

第7章 排序（ sorting）

7.1 概述

7.2 插入排序

7.3 交换排序

7.4 选择排序

7.5 归并排序

本章小结

7.1 概述

1. 排序： n 个对象的序列 $R[0], R[1], R[2], \dots, R[n-1]$ 按其关键码的大小，进行由小到大（非递减）或由大到小（非递增）的次序重新排序的。

2. 关键码（key）

3. 两大类： 内排序：对内存中的 n 个对象进行排序。
外排序：内存放不下，还要使用外存的排序。

4. 排序算法的稳定性:

如果待排序的对象序列中, 含有多个关键码值相等的对象, 用某种方法排序后, 这些对象的相对次序不变的, 则是稳定的, 否则为不稳定的。

例: 35 8₁ 20 15 8₂ 28
 8₁ 8₂ 15 20 28 35
 稳定的

5. 排序种类:

—内排序

插入排序, 交换排序, 选择排序, 归并排序,
基数排序

—外排序

6. 排序的算法分析:

- 1) 时间开销—比较次数, 移动次数
- 2) 所需的附加空间

下面是静态排序过程中所用到的数据表类定义:

| vector | | key | otherdata |
|---------------|-----|-----|-----------|
| 0 | | | |
| 1 | | | |
| | | | |
| currentsize-1 | ... | ... | ... |
| maxsize-1 | | | |

```
const int DefaultSize=100;
template<class Type>class datalist
template<class Type>class Element
{ private:
    Type key;
    field otherdata;
public:
    Type getKey( ){return key;}
    void setKey(const Type x){key=x;}
    Element <Type>&operator=(Element<Type> &x ){ this = x; }
    int operator==(Type & x){return !(this<x||x<this);}
    int operator!=(Type & x){return this<x||x<this;}
    int operator<=(Type & x){return !(this>x);}
    int operator>=(Type & x){return !(this<x);}
    int operator<(Type & x){return  this>x;}
}
```

```
template<class Type> class datalist
{ public:
    datalist(int MaxSz=DefaultSize):MaxSize(MaxSz),CurrentSize(0)
        { vector=new Element<Type>[MaxSz];}
    void swap (Element <Type> & x, Element<Type> & y)
        {Element <Type> temp=x; x=y; y=temp;}
private:
    Element <Type> * vector;
    int MaxSize; CurrentSize;
}
```

7.2 插入排序 (insert Sorting)

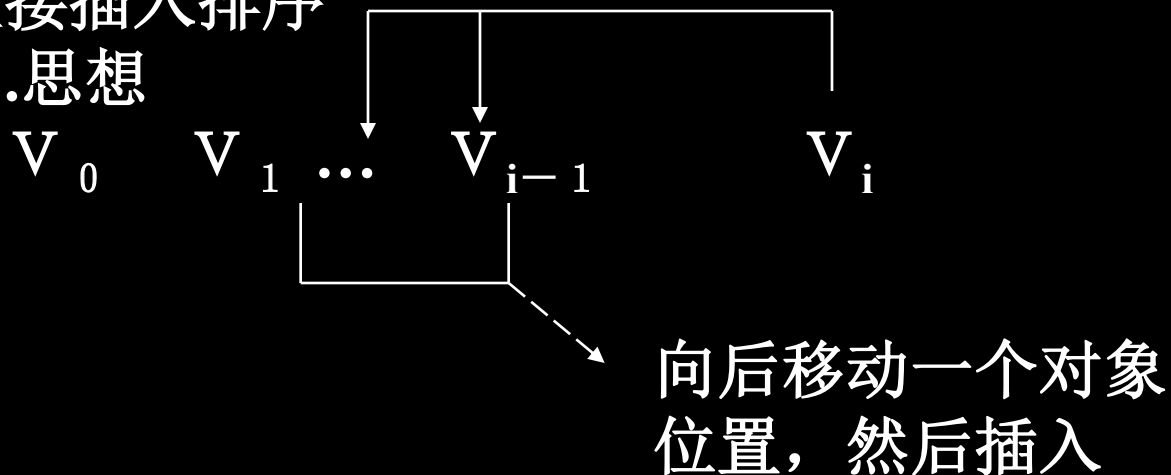
思想: V_0, V_1, \dots, V_{i-1} 个对象已排好序,
现要插入 V_i 到适当位置

例子: 体育课迟到的人

方法: 直接插入排序, 链表插入排序, 折半插入排序, 希尔排序

一、直接插入排序

1. 思想



2.例子

i=1

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 8 | 3 | 2 | 5 | 9 | 1 | 6 |

| | |
|---|---|
| 3 | 8 |
|---|---|

| | | |
|---|---|---|
| 2 | 3 | 8 |
|---|---|---|

| | | | |
|---|---|---|---|
| 2 | 3 | 5 | 8 |
|---|---|---|---|

...

3.主程序

```
template<class Type> void InsertionSort(datalist<Type> & list)
{ for (int i=1; i<list.CurrentSize; i++) Insert(list, i) ;
}
```


子程序

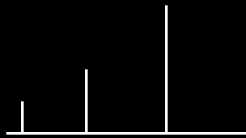
```
template<class Type> void Insert(datalist<Type> & list, int i)
{ Element<Type> temp=list.vector[i]; int j=i ;
  while(j>0&& temp.getkey( )<list.vector[j-1].getkey( ))
    { list.Vector[j]=list.Vector[j-1]; j-- }
  list.Vector[j]=temp;
}
```

java program

```
public static void insertionSort( Comparable [ ] a )  
{ int j;  
  for ( int p = 1; p < a.length; p++ )  
  { Comparable tmp = a[ p ];  
    for ( j = p; j > 0 && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )  
      a[ j ] = a[ j - 1 ];  
    a[ j ] = tmp;  
  }  
}
```

4.算法分析

1)n个对象已有序



比较总次数 $KCN = n - 1 = O(n)$

移动次数 $RMN = 0$

2) n个对象逆序



$$KCN = 1 + 2 + 3 + \dots + (n-1) = n(n-1)/2 = O(n^2)$$

$$\begin{aligned} RMN &= (1+2) + (2+2) + \dots + (n-1+2) = n(n-1)/2 + 2(n-1) \\ &= (n^2 + 3n - 4)/2 = O(n^2) \end{aligned}$$

3).对一般情况

处理第 i 个对象 r_i 时，它有 i 个可能的插入位置

↓
 $r_1, r_2, r_3, \dots, r_i, r_{i+1}, \dots, r_n$

- ▶ 不被移动(0次)，仍在第 i 个位置上。比较了一次；
- ▶ 可能向左移动了一个位置（比较2次，移动3次）、两个位置（比较3次，移动4次）、...直至 $i-1$ 个位置（比较 i 次，移动 $i+1$ 次）而到达表的前端。

假设 r_i 被插在这 i 个位置上的概率是相等的，
则：

不被移动的可能性为 $1/i$ (比较次数为1)；
被移动的可能性为 $(i-1)/i$ 。

∴被移动情况下的平均比较次数为 $(2+3+\dots+i-1+i)/(i-1)=i/2+1$
平均移动次数为 $(1+2+\dots+i-1)/(i-1)+2=i/2+2$

用各自的概率组合这两种情况:

第*i*个元素的平均比较与移动次数

$$C=1/i*1+((i-1)/i)*(i/2+1)=(i+1)/2$$

$$m=((i-1)/i)*(i/2+2) \quad m \leq (i+3)/2$$

*i*从2~*n*变化,所以结果为

$$KCN = \sum_{i=2}^n [(i+1)/2] = 1/4n^2 + 3/4n + 1 = O(n^2)$$

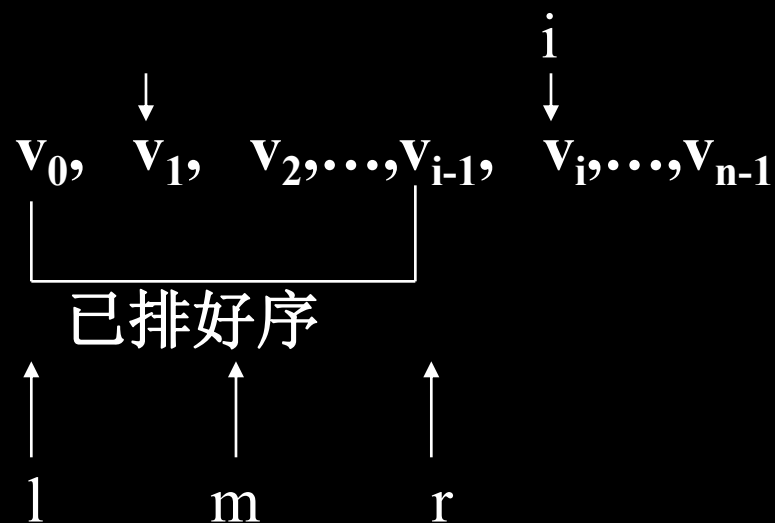
$$RMN \leq \sum_{i=2}^n [(i+3)/2] = 1/4n^2 + 7/4n - 2 = O(n^2)$$

5.稳定性: 稳定的

二、折半插入排序 (Binary Insert Sort)

也称二分法插入排序

1. 思想



2. 例子

| | | | | | | | |
|-----|-------|-----|-------|----|----|-----|-----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 28 | 13 | 72 | 85 | 39 | 41 | 6 | 20 |
| 6 | 13 | 28 | 39 | 41 | 72 | 85 | 20 |
| l | m_2 | l | m_1 | | | r | i |

left 的位置就是最后要插入的位置

3.算法

```
template <class Type> void BinaryInsertSort( datalist<Type> &list)
{  for (int i=1; i<list.currentSize; i++) BinaryInsert(list, i);
}

template <class Type> void BinaryInsert( datalist<Type> &list, int i)
{  int left=0, Right=i-1;
    Element<Type>temp=list.Vector[i];
    while (left<=Right)
    {  int middle=(left+Right)/2;
        if (temp.getkey( )<list.Vector[middle].getkey( ))
            Right=middle-1;
        else left=middle+1;
    }
    for (int k=i-1; k>=left ; k--) list.Vector[k+1]=list.Vector[k];
    list.Vector[left]=temp;
}
```

4. 算法分析

折半查找所需比较次数与初始排序无关，仅依赖于对象个数

比较次数: $v_0, v_1, v_2, \dots, v_{i-1}, \overset{\uparrow}{v_i}, \dots, v_{n-1}$

设 $n=2^k$ ，插入第 i 个对象时，需要经过 $\lfloor \log_2 i \rfloor + 1$ 次关键词比较

\therefore 折半查找所需的关键词比较次数为：

$$\begin{aligned}
 \sum_{i=1}^{n-1} (\lfloor \log_2 i \rfloor + 1) &= \underbrace{1}_{2^0 \text{ 个 } 1} + \underbrace{2+2}_{2^1 \text{ 个 } 2} + \underbrace{3+3+\dots+3}_{2^2 \text{ 个 } 3} + \underbrace{4+\dots+4+4+\dots+4}_{2^3 \text{ 个 } 4} + \dots + \underbrace{k+k+\dots+k}_{2^{k-1} \text{ 个 } k} \\
 &= 2^0 + 2^1 + 2^2 + \dots + 2^{k-1} \\
 &\quad + 2^1 + 2^2 + \dots + 2^{k-1} \\
 &\quad + 2^2 + \dots + 2^{k-1} \\
 &\quad \dots \\
 &\quad + 2^{k-2} + 2^{k-1} \\
 &\quad + 2^{k-1}
 \end{aligned}$$

$$\begin{aligned}
 &= \sum_{i=1}^k (2^k - 2^{i-1}) = k \cdot 2^k - \sum_{i=1}^k 2^{i-1} = k \cdot 2^k - 2^k + 1 \\
 &= n \cdot \log_2 n - n + 1 \approx n \cdot \log_2 n = O(n \cdot \log_2 n)
 \end{aligned}$$

5.稳定性： 稳定

三、希尔排序（Shell Sort）

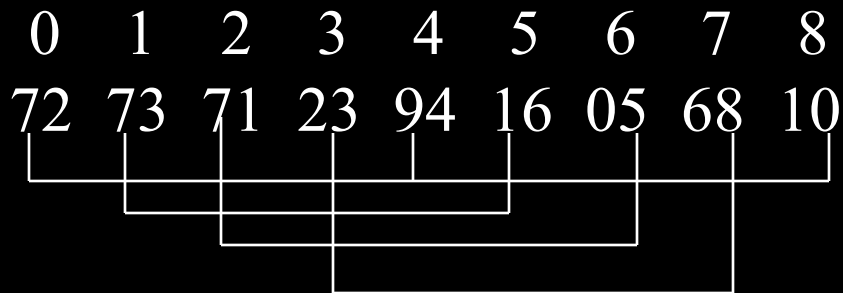
又称为缩小增量排序（**diminishing-increment sort**）

1.方法

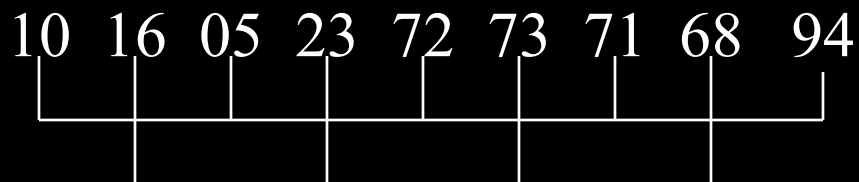
- 1) 取一增量（间隔 $gap < n$ ），按增量分组，对每组使用直接插入排序或其他方法进行排序。
- 2) 减少增量（分的组减少，但每组记录增多）。直至增量为1，即为一个组时。

2.例子

gap=4



gap=2



gap=1



05 10 16 23 68 71 72 73 94

3.算法

```
template <class Type> void Shellsort( datalist <Type> & list)
{   int gap=list.CurrentSize/2;
    while (gap)
    {   ShellInsert(list, gap); gap=gap==2? 1 : (int)(gap/2); }
}
```

```
template<class Type> void ShellInsert( datalist<Type> &list;
                                     const int gap)
{   for (int i=gap; i<list.CurrentSize; i++)
    {   Element<Type>temp=list.Vector[i];
        int j=i;
        while(j>=gap &&temp.getkey( )<list.Vector[j-gap].getkey( ))
        { list.Vector[j]=list.Vector[j-gap]; j-=gap; }
        list.Vector[j]=temp;
    }
}
```

java program

```
public static void shellsort( Comparable [ ] a )  
{ for ( int gap = a.length / 2; gap > 0; gap /= 2 )  
    for ( int i = gap; i < a.length; i++ )  
    { Comparable tmp = a[ i ];  
      int j = i;  
  
      for ( ; j >= gap && tmp.compareTo( a[ j- gap ] ) < 0; j -= gap )  
          a[ j ] = a[ j - gap ];  
      a[ j ] = tmp;  
    }  
}
```

4.稳定性：不稳定

5.算法分析：与选择的缩小增量有关，但到目前还不知如何选择最好结果的缩小增量序列。

平均比较次数与移动次数大约 $n^{1.3}$ 左右。

7.3 交换排序

方法的本质：不断的交换反序的对偶，直到不再有反序的对偶为止。

两种方法： 起泡排序（Bubble sort）
快速排序（Quick sort）

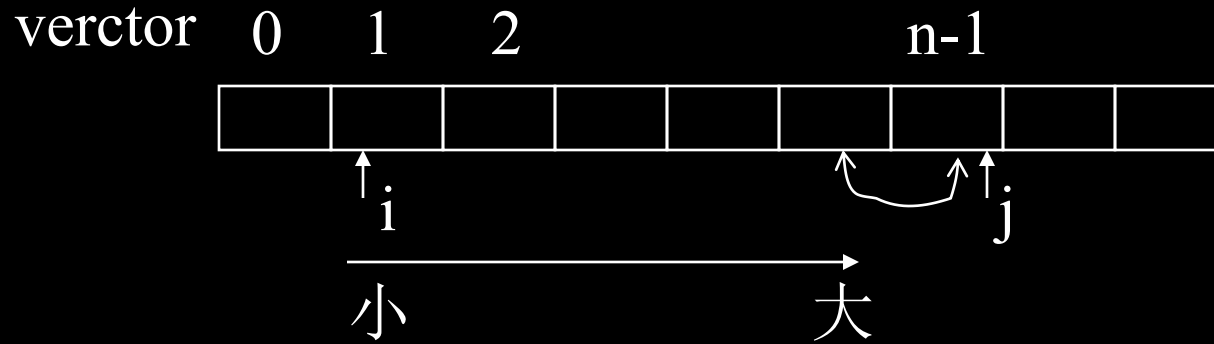
一、起泡排序

1. 方法：
 - 1) 从头到尾做一遍相邻两元素的比较，有颠倒则交换，记下交换的位置。一趟结束，一个或多个最大（最小）元素定位。
 - 2) 去掉已定位的元素，重复1，直至一趟无交换。

3.算法

```
template<class Type> void BubbleSort( datalist<Type> & list)
{  int pass=1;
   int exchange=1;
   while (pass<list.CurrentSize &&exchange)
   { BubbleExchange(list, pass, exchange); pass++;
   }
}
```

```
template<class Type> void BubbleExchange(datalist<Type> &list, const
                                         int i, int & exchange)
{  exchange=0;
   for (int j=list.CurrentSize-1; j>=i; j--)
       if (list.Vector[j-1].getkey( )>list.Vector[j].getkey( ))
       {  swap( list.Vector[j-1], list.Vector[j] );
          exchange=1;
       }
}
```



4.算法分析

最小比较次数

有序: $n-1$ 次比较, 移动次数为0

最大比较次数

逆序: $(n-1)+(n-2)+\dots+1=n(n-1)/2\approx O(n^2)$
(比较次数)

$$3 \sum_{i=1}^{n-1} i = (3/2)n(n-1)$$

(移动次数)

5.稳定性

起泡排序是稳定的

二、快速排序（分划交换排序）

1962年Hoare提出的。

1. 方法：

- 1) 在n个对象中，取一个对象（如第一个对象——基准pivot），按该对象的关键码把所有 \leq 该关键码的对象分划在它的左边。 $>$ 该关键码的对象分划在它的右边。
- 2) 对左边和右边（子序列）分别再用快排序。

2. 例子

| i ↓ | | | | | | | | | | j ↓ |
|----------|-----|----|------|----|-----|-----|-----|----|-------|----------|
| 46 | 13 | 55 | 42 | 94 | 05 | 17 | 70 | 82 | 100 | |
| [17 | 13 | 05 | 42] | 46 | [94 | 55 | 70 | 82 | 100] | |
| [05 | 13] | 17 | [42] | 46 | [94 | 55 | 70 | 82 | 100] | |
| 05 | 13 | 17 | 42 | 46 | [94 | 55 | 70 | 82 | 100] | |
| 05 | 13 | 17 | 42 | 46 | [82 | 55 | 70] | 94 | [100] | |
| 05 | 13 | 17 | 42 | 46 | [70 | 55] | 82 | 94 | 100 | |
| 05 | 13 | 17 | 42 | 46 | 55 | 70 | 82 | 94 | 100 | |

一次分划:



3.算法：用递归方法实现

```
template <class Type> void QuickSort( datalist <Type>& list,  
                                     const int left, const int right )  
{  if (left<right)  
    {  int pivotpos=partition (list, left, right);  
       QuickSort(list, left, pivotpos-1);  
       QuickSort(list, pivotpos+1, right);  
    }  
}
```

```

template <class Type> int partition(datalist<Type> &list,
                                   const int low, const int high)
{
    int i=low, j=high;
    Element<Type>pivot=list.Vector[low];
    while (i != j )
    {
        while(list.Vector[j].getkey( )>pivot.getkey( ) && i<j) j--;
        if (i<j) {list.Vector[i]=list.Vector[j]; i++;}

        while(list.Vector[i].getkey( )<pivot.getkey( ) && i<j) i++;
        if (i<j) {list.Vector[j]=list.Vector[i]; j--;}
    }
    list.Vector[i]=pivot ;
    return i;
}

```

4.稳定性:

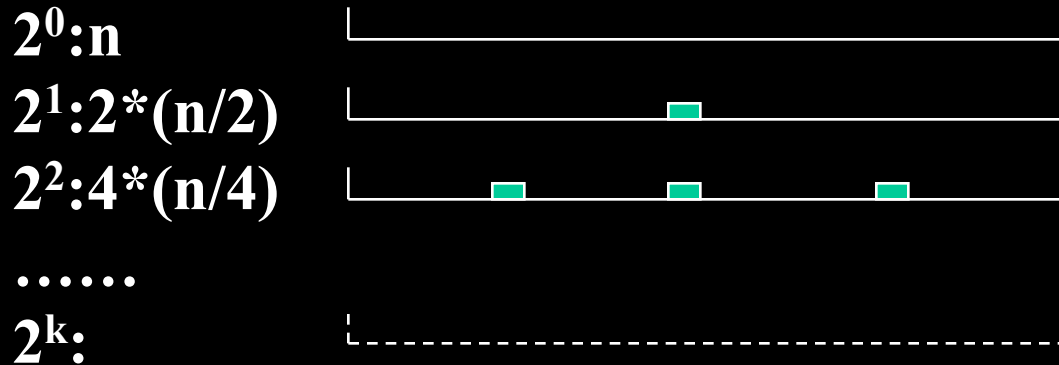
不稳定的排序方法

5. 算法分析

- 1) 最差的情况（当选第一个对象为分划对象时）
如果原对象已按关键码排好序

$$\left. \begin{array}{l} K_1 [\quad \quad \quad] \\ \quad K_2 [\quad \quad \quad] \\ \quad \quad K_3 [\quad \quad \quad] \\ \quad \quad \quad \dots\dots \end{array} \right\} O(n^2)$$

- 2) 最理想的情况
每次分划第一个对象定位在中间



设 $n=2^k$,一共做了K趟 $K=\log_2 n$

$$\begin{aligned}
 T(n) &\leq n + 2T(n/2) \leq n + 2(n/2 + 2T(n/4)) = 2n + 2^2T(n/2^2) \\
 &\leq 2n + 2^2(n/2^2 + 2T(n/2^3)) = 3n + 2^3T(n/2^3) \leq \dots \\
 &\leq kn + 2^kT(n/2^k) = n\log_2 n + nT(1) = O(n\log_2 n)
 \end{aligned}$$

可以证明QuickSort的平均计算时间也是 $O(n\log_2 n)$

下面讨论空间复杂性：

以上讨论的是递归算法，也可用非递归算法来实现。不管是递归（由编译程序来实现）还是非递归。第一次分划后，左部、右部要分别处理。

所以，栈来存放：

1) 存放什么：左部或右部的上、下界的下标。

2) 栈要多大： $O(\log_2 n) \sim O(n)$

↑
对应于有序情况

快速排序（分划交换排序）

1.选取pivot（枢纽元）

用第一个元素作pivot是不太好的。

方法1: 随机选取pivot, 但随机数的生成一般是昂贵的。

方法2: 三数中值分割法 (Median-of-Three partitioning)

N个数，最好选第 $\lceil N/2 \rceil$ 个最大数，这是最好的中值，但这是很困难的。

一般选左端、右端和中心位置上的三个元素的中值作为枢纽元。

8, 1, 4, 9, 6, 3, 5, 2, 7, 0 (8, 6, 0)

具体实现时：将 8, 6, 0 先排序，即 0, 1, 4, 9, 6, 3, 5, 2, 7, 8, 得到中值pivot为 6。

2.分割策略

将pivot与最后倒数第二个元素交换，使得pivot离开要被分割的数据段。然后，i 指向第一个元素，j 指向倒数第二个元素。

0, 1, 4, 9, 7, 3, 5, 2, 6, 8

↑ ↑

i j

然后进行分划

Java Quicksort algorithm

```
public static void quicksort( Comparable [ ] a)
{
    quicksort( a, 0, a.length - 1 );
}

private static Comparable median3( Comparable [ ] a, int left, int right )
{
    int center = ( left + right ) / 2;
    if ( a[center].compareTo( a[left ] < 0 )
        swapReferences( a, left, center );
    if ( a[ right ] . compareTo( a[left ] ) < 0 )
        swapReferences( a, left, right );
    if( a[right ] . compareTo( a[ center ] ) < 0 )
        swapReferences( a, center, right );

    swapReferences( a, center, right - 1 );
    return a[ right - 1 ];
}
```

8, 1, 4, 9, 6, 3, 5, 2, 7, 0 0, 1, 4, 9, 7, 3, 5, 2, 6, 8

Java Quicksort algorithm

```
private static void quicksort( Comparable [ ] a, int left, int right )
{   if( left + CUTOFF <= right )
    {   Comparable pivot = median3( a, left, right );
        int i = left, j = right - 1;
        for( ; ; )
        {   while( a[ ++i ].compareTo( pivot ) < 0 ) { }
            while( a[ --j ].compareTo( pivot ) > 0 ) { }
            if( i < j )
                swapReferences( a, i, j );
            else break;
        }
        swapReferences( a, i, right - 1 );
        quicksort( a, left, i - 1 );
        quicksort( a, i + 1, right );
    }
    else
        insertionSort( a, left, right );
}
```

7.4 选择排序

方法：1.直接选择排序
2.锦标赛排序
3.堆排序

一、直接选择排序

思想：首先在 n 个记录中选出关键码最小（最大）的记录，然后与第一个记录（最后第 n 个记录）交换位置，再在其余的 $n-1$ 个记录中选关键码最小（最大）的记录，然后与第二个记录（第 $n-1$ 个记录）交换位置，直至选择了 $n-1$ 个记录。

| | 0 | 1 | 2 | 3 | 4 | 5 |
|-----|----|-----|-----|---------------|-------------|-------------|
| 例子: | 21 | 25 | 49 | 25* | 16 | <u>08</u> |
| | 08 | [25 | 49 | 25* | <u>16</u> | 21] |
| | 08 | 16 | [49 | 25* | 25 | <u>21</u>] |
| | 08 | 16 | 21 | [<u>25</u> * | 25 | 49] |
| | 08 | 16 | 21 | 25* | [<u>25</u> | 49] |
| | 08 | 16 | 21 | 25* | 25 | 49 |

算法:

```

template <class Type> void SelectSort(datalist<Type> &list)
{ for ( int i=0; i<list.CurrentSize-1; i++)
    SelectExchange(list, i);
}

```

```

template <class Type> void SelectExchange( datalist<Type> &
                                           list, const int i)
{
    int k=i;
    for ( int j=i+1; j<list.CurrentSize; j++)
        if (list.Vector[j].getkey()<list.Vector[k].getkey()) k=j;
    if ( k!=i) Swap(list.Vector[i], list.Vector[k]);
}

```

算法分析： 比较次数： $n-1+n-2+\dots+1=n(n-1)/2=O(n^2)$
与原始记录次序无关。

稳定性 ： 不稳定的。

二、锦标赛排序（树形选择排序）

直接选择排序存在重复做比较的情况，锦标赛排序克服了这一缺点。

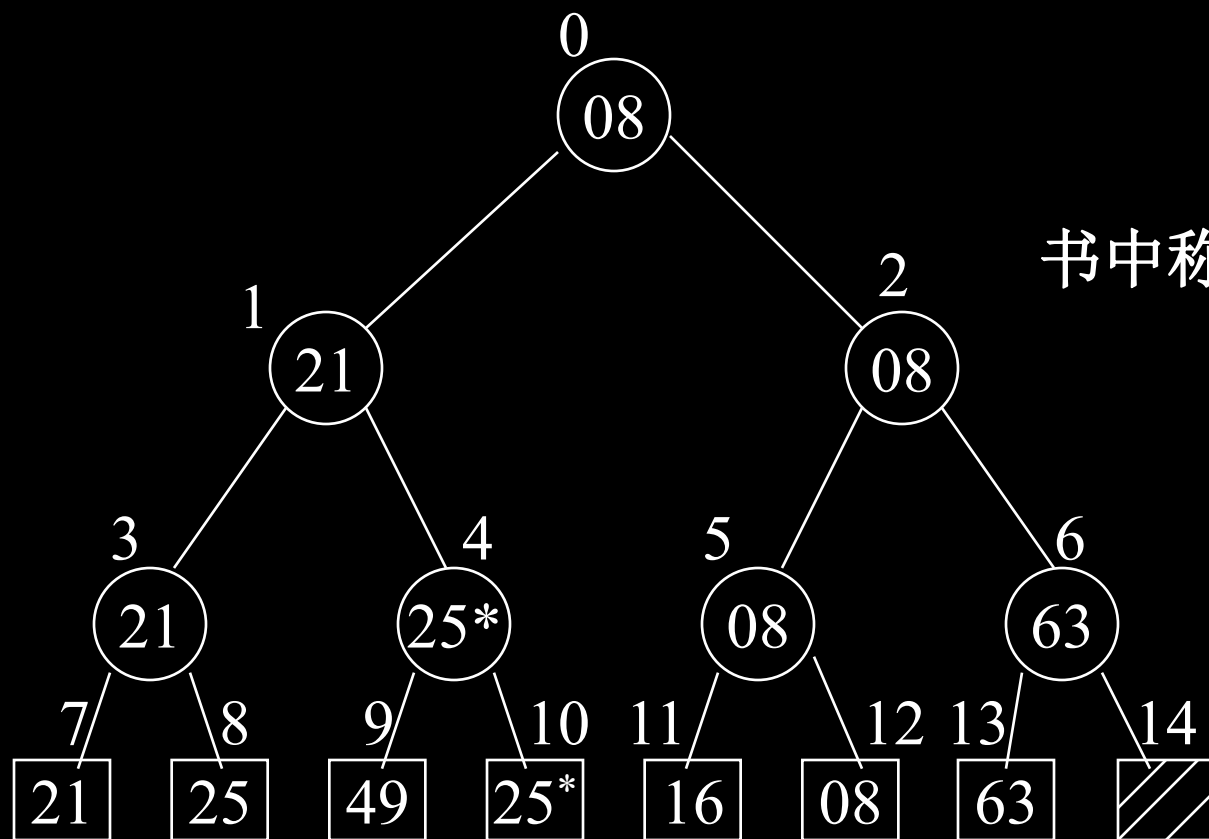
方法：

1. n 个对象的关键码两两比较得到 $\lceil n/2 \rceil$ 个 比较的优胜者(关键码小者)保留下来,再对这 $\lceil n/2 \rceil$ 个对象再进行关键码的两两比较,.....直至选出一个最小的关键码为止。

如果 n 不是2的 K 次幂, 则让叶结点数补足到满足 $2^k < n \leq 2^{k+1}$ 个。

2. 输出最小关键码。 再进行调整：

即把叶子结点上, 该最小关键码改为最大值后, 再进行由底向上的比较, 直至找到一个最小的关键码（即次小关键码）为止。重复2, 直至把关键码排好序。



书中称为胜者树

$$n_0 = n_2 + 1$$

算法分析： 树的深度 $\lceil \log_2 n \rceil$ 。总的比较次数

$$(n-1) + (n-1)(\lceil \log_2 n \rceil - 1) \approx (n-1) * \lceil \log_2 n \rceil = O(n \log_2 n)$$

↑ 建树 但需要 $n-1$ 个附加空间。而且 n 要满足 $2^{k-1} < n \leq 2^k$

算法实现：

每个结点： **data**—关键码

Active—表示此对象(结点)是否要参选

1:要 参选； 0: 不要参选

index—表示此对象在完全二叉树中的
序号

算法是稳定的。

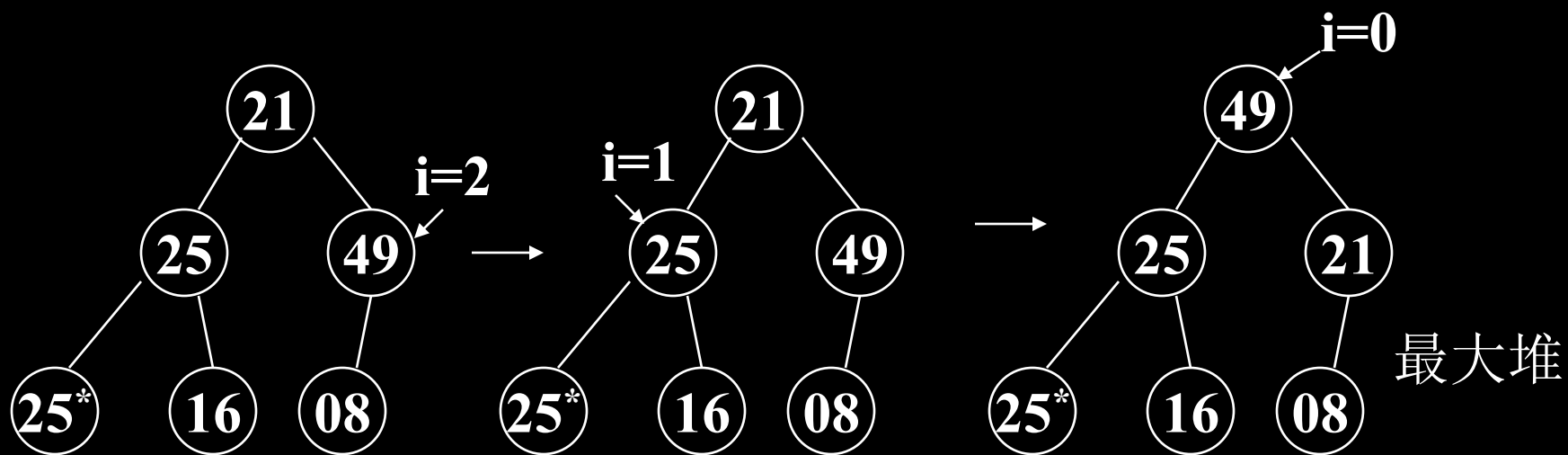
为了实现不要重复比较， 又不要增加额外的存储，
下面介绍堆排序。

三、堆排序（由J.W.J.Willman提出的）

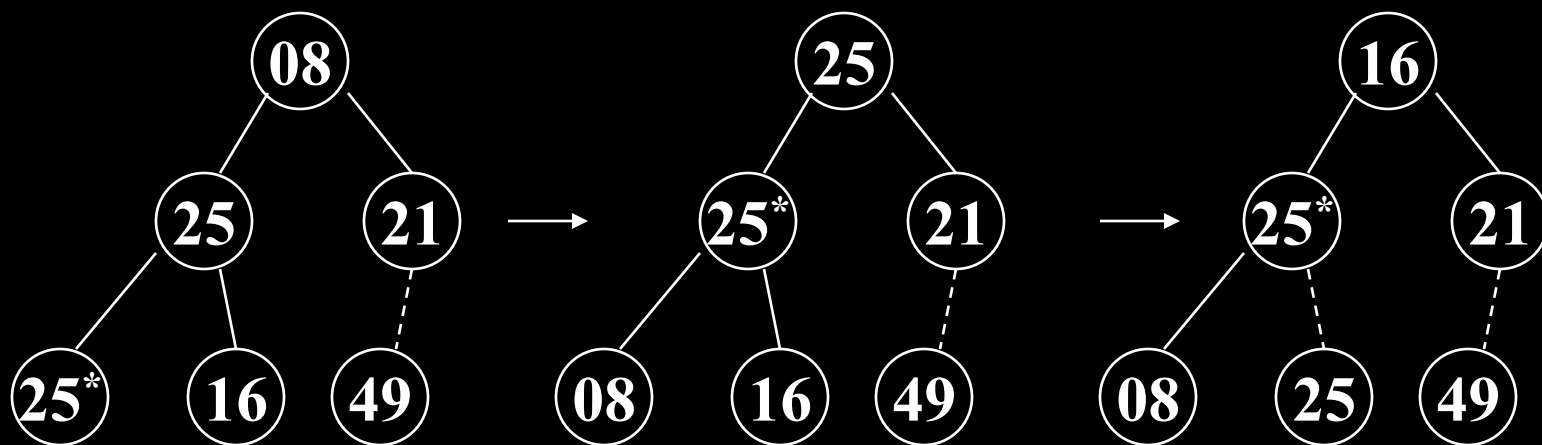
第6章已经介绍了堆结构和形成堆的算法。

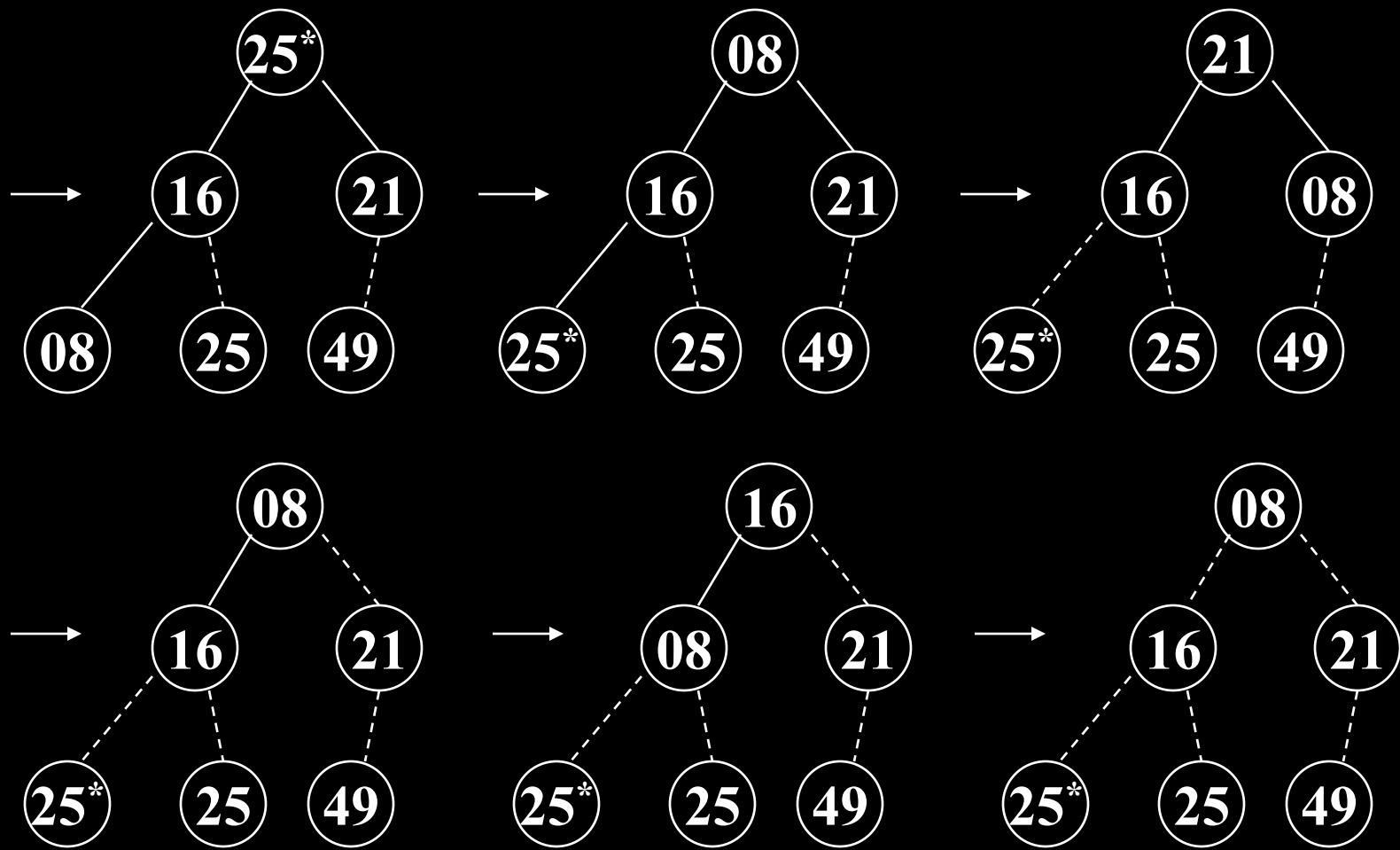
1.思想： 第一步，建堆，根据初始输入数据，利用堆的调整算法FilterDown（），形成初始堆。（形成最大堆）
第二步，一系列的对象交换和重新调整堆

2.例子： 书中的例子{21 25 49 25* 16 08}
 $i = \lfloor (n-1) / 2 \rfloor = \lfloor 5/2 \rfloor = 2, 1, 0$ 进行FilterDown（）



调整:





从以上例子可以看出堆排序是不稳定的

3.算法

```
Template<class  Type>void HeapSort(datalist<Type>&list)
{for(int i=(list.currentsize-1)/2;i>=0;i--)
    FilterDown(i,list.currentsize-1);
  for(i=list.currentsize-1;i>=1;i--)
    {Swap(list.Vector[0],list.vector[i]);
      FilterDown(0,i-1);
    }
}
```

其中，FilterDown()就是第6章中的，但要改一下：那里是建最小堆，这里是建最大堆。

Heap sort

java program

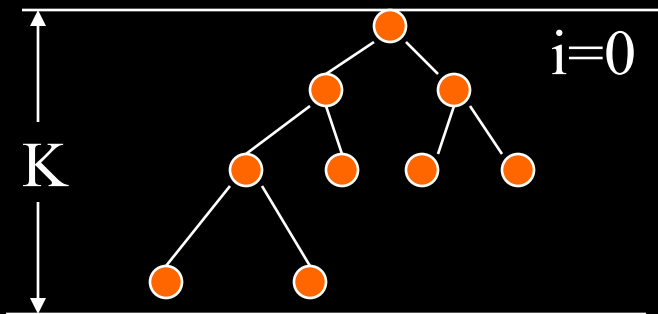
```
public static void heapsort( Comparable [ ] a )  
{ for( int i = a.length / 2; i >= 0; i-- )  
    percDown( a, i, a.length );  
    for( int i = a.length - 1; i > 0; i-- )  
        { swapReferences( a, 0, i );  
            percDown( a, 0, i);  
        }  
    }  
  
private static int leftChild( int i )  
{ return 2 * i + 1;  
}
```

Heap sort

```
private static void percDown( Comparable [ ] a, int i, int n )
{   int child;
    Comparable tmp;
    for( tmp = a[ i ]; leftChild( i ) < n ; i = child )
    {   child = leftchild( i );
        if( child != n - 1 && a[ child ] . compareTo( a[ child + 1 ] ) < 0 )
            child ++;
        if( tmp . compareTo( a[ child ] < 0 )
            a[ i ] = a[ child ];
        else
            break;
    }
    a[ i ] = tmp;
}
```


4.算法分析

初始建堆： n 个结点， $K=\lfloor \log_2 n \rfloor$ ， 从0层开始



第 i 层交换的最大次数为 $k-i$
第 i 层有 2^i 个结点

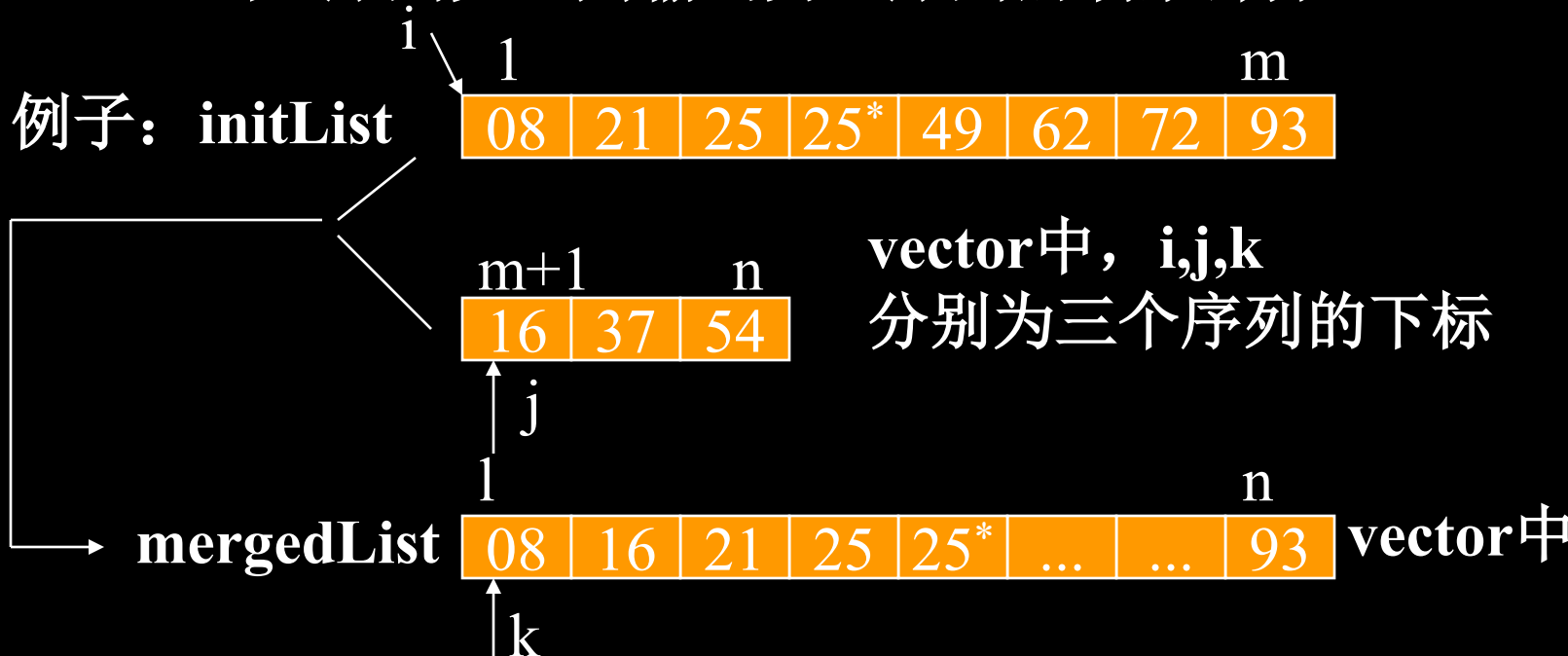
$$\begin{aligned} \text{总交换次数: } \sum_{i=0}^{k-1} 2^i \cdot (k-i) &= \sum_{j=1}^k j \cdot 2^{k-j} = \sum_{j=1}^k j(2^k \cdot 2^{-j}) \\ &\quad \uparrow \\ &\quad \text{令 } k-i=j \\ &= 2^k \cdot \sum_{j=1}^k j \cdot 2^{-j} \leq 2^k \cdot 2 \leq 2^{\log_2 n} \cdot 2 = 2n = O(n) \end{aligned}$$

调整 $n-1$ 次 `FilterDown()`， 时间为 $O(n \log_2 n)$

所以， $O(n) + O(n \log_2 n) = O(n \log_2 n)$

7.5 归并排序 (merge sort)

- 一、归并：两个（多个）有序的文件组合成一个有序文件
方法：每次取出两个序列中的小的元素输出之；
当一序列完，则输出另一序列的剩余部分



```

template<class Type> void merge(datalist<Type> & initList, datalist<Type>
    & mergedList, const int l, const int m, const int n)
{   int i=l, j=m+1, k=1;
    while ( i<=m && j<=n )
        if (initList.Vector[i].getkey( )<initList.Vector[j].getkey( ))
            { mergedList.Vector[k]=initList.Vector[i]; i++;k++;}

        else {mergedList.Vector[k]=initList.Vector[j]; j++; k++;}
    if ( i<=m)
        for (int n1=k, n2=i; n1<=n && n2<=m; n1++, n2++)
            mergedList.Vector[n1]=initList.Vector[n2];

    else
        for (int n1=k, n2=j; n1<=n && n2<=n; n1++, n2++)
            mergedList.Vector[n1]=initList.Vector[n2];
}

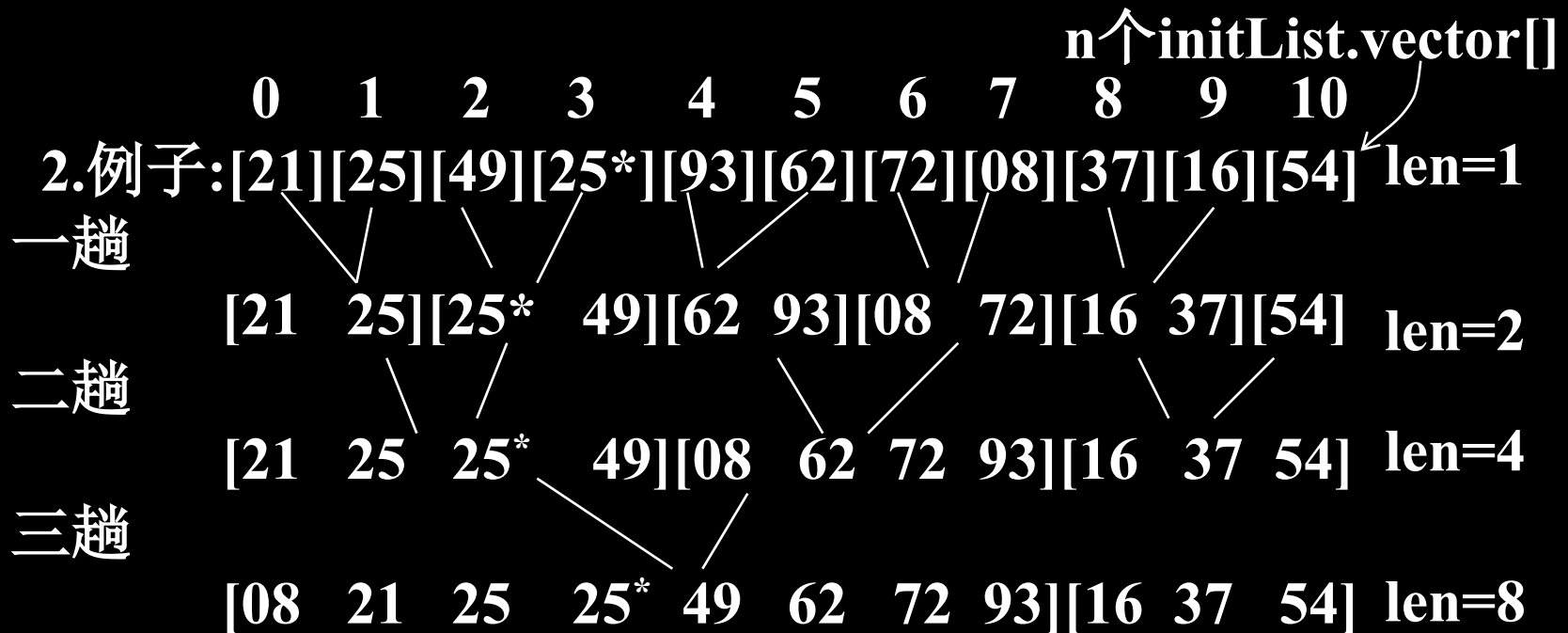
```

二、迭代的归并排序算法

例子: 21 25 49 25* 93 62 72 08 37 16 54

1.方法:

n个长为1的对象两两合并, 得 $n/2$ 个长为2的文件
 $n/2$ 个长为2.....得 $n/4$ 个长为4的文件
:
2个长为 $n/2$ 的对象两两合并,得1个长为n的文件



四趟

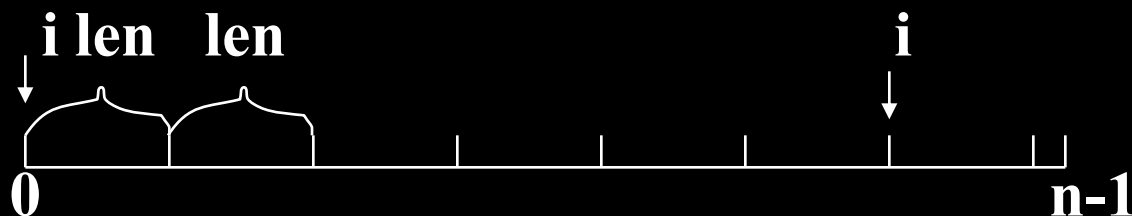
| | | | | | | | | | | | |
|-----|----|----|-----|-----|----|----|-----|-----|----|-----|--------|
| [08 | 21 | 25 | 25* | 49 | 62 | 72 | 93] | [16 | 37 | 54] | len=8 |
| | | | | | | | | | | | |
| [08 | 16 | 21 | 25 | 25* | 37 | 49 | 54 | 62 | 72 | 93] | len=16 |

3.算法:

主程序（多趟）→ 一趟 → 多次merge

```
template <class Type> void MergeSort(datalist <Type> & list)
{ datalist <Type> tempList(list.MaxSize);
  int len=1;
  while (len<list.CurrentSize)
  { MergePass(list, tempList, len); len *=2 ;
    if (len >= list.CurrentSize)
    { for (int i=0 ; i< list.CurrentSize; i++)
        list.Vector[i]=tempList.Vector[i];
      }
    else { MergePass (tempList, list, len); len*=2;}
  }
  delete[ ]tempList;
}
```

一趟归并算法: `mergepass(initList, mergedList, len)`



当两段均满len长时调用merge

当一长一短时也调用merge(但第二段的参数不同)

当只有一段时，则复抄

```

template <class Type> void MergePass( datalist<Type> & initList, datalist
                                     <Type> & mergedList, const int len)
{   int i=0;
    while (i+2*len<=initList.CurrentSize-1)
    {   merge( initList, mergedList, i, i+len-1, i+2*len-1);
        i+=2*len;
    }
    if (i+len <= initList.CurrentSize-1)
        merge( initList, mergedList, i, i+len-1, initList.CurrentSize-1);
    else for ( int j=i ; j<= initList.CurrentSize; j++)
            mergedList.Vector[j]=initList.Vector[j];
}

```

4.算法分析：合并趟数 $\log_2 n$,每趟比较 n 次，所以为 $O(n\log_2 n)$

5.稳定性：稳定。

Merge sort

java program (递归的归并排序算法)

```
public static void mergeSort( Comparable [ ] a )  
{   Comparable [ ] tmpArray = new Comparable[ a.length ];  
    mergeSort( a, tmpArray, 0, a.length - 1 );  
}
```

```
private static void mergeSort( Comparable [ ] a, Comparable [ ] tmpArray,  
                                int left, int right )  
{   if( left < right )  
    {   int center = ( left + right ) / 2;  
        mergeSort( a, tmparray, left, center );  
        mergeSort( a, tmpArray, center + 1, right );  
        merge( a, tmpArray, left, center + 1, right );  
    }  
}
```


Merge sort

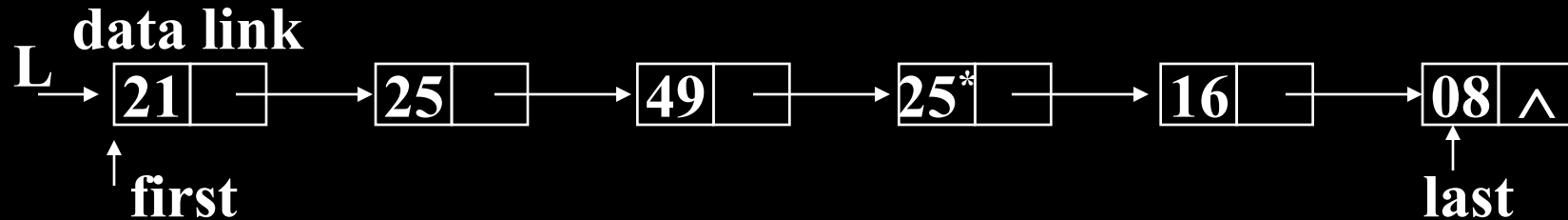
```
private static void merge( Comparable [ ] a, Comparable [ ] tmpArray,  
                           int leftPos, int rightPos, int rightEnd )  
{  int leftEnd = rightPos - 1;  
    int tmpPos = leftPos;  
    int numElements = rightEnd - leftPos + 1;  
    while( leftPos <= leftEnd && rightPos <= rightEnd )  
        if( a[ leftPos ].compareTo( a[ rightPos ] ) <= 0 )  
            tmpArray[ tmpPos++ ] = a[ leftPos++ ];  
        else tmpArray[ tmpPos++ ] = a[ rightPos++ ];  
    while( leftPos <= leftEnd )  
        tmpArray[ tmpPos++ ] = a[ leftPos++ ];  
    while( rightpos <= rightEnd)  
        tmpArray[ tmpPos++ ] = a[ rightpos++ ];  
    for( int i = 0; i < numElements; i++, rightEnd-- )  
        a[ rightEnd ] = tmpArray[ rightEnd ];  
}
```

三、递归的表归并排序

书中用静态链表的方法来实现

| | key | other | link |
|---|-----|-------|------|
| 0 | | | |
| 1 | 21 | | |
| 2 | 25 | | |
| 3 | 49 | | |
| | ⋮ | | |

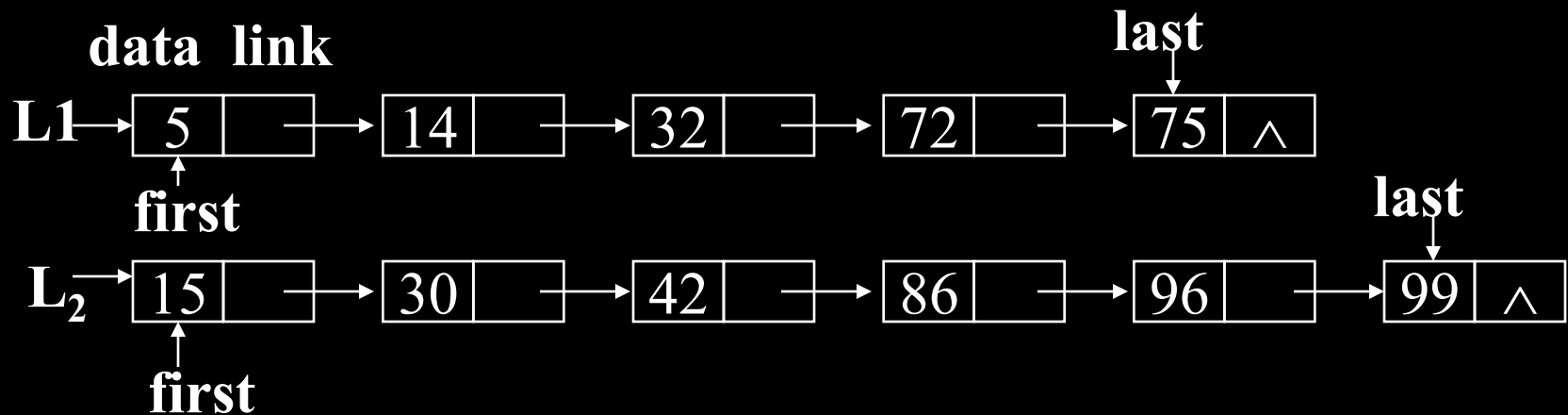
这里介绍用一般链表来实现归并排序（递归）



- 1.主程序 mergesort(L)
- 2.子程序 divide(L,L₁) 将L划分成两个子表
- 3.合并两有序序列 merge(L,L₁)

```
void MergeSort (List <Type> &L)
{
    List <Type> L1;
    if (L.first!=NULL)
        if (L.first->link != NULL)    //至少有两个结点
        {
            divide (L, L1);
            MergeSort(L);
            MergeSort(L1);
            L=merge( L, L1);
        }
}
```

下面讨论两个有序链表的 merge 算法



```

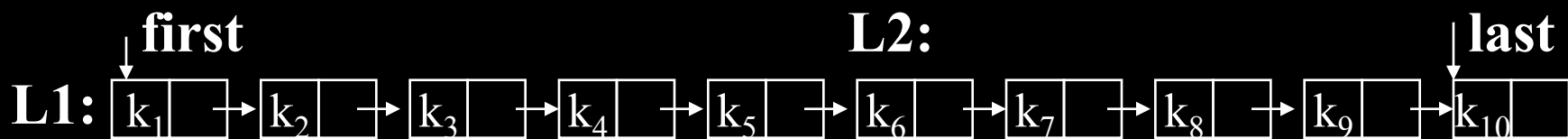
List<Type> & merge (List<Type> &L1, List<Type> & L2)
{  ListNode<Type>*p=L1.first, *q=L2.first, *r ;
   List<Type> temp ;
   1. if ((p==NULL) or (q==NULL))
   {   if (p!=NULL){temp.first=p; temp.last=L1.last;}
       else{temp.first=q ; temp.last=L2.last;}
   }
   else
   2. { 1) if (p->data<=q->data) {r = p ; p = p->link;}
        else{ r = q ; q = q->link ;}
        2) temp.first = r ;
        3) while((p!=NULL) && (q!=NULL))
        { if (p->data<=q->data)
            {r->link=p ; r=p ; p=p->link ; }
            else {r->link=q ; r=q ; q=q->link ; }
        }
        4) if (p==NULL) {r->link=q ; temp.last=L2.last ;}
            else {r->link=p ; temp.last=L1.last ;}
   }
   3. return temp ;
}

```

下面讨论 `divide(List<type>&L1,List<type>L2)`

将L1表分为两个长度几乎相等的表，L1.first指向前半部分，L2.first指向后半部分，要求被划分的表至少含有两个结点。

如何做？



方法：设两个流动指针p,q指向表的结点
一般来讲让p前进一步，q前进二步，最后当q=NULL时，这时p恰好指向前半张表的最后一个结点。

如果有10个结点，p走5次，q走10次正好走到表末尾。

```
void divide(List<Type> & L1, List <Type> & L2)
{  ListNode <Type> *p, *q; L2.last=L1.last;
   p=L1.first;
   q=p->link; q=q->link;
   while (q!=NULL)
   {  p=p->link;
      q=q->link;
      if (q!=NULL) q=q->link;
   }
   q=p->link ; p->link=NULL ; L1.last=p; L2.first=q;
}
```

第7章 内排序小结

| 排序方法 | 比较次数 | | 平均比较次数 | 稳定性 | 移动次数 | | 辅助存储 |
|---------|----------------|---------|----------------|-----|----------|---------|--------|
| | 最小 | 最大 | | | 最小 | 最大 | |
| 1、插入排序 | | | | | | | |
| 直接插入排序 | n | $n^2/2$ | $O(n^2)$ | 稳定 | $2n$ | $n^2/2$ | $O(1)$ |
| 二分法插入排序 | $n\log_2 n$ | | $O(n\log_2 n)$ | 稳定 | $2n$ | $n^2/2$ | $O(1)$ |
| 表插入排序 | n | $n^2/2$ | $O(n^2)$ | 稳定 | 0 | 0 | $O(n)$ |
| shell排序 | | | | 不稳定 | | | $O(1)$ |
| 2、选择排序 | | | | | | | |
| 直接选择排序 | $n(n-1)/2$ | | $O(n^2)$ | 不稳定 | $3(n-1)$ | | $O(1)$ |
| 堆排序 | $O(n\log_2 n)$ | | $O(n\log_2 n)$ | 不稳定 | | | $O(1)$ |

| 3、交换排序 | | | | | | | |
|-------------|-----------------------|------------|-----------------------|-----|-------------------|--------------------|---------------|
| Bubble Sort | $n-1$ | $n(n-1)/2$ | $O(n^2)$ | 稳定 | 0 | $3/2 \cdot n(n-1)$ | $O(1)$ |
| Quick Sort | $n \log_2 n$ | $n^2/2$ | $O(n \log_2 n)$ | 不稳定 | $\leq n \log_2 n$ | | $O(\log_2 n)$ |
| 4、归并排序 | $O(n \cdot \log_2 n)$ | | $O(n \cdot \log_2 n)$ | 稳定 | | | $O(n)$ |

以上量的分析与具体的算法、原始记录在机内存放方式、原始记录的排序情况有关。但一般取平均的比较次数。稳定性也有这个问题，不太严格。

Chapter 7

2009年统考题:

9. 若数据元素序列 11, 12, 13, 7, 8, 9, 23, 4, 5 是采用下列排序方法之一得到的第二趟排序后的结果, 则该排序算法只能是
- A. 起泡排序 B. 插入排序 C. 选择排序 D. 二路归并排序

Chapter 7

exercises:

1. Sort the sequence 3, 1, 4, 1, 5, 9, 2, 6, 5 using insertion sort.
2. Show the result of running Shellsort on the input 9, 8, 7, 6, 5, 4, 3, 2, 1 using the increments { 1, 3, 7 }.
3. Show how heapsort processes the input 142, 543, 123, 65, 453, 879, 572, 434, 111, 242, 811, 102.
4. Rewrite heapsort so that it sorts only items that are in the range low to high which are passed as additional parameters.
5. Sort 3, 1, 4, 1, 5, 9, 2, 6 using mergesort.