第 9 章 排序

静态数据表类定义

```
#include <iostream.h>
const int    DefaultSize = 100;
template <class Type> class dataList                          //数据表的前视声明

template <class Type> class Element {                         //数据表元素类的定义
friend class dataList <Type>;
private:
    Type key;                                                 //排序码
    field otherdata;                                          //其它数据成员
public:
    Type getKey ( ) { return key; }                           //取当前结点的排序码
    void setKey ( const Type x ) { key = x; }                 //将当前结点的排序码修改为 x
    Element<Type>& operator = ( Element<Type>& x )            //结点 x 的值赋给 this
        { key = x->key;    otherdata = x->otherdata; }
    int operator == ( Type& x ) { return key == x->key; }     //判 this 与 x 相等
    int operator <= ( Type& x ) { return key <= x->key; }     //判 this 小于或等于 x
    int operator > ( Type& x ) { return key > x->key; }       //判 this 大于 x
    int operator < ( Type& x ) { return key > x->key; }       //判 this 小于 x
}

template <class Type> class dataList {
//用顺序表来存储待排序的元素，这些元素的类型是 Type
private:
    Element <Type> * Vector;                                  //存储待排序元素的向量
    int MaxSize, CurrentSize;                                 //最大元素个数与当前元素个数
    int Partition ( const int low, const int high )           //用于快速排序的一次划分算法
public:
    datalist ( int MaxSz = DefaultSize ) : MaxSize ( Maxsz ), CurrentSize (0)
        { Vector = new Element <Type> [MaxSize]; }            //构造函数
    int length ( ) { return CurrentSize; }
    Element<Type>& operator [ ] ( int i ) { return Vector[i]; }
    void swap ( Element <Type>& x, Element <Type>& y )        //交换 x, y
        { Element <Type> temp = x;    x = y;    y = temp; }
    void Sort ( );                                            //排序
}
```

静态链表类定义

```
template <class Type> class staticlinkList;          //静态链表类的前视声明

template <class Type> class Element {                //静态链表元素类的定义
friend class staticlinkList<Type>;
private:
```

```
    Type key;                                   //排序码，其它信息略
    int link;                                   //结点的链接指针
public:
    Type getKey ( ) { return key; }             //取当前结点的排序码
    void setKey ( const Type x ) { key = x; }   //将当前结点的排序码修改为 x
    int getLink ( ) { return link; }            //取当前结点的链接指针
    void setLink ( const int ptr ) { link = ptr; }  //将当前结点的链接指针置为 ptr
}

template <class Type> class staticlinkList {    //静态链表的类定义
private:
    Element <Type> *Vector;                     //存储待排序元素的向量
    int MaxSize, CurrentSize;                   //向量中最大元素个数和当前元素个数
public:
    dstaticlinkList ( int Maxsz = DefaultSize ) : MaxSize ( Maxsz ), CurrentSize (0)
        { Vector = new Element <Type> [Maxsz]; }
}
```

9-1 什么是内排序？什么是外排序？什么排序方法是稳定的？什么排序方法是不稳定的？

【解答】内排序是排序过程中参与排序的数据全部在内存中所做的排序，排序过程中无需进行内外存数据传送，决定排序方法时间性能的主要是数据排序码的比较次数和数据对象的移动次数。外排序是在排序的过程中参与排序的数据太多，在内存中容纳不下，因此在排序过程中需要不断进行内外存的信息传送的排序。决定外排序时间性能的主要是读写磁盘次数和在内存中总的记录对象的归并次数。

不稳定的排序方法主要有希尔排序、直接选择排序、堆排序、快速排序。不稳定的排序方法往往是按一定的间隔移动或交换记录对象的位置，从而可能导致具有相等排序码的不同对象的前后相对位置在排序前后颠倒过来。其他排序方法中如果有数据交换，只是在相邻的数据对象间比较排序码，如果发生逆序(与最终排序的顺序相反的次序)才交换，因此具有相等排序码的不同对象的前后相对位置在排序前后不会颠倒，是稳定的排序方法。但如果把算法中判断逆序的比较“>(或<)”改写成“≥(或≤)”，也可能造成不稳定。

9-2 设待排序的排序码序列为{12, 2, 16, 30, 28, 10, 16*, 20, 6, 18}，试分别写出使用以下排序方法每趟排序后的结果。并说明做了多少次排序码比较。

(1) 直接插入排序　(2) 希尔排序(增量为 5,2,1)　(3) 起泡排序
(4) 快速排序　(5) 直接选择排序　(6) 锦标赛排序
(7) 堆排序　(8) 二路归并排序　(9) 基数排序

【解答】

(1) 直接插入排序

初始排列	0	1	2	3	4	5	6	7	8	9	排序码比较次数
i = 1	[12]	2	16	30	28	10	16*	20	6	18	1
i = 2	[2	12]	16	30	28	10	16*	20	6	18	1
i = 3	[2	12	16]	30	28	10	16*	20	6	18	1
i = 4	[2	12	16	30]	28	10	16*	20	6	18	2
i = 5	[2	12	16	28	30]	10	16*	20	6	18	5
i = 6	[2	10	12	16	28	30]	16*	20	6	18	3
i = 7	[2	10	12	16	16*	28	30]	20	6	18	3
i = 8	[2	10	12	16	16*	20	28	30]	6	18	3

i = 9	[2	6	10	12	16	16*	20	28	30]	18		8
	[2	6	10	12	16	16*	18	20	28	30]		

(2) 希尔排序(增量为 5,2,1)

初始排列	0	1	2	3	4	5	6	7	8	9	排序码比较次数
	12	2	16	30	28	10	16*	20	6	18	1+1+1+1+1 = 5
d = 5											
	10	2	16	6	18	12	16*	20	30	28	(1+1+2+1) + (1+1
d = 2											+1+1) = 9
	10	2	16	6	16*	12	18	20	30	28	1+1+3+1+3+1+1
d = 1											+1+2 = 14
	2	6	10	12	16	16*	18	20	28	30	

希尔(shell)本人采取的增量序列为 $\lfloor n/2 \rfloor, \lfloor \lfloor n/2 \rfloor /2 \rfloor, \lfloor \lfloor n/2 \rfloor /2 \rfloor /2 \rfloor, \cdots, 1$。一般地，增量序列可采用 $\lfloor n\alpha \rfloor, \lfloor \lfloor n\alpha \rfloor \alpha \rfloor, \lfloor \lfloor n\alpha \rfloor \alpha \rfloor \alpha \rfloor, \cdots, 1$。大量实验表明，取 $\alpha = 0.45454$ 的增量序列比取其他的增量序列的优越性更显著。计算 $\lfloor 0.45454n \rfloor$ 的一个简单方法是用整数算术计算 (5*n−1)/11。需要注意，当$\alpha < 1/2$ 时，增量序列可能不以 1 结束，需要加以判断和调整。

(3) 起泡排序

初始排列	0	1	2	3	4	5	6	7	8	9	排序码比较次数
i = 0	[12	2	16	30	28	10	16*	20	6	18]	9
i = 1	2	[12	6	16	30	28	10	16*	20	18]	8
i = 2	2	6	[12	10	16	30	28	16*	18	20]	7
i = 3	2	6	10	[12	16	16*	30	28	18	20]	6
i = 4	2	6	10	12	[16	16*	18	30	28	20]	5
i = 5	2	6	10	12	16	[16*	18	20	30	28]	4
i = 6	2	6	10	12	16	16*	[18	20	28	30]	3
	2	6	10	12	16	16*	18	20	28	30	

(4) 快速排序

Pivot	Pvtpos	0	1	2	3	4	5	6	7	8	9	排序码比较次数
12	0,1,2,3	**[12** ↑pos	**2** ↑pos	**16** ↑pos	**30** ↑pos	**28**	**10**	**16***	**20**	**6**	**18]**	9
6	0,1	**[6** ↑pos	**2** ↑pos	**10]**	12	[28	16	16*	20	30	18]	2
28	4,5,6,7,8	[2]	6	[10]	12	**[28** ↑pos	**16** ↑pos	**16*** ↑pos	**20** ↑pos	**30** ↑pos	**18]**	5
18	4,5,6	2	6	10	12	**[18** ↑pos	**16** ↑pos	**16*** ↑pos	**20]**	28	[30]	3
16*	4	2	6	10	12	**[16** * ↑pos	**16]**	18	[20]	28	30	1

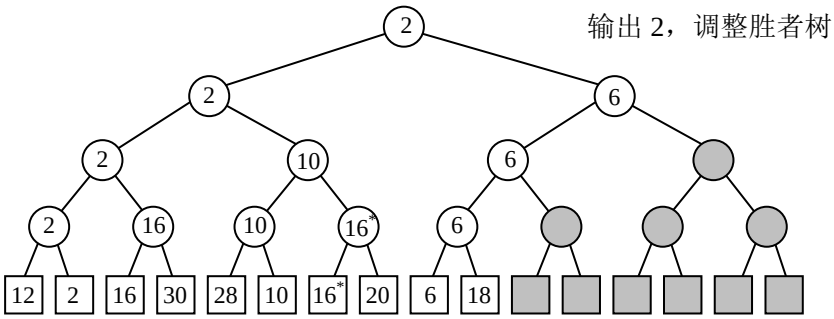
2 6 10 12 16* [16] 18 20 28 30

左子序列递归深度为 1，右子序列递归深度为 3。

(5) 直接选择排序

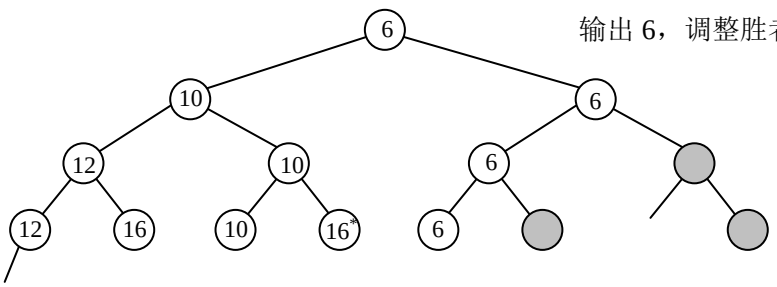
初始排列	0	1	2	3	4	5	6	7	8	9	排序码比较次数
i = 0	[12	2	16	30	28	10	16*	20	6	18]	9
i = 1	2	[12	16	30	28	10	16*	20	6	18]	8
i = 2	2	6	[16	30	28	10	16*	20	12	18]	7
i = 3	2	6	10	[30	28	16	16*	20	12	18]	6
i = 4	2	6	10	12	[28	16	16*	20	30	18]	5
i = 5	2	6	10	12	16	[28	16*	20	30	18]	4
i = 6	2	6	10	12	16	16*	[28	20	30	18]	3
i = 7	2	6	10	12	16	16*	18	[20	30	28]	2
i = 8	2	6	10	12	16	16*	16	20	[30	28]	1
	2	6	10	12	16	16*	16	20	28	[30]	

(6) 锦标赛排序

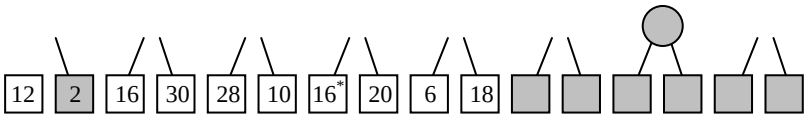


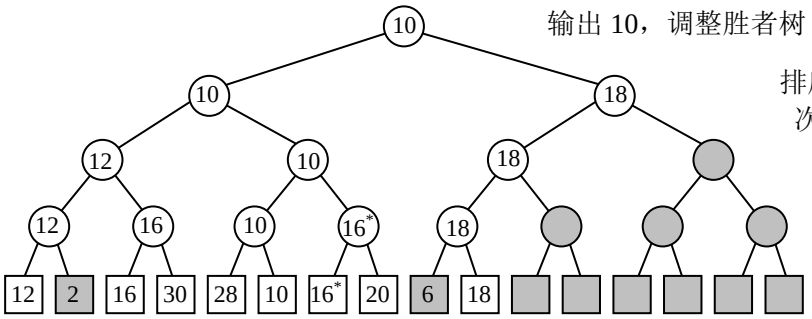
输出 2，调整胜者树

当参加排序的数据对象个数 n 不足 2 的某次幂时，将其补足到 2 的某次幂。本题的 n = 10，将叶结点个数补足到 $2^4 = 16$ 个。排序码比较次数 = 9。



输出 6，调整胜者树

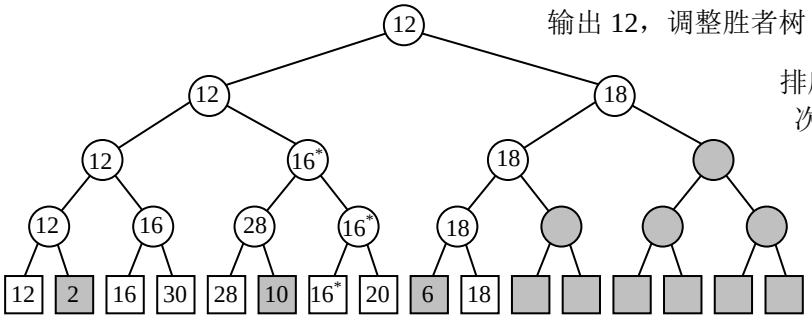




输出 10，调整胜者树

排序码比较次数 = 1。

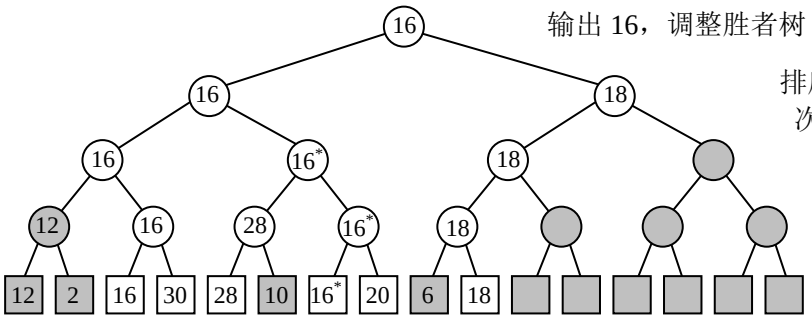
当某结点的比较对手的参选标志为“不再参选”，该结点自动升入双亲结点，此动作不计入排序码比较次数。



输出 12，调整胜者树

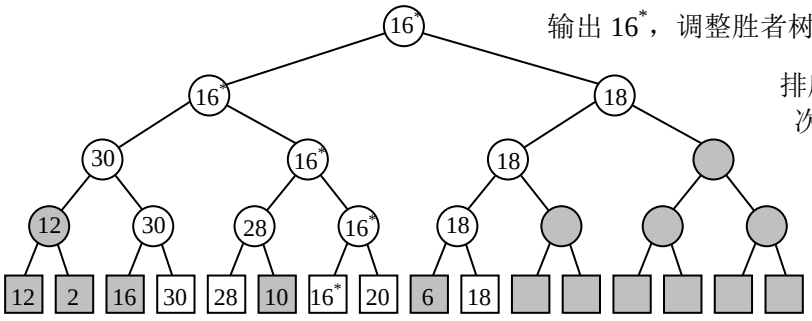
排序码比较次数 = 3。

排序码比较次数=3。某对象输出后，对手自动升到双亲，不计入排序码比较次数。



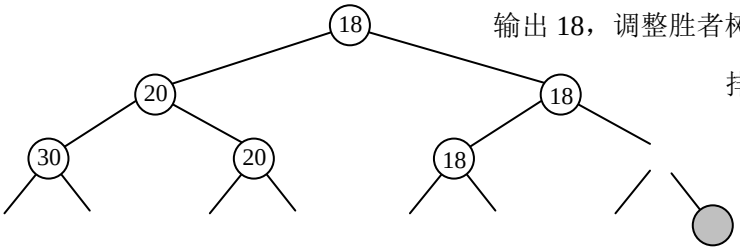
输出 16，调整胜者树

排序码比较次数 = 2。



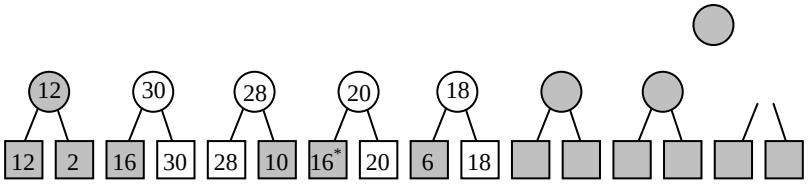
输出 16*，调整胜者树

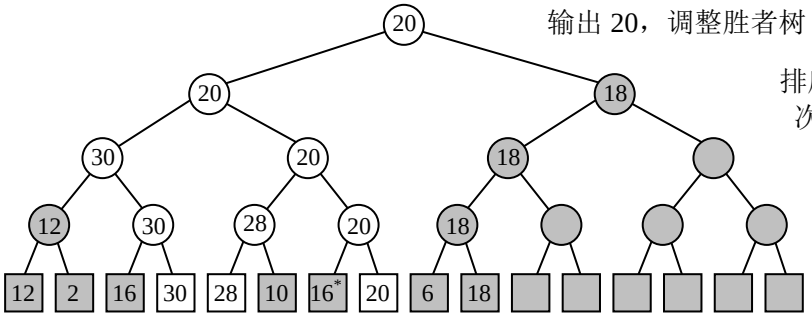
排序码比较次数 = 2。



输出 18，调整胜者树

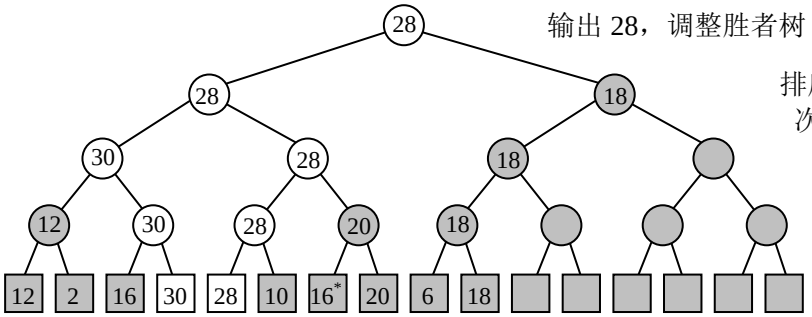
排序码比较次数 = 3。





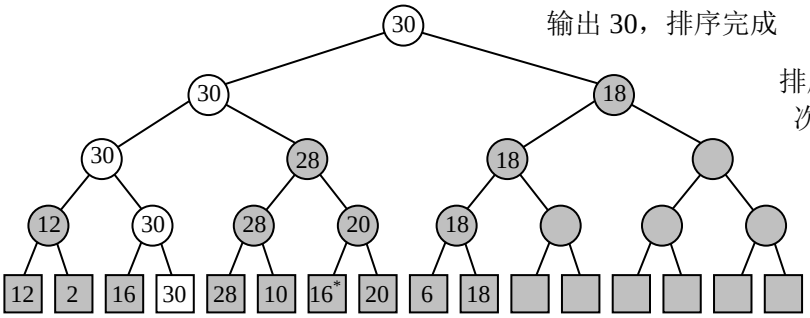
输出 20，调整胜者树

排序码比较
次数 = 0。



输出 28，调整胜者树

排序码比较
次数 = 2。

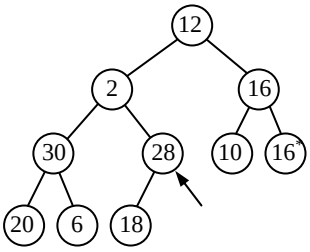


输出 30，排序完成

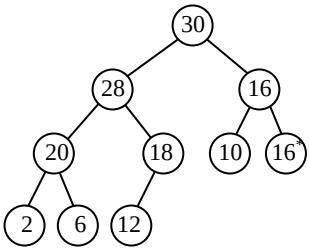
排序码比较
次数 = 0。

(7) 堆排序

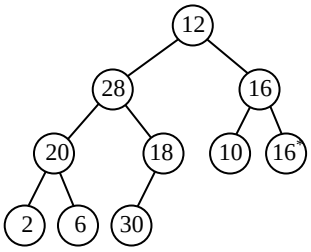
第一步，形成初始的最大堆 (略)，第二步，做堆排序。



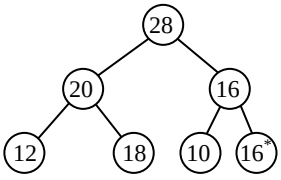
初始排列，不是最大堆

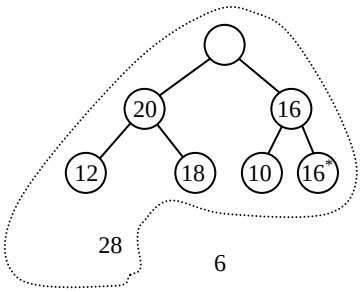


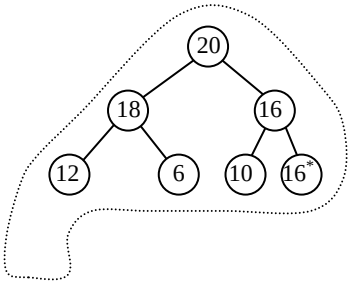
形成初始最大堆

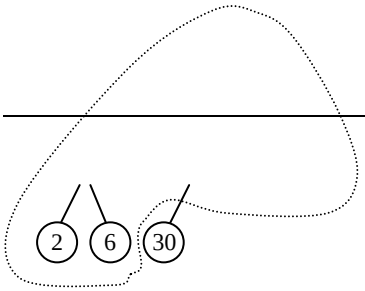


交换 $0^{\#}$ 与 $9^{\#}$ 对象

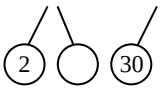




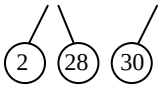




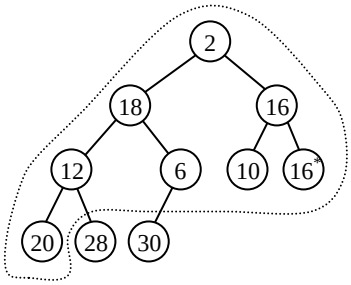
从 0# 到 8# 重新形成堆



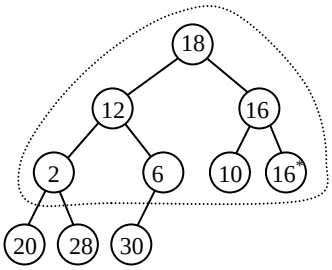
交换 0# 与 8# 对象



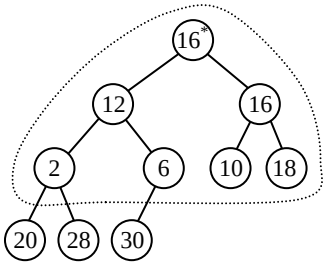
从 0# 到 7# 重新形成堆



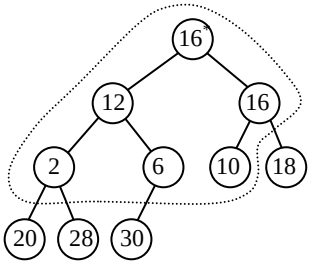
交换 0# 与 7# 对象



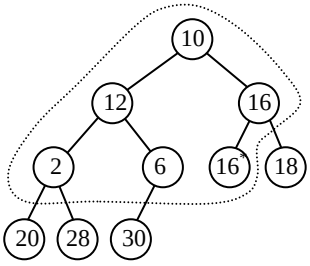
从 0# 到 6# 重新形成堆



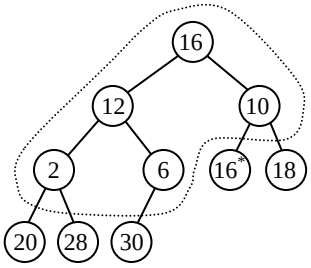
交换 0# 与 6# 对象



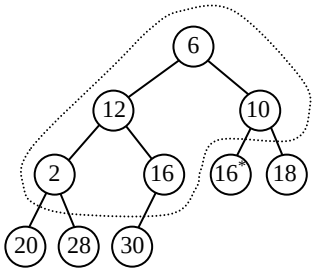
从 0# 到 5# 重新形成堆



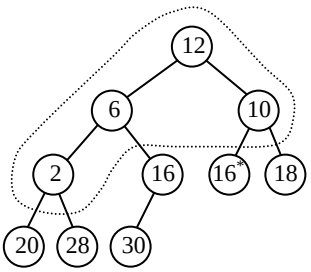
交换 0# 与 5# 对象



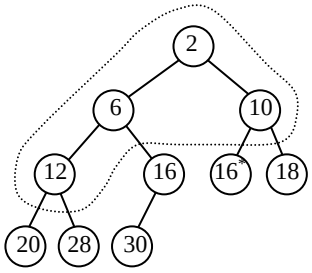
从 0# 到 4# 重新形成堆



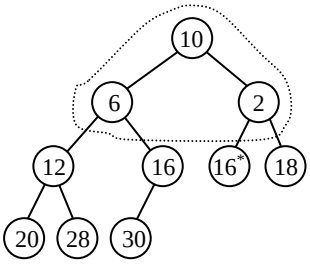
交换 0# 与 4# 对象



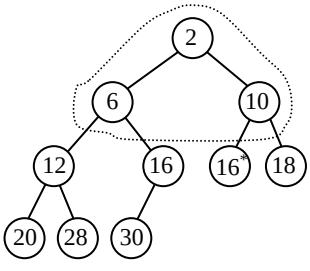
从 0# 到 3# 重新形成堆



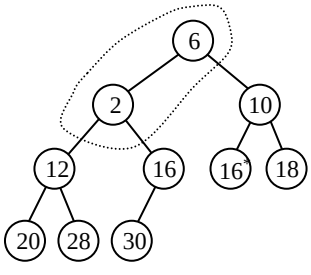
交换 0# 与 3# 对象



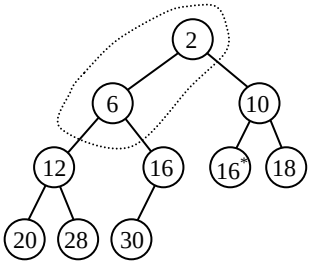
从 0# 到 2# 重新形成堆



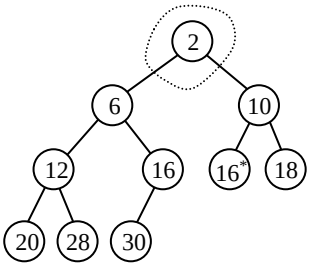
交换 0# 与 2# 对象



从 0# 到 1# 重新形成堆



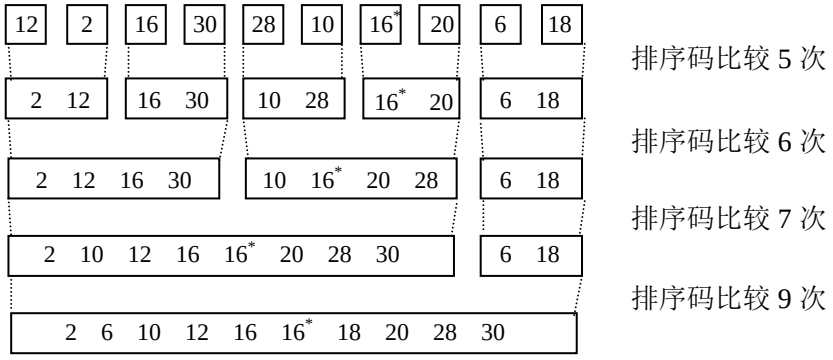
交换 0# 与 1# 对象



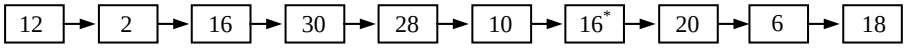
从 0# 到 1# 重新形成堆，得到结果

(8) 二路归并排序

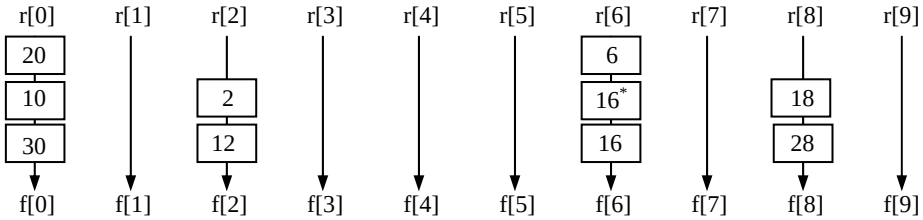
采用迭代的方法进行归并排序。设待排序的数据对象有 n 个。首先把每一个待排序的数据对象看作是长度为的初始归并项，然后进行两两归并，形成长度为 2 的归并项，再对它们两两归并，形成长度为 4 的归并项，如此一趟一趟做下去，最后得到长度为 n 的归并结果。

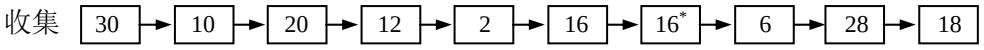


(9) 基数排序

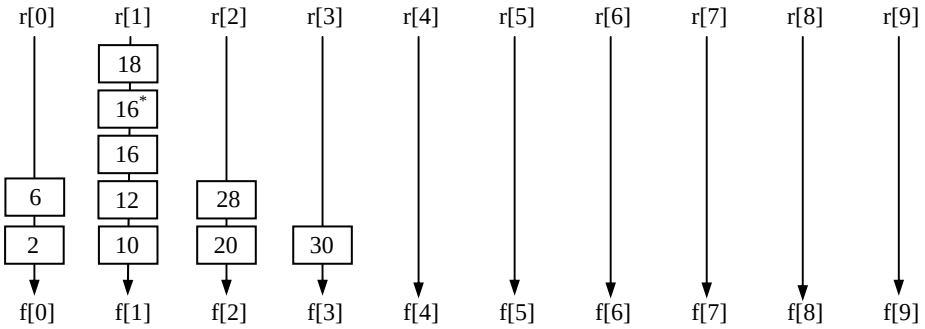


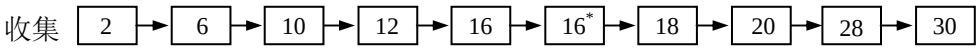
按最低位分配





按最高位分配

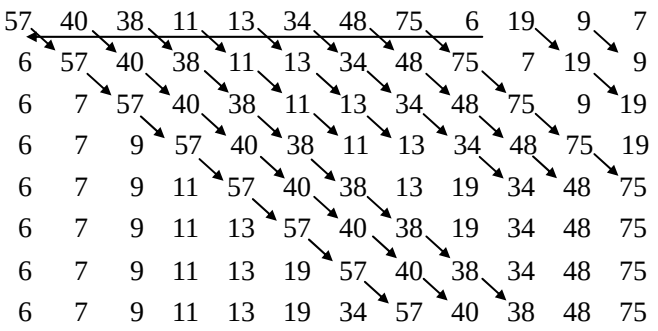




9-3 在起泡排序过程中，什么情况下排序码会朝向与排序相反的方向移动，试举例说明。在快速排序过程中有这种现象吗？

【解答】

如果在待排序序列的后面的若干排序码比前面的排序码小，则在起泡排序的过程中，排序码可能向与最终它应移向的位置相反的方向移动。例如，



9-4 试修改起泡排序算法，在正反两个方向交替进行扫描，即第一趟把排序码最大的对象放到序列的最后，第二趟把排序码最小的对象放到序列的最前面。如此反复进行。

【解答 1】

```
template <class Type> void dataList<Type> :: shaker_Sort ( ) {
//奇数趟对表 Vector 从前向后，比较相邻的排序码，遇到逆序即交换，直到把参加比较排序码序列
//中最大的排序码移到序列尾部。偶数趟从后向前，比较相邻的排序码，遇到逆序即交换，直到把
//参加比较排序码序列中最小的排序码移到序列前端。
    int i = 1, j;    int exchange;
    while ( i < CurrentSize ) {                          //起泡排序趟数不超过 n-1
        exchange = 0;                                    //假定元素未交换
        for ( j = CurrentSize-i; j >= i; j-- )           //逆向起泡
            if ( Vector[j-1] > Vector[j] ) {             //发生逆序
                Swap ( Vector[j-1], Vector[j] );         //交换，最小排序码放在 Vector[i-1]处
                exchange = 1;                            //做“发生了交换”标志
            }
        if ( exchange == 0 ) break;                      ////当 exchange 为 0 则停止排序
        for ( j = i; j <= CurrentSize-i-1; j++ )         //正向起泡
            if ( Vector[j] > Vector[j+1] ) {             //发生逆序
                Swap ( Vector[j], Vector[j+1] );         //交换，最大排序码放在 Vector[n-i]处
                exchange = 1;                            //做“发生了交换”标志
            }
      if ( exchange == 0 ) break;                        //当 exchange 为 0 则停止排序
      i++;
    }
}
```

【解答 2】

```
template <class Type> void dataList<Type> :: shaker_Sort ( ) {
    int low = 1, high = CurrentSize-1, i, j;    int exchange;
    while ( low < high ) {                               //当比较范围多于一个对象时排序
        j = low;                                         //记忆元素交换位置
        for ( i = low; i < high; i++ )                   //正向起泡
            if ( Vector[i] > Vector[i+1] ) {             //发生逆序
                Swap ( Vector[i], Vector[i+1] );         //交换
                j = i;                                   //记忆右边最后发生交换的位置 j
            }
        high = j;                                        //比较范围上界缩小到 j
```

```
        for ( i = high; i > low; i--)                    //反向起泡
          if ( Vector[i-1] > Vector[i] ) {                //发生逆序
              Swap ( Vector[i-1], Vector[i] );            //交换
              j = i;                                      //记忆左边最后发生交换的位置 j
          }
        low = j;                                          //比较范围下界缩小到 j
      }
    }
```

9-5 如果待排序的排序码序列已经按非递减次序有序排列，试证明函数 QuickSort( )的计算时间将下降到 $O(n^2)$。

【证明】

若待排序的 n 个对象的序列已经按排序码非递减次序有序排列，且设排序的时间代价为 T(n)。当以第一个对象作为基准对象时，应用一次划分算法 Partition，通过 n-1 次排序码比较，把只能把整个序列划分为：基准对象的左子序列为空序列，右子序列为有 n-1 个对象的非递减有序序列。对于这样的递归 QuickSort( )算法，其时间代价为

$T(n) = (n-1) + T(n-1)$
$= (n-1) + \{(n-2) + T(n-2) \}$
$= (n-1) + (n-2) + \{(n-3) + T(n-3) \}$
$= \cdots\cdots$
$= (n-1) + (n-2) + (n-3) + \cdots + \{ 2 + T(1) \}$
$= (n-1) + (n-2) + (n-3) + \cdots + 2 + 1$
$= n(n-1)/2 = O(n^2)$

9-6 在实现快速排序的非递归算法时，可根据基准对象，将待排序排序码序列划分为两个子序列。若下一趟首先对较短的子序列进行排序，试证明在此做法下，快速排序所需要的栈的深度为 $O(\log_2 n)$。

【解答】

由快速排序的算法可知, 所需递归工作栈的深度取决于所需划分的最大次数。如果在排序过程中每次划分都能把整个待排序序列根据基准对象划分为左、右两个子序列。假定这两个子序列的长度相等，则所需栈的深度为

$S(n) = 1 + S(n/2) =$
$= 1 + \{ 1 + S(n/4) \} = 2 + S(n/4)$
$= 2 + \{ 1 + S(n/8) \} = 3 + S(n/8)$
$= \cdots\cdots$
$= \log_2 n + S(1) = \log_2 n$ (假设 1 个对象的序列所用递归栈的深度为 0)

如果每次递归左、右子序列的长度不等，并且先将较长的子序列的左、右端点保存在递归栈中，再对较短的子序列进行排序，可用表示最坏情况的大 O 表示法表示。此时其递归栈的深度不一定正好是 $\log_2 n$，其最坏情况为 $O(\log_2 n)$。

9-7 在实现快速排序算法时，可先检查位于两端及中点的排序码，取三者之中的数值不是最大也不是最小的排序码作为基准对象。试编写基于这种思想的快速排序算法，并证明对于已排序的排序码序列，该算法的计算时间为 $O(n\log_2 n)$。

【解答】参看教科书

9-8 在使用非递归方法实现快速排序时，通常要利用一个栈记忆待排序区间的两个端点。那么能否用队列来代替这个栈? 为什么?

【解答】

可以用队列来代替栈。在快速排序的过程中，通过一趟划分，可以把一个待排序区间分为两个子区间，然后分别对这两个子区间施行同样的划分。栈的作用是在处理一个子区间时，保存另一个子区间的上界和下界，待该区间处理完成后再从栈中取出另一子区间的边界，对

其进行处理。这个功能利用队列也可以实现，只不过是处理子区间的顺序有所变动而已。

9-9 试设计一个算法，使得在 O(n)的时间内重排数组，将所有取负值的排序码排在所有取正值(非负值)的排序码之前。

【解答】

```
template<class Type> void reArrange ( dataList<Type>& L ) {
//数组元素类型 Type 只可能取 int 或 float
    int i = 0, j = L.length () – 1;
    while ( i != j ) {
        while ( L[i].getKey( ) < 0 ) i++;
        while ( L[j].getKey( ) >= 0 ) j--;
        swap ( L[i], L[j] );
        i++;   j--;
    }
}
```

9-10 奇偶交换排序是另一种交换排序。它的第一趟对序列中的所有奇数项 i 扫描，第二趟对序列中的所有偶数项 i 扫描。若 A[i] > A[i+1]，则交换它们。第三趟有对所有的奇数项，第四趟对所有的偶数项，…，如此反复，直到整个序列全部排好序为止。

(1) 这种排序方法结束的条件是什么？

(2) 写出奇偶交换排序的算法。

(3) 当待排序排序码序列的初始排列是从小到大有序，或从大到小有序时，在奇偶交换排序过程中的排序码比较次数是多少？

【解答】

(1) 设有一个布尔变量 exchange，判断在每一次做过一趟奇数项扫描和一趟偶数项扫描后是否有过交换。若 exchange = 1，表示刚才有过交换，还需继续做下一趟奇数项扫描和一趟偶数项扫描；若 exchange = 0，表示刚才没有交换，可以结束排序。

(2) 奇偶排序的算法

```
template<Type> void dataList<Type> :: odd-evenSort ( ) {
    int i, exchange;
    do {
        exchange = 0;
        for ( i = 1; i < CurrentSize; i += 2 )          //扫描所有奇数项
          if ( Vector[i] > Vector[i+1] ) {              //相邻两项比较，发生逆序
              exchange = 1;                             //作交换标记
              swap ( Vector[i], Vector[i+1] );          //交换
          }
        for ( i = 0; i < CurrentSize; i += 2 )          //扫描所有偶数项
          if ( Vector[i] > Vector[i+1] ) {              //相邻两项比较，发生逆序
              exchange = 1;                             //作交换标记
              swap ( Vector[i], Vector[i+1] );          //交换
          }
    } while ( exchange != 0 );
}
```

(3) 设待排序对象序列中总共有 n 个对象。序列中各个对象的序号从 0 开始。则当所有

待排序对象序列中的对象按排序码从大到小初始排列时，执行 m = ⌊(n+1)/2⌋ 趟奇偶排序。当所有待排序对象序列中的对象按排序码从小到大初始排列时，执行 1 趟奇偶排序。

在一趟奇偶排序过程中，对所有奇数项扫描一遍，排序码比较 ⌊(n−1)/2⌋ 次；对所有偶数项扫描一遍，排序码比较 ⌊n/2⌋ 次。所以每趟奇偶排序两遍扫描的结果，排序码总比较次数为 ⌊(n−1)/2⌋+⌊n/2⌋ = n−1。

9−11 请编写一个算法，在基于单链表表示的待排序排序码序列上进行简单选择排序。

【解答】

采用静态单链表作为存储表示。用 Vector[0]作为表头结点，各待排序数据对象从 Vector[1]开始存放。算法的思想是每趟在原始链表中摘下排序码最大的结点(几个排序码相等时为最前面的结点)，把它插入到结果链表的最前端。由于在原始链表中摘下的排序码越来越小，在结果链表前端插入的排序码也越来越小，最后形成的结果链表中的结点将按排序码非递减的顺序有序链接。

```
Template<class Type> void staticlinkList<Type> :: selectSort ( ) {
    int h = Vector[0].link, p, q, r, s;
    Vector[0].link = 0;
    while ( h != 0 ) {                          //原始链表未扫描完
        p = s = h;   q = r = 0;
        while ( p != 0 ) {                          //扫描原始链表, 寻找排序码最大的结点 s
            if ( Vector[p].data > Vector[s].data )   //记忆当前找到的排序码最大结点
                { s = p;   r = q; }
            q = p;   p = Vector[p].link;
        }
        if ( s == h ) h = Vector[h];            //排序码最大的结点是原始链表前端结点, 摘下
        else Vector[r].link = Vector[s].link;   //排序码最大的结点是原始链表表中结点, 摘下
        Vector[s].link = Vector[0].link;        //结点 s 插入到结果链表的前端
        Vector[0].link = s;
    }
}
```

9−12 若参加锦标赛排序的排序码有 11 个，为了完成排序，至少需要多少次排序码比较？

【解答】

对于有 n(n>0)个数据的序列，锦标赛排序选最小数据需进行 n−1 次数据比较，以后每选一个数据，进行数据比较的次数，均需 $\lfloor \log_2 n \rfloor - 1$ 次（在外结点层无比较）。对于有 11 个排序码的序列，第一次选具最小排序码的数据，需进行 10 次排序码比较，以后在剩下的序列中每选一个具最小排序码的数据，都需进行 $\lfloor \log_2 11 \rfloor - 1 = 2$ 次排序码比较，因此，为了完成排序，需要 10 + 2*10 = 30 次排序码比较。

9−13 试给出适用于锦标赛排序的胜者树的类型声明。并写一个函数，对 n 个参加排序的对象，构造胜者树。设 n 是 2 的幂。

【解答】

适用于锦标赛排序的胜者树的类型声明.

```
template <class Type> class DataNode {          //胜者树结点的类定义
public:
    Type data;                          //数据值
```

```
    int index;                          //树中的结点号，即在完全二叉树顺序存储中的下标
    int active;                         //是否参选的标志，  =1，参选；  =0，不再参选
}

template <class Type> void TournamentSort ( Type a[ ], int n ) {
//建立树的顺序存储数组 tree，将数组 a[ ]中的元素复制到胜者树中，对它们进行排序，并把结
//果返送回数组中, n 是待排序元素个数。
    DataNode<Type> *tree;               //胜者树结点数组
    DataNode<Type> item;
    int bottomRowSize = PowerOfTwo ( n );
                        //计算满足>=n 的 2 的最小次幂的数：树的底行大小 n=7 时它为 8
    int TreeSize = 2 * bottomRowSize − 1;           //计算胜者树的大小：内结点+外结点数
    int loadindex = bottomRowSize − 1;              //外结点开始位置
    tree = new DataNode<Type>[TreeSize];            //动态分配胜者树结点数组空间
    int j = 0;                                      //在数组 a 中取数据指针
    for ( int i = loadindex; i < TreeSize; i++ ) {  //复制数组数据到树的外结点中
        tree[i].index = i;                              //下标
        if ( j < n ) { tree[i].active = 1;   tree[i].data = a[j++]; }    //复制数据
        else tree[i].active = 0;                    //后面的结点为空的外结点
    }
    i = loadindex;                                  //进行初始比较寻找最小的项
    while ( i ) {
        j = i;
        while ( j < 2*i ) {                         //处理各对比赛者
            if ( !tree[j+1].active || tree[j].data <= tree[j+1].data )
                tree[(j−1)/2] = tree[j];            //胜者送入双亲
            else tree[(j−1)/2] = tree[j+1];
            j += 2;                                 //下一对参加比较的项
        }
        i = (i−1)/2;                                    //i 退到双亲，直到 i=0 为止
    }
    for ( i=0; i<n−1; i++) {                        //处理其它 n−1 元素
        a[i] = tree[0].data;                        //当前最小元素送数组 a
        tree[tree[0].index].active = 0;             //该元素相应外结点不再比赛
        UpdateTree ( tree, tree[0].index );         //从该处向上修改
    }
    a[n−1] = tree[0].data;
}

template <class Type> void UpdateTree ( DataNode<Type> *tree, int i ) {
//锦标赛排序中的调整算法：i 是表中当前最小元素的下标，即胜者。从它开始向上调整。
    if ( i %2 == 0 ) tree[(i−1)/2] = tree[i−1];     // i 为偶数，对手为左结点
    else tree[(i−1)/2] = tree[i+1];                 // i 为奇数，对手为右结点
    //最小元素输出之后，它的对手上升到父结点位置
    i = (i − 1) / 2;                                // i 上升到双亲结点位置
    while ( i ) {
        if ( i %2 == 0) j = i − 1;                  //确定 i 的对手是左结点还是右结点
        else j = i + 1;
        if ( !tree[i].active || !tree[j].active )   //比赛对手中间有一个为空
```

```
            if ( tree[i].active ) tree[(i−1)/2] = tree[i];
            else tree[(i−1)/2] = tree[j];                    //非空者上升到双亲结点
          else                                               //比赛对手都不为空
            if ( tree[i].data < tree[j].data ) tree[(i−1)/2] = tree[i];
            else tree[(i−1)/2] = tree[j];                    //胜者上升到双亲结点
          i = (i − 1) / 2;                                   // i 上升到双亲结点
      }
  }
```

9−14 手工跟踪对以下各序列进行堆排序的过程。给出形成初始堆及每选出一个排序码后堆的变化。

(1) 按字母顺序排序：Tim, Dot, Eva, Rom, Kim, Guy, Ann, Jim, Kay, Ron, Jan

(2) 按数值递增顺序排序：26, 33, 35, 29, 19, 12, 22

(3) 同样 7 个数字，换一个初始排列，再按数值的递增顺序排序：12, 19, 33, 26, 29, 35, 22

【解答】为节省篇幅，将用数组方式给出形成初始堆和进行堆排序的变化结果。阴影部分表示参与比较的排序码。请读者按照完全二叉树的顺序存储表示画出堆的树形表示。

(1) 按字母顺序排序

形成初始堆（按最大堆）

	0	1	2	3	4	5	6	7	8	9	10
	Tim	Dot	Eva	Rom	Kim	Guy	Ann	Jim	Kay	Ron	Jan
i=4	Tim	Dot	Eva	Rom	[Ron	Guy	Ann	Jim	Kay	Kim	Jan]
i=3	Tim	Dot	Eva	[Rom	Ron	Guy	Ann	Jim	Kay	Kim	Jan]
i=2	Tim	Dot	[Guy	Rom	Ron	Eva	Ann	Jim	Kay	Kim	Jan]
i=1	Tim	[Ron	Guy	Rom	Kim	Eva	Ann	Jim	Kay	Dot	Jan]
i=0	[Tim	Ron	Guy	Rom	Kim	Eva	Ann	Jim	Kay	Dot	Jan]

堆排序

	0	1	2	3	4	5	6	7	8	9	10	
j=10	[Jan	Ron	Guy	Rom	Kim	Eva	Ann	Jim	Kay	Dot	Tim]	交换
	[Ron	Rom	Guy	Kay	Kim	Eva	Ann	Jim	Jan	Dot]	Tim	调整
j=9	[Dot	Rom	Guy	Kay	Kim	Eva	Ann	Jim	Jan	Ron]	Tim	交换
	[Rom	Kim	Guy	Kay	Dot	Eva	Ann	Jim	Jan]	Ron	Tim	调整
j=8	[Jan	Kim	Guy	Kay	Dot	Eva	Ann	Jim	Rom]	Ron	Tim	交换
	[Kim	Kay	Guy	Jim	Dot	Eva	Ann	Jan]	Rom	Ron	Tim	调整
j=7	[Jan	Kay	Guy	Jim	Dot	Eva	Ann	Kim]	Rom	Ron	Tim	交换
	[Kay	Jim	Guy	Jan	Dot	Eva	Ann]	Kim	Rom	Ron	Tim	调整
j=6	[Ann	Jim	Guy	Jan	Dot	Eva	Kay]	Kim	Rom	Ron	Tim	交换
	[Jim	Jan	Guy	Ann	Dot	Eva]	Kay	Kim	Rom	Ron	Tim	调整
j=5	[Eva	Jan	Guy	Ann	Dot	Jim]	Kay	Kim	Rom	Ron	Tim	交换
	[Jan	Eva	Guy	Ann	Dot]	Jim	Kay	Kim	Rom	Ron	Tim	调整
j=4	[Dot	Eva	Guy	Ann	Jan]	Jim	Kay	Kim	Rom	Ron	Tim	交换
	[Guy	Eva	Dot	Ann]	Jan	Jim	Kay	Kim	Rom	Ron	Tim	调整
j=3	[Ann	Eva	Dot	Guy]	Jan	Jim	Kay	Kim	Rom	Ron	Tim	交换
	[Eva	Ann	Dot]	Guy	Jan	Jim	Kay	Kim	Rom	Ron	Tim	调整
j=2	[Dot	Ann	Eva]	Guy	Jan	Jim	Kay	Kim	Rom	Ron	Tim	交换

	[Dot	Ann]	Eva	Guy	Jan	Jim	Kay	Kim	Rom	Ron	Tim	调整
j=1	[Dot	Ann]	Eva	Guy	Jan	Jim	Kay	Kim	Rom	Ron	Tim	交换
	[Ann]	Dot	Eva	Guy	Jan	Jim	Kay	Kim	Rom	Ron	Tim	调整

(2) 按数值递增顺序排序

形成初始堆 (按最大堆)

	0	1	2	3	4	5	6
	26	33	35	29	19	12	22
i=2	26	33	[35	29	19	12	22]
i=0	26	[33	35	29	19	12	22]
i=1	[35	33	26	29	19	12	22]

堆排序

	0	1	2	3	4	5	6	
j=6	[22	33	26	29	19	12	35]	交换
	[33	29	26	22	19	12]	35	调整为堆
j=5	[12	29	26	22	19	33]	35	交换
	[29	22	26	12	19]	33	35	调整为堆
j=4	[19	22	26	12	29]	33	35	交换
	[26	22	19	12]	29	33	35	调整为堆
j=3	[12	22	19	26]	29	33	35	交换
	[22	12	19]	26	29	33	35	调整为堆
j=2	[19	12	22]	26	29	33	35	交换
	[19	12]	22	26	29	33	35	调整为堆
j=1	[12	19]	22	26	29	33	35	交换
	[12]	19	22	26	29	33	35	调整为堆

(3) 同样 7 个数字，换一个初始排列，再按数值的递增顺序排序

形成初始堆 (按最大堆)

	0	1	2	3	4	5	6
	12	19	33	26	29	35	22
i=2	12	19	[35	26	29	33	22]
i=0	12	[29	35	26	19	33	22]
i=1	[35	29	33	26	19	12	22]

堆排序

	0	1	2	3	4	5	6	
j=6	[22	29	33	26	19	12	35]	交换
	[33	29	22	26	19	12]	35	调整为堆
j=5	[12	29	22	26	19	33]	35	交换
	[29	26	22	12	19]	33	35	调整为堆
j=4	[19	26	22	12	29]	33	35	交换
	[26	19	22	12]	29	33	35	调整为堆
j=3	[12	19	22	26]	29	33	35	交换

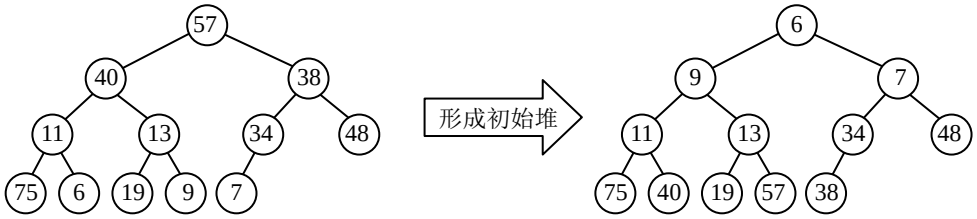
	[22	19	12]	26	29	33	35	调整为堆
j=2	[12	19	22]	26	29	33	35	交换
	[19	12]	22	26	29	33	35	调整为堆
j=1	[12	19]	22	26	29	33	35	交换
	[12]	19	22	26	29	33	35	调整为堆

9-15 如果只想在一个有 n 个元素的任意序列中得到其中最小的第 k (k<<n) 个元素之前的部分排序序列，那么最好采用什么排序方法？为什么？例如有这样一个序列：**{**503, 017, 512, 908, 170, 897, 275, 653, 612, 154, 509, 612*, 677, 765, 094**}**，要得到其第 4 个元素之前的部分有序序列: **{**017, 094, 154, 170**}**，用所选择的算法实现时，要执行多少次比较?

【解答】

一般来讲，当 n 比较大且要选的数据 k << n 时，采用堆排序方法中的调整算法 FilterDown( )最好。但当 n 比较小时，采用锦标赛排序方法更好。

例如，对于序列{ 57, 40, 38, 11, 13, 34, 48, 75, 6, 19, 9, 7 }，选最小的数据 6，需形成初始堆，进行 18 次数据比较；选次小数据 7 时，需进行 4 次数据比较；再选数据 9 时，需进行 6 次数据比较；选数据 11 时，需进行 4 次数据比较。



但如果选用锦标赛排序，对于有 n(n>0)个数据的序列，选最小数据需进行 n−1 次数据比较，以后每选一个数据，进行数据比较的次数，均需 $\lfloor \log_2 n \rfloor - 1$ 次。例如，同样 12 个数据，第一次选最小的数据 6，需进行 11 次数据比较，以后选 7、9、11 时，都是 $\lfloor \log_2 12 \rfloor - 1 = 2$ 次数据比较。

9-16 希尔排序、简单选择排序、快速排序和堆排序是不稳定的排序方法，试举例说明。

【解答】

(1) 希尔排序

{ 512 275 275* 061 **}** 增量为 2
{ 275* 061 512 275 **}** 增量为 1
{ 061 275* 275 512 **}**

(2) 直接选择排序

{ 275 275* 512 061 **}** i = 1
{ 061 275* 512 275 **}** i = 2
{ 061 275* 512 275 **}** i = 3
{ 061 275* 275 512 **}**

(3) 快速排序

{ 512 275 275* **}**
{ 275* 275 **512 }**

(4) 堆排序

{ 275 275* 061 170 **}** 已经是最大堆，交换 275 与 170
{ 170 275* 061 **275 }** 对前 3 个调整
{ 275* 170 061 **275 }** 前 3 个最大堆，交换 275*与 061
{ 061 170 **275*** **275 }** 对前 2 个调整
{ 170 061 **275*** **275 }** 前 2 个最大堆，交换 170 与 061
{ 061 **170** **275*** **275 }**

9-17 设有 n 个待排序元素存放在一个不带表头结点的单链表中，每个链表结点只存放一个元素，头指针为 r。试设计一个算法，对其进行二路归并排序，要求不移动结点中的元素，只改各链结点中的指针，排序后 r 仍指示结果链表的第一个结点。（提示：先对待排序的单链表进行一次扫描，将它划分为若干有序的子链表， 其表头指针存放在一个指针队列中。当队列不空时重复执行，从队列中退出两个有序子链表，对它们进行二路归并， 结果链表的表头指针存放到队列中。如果队列中退出一个有序子链表后变成空队列，则算法结束。这个有序子链表即为所求。)

【解答】

(1) 两路归并算法

```
template<Type> void staticlinkList<Type> :: merge ( int ha;   int hb;   int& hc ) {
//合并两个以 ha 和 hb 为表头指针的有序链表，结果链表的表头由 hc 返回
    int pa, pb, pc;
    if ( Vector[ha].data <= Vector[hb].data )          //确定结果链的表头
       { hc = ha;   pa = Vector[ha].link;   pb = hb; }
    else { hc = hb;   pb = Vector[hb].link;   pa = ha; }
    pc = hc;                                           //结果链的链尾指针
    while ( pa != 0 ) and ( pb != 0 )                  //两两比较，小者进结果链
       if ( Vector[pa].data <= Vector[pb].data )
          { Vector[pc].link = pa;   pc = pa;   pa = Vector[pa].link; }
       else { Vector[pc].link = pb;   pc = pb;   pb = Vector[pb].link; }
    if ( pa != 0 ) Vector[pc].link = pa;               // pb 链处理完, pa 链链入结果链
    else Vector[pc].link = pb;                         // pa 链处理完, pb 链链入结果链
}
```

(2) 归并排序主程序

```
template<class type> void staticlinkList<Type> :: merge_sort ( ) {
    int r, s, t;   Queue <int> Q;
    if ( Vector[0].link == 0 ) return;
    s = Vector[0].link;   Q.Enqueue( s );              //链表第一个结点进队列
    while ( 1 ) {
        t = Vector[s].link;                            //结点 t 是结点 s 的下一个链中结点
        while ( t != 0 && Vector[s].data <= Vector[t].data )
            { s = t;   t = Vector[t].link; }           //在链表中寻找一段有序链表
        Vector[s].link = 0;   s = t;
        if ( t != 0 ) Q.EnQueue( s );                  //存在一段有序链表，截取下来进队列
        else break;                                    //到链尾
    }
    while ( ! Q.IsEmpty( ) ) {
        r = Q.getFront( );   Q.DlQueue( );             //从队列退出一个有序链表的表头 r
        if ( Q.IsEmpty( ) ) break;                     //队列空，表示排序处理完成，退出
        s = Q.getFront( );   Q.DlQueue( );             //从队列再退出一个有序链表的表头 s
        merge( r, s, t );   Q.EnQueue( t );            //归并两个有序链表后结果链表进队列
    }
    Vector[0].link = r;
}
```

9-18 若设待排序排序码序列有 n 个排序码，n 是一个完全平方数。将它们划分为$\sqrt{n}$块，每块有$\sqrt{n}$个排序码。这些块分属于两个有序表。下面给出一种 O(1)空间的非递归归并算法：

step1: 在两个待归并的有序表中从右向左总共选出$\sqrt{n}$个具有最大值的排序码；

step2: 若设在 **step1** 选出的第 2 个有序表中的排序码有 s 个，则从第 1 个有序表选出的排序码有$\sqrt{n}$- s 个。将第 2 个有序表选出的 s 个排序码与第 1 个有序表选出的排序码左边的同样数目的排序码对调；

step3: 交换具有最大$\sqrt{n}$个排序码的块与最左块(除非最左块就是具有最大$\sqrt{n}$个排序码的块)。对最右块进行排序；

step4: 除去具有最大$\sqrt{n}$个排序码的块以外，对其它的块根据其最后的排序码按非递减顺序排序；

step5: 设置 3 个指针，分别位于第 1 块、第 2 块和第 3 块的起始位置，执行多次 **substep**，直到 3 个指针都走到第$\sqrt{n}$块为止。此时前$\sqrt{n}-1$块已经排好序。

☞ **subStep** 所做的工作是比较第 2 个指针与第 3 个指针所指排序码，将值小的与第 1 个指针所指排序码对调，相应指针前进 1 个排序码位置。

step6: 对最后第$\sqrt{n}$块中最大的$\sqrt{n}$个排序码进行排序。

(1) 设有 16 个排序码，分别存放于两个有序表**{**10, 12, 14, 16, 18, 20, 22, 25**}**和**{**11, 13, 15, 17, 19, 21, 23, 24**}**中，试根据上面的描述，写出排序的全过程，并说明它具有时间复杂度 O(n)和空间复杂度 O(1)。

(2) 编写相应的算法。要求两个待排序有序表的长度可以不同，但每一个表的长度都是$\sqrt{n}$的倍数。

(3) 假设两个有序表分别为(x_1, …, x_m)和(x_{m+1}, …, x_n)，编写一个算法归并这两个有序表，得到(x_1, …, x_n)。设 s = $\sqrt{n}$。

9-19 试编写一个算法，将对象序列(x_1, x_2, …, x_n)循环右移 p 个位置，$0 \le p \le n$。要求该算法的时间复杂度为 O(n)而空间复杂度为 O(1)。

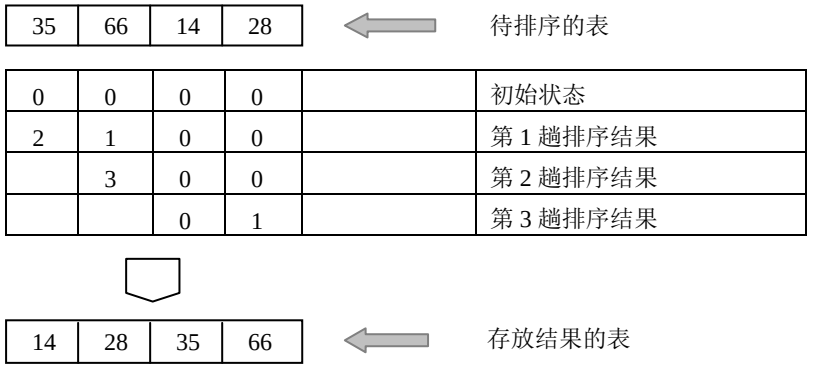
【解答】略

9-20 在什么条件下，MSD 基数排序比 LSD 基数排序效率更高？

【解答】由于高位优先的 MSD 方法是递归的方法，就一般情况来说，不像低位优先的 LSD 方法那样直观自然，而且实现的效率较低。但如果待排序的排序码的大小只取决于高位的少数几位而与大多数低位无关时，采用 MSD 方法比 LSD 方法的效率要高。

9-21 在已排好序的序列中，一个对象所处的位置取决于具有更小排序码的对象的个数。基于这个思想，可得计数排序方法。该方法在声明对象时为每个对象增加一个计数域 count，用于存放在已排好序的序列中该对象前面的对象数目，最后依 count 域的值，将序列重新排列，就可完成排序。试编写一个算法，实现计数排序。并说明对于一个有 n 个对象的序列，为确定所有对象的 count 值，最多需要做 n(n−1)/2 次排序码比较。

【解答】



```
template <class Type> void datalist<Type> :: count_sort ( ) {
//initList是待排序表，resultList是结果表
    int i, j;
    int *c = new datalist <Type>;                          // c是存放计数排序结果的临时表
    for ( i = 0; i < CurrentSize; i++ ) Vector[i].count = 0;        //初始化，计数值都为0
    for ( i = 0; i < CurrentSize−1; i++ )
        for ( j = i+1; j < CurrentSize; j++ )
            if ( Vector[j].key < Vector[i].key ) Vector[i].count++;
            else Vector[j].count++;                                //统计
    for ( i = 0; i < CurrentSize; i++ )                            //在c−>Vector[ ]中各就各位
        c−>Vector[ Vector[i].count ] = Vector[i];
    for ( i = 0; i < CurrentSize; i++ ) Vector[i] = c−>Vector[i];   //结果复制回当前表对象中
    delete c;
}
```

9−22 试证明对一个有 n 个对象的序列进行基于比较的排序，最少需要执行 $n\log_2 n$ 次排序码比较。

【解答】

基于比较的排序方法中，采用分治法进行排序是平均性能最好的方法。方法描述如下：

```
Sort ( List ) {
  if ( List 的长度大于 1) {
     将序列 List 划分为两个子序列 LeftList 和 Right List;
     Sort ( LeftList );     Sort ( RightList );        //分别对两个子序列施行排序
     将两个子序列 LeftList 和 RightList 合并为一个序列 List;
  }
}
```

典型的例子就是快速排序和归并排序。若设 T(n) 是对 n 个对象的序列进行排序所需的时间，而且把序列划分为长度相等的两个子序列后，对每个子序列进行排序所需的时间为 T(n/2)，最后合并两个已排好序的子序列所需时间为 cn（c 是一个常数）。此时，总的计算时间为：

$$
\begin{aligned}
T(n) &\leqslant cn + 2\,T(n/2) && \text{// c 是一个常数}\\
&\leqslant cn + 2\,(\,cn/2 + 2T(n/4)\,) = 2cn + 4T(n/4)\\
&\leqslant 2cn + 4\,(\,cn/4 + 2T(n/8)\,) = 3cn + 8T(n/8)\\
&\cdots\cdots\\
&\leqslant cn\log_2 n + nT(1) = O(n\log_2 n)
\end{aligned}
$$

9−23 如果某个文件经内排序得到 80 个初始归并段，试问

(1) 若使用多路归并执行 3 趟完成排序，那么应取的归并路数至少应为多少？

(2) 如果操作系统要求一个程序同时可用的输入/输出文件的总数不超过 15 个，则按多路归并至少需要几趟可以完成排序？如果限定这个趟数，可取的最低路数是多少？

【解答】

(1) 设归并路数为 k，初始归并段个数 m = 80，根据归并趟数计算公式 $S = \lceil \log_k m \rceil = \lceil \log_k 80 \rceil = 3$ 得：$k^3 \geqslant 80$。由此解得 $k \geqslant 3$，即应取的归并路数至少为 3。

(2) 设多路归并的归并路数为 k，需要 k 个输入缓冲区和 1 个输出缓冲区。1 个缓冲区对应 1 个文件，有 k +1 = 15，因此 k = 14，可做 14 路归并。由 $S = \lceil \log_k m \rceil = \lceil \log_{14} 80 \rceil = 2$。即至少需 2 趟归并可完成排序。

若限定这个趟数，由 $S=\lceil \log_k 80 \rceil = 2$，有 $80 \leqslant k^2$，可取的最低路数为 9。即要在 2 趟内完成排序，进行 9 路排序即可。

9-24 假设文件有 4500 个记录，在磁盘上每个页块可放 75 个记录。计算机中用于排序的内存区可容纳 450 个记录。试问：

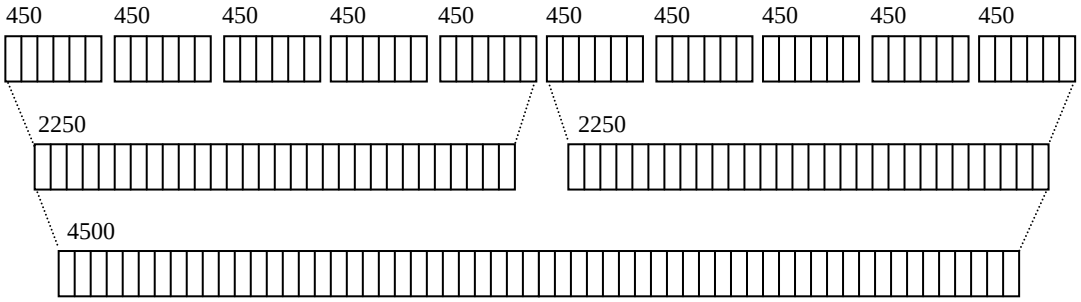
(1) 可建立多少个初始归并段？每个初始归并段有多少记录？存放于多少个页块中？

(2) 应采用几路归并？请写出归并过程及每趟需要读写磁盘的页块数。

【解答】

(1) 文件有 4500 个记录，计算机中用于排序的内存区可容纳 450 个记录，可建立的初始归并段有 4500 / 450 = 10 个。每个初始归并段中有 450 个记录，存于 450 / 75 = 6 个页块中。

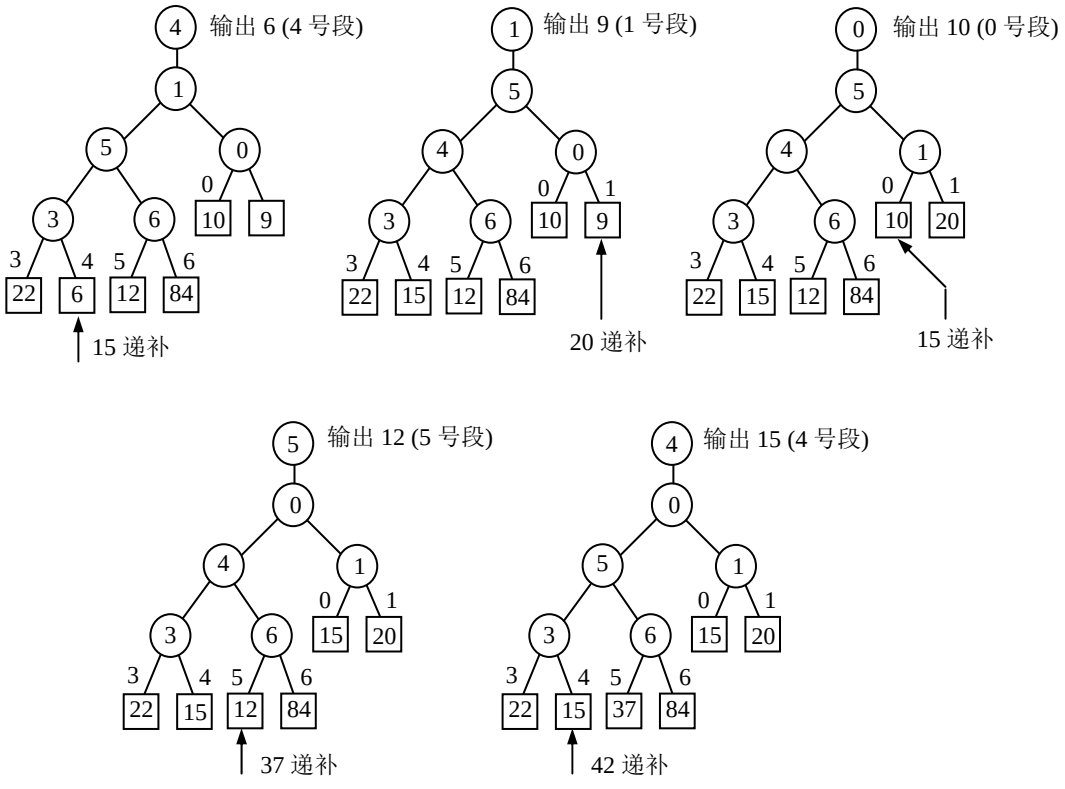
(2) 内存区可容纳 6 个页块，可建立 6 个缓冲区，其中 5 个缓冲区用于输入，1 个缓冲区用于输出，因此，可采用 5 路归并。归并过程如下：



共做了 2 趟归并，每趟需要读 60 个磁盘页块，写出 60 个磁盘页块。

9-25 设初始归并段为(10, 15, 31, ∞), (9, 20, ∞), (22, 34, 37, ∞), (6, 15, 42, ∞), (12, 37, ∞), (84, 95, ∞)，试利用败者树进行 k 路归并，手工执行选择最小的 5 个排序码的过程。

【解答】做 6 路归并排序，选择最小的 5 个排序码的败者树如下图所示。



9-26 设输入文件包含以下记录：14, 22, 7, 24, 15, 16, 11, 100, 10, 9, 20, 12, 90, 17, 13, 19, 26, 38, 30, 25, 50, 28, 110, 21, 40。现采用置换-选择方法生成初始归并段，并假设内存工作区可同时容纳 5 个记录，请画出选择的过程。

【解答】

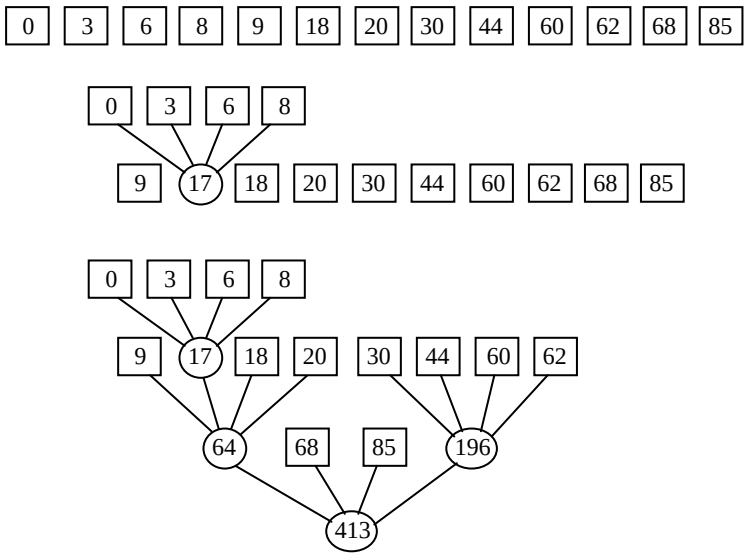
设内存工作区在某一时刻可以处理 6 个记录，利用败者树生成初始归并段的过程如下。

输入文件 InFile	内存工作区	输出文件 OutFile	动作
14, 22, 07, 24, 15, 16, 11, 100, 10, 09, 20, 12, 90, 17, 13, 19, 26, 38, 30, 25, 50, 28, 110, 21, 40			输入 6 个记录
11, 100, 10, 09, 20, 12, 90, 17, 13, 19, 26, 38, 30, 25, 50, 28, 110, 21, 40	14, 22, **07**, 24, 15, 16		选择 07, 输出 07, 门槛 07, 置换 11
100, 10, 09, 20, 12, 90, 17, 13, 19, 26, 38, 30, 25, 50, 28, 110, 21, 40	14, 22, **11**, 24, 15, 16	07	选择 11, 输出 11, 门槛 11, 置换 100
10, 09, 20, 12, 90, 17, 13, 19, 26, 38, 30, 25, 50, 28, 110, 21, 40	**14**, 22, 100, 24, 15, 16	07, 100	选择 14, 输出 14, 门槛 14, 置换 10
09, 20, 12, 90, 17, 13, 19, 26, 38, 30, 25, 50, 28, 110, 21, 40	10, 22, 100, 24, **15**, 16	07, 100, 14	选择 15, 输出 15, 门槛 15, 置换 09
20, 12, 90, 17, 13, 19, 26, 38, 30, 25, 50, 28, 110, 21, 40	10, 22, 100, 24, 09, **16**	07, 100, 14, 15	选择 16, 输出 16, 门槛 16, 置换 20
12, 90, 17, 13, 19, 26, 38, 30, 25, 50, 28, 110, 21, 40	10, 22, 100, 24, 09, **20**	07, 100, 14, 15, 16	选择 20, 输出 20, 门槛 20, 置换 12
90, 17, 13, 19, 26, 38, 30, 25, 50, 28, 110, 21, 40	10, **22**, 100, 24, 09, 12	07, 100, 14, 15, 16, 20	选择 22, 输出 22, 门槛 22, 置换 90
17, 13, 19, 26, 38, 30, 25, 50, 28, 110, 21, 40	10, 90, 100, **24**, 09, 12	07, 100, 14, 15, 16, 20, 22	选择 24, 输出 24, 门槛 24, 置换 17
13, 19, 26, 38, 30, 25, 50, 28, 110, 21, 40	10, **90**, 100, 17, 09, 12	07, 100, 14, 15, 16, 20, 22, 24	选择 90, 输出 90, 门槛 90, 置换 13
19, 26, 38, 30, 25, 50, 28, 110, 21, 40	10, 13, **100**, 17, 09, 12	07, 100, 14, 15, 16, 20, 22, 24, 90	选择 100, 输出 100, 门槛 100, 置换 19
26, 38, 30, 25, 50, 28, 110, 21, 40	10, 13, 19, 17, 09, 12	07, 100, 14, 15, 16, 20, 22, 24, 90, 100, ∞	无大于门槛的的记录, 输出段结束符
26, 38, 30, 25, 50, 28, 110, 21, 40	10, 13, 19, 17, **09**, 12		选择 09, 输出 09, 门槛 09, 置换 26
38, 30, 25, 50, 28, 110, 21, 40	**10**, 13, 19, 17, 26, 12	09	选择 10, 输出 10, 门槛 10, 置换 38
30, 25, 50, 28, 110, 21, 40	38, 13, 19, 17, 26, **12**	09, 10	选择 12, 输出 12, 门槛 12, 置换 30
25, 50, 28, 110, 21, 40	38, **13**, 19, 17, 26, 30	09, 10, 12	选择 13, 输出 13, 门槛 13, 置换 25
50, 28, 110, 21, 40	38, 25, 19, **17**, 26, 30	09, 10, 12, 13	选择 17, 输出 17, 门槛 17, 置换 50
28, 110, 21, 40	38, 25, **19**, 50, 26, 30	09, 10, 12, 13, 17	选择 19, 输出 19, 门槛 19, 置换 28

110, 21, 40	38, **25**, 28, 50, 26, 30	09, 10, 12, 13, 17, 19	选择 25, 输出 25, 门槛 25, 置换 110
21, 40	38, 110, 28, 50, **26**, 30	09, 10, 12, 13, 17, 19, 25	选择 26, 输出 26, 门槛 26, 置换 21
40	38, 110, **28**, 50, 21, 30	09, 10, 12, 13, 17, 19, 25, 26	选择 28, 输出 28, 门槛 28, 置换 40
	38, 110, 40, 50, 21, **30**	09, 10, 12, 13, 17, 19, 25, 26, 28	选择 30, 输出 30, 门槛 30, 无输入
	38, 110, 40, 50, 21, ∞	09, 10, 12, 13, 17, 19, 25, 26, 28, 30	选择 38, 输出 38, 门槛 38, 无输入
	−, 110, **40**, 50, 21, −	09, 10, 12, 13, 17, 19, 25, 26, 28, 30, 38	选择 40, 输出 40, 门槛 40, 无输入
	−, 110, −, **50**, 21, −	09, 10, 12, 13, 17, 19, 25, 26, 28, 30, 38, 40	选择 50, 输出 50, 门槛 50, 无输入
	−, **110**, −, −, 21, −	09, 10, 12, 13, 17, 19, 25, 26, 28, 30, 38, 40, 50	选择 110, 输出 110, 门槛 110, 无输入
	−, −, −, −, 21, −	09, 10, 12, 13, 17, 19, 25, 26, 28, 30, 38, 40, 50, 110, ∞	无大于门槛的的记录, 输出段结束符
	−, −, −, −, **21**, −		选择 21, 输出 21, 门槛 21, 无输入
	−, −, −, −, −, −	21, ∞	无大于门槛的的记录, 输出段结束符

9−27 给出 12 个初始归并段，其长度分别为 30, 44, 8, 6, 3, 20, 60, 18, 9, 62, 68, 85。现要做 4 路外归并排序，试画出表示归并过程的最佳归并树，并计算该归并树的带权路径长度 WPL。

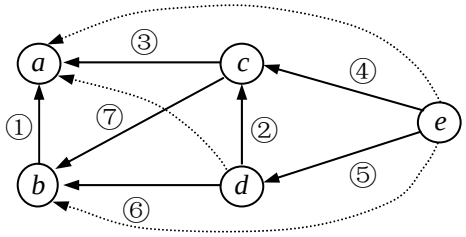
【解答】设初始归并段个数 n = 12，外归并路数 k = 4，计算(n−1) % (k−1) = 11 % 3 = 2 ≠ 0，必须补 k−2−1 = 1 个长度为 0 的空归并段，才能构造 k 路归并树。此时，归并树的内结点应有(n−1+1)/(k−1) = 12/3 = 4 个。



WPL = (3+6+8)*3 + (9+18+20+30+44+60+62)*2 + (68+85)*1 = 51 + 486 + 153 = 690

9-28 试构造排序 5 个整数最多用 7 次比较的算法。

【解答】算法的思想可以用如下的有向图来描述：



顶点	*a*	*b*	*c*	*d*	*e*
入度	4	3	2	1	0

在图中有 5 个顶点，代表 5 个可比较的整数 *a, b, c, d, e*。有向边的箭头从较大的整数指向较小的整数，虚线表示的有向边表示不用比较，而是通过传递性得到的。图中各有向边的编号给出 7 次比较的先后次序。

首先比较 *a* 与 *b* 和 *c* 与 *d*，得 $a < b, c < d$，这需要 2 次比较。然后比较 *a* 与 *c*，得 $a < c$，从而可得 $a < c < d$，这需要 3 次比较。

再比较 *c* 与 *e* 和 *d* 与 *e*，得 $c < e, d < e$，从而可得 $a < c < d < e$。最后 2 次比较，将 *b* 插入到 a 与 c 之间，得 $a < b < c < d < e$。

9-29 下面的程序是一个的两路归并算法 merge，只需要一个附加存储。设算法中参加归并的两个归并段是 A[left]~A[mid] 和 A[mid]~A[right]，归并后结果归并段放在原地。

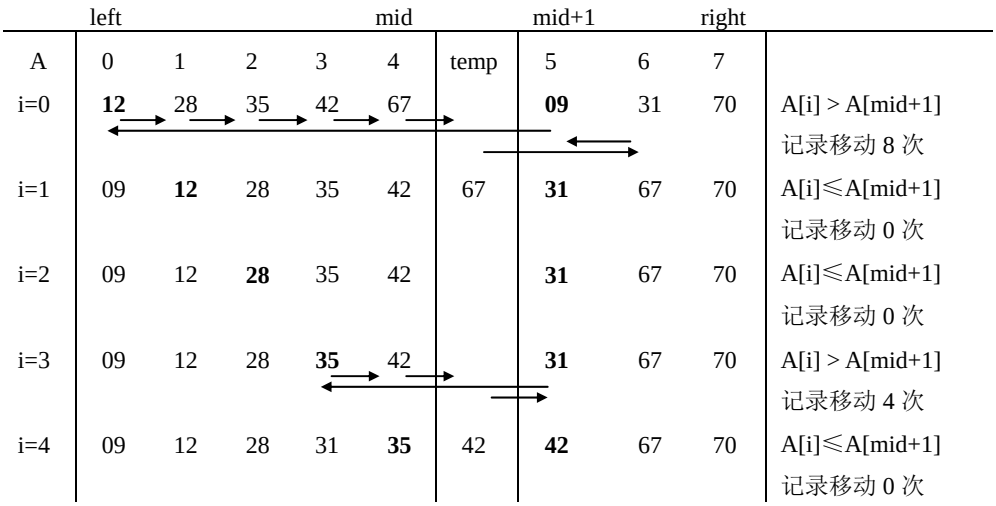
```
template<Type> void dataList<Type> :: merge( const int left, const int mid, const int right ) {
  int i, j;   Type temp;
  for ( i = left; i <= mid; i++ ) {
    if ( A[i] > A[mid+1] ) {
      temp = A[mid];
      for ( j = mod-1; j >= i; j--) A[j+1] = A[j];
      A[i] = A[mid+1];
      if ( temp <= A[mid+2] ) A[mid+1] = temp;
      else {
        for ( j = mid+2; j <= right; j++ )
          if ( temp > A[j] ) A[j-1] = A[j];
          else { A[j-1] = temp;   break; }
      }
    }
  }
}
```

(1) 若 A = { 12, 28, 35, 42, 67, 9, 31, 70 }, left = 0, mid = 4, right = 7。写出每次执行算法最外层循环后数组的变化。

(2) 试就一般情况 A[n]、left、mid 和 right，分析此算法的性能。

【解答】

(1) 数组 A 每次执行最外层循环后数组的变化如下：



(2) 本算法的记录比较次数和移动次数与待排序记录序列的初始排列有关。因此，性能分析需要讨论最好情况、最坏情况。

最好情况，例如参加排序的后一个有序表中所有记录(从 mid+1 到 right)的排序码均大于前一个有序表(从 left 到 mid)的排序码。此时，记录排序码的比较次数为 mid−left+1，与前一个有序表的长度相同，记录的移动次数为 0。

最坏情况，例如参加排序的后一个有序表中所有记录(从 mid+1 到 right)的排序码均小于前一个有序表(从 left 到 mid)的排序码，并且前一个表中有 n = mid−left+1 个元素，后一个表中有 m = right−mid 个元素，那么，估计记录排序码比较次数约为 n+m+m(m+4)/8，记录移动次数约为 (n(n+1)+m(m+1))/2。

9−30 当记录对象存放在数据表中时，进行排序时会移动许多记录对象，降低了排序的效率。为避免记录的移动, 使用静态链表进行排序。在排序过程中, 不移动记录对象，只修改链接指针。例如对如下的静态链表(图中只显示了排序码)进行排序：（V[0].link 是表头指针）

初始配置

	V[0]	V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]
data		49	65	38	27	97	13	76	49*
link	1	1	2	3	4	5	6	7	8

在排序结束后，各记录对象的排序顺序由各记录对象的 link 指针指示。V[0].link 指示排序码最小的记录对象，而排序码最大的记录对象的 link 指针为 0。

排序结果

	V[0]	V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]
data		49	65	38	27	97	13	76	49*
link	6	8	7	1	3	0	4	5	2

最后可以根据需要按排序码大小从小到大重排记录的物理位置。试设计一个算法，实现这种记录的重排。

【解答】

重排记录的基本思想是：从 i＝1 起, 检查在排序之后应该是第 i 个记录的记录是否正好在第 i 个记录位置。

i=1	V[0]	V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]	head	temp
data		**49**	65	38	27	97	**13**	76	49*		13
link	6	**8**	7	1	3	0	**4**	5	2	**6**	4

当 i = 1 时，第 1 个记录不是具有最小排序码的记录，具有最小排序码的记录地址在 head = Vector[0].link = 6。交换 Vector[head]与 Vector[i]并将原位置 head 记入 Vector[i].link。此时，在 temp.link 中存有下一个具次小排序码记录的地址 4，记入 head。再处理 i = 2 的情形。

i=2	V[0]	V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]	head	Temp
data		13	**65**	38	**27**	97	49	76	49*		13
link	6	6	**7**	1	**3**	0	8	5	2	4	4

当 i = 2 时，第 2 个记录不是具有次小排序码的记录，具有次最小排序码的记录地址在 head = 4。交换 Vector[head]与 Vector[i] 并将原位置 head = 4 记入 Vector[i].link。此时，在 temp.link 中存有下一个具次小排序码记录的地址 3，记入 head。再处理 i = 3 的情形。

i=3	V[0]	V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]	head	temp
data		13	27	**38**	65	97	49	76	49*		27
link	6	6	4	**1**	7	0	8	5	2	3	3

当 i = 3 时，第 3 个记录正应排在此位置(head == i)，原地交换并将位置 head = 3 记入 Vector[i]. link。此时，在 temp.link 中存有下一个具次小排序码记录的地址 head = 1，当下一次处理 i = 4 的情形时，head < i，表明它已处理过，但在 Vector[head].link 中记有原来此位置的记录交换到的新位置 6，令 head = Vector[head].link = 6。再处理 i = 4 的情形。

i=4	V[0]	V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]	head	Temp
data		13	27	38	**65**	97	**49**	76	49*		38
link	6	6	4	3	**7**	0	**8**	5	2	1,6	1

当 i = 4 时，交换 Vector[head]与 Vector[i]，并将位置 head = 6 记入 Vector[i].link。此时，在 temp.link 中存有下一个具次小排序码记录的地址 head = 8，再处理 i = 5 的情形

i=4	V[0]	V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]	head	temp
data		13	27	38	49	**97**	65	76	**49***		49
link	6	6	4	3	6	**0**	7	5	**2**	8	8

当 i = 5 时，交换 Vector[head]与 Vector[i]，并将位置 head = 8 记入 Vector[i].link。此时，在 temp.link 中存有下一个具次小排序码记录的地址 head = 8，再处理 i = 6 的情形

i=4	V[0]	V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]	head	temp
data		13	27	38	49	49*	**65**	76	97		49*
link	6	6	4	3	6	8	**7**	5	0	2	2

当 i = 6 时，应存放于此的记录不是 Vector[head]，因为 head = 2 时的 Vector[2]已处理过，通过 head = Vector[head].link = 4，此位置记录也已处理过，再求 head = Vector[head].link = 6 ≮ i, 原在此位置的记录还应在此位置(head == i)，原地交换并将位置 head = 6 记入 Vector[i]. link。此时，在 temp.link 中存有下一个具次小排序码记录的地址 head = 7。再处理 i = 7 的情形。

i=4	V[0]	V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]	head	temp
data		13	27	38	49	49*	65	**76**	97		65
link	6	6	4	3	6	8	6	**5**	0	7	7

当 i = 7 时，第 7 个记录正应排在此位置(head == i)，原地交换并将位置 head = 7 记入 Vector[i]. link。此时，在 temp.link 中存有下一个具次小排序码记录的地址 head = 5，当下一次处理 i = 8 的情形时，head < i，表明它已处理过，但在 Vector[head].link 中记有原来此位置的记录交换到的新位置 8，令 head = Vector[head].link = 8。再处理 i = 8 的情形。

i=4	V[0]	V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]	Head	temp
data		13	27	38	49	49*	65	76	**97**		76
link	6	6	4	3	6	8	6	7	**0**	5	5

当 i = 8 时，第 8 个记录正应排在此位置(head == i)，原地交换并将位置 head = 8 记入 Vector[i]. link。此时，在 temp.link 中存有下一个具次小排序码记录的地址 head = 0，它符合退出循环的条件，因此退出循环，算法结束。

i=4	V[0]	V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]	Head	temp
data		13	27	38	49	49*	65	76	97		97
link	6	6	4	3	6	8	6	7	8	0	0

下面给出重新安排物理位置的算法：

```
template <class Type> void StaticdataList<Type> :: ReArrange ( ) {
//按照已排好序的静态链表中的链接顺序，重新排列所有记录对象，使得所有对象按链接顺序物理
//地重新排列。
    int i = 1, head = Vector[0];   Element<Type> temp;
    while ( head != 0 ) {
        temp = Vector[head];   Vector[head] = Vector[i];   Vector[i] = temp;
        Vector[i].link = head;
        head = temp.link;
        i++;
        while ( head < i && head > 0 ) head = Vector[head].link;
    }
}
```