第10章 排序

Sort

主讲：顾为兵

第十章 排序(Sort)

目录

§10.1 排序概述
§10.2 插入排序
§10.3 交换排序
§10.4 选择排序
§10.5 归并排序
§10.6 基数排序

§10.1 排序概述

操作对象：同类型数据元素的集合。

操作目标：将无序的数据元素序列排列成按关键字值有序的序列。

稳定排序与非稳定排序：

设：R_i.key==R_j.key

假设排序前Ri的位置排列在R_j之前；

经排序后若R_i的位置仍然排列在R_j之前，则是稳定排序算法；

若不能保证这一点则是非稳定排序算法。

内部排序与外部排序：

内部排序----待排序的记录全部存放在内存；

外部排序----一部分放在内存，一部分在外存。排序过程中存在着内、外存的数据交换。

排序的基本动作：

①比较 ②移动

排序性能的评价：

对比较次数和移动次数的评估

先进排序算法：

时间复杂度----O(nlogn)

一般排序算法

时间复杂度----$O(n^2)$

排序方法：

插入排序

直接插入排序、折半插入排序、2路插入排序

表插入排序、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、树形选择排序

归并排序

2路归并排序

基数排序

第十章 排序(Sort)

目录

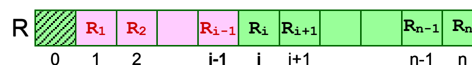
§10.1 排序概述
§10.2 插入排序
§10.3 交换排序
§10.4 选择排序
§10.5 归并排序
§10.6 基数排序

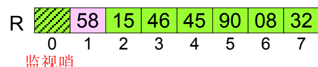
§10.2 插入排序

10.2.1 直接插入排序
10.2.2 折半插入排序
10.2.3 二路插入排序
10.2.4 表插入排序
10.2.5 希尔排序

直接插入排序算法思想：

- 排序区间R[1..n]；
- 在排序的过程中，整个排序区间被分为两个子区间：有序区R[1..i-1]和无序区R[i..n]；
- 共进行n－1趟排序，每趟排序都是把无序区的第一条记录R_i插到有序区的合适位置上。

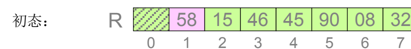




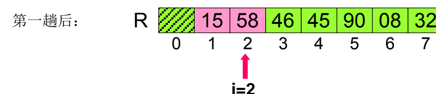
监视哨

初态：有序区为R[1]

初态：



第一趟后：

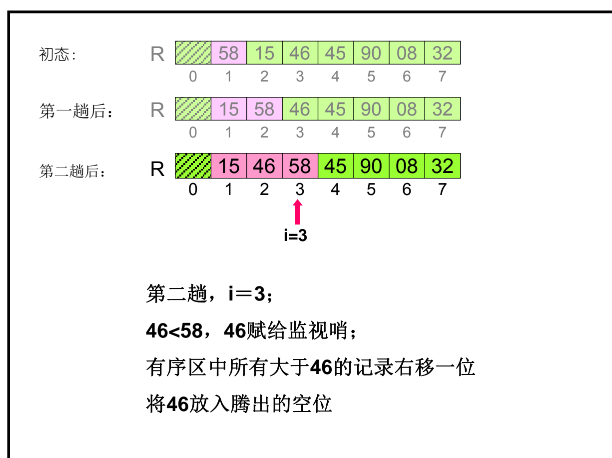


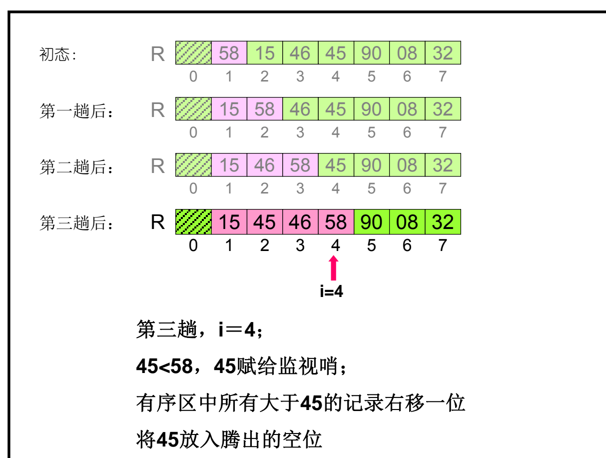
第一趟，i=2；

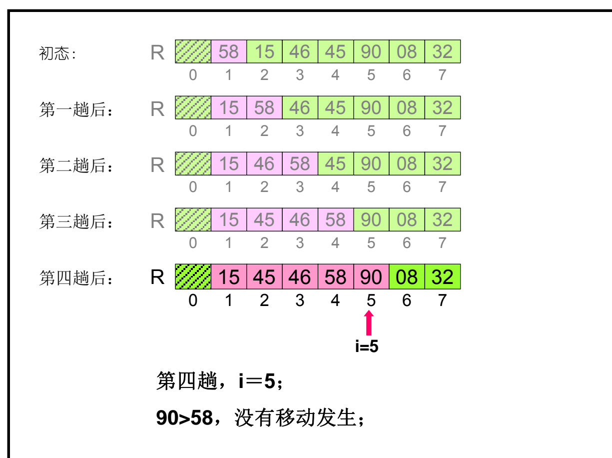
15<58，15赋给监视哨；

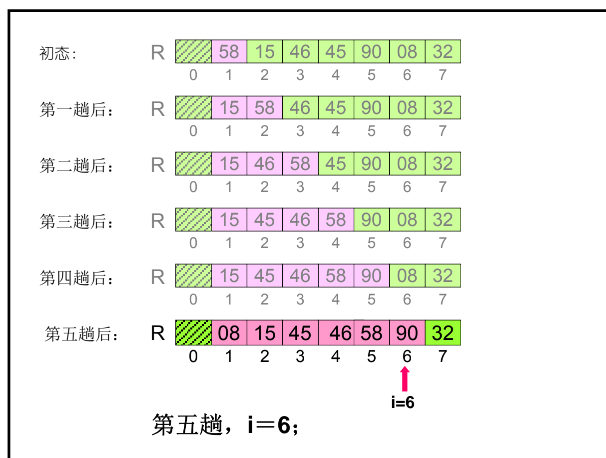
有序区中所有大于15的记录右移一位

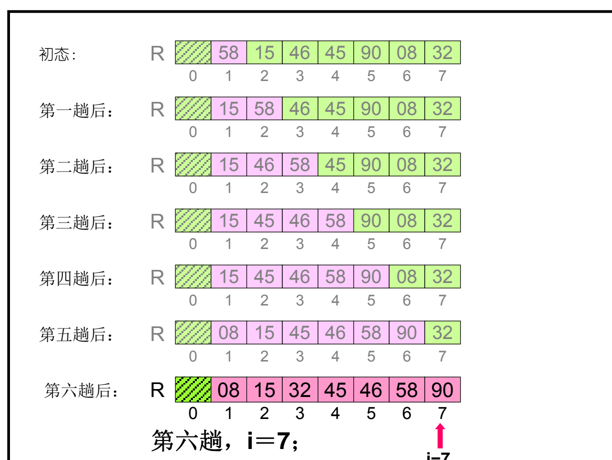
将15放入腾出的空位

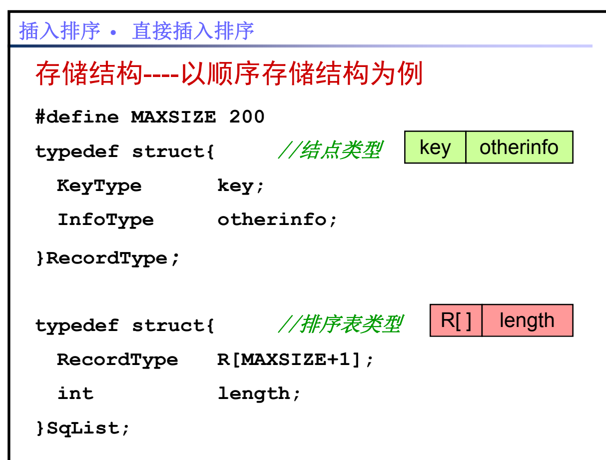

初态：
R 58 15 46 45 90 08 32
0 1 2 3 4 5 6 7
第一趟后：
R 15 58 46 45 90 08 32
0 1 2 3 4 5 6 7
第二趟后：
R 15 46 58 45 90 08 32
0 1 2 3 4 5 6 7
i=3
第二趟，i=3；
46<58，46赋给监视哨；
有序区中所有大于46的记录右移一位
将46放入腾出的空位


初态：
R 58 15 46 45 90 08 32
第一趟后：
R 15 58 46 45 90 08 32
第二趟后：
R 15 46 58 45 90 08 32
第三趟后：
R 15 45 46 58 90 08 32
0 1 2 3 4 5 6 7
i=4
第三趟，i=4；
45<58，45赋给监视哨；
有序区中所有大于45的记录右移一位
将45放入腾出的空位


初态：
R 58 15 46 45 90 08 32
第一趟后：
R 15 58 46 45 90 08 32
第二趟后：
R 15 46 58 45 90 08 32
第三趟后：
R 15 45 46 58 90 08 32
第四趟后：
R 15 45 46 58 90 08 32
0 1 2 3 4 5 6 7
i=5
第四趟，i=5；
90>58，没有移动发生；


初态：
R 58 15 46 45 90 08 32
第一趟后：
R 15 58 46 45 90 08 32
第二趟后：
R 15 46 58 45 90 08 32
第三趟后：
R 15 45 46 58 90 08 32
第四趟后：
R 15 45 46 58 90 08 32
第五趟后：
R 08 15 45 46 58 90 32
0 1 2 3 4 5 6 7
i=6
第五趟，i=6；


初态：
R 58 15 46 45 90 08 32
第一趟后：
R 15 58 46 45 90 08 32
第二趟后：
R 15 46 58 45 90 08 32
第三趟后：
R 15 45 46 58 90 08 32
第四趟后：
R 15 45 46 58 90 08 32
第五趟后：
R 08 15 45 46 58 90 32
第六趟后：
R 08 15 32 45 46 58 90
0 1 2 3 4 5 6 7
第六趟，i=7；
i=7


插入排序 • 直接插入排序
存储结构----以顺序存储结构为例
#define MAXSIZE 200
typedef struct{ //结点类型
KeyType key;
InfoType otherinfo;
}RecordType;
key
otherinfo
typedef struct{ //排序表类型
RecordType R[MAXSIZE+1];
int length;
}SqList;
R[ ]
length

直接插入排序算法：

R[0]有两个作用：
(1)保留R[i]的副本
(2)监视哨，监视j是否越界

```
void InsertSort( SqList &L){
    //对顺序表L作直接插入排序
   for(i=2; i<=L.length; i++)
   if(L.R[i].key<L.R[i-1].key){
       L.R[0].key=L.R[i].key;    //将R[i]赋给哨兵
       L.R[i]=L.R[i-1];
       for(j=i-2; L.R[j].key>L.R[0].key; j--)
       L.R[j+1]=L.R[j];  //将大于哨兵的记录右移
       L.R[j+1]=L.R[0];  //将R[i]放入有序区的正确位置
   } //end if
}
```


直接插入排序性能分析：

最好的情况：表的初态恰好是正序排列

比较次数：$C\min = \sum_{i=2}^{n} 1 = n-1$

移动次数：$M\min = 0$

最坏的情况：表的初态恰好是逆序排列

比较次数：$C\max = \sum_{i=2}^{n} i = \frac{(n+2)(n-1)}{2}$

移动次数：$M\max = \sum_{i=2}^{n}(i+1) = \frac{(n+4)(n-1)}{2}$

等概条件下平均情况：

平均比较次数：$\overline{C} = \frac{C\max + C\min}{2}$

平均移动次数：$\overline{M} = \frac{M\max + M\min}{2}$

时间复杂度：$O(n^2)$

直接插入排序是一种稳定的排序方法

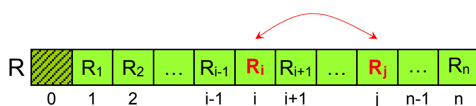
第十章 排序(Sort)

目录

§10.1 排序概述
§10.2 插入排序
§10.3 交换排序
§10.4 选择排序
§10.5 归并排序
§10.6 基数排序

交换排序概述

- 交换排序的基本思路是：
  比较两条记录Ri与Rj(i<j)的关键字，
  如果Ri.key>Rj.key----逆序排列，则交换Ri与Rj的位置
- 根据Ri与Rj的取法不同，分为冒泡排序和快速排序



§10.3 交换排序

10.3.1 冒泡排序

10.3.2 快速排序

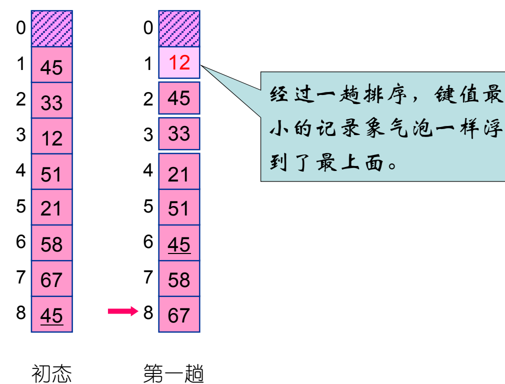
10.3.1 冒泡排序 (Bubble Sort)

自下而上(或上而下)扫描记录序列，

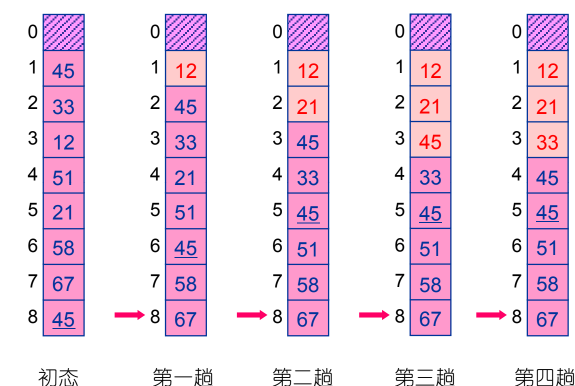
相邻的两条记录R_i与R_{i-1}(或R_{i+1})如果是逆序，

则交换位置

交换排序 • 冒泡排序



交换排序 • 冒泡排序



交换排序 • 冒泡排序

注意：在第四趟扫描过程中没有移动发生，表明所有记录已经排序完毕，算法可以提前结束了。

交换排序 • 冒泡排序

冒泡排序算法：

```
void BubbleSort( SqList  &L ) {
   //对顺序表L作冒泡排序
    for(i=2; i<=L.length; i++) {    //进行n－1趟扫描
        move=False ;                //move是交换与否的标志
        for( j=n;  j<=i;  j- - )
            if( L.R[ j ] < L.R[ j-1] ) {
                L.R[ j ]↔L.R[ j-1]; move=True; } //如逆序则交换
        if ( !move ) return;        //如果没有移动发生，则结束
    }
}
```

交换排序 • 冒泡排序

冒泡排序性能分析：

最好的情况： 表的初态恰好是正序排列，第一趟扫描没有移动发生

比较次数： Cmin = n-1

移动次数： Mmin = 0

最坏的情况： 表的初态恰好是逆序排列，需要进行n-1趟排序，每趟都要移动整个区间

比较次数： $C\max = \sum_{i=2}^{n}(n-i+1) = \frac{n(n-1)}{2}$

移动次数： $M\max = \sum_{i=2}^{n}(n-i+1)\times 3 = \frac{3n(n-1)}{2}$

等概条件下平均情况：

平均比较次数： $\overline{C} = \dfrac{C\max + C\min}{2}$

平均移动次数： $\overline{M} = \dfrac{M\max + M\min}{2}$

时间复杂度： $O(n^2)$

冒泡排序是一种**稳定的**排序方法

§10.3 交换排序

10.3.1 冒泡排序

10.3.2 快速排序

10.3.2 快速排序(Quick Sort)

一种速度很快的排序算法

快速排序算法思想:

- 设排序区间为R[low . . high];
- 在排序区间任选一个记录Rx做为基准;
- 经过一趟排序后，将排序区间分为左、右两个子区间：

  R[low . . i-1] Rx R[i+1 . . high]

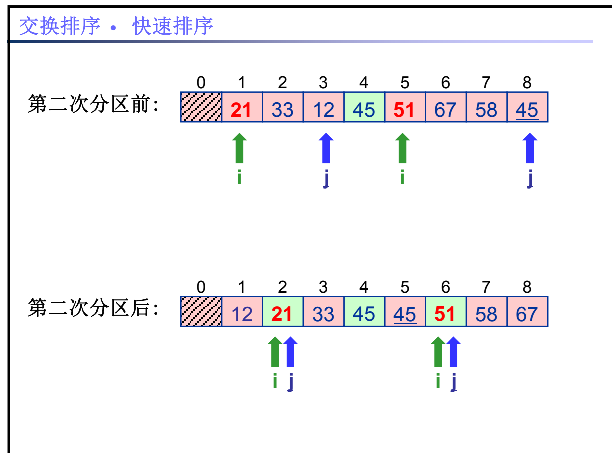
  使得： R[low. .i-1].key≤ Rx.key≤R[i+1. .high].key
- 然后再用相同的方法分别对左、右区间进行排序，直至每个区间长度都是1为止。

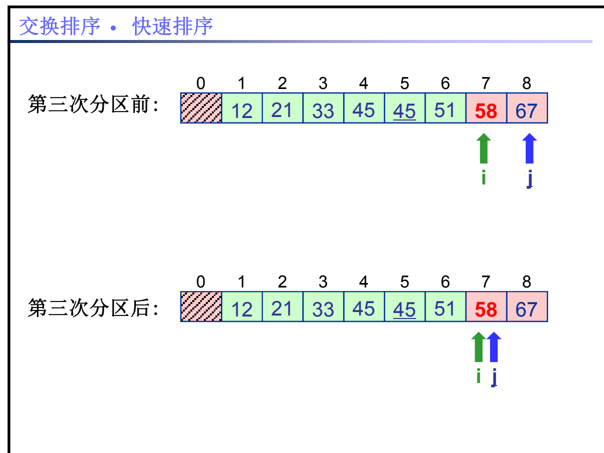
基准	0	1	2	3	4	5	6	7	8
		45	33	58	67	51	12	21	45
		i							j

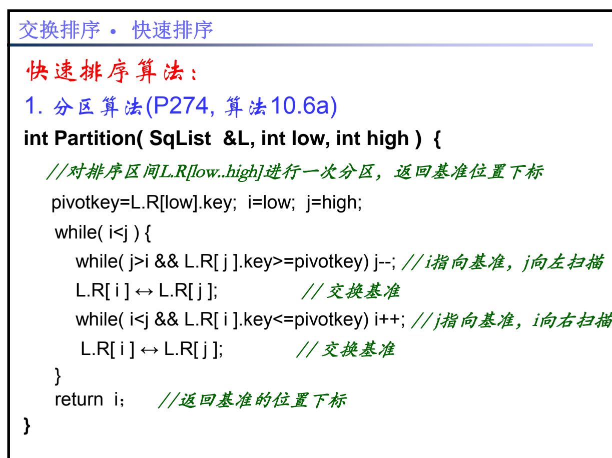
- 初态：low=1， high=n；
- 取区间的第一个元素为基准；
- 在区间的两端各设一个指针i和j，两个指针交替地向中间扫描；
- 首先，指针j向左扫描，当遇到键值小于基准的记录R[j]，就做一次交换R[i]↔R[j]；
- 然后i指针向右扫描，当遇到键值大于基准的记录R[i]，就做一次交换R[i]↔R[j]；
- 当两个指针相遇时，分区结束。

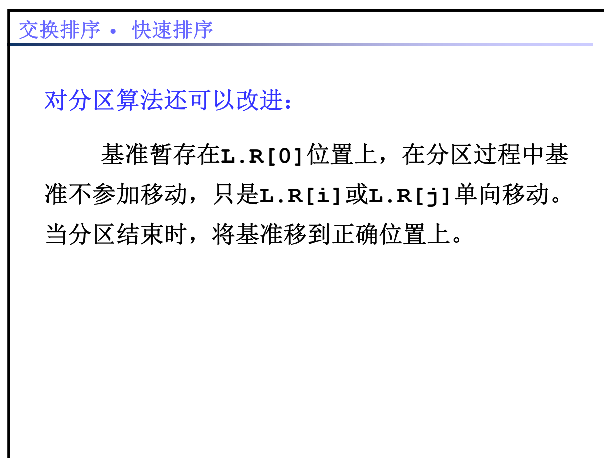
	0	1	2	3	4	5	6	7	8
第一次分区前：		45	33	58	67	51	12	21	45
		i							j

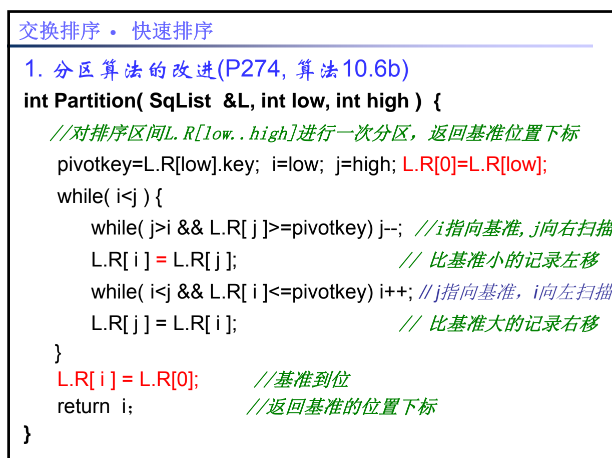
	0	1	2	3	4	5	6	7	8
第一次分区后：		21	33	12	45	51	67	58	45
					i j				

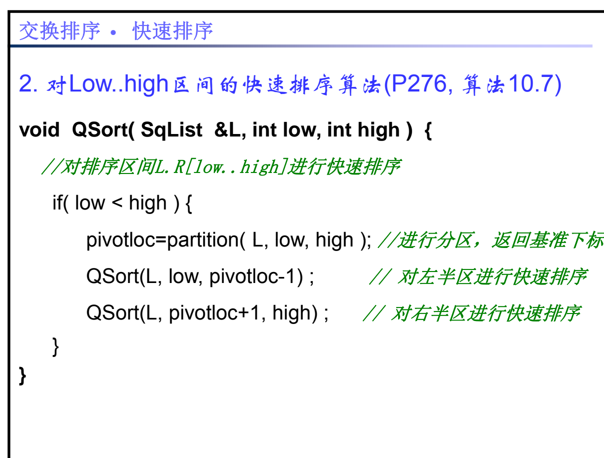

交换排序 • 快速排序
第二次分区前:
0 1 2 3 4 5 6 7 8
21 33 12 45 51 67 58 45
i j i j
第二次分区后:
0 1 2 3 4 5 6 7 8
12 21 33 45 45 51 58 67
i j i j


交换排序 • 快速排序
第三次分区前:
0 1 2 3 4 5 6 7 8
12 21 33 45 45 51 58 67
i j
第三次分区后:
0 1 2 3 4 5 6 7 8
12 21 33 45 45 51 58 67
i j


交换排序 • 快速排序
快速排序算法:
1. 分区算法(P274, 算法10.6a)
int Partition(SqList &L, int low, int high) {
//对排序区间L.R[low..high]进行一次分区，返回基准位置下标
pivotkey=L.R[low].key; i=low; j=high;
while(i<j) {
while(j>i && L.R[j].key>=pivotkey) j--; //i指向基准，j向左扫描
L.R[i] ↔ L.R[j]; // 交换基准
while(i<j && L.R[i].key<=pivotkey) i++; //j指向基准，i向右扫描
L.R[i] ↔ L.R[j]; // 交换基准
}
return i; //返回基准的位置下标
}


交换排序 • 快速排序
对分区算法还可以改进:
基准暂存在L.R[0]位置上，在分区过程中基准不参加移动，只是L.R[i]或L.R[j]单向移动。当分区结束时，将基准移到正确位置上。


交换排序 • 快速排序
1. 分区算法的改进(P274, 算法10.6b)
int Partition(SqList &L, int low, int high) {
//对排序区间L.R[low..high]进行一次分区，返回基准位置下标
pivotkey=L.R[low].key; i=low; j=high; L.R[0]=L.R[low];
while(i<j) {
while(j>i && L.R[j]>=pivotkey) j--; //i指向基准, j向右扫描
L.R[i] = L.R[j]; // 比基准小的记录左移
while(i<j && L.R[i]<=pivotkey) i++; //j指向基准，i向左扫描
L.R[j] = L.R[i]; // 比基准大的记录右移
}
L.R[i] = L.R[0]; //基准到位
return i; //返回基准的位置下标
}


交换排序 • 快速排序
2. 对Low..high区间的快速排序算法(P276, 算法10.7)
void QSort(SqList &L, int low, int high) {
//对排序区间L.R[low..high]进行快速排序
if(low < high) {
pivotloc=partition(L, low, high); //进行分区，返回基准下标
QSort(L, low, pivotloc-1) ; // 对左半区进行快速排序
QSort(L, pivotloc+1, high) ; // 对右半区进行快速排序
}
}

3. 对整个顺序表的快速排序算法(P276, 算法10.8)

```
void  QuickSort( SqList  &L) {
   // 对整个顺序表进行快速排序
     Qsort( L, 1, L.length );
}
```


快速排序性能分析：

最坏的情况：

表的初态恰好是正序或逆序排列。每次分区时，基准都恰好是区间的最大或最小键值，分区的结果是有一个区间为空。

对于初态是正序或逆序排列的表，需要进行n-1趟排序，每趟要进行n-i次比较：

$$Cmax=\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$$

快速排序退化成冒泡排序，时间复杂度达到$O(n^2)$.

最好的情况：每次分区时，基准都恰好是区间的中间值，分区的结果使得左、右两个区间长度一样，同步地收敛到1。

不妨设表长$n=2^k$，用C(m)表示对长度为m的表进行快速排序所需要的比较次数:

$Cmin=C(n)$

$\le n+2C(n/2)$

$\le n+2(n/2+2C(n/2^2))=2n+2^2C(n/2^2)$

$\le 2n+2^2(n/2^2+2C(n/2^3))=3n+2^3C(n/2^3)$

......

$\le kn+2^kC(n/2^k)=n\log_2 n+nC(1)$

$=O(n\log_2 n)$

- 就平均性能而言，快速排序的时间复杂度是**O(nlogn)**。
- 快速排序被认为是所有**O(nlogn)**级别的排序方法中平均性能最好的。
- 快速排序由于是递归实现的，需要消耗运行栈的空间
- 快速排序是非稳定的排序方法：

2 2 1 → 1 2 2

第十章 排序(Sort)

目录

§10.1 排序概述

§10.2 插入排序

§10.3 交换排序

§10.4 选择排序

§10.5 归并排序

§10.4 选择排序

10.4.1 简单选择排序

10.4.2 堆排序

简单选择排序算法思想：

- 设：排序区间R[1..n]；
- 在排序的过程中，整个排序区间被分为两个子区间：有序区R[1..i-1]和无序区R[i..n]；
- 共进行n－1趟排序，每趟排序都是选择无序区的最小记录R_{min}；将R_{min}与无序区的第一条记录位置互换，使得无序区长度减1，有序区长度增1。

R		R_1	R_2	…	R_{i-1}	R_i	R_{i+1}		R_{min}		R_n
	0	1	2		i-1	i	i+1			n-1	n

有序区　无序区

选择排序 • 简单选择排序

第一趟i=1

0	1	2	3	4	5	6	7	8
	45	33	58	67	51	12	21	45

i　j

此时，有序区为空，无序区为整个待排序区间。
最小值12与45交换

选择排序 • 简单选择排序

第二趟i=2

0	1	2	3	4	5	6	7	8
	12	33	58	67	51	45	21	73

i　j

无序区的最小值21与33交换

选择排序 • 简单选择排序

第三趟i=3

0	1	2	3	4	5	6	7	8
	12	21	58	67	51	45	33	73

i　j

无序区的最小值33与58交换

选择排序 • 简单选择排序

第四趟i=4

0	1	2	3	4	5	6	7	8
	12	21	33	67	51	45	58	73

i　j

无序区的最小值45与67交换

选择排序 • 简单选择排序

简单选择排序算法：

```
void SelectSort( SqList  &L ) {
    //对顺序表L作简单选择排序
    for(i=1; i<L.length; i++){
        min=i;
        for(j=i+1;j<=L.length;j++) //选最小的记录
            if(L.R[j].key<L.R[min].key)min=j;
        if(i!=min)L.R[min]↔L.R[i];
            //与无序区第一条记录交换
    }//end for
}
```

简单选择排序性能分析：

比较次数与表的初态无关：

$$C\min = C\max = \sum_{i=1}^{n-1}(n-i) = n(n-1)/2$$

最好的情况：表的初态恰好是正序排列

移动次数：$M\min = 0$

最坏的情况：每趟都有移动发生

移动次数：$M\max = 3(n-1)$

平均$O(n^2)$，不稳定的排序方法

§10.4 选择排序

10.4.1 简单选择排序

10.4.2 堆排序

10.4.2 堆排序(Heap Sort)

用建堆的方法来选择待排序区间的最大或最小键值。

一、堆定义

二、筛选操作

三、建堆算法

四、堆排序算法

一、堆定义

设n个元素的有限序列：

$K_1, K_2, K_3, \ldots, K_n$

如果满足

$K_i \le K_{2i}$ && $K_i \le K_{2i+1}$　*其中$1 \le i \le \lfloor n/2 \rfloor$*

则称这个序列为小根堆或小顶堆；

如果满足

$K_i \ge K_{2i}$ && $K_i \ge K_{2i+1}$　*其中$1 \le i \le \lfloor n/2 \rfloor$*

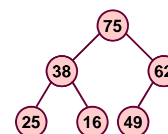
则称这个序列为大根堆或大顶堆；

- 回顾完全二叉树的性质，
- 编号为i的结点，其左孩子的编号为2i，右孩子的编号为2i+1
- 如果将堆序列看成完全二叉树的按层次遍历序列

  则这棵完全二叉树上每个结点的值比左孩子和右孩子值都要大(大根堆)，或比左孩子和右孩子值都要小(小根堆)。

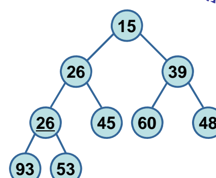
例：大根堆{ 75, 38, 62, 25, 16, 49 }

0	1	2	3	4	5	6
	75	38	62	25	16	49



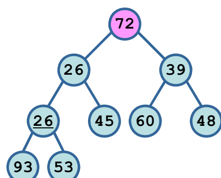
小根堆{ 15, 36, 39, 45, 36, 60, 48, 93, 53 }

0	1	2	3	4	5	6	7	8	9
	15	26	39	26	45	60	48	93	53

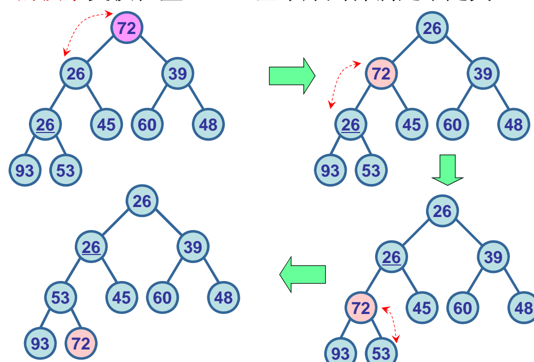


二、筛选操作

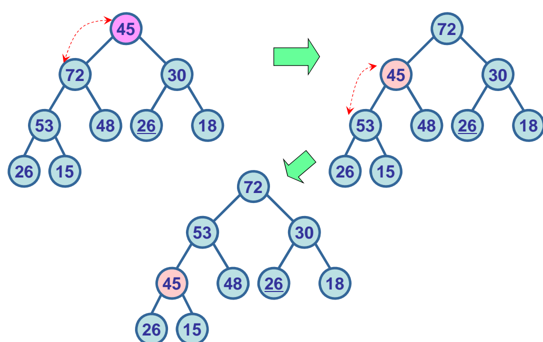
- 前提：根结点的左、右子树都是堆，而根结点不满足堆条件；
- 将根调整到合适的位置，使得整个序列成为堆。



对小根堆来说，筛选就是不断地将树根与较小的孩子交换位置，直至整个序列都满足堆定义。



对大根堆来说，筛选就是将树根与较大的孩子交换位置，直至整个序列都满足堆定义。



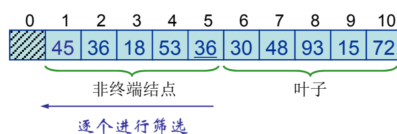
筛选算法，P281-282，算法10.10

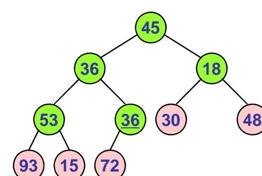
```
void HeapAdjustn ( SqList  &H,  int s,  int m ) {
    //已知H.R[s+1..m]满足大顶堆定义
    //调整H.R[s], H.R[s..m]使得成为大顶堆
   rc=H.R[s];
   for(j=2*s; j<=m; j*=2){
      if(j<m && H.R[j].key<H.R[j+1].key) j++;
                                    //令j指向较大的孩子
      if(rc.key>H.R[j].key) break;
      H.R[s]=H.R[j]; s=j;
   }
   H.R[s]=rc;
}
```

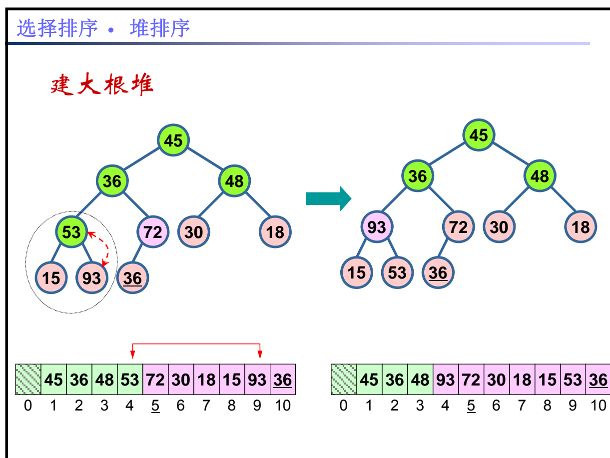

三、建堆

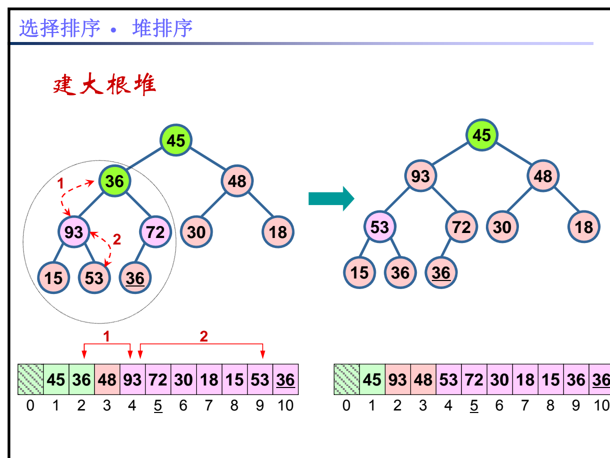
- 将一个任意序列建成一个小根堆或大根堆；
- 自下而上、自右向左地进行筛选，将以每一个非终端结点为根的子树筛选成堆；
- 具有n个结点的完全二叉树有⌈n/2⌉个叶子结点，一个叶子就是一个堆。
- 完全二叉树最后一个非终端结点的编号为⌊n/2⌋；

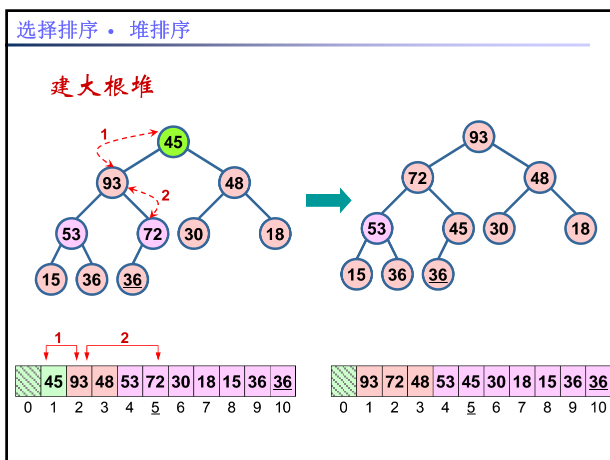
举例：将10个元素的序列建成一个大顶堆：











选择排序 • 堆排序

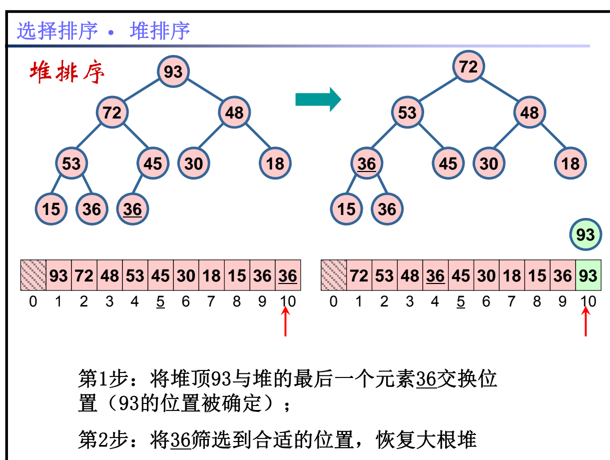
四、堆排序

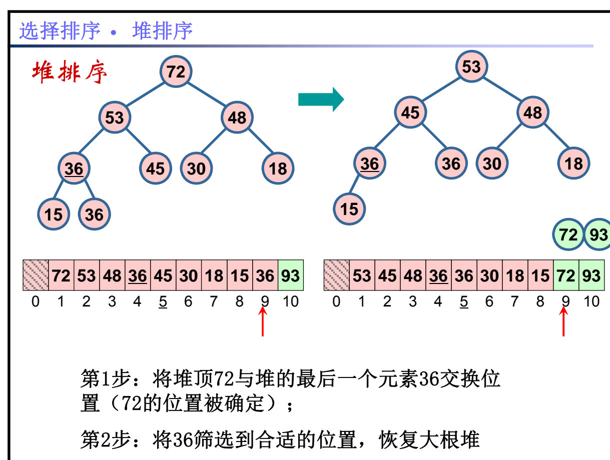
以大顶堆为例：

- 堆顶是排序区间最大的元素
- 去掉堆顶，将堆顶与堆的最后一个元素交换位置：

  ①最大元素归位；

  ②新树根不满足堆定义，需要通过筛选调整为堆





堆排序

	53	45	48	36	36	30	18	15	72	93
0	1	2	3	4	5	6	7	8	9	10

	48	45	30	36	36	15	18	53	72	93
0	1	2	3	4	5	6	7	8	9	10

第1步：将堆顶53与堆的最后一个元素15交换位置（53的位置被确定）；

第2步：将15筛选到合适的位置，恢复大根堆

堆排序

	48	45	30	36	36	15	18	53	72	93
0	1	2	3	4	5	6	7	8	9	10

	45	36	30	18	36	15	48	53	72	93
0	1	2	3	4	5	6	7	8	9	10

第1步：将堆顶48与堆的最后一个元素18交换位置（48的位置被确定）；

第2步：将18筛选到合适的位置，恢复大根堆

堆排序

	45	36	30	18	36	15	48	53	72	93
0	1	2	3	4	5	6	7	8	9	10

	36	36	30	18	15	45	48	53	72	93
0	1	2	3	4	5	6	7	8	9	10

第1步：将堆顶45与堆的最后一个元素15交换位置（45的位置被确定）；

第2步：将15筛选到合适的位置，恢复大根堆

第十章 排序(Sort)

目录

§10.1 排序概述

§10.2 插入排序

§10.3 交换排序

§10.4 选择排序

§10.5 归并排序

§10.6 基数排序

归并排序(Merging sort)算法思想：

- 一种基于将两个有序表异地归并成一个有序表的排序策略。
- 初态是将排序表中的每个元素看成是一个有序的子表，共有n个子表。

25 57 48 37 12 92 86 72 31 48

[25] [92] [48] [37] [12] [57] [86] [72] [31] [48]

- 经过一趟排序，将两个相邻的有序子表异地归并成一个有序子表；
- 共进行$\log_2 n$趟这样的归并，整个排序表就被归并成了一个有序表。

25 92 48 37 57 92 86 72 31 48

R: [25] [92] [48] [37] [12] [57] [86] [72] [31] [48]

T: [25 92] [37 48] [12 57] [72 86] [31 48]

R: [25 37 48 92] [12 57 72 86] [31 48]

T: [12 25 37 48 57 72 86 92] [31 48]

R: [12 25 31 37 48 48 57 72 86 92]

归并排序

归并排序算法（3级）：

(1) 两个相邻子表的归并:

```
void Merge(RecType SR[], RecType &TR[],
                         int i, int m, int n);
```

(2) 对区间[s..t]的归并排序

```
void MSort(RecType SR[], RecType &TR1[],
                              int s, int t);
```

(3) 对整个表的归并排序

```
void MergeSort(SqList &L)
```

归并排序

两个相邻子表的归并:

```
void Merge(RecType SR[], RecType &TR[],
                          int i, int m, int n){
  //将有序的SR[i..m]和SR[m+1..n]归并为有序的TR[i..n]
  for(j=m+1,k=i; i<m&&j<=n;  ++k){
     if(SR[i].key<=SR[j].key) TR[k]=SR[i++];
     else TR[k]=SR[j++];
  }
  if(i<=m) TR[k..n]=SR[i..m];
  if(j<=n) TR[k..n]=SR[j..m];
}  //end Merge
```

归并排序

对区间[s..t]的归并排序:

```
void MSort(RecType SR[], RecType &TR1[],
                              int s, int t){
  //将SR[s..t]归并排序为TR1[s..t]
  RecType TR2[n];   //定义辅助数组TR2
  if(S==t) TR1[s] = 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); //SR[s..m]排序成TR2[s..m]
    MSort(SR, TR2, m+1, t);//SR[m+1..t]排序成TR2[m+1..t]
    Merge(TR2, TR1, s, m, t); //两个有序的相邻区间归并
}  //end MSort
```

归并排序

整个表的归并排序

```
void MergeSort(SqList &L)
  //对顺序表L作归并排序
  MSort(L.r, L.r, 1, L.length);
}  //end MergeSort
```