8、查找算法

（1）对查找表经常进行的操作

查询：查询某个“特定的”数据元素是否在查找表中；

检索：检索某个“特定的”数据元素的各种属性；

插入：在查找表中插入一个数据元素；

删除：在查找表中删除某个数据元素。

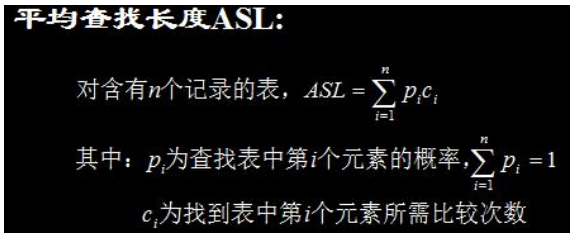
（2）查找表的分类

静态查找表：仅作**查询和检索**操作的查找表

动态查找表：在查找过程中还可以进行**插入和删除**数据元素的查找表

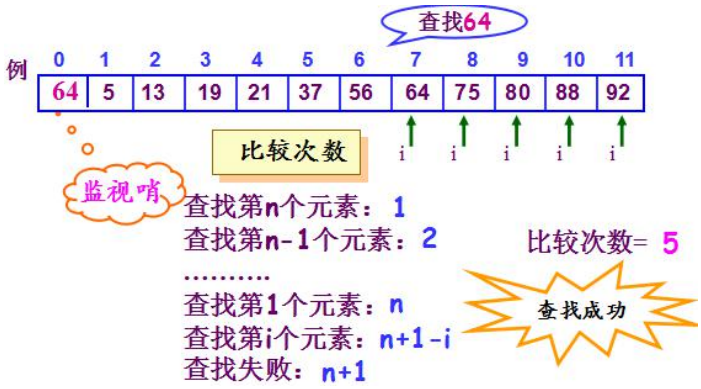
（3）静态查找方法

查找方法的一个重要评价指标：**平均查找长度 ASL**(Average Search Length)



①顺序查找

从表的一端开始，逐个进行关键字和给定值的比较，适用于以**顺序表**或**线性链表**表示的静态查找表，将位置 0 处设置为“监视哨”，放置给定值，如下图所示：



优点：算法简单，对表的逻辑次序和存储结构无要求

缺点：平均查找长度（ASL）较大

②折半查找

每次将待查记录所在**区间缩小一半**，适用于采用**顺序存储结构**的有序静态查找表

算法实现：

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。

初始时，令 low=1,high=n,mid=$\lfloor(low+high)/2\rfloor$，将 mid 指向的记录与 k 比较(假设有序表升序)

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；

重复上述操作，直至找到记录，查找成功；若 low>high，查找失败。

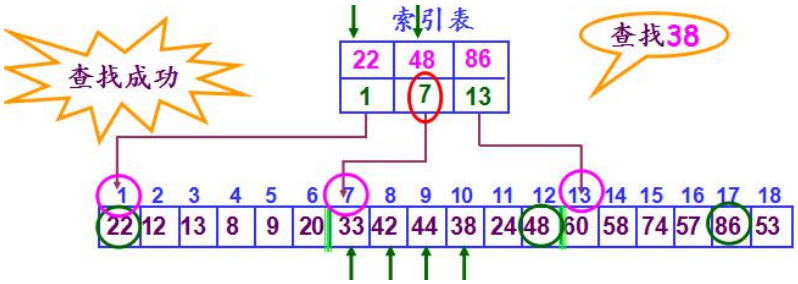
优点：平均查找长度（ASL）较小，查找效率高于顺序查找

缺点：只适用于顺序存储的有序表，不适用于一般的顺序表和链式存储结构

③分块查找（索引顺序查找）

将线性表分成几块，**块内无序，块间有序，**先确定待查记录所在块，再在块内查找，也是一个“**缩小区间**”的查找过程，适用于**分块有序表**。

算法实现：建立**索引表**，每个索引表结点都含有一个**数据域**（本块最大关键字）和一个**指针域**（指向本块第一个结点），将给定值与索引表中的数据域逐个比较，先确定给定值所在的分块，然后再在块内顺序查找，确定是否存在给定值，具体实例如下：



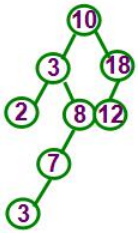
注意：索引表查找可以使用顺序查找和折半查找，但块内查找只能使用顺序查找

④静态查找方法的比较

	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

（4）动态查找表的分类

①二叉排序树（二叉查找树）



定义：二叉排序树或是一棵空树，或是具有下列性质的二叉树：

左、右子树本身又各是一棵二叉排序树。

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**均大于或等于**它的根结点的值；

注意：**中序遍历**二叉排序树可得到一个关键字的**有序序列**

查找过程：若给定值**等于**根结点的关键字，则**查找成功**；

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；

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。

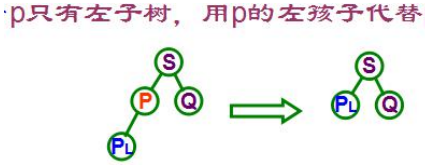
注意：二叉排序树的查找过程类似折半查找，折半查找比较的是给定值和中点值，而二叉排序树比较的是给定值和根节点的值。

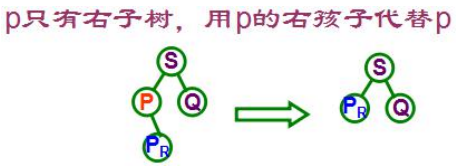
插入过程：相当于在有序序列中插入一条记录，若二叉排序树为空，则插入结点应为新的根结点；否则，根据**查找规则**从根结点开始查找，比较给定值与根节点的值的大小关系，直到某结点的左子树或右子树为空为止，将**新结点作为叶子结点**插入。

删除过程：相当于在有序序列中删除一条记录，要删除二叉排序树中的 p 结点，分三种情况：

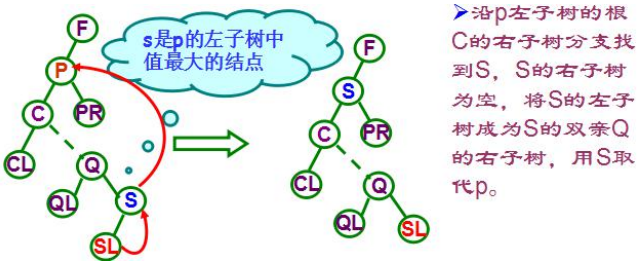
A、若 P 为叶子结点，只需修改 p 双亲 f 的指针 f->lchild=NULL 或 f->rchild=NULL；

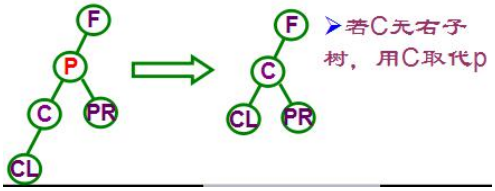
B、若 **p 只有左子树或右子树**，用 p 的左孩子或右孩子代替 p；（**下面一层结点上移**）



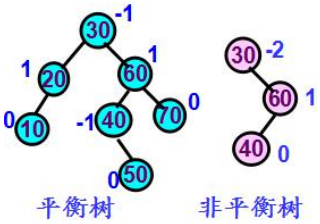


若 **p 左、右子树均非空**，沿 p 左子树的根结点 C 的右子树分支找到 S，此时 S 的右子树为空，让 S 的左子树成为 S 的双亲 Q 的右子树，用 S 取代 p；若 C 无右子树，用 C 取代 p。**（找到 p 的左子树所有结点中值最大的结点 S，用 S 代替 p，然后 S 的整个左子树结点上移）**





②平衡二叉树（具有平衡特性的二叉排序树）



定义：它或是一棵空树，或是具有下列性质的二叉树：

它的左、右子树都是平衡二叉树；

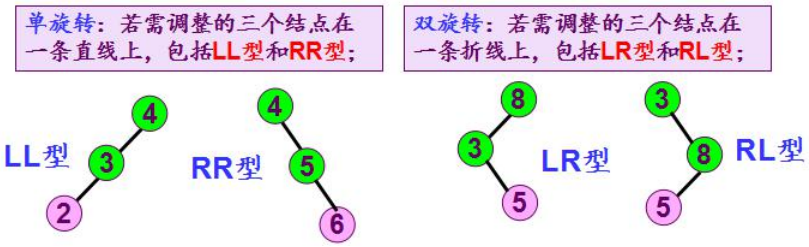
它的**左、右子树深度之差**（平衡因子 BF）**的绝对值不大于 1**。

注意：平衡因子 BF=**左**子树深度-**右**子树深度，可能的值有 1，0，-1

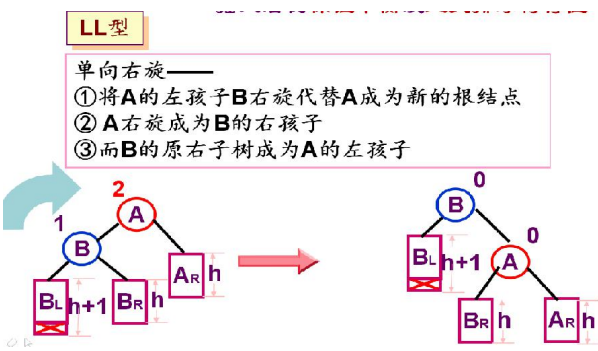
插入过程：插入后仍需保证平衡及二叉排序树特性

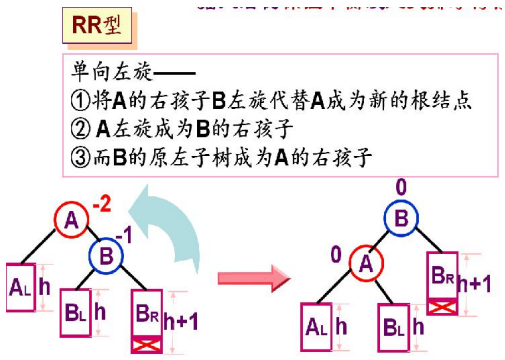
若插入新结点破坏了平衡二叉树的平衡性，则从插入点逆向向根结点方向，将**第一个失衡结点作为新的根结点**，则该结点及其子树构成了**最小不平衡子树**，然后进行调整。

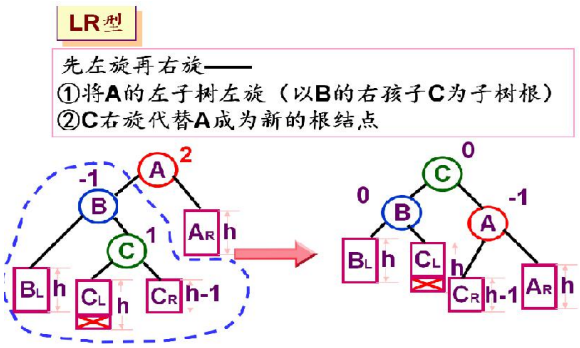
最小不平衡子树类型分为 LL 型、RR 型、LR 型、RL 型，如下图：

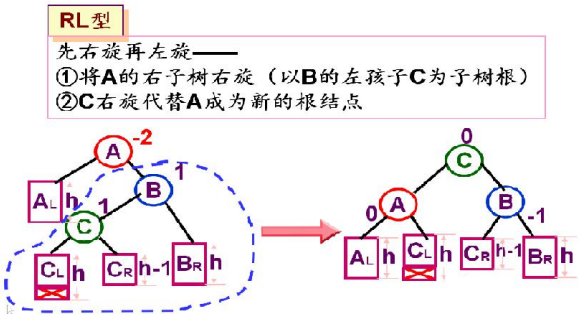


四种最小不平衡子树的插入调整过程为：

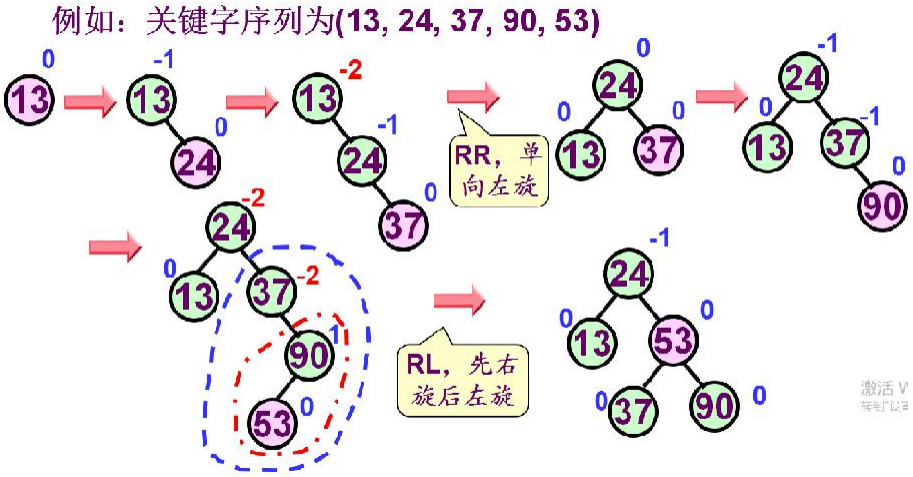








具体实例：给定一个关键字序列，构造平衡二叉树



（5）哈希表及哈希函数

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，**是从**关键字集合**到**存储地址集合**的一种映象

哈希函数可写成：$addr(a_i)=H(k_i)$

- a_i是表中的一个元素
- $addr(a_i)$是a_i的存储地址
- k_i是a_i的关键字

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

哈希查找：又叫散列查找，利用**哈希函数**在**哈希表**中进行查找的过程

哈希函数的构造方法：

①**直接定址法：**取**关键字或关键字的某个线性函数**作哈希地址，即 **H(key)=key 或 H(key)=a • key+b，**该方法所得地址集合与关键字集合大小相等，不会发生冲突，但实际中能用这种哈希函数的情况很少

②**数字分析法（数字选择法）：**对关键字进行分析，取**关键字的若干位或其组合**作哈希地址，适用于关键字位数比哈希地址位数大，且事先知道可能出现的全部关键字的情况

③**平方取中法：**取**关键字平方后的中间几位**作哈希地址，目的是“扩大差别”，平方值的中间位能受到整个关键字中各位的影响，适于不知道全部关键字的情况

④**除留余数法：**取**关键字被 p 除后所得余数**作哈希地址（p 是某个不大于哈希表长 m 的数），即 H(key)=key MOD p，p<=m

特点是：简单、常用；p 的选取很重要； p 选的不好，容易发生冲突

p 一般取小于等于表长 m 的最大素数

哈希冲突：key1!=key2，但 H(key1)=H(key2)的现象叫冲突，又称碰撞（即不同的关键字根据哈希函数存入了同一个地址）

注意：**哈希函数通常是一种压缩映象，所以冲突不可避免，**只能选择一个好的哈希函数，尽量减少产生冲突的机率；同时冲突发生后，应该有**处理冲突的方法**

选择一个好的哈希函数的标准：**能将关键字均匀的分布在存储空间中**

处理冲突的方法：为产生冲突的地址寻找下一个哈希地址

①开放地址法：

●方法：当冲突发生时，形成一个探查序列d_i；沿此序列逐个地址探查，直到找到一个空位置（开放的地址），将发生冲突的记录放到该地址中。

令：Hi——新的哈希地址
H(key)——哈希函数
d_i——增量序列
m——哈希表表长

则：$Hi=(H(key)+di)\ MOD\ m$

●分类

◆线性探测再散列：$d_i=1,2,3,\ldots\ldots m-1$

◆二次探测再散列：$d_i=1^2,-1^2,2^2,\ldots,\pm k^2(k\le m/2)$

◆伪随机探测再散列：d_i=伪随机数序列

开放定址法处理冲突举例

例 表长为12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，
按三种探测再散列处理冲突的方法，将它填入哈希表中

哈希函数值：H(38)=38 MOD 11=5 冲突

(1) 线性探查：$H_1=(5+1)$ MOD 12=6 冲突
$H_2=(5+2)$ MOD 12=7 冲突
$H_3=(5+3)$ MOD 12=8 不冲突

(2) 二次探查：$H_1=(5+1^2)$ MOD 12=6 冲突
$H_2=(5-1^2)$ MOD 12=4 不冲突

(3)设伪随机数序列为9，则
$H_1=(5+9)$ MOD 12=2 不冲突

②再哈希法

●方法：构造若干个哈希函数，当发生冲突时，使用另外一个哈希函数计算哈希地址，即：
Hi=Rhi(key) i=1,2,……k，直到不发生冲突为止

其中：Rhi——不同的哈希函数

●特点：不易再次产生冲突，但计算时间增加

例如：关键字集合 { 19, 01, 23, 14, 55, 68, 11, 82, 36 }
设定哈希函数 H_1(key) = key MOD 11 (表长m=11)
设 H_2(key)=(3*key) MOD 10 +1

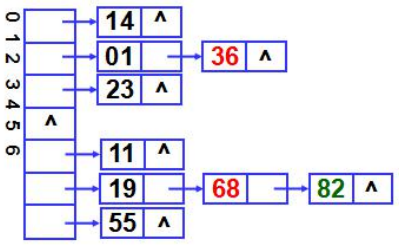
H(19)=8 不冲突
H(01)=1 不冲突
H(23)=1 冲突，H1=(3*23) MOD10+1=10
H(14)=3 不冲突
H(55)=0 不冲突
H(68)=2 不冲突
H(11)=0 冲突，H1=(3*11) MOD10+1=4
H(82)=5 不冲突
H(36)=3 冲突，H1=(3*36) MOD10+1=9

构造结束

③链地址法（外链法）

●方法：将所有关键字为同义词的记录存储在一个单链表中，并用一维数组存放单链表的头指针

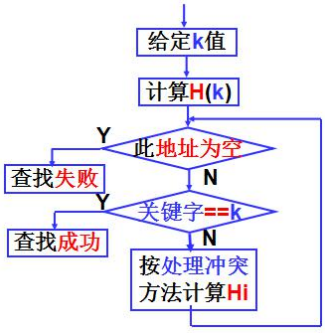
例 已知一组关键字(19, 01, 23, 14, 55, 68, 11, 82, 36)
5 1 2 0 6 5 4 5 1
哈希函数为：H(key)=key MOD 7,
用链地址法处理冲突



哈希查找过程与分析

由于冲突的存在，哈希查找过程仍是一个给定值与关键字进行比较的过程

评价哈希查找效率仍要用 ASL，选取不同的哈希函数及处理冲突的方法，其 ASL 的计算不同



9、排序算法

（1）排序

定义:将一个数据元素(或记录)的任意序列，重新排列成一个按**关键字有序**的序列

分类：（按照排序依据规则）

插入排序：直接插入排序、折半插入排序、**希尔排序**

交换排序：冒泡排序、**快速排序**

选择排序：简单选择排序、堆排序

归并排序

基数排序

排序基本操作都包括两步：**比较**两个关键字大小；将记录从一个位置**移动**到另一个位置

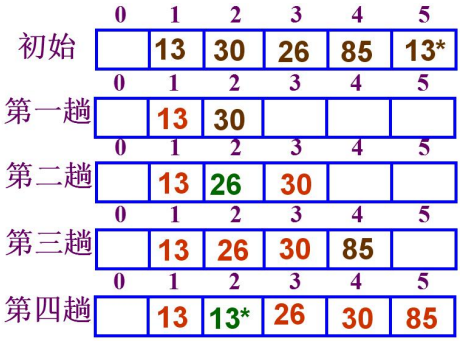
排序算法的**稳定性**：待排序数列中如果有关键字相等的记录，经某一种算法排序后，关键字相等的记录**其先后次序始终不变**，则称排序算法为稳定的，具有稳定性；否则具有不稳定性

（2）插入排序

每次将一个待排序的记录，**按关键字的大小插入到**已排好序的子序列中的**适当位置**，直到全部记录插入完毕为止

①直接插入排序（稳定的排序方法）

排序过程：先将序列中**第 1 个记录**看成是**一个有序子序列**，然后**从第 2 个记录开始**，逐个进行插入，直至整个序列有序；整个排序过程为 **n-1 趟插入**。



✺待排序记录为R[i]，将其与R[j] (j=i-1)进行比较——
➢若R[i]>R[j]——R[i]位置不变
➢若R[i]<R[j]——①将R[i]放在“监视哨”的位置；
②R[j]后移，j--，直到R[i] ≧R[j]；
③将R[0]放置于R[j+1]的位置

算法描述：

❖算法描述

```
void InsertSort( SqList &L)
{
  int i, j;
  for(i=2; i<=L.length; i++)
  {
    if(L.r[i].key<L.r[i-1].key)
    {  L.r[0]=L.r[i];
       for(j=i-1;L.r[0].key<L.r[j].key;j--)
             L.r[j+1]=L.r[j];
       L.r[j+1]=L.r[0];
     }
   }
}
```

从第2个记录开始排序

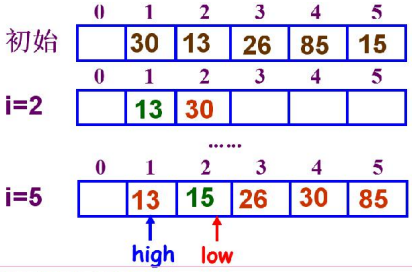
与有序子序列的最后一个记录比较

若待排序记录较小，则元素后移

将待排序记录放置在r[j+1]的位置

②折半插入排序

排序过程：用**折半查找方法**确定插入位置



- 待排序记录为R[i]
- low=1，high=i-1，mid=(low+high)/2
- 将R[i]与R[mid]比较，进行折半查找，直到low>high
- 将i之前low之后元素后移
- 将R[0]插入到low所指示位置

```
void BinSort( SqList &L)                    从第2个记录开始排序
{ int i, j, high, low, mid;
  for(i=2; i<=L.length; i++)
  { L.r[0]=L.r[i];                          设置监视哨
    low=1; high=i-1;
    while(low<=high)                        折半查找
    {  mid=(low+high)/2;
       if(L.r[0].key<L.r[mid].key)
            high=mid-1;
       else  low=mid+1;     }
    for(j=i-1; j>=low; j--)                 记录后移
       L.r[j+1]=L.r[j];
    L.r[low]=L.r[0];                        插入
  }
}
```

③希尔排序（不稳定的排序算法）

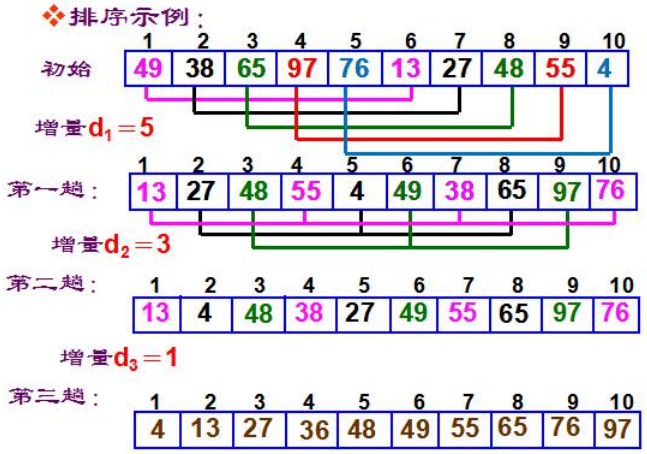
基本思想：先将整个待排序记录**“跳跃式”分割成若干个子序列**分别进行**直接插入排序**，待整个序列中的记录“基本有序”时，再对**全体记录**进行一次**直接插入排序**

对待排记录先作“宏观”调整，再作“微观”调整。

“宏观”调整——“跳跃式”的插入排序。

特点：子序列的构成不是简单的“逐段分割”，而是将相隔某个增量的记录组成一个子序列；希尔排序可提高排序速度，因为时间复杂度减小了，关键字较小的记录跳跃式前移；

排序过程：先取一个**正整数 d1<n，把所有相隔 d1 的记录放一组**，各组内进行直接插入排序；然后取 **d2<d1**，重复上述分组和排序操作；直至 **di=1**，即所有记录放进一个组中排序为止（增量序列取无除 1 以外的公因子，最后一个增量值必须为 1）



（3）交换排序

两两比较待排序记录的关键值，**交换**不满足顺序要求的记录，直到全部满足顺序要求为止

①冒泡排序：每次比较相邻的两个记录（稳定的排序算法）

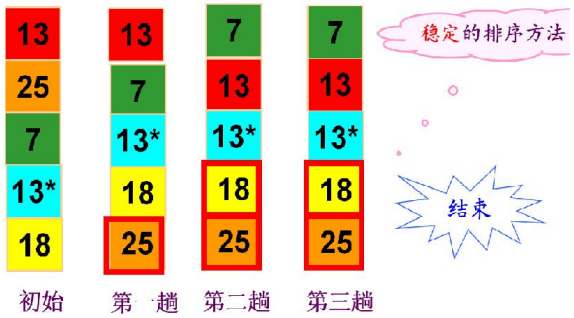
排序过程：将**第一个**记录与**第二个**记录的关键字进行比较，若为**逆序则交换**；然后比较**第二个**记录与**第三个**记录；依次类推，直至**第 n-1 个**记录和**第 n 个**记录比较为止。

第一趟冒泡排序，结果**关键字最大的记录**被安置在**最后一个记录**上；

对**前 n-1 个记录**进行**第二趟**冒泡排序，结果使**关键字次大的记录**被安置在**第 n-1 个记录**位置；

重复上述过程，直到“**在一趟排序过程中没有进行过交换记录的操作**”为止

特点：冒泡排序最多执行 n-1 趟；第 i 趟需要比较 n-i 次；
适用于元素较少或初始序列基本有序的情况




```
void BubbleSort( SqList &L)
{  int m, j, flag=1;          上一趟排序有无交换操作，1(有)0(无)
   m=L.length-1;              最多进行L.length-1趟排序
   while((m>0)&&(flag= =1))   当flag为1且排序趟数小于L.length-1时进行冒泡排序
   {  flag=0;
      for(j=1;j<=m;j++)
         if(L.r[j].key>L.r[j+1].key)   若相邻的两个元素逆序
         {  flag=1;                    交换
            L.r[0]=L.r[j];  L.r[j]=L.r[j+1];  L.r[j+1]=L.r[0];
         }
      m--;                       排序趟数-1
   }
}
```

②快速排序：每次比较不相邻的两个记录（不稳定的排序算法）

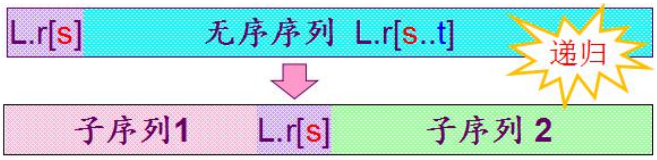
基本思想：在待排序的 n 个记录中任取一个记录(通常为**第一个记录**),称其为“**枢轴**”。再以“**枢轴**”的关键值为标准，将其它记录分为两组:

第一组中各记录的关键值**均小于**枢轴的键值；

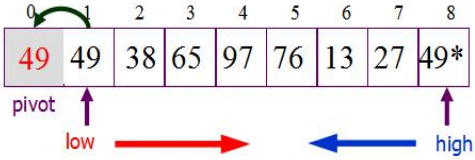
第二组中各记录的关键值**均大于**枢轴的键值；

然后把**枢轴**排在这**两组的中间**(即该记录最终的位置)。此称为**一趟快速排序**。

再分别对两个子序列重复上述过程，直到所有记录有序。



排序过程：

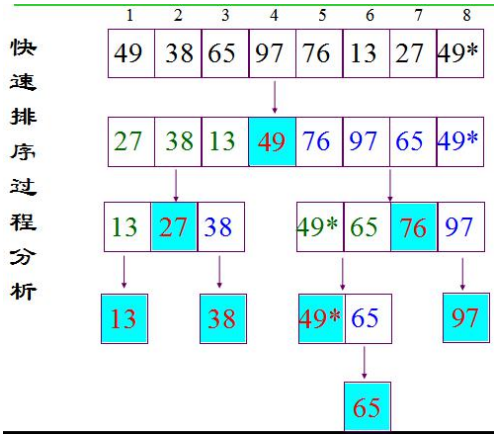


设置位置标志low和high，初始指向待排序记录的首尾；

首先，从high端开始向前搜索，直到找到一个关键值小于pivot的记录，将其放置在low所指位置，low后移；

然后，从low端开始向后搜索，直到找到一个关键值大于pivot的记录，将其放置在high所指位置，high前移；

直到low==high,一趟快速排序结束。



算法分析：（递归实现）

首先对 L.r 中**从 low 到 high 的记录**（low 初始为 1，high 初始为 L.length）进行一趟快速排序，

记录**枢轴位置 pivotloc**；

对 L.r 中从 **low 到枢轴位置 pivotloc -1** 的记录进行快速排序；

对 L.r 中从枢轴位置 **pivotloc +1 到 high** 的记录进行快速排序；

不断重复上述步骤，直到 low=high，快速排序结束。

算法描述：

❖算法描述 一趟快速排序

```
int Partition(SqList &L, int low, int high)
{  KeyType pivotkey;
   L.r[0]=L.r[low]; pivotkey=L.r[low].key;   保存枢轴
   while(low<high)                            一趟快速排序
   {
     while(L.r[high].key>=pivotkey &&low<high) high--;
     if(low<high) {L.r[low]=L.r[high]; low++;}
     while(L.r[low].key<=pivotkey&&low<high ) low++;
      if(low<high) {L.r[high]=L.r[low]; high--;}
   }
   L.r[low]=L.r[0];                           放置枢轴
   return low;
}
```

❖算法描述 快速排序

```
void QuickSort(SqList &L, int low, int high)
{
   int pivotloc;
   if(low<high)
   {
      pivotloc=Partition(L, low, high);
      QuickSort(L, low, pivotloc-1);
      QuickSort(L, pivotloc+1, high);
   }
}
```

第一次调用函数 QuickSort 时，待排序记录序列的上下界分别为 1 和 L.length。 QuickSort(L,1,L.length);

算法评价：

时间复杂度：最好情况（每次总是选到**中间值**作枢轴）$T(n)=O(n\log_2 n)$

最坏情况（每次总是选到**最小或最大元素**作枢轴）$T(n)=O(n^2)$

空间复杂度：需**栈空间**以实现递归　　最坏情况：$S(n)=O(n)$　　一般情况：$S(n)=O(\log_2 n)$

就**平均时间**而言，**快速排序算法**被认为是**内部排序方法中最好的一种**，但快速排序不适合对小规模的序列进行排序

（4）选择排序

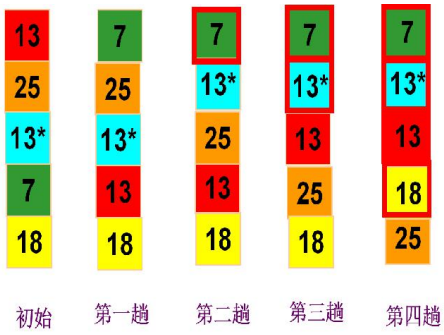
每次从待排序记录中**选出关键字最小的记录，顺序放在**已排好序的记录序列的**后面**，直到全部排完为止

①简单选择排序（不稳定的排序算法）

排序过程：首先通过 **n-1 次关键字比较**，从 n 个记录中找出**关键字最小的记录**，将它**与第一个记录交换；**

再通过 **n-2 次比较**，从剩余的 n-1 个记录中找出**关键字次小的记录**，将它与**第二个记录交换**

重复上述操作，共进行 **n-1 趟排序**后，排序结束



算法描述：

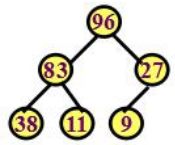
```
void SelectSort( SqList &L)
{  int i, j, k;   k——最小记录下标
   for(i=1;i<L.length;i++)        L.length-1趟排序
   { k=i;
     for(j=i+1;j<=L.length;j++)
        if(L.r[j].key<L.r[k].key)   查找最小记录
           k=j;
     if(i!=k)   若最小记录不在“适当”位置上
     { L.r[0].key=L.r[i];          交换
       L.r[i]=r[k];
       L.r[k]=L.r[0].key;    }
   }
}
```

②堆排序（不稳定的排序算法）

堆定义：n 个元素的序列($k_1,k_2,\cdots\cdots k_n$)，当且仅当满足下列关系时称之为堆：

$$\begin{cases} k_i \le k_{2i} \\ k_i \le k_{2i+1} \end{cases} \text{或} \begin{cases} k_i \ge k_{2i} \\ k_i \ge k_{2i+1} \end{cases} \quad (i=1,2,\ldots\ldots\lfloor n/2 \rfloor)$$

例 （96，83，27，38，11，9）



大项堆(max heap)

例 （13，38，27，50，76，65，49，97）

可将堆序列看成完全二叉树，则堆顶元素（完全二叉树的根）必为序列中n个元素的最小值或最大值

小顶堆(min heap)

基本思想：将无序序列**建成一个堆**，得到**关键字最小(或最大)**的记录；输出堆顶的最小(大)值后，使**剩余的 n-1 个元素重又建成一个堆**，则可得到 n 个元素的次小值；重复执行，得到一个有序序列，这个过程叫堆排序

堆排序需解决的两个问题及解决方法：

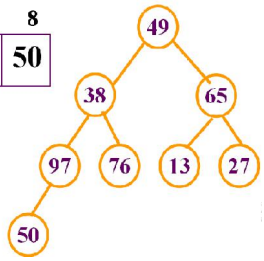
A、如何由**一个无序序列建成一个堆**？

方法：从无序序列的**第$\lfloor n/2 \rfloor$个元素**（即此无序序列对应的**完全二叉树的最后一个非叶子结点**）起，至第一个元素止，进行反复筛选。

例如，对无序序列

1	2	3	4	5	6	7	8
49	38	65	97	76	13	27	50

通过反复筛选
初建成小顶堆

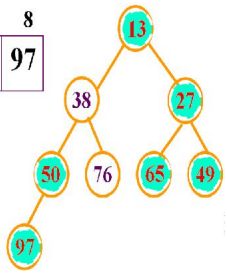


例如，对无序序列

1	2	3	4	5	6	7	8
13	38	27	50	76	65	49	97

通过反复筛选
初建成小顶堆

初建堆结束

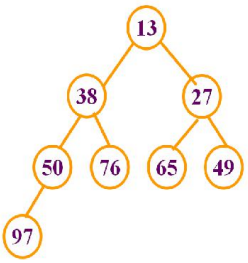


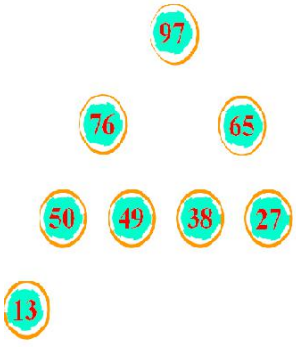
B、如何在输出堆顶元素之后**调整剩余元素**，使之成为一个新的堆？

方法：输出**堆顶元素(根)**之后，以堆中**最后一个元素替代**之；然后将根结点值与左、右子树的根结点值进行比较，并与其中小(大)者进行交换；重复上述操作，直至叶子结点，将得到新的堆，称这个从堆顶至叶子的调整过程为“筛选”。

例如，已知小顶堆，通过筛选，重新调整成小顶堆

1	2	3	4	5	6	7	8
13	38	27	50	76	65	49	97





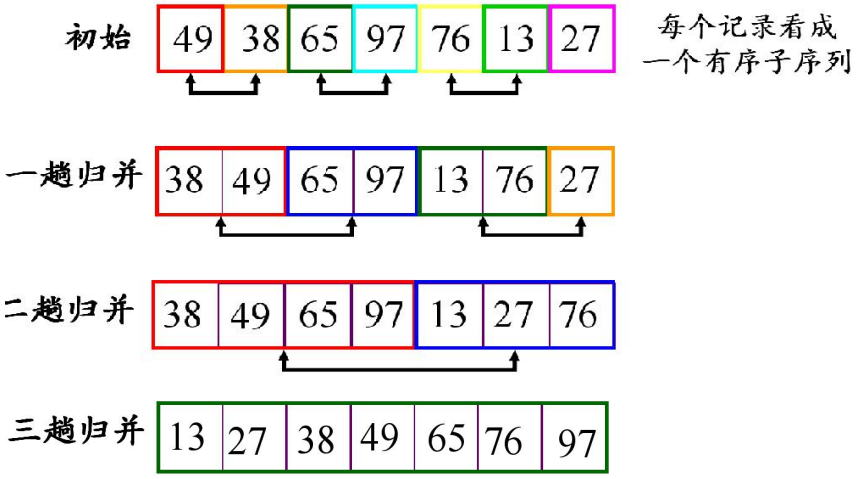
① 取得堆顶元素值
用最后一个元素代替
堆顶元素

② 将剩余元素调整成一个新的小顶堆

利用小顶堆得到的是降序序列

（5）归并排序（稳定的排序算法）

将两个或两个以上的有序表组合成一个新的有序表



（6）基数排序（稳定的排序算法）

基数排序是一种借助“**多关键字排序**”的思想来实现“**单逻辑关键字排序**”的内部排序算法。

（7）排序方法的比较

排序方法	平均时间	最坏时间	辅助空间	稳定性
直接插入	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
简单选择	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	不稳定
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定
基数排序	$O(d(n+rd))$	$O(d(n+rd))$	$O(n+rd)$	稳定

选择排序方法考虑因素：

■记录个数n　　■记录大小　　■记录键值分布

■稳定性要求　　■辅助存储空间大小

排序方法的选取规则：

(1)**n 较小**时，可采用简单排序方法(**直接插入、简单选择和冒泡排序**)；

(2)**n 较大**时，应采用**快速排序或堆排序**。要求**稳定性**时，可采用**归并排序**；

(3)若待排序记录已基本有序时，应采用冒泡排序；

(4)多关键字时（或关键字可分解时），可采用基数排序；

(5)多种排序方法可结合使用。