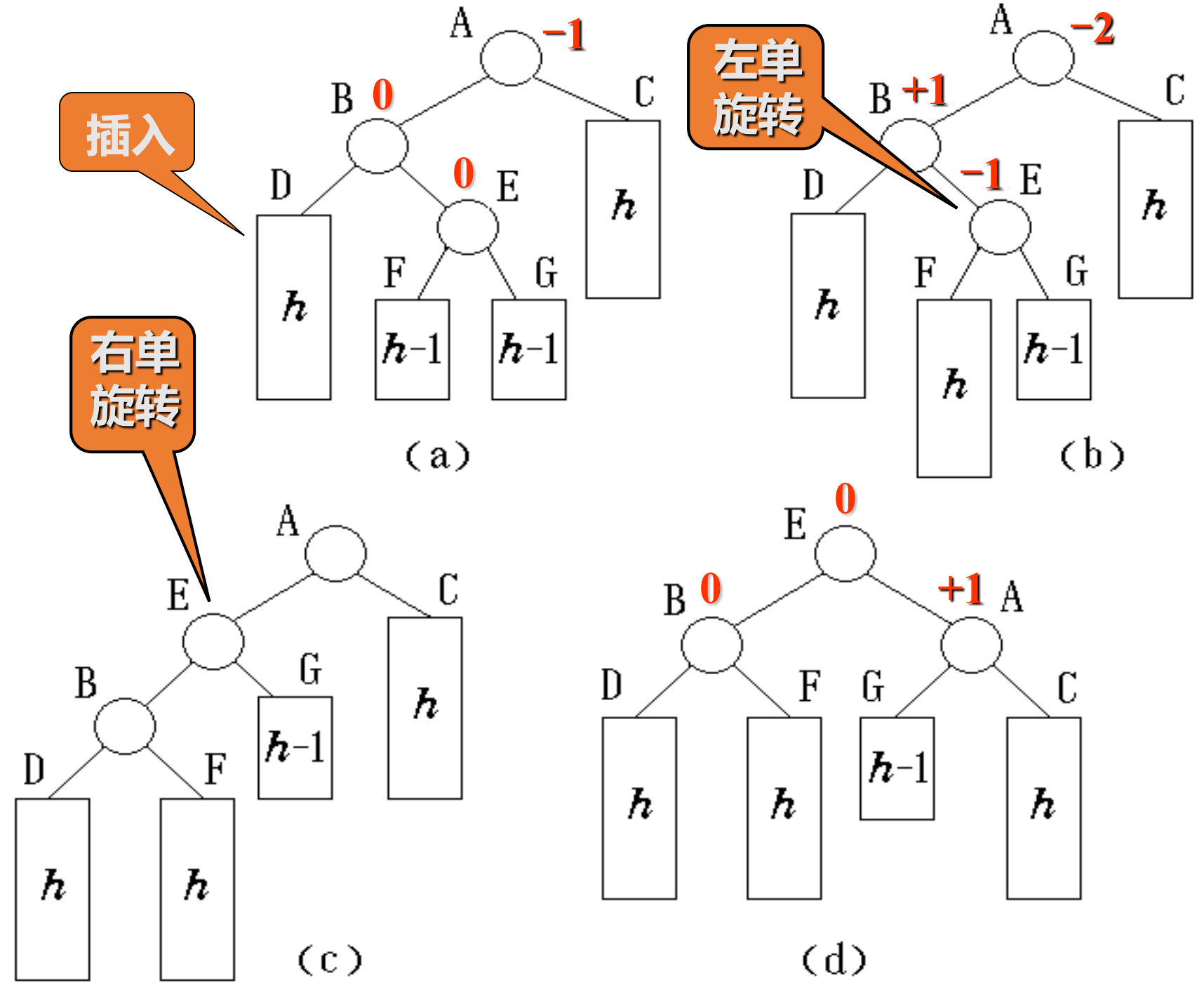



- 在左子树D上插入新结点使其高度增1，导致结点A的平衡因子增到 -2，造成了不平衡。
- 为使树恢复平衡，从A沿插入路径连续取3个结点A、B和D，它们处于一条方向为“/”的直线上，需要做右单旋转。
- 以结点B为旋转轴，将结点A顺时针旋转。

```
void R_Rotate(BSTree &p){
//右单旋转的算法（p236 算法9.9）
        BSTree lc;
        lc=p->lchild;
        p->lchild=lc->rchild;
        lc->rchild=p; p=lc;
}
```

# 先左后右双旋转 (RotationLeftRight)

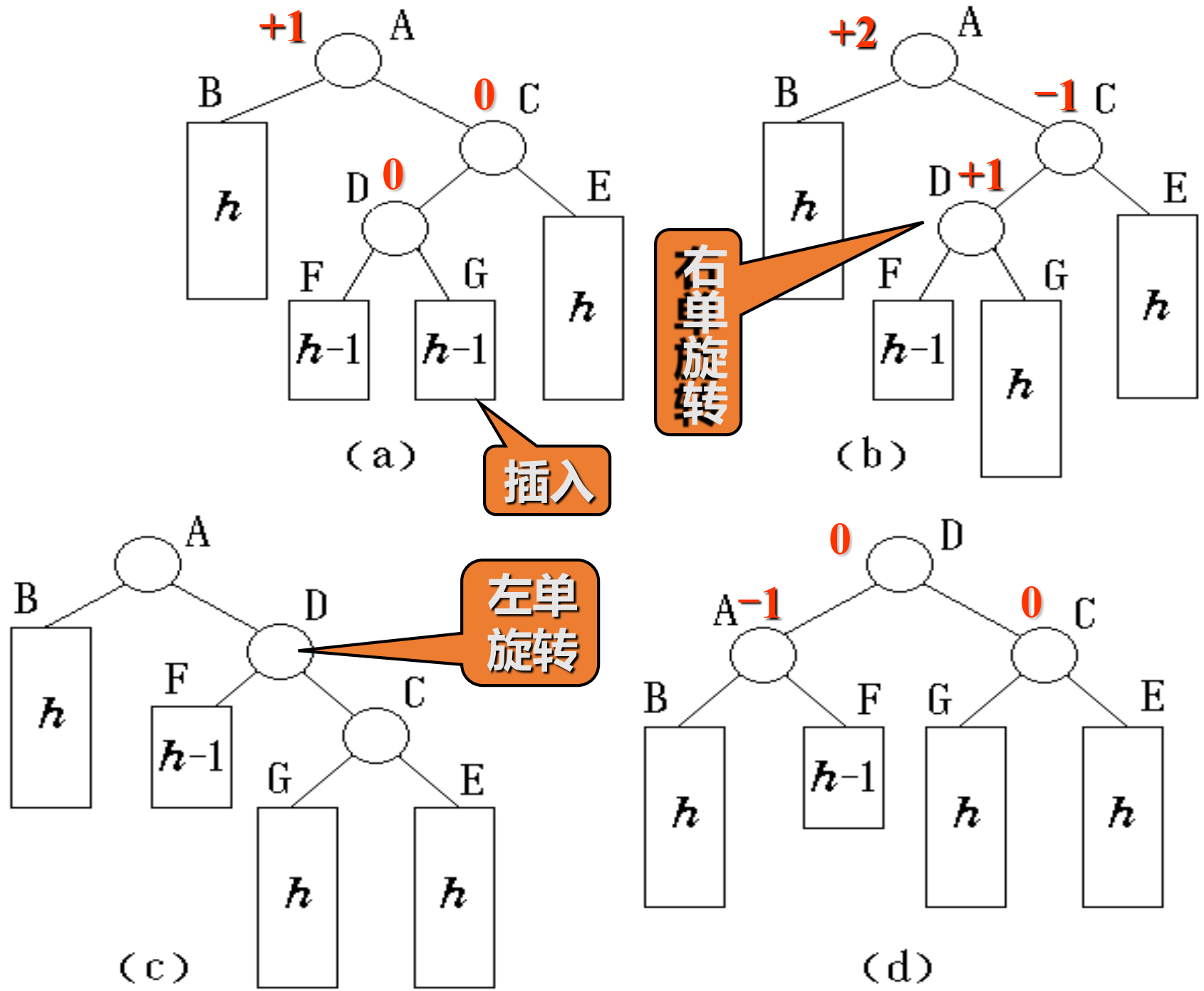
- 在子树F或G中插入新结点，该子树的高度增1。结点A的平衡因子变为 -2，发生了不平衡。
- 从结点A起沿插入路径选取3个结点A、B和E，它们位于一条形如“〈”的折线上，因此需要进行先左后右的双旋转。
- 首先以结点E为旋转轴，将结点B反时针旋转，以E代替原来B的位置，做左单旋转。
- 再以结点E为旋转轴，将结点A顺时针旋转，做右单旋转。使之平衡化。


插入
A
-1
B
0
C
D
0
E
h
F
G
h
h-1
h-1
(a)
左单旋转
A
-2
B
+1
C
D
-1
E
h
F
G
h
h
h-1
(b)
右单旋转
A
E
C
B
G
h
D
F
h-1
h
h
(c)
E
0
B
0
+1
A
D
F
G
C
h
h
h-1
h
(d)

```
void LeftBalance(BSTree &T){//左平衡化的算法
      BSTree lc,rd;    lc=T->lchild;
      switch(lc->bf){
      case LH:   T->bf = lc->bf = EH;
            R_Rotate(T); break;
      case RH:   rd=lc->rchild;
            switch(rd->bf){
            case LH: T->bf=RH;  lc->bf=EH; break;
            case EH: T->bf=lc->bf=EH; break;
            case RH: T->bf = EH; lc->bf=LH; break; }
            rd->bf=EH;
            L_Rotate(T->lchild);
            R_Rotate(T);
      }
}
```

# 先右后左双旋转（RotationRightLeft）

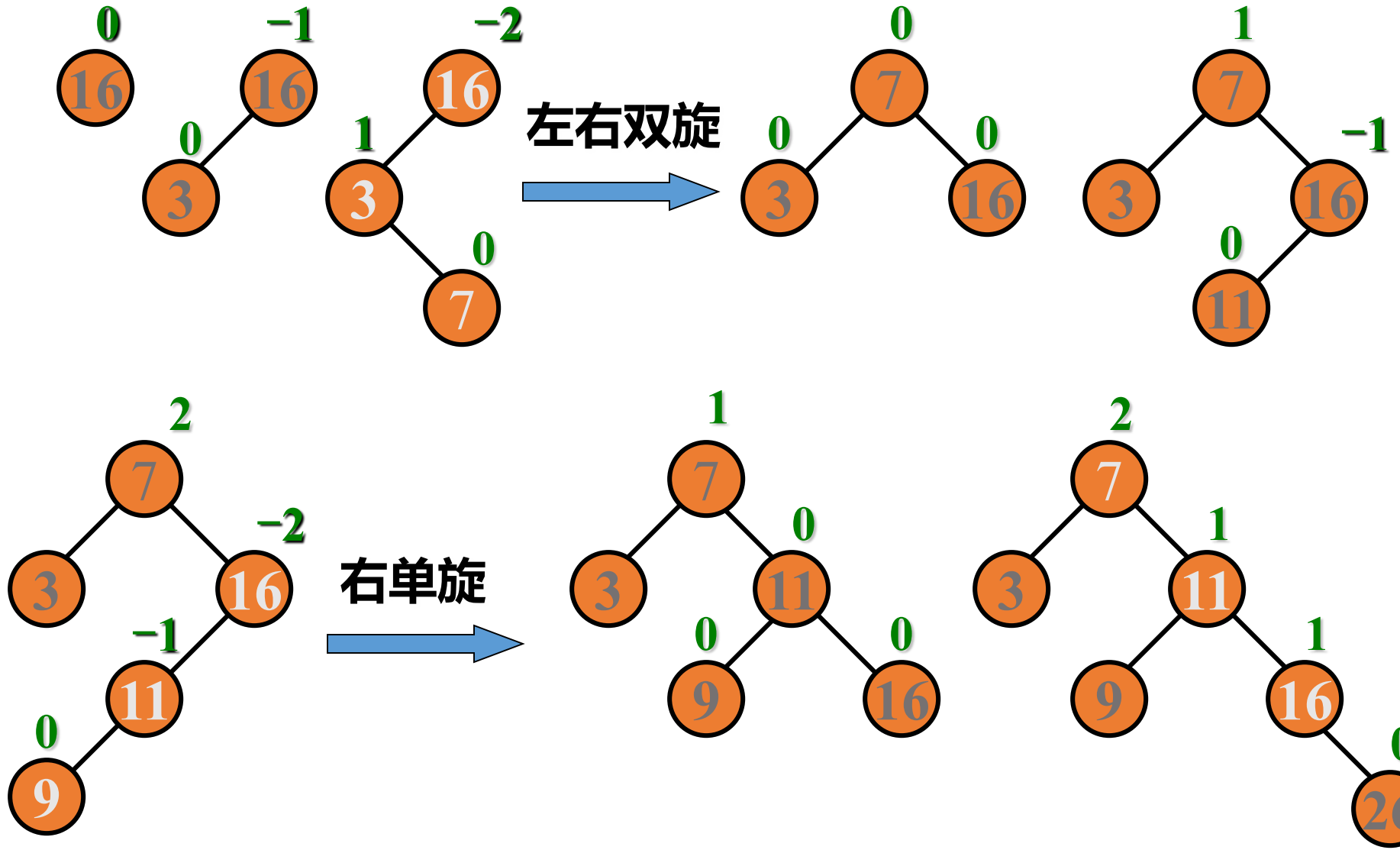
- 右左双旋转是左右双旋转的镜像。
- 在子树F或G中插入新结点，该子树高度增1。结点A的平衡因子变为2，发生了不平衡。
- 从结点A起沿插入路径选取3个结点A、C和D，它们位于一条形如“〉”的折线上，需要进行先右后左的双旋转。
- 首先做右单旋转：以结点D为旋转轴，将结点C顺时针旋转，以D代替原来C的位置。
- 再做左单旋转：以结点D为旋转轴，将结点A反时针旋转，恢复树的平衡。

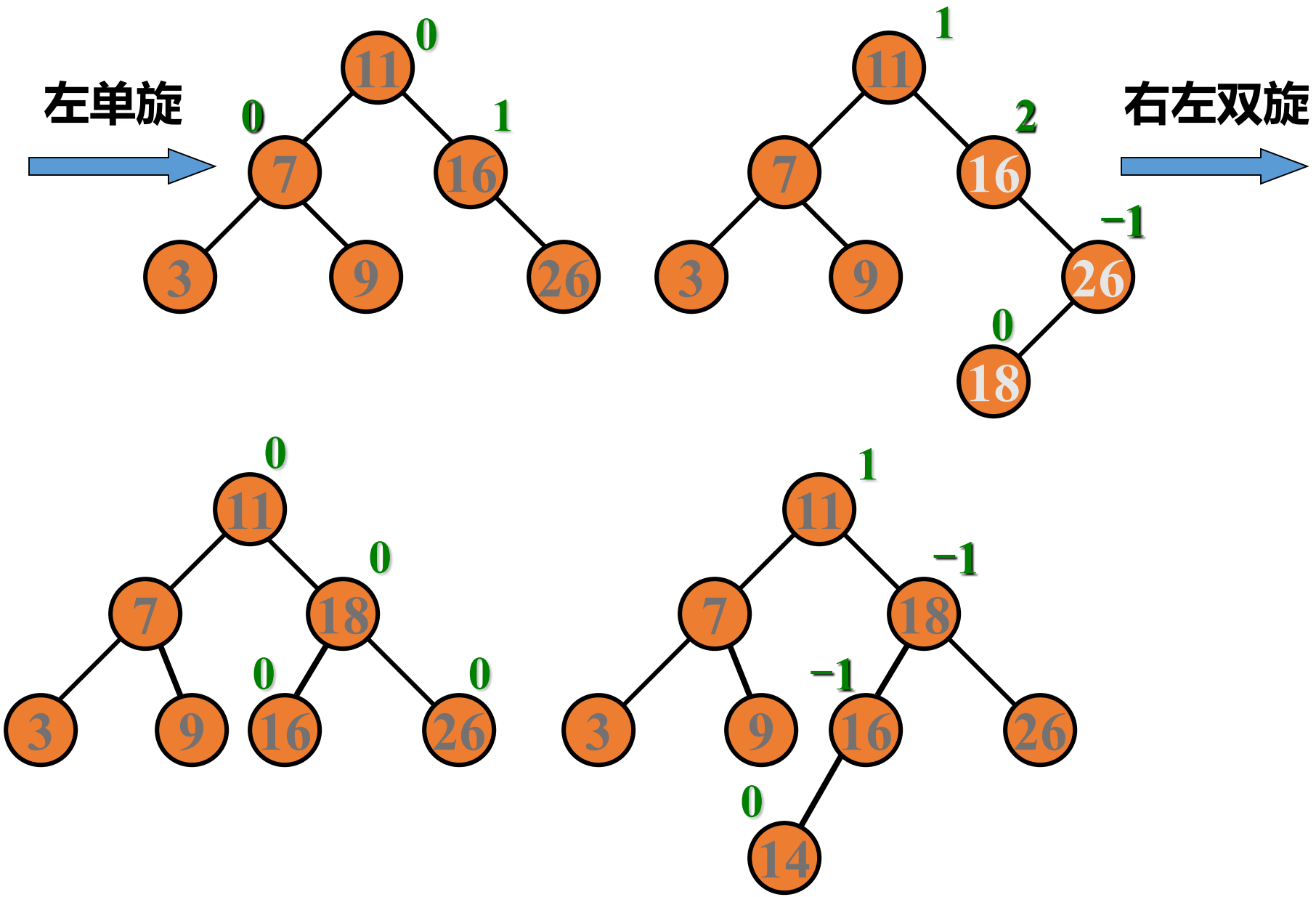

+1
A
B
0
C
D
0
E
F
G
h
h-1
h-1
h
(a)
插入
+2
A
B
-1
C
D
+1
E
F
G
h
h-1
h
h
(b)
右单旋转
A
B
D
F
C
G
E
h
h-1
h
h
(c)
左单旋转
0
D
A
-1
0
C
B
F
G
E
h
h-1
h
h
(d)

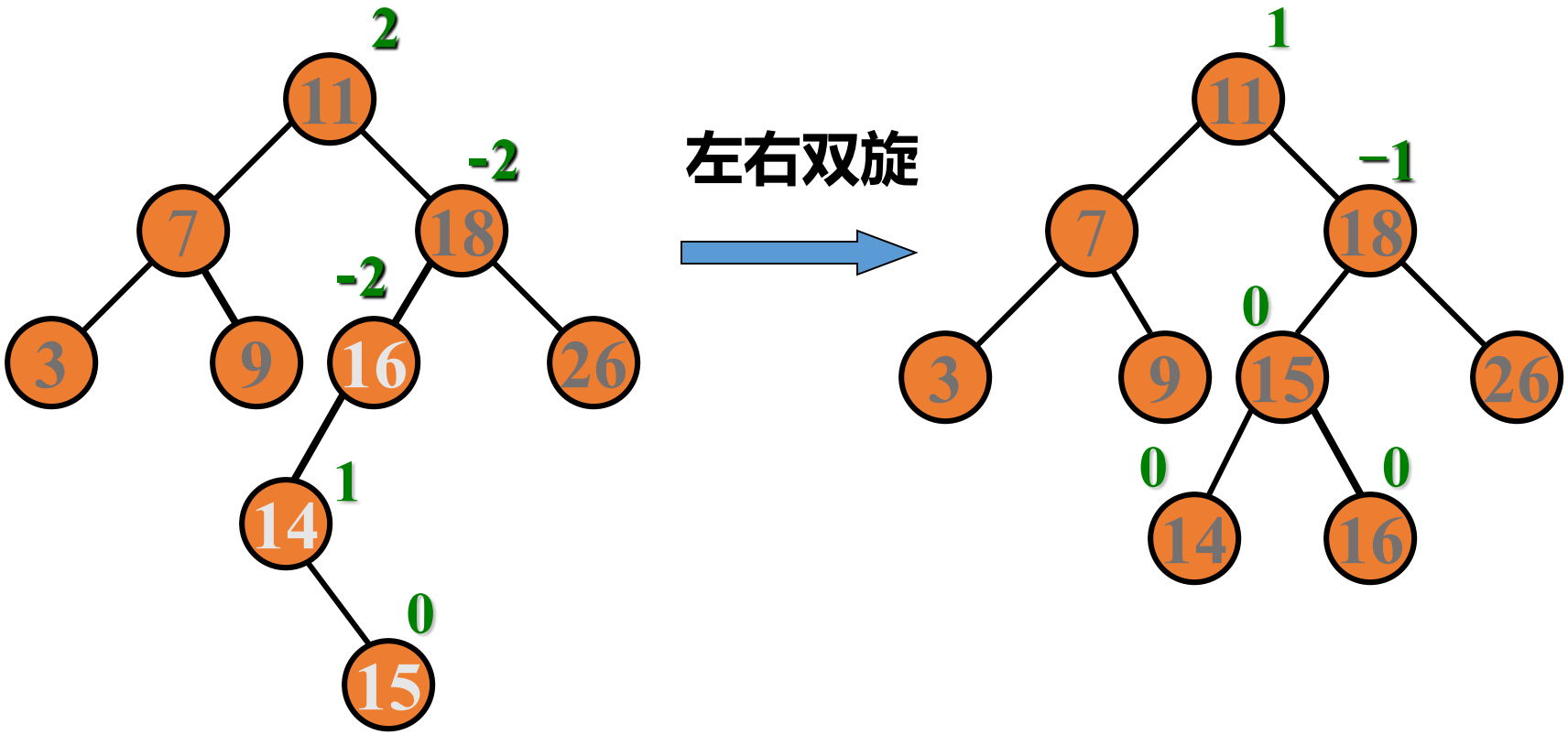
# *AVL*树的插入

- 在向一棵本来是高度平衡的*AVL*树中插入一个新结点时，如果树中某个结点的平衡因子的绝对值 $|balance| > 1$，则出现了不平衡，需要做平衡化处理。
- 算法从一棵空树开始，通过输入一系列对象的关键字，逐步建立*AVL*树。在插入新结点时使用了前面所给的算法进行平衡旋转。

例，输入关键字序列为｛16，3，7，11，9，26，18，14，15｝，插入和调整过程如下。




左单旋
右左双旋
11
7
16
3
9
26
18
14
0
1
2
−1



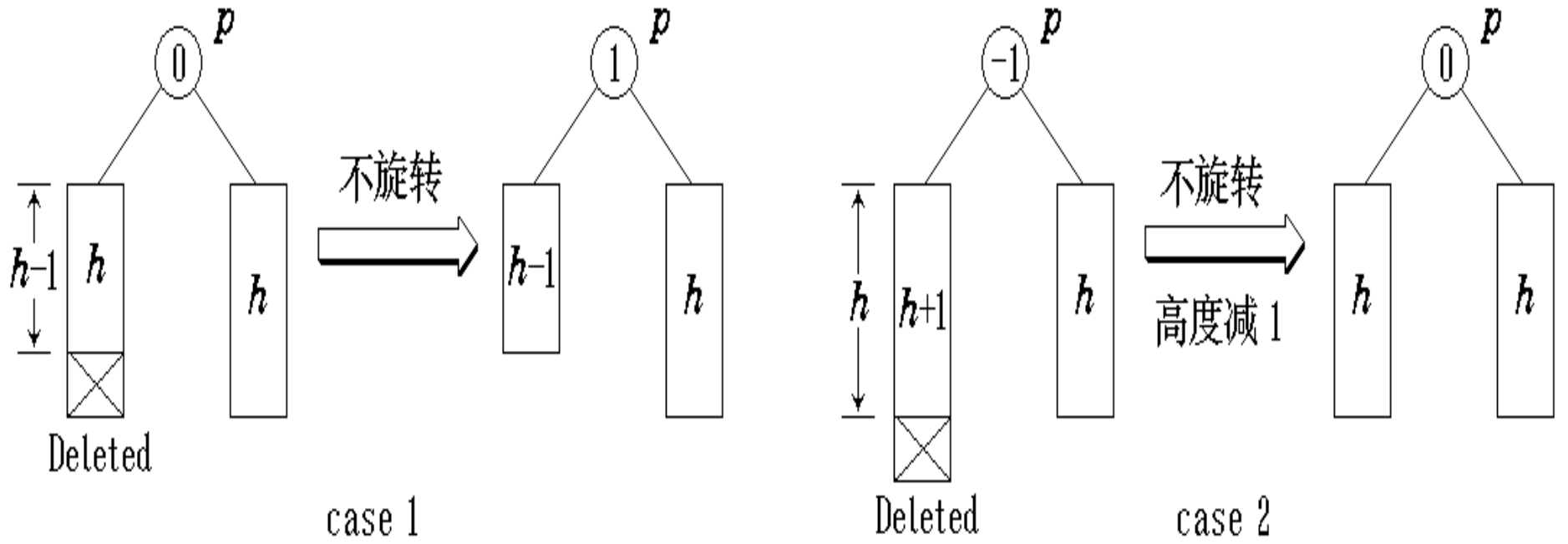

# 从空树开始的建树过程

# AVL树的删除

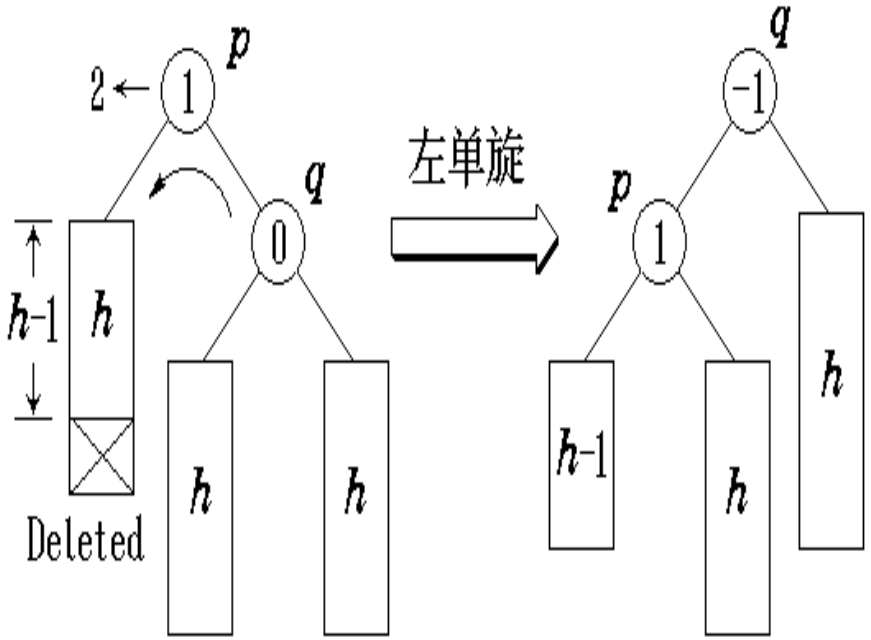
- 如果被删结点x最多只有一个子女，那么问题比较简单。如果被删结点x有两个子女，首先查找 x 在中序次序下的直接前驱 y（同样可以找直接后继）。再把 结点y 的内容传送给结点x，现在问题转移到删除结点 y。
- 把结点y当作被删结点x。
- 将结点x从树中删去。因为结点x最多有一个子女，我们可以简单地把x的双亲结点中原来指向x的指针改指到这个子女结点；如果结点x没有子女，x双亲结点的相应指针置为NULL。然后将原来以结点x为根的子树的高度减1，

- 必须沿 x 通向根的路径反向追踪高度的变化对路 径上各个结点的影响。
- 用一个布尔变量 shorter 来指明子树的高度是否被缩短。在每个结点上要做的操作取决于 shorter 的值和结点的 balance，有时还要依赖子女的 balance 。
- 布尔变量 shorter 的值初始化为True。然后对于从 x 的双亲到根的路径上的各个结点 p，在 shorter 保持为 True 时执行下面的操作。如果 shorter 变成False，算法终止。

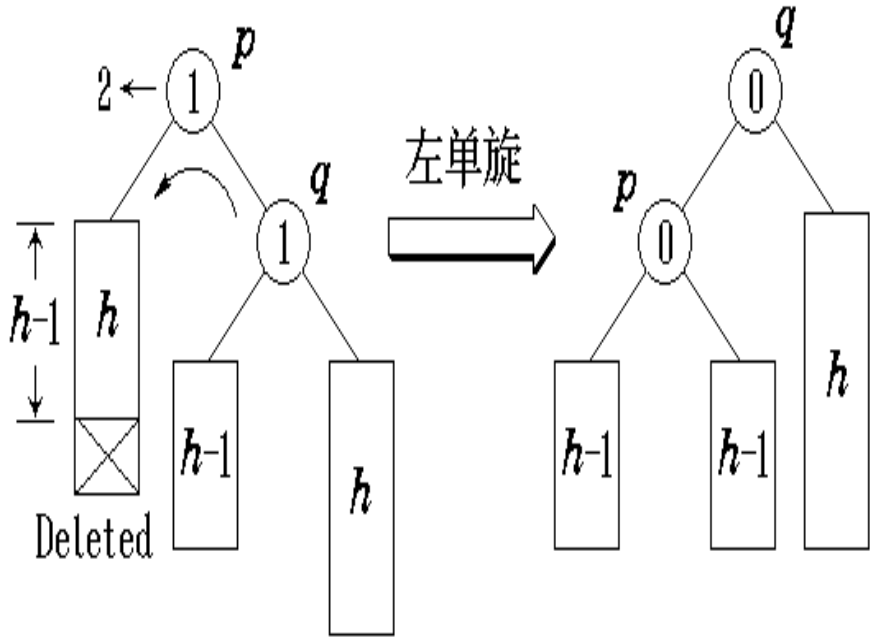
- case 1 ： 当前结点 p 的balance为0。如果它的左子树或右子树被缩短，则它的 balance改为 1 或-1，同时 shorter 置为False。
- case 2 ： 结点 p 的balance不为0，且较高的子树被缩短，则 p 的balance改为0，同时 shorter 置为True。



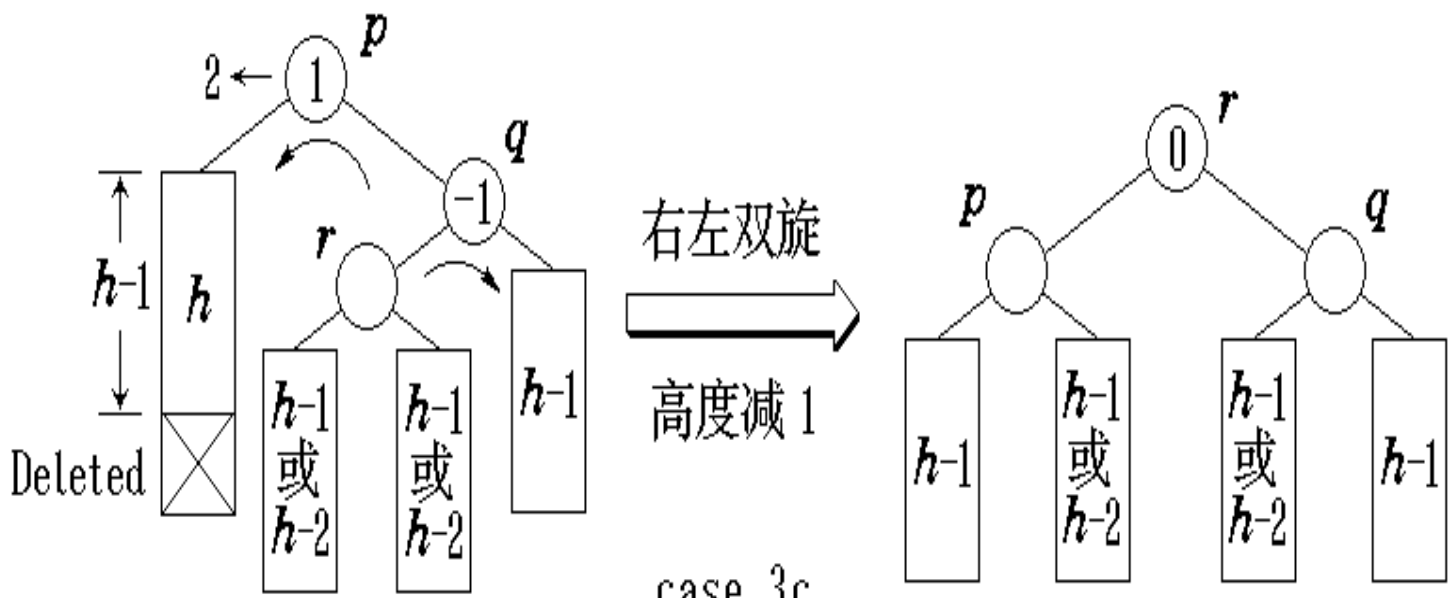
- case 3 ：结点 p 的balance不为0，且较矮的子树又被缩短，则在结点 p 发生不平衡。需要进行平衡化旋转来恢复平衡。令 p 的较高的子树的根为 q（该子树未被缩短），根据 q 的balance，有如下 3 种平衡化操作。
- case 3a ：如果 q 的balance为0，执行一个单旋转来恢复结点 p 的平衡，置shorter为False。
- case 3b ：如果 q 的balance与 p 的balance相同，则执行一个单旋转来恢复平衡，结点 p 和 q 的balance均改为0，同时置shorter为True。




case 3a

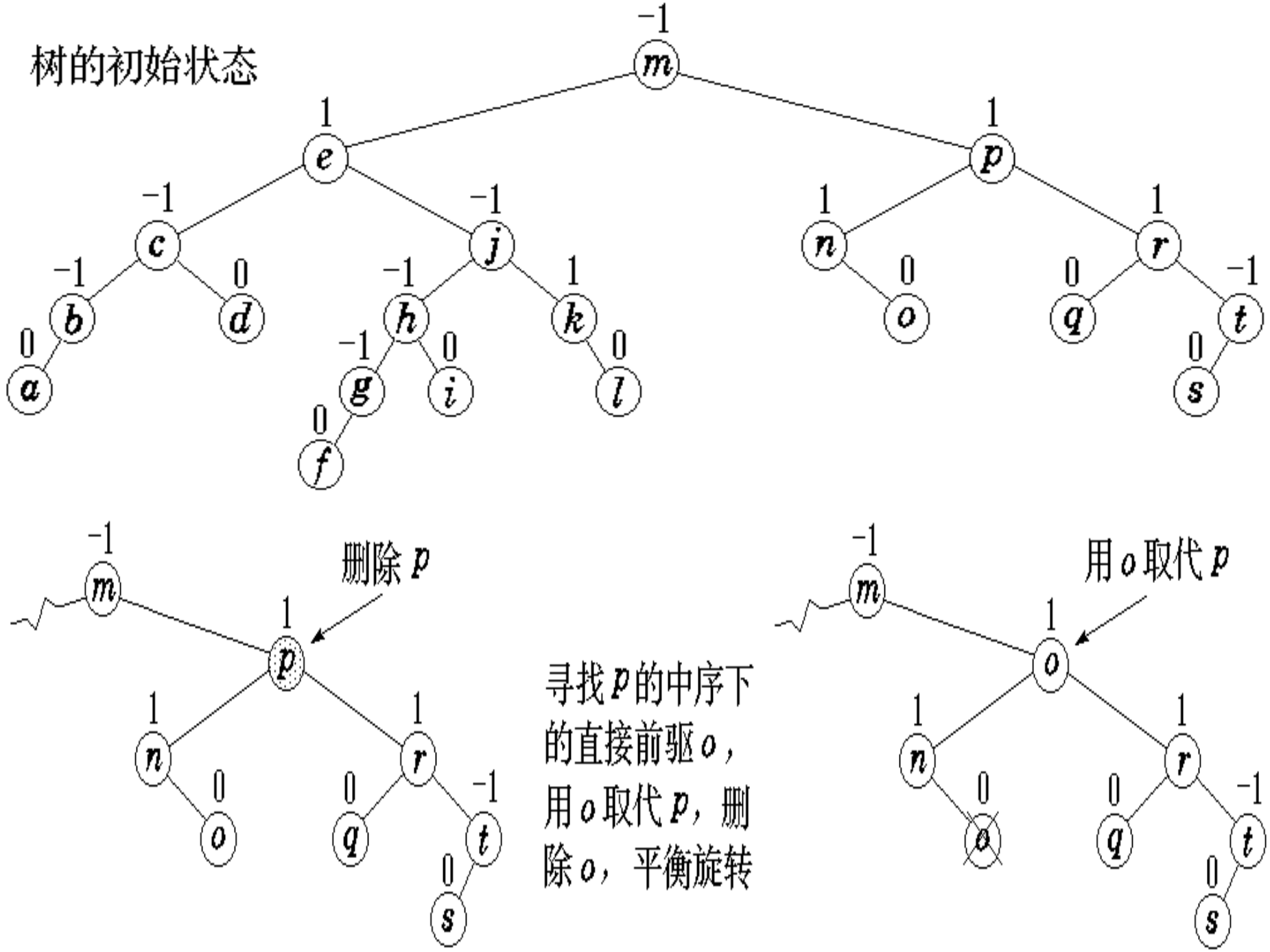


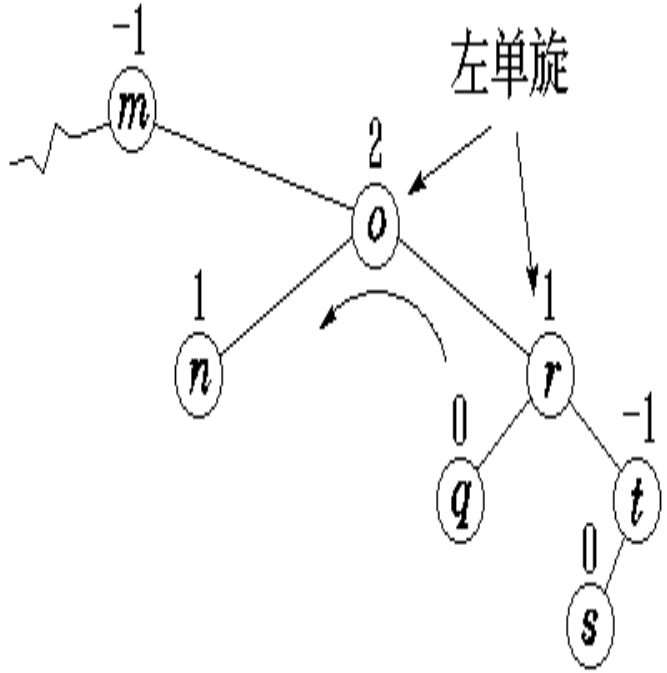

case 3b



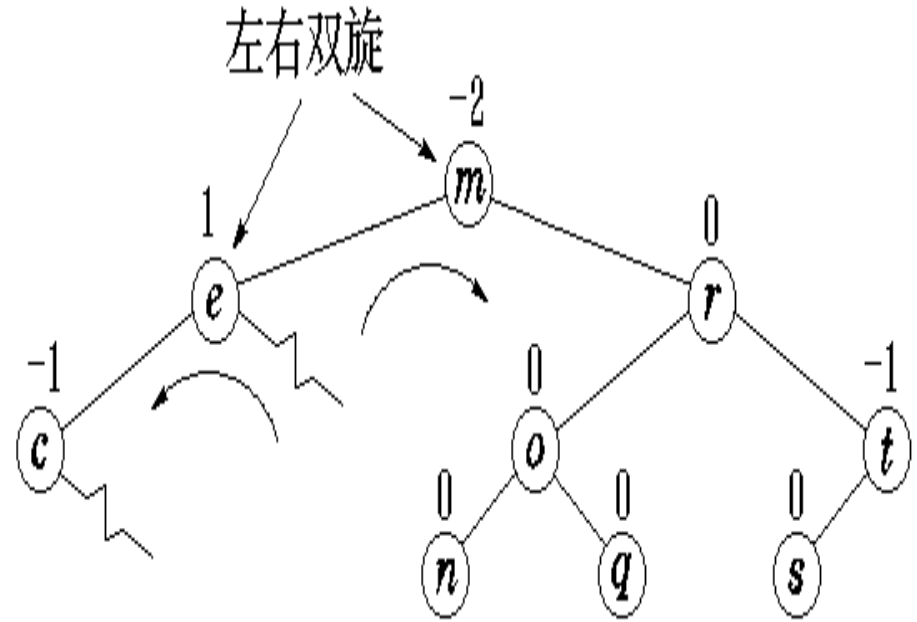

case 3c

- case 3c ： 如果 p 与 q 的balance相反，则执行一个双旋转来恢复平衡，先围绕 q 转再围绕 p 转。新的根结点的balance置为0，其它结点的balance相应处理，同时置shorter为True。
- 在case 3a, 3b和3c的情形中，旋转的方向取决于是结点 p 的哪一棵子树被缩短。

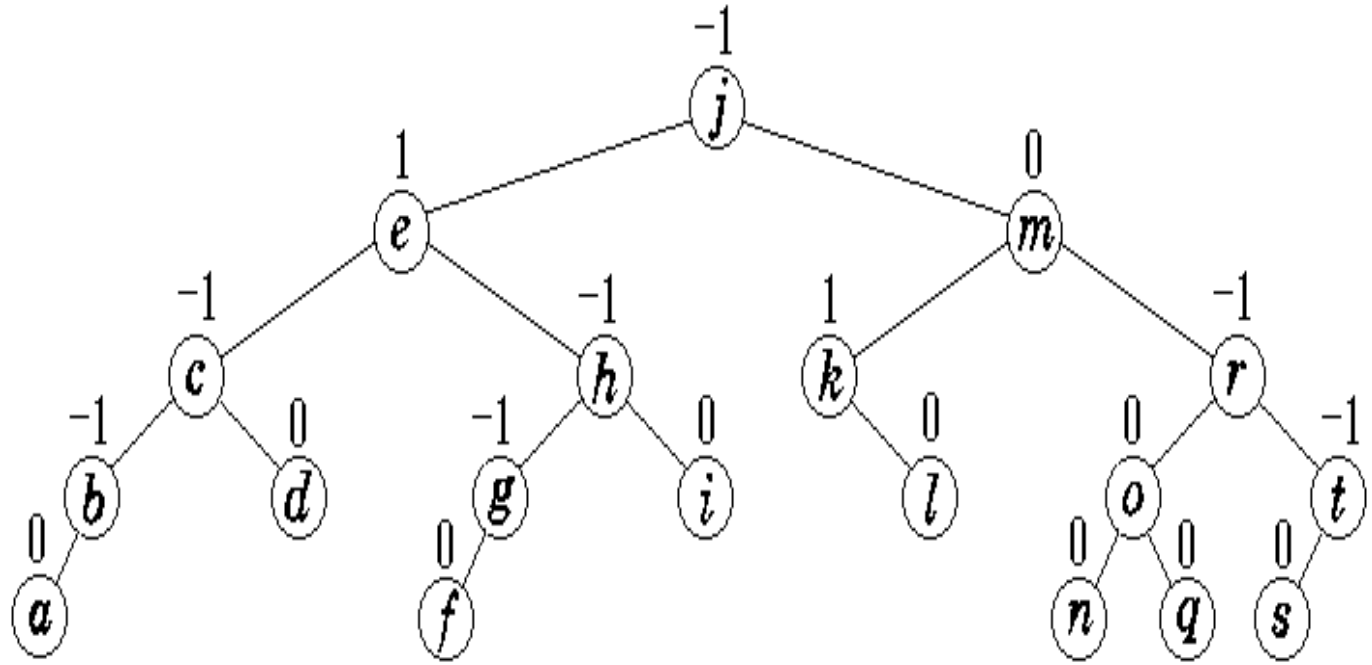

树的初始状态
删除 p
寻找 p 的中序下的直接前驱 o，用 o 取代 p，删除 o，平衡旋转
用 o 取代 p




以 $r$ 为旋转轴作左单旋，子树的高度减 1 $m$ 发生不平衡




首先以 $j$ 为旋转轴作左单旋，再以 $j$ 为旋转轴作右单旋，让 $e$ 成为 $j$ 的左子女，$m$ 成为 $j$ 的右子女。树的高度减 1



# AVL树的高度

- 设在新结点插入前AVL树的高度为 h，结点个数为 n，则插入一个新结点的时间是O(h)。对于AVL树来说，h多大？
- 设 $N_h$ 是高度为 h 的AVL树的最小结点数。根的一棵子树的高度为 h-1，另一棵子树的高度为 h-2，这两棵子树也是高度平衡的。因此有
  - $N_{-1} = 0$ （空树）
  - $N_0 = 1$ （仅有根结点）
  - $N_h = N_{h-1} + N_{h-2} + 1$ ，$h > 0$
- 可以证明，对于 $h \geq 0$，有 $N_h = F_{h+3} - 1$ 成立。

- 有 n 个结点的AVL树的高度不超过

$$\frac{3}{2}\log_2(n+1)$$

- 在AVL树删除一个结点并做平衡化旋转所需时间为 $O(\log_2 n)$。
- 二叉查找树适合于组织在内存中的较小的索引(或目录)。对于存放在外存中的较大的文件系统，用二叉查找树来组织索引不太合适。
- 在文件检索系统中大量使用的是用B_树或B+树做文件索引。

# 9.3 哈希(散列)查找

**基本思想**：在记录的存储地址和它的关键字之间建立一个确定的对应关系；这样，不经过比较，一次存取就能得到所查元素的查找方法。

例 30个地区的各民族人口统计表

| 编号 | 省、市(区) | 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

例：

已知线性表关键码集合为： S = { and, array, begin, do, else, end, for, go, if, repeat, then, until, while, with}

可设散列表为： char ht2[26][8]

散列函数1: h1[key]=key[0]-’ a’

| 散列地址 | 关键码 |
|---|---|
| 0 | (and, array) |
| 1 | begin |
| 2 | |
| 3 | do |
| 4 | (end, else) |
| 5 | for |
| 6 | go |
| 7 | |
| 8 | if |
| 9 | |
| 10 | |
| 11 | |
| 12 | |

| 散列地址 | 关键码 |
|---|---|
| 13 | |
| 14 | |
| 15 | |
| 16 | |
| 17 | repeat |
| 18 | |
| 19 | then |
| 20 | until |
| 21 | |
| 22 | (while, with) |
| 23 | |
| 24 | |
| 25 | |

# 散列函数2：首尾字母在字母表中序号的平均值

```
int H3(char key[]) {
    int i = 0;    while ((i<8) && (key[i]!= '\0')) i++;
   return((key[0] + key(i-1) - 2*' a' ) /2 )
}
```

| 散列地址 | 关键码 |
|---|---|
| 0 | |
| 1 | and |
| 2 | |
| 3 | end |
| 4 | else |
| 5 | |
| 6 | if |
| 7 | begin |
| 8 | do |
| 9 | |
| 10 | go |
| 11 | for |
| 12 | array |

| 散列地址 | 关键码 |
|---|---|
| 13 | while |
| 14 | with |
| 15 | until |
| 16 | then |
| 17 | |
| 18 | repeat |
| 19 | |
| 20 | |
| 21 | |
| 22 | |
| 23 | |
| 24 | |
| 25 | |

# 9.3.1 基本概念

**哈希函数：** 在记录的关键字与记录的存储地址之间建立的一种对应关系叫哈希函数。

哈希函数是一种映像，是从关键字空间到存储地址空间的一种映像。可写成：$addr(a_i)=H(k_i)$，其中i是表中一个元素，$addr(a_i)$是$a_i$的地址，$k_i$是$a_i$的关键字。

**哈希表：** 应用哈希函数，由记录的关键字确定记录在表中的地址，并将记录放入此地址，这样构成的表叫哈希表。

**哈希查找(又叫散列查找)**：利用哈希函数进行查找的过程叫哈希查找。

**冲突**：对于不同的关键字$k_i$、$k_j$，若$k_i \neq k_j$，但$H(k_i)=H(k_j)$的现象叫冲突(collision) 。

**同义词**：具有相同函数值的两个不同的关键字，称为该哈希函数的同义词。

**负载因子**：$\boldsymbol{\alpha = \frac{n}{m}}$

m:哈希表的空间大小

n:填入表中的结点数

哈希函数通常是一种压缩映像，所以冲突不可避免，只能尽量减少；当冲突发生时，应该有处理冲突的方法。

设计一个散列表应包括：

① 散列表的空间范围，即确定散列函数的值域；

② 构造合适的散列函数，使得对于所有可能的元素(记录的关键字)，函数值均在散列表的地址空间范围内，且出现冲突的可能尽量小；

③ 处理冲突的方法。即当冲突出现时如何解决。

# 9.3.2 哈希函数的构造

哈希函数是一种映象，其设定很灵活，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。

哈希函数“好坏”的主要评价因素有：

- 散列函数的构造简单；
- 能“均匀”地将散列表中的关键字映射到地址空间。所谓“均匀”(uniform)是指发生冲突的可能性尽可能最少。

常用哈希函数选取方法：

除(留)余(数)法、平方取中法、数字分析法、折叠法、直接定址法、ELFhash字符串散列函数

# 1. 直接定址法

取关键字或关键字的某个线性函数作哈希地址，即 H(key)=key　　或　　H(key)=a • key+b(a,b为常数)

特点：直接定址法所得地址集合与关键字集合大小相等，不会发生冲突，但实际中很少使用。

| 序号 | 出生年份 | 出生人口 |
|---|---|---|
| 0 | 1995 | 7.11万 |
| 1 | 1996 | 6.79万 |
| 2 | 1997 | 6.42万 |
| 3 | 1998 | 6.17万 |
| 4 | 1999 | 6.56万 |
| 5 | 2000 | 6.95万 |
| 6 | 2001 | 5.76万 |
| 7 | 2002 | 6.2万 |
| 8 | 2003 | 5.73万 |
| 9 | 2004 | 8.09万 |

**H(key)=key-1995**

**出生年份：key**

**出生人口：value(attribute)**

# 2. 数字分析法

对关键字进行分析，取关键字的若干位或组合作为哈希地址。

适用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。

例如：取18位身份证号码的其中5位做关键字，可定义如下哈希函数：

```
H(key)=(key[6]-'0') *10^4 +
(key[10]-'0')*10^3+(key[14]-'0')*10^2 +
(key[16]-'0')*10+key[17]-'0'
```

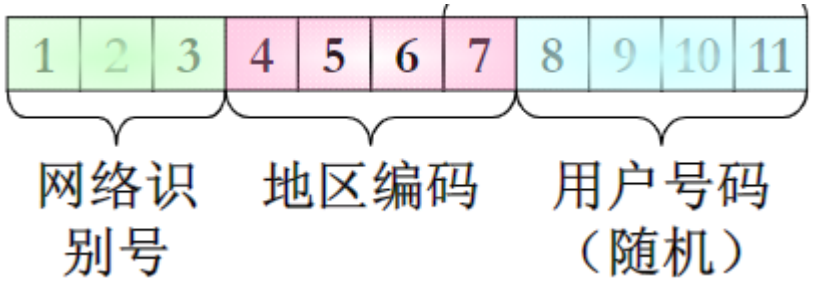
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 3 | 0 | 1 | 0 | 6 | 1 | 9 | 9 | 0 | 1 | 0 | 0 | 8 | 0 | 4 | 1 | 9 |
| 省 | | 市 | | 区（县）下属辖区编号 | | （出生）年份 | | | | 月份 | | 日期 | | 该辖区中的序号 | | | 校验 |

例： 设有80个记录，关键字为8位十进制数，哈希地址为2位十进制数。

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

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

例如：取手机号码的后四位做关键字；



# 3. 平方取中法

将关键字平方后取中间几位作为哈希地址。

一个数平方后中间几位和数的每一位都有关，则由随机分布的关键字得到的散列地址也是随机的。散列函数所取的位数由散列表的长度决定。这种方法适于不知道全部关键字情况，是一种较为常用的方法。

| 关键字 | （关键字）$^2$ | 函数地址 |
|---|---|---|
| 0100 | 0010000 | 010 |
| 1100 | 1210000 | 210 |
| 1200 | 1440000 | 440 |
| 1160 | 1370400 | 370 |
| 2061 | 4310541 | 310 |

# 4. 折叠法

将关键字分割成位数相同的几部分(最后一部分可以不同)，然后取这几部分的叠加和作为哈希地址。

数位叠加有移位叠加和间界叠加两种。适于关键字位数很多，且每一位上数字分布大致均匀情况。

- **移位叠加：**将分割后的几部分低位对齐相加。
- **间界叠加：**从一端到另一端沿分割界来回折迭，然后对齐相加。

例： 设关键字为0442205864，哈希地址位数为4 。

```
  5 8 6 4
  4 2 2 0
    0 4
-------------
  1 0 0 8 8
```

移位叠加

H(key)=0088

```
  5 8 6 4
  0 2 2 4
    0 4
-------------
  6 0 9 2
```

间界叠加

H(key)=6092

# 5. 除留余数法

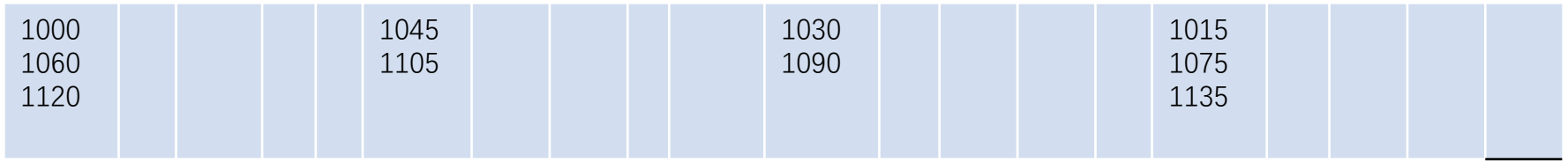
取关键字被某个不大于哈希表表长m的数p除后所得余数作哈希地址，即H(key)=key MOD p (p≤m)

是一种简单、常用的哈希函数构造方法。

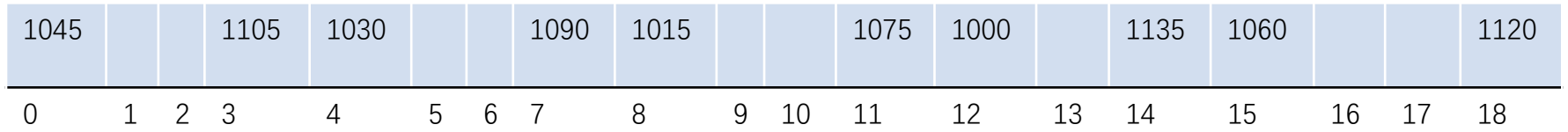
利用这种方法的关键是p的选取，p选的不好，容易产生同义词。p的选取的分析:

- 选取$p=2^i$(p≤m)：运算便于用移位来实现，但等于将关键字的高位忽略而仅留下低位二进制数。高位不同而低位相同的关键字是同义词。
- 选取p=q×f(q、f都是质因数，p≤m)：则所有含有q或f因子的关键字的散列地址均是q或f的倍数。
- 选取p为素数或p=q×f(q、f是质数且均大于20，p≤m)：常用的选取方法，能减少冲突出现的可能性。

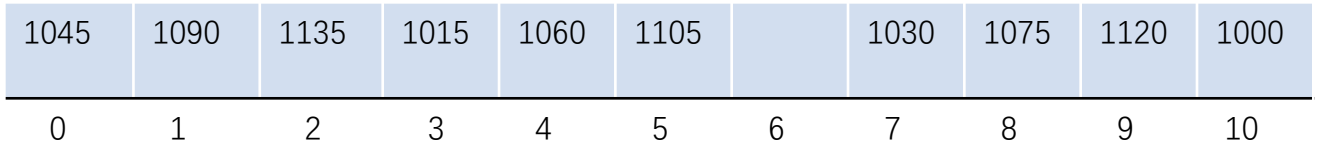
- 例如：有如下10个关键码{1000，1015，1030，1045，1060，1075，1090，1105，1120，1135}
- 散列函数：h(key)=key%M
- (a)表长=20, M=20




- (b)表长=19，M=19




- (c) 表长=11，M=11



# 6. 乘余取整法

先让关键码 key 乘上一个常数 A (0<A<1)，提取乘积的小数部分；

再用整数 n 乘以这个值，对结果向下取 整，把它作为散列地址

散列函数为： hash(key)= ⌊n * A * key % 1 ⌋)

“A * key % 1” 表示取 A * key 小数部分；

A * key % 1 = A * key - ⌊A * key ⌋ 。

例如：设关键码 key = 123456, n = 10000且取 A = = 0.6180339， hash(123456)= ⌊10000*123456 * 0.6180339 * % 1⌋ = ⌊10000*(76300.0041151…%1)⌋ = ⌊ 10000 * 0.0041151… ⌋ = 41

乘余取整法参数取值的考虑：

- 若地址空间为 p 位，就取 n = $2^p$
    - 所求出的散列地址正好是计算出来的
    - A * key % 1 = A * key - ⌊ A * key⌋ 值的小数点后最左 p 位（bit）值
    - 此方法的优点:对 n 的选择无关紧要
- Knuth认为：A 可以取任何值，与数据特征有关。一般情况下取黄金分割最理想

# 8. 基数转换法

- 把关键码看成是另一进制上的数后
- 再把它转换成原来进制上的数
- 取其中若干位作为散列地址
- 一般取大于原来基数的数作为转换的基数，并且两个基数要互素

例如：给定一个十进制数的关键码是(210485)10，把 它看成以 13 为基数的十三进制数 (210485)13 ，再把 它转换为十进制

- (210485)13 = 2×13^5 + 1×13^4 + 4×13^2 + 8×13 + 5 = (771932)10
- 假设散列表长度是 10000，则可取低 4 位1932 作为散列地址

## 选取哈希函数，考虑以下因素

- 计算哈希函数所需时间；
- 关键字的长度；
- 哈希表长度（哈希地址范围）；
- 关键字分布情况；
- 记录的查找频率。

# 9.3.3 冲突处理的方法

**冲突处理**：当出现冲突时，为冲突元素找到另一个存储位置。常用方法有如下几种：

- **开放定址法**
- **再哈希法**
- 链地址法
- 建立公共溢出区

# 1 开放定址法

**基本方法：** 当冲突发生时，形成某个探测序列；按此序列逐个探测散列表中的其他地址，直到找到给定的关键字或一个空地址(开放的地址)为止，将发生冲突的记录放到该地址中。

散列地址的计算公式是:

$H_i(key)=(H(key)+d_i)\ MOD\ m,\ i=1,\ 2,\ \cdots,\ k(k\leq m-1)$

其中：H(key)：哈希函数;

m：散列表长度;

$d_i$：第i次探测时的增量序列;

$H_i(key)$：经第i次探测后得到的散列地址。

# (1) 线性探测法

将散列表T[0 …m-1]看成循环向量。当发生冲突时，从初次发生冲突的位置依次向后探测其他的地址。

增量序列为：$d_i$=1，2，3，…，m-1

设初次发生冲突的地址是h，则依次探测T[h+1]，T[h+2]…，直到T[m-1]时又循环到表头，再次探测T[0]，T[1]…，直到T[h-1]。探测过程终止的情况是：

- **探测到的地址为空**：表中没有记录。若是查找则失败；若是插入则将记录写入到该地址；
- **探测到的地址有给定的关键字**：若是查找则成功；若是插入则失败；
- 直到T[h]：仍未探测到空地址或给定的关键字，散列表满。

**例：** 设散列表长为7，记录关键字组为：{15, 14, 28, 26, 56, 23}，散列函数：H(key)=key MOD 7，冲突处理采用线性探测法。

**解：** H(15)=15 MOD 7=1　　H(14)=14 MOD 7=0

H(28)=28 MOD 7=0 冲突 $H_1(28)=1$ 又冲突 $H_2(28)=2$

H(26)=26 MOD 7=5

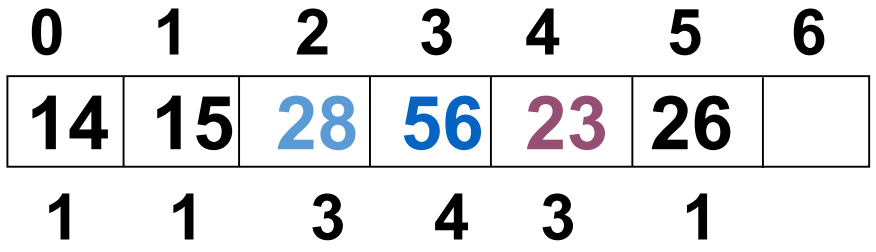
H(56)=56 MOD 7=0 冲突 $H_1(56)=1$ 又冲突 $H_2(56)=2$

又冲突 $H_3(56)=3$

H(23)=23 MOD 7=2 冲突 $H_1(23)=3$ 又冲突 $H_2(23)=4$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 14 | 15 | 28 | 56 | 23 | 26 | |
| 1 | 1 | 3 | 4 | 3 | 1 | |

{15, 14, 28, 26, 56, 23}




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 说明 |
|---|---|---|---|---|---|---|---|---|
| 插入15 | | 15 | | | | | | 无冲突 |
| 插入14 | 14 | 15 | | | | | | 无冲突 |
| 插入28 | 14 | 15 | 28 | | | | | d2=2 |
| 插入26 | 14 | 15 | 28 | | | 26 | | 无冲突 |
| 插入56 | 14 | 15 | 28 | 56 | | 26 | | d3=3 |
| 插入23 | 14 | 15 | 28 | 56 | 23 | 26 | | d2=2 |
| | | | | | | | | |

**平均查找长度=(1+1+3+4+3+1)/6=2.16**

# 线性探测法的特点

- **优点**：只要散列表未满，总能找到一个不冲突的散列地址；
- **缺点**：每个产生冲突的记录被散列到离冲突最近的空地址上，从而又增加了更多的冲突机会(这种现象称为冲突的“聚集”)。

## (2) 二次探测法

增量序列为：$d_i=1^2,-1^2,2^2,-2^2,3^2,\cdots\cdots\pm k^2$ $(k\le\lfloor m/2\rfloor)$

**解：** H(15)=15 MOD 7=1

H(14)=14 MOD 7=0

H(28)=28 MOD 7=0 冲突 $H_1(28)=1$ 又冲突$H_2(28)=6$

H(26)=26 MOD 7=5

H(56)=56 MOD 7=0 冲突 $H_1(56)=1$ 又冲突$H_2(56)=6$

又冲突 $H_3(56)=4$

H(23)=23 MOD 7=2

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 14 | 15 | 23 | | 56 | 26 | 28 |

**平均查找长度**
**=(1+1+1+4+1+3)/6**
**=1.83**

## 二次探测法的特点：

- **优点**：探测序列跳跃式地散列到整个表中，不易产生冲突的“聚集”现象；
- **缺点**：不能保证探测到散列表的所有地址。

例如：关键字{0，4，1，5，10}，表长为5，M=5。

H(key)=key%5

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 0 | 1 | | | 4 |

(1)插入0

(2)插入4

(3)插入1

(4)插入5：0，1，4，4，1，1，4，0，1，4…无法插入

## (3) 伪随机探测法

增量序列使用一个伪随机函数来产生一个落在闭区间[1，m-1]的随机序列。

例 ： 表长为11的哈希表中已填有关键字为17，60，29的记录，散列函数为H(key)=key MOD 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。

(1) H(38)=38 MOD 11=5 冲突

H1=(5+1) MOD 11=6 冲突

H2=(5+2) MOD 11=7 冲突

H3=(5+3) MOD 11=8 不冲突

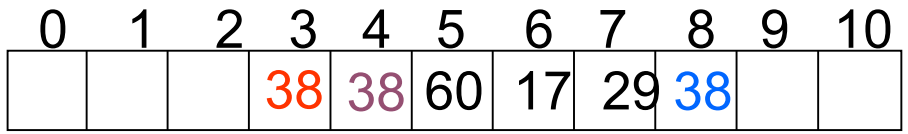
(2) H(38)=38 MOD 11=5 冲突

$H_1$=(5+$1^2$) MOD 11=6 冲突

$H_2$=(5-$1^2$) MOD 11=4 不冲突

(3) H(38)=38 MOD 11=5 冲突

设伪随机数序列为9，则$H_1$=(5+9) MOD 11=3 不冲突



# 2 再哈希法

构造若干个哈希函数，当发生冲突时，利用不同的哈希函数再计算下一个新哈希地址，直到不发生冲突为止。即：$H_i=RH_i(key)$　　$i=1，2，\cdots，k$

$RH_i$：一组不同的哈希函数。第一次发生冲突时，用$RH_1$计算，第二次发生冲突时，用$RH_2$计算…依此类推知道得到某个$H_i$不再冲突为止。

- 优点：不易产生冲突的“聚集”现象；
- 缺点：计算时间增加。

# 3　链地址法

**方法**：将所有关键字为同义词(散列地址相同)的记录存储在一个单链表中，并用一维数组存放链表的头指针。设散列表长为m，定义一个一维指针数组：

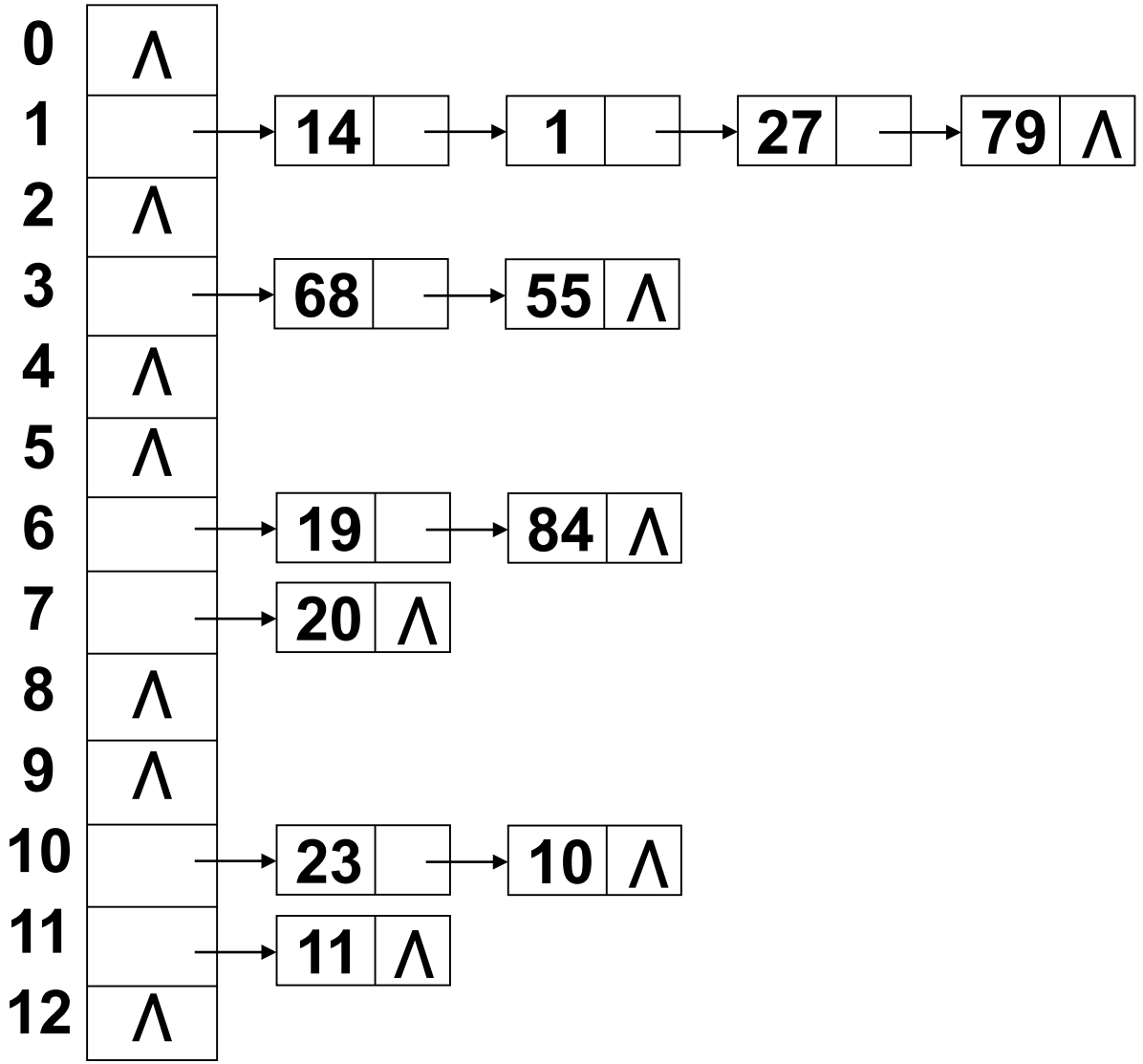
RecNode *linkhash[m]，其中RecNode是结点类型，每个分量的初值为空。凡散列地址为k的记录都插入到以linkhash[k]为头指针的链表中，插入位置可以在表头或表尾或按关键字排序插入。

# 3 链地址法

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(key)=key MOD 13, 用链地址法处理冲突, 如下图所示 。

优点: 不易产生冲突的“聚集”;

删除记录也很简单。




用链地址法处理冲突的散列表

# 4　建立公共溢出区

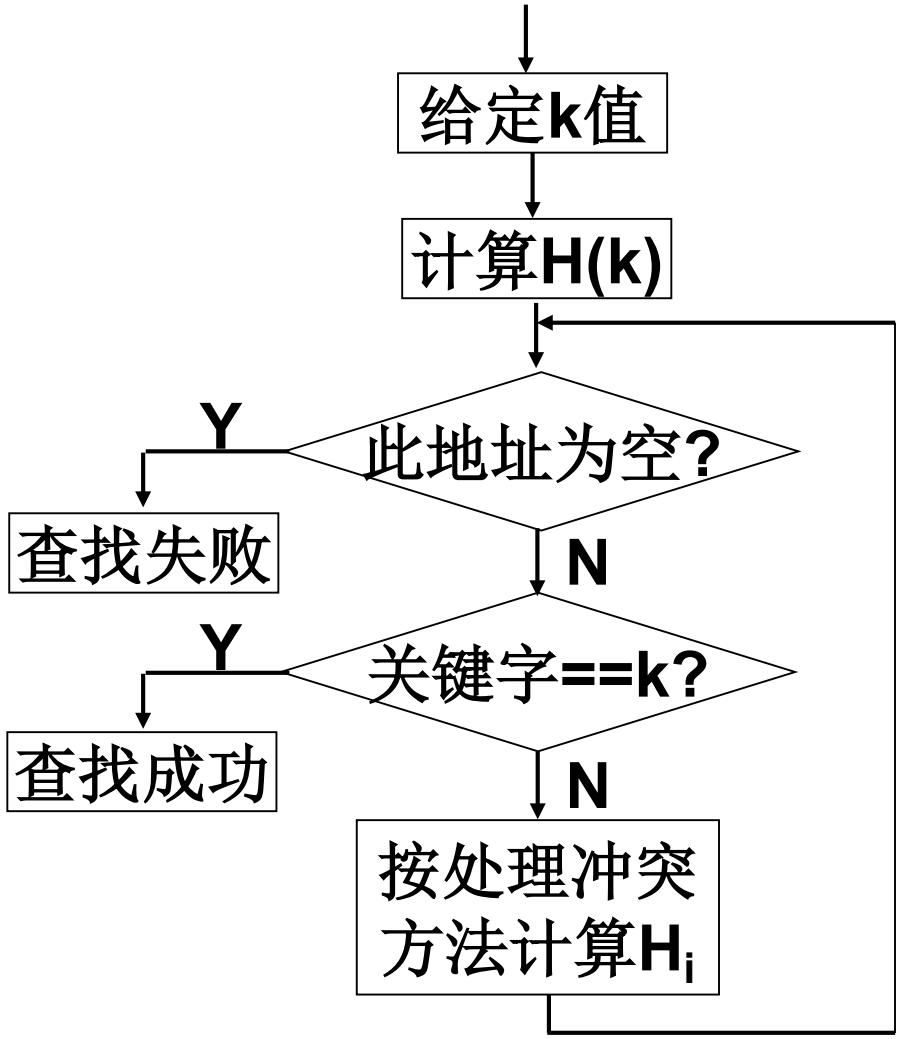
**方法**：在基本散列表之外，另外设立一个溢出表保存与基本表中记录冲突的所有记录。

设散列表长为m，设立基本散列表hashtable[m]，每个分量保存一个记录；溢出表overtable[v]，一旦某个记录的散列地址发生冲突，都填入溢出表中。

# 9.3.4 哈希查找

## 基本思想

给定K值，根据造表时设定的哈希函数求得哈希地址，若表中此位置上没有记录，则查找不成功；否则比较关键字，若和给定值相等，则查找成功；否则根据造表时设定的处理冲突的方法找“下一地址”，直至哈希表中某个位置为“空”或者表中所填记录的关键字等于给定值时为止。




散列表的查找过程

# 哈希查找的特点：

(1)优点：插入查找的速度快；

(2)缺点：

- 通过哈希函数计算哈希地址时，占用一定的计算时间。
- 占用的存储空间多。为减少冲突的发生，哈希表的长度应大于记录的长度。
- 在哈希表中只能按关键字进行查找。

# 小结　需要复习的知识点

- 掌握　简单的查找结构：
  - 查找的概念
    - 查找结构
    - 查找的判定树
    - 平均查找长度
  - 静态查找
    - 顺序查找算法、分析
    - 折半查找算法、分析

- 二叉查找树
  - 定义
  - 查找、平均查找长度
  - 插入、删除、
- *AVL*树
  - 定义
  - 插入、平衡化旋转
  - 删除、平衡化旋转
  - 高度

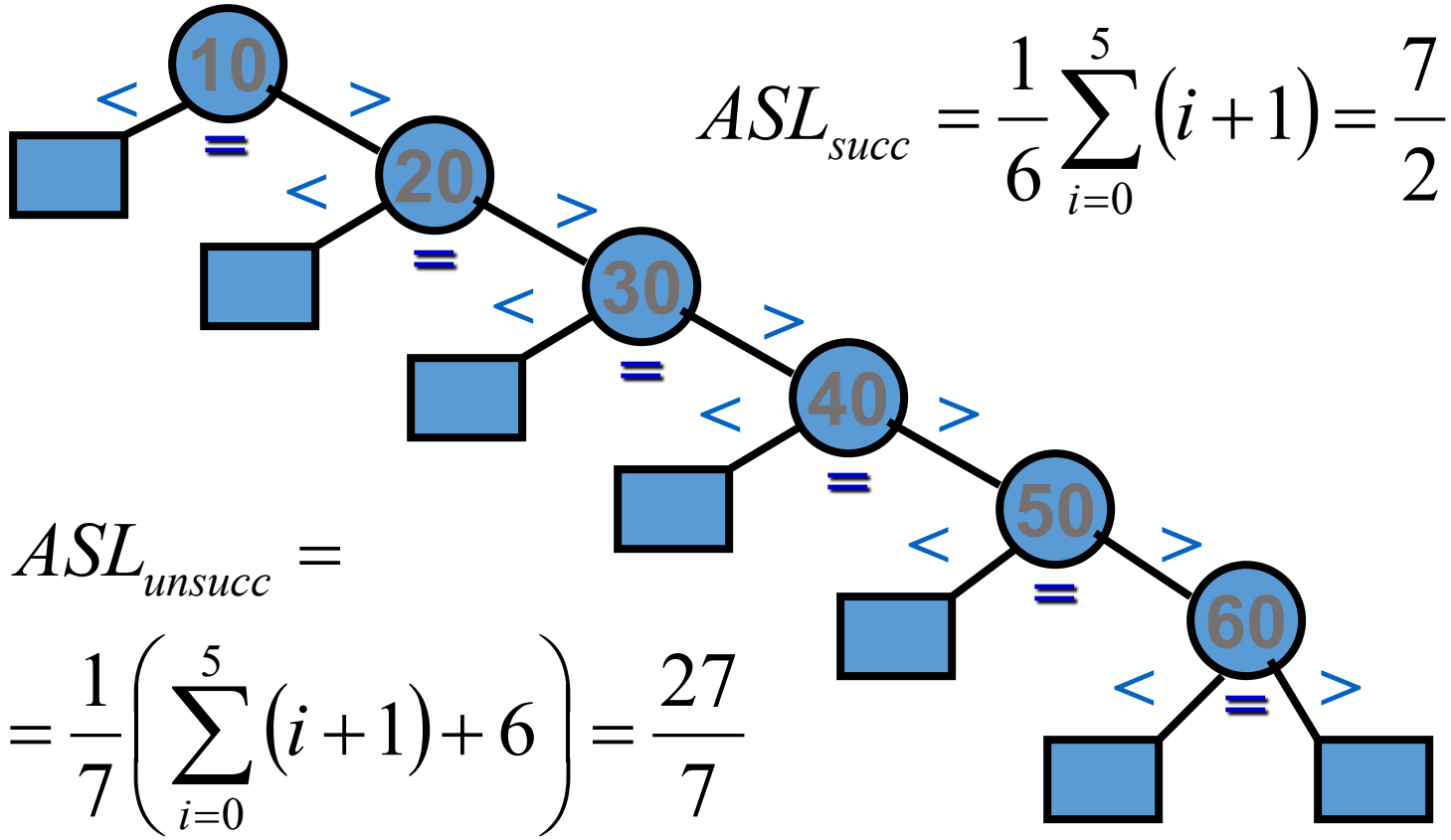
## 查找的概念

- 查找结构决定查找的效率
- 查找算法基于查找结构
- 查找效率用平均查找长度衡量
- 平均查找长度表明查找算法的整体性能，避开偶然因素
- 平均查找长度分查找成功与查找不成功两种情况

- 静态查找结构
  - 顺序查找 — 顺序表、链表
  - 折半查找 — 有序顺序表
- 动态查找结构
  - 二叉排序树 — 无重复关键字
  - *AVL*树 — 平衡二叉排序树

## ■有序顺序表的顺序查找

（ 10， 20， 30， 40， 50， 60 ）



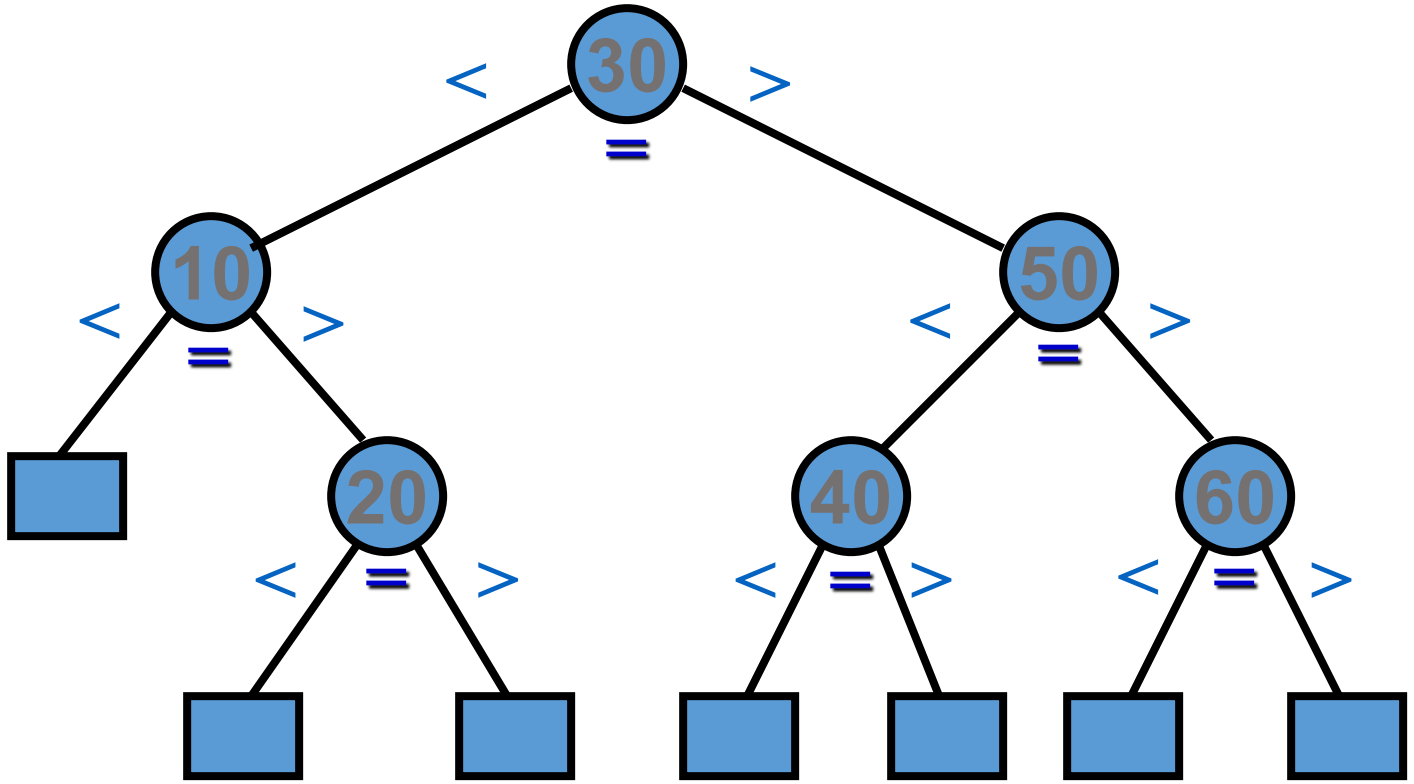

$$ASL_{succ} = \frac{1}{6}\sum_{i=0}^{5}(i+1) = \frac{7}{2}$$

$$ASL_{unsucc} =$$

$$= \frac{1}{7}\left(\sum_{i=0}^{5}(i+1)+6\right) = \frac{27}{7}$$

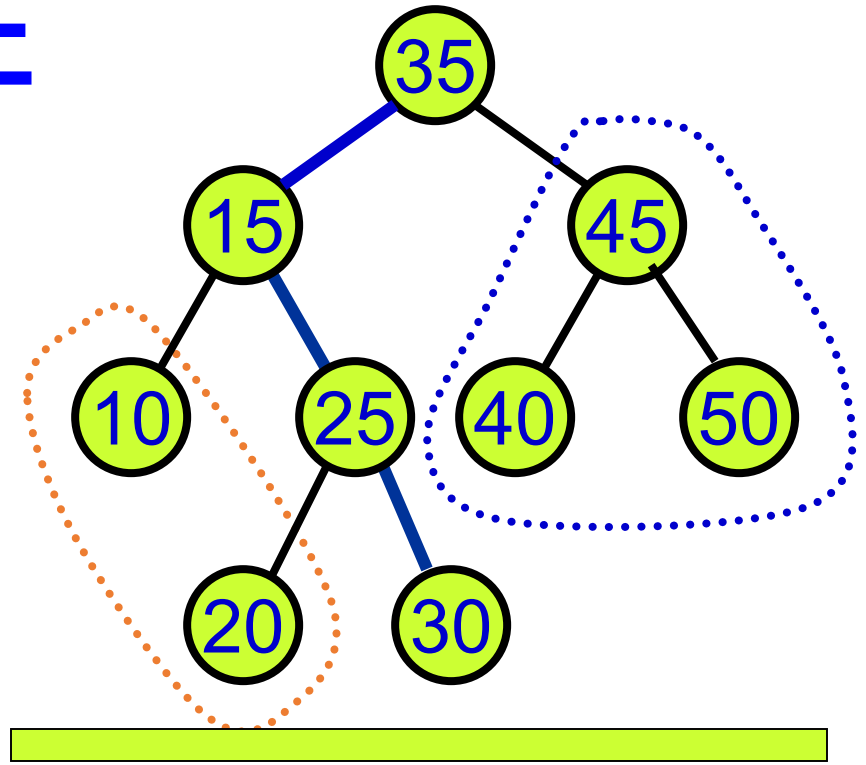
## ■有序顺序表的折半查找

（ 10， 20， 30， 40， 50， 60 ）



# 二叉查找树

- 二叉查找树的子树是二叉查找树
  - **结点左子树上所有关键字小于结点关键字**
  - **右子树上所有关键字大于结点关键字**

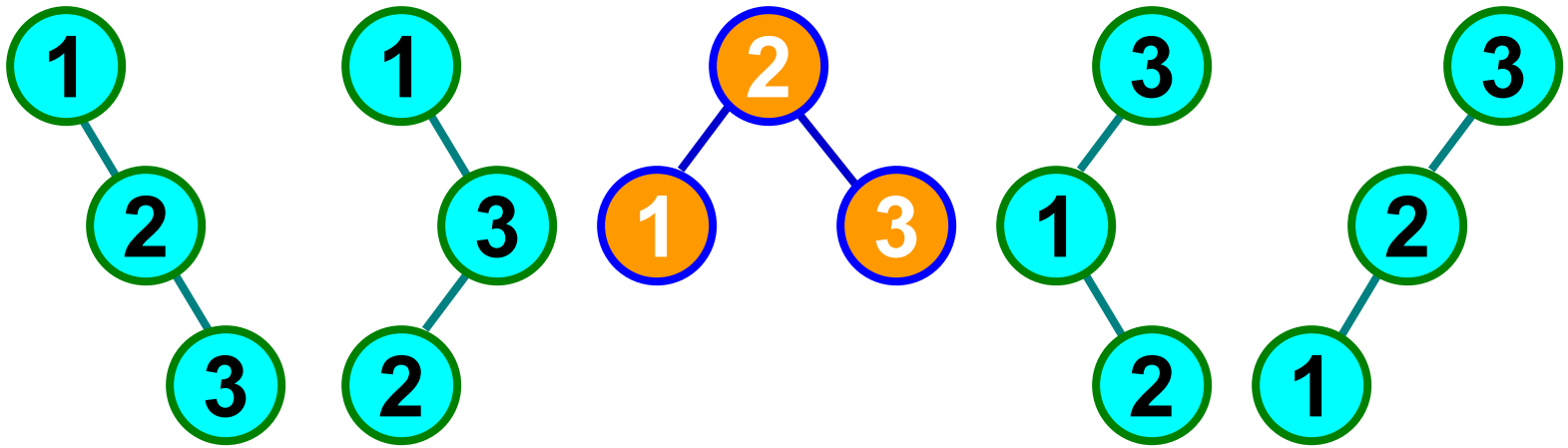


# $n$ 个结点的二叉查找树的数目

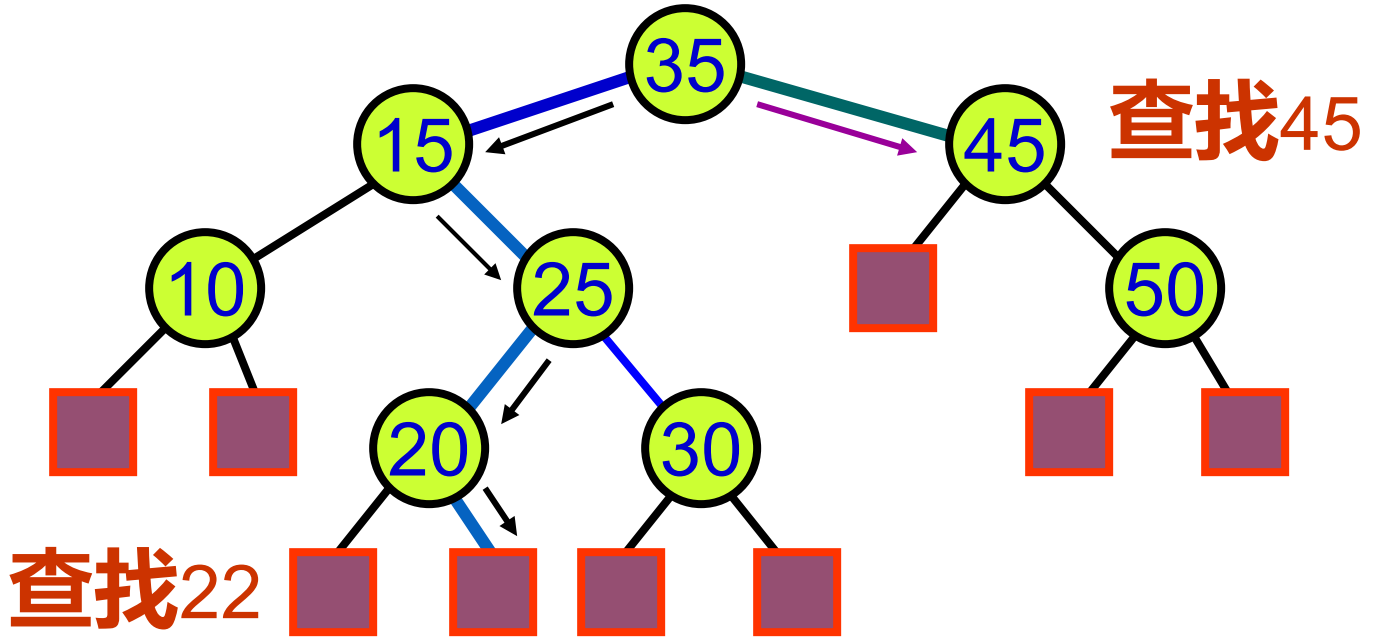
## 【例】3 个结点的二叉查找树

$$\frac{1}{3+1}C_{2*3}^{3}=\frac{1}{4}*\frac{6*5*4}{3*2*1}=5$$

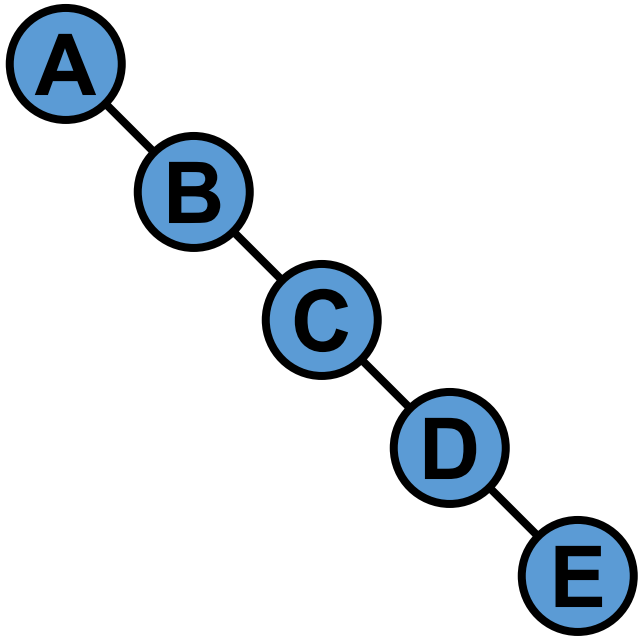
{123} {132} {213} {231} {312} {321}

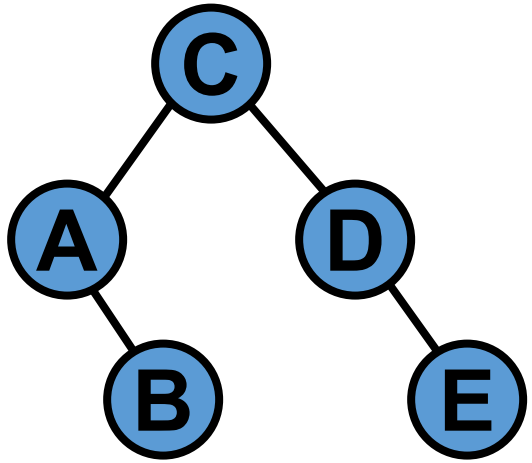


- 查找成功时检测指针停留在树中某个结点。
- 查找不成功时检测指针停留在某个外结点（失败结点）。



- 二叉查找树的高度越小，平均查找长度越小。
- $n$ 个结点的二叉查找树的高度最大为 $n-1$，最小为 $\lfloor \log_2 n \rfloor$.



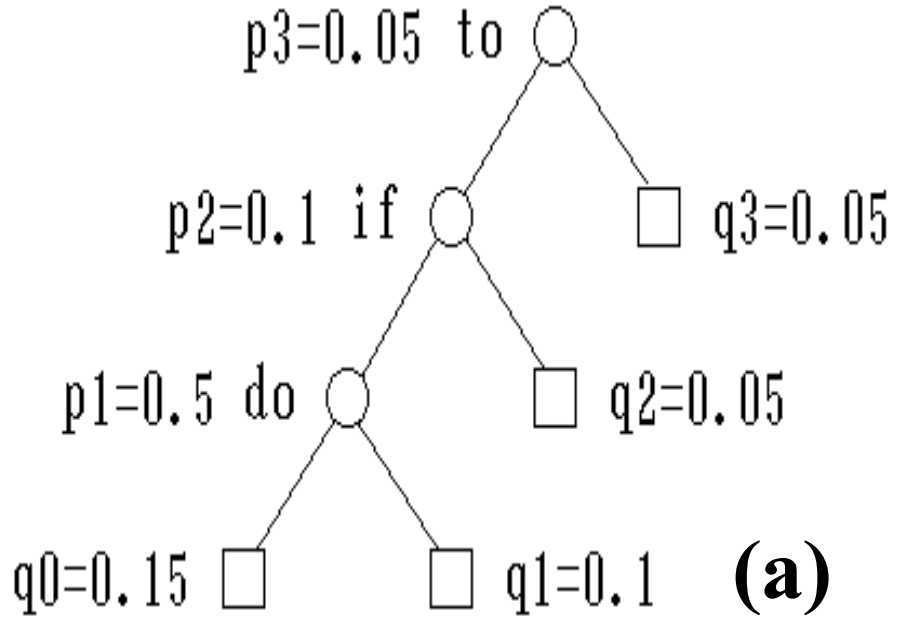



- *AVL*树
  - 理解：*AVL*树的子树也是*AVL*树
  - 掌握：插入新结点后平衡化旋转的方法
  - 掌握：删除结点后平衡化旋转的方法
  - 掌握：结点高度 $h$ 与结点数 $n$ 的关系

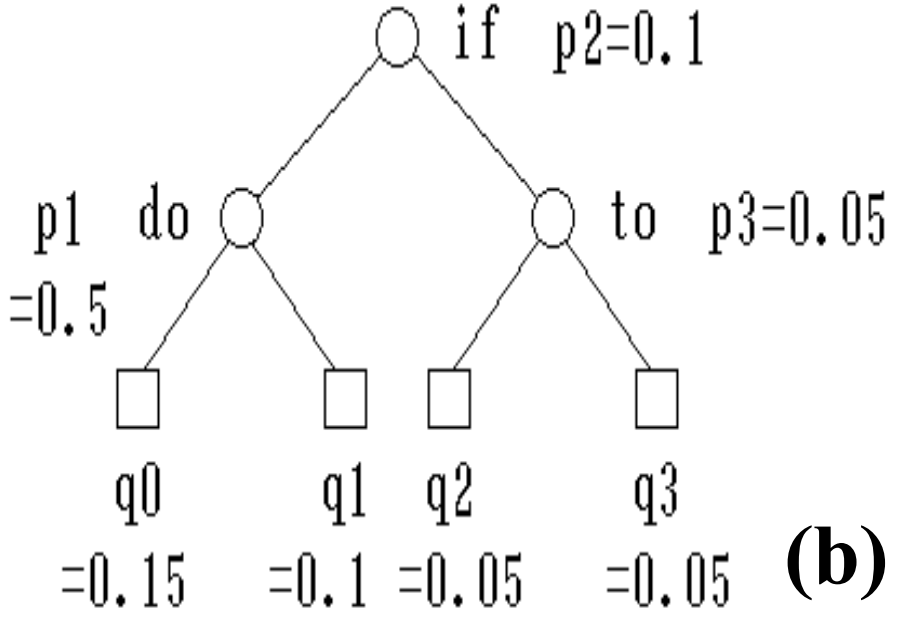
# 最优二叉查找树

- 对于有 $n$ 个关键字的集合，其关键字有 $n!$ 种不同的排列，可构成的不同二叉查找树有 $\frac{1}{n+1}C_{2n}^{n}$（棵）

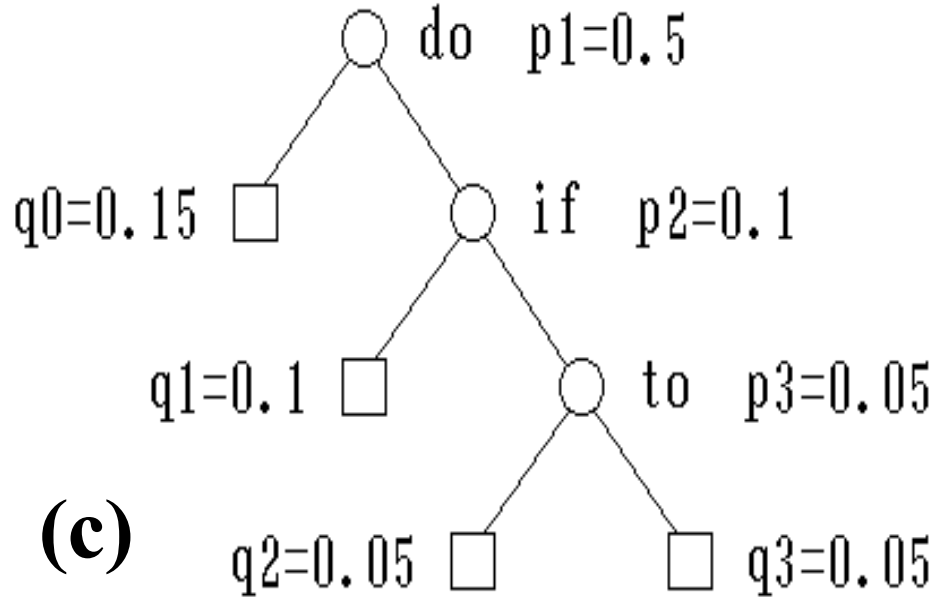
- 如何评价这些二叉查找树，可以用树的查找效率来衡量。
- 例如，已知关键字集合 { a1, a2, a3 } = { do, if, to }，对应查找概率为 p1=0.5，p2=0.1，p3=0.05，在各个查找不成功的间隔内查找概率又分别为 q0=0.15，q1=0.1，q2=0.05，q3=0.05。可能的二叉查找树如下所示。


p3=0.05 to
p2=0.1 if
q3=0.05
p1=0.5 do
q2=0.05
q0=0.15
q1=0.1

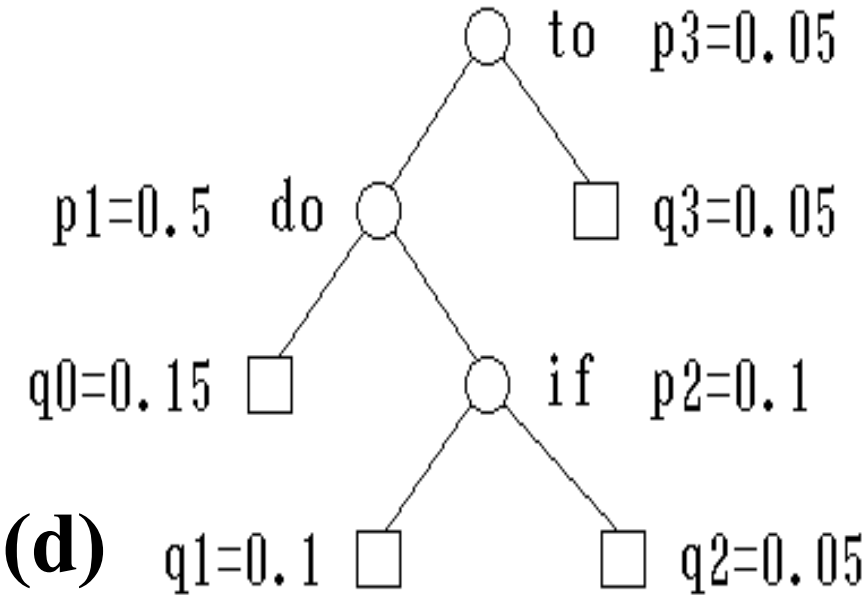
(a)


if p2=0.1
p1 do
=0.5
to p3=0.05
q0
q1
q2
q3
=0.15
=0.1
=0.05
=0.05

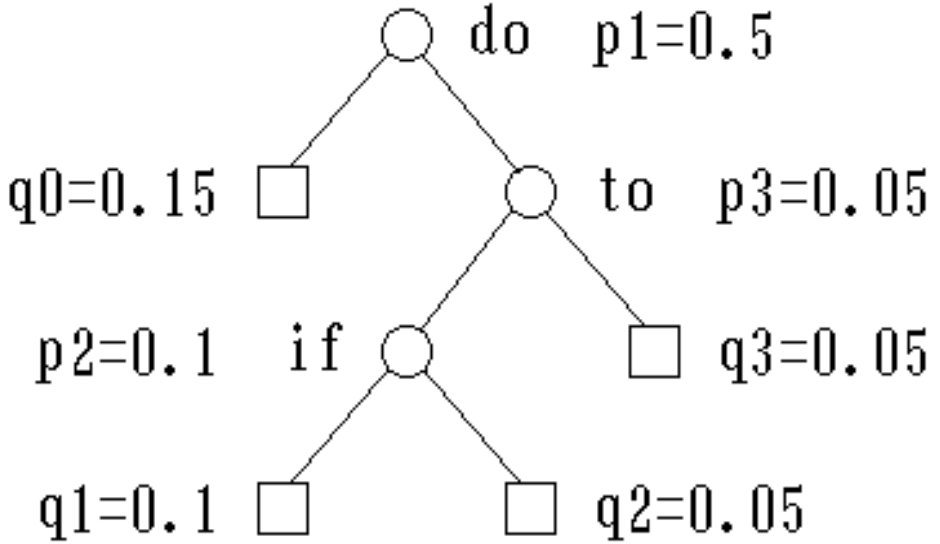
(b)


do p1=0.5
q0=0.15
if p2=0.1
q1=0.1
to p3=0.05
q2=0.05
q3=0.05

(c)


to p3=0.05
p1=0.5 do
q3=0.05
q0=0.15
if p2=0.1
q1=0.1
q2=0.05

(d)




**(e)**

- 在扩充二叉查找树中
  - ○表示内部结点，包含了关键字集合中的某一个关键字；
  - □表示外部结点，代表造成查找失败的各关键字间隔中的不在关键字集合中的关键字。
- 在每两个外部结点之间必然存在一个内部结点。

设二叉树的内部结点有 $n$ 个，外部结点有 $n+1$ 个。如果定义每个结点的路径长度为该结点的层次，并用 $I$ 表示所有 $n$ 个内部结点的路径长度之和，用 $E$ 表示 $n+1$ 个外部结点的路径长度之和，可以用归纳法证明，$E = I+2n$。

一棵扩充二叉查找树的查找成功的平均查找长度 $ASL_{succ}$ 可以定义为该树所有内部结点上的权值p[$i$]与查找该结点时所需的关键字比较次数$c[i]$ ($= l[i] + 1$) 乘积之和：

$$ASL_{succ} = \sum_{i=1}^{n} \mathbf{p}[i]*(l[i]+1).$$

扩充二叉查找树查找不成功的平均查找长度$ASL_{unsucc}$为树中所有外部结点上权值q[j]与到达外部结点所需关键字比较次数c'[j](= l'[j])乘积之和:

$$ASL_{unsucc} = \sum_{j=0}^{n} q[j] * l[j].$$

扩充二叉查找树总的平均查找长度为:

$$ASL = ASL_{succ} + ASL_{unsucc}$$

所有内、外部结点上的权值关系为

$$\sum_{i=1}^{n} p[i] + \sum_{j=0}^{n} q[j] = 1$$

# (1) 相等查找概率的情形

若设树中所有内、外部结点的查找概率都相等：

$p[i] = q[j] = 1/7,\ 1 \le i \le 3,\quad 0 \le j \le 3$

图(a)：$ASL_{succ} = 1/7*3+1/7*2+1/7*1 = 6/7,$

$ASL_{unsucc} = 1/7*3*2+1/7*2+1/7*1 = 9/7$。

总平均查找长度 $ASL = 6/7 + 9/7 = 15/7$。

图(a)： ASL = 15/7图(d)：ASL = 15/7

图(b)：ASL = 13/7 图(e)：ASL = 15/7

图(c)：ASL = 15/7

图(b)的情形所得的平均查找长度最小。一般把平均查找长度达到最小的扩充二叉查找树称作最优二叉查找树，

在相等查找概率的情形下，因为所有内部结点、外部结点的查找概率都相等，把它们的权值都视为 1。同时，第 $k$ 层有 $2^k$ 个结点，$k = 0, 1, \ldots$。则有 $n$ 个内部结点的扩充二叉查找树的内部路径长度 $I$ 至少等于序列

0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, …

的前$n$项的和。

因此，最优二叉查找树的查找成功的平均查找长度和查找不成功的平均查找长度分别为:

$$ASL_{succ} = \sum_{i=1}^{n} (\lfloor \log_2 i \rfloor + 1).$$

$$ASL_{unsucc} = \sum_{i=n+1}^{2n+1} \lfloor \log_2 i \rfloor.$$

## (2) 不相等查找概率的情形

- 设树中所有内、外部结点的查找概率互不相等，p[1] = 0.5，p[2] = 0.1，p[3] = 0.05

  q[0] = 0.15，q[1] = 0.1，q[2] = 0.05，q[3] = 0.05

  图(a)：$ASL_{succ}$ = 0.5*3 + 0.1*2 + 0.05*1 = 1.75，

  $ASL_{unsucc}$ = 0.15*3 + 0.1*3 + 0.05*2 + 0.05*1= 0.9。

  ASL = $ASL_{succ}$ + $ASL_{unsucc}$ = 2.65。

图(a)：ASL = 2.65　　图(d)：ASL = 2.15

图(b)：ASL = 1.9 图(e)：ASL = 1.6

图(c)：ASL = 0.85

- 由此可知，图(c)的情形下树的平均查找长度达到最小，因此，图(c)的情形是最优二叉查找树。

不相等查找概率情形下的最优二叉查找树可能不同于完全二叉树的形态。

考虑在不相等查找概率情形下如何构造最优二叉查找树。

假设关键字集合放在一个有序的顺序表中

$\{ key[1], key[2], key[3], \cdots key[n] \}$

设最优二叉查找树为T[0][$n$]，其平均查找长度为：

$$ASL = \sum_{i=1}^{n} p[i]*(l[i]+1) + \sum_{j=0}^{n} q[j]*l[j] = C[0][n]$$

l [i]是内部结点 a[i] 所在的层次号，l [j] 是外部结点 j 所在的层次号。C[0][n] 是树的代价。

为计算方便，将 p[i] 与 q[j] 化为整数。

# 构造最优二叉查找树

- 要构造最优二叉查找树，必须先构造它的左子树和右子树，它们也是最优二叉查找树。

对于任一棵子树T[$i$][$j$]，它由

{ $key[i+1]$, $key[i+2]$,…, $key[j]$ }

组成，其内部结点的权值为

{ $p[i+1]$, $p[i+2]$,…, $p[j]$ }

外部结点的权值为

{ $q[i]$, $q[i+1]$, $q[i+2]$,…, $q[j]$ }

使用数组$W[i][j]$来存放它的累计权值和：

$$W[i][j] = q[i] + p[i+1] + q[i+1] + p[i+2] + q[i+2] + \cdots + p[j] + q[j].$$

$$0 \le i \le j \le n$$

计算W[i][j]可以用递推公式:

W[i][i] = q[i]

//不是二叉树，只有一个外部结点

W[i][i+1] = q[i] + p[i+1] + q[i+1]

= W[i][i] + p[i+1] + q[i+1]

//有一个内部结点及两个外部结点的二叉树

W[i][i+2] = W[i][i+1] + p[i+2] + q[i+2]

//加一个内部结点和一个外部结点的二叉树

一般地，

W[i][j] = W[i][j-1] + p[j] + q[j]

//再加一个内部结点和一个外部结点

对于一棵包括关键字

{ key[i+1], …, key[k-1], key[k], key[k+1], …, key[j] }

的子树T[i][j]，若设它的根结点为k，i < k ≤ j，

它的代价为：

C[i][j] = p[k]+C[i][k-1]+W[i][k-1]+C[k][j]+W[k][j]

- C[i][k-1]是其包含关键字 { key[i+1], key[i+2],…, key[k-1] } 的左子树T[i][k-1]的代价
- C[i][k-1]+W[i][k-1]等于把左子树每个结点的路径长度加1而计算出的代价
- C[k][j]是包含关键字 { key[k+1], key[k+2],…, key[j] } 的右子树T[k][j]的代价
- C[k][j] + W[k][j]是把右子树每个结点的路径长度加1之后计算出的代价。

因此，公式

$$C[i][j] = p[k]+C[i][k-1]+W[i][k-1]+C[k][j]+W[k][j]$$

表明：整个树的代价等于其左子树的代价加上其右子树的代价，再加上根的权值。

因为

$$W[i][j] = p[k] + W[i][k-1] + W[k][j],$$

故有

$$C[i][j] = W[i][j] + C[i][k-1] + C[k][j]$$

可用 $k = i+1, i+2, \cdots, j$ 分别计算上式，选取使得$C[i][j]$达到最小的 $k$。这样可将最优二叉查找树T[i][j]构造出来。

构造最优二叉查找树的方法就是自底向上逐步构造的方法。

# 构造的步骤

- 第一步，构造只有一个内部结点的最优查找树：T[0][1]，T[1][2]，…，T[$n$-1][$n$]。

  在T[$i$-1][$i$] ($1 \le i \le n$) 中只包含关键字 $key[i]$。其代价分别是 $C[i-1][i] = W[i-1][i]$。另外设立一个数组 $R[0..n][0..n]$ 存放各最优二叉查找树的根。$R[0][1] = 1$，$R[1][2] = 2$，…，$R[n-1][n] = n$。

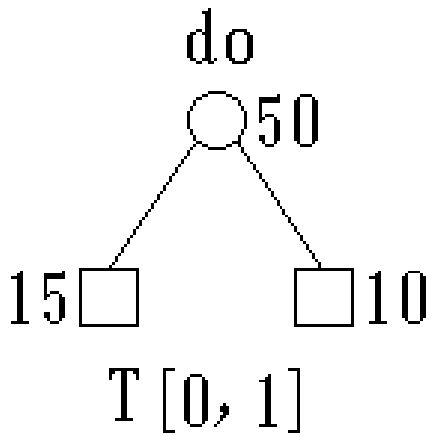
- 第二步，构造有两个内部结点的最优二叉查找树：T[0][2]，T[1][3]，…，T[$n$-2][$n$]。在T[$i$-2][$i$] 中包含两个关键字 $\{ key[i-1], key[i] \}$。其代价取分别以 $key[i-1]$，$key[i]$ 做根时计算出的 $C[i-2][i]$ 中的小者。

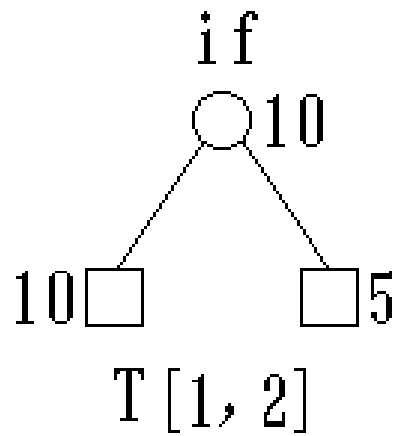
- 第三步，第四步，…，构造有 3 个内部结点，有 4 个内部结点，… 的最优二叉查找树。
- 最后构造出包含有 $n$ 个内部结点的最优二叉查找树。对于这样的最优二叉查找树，若设根为 $k$，则根 $k$ 的值存于 $R[0][n]$ 中，其代价存于 $C[0][n]$ 中，左子树的根存于 $R[0][k-1]$ 中，右子树的根存于 $R[k][n]$ 中。

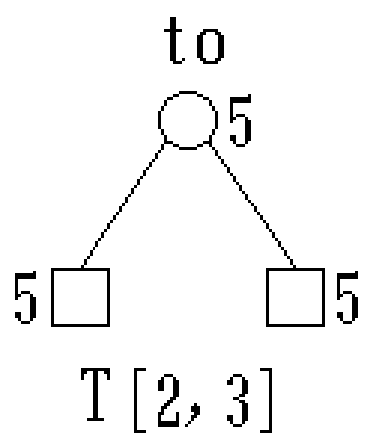
# 例：给出关键字集合和内、外部结点的权值序列

| 关键字集合 | | key1 | key2 | key3 |
|---|---|---|---|---|
| 实　例 | | do | if | to |
| 对应 | 内部结点 | p1=50 | p2=10 | **p3=5** |
| 权值 | 外部结点 | q0=15 | q1=10 | q2=5 | **q3=5** |

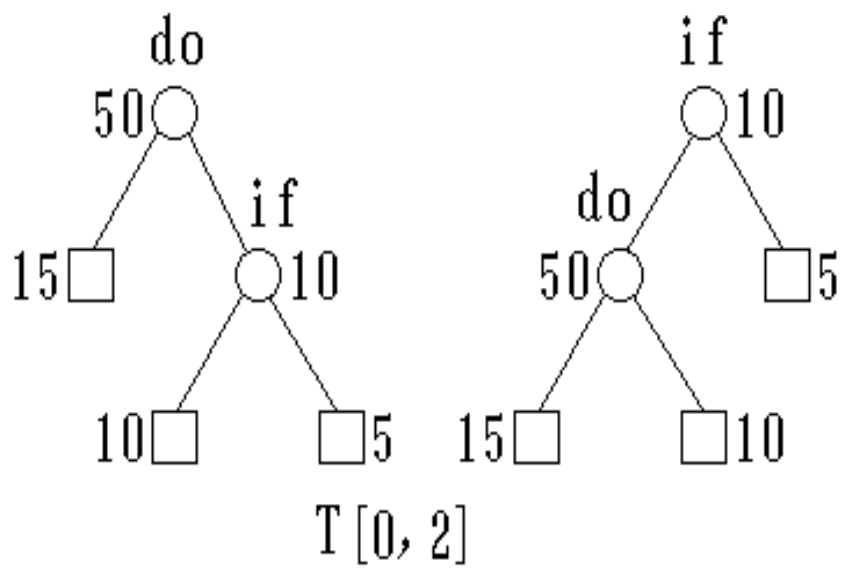
## 第一步



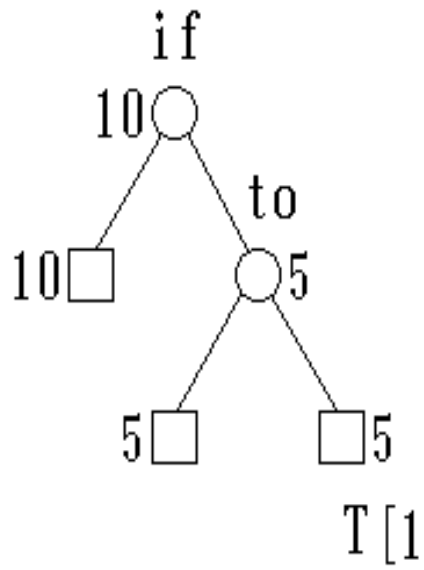



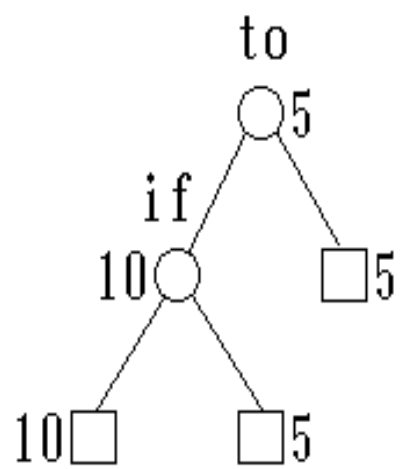
# 第二步





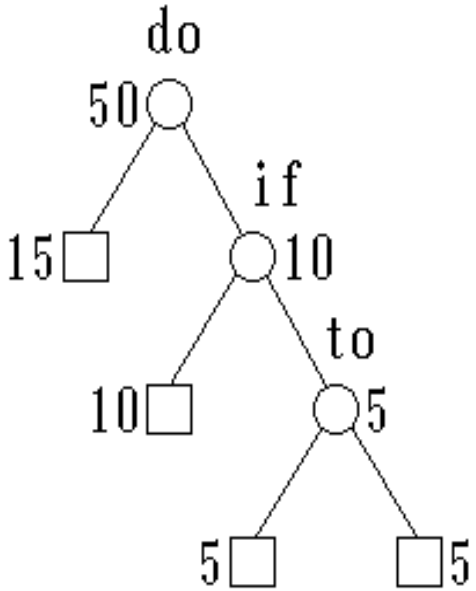




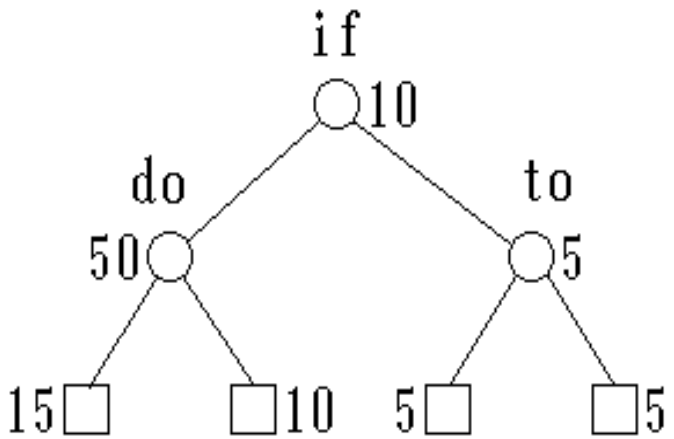




| | | | | |
|---|---|---|---|---|
| *C* | 115 | 165 | **50** | 60 |
| *W* | 90 | 90 | **35** | 35 |
| *R* | 1 | 2 | **2** | 3 |
| 左子树 | T[0, 0] | T[0, 1] | **T[1, 1]** | T[1, 2] |
| 右子树 | T[1, 2] | T[2, 2] | **T[2, 3]** | T[3, 3] |

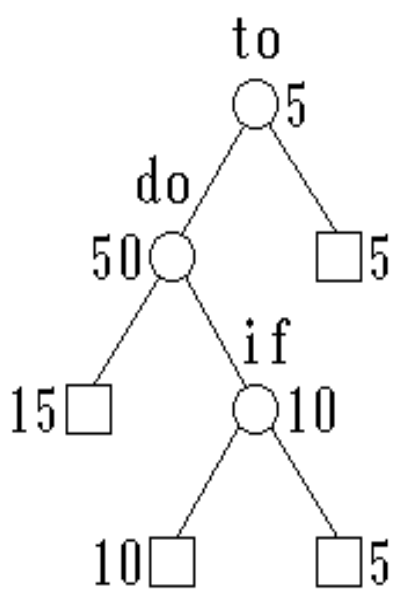
# 第三步




T [0, 3]




T [0, 3]




T [0, 3]

| | | | |
|---|---|---|---|
| ***C*** | **150** | **190** | **215** |
| ***W*** | **100** | **100** | **100** |
| ***R*** | **1** | **2** | **3** |
| **左子树** | **T[0,0]** | **T[0,1]** | **T[0,2]** |
| **右子树** | **T[1,3]** | **T[2,3]** | **T[3,3]** |

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 15 | 75 | 90 | 100 |
| 1 | | 10 | 25 | 35 |
| 2 | | | 5 | 15 |
| 3 | | | | 5 |

*W*[*i*][*j*]

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 75 | 115 | 150 |
| 1 | | 0 | 25 | 50 |
| 2 | | | 0 | 15 |
| 3 | | | | 0 |

*C*[*i*][*j*]

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 |
| 1 | | 0 | 2 | 2 |
| 2 | | | 0 | 3 |
| 3 | | | | 0 |

*R*[*i*][*j*]

**3个关键字 { do, if, to } 的最优二叉查找树**

p1=50, p2=10, p3=5

q0=15, q1=10, q2= 5, q3= 5