

# Order

- 定义
    - 内部排序：在内存中排序
    - 外部排序：待排数据放在外部存储器，需要内外存交换数据
  - 排序效率 和待排数据的数据结构 密切相关
    - 集合
    - 数型结构
  - 数据表：有限集合
  - 稳定性：排序不改变“关键码相同”的对象的顺序
  - 排序的时间、空间开销：时间复杂度、空间复杂度
- 
- 插入排序
  - 交换排序
  - 选择排序
  - 归并排序
  - 基数排序

## 10.1 插入排序

- 基本方法：每一步，将待排对象插入到一组已排好序的对象的相应位置

### 直接插入排序

- 按顺序逐一进行比较，得到插入位置。
- 分析：
  - 若共有 $n$ 个，则进行 $n-1$ 次插入
  - 最好的情况下：每趟只需与前面的有序对象序列的最后一个对象比较1次，移动2次对象。插入 $n$ 个元素，比较 $n-1$ 次，总移动次数 $2(n-1)$
  - 最坏的情况下：第 $i$ 趟插入时，第 $i$ 个对象和前面 $i-1$ 个对象进行比较，且每次比较需要移动1次数据，则总的次数为： $n^2/2$

**KCN**和**对象移动次数RMN**分别为

$$\begin{aligned} \circ \quad KCN &= \sum_{i=1}^{n-1} i = n(n-1)/2 \approx n^2/2, \\ \circ \quad RMN &= \sum_{i=1}^{n-1} (i+2) = (n+4)(n-1)/2 \approx n^2/2 \end{aligned}$$

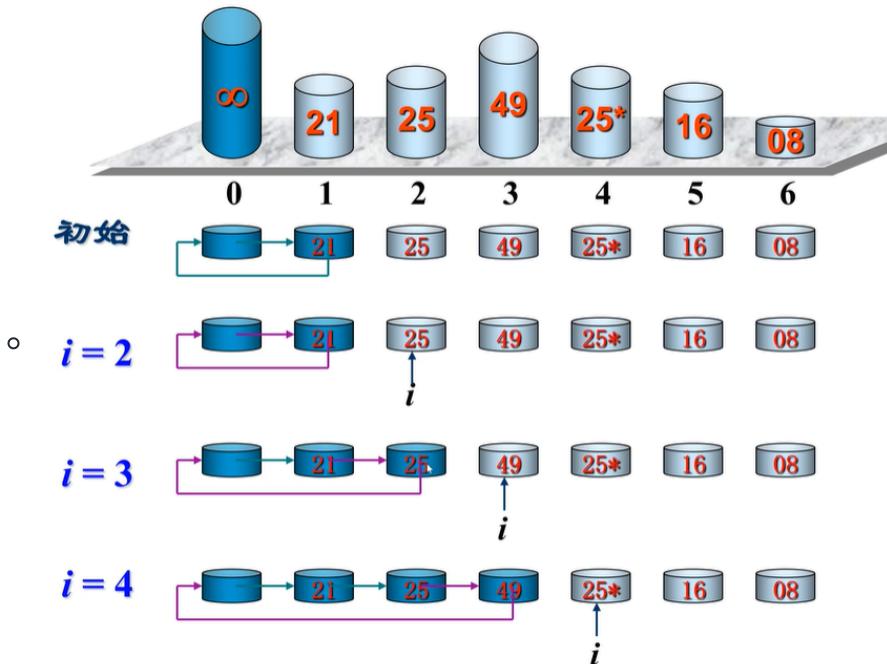
- 直接插入排序的时间复杂度： $O(n^2)$ ，考虑一般情况
- 稳定排序

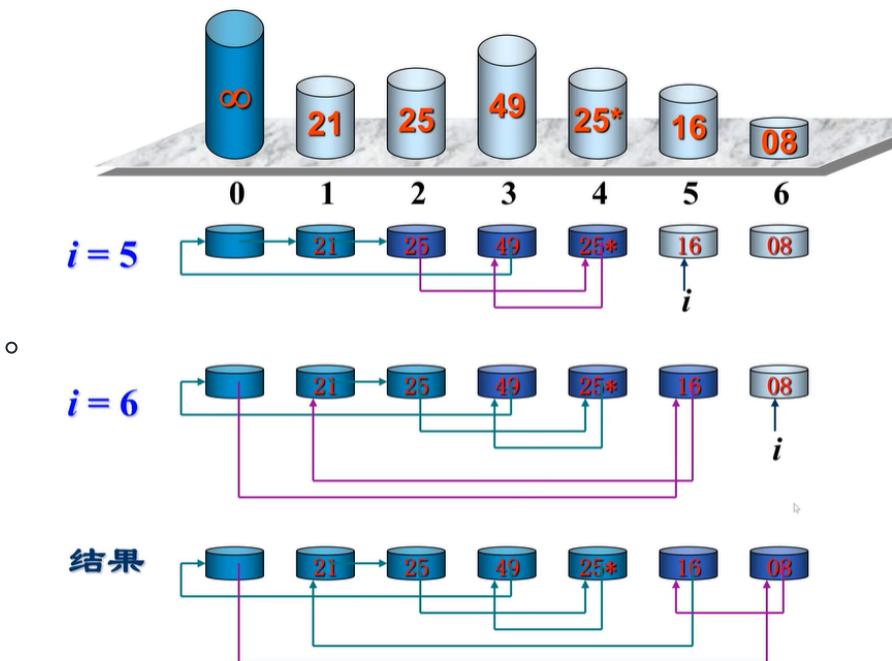
## 折半插入排序

- 对直接插入排序进行改进
- 通过排序过程中部分元素有序，提高排序效率
- 分析：
  - 查找快
  - 比较次数： $n * \log_2(n)$  次
  - 也是稳定的

## 表插入排序

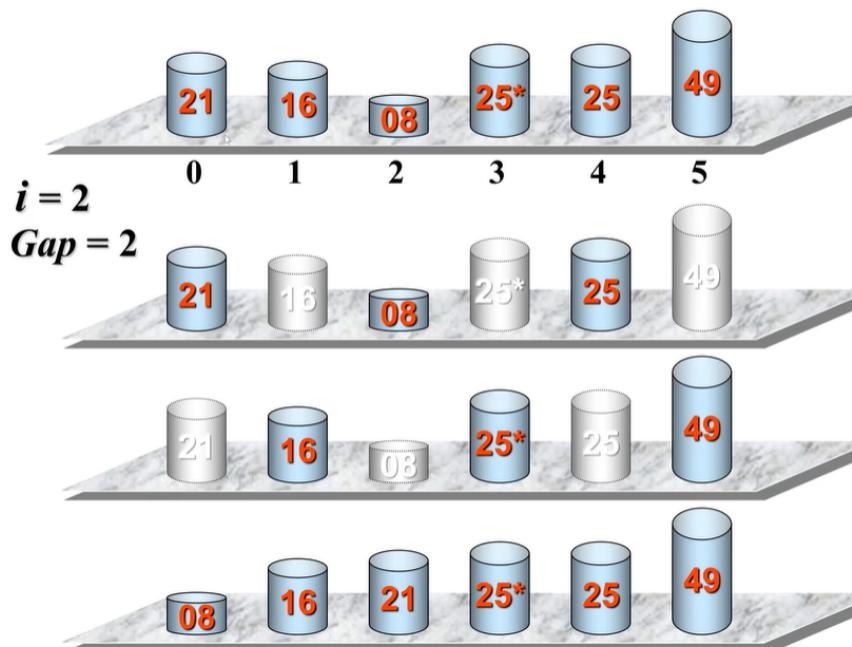
- 目标：
  - 减少插入排序中，移动元素的次数
  - 尽量不移动元素
- 方法：链式存储（静态链表）
- 和表头形成循环链表





## 希尔排序

- 基本思想：分组排序，每组分别进行直接插入排序
- 对于数据量大时挺好用
- 基本过程：
  - $n$ 个对象，先取一个整数 $gap < n$ 为间隔，
  - 所有距离为 $gap$ 的对象在同一个子序列中，每个子序列中分别进行直接排序
  - 缩小 $gap$ ，重复，直到 $gap$ 取1
    - $gap$ 为“脚标”之差



- 时间复杂度：平均为  $n^{1.25} \sim 1.6 * n^{1.25}$

## 10.2 交换排序

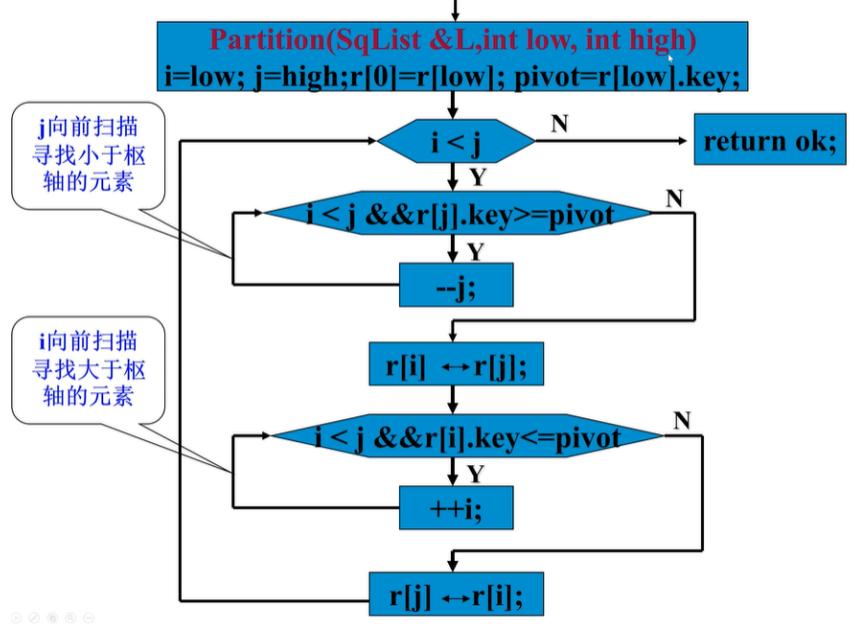
### 冒泡排序

- 分析：
  - 最好的情况：n-1次比较，不移动
  - 最坏的情况：比较次数 $1/2 * n(n-1)$ ，移动次数： $3/2 * n(n-1)$
- 稳定的排序
- $O(n^2)$ 的时间复杂度

### 快速排序

- 基本思想：
  - 在冒泡排序上进行大范围交换
  - 采用：树形结构，借鉴二叉树的思想
- 基本过程：
  - 任取一个对象，作为基准
    - 左侧序列都小于或等于基准
    - 右侧都大于基准
  - 对两个子序列重复上述方法
  - 类似于二叉排序树，在一维数组层次上进行存储或者处理
  - 设定i, j指针先后扫描，当*i==j*时扫描结束；扫描期间，大者向后，小者向前

- 一趟快速排序算法流程图（算法10.6a）**



o

```

void QuickSort ( datalist &list, int low, int high ) {
    //对表list中list[ low~high] 进行快速排序
    if ( low < high ) {
        pivot = Partition ( list, low, high ); //一分为二
        QuickSort ( list, low, pivot-1 );
            //在左子区间进行快速排序
        QuickSort ( list, pivot+1, high );
            //在右子区间进行快速排序
    }
}

```

注：调整QuickSort先后次序可以有助于减少递归深度！

## 快速排序分析

- 快排是递归的，需要使用栈
- 可以证明：quicksort的平均计算时间： $O(n * \log_2(n))$
- 最大递归调用层于递归树的深度一样，存储开销： $O(\log_2(n))$
- 每次划分确定对象定位后，左侧序列和右侧序列长度相同，则下一步将是对两个长度减半的子序列进行排序，理想，趟数最少
- 最坏的情况：比较次数： $n^2 / 2$ 
  - 例如 **从小到大** 的已经排好的序列
  - 递归树成为单支树
  - 改进：选基准元素：中间的而不是两端的
- 快排是 **不稳定** 的排序方法
- 小结：
  - $T(n) = O(n * \log_2(n))$
  - $S(n) = O(\log_2(n))$
  - 不稳定

## 10.3 选择排序

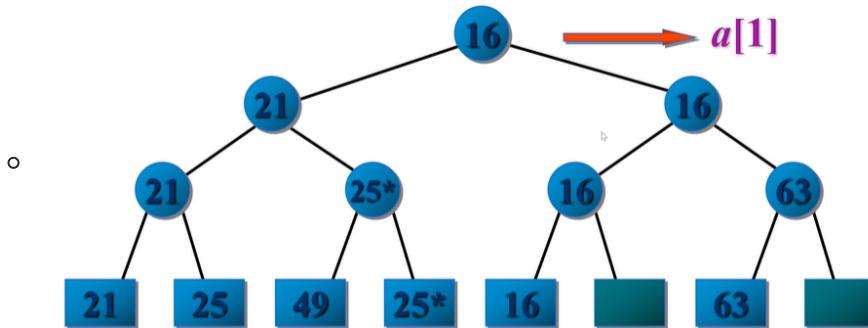
- 基本思想：每一趟在 $n-i$ 个待排序对象中，选出最小/最大的对象

### 直接选择排序

- 算法分析：
  - 重复扫描，时间复杂度： $O(n^2)$

# 锦标赛排序

- 基本思想：
  - 消除直接排序中的重复扫描
  - 采用树形结构进行选择排序
- 具体过程：
  - 进行两两对比，直到决出“冠军”，比较次数为  $n-1$
  - Winner(胜者)



## 锦标赛排序分析

- 满完全二叉树，深度为 上取整( $\log_2(n)$ ) + 1
- 时间复杂度： $O(n * \log_2(n))$
- 需要附加空间： $n-1$ 个结点
- 稳定
- - 最小关键字排序比较次数：满完全二叉树中间节点个数(n-1)
  - 其他关键字排序比较次数：满完全二叉树深度减1，即比较次数为： $\lceil \log_2 n \rceil$
- 小结：
  - $T(n) = O(n * \log_2(n))$
  - $S(n) = O(n)$
  - 稳定

# 堆排序

- 基本目标：
  - 实现  $O(n * \log_2(n))$  的时间复杂度
  - 减少/消除辅助空间
- 基本思想：
  - 二叉树中所有节点均存放在待排元素，而不是只将待排元素放在叶子节点上

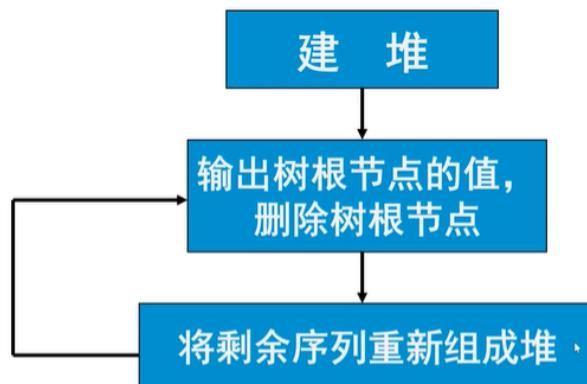
## 1. 堆的概念

设有n个元素的序列  $k_1, k_2, \dots, k_n$ , 当且仅当满足下述关系之一时, 称之为堆。

$$\left\{ \begin{array}{l} k_i \leq k_{2i} \\ k_i \leq k_{2i+1} \end{array} \right. \quad \text{或者} \quad \left\{ \begin{array}{l} k_i \geq k_{2i} \\ k_i \geq k_{2i+1} \end{array} \right. \quad i=1,2,\dots,n/2$$

- 本质: 在逻辑上看成一棵完全二叉树, 所有节点的值均大于(小于)其左右节点的值, 于是, 根节点就是“冠军”
- 大根堆 / 小根堆
- 如何建这个堆呢?

## 2. 流程



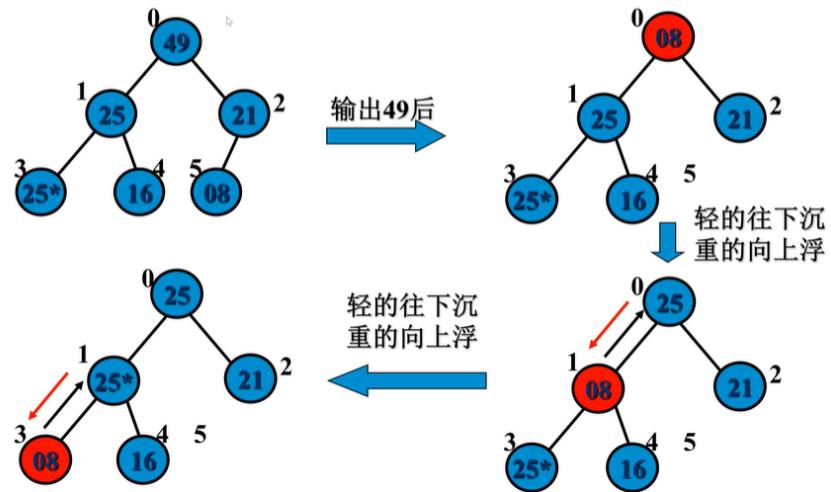
因此, 实现堆排序需要解决两个问题:

- 1) 如何将一个任意序列建成堆;
- 2) 输出顶点后, 如何将剩余序列重新调整为堆。

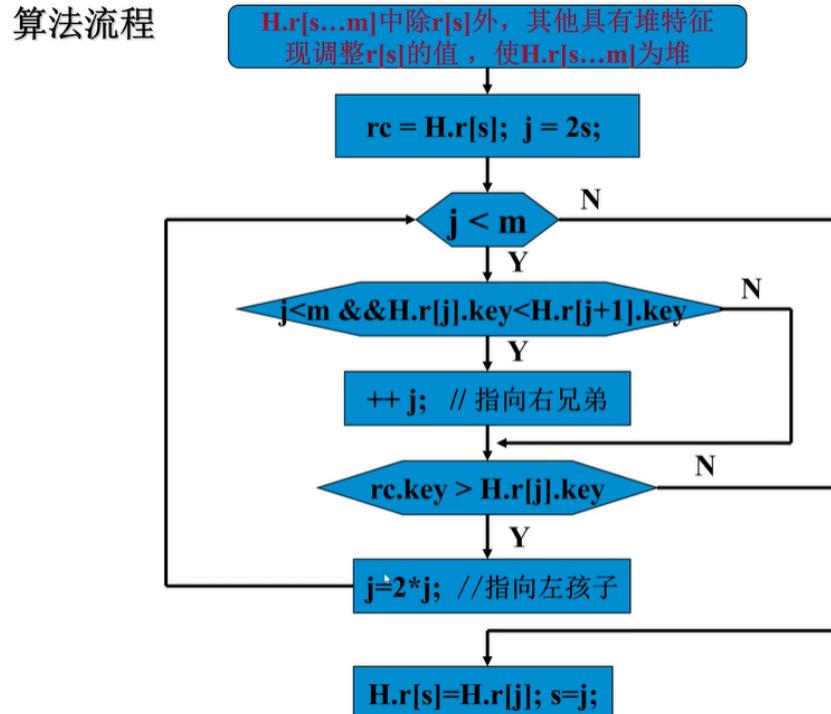
## 3. 堆的建立

- 先思考输出顶点后如何调整:
- 轻的下沉, 重的上浮





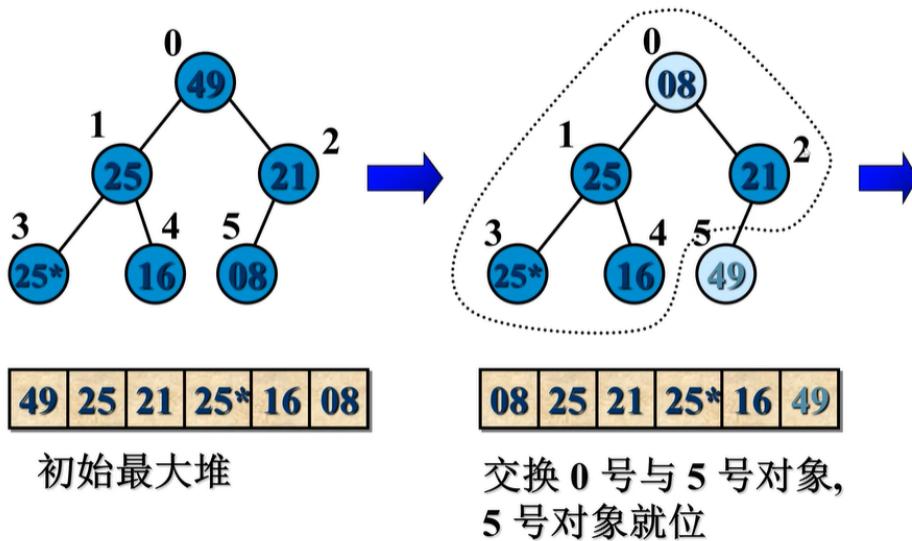
◦ 算法流程

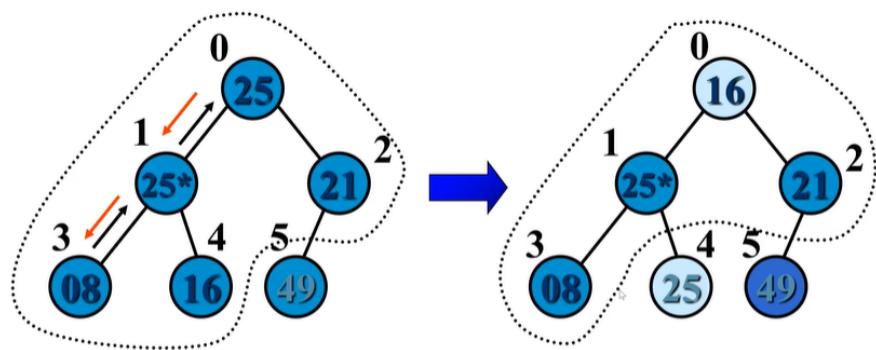


◦ 堆的建立：

◦ 从最后一个非终端节点开始，往前逐步调整，直到根节点为止，序列变为堆。

#### 4. 基于初始堆的堆排序



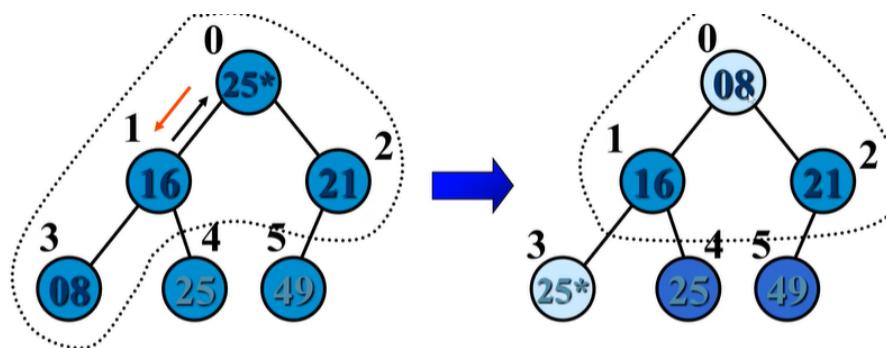


25	25*	21	08	16	49
----	-----	----	----	----	----

从 0 号到 4 号 重新  
调整为最大堆

16	25*	21	08	25	49
----	-----	----	----	----	----

交换 0 号与 4 号对象，  
4 号对象就位

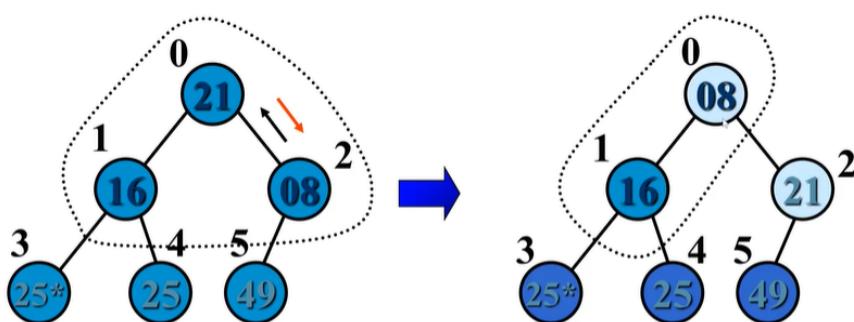


25*	16	21	08	25	49
-----	----	----	----	----	----

从 0 号到 3 号 重新  
调整为最大堆

08	16	21	25*	25	49
----	----	----	-----	----	----

交换 0 号与 3 号对象，  
3 号对象就位



21	16	08	25*	25	49
----	----	----	-----	----	----

从 0 号到 2 号 重新  
调整为最大堆

08	16	21	25*	25	49
----	----	----	-----	----	----

交换 0 号与 2 号对象，  
2 号对象就位

- 堆排序算法

```

void HeapSort (HeapType &H) {
    //对顺序表H进行堆排序
    for ( i = H.length / 2; i > 0; -- i )
            HeapAdjust(H,I, H.length );       //初始堆
    for ( i = H.length; i > 1; --i) {
            H.r[1] ← H.r[i];                    //交换
            HeapAdjust( H, 1,i-1 );            //重建最大堆
    }
}

```

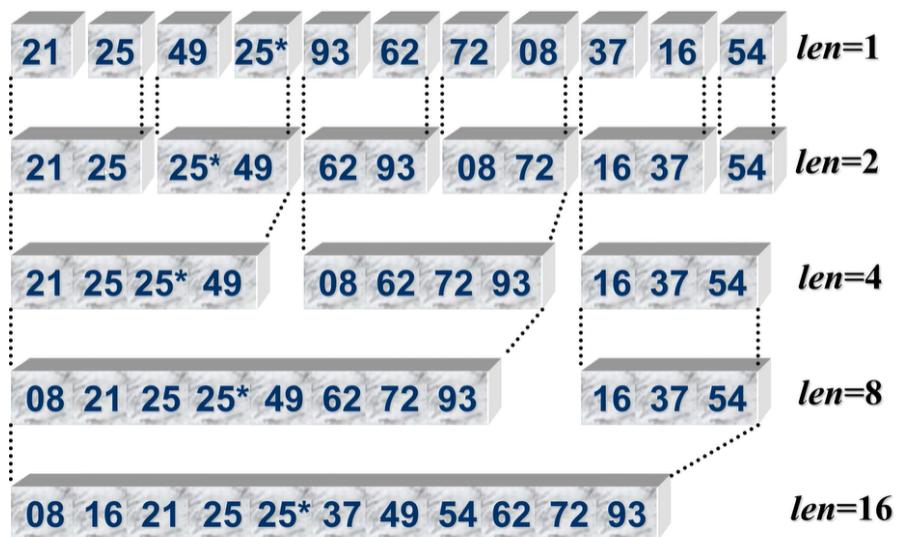
## 堆排序分析

- 调用  $n-1$  次 `HeapAdjust()`, 时间复杂度  $O(n * \log_2(n))$
- 附加存储, 在for里面的临时空间, 空间复杂度为  $O(1)$
- 堆排序, 不稳定
- 小结:
  - $T(n) = O(n * \log_2(n))$
  - $S(n) = O(1)$
  - 不稳定

## 10.5 归并排序

- 基本思路: 就是MergeSort

归并排序算法示意图



## 归并排序分析

- $T(n) = O(n * \log_2(n))$
- 归并时用栈
- 稳定
- 小结：
  - $T(n) = O(n * \log_2(n))$
  - $S(n) = O(n)$
  - 稳定

## 10.6 基数排序

- 基本思想：关键字不同的“位值”进行排序
  - 例如扑克牌：先用“花色”排序，再用“面值”排序
- 假定n个对象序列：{  $V_0, V_1, \dots, V_{n-1}$  }
- 进队列扫描，出队列即可排序
- 算法分析：
  - 假设有n个记录，每个记录的关键字有d位，每个关键字的取值有rd个，则需要rd个队列，进行d趟“分配”与“收集”。总的时间复杂度为 $O(d(n+rd))$ 。
  - 若关键字的rd相同，对于对象个数较多而关键码位数较少的情况，使用链式基数排序较好。
  - 基数排序是稳定的排序方法。

## 排序方式的比较

## 各种排序方法的比较

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	$n \log_2 n$	$n^2$	$n \log_2 n$	$n^2$	✗	$\log_2 n$	$n^2$
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	$n \log_2 n$		$n \log_2 n$		√	$n$	