第九章　内部排序

- ☑ 概述
- ☑ 插入排序
- ☑ 快速排序
- ☑ 选择排序
- ☑ 归并排序
- ☑ 基数排序

概述 --相关概念

- **排序：**将一组杂乱无章的数据按一定的规律顺次排列起来。
- **数据表(datalist)：**它是待排序数据元素的有限集合。
- **排序码(key)：**通常数据元素有多个属性域, 即多个数据成员组成, 其中有一个属性域可用来区分元素, 作为排序依据。该域即为排序码。每个数据表用哪个属性域作为排序码，要视具体的应用需要而定。

概述 --相关概念

- **排序算法的稳定性：**如果在元素序列中有两个元素$r[i]$和$r[j]$, 它们的排序码 $k[i] == k[j]$, 且在排序之前, 元素$r[i]$排在$r[j]$前面。如果在排序之后, 元素$r[i]$仍在元素$r[j]$的前面, 则称这个排序方法是稳定的, 否则称这个排序方法是不稳定的。
- **内排序与外排序：**内排序是指在排序期间数据元素全部存放在内存的排序；外排序是指在排序期间全部元素个数太多，不能同时存放在内存，必须根据排序过程的要求，不断在内、外存之间移动的排序。

概述 --相关概念

- **排序的时间开销：**排序的时间开销是衡量算法好坏的最重要的标志。排序的时间开销可用算法执行中的**数据比较次数与数据移动次数来衡量**。

  算法运行时间代价的大略估算一般都按平均情况进行估算。对于那些受元素排序码序列初始排列及元素个数影响较大的，需要按最好情况和最坏情况进行估算。
- **算法执行时所需的附加存储：**评价算法好坏的另一标准。

概述 --内部排序方法的分类

- 按排序依据原则
  - 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：2-路归并排序
  - 基数排序

概述 --内部排序的存储结构

```
#define MAXSIZE 20
typedef int KeyType;
typedef struct{
    KeyType key;  //关键字项
    InfoType otherinfo;  //其它数据项
}RedType;  //记录类型
typedef struct{
    RedType r[MAXSIZE+1];  //r[0]闲置或作哨兵
    int length;
}SqList;
```


插入排序

插入排序 (Insert Sorting)的基本方法是：每步将一个待排序的元素，按其排序码大小，插入到前面已经排好序的一组元素的适当位置上, 直到元素全部插入为止。

- 直接插入排序
  - 基本思想是：当插入第i (i≥1) 个元素时，前面的V[0], V[1], …, V[i-1]已经排好序。这时，用V[i]的排序码与V[i-1], V[i-2], …的排序码顺序进行比较，找到插入位置即将V[i]插入，原来位置上的元素向后顺移。

插入排序 --直接插入排序(过程演示)

下标：	0	1	2	3	4	5	6	7	8
初始关键字:	(49)	38	65	97	76	13	27	49	
i=2:	38	(38	49)	65	97	76	13	27	49
i=3:	65	(38	49	65)	97	76	13	27	49
i=4:	97	(38	49	65	97)	76	13	27	49
i=5:	76	(38	49	65	76	97)	13	27	49
i=6:	13	(13	38	49	65	76	97)	27	49
i=7:	27	(13	27	38	49	65	76	97)	49
i=8:	49	(13	27	38	49	49	65	76	97)

哨兵

插入排序 --直接插入排序算法

```
// 对顺序表L作直接插入排序
void InsertSort(SqList &L) {
  for (i=2; i<=L.length; ++i)
    if (LT(L.r[i].key, L.r[i-1].key)) {
        L.r[0] = L.r[i];                // 复制为哨兵
        L.r[i]=L.r[i-1];
        for (j=i-2;  LT(L.r[0].key, L.r[j].key);  --j)
              L.r[j+1] = L.r[j];            // 记录后移
        L.r[j+1] = L.r[0];              // 插入到正确位置
    }
} // InsertSort
```


插入排序 --直接插入排序

性能分析：

平均情况下排序的时间复杂度为 $O(n^2)$。

空间复杂度为$O(1)$。

直接插入排序是一种稳定的排序方法。

插入排序 --折半插入排序

- 基本思想是：设在顺序表中有一个元素序列 V[0], V[1], …, V[n-1]。其中, V[0], V[1], …, V[i-1] 是已经排好序的元素。在插入V[i] 时, 利用折半搜索法寻找V[i] 的插入位置。
- 折半插入排序算法

```
// 对顺序表L作折半插入排序。
void BInsertSort(SqList &L) {
   int i,j,high,low,m;
```


插入排序 --折半插入排序

```
for (i=2; i<=L.length; ++i) {
   L.r[0] = L.r[i];      // 将L.r[i]暂存到L.r[0]
   low = 1;   high = i-1;
   while (low<=high) {
     m = (low+high)/2;   // 折半
     if (LT(L.r[0].key, L.r[m].key)) high = m-1;
     else  low = m+1; }
   for (j=i-1; j>=high+1; --j) L.r[j+1] = L.r[j];  // 后移
   L.r[high+1] = L.r[0];      // 插入
 }} // BInsertSort
```


插入排序 --折半插入排序

- 分析
  - 折半插入排序是一个稳定的排序方法。
  - 当 n 较大时，总排序码比较次数比直接插入排序的最坏情况要好得多，但比其最好情况要差。
  - 在元素的初始排列已经按排序码排好序或接近有序时，直接插入排序比折半插入排序执行的排序码比较次数要少。折半插入排序的元素移动次数与直接插入排序相同，依赖于元素的初始排列。

插入排序--希尔排序

- 基本思想

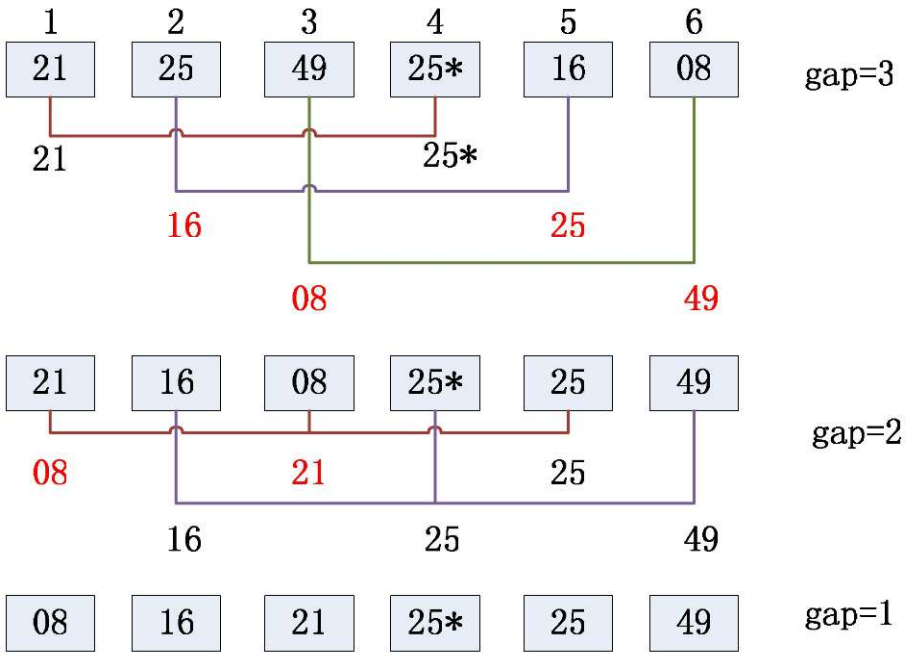
  希尔排序方法又称为缩小增量排序。该方法的基本思想是：设待排序元素序列有 n 个元素, 首先取一个整数 gap < n 作为间隔，将全部元素分为一些子序列，所有距离为 gap 的元素放在同一个子序列中，在每一个子序列中分别施行直接插入排序。然后缩小间隔 gap, 例如取 gap = gap/2,重复上述的子序列划分和排序工作。直到最后取 gap == 1，将所有元素放在同一个序列中排序为止。

插入排序--希尔排序

开始时 gap 的值较大，子序列中的元素较少，排序速度较快; 随着排序进展，gap 值逐渐变小, 子序列中元素个数逐渐变多，由于前面工作的基础，大多数元素已基本有序，所以排序速度仍然很快。

希尔排序是一种不稳定的排序方法。

希尔排序 --排序过程演示



希尔排序

希尔排序 --希尔排序的算法

```
void ShellInsert(SqList &L, int dk) {
  // 对顺序表L作一趟希尔插入排序。
  //本算法对算法10.1作了以下修改：
  //    1. 前后记录位置的增量是dk，而不是1；
  //    2. r[0]只是暂存单元，不是哨兵。
  //当j<=0时，插入位置已找到。
  int i,j;
```


希尔排序 --希尔排序的算法

```
for (i=dk+1; i<=L.length; ++i)
   if (LT(L.r[i].key, L.r[i-dk].key)) {
      // 需将L.r[i]插入有序增量子表
      L.r[0] = L.r[i];  // 暂存在L.r[0]
      for (j=i-dk; j>0 && LT(L.r[0].key, L.r[j].key); j-=dk)
            L.r[j+dk] = L.r[j]; // 记录后移，查找插入位置
      L.r[j+dk] = L.r[0];   // 插入
   }
} // ShellInsert
```


希尔排序 --希尔排序的算法

```
void ShellSort(SqList &L, int dlta[], int t) {
  // 按增量序列dlta[0..t-1]对顺序表L作希尔排序。
  for (int k=0; k<t; ++k)
    ShellInsert(L, dlta[k]);  // 一趟增量为dlta[k]的插入排序
} // ShellSort
```


起泡排序--基本思想

- 将第一个记录的关键字与第二个记录的关键字进行比较，若为逆序r[1].key>r[2].key，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

起泡排序--排序过程演示

初始	一	二	三	四	五	六	第几趟排序后
49	38	38	38	38	13	**13**	
38	49	49	49	13	27	**27**	
65	65	65	13	27	38	**38**	
97	76	13	27	49	**49**	**49**	
76	13	27	49	**49**	**49**	**49**	
13	27	49	**65**	**65**	**65**	**65**	
27	49	**76**	**76**	**76**	**76**	**76**	
49	**97**	**97**	**97**	**97**	**97**	**97**	

快速排序--基本思想

- 任取待排序元素序列中的某个元素 (例如取第一个元素) 作为基准，按照该元素的排序码大小，将整个元素序列划分为左右两个子序列：
  - 左侧子序列中所有元素的排序码都小于或等于基准元素的排序码
  - 右侧子序列中所有元素的排序码都大于基准元素的排序码
- 基准元素则排在这两个子序列中间(这也是该元素最终应安放的位置)。
- 然后分别对这两个子序列重复施行上述方法，直到所有的元素都排在相应位置上为止。

快速排序 --排序过程演示

pivotkey

49	38	65	97	76	13	27	49
i							j
27	38	65	97	76	13		49
i						j	
27	38		97	76	13	**65**	49
		i				j	
27	38	**13**	97	76		65	49
		i			j		
27	38	13		76	**97**	65	49
			i		j		
27	38	13		76	97	65	49
			ij				
27	**38**	**13**	**49**	**76**	**97**	**65**	**49**

完成一趟排序

快速排序--算法描述

```
void QSort(SqList &L, int low, int high) {
  // 对顺序表L中的子序列L.r[low..high]进行快速排序
  int pivotloc;
  if (low < high) {// 长度大于1
    pivotloc = Partition(L, low, high);  // 划分
    QSort(L, low, pivotloc-1); // 对低子表递归排序
    QSort(L, pivotloc+1, high); // 对高子表递归排序
  }
} // QSort
```


快速排序--快速排序的算法

```
int Partition(SqList &L, int low, int high) {
// 交换顺序表L中子序列L.r[low..high]的记录，使枢
//轴记录到位，并返回其所在位置，此时，在它之前
//（后）的记录均不大（小）于它
  KeyType pivotkey;
  L.r[0] = L.r[low]; // 用子表的第一个记录作枢轴记录
  pivotkey = L.r[low].key;     // 枢轴记录关键字
```


快速排序--快速排序的算法

```
while (low<high) {// 从表的两端交替地向中间扫描
    while (low<high && L.r[high].key>=pivotkey) --high;
    L.r[low] = L.r[high];
    while (low<high && L.r[low].key<=pivotkey) ++low;
    L.r[high] = L.r[low];
}
  L.r[low] = L.r[0];          // 枢轴记录到位
  return low;                 // 返回枢轴位置
} // Partition
```


快速排序

- 分析
  - 快速排序是一种不稳定的排序方法。
  - 对于 n 较大的平均情况而言, 快速排序是“快速”的, 但是当 n 很小时, 这种排序方法往往比其它简单排序方法还要慢。
  - 因此， 当n很小时可以用直接插入排序方法。

选择排序

- 基本思想

  每一趟 (例如第 i 趟, i = 1, …, n-1) 在后面 n-i+1 个待排序元素中选出排序码最小的元素，作为有序元素序列的第 i 个元素。待到第 n-1 趟作完，待排序元素只剩下1个，就不用再选了。

选择排序--直接选择排序

- ☑ 直接选择排序的基本步骤是：

① 在一组元素 L.r[i]～L.r[n] 中选择具有最小排序码的元素；

② 若它不是这组元素中的第一个元素，则将它与这组元素中的第一个元素对调；

③ 在这组元素中剔除这个具有最小排序码的元素。在剩下的元素L.r[i+1] ～ L.r[n]中重复执行第①、②步，直到剩余元素只有一个为止。

选择排序--直接选择排序

- 排序过程演示

	1	2	3	4	5	6
i	21	25	49	25*	16	08
1	08	25	49	25*	16	21
2	08	16	49	25*	25	21
3	08	16	21	25*	25	49
4	08	16	21	25*	25	49
5	08	16	21	25*	25	49
	08	16	21	25*	25	49

选择排序

选择排序--直接选择排序的算法

```
void SelectSort(SqList &L) {  // 算法10.9 对顺序表L作简单选择排序。
for ( i=1; i<L.length; ++i) {
    j = SelectMinKey(L, i);  // 在L.r[i..L.length]中选择key最小的记录
    if (i!=j) {              // L.r[i]←→L.r[j];  与第i个记录交换
     RedType temp;
     temp=L.r[i];
     L.r[i]=L.r[j];
     L.r[j]=temp; }   }
} // SelectSort
```

- ☑ 直接选择排序是一种不稳定的排序方法。

选择排序--锦标赛排序

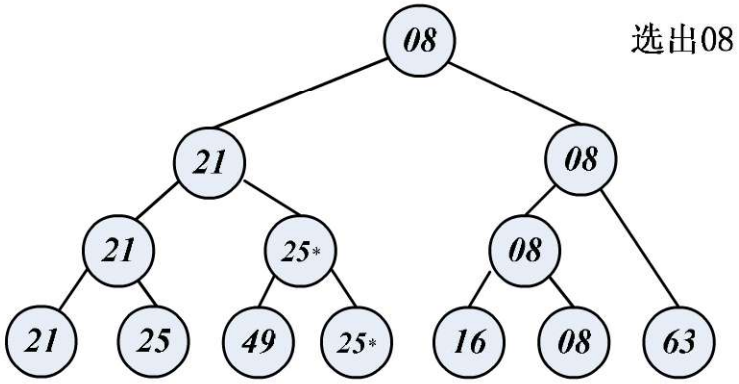
- 锦标赛排序思想

  它的思想与体育比赛时的淘汰赛类似。首先取得 n 个元素的排序码，进行两两比较，得到$\lceil n/2 \rceil$个比较的优胜者(排序码小者)，作为第一步比较的结果保留下来。然后对这$\lceil n/2 \rceil$个元素再进行排序码的两两比较, …, 如此重复，直到选出一个排序码最小的元素为止。

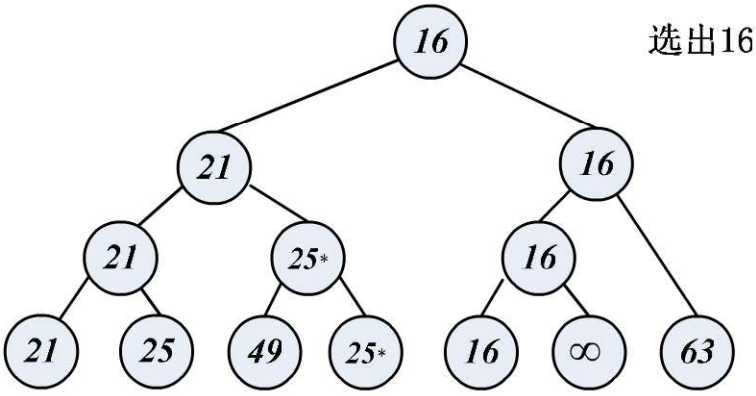
  由于每次两两比较的结果是把排序码小者作为优胜者上升到双亲结点，所以称这种比赛树为**胜者树**。

选择排序--锦标赛排序

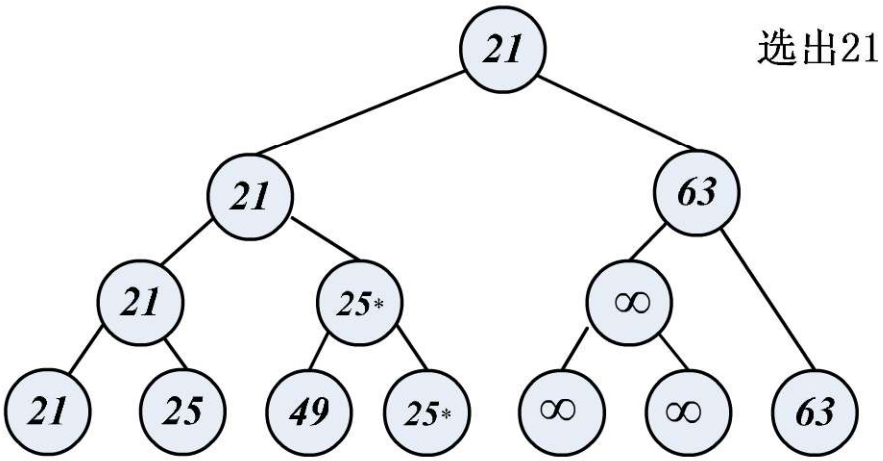
- 锦标赛排序算法演示



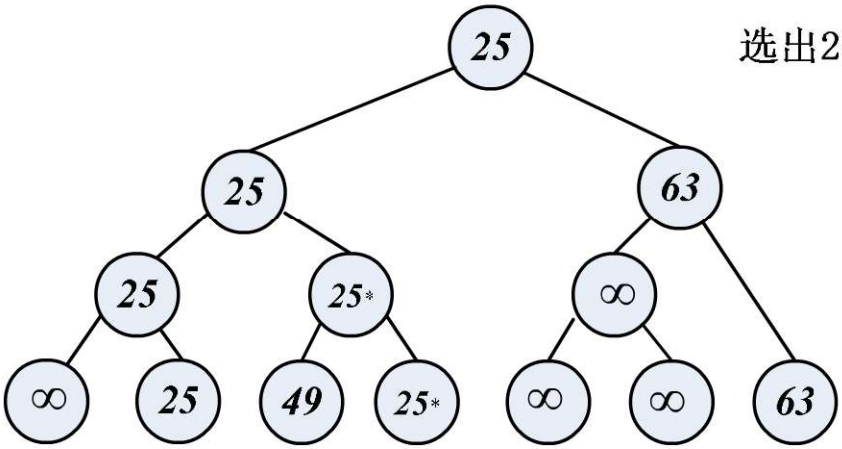
选择排序--锦标赛排序



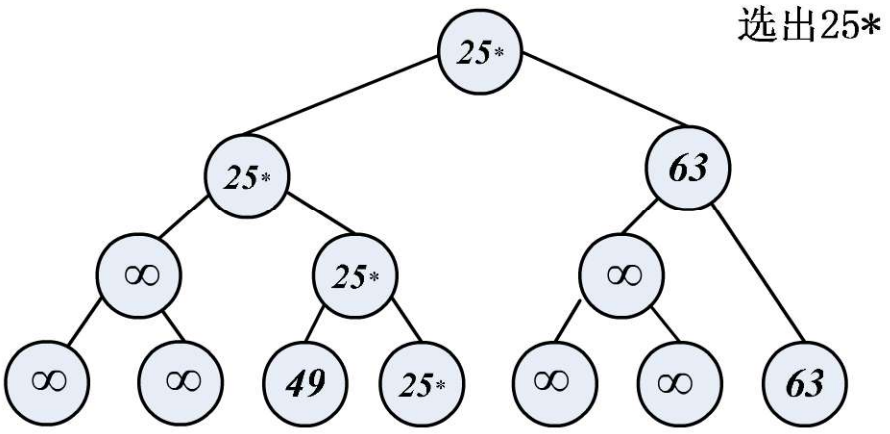
选择排序--锦标赛排序



选择排序--锦标赛排序

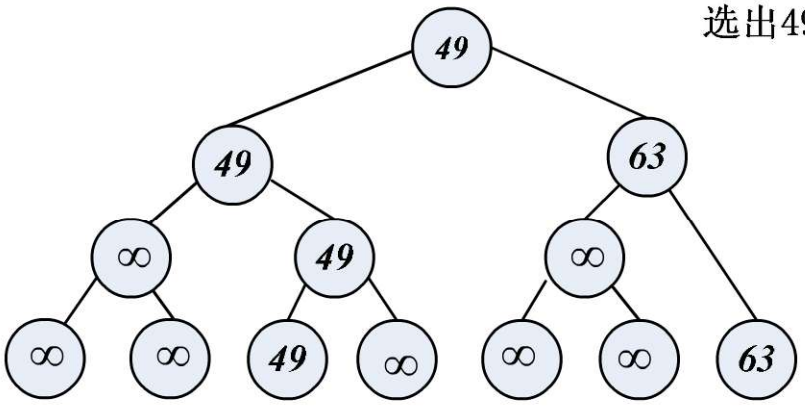


选择排序--锦标赛排序

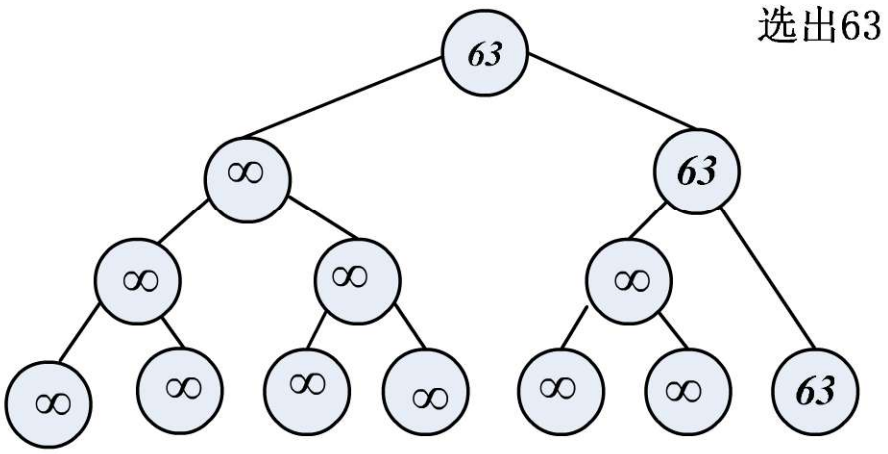


选择排序--锦标赛排序

选出49



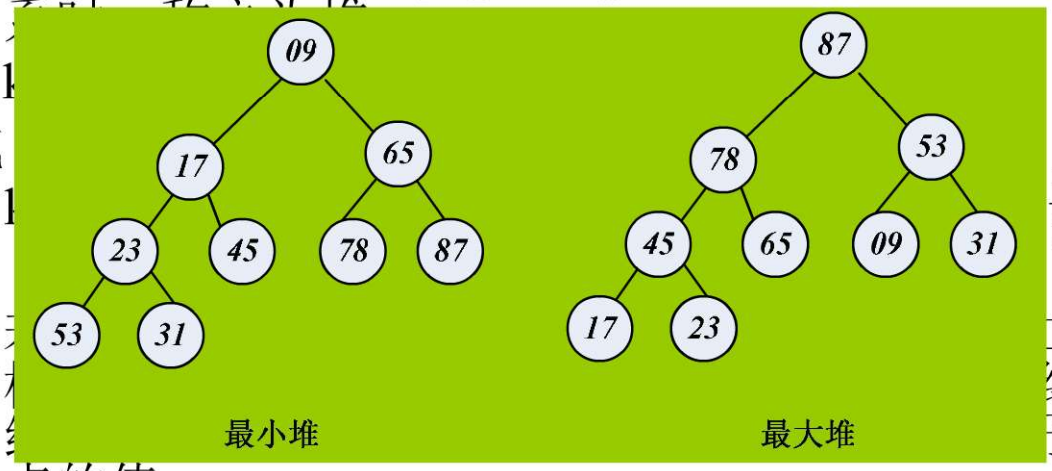
选择排序--锦标赛排序



补充: 堆　　--最小堆和最大堆

- 堆的定义

  n个元素的序列{$k_1,k_2,\ldots,k_n$}当且仅当满足下列关



或

$\lfloor \frac{n}{2} \rfloor$

二叉
终端
子结

点的值。

选择排序--堆排序

- ☑ 堆排序思想

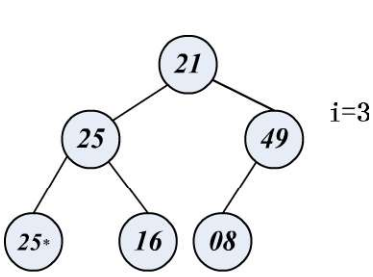
堆排序分为两个步骤：

① 根据初始输入数据，利用堆的调整算法HeapAdjust( )形成初始堆;

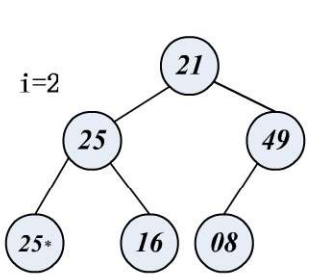
② 通过一系列的元素交换和重新调整堆进行排序。

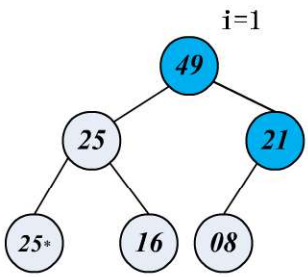
为了实现元素按排序码从小到大排序，要求建立最大堆。

选择排序--堆排序



i=3





1	2	3	4	5	6
21	25	49	25*	16	08

1	2	3	4	5	6
21	25	49	25*	16	08

1	2	3	4	5	6
49	25	21	25*	16	08

建立初始的最大堆

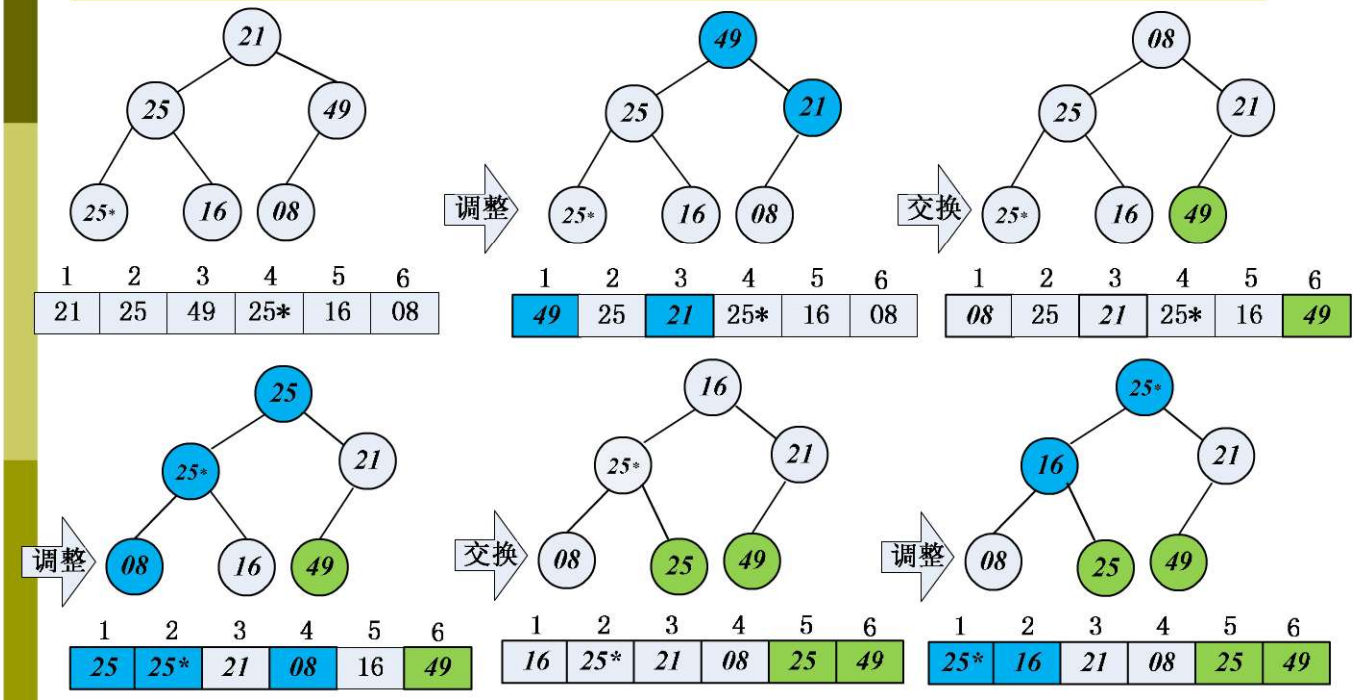
选择排序--堆排序

```
void HeapAdjust(HeapType &H, int s, int m) {
// 算法10.10 .已知H.r[s..m]中记录的关键字除
//H.r[s].key之外均满足堆的定义，本函数调整H.r[s]
//的关键字，使H.r[s..m]成为一个大顶堆（对其中记
//录的关键字而言）
  int j;
  RedType rc;
  rc = H.r[s];
```

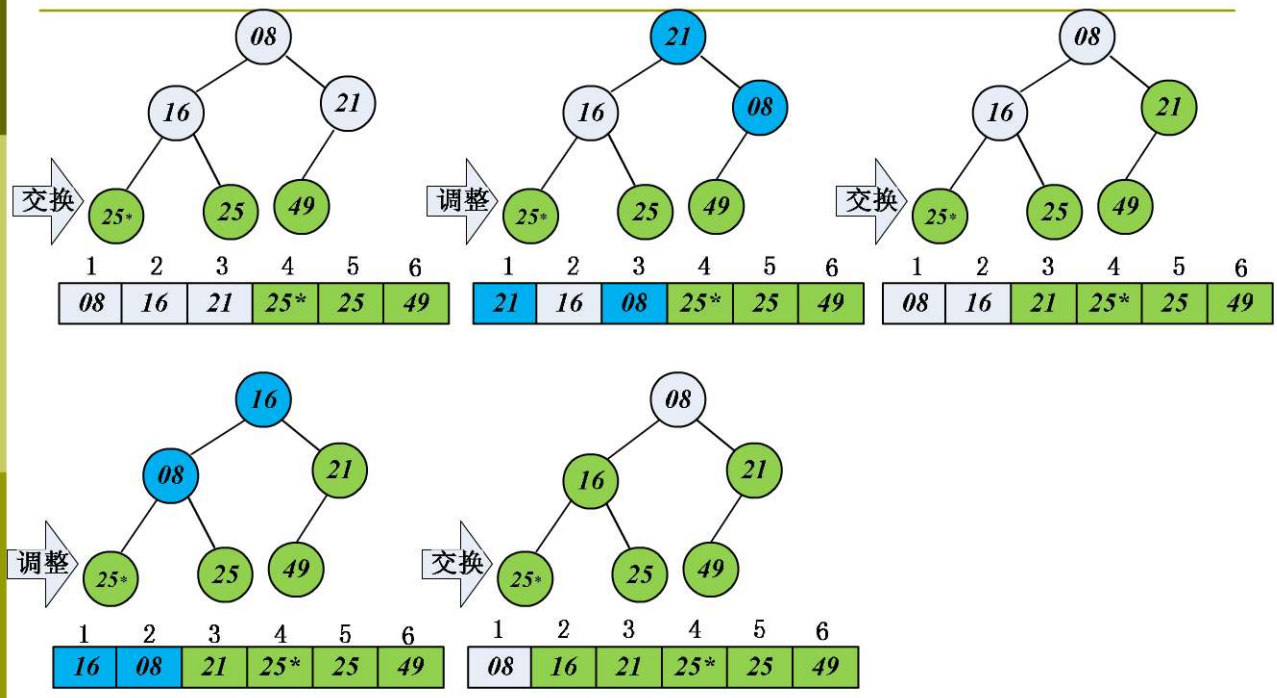

选择排序--堆排序

```
for (j=2*s; j<=m; j*=2) { // 沿key较大的孩子结点向下筛选
    if (j<m && H.r[j].key<H.r[j+1].key) ++j;
                        // j为key较大的记录的下标
    if (rc.key >= H.r[j].key) break; // rc应插入在位置s上
    H.r[s] = H.r[j];  s = j;
}
H.r[s] = rc;  // 插入
} // HeapAdjust
```


选择排序--堆排序


21
25
49
25*
16
08
调整
49
25
21
25*
16
08
交换
08
25
21
25*
16
49
1 2 3 4 5 6
21 25 49 25* 16 08
1 2 3 4 5 6
49 25 21 25* 16 08
1 2 3 4 5 6
08 25 21 25* 16 49
调整
25
25*
21
08
16
49
交换
16
25*
21
08
25
49
调整
25*
16
21
08
25
49
1 2 3 4 5 6
25 25* 21 08 16 49
1 2 3 4 5 6
16 25* 21 08 25 49
1 2 3 4 5 6
25* 16 21 08 25 49

选择排序--堆排序



堆排序过程

选择排序--堆排序

- 基于初始堆进行堆排序
  - 最大堆堆顶H.r[1]具有最大的排序码，将H.r[1]与H.r[n]对调，把具有最大排序码的元素交换到最后，再对前面的*n*-1个元素，使用堆的调整算法HeapAdjust(H,1, *n*-1)，重新建立最大堆，具有次最大排序码的元素又上浮到H.r[1]位置。
  - 再对调H.r[1]和H.r[n-1]，再调用HeapAdjust(H,1,*n*-2) 对前面的*n*-2个元素重新调整，…。
  - 如此反复执行，最后得到全部排序好的元素序列。这个算法即堆排序算法，其细节在下面的程序中给出。

选择排序--堆排序的算法

```
void HeapSort(HeapType &H) {  // 算法10.11
  // 对顺序表H进行堆排序。
   for (i=H.length/2; i>0; --i)  // 把H.r[1..H.length]建成大顶堆
     HeapAdjust ( H, i, H.length );
   for (i=H.length; i>1; --i) {
       temp=H.r[i]; H.r[i]=H.r[1]; H.r[1]=temp;
    // 将堆顶记录和当前未经排序子序列Hr[1..i]中最后一个记录相互交换
       HeapAdjust(H, 1, i-1);  // 将H.r[1..i-1] 重新调整为大顶堆
    }} // HeapSort
```

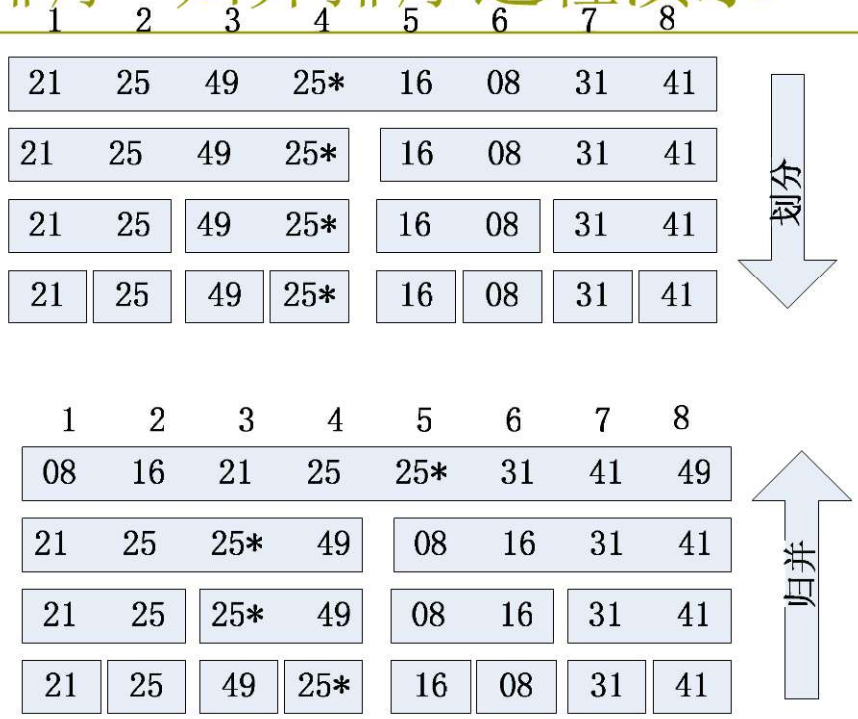

选择排序--堆排序

- 分析：
  - 堆排序的时间复杂性为$O(n\log_2 n)$。
  - 该算法的附加存储主要是在第二个 for 循环中用来执行元素交换时所用的一个临时元素。因此，该算法的空间复杂性为O(1)。
  - 堆排序是一个不稳定的排序方法。

归并排序

- **归并**，是将两个或两个以上的有序表合并成一个新的有序表。
- 元素序列L中有两个有序表L.r[left..mid]和L.r[mid+1..right]。它们可归并成一个有序表, 存于L.r[left..right] 中。
- 这种方法称为**两路归并** (2-way merging)。

归并排序--归并排序过程演示



归并排序过程

归并排序--两路归并算法

```
void Merge (RedType SR[], RedType TR[], int i, int m, int n) {
  // 算法10.12. 将有序的SR[i..m]和SR[m+1..n]归并为有序的TR[i..n]
  int j,k;
  for (j=m+1, k=i;  i<=m && j<=n;  ++k) {
    // 将SR中记录由小到大地并入TR
    if LQ(SR[i].key,SR[j].key) TR[k] = SR[i++];
    else TR[k] = SR[j++];
  }
  if (i<=m)  // TR[k..n] = SR[i..m];  将剩余的SR[i..m]复制到TR
    while (k<=n && i<=m) TR[k++]=SR[i++];
  if (j<=n)  // 将剩余的SR[j..n]复制到TR
    while (k<=n &&j <=n) TR[k++]=SR[j++];
} // Merge
```


归并排序--两路归并算法

```
void MSort(RedType SR[], RedType TR1[], int s, int t) {
// 算法10.13. 将SR[s..t]归并排序为TR1[s..t]。
  if (s==t) TR1[t] = SR[s];
  else {
    m=(s+t)/2;          // 将SR[s..t]平分为SR[s..m]和SR[m+1..t]
    MSort(SR,TR2,s,m);   // 递归地将SR[s..m]归并为有序的TR2[s..m]
    MSort(SR,TR2,m+1,t); // 将SR[m+1..t]归并为有序的TR2[m+1..t]
    Merge(TR2,TR1,s,m,t); // 将TR2[s..m]和TR2[m+1..t]归并到TR1[s..t]
  }
} // MSort
```


归并排序--(两路)归并排序的主算法

- ☑ 分析

1. 算法总的时间复杂度为$O(n\log_2 n)$。
2. 归并排序占用附加存储较多, 需要另外一个与原待排序元素数组同样大小的辅助数组。这是这个算法的缺点。
3. 归并排序是一个稳定的排序方法。

基数排序

- 基数排序是采用“分配”与“收集”的办法，用对多排序码进行排序的思想实现对单排序码进行排序的方法。
- 多排序码排序

  以扑克牌排序为例。每张扑克牌有两个“排序码”：花色和面值。其有序关系为：

花色：♣ < ♦ < ♥ < ♠

面值：2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < J < Q < K < A

如果我们把所有扑克牌排成以下次序：

♣ 2, …, ♣ A, ♦ 2, …, ♦ A, ♥ 2, …, ♥ A, ♠ 2, …, ♠ A

基数排序 --多排序码排序

这就是**多排序码排序**。排序后形成的有序序列叫做**词典有序序列**。

对于上例两排序码的排序，可以先按花色排序，之后再按面值排序；也可以先按面值排序，再按花色排序。

一般情况下，假定有一个 n 个元素的序列 $\{V_0, V_1, \ldots, V_{n-1}\}$，且每个元素 V_i 中含有 d 个排序码

$$\left(K_i^1, K_i^2, \cdots, K_i^d\right)$$

如果对于序列中任意两个元素 V_i 和 V_j $(0 \leqslant i < j \leq n-1)$ 都满足：$\left(K_i^1, K_i^2, \cdots, K_i^d\right) < \left(K_j^1, K_j^2, \cdots, K_j^d\right)$

则称序列对排序码 $(K^1, K^2, \ldots, K^d)$ **有序**。其中，K^1 称为**最高位排序码**，K^d 称为**最低位排序码**。

基数排序 --多排序码排序

如果排序码是由多个数据项组成的数据项组，则依据它进行排序时就需要利用多排序码排序。

- 实现多排序码排序有两种常用的方法：
  - 最高位优先MSD(Most Significant Digit first)
  - 最低位优先LSD(Least Significant Digit first)

基数排序 --多排序码排序

- ☑ **最高位优先法**通常是一个递归的过程：

1. 先根据最高位排序码 K^1排序, 得到若干元素组, 元素组中各元素都有相同排序码K^1。
2. 再分别对每组中元素根据排序码 K^2 进行排序, 按 K^2 值的不同, 再分成若干个更小的子组, 每个子组中的元素具有相同的 K^1和 K^2值。

   依此重复, 直到对排序码K^d完成排序为止。
3. 最后, 把所有子组中的元素依次连接起来， 就得到一个有序的元素序列。

基数排序 --多排序码排序

- 最低位优先法

  首先依据最低位排序码K^d对所有元素进行一趟排序，再依据次低位排序码K^{d-1}对上一趟排序的结果再排序，依次重复，直到依据排序码K^1最后一趟排序完成，就可以得到一个有序的序列。使用这种排序方法对每一个排序码进行排序时，不需要再分组，而是整个元素组都参加排序。

- LSD和MSD方法也可应用于对一个排序码进行的排序。此时可将单排序码 K^i 看作是一个子排序码组：

$$\left(K_i^1,\ K_i^2,\ \cdots,\ K_i^d\right)$$

基数排序 --链式基数排序

- 链式基数排序是典型的LSD排序方法, 利用“分配”和“收集”对单排序码进行排序。在这种方法中，把单排序码 K^1 看成是一个d元组：

$$\left(K_i^1,\ K_i^2,\ \cdots,\ K_i^d\right)$$

其中的每一个分量$K_i^j(1 \leqslant j \leqslant d)$ 也可看成是一个排序码。分量K_i^j有$radix$种取值, 称$radix$为基数。例如，排序码984可以看成是一个3元组(9, 8, 4), 每一位有 0, 1, …, 9 等10种取值，基数$radix = 10$。排序码‘data’可以看成是一个4元组(d,a,t,a), 每一位有‘a’,‘b’, …, ‘z’等26种取值，$radix = 26$。

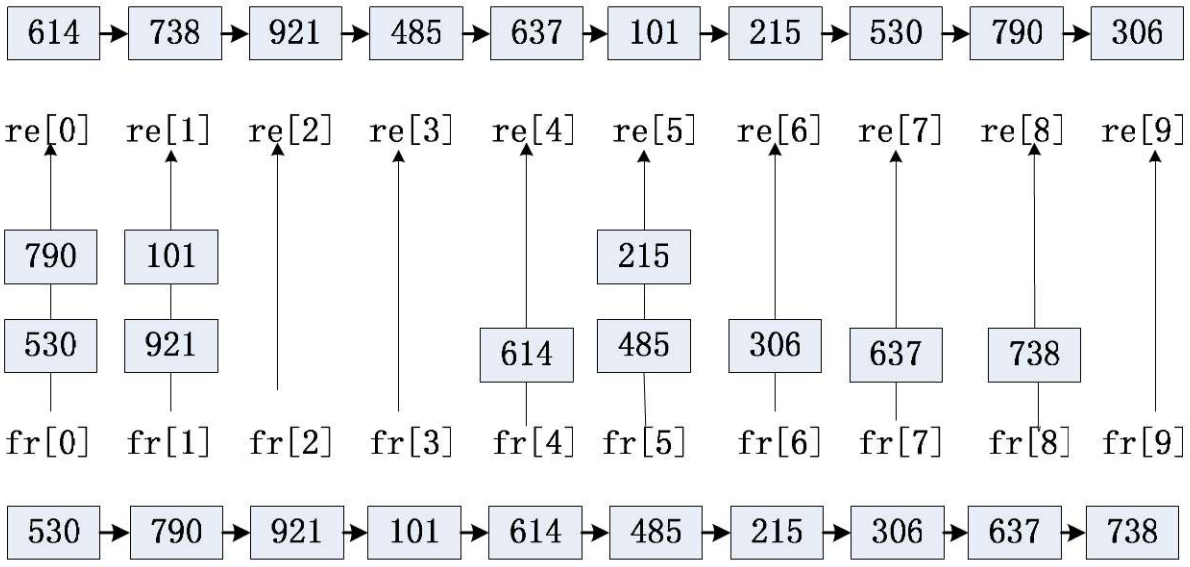
基数排序 --链式基数排序

- 针对d元组中的每一位分量, 把元素序列中的所有元素, 按K_i^j 的取值，先“分配”到rd个队列中去。然后再按各队列的顺序，依次把元素从队列中“收集”起来，这样所有元素按取值K_i^j排序完成。
- 如果对于所有元素的排序码$K^0, K^1, \ldots, K^{n-1}$, 依次对各位的分量, 让$j = d, d\text{-}1, \ldots, 1$, 分别用“分配”、“收集”的运算逐趟进行排序，在最后一趟“分配”、“收集” 完成后, 所有元素就按其排序码的值从小到大排好序了。

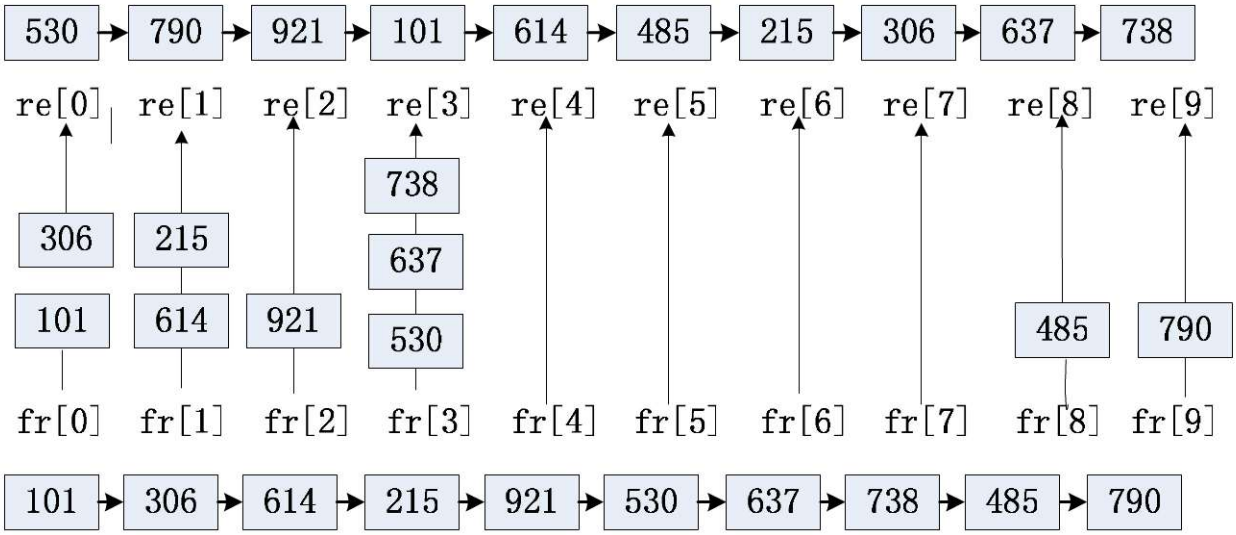
基数排序 --链式基数排序

- 各队列采用链式队列结构, 分配到同一队列的排序码用链接指针链接起来。每一队列设置两个队列指针：int front [radix]指示队头，int rear [radix] 指向队尾。
- 为了有效地存储和重排 n 个待排序元素，以静态链表作为它们的存储结构。

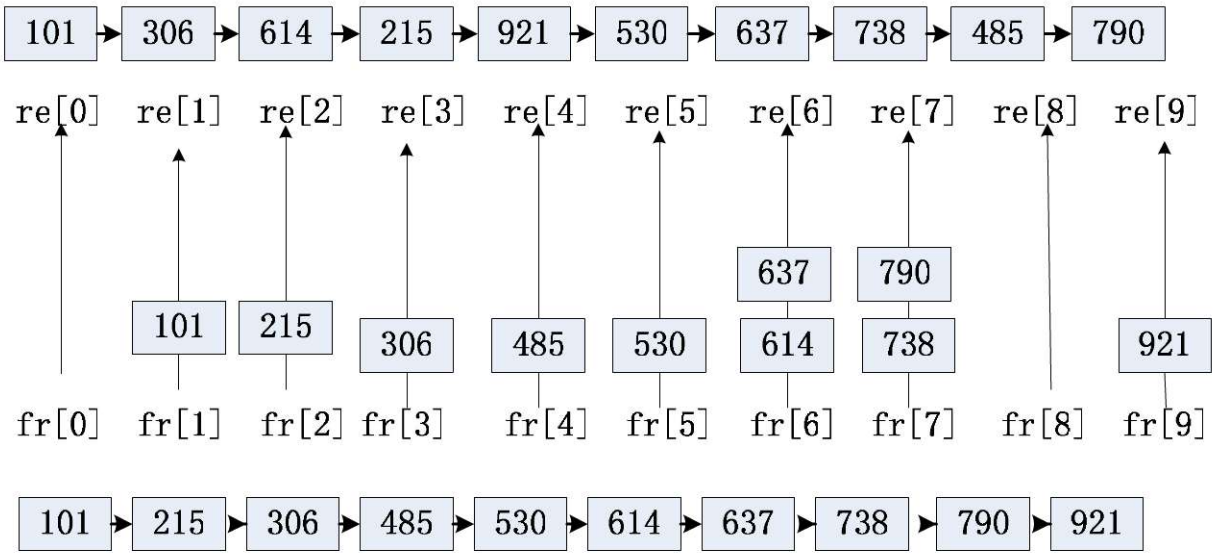
基数排序 --链式基数排序过程演示



基数排序 --链式基数排序过程演示



基数排序 --链式基数排序过程演示



链式基数排序过程

基数排序 --链式基数排序算法

- 分析：
  1. 若每个排序码有d位, 需要重复执行d趟“分配”与“收集”。每趟对n个元素进行“分配”，对radix个队列进行“收集”。总时间复杂度为O(d(n+radix))
  2. 若基数radix相同, 对于元素个数较多而排序码位数较少的情况, 使用链式基数排序较好。
  3. 基数排序需要增加n+2radix个附加链接指针。
  4. 基数排序是稳定的排序方法。

基数排序 --各种内部排序方法的比较

排序方法	比较次数		移动次数		稳定性	附加存储	
	最好	最差	最好	最差		最好	最差
直接插入排序	n	n^2	0	n^2	√	1	
折半插入排序	$n \log_2 n$		0	n^2	√	1	
起泡排序	n	n^2	0	n^2	√	1	
快速排序	$n\log_2 n$	n^2	$\log_2 n$	n	×	$\log_2 n$	n
简单选择排序	n^2		0	n	×	1	
锦标赛排序	$n \log_2 n$		$n \log_2 n$		√	n	
堆排序	$n \log_2 n$		$n \log_2 n$		×	1	
归并排序	$n \log_2 n$		$n \log_2 n$		√	n