第六章　集合与字典

基本内容：

- 集合
- 字典的基本概念
- 顺序表检索
  - 顺序检索
  - 二分法检索
  - 分块检索
- 散列表检索
- 树表检索
  - 二叉树表示
  - AVL树表示
  - B树表示

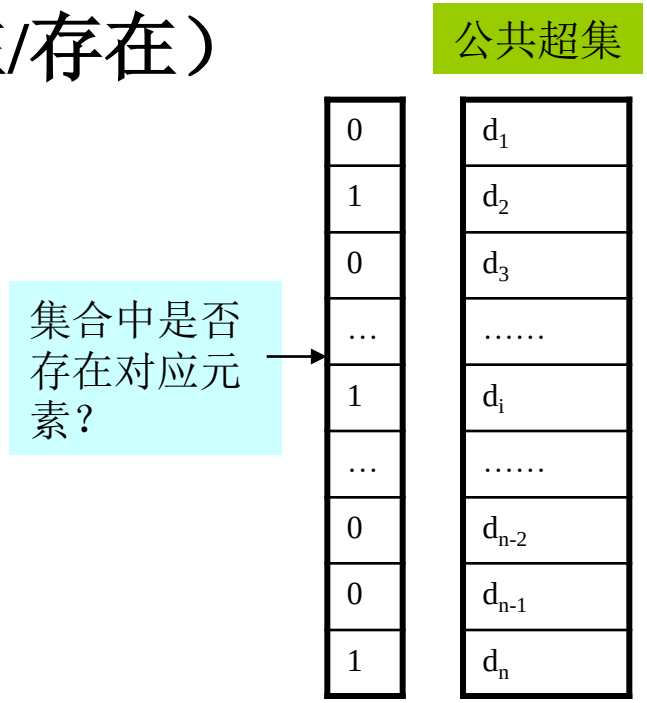
重点：各种检索方法的思想、算法、时间效率度量

6.1 集合

1. 基本概念：集合、成员、空集、有序集
2. 主要运算：并、交、差、子集、相等
3. 集合的实现：顺序（位向量），链接（单链表）

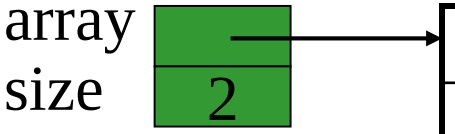
元素是否存在的位向量：0/1（不存在/存在）

```
typedef struct
{
    int size;          //字符数组长度
    char *array; //位向量空间，
                       //一个字节表达8个元素
} BitSet
```



S={1,3,5,7,9}

7	6	5	4	3	2	1	0
						9	8



1	0	1	0	1	0	1	0
						1	0

元素个数与字符数组长度的关系：

size = (n+7)/8

每个字符8位，最后一个字符可能不满。

给定成员位置idx，确定在那个字符中：

which_char = idx >> 3;　　//即：idx/8

确定哪一位：

which_bit = idx & 07;　　//即：idx%8 (后三位)

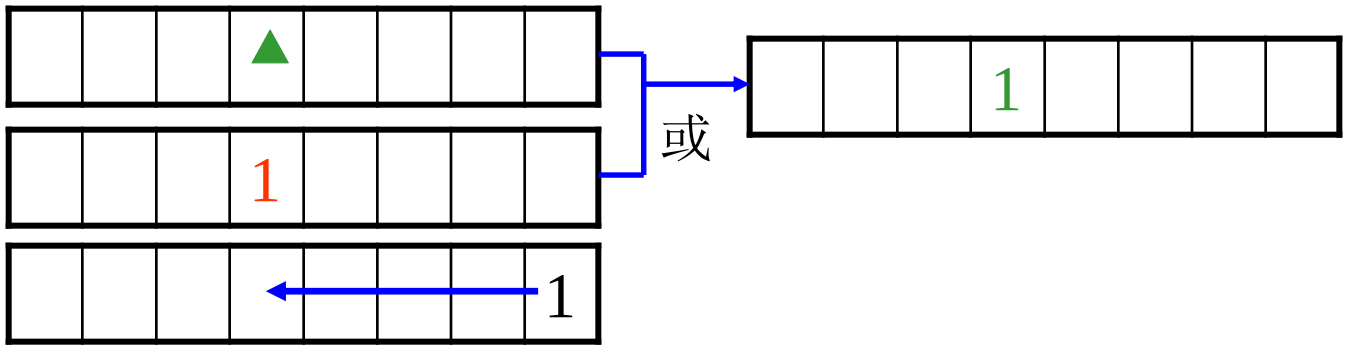
创建空集合：

```
BitSet *CreateEmptySet(int n)
{   BitSet *s = (BitSet *)malloc(sizeof(BitSet));
    if (!s) return NULL;
    s->size = (n+7)/8;
    s->array = (char *)malloc(s->size);
    memset(s->array, 0, s->size);
    return s;
}
```

插入运算：

```
int Insert(BitSet *s, int idx)
{
    if (idx > = 0 && (idx >>3 < s->size))
    {
        s->array[idx >> 3] |= (1 << (idx & 07));
        return 1;
    }
    return 0;
}
```

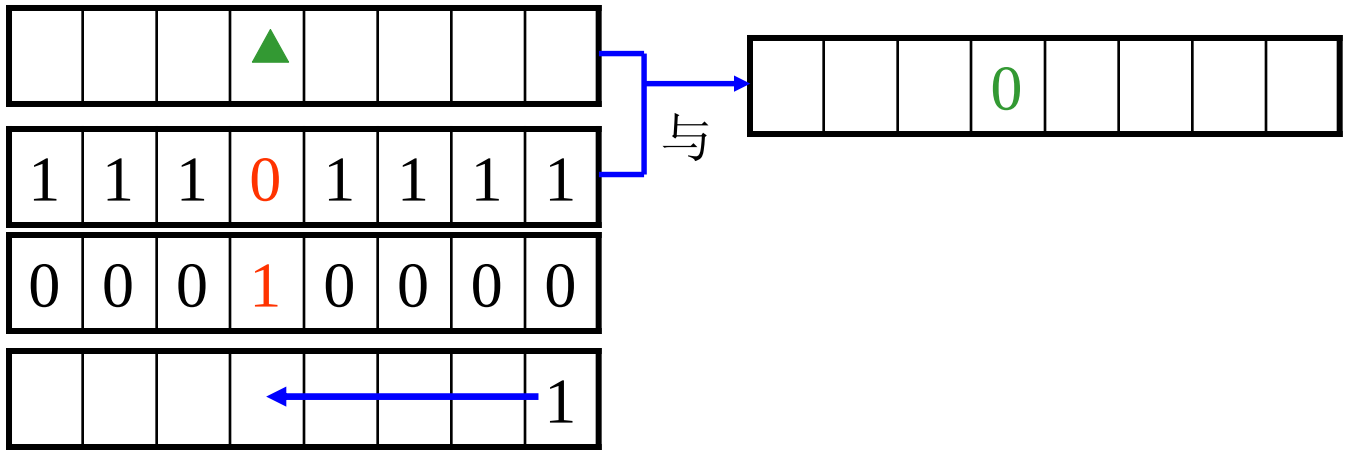
模板，对应位置1



删除运算：

```
int Delete(BitSet *s, int idx)
{
    if (idx > = 0 && (idx >>3 < s->size))
    {
        s->array[idx >> 3] &= ~(1 << (idx & 07));
        return 1;
    }
    return 0;
}
```

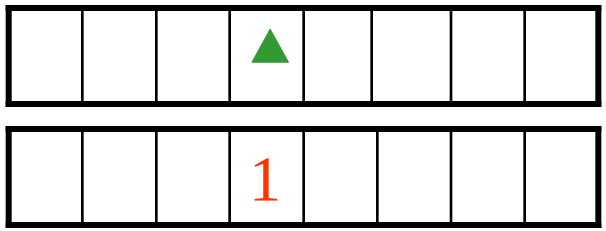
模板，对应位置0



属于判断：

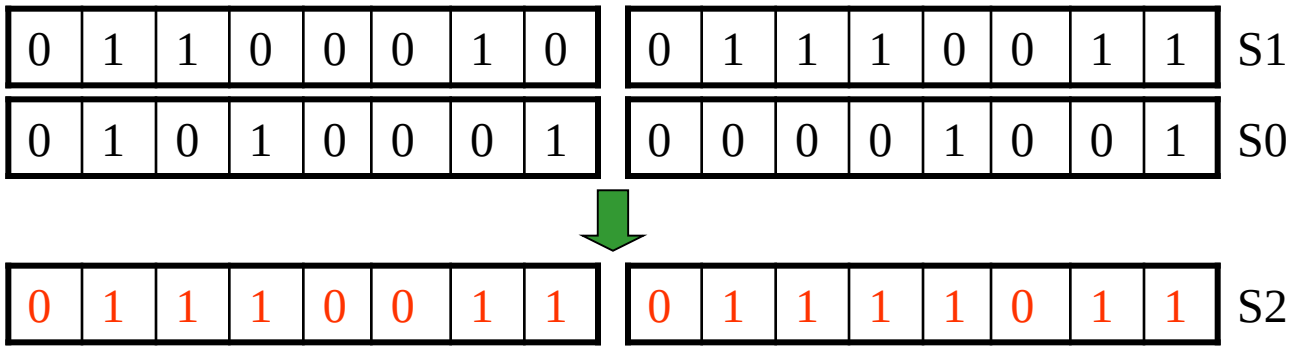
```
int Member(BitSet *s, int idx)
{
    if (idx > = 0 && (idx >>3 < s->size))
    {
        return (s->array[idx >> 3] & (1 << (idx & 07)));
    }
    return 0;
}
```

1与对应位“与”运算



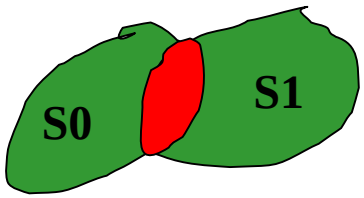
集合并：通过对应位“或”运算完成

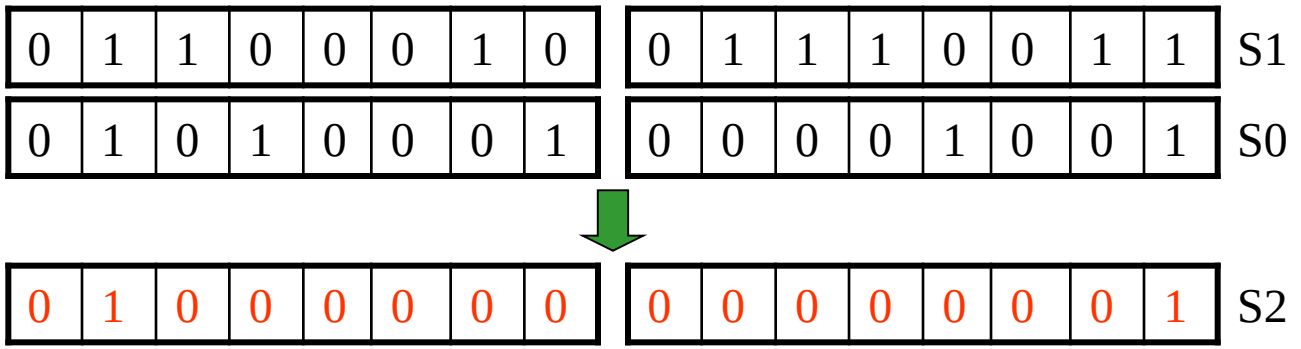
```
int Union(BitSet *s0, BitSet *s1, BitSet *s2)
{   int i;
    if (s0->size != s1->size || s2->size != s1->size) return 0;
    for (i = 0; i < s1->size; i++)
    {
        s2->array[i] = s0->array[i] | s1->array[i];
    }
    return 1;
}
```



集合交：通过对应位“与”运算完成

```
int Intersection(BitSet *s0, BitSet *s1, BitSet *s2)
{   int i;
    if (s0->size != s1->size || s2->size != s1->size) return 0;
    for (i = 0; i < s1->size; i++)
    {
        s2->array[i] = s0->array[i] & s1->array[i];
    }
    return 1;
}
```

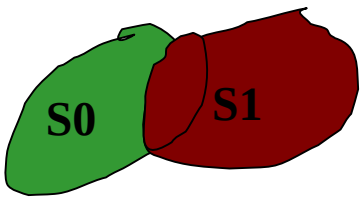


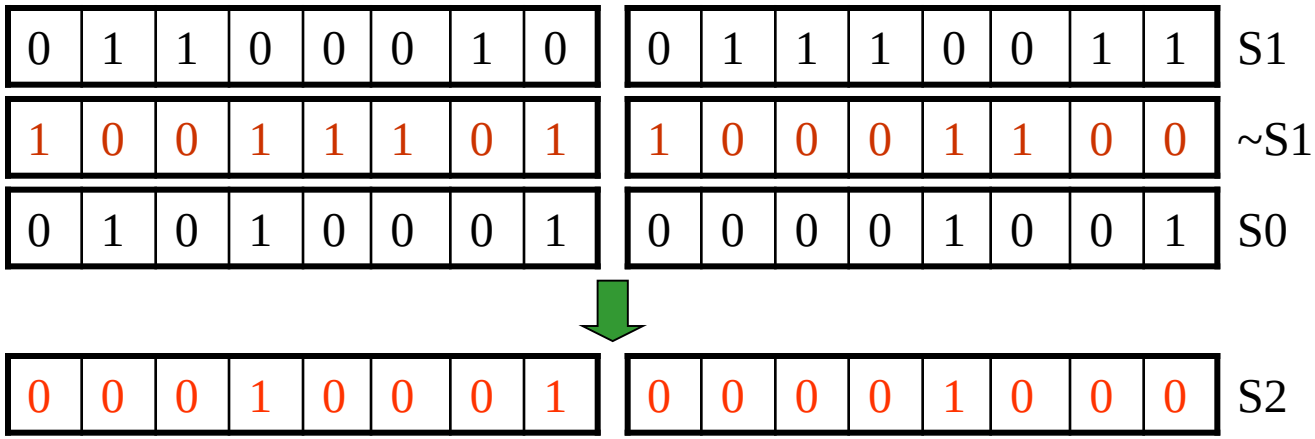


集合差：在第一个集合中，同时不在第二个集合中

第一个集合与第二个集合的逆做“与”运算

```
int Difference(BitSet *s0, BitSet *s1, BitSet *s2)
{   int i;
    if (s0->size != s1->size || s2->size != s1->size) return 0;
    for (i = 0; i < s1->size; i++)
        s2->array[i] = s0->array[i] & ~(s1->array[i]);
    return 1;
}
```





单链表：有序单链表，各个结点存放成员本身。

```
struct Node
{
    DataType info;
    struct Node *link;
};
typedef struct Node *PNode;
```

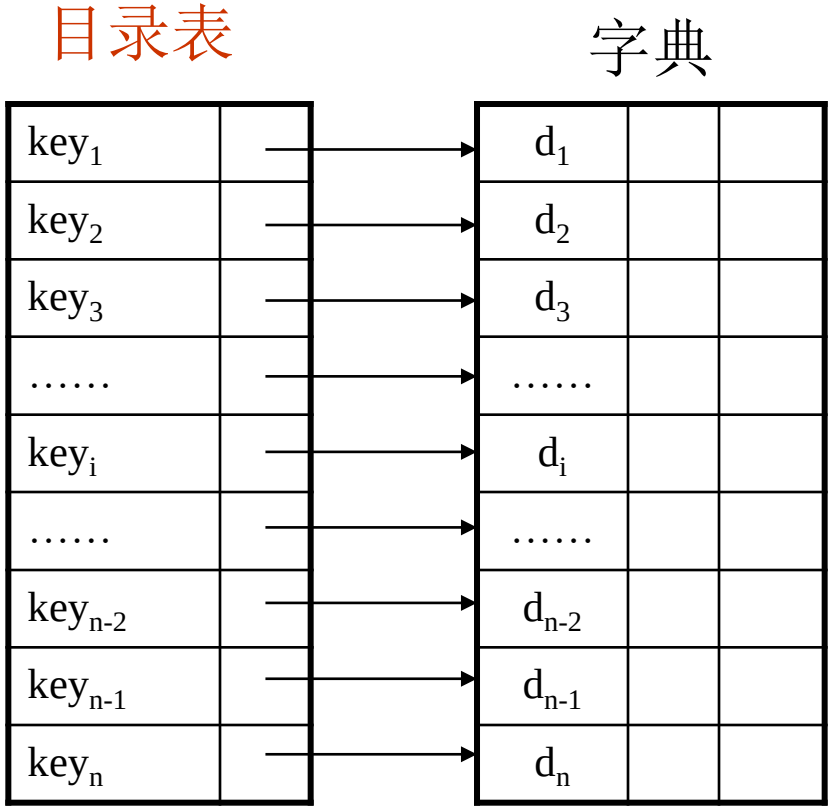
基本运算实现：p177~180

注意两个有序单链表运算如何实现集合运算。

6.2 字典的基本概念

1. 字典的组成： 关键码+属性

2. 字典与索引：



字典相关的运算（排序、检索等），直接在字典的目录表（索引）下进行，不用关心字典本身。

3. 检索： 按照关键码进行

给定一个值key，按照某种次序在字典中进行检索（与关键码进行比较），如果找到则检索成功，否则失败。

4. 存储结构：静态、动态

静态：一旦建立，不再修改，只有检索运算

动态：除检索外，字典经常变化（插入、删除）

5. 衡量检索算法的标准

平均查找长度

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

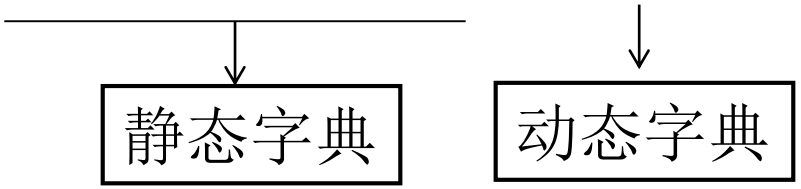
p_i为查找第i个元素的概率，

c_i为查找第i个元素的比较次数

如不特别声明为等概率查找，$p_i=1/n$

6. 检索中的基本运算： 关键码与给定值的比较

7. 字典（索引）的表示：顺序表，散列表，树表



6.3 顺序表检索

KeyType: int
DataType: int

顺序检索、二分法检索、分块检索

1. 顺序表的表示

```
typedef struct                  /* 元素结构 */
{   KeyType  key;               /* 元素的关键码 */
    DataType other;             /* 其它属性字段 */
} DicElement;
typedef struct                  /* 字典结构 */
{   DicElement  element[MAXNUM];    /* 字典数组 */
     int  n;                    /* 实际元素个数 */
} SeqDictionary;
```

2. 顺序检索

（1）思想：从字典一端开始顺序扫描，将字典元素的关键码与给定值进行比较。如果相等，则检索成功；当全部元素皆比较完，到达了字典的另一端，则检索失败。

（2）算法：

```
int SeqSearch( SeqDictionary *pdic, KeyType key, int *pos)
{   int i;
    for ( i = 0; i < pdic->n; i++)              /* 从头开始 */
    {
        if (pdic->element[i].key == key)  /* 成功 */
        {   *pos = i;        return TRUE;     }
    }
    *pos = i;    return FALSE;                  /* 失败     */
}
```

(3) 时间效率分析

从算法可以看出：$c_1 = 1, c_2 = 2, \ldots, c_i = i, \ldots$

(找到第i个元素），

因此，查找成功 $$ASL = \sum_{i=1}^{n} i.p_i$$

如果等概率查找，则$p_i = 1/n$,

ASL=(1+2+…+i+…+n)/n=(n+1)/2

查找失败的比较次数： n次

时间复杂度为：O(n)

(4) 不等概率情况下的改进：

在不等概率情况下，当$P_1 \geq P_2 \ldots \geq P_n$时，ASL最小。

因此，在不等概率时，应该保持概率最大的元素在最前面，概率最小的元素在最后面。

通常，无法预先知道各个元素的查找概率，解决的办法是为用检索成功次数代替查找概率，当检索元素成功时，其检索成功次数加1，保持检索成功次数最大的元素在前面，检索成功次数最小的元素在最后面。

Rs	…								…
Cmax	…								Cmin

(5) 顺序检索的优缺点：

优点：算法简单，适应面广，对字典结构无要求，无论是否按关键码有序，适用于顺序表和链表。

缺点：ASL长，O(n)数量级，大约一半元素个数的比较次数。

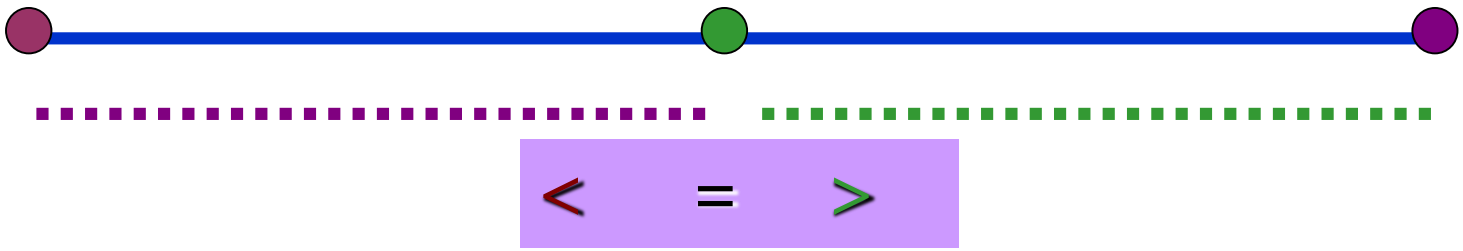
3. 二分法检索（折半检索）

（1）思想：

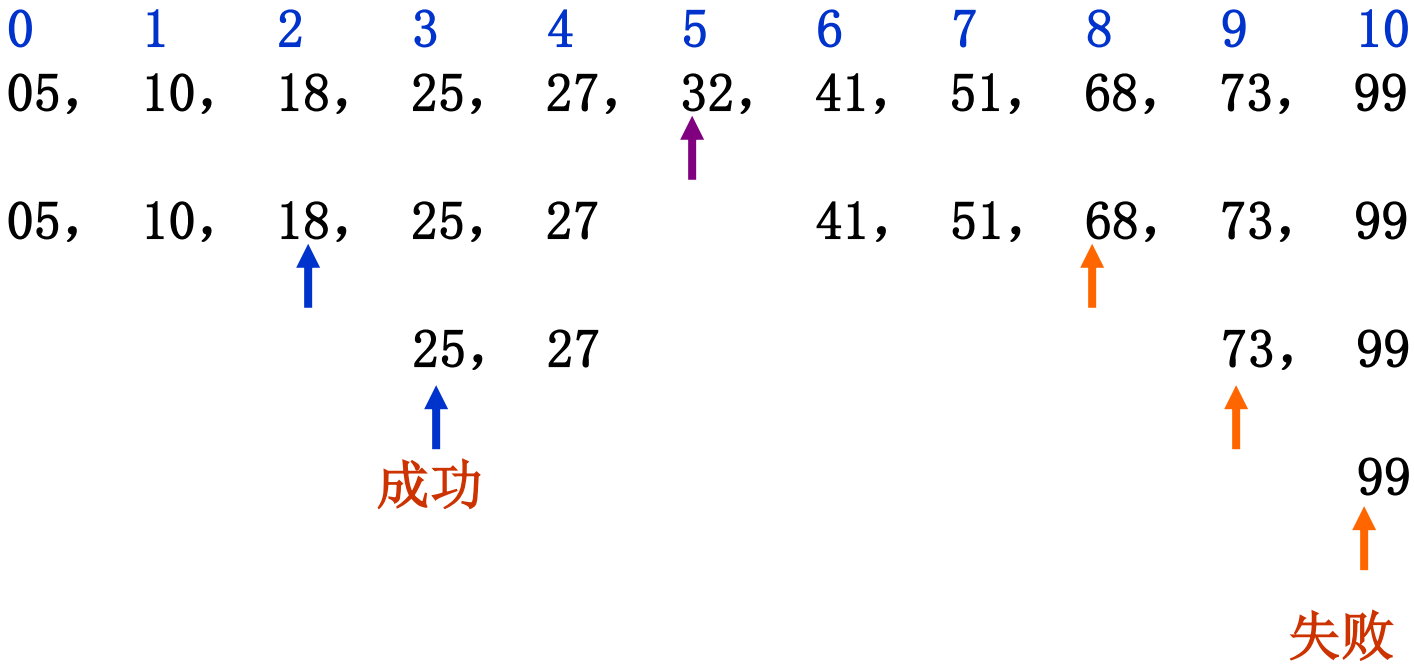
当字典按关键码有序时，将字典元素按关键码从小到大保存在数组中。

首先给定值key与字典的中间位置元素进行比较，如果相等，则检索成功；否则根据key与中间位置元素的关键码的比较确定在左边进行继续查找，还是在右边继续查找。如key小，则在左边继续进行二分法检索，否则，在右边继续二分法检索。

从上面可以看出，折半检索的实质是逐步缩小查找区间的检索方法。



例如给定值key = 25和78的检索：



需要指定搜索区间。对于顺序表：[低界，高界]

（2） 算法

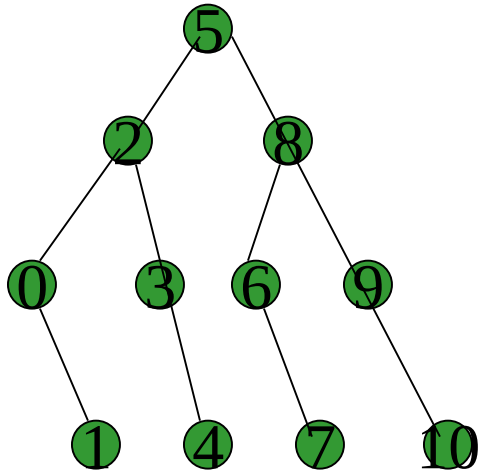
```
int BinSearch( SeqDictionary *pdic, KeyType key, int *pos)
{
    int low, mid, high;
    low = 0;  high = pdic->n-1;                    /* 初始搜索区间 */
    while (low <= high)
    {
       mid = (low+hight)/2;                        /* 中间位置 */
       if (key == pdic->element[mid].key)  /* 查找成功 */
       {    *pos = mid;   return TRUE;     }
       else if (key < pdic->element[mid].key)
       {    high = mid-1;   }                      /* 左边区间继续 */
       else
       {    low = mid+1;   }                       /* 右边区间继续 */
    }
    *pos = low;   return FALSE;                    /* 查找失败 */
}
```

（3） 时间效率分析

每比较一次缩小一半的查找区间。为了分析折半检索的性能，可以用二叉树来描述折半检索的过程。当前检索区间中间记录作为根，左半区间和右半区间中的记录分别为根的左子树和右子树，这样得到的二叉树称为折半检索二叉树。

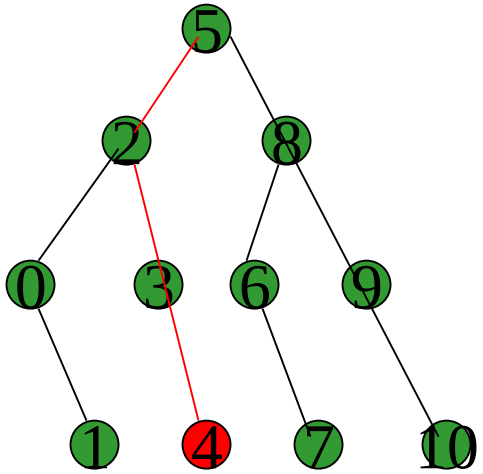
树中结点数字表示结点在有序表中的位置。

右图为11个记录的折半检索二叉树。如要检索第6个记录，则只需要一次比较；检索第3、9个记录需要2次比较；检索第1、4、7、10个记录需要3次比较；检索第2、5、8、11个记录需要4次比较。



由此可见，折半检索的过程恰好是在判定树中走了一条从根到检索结点的路径，比较的关键码个数恰为该结点在二叉树中的层数。因此，成功折半检索关键码比较次数不会超过树的深度：$\lceil \log_2(n+1) \rceil$

那么，折半检索的**ASL**等于多少？



假定记录数$n=2^h-1$， 即$h=\log_2(n+1)$，则描述折半检索的判定树是一棵深度为h的全满二叉树。在该全满二叉判定树中，有：

第1层的结点个数为2^0， 比较1次；
第2层的结点个数为2^1， 比较2次；
……
第k层的结点个数为2^{k-1}，比较k次；
……
第h层的结点个数为2^{h-1}，比较h次；

等概率检索下，检索成功时的

$$ASL=\sum_{k=1}^{h}p_kc_k=\frac{1}{n}\sum_{k=1}^{h}k\times 2^{k-1}=\frac{n+1}{n}\log_2(n+1)-1+\frac{1}{n}$$

当n很大时，如n>100，则得到: $ASL\approx\log_2(n+1)-1$

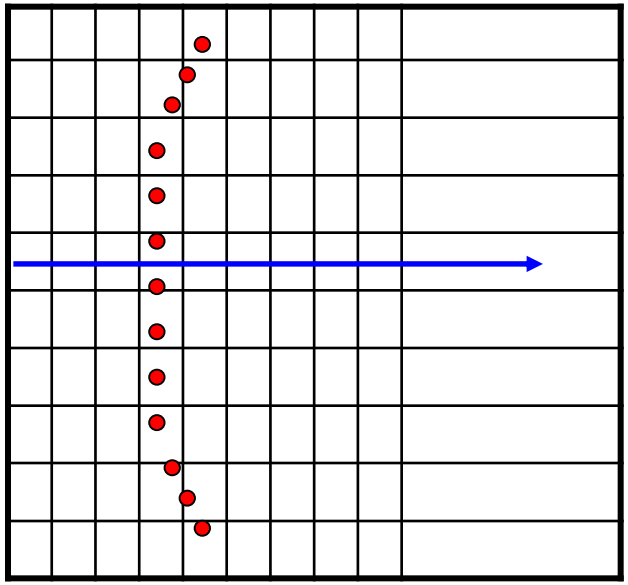
由此可见，折半检索的效率要高于顺序检索。如n=1000，顺序ASL=500，而折半ASL≈9。

（4）优缺点

优点：检索效率高， $ASL \approx \log_2(n+1)-1$

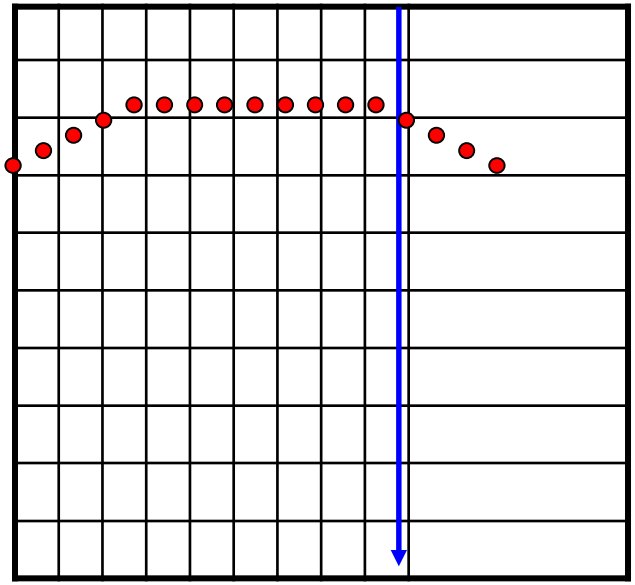
缺点：只适用于数据元素很少变动的有序顺序表，应用范围有限制。

经度数组



每行中：列按照经度有序

纬度数组



每列中：行按照纬度有序

问题描述：

地球观测网格点m行m列，网格点经纬度位置分别存储在经度数组和纬度数组中。对于经度数组，各行按照列经度有序；对于纬度数组，各列按照行纬度有序。

算法设计：

给定一个经纬度值（lon, lat），检索最近网格点的行列坐标。

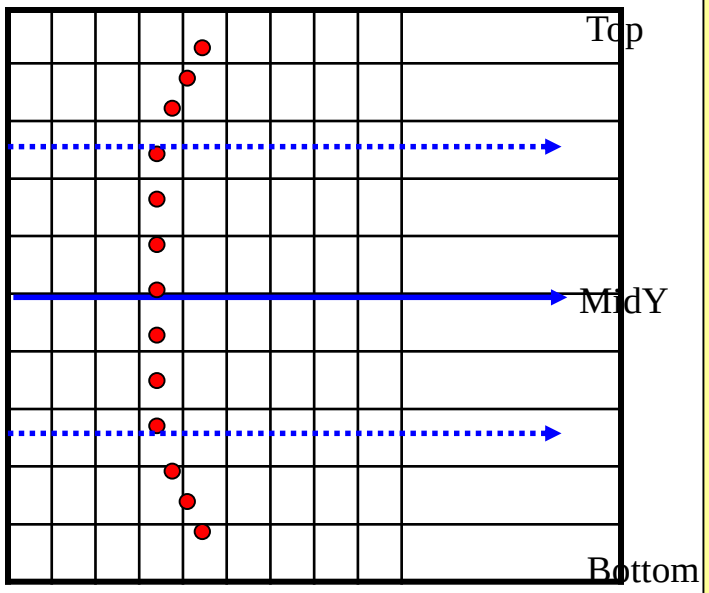
void geo2grid(float lon, float lat, int *gx, int *gy);

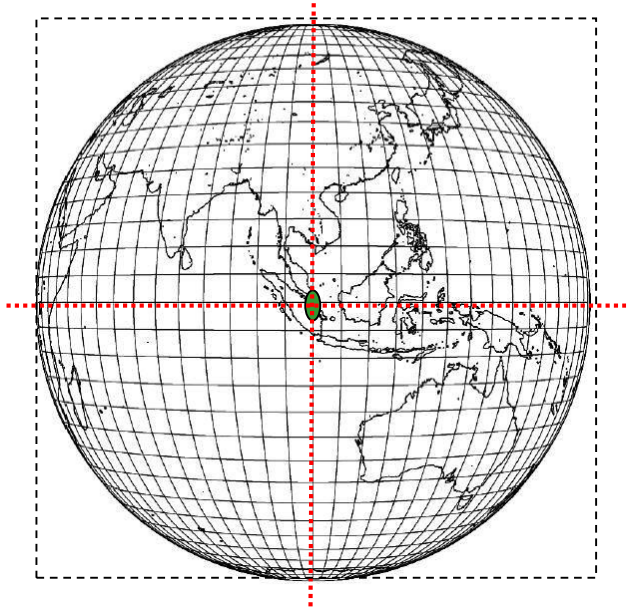
------双折半实现，速度较顺序检索，大大提高。

顺序：$m^2/2$，　双折半：$(\log_2(m+1)-1)^2$，分区双折半：$(\log_2(m/2+1)-1)^2$。

如：m = 2000, ASL ≈ 【2000000, 100, 82(81+1)】

经度数组(每行中：经度有序)





```
void  geo2grid(float lon, float lat, int *gx, int *gy)
{
    int top = 0, bottom = rows-1, midy;
    int left, right, midx;
    while (top <= bottom)          //纬度控制的行折半检索
    {
        midy = (top+bottom)/2;  //中间行
        left = 0, right = cols-1;    //在中间行中折半经度检索
        while (left <= right)
        {
            midx = (left+right)/2;                    //中间列
            if (lon == lons[midy][midx]) break; //找到列
            else if (lon < lons[midy][midx])
                  right = midx-1;
            else left = midx+1;
        }
        if (lat == lats[midy][midx]) break;       //成功
        else if (lat < lats[midy][midx])
             bottom = midy-1;
        else top = midy+1;
     }
     *ix = midx;  *iy = midy;   //近似解
}
```

进一步改进: (lon,lat)先与中心点经纬度进行比较，确定在那个1/4区域，再在该区域内双折半检索。

4. 分块检索(Blocking Search)

（1）数据元素排列

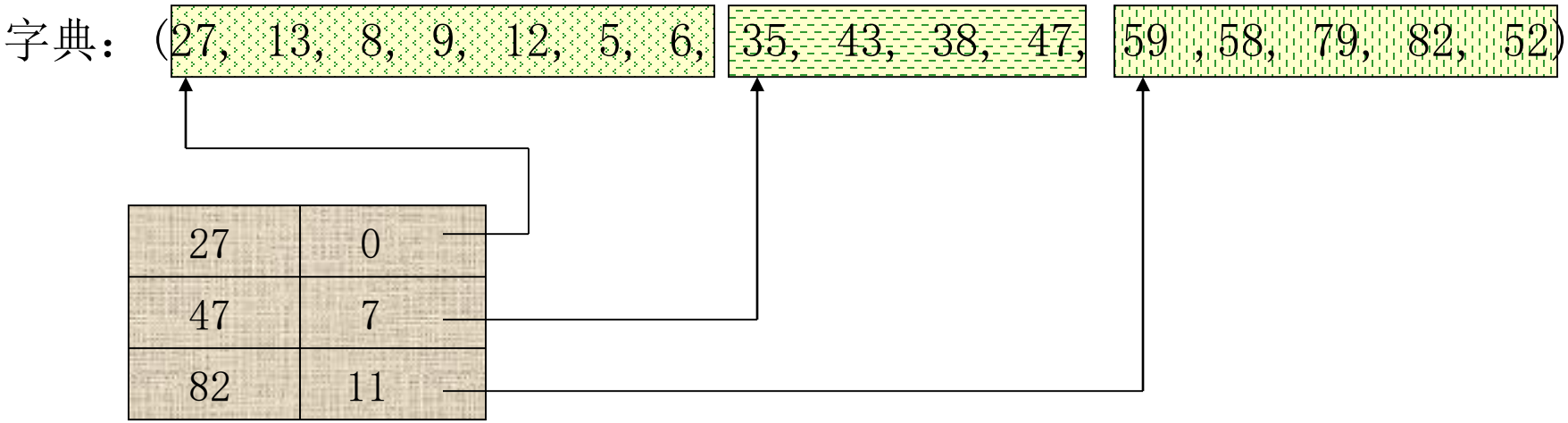
又称索引顺序检索，是介于顺序、折半之间的检索方法。

分块检索要求所有的记录分成若干块，并且块间按关键码有序，而块内不一定有序。即：第一块中所有记录的关键码均不超过第二块中任意记录的关键码，而第二块中所有记录的关键码均不超过第三块中任意记录的关键码，……，块内记录可以任意排列。

这样，可以建立一个块索引表，存储每块的最大关键码值以及该块第一个元素在字典中的位置。

块索引表数据元素结构：

最大关键码值	第一个记录的位置



块索引表

5	6
35	10
52	15

最小关键码值	最后一个记录的位置

（2）检索思想

给定值key, 首先在块索引表中进行检索确定下一步要检索的块，然后到确定的块中进行检索。由于块索引表按关键码有序，因此块间检索可以采用折半检索，也可以采用顺序检索。但由于块内记录无序，任意排列，只能采用顺序检索。

由此可以看出，分块检索有两步组成：块间检索和块内检索。

如上例中检索key=38，首先在块索引表中检索发现在第二块，然后在第二块中顺序检索，得到记录是第二块的第三个记录（总的字典位置为7+2)。

又如key=15，有块索引表得到如果记录存在，必定在第一块。在第一块中顺序检索失败，因此字典中不存在这样的记录，分块检索失败。

（3）检索算法

索引表结构

```
typedef struct
{    KeyType maxkey;      /* 块中最大关键字 */
     int     pos_1rec;    /* 块的第一个元素的位置 */
} IndexNode;

typedef struct
{    IndexNode idx_block[MAXNUM];  /* 块索引表 */
     int    num_block;             /* 实际块数 */
} *Pindex;
```

```
int BlockSearch(SeqDictionary *pdic, Pindex idx, KeyType key, int *pos)
{   int i=0, j, last_j;

   //在块索引表中确定块
   while (i<idx->num_block && key > idx->idx_block[i].maxkey)
          i++;
   if (i > idx->num_block)  *pos = -1;              //块间检索失败
   else
   {   //块内顺序检索，确定检索范围
       j = idx->idx_block[i].pos_1rec;                    //起始记录
       if (i == idx->num_block)   last_j = pdic->n-1;  //终止记录
       else  last_j = idx->idx_block[i+1].pos_1rec-1;
       while (j <= last_j && key != pdic->element[j].key)
              j++;
       if ( j > last_j)    *pos = -1;                  //块内检索失败
       else  *pos = j;                                 //检索成功
   }
```

```
    //检索成功或失败
    if (*pos > -1)  return TRUE;
    else            return FALSE;
}
```

（4）检索效率 $ASL_{bs} = L_b + L_w$

其中，L_b为块间检索的ASL，L_w为块内检索的ASL
假定总共有n个记录，分成b块，每块含有s个记录，即s=n/b。
假定块间检索为等概率：1/b，块内检索仍然等概率：1/s。根据块间检索方法不同，ASL计算不同。

- 块间顺序检索

$L_b = (b+1)/2$　　$L_w = (s+1)/2$

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

由此可见，ASL不仅与n有关，而且与每块记录数目s有关。对上式ASL求导，可以得到当 $s=\sqrt{n}$ 时，ASL取最小值 $\sqrt{n}+1$。

如：n=1000时，$s=\sqrt{n}\approx 31$，ASL ≈32

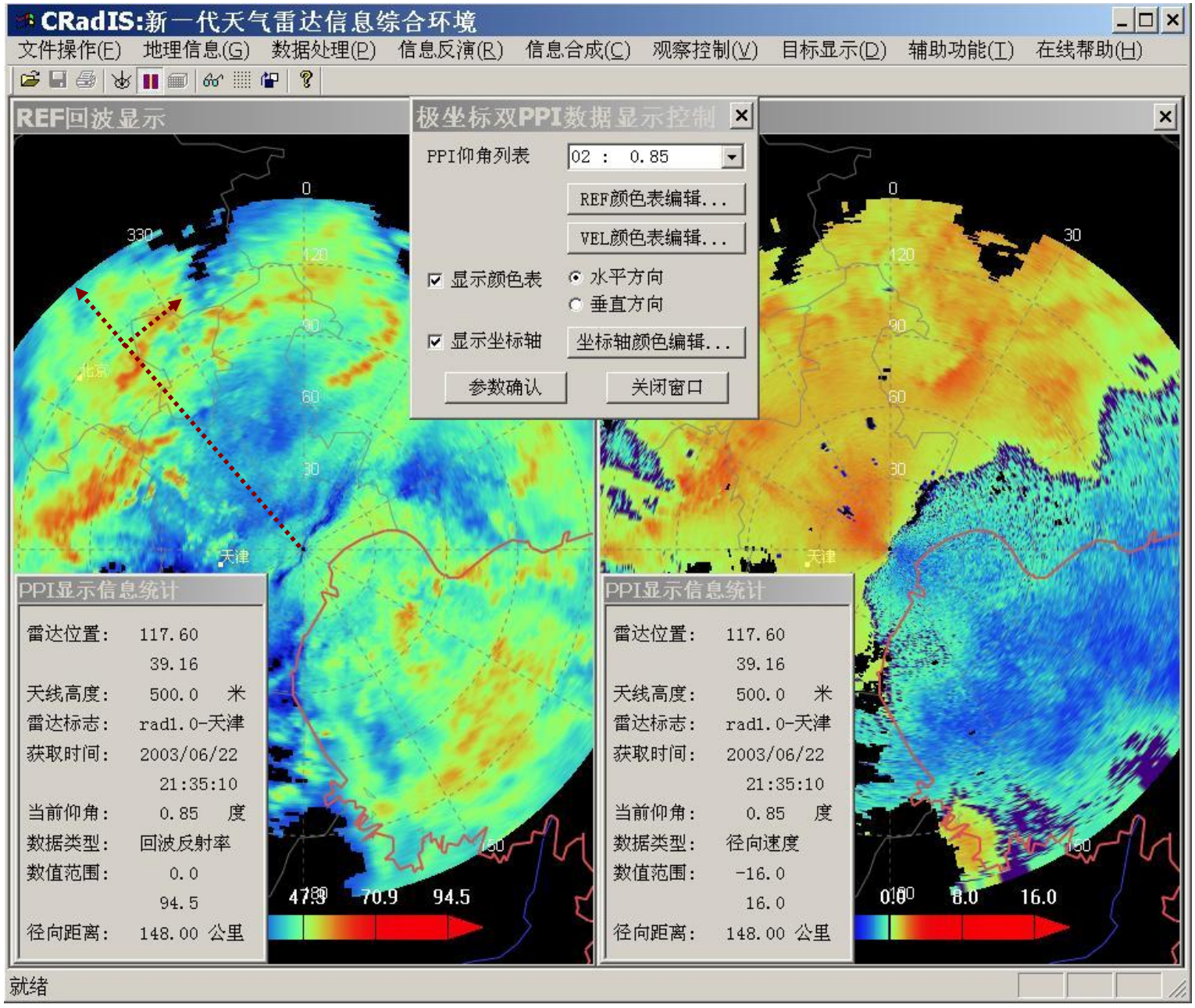
顺序检索： ASL ≈500

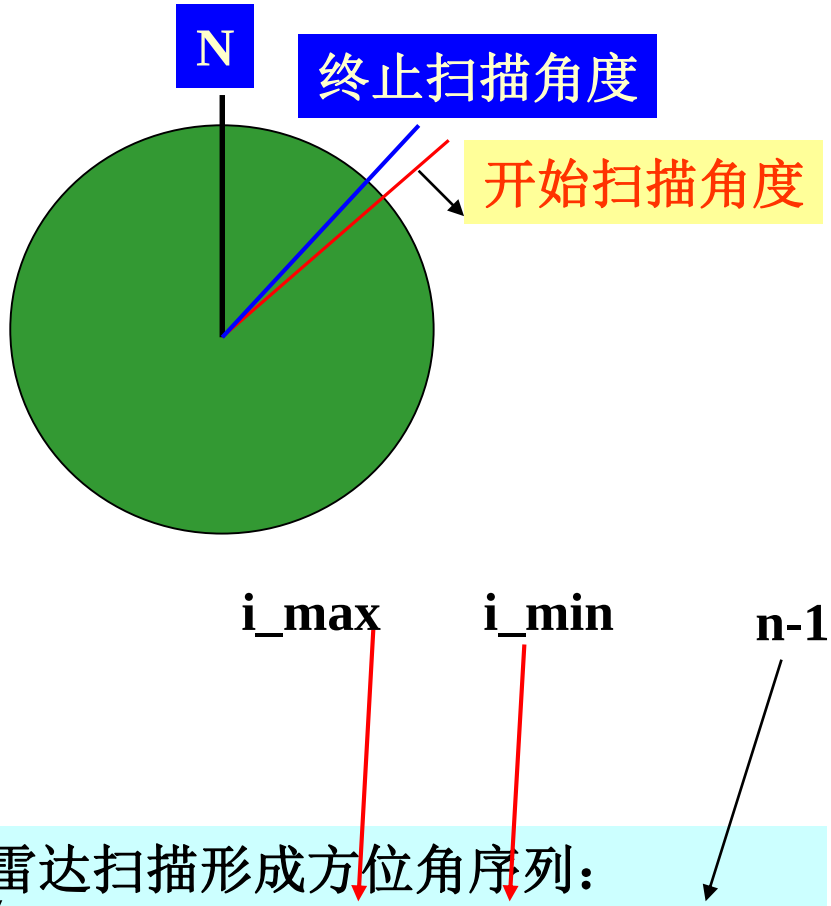
折半检索： ASL ≈9

因此，分块检索介于顺序、折半检索之间。

- 块间折半检索

$ASL = \log_2(n/s+1)-1+(s+1)/2 \approx \log_2(n/s+1)+s/2$


CRadIS:新一代天气雷达信息综合环境
文件操作(F)
地理信息(G)
数据处理(P)
信息反演(R)
信息合成(C)
观察控制(V)
目标显示(D)
辅助功能(I)
在线帮助(H)
REF回波显示
极坐标双PPI数据显示控制
PPI仰角列表
02 : 0.85
REF颜色表编辑...
VEL颜色表编辑...
显示颜色表
水平方向
垂直方向
显示坐标轴
坐标轴颜色编辑...
参数确认
关闭窗口
PPI显示信息统计
雷达位置: 117.60
39.16
天线高度: 500.0 米
雷达标志: rad1.0-天津
获取时间: 2003/06/22
21:35:10
当前仰角: 0.85 度
数据类型: 回波反射率
数值范围: 0.0
94.5
径向距离: 148.00 公里
PPI显示信息统计
雷达位置: 117.60
39.16
天线高度: 500.0 米
雷达标志: rad1.0-天津
获取时间: 2003/06/22
21:35:10
当前仰角: 0.85 度
数据类型: 径向速度
数值范围: -16.0
16.0
径向距离: 148.00 公里
天津
47.3
70.9
94.5
8.0
16.0
就绪



雷达扫描形成方位角序列：

$a_0, a_1, a_2, \ldots, 359.88, 0.01, \ldots, a_0-0.1$

分成两块：

$a_0, a_1, a_2, \ldots, 359.88$

$0.01, \ldots, a_0-0.1$

```
int find_bin(float azim)
{
     int si, ei, mi;
     if (azim >= a[0])
    {    si = 0; ei = i_max; }
    else
    {    si = i_min, ei = n-1; }
    while (si <= ei)
    {
          mi = (si+ei)/2;
          if (azim == a[mi])
               return mi;
          else if (azim < a[mi])
               ei = mi-1;
          else si = mi+1;
    }
    return mi;
}
```

块间有序、块内有序的分块检索应用

索引存储结构及其检索：

1）密集索引:每个元素一个索引

顺序表

链表

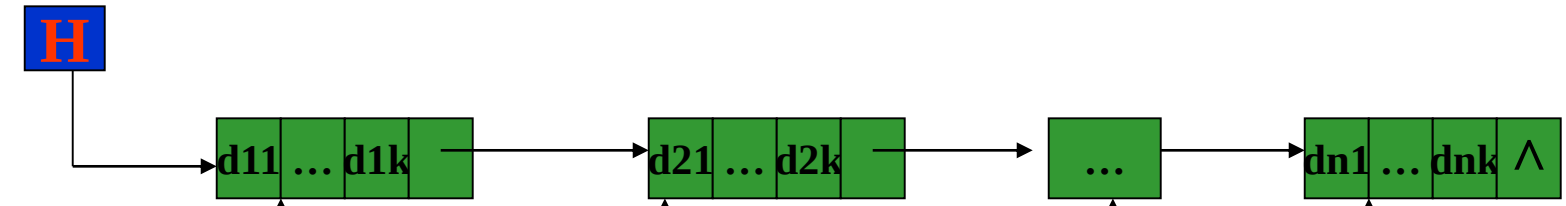
顺序表

d11	…	d1n
d21	…	d2n
dk1	…	dkn

索引

k1	
k2	
…	…
kn	

链表



索引

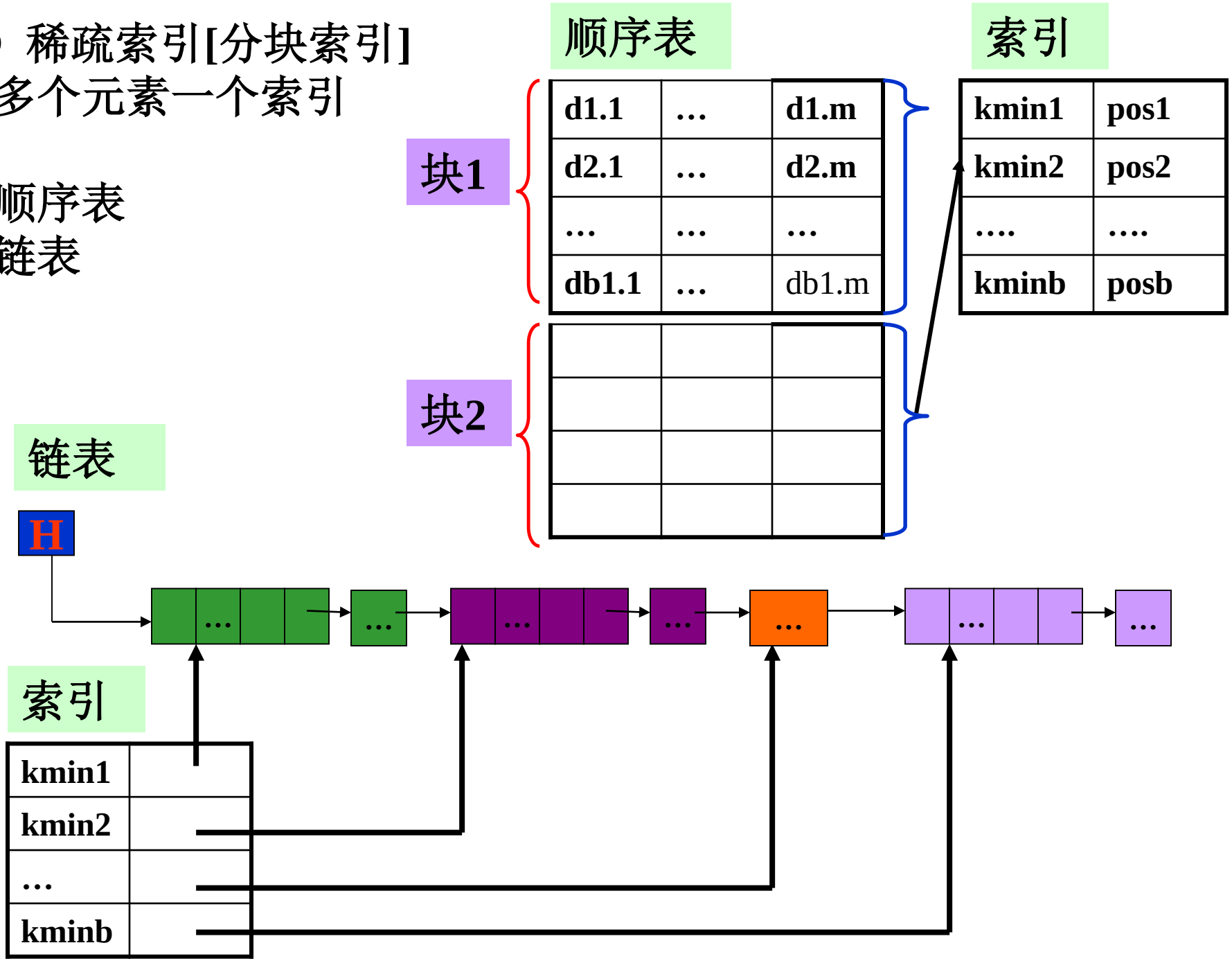
k1	
k2	
…	
kn	

2）稀疏索引[分块索引]

多个元素一个索引

顺序表

链表



6.4 散列表检索

1. 思想与基本概念

前面的检索方法都是通过给定值与关键码的比较才能确定记录位置，检索效率与检索过程中进行的比较次数有关。本节介绍的Hash表检索是一种基于“尽可能不通过比较操作而直接得到记录的存储位置”的想法而提出来的检索方法。

其基本思想是以记录关键码key为自变量，通过某种确定的函数h(key)计算的函数值作为存储地址，将该记录（或记录的关键码）存放到该位置上。

检索时仍然需要h(key)计算得到关键码所在记录的存储地址。显然，按照这种思路可以不需要比较运算就能直接确定记录。

散列函数（哈希函数）： h(key)
散列地址（哈希地址）： h(key)的值
散列表（哈希表）： 通过h(key)建立起来的线性表，称为Hash表

到目前为止，已经接触了四种存储结构：
顺序、链接、索引、散列。

给定一个散列函数h，可能出现两个或多个不同的关键码具有相同的散列地址，称这些不同的关键码为“同义词”，而这种现象称为“碰撞”。

对于任意的散列函数，都可能出现“碰撞”现象。因此，如何选择合适的散列函数减少碰撞发生是Hash表构造中的一个关键问题。

通过散列函数得到的散列地址空间称为“基本区域”，发生碰撞时，同义词可以放在基本区域中未被占用的空间，也可以放到基本区域以外另开辟的区域（溢出区）。

负载因子：$$\alpha = \frac{\text{散列表中结点数目}}{\text{基本区域能容纳的结点 数}}$$

当α>1时，碰撞不可避免。

由此可以看出，散列表的构造和检索中，需要解决的主要问题有：

（1）如何选择合适的散列函数构造散列表？

（2）碰撞发生时如何处理？

（3）散列表如何检索？检索效率如何？检索效率与那些因素有关？

2. 散列函数的选择

散列函数的选择在散列表的构造和检索中是非常重要的，合适的散列函数可以减少碰撞的发生，提高检索效率。

散列函数的选择标准：

- 字典中的关键码均匀分布在散列地址空间中，减少碰撞的发生；
- 散列函数尽可能简单，提高计算速度。

常用的散列函数：

（1）直接定址法

$$h(key) = a*key+b$$

适用于关键码分布基本连续情况。若关键码分布不连续，容易造成空单元，存储空间浪费。

（2）除留余数法

h(key) = key%p

p的选择非常重要，通常选择小于等于散列表长度m的某个最大素数。

如：m = 600，则p可以选择为599。

除留余数法计算简单，许多情况下效果较好，是一种常用的散列函数。

（3）数字分析法

对关键码的数字位进行分析，然后用其中的某些较分散的数字位构成散列函数。

通常用于已知记录关键码，并且关键码各位分布已经知道的情况。

例如：(395003, 395010, 395012, 395085, 395097)中，前四位相同，后两位随机分布，故可以取后两位构成散列函数，得到的散列地址为（03，10，12，85，97）

（4）折叠法

如果关键码的位数多于地址位数，且各位分布均匀，此时可以将关键码分成若干块，块长度不超过地址位数。各块相加，舍弃进位，最后的和作为散列地址。

如：**key = 0582422241**，地址位数为**4**。

```
   2241
   8242
+)   05
-------
 [1]0488
```

移动叠加

```
   2241
   2428
+)   05
-------
   4672
```

间界叠加

（5）平方取中法

先求出关键码的平方，然后取中间若干位构成散列函数。

例如：key = 4731，key^2 = 22382361，如地址码为3位，则可以取382为散列地址，即h(4731) = 382

（6）基数转换法

将关键码首先看作是另一进制的表示，然后再转换为原来的进制数，用数字分析法取若干位作为散列地址。一般转换基数大于原基数，且最好二者互素。

例如： key = $(236075)_{10}$, 把它看作13进制数$(236075)_{13}$,再转换为10进制，地址位数为4：

$(236075)_{13} = 2*13^5+3*13^4+6*13^3+7*13^1+5 = (841547)_{10}$

选择2~5，得到散列地址h(236075) = 4154

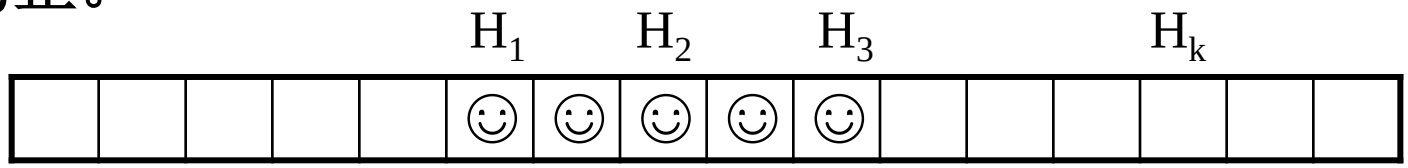
上面为常用的散列函数选择，实际情况应该根据问题的要求选择适合散列表构造和检索的散列函数。

从上面的散列函数选择可以看出，无论选择何种散列函数，碰撞都可能发生。换句话讲，合适的散列函数可以减少碰撞发生的几率，但不能保证不发生碰撞。碰撞发生时如何处理？？

3. 碰撞的处理（开放地址法，拉链法）

(1) 开放地址法（基本区域内解决）

假定散列表地址空间为[0，m-1]，碰撞指的是由关键码得到的散列地址为j(0≤j≤m-1)的位置上已经存在记录，则处理碰撞就是为该关键码找到一个空的散列地址。找到该空的散列地址可能需要经过一个地址序列H_i(i=0,1,2,…,k)，即在处理碰撞时，如果得到的散列地址H_1仍然冲突，则再求下一个地址H_2，如H_2冲突，再求H_3，…，直到H_k不发生冲突为止。



$H_i = (H(key)+d_i) \% m$　　(i=1,2, …, k [k≤m-1])

其中，H(key)为散列函数，

m为散列表长度，

d_i为增量序列。

d_i有下面三种形式：

- [a] 线性探测： $d_i=1,2,3,\ldots,m-1$
- [b] 平方探测： $d_i=1^2,-1^2,2^2,-2^2,\ldots,k^2,-k^2$ $(k\leq m/2)$
- [c] 随机探测： d_i=伪随机数序列

线性探测容易出现不同的关键码争夺同一个散列地址问题，这种现象称为“二次聚集[堆积]”。

例如：长度为11的散列表已填有关键码分别为17，60，29的记录[散列函数h(key)=key%11]，现第4个记录的关键码为38，由散列函数得到其散列地址为5，由于该地址已经存放关键码为60的记录，产生冲突。

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

如何寻找下一个空的地址？

按照线性探测得到下一个地址为6，仍然冲突，继续探测得到下一个地址为7，仍然冲突，再继续探测下一个地址为8，该地址为空，因此将关键码为38的记录存放在地址为8的单元。

如果按照平方探测方法，则得到地址为4的空地址用于存放关键码为38的记录。

如果按照随机探测方法，则得到地址为3的空地址用于存放关键码为38的记录。

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

在线性探测中可以发现，当表中i、i+1、i+2位置上已填有记录时，散列地址为i、i+1、i+2、i+3的记录都将填入i+3的位置上，此时出现“二次聚集[堆积]”问题。这不利于查找。但是，当散列表未满时总可以找到一个空单元。平方探测和随机探测不能保证一定能找到下一个空单元。

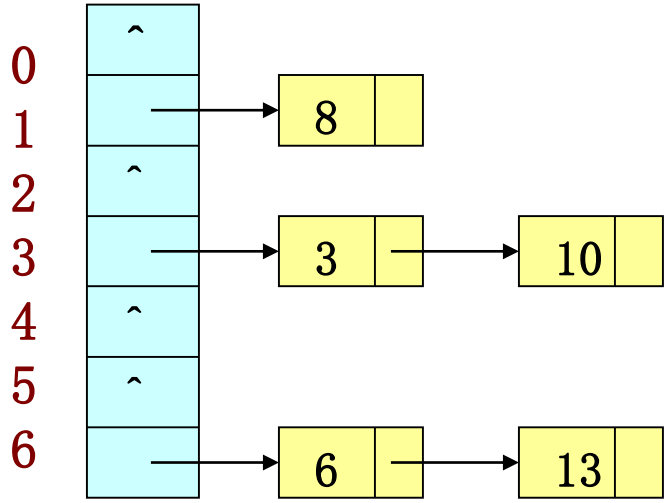
再散列:当发生碰撞时，H_i的计算可以采用其它的散列函数得到。

$$H_i = Rh_i(key),\ i=0,1,2,\ldots,\ k$$

(2)拉链法（溢出区内解决）

将所有关键码为同义词的记录存储在同一个线性链表中。假定由散列函数得到的散列地址空间为[0，m-1]，则可以建立一个指针向量： Chain ChainHash[m]

其初始状态皆为空指针。
散列地址为i的记录都插入到头指针为ChainHash[i]的链表中。
链表中记录排列可以是任意的，也可以按照关键码有序。



如: (8, 13, 3, 6, 10), H(key)=key%7

4. 散列表的建立与检索

散列表的建立与检索是通过相同的散列函数进行的，并且当出现碰撞时应采用相同的解决碰撞的方法。

对于给定关键码值**key**，计算**H(key)**，得到散列地址。

对于散列表的建立，如果该散列地址对应的单元为空，表明没有冲突，直接将记录插入；否则如果该散列地址对应的单元非空，表明该单元已经被前面的记录占用，出现碰撞问题。此时，需要按照解决碰撞的方法进行处理[如果采用开放地址法，则按照一定的序列寻找下一个空单元；如果是采用拉链法，则在碰撞地址对应的同义词链表中完成插入]。

对于散列表的检索，如果该散列地址对应的单元（或指针）为空，表明检索失败；否则：

(1) 如果散列表是采用开放地址法解决碰撞建立的，那么给定值key与该单元存储的记录的关键码进行比较。如果相等，表明检索成功；否则按照散列表建立时寻找下一个单元的序列寻找下一个地址，如此反复进行，直到关键码比较相等（检索成功）或找到空单元（检索失败）为止；

(2) 如果散列表是采用拉练法解决碰撞建立的，那么沿着该非空指针进入同义词单链表，在该链表中进行检索。如果找到某个结点的存储记录的关键码与给定值key相同，检索成功；否则，检索失败。

```
//开放地址法，散列表结构定义：
typedef struct
{
    DicElement element[REGION_LEN];
    int m;                  /* m=REGION_LEN, 基本区域长度 */
} HashDictionary;
```

```
// 使用开放地址法中的线性探测方法解决碰撞的散列表检索
int LinearSearch(HashDictionary *phash, KeyType key, int *pos)
{
    int d, inc;
    d = H(key);                          //计算散列地址
    for (inc=0; inc < phash->m; inc++)
    {
        if (phash->element[d].key == key)   //检索成功
        {  *pos = d;   return TRUE;  }
        else if (phash->element[d].key == NIL)
        //检索失败，找到插入位置
        {  *pos = d;   return FALSE; }
        d = (d+1)%phash->m;     //采用线性探测找下一个探测地址
    }
    *pos = -1;  return FALSE;  //散列表溢出
}
```

```
// 使用开放地址法中的线性探测方法解决碰撞的散列表建立
int LinearInsert(HashDictionary *phash, KeyType key)
{
    int pos;
    if (LinearSearch(phash, key, &pos)== TRUE)
        printf("找到相同的记录！ \n");
        //散列表中已经存在关键码为key的记录，不能再插入
    else if (pos != -1)
        phash->element[pos].key = key;  //插入记录
    else return FALSE;                          //散列表溢出
    return TRUE;
}
```

5. 检索效率分析

从散列表的检索过程可以发现：

（1）虽然散列表在关键码与记录的存储位置之间建立了直接映象，但由于“碰撞”的产生，使得散列表的检索过程仍然是一个给定值和记录关键码进行比较的过程。因此，仍需要以平均查找长度作为衡量散列表查找效率的度量。

（2）查找过程中需和关键码进行比较的次数取决于下列三个因素：

- 散列函数
- 处理碰撞的方法
- 装填因子

散列函数的“好坏”首先影响出现冲突的频率。对于同一组随机的关键码，“均匀分布”的散列函数产生碰撞的可能性相同，则可不考虑它对ASL的影响。

对同样一组关键码，相同的散列函数，但不同的碰撞处理方法得到的散列表不一样，其ASL也必然不同，通常拉链法的ASL要小于开放地址法。
线性探测容易产生“二次聚集[堆积]”问题，而拉链法不会出现这种情况。

另外，处理碰撞相同的散列表，其ASL依赖于散列表的装填因子。装填因子表示散列表的装满程度。装填因子越小，发生碰撞的可能性越小；反之，装填因子越大，表中已填入的记录多，再填记录时，发生碰撞的可能性就越大，检索时比较次数就越多。

6. 散列表的删除

在“开放地址法”处理碰撞时，删除的元素需要特别标注，否则今后会出现检索错误。
例如：散列函数 key%11

删除5后，再检索16：失败！

特殊标志，如-1

检索时，碰到-1，继续按照找空单元的方法检索下一单元。
插入时，碰到-1，与空单元一样处理【可以插入】

6.5 树表的检索（动态字典）

折半检索效率高，但只适合于有序的顺序表。

顺序表中插入、删除结点需要数据元素移动，效率低。能否找到一种既能把链式存储结构的优点与折半检索高效率结合，并且又不要求有序表的检索方法？

树表：二叉排序树、平衡二叉排序树（AVL树）、 B树

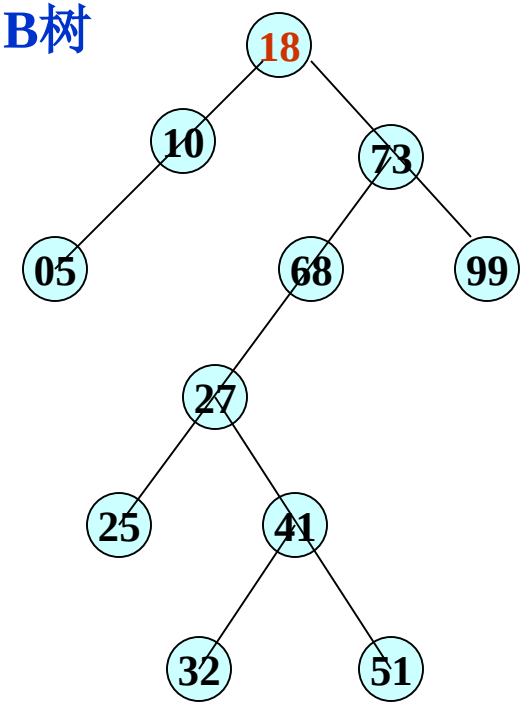
1. 二叉排序树

（1）定义：

或者是一棵空二叉树；

或者具有下列性质的二叉树：

- 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；
- 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
- 它的左右子树也分别为二叉排序树。

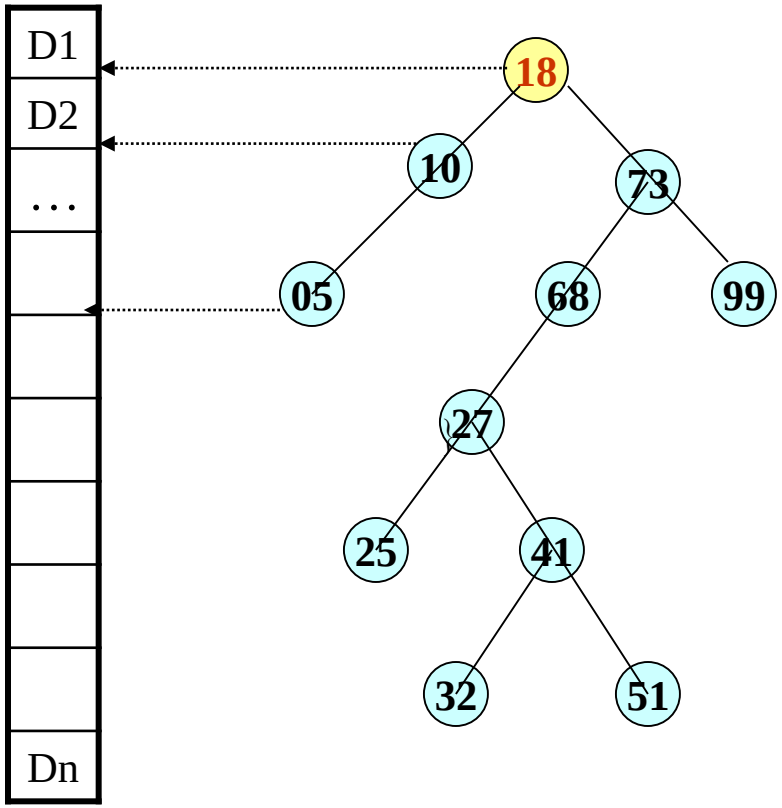


K={18, 73, 10, 05, 68, 99, 27, 41, 51, 32, 25}

字典的二叉排序树索引

字典的二叉排序树表示

二叉排序树



（2）为什么二叉排序树是动态字典的合适表达？

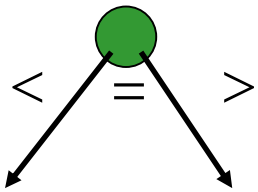
插入、删除不需要数据元素的移动，方便。检索是缩小区间的检索方法，检索效率快。无序序列构造得到二叉排序树，中序遍历，可得到一个有序的序列（可用于排序）。因此二叉排序树是动态字典的合适表达。

（3）检索思想

首先根据无序序列的先后顺序，构造二叉排序树。然后根据二叉排序树的定义进行检索：

若二叉排序树为空树，检索失败；

否则，　如给定值**key**等于二叉排序树的根结点的关键字，则检索成功；



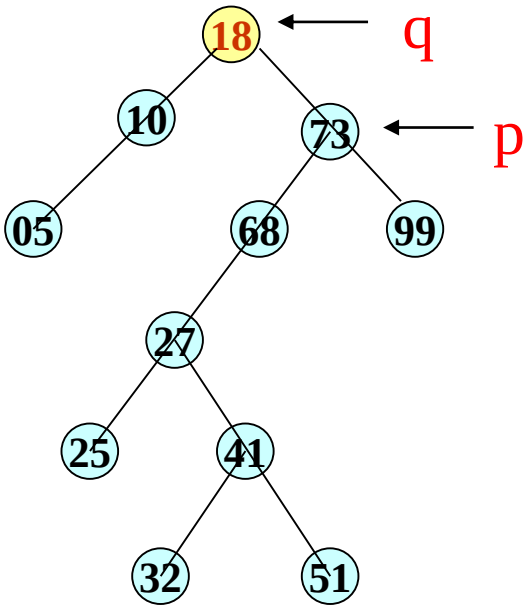
如给定值**key**小于二叉排序树的根结点的关键字，则说明待检索记录只可能在左子树中，继续在左子树检索；

如给定值**key**大于二叉排序树的根结点的关键字，则说明待检索记录只可能在右子树中，继续在右子树检索；

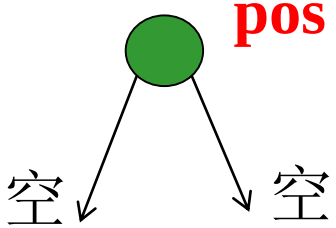
与折半相似
缩小区间搜索

（4）二叉排序树结构描述与检索算法

```
typedef struct BinNode
{   KeyType      key;          /* 结点的关键码字段 */
    DataType     other;        /* 结点的属性字段 */
    struct BinNode  *llink;    /* 二叉树的左指针 */
    struct BinNode  *rlink;    /* 二叉树的右指针 */
}BinTreeNode, *PBinTreeNode, BinTree, *PBinTree;
```



```
int SearchNode(PBinTree ptree, KeyType  key, PBinTreeNode pos)
{    PBinTreeNode p, q;
     p = ptree;  q = NULL;
     while (!p)
     {
          if (p->key == key)
          {   pos = p;      return(TRUE);  }      //检索成功
          q = p;                                  //q指向父结点位置
          if (p->key > key)
                    p = p->llink;                 //进入左子树
          else
                    p = p->rlink;                 //进入右子树
      }
      pos = q;                                    //检索失败
      return(FALSE);
}
```



（5）二叉排序树的插入与构造

插入新的结点后，必须保证插入后仍然满足二叉排序树的性质。

插入新结点的方法:

- 如果二叉排序树为空，则新结点作为根结点。
- 如果二叉排序树非空，则将新结点的关键码与根结点的关键码比较，

  若小于根结点的关键码，

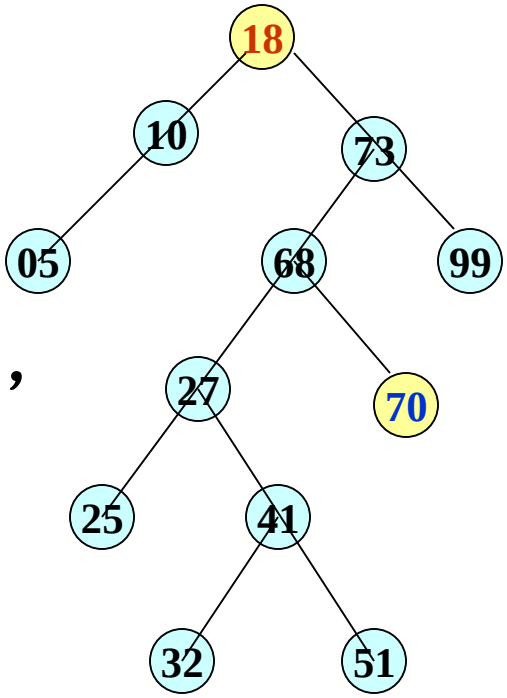
  则将新结点插入到根结点的左子树中；

  否则，插入到右子树中。
- 子树中的插入过程和树中的插入过程相同，

  如此进行下去，直到找到该结点，

  或者直到新结点成为叶子结点为止。

如插入70，二叉排序树形式如右图。



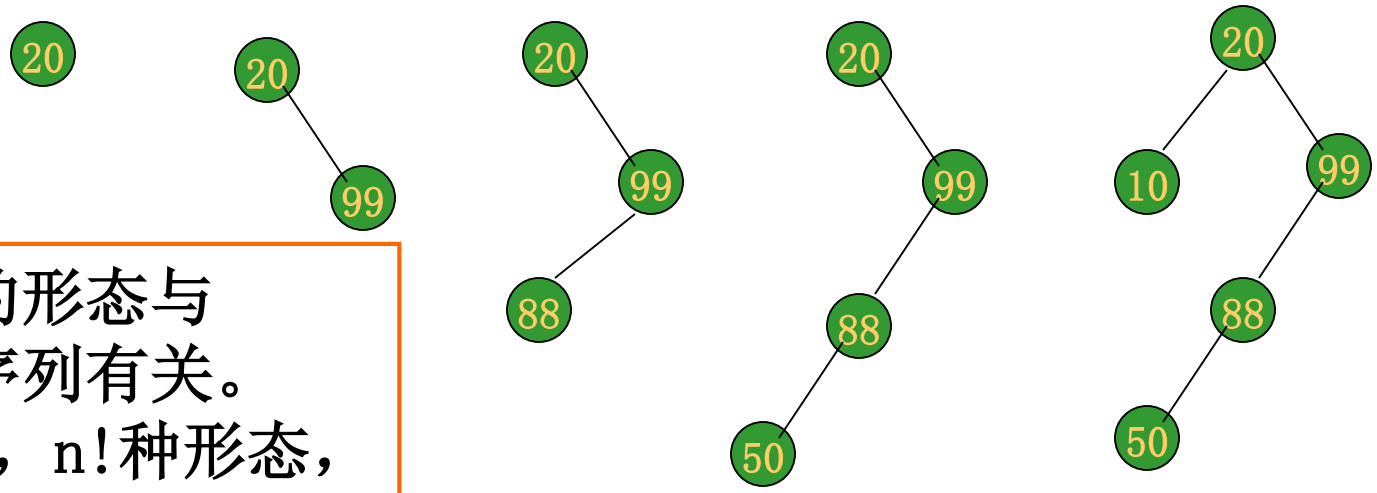
```
void InsertNode(PBinTree ptree, KeyType key)
{   PBinTreeNode p, q;
    if (SearchNode(ptree, key, q) == TRUE)
    {   return; }                    //已经存在关键码为key的结点
    //q返回插入位置的父结点
    p = (PBinTreeNode)malloc(sizeof(struct BinTreeNode));
    if ( p == NULL )                          //内存申请错误 */
    {   printf("Memory allocated Error!\n"); exit(1); }
    p->key = key;
    p->llink = p->rlink = NULL;
    if (q == NULL)                            //空树插入，直接为根
            ptree = p;
    else if (key < q->key)                    //插入为q的左子女
            q->llink = p;
    else                                      //插入为q的右子女
            q->rlink = p;
}
```

二叉排序树的构造：从空树开始，将元素逐个插入到二叉排序树中。

```
void CreateTree(PBinTree ptree, SeqDictionary dic)
{   int i;

    for (i = 0; i < dic.n; i++)
        InsertNode(ptree, dic.element[i].key);
}
```

K= { 20, 99, 88, 50, 10 }



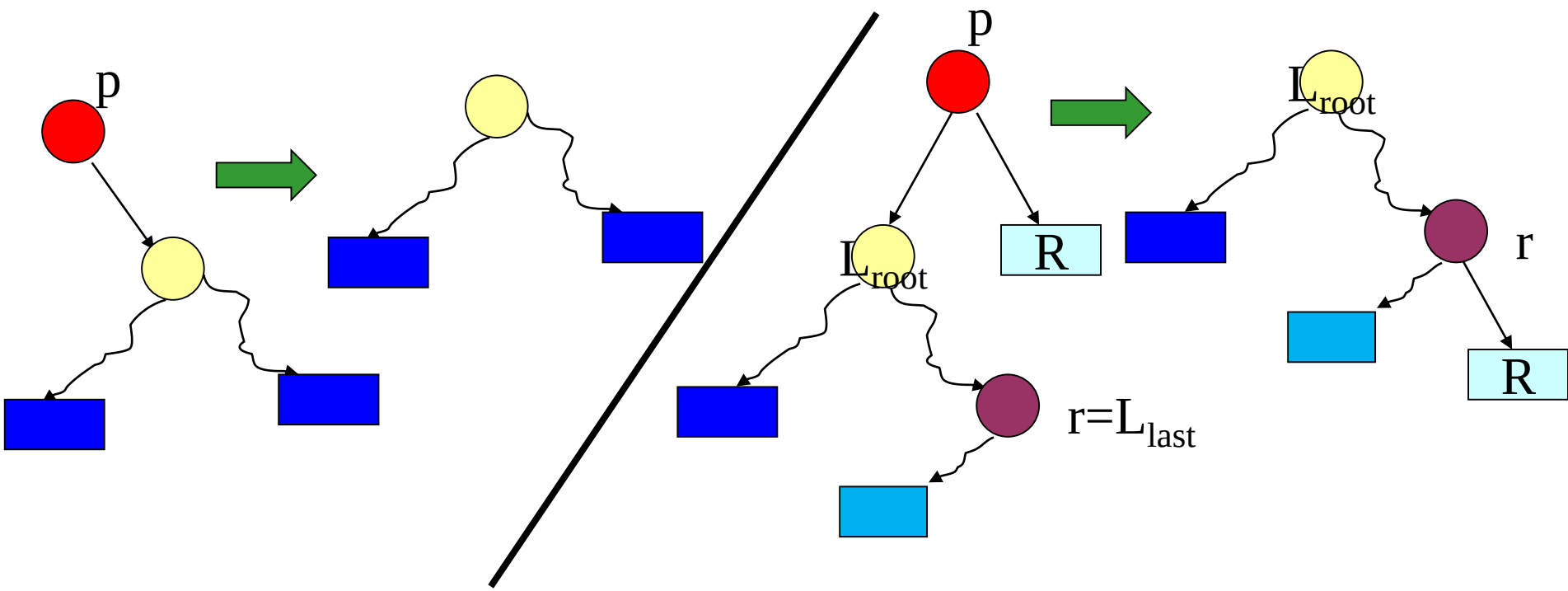
二叉树的形态与关键码序列有关。
n个记录，n!种形态，
那种检索效率最高？

（5）二叉排序树的删除

删除结点后，仍然满足二叉排序树的性质。 …L_{root}…L_{last}PR…

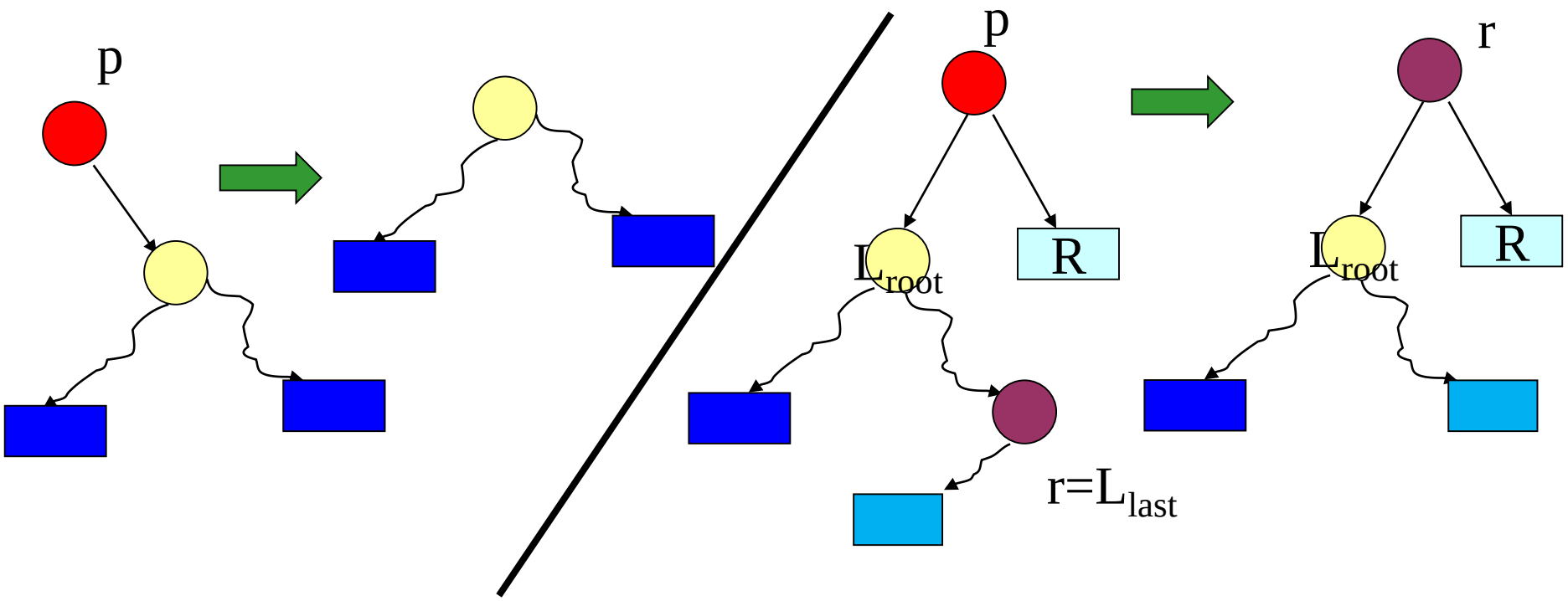
方法1:

- 如果被删除结点p没有左子树，则用p的右子女代替p即可。
- 否则，在p的左子树中，中序遍历找出最后一个结点r(r一定无右子女)，将r的右指针指向p的右子女，用p的左子女代替p结点。

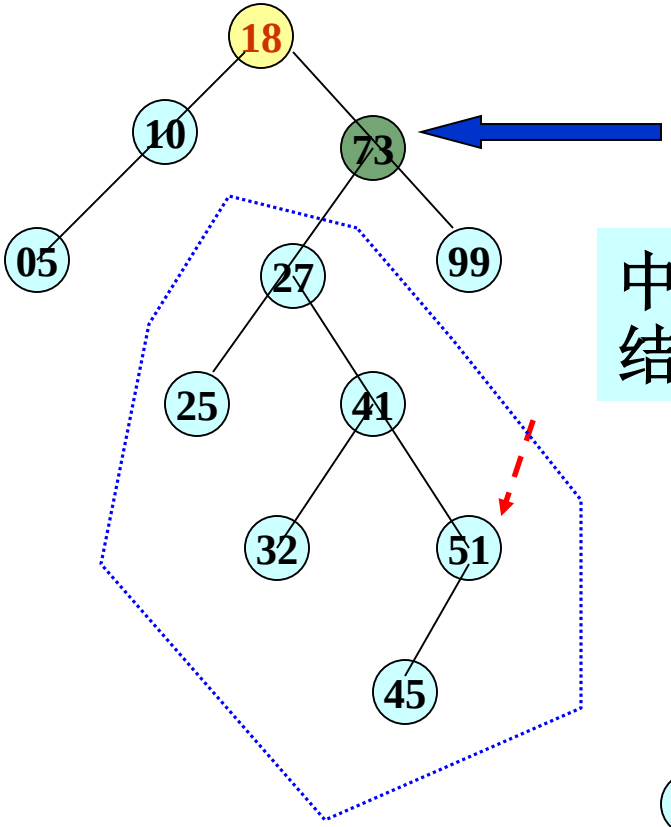


方法2:

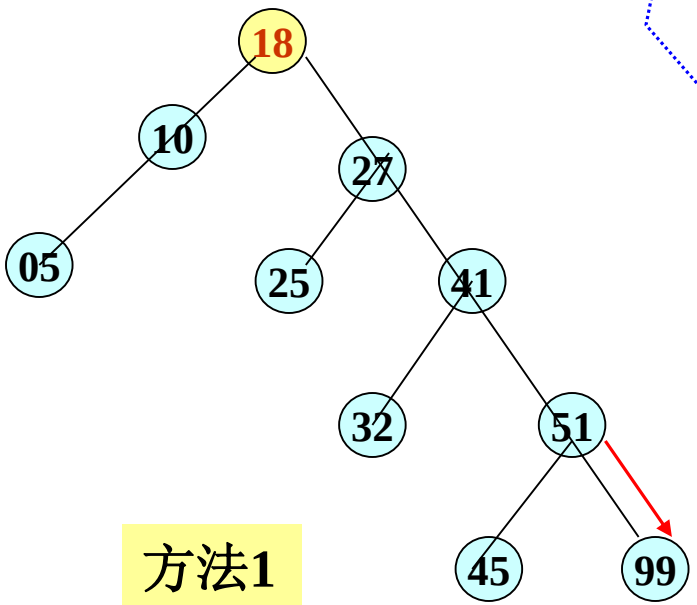
- 如果被删除结点p没有左子树，则用p的右子女代替p即可；
- 否则，在p的左子树中，中序遍历找出最后一个结点r（r一定无右子女），用r结点代替被删除的结点p，用r的左子女代替r结点。



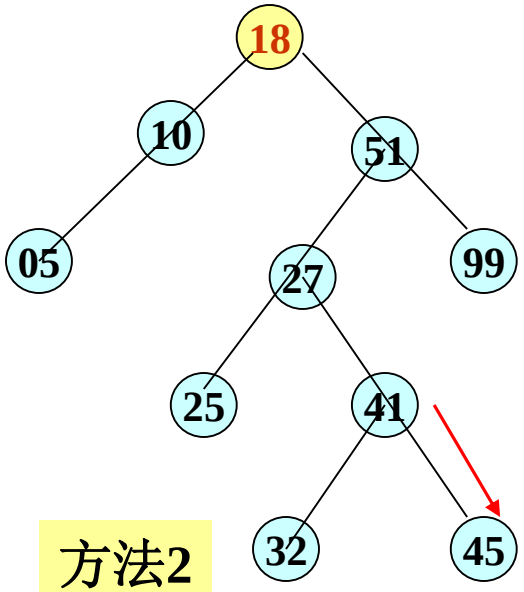
删除key=73结点



中序遍历左子树最后结点，其关键字最大



方法1



方法2

方法1算法：

```
void DeleteNode(PBinTree ptree, KeyType key)
{    PBinTreeNode parentp, p, r;
     p = ptree;   parentp = NULL;
     while( p != NULL )                    //找关键码为key的结点
     {   if (p->key == key) break;
         parentp = p;                      //保存其父结点
         if (p->key > key)  p = p->llink;       //进入左子树
         else               p = p->rlink;       //进入右子树
     }
     if (p==NULL) return;                  //无关键码为key的结点
     if (p->llink == NULL)                 //结点p无左子树
     {   if (parentp == NULL)    ptree = p->rlink;  //p为根
         else if (parentp->llink == p)
                   parentp->llink = p->rlink;       //左
         else      parentp->rlink = p->rlink;       //右
     }
```

```
    else                                    //结点p有左子树
    {
        r = p->llink;                       //在p的左子树中找最右下结点r
        while( r->rlink != NULL)    r = r->rlink;
        r->rlink = p->rlink;        //用r的右指针指向p的右孩子

        //用p的左孩子代替p
        if (parentp == NULL)
            ptree = p->llink;                   //p为根
        else if (parentp->llink == p)
            parentp->llink = p->llink;      //左
        else
            parentp->rlink = p->llink;      //右
    }
    //删除p
    free(p);
}
```

给定一组n个记录的记录序列（无序），构造的二叉排序树有n!种形态，各种形态的查找效率不相同。

如何构造高效率的二叉排序树？

等概率：　折半检索的二叉判定树或平衡二叉树（AVL树）

不等概率：构造最佳二叉排序树或次优查找树

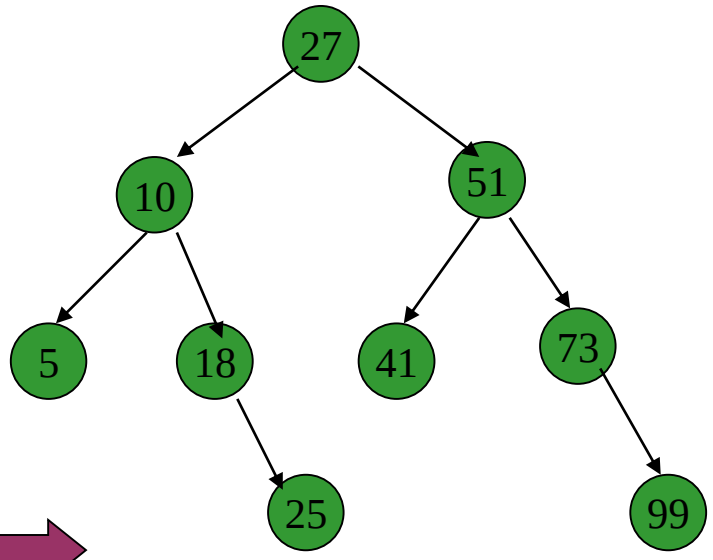
二叉判定树构造：

1）记录序列按照关键码排序；

2）按照关键码的折半检索次序，在二叉排序树中加入记录。

27，73，10，5，18，41，99，51，25

5，10，18，25，27，41，51，73，99



2. 平衡二叉排序树（AVL树）

对于一棵二叉排序树，其检索效率取决于树的形态，而构造一棵形态均称的二叉排序树与结点的输入顺序有关。因此，为了保证高效率检索就必须对构造的二叉排序树进行动态平衡，使得其形态均称。

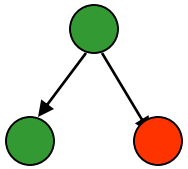
平衡二叉树：又称AVL树。它或者是一棵空树，或者是具有下列性质的二叉树：它的左右子树均为平衡二叉树，且左右子树的深度之差的绝对值不超过1。

平衡因子(Balance Factor)：结点的右子树的深度减去左子树的深度。

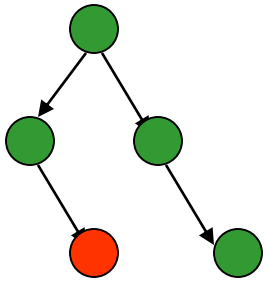
对于AVL树，各个结点的BF值只可能为-1, 0, 1。

构造过程中，每插入一个新的结点，都需要判断是否破坏了原来的平衡条件。如果破坏，需要进行平衡处理。

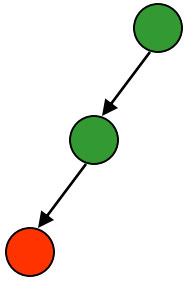
平衡二叉树在插入一个新结点后，可能出现三种情况：



- 新结点插入后不影响其父结点为根的子树深度，则不会破坏整个二叉排序树的平衡；



- 父结点为根的子树深度增加了，但在其祖先的某一层上不再影响子树的深度，则整个二叉排序树仍然是平衡的；



- 父结点为根的子树深度增加，且在其祖先的某一层上破坏了平衡的要求，使整个二叉排序树不再平衡。

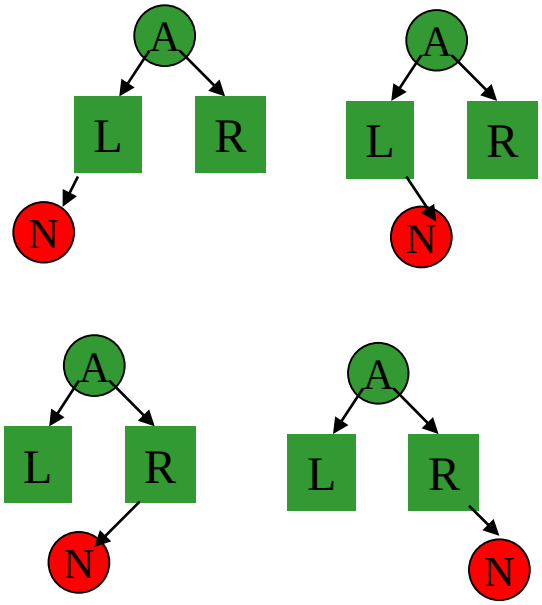
失去平衡后的处理方法：

首先找出最小不平衡子树，在保证二叉排序树性质的前提下，调整最小不平衡子树中各结点的连接关系，以达到新的平衡。

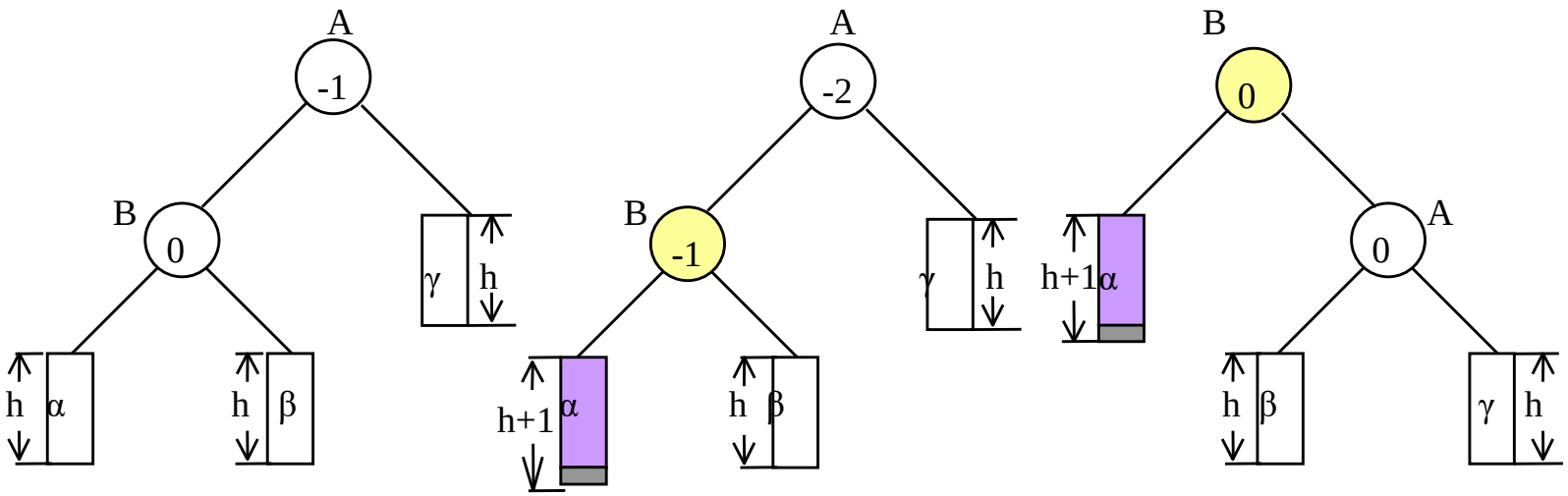
最小不平衡子树：离插入结点最近，平衡因子BF绝对值大于1的结点为根的子树

假设最小不平衡子树的根结点为A，根据新插入结点的位置，调整子树的操作可归纳为以下四种情况：

- LL（A的左子女[L]的左子树[L]上插入新结点）[-1 to -2]
- LR（A的左子女[L]的右子树[R]上插入新结点）[-1 to -2]
- RL（A的右子女[R]的左子树[L]上插入新结点）[1 to 2]
- RR（A的右子女[R]的右子树[R]上插入新结点）[1 to 2]



（1）LL型调整

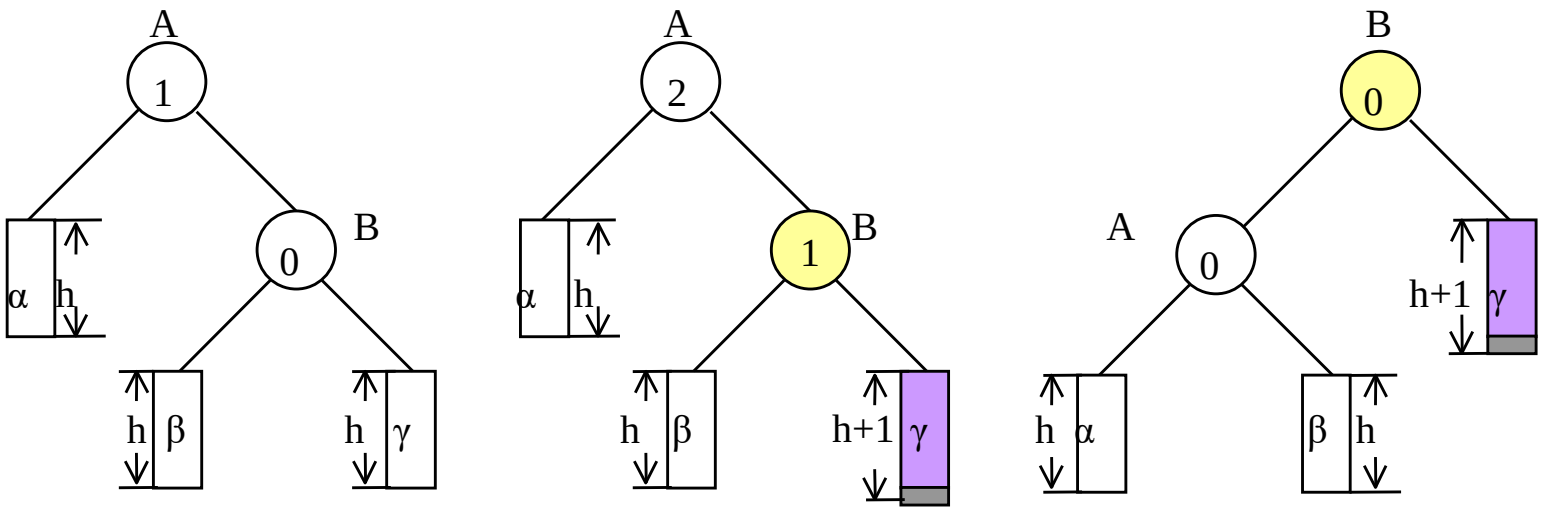


LL型调整操作示意图：(αBβ)A(γ)= (α)B(βAγ)

调整规则是：

- **将A的左子女B提升为新二叉树的根；**
- **原来的根A连同其右子树向右下旋转成为B的右子树；**
- **B的原右子树β作为A的左子树。**

（2）RR型调整

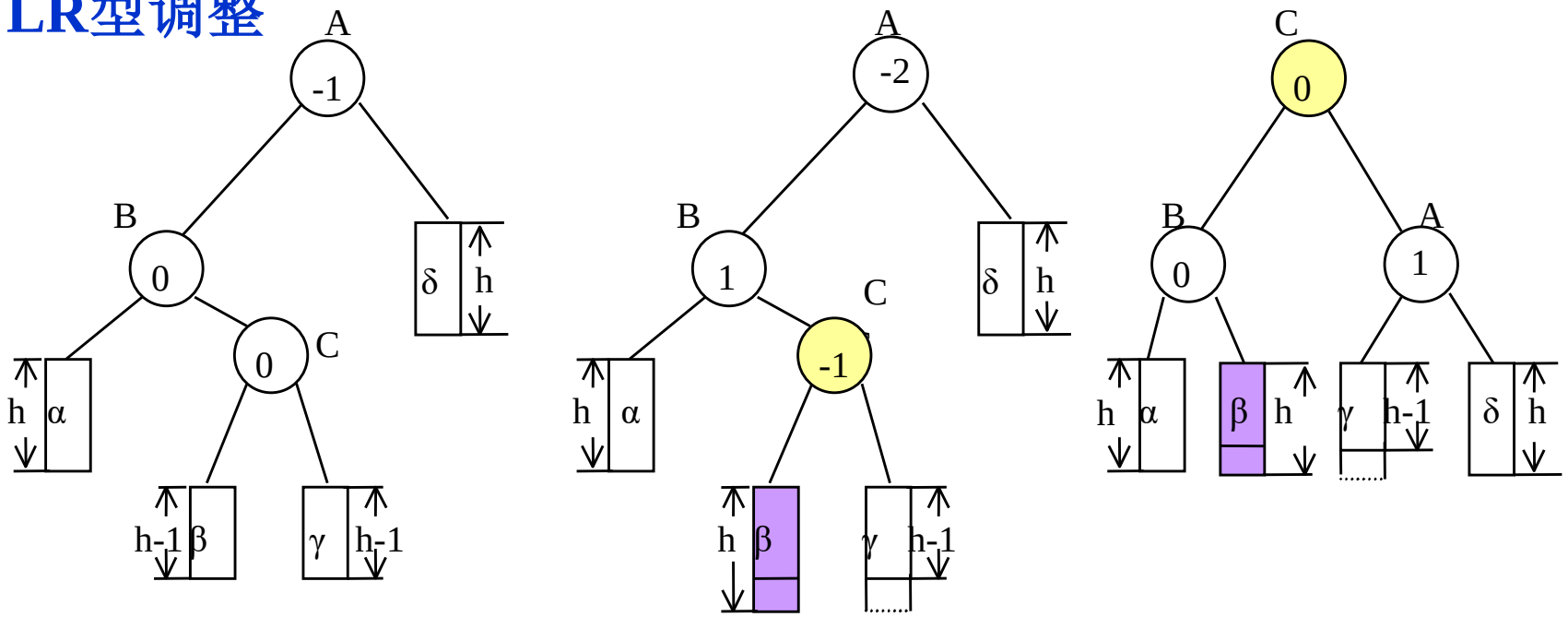


RR型调整操作示意图：　(α)A(βBγ)= (αAβ)B(γ)

调整规则：

- **将A的右子女B提升为新二叉树的根；**
- **原来的根A连同其左子树向左下旋转成为B的左子树；**
- **B的原左子树作为A的右子树。**

（3）LR型调整

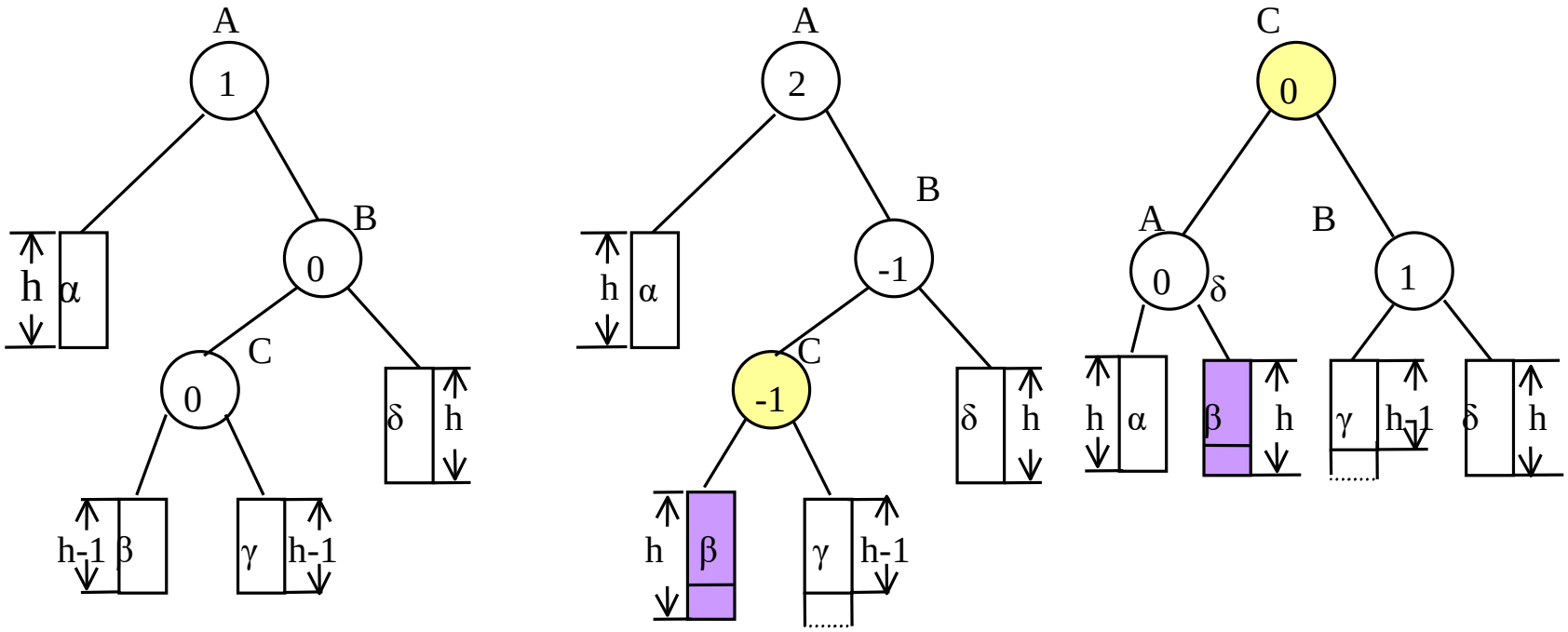


LR型调整操作示意图：　((α)B(βCγ))A(δ)= (αBβ)C(γAδ)

调整规则:

- **设C为A的左子女B的右子女，将A的孙子结点C提升为新二叉树的根；**
- **原C的父结点B连同其左子树α向左下旋转成为新根C的左子树，原C的左子树β成为B的右子树；**
- **原根A连同其右子树δ向右下旋转成为新根C的右子树，原C的右子树γ成为A的左子树。**

(4) RL型调整



RL型调整操作示意图：(α)A((βCγ)B(δ))= (αAβ)C(γBδ)

调整规则：

- **设C为A的右子女B的左子女，将A的孙子结点C提升为新二叉树的根；**
- **原来C的父结点B连同其右子树δ向右下旋转成为新根C的右子树，C的原右子树γ成为B的左子树；**
- **原来的根A连同其左子树α向左下旋转成为新根C的左子树，原来C的左子树β成为A的右子树。**

结点插入算法的关键部分：

- 新结点插入导致失去平衡，需找最小不平衡子树。

  令A始终指向离插入位置最近、且平衡因子不为零的结点，若新结点插入后失去平衡，则A是最小不平衡子树的根。

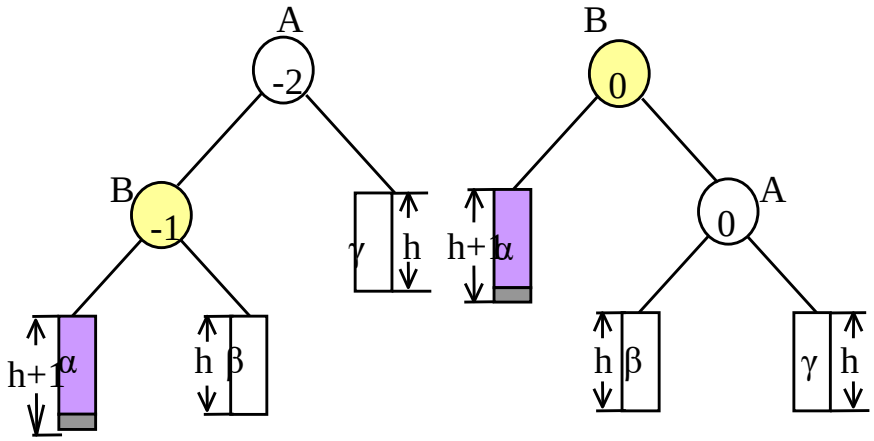
- 新结点插入后，修改一些结点的平衡因子。
- 判别以A为根的子树是否失去了平衡。
- 失去平衡后，按照四种类型之一调整。

AVL树的结点结构：

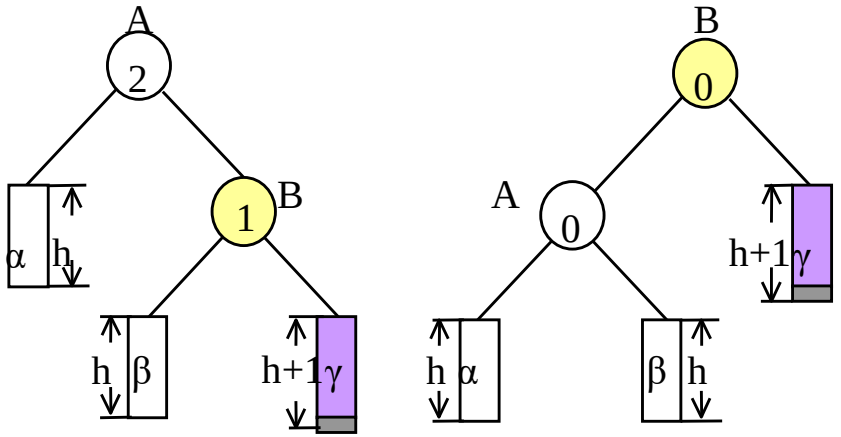
```
typedef struct AVLNode
{
        KeyType      key;          /* 结点的关键码 */
        DataType     other;        /* 结点的其它信息 */
        int          bf;           /* 结点的平衡因子 */
        struct AVLNode *llink,     /* 指向结点的左子女 */
        struct AVLNode *rlink;     /* 指向结点的右子女 */
}AVLNode, *PAVLNode, AVLTree, *PAVLTree;
```

下面只讨论四种调整，有关**AVL树的检索与插入算法**见教材p229~232，希望大家课后阅读，时间效率分析$O(\log_2 n)$。

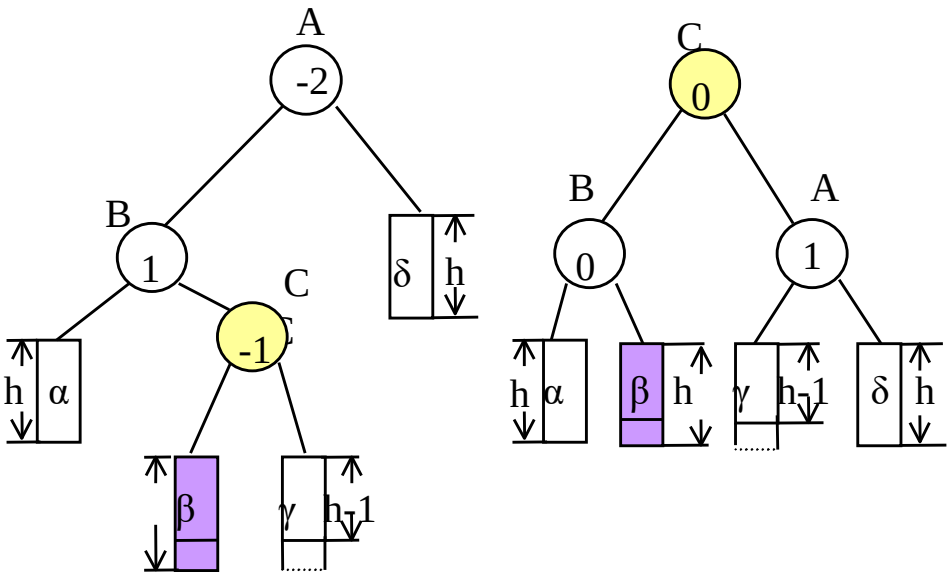
```
PAVLNode  LL(PAVLNode  a)
{       PAVLNode     b = a->llink;
        a->bf = 0;
        a->llink = b->rlink;
        b->bf = 0;
        b->rlink = a;
        return(b);      //b指向调整后的子树的根结点
}
```



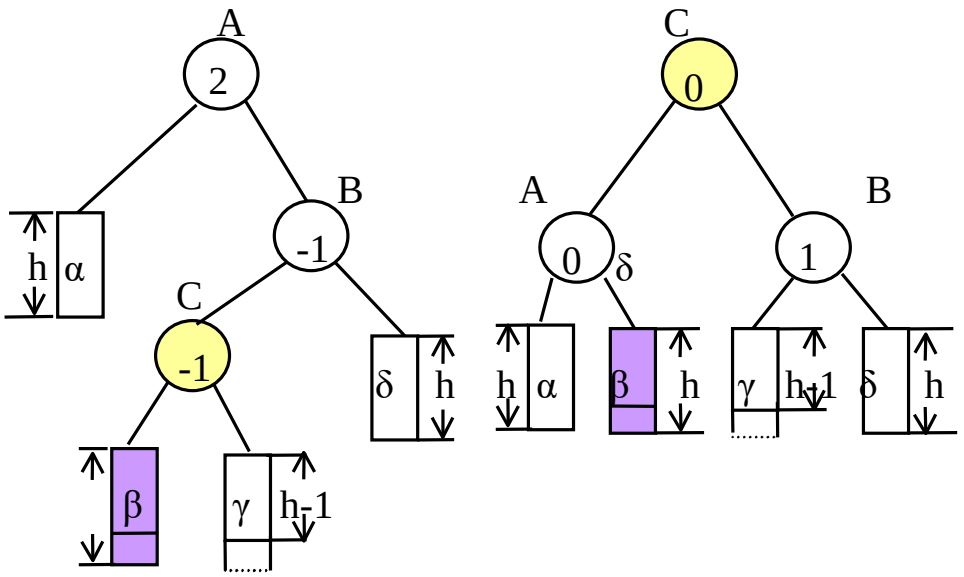

```
PAVLNode  RR(PAVLNode a)
{       PAVLNode     b = a->rlink;
        a->bf = 0;
        a->rlink = b->llink;
        b->bf = 0;
        b->llink = a;
        return(b);      //b指向调整后的子树的根结点
}
```



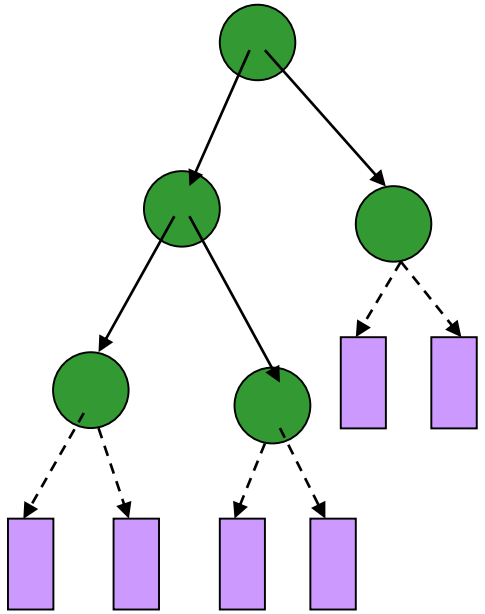
```
PAVLNode  LR(PAVLNode a)
{  PAVLNode        b, c;
   b = a->llink;
   c = b->rlink;
   a->llink=c->rlink;
   b->rlink=c->llink;
   c->llink=b;  c->rlink=a;
   switch(c->bf)
   {
      case 0:  a->bf=0;  b->bf=0; break;   //LR(0)型调整
      case 1:  a->bf=1;  b->bf=0; break;   //新结点插在*c的左子
                                           //树中，LR(L)型调整
      case –1: a->bf=0; b->bf=-1; break; //新结点插在*c的右子
                                           //树中，LR(R)型调整
   }
    c->bf=0;
    return(c);
}
```



```
PAVLNode  RL(PAVLNode a)
{    PAVLNode b, c;
     b = a->rlink;
     c=b->llink;
     a->rlink=c->llink;
     b->llink=c->rlink;
     c->llink=a;  c->rlink=b;
     switch(c->bf)
     {  case 0: a->bf=0; b->bf=0; break;  //c本身就是插入结点，
                                          //RL(0)型调整
        case 1: a->bf=0; b->bf=1; break;  //插在*c的左子树中，
                                          //RL(L)型调整
        case –1:a->bf=-1; b->bf=0; break;//插在*c的右子树中，
                                          //RL(R)型调整
     }
      c->bf=0;
      return(c);
}
```



3. 最佳二叉排序树



扩充二叉排序树：按照扩充二叉树的方法对二叉排序树进行扩充，得到扩充二叉排序树。

如果检索终止于内部结点：检索成功；终止于外部结点：检索失败。

中序遍历扩充二叉排序树得到：最左下角的外部开始，到最右下角外部结点结束，内外部结点交叉排列，第i个内部结点位于第i、i+1个外部之间。

同一个关键码集合，由于结点插入的先后次序不同，所构造的二叉排序树的形态和深度是不同的。二叉排序树的检索长度和树的形态有关，可以用平均检索长度E(n)来评价二叉排序树的好坏。

成功　　失败

$$E(n)=\frac{1}{w}\left[\sum_{i=1}^{n}p_i(l_i+1)+\sum_{i=0}^{n}q_i l'_i\right] \qquad w=\sum_{i=1}^{n}p_i+\sum_{i=0}^{n}q_i$$

其中l_i是第i个内部结点的层数，l'_i是第i个外部结点的层数，p_i是检索第i个内部结点关键码的频率，q_i是被检索的关键码属于第i个外部结点关键码集合的频率。

内部结点：代表一个记录的关键码；**外部结点**：代表与其相邻的两个内部结点关键码之间的那些关键码值。

最佳二叉排序树：检索中平均比较次数最小，即E(n)最小的二叉排序树。

1) 所有结点的检索概率都相等:

$$\frac{p_1}{w}=\frac{p_2}{w}=\ldots=\frac{p_n}{w}=\frac{q_0}{w}=\frac{q_1}{w}=\ldots=\frac{q_n}{w}=\frac{1}{2n+1}$$

IPL:内部路径长度； EPL:外部路径长度

EPL = IPL+2n, 对E(n)推导得：

E(n) = (IPL+n+EPL) / (2n+1) = (2IPL+3n)/(2n+1)

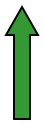
其中：$IPL = \sum_{k=1}^{n} \lfloor \log_2 k \rfloor$

$$= (n+1)\lfloor \log_2 n \rfloor - 2^{\lfloor \log_2 n \rfloor + 1} + 2$$

IPL最小，则E(n)最小；IPL最大，则E(n)最大。

IPL最小的二叉树形态是：只有最后两层的结点度小于2的二叉树。

IPL最大的二叉树形态是：单枝树。



与折半查找的判定树形态相同

构造方法：

（a）将结点按关键码排序

（b）按二分法依次检索关键码，将检索中遇到的还未在二叉排序树中的结点插入到已经得到的二叉排序树中。

时间复杂度：$O(\log_2 n)$。

2) 各个结点的检索概率不相等

不等概率时，按照折半检索思路构造的二叉排序树并不是合适的。

最佳二叉排序树的特点：它的任何子树都是最佳二叉排序树。

设关键码key_{i+1}，key_{i+2},…,key_j为内部结点，相应的权为(q_i，p_{i+1}，q_{i+1},…,p_j，q_j)的最佳二叉排序树为T(i, j)，

构造步骤：

- **构造包括一个结点的最佳二叉排序树，**

  就是找T(0,1), T(1,2),…,T(n-1,n)

- **构造包括两个结点的最佳二叉排序树，**

  就是找T(0,2), T(1,3),…,T(n-2,n)

- **再构造包括三个、四个、…结点的最佳二叉排序树，**

  直到最后构造T(0,n)

最优二叉排序树的构造代价太大，通常用于静态字典。
对于动态字典，通常采用“次优查找树”，即：
次优二叉排序树: IPL并不最小，而是近似最小。

检索效率取决于二叉排序树的形态，形态由各个子树根的选择决定。

次优二叉排序树的构造：

$(key_i, key_{i+1},\ldots,key_j)$ ⟷ $(p_i, p_{i+1},\ldots,p_{k-1},p_k,p_{k+1},\ldots,p_j)$

关键字已经有序：$key_i < key_{i+1} < \ldots < key_j$

计算j-i+1个记录的

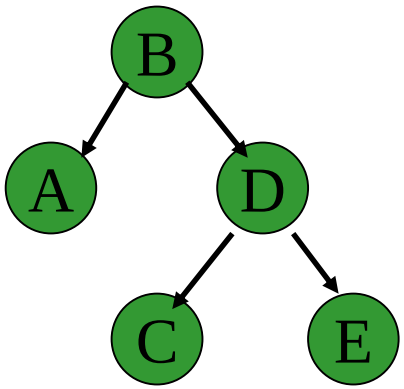
$$\Delta P_k = \left| \sum_{m=k+1}^{j} p_m - \sum_{m=i}^{k-1} p_m \right| \qquad (i \le k \le j)$$

选择ΔP_k最小的记录构造根结点 $\mathbf{RP_k}$。

然后分别对子序列$\{r_i, r_{i+1},\ldots, r_{k-1}\}$和$\{r_{k+1},r_{k+2},\ldots, r_j\}$构造次优二叉排序树，并分别作为根 $\mathbf{RP_k}$ 的左、右子树。

另外，由于构造次优二叉排序树过程中，并没有考虑单个关键字的查找概率，可能出现根关键字的查找概率比它相邻的关键字的查找概率小。此时，应该适当调整：选择邻近的查找概率较大的关键字作为根。

key:	A	B	C	D	E
P:	1	30	2	29	3



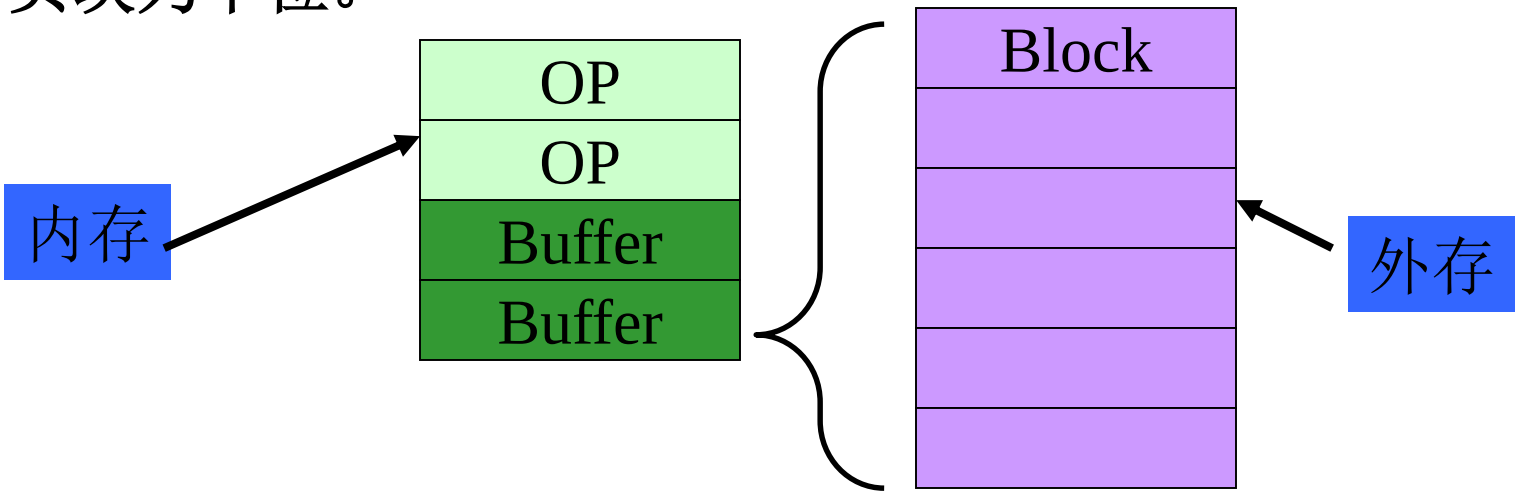
算法：

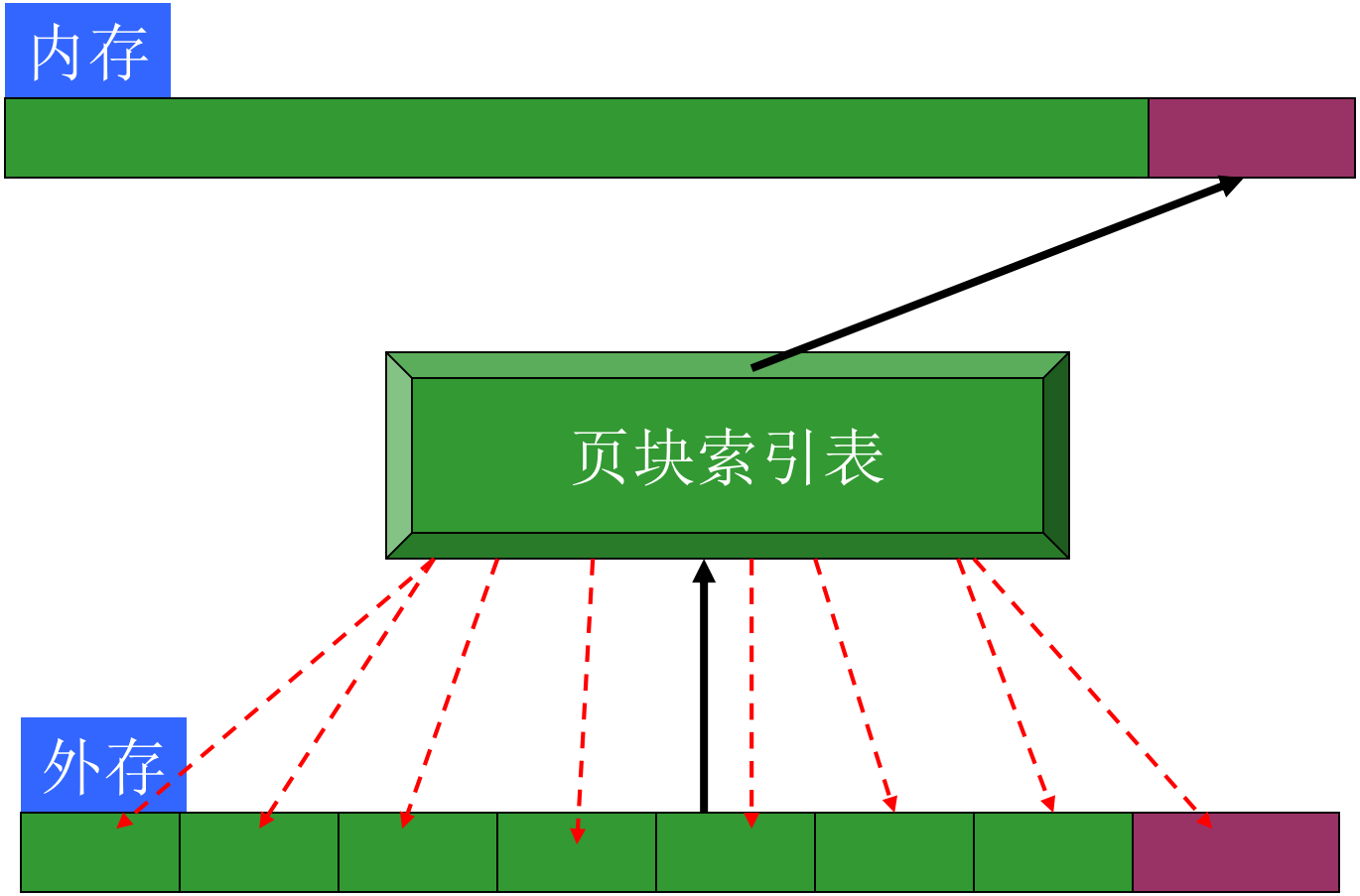
```
int root_selector(int i, int j, float p[])
{   float sp1, sp2, delta_p = 999999;
    int is, m = -1;
    for (k = i+1; k <= j-1; k++)
    {  sp1 = sp2 = 0;
       for (m = i; m <= k-1; m++)  sp1 += p[m];
       for (m = k+1; m <= j; m++) sp2 += p[m];
       if (fabs(sp2-sp1) < delta_p)
       {
           delta_p = fabs(sp2-sp1);
           m = k;
       }
    }
    //原来的：return m;
    //改进后：m-1, m, m+1三者中p值大的返回
}
```

4. B树与B+树

用途：文件组织中，索引文件的索引表结构，方便插入、删除和检索。二叉排序树适用于组织较小的、内存中可以容纳的字典。对于较大的必须存放在外存储器上的字典，通常采用B或B+树进行索引组织。

外存访问的特点: 存取速度慢，每次访问需要定位。采用分块存取减少内外存交换的次数提高效率：即在内存开辟一个或多个与外存页块（物理记录）大小相同的缓冲区，内外存交换以整个页块为单位。




内存
页块索引表
外存

(1) B树

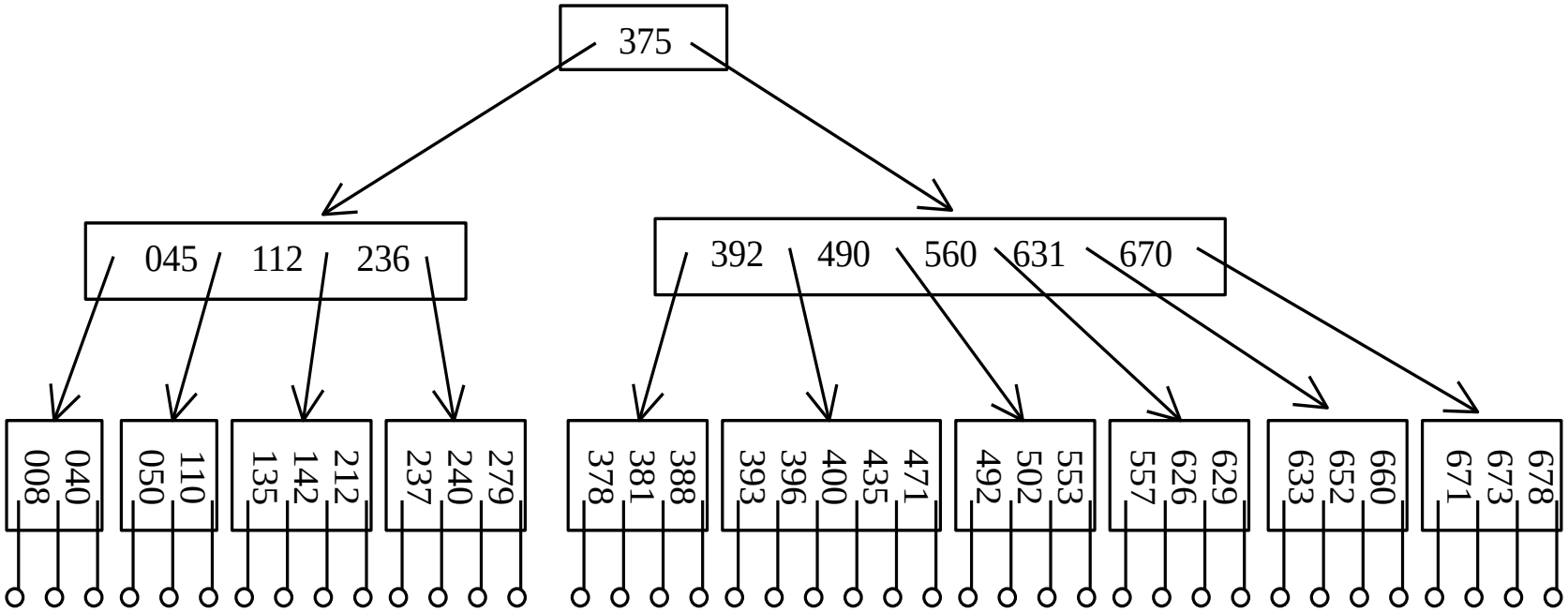
a) B树定义：

一棵m阶的B树，或为空树，或是满足下列特性的m叉树：

- 每个结点至多有m棵子树。
- 除根结点外，其它非终端结点至少有⌈ m/2⌉棵子树。
- 根结点至少有两棵子树(除非根结点为叶结点，此时B树只有一个结点)。
- 所有叶结点（外部结点或查找失败的结点，实质上不存在）在同一层上，叶结点不包含任何关键码信息。
- 有K个孩子的非叶结点恰好有K-1个关键码，关键码按递增次序排列。结点中包含的信息为：$(p_0,k_1,p_1,k_2,p_2,\ldots,k_n,p_n)$其中，$k_i(1\le i\le n)$为关键码，且满足 $k_i<k_{i+1}(i=1,2,\ldots,n-1)$

$p_i(0\le i\le n)$为指向子树根结点的指针

(子树中所有关键码满足： $k_i < k < k_{i+1}$)。



6阶B树

b) 运算：

检索： 类似二叉排序树。给定值**key**，在根结点的关键字集合中进行检索（顺序或折半），如**key=K_i**，则检索成功；否则，**key**必定在某**K_i**和**K_{i+1}**之间（即：如果存在必定在**P_i**指向的子树中），进入**P_i**继续检索，重复上述检索过程，直到检索成功或**P_i**为空为止（检索失败）。

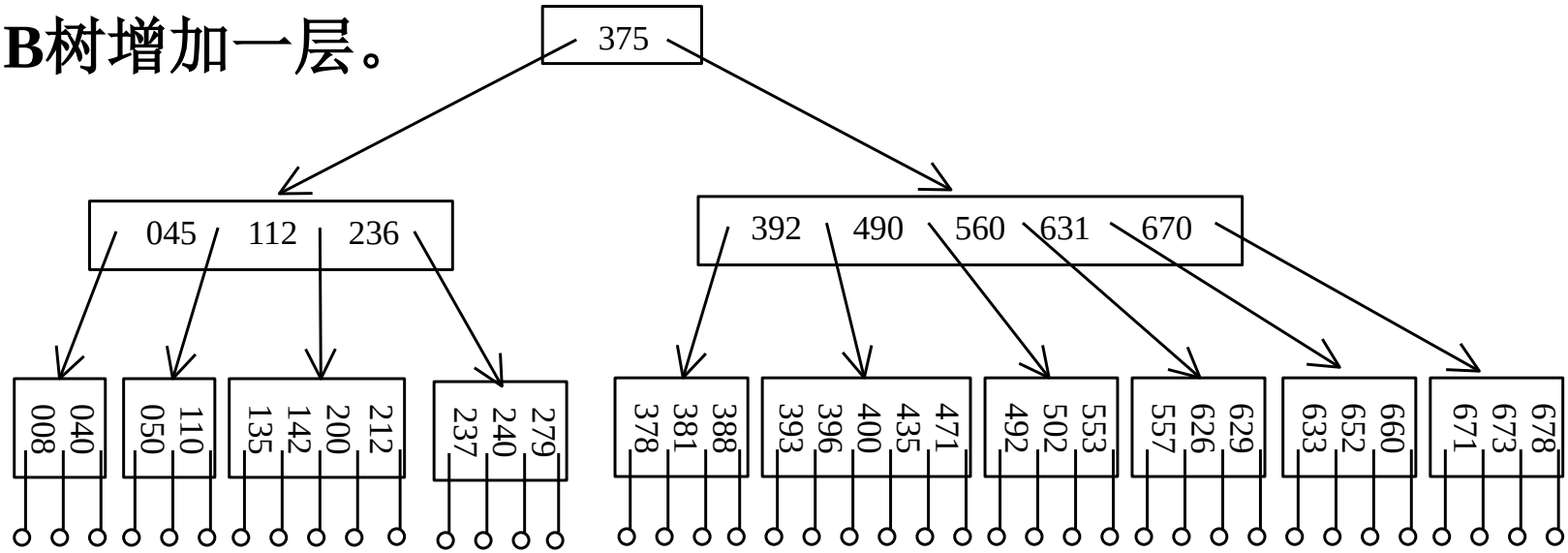
插入：

- 若该结点中关键码个数小于m-1，则直接插入即可。
- 若该结点中关键码个数等于m-1，则将引起结点的分裂。

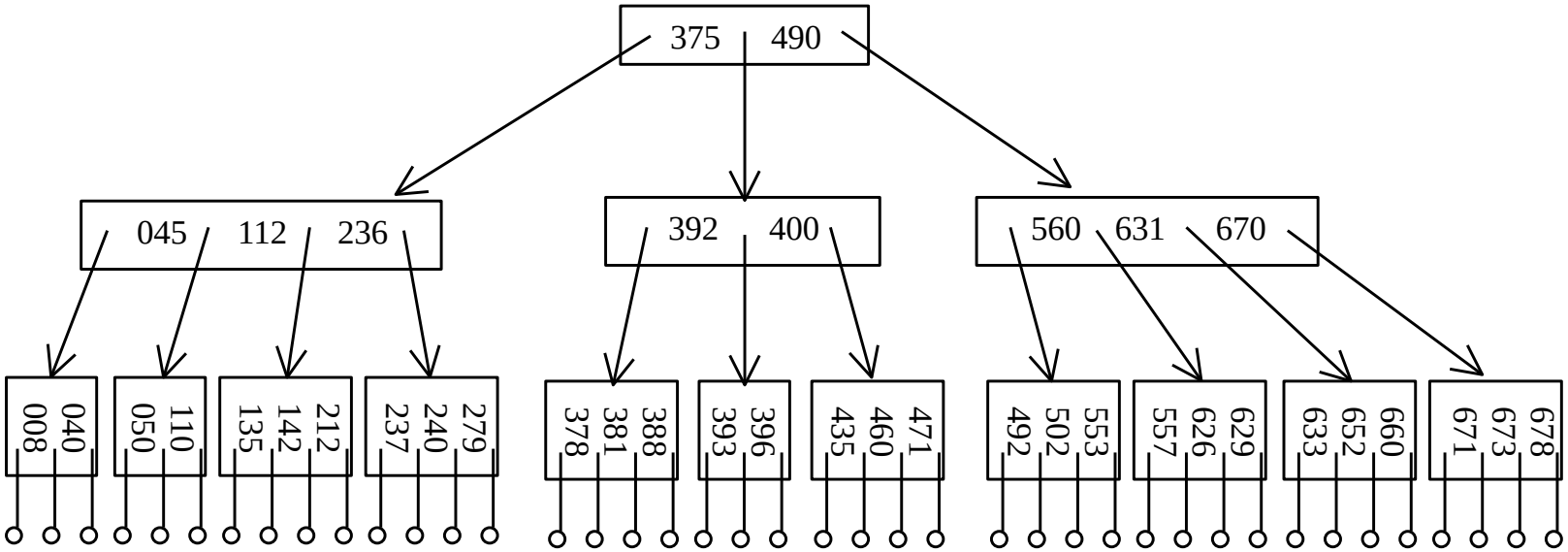
以中间关键码为界将结点一分为二，产生一个新结点，并把中间关键码插入到父结点中；

若父结点中关键码数也为m-1，则需要再次分裂。

最坏情况一直分裂到根结点，建立一个新的根结点，整个B树增加一层。



(a) 插入200后的B树



(b) 插入460后的B树

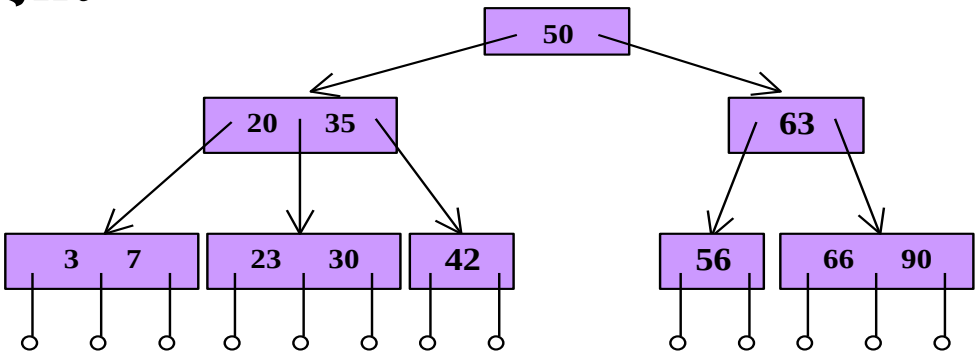
删除:

□若结点在第h层（B树深度为h），且结点中关键码个数大于⌈m/2⌉－1，直接删除即可。

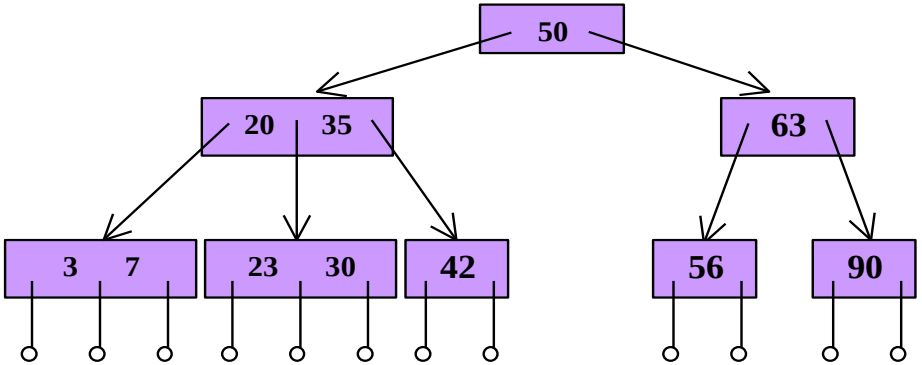
□若结点在第h层，且关键码个数等于⌈m/2⌉－1，结点左(或右)兄弟结点的关键码个数大于⌈m/2⌉－1，则把左(或右)兄弟结点中最大(或最小)的关键码移到父结点中，并将父结点中大于(或小于)其值的关键码移到被删除关键码所在的结点中。

- 若结点在第h层，且关键码个数及结点左、右兄弟中的关键码个数都等于⌈ m/2⌉−1 ，则需要合并该结点、其兄弟结点及父结点中的一个关键码。

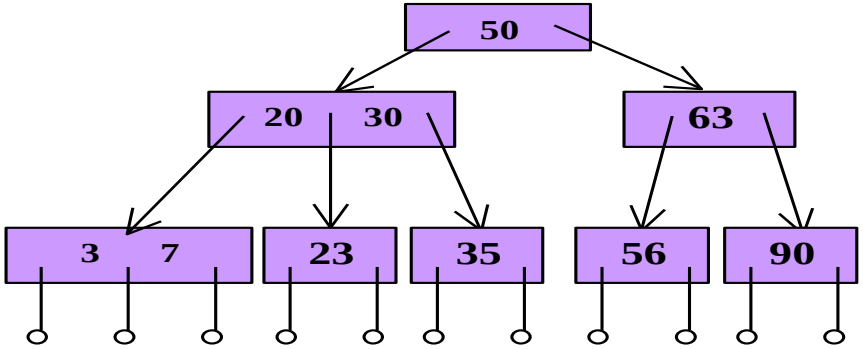
- 若结点的层数小于h，设删除的关键码为结点中的第i个关键码k_i，则用p_i所指的子树中的最小关键码k代替k_i，然后再删除关键码k。



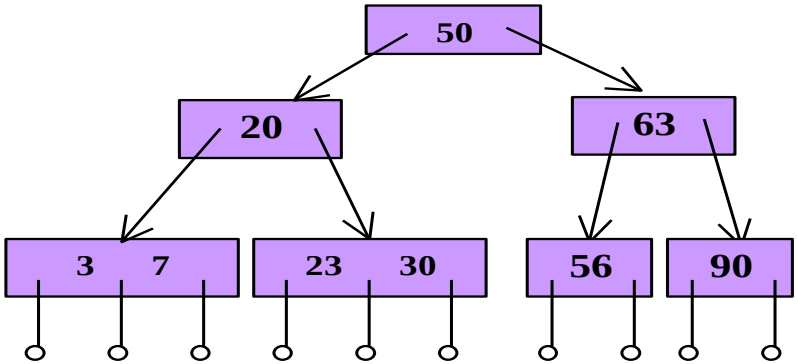
(a) 3阶B树



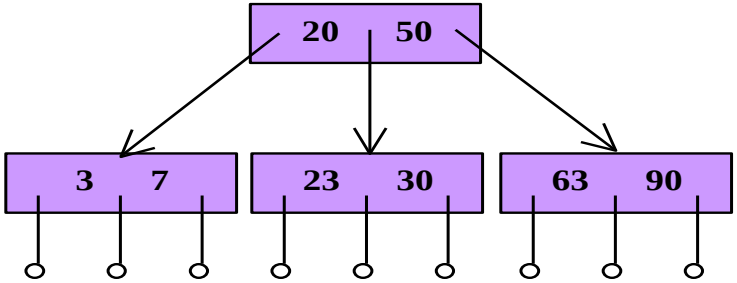
(b) 从图a中删除关键码66



(c) 从图b中删除关键码42



(d) 从图c中删除关键码35



(e) 从图d中删除关键码56

(2) B+树

a) B^+树的定义

一个m阶的B+树满足下列条件:

- 每个结点至多有m棵子树。
- 除根结点外，其它每个分支至少有⌈m/2⌉棵子树。
- 根结点至少有两棵子树。
- 有n棵子树的结点有n个关键码
- 叶结点中存放数据文件中记录的关键码及指向该记录的指针，且叶子结点本身依关键字从小到大顺序链接。
- 分支结点中仅包含它的各个子结点中最大(或最小)关键码及指向子结点的指针。

所有信息存放在叶结点，中间结点实质上是索引。

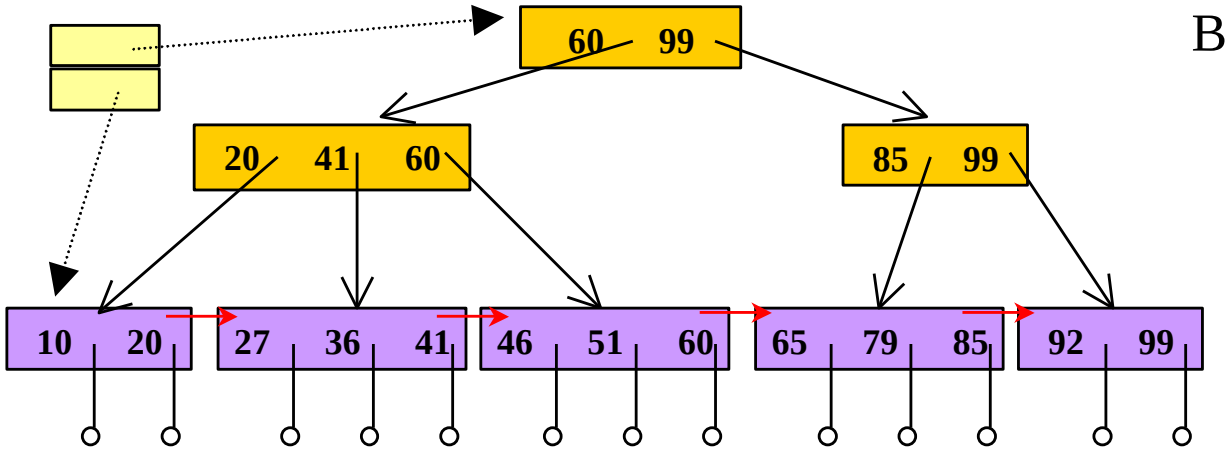
b) 运算

- 检索（2种方法）：

A. 直接从最小关键码开始顺序检索[叶结点构成链表]；

B. 从B+树的根结点开始随机检索；

- 插入：与B树的插入操作相似，但总是插在叶结点上。
- 删除：仅在叶子结点删除关键码。



B+树示例

B树和B+树的比较:

B+树是B树的变形，每个结点存贮在外存储器的一个页块上，对于相同大小的页块，同一字典分别采用B+树或B树进行存贮时，B+树的阶数要比B树的阶数多。因此，在同等条件下，B+树的检索速度快。

小结：

ASL：平均检索长度，基本运算为比较。

静态：只有检索。

顺序（顺序、折半、分块）、

散列（散列函数、碰撞处理）

各种检索的思想、算法、时间复杂度分析（ASL）。

散列检索效率分析，为什么仍然用ASL衡量？

动态：除检索外，还有插入、删除。

二叉排序树（定义、性质、检索、插入、删除，中序遍历）

等概率时：先排序，再折半构造

不等概率时：构造最优二叉排序树

(麻烦，通常用次优查找树代替)

AVL树，B树、B+树（用于外部存储检索）

目前学过的二叉树种类总结：

1）一般二叉树(二叉链表、三叉链表、线索链表)
满二叉树、完全二叉树、扩充二叉树
二叉树的7个重要性质
2）Huffman树（特点，Huffman编码与译码）

3）缩小搜索区间检索（如折半检索）的二叉判定树
（检索过程，特点）
4）二叉排序树（特点、插入与构造、删除、检索）
查找概率不同时：最优查找树、次优查找树；
查找概率相同时：保持平衡、提高检索效率的AVL树。

折半检索的二叉判定树是AVL树？

5）下一章要学习的“堆”排序中二叉树

上机作业：

1）实现分块检索算法

要求：块间采用折半检索，块内采用顺序检索。

采用数据文件方式输入记录序列。

格式：

36，6

10， 30， 20， 15， 25， 5

100， 200，150，250，300，50

500， 550，510，450，580，400

……

1000, 1500, 1200, 2000, 1800, 1300

首先根据文件建立块索引，然后根据给定值检索。

2）实现二叉排序树建立和检索算法

作业（p199~200)：

复习题：2、3、4、5
算法题：3、5

作业（p248~249):

复习题：1、2、3、7
算法题：1

上机作业：1）实现分块检索算法
2）实现二叉排序树建立和检索算法