

第三章

查找与排序(下)



本节内容

通过本单元的学习，了解、掌握有关排序的：

- 基本概念：
 - **排序、排序分类、算法稳定性**
- 典型的排序算法：
 - **插入排序、选择排序、交换排序**
 - **归并排序、基数排序**

3.3 基本的排序技术

排序的基本概念

- **定义**：将记录按**关键字**递增(递减)的次序排列起来，形成新的有序序列，称为排序。

- **描述**：

设n个记录的序列为 $\{R_1, R_2, \dots, R_n\}$,其相应关键字序列为 $\{K_1, K_2, \dots, K_n\}$,需确定一种排序 P_1, P_2, \dots, P_n ,使其相应的关键字满足递增(升序),或递减(降序)的关系:

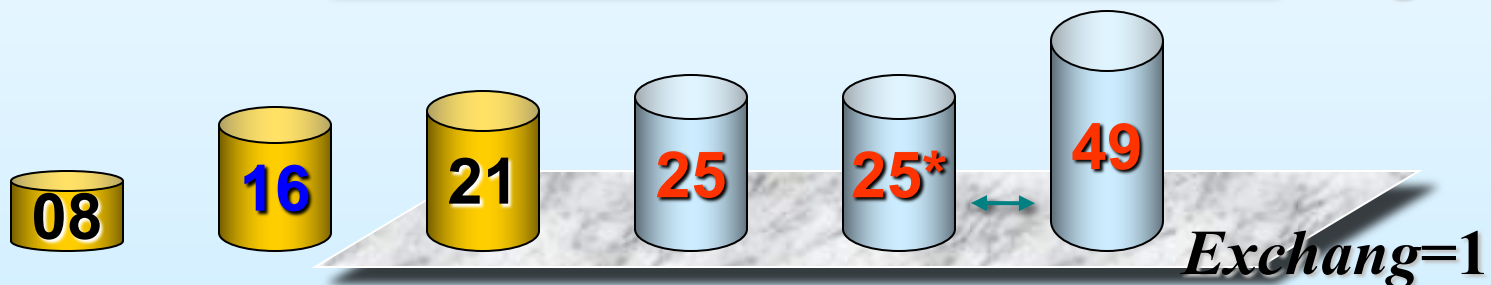
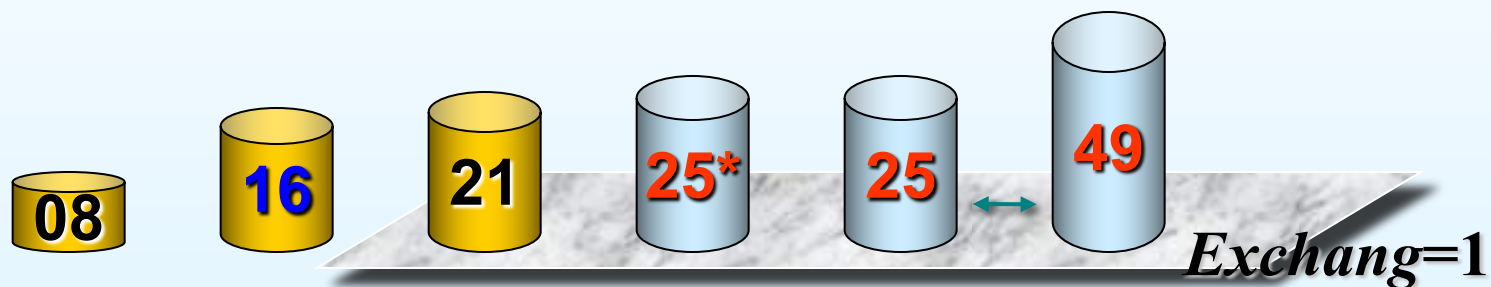
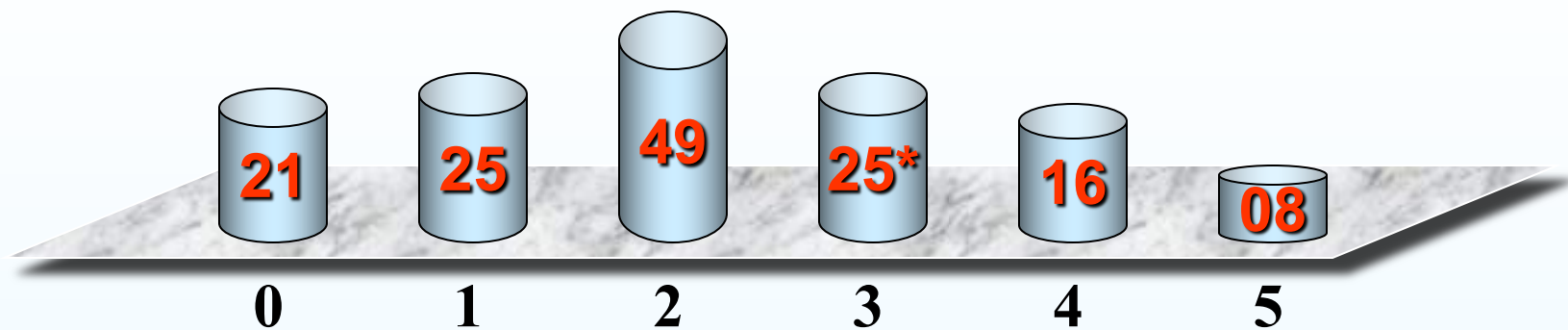
$$K_{p_1} \leq K_{p_2} \leq \dots \leq K_{p_n}$$

或

$$K_{p_1} \geq K_{p_2} \geq \dots \geq K_{p_n}$$

- 虽然排序算法是一个简单的问题，但是从计算机科学发展以来，已经有大量的研究在此问题上。举例而言，**冒泡排序**在**1956**年就已经被研究。虽然大部分人认为这是一个已经被解决的问题，有用的新算法仍在不断的被发明。（例子：**图书馆排序**在**2004**年被发表）

算法稳定性



算法稳定性

- 当相等的元素是无法分辨的，比如像是整数，稳定性并不是一个问题。然而，假设以下的数对将要以他们的第一个数字来排序。
- (4, 1) (3, 1) (3, 7) (5, 6)
- (3, 1) (3, 7) (4, 1) (5, 6) (保持次序)
- (3, 7) (3, 1) (4, 1) (5, 6) (次序被改变)

- 不稳定排序算法可能会在相等的键值中改变纪录的相对次序。
- 不稳定排序算法可以被特别地实现为稳定。方法是 人工扩充键值的比较。然而，要记住这种次序通常牵 涉到额外的空间负担。

稳定的

- 冒泡排序 (bubble sort) — $O(n^2)$
- 鸡尾酒排序 (Cocktail sort, 双向的冒泡排序) — $O(n^2)$
- 插入排序 (insertion sort) — $O(n^2)$
- 桶排序 (bucket sort) — $O(n)$; 需要 $O(k)$ 额外空间
- 计数排序 (counting sort) — $O(n+k)$; 需要 $O(n+k)$ 额外空间
- 合并排序 (merge sort) — $O(n \log n)$; 需要 $O(n)$ 额外空间
- 原地合并排序 — $O(n^2)$
- 二叉排序树排序 (Binary tree sort) — $O(n \log n)$ 期望时间; $O(n^2)$ 最坏时间; 需要 $O(n)$ 额外空间
- 鸽巢排序 (Pigeonhole sort) — $O(n+k)$; 需要 $O(k)$ 额外空间
- 基数排序 (radix sort) — $O(n \cdot k)$; 需要 $O(k)$ 额外空间

不稳定

- 选择排序 (selection sort) — $O(n^2)$
- 希尔排序 (shell sort) — $O(n \log n)$ 如果使用最佳的现在版本
- 组合排序 — $O(n \log n)$
- 堆排序 (heapsort) — $O(n \log n)$
- 平滑排序 — $O(n \log n)$
- 快速排序 (quicksort) — $O(n \log n)$ 期望时间, $O(n^2)$ 最坏情况; 对于大的、乱数列表一般相信是最快的已知排序

排序算法的数据结构

- 简单起见，这里用顺序存储结构描述待排序的记录。
- 顺序存储结构（C语言描述）：

```
#define N n  
typedef struct record {  
    int key ;    /* 关键字项 */  
    int otherterm ; /* 其它项 */  
} ;  
typedef struct record RECORD ;  
RECORD file[N+1] ; /*RECORD型的N+1元数组  
*/
```

典型排序算法

- 冒泡排序
- 快速排序
- 简单插入排序
- 希尔排序
- 简单选择排序
- 堆排序
- 归并排序
- 基数排序
- 二叉排序树

一、冒泡排序

■ 1.指导思想：

两两比较待排序记录的关键字，并交换**不满足顺序**要求的那些偶对元素，直到全部数列满足有序为止。

- **冒泡排序 (Bubble sort)** 是基于交换排序的一种算法。它是依次两两比较待排序元素；若为逆序（递增或递减）则进行交换，将待排序元素从左至右比较一遍称为一趟“冒泡”。每趟冒泡都将待排序列中的最大关键字交换到最后（或最前）位置。直到全部元素有序为止。

$\cap \cap \cap \dots \cap$
 $a_1 \ a_2 \ a_3 \ \dots \ a_{n-1} \ a_n$

↑
—— 最大值

2.冒泡排序算法

- **step1** 从待排序队列的前端开始(a_1)两两比较记录的关键字值, 若 $a_i > a_{i+1}$ ($i=1, 2, \dots, n-1$), 则交换 a_i 和 a_{i+1} 的位置, 直到队列尾部。一趟冒泡处理, 将序列中的最大值交换到 a_n 的位置。
- **step2** 如法炮制, 第 k 趟冒泡, 从待排序队列的前端开始(a_1)两两比较 a_i 和 a_{i+1} ($i=1, 2, \dots, n-k$), 并进行交换处理, 选出序列中第 k 大的关键字值, 放在有序队列的最前端。(思考: 为什么 $i=1, \dots, n-k$?)
- **Step3** 最多执行 $n-1$ 趟的冒泡处理, 序列变为有序。
- **从小到大排序**

冒泡排序算法举例

设有数列{ 65,97,76,13,27,49,58 } 比较次数

第1趟 {65 , 76 , 13 , 27 , 49 , 58} , {97} 6

第2趟 {65 , 13 , 27 , 49 , 58} , {76 , 97} 5

第3趟 {13 , 27 , 49 , 58} , {65 , 76 , 97} 4

第4趟 {13 , 27 , 49} , {58 , 65 , 76 , 97} 3

第5趟 {13 , 27} , {49 , 58 , 65 , 76 , 97} 2

第6趟 {13} , {27 , 49 , 58 , 65 , 76 , 97} 1

总计： 21 次

3.冒泡排序实现

```
bubble(int *item,int count)
{ int a,b , t;
  for(a=1;a<count; a++)    /*n-1趟冒泡处理*/
    for(b=1;b<=count-a;b++)/*每趟n-i次的比较*/
      if(item[b-1]>item[b])/*若逆序，则交换*/
      { t=item[b-1];    /* 它们的位置 */
        item[b-1]=item[b];
        item[b]=t;
      }
}
```

4.改进的冒泡排序

- 从上述举例中可以看出，从第4趟冒泡起，序列已有序，结果是空跑了3趟。**当某一趟冒泡排序没有元素交换时，说明序列已有序，则停止交换。**这就是改进的冒泡算法的处理思想。
- 用改进的冒泡算法进行处理，比较次数有所减少。

对于数列{ 65,97,76,13,27,49,58 } 比较次数

第1趟 {65 , 76 , 13 , 27 , 49 , 58} , {97} 6

第2趟 {65 , 13 , 27 , 49 , 58} , {76 , 97} 5

第3趟 {13 , 27 , 49 , 58} , {65 , 76 , 97} 4

第4趟 {13 , 27 , 49} , {58 , 65 , 76 , 97} 3

总计：18 次

```
bubble_a(int *item,int count)
```

```
{  int a,b,t,c;
```

```
    for(a=1;a<count;++a) /* n-1趟的循环 */
```

```
    {  c=1;          /* 设置交换标志 */
```

```
        for(b=1;b<=count-a;b++)/* n-1趟处理 */
```

```
        {  if(item[b-1]>item[b])/* 若逆序，则 */
```

```
            {  t=item[b-1];    /* 交换位置 */
```

```
                item[b-1]=item[b];
```

```
                item[b]=t;
```

```
                c=0;  } /* 若有交换，则 */
```

```
        } /* 改变交换标志 */
```

```
    if(c) break; /* 若没有交换，则 */
```

```
    } /* 退出处理 */
```

```
}
```



5. 算法评价

- ❖ 若待排序列有序(递增或递减)，则只需进行一趟冒泡处理即可；若反序，则需进行 $n-1$ 趟的冒泡处理。在最坏的情况下，冒泡算法的时间复杂度是 $O(n^2)$ 。
- ❖ 当待排序列基本有序时，采用冒泡排序法效果较好。
- ❖ 冒泡排序算法是稳定的。

课堂练习

- 对下列数据进行冒泡排序
- **23 ,34,69,21,5,66,7,8,12,34**

二、快速排序

- 快速排序法是对冒泡排序法的一种改进，也是基于交换排序的一种算法。因此，被称为“分区交换排序”。
- 快速排序法是一位计算机科学家 C.A.R.Hoare 推出并命名的。曾被认为是最好的一种排序方法。

1.快速排序基本思想

- 在待排序序列中按某种方法选取一个元素K，以它为分界点，用交换的方法将序列分为两个部分：比该值小的放在左边，否则放在右边。形成

{左子序列}K{右子序列}

再分别对左、右两部分实施上述分解过程，直到各子序列长度为1，即有序为止。

- **分界点元素值K的选取方法不同，将构成不同的排序法，也将影响排序的效率：**
 - 取左边第1个元素为分界点；
 - 取中点 $A[(left+right)/2]$ 为分界点；
 - 选取最大和最小值的平均值为分界点等。

2. 快速排序算法

- **Step1** 分别从两端开始，指针*i*指向第一个元素A[left]，指针*j*指向最后一个元素A[right]，分界点取K；
- **Step2** 循环 ($i \leq j$)
 - 从左边开始进行比较：
若 $K > A[i]$ ，则 $i = i + 1$ ，再进行比较；
若 $K \leq A[i]$ ，则将A[i]交换到右边。
 - 从右边开始进行比较：
若 $K \geq A[j]$ ，则将A[j]交换到左边；
若 $K < A[j]$ ，则 $j = j - 1$ ，再进行比较；
 - 当*i=j*时，一次分解操作完成。
- **Step3** 在对分解出的左、右两个子序列按上述步骤继续进行分解，直到子序列长度为1（不可再分）为止，也即序列全部有序。

```

qs(int *item,int left,int right)
{ int i,j,x,y,k; i=left; j=right;
  x=item[(left+right)/2];          /* 计算中点位置      */
  do{                               /* i≤j 的循环处理    */
    while(item[i]<x && i<right )
      i++ ;                        /* 确定i点交换位置  */
    while(x<item[j] && j>left)
      j--;                          /* 确定j点交换位置  */
    if(i<=j)                       /* 如果i、j位置合法，则交换 */
    { y=item[i];                  /* A[i]和A[j]的位置      */
      item[i]=item[j];
      item[j]=y; i++; j--;        }
  } while(i<=j);
  if(left<j) qs(item,left,j);      /* 对分割出的左部处理*/
  if(i<right) qs(item,i,right);    /*对分割出的右部处理*/ }

```

快速排序算法举例

对于数列{49, 38, 60, 90, 70, 15, 30, 49},

采用中点分界法:

初始状态: 49 38 60 90 70 15 30 49 比较次数

$k = 90$

第1趟

49 38 60 90 70 15 30 49

$i \uparrow$

$j \uparrow$

49 38 60 90 70 15 30 49 5 ($i4, j1$)

$i \uparrow$

$j \uparrow$

49 38 60 49 70 15 30 90

5 ($i4, j1$) $j \uparrow$

{ 49 38 60 49 70 15 30 } 90

小计: 10

快速排序算法举例（续一）

初始状态: 49 38 60 49 70 15 30 比较次数

第2趟

↓ $k = 49$

49 38 60 49 70 15 30

$2(i1, j1)$

$i \uparrow$ $j \uparrow$

30 38 60 49 70 15 49

$i \uparrow$ $j \uparrow$

30 38 60 49 70 15 49

$i \uparrow$ $j \uparrow$

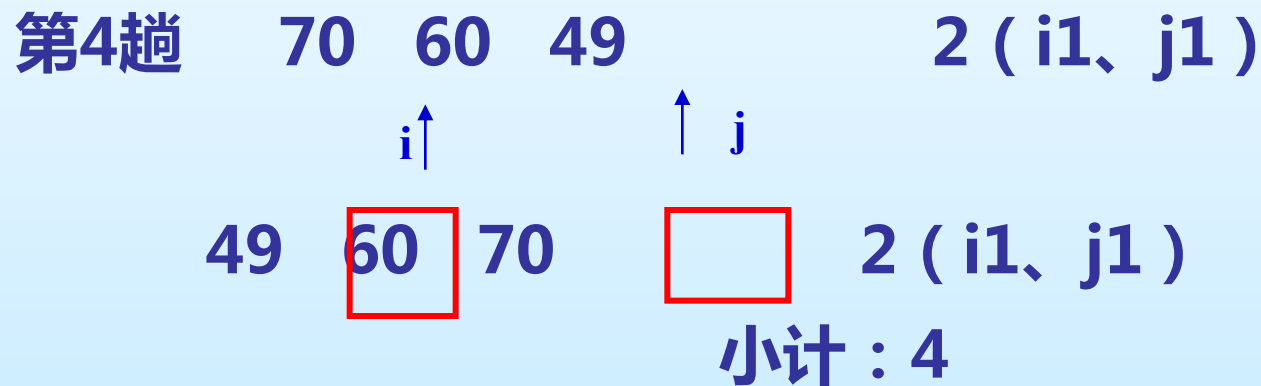
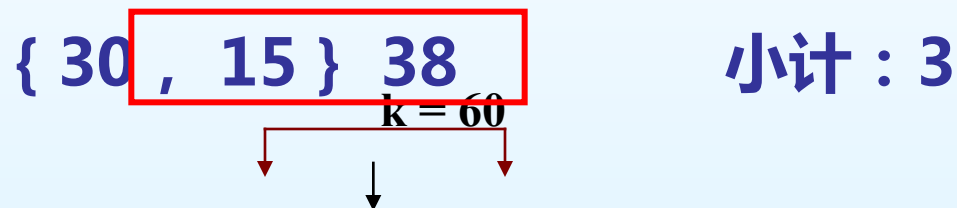
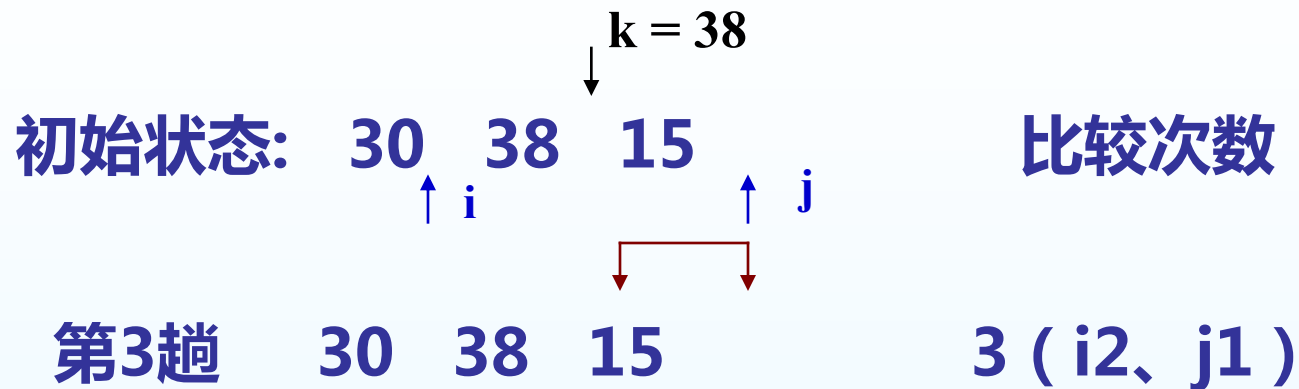
3 ($i2, j1$)

30 38 15 49 70 60 49

3 ($i1, j2$)

{ 30 38 15 } 49 { 70 60 49 } 小计 : 8

快速排序算法举例（续二）



快速排序算法举例（续三）

初始状态: 30 15
 ↓ $k = 30$
 ↑ i ↑ j

比较次数

第5趟 30 15

2 ($i1$ 、 $j1$)

15 30

小计 : 2

最后状态 :

{ 15 30 38 49 49 60 70 90 } 总计 : 27

课堂练习

■ P233 3.9

数据（2）

4. 算法评价

- ❖ 分界点选取方法不同，排序效果差异很大；
- ❖ 比较次数为 $n \log n$ ，即为： $O(n \log n)$ 。
- ❖ 快速排序算法是**不稳定的**。

三、简单插入排序

1.基本思想：

- 将n个元素的数列分为已有序和无序两个部分。

$\{\{a_1\}, \{a_2, a_3, a_4, \dots, a_n\}\}$

$\{\{a_1^{(1)}, a_2^{(1)}\}, \{a_3^{(1)}, a_4^{(1)}, \dots, a_n^{(1)}\}\}$

.....

$\{\{a_1^{(n-1)}, a_2^{(n-1)}, \dots\}, \{a_n^{(n-1)}\}\}$

有序

无序

- 每次处理：将无序数列的第一个元素与有序数列的元素从后往前逐个进行比较,找出插入位置,将该元素插入到有序数列的合适位置中。
- 从前往后，若比 a_i 小，则放在 a_i 前面
- 从后往前，若比 a_i 大，则放在 a_i 后边。

2.插入排序算法步骤

- *Step1* 从有序数列 $\{a_1\}$ 和无序数列 $\{a_2, a_3, \dots, a_n\}$ 开始进行排序;
- *Step2* 处理第 i 个元素时($i=2, 3, \dots, n$), 数列 $\{a_1, a_2, \dots, a_{i-1}\}$ 是已有序的, 而数列 $\{a_i, a_{i+1}, \dots, a_n\}$ 是无序的。用 a_i 与 a_{i-1} 、 a_{i-2}, \dots, a_1 进行比较, 找出合适的位置将 a_i 插入。(从后往前比较)
- *Step3* 重复*Step2*, 共进行 $n-1$ 的插入处理, 数列全部有序。(从小到大排序)

插入排序举例

设有数列{ 18, 12, 10, 12, 30, 16 }

初始状态: {18}, {12, 10, 12, 30, 16} 比较次数

i=1 {18}, {12, 10, 12, 30, 16} 1

i=2 {12, 18}, {10, 12, 30, 16} 2

i=3 {10, 12, 18}, {12, 30, 16} 2

i=4 {10, 12, 12, 18}, {30, 16} 1

i=5 {10, 12, 12, 18, 30}, {16} 3

{10, 12, 12, 16, 18, 30 }

总计: 9 次

插入排序算法实现

```
insert_sort(item , n )  
int *item ,   n ;  
{ int i, j, t ;  
  for (i=1; i<n ; i++ )  
  { t=item[i];  
    j = i - 1;  
    while(j>=0 && t < item[j])  
    { item[j+1]=item[j];  
      j- - ;  
    }  
    item[j+1]=t;  
  }  
}
```

//n-1次循环
// 要插入的元素
// 有序部分起始位置
// 寻找插入位置
// 当前元素后移

// 插入该元素

4. 算法评价

- 插入算法比较次数和交换次数约为 $n^2/2$, 因此, 其时间复杂度为 $O(n^2)$ 。
- 该算法是稳定的。

四.希尔 (Shell) 排序

1.思想：

- 希尔排序是一种快速排序法，出自 D.L.Shell,

- **指导思想：**

仍采用插入法，排序过程通过调换并移动数据项来实现。每次**比较指定间距的两个数据项**，若右边的值小于左边的值，则交换它们的位置。间距d按给定公式减少：

$d_{i+1} = (d_i + 1) / 2$ ，直到d等于1为止。d取 {9, 5, 3, 2, **1**}。

2. 算法步骤

- **Step1** 将n个元素的数列分为m个部分，元素比较间距为d。

- **Step2** 在第i步，两元素比较的间距取

$$d_{i+1} = (d_i + 1) / 2 \quad \{9, 5, 3, 2, 1\}$$

若 $a_i > a_{i+1}$ ，则交换它们的位置。

- **Step3** 重复上述步骤，直到 $d_K = 1$ 。

希尔排序例子

[13 14 94 33 82 25 59 94 65 23 45 27 73 25 39 10],

d=5

13	14	94	33	82
25	59	94	65	23
45	27	73	25	39
10				

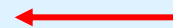
插入排序



10	14	73	25	23
13	27	94	33	39
25	59	94	65	82
45				

最后以1步长进行排序
(此时就是简单的插入排序了)

10	14	13
25	23	33
27	25	59
39	65	73
45	94	82
94		



10	14	73
25	23	13
27	94	33
39	25	59
94	65	82
45		

d=3

- 希尔排序是基于插入排序的以下两点性质而提出改进方法的：
 - 1) 插入排序在对几乎已经排好序的数据操作时，效率高，即可以达到线性排序的效率；
 - 2) 但插入排序一般来说是低效的，因为插入排序每次只能将数据移动一位。

3. SHELL排序算法(c++语言)

```
shellsort(T item[],int n)
{  int i,j,h; T t;
   h=n/2 ;
   while(h>0)
   {   for(i=h;i<n;i++) //内部为插入排序
       {   t=item[i]; j=i-h;
           while(t<item[j]&& j>=0)
               {   item[j+h]=item[j];  j=j-h; }
           item[j+h]=t;
       }
       h=h/2;
   }
}
```

4. 算法评价

- 希尔排序算法比较次数约为 $n^{1.5}$ ，因此，其时间复杂度为 $O(n^{1.5})$ 。
- 该算法是**不稳定的**。

希尔排序课堂练习

- 23 33 21 1 24 14 2 26 90 43
- d=5 3 1

五、简单选择排序

- **1.基本思想**：每次从待排序的记录中选出关键字最小（或最大）的记录，顺序放在已有序的记录序列的最后（或最前）面，直到全部数列有序。

$$\{\{\mathbf{a1}\}, \{\mathbf{a2}, \mathbf{a3}, \mathbf{a4}, \dots, \mathbf{an}\}\}$$
$$\{\{\mathbf{a1}^{(1)}, \mathbf{a2}^{(1)}\}, \{\mathbf{a3}^{(1)}, \mathbf{a4}^{(1)} \dots, \mathbf{an}^{(1)}\}\}$$

● ● ● ● ● ●

$\{\{ \mathbf{a1^{(n-1)}} , \mathbf{a2^{(n-1)}} , ... \}, \{ \mathbf{an^{(n-1)}} \} \}$

有序
无序

2.选择排序算法步骤

- *Step1* 从原始数列 $\{a_1, a_2, a_3, \dots, a_n\}$ 开始进行排序，比较 $n-1$ 次，从中选出最小关键字，放在有序数列中，形成 $\{a_1\}$ 、 $\{a_2, a_3, \dots, a_n\}$ 有序数列和无序数列两部分，完成第1趟排序。
- *Step2* 处理第 i 趟排序时($i=2, 3, \dots, n$)，从剩下的 $n-i+1$ 个元素中找出最小关键字，放在有序数列的后面。
- *Step3* 重复*Step2*，共进行 $n-1$ 趟的选择处理，数列全部有序。

选择排序举例

设有数列{ 18, 12, 10, 12, 30, 16 }

初始状态: {}, {18, 12, 10, 12, 30, 16} 比较次数

i=1 {10}, {18, 12, 12, 30, 16} 5

i=2 {10, 12}, {18, 12, 30, 16} 4

i=3 {10, 12, 12}, {18, 30, 16} 3

i=4 {10, 12, 12, 16}, {18, 30} 2

i=5 {10, 12, 12, 16, 18}, {30} 1

总计: 15 次

3.选择排序算法

```
select_sort(int *item,int count)
```

```
{ int i, j, k, t;
```

```
  for(i=0;i<count-1;++i)    // n-1次循环
```

```
  { k=i;                    // 无序部分第1个元素
```

```
    t=item[i];              // 及位置
```

```
    for(j=i+1;j<count;++j)  // 寻找最小值循环
```

```
      { if(item[j]<t)
```

```
        { k=j; t=item[j]; }    // 记录最小值及位置
```

```
      }
```

```
      item[k]=item[i];         // 交换最小值与有序
```

```
      item[i]=t;               // 部分最后1个元素位置
```

```
    }
```

```
}
```

4. 算法评价

- 每次选出当前最小关键字，但没有为以后的选择留下任何信息，比较次数仍为 $O(n^2)$ 。
- 选择排序算法是不稳定的。

六、堆排序

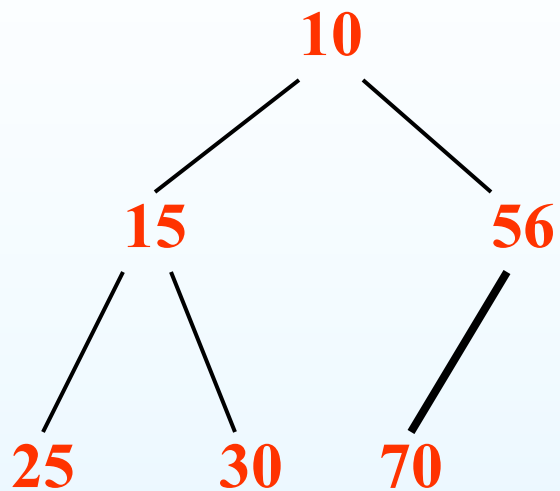
- 堆排序是一种树型**选择排序**。
- 在排序过程中，将 $R[0]$ 到 $R[n-1]$ 看成是一个**完全二叉树顺序存储结构**，利用**完全二叉树**中双亲结点和孩子结点之间的内在关系来选择关键字最小记录。
- 堆排序分为两个步骤：
 - 1、根据初始输入，形成初始堆。
 - 2、通过一系列的对象交换和重新调整进行排序。

1. 堆的定义

n 个关键字序列 K_1, k_2, \dots, K_n 称为堆，当且仅当该序列满足特性：

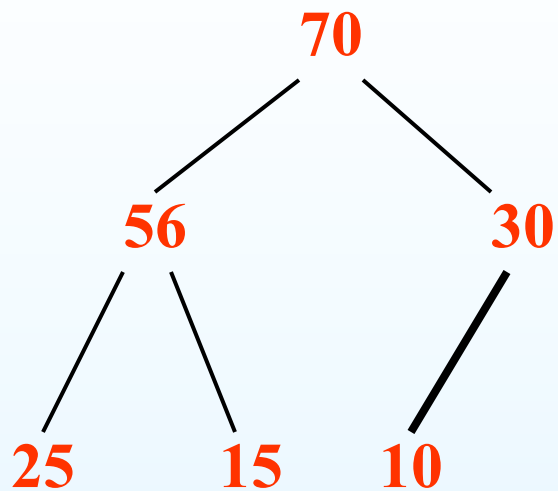
$$K_i \leq K_{2i} \text{ 和 } K_i \leq K_{2i+1} \quad (0 \leq i \leq low = (n-1)/2)$$

从堆的定义可以看出，堆实质上是满足如下性质的二叉树：树中任一非叶子结点的关键字均小于或等于它的孩子结点的关键字。



10	15	56	25	30	70
----	----	----	----	----	----

小根堆示例



70	56	30	25	15	10
----	----	----	----	----	----

大根堆示例

* 2. 建堆

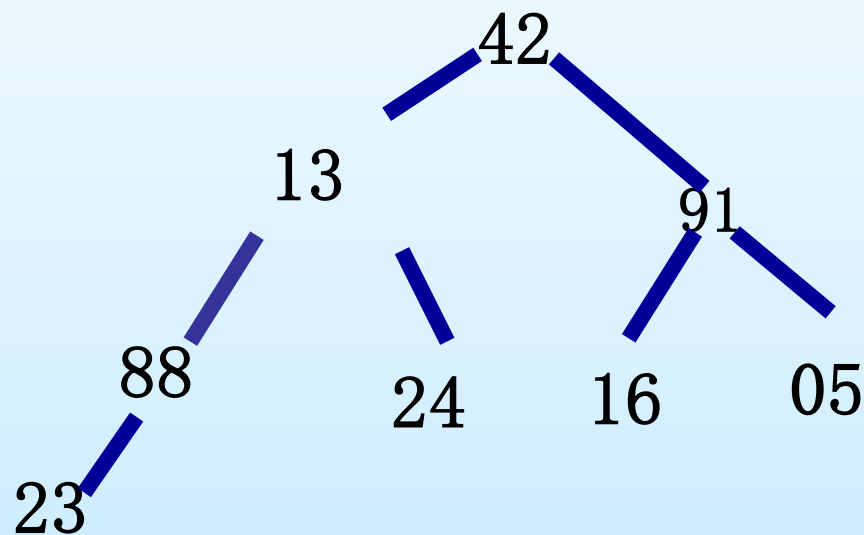
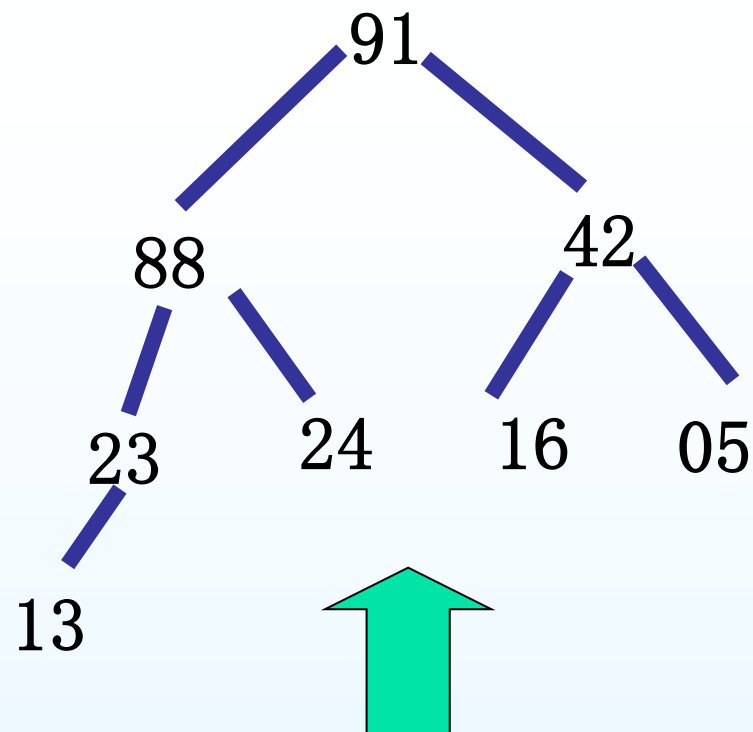
■建堆的方法

次序

思想

举例

■实现算法



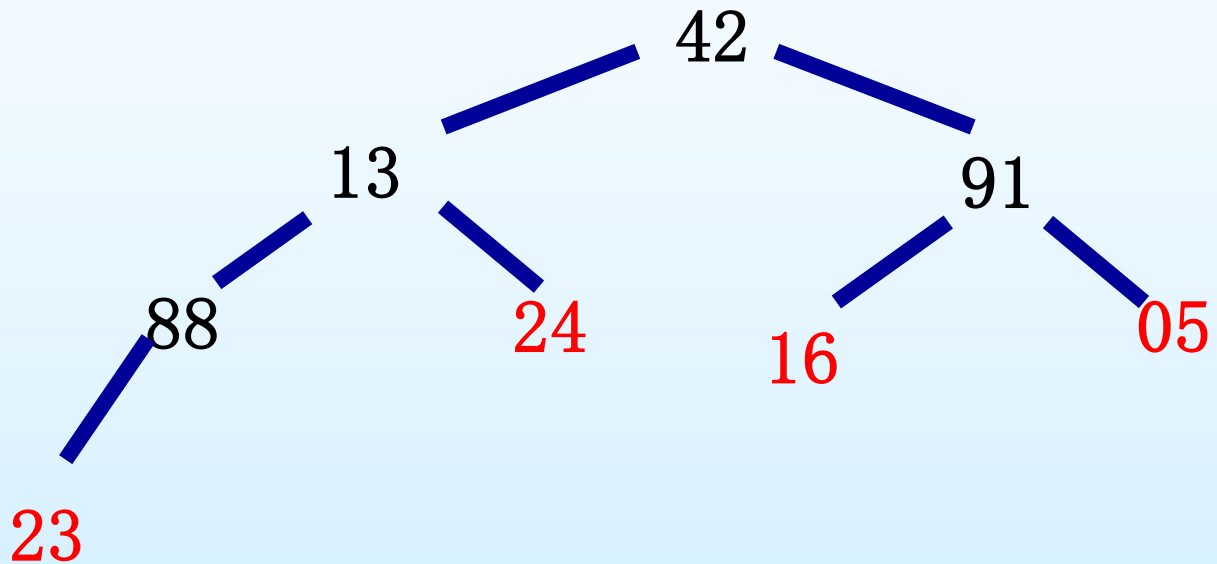
(1) 建堆次序

(a) 只有一个结点的树是堆

(b) 而在完全二叉树中，所有序号 $i \geq \text{low}(n/2)$ 的结点都是叶子，因此以这些结点为根的子树都已是堆。



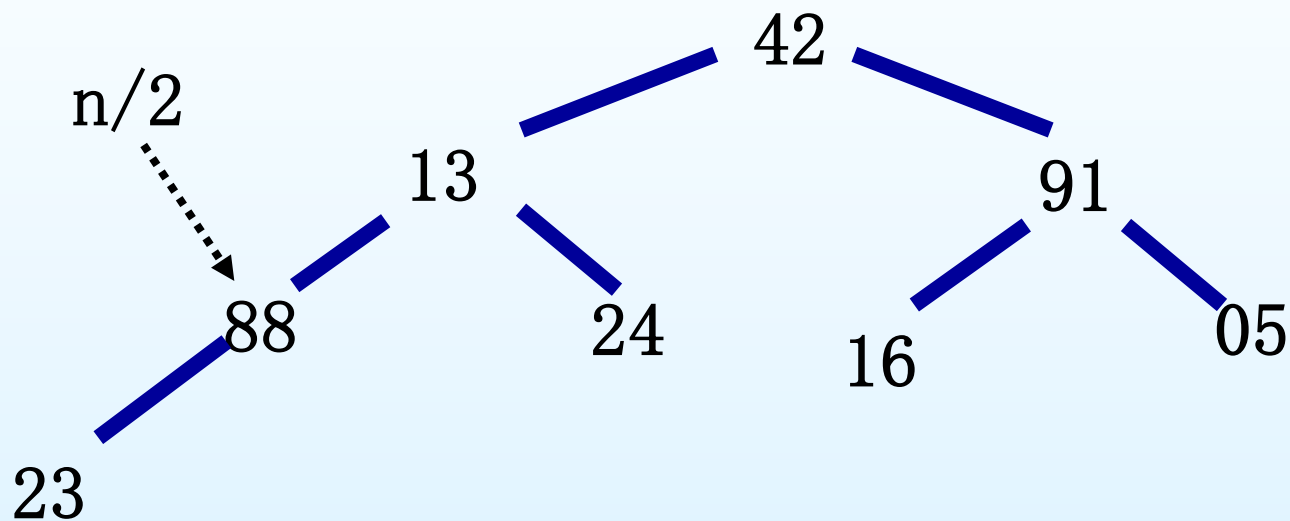
一个结点的树是堆



建大根堆

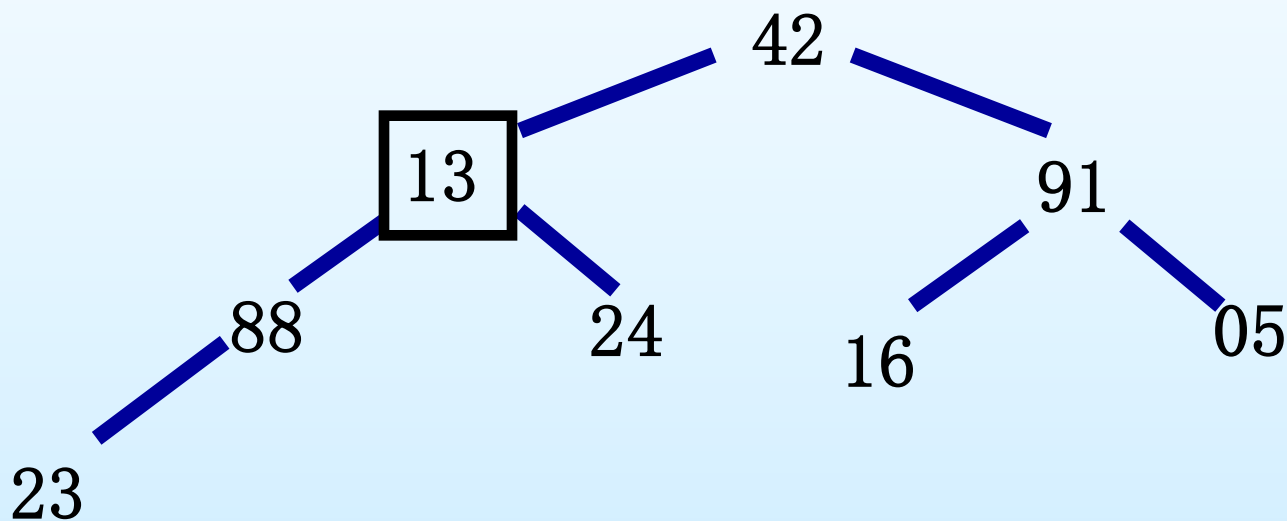
(1) 建堆次序

(c) 只需依次将序号为 $\text{low}(n/2)$ $\text{low}(n/2)-1$, ... , 1的结点作为根的子树都调整为堆即可。

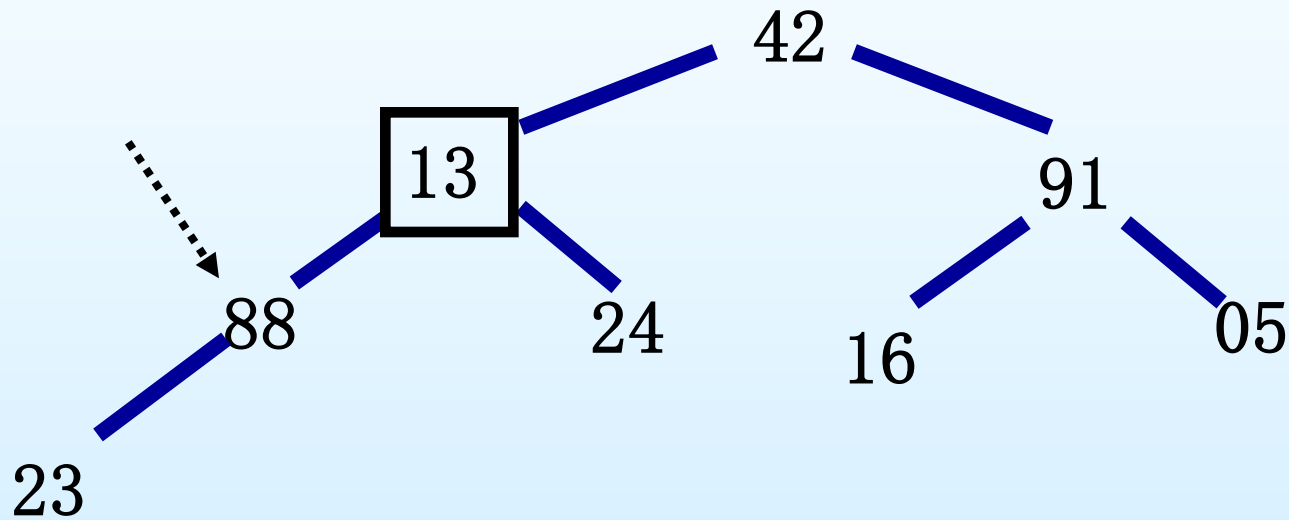


(2) 建堆方法--“筛选法”

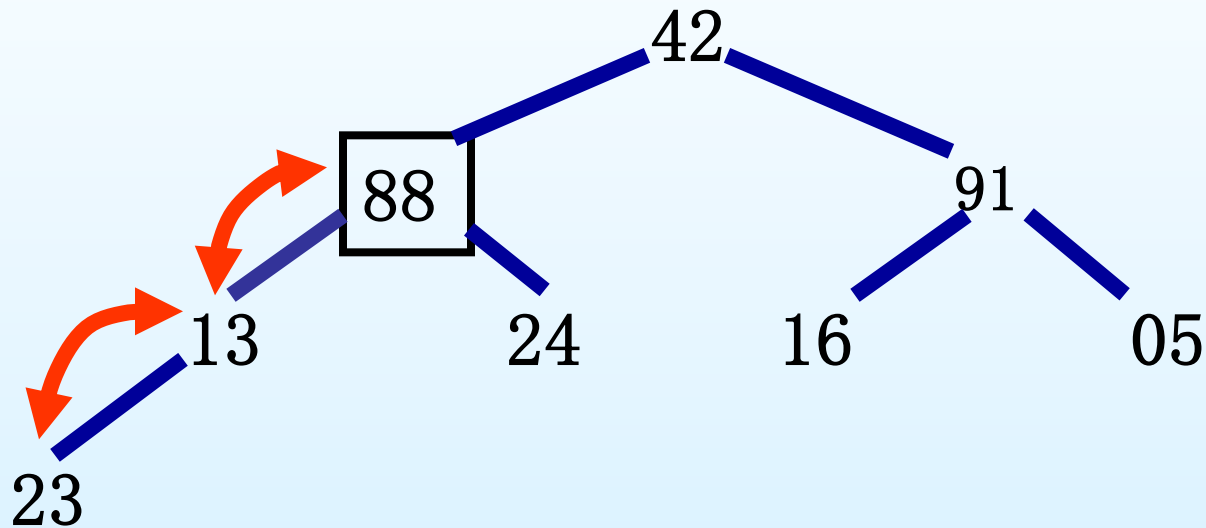
一: 如果 $R[i]$ 的左右子树已是堆，这两棵子树的根分别是各自子树中关键字最大的结点。

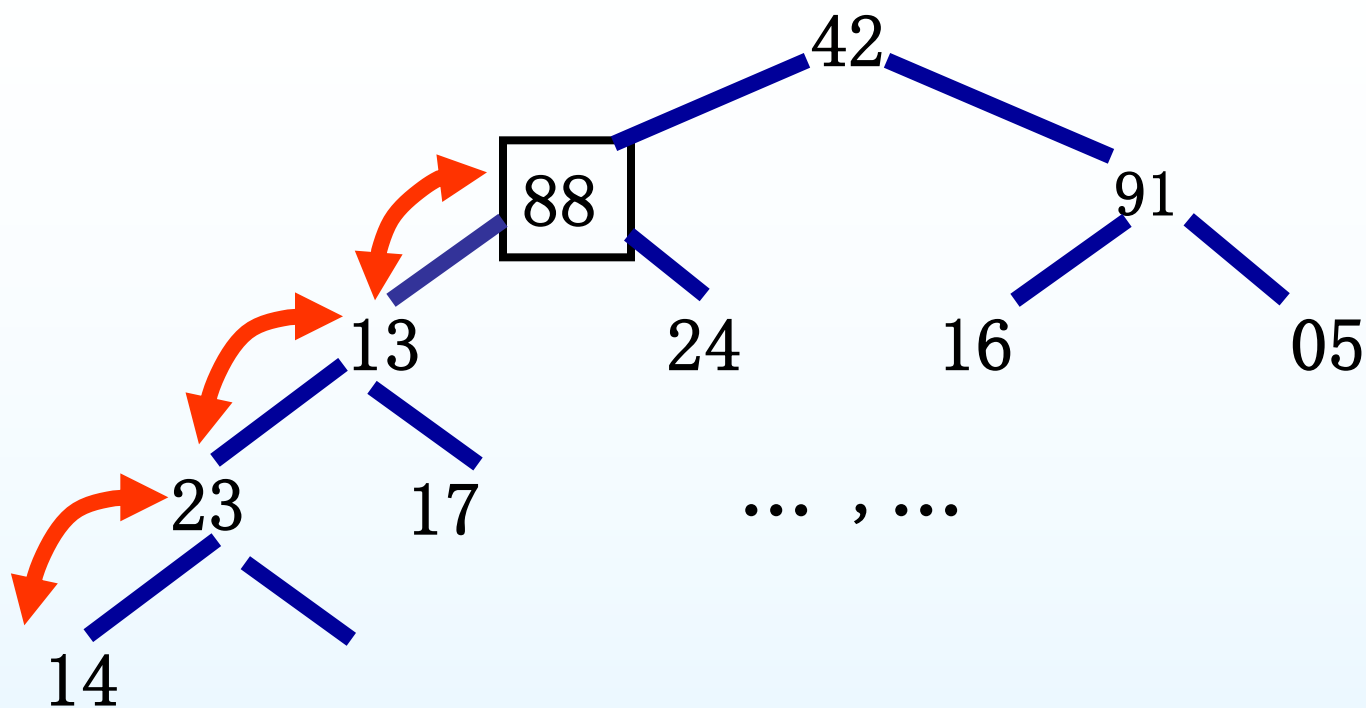


二:若根的关键字已是三者（根、左孩子、右孩子）中的最大者，则无须做任何调整；**否则**必须将具有**最大关键字**的孩子与根交换。



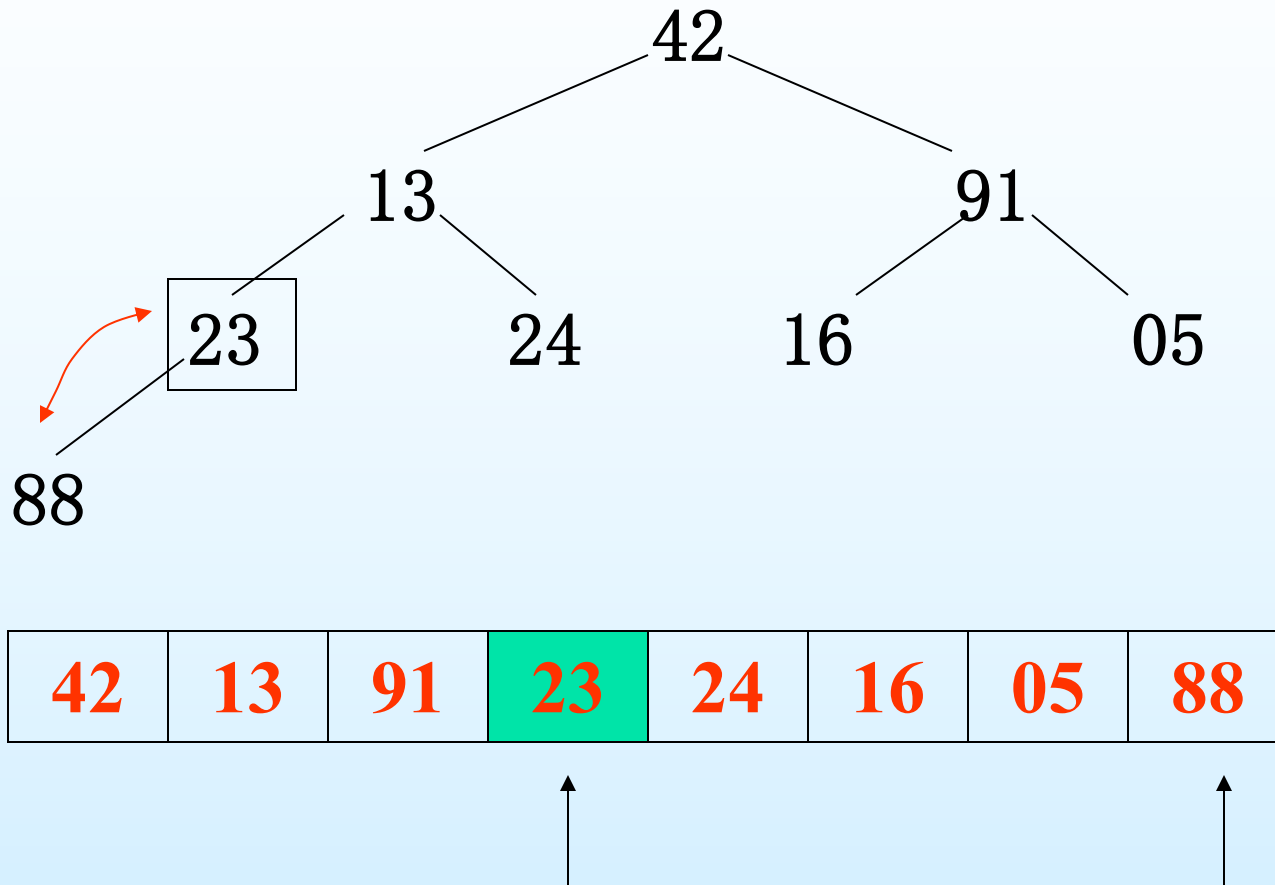
三: 交换之后有可能导致新子树不再是堆, 需要将新子树调整为堆。如此逐层递推下去, 直到调整到树叶为止。

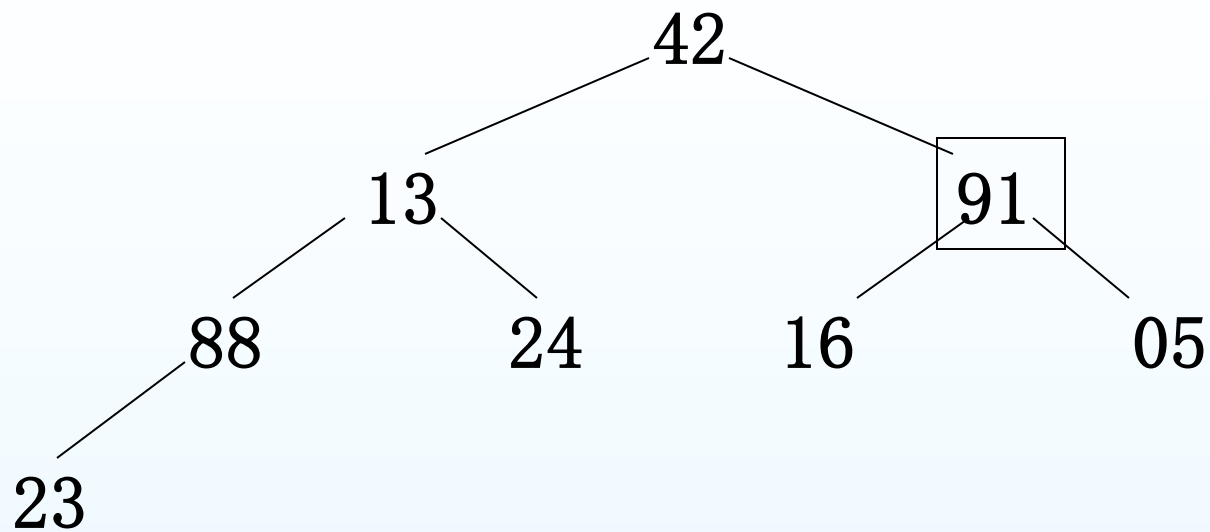




思考:如果最后一个节点不是14, 而是12将如何?

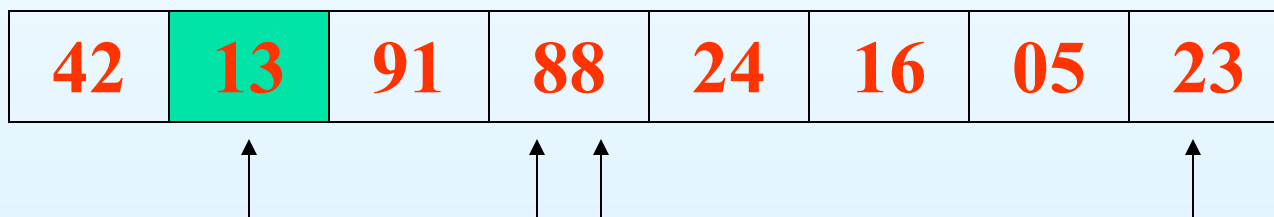
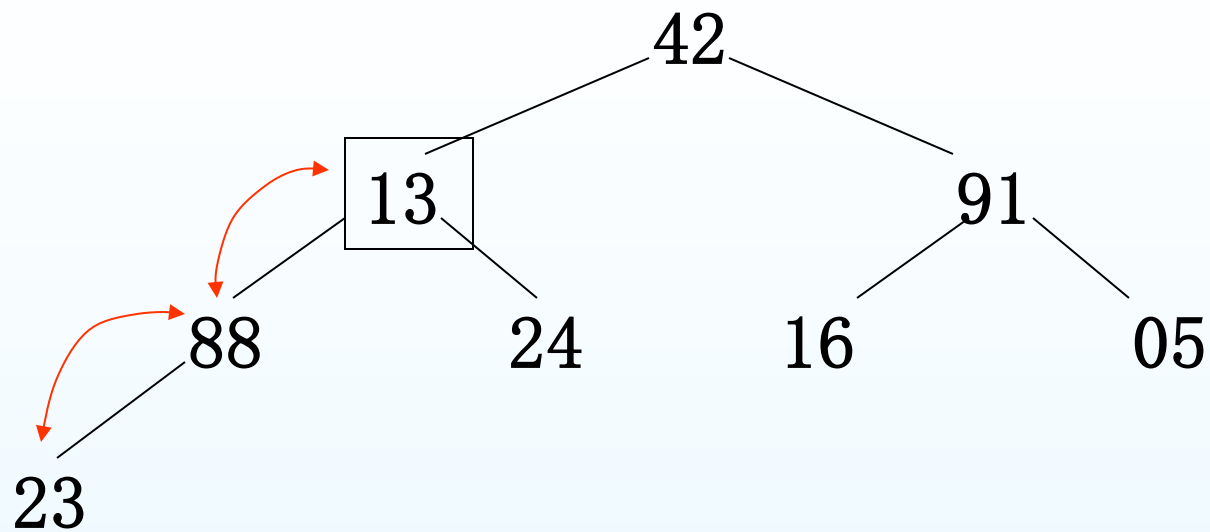
例子：关键字序列为 42, 13, 91, 23, 24, 16, 05, 88, $n=8$, 故从第四个结点开始调整

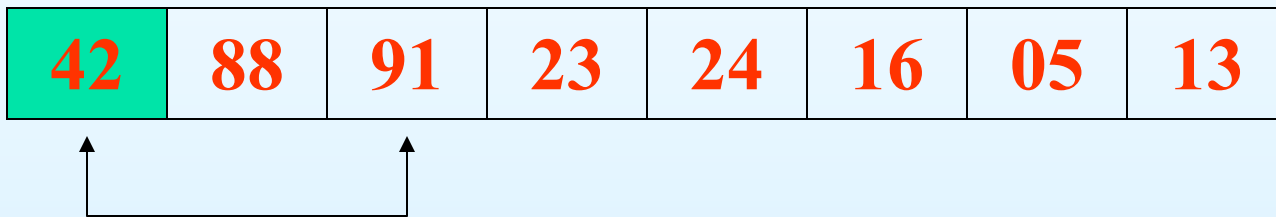
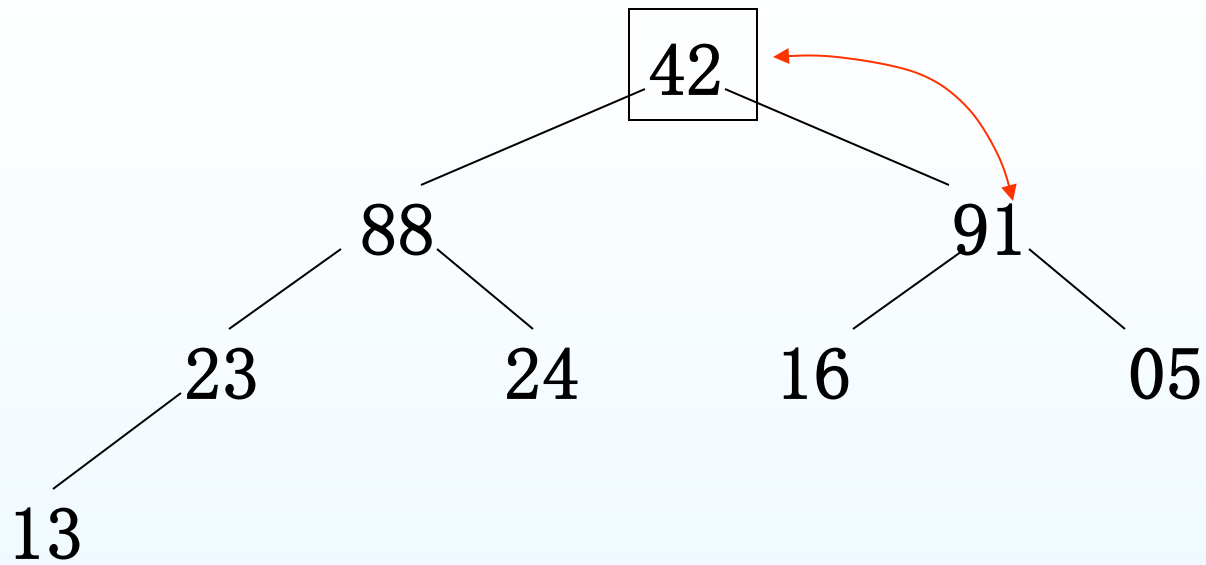


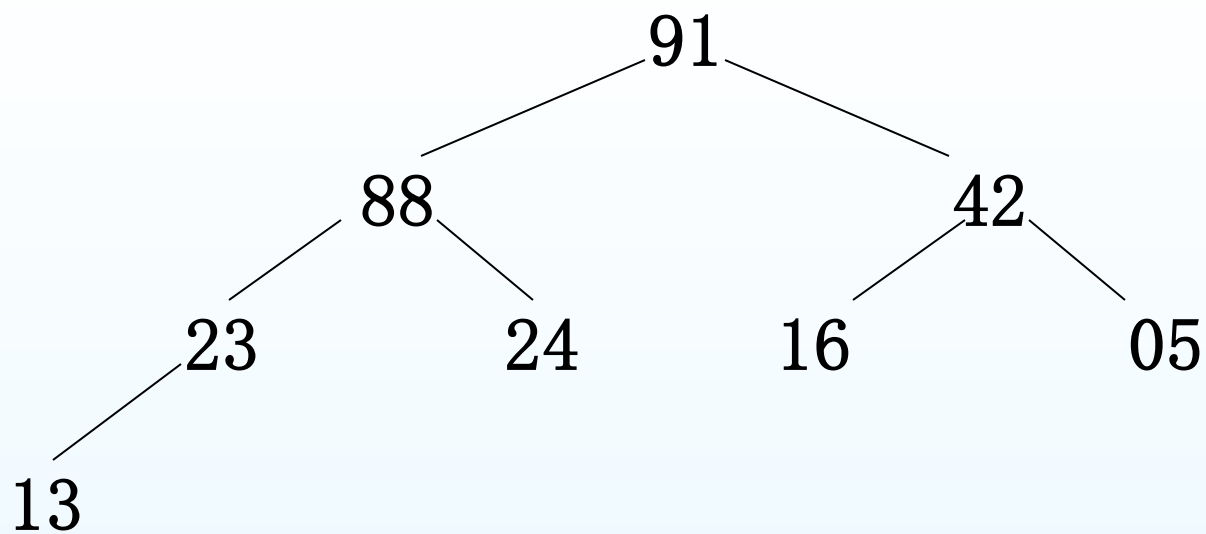


42	13	91	88	24	16	05	23
----	----	----	----	----	----	----	----

不调整







91	88	42	23	24	16	05	13
----	----	----	----	----	----	----	----

建成的堆

```

SIFT(ET h[], int n; int m)
{
    int j;  ET t;  t=h[m];
    j=2 * m;
    while (j <= n) //处理到叶子
    {
        if ((j<n)&&(h[j] <h[j+1]))
            j++; //两颗子树比较
        if (t<h[j]) { //exchange
            h[m]=h[j];
            h[j]=t
            m=j;
            j=2 * m; }
        else break;
    }
}

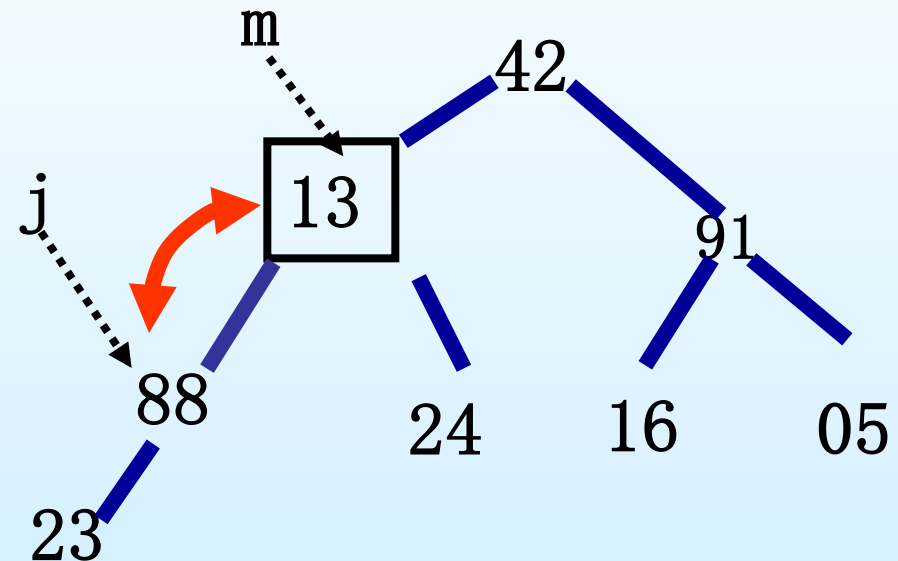
```

$m = 2$

$h[m] = t = 13$

$j = 4 \quad h[j] = 88$

$h[j+1] = 24$



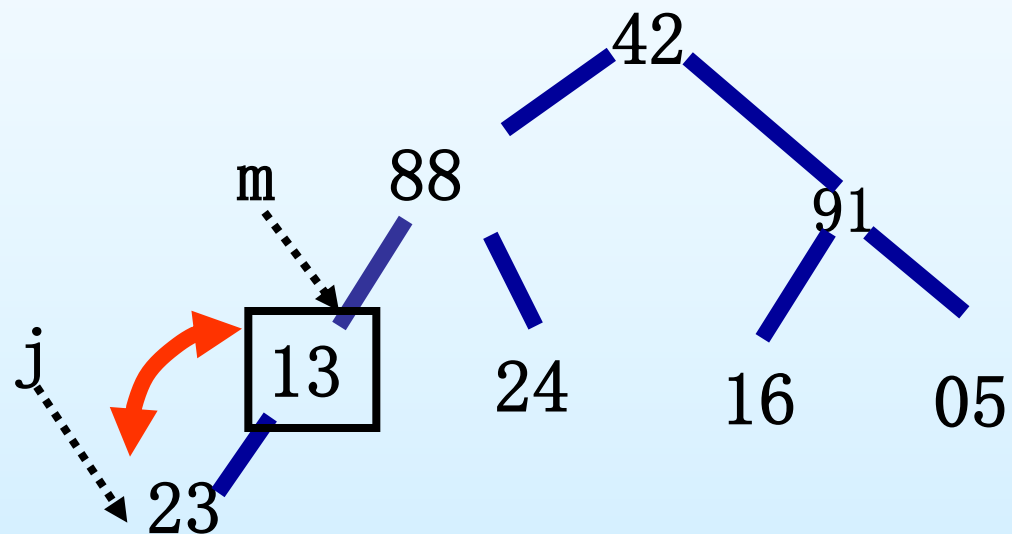
42	13	91	88	24	16	05	23
----	----	----	----	----	----	----	----

```

SIFT(ET h[], int n; int m)
{ int j;  ET t;  t=h[m];
  j=2 * m;
  while (j <= n) //处理到叶子
  { if ((j<n)&&(h[j] <h[j+1]))
    j++;  //两颗子树比较
    if (t<h[j]) { //exchange
      h[m]=h[j];
      h[j]=t
      m=j;
      j=2 * m; }
    else break;
  }
}

```

$m = 4$ $t = 13$
 $h[m] = 13$
 $j = 8$ $h[j] = 23$
 $h[n+1] = 0$



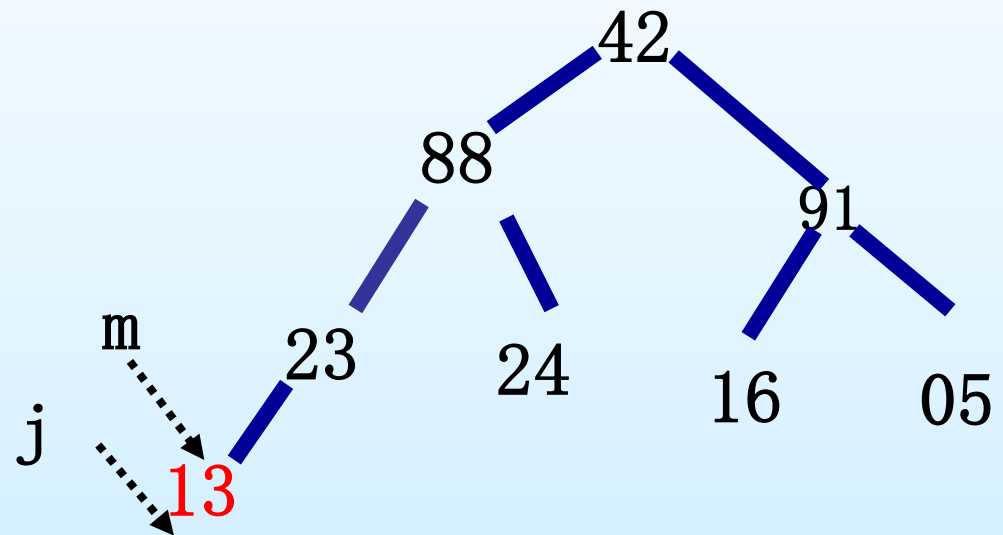
42	88	91	88	24	16	05	23
----	----	----	----	----	----	----	----

```

SIFT(ET h[], int n; int m)
{
    int j;  ET t;  t=h[m];
    j=2 * m;
    while (j<=n) //处理到叶子
    {
        if ((j<n)&&(h[j] <h[j+1]))
            j++; //两颗子树比较
        if (t<h[j]) { //exchange
            h[m]=h[j];
            h[j]=t;
            m=j;
            j=2 * m; }
        else break;
    }
}

```

$m = 8$ $t = 13$
 $h[m] = 23$
 $j = 16$



42	13	91	88	24	16	05	23
----	----	----	----	----	----	----	----

上述算法只是对一个指定的结点进行调整。有了这个算法，将初始的无序区 $R[1]$ 到 $R[n]$ 建成一个大根堆，**如何实现？**

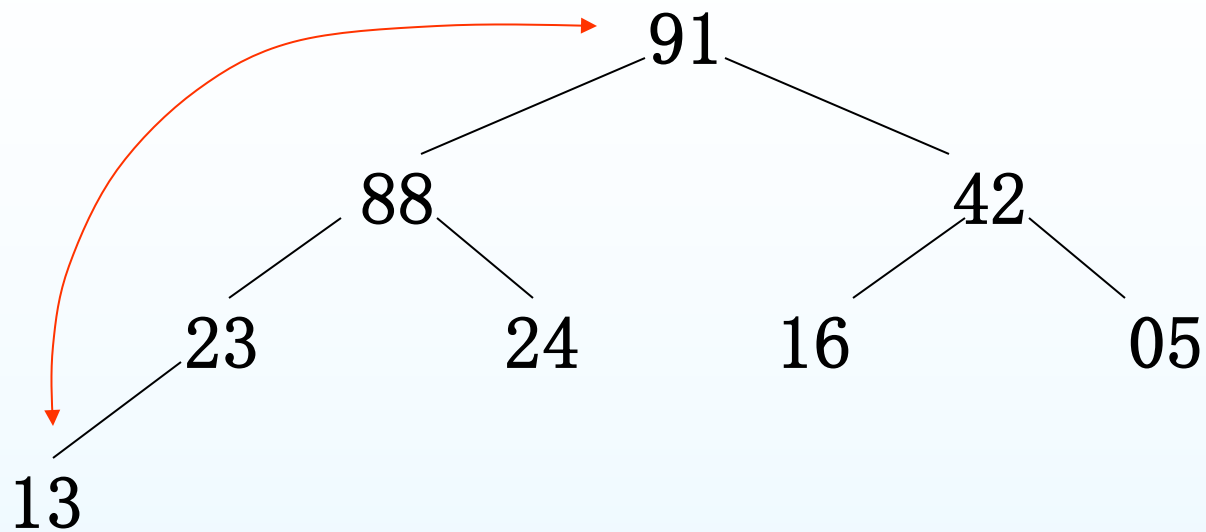
```
for (i = n/2 - 1; i >= 0; i--)
```

```
    SIFT(R, n - 1, i)
```

3.堆排序算法

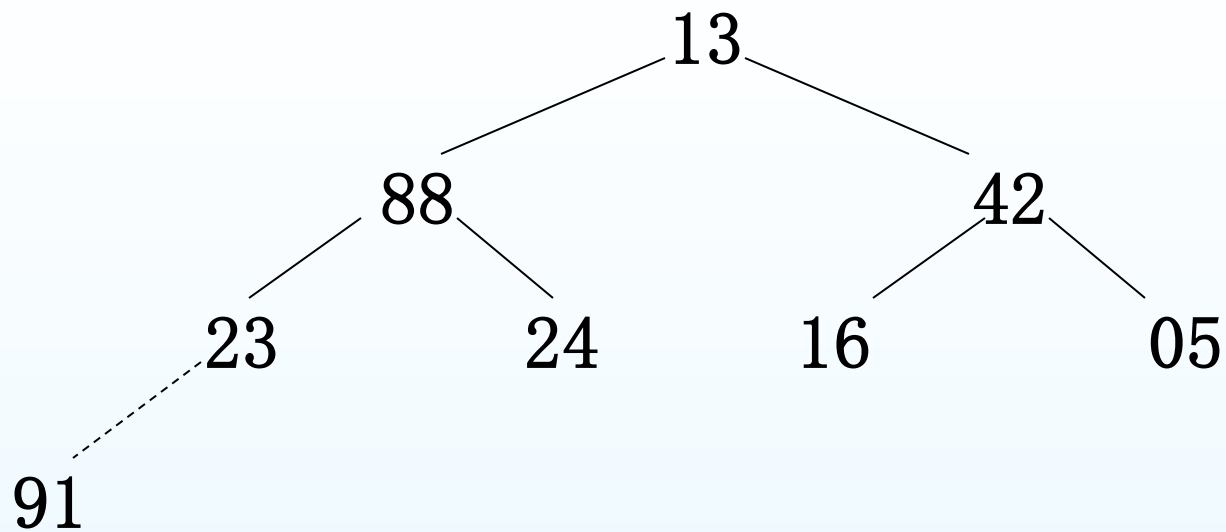
- 由于建堆的结果把关键字最大的记录选到了堆顶的位置，而排序的结果应是升序，如何解决？
- 应该将 $R[0]$ 和 $R[n-1]$ 交换，这就得到了第一趟排序的结果。
- 第二趟排序的操作首先是将无序区 $R[0]$ 到 $R[n-2]$ 调整为堆。这时对于当前堆来说，它的左右子树仍然是堆，所以，可以调用SIFT函数将 $R[0]$ 到 $R[n-2]$ 调整为大根堆，选出最大关键字放入堆顶，然后将堆顶与当前无序区的最后一个记录 $R[n-2]$ 交换，如此反复进行下去。

举例：



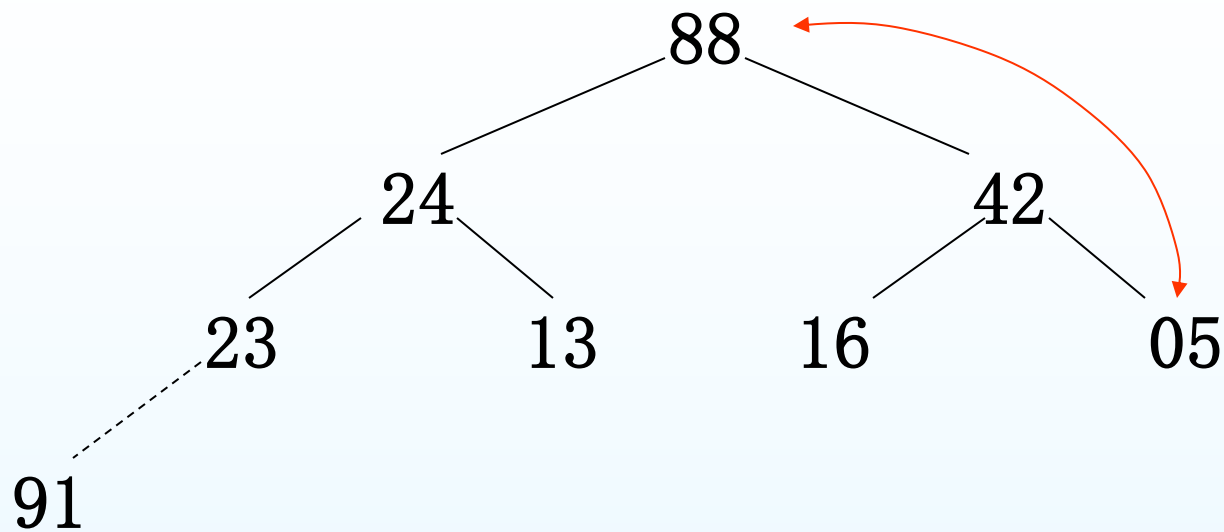
91	88	42	23	24	16	05	13
----	----	----	----	----	----	----	----

(a) 初始堆R[1]到R[8]



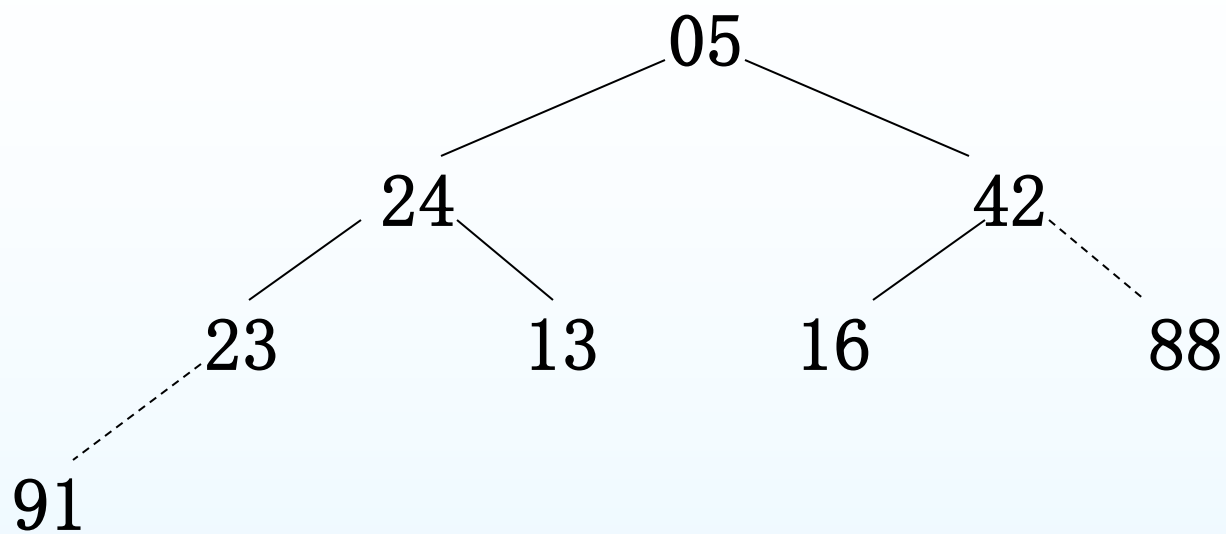
13	88	42	23	24	16	05	91
----	----	----	----	----	----	----	----

(b) 第一趟排序之后



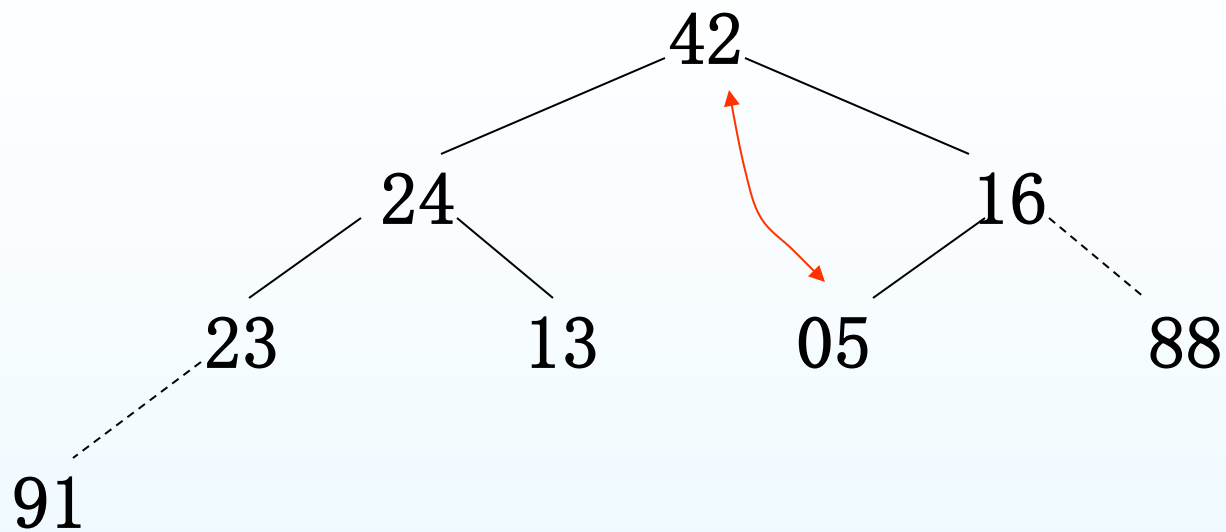
88	24	42	23	13	16	05	91
----	----	----	----	----	----	----	----

(c) 重建的堆R[1]到R[7]



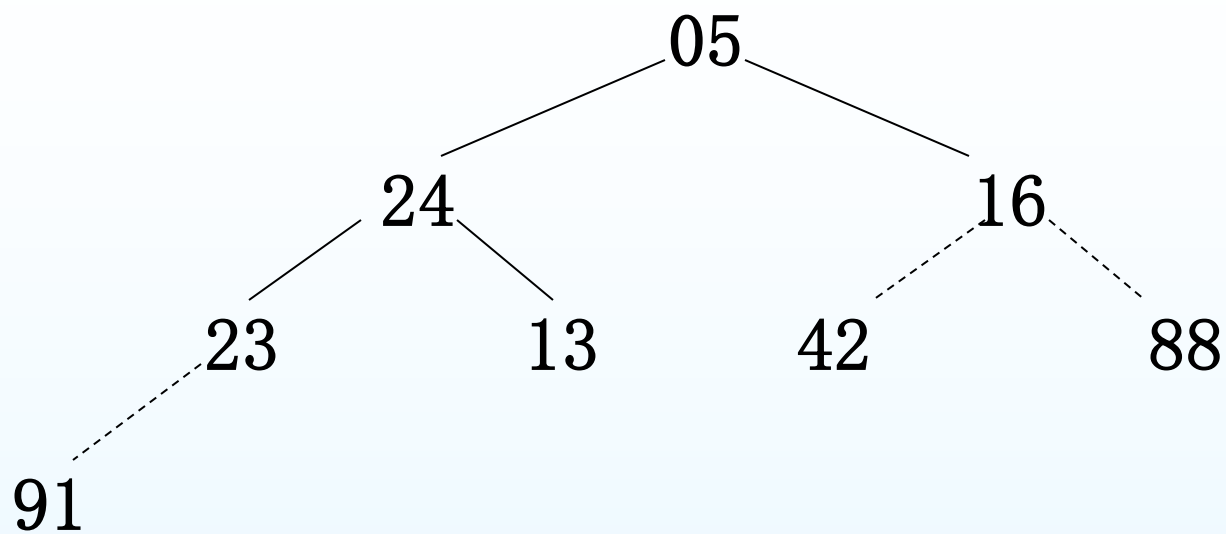
05	24	42	23	13	16	88	91
----	----	----	----	----	----	----	----

(d) 第二趟排序之后



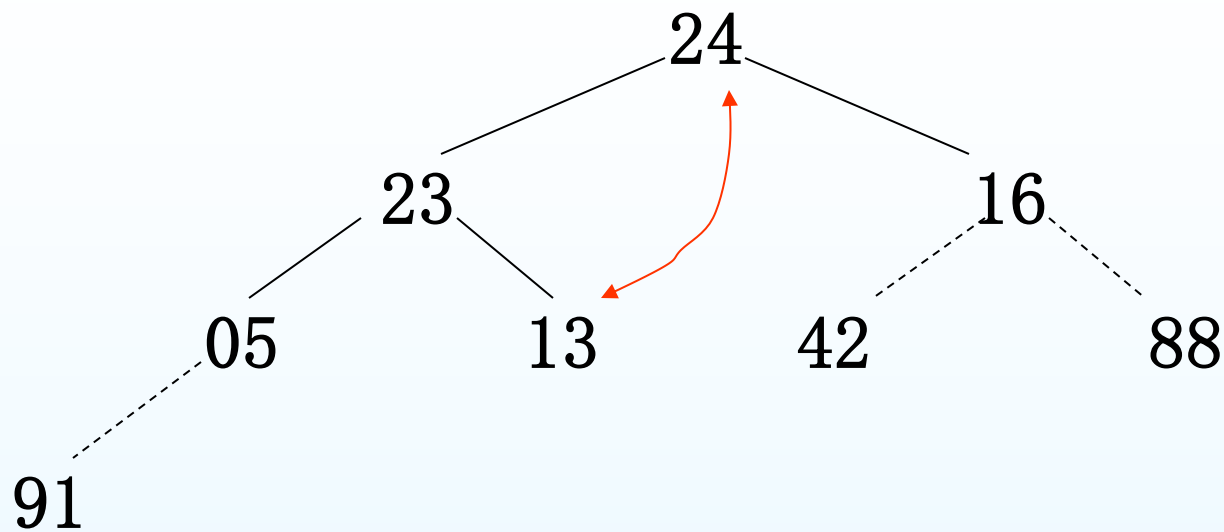
42	24	16	23	13	05	88	91
----	----	----	----	----	----	----	----

(e) 重建的堆R[1]到R[6]



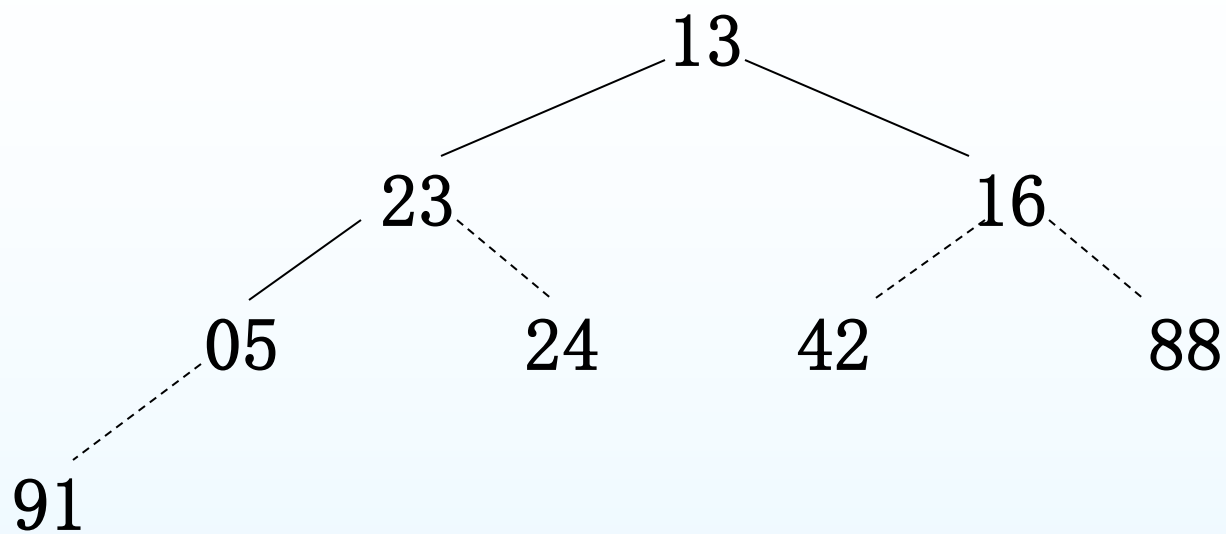
05	24	16	23	13	42	88	91
----	----	----	----	----	----	----	----

(f) 第三趟排序之后



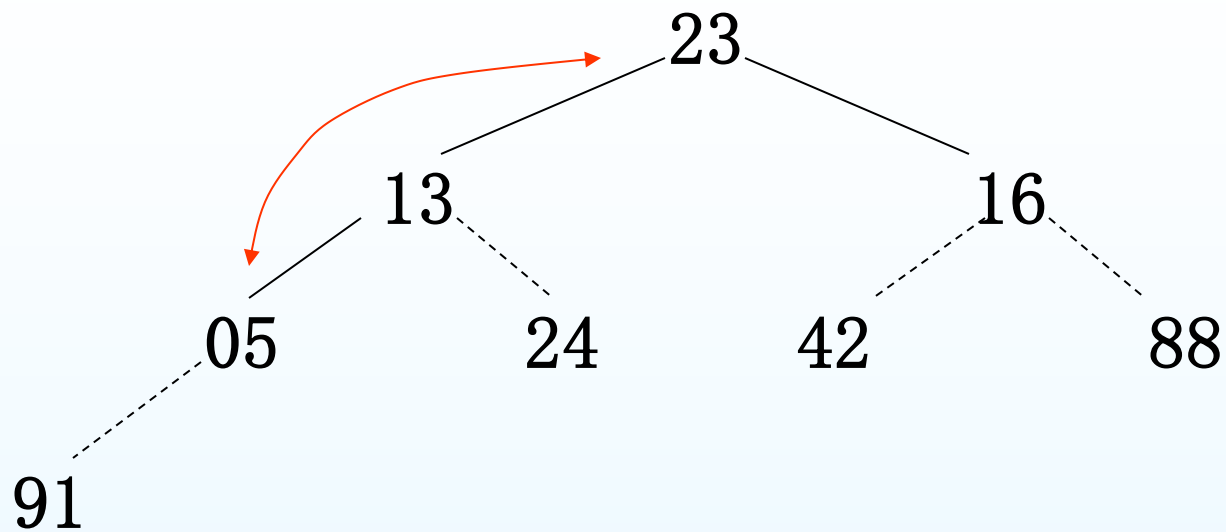
24	23	16	05	13	42	88	91
----	----	----	----	----	----	----	----

(g) 重建的堆R[1]到R[5]



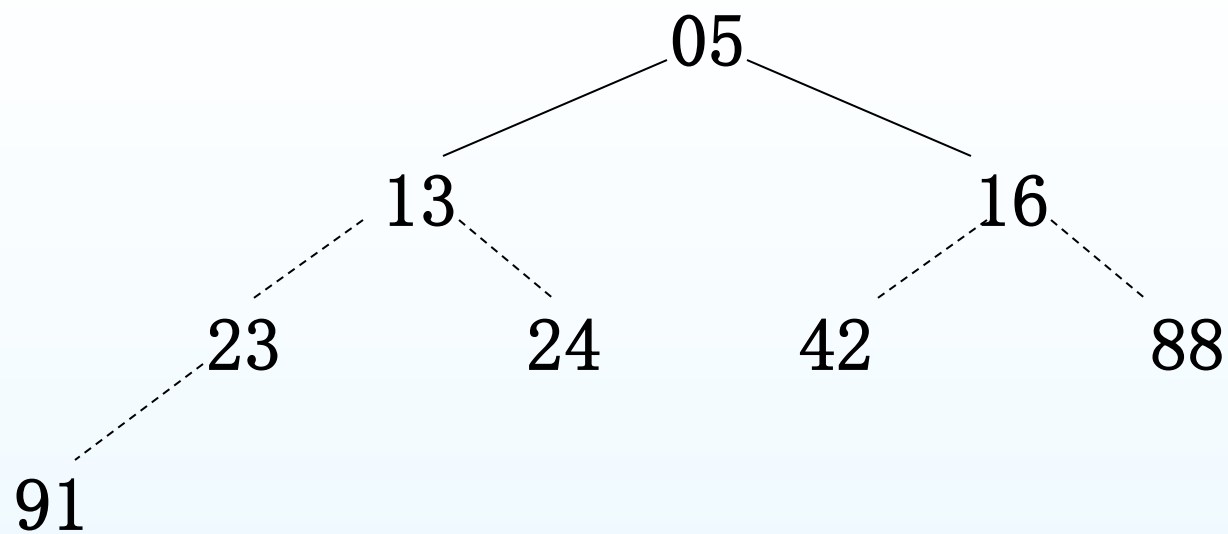
13	23	16	05	24	42	88	91
----	----	----	----	----	----	----	----

(h) 第四趟排序之后



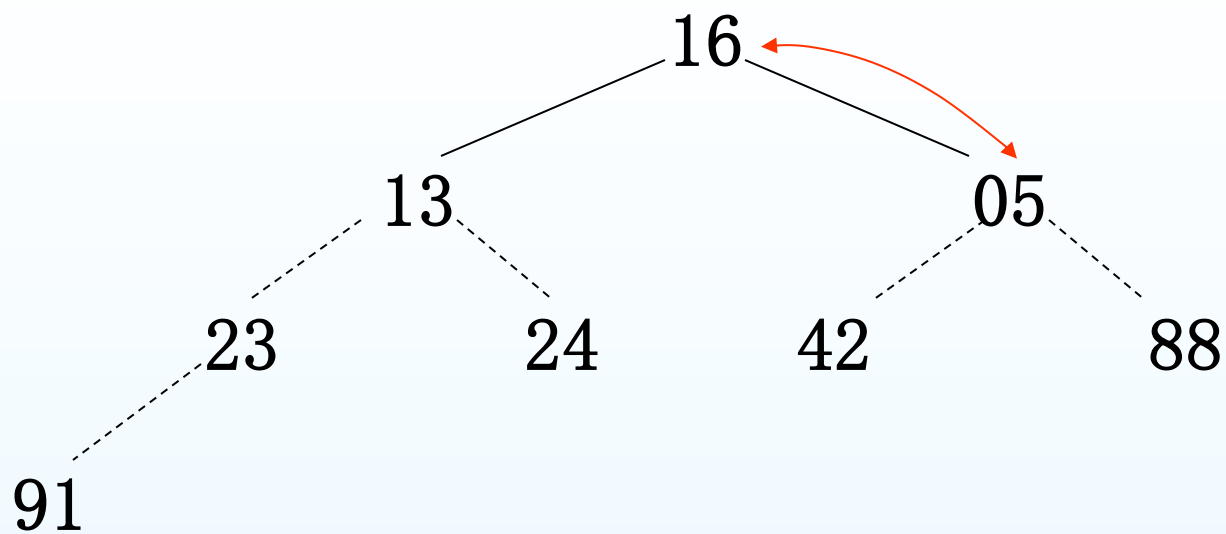
23	13	16	05	24	42	88	91
----	----	----	----	----	----	----	----

(i) 重建的堆R[1]到R[4]



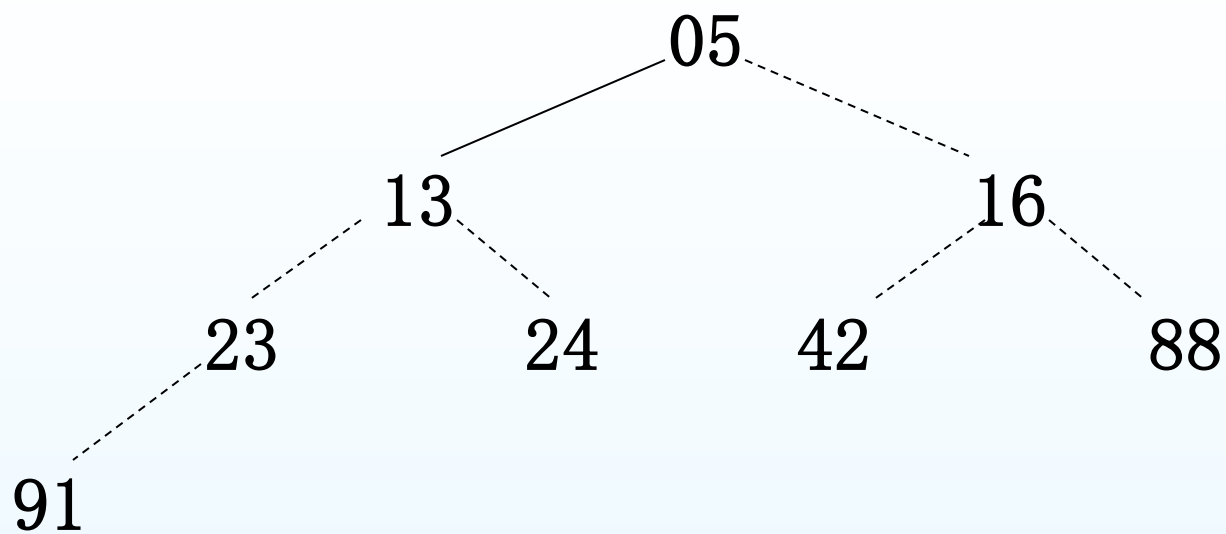
05	13	16	23	24	42	88	91
----	----	----	----	----	----	----	----

(j) 第五趟排序之后



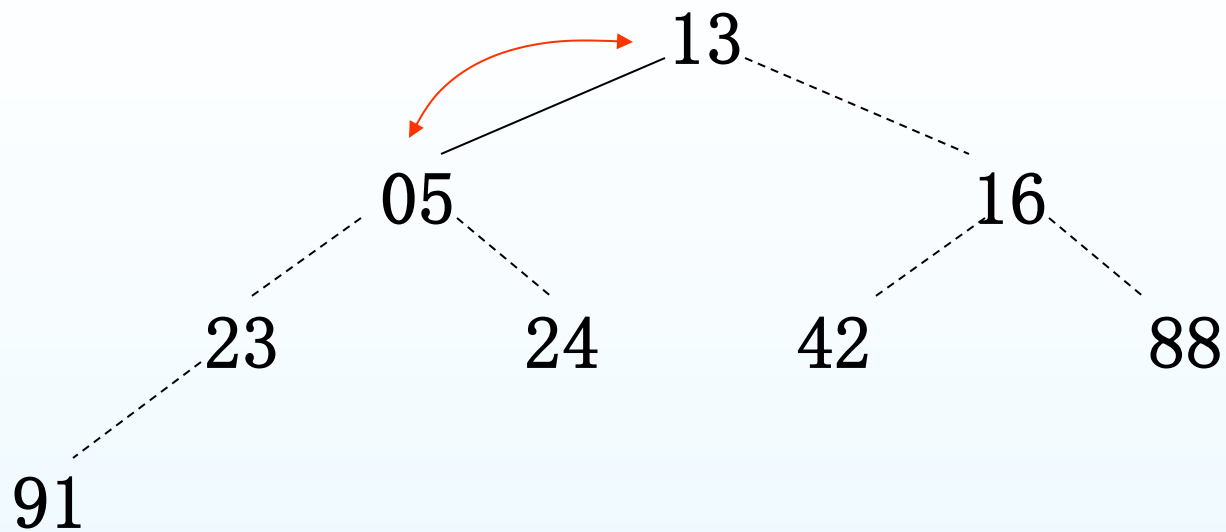
16	13	05	23	24	42	88	91
----	----	----	----	----	----	----	----

(k) 重建的堆R[1]到R[3]



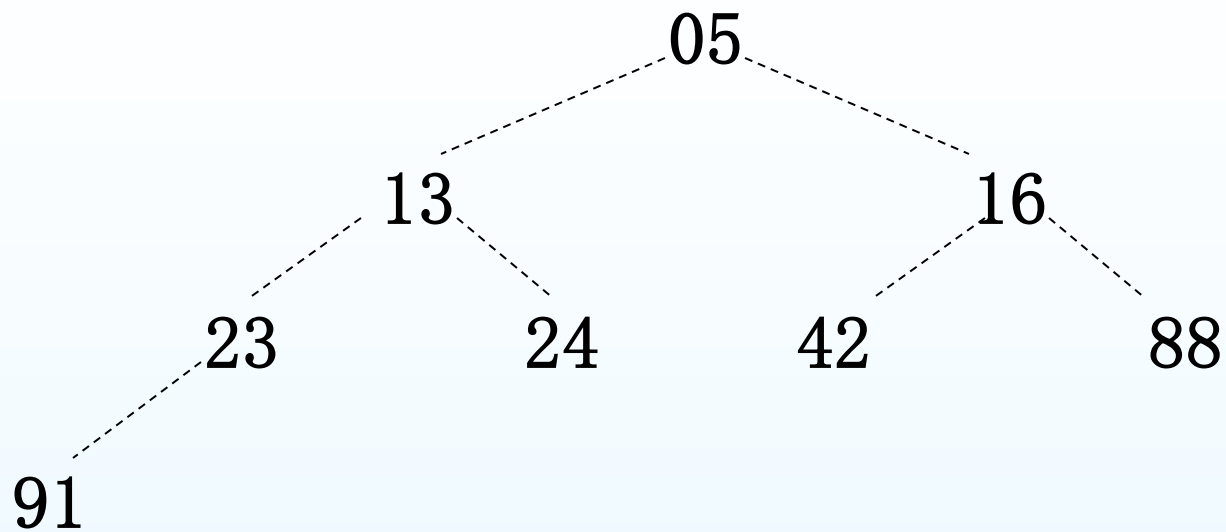
05	13	16	23	24	42	88	91
----	----	----	----	----	----	----	----

(1) 第六趟排序之后



13	05	16	23	24	42	88	91
----	----	----	----	----	----	----	----

(m) 重建的堆R[1]到R[2]



05	13	16	23	24	42	88	91
----	----	----	----	----	----	----	----

(n) 第七趟排序之后

HEAPSORT (ET p[])

```
{  int i; ET t;
    for (i=n/2 -1; i>=1; i--)
        sift(p, n-1, i);
    for (i=n-1; i>=1; i--)
    {
        t=p[0]; p[0]=p[i];
        p[i]=t;
        sift(p, i-1, 0);
    }
}
```

4.堆排序的时间复杂度

- 堆排序的时间复杂度为 $O(n\log_2 n)$
- 空间复杂度为 $O(1)$

堆排序是不稳定的排序方法。

堆排序课堂练习

- 23 33 21 1 24 14 2 26 90 43
- 1) 先建大根堆（写出过程）
- 2) 排序！

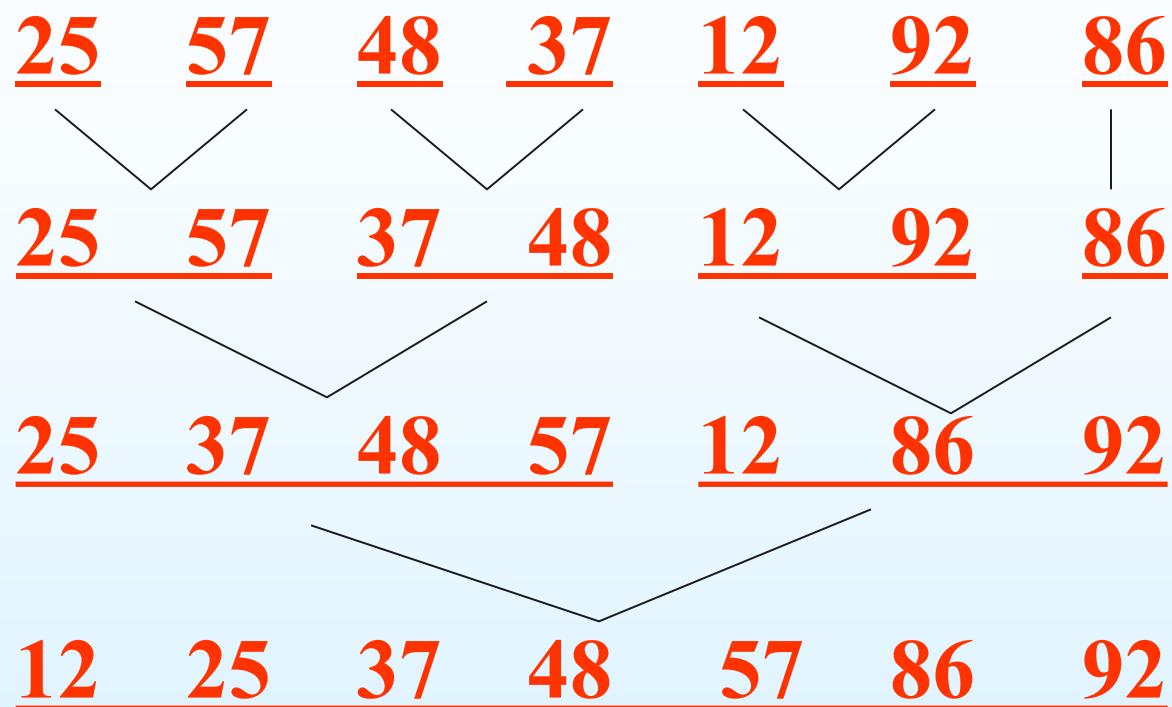
七、归并排序

- **基本原理**：通过对两个有序结点序列的合并来实现排序。
- 所谓**归并**是指将若干个已排好序的部分合并成一个有序的部分。
- 若将两个有序表合并成一个有序表，称为2-路归并。

1.两路归并的基本思想

- (1) 设有两个有序表A和B，对象个数分别为 a_1 和 b_1 ，变量i和j分别是两表的当前指针。
- (2) 设表C是归并后的新有序表，变量k是它的当前指针。
- (3) i和j对A和B遍历时，依次将关键字小的对象放到C中，当A或B遍历结束时，将另一个表的剩余部分照抄到新表中。

两路归并的示例



归并排序就是利用上述归并操作实现排序的。

(1)将待排序列 $R[1]$ 到 $R[n]$ 看成 n 个长度为1的有序子序列，把这些子序列两两归并，便得到 $\text{high}(n/2)$ 个有序的子序列。

(2)然后再把这 $\text{high}(n/2)$ 个有序的子序列两两归并，如此反复，直到最后得到一个长度为 n 的有序序列。

(3)上述每次归并操作，都是将两个子序列归并为一个子序列，这就是“二路归并”，类似地还可以有“三路归并”或“多路归并”。

算法评价

- 空间复杂度为 $O(n)$,
用辅助空间L1、L2、(Swap)
- 时间复杂度为 $O(n \log n)$
- 2-路归并排序算法是稳定的。

八、基数排序

- 多关键字排序技术：多关键字（ K_1, K_2, \dots, K_t ）；
例如：关键字 K_1 小的结点排在前面。如关键字 K_1 相同，则比较关键字 K_2 ，关键字 K_2 小的结点排在前面，依次类推.....

1. 举例


- 假定给定的是 $t = 2$ 位十进制数，存放在数组 B 之中。现要求通过基数排序法将其排序。
- 设置十个口袋，因十进制数分别有数字：0, 1, 2, ..., 9，分别用 B_0 、 B_1 、 B_2 、.....、 B_9 进行标识。
- 执行 $j = 1 \dots t$ (这里 $t = 2$) 次循环，每次进行一次分配动作，一次收集动作。
- 将右起第 j 位数字相同的数放入同一口袋。比如数字为 1 者，则放入口袋 B_1 ，余类推 收集：按 B_0 、 B_1 、 B_2 、..... B_9 的顺序进行收集。

基数排序举例

e.g: B = 5、2、9、7、18、17、52 用基数排序法进行排序。

B = 5、2、9、7、18、17、52



根据  所指向的数，进行分配动作，将其分配到相应的口袋。

口袋

B_0

B_1

B_2

B_3

B_4

B_5

B_6

B_7

B_8


B_9

基数排序举例

e.g: B = 5、2、9、7、18、17、52 用基数排序法进行排序。

B = 5、2、9、7、18、17、52



根据  所指向的数，进行分配动作，将其分配到相应的口袋。

口袋

B_0

B_1

B_2

B_3

B_4

B_5 5

B_6

B_7

B_8

B_9

基数排序举例

e.g: B = 5、2、9、7、18、17、52 用基数排序法进行排序。

B = 5、2、9、7、18、17、52



口袋

B_0

B_1

B_2

B_3

B_4


B_5 5

B_6

B_7

B_8

B_9

根据  所指向的数，进行分配动作，将其分配到相应的口袋。

基数排序举例

e.g: B = 5、2、9、7、18、17、52 用基数排序法进行排序。

B = 5、2、9、7、18、17、52



口袋

B₀

B₁

B₂ 2

B₃

B₄


B₅ 5

B₆

B₇

B₈

B₉

根据  所指向的数，进行分配动作，将其分配到相应的口袋。


基数排序举例

e.g: B = 5、2、9、7、18、17、52 用基数排序法进行排序。

B = 5、2、9、7、18、17、52



口袋

根据  所指向的数，进行分配动作，将其分配到相应的口袋。

B₀

B₁

B₂ 2

B₃

B₄

B₅ 5

B₆

B₇

B₈

B₉


基数排序举例

e.g: B = 5、2、9、7、18、17、52 用基数排序法进行排序。

B = 5、2、9、7、18、17、52



口袋

根据  所指向的数，进行分配动作，将其分配到相应的口袋。

B₀

B₁

B₂ 2

B₃

B₄

B₅ 5

B₆

B₇

B₈

B₉ 9


基数排序举例

e.g: B = 5、2、9、7、18、17、52 用基数排序法进行排序。

B = 5、2、9、7、18、17、52



口袋

根据  所指向的数，进行分配动作，将其分配到相应的口袋。

B_0

B_1

B_2 2

B_3

B_4

B_5 5

B_6

B_7

B_8

B_9 9


基数排序举例

e.g: B = 5、2、9、7、18、17、52 用基数排序法进行排序。

B = 5、2、9、7、18、17、52



口袋

根据  所指向的数，进行分配动作，将其分配到相应的口袋。

B₀

B₁

B₂ 2

B₃

B₄

B₅ 5

B₆

B₇ 7

B₈

B₉ 9


基数排序举例

e.g: B = 5、2、9、7、18、17、52 用基数排序法进行排序。

B = 5、2、9、7、18、17、52



口袋

根据  所指向的数，进行分配动作，将其分配到相应的口袋。

B_0

B_1

B_2 2

B_3

B_4

B_5 5

B_6

B_7 7

B_8

B_9 9


基数排序举例

e.g: B = 5、2、9、7、18、17、52 用基数排序法进行排序。

B = 5、2、9、7、18、17、52



口袋

根据  所指向的数，进行分配动作，将其分配到相应的口袋。

B_0

B_1

B_2 2

B_3

B_4

B_5 5

B_6

B_7 7

B_8 18

B_9 9


基数排序举例

e.g: B = 5、2、9、7、18、17、52 用基数排序法进行排序。

B = 5、2、9、7、18、17、52



口袋

根据  所指向的数，进行分配动作，将其分配到相应的口袋。

B₀

B₁

B₂ 2

B₃

B₄

B₅ 5

B₆

B₇ 7

B₈ 18

B₉ 9


基数排序举例

e.g: B = 5、2、9、7、18、17、52 用基数排序法进行排序。

B = 5、2、9、7、18、17、52



口袋

根据  所指向的数，进行分配动作，将其分配到相应的口袋。

B₀

B₁

B₂ 2

B₃

B₄

B₅ 5

B₆

B₇ 7 17

B₈ 18

B₉ 9


基数排序举例

e.g: B = 5、2、9、7、18、17、52 用基数排序法进行排序。

B = 5、2、9、7、18、17、52



口袋

根据  所指向的数，进行分配动作，将其分配到相应的口袋。

B₀

B₁

B₂ 2

B₃

B₄

B₅ 5

B₆

B₇ 7 17

B₈ 18

B₉ 9


基数排序举例

e.g: B = 5、2、9、7、18、17、52 用基数排序法进行排序。

B = 5、2、9、7、18、17、52



口袋

根据  所指向的数，进行分配动作，将其分配到相应的口袋。

B₀

B₁

B₂ 2 52

B₃

B₄

B₅ 5

B₆

B₇ 7 17

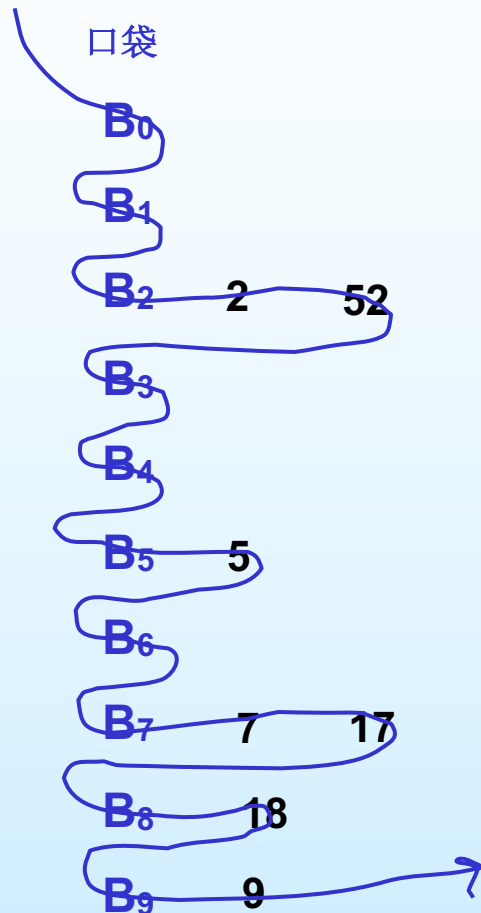
B₈ 18

B₉ 9

基数排序举例

e.g: B = 5、2、9、7、18、17、52 用基数排序法进行排序。

B = 5、2、9、7、18、17、52



分配完毕，按照
箭头所指的方向进行
收集动作。注意：收
集后的序列已经按照
右起第一位（个位数
字）排好序了。

收集后的序列：2、52、5、7、17、18、9

、

基数排序举例

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

B_0

B_1

B_2

B_3

B_4


B_5

B_6

B_7

B_8

B_9

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进行分配。

基数排序举例

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

B₀ 2

B₁

B₂

B₃

B₄


B₅

B₆

B₇

B₈

B₉

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进行分配。

基数排序举例

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

B₀ 2

B₁

B₂

B₃

B₄


B₅

B₆

B₇

B₈

B₉

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进分配。

基数排序举例

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

B₀ 2

B₁

B₂

B₃

B₄


B₅ 52

B₆

B₇

B₈

B₉

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进分配。


基数排序举例

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进分配。

B₀ 2

B₁

B₂

B₃

B₄

B₅ 52

B₆

B₇

B₈

B₉


基数排序举例

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进分配。

B₀ 2 5

B₁

B₂

B₃

B₄

B₅ 52

B₆

B₇

B₈

B₉


基数排序举例

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进分配。

B₀ 2 5

B₁

B₂

B₃

B₄

B₅ 52

B₆

B₇

B₈

B₉


基数排序举例

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进分配。

B₀ 2 5 7

B₁

B₂

B₃

B₄

B₅ 52

B₆

B₇

B₈

B₉


基数排序举例

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进分配。

B₀ 2 5 7

B₁

B₂

B₃

B₄

B₅ 52

B₆

B₇

B₈

B₉


基数排序举例

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进分配。

B₀ 2 5 7

B₁ 17

B₂

B₃

B₄

B₅ 52

B₆

B₇

B₈

B₉


基数排序举例

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进分配。

B₀ 2 5 7

B₁ 17

B₂

B₃

B₄

B₅ 52

B₆

B₇

B₈

B₉


基数排序举例

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进行分配。

B₀ 2 5 7

B₁ 17 18

B₂

B₃

B₄

B₅ 52

B₆

B₇

B₈

B₉


基数排序举例

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进分配。

B₀ 2 5 7

B₁ 17 18

B₂

B₃

B₄

B₅ 52

B₆

B₇

B₈

B₉


基数排序举例

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



口袋

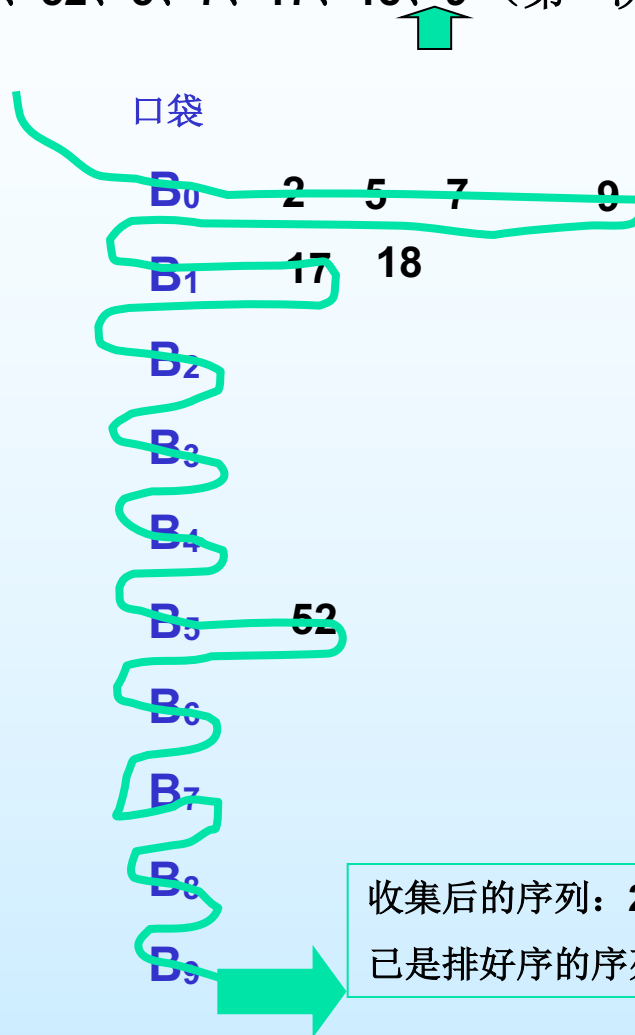
根据  所指向的数，进行分配动作，将其分配到相应的口袋。和第一次不同，这次根据右起第二位数字（十位数字）进分配。

B₀	2	5	7	9
B₁	17	18		
B₂				
B₃				
B₄				
B₅	52			
B₆				
B₇				
B₈				
B₉				

基数排序举例

e.g: B = 5 、 2、 9、 7、 18、 17、 52 用基数排序法进行排序。

B = 2、 52、 5、 7、 17、 18、 9 （第一次收集的结果）



分配完毕，按照箭头所指的方向进行第二次收集动作。注意：收集后的序列已经按照右起第一位（个位数字）、右起第二位（十位数字）排好序了。

收集后的序列：2、5、7、9、17、18、52
已是排好序的序列。

性能分析

· 空间：采用顺序分配，显然不合适。由于每个口袋都有可能存放所有的待排序的整数。所以，额外空间的需求为 $10n$ ，太大了。采用链接分配是合理的。额外空间的需求为 n ，通常再增加指向每个口袋的首尾指针就可以了。在一般情况下，设每个键字的取值范围为 rd ，首尾指针共计 $2 \times rd$ 个，总的空间为 $O(n + 2 \times rd)$ 。

· 时间：上例中每个数计有 $t = 2$ 位，因此执行 $t = 2$ 次分配和收集就可以了。在一般情况下，每个结点有 d 位关键字，必须执行 $t = d$ 次分配和收集操作。

分配的代价： $O(n)$

收集的代价： $O(rd)$

总的代价为： $O(d \times (n + rd))$

课堂练习

- 23,44,55,45,21,124,115,7,129,99

3.4 二叉排序树及其查找

一、定义

所谓二叉排序树是指满足下列条件的二叉树：

- (1) 左子树上的所有结点值均小于根结点值。
- (2) 右子树上的所有结点值均不小于根结点值。
- (3) 左、右子树也满足上述两个条件

二、 二叉排序树的特性

- ❖ 二叉排序树有一个**重要特性**：**中序遍历**二叉排序树可以得到有序序列。因此，由无序序列构造二叉排序树实际上就将一个无序序列变成了有序序列。
- ❖ 另外，由于在二叉排序树中插入的新结点都是叶子结点，因此，在对二叉排序树进行插入运算时，不需移动其他结点，而只需改动插入位置上的叶子结点指针即可。

三、二叉排序树的构造

依次读入给定序列中的每一个元素，然后进行如下处理：

（1）若当前的二叉排序树为空，则读入的元素为根结点。

（2）若读入的元素值小于根结点值，则将元素插入到左子树中。

（3）若读入的元素值不小于根结点值，则将元素插入到右子树中。

无论是插入到左子树还是右子树，都是同样按照上述方法处理。

四、二叉排序树的删除

为了删除一个元素，首先要找到被删除元素所在的结点p和它的父结点q，然后分以下3种情况进行处理：

(1) p为叶子结点：直接删除，修改父结点指针。

(2) p为单支树：将P的子树链到q上。

(3) p的左右子树均不空：

■删除节点p左子树的最右边的元素替代之，相当于用前继节点替代。

■删除节点p右子树的最左边的元素替代之，相当于用后继节点替代

推荐：二叉排序树的删除原理及动画演示

http://student.zjzk.cn/course_ware/data_structure/web/chazhao/chazhao9.3.1.3.htm

五、二叉排序树查找算法

算法描述:

- 输入一个值，在该树中查找，若找到输出该结点值；否则，显示查找失败。
- 与根比，相等，查找成功，
- 比根小，在左子树查
- 比根大，在右子树查
- 查左右子树时按同样的方法查

```
struct tree *search_btree(struct tree * root, char key)
{  if (!root)
    { printf( "Empty tree" ); return root; }
  while(root->info!=key) /* 查找key 的循环 */
  {  if(key<root->info)
      root=root->left; /* 沿左路查找 */
    else
      root=root->right;
    if(root==0)
    { printf( "Search Failure" );
      break ;
    }
  } /* while(root->info!=key) */
```


实际测试结果

OS: winxp, Compiler: vc8, CPU: Intel T7200, Memory: 2G

不同数组长度下调用6种排序1000次所需时间(秒)

length	shell	quick	merge	insert	select	bubble
100	0.0141	0.359	1.875	0.204	0.313	0.421
1000	0.218	0.578	2.204	1.672	2.265	4
5000	1.484	3.25	14.14	41.392	63.656	101.703
10000	3.1	7.8	23.5	253.1	165.6	415.7
50000	21.8	40.6	121.9	411.88	6353.1	11648.5
100000	53.1	89	228.1	16465.7	25381.2	44250

结论:

数组长度不大的情况下不宜使用归并排序, 其它排序差别不大。

数组长度很大的情况下shell最快, Quick其次, 冒泡最慢。

作业

- **P233 3.9**（按下面要求进行排序）
- 数据（1）
- 1）快速排序
- 2）希尔排序
- 3）使用堆排序
- 写出中间步骤和结果