

第七章 内部排序

基本内容:

- 排序的基本概念
- 插入排序
 - 直接插入
 - 折半插入
 - 2-路插入
 - 表插入
 - Shell排序

重点：各种排序方法的思想、算法、时间效率度量

- 选择排序
 - 直接选择
 - 堆排序
- 交换排序
 - 起泡排序
 - 快速排序
- 分配排序 (**)
 - 基数排序
- 归并排序
 - 二路归并

8.1 基本概念

1. 排序的定义:

假设含 n 个记录的序列为 $\{R_1, R_2, \dots, R_n\}$, 其相应的排序码序列为 $\{K_1, K_2, \dots, K_n\}$, 需确定 $1, 2, \dots, n$ 的一种排列 p_1, p_2, \dots, p_n , 使其相应的关键字满足如下的非递减（或非递增）关系:

$K_{p_1} \leq K_{p_2} \leq \dots \leq K_{p_n}$, 使序列成为一个按关键字有序的序列:

$R_{p_1} \leq R_{p_2} \leq \dots \leq R_{p_n}$, 这样的操作称为**排序**。

由小到大: 正排序; 由大到小: 逆排序。

排序码可以是关键码, 也可以是非关键码。

2. 稳定排序法和不稳定排序法:

在待排序的文件中，如果存在多个排序码相同的记录，经过排序后记录的相对次序保持不变，则这种排序方法称为是“稳定的”，否则是“不稳定的”。

3. 排序中的基本操作: 排序码大小比较+记录移动

4. 排序评价标准: 执行时间(比较次数+移动次数)

所需附加空间

算法复杂程度

五类（11种）：

插入排序(5)、选择排序(2)、交换排序(2)、分配排序(1)和归并排序(1)

8.2 插入排序

思想：每步将一个待排序的记录按其排序码大小插入到**前面已排序**的表中适当位置，直到全部插入完为止。

关键问题：找到正确的位置，完成插入。

1. 直接插入排序：

(1) 思想：依次将待排序的记录逐一地按照其排序码的大小插入到一个已经排好序的有序序列中，得到一个新的有序序列（长度加1），直到所有的记录插入完为止。

初始有序表（只包含第一个记录）。每趟排序插入一个记录， n 个记录需要 $n-1$ 趟直接插入排序。

找位置：顺序检索

插入：记录后移，腾出位置，插入。

(2) 示例: {23, 11, 55, 97, 19, 80}

第一趟: {23}, [起始只有一个记录]

{11, 23}

第二趟: {11, 23},

{11, 23, 55}

第三趟: {11, 23, 55},

{11, 23, 55, 97}

第四趟: {11, 23, 55, 97},

{11, 19, 23, 55, 97}

第五趟: {11, 19, 23, 55, 97},

{11, 19, 23, 55, 80, 97}

11

55

97

19

80



(3) 算法实现:

待排序记录集结构定义（顺序存储）：

```
typedef int KeyType;  
typedef int DataType;  
typedef struct  
{  
    KeyType    key;    /* 排序码字段 */  
    DataType  info;   /* 记录的其它字段 */  
}RecordNode;  
typedef struct  
{  
    RecordNode record[MAXNUM];  
    int         n;     /* 记录个数, n<MAXNUM */  
}SortObject;
```

```
void InsertSort(SortObject *pv)
{  int i, j;
   RecordNode temp;

   for (i = 1; i < pv->n; i++)
   {  temp = pv->record[i];
      j = i-1;
      /* 从后往前顺序检索，找插入位置 */
      while ((temp.key < pv->record[j].key) && j >= 0)
      {
         pv->record[j+1] = pv->record[j];
         j--;
      }
      /* 需要插入吗？ */
      if (j != i-1)  pv->record[j+1] = temp;
   }
}
```

(4) 时间效率分析:

第*i*趟排序: 比较次数 C_i 最多为*i*次 (正确位置是第一个记录),
最少为1次 (正确位置是最后记录);

移动次数 M_i 包含temp复制一次,
最多为*i*+1次 (正确位置是第一个记录),
最少为1次 (正确位置是最后记录)。

n-1趟总的:

最少比较次数: $C_{\min} = n-1$

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

平均比较次数:
$$\begin{aligned}\bar{C} &= \sum_{i=1}^{n-1} \bar{C}_i = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} P_j C_j = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} \frac{1}{i} (j+1) = \sum_{i=1}^{n-1} \frac{i+1}{2} \\ &= \frac{n-1}{2} + \frac{1}{2} \sum_{i=1}^{n-1} i = \frac{n-1}{2} + \frac{n(n-1)}{4} = O(n^2)\end{aligned}$$

最小移动次数: $M_{\min} = n-1$

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

平均移动次数: $\bar{M} = O(n^2)$

时间复杂度: $T(n)=O(n^2)$, 辅助空间1个记录单位: Temp, 稳定的排序。

直接插入排序算法简单，容易实现。当记录数目较少时或待排序的记录序列基本有序情况下，直接插入排序是一种很好的排序方法。但从直接插入排序算法时间效率分析可以看出：当待排序记录数 n 较大时，记录的比较、移动次数都非常大。如何减少记录的比较次数和移动次数？

后面的几种插入排序方法，都是从减少比较次数和移动次数方面改进直接插入排序。

2. 折半插入排序：

（1）思想：找插入位置时，采用折半检索提高检索效率，减少比较次数。

找位置：折半检索（有序顺序表）

插入：记录后移，腾出位置，插入。

(2) 示例: {23, 11, 55, 97, 19, 80}

第五趟: {11, 19, 23, 55, 97}, 80

{11, 19, 23, 55, 97}

{11, 19, 23, 55, 80, 97}

(3) 算法实现:

```
void BinSort(SortObject *pv)
```

```
{ int i, j, left, mid, right;
```

```
    RecordNode temp;
```

```
for (i=1; i < pv->n; i++)  
{  temp = pv->record[i];  
  left = 0; right = i-1;          /* 折半检索确定位置 */  
  while (left <= right)  
  {  mid = (left+right)/2;  
    if (temp.key < pv->record[mid].key) right=mid-1;  
    else left =mid+1;  
  }  
  for (j = i-1; j >= left; j--) /* 记录移动 */  
    pv->record[j+1]=pv->record[j];  
  if (left != i)  pv->record[left] = temp;  
}  
}
```

(4) 时间效率分析:

折半插入排序的比较次数与待排序记录的初始状态无关，仅依赖于记录的个数. (总比较次数: $n\log_2 n$, p250推导)

由于检索时采用了折半方法，因此减少了记录的比较次数（平均性能），但并没有减少记录的移动次数。因此，总的**时间复杂度**仍然为 $O(n^2)$ 。

辅助空间：1个记录（temp）

稳定性：在检索时采用 $left > right$ 结束，left、right的修改原则是：temp.key < pv->record[mid].key，保证排序是稳定的。

3. 2-路插入排序:

(1) 思想: 在折半插入排序的基础上, 花费 n 个记录的存储空间, 减少记录的移动次数。

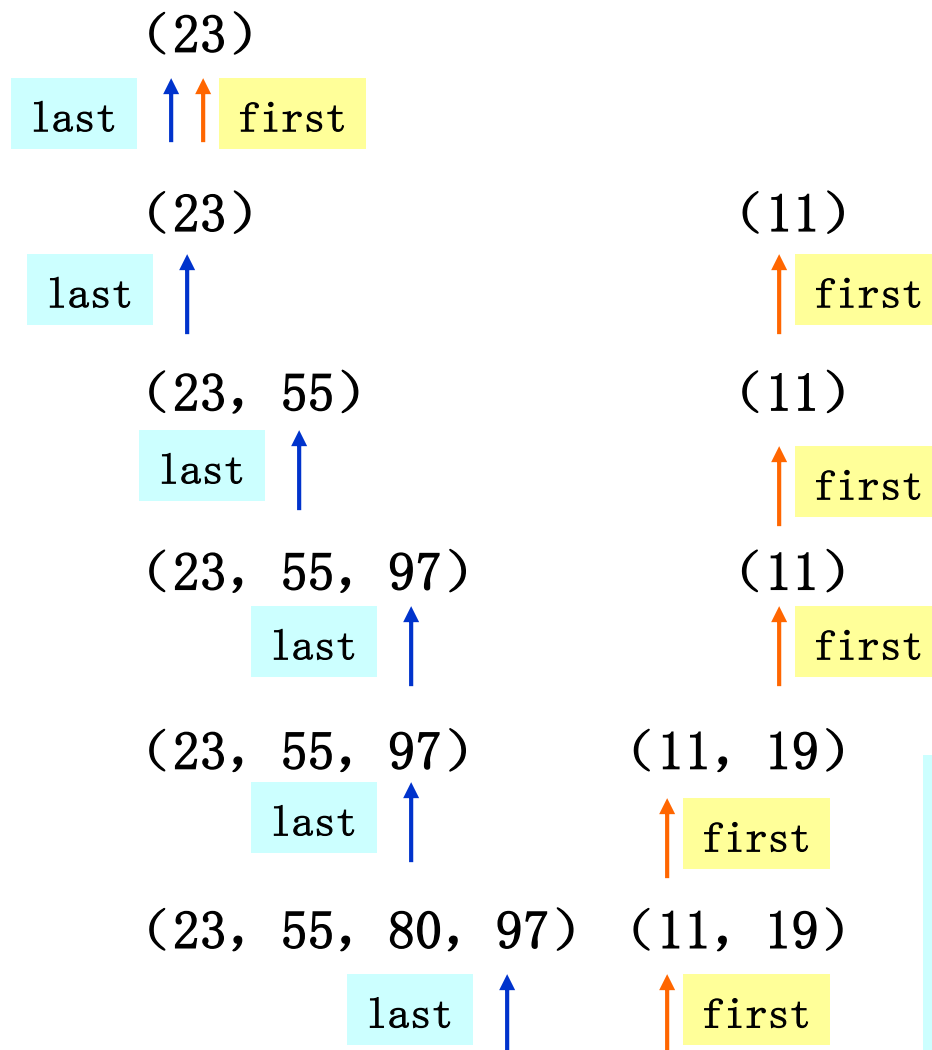
具体做法: 另设一个与 R 同类型的数组 D , 首先将 $R[1]$ 赋给 $D[1]$, 并将 $D[1]$ 看成是在排好序的序列中处于中间位置的记录。然后从 R 的第二个记录开始, 比较其关键字值与 $D[1]$ 的关键字值的大小: 如果小于 $D[1]$, 则插入到 $D[1]$ 前面的有序表中, 否则插入到 $D[1]$ 后面的有序表中。

【小于 $D[1]$ 的有序表】 【 $D[1]$, 大于 $D[1]$ 的有序表】

找位置: $D[1]$ 比较, 决定前后; 然后在前或后有序表中折半检索有序顺序表。

插入: 前或后有序表中记录后移, 腾出位置, 插入。

(2) 示例: {23, 11, 55, 97, 19, 80}



First指向前表的第一个记录; **Last**指向后表的最后记录。从**First**->**Last**构成有序表

(3) 算法实现:

D为循环数组，另设两个指针(first,last)。课后自己实现。

(4) 时间效率分析:

与直接插入排序比较：比较、移动次数都减少；

与折半插入排序比较：比较次数不变，减少了移动次数
(前或后有序表中移动)：大约为 $n^2/8$ 。

特殊情况：当R[1]为待排序序列中最小或最大时，

移动次数达到最大（前后表变为一个，与折半相同）。

减少移动次数的代价：n个记录辅助空间，
空间复杂度为 $O(n)$ 。

时间效率： $O(n^2)$

顺序表存储时，记录的插入必然通过记录的移动完成。

因此，避免移动的办法是：改变存储结构，通过指针修改完成记录的插入。



4. 表插入排序:

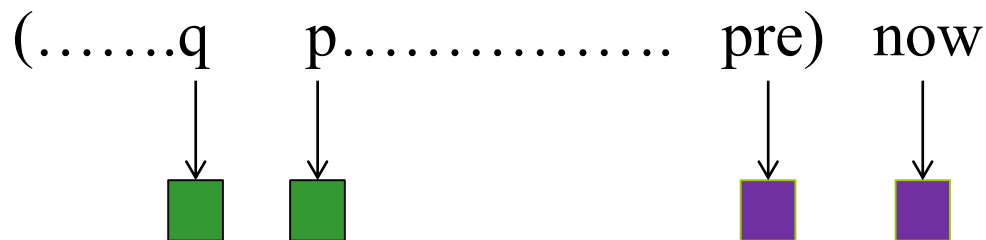
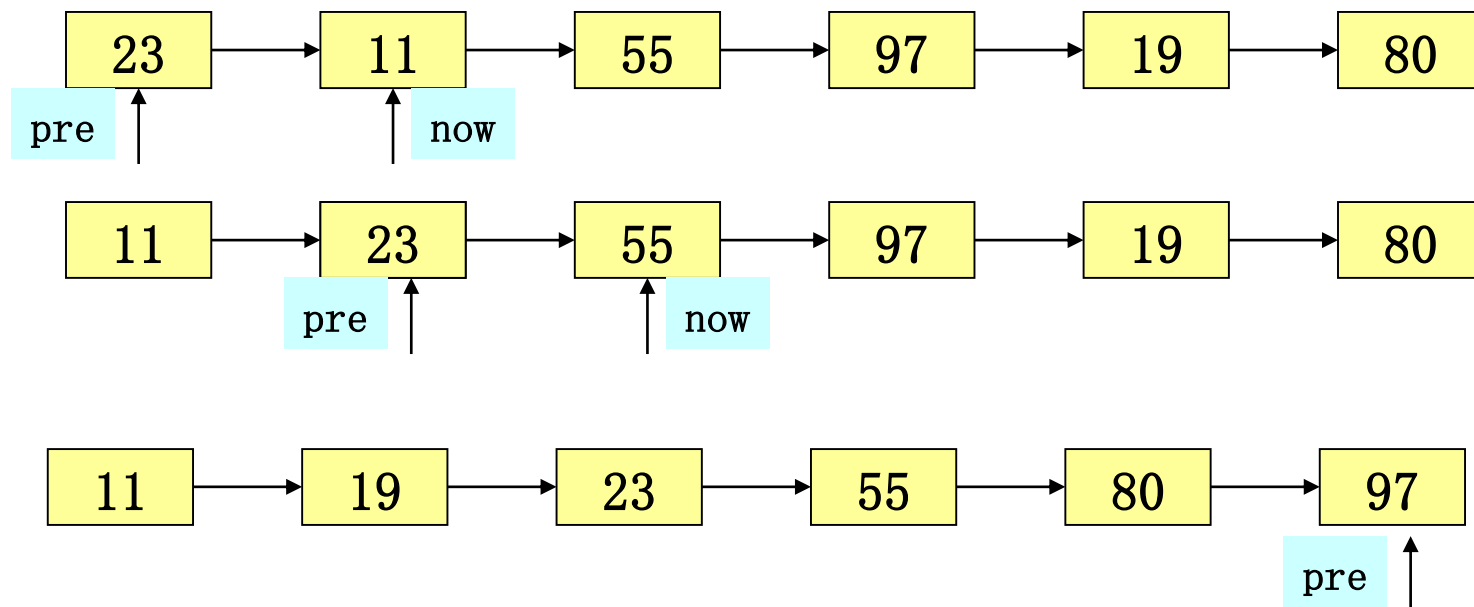
(1) 思想: 将待排序的记录序列采用链表存储, 通过指针修改完成记录的插入, 避免记录移动。

第 i 趟表插入: 插入 R_i 前, R_0 到 R_{i-1} 已经有序。因此, 确定 R_i 的位置只能从表头开始顺序检索。然后通过前后记录指针修改完成 R_i 的插入。

找位置: 顺序检索 (链表如何检索? 从头顺链一个一个比较)

插入 : 修改指针。

(2) 示例: {23, 11, 55, 97, 19, 80}



(3) 算法实现:

struct Node

```
{  KeyType Key;          /* 关键字 */
    DataType info;       /* 其它信息 */
    struct Node *next;    /* 指针 */
};
```

typedef struct Node ListNode, *LinkList;

void ListSort(LinkList plist)

```
{  ListNode *now, *pre, *p, *q, *head;
    head = plist;                                /* 表头 */
    pre = head->next;  now = pre->next;          /* 初始化 */
    if (pre == NULL || now == NULL) return; /*不需要排序 */
```

```

while (now != NULL)
{
    q = head;  p = head->next;
    while (p != now && p->key <= now->key)
    {
        /* 找位置 */
        q = p;  p = p->next;
    }
    if (p == now)
    {
        /* 位置不变, 继续 */
        pre = pre->next;  now = pre->next;  continue;
    }
    pre->next = now->next;
    q->next = now;
    now->next = p;
    /* now记录脱链 */
    /* 插入now */
    /* 下一个, 继续 */
}
}

```

(4) 时间效率分析:

第*i*趟排序: 最多比较次数*i*次, 最少比较次数1次。

n-1趟总的比较次数:

$$\begin{aligned} \text{最多: } & \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \\ \text{最少: } & n-1 \end{aligned}$$

记录移动次数: 0

时间效率: $O(n^2)$

辅助空间: $O(n)$ [指针]

稳定性: $p.\text{key} \leq \text{now} \rightarrow \text{key}$ 保证稳定的排序。

5. Shell(希尔)排序:

(1) 思想: 缩小增量法, D.L.Shell (1959年) 提出对直接插入排序改进。

直接插入排序中, 当初始序列为逆序时, 时间效率最差。若初始序列**基本有序**时, 则大多数记录不需要插入, 时间效率大大提高。另外, 当**记录数 n 较小**时, n^2 值受 n 的值影响不大。Shell排序正是从这两个方面出发对直接插入排序进行改进。

基本作法：先取一个小于 n 的整数 d_1 作为第一个增量。把待排序的记录分成 d_1 组，所有距离为 d_1 的记录放在同一组内，在各组内实行直接插入排序；然后，取第二个增量 $d_2 < d_1$ ，重复上述分组和排序过程，直到所取增量 $d_k = 1 (d_k < d_{k-1} < \dots < d_2 < d_1)$ ，此时所有的记录放置在同一组内进行直接插入排序。

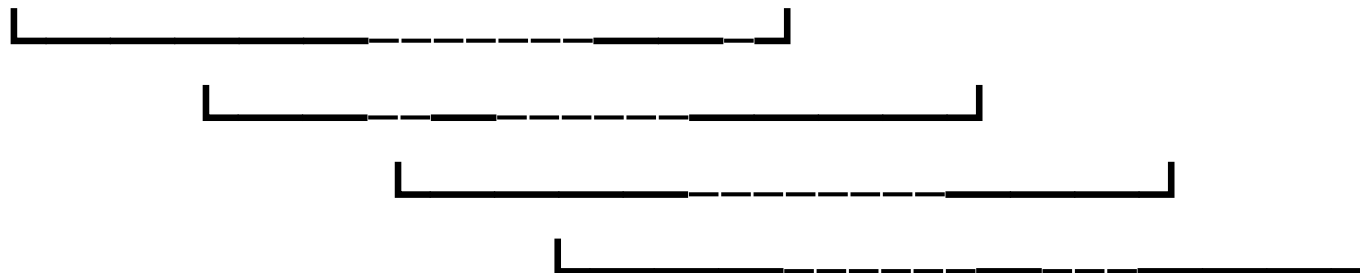
对于增量的选择：Shell提出 $d_1 = \left\lceil \frac{n}{2} \right\rceil, d_{i+1} = \left\lceil \frac{d_i}{2} \right\rceil$ ；Knuth提出： $d_{i+1} = \left\lceil \frac{d_i}{3} \right\rceil$ 还有其它选择，但何种最佳尚无法证明。无论如何，最后增量必须为1。

(2) 示例: {49, 38, 65, 97, 13, 76, 27, 49'}

原始序列

49 38 65 97 13 76 27 49'

d=4



d=2

13 38 27 49' 49 76 65 97



d=1

13 38 27 49' 49 76 65 97



排序结果:

13 27 38 49' 49 65 76 97

(3) 算法实现：（增量减半）

```
void ShellSort(SortObject *pv, int d)
```

```
{  int i, j, inc;
```

```
    RecordNode temp;
```

```
    for (inc = d; inc > 0; inc /= 2)          /* 增量序列 */
```

```
    {  for (i = inc; i < pv->n; i++)          /* 每组直接插入排序 */
```

```
        {  temp = pv->record[i];    j = i-inc;
```

```
            while (j >= 0 && temp.key < pv->record[j].key)
```

```
            {  pv->record[j+inc] = pv->record[j];    j -= inc;    }
```

```
            pv->record[j+inc] = temp;
```

```
        }
```

```
    }
```

```
}
```

(4) 时间效率分析:

Shell排序比直接插入排序快，其时间复杂度分析比较困难。

Knuth证明：在增量 $1/3$ 递减时，Shell排序的平均比较次数和平均移动次数都为 $n^{1.3}$ 。

辅助空间： 1 [temp]

稳定性： 不稳定的排序。

插入排序小结：

除Shell外，其它的插入排序的时间复杂度为 $O(n^2)$ ，并且是稳定的。
所有的插入排序都首先确定位置，然后插入。

直接插入排序简单，容易实现，但当 n 较大时，时间效率低。
其它插入排序方法都是从减少比较次数、移动次数出发对直接插入排序进行改进。

- ① **直接插入**：顺序检索确定位置，记录移动实现插入；
- ② **折半插入**：减少比较次数（折半检索确定位置，记录移动实现插入）；
- ③ **2-路插入**：减少比较次数和移动次数（通过首记录划分两个子有序表，在前或后有序表中折半检索确定位置，记录移动实现插入）；
- ④ **表插入**：减少移动次数（采用链表存储，顺序检索确定位置，指针修改代替记录移动完成插入）；
- ⑤ **Shell排序**：分组，缩小增量（来自：当待排序序列基本有序，并且 n 较小时，直接插入排序效率大大提高），不稳定排序。

8.3 选择排序

思想： 每趟从待排序的记录序列中选择排序码最小的记录放置到待排序表的最前位置，直到全部排完。

关键问题： 在剩余的待排序记录序列中找到最小排序码记录。

方法： 直接选择排序和堆排序。

1. 直接选择排序:


(1) 思想: 首先在所有记录序列中选择排序码最小的记录与第一个记录交换; 然后从第二个记录开始, 再选择排序码最小的记录与第二个记录交换; 依次类推, 直到所有记录排好序。


每趟排序选择一个记录, n 个记录需要 $n-1$ 趟直接选择排序。


第 i 趟排序从第 i 到第 n 个记录中选择排序码最小的, 交换到第 i 个位置。


最小排序码选择: 从一端开始顺序比较。


(2) 示例: {23, 11, 55, 97, 19, 80}

第一趟: 11, {23, 55, 97, 19, 80} 11


第二趟: 11, 19, {55, 97, 23, 80} 19


第三趟: 11, 19, 23, {97, 55, 80} 23


第四趟: 11, 19, 23, 55, {97, 80} 55


第五趟: 11, 19, 23, 55, 80, {97} 80


最后剩下97, 为最大的记录。

另外, 第*i*趟排序中, 如果最小记录位置为第*i*个, 不需要交换。

(3) 算法实现:

```
void SelectSort(SortObject *pv)
{  int i, j, k;
   RecordNode temp;
   for (i = 0; i < pv->n-1; i++)           /* n-1趟直接选择排序 */
   {   k = i;
       for (j = i+1; j < pv->n; j++)        /* 选择最小排序码 */
       {
           if (pv->record[j].key < pv->record[k].key)  k = j;
       }
       if (k != i)                               /* 需要交换吗? */
       {   temp = pv->record[i];
           pv->record[i] = pv->record[k];
           pv->record[k] = temp;
       }
   }
}
```

(4) 时间效率分析:

选择排序的比较次数与记录的初始状态无关。

第*i*趟排序: 从第*i*个记录开始, 顺序比较选择最小排序码记录需要*n-i*次比较。

总的比较次数:
$$\sum_{i=1}^{n-1} (n - i) = \frac{n(n-1)}{2}$$

最少移动次数: $M_{\min} = 0$ (初始为正序时)

最多移动次数: $M_{\max} = 3(n-1)$ (每趟1次交换, 3次移动完成)

时间复杂度: $T(n) = O(n^2)$, 辅助空间1个记录单位: Temp,

稳定性: 不稳定的排序 (交换可能改变相对次序)。

效率低的原因分析: 大量的重复比较, 后面没有利用前面已经比较的结果。

2. 堆排序:

(1) 思想: 直接选择排序的改进, 在后面的选择过程中利用前面选择时的比较结果以减少比较次数。

堆的定义: n 个元素的序列 $\{k_1, k_2, \dots, k_n\}$, 当且仅当满足如下关系时, 称之为堆。

$$\begin{cases} k_i \leq k_{2i} \\ k_i \leq k_{2i+1} \end{cases} \quad \text{或者} \quad \begin{cases} k_i \geq k_{2i} \\ k_i \geq k_{2i+1} \end{cases} \quad (i = 1, 2, \dots, \lfloor n/2 \rfloor)$$

从定义看出: 堆实质上可以用一棵完全二叉树描述, 根大于 (大根堆) 或小于 (小根堆) 其左右子女。

大根堆示例:

77, 61, 59, 48, 19, 11, 26, 15, 1, 5

.....

.....

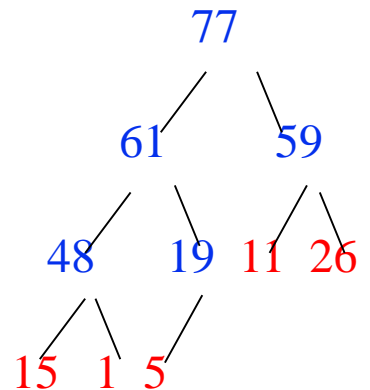
.....

.....

.....

.....

.....



堆(大根堆)排序的主要思想:

- 设法把原始序列构造成一个堆, 使得n个元素的最大值处于序列的第一个位置;
- 然后交换序列第一个元素(最大值元素)与最后一个元素;
- 再把序列的前n-1个元素组成的子序列构成一个新堆, 得到第二大元素, 把序列的第一个元素与第n-1个元素交换。此后再把序列的前n-2个元素构成一个新堆....., 如此操作, 最终整个序列成为有序序列。

$(K_1, K_2, K_3, \dots, K_n)$

$(\mathbf{K_{max1}}, [\dots]) \rightarrow [\dots], \mathbf{K_{max1}}$

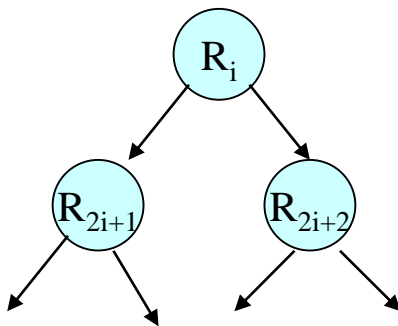
$(\mathbf{K_{max2}}, [\dots], K_{max1}) \rightarrow [\dots], \mathbf{K_{max2}}, K_{max1}$

.....

关键问题：如何将原始序列构成初始堆，丢掉最大值后如何构造新堆？

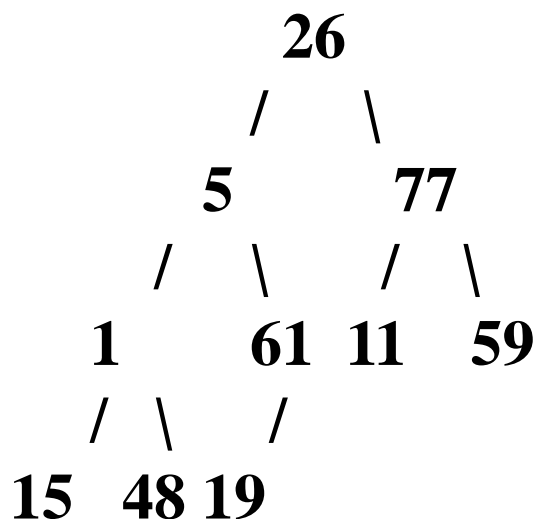
初始完全二叉树中， $\lfloor n/2 \rfloor$, $\lfloor n/2 \rfloor + 1$, ..., $n-1$ 为叶子，以其为根的子树必然为堆。因此，**初始堆建立**时，只需要将所有非终端结点为根的子树调整为堆。

建堆方法：采用“**筛选法**”为以 R_i 为根的完全二叉树建堆。这时 R_i 的左、右子树都是堆，可以把 R_i 与其左、右子树根结点 R_{2i+1} 、 R_{2i+2} 中最大者交换位置。若交换位置后破坏了子树的堆特性，则再对这棵子树重复交换过程，直到以 R_i 为根结点的子树成为堆。

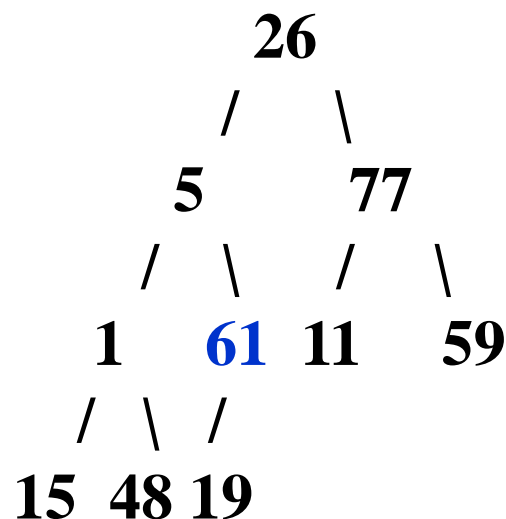


(2) 示例（大根堆）

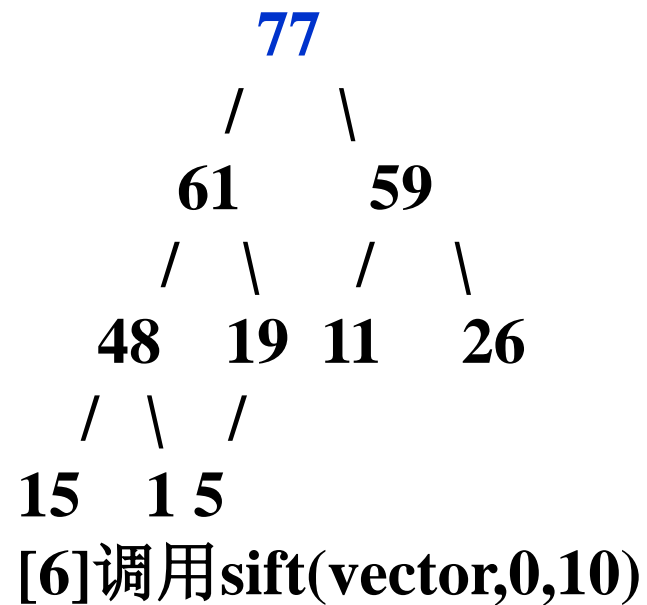
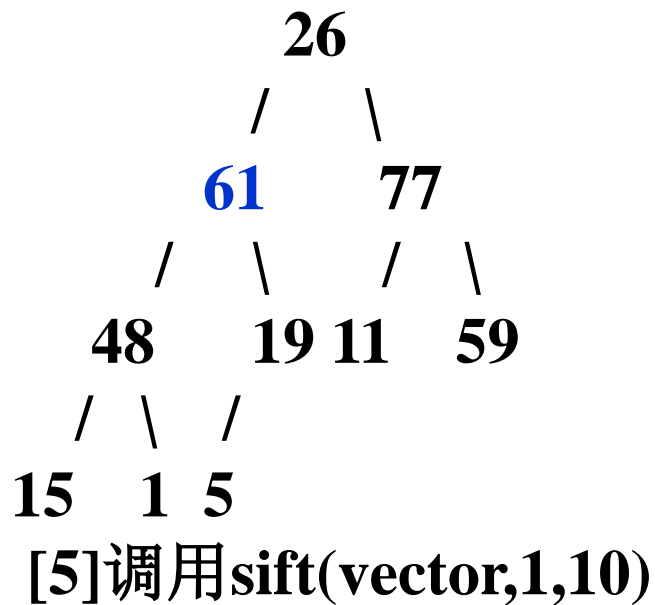
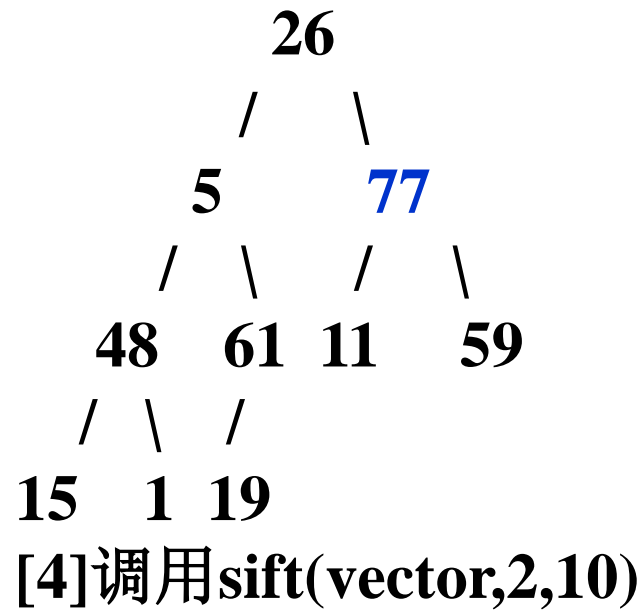
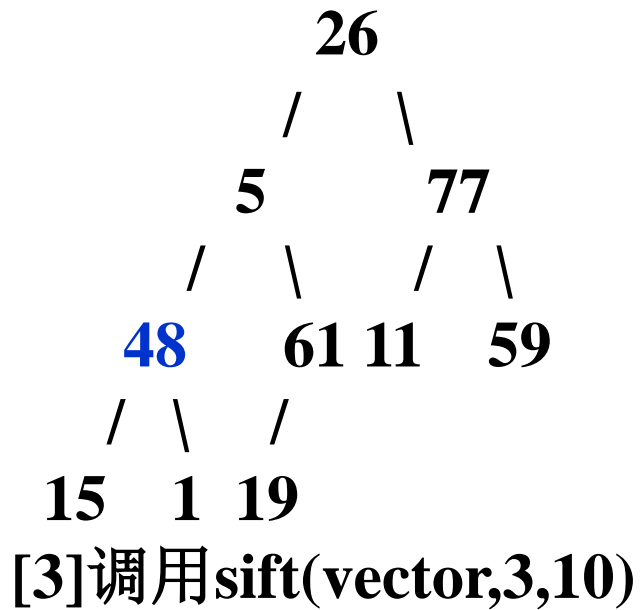
初始序列为 26, 5, 77, 1, 61, 11, 59, 15, 48, 19,

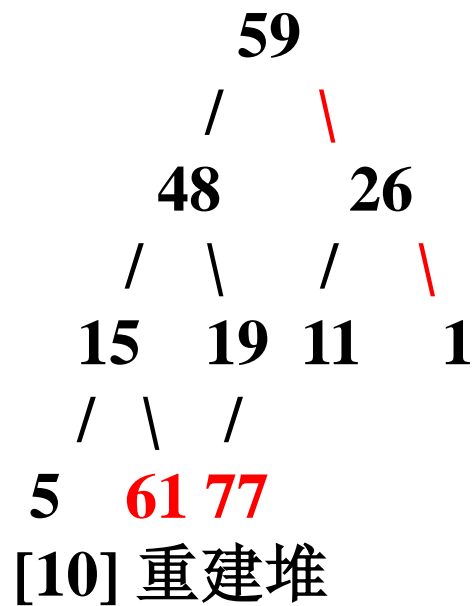
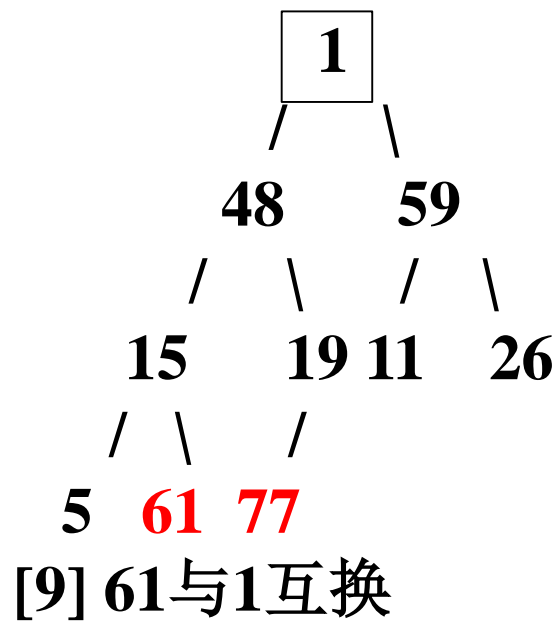
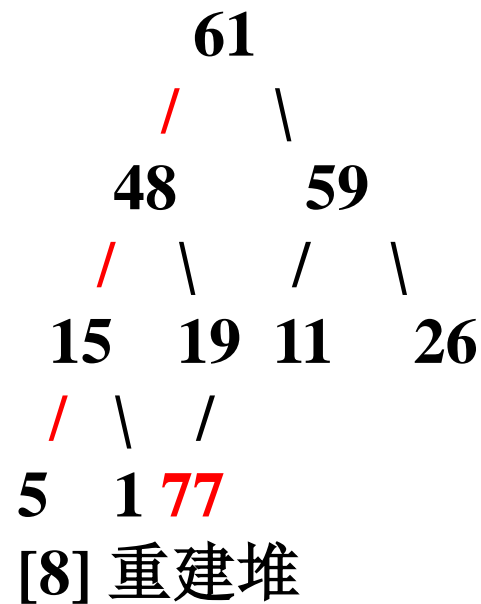
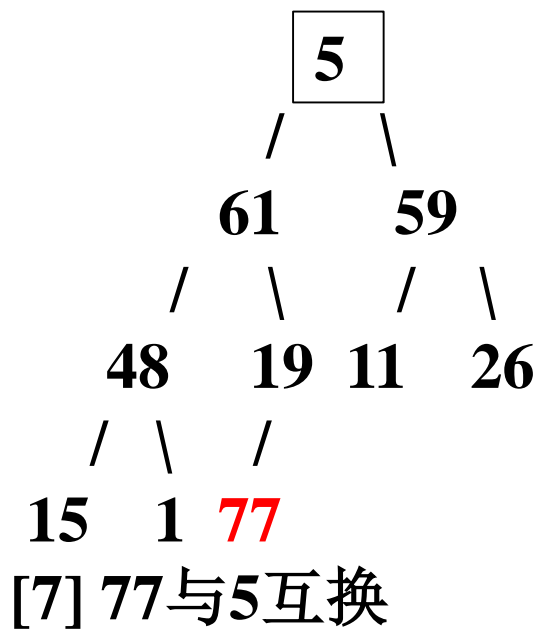


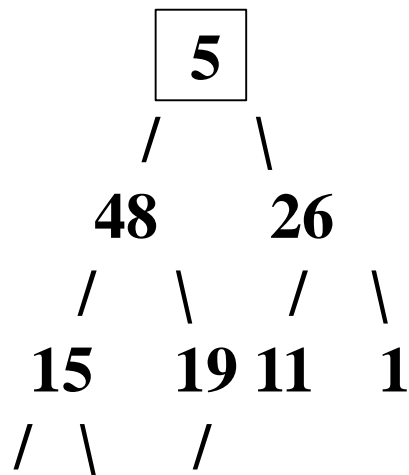
[1] 初始完全二叉树



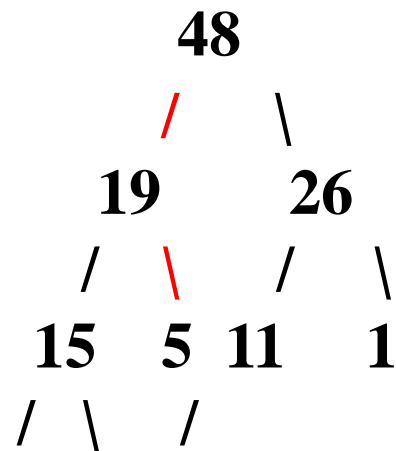
[2] $i = \lfloor n/2 \rfloor - 1 = 4$, 调用 `sift(vector, 4, 10)`



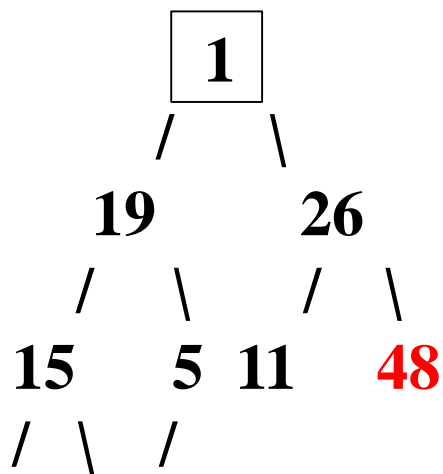




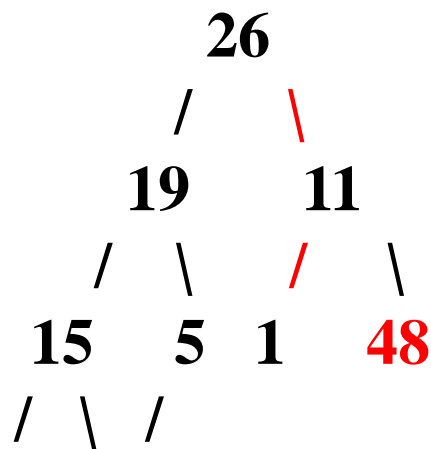
59 61 77
 [11] 59与5互换



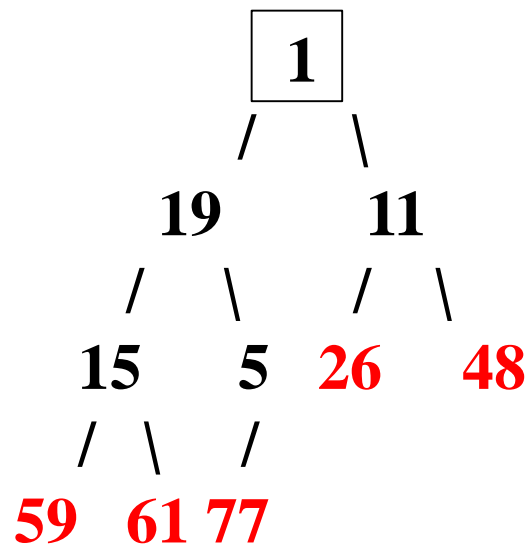
59 61 77
 [12] 重建堆



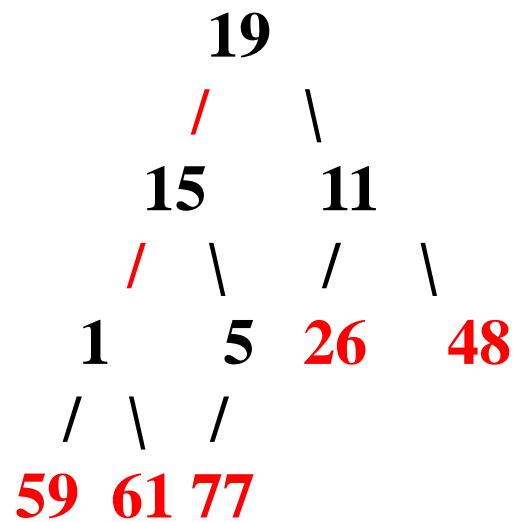
59 61 77
 [13] 48与1互换



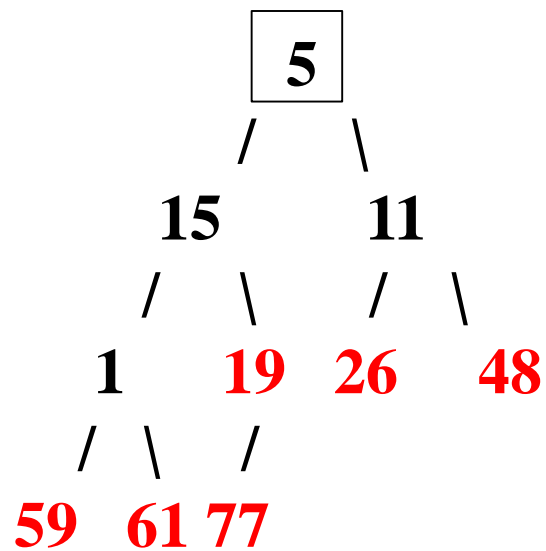
59 61 77
 [14] 重建堆



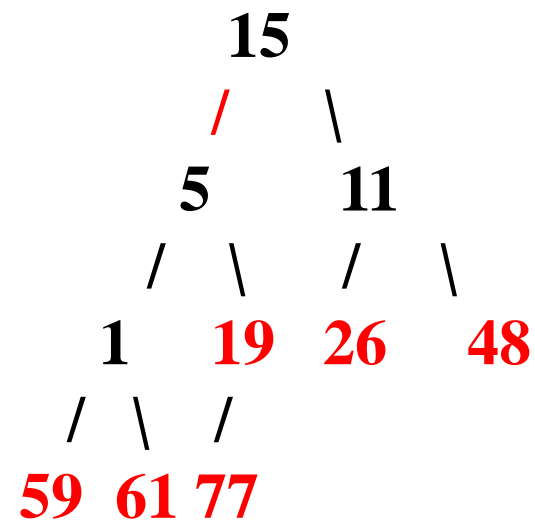
[15] 26与1互换



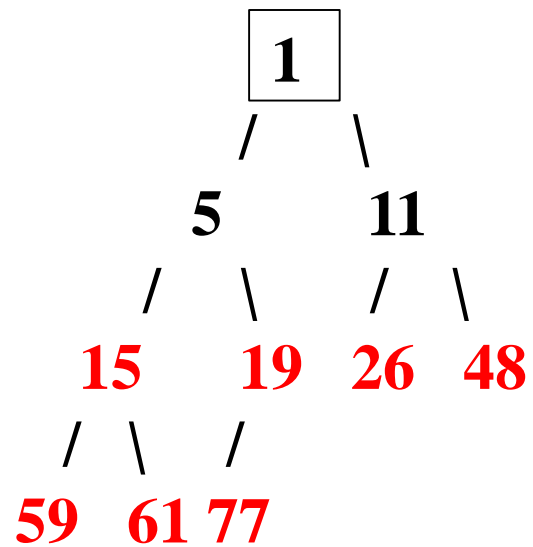
[16] 重建堆



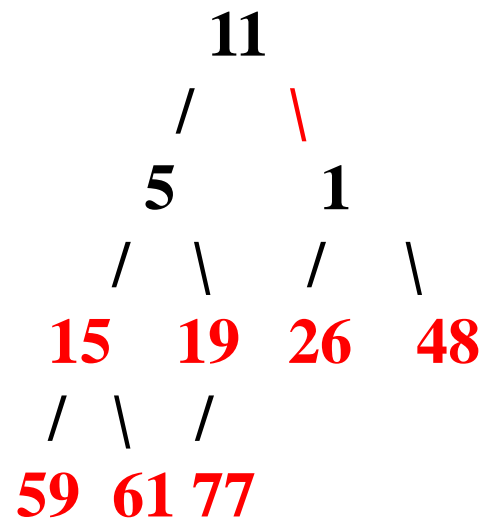
[17] 19与5互换



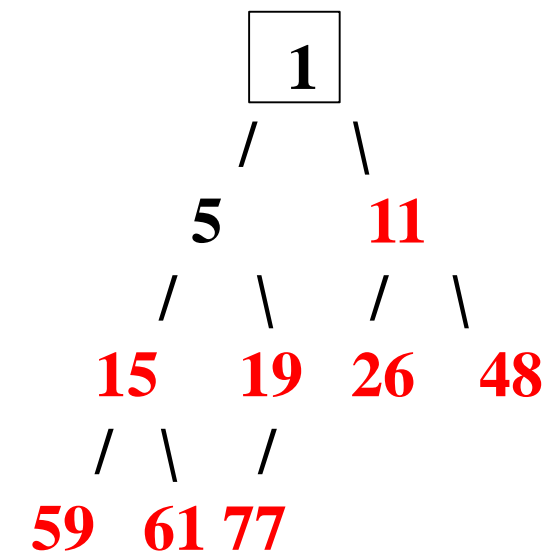
[18] 重建堆



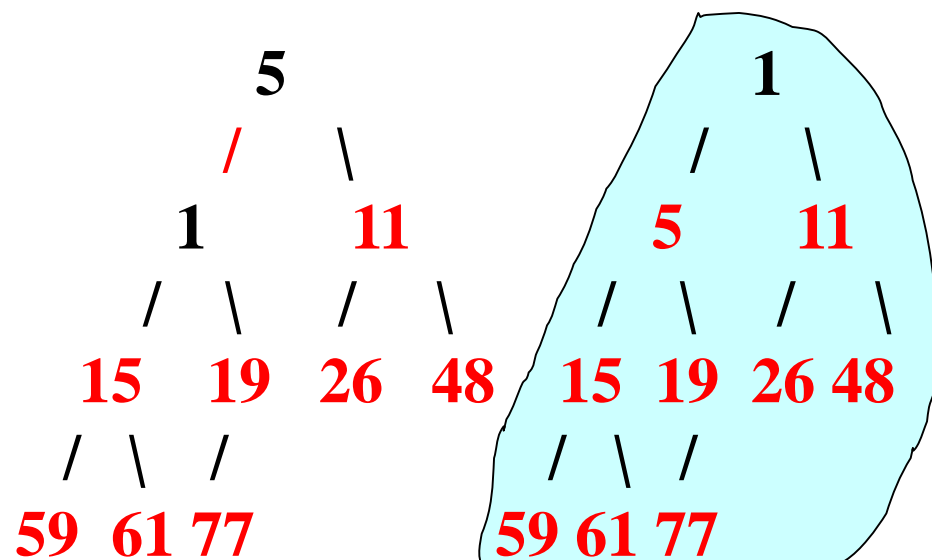
[19] 15与1互换



[20] 重建堆



[21] 11与1互换



[22] 重建堆

(3) 算法实现:

筛选算法: $\{k_0, k_1, \dots, k_{n-1}\}$, k_j ($j=i+1, i+2, \dots, n-1$) 为根的子树均满足堆定义, 筛选结束后, 以 k_i 为根的子树也是堆。

```
#define leftchild(i)  2*i+1
```

```
void sift(SortObject *pv, int i, int n)
```

```
{  int child;
```

```
    RecordNode temp = pv->record[i];
```

```
    child = leftchild(i);
```

```
    while (child < n)
```

```
    {  if (child < n-1 && pv->record[child].key < pv->record[child+1].key)
```

```
        child++;
```

/* 选择左右子女中较大者 */

```
    if (temp.key < pv->record[child].key)
```

```
    {  pv->record[i] = pv->record[child]; /* 交换并且继续往下探索 */
```

```
        i = child;  child = leftchild(i);
```

```
    }
```

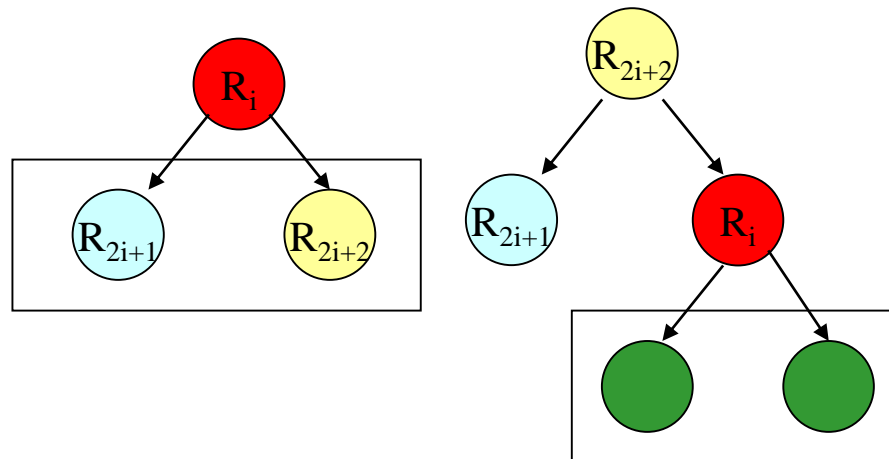
```
    else  break;
```

/* 没有交换, 终止探索 */

```
}
```

```
pv->record[i] = temp;
```

```
}
```



堆排序算法:

```
void HeapSort(SortObject *pv)
{
    int i, n;
    RecorNode temp;
    /* 构造初始堆 */
    n = pv->n;
    for ( i = n/2-1; i>=0; i--)
        sift(pv, i, n);
    /* 交换并且调整堆 */
    for (i = n-1; i>0; i--)
    {
        temp = pv->record[0];
        pv->record[0] = pv->record[i];
        pv->record[i] = temp;
        sift(pv, 0, i);
    }
}
```

(4) 时间效率分析[p223]

建初始堆比较次数 C_1 : $O(n)$

重新建堆比较次数 C_2 :

$O(n\log_2 n)$

总比较次数= C_1+C_2

移动次数小于比较次数

因此,

时间复杂度: $O(n\log_2 n)$

空间复杂度: $O(1)$

适用于 n 值较大的情况。

算法稳定性: 不稳定

二叉树深度 $h = \log_2(n+1)$

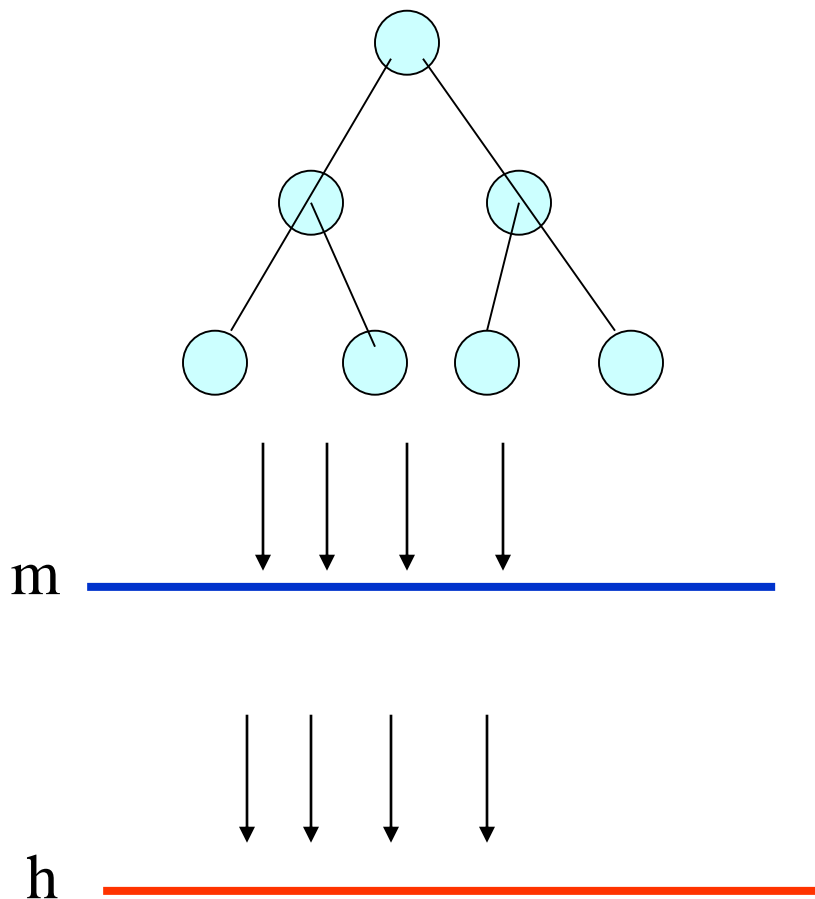
m 层结点个数: 2^m , 每个结点最多与两个子结点比较, 到叶子共 $h-m$ 层, 因此初始建堆该层上的结点最多比较次数为 $2(h-m)$, m 层所有结点建堆最多比较次数: $2(h-m) \cdot 2^m$
总共 h 层, 因此

$$C_1 = \sum_{m=0}^{h-1} (h-m) 2^{m+1} \leq 4n = O(n)$$

重新建堆:

$$C_2 = \sum_{j=1}^{n-1} 2(\lceil \log_2(n-j+1) \rceil) < 2n \log_2 n = O(n \log_2 n)$$

总的比较次数: $C = C_1 + C_2 = O(n) + O(n \log_2 n) = O(n \log_2 n)$



8.4 交换排序

思想： 每趟在待排序的记录序列中两两比较，若发现两个记录的排序码值次序相反，则进行交换，直到没有反序的记录为止。

关键问题： 两个记录的选择，相邻比较[起泡排序]，
划分比较[快速排序]。

方法： 起泡排序和快速排序。

1. 起泡排序:

(1) 思想: 首先 R_0 与 R_1 比较, 若前者大于后者, 二者交换, 否则不交换; 然后, R_1 与 R_2 比较, 做同样的处理; 依次类推, 直到处理完 R_{n-2} 与 R_{n-1} 。这样, 经过 $n-1$ 次比较和交换 (或不交换) 过程, 将最大排序码的记录移到最后位置。此后再对前面 $n-1$ 个记录进行同样的处理, 使这 $n-1$ 个记录中最大排序码记录移动到第 $n-1$ 个位置;, 重复上述过程直到一趟起泡排序中无交换为止。

每趟起泡排序将待排序的记录序列中最大排序码记录移动到最后;

最多 $n-1$ 趟起泡排序 (逆序), 最小一趟 (正序)。

(2) 示例: {23, 11, 55, 97, 19, 80}

第一趟: {11, 23, 55, 19, 80}, 97

第二趟: {11, 23, 19, 55}, 80, 97

第三趟: {11, 19, 23}, 55, 80, 97

第四趟: 11, 19, 23, 55, 80, 97 /* 无交换, 结束*/

(3) 算法实现:

```
void BubbleSort(SortObject *pv)
{   int i, j, noswap;
    for (i=0; i <pv->n-1;i++)
    {
        noswap = TRUE;
        for (j = 0; j<pv->n-i-1; j++)
        {
            if (pv->record[j+1].key < pv->record[j].key)
            {
                pv->record[j] <=> pv->record[j+1];
                noswap = FALSE;
            }
        }
        if (noswap) break;
    }
}
```

(4) 时间效率分析:

空间复杂度: $O(1)$

时间复杂度: $O(n^2)$

正序: 比较次数 $n-1$ 次

移动次数 0 次

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

移动次数: $3 * C_{\max}$

算法稳定性: 稳定

2. 快速排序:

(1) 思想: 起泡排序是相邻记录的比较和交换, 每次只能上移或下移一个位置, 速度慢。快速排序是对起泡排序的改进: 记录的比较和交换是从两端向中间进行(分区交换排序), 这样排序码较大的记录一次就能从前面移到后面, 同样的排序码较小的记录一次就能从后面移到前面, 记录每次移动的距离较远, 从而减少总的比较次数和移动次数。快速排序是目前内部排序中速度最快的排序方法。

基本做法：从 n 个待排序记录序列中任选一个记录（通常选择第一个）为基准，使得序列中所有小于和等于基准的记录均放置在该记录的前面，而所有大于基准的记录均放置在该记录的后面。这样，以该记录的排序码为基准，将待排序的记录序列分割成两组（该记录目前的位置就是其最终位置）。然后，分别对前后两组分别进行快速排序。重复上述过程，直到所有的记录都得到最后位置为止。

每趟快速排序实质上就是找基准记录的最终位置，如何实现？

(2) 示例:

第一趟: {23, 11, 55, 97, 19, 80}

$\uparrow i$ $\leftarrow \uparrow j$

j往前扫描找小于

{19, 11, 55, 97, 23, 80}

基准记录与基准交换:

$\uparrow i \rightarrow \uparrow j$

i往后扫描找大于

{19, 11, 23, 97, 55, 80}

基准记录与基准交换:

$\uparrow i \leftarrow \uparrow j$

j往前扫描找小于

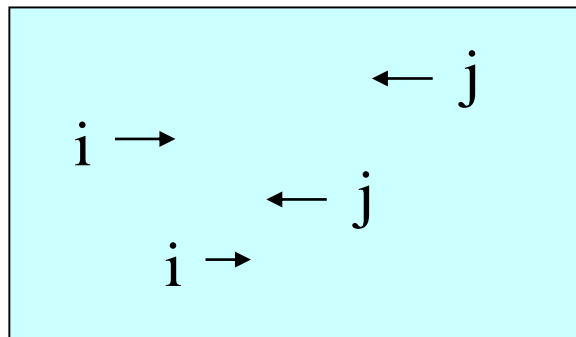
{19, 11, 23, 97, 55, 80}

基准记录与基准交换:

$\uparrow i \uparrow j$

(碰到*i=j*,表示一趟结束, 找到位置)

i、j交替进行



各趟排序结果: {23, 11, 55, 97, 19, 80}

{19, 11}, 23, {97, 55, 80}

{11}, 19, 23, {97, 55, 80}

11, 19, 23, {97, 55, 80}

11, 19, 23, {80, 55}, 97

11, 19, 23, {55}, 80, 97

11, 19, 23, 55, 80, 97

往前扫描一次，再往后扫描一次，交替进行，直到相碰为止。

具体实现时，首先保存基准记录，中间不参与每次交换。最后将基准记录移动到最终位置。

(3) 算法实现:

```
void QuickSort(SortObject *pv, int L, int R)
{
    int v, j;    RecordNode temp;
    if (L >= R) return;                                /* 边界不合法, 结束 */
    i = L; j = R; temp = pv->record[i]; /* 设置左右指针并保存基准记录 */
    while (i != j)
    {
        /* 向前扫描 */
        while ( pv->record[j].key >= temp.key && j > i) j--;
        if (i < j) pv->record[i++] = pv->record[j]; /* 交换 */
        /* 向后扫描 */
        while ( pv->record[i].key <= temp.key && i < j) i++;
        if (i < j) pv->record[j--] = pv->record[i]; /* 交换 */
    }
    pv->record[i] = temp; /* 基准移动到最终位置 */
    QuickSort(pv, L, i-1); /* 分别对左右区间进行快速排序 */
    QuickSort(pv, i+1, R);
}
```


(4) 时间效率分析:

若把每次划分所用的基准记录作为根，把划分得到的左区间和右区间分别作为根的左右子树，那么整个排序过程就对应着一棵二叉树。所需划分的次数等于二叉树的高度减1。

快速排序中，记录的移动次数小于或等于比较次数。

正序时算法执行时间最长:

第一趟需要 $n-1$ 次比较，基准记录位置不变，
右区间为有 $n-1$ 个记录的正序序列，.....

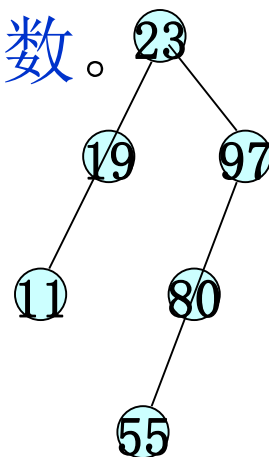
总的比较次数:

$$C_{\max} = \sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2} = o(n^2)$$

最好情况下是每次划分的两个区间长度大致相等:

$$C_{\min}(n) \approx n \log_2 n + n C_{\min}(1) = \mathbf{O(n \log_2 n)}$$

[教材证明, p263]



为了改善最坏情况下的时间性能，选择基准记录时，采用“三者取中”法。即： $R[L].key$, $R[R].key$ 和 $R[(L+R)/2].key$ 中间值记录为基准记录(具体实现时可以先将该基准记录与 $R[L]$ 交换,然后采用前面的快速排序实现)。

快速排序的**平均时间效率**为 $O(n\log_2 n)$ 。

算法需要一个**栈实现递归**，栈的深度取决于划分次数，最多不超过 n 。

如果每次都选较大的部分进栈，先处理较短的部分，则递归层数不超过 $\log_2 n$ ，因此快速排序的辅助空间为 $O(\log_2 n)$ 。

快速排序是不稳定的。

(4) 改进:

- ❑ 基准记录采用“三者取中”法选择;
 - ❑ 在向前、向后扫描过程中同时进行“起泡”操作, 即: 相邻记录逆序时进行互换。同时在算法中附设两个BOOL型变量, 分别指示向前扫描和向后扫描的中间移动过程中是否进行过记录交换。若向前扫描时无交换, 则不需要再对右区间进行排序; 同样, 若向后扫描时无交换, 则不需要再对左区间进行排序。
 - ❑ 一趟排序后, 比较左右区间的长度, 先对长度短的区间进行快速排序, 避免基准记录位于两端, 减少递归层数。
-

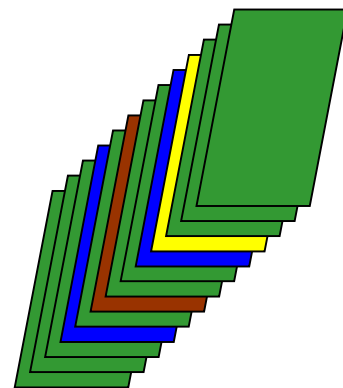
8.5 分配排序

思想： 分配排序是一种借助多排序码排序思想对单逻辑排序码排序的方法。

52张扑克牌，按照梅花、方块、红心、黑桃四种花色的面值从小到大排序
(花色的地位高于面值)

两种排法：

- ① 按照花色分成4堆，按照花色从小到大收集后，再按照面值分成13堆，按照面值收集。
- ② 按照面值分成13堆，按照面值从小到大收集后，再按照花色分成4堆，按照花色收集。



1. 概述:

假设记录序列F有n个记录: $F=(R_0, R_1, \dots, R_{n-1})$

且每个记录 R_i 中含有d个排序码 $(k_i^0, k_i^1, \dots, k_i^{d-1})$, 则F对

排序码 $(k^0, k^1, \dots, k^{d-1})$ 有序是指:任意两个记录 R_i 和

$R_j(0 \leq i \leq j \leq n-1)$ 满足词典次序有序关系

$$(k_i^0, k_i^1, \dots, k_i^{d-1}) < (k_j^0, k_j^1, \dots, k_j^{d-1})$$

其中 k^0 称为最高位排序码, k^{d-1} 称为最低位排序码。

实现多排序码排序有两种方法: 高位优先和低位优先法。

高位优先：先对最高位排序码 k^0 排序，将F分成若干堆，每堆中的记录具有相同的 k^0 ；然后分别对每堆按排序码 k^1 排序，分成若干子堆，如此重复，直到对 k^{d-1} 排序。最后，将各堆按次序叠放在一起，成为有序序列。

低位优先：从最低位排序码 k^{d-1} 起排序，然后再对 k^{d-2} 排序，如此重复，直到对 k^0 排序。

2. 基数排序

(1) 思想:

基数排序: 低位优先法对单逻辑排序码排序。

排序方法: 把每个排序码看成是一个d元组:

$$K_i = (K_i^0, K_i^1, \dots, K_i^{d-1})$$

其中每个 K_i^j 都是集合 $\{C_0, C_1, \dots, C_{r-1}\}$ ($C_0 < C_1 < \dots < C_{r-1}$) 中的值, 即 $C_0 \leq K_i^j \leq C_{r-1}$ ($0 \leq i \leq n-1, 0 \leq j \leq d-1$), 其中r称为**基数**。排序时先按 K_i^{d-1} 从小到大将记录分配到r个堆中, 然后依次收集; 再按 K_i^{d-2} 分配到r个堆中, 然后收集; 如此反复, 直到对 K_i^0 分配、收集, 得到的便是排好序的序列。

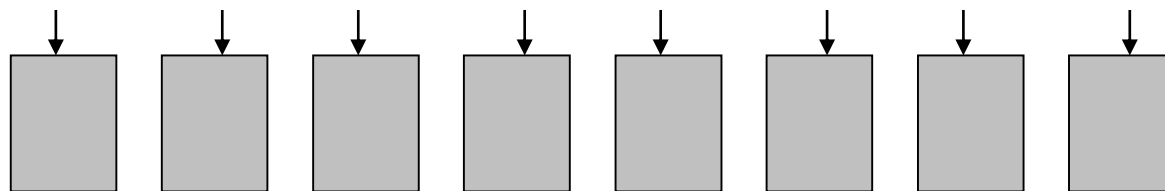
十进制关键码: $r=10$, $C_0=0$, $C_9=9$

小写字母串: $r=26$, $C_0='a'$, $C_{25}='z'$

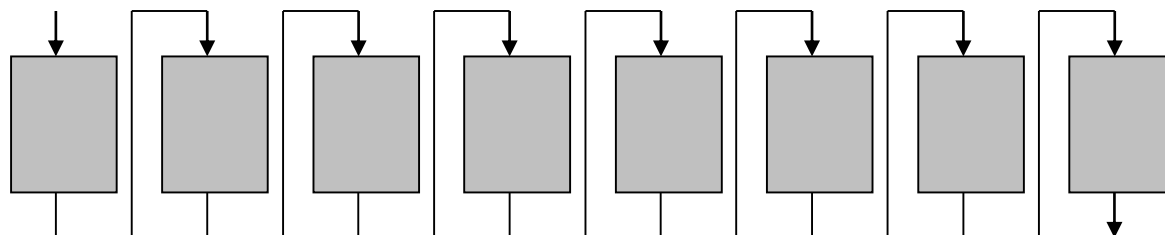
ASCII(1): $r = 128$; ASCII(f): $r = 256$;

基数排序的实现: 增设 r 个队列, 排序前为空队列, 分配时按照排序码位将记录插入到各自的队列中, 收集时将队列中的记录排列在一起。

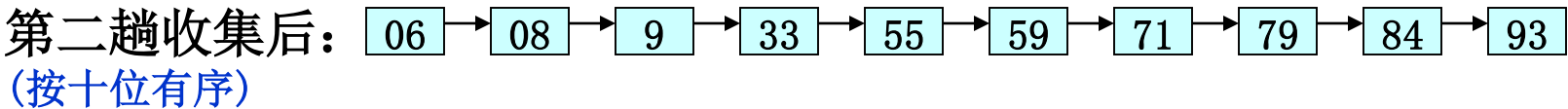
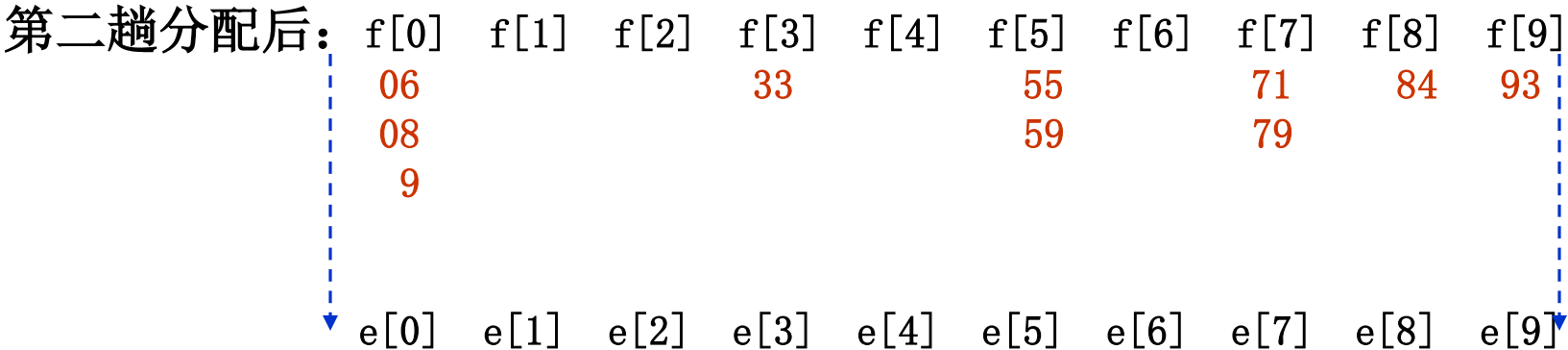
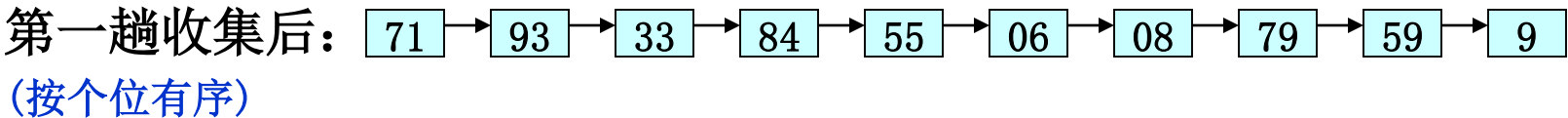
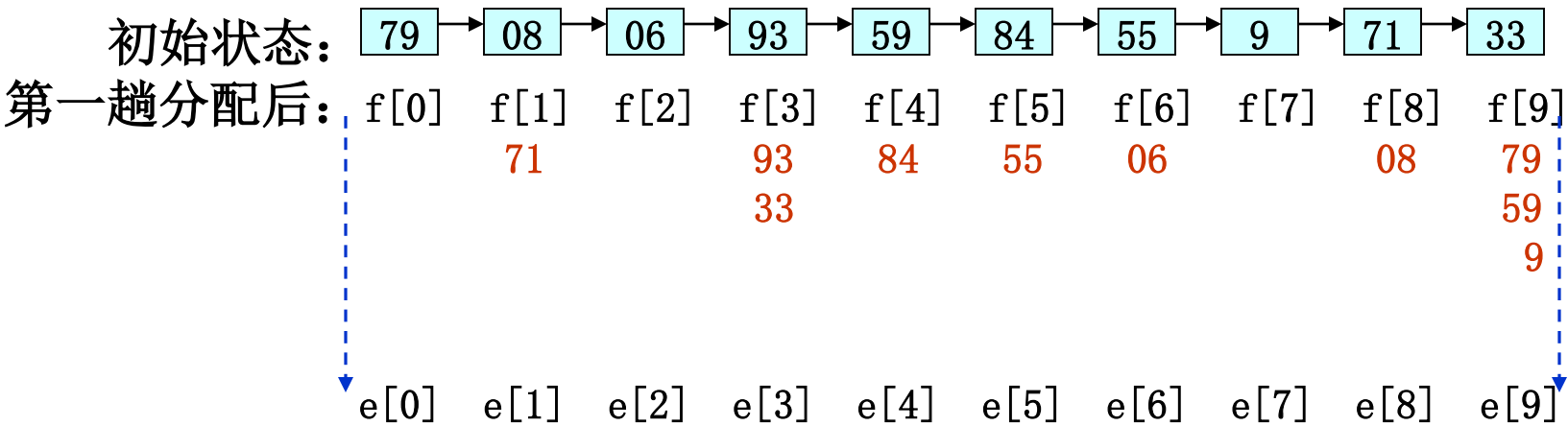
分配



收集



(2) 示例：初始序列 { 79, 08, 06, 93, 59, 84, 55, 9, 71, 33 }



(3) 算法实现:

```
#define D      3                /* D为排序码的最大位数 */
#define R      10              /* R为基数 */
typedef struct Node              /* 单链表结点类型 */
{
    KeyType      key[D];
    DataType     info;
    struct Node  *next;
}RadixNode, RadixList, PRadixList;

typedef struct QueueNode
{
    RadixNode    *f;    /* 队列的头指针 */
    RadixNode    *e;    /* 队列的尾指针 */
}Queue;

Queue queue[R];    //R个队列（分配前，空）
```

```

void RadixSort(RadixList * plist, int d, int r)
{
    int          i, j, k;
    RadixNode    *p, *head;

    head=(*plist)->next;
    for(j=d-1; j>=0 ; j--) /* 按位循环 */
    {
        for(i=0; i<r ; i++) /* 初始化 */
        {
            queue[i].f=queue[i].e=NULL;
        }
        /*分配到相应的队列中*/
        p=head;
        while(p!=NULL)
        {
            k=p->key[j]; /* 第j位 */
            if(queue[k].f==NULL) /* p链接到第k个队列 */
                queue[k].f=p;
            else
                (queue[k].e)->next=p; /* 中 */
            queue[k].e=p;
            p=p->next; /* 下一个记录 */
        }
    }
}

```

/* 收集成一个链表 */

/* 找第一个非空队列 */

i = 0;

while(queue[i].f==NULL) i++;

head=queue[i].f;

p=queue[i].e;

/* 下一个队列 */

for(i++; i<r ; i++)

{

if(queue[i].f!=NULL)

{ /* 链接到链表中 */

p->next=queue[i].f;

p=queue[i].e;

}

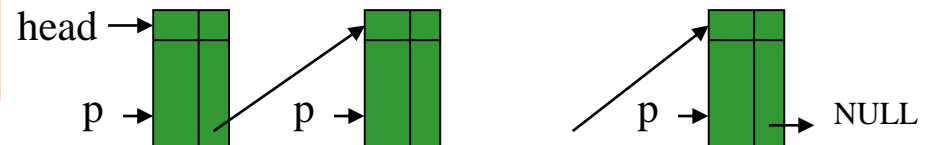
}

p->next=NULL; /* 完成收集 */

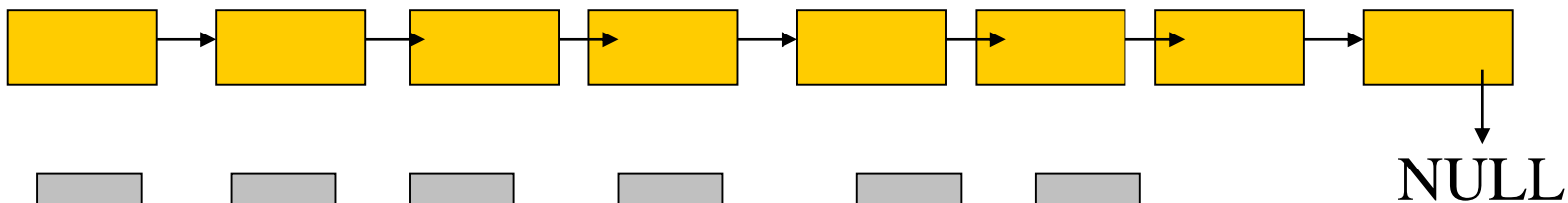
}

(*plist)->next=head;

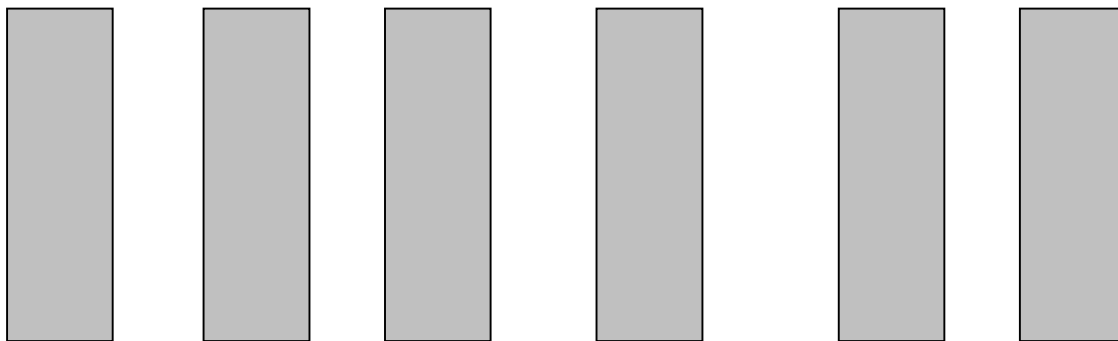
}



分配



f

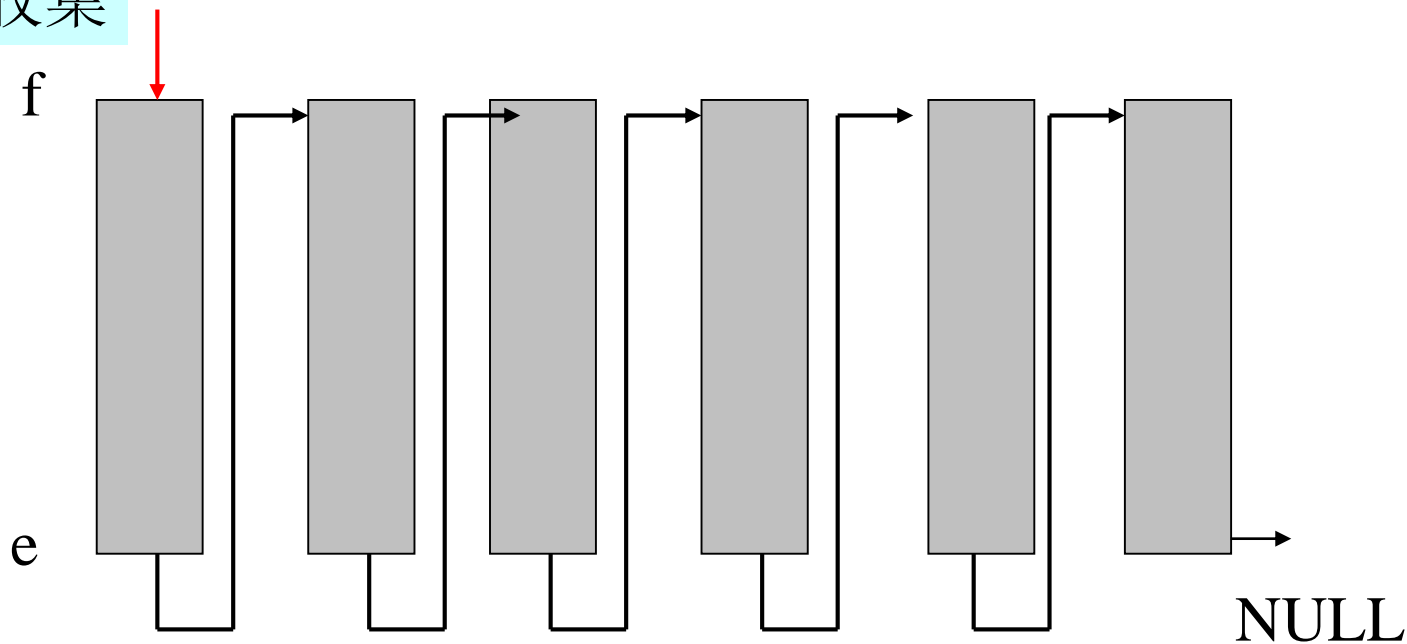


e

利用原链表结点构成队列

收集

f



(4) 时间效率分析:

基数排序过程中没有排序码的比较和记录的移动，只是对链表的扫描和指针的赋值。因此，时间主要耗费在指针的修改上。

每趟排序中：清队列的时间为 $O(r)$ ，
将 n 个记录分配到队列的时间为 $O(n)$ ，
收集的时间为 $O(r)$ 。
因此，一趟排序时间为 $O(r+n)$

总共需要 d 趟排序，因此：

总的时间复杂度： $T(n)=O(d(n+r))$

可见： n 较大， d 较小时，基数排序非常有效。

辅助空间： $S(n)=O(n+r)$ ，每个记录一个增加指针项，
需要一个包含 r 个队列的辅助队列数组

稳定性：稳定的排序。

8.6 归并排序

思想：

归并排序是借助多个有序表合并的思想进行排序的方法。

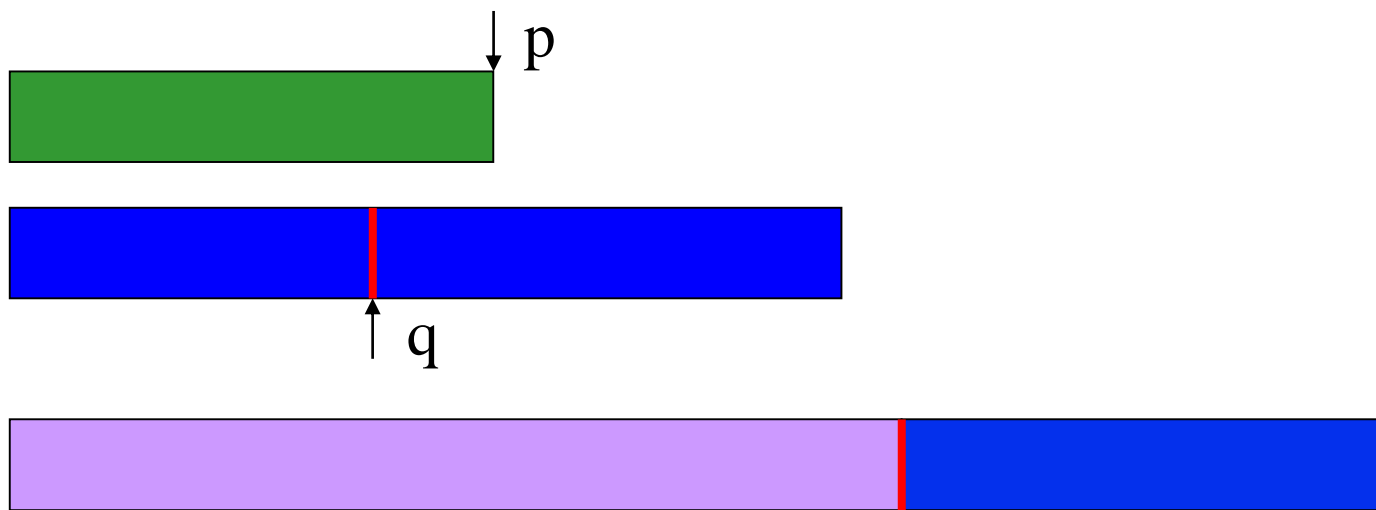
归并有：二路归并（每次合并两个有序表），
多路归并（每次合并多个有序表）

1. 2路归并的思想：

n 个记录首先看成 n 个有序表，每个有序表包含1个记录。 n 个有序表两两合并得到 $n/2$ 个有序表，每个有序表包含2个或1个记录（ n 为奇数时，最后一个记录无法合并）。然后，再两两合并得到包含更多记录的有序表，如此重复，直到得到包含 n 个记录的有序表为止。此时，得到记录的排序结果。

两个有序表的合并方法：首先比较两个表中的第一个记录，将最小者取出，再继续比较两个有序表的其它记录，如此反复直到一个或两个有序表中无记录为止，然后将包含记录的有序表直接连接到排序序列中。

3个循环实现两个有序表的合并。



2. 2路归并的示例

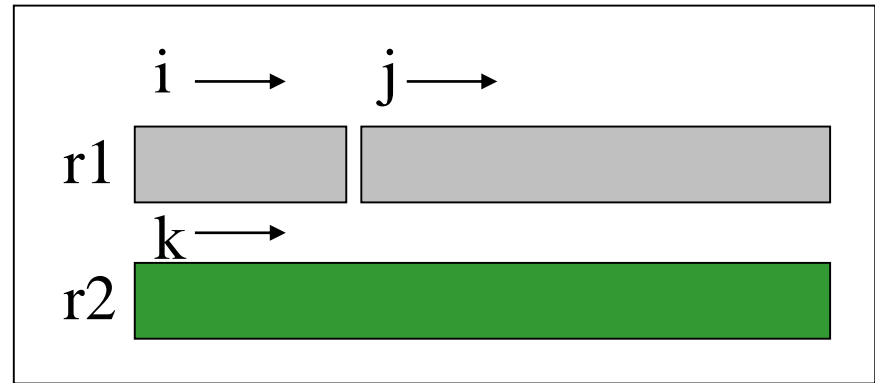
初始序列为:	25, 57, 48, 37, 12, 82, 75, 29, 16
初始排序码	<u>25 57</u> <u>48 37</u> <u>12 82</u> <u>75 29</u> <u>16</u>
	\ / \ / \ / \ /
第一趟归并后	<u>25 57</u> <u>37 48</u> <u>12 82</u> <u>29 75</u> <u>16</u>
	\ / \ / \ /
第二趟归并后	<u>25 37</u> <u>48 57</u> <u>12 29</u> <u>75 82</u> <u>16</u>
	\ / \ / \ /
第三趟归并后	<u>12 25</u> <u>29 37</u> <u>48 57</u> <u>75 82</u> <u>16</u>
	\ / \ / \ /
第四趟归并后	<u>12 16</u> <u>25 29</u> <u>37 48</u> <u>57 75</u> <u>82</u>

排序后的结果为: 12, 16, 25, 29, 37, 48, 57, 75, 82

3. 2路归并的算法

/ r1[low..m], r1[m+1..high]的合并, 得到r2[low..high] */*

```
void merge(RecordNode *r1, RecordNode *r2, int low, int m, int high)
{
    int i, j, k;
    i = low;
    j = m+1;
    k = low;
    while( (i<=m) && (j<=high) )
    {
        if(r1[i].key<=r1[j].key)  r2[k++]=r1[i++];
        else                      r2[k++]=r1[j++];
    }
    while (i<=m)    r2[k++]=r1[i++];    //表2结束
    while (j<=high) r2[k++]=r1[j++];    //表1结束
}
```



/* 对r1做一趟归并，结果放在r2中(length为子表长度) */

```
void mergePass(RecordNode r1[], RecordNode r2[], int n, int length)  
{  
    int i = 0, j;  
    //归并两个长度为length的子表  
    while(i+2*length-1<n)  
    {  
        merge(r1, r2, i, i+length-1, i+2*length-1);  
        i+=2*length;  
    }  
    if(i+length-1<n-1) //剩下两个子表，其中一个的长度小于length  
        merge(r1, r2, i, i+length-1, n-1);  
    else  
    {  
        for(j=i; j<n; j++) r2[j]=r1[j]; //最后一个子表直接复制到r2中  
    }  
}
```

/* 2路归并算法 */

void mergeSort(SortObject * pvector)

{

RecordNode *r1 = pvector->record;

RecordNode r2[MAXNUM];

int n = pvector->n;

int length = 1;

while (length < n) //循环一次，做两趟2路归并

{

//一趟归并，结果放在r2中

mergePass(r1, r2, n, length);

length*=2;

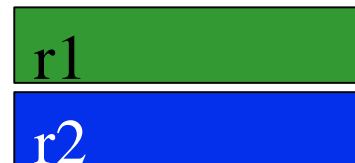
//一趟归并，结果放在pvector-> record中

mergePass(r2, r1, n, length);

length*=2;

}

}



4. 2路归并的时间效率分析

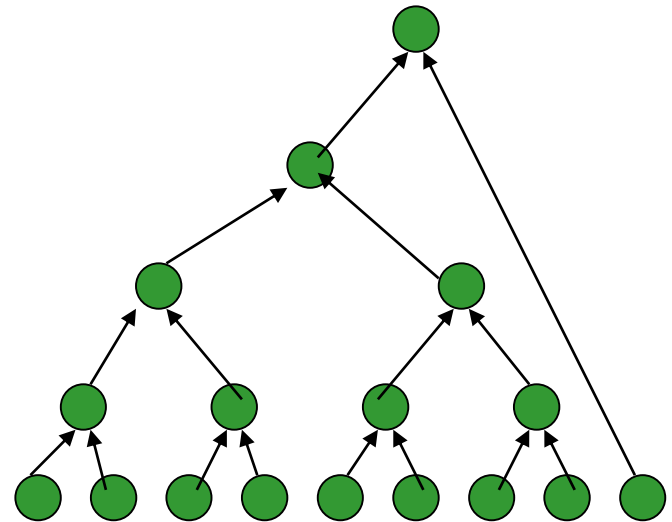
第 i 趟归并后，最长有序子表的长度不超过 2^i 个。

n 个记录，需要 $\lceil \log_2 n \rceil$ 趟归并，每趟归并时间为 $O(n)$ ，

因此：2路归并的时间效率为： **$T(n) = O(n \log_2 n)$**

空间复杂度： **$S(n) = O(n)$** ，一个辅助数组 $r2$ （包含 n 个记录）

稳定性：**稳定的内部排序。**



各种排序法的比较

排序法比较时考虑的因素：

- 算法的时间复杂度
- 算法的辅助空间
- 排序的稳定性
- 算法结构的复杂性
- 参加排序的数据的规模

结论:

- ❑ 当数据规模 n 较小时, n^2 和 $n\log_2 n$ 的差别不大, 采用简单排序方法比较合适。
- ❑ 初态已基本有序时, 可选择简单的排序方法。
- ❑ 当数据规模 n 较大时, 应选用速度快的排序算法, 其中快速排序法最快。
- ❑ 堆排序不会出现象快速排序那样的最坏情况, 且堆排序所需的辅助空间比快速排序少。如果要求排序是稳定的, 则可以选择归并排序方法。
- ❑ 当 n 较大, 记录的排序码位数较少且可以分解时, 采用基数排序方法较好。
- ❑ 归并排序法也可以用于外排序。

排序方法	最坏时间复杂度	平均时间复杂度	辅助空间	稳定性
直接插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定的
折半插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定的
表插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	稳定的
Shell排序	$O(n^{1.3})$	$O(n^{1.3})$	$O(1)$	不稳定的
直接选择排序	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定的
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定的
起泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定的
快速排序	$O(n^2)$	$O(n\log_2 n)$	$O(\log_2 n)$	不稳定的
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(r+n)$	稳定的
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定的

上机作业:

1) 给定一组英文单词（假定全部为小写字符，单词中只有a~z 26个字符构成，最大长度为d个），设计并完成按照基数排序对英文单词字典排列的算法和程序。

所有长度不足d个字母的单词，都在尾部补足空格。排序时设置27个箱子，分别与空格, a, b, c, ..., z对应。

2) p279, 应用题

直接插入、直接选择、起泡、Shell、快速和堆排序实现并比较。

书面作业: P278

复习题: (1)、(3)、(5)

算法题: (4)