第九章 查找

- 查找的概念
- 静态查找表
- 动态查找表
- 哈希表

查找的概念

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数据结构。

对查找表的操作:

- 查询某个“特定的”数据元素是否在查找表中;
- 检索某个“特定的”数据元素的各种属性;
- 在查找表中插入一个数据元素;
- 从查找表中删去某个数据元素

查找表的分类：

静态查找表

仅作查询和检索操作的查找表。

动态查找表

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,此类表为动态查找表。

关键字

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

查找

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数据元素或（记录）。

若查找表中存在这样一个记录，则称“查找成功”，查找结果：给出整个记录的信息，或指示该记录在查找表中的位置；否则称“查找不成功”，查找结果：给出“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。

查找方法评价

- 查找速度
- 占用存储空间多少
- 算法本身复杂程度
- 平均查找长度ASL(Average Search Length)：为确定记录在表中的位置，需和给定值进行比较的关键字的个数的期望值叫查找算法的~

对含有n个记录的表，$ASL=\sum_{i=1}^{n} p_i c_i$

其中：p_i为查找表中第i个元素的概率，$\sum_{i=1}^{n} p_i = 1$

c_i为找到表中第i个元素所需比较次数

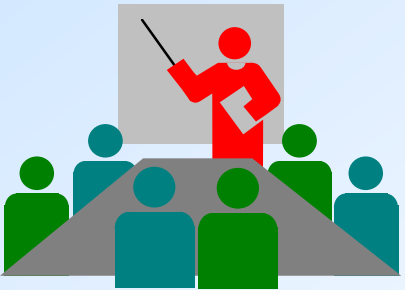
静 态 查 找 表

抽象数据类型静态查找表的定义:

ADT StaticSearchTable {

数据对象D: **D是具有相同特性的数据元素的集合。每个数据元素含有类型相同的关键字，可唯一标识数据元素。**

数据关系R：数据元素同属一个集合。



基本操作 P：

Create(&ST, n); //构造一个含 n 个数据元素的静态查找表ST。

Destroy(&ST); //销毁表ST。

Search(ST, key); //查找 ST 中其关键字等于kval 的数据元素。

Traverse(ST, Visit()); //按某种次序对ST的每个元素调用函数Visit()一次且仅一次，

} ADT StaticSearchTable

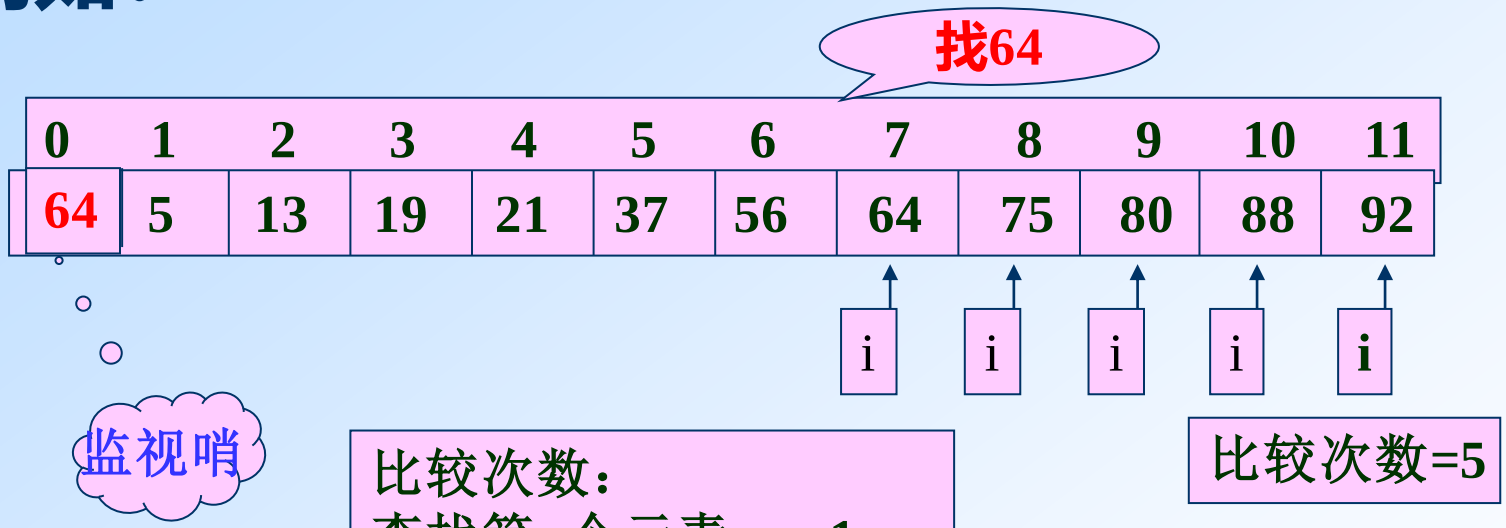
▪顺序表的查找

以顺序表表示静态查找表，则Search函数可用顺序查找来实现。其顺序存储结构如下：

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

查找过程：从表的一端开始逐个进行记录的关键字和给定值的比较。

例如：



比较次数：

查找第n个元素： 1

查找第n-1个元素： 2

……

查找第1个元素： n

查找第i个元素： n+1-i

查找失败： n+1

算法描述：

```
int Search_Seq(SSTable ST,
                                KeyType kval) {
   // 在顺序表ST中顺序查找其关键字等于
   //  key的数据元素。若找到，则函数值为
   // 该元素在表中的位置，否则为0。
   ST.elem[0].key = kval;       // 设置“哨兵”
   for (i=ST.length; ST.elem[i].key!=kval;  --i);
                                // 从后往前找
   return i;                    // 找不到时，i为0
} // Search_Seq
```

顺序查找性能分析

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

C_i为找到该记录时，曾和给定值比较过的关键字的个数

对顺序表而言，$C_i = n-i+1$

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

在等概率查找的情况下，$P_i = \frac{1}{n}$

顺序表查找的平均查找长度为：

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

在不等概率查找的情况下，ASL_{ss} 在

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

时取极小值。表中记录按查找概率由小到大重新排列,以提高查找效率。

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。

▪有序表的查找

顺序表的查找算法简单，但平均查找长度较大，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

查找过程： 每次将待查记录所在区间缩小一半。

适用条件： 采用顺序存储结构的有序表。

折半查找算法实现

1.设表长为n，low、high和mid分别指向待查元素所在区间的上界、下界和中点,k为给定值。

2.初始时，令low=1,high=n,mid=⌊(low+high)/2⌋

让k与mid指向的记录比较

若k==r[mid].key，查找成功

若k<r[mid].key，则high=mid-1

若k>r[mid].key，则low=mid+1

3.重复上述操作，直至low>high时，查找失败。

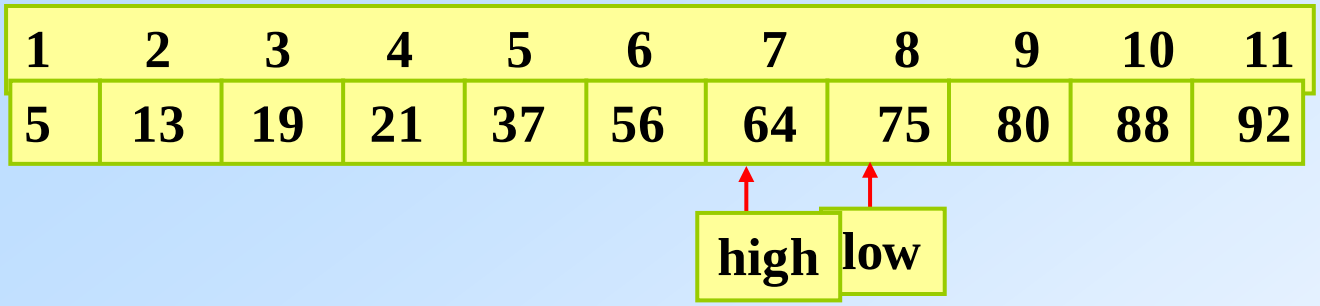
例如 key =21 的查找过程

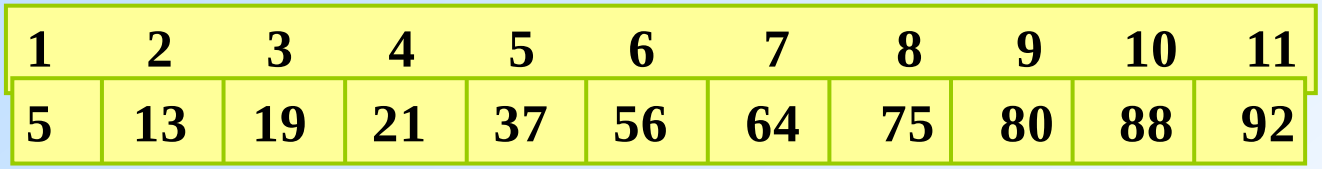
low 指示查找区间的下界；

high 指示查找区间的上界；

mid = (low+high)/2。

例key =70 的查找过程





当下界low大于上界high时，则说明表中没有关键字等于Key的元素，查找不成功。

折半查找算法

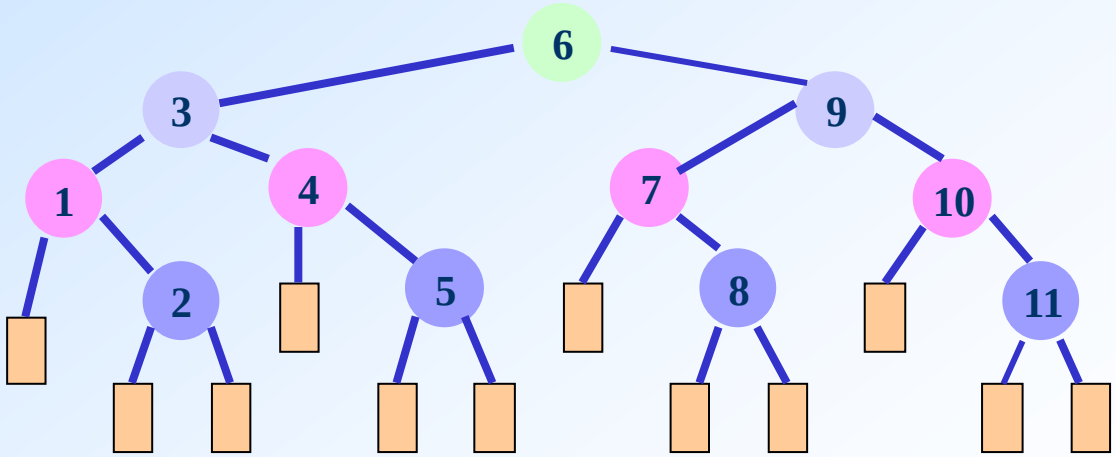
```
int Search_Bin ( SSTable ST, KeyType kval ) {
   low = 1;  high = ST.length;     // 置区间初值
   while (low <= high) {
      mid  = (low + high) / 2;
      if   (kval == ST.elem[mid].key   )
          return  mid;          // 找到待查元素
      else  if ( kval < ST.elem[mid].key)  )
        high = mid - 1;         // 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
```

折半查找的性能分析

- 判定树：描述查找过程的二叉树。
- 有n个结点的判定树的深度为$\lfloor \log_2 n \rfloor + 1$
- 折半查找法在成功查找过程中进行的比较次数最多不超过$\lfloor \log_2 n \rfloor + 1$

先看一个有11个元素的表的例子：n=11

i	1	2	3	4	5	6	7	8	9	10	11
Ci	3	4	2	3	4	1	3	4	2	3	4



假设 有序表的长度$n=2^h-1$（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，层次为2的结点有2个，层次为h的结点有2^{h-1}个。假设表中每个记录的查找概率相等

则查找成功时折半查找的平均查找长度

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

在 n>50 时，可得近似结果

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

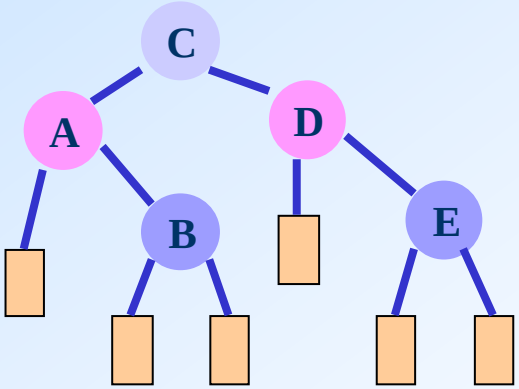
- 折半查找的效率比顺序查找高。
- 折半查找只能适用于有序表，并且以顺序存储结构存储。

顺序表和有序表的比较

	顺序表	有序表
表的特性	无序	有序
存储结构	顺序 或 链式	顺序
插删操作	易于进行	需移动元素
ASL的值	大	小

折半查找树

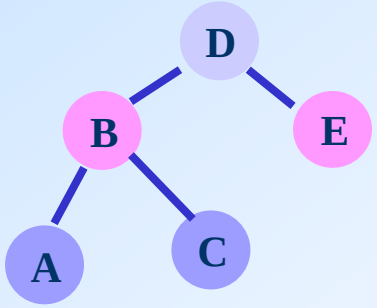
Key	A	B	C	D	E
W_j	0.1	0.2	0.1	0.4	0.2



$$ASL = \sum_{i=1}^{n} P_i C_i = 0.1*2+0.2*3+0.1*1+0.4*1+0.2*3=2.3$$

静态次优查找树

Key	A	B	C	D	E
W_j	0.1	0.2	0.1	0.4	0.2



$$ASL = \sum_{i=1}^{n} P_i C_i = 0.1*3+0.2*2+0.1*3+0.4*1+0.2*2=1.8$$

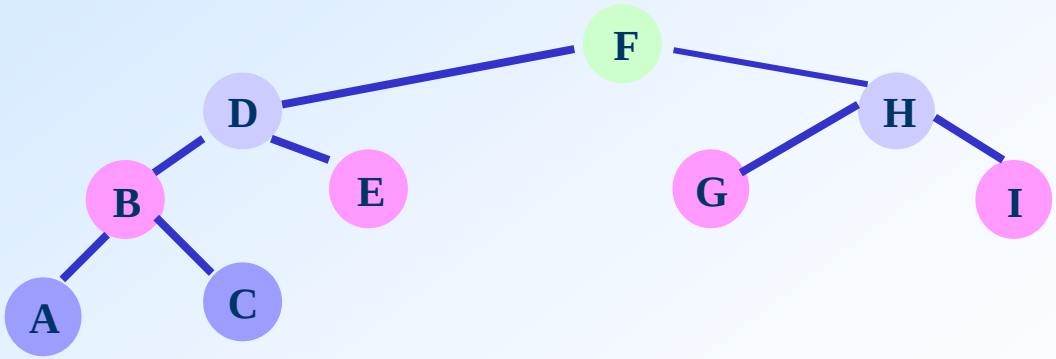
静态次优查找树

- J 0 1 2 3 4 5 6 7 8 9
- Key A B C D E F G H I
- W_j 0 1 1 2 5 3 4 4 3 5
- SW_j 0 1 2 4 9 12 16 20 23 28
- $\triangle P_j$ 27 25 22 15 7 0 8 15 23
- $\triangle P_j$ 11 9 6 1 9 8 1 7
- $\triangle P_j$ 3 1 2 0 0 0
- $\triangle P_j$ 0 0

$\{r_l, r_{l+1}, \ldots, r_{i-1}\}$ r_i $\{r_{i+1}, \ldots, r_h\}$

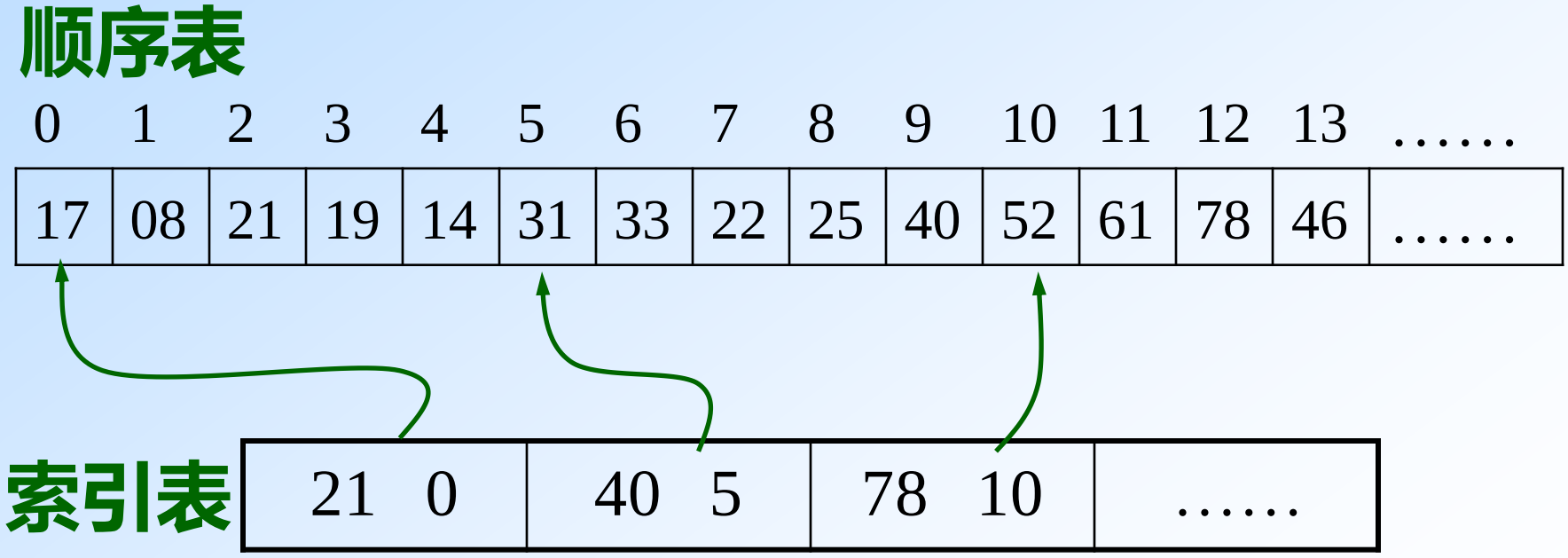
$$\triangle P_j = \left| (SW_h - SW_i) - (SW_{i-1} - SW_{l-1}) \right|$$

• Key		A	B	C	D	E	F	G	H	I
• W_j	0	1	1	2	5	3	4	4	3	5
• SW_j	0	1	2	4	9	12	16	20	23	28
• $\triangle P_j$		27	25	22	15	7	0	8	15	23
• $\triangle P_j$		11	9	6	1	9		8	1	7
• $\triangle P_j$		3	1	2		0		0		0
• $\triangle P_j$		0		0						



▪索引顺序表

在建立顺序表的同时，建立一个索引项，包括两项：关键字项和指针项。索引表按关键字有序，表则为分块有序



索引顺序表 = 索引 + 顺序表

索引顺序查找

又称分块查找

查找过程：将表分成几块，块内无序，块间有序；先确定待查记录所在块，再在块内查找

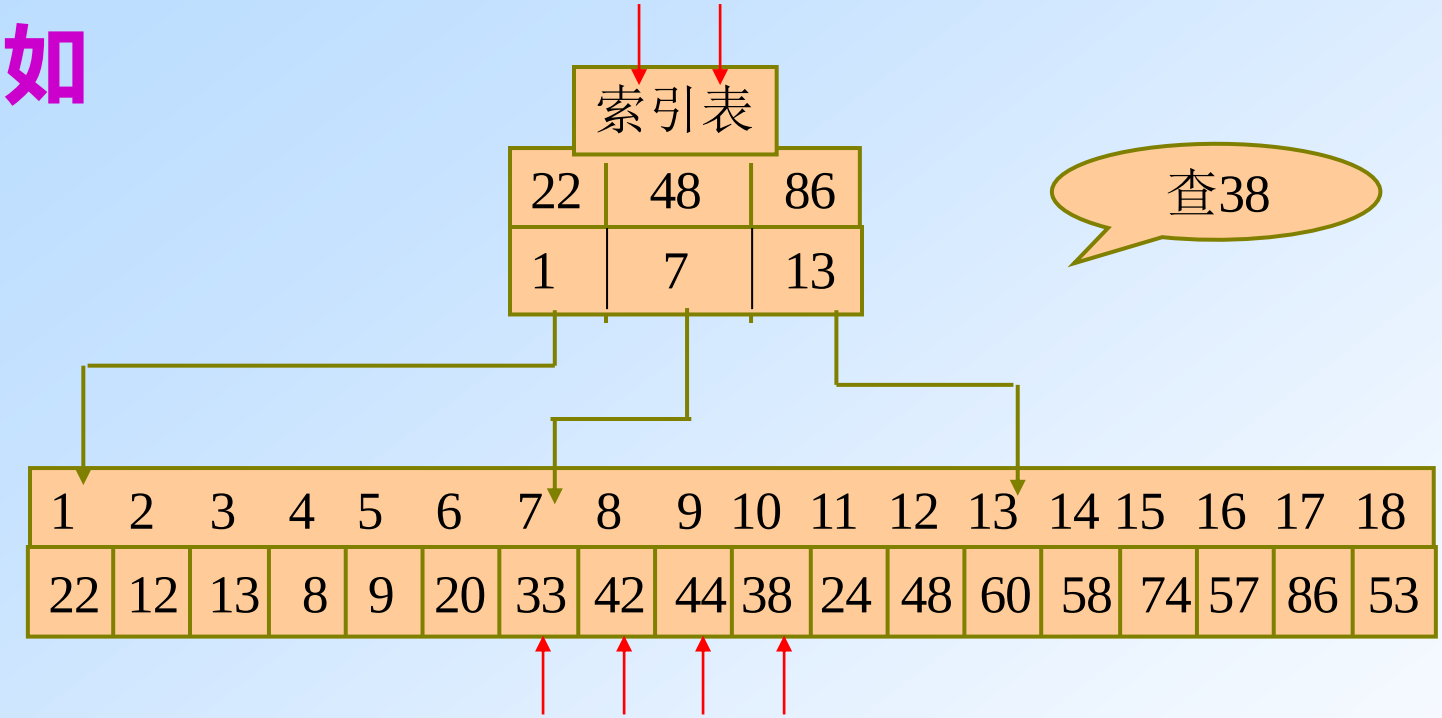
适用条件：分块有序表

算法实现：

用数组存放待查记录,每个数据元素至少含有关键字域

建立索引表，每个索引表结点含有最大关键字域和指向本块第一个结点的指针

例如


索引表
22
48
86
1
7
13
查38
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
22 12 13 8 9 20 33 42 44 38 24 48 60 58 74 57 86 53

分块查找方法评价

$$ASL_{bs} = L_b + L_w$$

其中：L_b ——查找索引表确定所在块的平均查找长度

L_w ——在块中查找元素的平均查找长度

若将表长为n的表平均分成b块，每块含s个记录，并设表中每个记录的查找概率相等，则：

(1)用顺序查找确定所在块：$ASL_{bs} = \frac{1}{b}\sum_{j=1}^{b} j + \frac{1}{s}\sum_{i=1}^{s} i$

$$= \frac{b+1}{2} + \frac{s+1}{2} = \frac{1}{2}\left(\frac{n}{s} + s\right) + 1$$

(2)用折半查找确定所在块：$ASL_{bs} \approx \log_2\left(\frac{n}{s} + 1\right) + \frac{s}{2}$

查找方法比较

	顺序查找	折半查找	分块查找
ASL	最大	最小	两者之间
表结构	有序表、无序表	有序表	分块有序表
存储结构	顺序存储结构 线性链表	顺序存储结构	顺序存储结构 线性链表

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)

结论：

- 从查找性能看，最好情况能达O(logn)，此时要求表有序；
- 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。

思考：已知一个长度为16的顺序表，其元素按关键字有序，若采用折半查找法查找一个不存在的元素，则比较次数最多是几？ 5

动态查找表

动态查找表的特点:表结构本身是在查找过程中动态生成。若表中存在其关键字等于给定值key的记录,表明查找成功；否则插入关键字等于key的记录。

抽象数据类型动态查找表的定义:

ADT DynamicSearchTable {

数据对象D: D是具有相同特性的数据元素的集合。

每个数据元素含有类型相同的关键字， 可唯一标识数据元素。

数据关系R: 数据元素同属一个集合。

基本操作：

InitDSTable(&DT)//构造一个空的动态查找表DT。

DestroyDSTable(&DT)//销毁动态查找表DT。

SearchDSTable(DT, key);//查找 DT 中与关键字key等值的元素。

InsertDSTable(&DT, e);//若 DT 中不存在其关键字等于 e.key 的 数据元素，则插入 e 到 DT。

DeleteDSTable(&T, key);//删除DT中关键字等于key的数据元素。

TraverseDSTable(DT, Visit());//按某种次序对DT的每个结点调用函数 Visit() 一次且至多一次。

}ADT DynamicSearchTable

二叉排序树（二叉查找树）

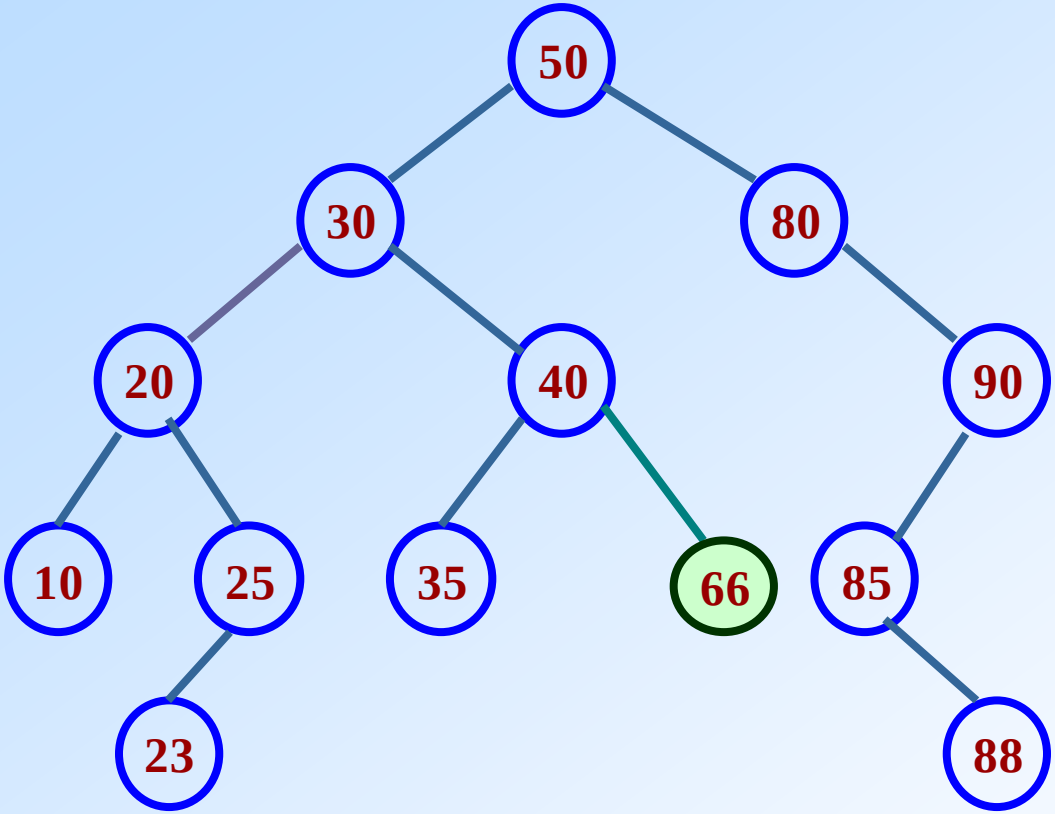
定义

二叉排序树或者是一棵空树；或者是具有如下特性的二叉树：

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；

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；

它的左、右子树也都分别是二叉排序树。



不是二叉排序树。

二叉排序树的存储结构

以二叉链表形式存储

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

typedef struct Node { // 结点结构
   ElemType      data;
   struct BiTNode  *lchild, *rchild; // 左右指针
 } BiTNode, *BiTree;
```

二叉排序树的查找算法

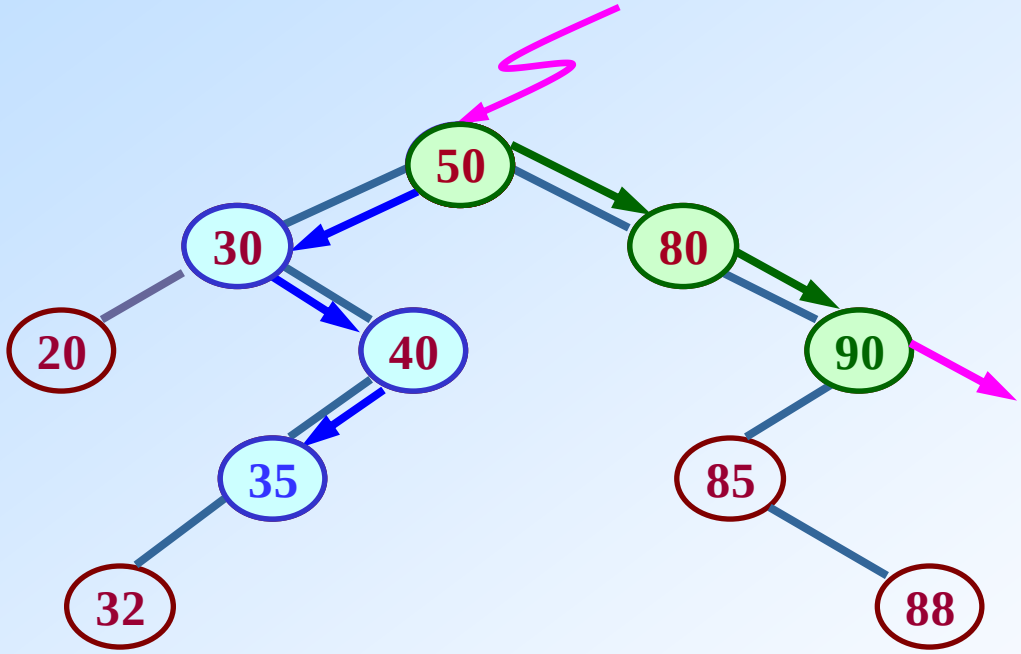
若二叉排序树为空，则查找不成功；
否则

1）若给定值等于根结点的关键字，则查找成功；

2）若给定值小于根结点的关键字，则继续在左子树上进行查找；

3）若给定值大于根结点的关键字，则继续在右子树上进行查找。

在二叉排序树中查找关键字值等于50,35,90,95



递归查找算法

```
BiTree SearchBST(BiTree T, KeyType xkey)
{   if(!T)|| EQ(xkey, T->data.key)  return(T);
    else if(LT(xkey, T->data.key))  return(SearchBST(T->lchild, xkey));
    else return(SearchBST(T->rchild, xkey));
}
```

非递归查找算法

```
BiTree SearchBST( BiTree T, KeyType xkey )
{   p=T;       // 查找成功，返回指向该结点的指针，否则返回NULL。
    while( p && !EQ( xkey, p->data.key ))
        if( LT( xkey, p->data.key ))  p = p->lchild;
        else  p = p->rchild;
    return( p );
}
```

二叉排序树的插入算法

- 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。

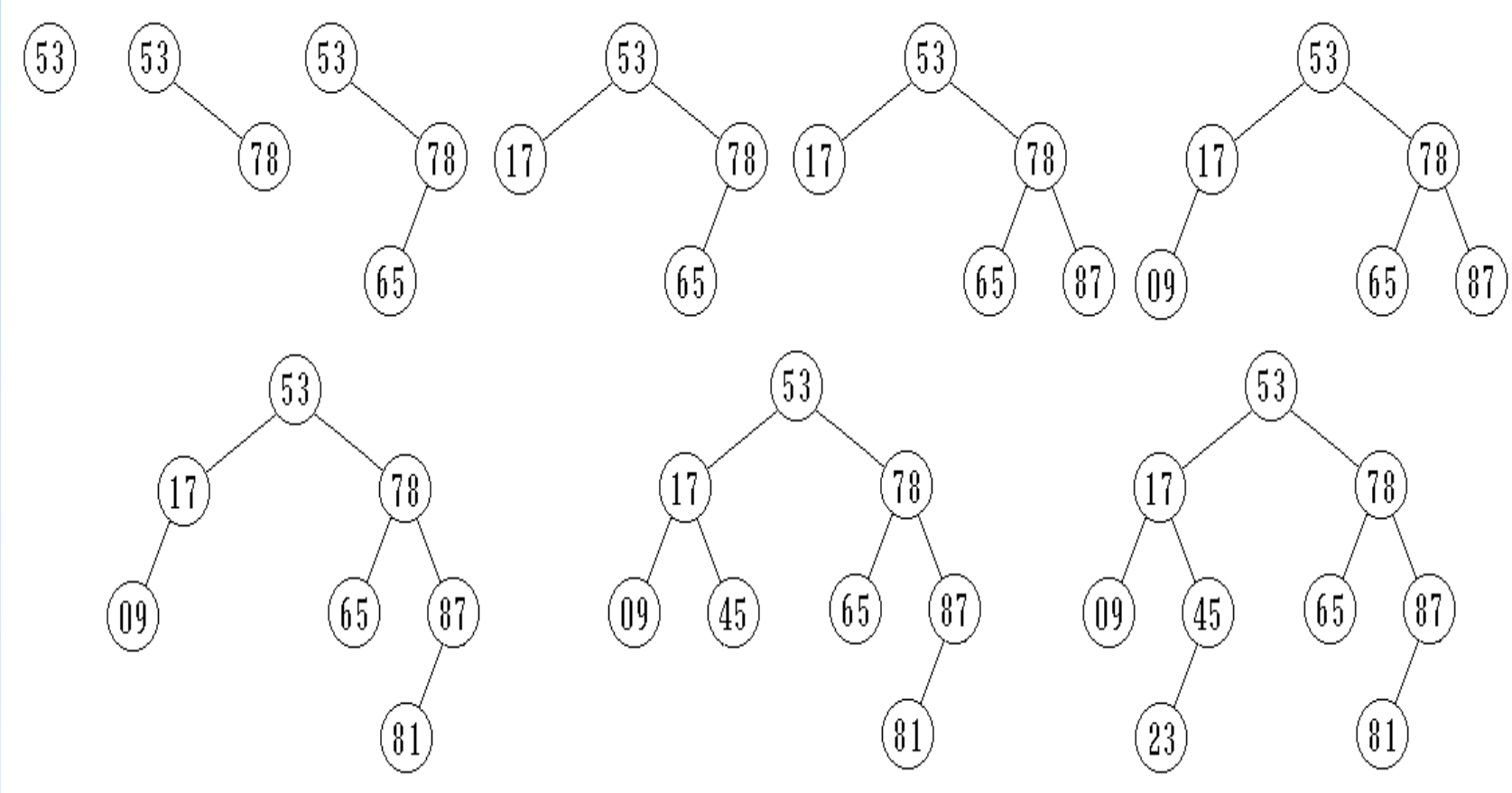
修改后的非递归查找算法

```
BiTree SearchBST(BiTree T, KeyType xkey, BiTree *pre)
{ // 查找成功，返回指向要找的结点的指针，pre指向该结点的父结点；
  // 否则返回NULL，pre指向查找路径上的最后一个结点。
    BiTree p;       p = T;
     *pre = NULL;
     while(p && !(xkey== p->data.key)
     { (* pre) = p;
         if((xkey< p->data.key))  p = p->lchild;
         else  p = p->rchild;
     }
     return( p );
}
```

二叉排序树的插入算法

```
Void InsertBST( BiTree T, ElemType e )
{  BiTree pre,s;
   if( !SearchBST( T, e.key, &pre ))  // 查找不成功
   {   s = ( BiTNpde * )malloc(sizeof( BiTNode )); // 构造新结点
       s->data = e; s->lchild = s->rchild = NULL;
       if( !pre ) T = s;                    // 原树为空时，新结点为根
       else if( LT( e.key, pre->data.key ) )
                  pre->lchild = s;      // 新结点为左子结点
       else   pre->rchild = s;          // 新结点为右子结点
}
}
```

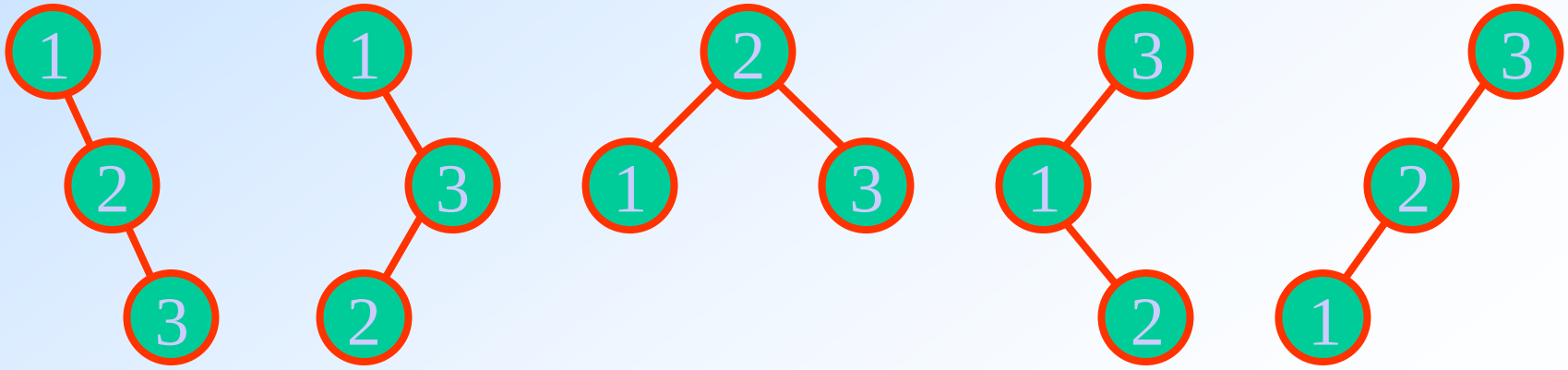
• 输入数据序列 { 53, 78, 65, 17, 87, 09, 81, 45, 23 }， 建立二叉排序树的过程



- 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，使得二叉排序树的深度达到最大，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}



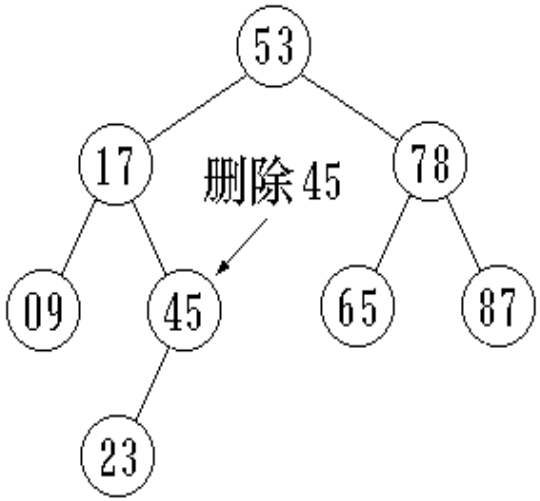
结论

- 一个无序序列可以通过构造一棵二叉排序树而变成一个有序序列
- 每次插入的新结点都是二叉排序树上新的叶子结点
- 插入时不必移动其它结点，仅需修改某个结点的指针

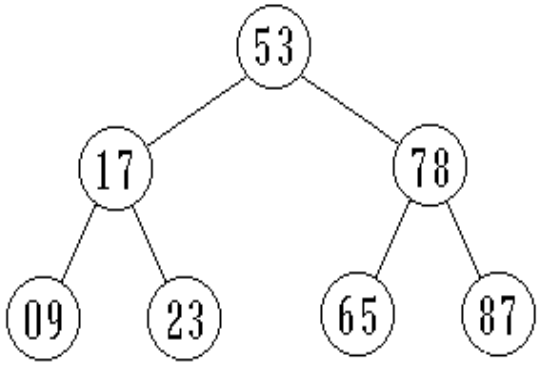
二叉排序树的删除

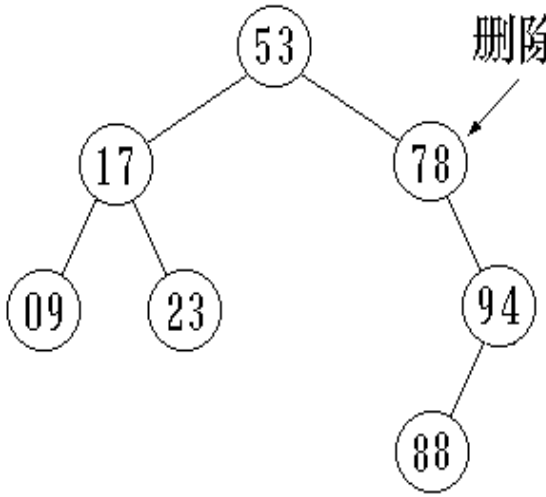
在二叉排序树中删除一个结点时，必须将因删除结点而断开的二叉链表重新链接起来，同时确保二叉排序树的性质。

- 删除叶结点，将其双亲结点指向它的指针清零，再释放。
- 被删结点缺右子树，左子女结点顶替它的位置，再释放它。
- 被删结点缺左子树，右子女结点顶替它的位置，再释放它。
- 被删结点左、右子树都存在，在它的右子树中寻找中序下的第一个结点(关键字最小),用它的值填补到被删结点中，再来处理这个结点的删除问题。

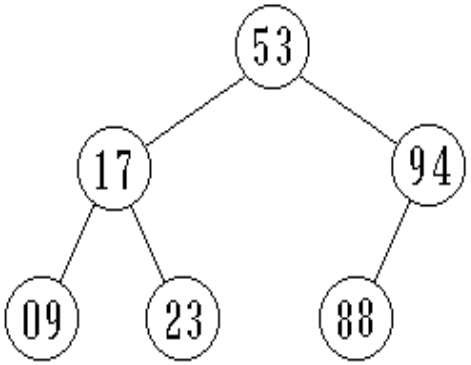


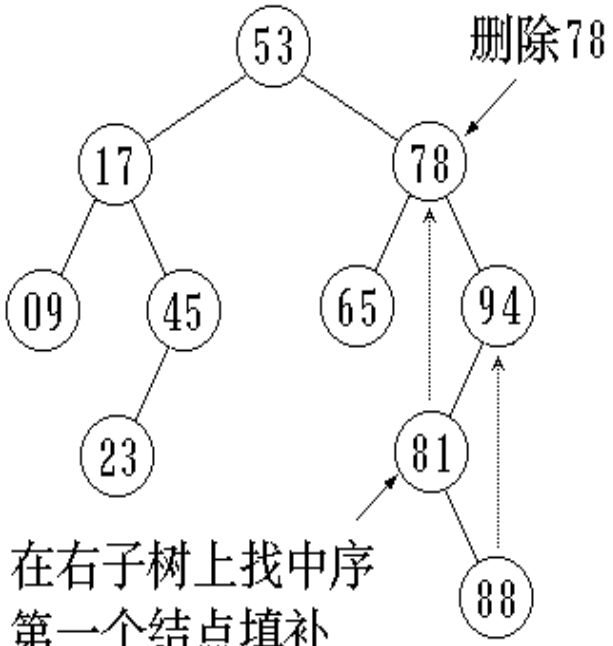
缺右子树用左子女填补



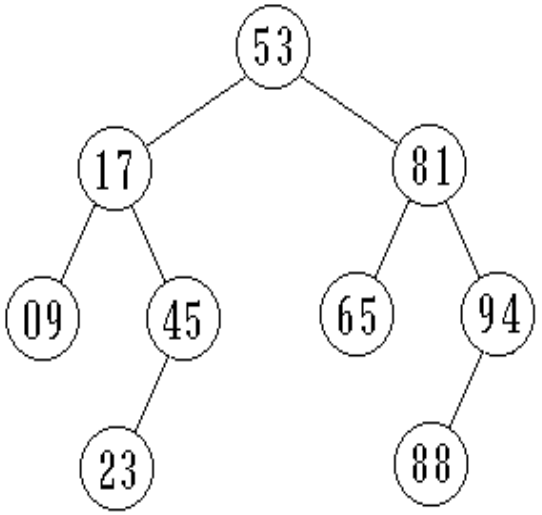


缺左子树用右子女填补





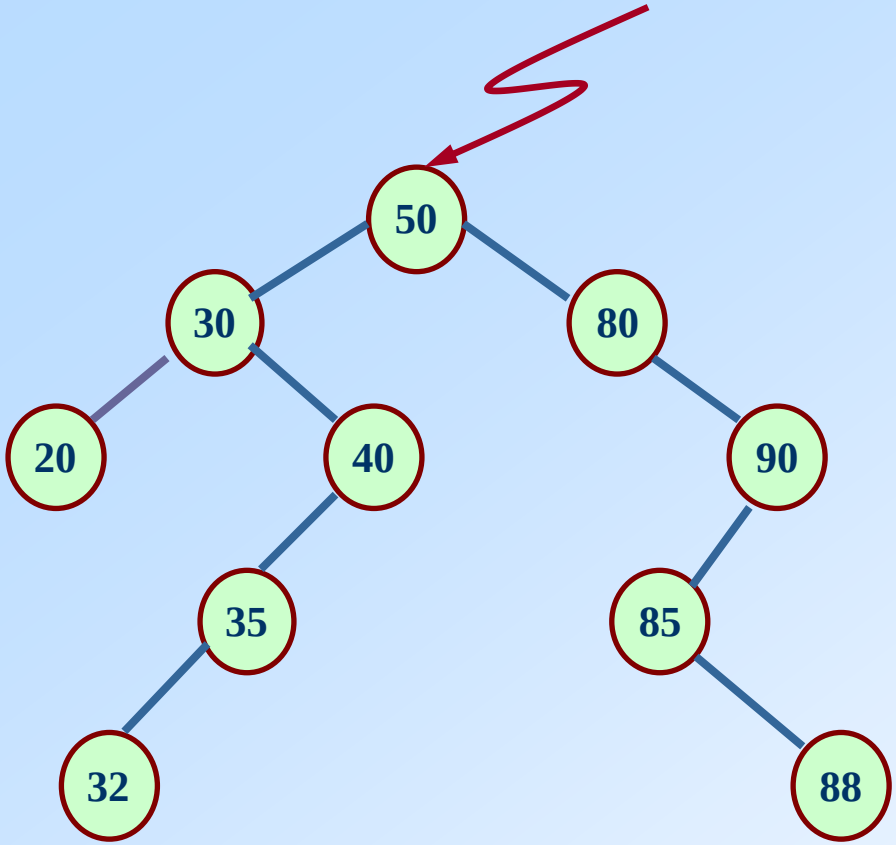
在右子树上找中序第一个结点填补

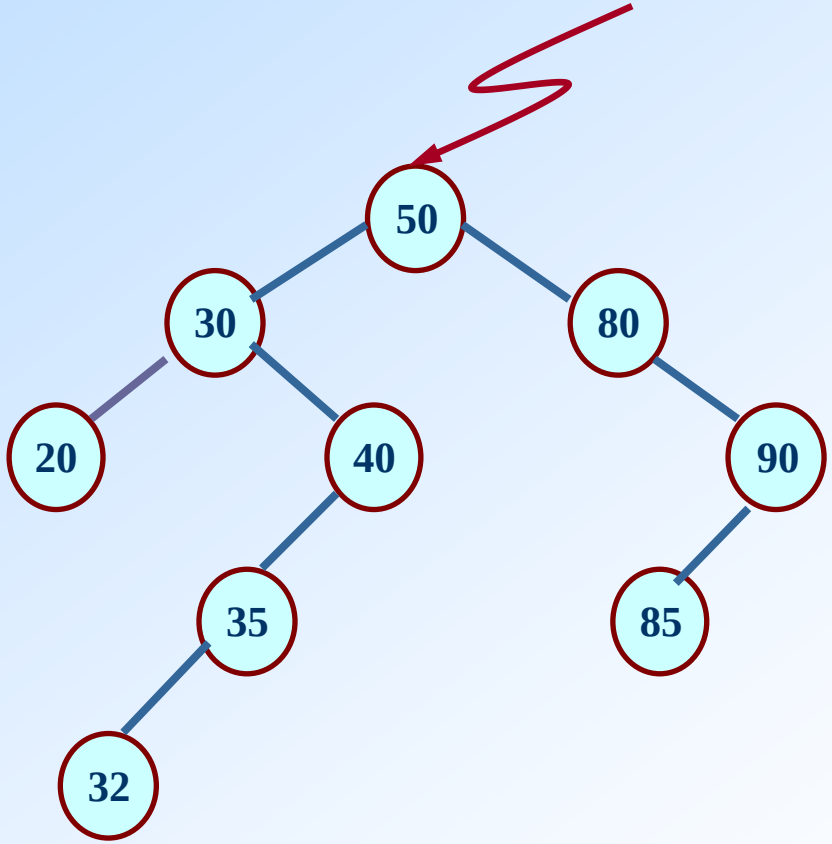


二叉排序树的删除算法

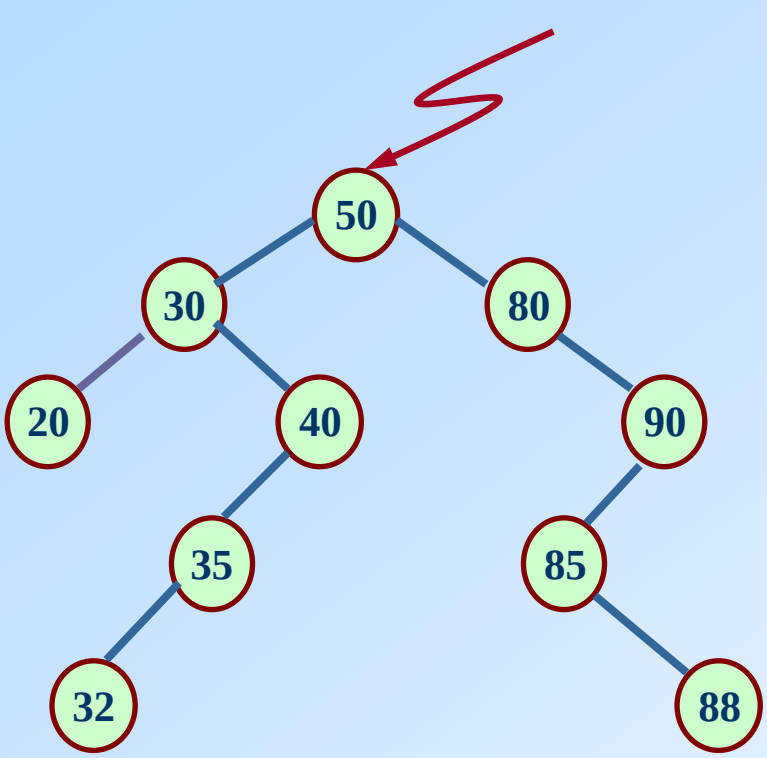
```
void DeleteBST( BiTree T, KeyType key )
{  BiTree  p, f, s, q;
   p = SearchBST( T, key, &f );
   if( !p )  return( FALSE );            // 不存在待删除结点
   if( !p->lchild || !p->rchild)         // 待删除结点无左子树或右子树
   {  q = ( p->lchild )?  p->lchild: p->rchild;
      if( !f )  T = q;
      else if( p==f->lchild)   f->lchild = q;
      else  f->rchild = q;
      free( p );
   }
```

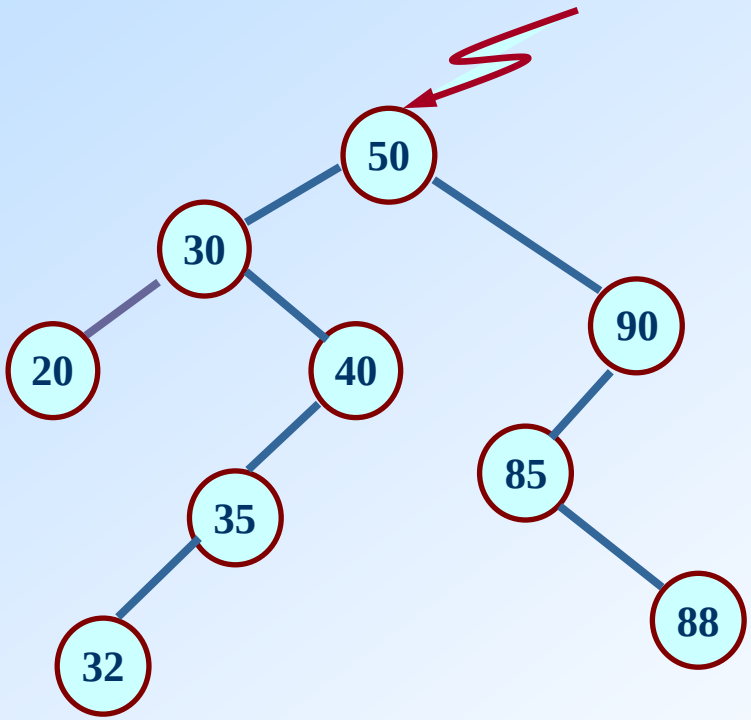
```
    else                                    //待删除结点存在左、右子树
    {    q = p;
         s = p->rchild;                     // 找寻*p右子树中的最左结点*s
         while( s->lchild )
         {   q = s;                         // *q为*s的父结点
             s = s->lchild;
         }
         p->data = s->data;      // 用*s代替*p
        if (q != p )  q->rchild = s->lchild;
        else  q->lchild = s->lchild;
         free( s );
    }
}
```





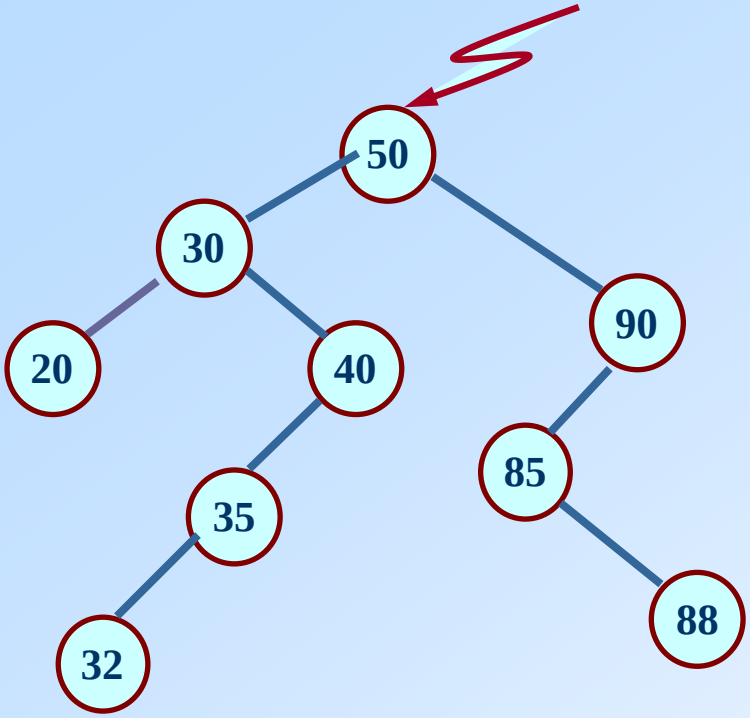
被删除的结点是叶子结点,如Key = 88
其双亲结点中相应指针域的值改为空

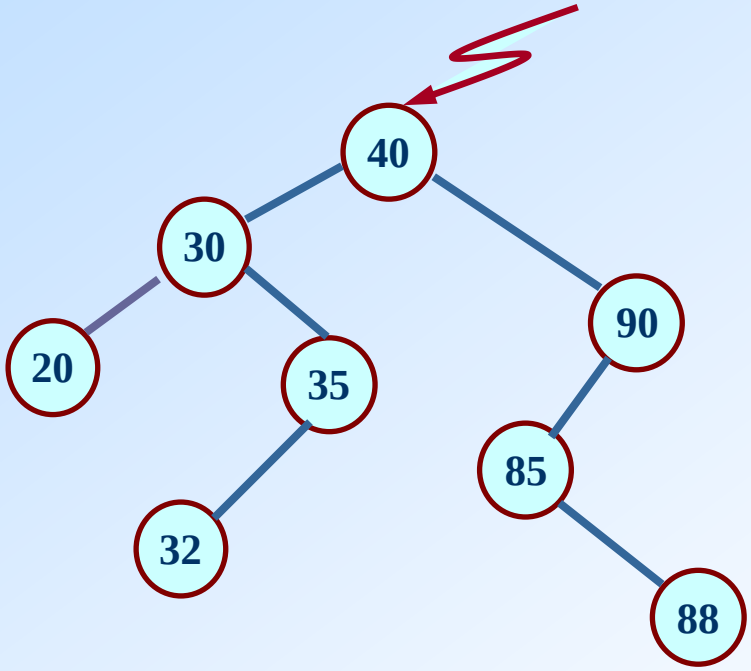




被删除的结点只有左子树或者只有右子树如key=80

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”。





被删除的结点既有左子树，也有右子树

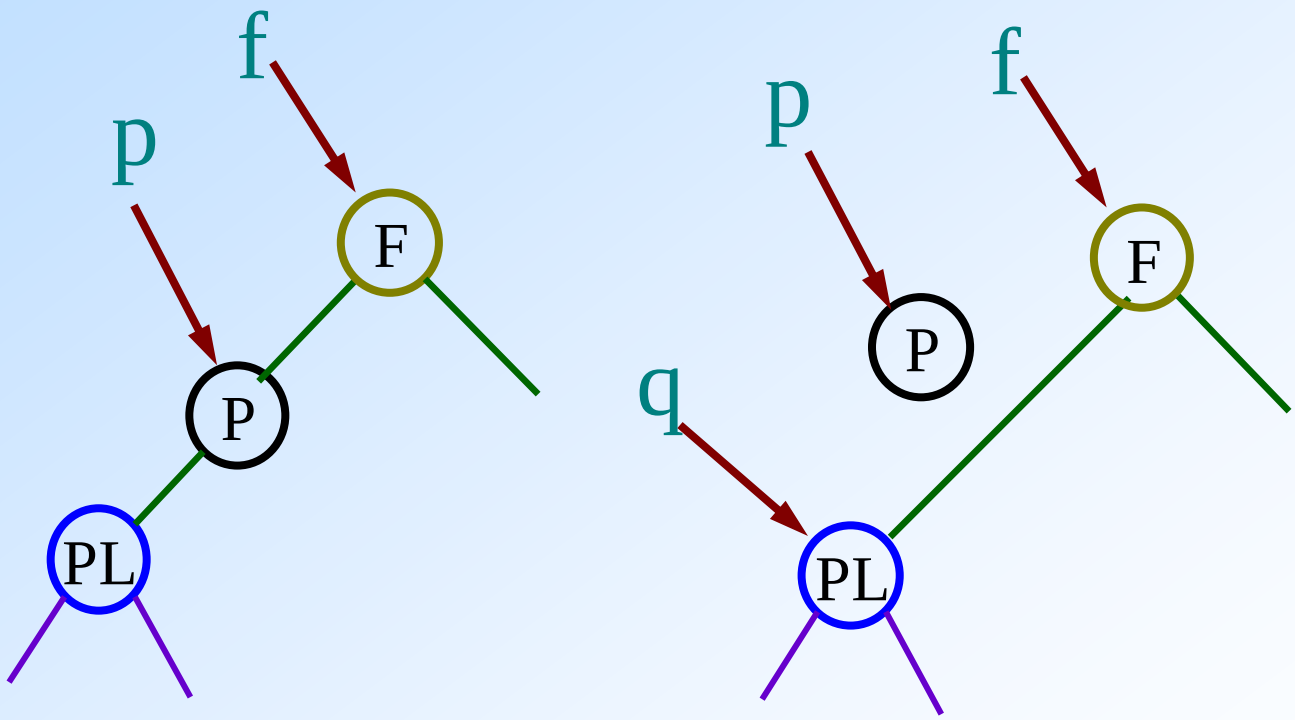
如被删关键字key = 50

以其前驱替代之，然后再删除该前驱结点

右子树为空树则只需重接它的左子树

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

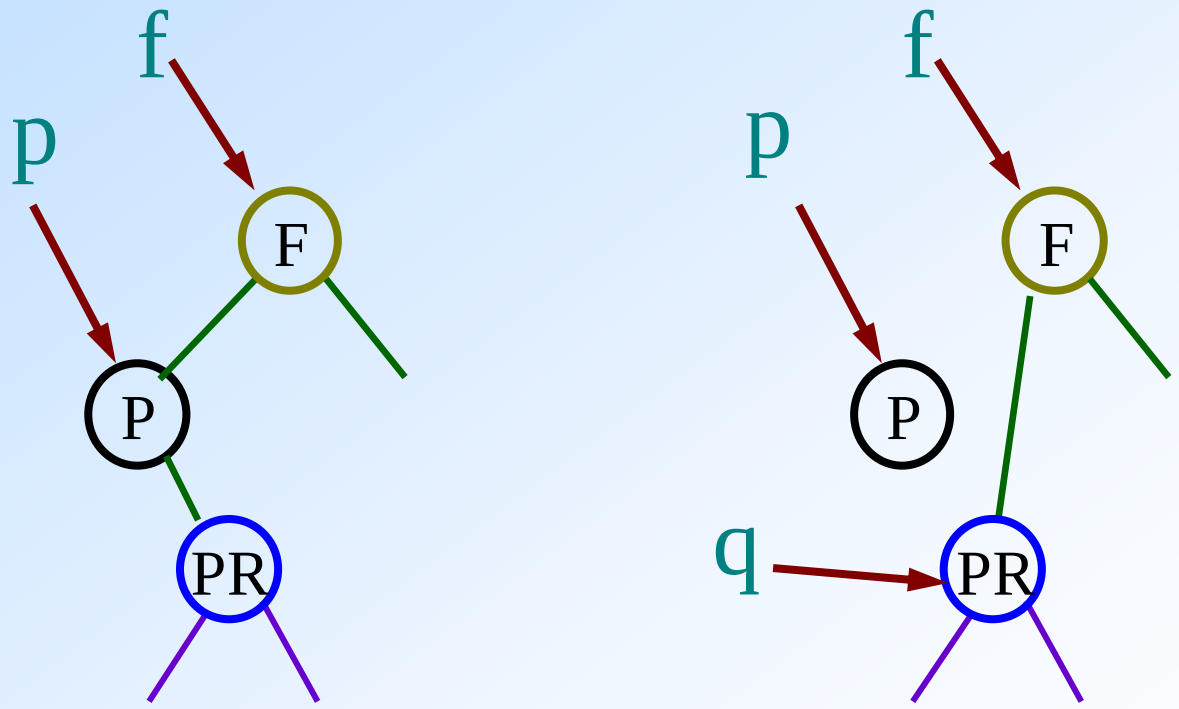
(或f->rchild=q)



左子树为空树只需重接它的右子树

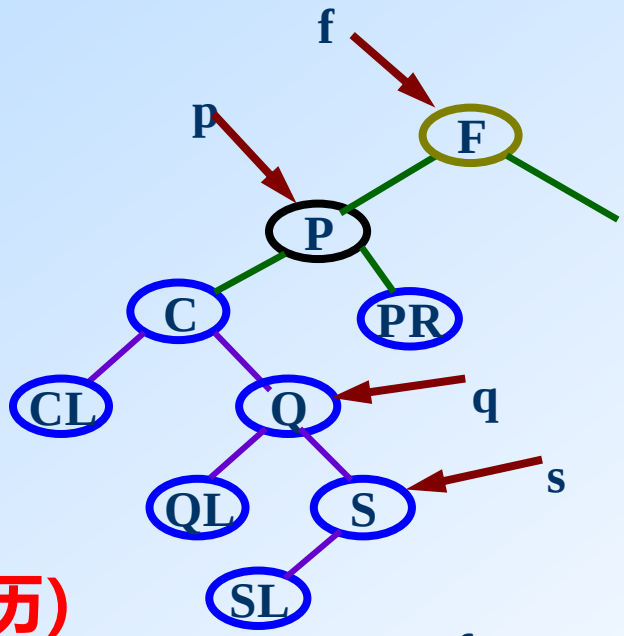
q = p->rchild; f->lchild=q; free(p);

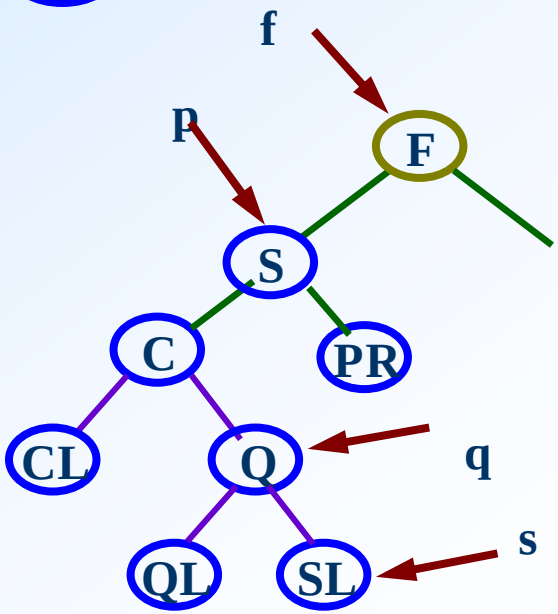
(或f->rchild=q)



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(中序遍历)
p->data = s->data;
if (q != p )  q->rchild = s->lchild;
else  q->lchild = s->lchild; //当p的左
子树根是s
delete(s);
```



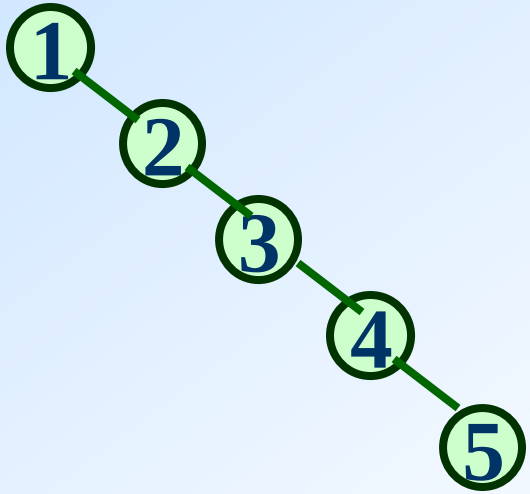


二叉排序树查找性能的分析

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。

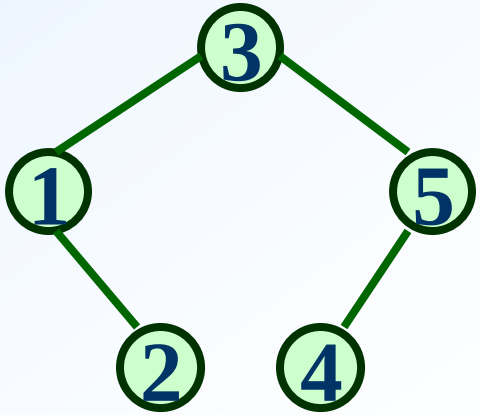
例如：

由关键字序列 1，2，3，4，5构造而得的二叉排序树，



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

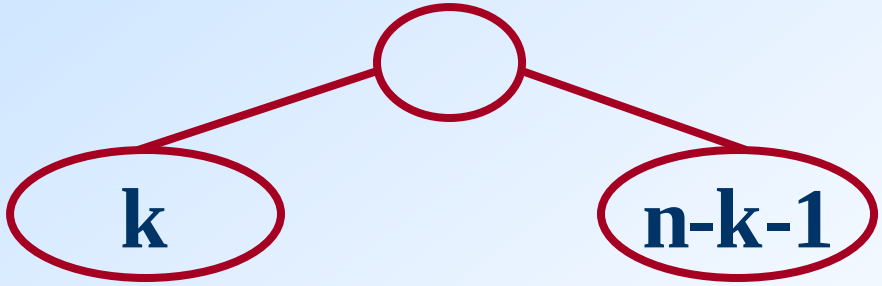
由关键字序列 3，1，2，5，4构造而得的二叉排序树



$$ASL = (1+2+3+2+3) / 5$$
$$= 2.2$$

下面讨论平均情况:

不失一般性，假设长度为 n 的序列中有 k 个关键字小于第一个关键字，则必有 n-k-1 个关键字大于第一个关键字,由它构造的二叉排序树



的平均查找长度是 n 和 k 的函数

$$P(n, k) \quad (0 \le k \le n-1)$$

$$P(n,k) = \frac{1}{n}\sum_{i=1}^{n} C_i = \frac{1}{n}\left(C_{root} + \sum_{L} C_i + \sum_{R} C_i \right)$$

$$= \frac{1}{n}\left(1 + k(P(k)+1) + (n-k-1)(P(n-k-1)+1)\right)$$

$$= 1 + \frac{1}{n}\left(k \times P(k) + (n-k-1) \times P(n-k-1)\right)$$

含 n 个关键字的二叉排序树的平均查找长度

$$ASL = P(n) = \frac{1}{n}\sum_{k=0}^{n-1} P(n,k)$$

由此

$$P(n)=\frac{1}{n}\sum_{k=0}^{n-1}\left(1+\frac{1}{n}\left(k\times P(k)+(n-k-1)\times P(n-k-1)\right)\right)$$

$$=1+\frac{2}{n^2}\sum_{k=1}^{n-1}\left(k\times P(k)\right)$$

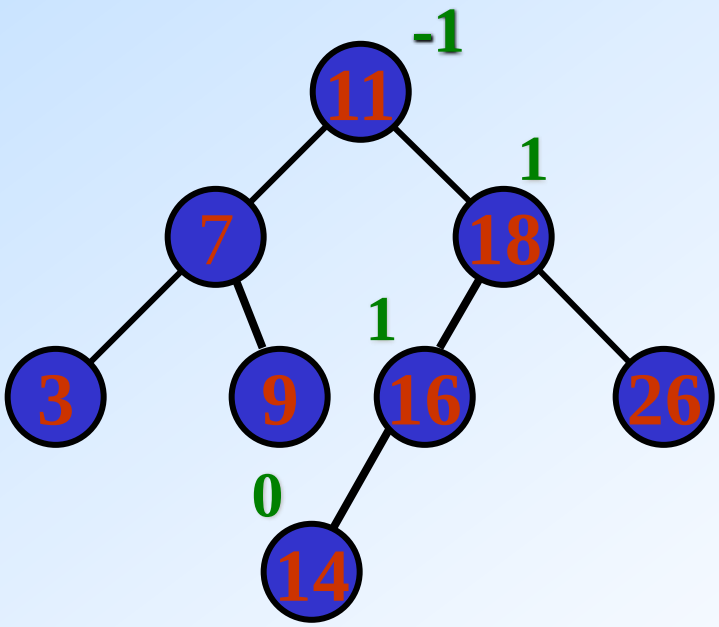
可类似于解差分方程，此递归方程有解：

$$P(n)=2\frac{n+1}{n}\log n+C$$

N	≈ 1 000	≈ 1 000 000	≈ 1 000 000 000
log **N**	10	Only 20	Only 30

平衡二叉树（AVL）

1 *AVL*树的定义：一棵*AVL*（1962年由Adelson, Velskli和Landis提出）树或者是空树，或者是具有下列性质的二叉排序树：它的左子树和右子树都是*AVL*树，且左子树和右子树的深度之差的绝对值不超过1。



深度平衡的二叉排序树

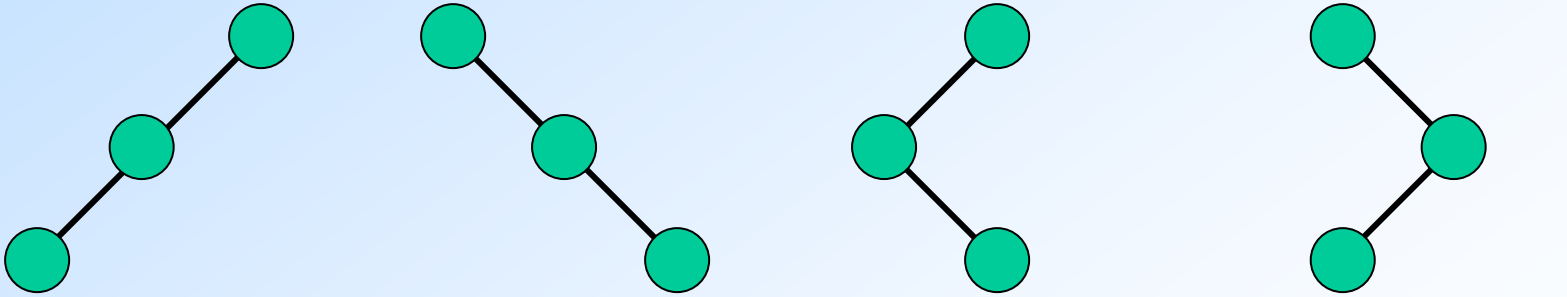
2 结点的平衡因子*balance* (balance factor)：

- 每个结点附加一个数字，给出该结点左子树的深度减去右子树的深度（或左子树的深度减去右子树的深度）所得的深度差。这个数字即为结点的平衡因子*balance* 。
- 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，则这棵二叉排序树就失去了平衡，不再是*AVL*树。
- 如果一棵二叉排序树是深度平衡的，它就成为 *AVL*树。如果它有 *n* 个结点，其深度可保持在$O(\log_2 n)$，平均查找长度也可保持在$O(\log_2 n)$。

3 平衡化旋转：

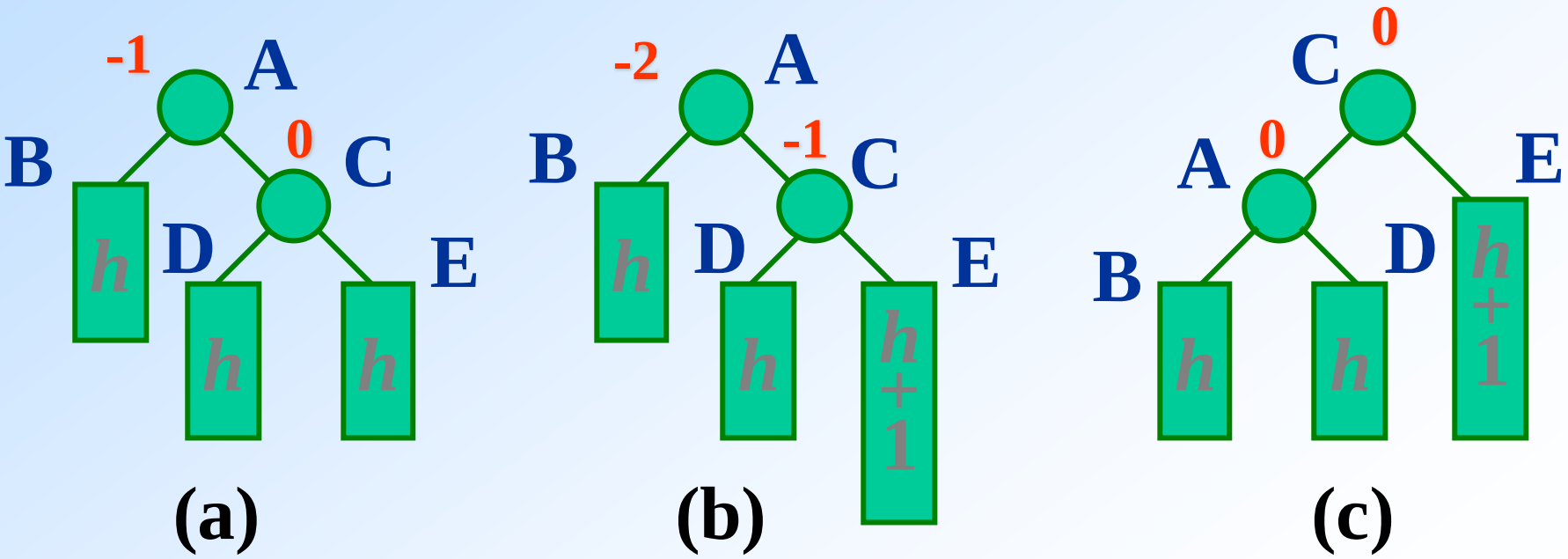
- 如果在一棵平衡的二叉排序树中插入一个新结点，造成了不平衡。此时必须调整树的结构，使之平衡化。
- 必须保证平衡化后的二叉树依然是一棵二叉排序树。
- 每插入一个新结点时，*AVL*树中相关结点的平衡状态会发生改变。因此，在插入一个新结点后，需要从插入位置沿通向根的路径回溯，检查各结点的平衡因子(左、右子树的深度差)。
- 如果在某一结点发现深度不平衡，停止回溯。
- 从发生不平衡的结点起，沿刚才回溯的路径取直接下两层的结点。

- **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。



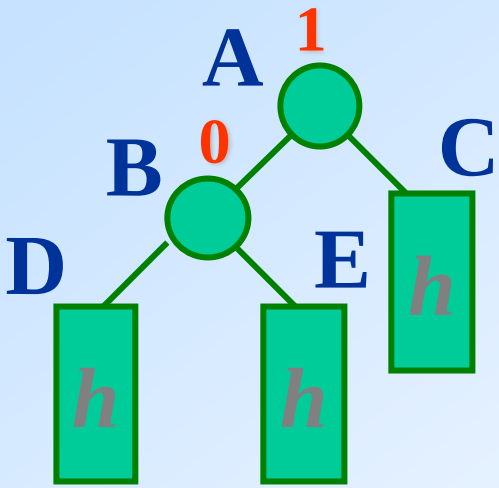
左单旋转 (RR型):

- 如果在子树E中插入一个新结点，该子树深度增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。

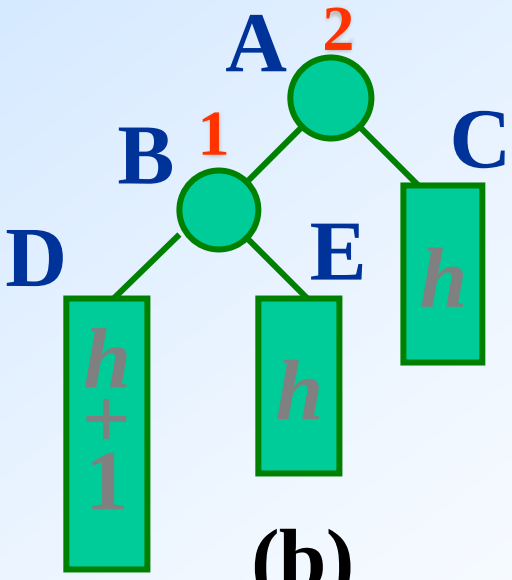


右单旋转 (LL型):

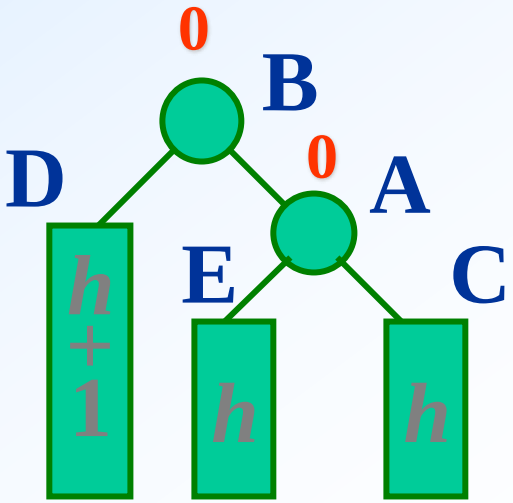
- 在左子树D上插入新结点使其深度增1，导致结点A的平衡因子增到 -2，造成了不平衡。
- 为使树恢复平衡，从A沿插入路径连续取3个结点A、B和D，它们处于一条方向为“/”的直线上，需要做右单旋转。
- 以结点B为旋转轴，将结点A顺时针旋转。



(a)



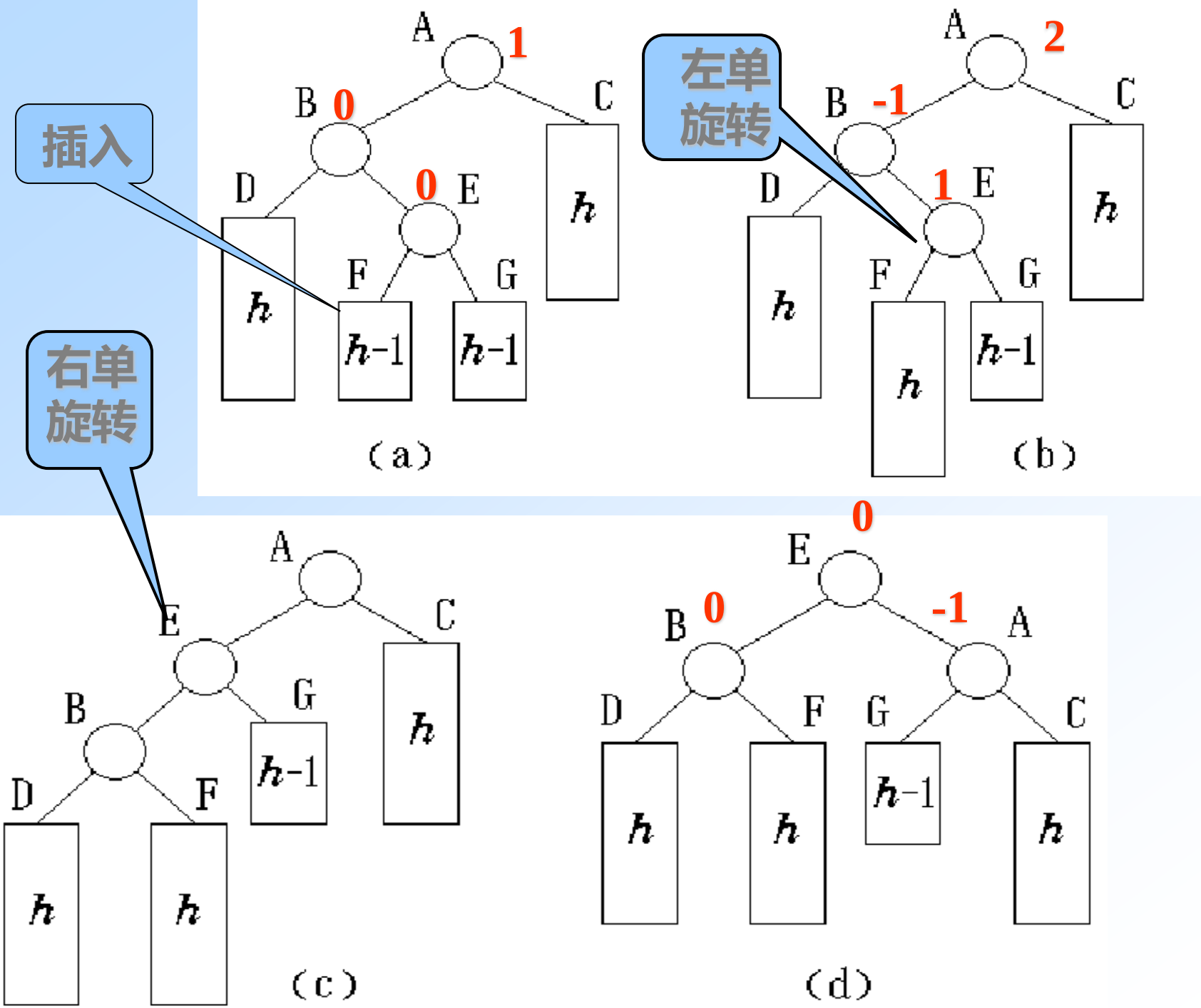
(b)



(c)

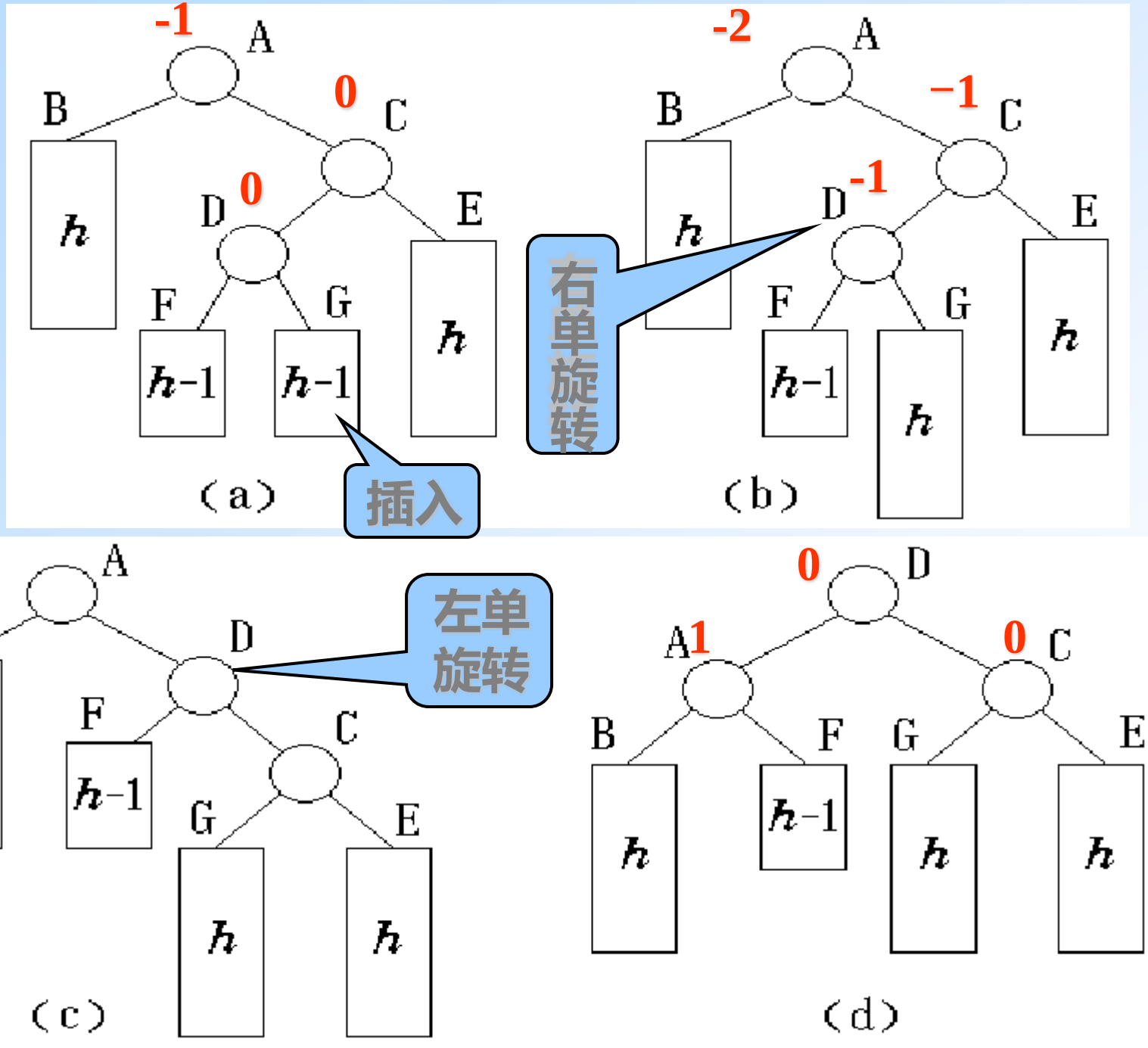
先左后右双旋转 (LR型):

- 在下图中子树F或G中插入新结点，该子树的深度增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。

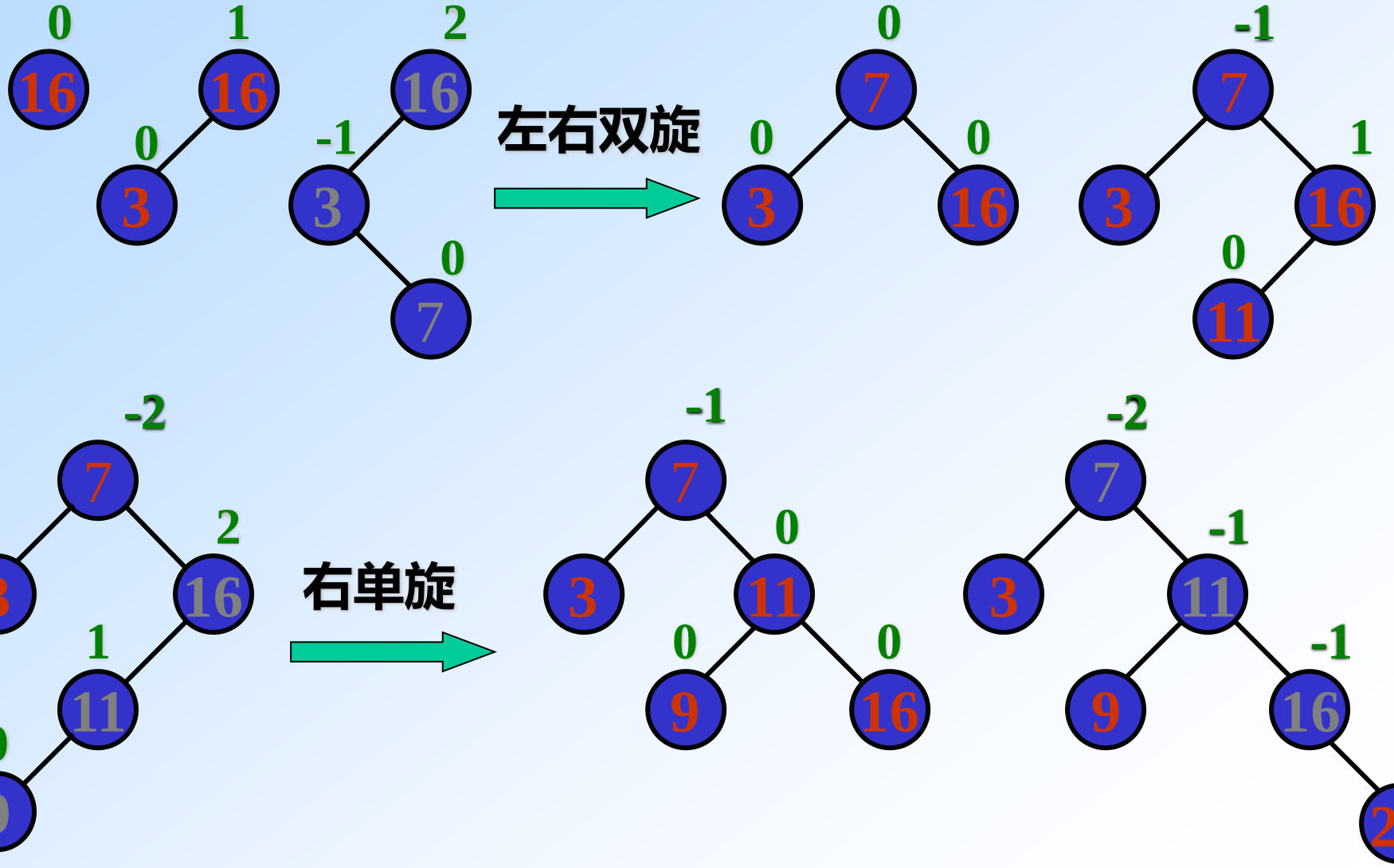

插入
A
1
B
0
C
D
0
E
h
F
G
h
h-1
h-1
(a)
左单旋转
A
2
B
-1
C
D
1
E
h
F
G
h
h-1
(b)
右单旋转
A
E
C
B
G
h
D
F
h-1
h
h
(c)
0
E
B
0
-1
A
D
F
G
C
h
h
h-1
h
(d)

先右后左双旋转 (RL型)：

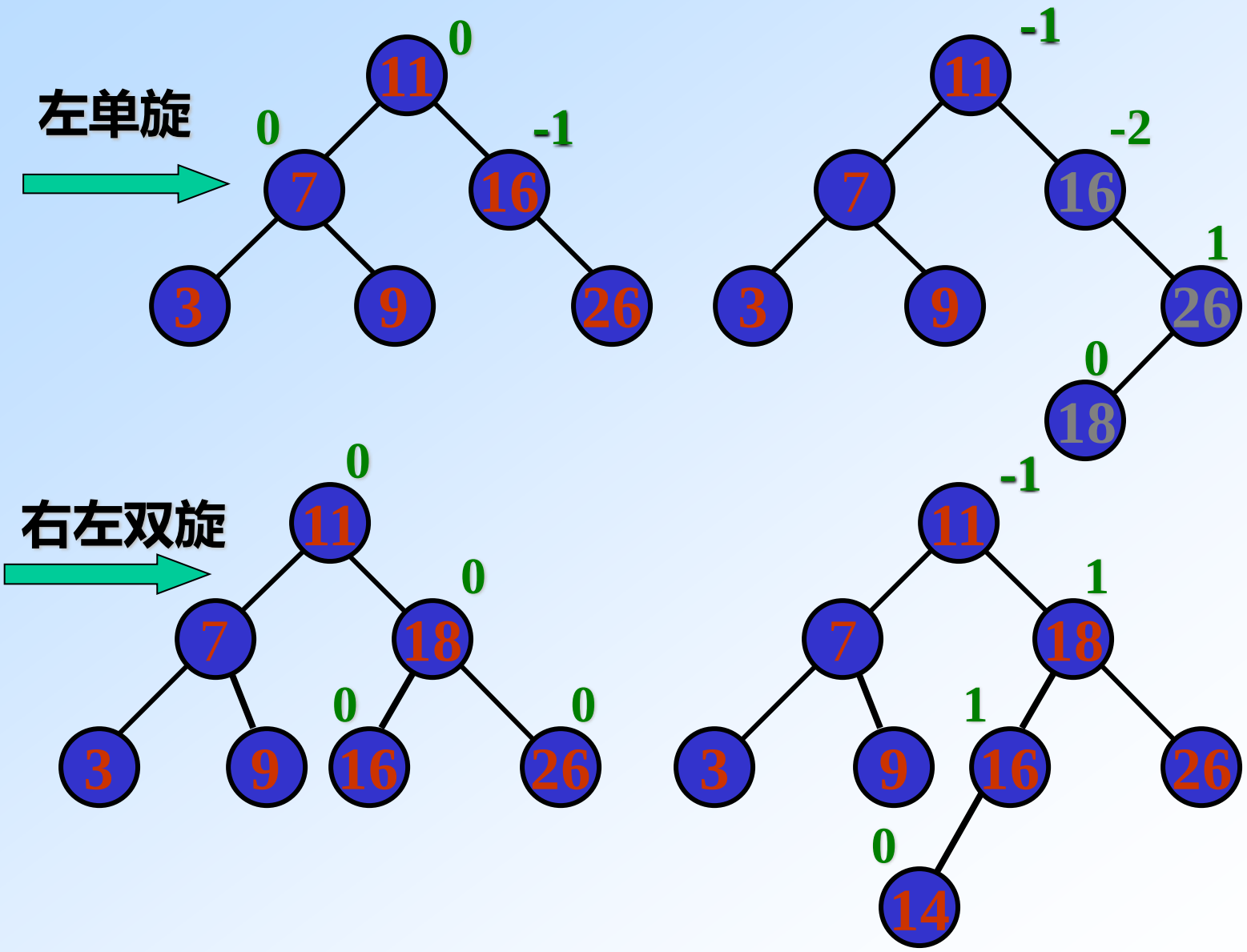
- 右左双旋转是左右双旋转的镜像。
- 在下图中子树F或G中插入新结点，该子树深度增1。结点A的平衡因子变为2，发生了不平衡。
- 从结点A起沿插入路径选取3个结点A、C和D，它们位于一条形如“〉”的折线上，需要进行先右后左的双旋转。
- 首先做右单旋转：以结点D为旋转轴，将结点C顺时针旋转，以D代替原来C的位置。
- 再做左单旋转：以结点D为旋转轴，将结点A反时针旋转，恢复树的平衡。


-1
A
B
0
C
h
D
0
E
F
G
h
h-1
h-1
h
(a)
插入
右单旋转
-2
A
B
-1
C
h
D
-1
E
F
G
h-1
h
h
(b)
A
B
D
h
F
C
h-1
G
E
h
h
(c)
左单旋转
0
D
A1
0
C
B
F
G
E
h
h-1
h
h
(d)

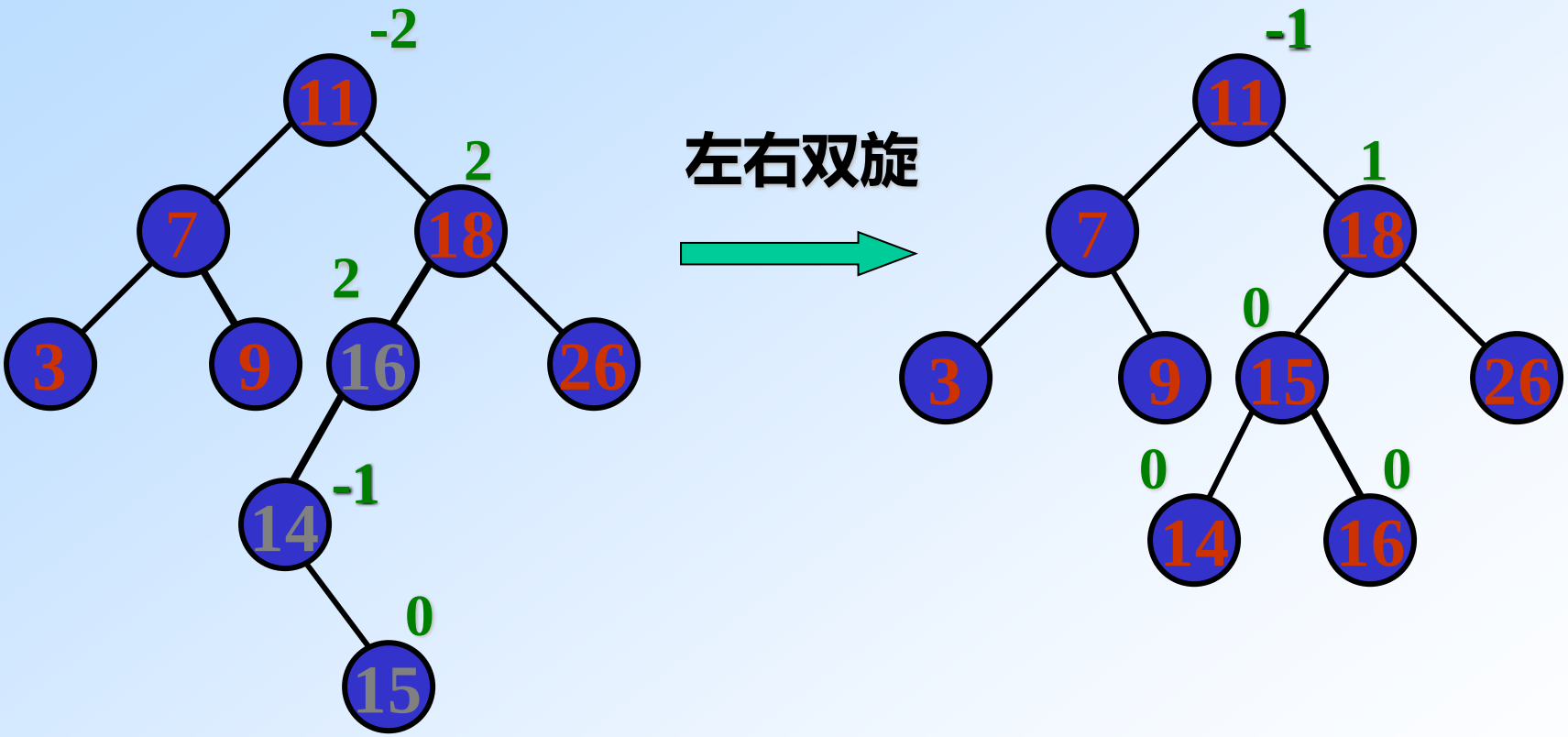
例：输入关键字序列 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }，
建立一棵AVL树。



例：输入关键字序列 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }，
建立一棵AVL树。



例：输入关键字序列 { 16, 3, 7, 11, 9, 26, 18, 14, 15 }，
建立一棵AVL树。



从空树开始的建树过程

AVL树查找的分析

深度 $h = 1; 2; 3; 4; 5; 6; 7;$

最少结点数 $N = 1; 2; 4; 7; 12; 20; 33;$

- 在*AVL*树上进行查找的过程与排序树相同，因此在查找过程中与给定值进行比较的关键字个数不超过树的深度。
- 设 N_h 是深度为 h 的*AVL*树的最小结点数。根的一棵子树的深度为 h-1，另一棵子树的深度为h-2，这两棵子树也是*AVL*树。因此有：

$$N_0 = 0;\quad N_1 = 1;\quad N_h = N_{h-1} + N_{h-2} + 1,\ h > 0$$

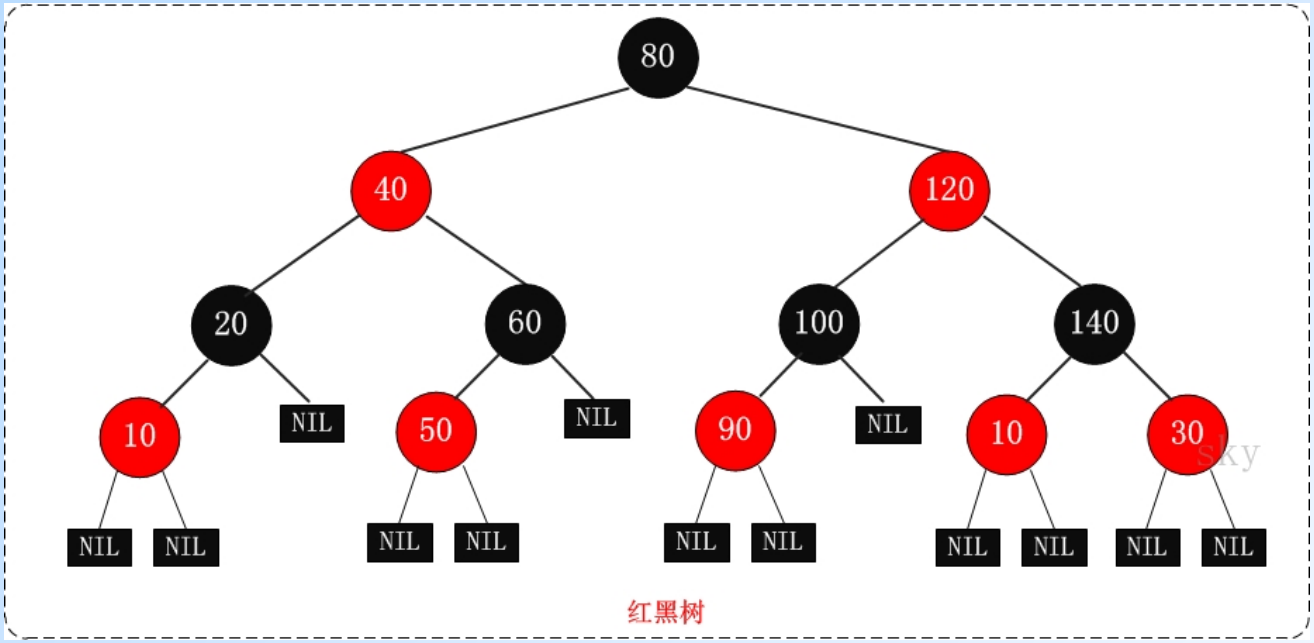
- 可以证明，对于 $h \geq 0$，有 $N_h = F_{h+2} - 1 = \dfrac{\varphi^{h+2}}{\sqrt{5}} - 1$，其中 $\varphi = \dfrac{1+\sqrt{5}}{2}$
- 有 n 个结点的*AVL*树的深度为：

$$h = \log_{\varphi}(\sqrt{5}(n+1)) - 2 \approx 1.44\log_2(n+1)$$

- 在*AVL*树进行查找的时间复杂度为 $O(\log_2 n)$。

红黑树(Red Black Tree)

- 红黑树 是一种自平衡二叉查找树。在1972年由Rudolf Bayer发明的，1978年被 Leo J. Guibas 和 Robert Sedgewick 修改为“红黑树”。
- 红黑树和AVL树类似，都是在进行插入和删除操作时，通过特定操作保持二叉查找树的平衡，从而获得较高的查找性能。

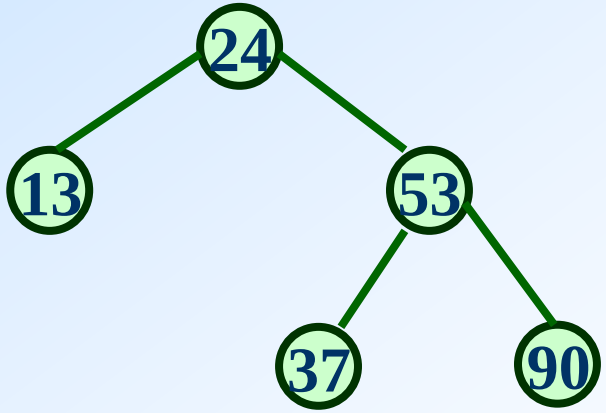


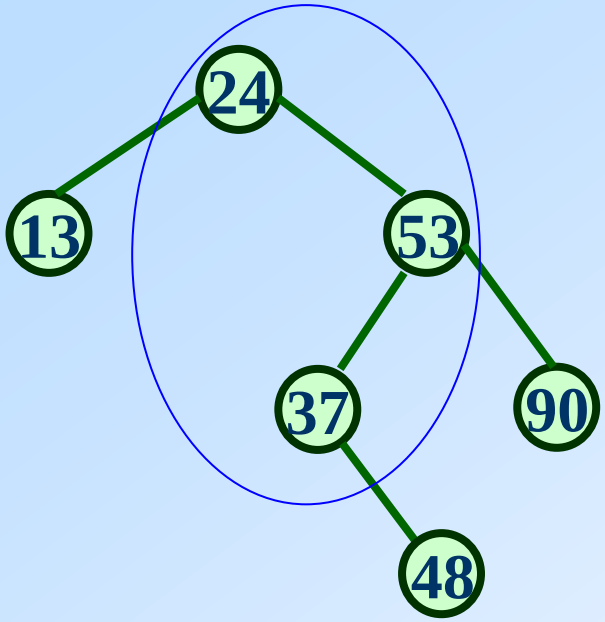
- 性质1. 节点是红色或黑色。
- 性质2. 根节点是黑色。
- 性质3 每个叶节点（NIL节点，空节点）是黑色的。
- 性质4 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)
- 性质5. 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。（没有路径能多于任何其他路径的两倍长。）

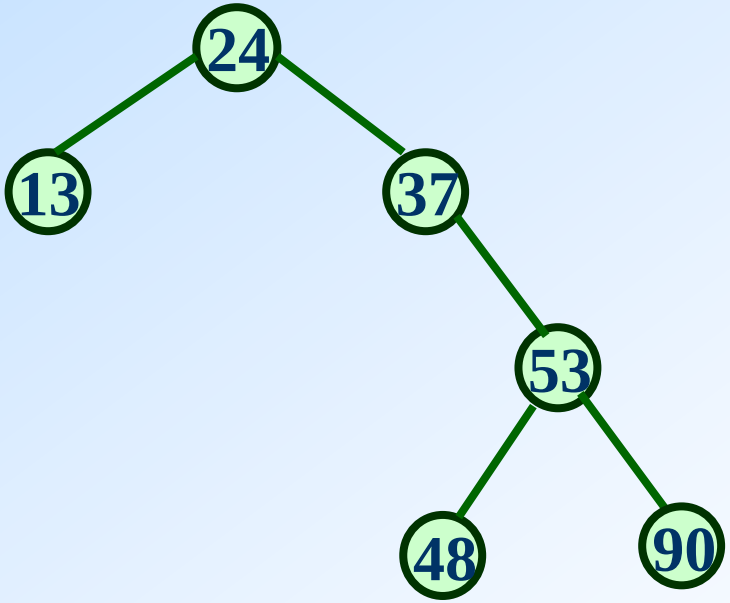
- 统计性能要好于平衡二叉树，红黑树在很多地方都有应用。java中的TreeSet和TreeMap数据结构。在C++ STL中，很多部分(包括set, multiset, map, multimap)应用了红黑树的变体。

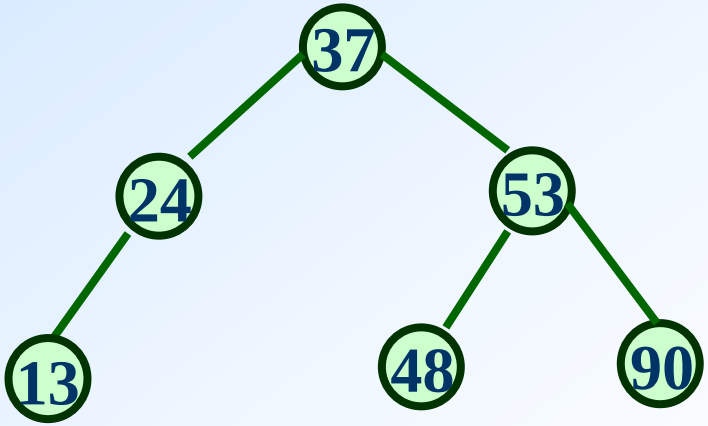
思考：如图所示的平衡二叉树中插入关键字48后，得到一棵新平衡二叉树，在新平衡二叉树中，关键字37所在结点的左、右子节点中保存的关键字分别为 C

A）13,48　　B）24,48　　C）24,53　　D）24,90




24
13
53
37
90
48

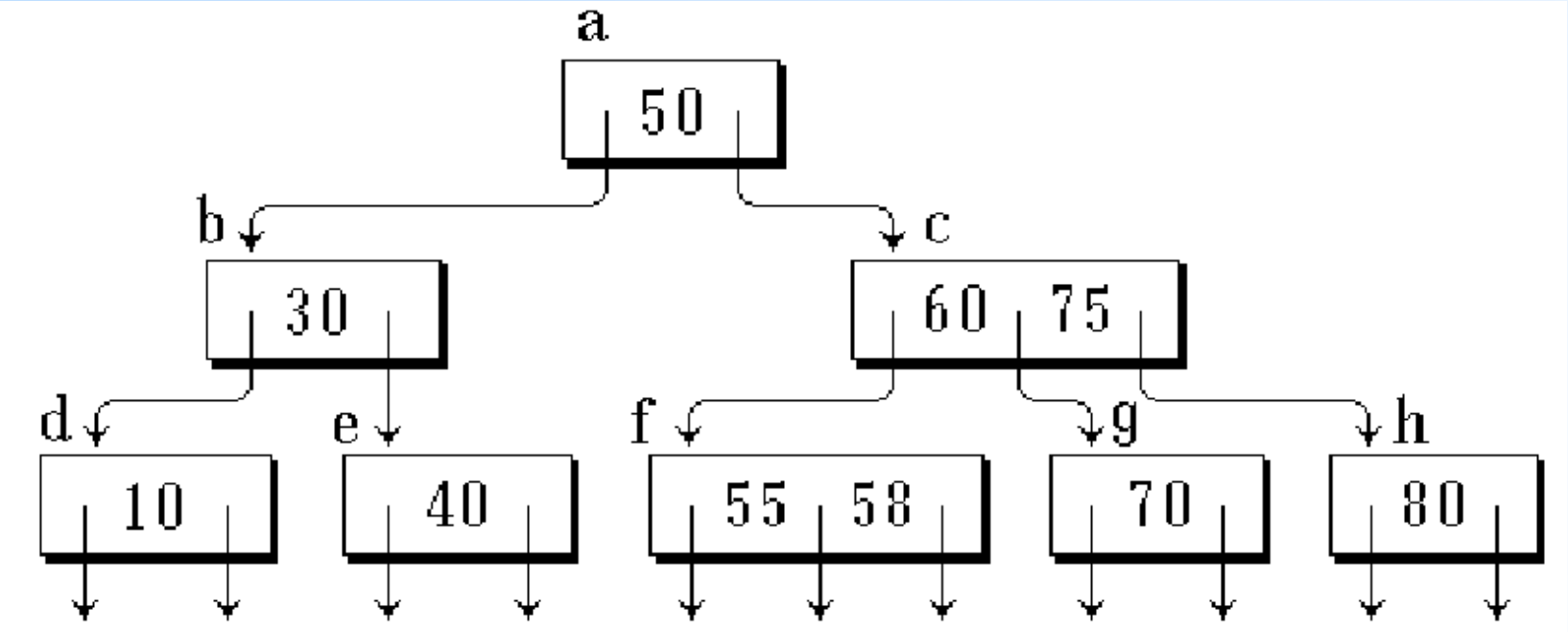

24
13
37
53
48
90


37
24
53
13
48
90

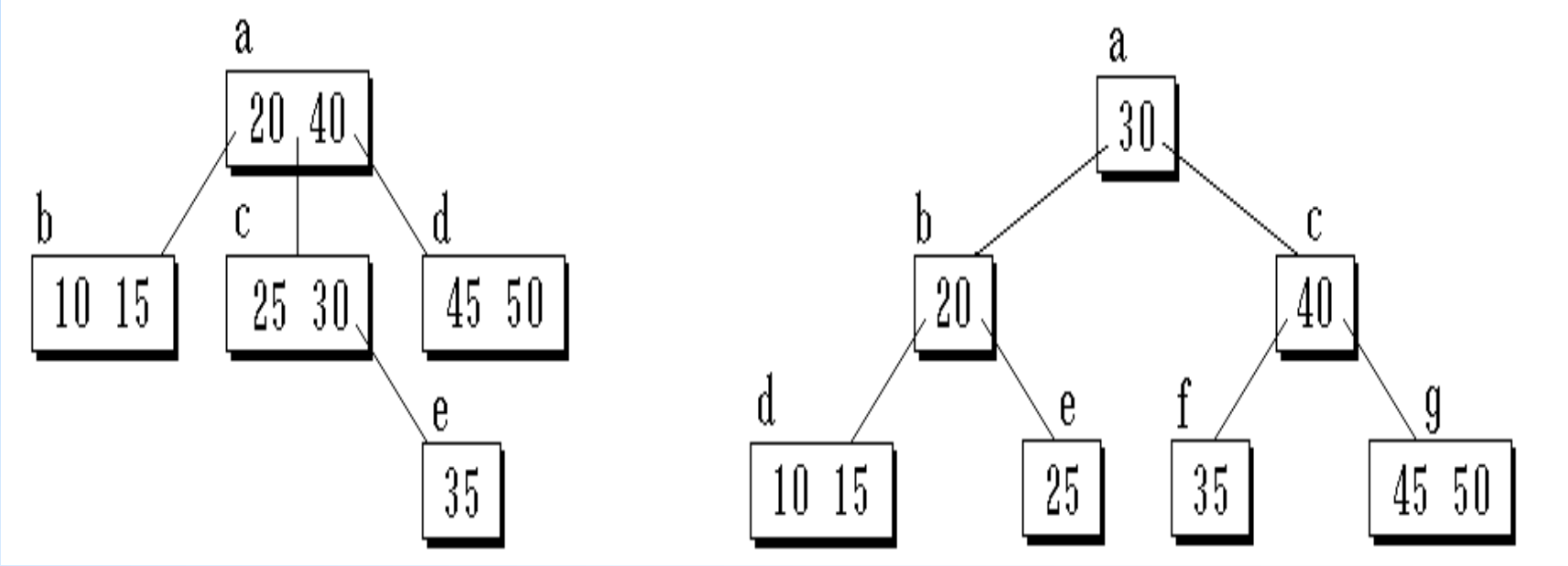
9.2.2 B_树和B+树

一、B_树及其查找

- **一棵 *m* 阶B_树是一棵 *m* 路查找树，它或者是空树，或者是满足下列性质的*m*叉树：**
  - **根结点不是叶子结点时至少有 2 棵子树。**
  - **树中每个结点至多有*m*棵子树。**
  - **除根结点以外的所有非叶子结点至少有⌈*m*/2⌉棵子树。**
  - **所有非叶子结点的组成为$(n,A_0,K_1,A_1,K_2,A_2,\ldots,K_n,A_n)$， n为关键字个数，$K_i$为关键字，$A_i$为指向子树的指针。$A_{i-1}$指向子树的结点的关键字$< K_i < A_i$指向子树的结点的关键字.**
  - **所有的叶子结点都位于同一层。**
- **一棵 *m* 阶B_树是一棵 *m*叉有序平衡树**


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

- 非B_树和 B_树示例：

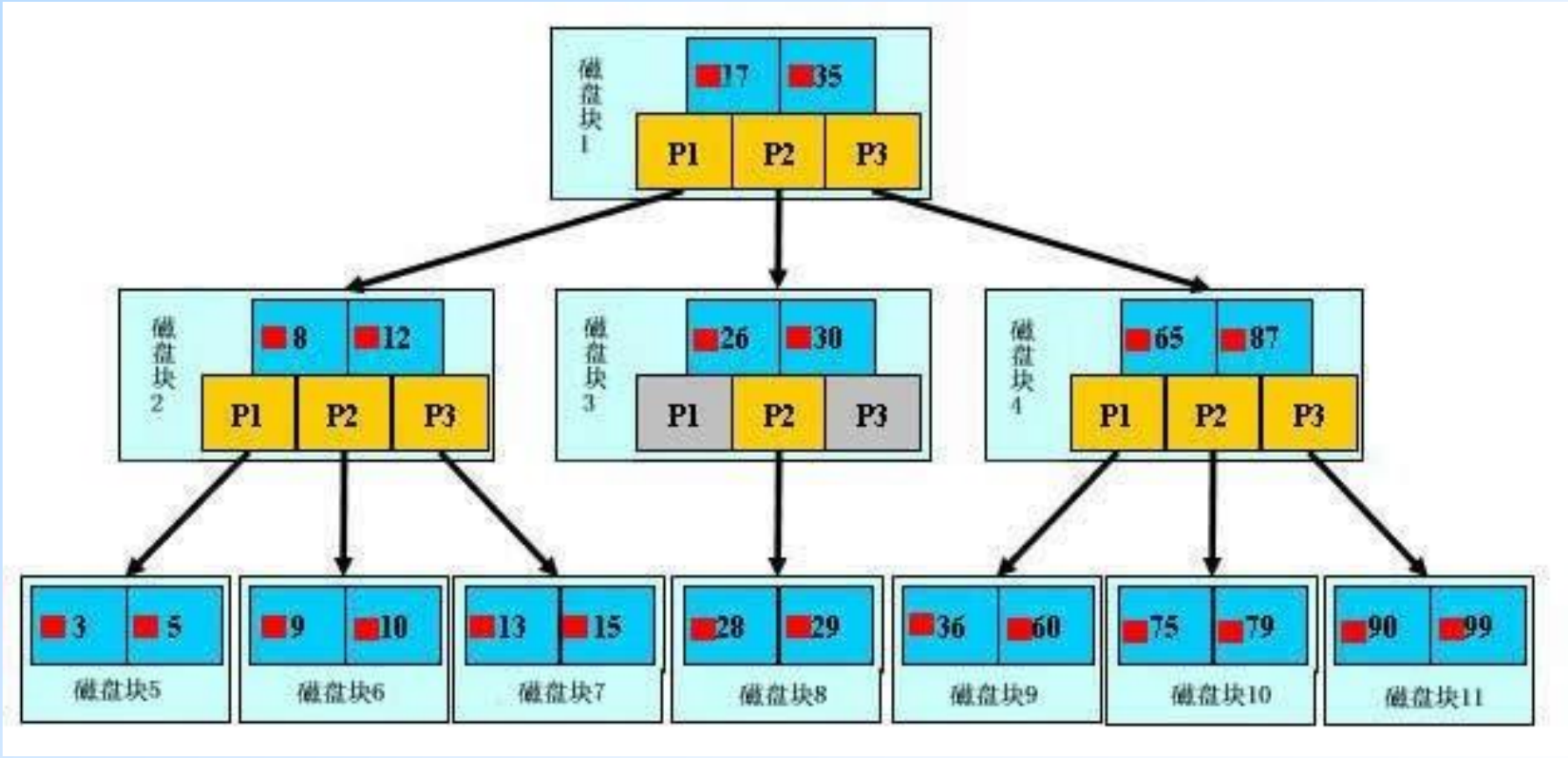


非B_树　　B_树

- 为什么采用B_ 树和 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问外存次数。

- 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。数据组织方式要求树尽量低，这样进行的I/O操作就会降低到最少！


磁盘块1
17
35
P1
P2
P3
磁盘块2
8
12
P1
P2
P3
磁盘块3
26
30
P1
P2
P3
磁盘块4
65
87
P1
P2
P3
3
5
磁盘块5
9
10
磁盘块6
13
15
磁盘块7
28
29
磁盘块8
36
60
磁盘块9
75
79
磁盘块10
90
99
磁盘块11

- 模拟查找文件29的过程：
- A.根据根结点指针找到文件目录的根磁盘块1，将其中的信息导入内存。【磁盘IO操作 1次】
- B.此时内存中有两个文件名17、35和三个存储其他磁盘页面地址的数据。根据算法我们发现17<29<35，因此我们找到指针p2。
- C.根据p2指针，我们定位到磁盘块3，并将其中的信息导入内存。【磁盘IO操作 2次】
- D.此时内存中有两个文件名26，30和三个存储其他磁盘页面地址的数据。根据算法我们发,26<29<30，因此我们找到指针p2。
- E.根据p2指针，我们定位到磁盘块8，并将其中的信息导入内存。【磁盘IO操作 3次】
- F.此时内存中有两个文件名28，29。根据算法我们查找到文件名29，并定位了该文件内存的磁盘地址。

- 分析上面的过程，发现需要3次磁盘IO操作和3次内存查找操作。关于内存中的文件名查找，由于是一个有序表结构，可以利用折半查找提高效率。至于IO操作是影响整个B树查找效率的决定因素。

- **B_树的结点类型说明：**

```
#define m 3                          //一棵m阶的B_树
typedef struct BTNode{               //结点定义
    int              keynum;         //结点中关键字个数
    struct BTNode *parent;           //双亲指针
    KeyType          key[m+1];       //关键字数组 1~m-1
    struct BTNode *ptr[m+1];         //子树指针数组 0~m
};
typedef struct{
    BTNode       *pt;                //指向找到的结点
    int              i;              //1..m, 在结点中的关键字序号
    int              tag;            //1: 查找成功; 0: 查找失败
}Result;                             //B_树的查找结果类型
```

- **B_树的查找算法**

```
Result SearchBTree(Btree T, KeyType K)
{ // 返回(pt,i,tag)。查找成功，tag=1，pt所指结点中第i个关键字等于K。
  // 查找失败，则 tag=0，K应插到pt所指结点中第i和第i+1个关键字之间。
    p=T;  q=NULL;  found=FALSE;
    while(p && !found)
    {   for(i=p->keynum, p->key[0]=K; K<p->key[i]; i--);
                                    //查找i,使p->key[i]<=K<p->key[i+1]
        if( i>0 && p->key[i]==K)  found=TRUE;
        else { q=p; p=p->ptr[i]; }
    }
    if (found)  return(p,i,1);  //查找成功
    else return(q,i,0);         //查找不成功,返回插入位置
}
```

B_树查找分析

- B_树的查找过程是一个在结点内查找和循某一条路径向下一层查找交替进行的过程。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深度。从B_树的定义知, 每层关键字个数 N 最少有：
  - 第1层　1 个结点
  - 第2层　至少 2 个结点
  - 第3层　至少 $2\lceil m/2 \rceil$ 个结点
  - 第4层　至少 $2\lceil m/2 \rceil^2$ 个结点
  - 如此类推，……
  - 第h 层至少有$2\lceil m/2 \rceil^{h-2}$个结点。所有这些结点都不是失败结点。

- 若树中关键字有 N 个, 则失败结点数为 $N+1$。这是因为失败一般发生在 $K_i < x < K_{i+1}, 0 \le i \le N$，设$K_0 = -\infty$，$K_{N+1} = +\infty$。因此有

  $N+1$ = 失败的结点数目= 位于第 $h+1$层的结点数 $\ge 2\lceil m/2 \rceil^{h-1}$

  $N \ge 2\lceil m/2 \rceil^{h-1} - 1$

  反之，$h \le \log_{\lceil m/2 \rceil}((N+1)/2) + 1$
- 示例：若B_树的阶数 $m = 199$，关键字总数 $N = 1999999$，则B_树的深度 h 不超过

  $$\log_{100} 1000000 + 1 = 4$$
- 若B_树的阶数 $m = 3$，深度 $h = 4$，则关键字总数至少为

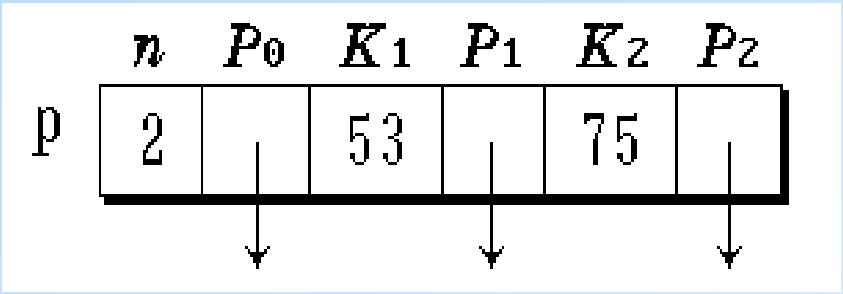
  $$N = 2\lceil 3/2 \rceil^{4-1} - 1 = 15$$

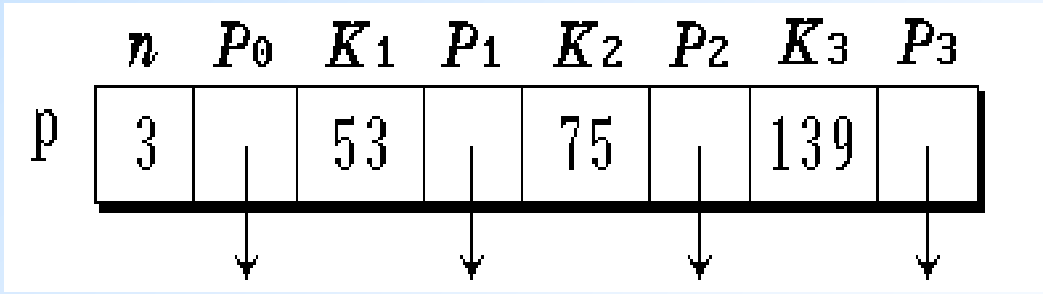
B_树的插入

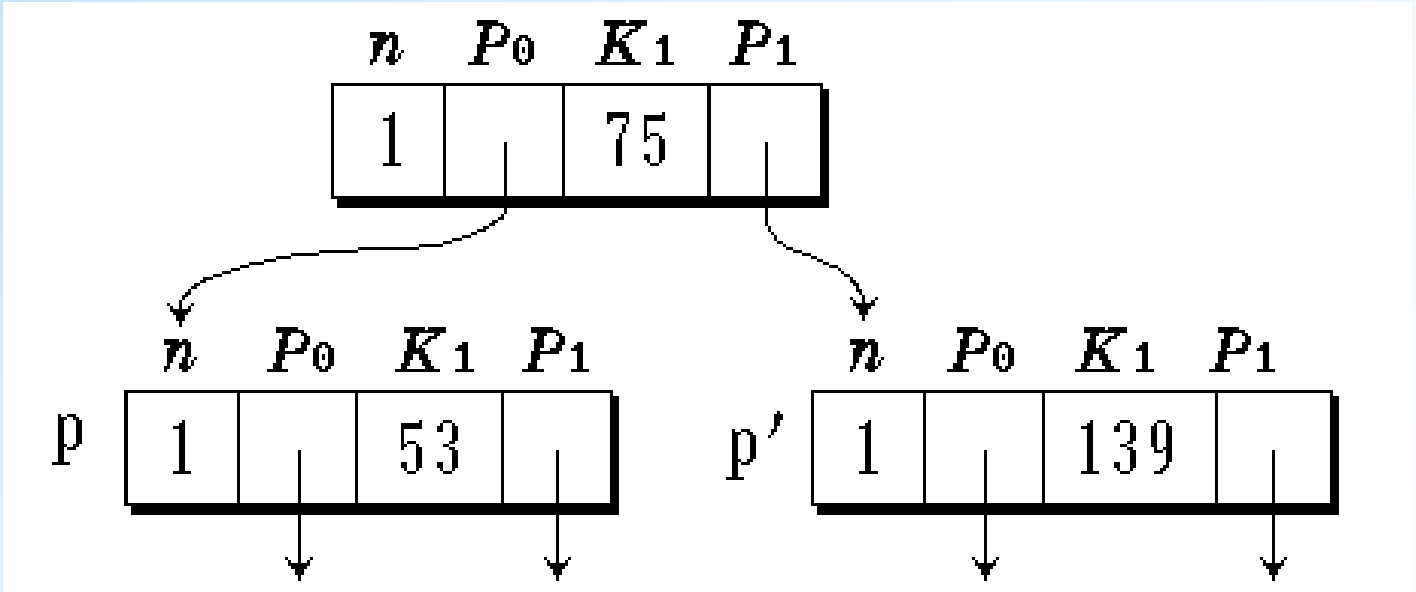
- B_树是从空树起，逐个插入关键字而生成的。
- 插入从某个叶子结点开始。如果在关键字插入后结点中的关键字个数超出了上界 m-1，则结点需要“分裂”，否则可以直接插入。
- 实现结点“分裂”的原则是：
  - 设结点 p 中已经有 m-1 个关键字，当再插入一个关键字后结点中的状态为($m, P_0, K_1, P_1, K_2, P_2, \ldots\ldots, K_m, P_m$)

    其中 $K_i < K_{i+1}, 1 \le i < m$
  - 把结点 p 分裂成两个结点 p 和 q，它们包含的信息分别为：
  - 结点 p：($\lceil m/2 \rceil$ -1, $P_0, K_1, P_1, \ldots\ldots, K_{\lceil m/2 \rceil -1}, P_{\lceil m/2 \rceil -1}$)
  - 结点 q：($m - \lceil m/2 \rceil, P_{\lceil m/2 \rceil}, K_{\lceil m/2 \rceil +1}, P_{\lceil m/2 \rceil +1}, \ldots\ldots, K_m, P_m$)
  - 位于中间的关键字 $K_{\lceil m/2 \rceil}$ 与指向新结点 q 的指针形成一个二元组 ($K_{\lceil m/2 \rceil}$, q)，插入到这两个结点的双亲结点中去。

3阶B_树非叶子结点最多有2个关键字（3棵子树），最少有一个关键字（2棵子树）

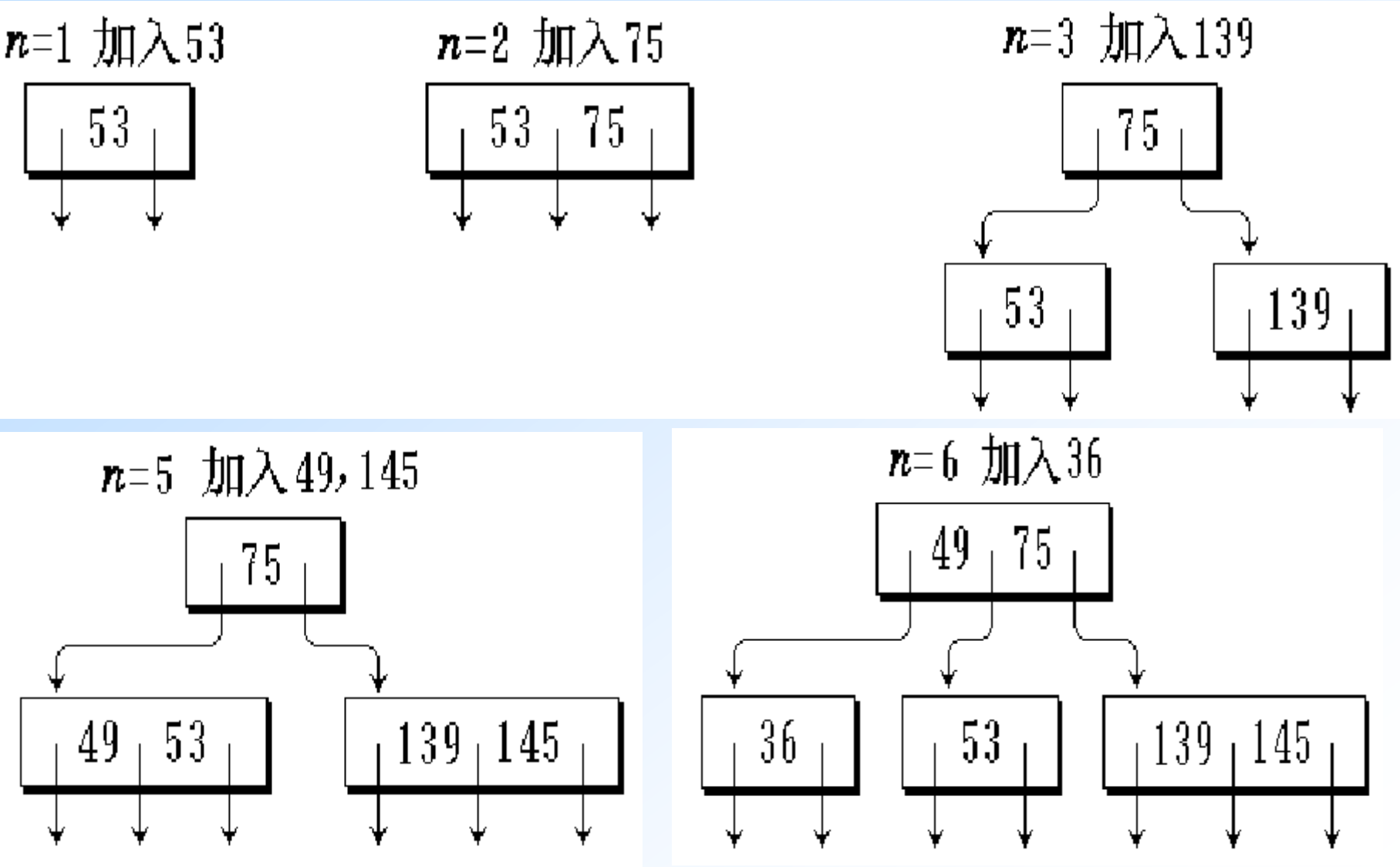


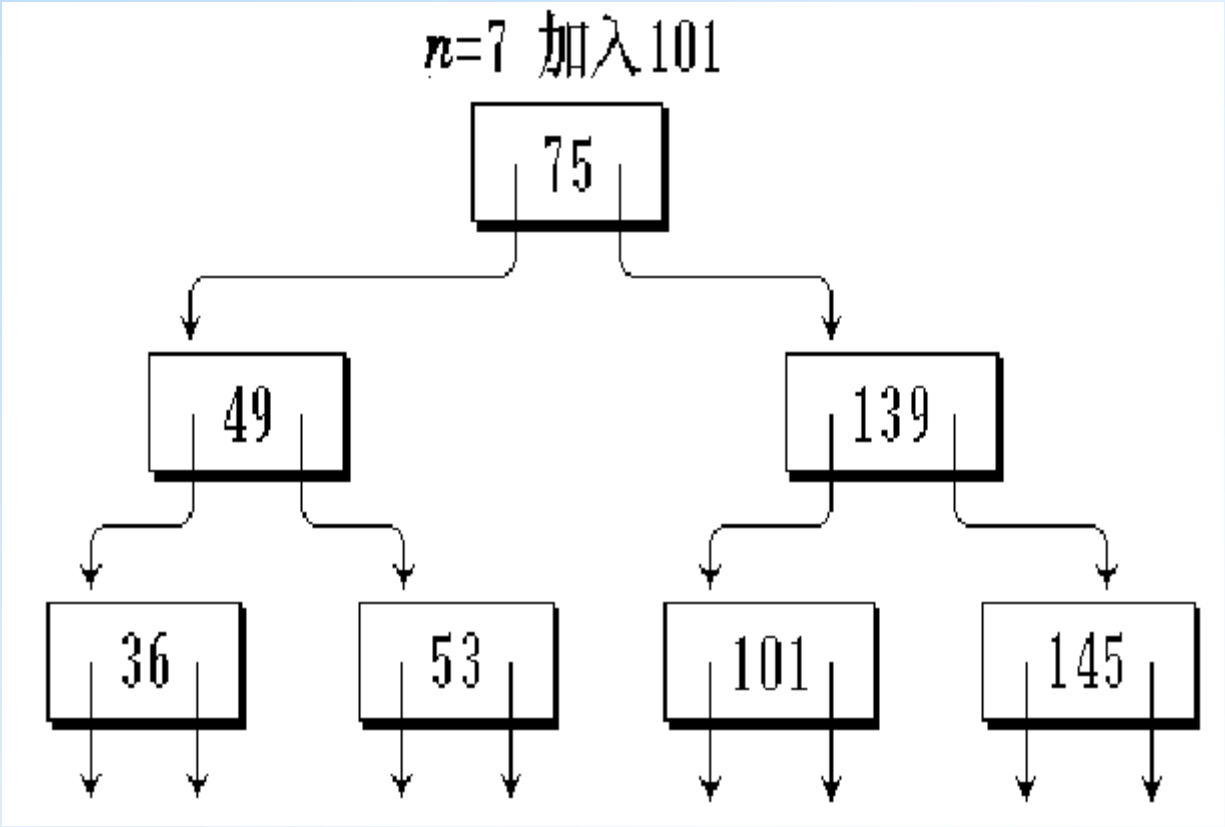




结点“分裂”的示例

示例：从空树开始逐个加入关键字建立3阶B_树





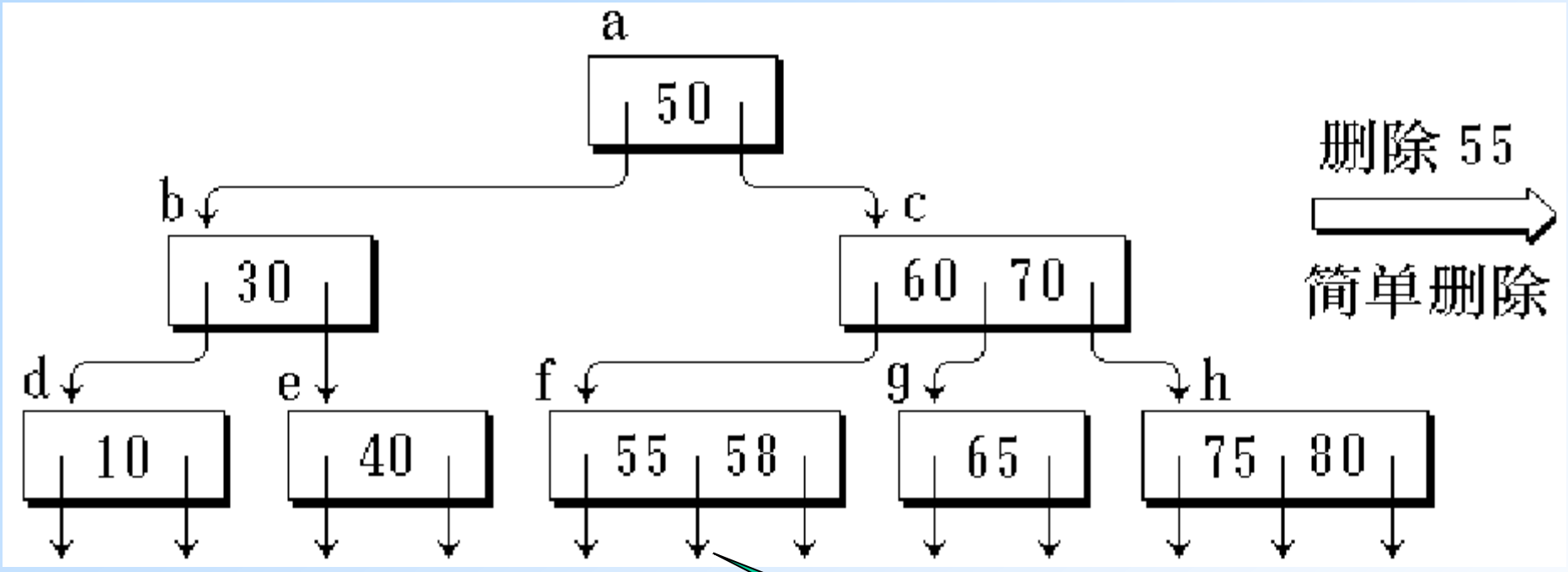
- **在插入新关键字时，需要自底向上分裂结点，最坏情况下从被插关键字所在叶结点到根的路径上的所有结点都要分裂。**

B_树的删除

- 在B_树上删除一个关键字时，首先需要找到这个关键字所在的结点，从中删去这个关键字。若该结点不是叶子结点，且被删关键字为 K_i，$1 \le i \le n$，则在删去该关键字之后，应以该结点 P_i 所指示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 所在的叶子结点中删除 x。
- 在叶子结点上的删除有 4 种情况。

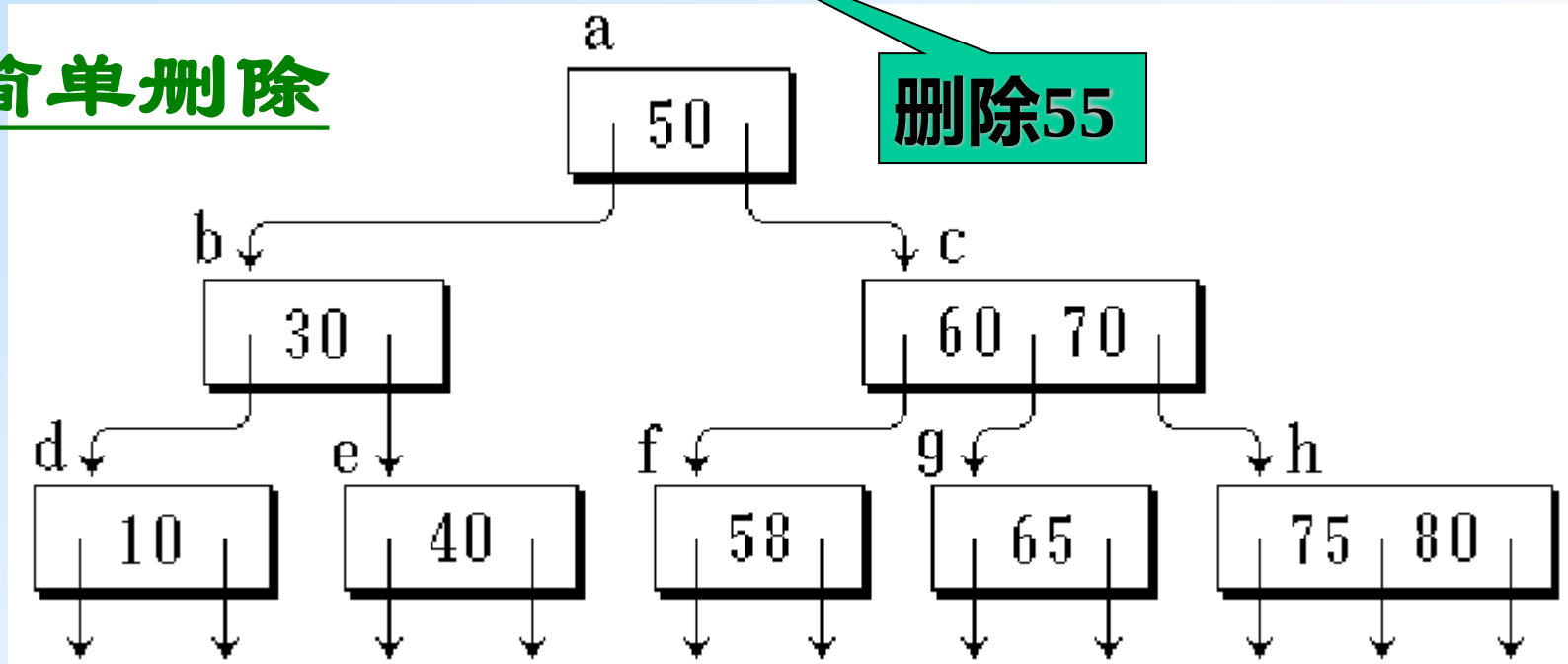
  1） 被删关键字所在叶子结点同时又是根结点且删除前该结点中关键字个数 $n \ge 2$，则直接删去该关键字。

  2）被删关键字所在叶子结点不是根结点且删除前该结点中关键字个数 $n \ge \lceil m/2 \rceil$，则直接删去该关键字。


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

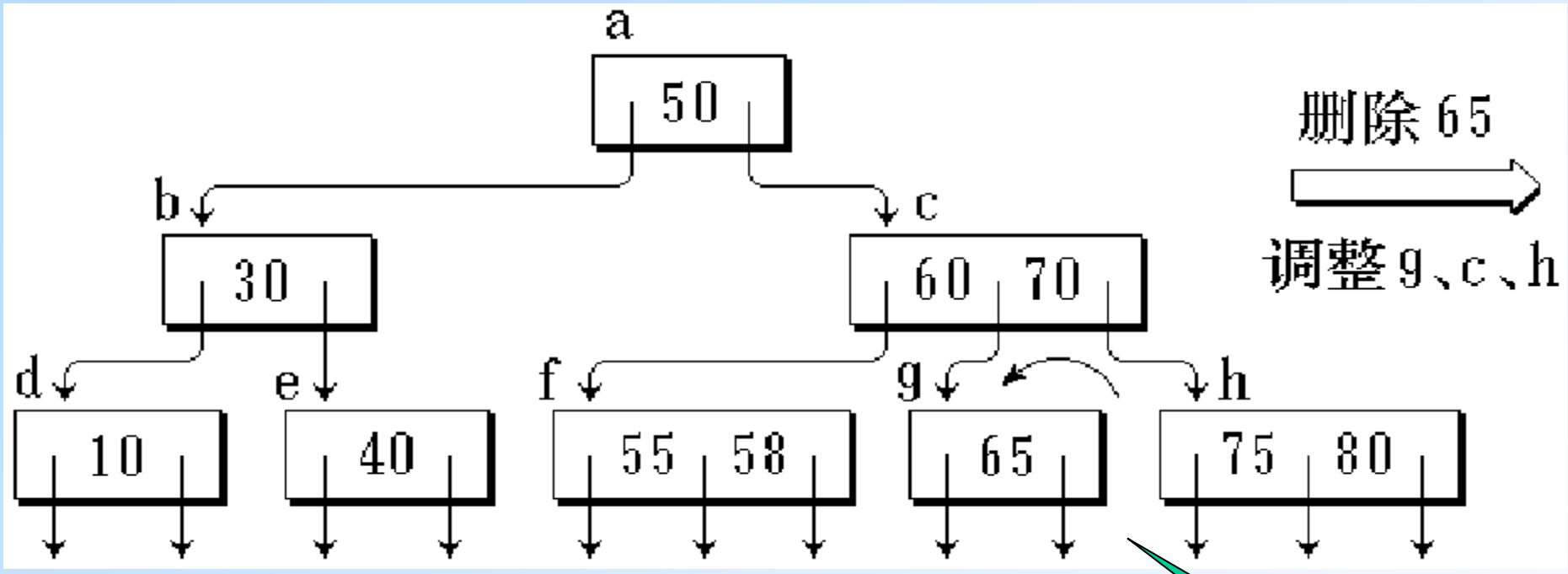
简单删除

删除55

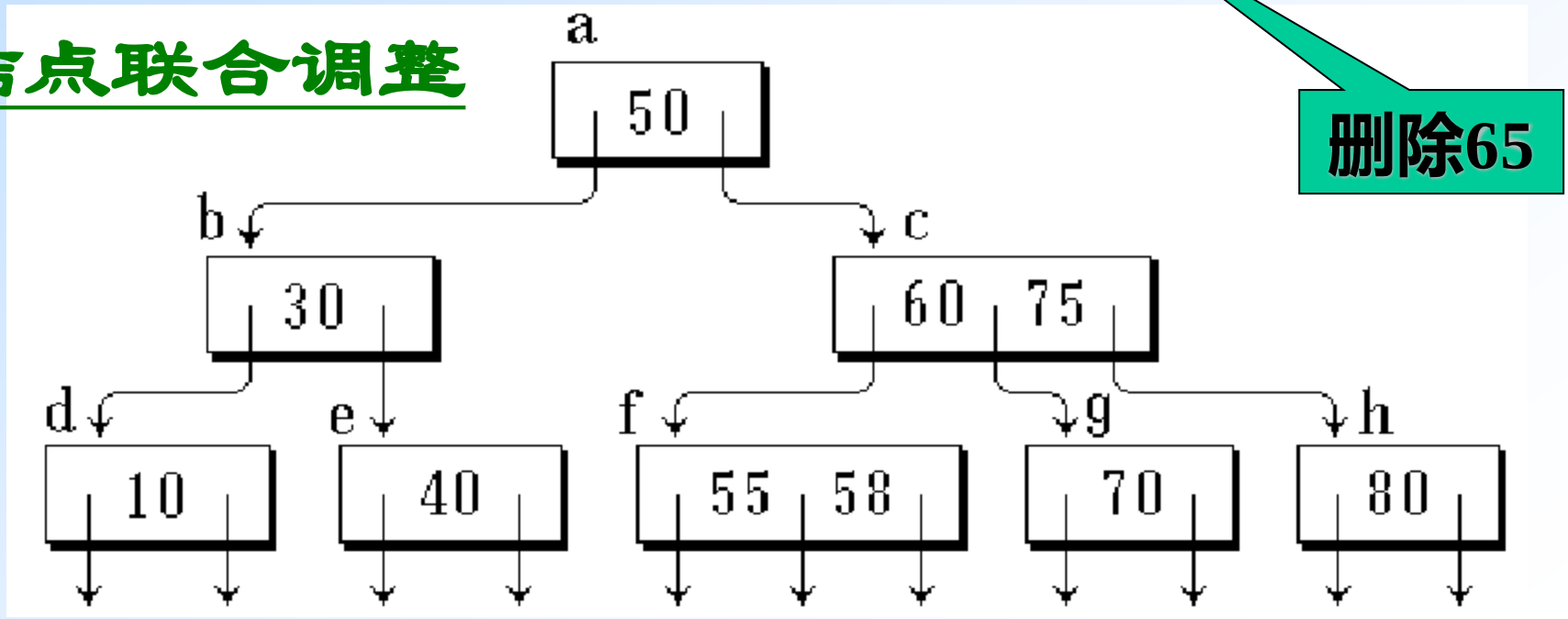

a
50
b
30
c
60 70
d
10
e
40
f
58
g
65
h
75 80

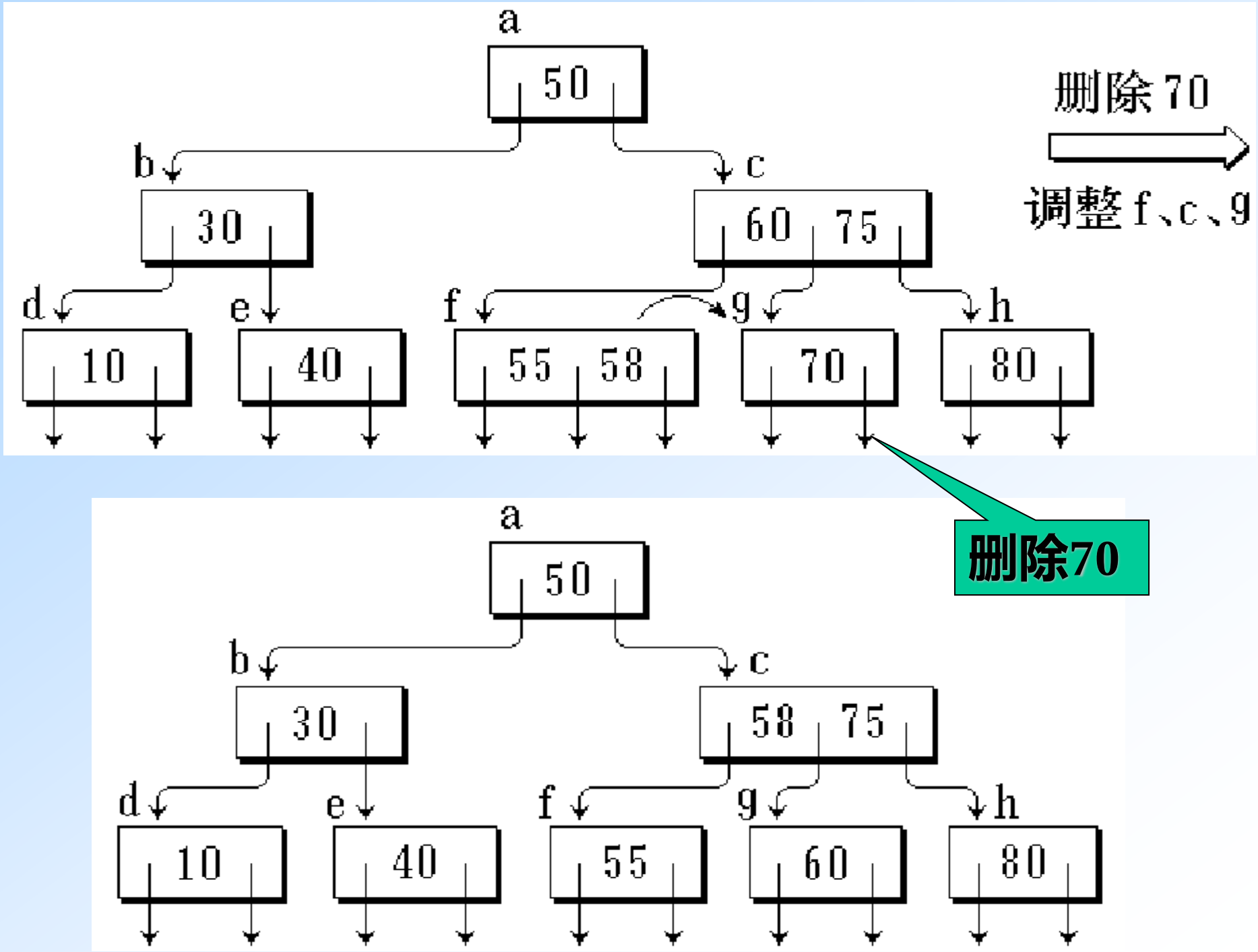
3）被删关键字所在叶子结点删除前关键字个数 $n=\lceil m/2\rceil-1$，若这时与该结点相邻的右兄弟 (或左兄弟) 结点的关键字个数 $n\geq\lceil m/2\rceil$，则可按以下步骤调整该结点、右兄弟 (或左兄弟) 结点以及其双亲结点，以达到新的平衡。

- 将双亲结点中刚刚大于 (或小于) 该被删关键字的关键字 K_i ($1\leq i\leq n$) 下移到被删除的关键字位置；
- 将右兄弟 (或左兄弟) 结点中的最小 (或最大)关键字上移到双亲结点的 K_i 位置；
- 将右兄弟 (或左兄弟) 结点中的最左 (或最右) 子树指针平移到被删关键字所在结点中最后 (或最前) 子树指针位置；
- 在右兄弟 (或左兄弟) 结点中，将被移走的关键字和指针位置用剩余的关键字和指针填补、调整。再将结点中的关键字个数减1。


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

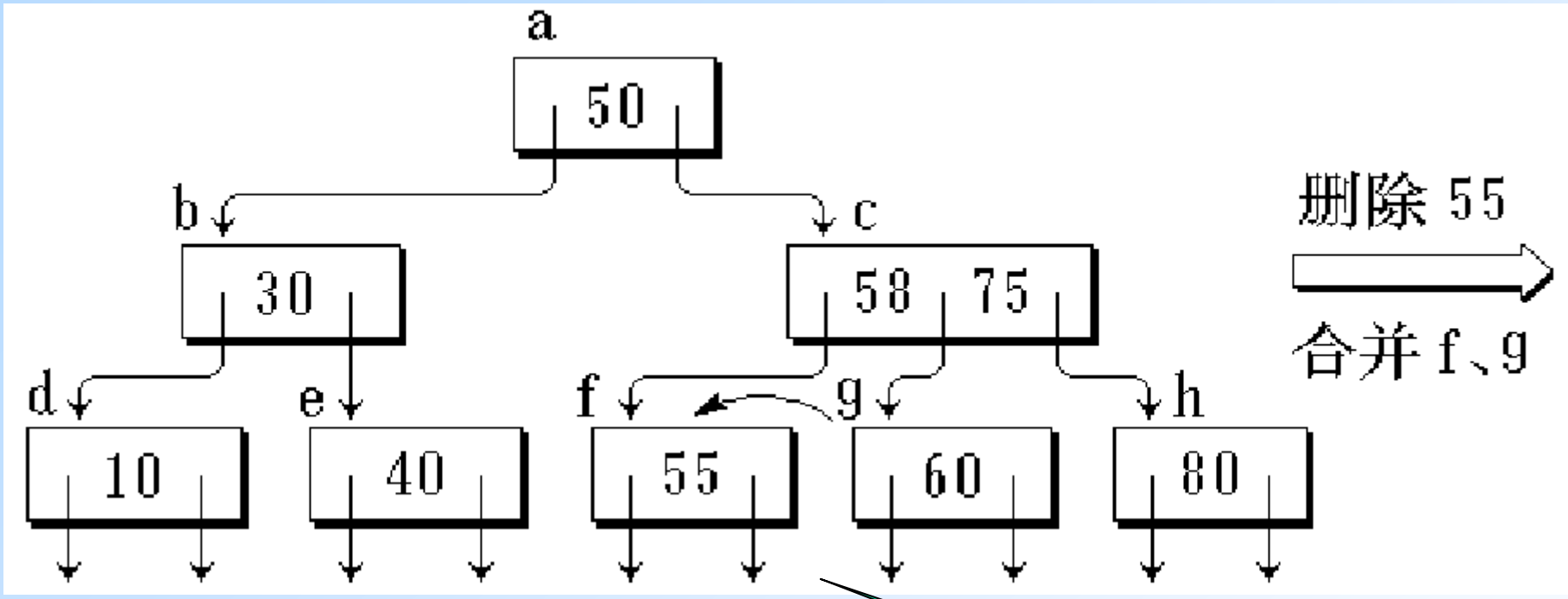
结点联合调整


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

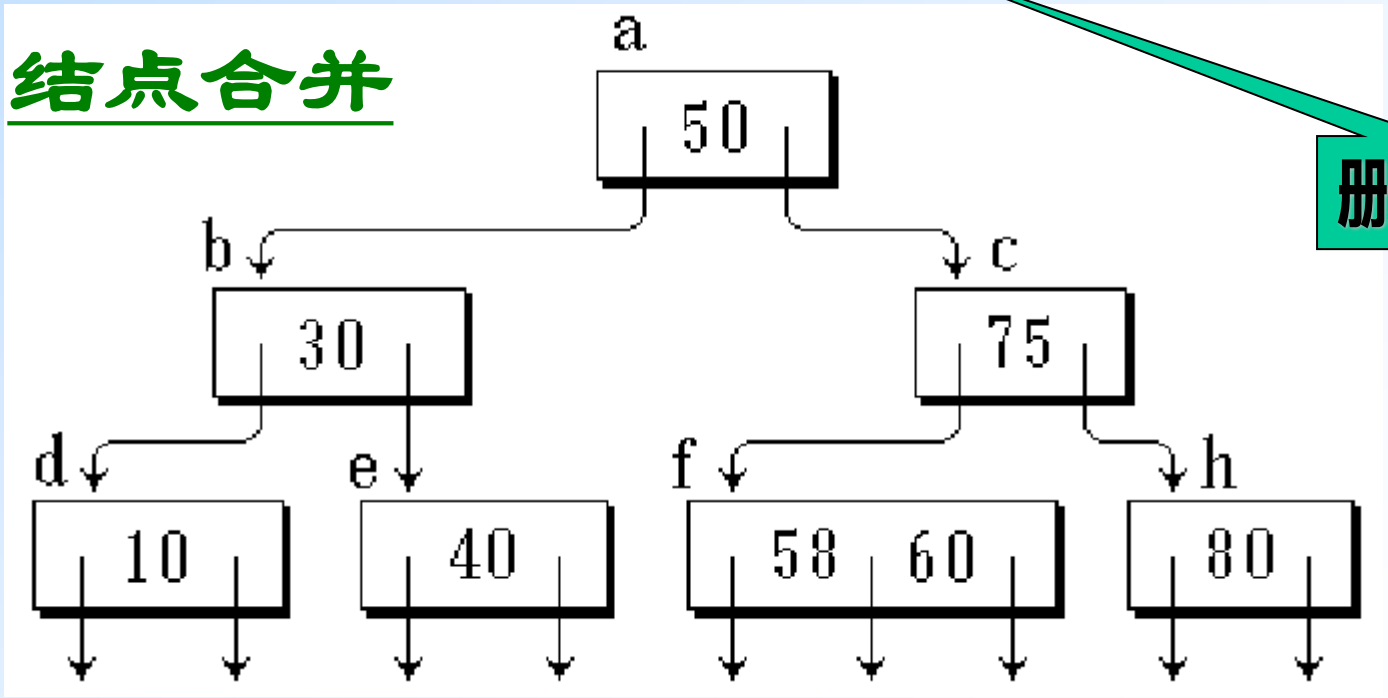

a
50
b
30
c
60 75
d
10
e
40
f
55 58
g
70
h
80
删除70
调整f、c、g
删除70
a
50
b
30
c
58 75
d
10
e
40
f
55
g
60
h
80

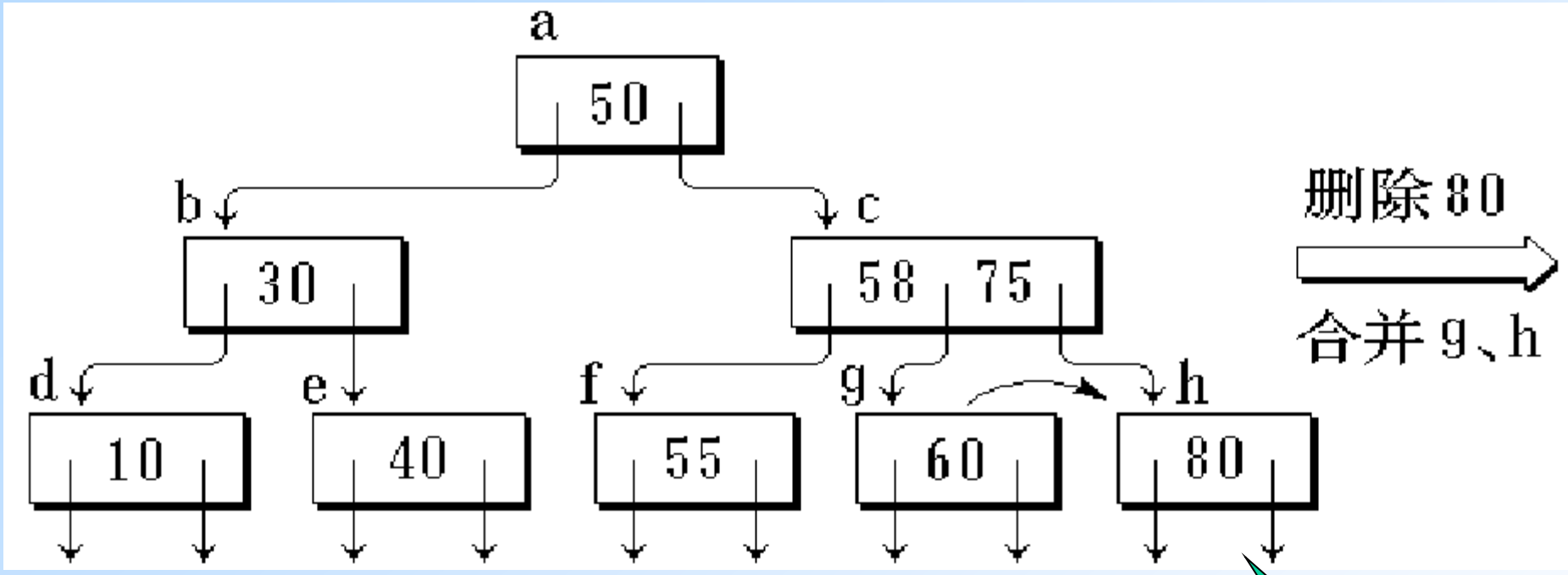
4）被删关键字所在叶子结点删除前关键字个数 $n=\lceil m/2\rceil-1$，若这时与该结点相邻的右兄弟 (或左兄弟) 结点的关键字个数 $n=\lceil m/2\rceil-1$，则必须按以下步骤合并这两个结点。

- 将双亲结点 p 中相应关键字下移到选定保留的结点中。若要合并 p 中的子树指针 P_i 与 P_{i+1} 所指的结点，且保留 P_i 所指结点，则把 p 中的关键字 K_{i+1} 下移到 P_i 所指的结点中。
- 把 p 中子树指针 P_{i+1} 所指结点中的全部指针和关键字都照搬到 P_i 所指结点的后面。删去 P_{i+1} 所指的结点。
- 在结点 p 中用后面剩余的关键字和指针填补关键字 K_{i+1} 和指针 P_{i+1}。
- 修改结点 p 和选定保留结点的关键字个数。

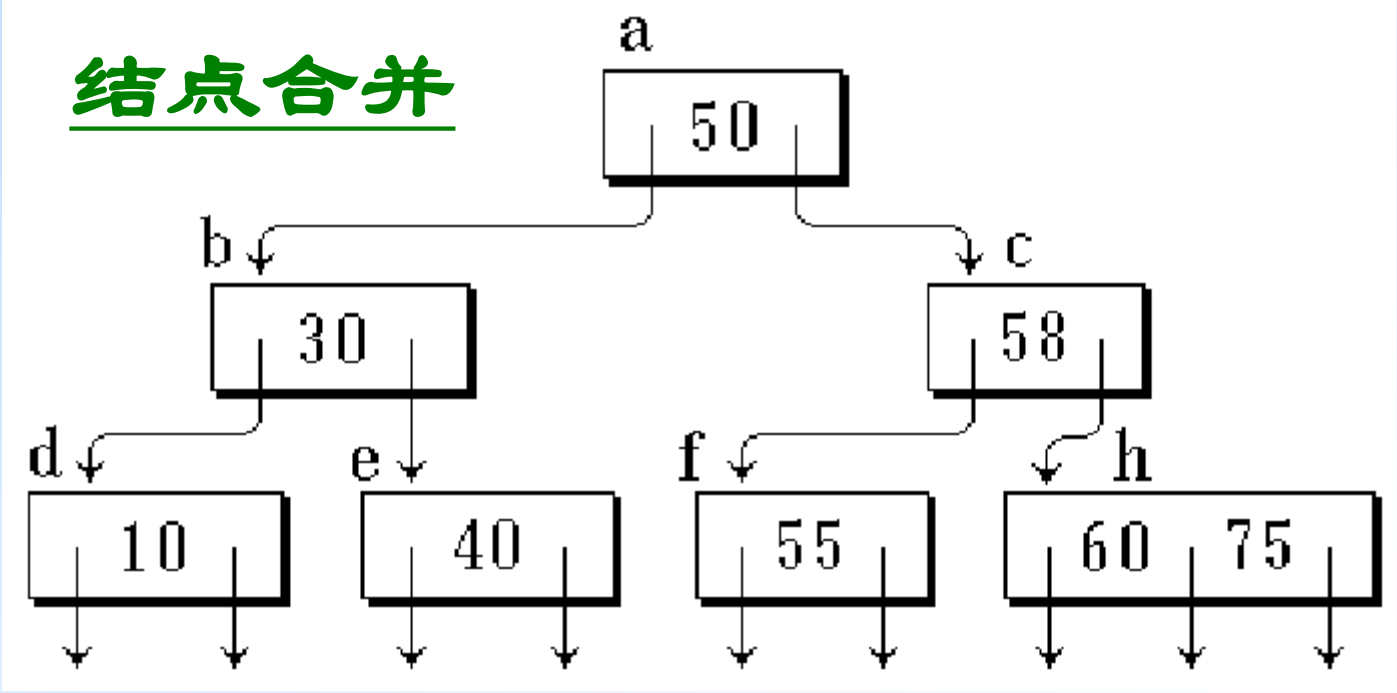

a
50
b
30
c
58 75
d
10
e
40
f
55
g
60
h
80
删除 55
合并 f、g

结点合并

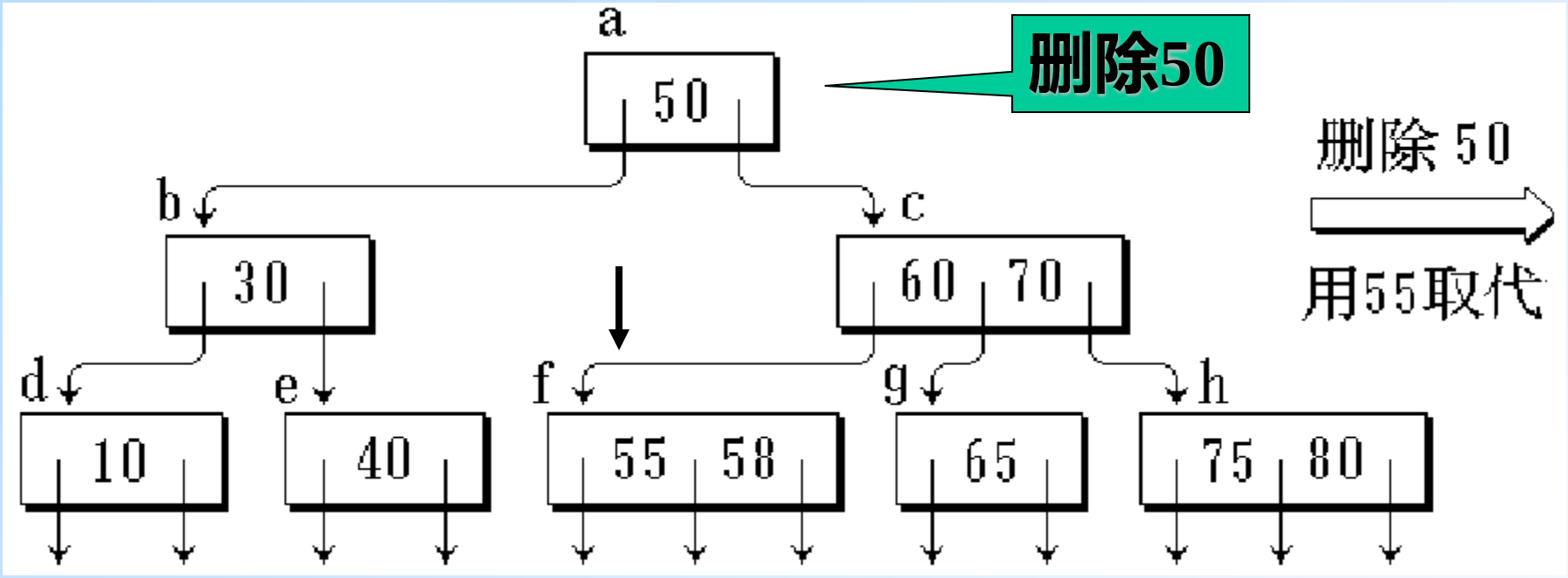

a
50
b
30
c
75
d
10
e
40
f
58 60
h
80
删除55

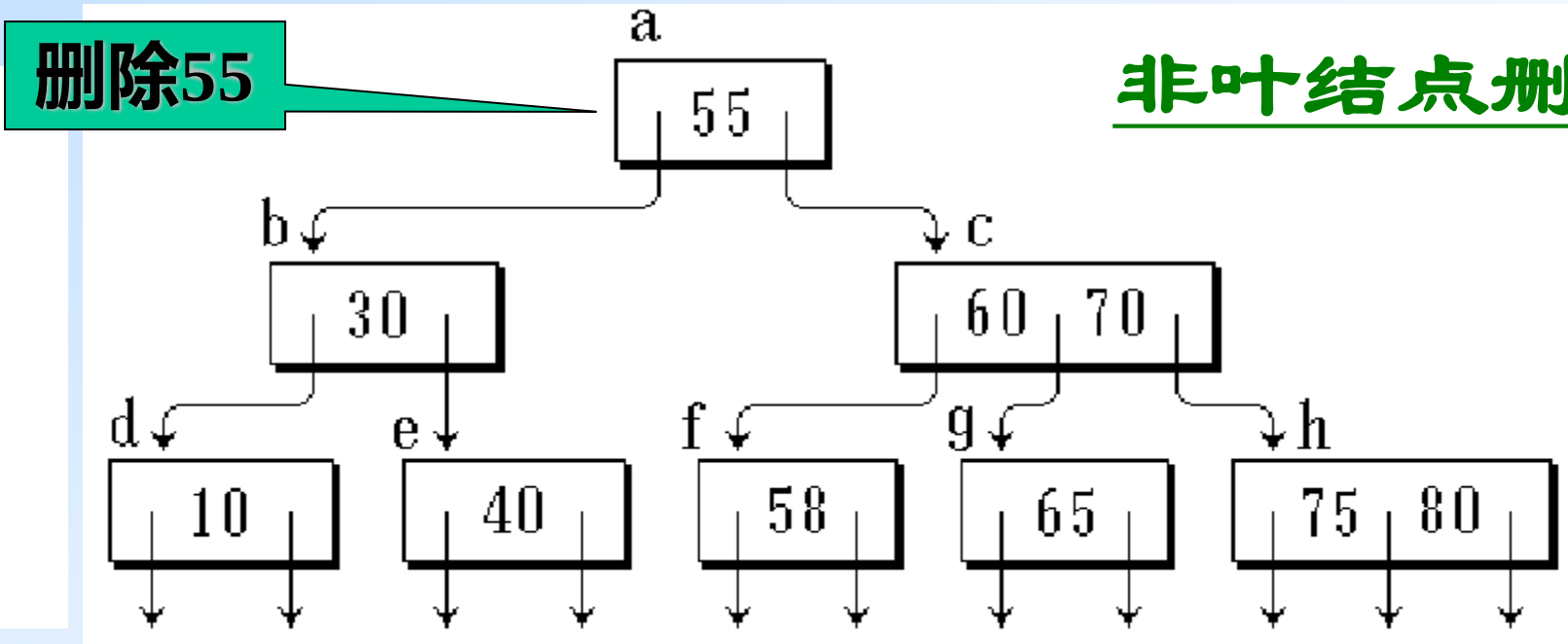

a
50
b
30
c
58 75
d
10
e
40
f
55
g
60
h
80
删除80
合并g、h

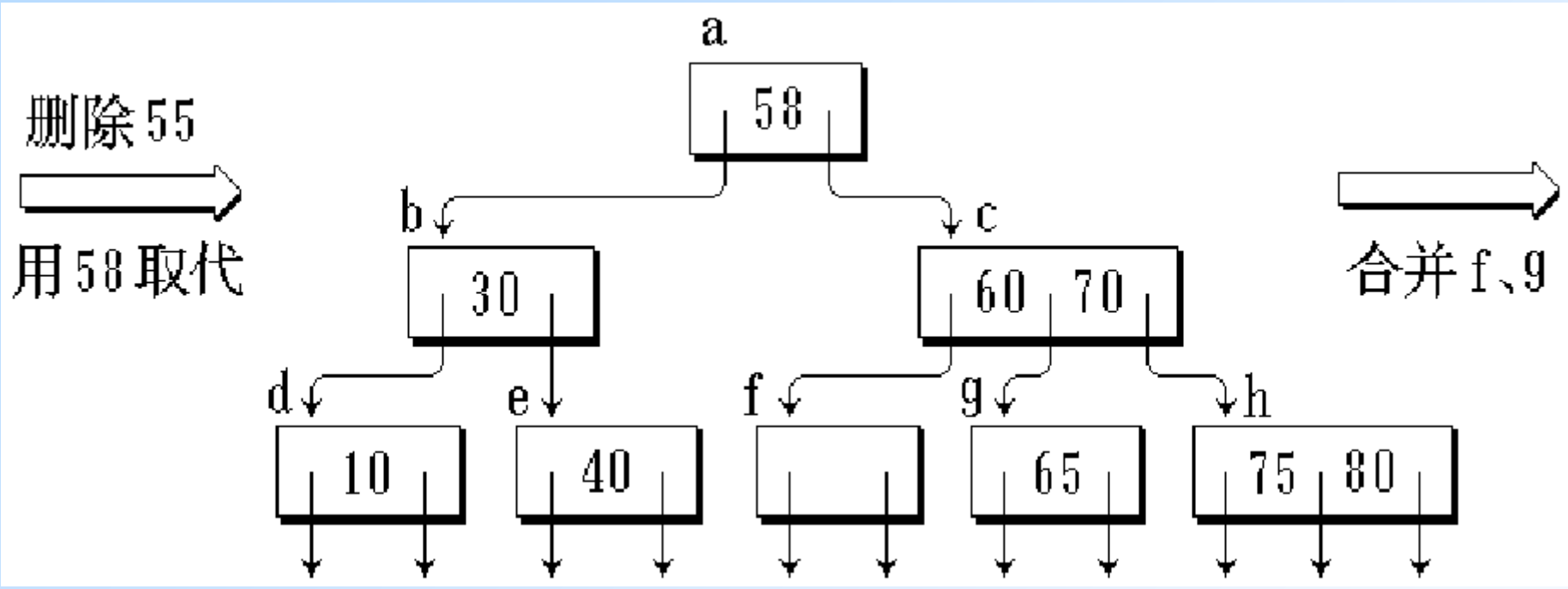
结点合并


a
50
b
30
c
58
d
10
e
40
f
55
h
60 75

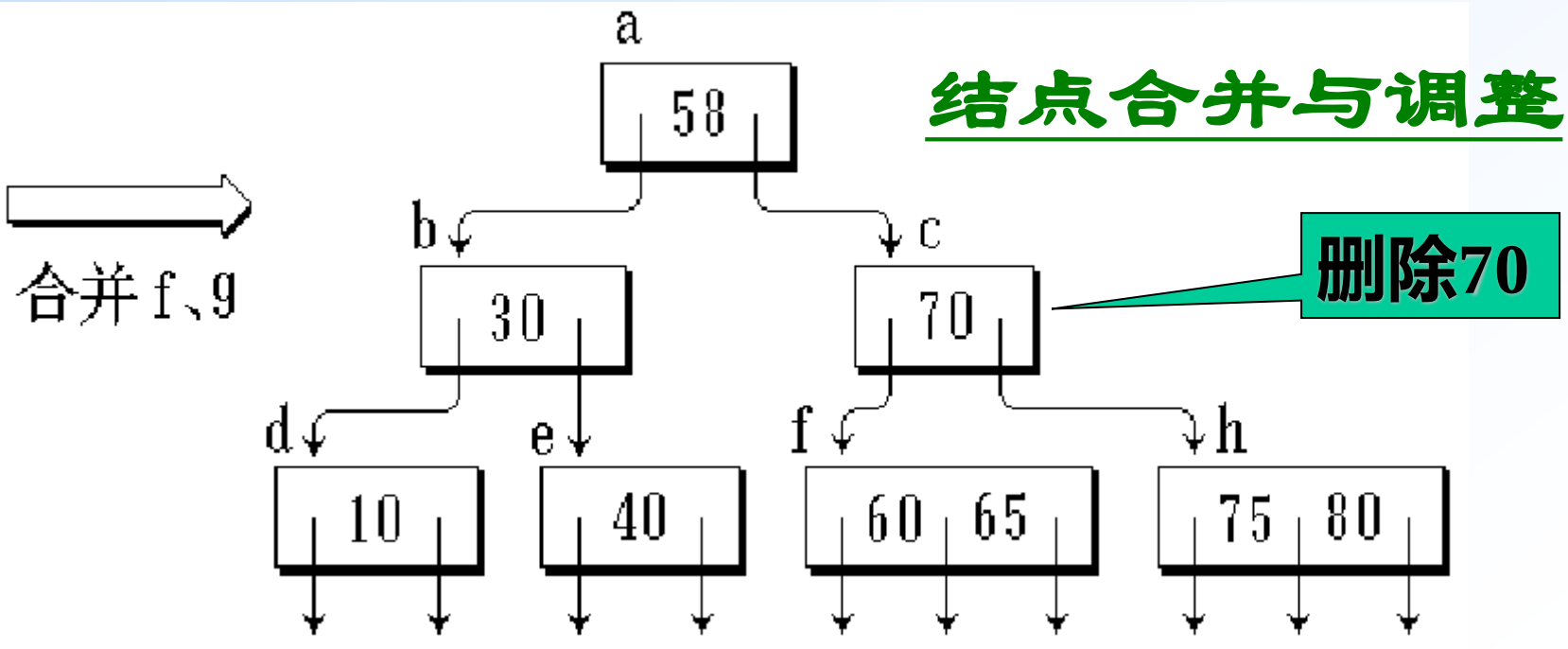
删除80

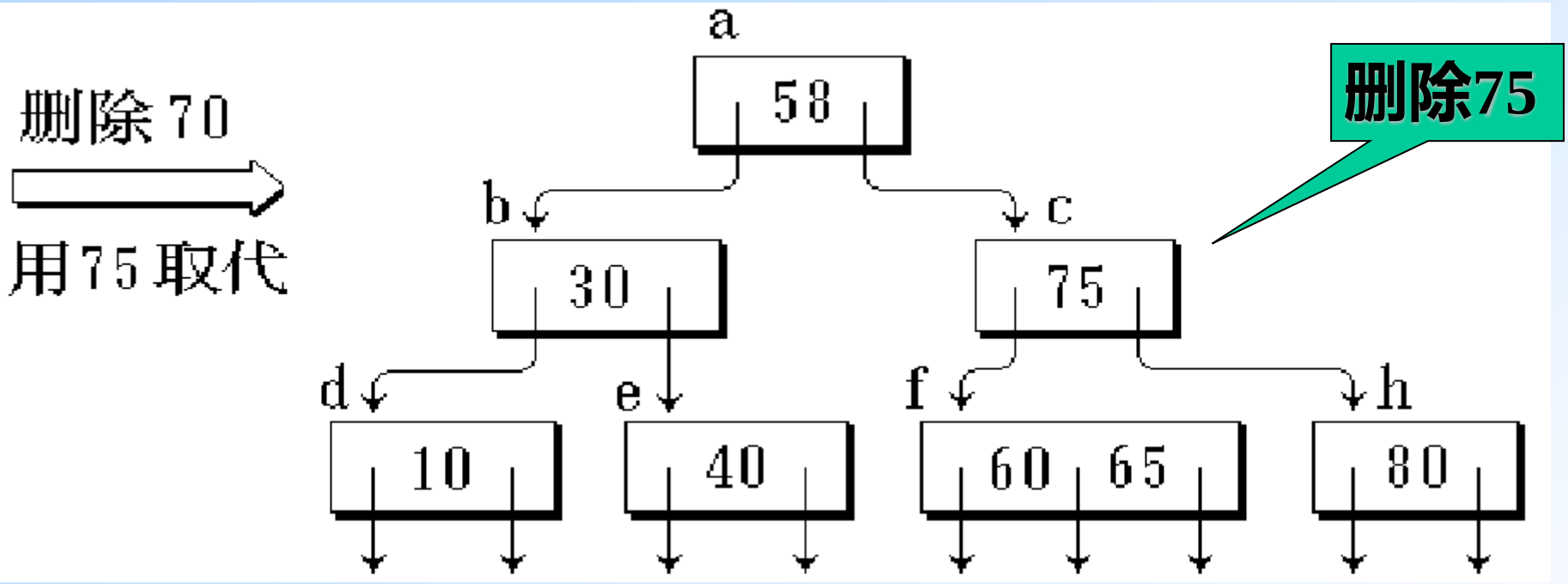

a
50
删除50
删除 50
用55取代
b
30
c
60 70
d
10
e
40
f
55 58
g
65
h
75 80

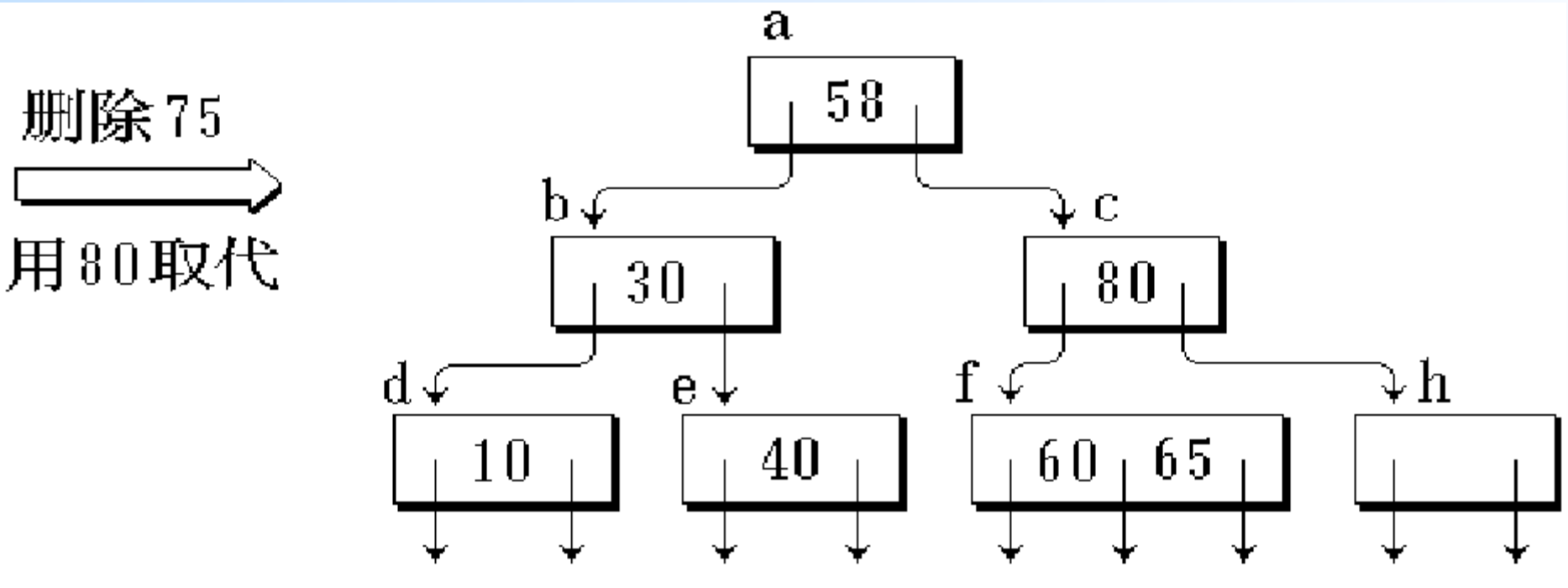

删除55
a
55
非叶结点删除
b
30
c
60 70
d
10
e
40
f
58
g
65
h
75 80

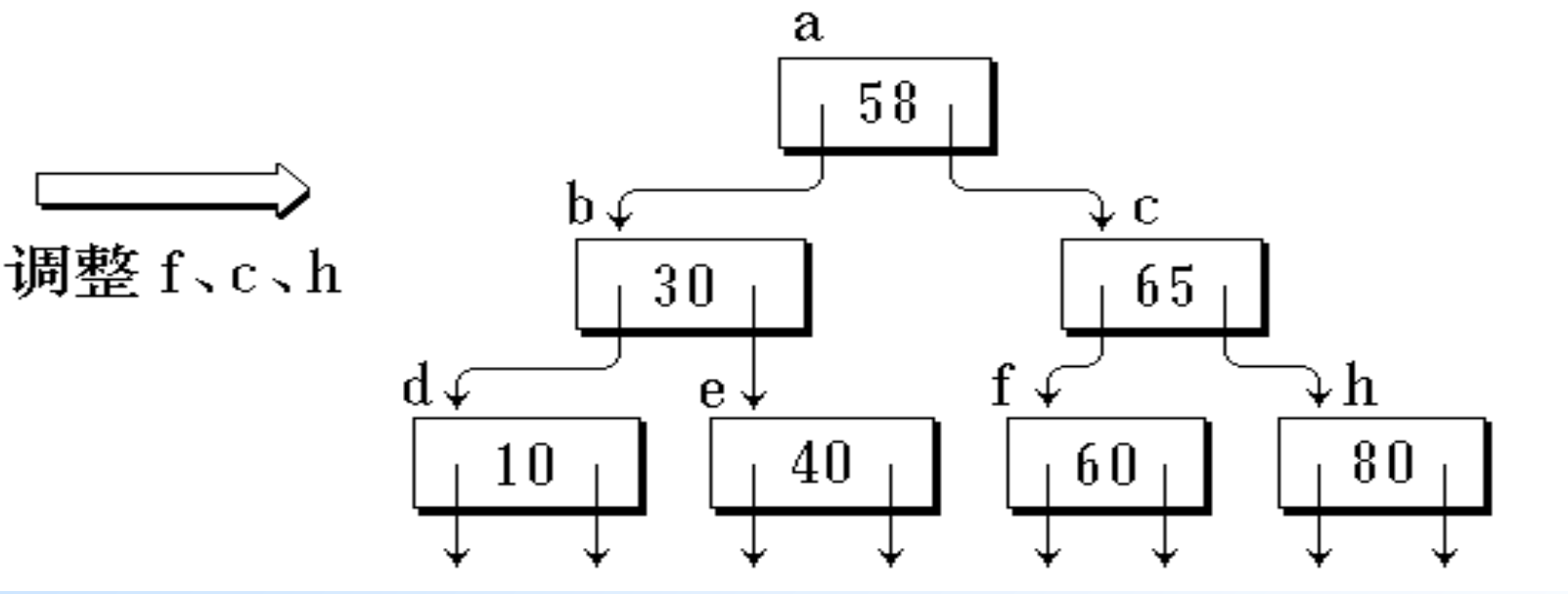

删除55
用58取代
a
58
b
30
c
60 70
d
10
e
40
f
g
65
h
75 80
合并f、g

结点合并与调整


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


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

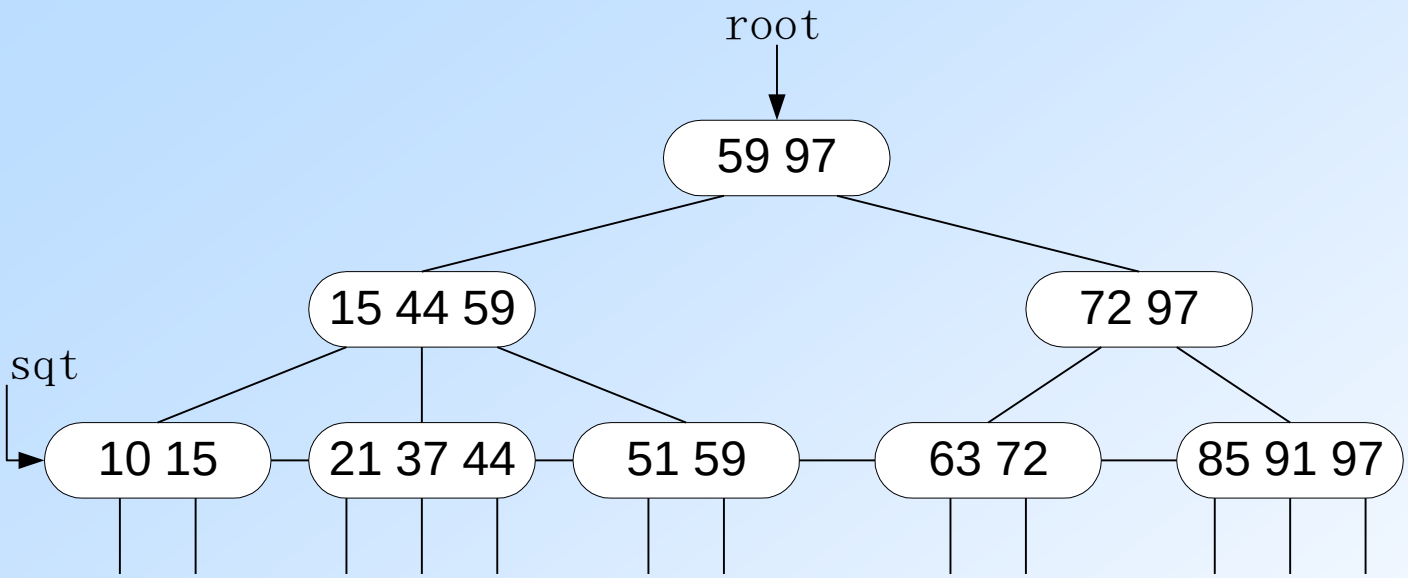

删除75
用80取代
a
58
b
30
c
80
d
10
e
40
f
60
65
h



- 在合并结点的过程中，双亲结点中的关键字个数减少了。
- 若双亲结点是根结点且结点关键字个数减到 0，则该双亲结点应从树上删去，合并后保留的结点成为新的根结点；否则将双亲结点与合并后保留的结点都写回磁盘，删除处理结束。
- 若双亲结点不是根结点，且关键字个数减到$\lceil m/2 \rceil - 2$，又要与它自己的兄弟结点合并，重复上面的合并步骤。最坏情况下这种结点合并处理要自下向上直到根结点。

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。
- 一棵m阶B+树与m阶B_树的主要差别如下：
  - 有n棵子树的结点含有n个关键字；
  - 所有的叶子结点都处于同一层次上，包含了全部关键字及指向相应数据元素存放地址的指针，且叶子结点本身按关键字从小到大顺序链接；
  - 所有的非叶子结点可以看成是索引部分，结点中关键字 K_i 与指向子树的指针 P_i 构成对子树 (即下一层索引块) 的索引项 (K_i, P_i)，K_i 是子树中最大的关键字。



- 在B+树中有两个头指针：一个指向B+树的根结点，一个指向关键字最小的叶子结点。可对B+树进行两种查找运算：一种是循叶子结点链的顺序查找，另一种是从根结点开始的随机查找。
- 在B+树上进行随机查找、插入和删除的过程基本上与B_树类似。只是在查找过程中，如果非叶子结点上的关键字等于给定值，查找并不停止，而是继续沿右指针向下，一直查到叶子结点上的这个关键字。B+树的查找分析类似于B_树。

下列叙述中，不符合m阶B-树定义的是

A）根节点最多有m棵子树　　D

B）所有叶子节点都在同一层上

C) 各节点内关键字均升序或降序排列

D）叶子节点之间通过指针链接

哈 希 表

哈希表的相关定义
哈希函数的构造方法
处理冲突的方法
哈希表的查找
哈希表的插入
哈希查找分析

哈希表的相关定义

哈希查找

又叫散列查找，利用哈希函数进行查找的过程。

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。

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。哈希函数是一种映象，是从关键字空间到存储地址空间的一种映象。

可写成，addr(ai)=H(k)

其中：　ai是表中的一个元素

addr(ai)是ai的存储地址

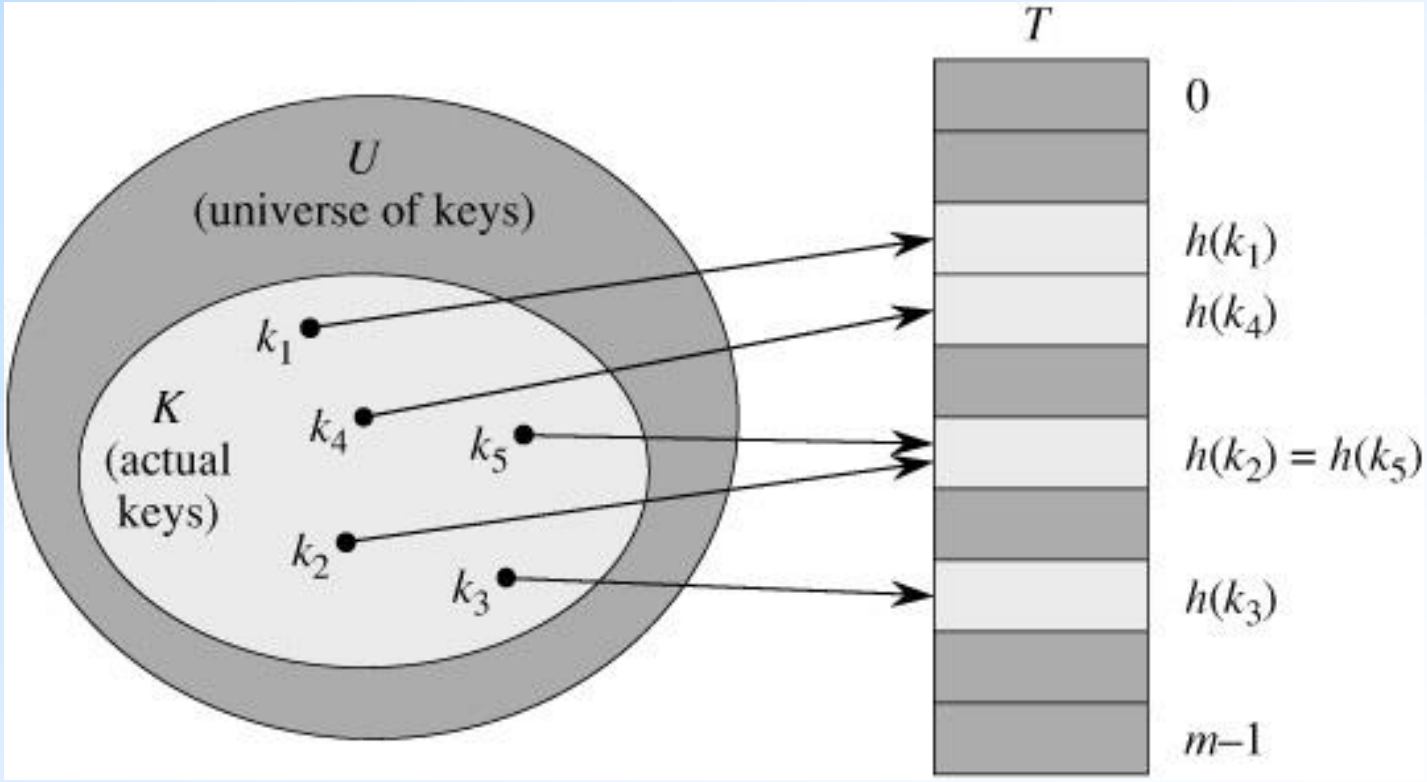
k是ai的关键字

哈希表

根据设定的哈希函数 H(key) 和所选中的处理冲突的方法，将一组关键字映象到一个有限的、地址连续的地址集 (区间) 上，并以关键字在地址集中的“象”作为相应记录在表中的存储位置，如此构造所得的查找表称之为“哈希表”。

❖哈希表(Hash Tables)

给定关键字，期望通过计算，即利用哈希函数$h(x)$计算出关键字在表T中的位置。



例如：对于如下 9 个关键字：

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

设哈希函数 f (key) = ⌊(Ord(关键字首字母) - Ord('A') + 1) / 2⌋

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

Zhou

问题：若添加关键字 Zhou，会出现什么情况？

从这个例子可见：

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

2) 由于哈希函数是一个压缩映像，因此，在一般情况下，很容易产生“冲突”现象，即：key1 ≠ key2，而 f (key1) = f (key2)。这种具有相同函数值的关键字称为同义词。

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。

冲突：$key1 \neq key2$，

但$H(key1)=H(key2)$的现象

哈希函数构造的方法

- 直接定址法
- 数字分析法
- 平方取中法
- 折叠法
- 除留余数法
- 随机数法

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

其中a和b为常数

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 (u_1, u_2, …, u_s)，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 ③ ④⑤⑥ 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

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。

例：2589的平方值为6702921，可以取中间的029为地址。

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。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

间界叠加: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。

例　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

除留余数法

设定哈希函数为:

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

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

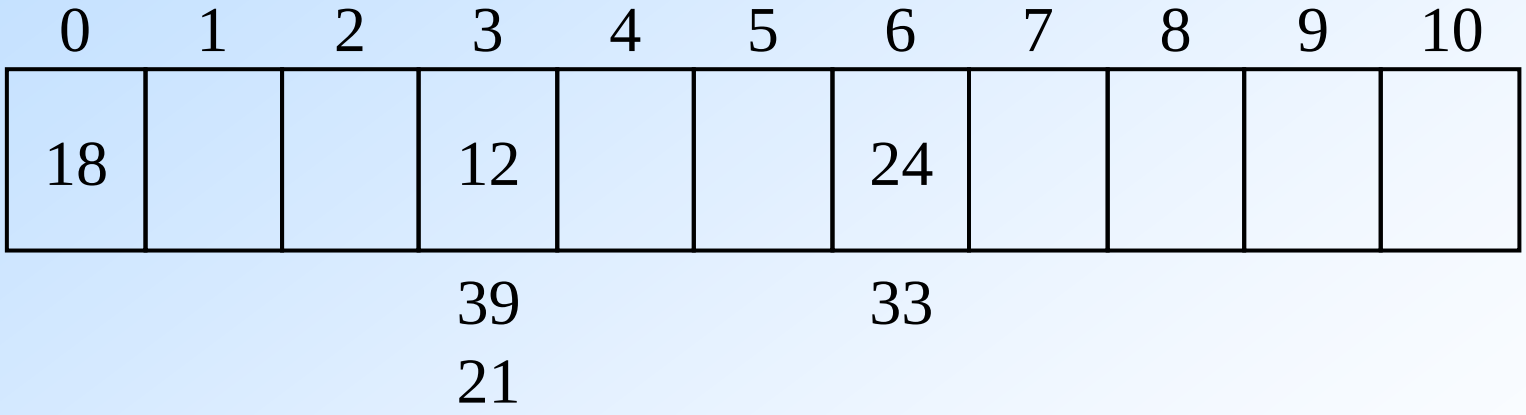
❖除留余数法(最常用)

给定一组关键字：12, 39, 18, 24, 33, 21，若取 $p = 11$, 则他们对应的哈希函数值将为

0	1	2	3	4	5	6	7	8	9	10
33	12	24				39	18			21

❖除留余数法(最常用)

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



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。

随机数法

设定哈希函数为:

H(key) = Random(key)

其中，Random 为伪随机函数

此法用于对长度不等的关键字构造哈希函数。

选取哈希函数考虑的因素:

计算哈希函数所需时间

关键字长度

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

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。

- 开放定址法
- 再哈希法
- 链地址法

开放定址法

为产生冲突的地址 H(key) 求得一个地址序列：

$H_0, H_1, H_2, \ldots, H_s \quad 1 \le s \le m-1$

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

其中： $i=1, 2, \ldots, s$

H(key)为哈希函数;m为哈希表长;

d_i为增量序列,有下列三种取法:

对增量 d_i 的三种取法：

1) 线性探测再散列

$d_i = c \times i$ 最简单的情况 $c=1$

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)

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 $s(m-1)$个 H_i 值能覆盖哈希表中所有地址。则要求：

※ 平方探测时的表长 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。

例 表长为11的哈希表中已填有关键字为17，60，29的记录，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

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

(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
H1=(5+1^2) MOD 11=6 冲突
H2=(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：
H1=(5+9) MOD 11=3 不冲突

例如：给定关键字集合构造哈希表

{ 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

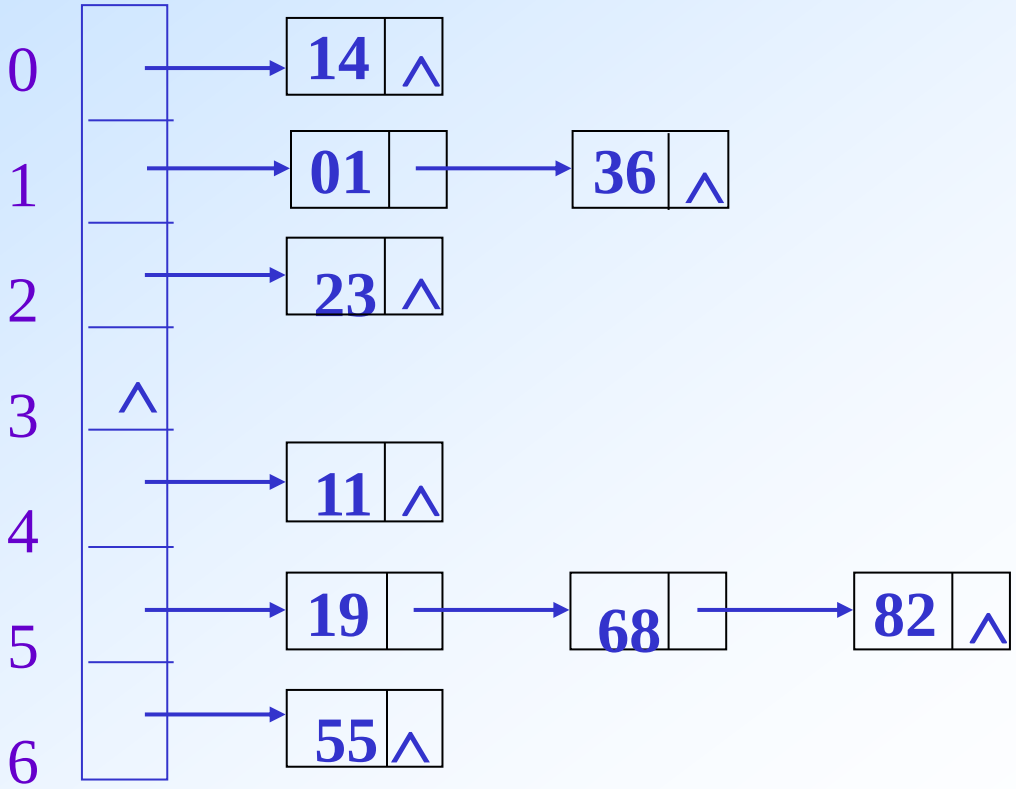
0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	11	82	68	36	19		

链地址法

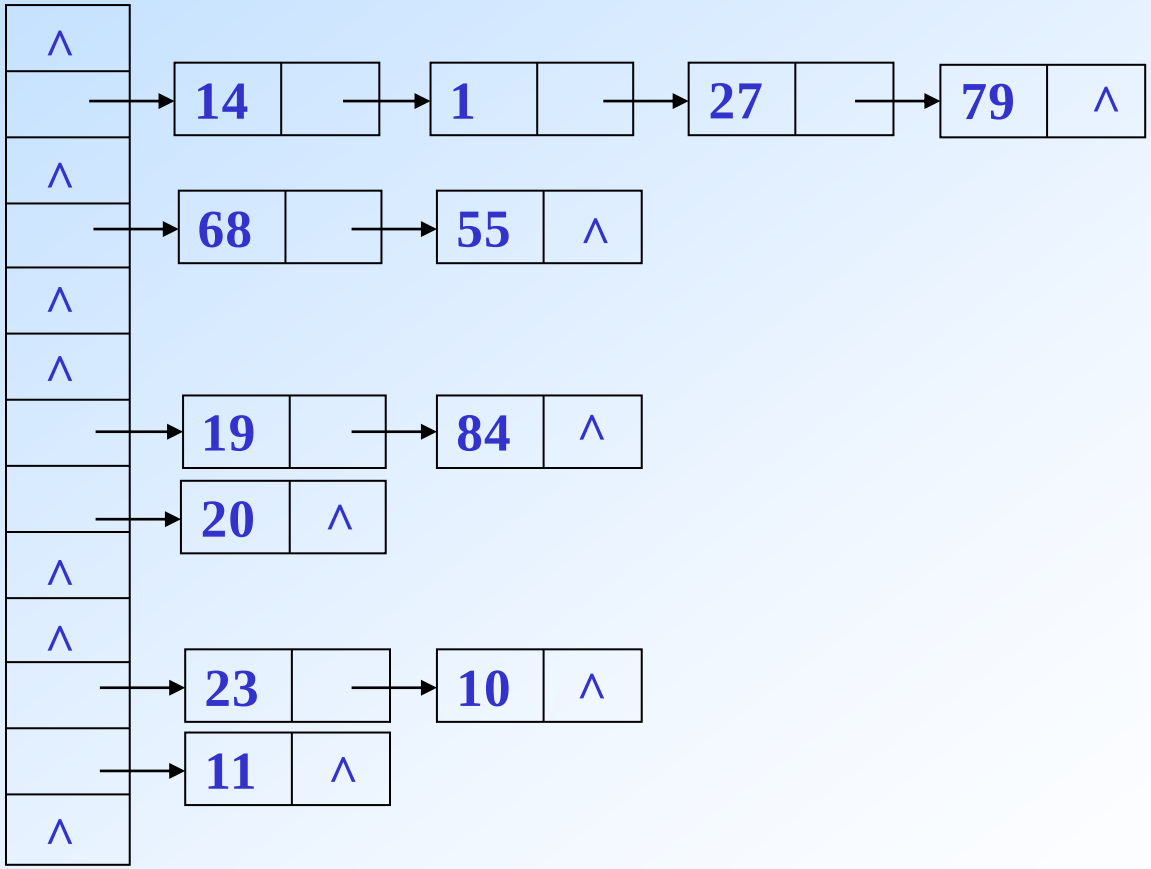
将所有哈希地址相同的记录都链接在同一链表中。

例:给定关键字{ 19, 01, 23, 14, 55, 68, 11, 82,36 }

哈希函数为 H(key)=key MOD 7



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



再哈希法

方法：构造若干个哈希函数，当发生冲突时，计算下一个哈希地址，

直到冲突不再发生。

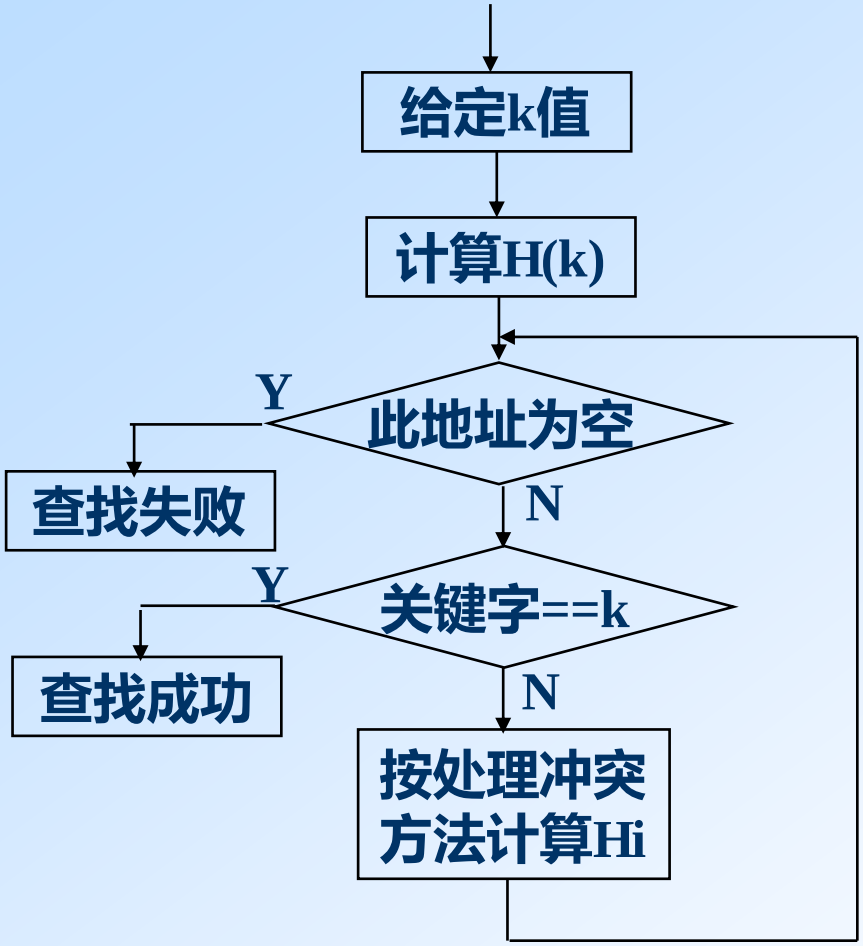
即：$H_i=Rh_i(key)$　　$i=1,2,\ldots\ldots k$

其中：Rh_i——不同的哈希函数

特点：计算时间增加

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。

开放定址哈希表的存储结构

```
int  hashsize[] = { 997, ... };
typedef  struct {
  ElemType   *elem;
  int  count;                  // 当前数据元素个数
  int  sizeindex;
                  // hashsize[sizeindex]为当前容量
} HashTable;
#define  SUCCESS  1
#define  UNSUCCESS  0
#define  DUPLICATE  -1
```

查找算法

```
Status SearchHash (HashTable H, KeyType K,
                                  int &p, int &c) {
  // 在开放定址哈希表H中查找关键码为K的记录
      p = Hash(K);        // 求得哈希地址
      while ( H.elem[p].key != NULLKEY &&
                          !EQ(K, H.elem[p].key))
          collision(p, ++c);       // 求得下一探查地址 p
      if (EQ(K, H.elem[p].key)) return SUCCESS;
       // 查找成功，返回待查数据元素位置 p
      else return UNSUCCESS;    // 查找不成功
} // SearchHash
```

哈希表的插入

```
Status InsertHash (HashTable &H, Elemtype e){
if ( HashSearch ( H, e.key, p, c ) == SUCCESS )
    return DUPLICATE;
                        // 表中已有与 e 有相同关键字的元素
elseif ( c < hashsize[H.sizeindex]/2 ) {
                // 冲突次数 c 未达到上限，（阀值 c 可调）
    H.elem[p] = e;  ++H.count;  return OK;
                        // 查找不成功时，返回 p为插入位置
}
else  RecreateHashTable(H);        // 重建哈希表
 } // InsertHash
```

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, 哈希表长为m=16， 用线性探测再散列处理冲突得到哈希表。

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

给定值K=84的查找过程为:

H(84)=6 不空且不等于84, 冲突

H1=(6+1)MOD16=7不空且不等于84, 冲突

H2=(6+2)MOD16=8不空且等于84, 查找成功, 返回记录在表中的序号。

给定值K=38的查找过程为:

H(38)=12 不空且不等于38,冲突

H1=(12+1)MOD16=13空记录

表中不存在关键字等于38 的记录,查找不成功。

哈希表查找的分析

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。

决定哈希表查找的ASL的因素：

1) 选用的哈希函数;

2) 选用的处理冲突的方法;

3) 哈希表饱和的程度，装载因子 $\alpha=n/m$ 值的大小（n—表中填入的记录数，m—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。

因此，哈希表的ASL是处理冲突方法和装载因子的函数。

例如：前述例子

线性探测处理冲突时，　ASL = 22/9

链地址法处理冲突时，　ASL = 13/9

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

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

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。

思考题：已知带头结点的单链表, L为表头指针。试设计一个算法，删除单链表中所有重复的结点（对于多个重复结点，只保留第一个）。

遍历单链表的过程中，把遍历的值存储到一个哈希表中。若当前访问的值在哈希表中已经存在，则说明这个数据是重复的，就可以删除。时间复杂度O(n)。

抽象数据类型“映射”:ADT Map

- ADT Map的结构是键-值关联的无序集合。
- 关键码key具有唯一性，通过关键码可以唯一确定一个数据值。
- 使用字典的优势在于，给定关键码key，能够很快得到关联的数据值data (value) 。
- 为了达到快速查找的目标，需要一个支持高效查找的ADT实现。

- 可以采用列表数据结构加顺序查找或者二分查找。
- c++STL 中，map一般采用红黑树(RB Tree)实现。

◆更为合适的是使用散列表来实现，这样查找可以达到最快O(1)的性能。 **Python最有用的内置数据类型之一是“字典Dictionary”, 可以保存(key-data)键值对的数据类型**

c++STL 中，map一般采用红黑树(RB Tree)实现。100万条记录，最多只要20次的比较。最坏的情况下其复杂度也不会超过O(logN)。

hash_map采用hash表存储，最好的情况是O(1)，有可能一次或者两次的比较。但是最坏的情况是O(N)，不确定。哈希表的特点是以消耗比较多的内存为代价，把数据的存储和查找消耗的时间大大降低，几乎可以看成是常数时间；在可利用内存越来越多的情况下，用空间换时间。

针对关键字查询操作的次数，以及所要求的是总体查询时间还是单个查询时间来进行选择。如果操作次数少，可使用单次处理时间恒定的map，保证整体的稳定性。

除了用于在散列表中安排数据项的存储位置，散列技术还用在信息处理的很多领域。

由于完美散列函数能够对任何不同的数据生成不同的散列值，如果把散列值当作数据的“指纹”或者“摘要”，这种特性被广泛应用在数据的一致性校验上。

哈希(Hash)函数在中文中有很多译名，有些人根据Hash的英文原意译为“散列函数”或“杂凑函数”，音译为“哈希函数”，还有根据Hash函数的功能译为“压缩函数”、“消息摘要函数”、“指纹函数”、“单向散列函数”等等。

Hash算法是把任意长度的输入数据经过算法压缩，输出一个尺寸小了很多的固定长度的数据，即哈希值。哈希值也称为输入数据的数字指纹（Digital Fingerprint）或消息摘要（Message Digest）等。

Hash函数具备以下的性质：
1、给定输入数据，很容易计算出它的哈希值；
2、反过来，给定哈希值，倒推出输入数据则很难，计算上不可行。这就是哈希函数的单向性，在技术上称为抗原像攻击性；
3、给定哈希值，想要找出能够产生同样的哈希值的两个不同的输入数据，(这种情况称为碰撞，Collision)，计算上不可行，在技术上称为抗碰撞攻击性；
4、哈希值不表达任何关于输入数据的信息。

好的散列函数需要具备3个特性：冲突最少（近似完美）、计算难度低（额外开销小）、充分分散数据项（节约空间）

- 最著名的近似完美散列函数是MD5和SHA系列函数：
  (Python自带MD5和SHA系列的散列函数库：hashlib
  包括了md5/sha1/sha224/sha256/sha384/sha512等6种散列函数)

- MD5（Message Digest）将任何长度的数据变换为固定长为128位（16字节）的“摘要”.

- SHA（Secure Hash Algorithm）是另一组散列函数，SHA-0/SHA-1输出散列值160位（20字节），SHA-256/SHA-224分别输出256位、224位，SHA-512/SHA-384分别输出512位和384位.

散列函数MD5/SHA系列用于数据一致性校验

数据文件一致性判断

为每个文件计算其散列值，仅对比其散列值即可得知是否文件内容相同；

用于网络文件下载完整性校验；

用于文件分享系统：网盘中相同的文件（尤其是电影）可以无需存储多次。

加密形式保存密码

仅保存密码的散列值，用户输入密码后，计算散列值并比对；

无需保存密码的明文即可判断用户是否输入了正确的密码。

防止文件篡改：原理同数据文件一致性判断

当然还有更多密码学机制来保护数据文件，

防篡改，防抵赖，是电子商务的信息技术基础。

❖哈希表的应用

Hash算法在信息安全方面的应用主要体现在以下的3个方面：

(1) 文件校验

(2) 数字签名

(3) 鉴权协议

一、为提高散列（Hash)表的查找效率，可以采取的正确措施是：

A)增大装填因子

B）设计冲突少的散列函数

C)处理冲突时避免产生聚集现象

B 和C

二、将关键字序列{7,8,30,11,18,23,14}散列存储到散列表中，散列表的存储空间是一个下标从0开始的一维数组，散列函数为H(key)=(key*3)MOD7，处理冲突采用线性探测再散列法，要求装填因子为0.7。

（1）请画出所构造的散列表。

（2）分别计算等概率情况下，查找成功和查找不成功的平均查找长度。

	0	1	2	3	4	5	6	7	8	9
key	7	14		8		11	30	18	23	
	1	2		1		1	1	3	3	
	3	2	1	2	1	5	4			

ASL（成功）=（1+2+1+1+1+3+3）/7=1.71

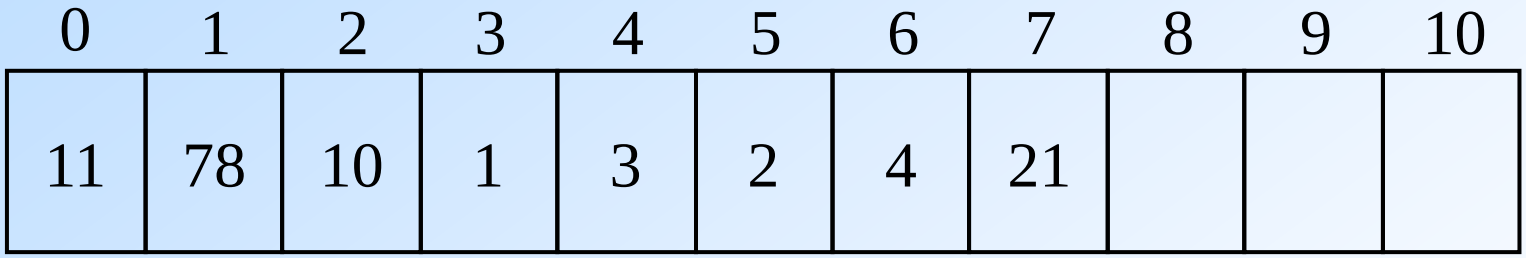
ASL（不成功）=（3+2+1+2+1+5+4）/7=2.57

❖哈希查找的性能分析

例：给定关键字序列{11,78,10,1,3,2,4,21}，试分别用顺序查找、二分查找、二叉排序树查找、平衡二叉树查找、哈希查找(用开放定址法的线性探测法和链地址法)来实现查找，试画出它们的对应存储形式(顺序查找的顺序表，二分查找的判定树，二叉排序树查找的二叉排序树及平衡二叉树查找的平衡二叉树，两种处理冲突的哈希表)，并求出每一种查找的成功（或失败）平均查找长度。哈希函数$H(k)=k \mod 11$。

{11,78,10,1,3,2,4,21}

顺序查找的线性表(一维数组)



由图可得：顺序查找的成功平均查找长度为

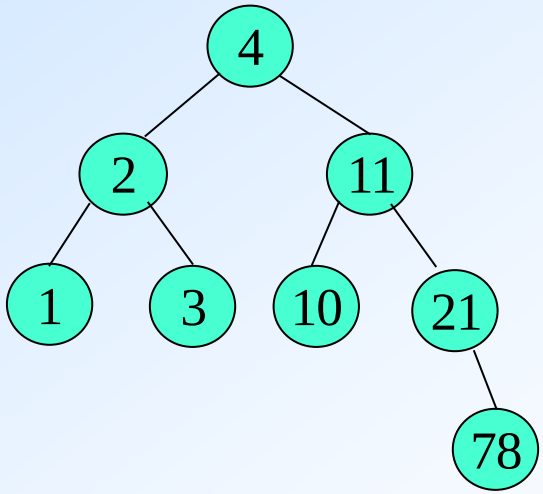
$$ASL=(1+2+3+4+5+6+7+8)/8=4.5$$

{11,78,10,1,3,2,4,21}

折半查找的判定树

折半查找的有序表(一维数组)

0	1	2	3	4	5	6	7
1	2	3	4	10	11	21	78



二分查找的成功平均查找长度为

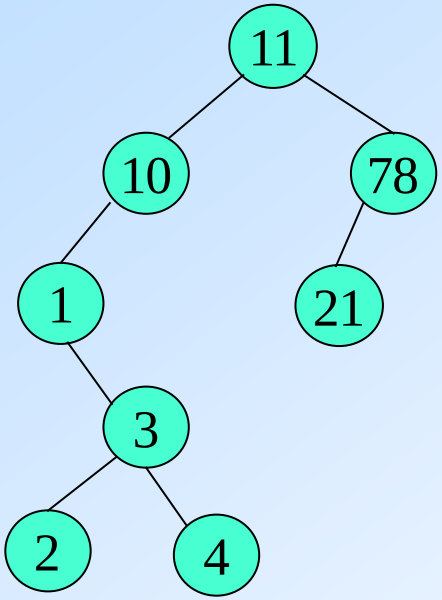
ASL=(1*1+2*2+3*4+4*1)/8=2.625

二分查找失败的平均查找长度为

ASL=(3*7+4*2)/9=29/9

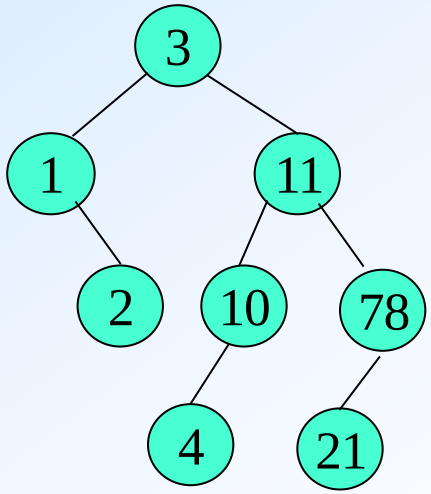
{11,78,10,1,3,2,4,21}

二叉排序树



ASL=(1+2*2+3*2+4+5*2)/8=3.125

平衡二叉树



ASL=(1+2*2+3*3+4*2)/8=2.75

查找不成功的平均查找长度为

ASL=(2*1+3*4+4*4)/9=30/9

{11,78,10,1,3,2,4,21}

开放定址法（线性探测法）解决冲突的哈希表

0	1	2	3	4	5	6	7	8	9	10
11	78	1	3	2	4	21				10

由图可得：线性探查法的成功平均查找长度为

$ASL=(1+1+2+1+3+2+8+1)/8=2.375$

{11,78,10,1,3,2,4,21}

链地址法

解决冲突的哈希表

链地址法成功平均查找长度为

ASL=(1*6+2*2)/8=1.25