第 9 章　查 找

- 静态查找表
  - 顺序表的查找
  - 有序表的查找
  - 索引顺序表的查找
- 动态查找表
  - 二叉排序树和平衡二叉树
  - B_树和B^+树
- Hash表
  - 什么是Hash表
  - Hash函数的构造方法
  - 处理冲突的方法
  - Hash表的查找及其分析

基本概念

- 查找表：是由同一类型的数据元素（或记录）构成的集合。表中的每个记录由若干个数据项构成，其中可以唯一地标识一个记录的数据项称为主关键字。

- 查找操作类型：
  ① 在查找表中查询某个记录是否存在
  ② 检索某个记录的各种属性
  ③ 在查找表中插入一个记录
  ④ 从查找表中删除一个记录

- 根据对查找表所进行查找操作的不同，查找表可被分为两类：
  - 静态查找表
  - 动态查找表

- 平均查找长度：为确定记录在查找表中的位置，通常把查找过程中对关键字需要进行的平均比较次数，称为平均查找长度**ASL**。

9.1 静 态 查 找 表

- 若对查找表只作查询和检索操作，则称此类查找表为静态查找表。

- 静态查找表可以有不同的表示方式，在不同的表示方法中，实现查找操作的方法也不同：
  - 顺序表的查找
  - 有序表的查找
  - 静态树表的查找
  - 顺序索引表的查找

9.1.1 顺序表的查找

- ✓ 当以顺序表或线性链表表示静态查找表时，可采用顺序查找法。

- ✓ 顺序查找的过程为：

  从表的一端开始，顺序扫描线性表，依次将扫描到的记录关键字与给定值**K**相比较，若相等，则查找成功；若扫描结束后，仍未找到，则查找失败。

查找表的类型描述：

```
struct Record{
    int key;
    ……
};
Record r[n];
```

顺序表查找算法（基于顺序存储结构）：

```
int SeqSearch( Record r[], int n, int key)
{
   // 在顺序表r中查找其关键字等于给定值的数据元素，
   // 若找到，则返回其在r中的位序，否则返回-1
   i = 0;       // i的初值为第1个元素的位序
   while( i<n && r[i].key!=key )
         ++i;
   if(i<n) return i;
   else return -1;
}
```

顺序表查找算法（改进算法）：

```
int SeqSearch2 (Record r[], int n, int key)
 {
     i=0;
     r[n].key=key;              // 哨兵
     while( r[i].key != key) i++;
     if( i<n ) return i;
     return -1;     // 找不到时，返回-1
 }
```

性能分析：

在等概率的情况下，顺序查找成功时的平均查找长度为：

ASL = (n+1) / 2

不成功时的平均查找长度为：**ASL = n+1**。

所以顺序查找的平均查找长度为：**ASL= 3(n+1) / 4**。

当记录的查找概率不相等时，应把记录按查找概率从大到小重新排列，以提高查找效率。在查找概率无法预先确定时，可以在记录中附设一个频度域，并使顺序表中的记录始终按访问频度非递减有序排列。

顺序查找的优点是：算法简单，且对表的结构无任何要求。

缺点是：查找效率低，当n较大时，不宜采用顺序查找。

9.1.2 有序表的查找

- 当静态查找表是有序表，即表中结点是按关键字有序，并采用顺序存储结构时，可用折半查找法。

- 折半查找的查找的过程是：

  先确定待查记录所在的范围（区间），然后逐步缩小查找范围，直到找到该记录或者找不到为止。

  所谓“折半”的含义是指，先将给定值和所查区间中间位置的记录的关键字进行比较，若相等，则查找成功，否则，依给定值大于或小于该关键字继续在后半个区间或前半个区间中进行查找。

折半查找非递归算法：

```
int BinSearch (Record r[], int n, int key) {
    low=0;                  // 置区间初值
    high=n-1;
    while( low<=high ){
        mid = (low+high)/2;
        if (key == r[mid].key )    // 找到待查元素
            return mid;
        else if (key < r[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 -1;              // 顺序表中不存在待查元素
}
```

折半查找递归算法?

```
int BinSearch2 (Record r[], int low, int high, int key)
{
    if( low>high ) return -1;
    mid = (low+high)/2;
    if (key == r[mid].key )    // 找到待查元素
            return mid;
    else if( key < r[mid].key )
            return BinSearch2 (r, low, mid-1, key);
        else
            return BinSearch2 (r, mid+1, high, key);
}
```

性能分析：

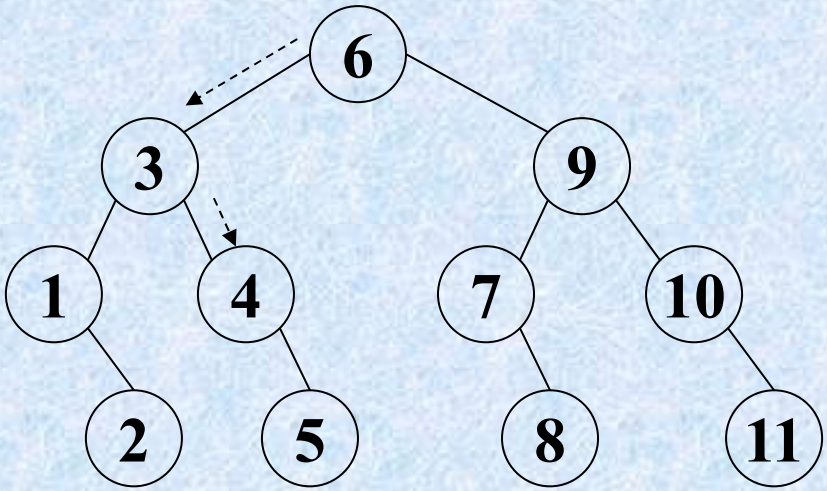
以下面11个数的查找为例：

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

用折半查找法可知：

找到第6个数仅需比较1次，找到第3和第9个数需2次，找到第1，4，7，10个数需3次，找到第2，5，8和11个数需4次。

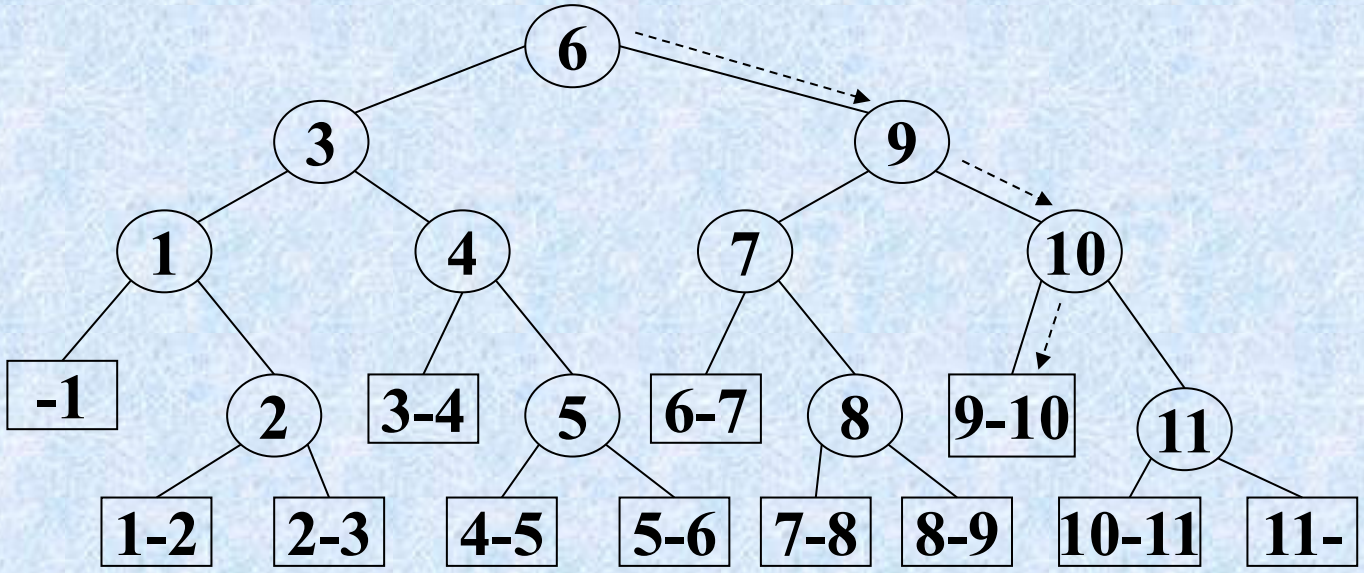
- ✓ 这个查找过程可以用右图的二叉树来表示。
- ✓ 查找某个结点所比较的次数等于该结点的层次数。
- ✓ 实际上查找某个结点的过程就是从根结点到与该记录相应的结点的路径。



二叉判定树

因此，用折半查找法在查找成功时进行比较的次数最多不超过该树的深度。而具有n个结点的判定树的深度为$\lfloor \log_2 n \rfloor+1$。所以折半查找法在查找成功时的比较次数最为$\lfloor \log_2 n \rfloor+1$次。

如果考虑到查找不成功的情况，则判定树如下所示（方框表示查找不成功的情况）：



可见，查找不成功时的最多比较次数也是$\lfloor \log_2 n \rfloor+1$。

现在讨论一下其平均查找长度：

从判定树可见，判定树上同一层次上的所有结点的比较次数相等。第*j*层的比较次数为*j*，该层上的元素个数为2^{j-1}。因此，

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

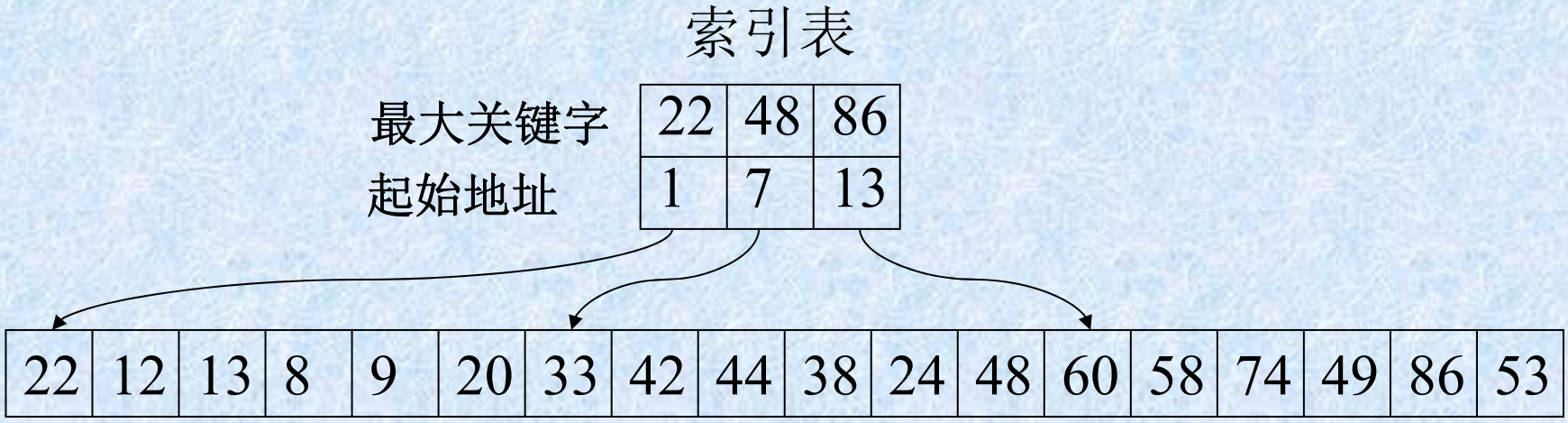
折半查找法的优缺点：

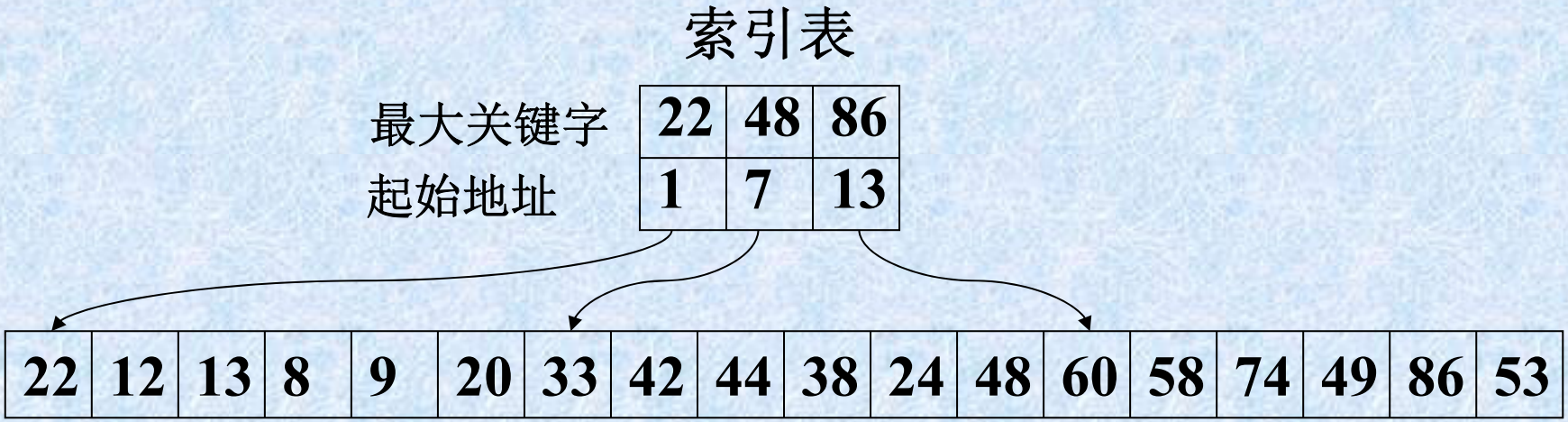
优点：速度快

缺点：只能适用于有序表，且仅限于顺序存储结构

9.1.3 分块查找

- ✓ 分块查找：是一种性能介于顺序查找和二分查找之间的一种查找方法，但它要求先对查找表建立一个索引表，再进行分块查找。
- ✓ 索引表建立方法：先对查找表进行分块，要求“分块有序”，即块内关键字不一定有序，但块间有序。然后，抽取各块中的最大关键字及其起始位置构成一个索引表。





- 分块查找方法：
  - 分两步进行：
    - 先查找索引表，因其有序，故可采用二分法查找，以确定待查记录在哪一块；
    - 然后，在已确定的块中进行再顺序查找。

性能分析：

一般情况下，为进行分块查找，可将长度为n的表均匀地分成b块，每块含有s个记录，即$b=\lceil n/s \rceil$；再假定表中每个记录的查找概率相等，则每块查找的概率为1/b，块中每个记录的查找概率为1/s。

若用顺序查找确定所在块，则

$$ASL = L_b + L_w = (b+1)/2 + (s+1)/2$$

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

9.3 Hash 表

- 什么是Hash表
- Hash函数的构造方法
- 处理冲突的方法
- Hash表的查找及其分析

9.3.1 什么是Hash表

- Hash是一种重要的存储方法，也是一种重要的查找方法。
- 它的基本思想是：
  - 以关键字K为自变量，通过一个确定的函数f，计算出对应的函数值f (k)，把这个值解释为关键字等于K的结点的存储地址；
  - 查找时，再根据要查找的关键字用同样的函数计算地址，然后到相应的存储单元取出要查找的结点；
  - 按这个思想建立的表，称为Hash表，称函数f 为Hash函数，称f (k)的值为Hash地址。

Hash表实例：

已知线性表的关键字集合为：

S = {and, begin, do, end, for, go, if, then, until }

则可设Hash表为： **char HT[26][8]**

Hash函数H(key)的值，可取关键字key中第一个字母在字母表中的序号（0~25），即

H(key) = key[0]- ‘a’

- **Hash函数是一个映射，其设定可以很灵活，只要使得任何关键字的Hash函数值都落在表长允许范围内即可。**

- **对不同关键字可能得到同一Hash地址，这一现象称为“冲突”，而发生冲突的关键字对于该Hash函数来说，称为“同义词”。**
  - ✓ 因关键字集合比Hash表长度大，故冲突不可避免。

- **一般情况下，Hash表的空间必须比结点的集合大，此时虽然浪费了一定的空间，但换取的是查找效率。设Hash表空间大小为m，填入表中的结点数是 n, 则称$\alpha = n/m$ 为Hash表的装填因子。**

9.3.2 Hash函数的构造方法

构造Hash函数时的几点要求：

- **Hash函数的定义域必须包括需要存储的全部关键字，如果Hash表允许有 *m* 个地址时，其值域必须在 0 到 *m*-1 之间。**
- **Hash函数计算出来的地址应能均匀分布在整个地址空间中：若 *key* 是从关键字集合中随机抽取的一个关键字，Hash函数应能以同等概率取 0 到 *m*-1 中的每一个值。**
- **Hash函数应是简单的，能在较短的时间内计算出结果。**

直接定址法

此类函数取关键字的某个线性函数值作为Hash地址：

$$Hash(key) = a * key + b \qquad \{a, b\text{为常数}\}$$

✓ 这类Hash函数是一对一的映射，一般不会产生冲突。但是，它要求Hash地址空间的大小与关键字集合的大小相同。

数字分析法

设有 n 个 d 位数，每一位可能有 r 种不同的符号。

这 r 种不同的符号在各位上出现的频率不一定相同，可能在某些位上分布均匀些；在某些位上分布不均匀，只有某几种符号经常出现。

可根据Hash表的大小，选取其中各种符号分布均匀的若干位作为Hash地址。

- 例如已知80个记录的关键字为8位十进制数（下图列出其中部分），假设Hash表的表长为100，即地址为00～99。

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

✓ 由于关键字中的第 1、2、3 和 8 位取值集中在某几个数上，则应取其余四位中的任意两位或其中两位与另两位之叠加和（舍去进位）作为Hash地址。

- 数字分析法仅适用于事先明确知道表中所有关键字每一位数值的分布情况，它完全依赖于关键字集合。

- 如果换一个关键字集合，选择哪几位要重新决定。

③ 平方取中法

- 通常，要预先估计关键字的数字分布并不容易，要找数字均匀分布的位数则更难。
- **如 （0100，0110，1010，1001，0111）**
- 如果关键字的所有各位分布都不均匀，则可取关键字的平方值的中间若干位作为**Hash**表的地址。
- 由于一个数的平方值的中间几位数受该数所有位影响，将使随机分布的关键字得到的**Hash**函数值也随机。

例如：下列八进制数的关键字及其Hash地址

关键字	(关键字)^2	Hash地址
0100	0010000	010
1100	1210000	210
1200	1440000	440
1160	1370400	370
2061	4310541	310
2062	4314704	314
2161	4734741	734
2162	4741304	741
2163	4745651	745
…		

• 此方法在词典处理中使用十分广泛。它先计算构成关键字的标识符的内码的平方，然后按照Hash表的大小取中间的若干位作为Hash地址。

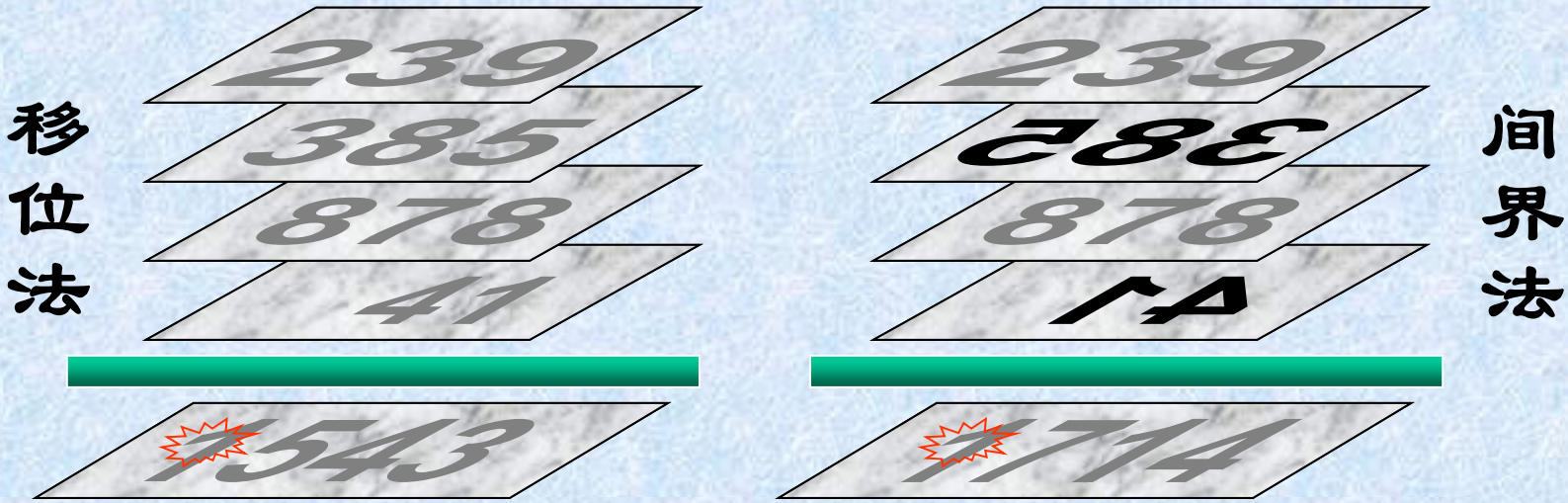
④ 折叠法

- 若关键字的位数很多，且每一位上数字分布大致均匀，则可采用折叠法。
- 方法把关键字自左到右分成位数相等的几部分，每一部分的位数应与Hash表地址位数相同，只有最后一部分的位数可以短一些。
- 把这些部分的数据叠加起来，就可以得到具有该关键字的记录的Hash地址。
- 有两种叠加方法：
  - 移位法 — 把各部分的最后一位对齐相加；
  - 间界法 — 各部分不折断，沿各部分的分界来回折叠，然后对齐相加，将相加的结果当做Hash地址。

- **示例**：设给定的关键字为 $key = 23938587841$，若存储空间限定 3 位, 则划分结果为每段 3 位. 上述关键字可划分为 4段：

  239　385　878　41

- 把超出地址位数的最高位删去, 仅保留最低的3位，作为可用的Hash地址。



⑤ 除留余数法

- 设Hash表中允许的地址数为 m, 取一个不大于 m，但最接近于或等于 m 的质数 p作为除数，利用以下公式把关键字转换成Hash地址。Hash函数为：

$$hash(key) = key \% p \qquad p \le m$$

- 其中, “%”是整数除法取余的运算，要求这时的质数 p 不是接近2的幂。

- 示例：有一个关键字 $key = 962148$，Hash表大小 $m = 25$，即 $HT[25]$。取质数 $p = 23$。Hash函数 $hash(key) = key \% p$。则Hash地址为

$$hash(962148) = 962148 \% 23 = 12。$$

- 可以按计算出的地址存放记录。需要注意的是，使用上面的Hash函数计算出来的地址范围是 0到 22，因此，从23到24这几个Hash地址实际上在一开始是不可能用Hash函数计算出来的，只可能在处理冲突时达到这些地址。

- 以上介绍了几种常用的Hash函数。在实际工作中应根据关键字的特点，选用适当的方法。

- 有人曾用统计分析方法对它们进行了模拟分析，结论是平方取中法最接近于“随机化”。

9.3.3 处理冲突的方法

- 两种处理冲突的基本方法：

  （1）开放定址法

  （2）拉链法

冲突的处理之----开放定址法

- 当冲突发生时，使用某种方法在**Hash**表中形成一个探测序列，沿着此序列逐个地址去探查，直到找到一个开放的地址（空位置），将发生冲突的键值放到该地址中。

- 令探测序列为：

  $H_i = (H(key) + d_i) \% m \quad i=1,2,\ldots,k \ (k<=m)$

- 对于 d_i 通常可有三种设定方法：

  1）$d_i = 1,2,3,\ldots,m-1$，称这种处理冲突的方法为“线性探测再Hash”。

  2）$d_i = 1^2, -1^2, 2^2, -2^2, \ldots, \pm k^2 (k \le m/2)$，称这种处理冲突的方法为“平方（二次）探测再Hash”。

  3）d_i为伪随机数列，称这种处理冲突的方法为"伪随机探测再Hash"。

冲突的处理
----开放定址法之线性探测法

- 对给定的键值k，若地址d(即h(k)=d)的单元发生冲突，则依次探查下述地址单元（设m为表长）：

  d+1, d+2, …., m-1, 0, 1, … d-1

- 插入操作：

  如果找到一个空位置，就把k插入，如果用完整个地址序列仍未找到空位置，则插入失败。

- 查找操作：

  先计算h(k)=d, 到位置d找k，若找不到，则按线性探测地址序列进行查找，若在某地址处找到键值k，则查找成功；否则，若找到一个空位置或用完整个地址序列仍未找到，则查找失败。

例：**假设关键字序列为**

{ 26, 36, 41, 38, 44, 15, 68, 12, 06, 51, 25 }，

用线性探测法构造这组关键字的**Hash**表。

Hash函数为： H(key)=key % 13

Key	26	36	41	38	44	15	68	12	06	51	25
Hash	0	10	2	12	5	2	3	12	6	12	12

0	1	2	3	4	5	6	7	8	9	10	11	12

线性探测法

- 使用线性探测法会产生“堆积”现象：
  - 本不是同义词的键值被当成同义词来处理。
  - 堆积现象增加了查找长度。
  - 在用开放地址法处理冲突时，要删除一个键值k，只能对它作一个删除标记，而不能真正删除。
    - 在插入时，凡遇到删除标记的位置就可以插入；
    - 在查找时，凡遇到有删除标记的位置还应继续查找下去。

双Hash探测法

- “堆积现象”的产生主要是由于探测序列不变引起的，若设法使不同的键值具有不同的探测序列，可尽量避免产生堆积（如双Hash探测方法）。

- 思想方法：

  当h1(k1)=h1(k2)=d时，设法使：

  h2(k1)<>h2(k2), 可避免产生堆积。

双Hash探测法

- 设m为表长，当h1(k1)=h1(k2)=d时，使探测序列为：

  （d+h2(k))%m,（d+2h2(k))%m,（d+3h2(k))%m,

  ……..

冲突的处理之 ---- 拉链法

- 拉链法首先对关键字集合用某一个Hash函数计算它们的存放位置。
- 若设Hash表地址空间的所有位置是从0到m-1，则关键字集合中的所有关键字被划分为m个子集，具有相同地址的关键字归于同一子集。
  - ✓ 同一子集中的关键字互为同义词。每一个子集称为一个桶。
- 通常各个桶中的表项通过一个单链表链接起来，称之为同义词子表。所有桶号相同的表项都链接在同一个同义词子表中，各链表的表头结点组成一个向量。

- **示例：给出一组表项关键字{ Burke, Ekers, Broad, Blum, Attlee, Alton, Hecht, Ederly }。Hash函数为：** ***Hash*** **(*x*) = x[0] - 'A'。**
- **用它计算可得：**

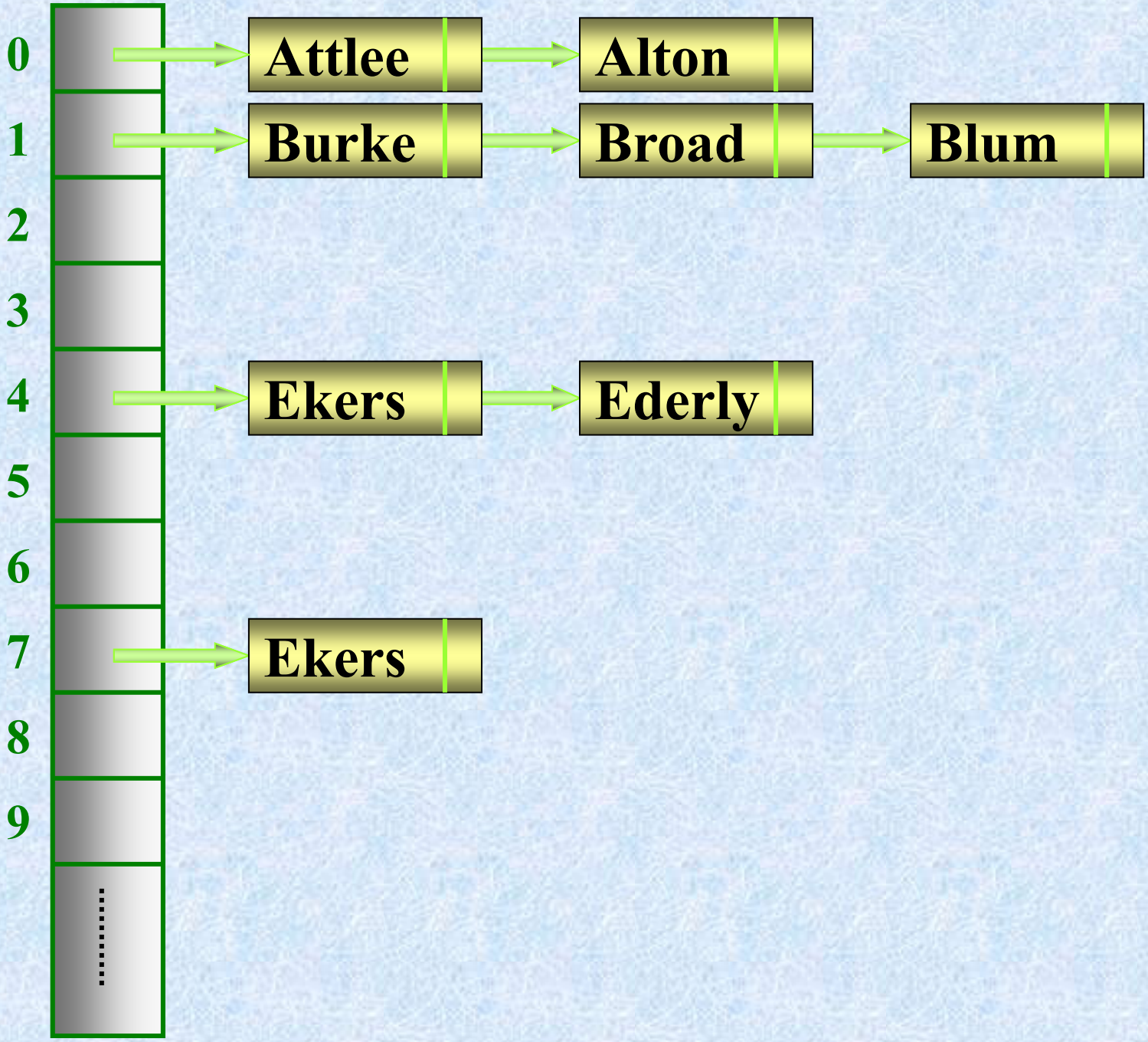
Hash **(Burke) = 1** ***Hash*** **(Ekers) = 4**

Hash **(Broad) = 1** ***Hash*** **(Blum) = 1**

Hash **(Attlee) = 0** ***Hash*** **(Hecht) = 7**

Hash **(Alton) = 0** ***Hash*** **(Ederly) = 4**

- **Hash表为 *HT*[0..25]，*m* = 26。**


0
1
2
3
4
5
6
7
8
9
Attlee
Alton
Burke
Broad
Blum
Ekers
Ederly
Ekers

拉链法的优点

- 不产生“堆积”现象；
- 更适合于造表前无法确定表长的情况；
- 删除结点容易。

9. 3. 4 Hash表的查找和分析

- ✓ 在Hash表上进行查找和Hash建表的过程基本一致。

- ✓ 给定K值，根据建表时设定的Hash函数求得Hash地址，若：
  - 表中没有记录，则查找不成功；
  - 否则比较关键字，若和给定值相等，则查找成功；
  - 否则根据建表时设定的冲突处理方法找“下一地址”，直到Hash表某个位置为空或表中所填记录的关键字等于给定值时为止。

Hash表查找算法 (线性探测法解决冲突):

```
int SearchHash ( int hash[], int m, int key)
{
      pos=k%m;    // 求得Hash地址
      t=pos;
      while ( hash[pos] != EMPTY )  {
          if( hash[pos] == key)
              return pos;
          else
              pos = (pos+1)%m;
          if( pos==t) return -1;
      }
      return -1;
}
```

Hash表的查找性能：

✓ 以Hash表表示查找表原希望其平均查找长度为0，但由于构建Hash表时不可避免会产生冲突，因此在Hash表上进行查找还是需要通过“比较”来确定查找是否成功，因此仍然存在平均查找长度的问题。

✓一般情况下，在Hash函数为"均匀"的前提下，Hash表的平均查找长度仅取决于处理冲突的方法和Hash表的装填因子。

Hash表的查找性能：

在等概率查找的情况下，可以证明：

线性探测再**Hash**查找成功时的平均查找长度为：

$$S_{nl} \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$$

二次探测法与双**Hash**法的平均查找长度为：

$$S_{nr} \approx -\frac{1}{\alpha}\ln(1-\alpha)$$

链地址法的平均查找长度为：

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

- 由分析可见，由于**Hash**表的平均查找长度不是 **n** 的函数，而是 α 的函数，因此虽然不能做到平均查找长度为**0**，但可以设计一个**Hash**表，使它的平均查找长度控制在一个期望值之内。

9.2 树表的查找

1. 二叉排序树

2. 平衡二叉树

3. B- 树和B^+ 树

9.2.1 二叉排序树和平衡二叉树

1. 定义

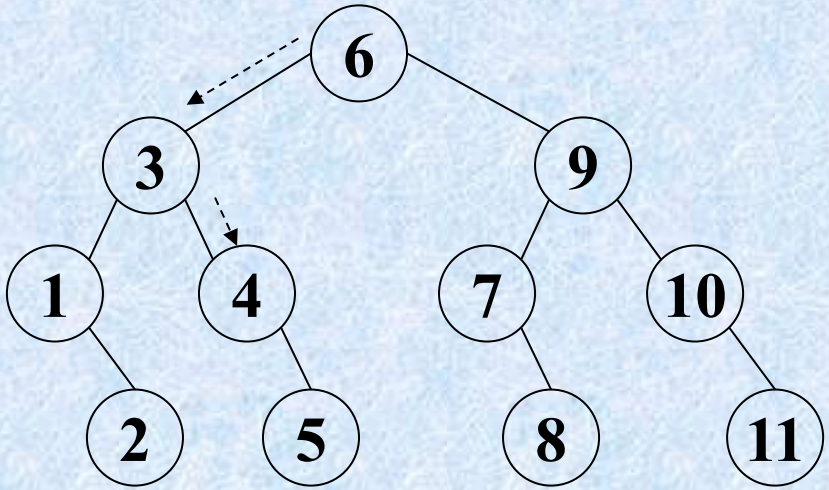
二叉排序树(Binary Sort Tree)或者是一棵空二叉树；或者具有下列性质的二叉树：

(1) 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；

(2) 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；

(3) 它的左右子树也分别为二叉排序树。

实际上，折半查找法的判定树就是一棵二叉排序树。



二叉排序树的存储结构：

```
struct BiNode {
        int key;
        BiNode *lchild, *rchild;
};
```

```
class BiSortTree{
    BiNode *root;
    void Insert(BiNode *&ptr, int k);          //供插入函数调用
    BiNode* Search(BiNode *ptr, int k);        //供查找函数调用
    void Delete (BiNode *&ptr, int k);         //供删除函数调用
   void Free(BiNode *ptr);                     //供析构函数调用
public:
    BiSortTree(int a[ ], int n);    //根据数组a[n]建立二叉排序树
    ~BiSortTree();                  //析构函数
    void Insert(int k);             //插入
    bool Search(int k);             //查找
    void Delete (int k);            //删除
};
```

- 从二叉排序树的定义可得出二叉排序树的一个重要性质：
  - 按中序遍历该树得到的中序序列是一个递增有序序列

- 练习题：设计一个算法，判断一棵二叉树是否为二叉排序树。

```
template <class T>
bool JudgeBiOrder(BiNode<T> *p)
{   if(p==NULL) return true;
    if(p->lchild && p->data<=p->lchild->data)
        return false;
    if(p->rchild && p->data>=p->rchild->data)
        return false;
    if(JudgeBiOrder(p->lchild)
         return JudgeBiOrder(p->rchild)
    else return false;
}
```

二叉查找树的查找算法

- 若二叉查找树为空，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。

```
//在以ptr为根的二叉排序树中查找关键字为k的结点
BiNode*  BiSortTree::Search(BiNode *ptr, int k)
{
   if (ptr==NULL) return NULL;
   else
        if (ptr->key==k)       //查找成功，返回
            return ptr;
        else if (k<ptr->key)
            return Search (ptr->lchild, k);  //查找左子树
        else
            return Search(ptr->rchild, k);  //查找右子树
}
```

二叉排序树查找的非递归算法：

```
BiNode*  BiSortTree::Search2(BiNode *ptr, int k)
{
   while (ptr)
   {    if (k==ptr->key)  return ptr;
        else if (k<ptr->key)  ptr=ptr->lchild;
        else ptr=ptr->rchild;
   }
   return NULL;
}
```

二叉排序树结点的插入

- ✓ 当树中不存在关键字等于给定值的结点时，需要生成新结点并插入到二叉树中。
- ✓ 而新插入的结点必定是一个叶子结点，并且是查找不成功时查找路径上访问的最后一个结点左孩子或右孩子结点.

算法思想：

1. 如果二叉排序树为空，则新结点作为根结点。
2. 如果二叉排序树非空，则将新结点的关键字与根结点的关键字比较，若小于根结点的关键字，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。
3. 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。

```
//在以ptr为根的二叉排序树中插入值为k的结点
void BiSortTree::Insert(BiNode *& ptr, int k)
{
    if (ptr==NULL){
        ptr=new BiNode;
        ptr->key=k;
        ptr->lchild=ptr->rchild=NULL;
    }
    else{
        if (k<ptr->key) Insert(ptr->lchild, k);   //插入到左子树
        else if(k>ptr->key)  Insert(ptr->rchild, k); //插入到右子树
    }
}
```

二叉排序树的删除算法

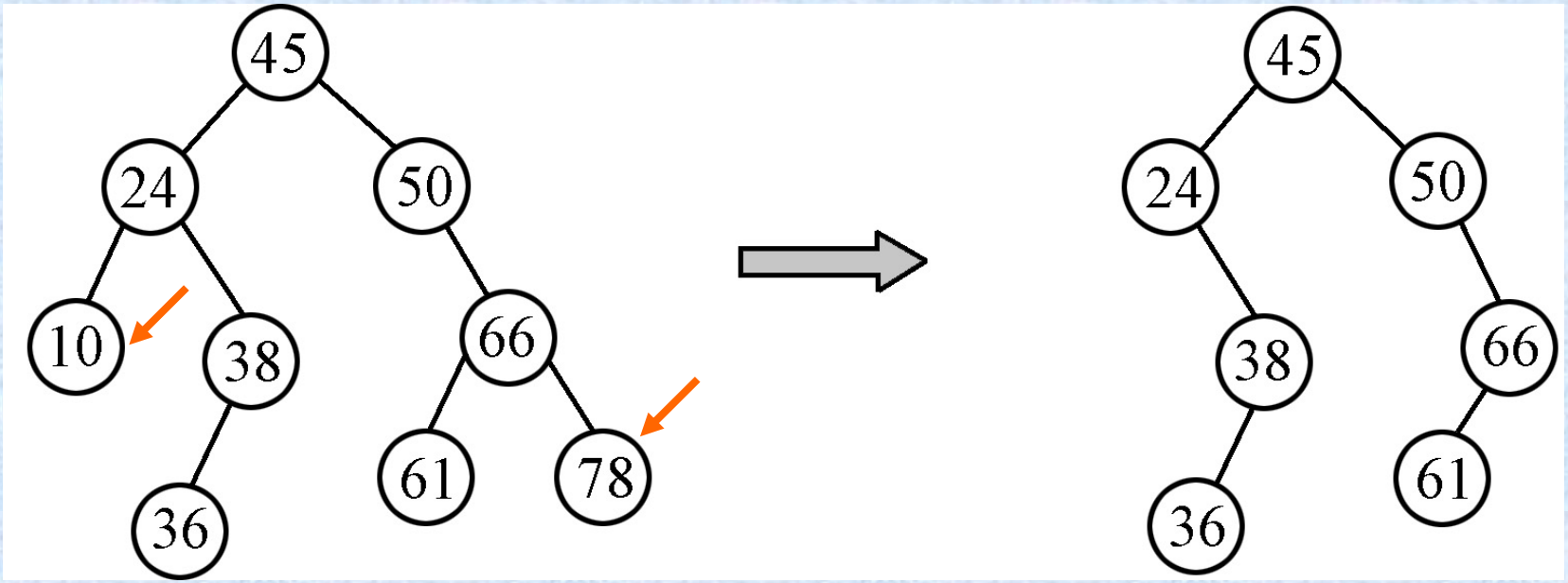
- 在二叉排序树上删除一个结点之后应该仍是一棵二叉树，并保持其二叉排序树的特性。

- 那么在二叉排序树上如何删除一个结点，即如何修改结点的指针？分三种情况讨论：

(1)被删结点为“叶子”

此时删除该结点不影响其它结点之间的关系，因此仅需修改其双亲结点的相应指针即可；

删除关键字为10、78的叶子结点



二叉排序树的删除算法（续）

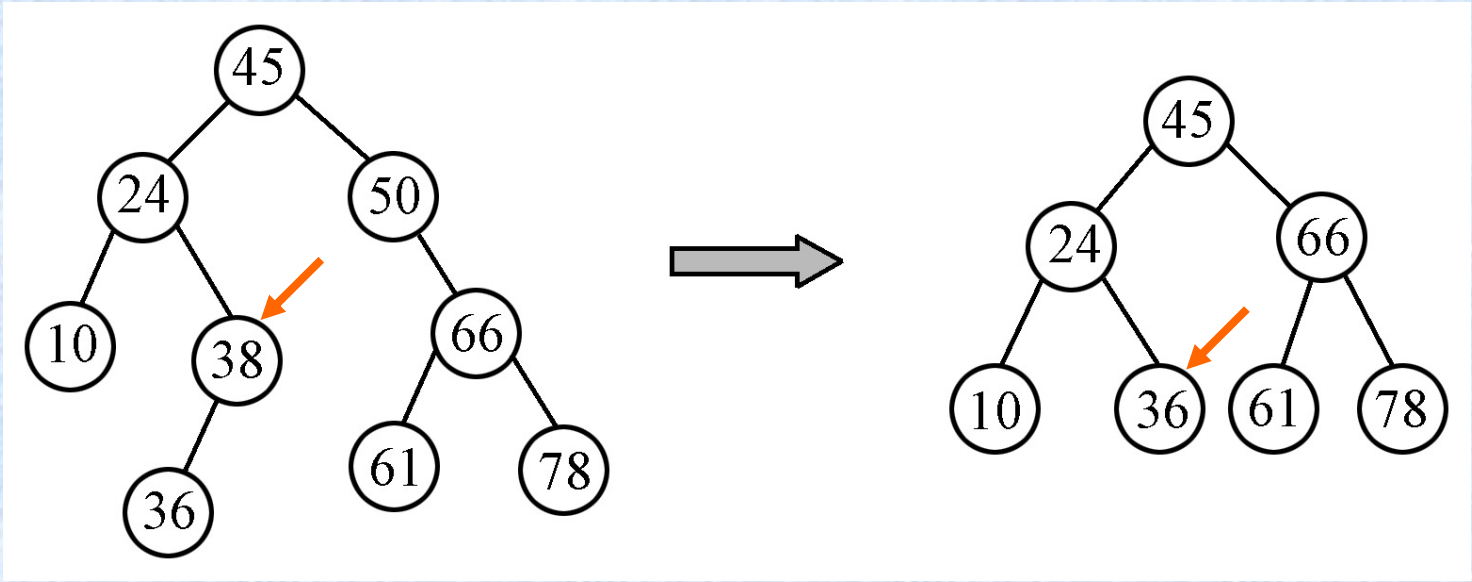
- 那么在二叉排序树上如何删除一个结点，即如何修改结点的指针？分三种情况讨论：

(1) 被删结点为“叶子”

(2) 被删结点只有左子树或右子树

只需保持该结点的子树和其双亲之间原有的关系即可，即删除该结点之后，将其左子树或右子树直接链接到其双亲结点成为其双亲的子树即可；

删除关键字为38的单支结点



二叉排序树的删除算法（续）

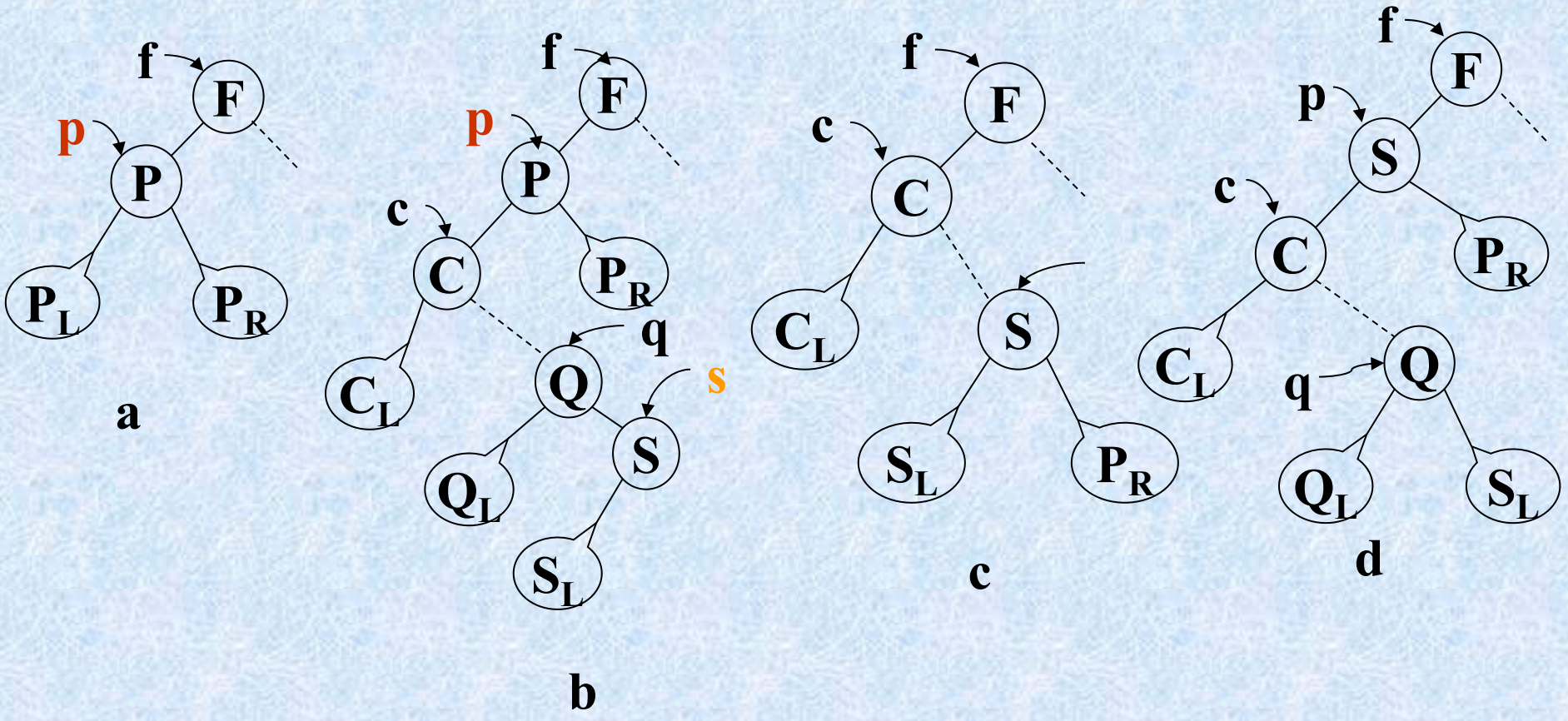
- 那么在二叉排序树上如何删除一个结点，即如何修改结点的指针？分三种情况讨论：

(1) 被删结点为“叶子

(2)被删结点只有左子树或右子树，

(3) 被删结点的左右子树均不空。

此时若将二叉排序树视作一个有序序列，为保持其左子树和其右子树间的有序关系，可将其中序"前驱"替代被删数据元素，即将被删结点的数据元素赋值为它的"前驱"，然后从二叉排序树上删去这个"前驱"结点，使得删除一个结点之后的二叉排序树上其余结点之间的"有序"关系不变，而其前驱结点由于只有左子树容易删除。


f
F
p
P
P_L
P_R
a
f
F
p
P
c
C
P_R
q
C_L
Q
s
S
Q_L
S_L
b
f
F
c
C
C_L
S
S_L
P_R
c
f
F
p
S
c
C
P_R
C_L
q
Q
Q_L
S_L
d

```
void BiSortTree::Delete(BiNode *&ptr, int k){
    if (ptr!=NULL){
        if (k<ptr->key)  Delete(ptr->lchild,k);
        else if (k>ptr->key)  Delete(ptr->rchild,k);
            else{    //ptr指向的结点就是要删除的结点
                if (ptr->lchild!=NULL&&ptr->rchild!=NULL){
                    pre=ptr->lchild;
                    while (pre->rchild!=NULL)
                        pre=pre->rchild;
                    ptr->key = pre->key;
                    Delete(ptr->lchild, pre->key);
                }
                else{
                    temp=ptr;
                    if (ptr->lchild==NULL)  ptr=ptr->rchild;
                    else if (ptr->rchild==NULL)  ptr=ptr->lchild;
                    delete temp;
                }
            }
    }
}
```

三、性能分析

在二叉排序树上查找关键字实际上是走了一条从根结点到某个结点的路径的过程，和给定值比较的次数等于路径长度加1。因此，比较的次数不超过树的深度。

但是具有n个结点二叉树可以有不同的形态，而不同形态的二叉树具有不同的深度，因此，对于不同形态的二叉排序树，其平均查找长度也是不同的。

最坏的情况下蜕变为单支树，此时的平均查找长度为（n+1)/2。

在随机的情况下，平均性能为：

$$P(n) <= 2(1+1/n)ln n$$

由此可见，在随机情况下的平均查找长度和$log n$是等数量级的，然而在某些情况下，还需进行“平衡化”处理。

四. 平衡二叉树 (AVL树)

- 平衡二叉树或者是一棵空树，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。

- 若定义二叉树上结点的平衡因子**BF(Balance Factor)**为该结点的左子树的深度减去右子树的深度，在平衡二叉树上所有结点平衡因子只可能为**-1, 0, 1**。

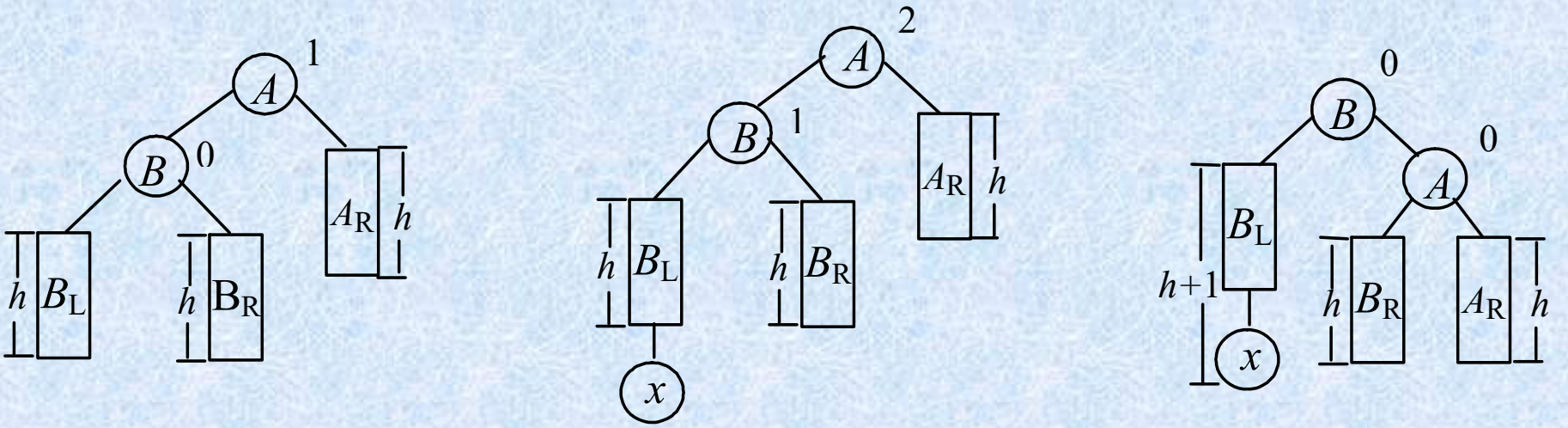
- 只要二叉树上有一个结点的平衡因子的绝对值大于**1**，则该二叉树就是不平衡的。

- 对于平衡二叉树来说，它的深度和***log*n**是同数量级的，因此我们希望任何初始序列构成的二叉排序树都是**AVL**树。
- 如何使构成的二叉排序树成为平衡二叉树呢？可以通过对结点的旋转操作来实现。下面讨论一般的情况。

假设由于在二叉排序树上插入结点而失去平衡的最小子树根结点为A（即A是离插入结点最近，且平衡因子绝对值大于1的祖先结点），则失去平衡后进行调整的规律可归纳为下列四种情况：

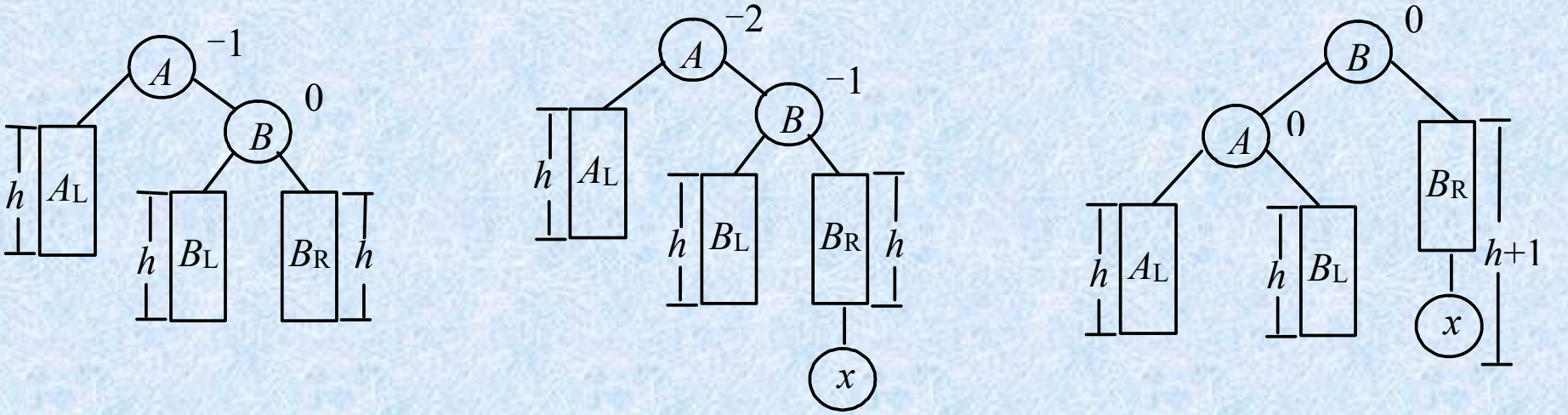
（1）单向右旋平衡处理(LL型)：在A的左孩子的左子树上插入新结点，使得*A*结点的平衡因子由1变为2。

（2）单向左旋平衡处理(RR型)：在A的右孩子的右子树上插入新结点，使得*A*结点的平衡因子由–1变为–2。



(a) 插入前　　(b) 插入后，调整前　　(c) 调整后

LL型调整过程



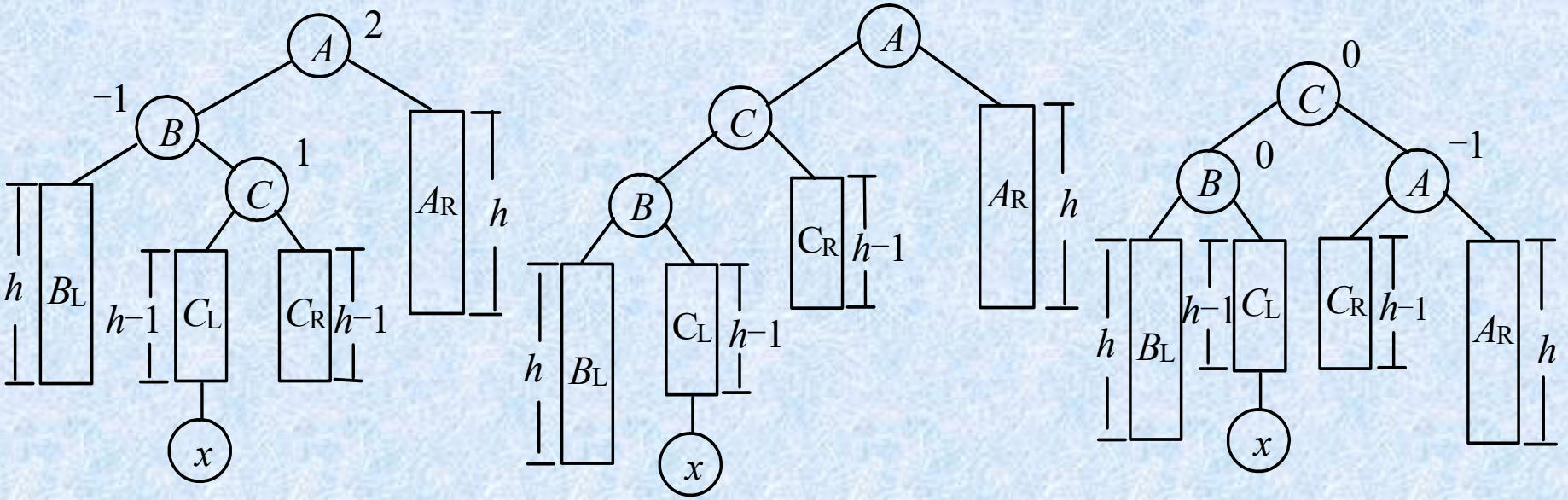
(a) 插入前　　(b) 插入后，调整前　　(c) 调整后

RR型调整过程

假设由于在二叉排序树上插入结点而失去平衡的最小子树根结点为A（即A是离插入结点最近，且平衡因子绝对值大于1的祖先结点），则失去平衡后进行调整的规律可归纳为下列四种情况：

（3）双向旋转（先左后右）平衡处理(LR型)：在A的左孩子的右子树上插入新结点，使得*A*结点的平衡因子由1变为2。

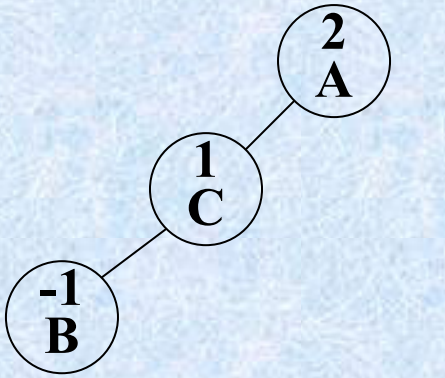
（4）双向旋转（先右后左）平衡处理(RL型)：在A的右孩子的左子树上插入新结点，使得*A*结点的平衡因子由-1变为-2。



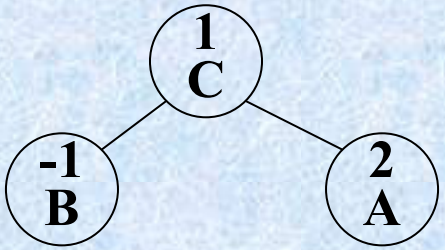
(a) 插入后，调整前　　(b) 先逆时针旋转　　(c) 再顺时针旋转

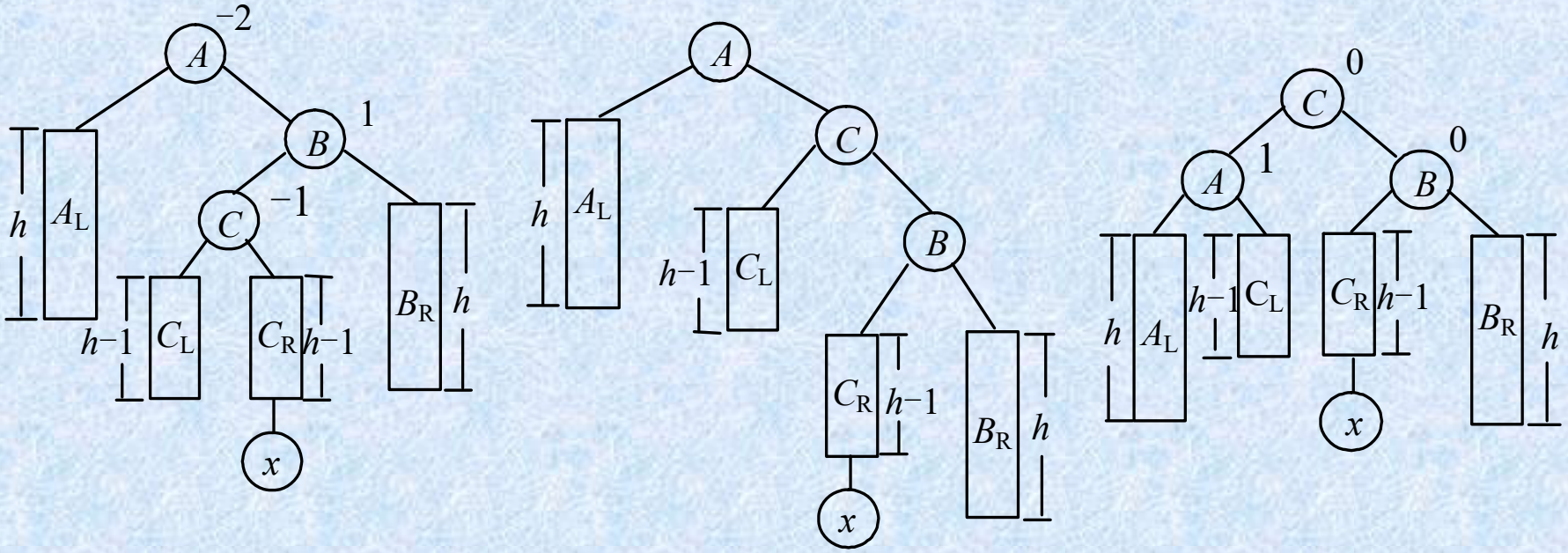
LR型调整过程

先左旋：先对A的左孩子B进行逆时针左旋处理，即变为右面的样子：



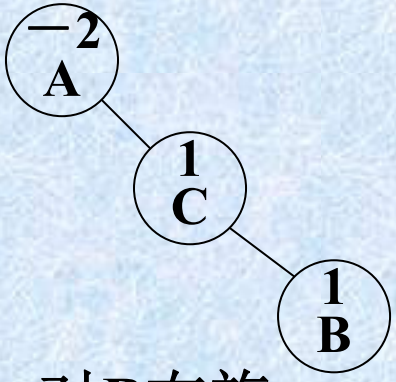
后右旋：然后对A进行顺时针右旋处理，即变为右面的样子：



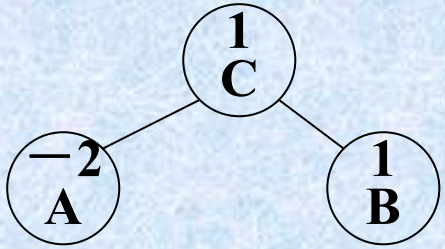


(a) 插入后，调整前　　(b) 先顺时针旋转　　(c) 再逆时针旋转

RL型调整过程



对B右旋



对A左旋

【例】 设一组关键字序列为{4，5，7，2，1，3，6}，试建立一棵平衡二叉树。

The End!