第九章

文件及查找

本章内容

9.1 文件的基本概念

9.2 顺序文件

9.3 索引文件

9.4 B-树和B+树

9.5 散列(Hash)文件

9.1 文件的基本概念

例1 花名册

学号	姓 名	性别	年龄	其 他
99001	张 三	女	17	……
99002	李 四	男	18	……
99003	王 五	男	17	……
…	…	…	…	……
…	…	…	…	……
…	…	…	…	……
99050	刘 末	女	19	……

例2 商品清单

代号	名称	数量	入库时间	其 他
110025	电视机	300	2000.2.3	……
120040	洗衣机	125	2000.6.11	……
150003	空调机	50	2000.6.5	……
180024	电冰箱	30	2000.8.15	……
…	…	…	…	……
…	…	…	…	……
…	…	…	…	……

一. 名词术语

学号	姓名	性别	年龄	其　他
99001	张　三	女	17	……
99002	李　四	男	18	……
99003	王　五	男	17	……
…	…	…	…	……
99050	刘　末	女	19	……

字段、数据项

属性：描述一个客体某一方面特征的数据信息。

记录：由若干属性值构成的集合。

文件：具有相同性质的记录的集合。

关键字：区分不同记录的属性或属性组。

二. 文件的逻辑结构

记录呈现在用户眼前的排列的先后次序关系。
(线性结构)

三. 文件的物理结构

文件在存储介质上的组织方式。

1. 连续组织方式(顺序组织方式)
2. 链接组织方式
3. 索引组织方式
4. 随机组织方式(散列组织方式)

顺序文件　索引文件　散列文件

四. 文件的基本操作

查找

在文件中确定某个特定记录存在与否的过程。

结论： 查找成功, 给出被查到记录的位置;
查找失败, 给出相应的信息。

(1). 查找文件的第i个记录。
(2). 查找当前位置的下一个记录。
(3). 按关键字值查找记录。

插入

删除

修改

以查找操作为基础

排序 使记录按关键字值有序排列的过程。

9.2 顺序文件

一. 顺序文件的基本概念

在物理结构中记录排列的先后次序与在逻辑结构中记录排列的先后次序一致的文件称为 ***顺序文件***

记录的排列按关键字值有序的顺序文件称为排序顺序文件，否则，称为一般顺序文件。 逻辑上划分

在存储介质上采用连续组织方式的顺序文件称为连续顺序文件；采用链接组织方式的顺序文件称为链接顺序文件。 物理上划分

若排序顺序文件在存储介质上采用连续组织方式，称之为 ***排序连续顺序文件***。

关键字

排序顺序文件

学号	姓 名	性别	年龄	其 他
99001	张 三	女	17	……
99002	李 四	男	18	……
99003	王 五	男	17	……
…	…	…	…	……
…	…	…	…	……
…	…	…	…	……
99050	刘 末	女	19	……

关键字

一般顺序文件

排序连续顺序文件

二. 连续顺序文件的查找

1. 顺序查找法

查找思想: 从文件的第一个记录开始，将用户给出的关键字值与当前被查找记录的关键字值进行比较，若匹配，则查找成功，给出被查到的记录在文件中的位置，查找结束。若所有n 个记录的关键字值都已比较，不存在与用户要查的关键字值匹配的记录，则查找失败，给出信息0。

关键字集合

$(\ key_1,\ key_2,\ key_3,\ \ \ldots\ldots,\ key_n\ )$

k

被查找记录的关键字值

算法

```
int  SEQSEARCH( keytype key[ ],  int  n,
                                 keytype k )
{
      int  i;
      for( i=1; i<=n; i++ )
           if( key[i]==k )
                return i;
      return 0;
}
```

递归算法如何设计？

例

key[1:10]　38　75　19　57　100　48　50　7　62　11

若查找 k=48　经过六次比较, 查找成功, 返回i = 6

若查找 k=35　查找失败，返回信息0

递归算法

```
int SEQSEARCH2(keytype key[ ],int n,keytype k,int i)
{
        if(i>n)
            return 0;
        if(key[i]==k)
            return i;
        return SEQSEARCH2(key,n,k,i+1);
}
```

调用方式

```
pos=SEQSEARCH2(key,n,k,1);
```

返回位置

查找效率如何

平均查找长度ASL (Average Search Length)

确定一个记录在文件中的位置所需要进行的关键字值的比较次数的期望值(平均值)。

对于具有n个记录的文件，有

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

其中，p_i为查找第i个记录的概率，c_i为查找第i个记录所进行过的关键字的比较次数。

结论

对于具有n个记录的顺序文件，若查找概率相等，则有

$$ASL=\sum_{i=1}^{n}p_i c_i = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}$$

算法的时间复杂度为O(n)！

优点：

- 查找原理和过程简单，易于理解。
- 对于被查找对象的排列次序没有限制。

缺点：

- 查找的时间效率低。

2. 排序连续顺序文件的折半查找法

（二分查找法、 对半查找法）

查找思想：

将要查找的关键字值与当前查找范围内位置居中的记录的关键字的值进行比较，

若匹配，则查找成功，给出被查到记录在文件中的位置，查找结束。

若要查找的关键字值小于位置居中的记录的关键字值，则到当前查找范围的前半部分重复上述查找过程，否则，到当前查找范围的后半部分重复上述查找过程，直到查找成功或者失败。

若查找失败，则给出信息0。

几个变量

n 排序连续顺序文件中记录的个数。

low 当前查找范围内第一个记录在文件中的位置。 初值 **low=1**

high 当前查找范围内最后那个记录在文件中的位置。 初值 **high=n**

mid 当前查找范围内位置居中的那个记录在文件中的位置。 $mid = \lfloor \frac{low+high}{2} \rfloor$

key[1: n]　n=11　k=23

1	2	3	4	5	6	7	8	9	10	11
2	**5**	**7**	**11**	**14**	**16**	**19**	**23**	**27**	**32**	**50**
low					mid	low mid	high low mid	mid		high

经过四次元素之间的比较，查找成功，给出被查到记录在文件中的位置**8(mid)**。

查找失败的标志：low>high

key[1:n]　n=11　k=9

1	2	3	4	5	6	7	8	9	10	11
2	5	7	11	14	16	19	23	27	32	50
low		mid	low	high	mid					high
		high	mid							

经过3次元素之间的比较，未能查到匹配的记录，查找失败。给出信息0。

非递归算法

```
int  BINSEARCH 1( keytype key[ ],  int n,  keytype k )
{
      int low=1, high=n, mid;
      while ( low<=high ) {
           mid=(low+high)/2;
           if ( key[mid]==k )
                return mid;                    /*  查找成功  */
           if ( k>key[mid] )
                low=mid+1;                     /*  查找后半部分 */
           else
                high=mid–1;                    /*  查找前半部分 */
       }
       return 0;                               /*   查找失败  */
}
```


```
int  BINSEARCH 2( keytype key[ ],  int low,  int high,
                                   keytype k )
{
    int  mid;
    if ( low>high )
        return 0;
    else {
        mid=(low+high)/2;
        if ( key[mid]==k )
            return mid;
        else
            if( k<key[mid] )
                return BINSEARCH2( key, low, mid–1, k );
            else
                return BINSEARCH2( key, mid+1, high, k );
        }
}
```

在调用算法中

```
low=1;
high=n;
pos=BINSEARCH2( KEY,low,high,k );
```

查找效率如何？

平均查找长度ASL？

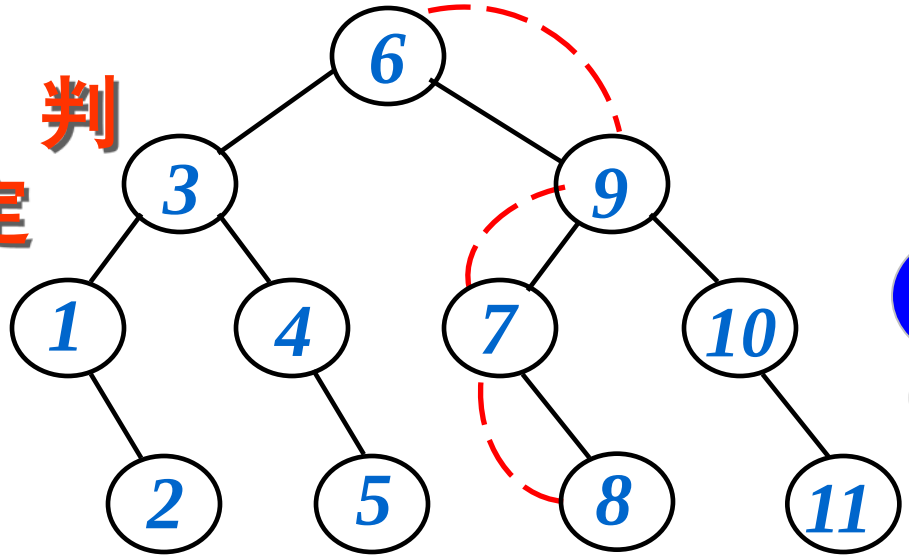
判定树

若把当前查找范围内居中的记录的**位置**作为根结点，前半部分与后半部分的记录的**位置**分别构成根结点的左子树与右子树，则由此可得到一棵称为“**判定树**”的二叉树，利用它来描述折半查找的过程。

位置	2	5	7	11	14	16	19	23	27	32	50
	1	2	3	4	5	6	7	8	9	10	11

查找23

判定树



成功！

结论

成功的查找过程正好等于走了一条从根结点到被查找记录的路径，经历的比较次数恰好是被查找记录在二叉树中所处的层次数！

平均查找长度

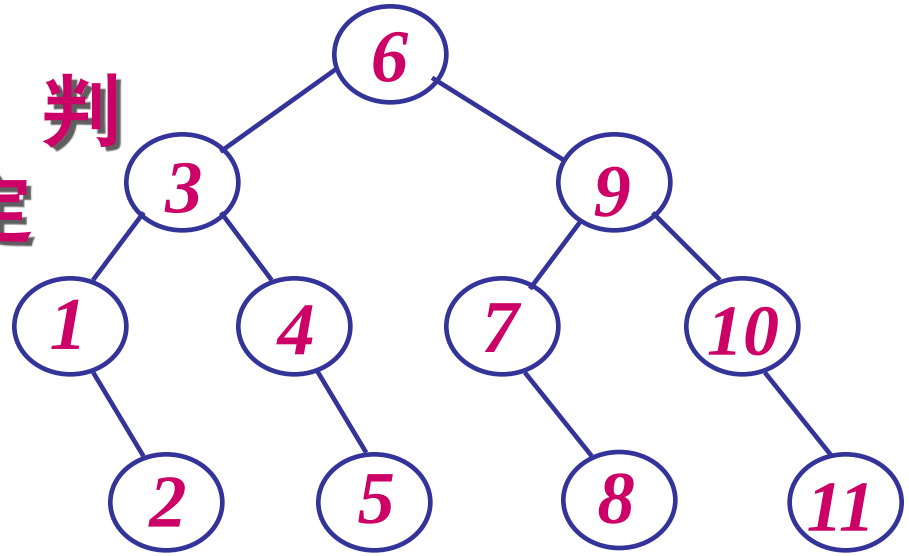
第j层结点数的最大值

对于排序连续顺序文件，若查找概率相等，则有

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

第j层每个结点的比较次数

判定树



当n足够大时，有

$$ASL=\log_2(n+1)-1$$

算法的时间复杂度：$O(\log_2 n)$

例3

若在长度为13、且元素按值从小到大排列的顺序表中采用折半查找法查找某个元素，依次被比较过的元素在表中的位置不可能是___。

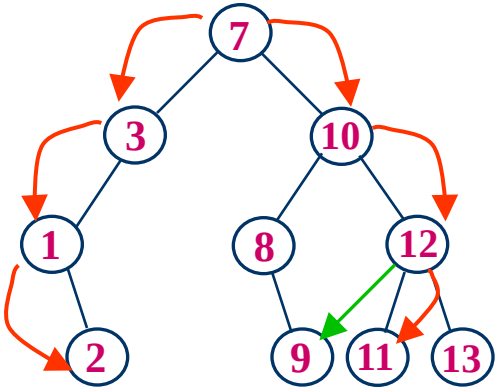
A. 7, 3, 1, 2 ✓

B. 7, 10, 12, 11 ✓

C. 7, 10, 12, 9 ✗

D. 7, 10, 8, 9 ✓

（位置） 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13



判定树

优点：

- 查找原理和过程简单，易于理解。
- 查找的时间效率较高。

缺点：

- 要求文件的记录按照关键字值有序排列。
- 为保持文件的记录按照关键字值有序排列，在文件中插入和删除记录时需要移动大量的其它记录。

折半查找方法特别适用于一经建立就很少改动、而又经常需要查找的文件！

思考

1. 若对一个线性表采用折半查找方法查找一个数据元素，该线性表应该满足什么条件？

数据元素按值有序排列

必须采用顺序存储结构

2. 在线性表中采用折半查找方法查找数据元素，要求该线性表中数据元素必须按值有序排列，这个要求将带来什么负作用？

增加插入删除操作的时间开销

例4

请写出在数据元素按值递增排列的顺序表A[1..n]中插入一个新元素item的算法。要求：

1. 插入后表中元素仍然保持按值递增排列；
2. 分别采用顺序查找法和折半查找法确定插入位置。

需要做的工作

1. 确定插入位置(采用顺序查找法和折半查找法)；
2. 将新元素插入指定位置：
   ① 将相关元素依次后移一个位置；
   ② 插入新元素；
   ③ 修改表的长度。

采用顺序查找法确定插入位置

$(a_1,\ a_2,\ a_3,\ \cdots,\ a_i,\ a_{i+1},\ \cdots,\ a_{n-1},\ a_n)$ item


```
void INSERT1(ElemType A[ ], int &n, item)
{       int i,j;
          i=1;
          while(i<=n){                          /* 确定插入位置 */
                if(item<A[i])
                     break;
                i++;
          }
          for(j=n;j>=i;j--)
                A[j+1]=A[j];
          A[i]=item;                            /*  插入新元素 */
          n++;                                  /*  修改表长 */
}
```

采用折半查找法确定插入位置

插入位置

2, 4, 6, 8, 10, 12, 14, 16, 18, 20)

item=13

high

low

mid


```
void INSERT2(ElemType A[ ], int &n, item)
{       int i,j,low,high,mid;
        low=1;
        high=n;
        while(low<=high){                       /*  确定插入位置 */
              mid=(low+high)/2;
              if(item<A[mid])
                    high=mid−1;
              else
                    low=mid+1;
        }
        for(j=n;j>=low;j−−)                     /* 后移相关元素的位置 */
             A[j+1]=A[j];
        A[low]=item;                            /*  插入新元素 */
        n++;                                    /*  修改表长 */
}
```

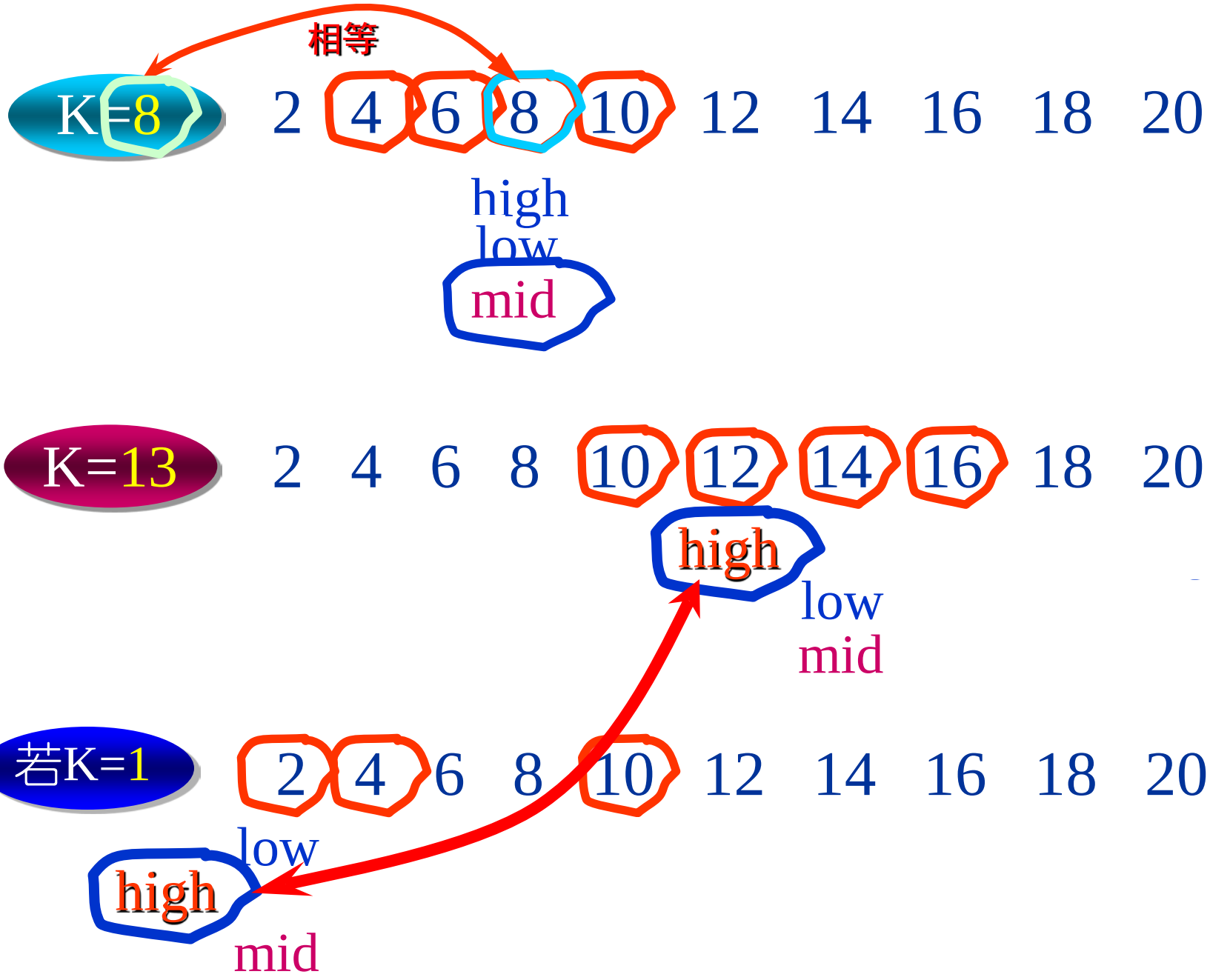
例5

请写一非递归算法，该算法在长度为 n、且元素按值严格递增排列的顺序表A[1..n]中采用 **折半查找法** 查找值不大于k的最大元素，若表中存在这样的元素，则算法返回该元素在表中的位置，否则，返回0。

例

2 4 6 8 10 12 14 16 18 20

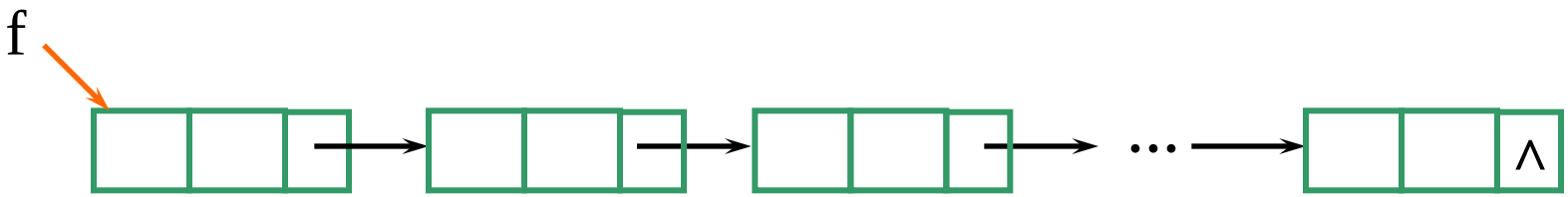
K=8　　K=13　　K=1


相等
K=8
2 4 6 8 10 12 14 16 18 20
high
low
mid
K=13
2 4 6 8 10 12 14 16 18 20
high
low
mid
若K=1
2 4 6 8 10 12 14 16 18 20
low
high
mid

折半查找算法


```
int BINSEARCH(keytype A[ ],int n,keytype k)
{
      int low=1, high=n, mid;
      while(low<=high){
          mid=(low+high)/2;
          if(A[mid]==k)
                return mid;          /*  返回mid  */
          if(k>A[mid])
                low=mid+1;           /*  准备查找后半部分 */
          else
                high=mid–1;          /* 准备查找前半部分 */
       }
       return high;                  /* 返回high */
}
```

三. 链接顺序文件的查找



算法

```
typedef  struct node {
      keytype   key;
      rectype    rec;
      struct node  *link;
} *KeyLink;
```

```
KeyLink SEARCH( KeyLink F,  keytype  k )
{
      KeyLink  p;
      p=F;
      while( p!=NULL ) {
          if( p->key==k )
              return( p );              /* 查找成功 */
          p=p->link;
      }
      return( NULL );                   /* 查找失败 */
}
```

递归算法

```
KeyLink SEARCH(KeyLink F,  keytype  k)
{
      if(F!=NULL)
          if(F->key==k )
              return F;                    /* 查找成功 */
          else
              return SEARCH(F->link, k );
      return NULL;                         /* 查找失败 */
}
```

9.3 索引文件

一. 索引文件的基本概念

1. 索引

记录关键字值与记录的存储位置之间的对应关系。

2. 索引文件

由基本数据与索引表两部分组成的数据文件称为索引文件。

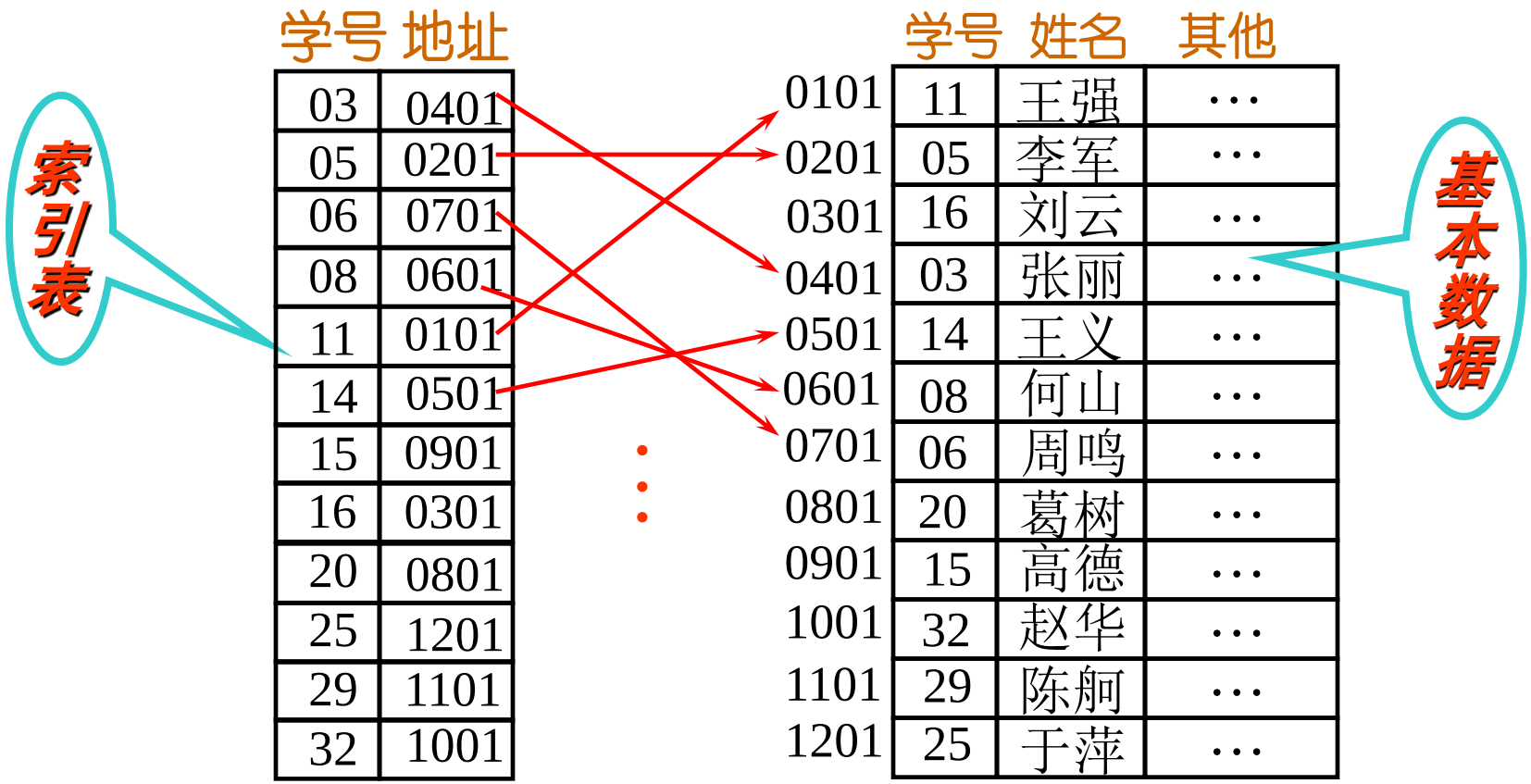
3. 索引表的特点

(1). 索引表是由系统自动产生的。

(2). 索引表中表项按关键字值有序排列。

二. 稠密索引文件

特点 文件的基本数据中的每一个记录在索引表中都占有一项。

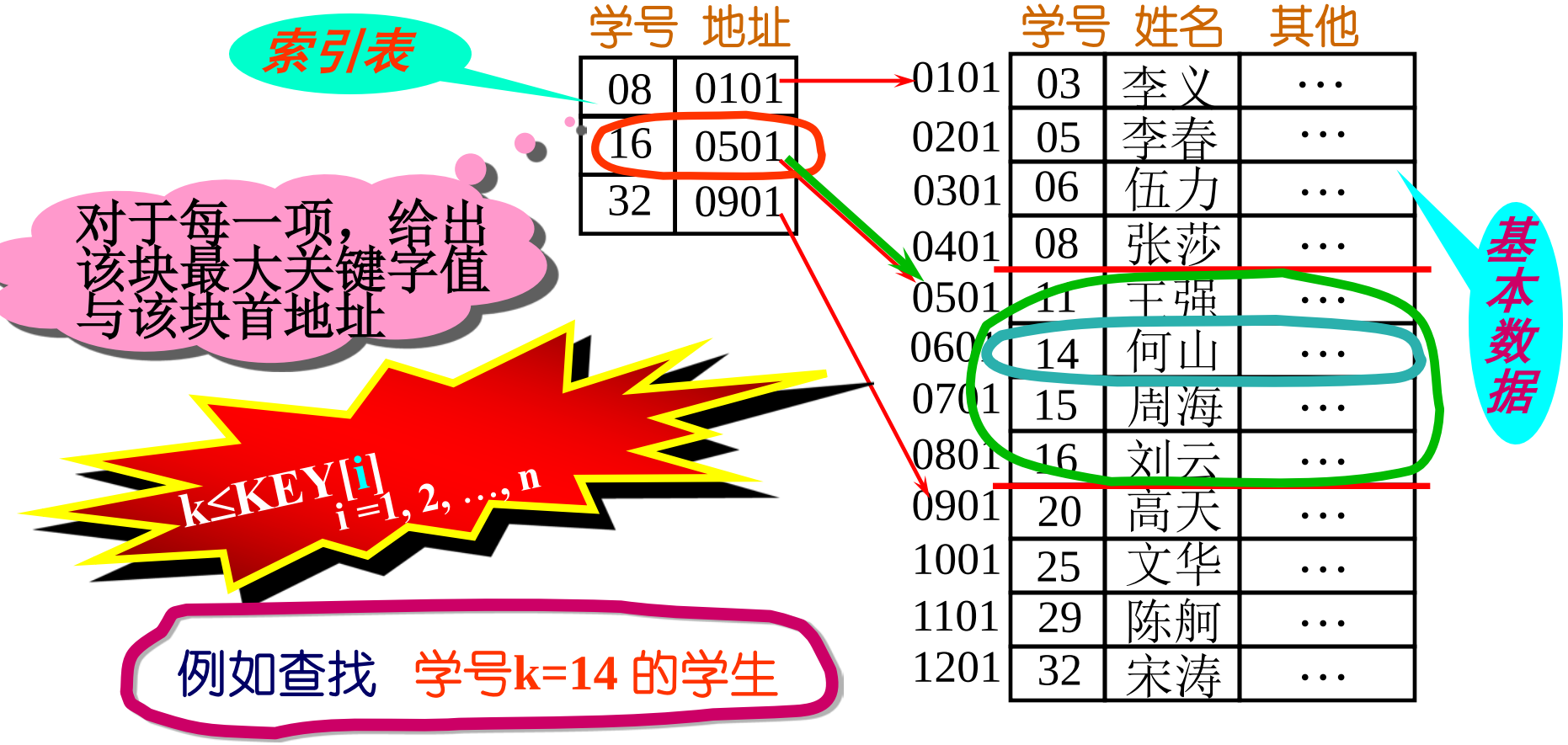


结论

在稠密索引文件中查找一个记录存在与否的过程是直接查找索引表。

三. 非稠密索引分块文件

特点 将文件的基本数据中记录分成若干块(块与块之间记录按关键字值有序， 块内记录是否按关键字值有序无所谓)，索引表中为每一块建立一项。



结论

在非稠密索引(分块)文件中查找一个记录存在与否的过程是：

先查找索引表(确定被查找记录所在块)，然后在相应块中查找被查记录存在与否。

四. 多级索引文件

当索引文件的索引本身非常庞大时，可以把索引分块，建立索引的索引，形成 ***树形结构的多级索引*** 。

二叉排序树多级索引结构、
多分树索引结构

树形结构的多级索引

9.4 B-树和B+树

P_i指的结点中所有关键字值都大于key_i

一. B- 树的定义

一个m阶的B-树为满足下列条件的m叉树：

(1). 每个分支结点最多有m 棵子树；

(2). 除根结点外，每个分支结点最少有⌈m/2⌉棵子树；

(3). 根结点最少有两棵子树(除非根为叶结点，此时B -树只有一个结点)；

(4). 所有“叶结点”都在同一层上，叶结点不包含任何关键字信息(可以把叶结点视为实际上不存在的外部结点，指向这些“叶结点”的指针为空)；

(5). 所有分支结点中包含下列信息：

$$n,\ p_0,\ key_1,\ p_1,\ key_2,\ p_2,\ \ldots,\ key_n,\ p_n$$

还含有n个指向相应记录的指针

其中，n 为结点中关键字值的个数，

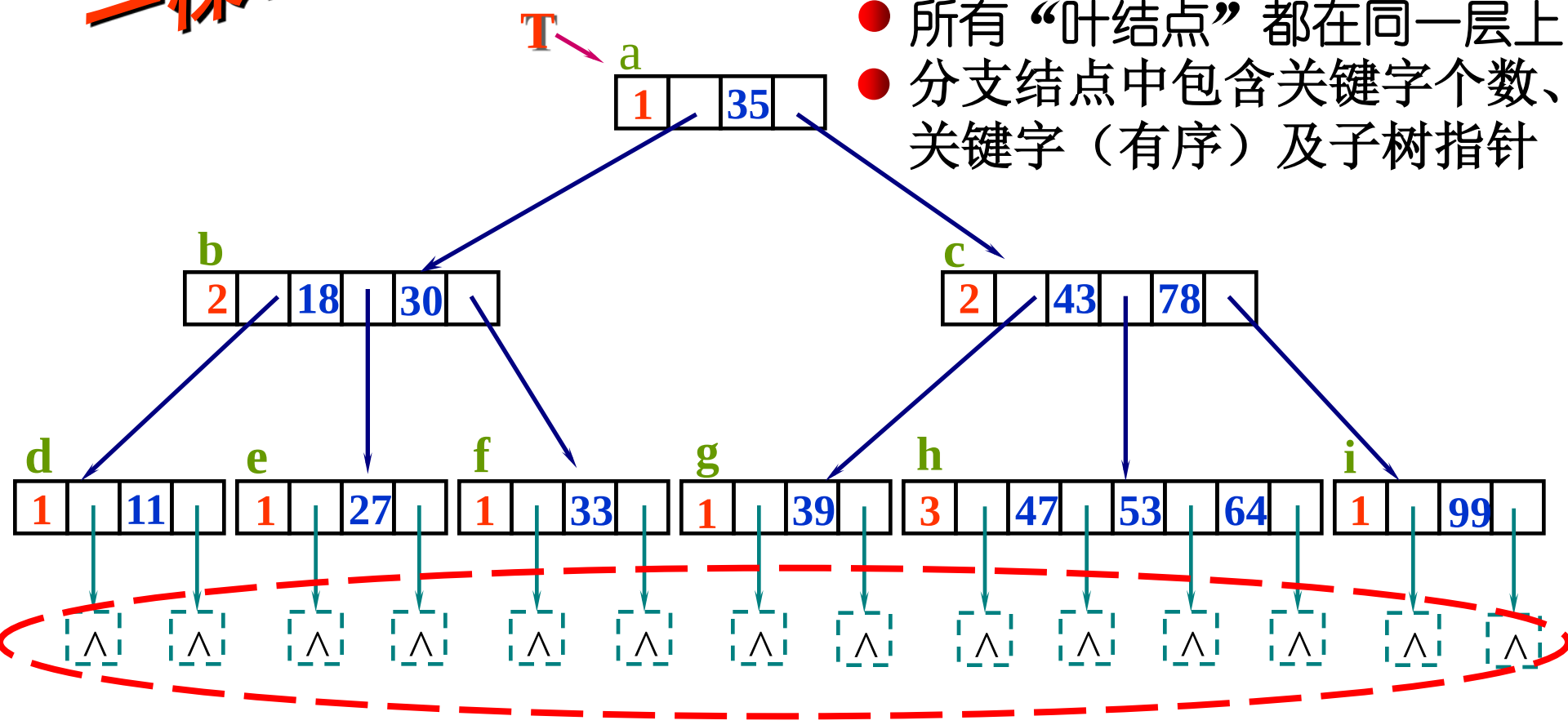
key_i为关键字，且满足 $key_i<key_{i+1}$ $1\le i<n$

p_i为指向该结点的第i+1棵子树的根的指针。 $0\le i\le n$

例

一棵4阶B-树

- 每个分支结点最多有4棵子树
- 每个分支结点最少有2棵子树
- 根结点最少有2棵子树
- 所有“叶结点”都在同一层上
- 分支结点中包含关键字个数、关键字（有序）及子树指针

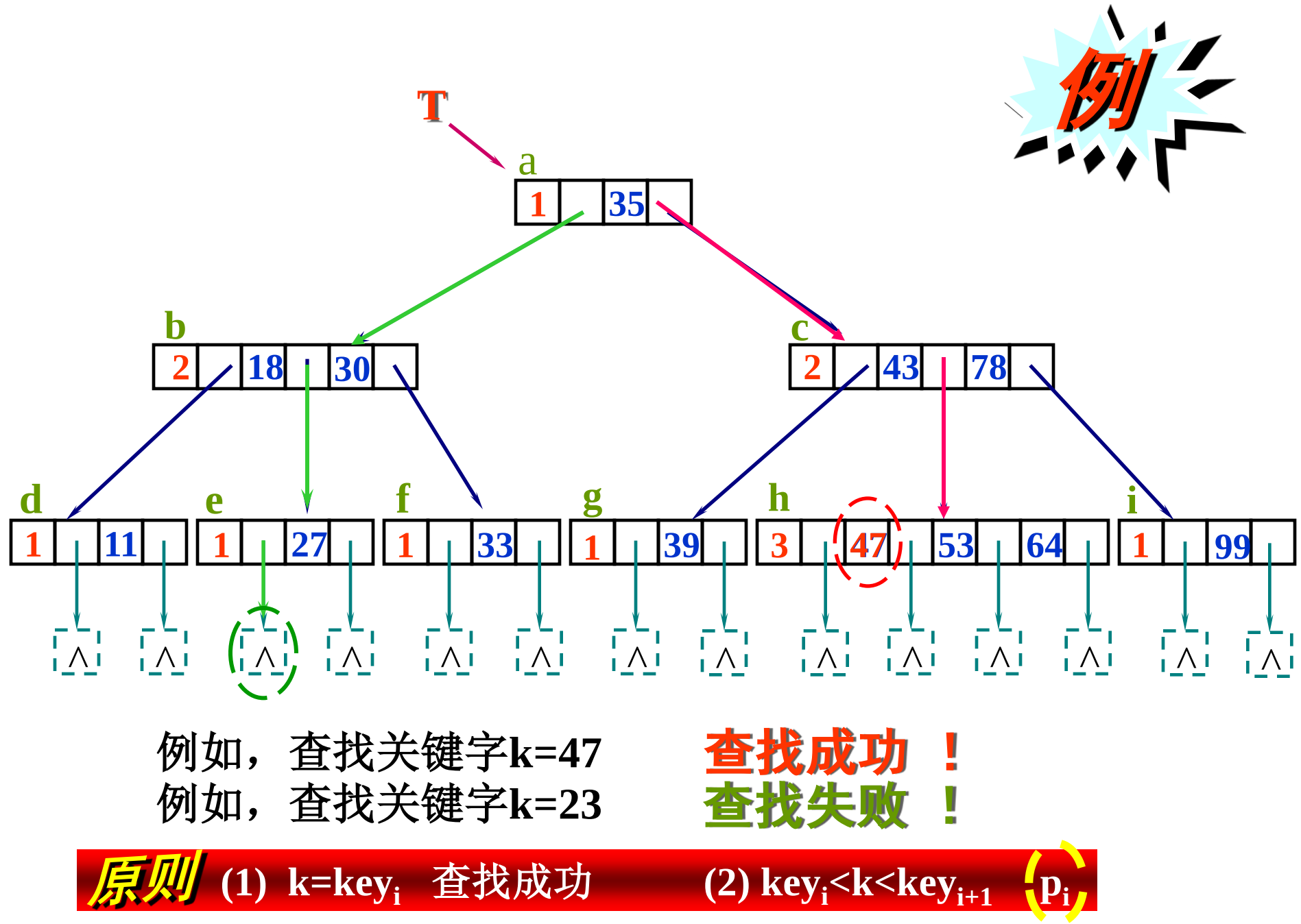


二. B- 树的查找

$n,\ p_0,\ key_1,\ p_1,\ key_2,\ p_2, \ldots,\ key_n,\ p_n$

首先将给定的关键字k在B-树的根结点的关键字集合中采用顺序查找法或者折半查找法进行匹配查找，若有$key_i=k$，则查找成功，根据相应的指针取得记录。否则，若k 在key_i与 key_{i+1} 之间，则在指针p_i所指的结点中重复上述查找过程，直到在某结点中查找成功，或者在某结点中出现$p_i=nil$，查找失败。

类似于在二叉排序树中查找一个结点的过程



例如，查找关键字k=47　　查找成功 ！

例如，查找关键字k=23　　查找失败 ！

原则 (1) $k=key_i$ 查找成功　　(2) $key_i<k<key_{i+1}$ p_i

类型定义

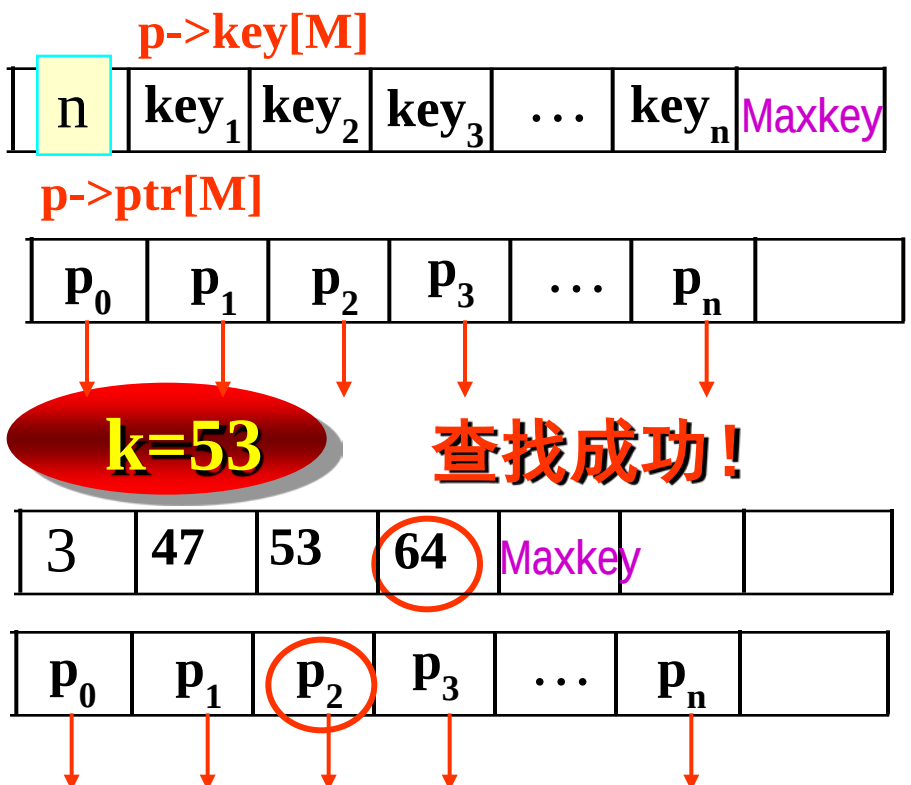
```
#define M     1000
typedef struct node {
      int  keynum;
      keytype  key[M+1];
      struct node  *ptr[M+1];
      rectype * recptr[M+1];
} *BTree;
```

算法

```
int  MBSEARCH( Btree T, keytype k )
{
      int i,n;
      BTree p=T;
      while( p!=NULL ) {
          n=p->keynum;
          p->key[n+1]=Maxkey;
          i=1;
          while( k>p->key[i] )
              i++;
          if( p->key[i]==k )
              return( p->recptr[i] );
          else
              p=p->ptr[i-1];
      }
      return (-1);
}
```

```
#define M     1000
typedef struct node {
      int  keynum;
      keytype  key[M+1];
      struct node  *ptr[M+1];
      rectype * recptr[M+1];
} *BTree;
```

在p指结点的关键字集合中查找k



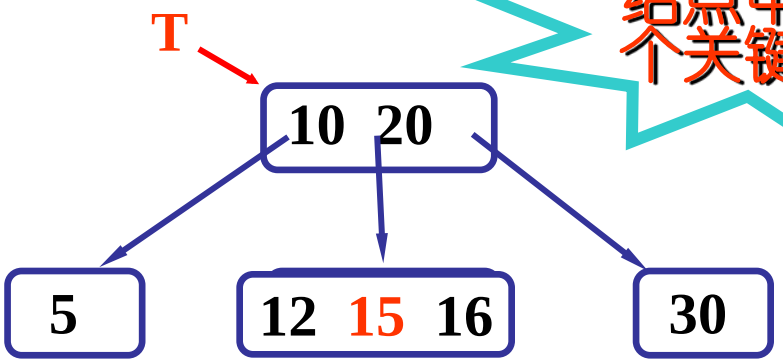
k=62

沿着新的指针继续查找！

三. B-树的插入

一棵m阶B-树的结点中最多有m-1个关键字值

一棵3阶B-树



插入 k=15

基本思想

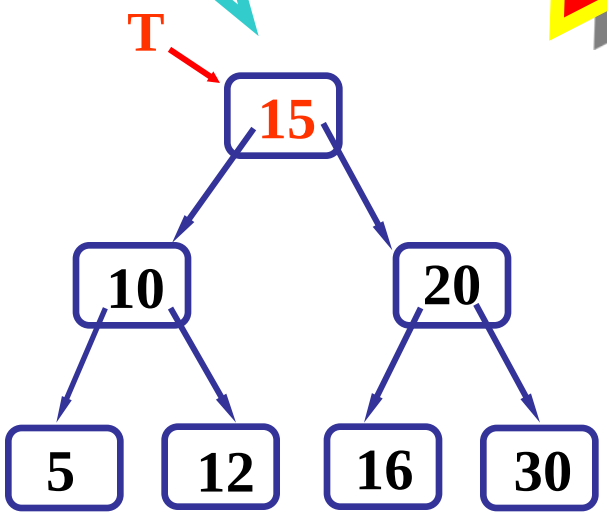
若将k插入到某结点后使得该结点中关键字值数目超过m-1时，则要以该结点位置居中的那个关键字值为界将该结点一分为二，产生一个新结点。并把位置居中的那个关键字值插入到双亲结点中；如双亲结点也出现上述情况，则需要再次进行分裂。最坏情况下，需要一直分裂到根结点，以致于使得B-树的深度加1。

完整地看这个例子

B-树的深度增加1层

一棵3阶B-树
的结点中最多
有2个关键字值

3阶B-树



插入 k=15

插入15 以后的3阶B-树

一般情况下

若某结点已有m−1个关键字值，在该结点中插入一个新的关键字值，使得该结点内容为

q

m	key_1	key_2	key_3	…	$key_{\lceil m/2\rceil-1}$	$key_{\lceil m/2\rceil}$	…	key_{m-1}	key_m

则需要将该结点分解为两个结点q与q’，结点内容为

q

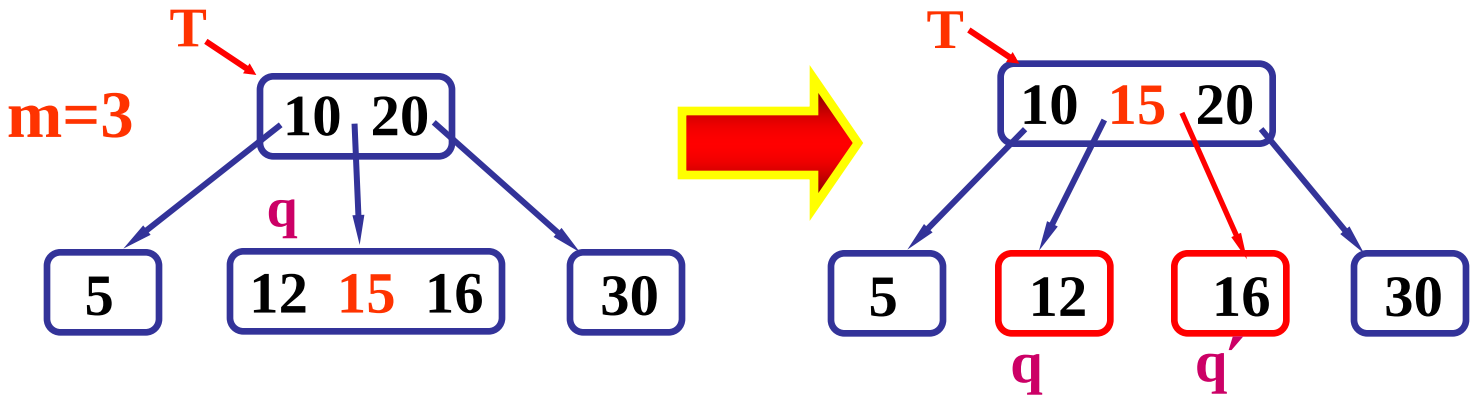
$\lceil m/2\rceil-1$	key_1	key_2	…	$key_{\lceil m/2\rceil-2}$	$key_{\lceil m/2\rceil-1}$

q’

$m-\lceil m/2\rceil$	$key_{\lceil m/2\rceil+1}$	$key_{\lceil m/2\rceil+2}$	…	key_{m-1}	key_m

并且将关键字值$key_{\lceil m/2\rceil}$与一个指向q’的指针插入到q的双亲结点中。

例如



四. B +树的定义

一个m阶的B+树为满足下列条件的m叉树：

同B-树

(1). 每个分支结点最多有m 棵子树；

(2). 除根结点外，每个分支结点最少有⌈m/2⌉棵子树；

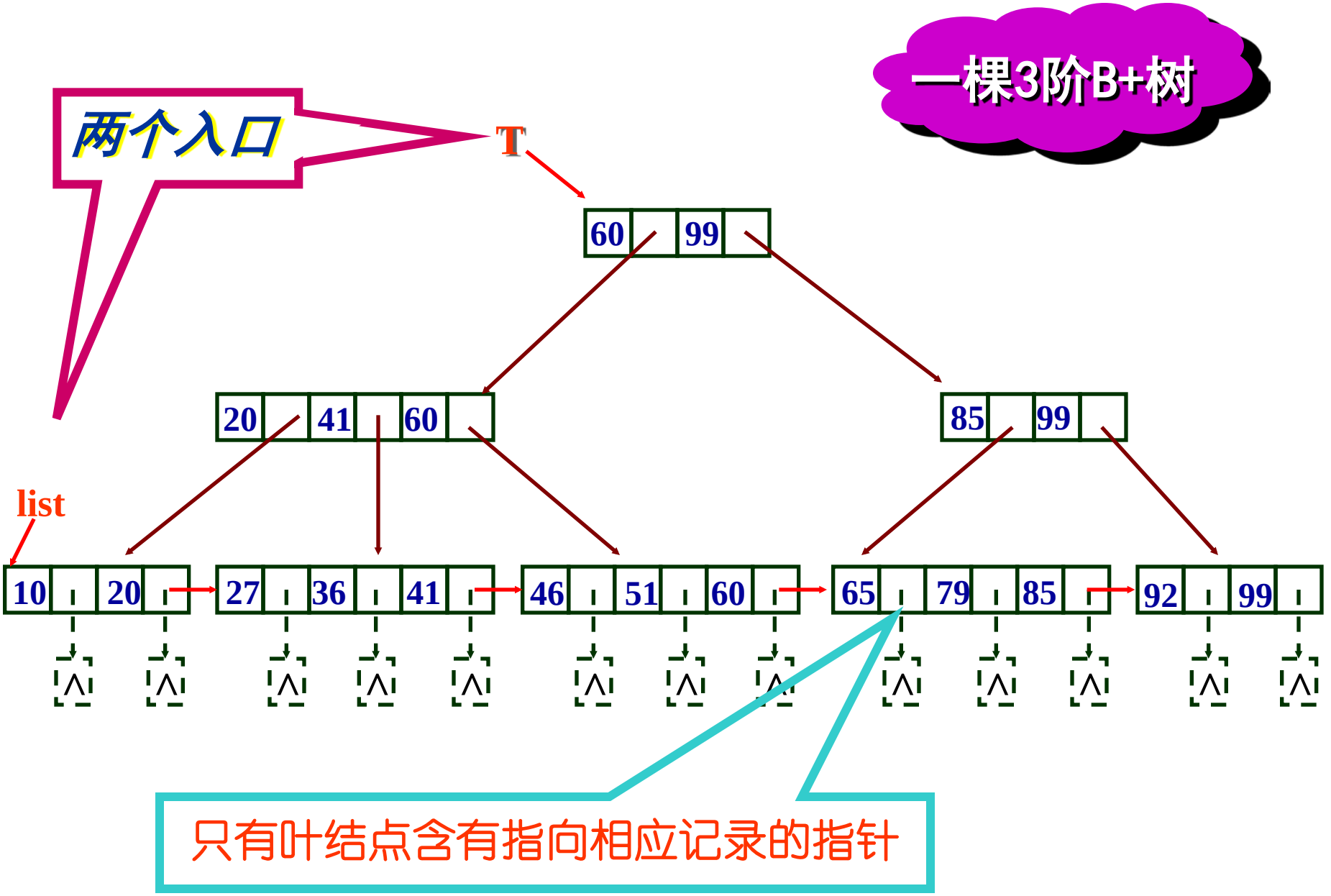
(3). 根结点最少有两棵子树(除非根为叶结点结点，此时B+树只有一个结点)；

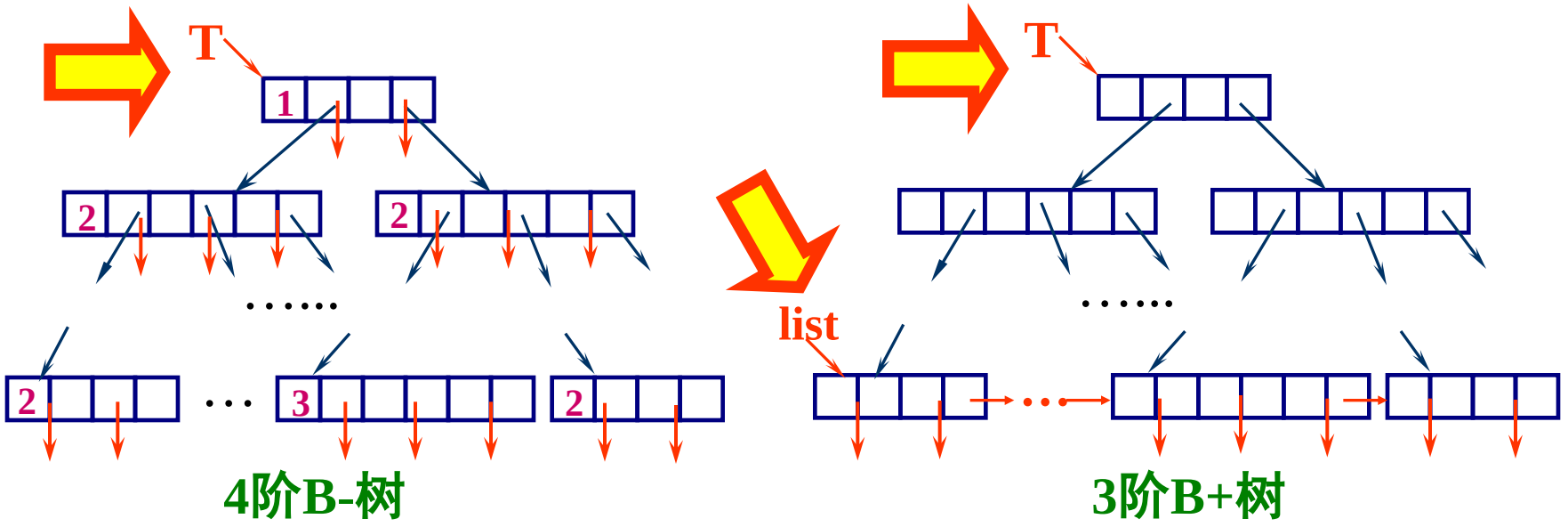
(4). 具有n 棵子树的结点中一定有n 个关键字；

(5). 叶结点中存放记录的关键字以及指向记录的指针，或者数据分块后每块的最大关键字值及指向该块的指针，并且叶结点按关键字值的大小顺序链接成线性链表。

key_1 p_1 key_2 p_2 …… key_n p_n

(6). 所有分支结点可以看成是索引的索引，结点中仅包含它的各个孩子结点中最大(或最小)关键字值和指向孩子结点的指针。


一棵3阶B+树
两个入口
T
60
99
20
41
60
85
99
list
10
20
27
36
41
46
51
60
65
79
85
92
99
只有叶结点含有指向相应记录的指针



B-树与B+树的区别 (从结构上看)

1. B-树的每个分支结点中含有该结点中关键字值的个数，B+树没有。
2. B-树的每个分支结点中含有指向关键字值对应记录的指针，而B+树只有叶结点有指向关键字值对应记录的指针。
3. B-树只有一个指向根结点的入口，而B+树的叶结点被链接成为一个不等长的链表，因此，B+树有两个入口，一个指向根结点，另一个指向最左边的叶结点(即最小关键字所在的叶结点)。

9.5 散列(Hash)文件

一. 散列文件的基本概念

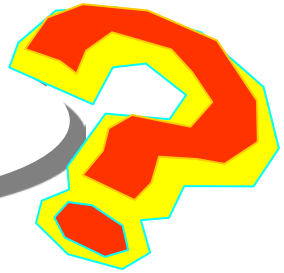
顺序文件的**顺序查找法**和**折半查找法**、**索引文件查找法**以及在**B-树B+树中进行的查找方法**。

学 号	姓 名	年龄	…
99001	王　亮	17	…
99002	张　云	18	…
99003	李海民	20	…
99004	刘志军	19	…
…	…		
99049	周　颖	18	…
99050	罗　杰	16	…

基于关键字值比较的查找方法

查找的时间效率取决于查找过程中的比较次数

能否有一种不经过任何关键字值的比较或者经过很少次的关键字值的比较就能够达到目的的方法



答案是肯定的。需要建立记录的关键字值与记录的存储位置之间的关系。

1. 基本概念

$$A = H(k)$$

其中，k 为记录的关键字，H(K)称为散列函数，或哈希(Hash)函数，或杂凑函数。函数值A为k对应的记录在文件中的位置

关键字

学 号	姓名	性别	…
99001	张 云	女	…
99002	王 民	男	…
99003	李 军	男	…
99004	汪 敏	女	…
……	…	…	…
99030	刘小春	男	…

1	张 云 …
2	王 民 …
3	李 军 …
4	汪 敏 …
:	……
:	
30	刘小春 …

地址范围: **[1：30]**

散列函数: **H(k)=k–99000**

例2

关键字

学 号	姓名	性别	…
99001	张 云	女	…
99002	王 民	男	…
99003	李 军	男	…
99004	汪 敏	女	…
……	…	…	…
99030	刘小春	男	…

1	李 军 …
2	张 云 …
3	
4	王汪 民敏……
:	……
:	
30	

地址冲突

地址范围：[1：30]

散列函数：

H(k) = “将组成关键字k 的串转换为一个1—30之间的代码 ”

一个处理过程

H(张云)=2
H(王民)=4
H(李军)=1
H(汪敏)=4

2. 散列冲突

对于不同的关键字k_i与k_j，经过散列得到相同的散列地址，即有

$$H(k_i) = H(k_j)$$

这种现象称为**散列冲突**。

称k_i与k_j为“同义词”

3. 什么是散列文件？

根据构造的散列函数与处理冲突的方法将一组关键字映射到一个有限的连续地址集合上，并以关键字在该集合中的“象”作为记录的存储位置，按照这种方法组织起来文件称为**散列文件**，或称**哈希文件**，或称**杂凑文件**；建立文件的过程称为哈希造表或者散列，得到的存储位置称为散列地址或者杂凑地址。

二. 散列函数的构造

1. 原 则

- 散列函数的定义域必须包括将要存储的全部关键码；若散列表允许有m个位置时，则函数的值域为[0:m–1](地址空间)。
- 利用散列函数计算出来的地址应能尽可能均匀分布在整个地址空间中。
- 散列函数应该尽可能简单，应该在较短的时间内计算出结果。

一个“好”的散列函数

2. 步骤

- 确定散列的地址空间(地址范围)；
- 构造合适的散列函数；
- 选择处理冲突的方法。

详见教材P307-309

3. 方法

1. 直接定址法
2. 数字分析法
3. 平方取中法
4. 叠加法
5. 基数转换法
6. 除留余数法

一般形式

$H(k)=ak+b$

$H(k)=k-99000$

除留余数法

$$H(k) = k \ \text{MOD} \ p$$

其中，若m为地址范围大小(或称表长)，
则p可为小于等于m的素数。

三. 冲突的处理方法

所谓处理冲突，是在发生冲突时，为冲突的元素找到另一个散列地址以存放该元素。如果找到的地址仍然发生冲突，则继续为发生冲突的这个元素寻找另一个地址，直到不再发生冲突。

1. 开放地址法

闭散列方法

所谓开放地址法是在散列表中的“空”地址向处理冲突开放。即当散列表未满时，处理冲突需要的“下一个”地址在该散列表中解决。

$$D_i = (H(k)+d_i) \text{ MOD } m \qquad i=1, 2, 3, \ldots$$

其中，H(k)为哈希函数，m为表的长度，d_i为地址增量，有：

(1) $d_i=1, 2, 3, \ldots, m-1$　　称为线性再散列

(2) $d_i=1^2, 2^2, \ldots, (\lfloor m/2 \rfloor)^2$　　称为二次再散列

(3) d_i=伪随机数序列　　称为伪随机再散列

例3

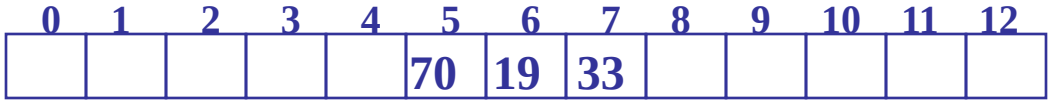
除留余数法

设散列函数为

$$H(k) = k \ \ MOD \ \ 13$$

散列表为[0: 12]， 表中已分别有关键字为19，70，33的记录，现将第四个记录(关键字值为18)插入散列表中。

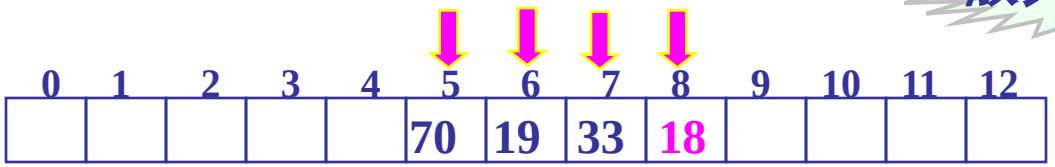
插入前



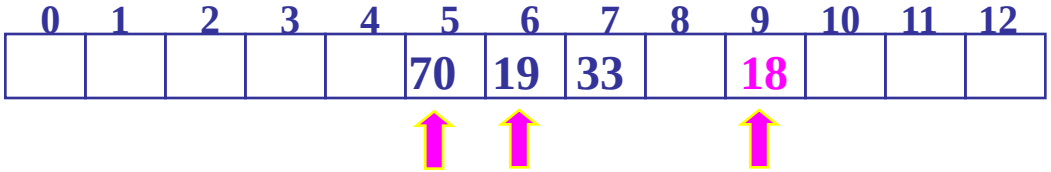
$$D_i = (k \ \ MOD \ \ 13 + d_i) \ \ MOD \ \ 13$$

散列地址为5

线性再散列



二次再散列



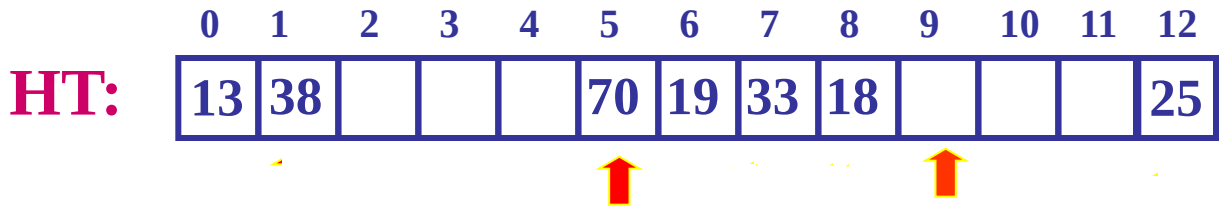
例4 查找

已知有长度为M的散列表HT[0:M–1]，散列函数为H(k)，并且采用线性探测再散列方法处理冲突。请写出在该散列表中查找关键字值为key的元素存在与否的算法。若存在，则给出它在表中的位置，否则，给出相应信息。

$H(k)=k \ MOD \ 13$

例

$D_i = (k \ MOD \ 13 + d_i) \ MOD \ 13$



key=70　key=18　key=38　key=20

算法

```
int  HASHSEARCH(ElemType HT[], int M, ElemType key)
     int i,D;
     i=H(key);
     D=i;
     while (HT[D] !=空 && HT[D]!=key ) {
          D=(D+1) % M;
          if (D==i)              // 转了一圈回到开始点(表满), 查找失败 //
               return (–1);
      }
          if (HT[D]==key)
             return(D);                 // 查找成功 //
          return (–1);                 //查找失败 //
}
```

聚集 —— 散列地址不同的元素争夺同一后继散列地址的现象。

产生聚集的主要原因

1. 散列函数选择得不合适；
2. 负载因子过大。

负载因子 —— 衡量散列表的 **饱满程度** 。

$$\alpha = \frac{\text{散列表中实际存入的元素数}}{\text{散列表中基本区的最大容量}}$$

例5

已知要将给定的关键字值序列(25,33,26,47,38,59,64)进行散列存储，并且要求装填因子α=0.5，

1. 请利用除留余数法构造出合适的散列函数；
2. 请画出利用该散列函数依次将序列中各关键字值插入到散列表后表的状态。设散列表初始为空，并且采用线性探测再散列法处理冲突。

1. 由于$\alpha=\frac{\text{散列表中实际存入的元素数}}{\text{散列表的长度m}}$

得到散列表长度$m=\frac{\text{散列表中实际存入的元素数}}{\alpha}=\frac{7}{0.5}=14$

于是，构造散列函数为 H(k)=k MOD 13

2. 散列表：

0	1	2	3	4	5	6	7	8	9	10	11	12	13
26	64						33	47	59			25	38

思考

若n个关键字值具有相同的散列地址，并且采用线性探测再散列法处理冲突，将这些关键字值散列到初始为空散列表中，一共要进行多少次探测？

$k_1,\ k_2,\ k_3,\ k_4,\ \cdots,\ k_n$

$D=H(k)$

$0+1+2+3+\cdots+(n-1)=n\times(n-1)/2$(次)

特点

- “线性探测法”容易产生元素“聚集”的问题。
- “二次探测法”可以较好地避免元素“聚集”的问题，但不能探测到表中的所有元素(至少可以探测到表中的一半元素)。
- 只能对表项进行逻辑删除(如做删除标记)，而不能进行物理删除。使得表面上看起来很满的散列表实际上存在许多未用位置。

2. 再散列法

$D_i = H_i(k)$　　i=1, 2, 3, …

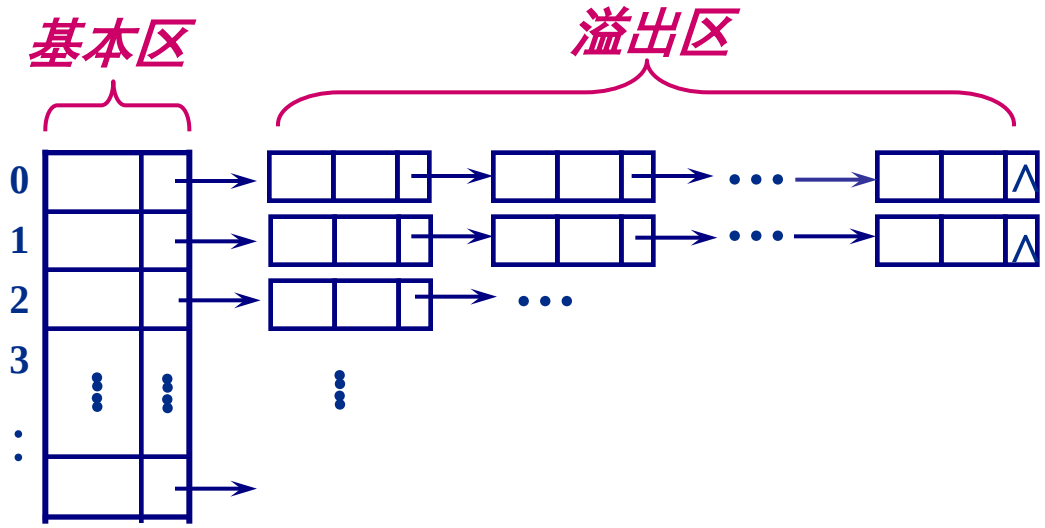
其中， D_i为散列地址，$H_i(k)$为不同的散列函数。

3. 链地址法

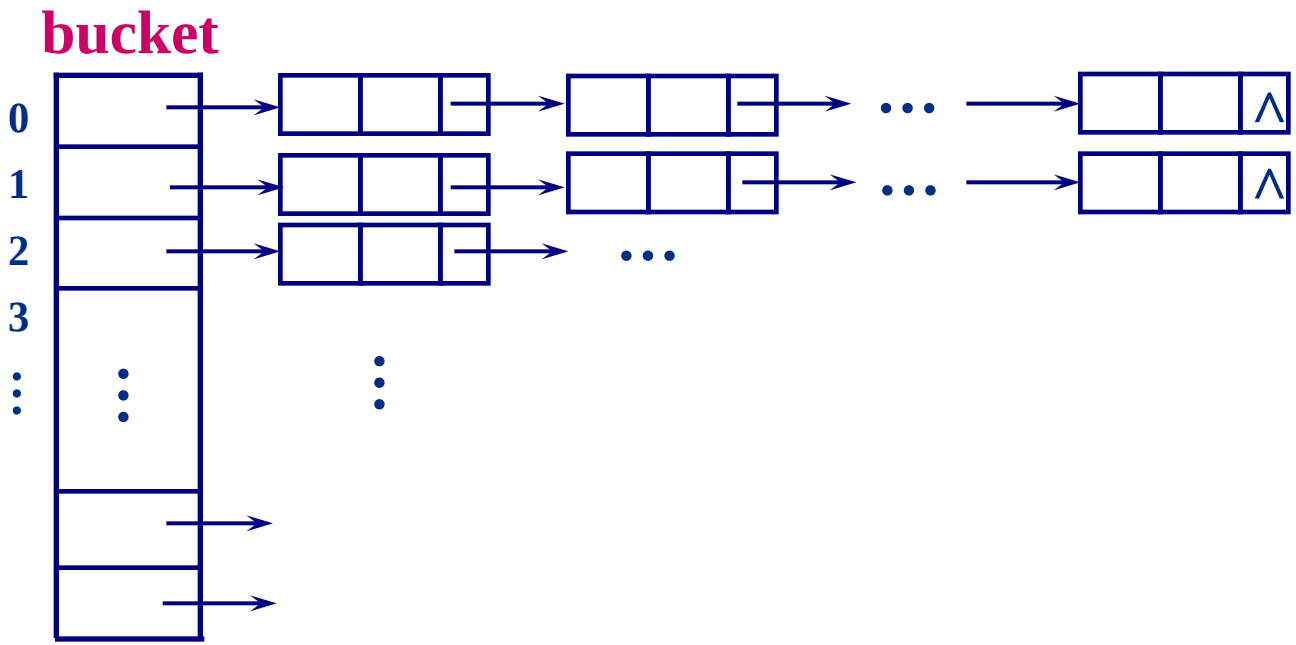
方法一

将哈希文件分为基本区与溢出区两个部分，没有发生冲突的记录放在基本区，当发生冲突时，把具有相同散列地址的记录存放在溢出区，并把具有相同地址的记录采用一个线性链表链接起来。

“同义词”



方法二 不分基本区与溢出区。将所有散列地址相同的记录链接成一个线性链表。若散列范围为[0: m-1], 则定义一个指针数组bucket[0 : m-1]分别存放m个链表的头指针。

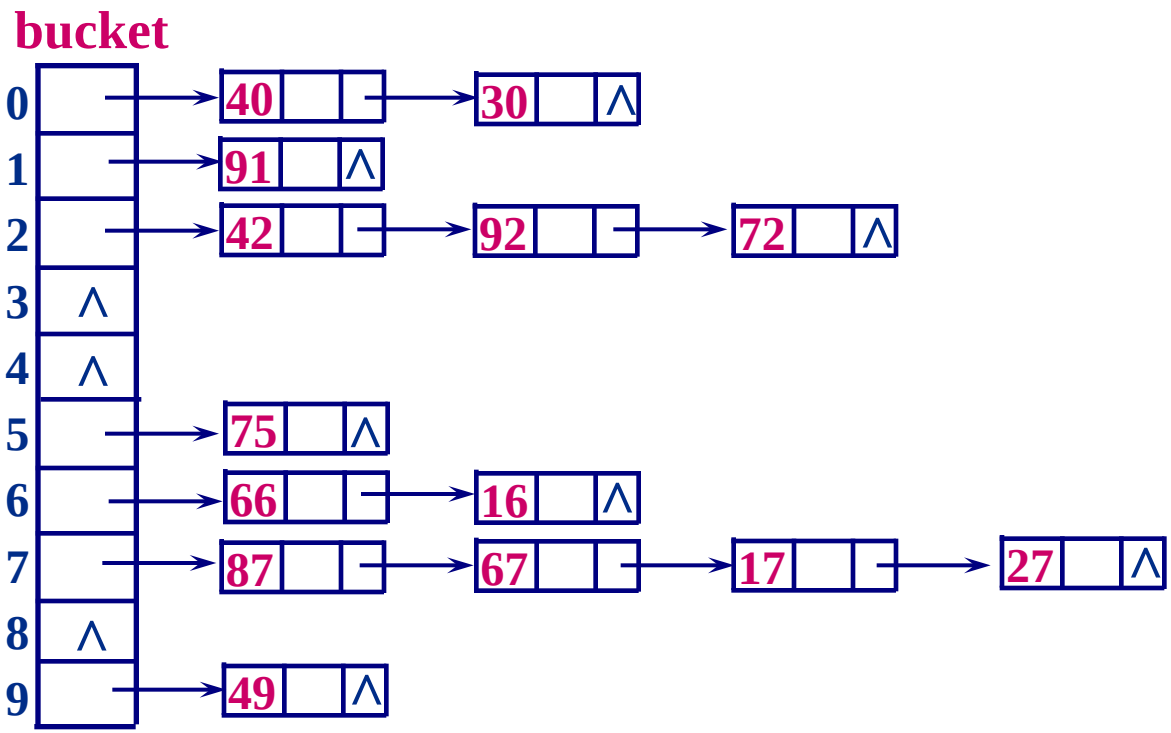


例6

设散列函数为

$$H(k) = k \ \mathrm{MOD}\ 10$$

散列表为[0 ： 9]，采用链地址法处理冲突， 则关键字序列{75, 66, 42, 192, 91, 40, 49, 87, 67, 16, 17, 30, 72, 27}对应的记录插入散列表后的散列文件



特点

- 处理冲突简单，不会产生元素“聚集”现象，因此，平均查找长度较小。
- 特别适合建立散列表之前难以确定表长的情况。
- 在“采用链地址法”建立的散列表中进行插入删除操作简单。
- 由于指针域需占用额外空间，当规模较小时，不如“开放地址法”节省空间。

在各种查找方法中，

只有 *散列查找法*

的平均查找长度ASL与

元素的个数n无关

为什么？

在散列函数与散列地址范围都分别相同的前提下,为什么说采用链地址法处理冲突比采用开放地址法的时间效率高？

为什么？

顺序查找法、折半查找法、树型查找法和散列查找法四种查找方法中，只能在顺序存储结构上进行的查找方法是哪一种？

本章内容小结

文件及查找

一、文件的基本概念

1. 基本名称术语
2. 文件的逻辑结构与物理结构
3. 文件的基本操作(查找、排序)

二、顺序文件及其查找

1. 顺序文件的基本概念
   - 一般顺序文件、排序顺序文件
   - 连续顺序文件、链接顺序文件
   - 排序连续顺序文件
2. 连续顺序文件的查找
   - 顺序查找法
   - 折半查找法

   } (递归和非递归过程)
   - 时间复杂度分析(判定树)
3. 链接顺序文件的查找

三、索引文件及其查找

1. 索引文件的基本概念
   - 索引与索引表
   - 索引表的特点
2. 索引文件的查找

   稠密索引文件与非稠密索引文件的查找

四、B-树与B+树

1. B-树的结构
2. B-树的查找
3. B-树的插入（结点的分解原则）
4. B+树的结构
5. B-树与B+树的异同

五、散列(Hash)文件及其查找

1. 散列文件的基本概念

- 散列函数及其构造方法(原则)
- 散列冲突

2. 散列冲突的处理方法

- 开放地址法
- 再散列法
- 链地址法