第9章 排序

**一、复习要点**

排序是使用最频繁的一类算法。排序分内排序与外排序。内排序算法主要分5大类，有12个算法。在插入排序类中讨论了直接插入排序、二分法插入排序、表插入排序和shell排序算法；在交换排序类中讨论了起泡排序和快速排序算法；在选择排序类中讨论了简单选择排序、锦标赛排序和堆排序算法；在归并排序类中讨论了迭代的两路归并排序和递归的表归并排序算法；在多排序码排序类中讨论了最低位优先的链表基数排序算法。其中，不稳定的排序方法有shell排序、简单选择排序、快速排序和堆排序；适合于待排序对象数目n比较大的排序方法有快速排序、堆排序、归并排序和基数排序；排序码比较次数不受对象排序码初始排列影响的排序方法有折半插入排序、简单选择排序、锦标赛排序、两路归并排序和基数排序，其中，当排序码的初始排列接近有序时，直接插入排序和起泡排序等增长很快，而快速排序则变成慢速排序。

外排序是基于外存的排序方法。由于外存以顺序存取的效率最高，以归并排序最为适合。因此，外排序以k路平衡归并为主。在k个对象排序码中选取最小排序码，采用了败者树。这是一种高效的选择算法。此外，还讨论了初始归并段生成的方法，最佳归并树等问题。

本章复习的要点是：

**1、**基本知识点

要求理解排序的基本概念，包括什么是排序，排序的稳定性，排序的性能分析，如时间代价（对象排序码的比较次数和对象的移动次数）和空间代价（附加对象个数）。掌握插入排序（直接插入排序；折半插入排序；链表插入排序）、交换排序（起泡排序；快速排序）、选择排序（直接选择排序；链表选择排序；锦标赛排序；堆排序）、迭代的归并排序等内排序的方法及其性能分析方法。理解基数排序方法及其性能分析方法。理解多路平衡归并等外排序方法及败者树构造方法。理解生成初始归并段及败者树构造方法。理解最佳归并树的建立方法。

**2、**算法设计

⮚ 插入排序：直接插入排序算法、折半插入排序算法、链表插入排序算法

⮚ 交换排序：起泡排序算法，快速排序的递归算法和用栈实现的非递归算法

⮚ 选择排序：直接选择排序算法，链表选择排序算法，堆排序算法

⮚ 归并排序：两路归并算法，迭代的归并排序算法；递归的链表归并排序算法和用队列实现的非递归链表归并排序算法

⮚ 其它排序算法：计数排序算法，奇偶排序算法

**二、难点和重点**

1、基本概念：排序码、初始排序码排列、排序码比较次数、数据移动次数、稳定性、附加存储、内部排序、外部排序

2、插入排序

⮚ 当待排序的排序码序列已经基本有序时，用直接插入排序最快

3、选择排序

⮚ 用直接选择排序在一个待排序区间中选出最小的数据时，与区间第一个数据对调，而不是顺次后移。这导致方法不稳定。

* 当在n个数据(n很大)中选出最小的5 ~ 8个数据时，锦标赛排序最快

⮚ 锦标赛排序的算法中将待排序的数据个数n补足到2的k次幂2k-1 < n ≤ 2k

⮚ 在堆排序中将待排序的数据组织成完全二叉树的顺序存储。

4、交换排序

⮚ 快速排序是一个递归的排序方法

⮚ 当待排序排序码序列已经基本有序时，快速排序显著变慢。

5、二路归并排序

⮚ 归并排序可以递归执行

⮚ 归并排序需要较多的附加存储。可以采用一种“推拉法”实现归并排序，算法的时间复杂度为O (n)、空间复杂度为O(1)

⮚ 归并排序对待排序排序码的初始排列不敏感，排序速度较稳定

6、外排序

⮚ 多路平衡归并排序的过程、I/O缓冲区个数的配置

⮚ 外排序的时间分析、利用败者树进行多路平衡归并

⮚ 利用置换选择方法生成不等长的初始归并段

* 最佳归并树的构造及WPL的计算

**三、教材中习题的解析**

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 ⎣n/2⎦, ⎣⎣n/2⎦/2⎦, ⎣⎣n/2⎦/2⎦/2⎦, …,1。一般地，增量序列可采用⎣nα⎦, ⎣⎣nα⎦α⎦, ⎣⎣nα⎦α⎦α⎦, …, 1。大量实验表明，取α=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 ⎣0.45454n⎦ 的一个简单方法是用整数算术计算(5\*n-1)/11。需要注意，当α< 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  pos  pos  pos  pos | 0,1,2,3 | **[ 12** | **2** | **16** | **30** | **28** | | **10** | **16\*** | | | **20** | **6** | | **18 ]** | 9 |
|  |  |  |  |  |  |  | |  |  | | |  |  | |  |  |
| 6  pos  pos | 0,1 | **[ 6** | **2** | **10 ]** | 12 | [ 28 | | 16 | 16\* | | | 20 | 30 | | 18 ] | 2 |
|  |  |  |  |  |  |  | |  |  | | |  |  | |  |  |
| 28  pos  pos  pos  pos  pos | 4,5,6,7,8 | [ 2 ] | 6 | [ 10 ] | 12 | **[ 28** | | **16** | **16\*** | | | **20** | **30** | | **18 ]** | 5 |
|  |  |  |  |  |  |  | |  |  | | |  |  | |  |  |
| 18  pos  pos  pos | 4,5,6 | 2 | 6 | 10 | 12 | **[ 18** | | **16** | **16\*** | | | **20 ]** | 28 | | [ 30 ] | 3 |
|  |  |  |  |  |  |  | |  |  | | |  |  | |  |  |
| 16\*  pos | 4 | 2 | 6 | 10 | 12 | **[ 16\*** | | **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

2

6

2

10

6

2

16\*

10

16

6

2

18

6

20

16\*

10

28

16

30

2

12

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24 = 16个。排序码比较次数 = 9。

输出6，调整胜者树

6

6

10

10

6

12

6

16\*

10

16

12

18

6

20

16\*

10

28

30

16

2

12

10

输出10，调整胜者树

排序码比较

次数 = 1。

10

10

18

18

12

10

18

16\*

16

12

12

18

6

20

16\*

10

28

30

16

2

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。

12

输出12，调整胜者树

排序码比较

次数 = 3。

18

12

16\*

18

12

16\*

28

16

18

12

18

6

20

16\*

10

28

30

16

2

12

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

16

排序码比较

次数 = 2。

16

18

16

18

16\*

28

16

18

12

16\*

6

20

16\*

10

28

30

16

2

12

18

16\*

输出16\*，调整胜者树

16\*

18

排序码比较

次数 = 2。

16\*

30

18

18

16\*

12

30

28

6

20

16\*

10

28

30

16

2

12

18

18

输出18，调整胜者树

排序码比较

次数 = 3。

20

18

20

30

18

20

30

18

12

28

18

20

16\*

10

28

30

16

2

12

6

20

20

排序码比较

次数 = 0。

20

18

30

20

18

12

30

18

输出20，调整胜者树

28

6

20

16\*

10

28

30

16

2

12

18

28

输出28，调整胜者树

28

排序码比较

次数 = 2。

18

28

18

30

20

28

30

18

12

20

6

16\*

10

28

30

16

2

12

18

30

输出30，排序完成

30

排序码比较

次数 = 0。

18

28

18

30

20

18

28

30

12

18

20

16\*

10

28

30

16

2

12

6

(7) 堆排序

第一步，形成初始的最大堆 (略)，第二步，做堆排序。

6

30

12

16

28

16

28

16

2

16\*

18

20

10

18

16\*

10

20

16\*

10

28

30

30

12

2

2

12

6

18

6

20

初始排列，不是最大堆 形成初始最大堆 交换0# 与9# 对象

20

28

28

18

16

20

16

16

20

6

10

12

16\*

18

18

16\*

12

10

10

12

16\*

30

28

2

6

30

2

30

6

2

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

16\*

28

28

2

12

16

16

12

16

18

18

2

2

16\*

10

10

6

6

12

10

16\*

6

18

30

20

30

20

28

20

30

交换0# 与7# 对象 从0# 到6# 重新形成堆 交换0# 与6# 对象

16

16\*

10

10

12

16

12

12

16

6

18

16\*

2

16\*

18

18

6

2

6

2

10

30

28

20

28

20

30

28

20

30

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

2

12

6

6

10

6

10

10

12

12

18

16

16\*

16

16\*

18

2

16

18

2

16\*

28

20

30

28

20

30

28

30

20

交换0# 与4# 对象 从0# 到3# 重新形成堆 交换0# 与3# 对象

6

2

10

10

2

6

10

2

6

18

16

12

16\*

16\*

12

16

18

12

18

16\*

16

28

20

30

28

20

30

30

28

20

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

2

2

6

10

10

6

16\*

16

12

18

18

16

12

16\*

28

20

30

30

20

28

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。

排序码比较5次

18

20

16\*

2

16

30

28

10

6

12

6 18

16\* 20 12

10 28

16 30

2 12

排序码比较6次

6 18

10 16\* 20 28

2 12 16 30

排序码比较7次

排序码比较9次

2 10 12 16 16\* 20 28 30

6 18

2 6 10 12 16 16\* 18 20 28 30

(9) 基数排序

6

2

16

30

10

28

16\*

20

18

12

按最低位分配

r[0] r[1] r[2] r[3] r[4] r[5] r[6] r[7] r[8] r[9]

6

20

2

18

16\*

10

28

16

30

12

f[0] f[1] f[2] f[3] f[4] f[5] f[6] f[7] f[8] f[9]

收集

28

6

2

10

18

16\*

16

12

20

30

按最高位分配

18

r[0] r[1] r[2] r[3] r[4] r[5] r[6] r[7] r[8] r[9]

16\*

16

28

6

f[0] f[1] f[2] f[3] f[4] f[5] f[6] f[7] f[8] f[9]

12

30

20

2

10

30

28

20

18

16

10

12

6

2

收集

16\*

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，

57 40 38 11 13 34 48 75 6 19 9 7 如9向相反方向移动

6 57 40 38 11 13 34 48 75 7 19 9 如19向相反方向移动

6 7 57 40 38 11 13 34 48 75 9 19 如9向最终方向移动

6 7 9 57 40 38 11 13 34 48 75 19 如13向相反方向移动

6 7 9 11 57 40 38 13 19 34 48 75 如13向最终方向移动

6 7 9 11 13 57 40 38 19 34 48 75 如34向相反方向移动

6 7 9 11 13 19 57 40 38 34 48 75

6 7 9 11 13 19 34 57 40 38 48 75

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】

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 ==** ( Element<**Type>** x ) **{ return** key ==x.key**;** **}** //判**this**与x相等

**int operator <=** ( Element<**Type>** x ) **{ return** key <=x.key**; }** //判**this**小于或等于x

**int operator >** ( Element<**Type>** x ) **{ return** key >x.key**; }** //判**this**大于x

**int operator <** ( Element<**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

**}**

**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(n2)。

【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) }

= ……

= (n-1) + (n-2) + (n-3) + … + { 2 + T(1) }

= (n-1) + (n-2) + (n-3) + … + 2 + 1

= n(n-1)/2 = O(n2)

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(log2n)。

【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)

= ……

= log2n + S(1) = log2n (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log2n，其最坏情况为O(log2n)。

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(nlog2n)。

【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。

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 ( **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 ( **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]**; }**

**void** Sort ( )**;**

**}**

**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 ⎣log2n⎦ -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 ⎣log211⎦ -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。

6

57

7

9

38

40

形成初始堆

48

48

34

34

13

13

11

11

19

7

9

19

6

75

75

38

40

57

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 ⎣log2n⎦ -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 ⎣log212⎦ -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

t**emplate<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。将它们划分为块，每块有个排序码。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- 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:** 交换具有最大个排序码的块与最左块(除非最左块就是具有最大个排序码的块)。对最右块进行排序；

**step4:** 除去具有最大个排序码的块以外，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个指针都走到第块为止。此时前块已经排好序。

☞ **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:** 对最后第块中最大的个排序码进行排序。

(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的倍数。

(3) 假设两个有序表分别为(x1, …, xm)和(xm+1, …, xn)，编写一个算法归并这两个有序表，得到(x1, …, xn)。设s = 。

【解答】略

9-19 试编写一个算法，将对象序列 (x1, x2, …, xn) 循环右移p个位置，0 ≤ p ≤ 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】

**void** siftk ( **int** X[ ], **int** n, **int** p ) **{**

**int** i, j, m = GCD ( n, p )**;**  //寻找n 与p的最大公约数

**for** ( **int** k = 0**;** k < m**;** k++ ) **{**

**int** temp = X[k]**;** i = k**;** j = (n + i - p) % n**;**

**while** ( j != k ) **{**

X[i] = X[j]**;** i = j**;** j = (n + j - p) % n**;**

**}**

X[i] = temp**;**

**}**

**}**

**int** GCD( **int** dividend, **int** divisor ) **{**

//函数返回dividend与divisor的最大公约数, 若dividend与divisor互质, 则函数返回1.

**if** ( dividend < divisor ) **{ int** temp = dividend**;** dividend = divisor**;** divisor = temp**; }**

**int** remainder = dividend % divisor**;**

**while** ( remainder != 0 ) **{**

dividend = divisor**;** divisor = remainder**;**

remainder = dividend % divisor**;**

**}**

**return** divisor**;**

**}**

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】

待排序的表

35 66 14 28

14 28 35 66

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 |  | 初始状态 |
| 2 | 1 | 0 | 0 |  | 第1趟排序结果 |
|  | 3 | 0 | 0 |  | 第2趟排序结果 |
|  |  | 0 | 1 |  | 第3趟排序结果 |

存放结果的表

**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nlog2n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：

T(n)≤cn + 2 T(n/2 ) // c是一个常数

≤ cn + 2 ( cn/2 + 2T(n/4) ) = 2cn + 4T(n/4)

≤ 2cn + 4 ( cn/4 +2T(n/8) ) = 3cn + 8T(n/8)

………

≤ cn log2n + nT(1) = O(n log2n )

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 = ⎡logkm⎤ = ⎡logk80⎤ = 3得：k3≥80。由此解得k≥3，即应取的归并路数至少为5。

(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 = ⎡logkm⎤ = ⎡log1480⎤ = 2。即至少需2趟归并可完成排序。

若限定这个趟数，由S = ⎡logk80⎤ = 2，有80≤k2，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：

450 450 450 450 450 450 450 450 450 450

2250 2250

4500

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。

输出10 (0号段)

0

输出9 (1号段)

1

4

输出6 (4号段)

5

5

1

1

1

4

4

0

0

5

1

0

0

84

0

10

20

10

3

6

6

9

3

9

10

3

6

4

6

4

3

6

3

5

5

6

5

4

3

22

15递补

15

20递补

15

84

1

22

12

12

22

6

84

12

15递补

输出15 (4号段)

输出12 (5号段)

4

5

0

0

5

12

6

22

5

1

1

4

1

0

1

0

15

20

15

20

6

3

3

6

6

4

3

6

4

3

5

5

15

15

84

84

22

37

12

22

42递补

37递补

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  4 | 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。

85

68

62

60

44

30

20

18

9

6

8

3

0

6

3

0

8

85

68

62

60

44

30

17

9

20

18

8

3

0

6

60

44

30

62

17

20

18

9

196

68

85

64

413

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

**三、其他练习题**

9-28 填空题

(1) 每次从无序表中取出一个元素，把它插入到有序表中的适当位置，此种排序方法叫做\_\_\_\_\_\_\_排序；每次从无序表中挑选出一个最小或最大元素，把它交换到有序表的一端，此种排序方法叫做\_\_\_\_\_\_\_\_排序。

(2) 每次直接或通过基准元素间接比较两个元素，若出现逆序排列时就交换它们的位置，此种排序方法叫做\_\_\_\_\_\_\_\_排序；每次使两个相邻的有序表合并成一个有序表的排序方法叫做\_\_\_\_\_\_\_\_排序。

(3) 在直接选择排序中，记录比较次数的时间复杂度为\_\_\_\_\_\_\_\_，记录移动次数的时间复杂度为\_\_\_\_\_\_\_\_。

(4) 在堆排序的过程中，对n个记录建立初始堆需要进行\_\_\_\_\_\_\_\_次调整运算，由初始堆到堆排序结束，需要对树根结点进行\_\_\_\_\_\_\_\_次调整运算。

(5) 在堆排序的过程中，对任一分支结点进行调整运算的时间复杂度为\_\_\_\_\_\_\_\_，整个堆排序过程的时间复杂度为\_\_\_\_\_\_\_\_。

(6) 假定一组记录的排序码为(46,79,56,38,40,84)，则利用堆排序方法建立的初始堆为\_\_\_\_\_\_\_\_。

(7) 快速排序在平均情况下的时间复杂度为\_\_\_\_\_\_\_\_，在最坏情况下的时间复杂度为\_\_\_\_\_\_\_\_。

(8) 快速排序在平均情况下的空间复杂度为\_\_\_\_\_\_\_\_，在最坏情况下的空间复杂度为\_\_\_\_\_\_\_\_。

(9) 在快速排序方法中，进行每次划分时，是从当前待排序区间的\_\_\_\_\_\_\_\_向\_\_\_\_\_\_\_\_依次查找出处于逆序的元素并交换之，最后将基准元素交换到一个确定位置，从而以该位置把当前区间划分为前后两个子区间。

(10) 假定一组记录的排序码为 (46,79,56,38,40,80)，对其进行快速排序的一次划分的结果为\_\_\_\_\_\_\_\_。

(11) 在二路归并排序中，对n个记录进行归并的趟数为\_\_\_\_\_\_\_\_。

(12) 在归并排序中，进行每趟归并的时间复杂度为\_\_\_\_\_\_\_\_，整个排序过程的时间复杂度为\_\_\_\_\_\_\_\_，空间复杂度为\_\_\_\_\_\_\_\_。

(13) 对20个记录进行归并排序时，共需要进行\_\_\_\_\_\_\_\_趟归并，在第三趟归并时是把长度为\_\_\_\_\_\_\_\_的有序表两两归并为长度为\_\_\_\_\_\_\_\_的有序表。

(14) 假定一组记录的排序码为 (46,79,56,38,40,80)，对其进行归并排序的过程中，第二趟归并后的结果为\_\_\_\_\_\_\_\_。

【解答】

(1) 插入, 选择 (2) 交换, 二路归并

(3) O(n2), O(n) (4) ⎣n/2⎦ , n-1

(6) O(log2n), O(nlog2n) (6) (84,79,56,38,40,46)

(7) O(nlog2n), O(n2) (8) O(log2n), O(n)

(9) 两端, 中间 (10) [38 40]46[56 79 84]

(11) 4, 4 (12) O(n), O(nlog2n), O(n)

(13) 5, 4, 8 (14) [38 46 56 79][40 84]

9-29 当所有待排序记录的排序码都相等，计算下列排序方法的运行时间：

(1) 直接插入排序； (2) 堆排序；

(3) 起泡排序； (4) 直接选择排序。

【解答】

(1) n-1 (2) 2 ⎣n/2⎦ ( n为奇数), 2 ⎣n/2⎦ -1( n为偶数)

(3) n-1 (4) n(n-1)/2

9-30 填空题

(1) 对n个不同的排序码进行起泡排序，在（ ① ）情况下排序码比较次数最小；在（ ② ）情况下排序码比较次数最大。

(2) 快速排序在（③ ）情况下最不利于发挥其长处，在（④）情况下最易发挥其长处。

(3) 将5个不同的数据进行排序，最少需要（ ⑤）次数据比较，最多需要（ ⑥ ）次数据比较。

(4) 堆的形状是一棵（ ⑦ ）树。

1. 就平均时间而言，（ ⑧ ）排序最好。

【解答】

① 排序对象已经按其排序码从小到大排好

② 排序对象已经按其排序码从大到小排好，现要求全部逆转

③ 排序对象已经按其排序码从小到大排好

④ 每次划分对象定位后，左侧子序列与右侧子序列长度相同的

⑤ 5

⑥ 25

⑦ 完全二叉

⑧ 快速

9-31 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：

③

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 顶点 | *a* | *b* | *c* | *d* | *e* |
| 入度 | 4 | 3 | 2 | 1 | 0 |

④

*c*

*a*

⑦

②

①

*e*

⑤

⑥

*d*

*b*

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*。

9-32 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 = mid**-**1**;** j >= i**;** j***--*** ) A[j+1] = A[j]**;**

A[i] = A[mid+1]**;**

**for** ( j = mid+2**;** j <= right**;** j++ )

**if** ( temp > A[j] ) A[j**-**1] = A[j]**;**

**else break;**

A[j**-**1] = temp**;**

**}**

**}**

**}**

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：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | left |  |  |  | mid |  | mid+1 |  | right |  |
| A | 0 | 1 | 2 | 3 | 4 | temp | 5 | 6 | 7 |  |
| i=0 | **12** | 28 | 35 | 42 | 67 |  | **09** | 31 | 70 | A[i] > A[mid+1] |
|  |  |  |  |  |  |  |  |  |  | 记录移动8次 |
| i=1 | 09 | **12** | 28 | 35 | 42 | 67 | **31** | 67 | 70 | A[i]≤A[mid+1] |
|  |  |  |  |  |  |  |  |  |  | 记录移动0次 |
| i=2 | 09 | 12 | **28** | 35 | 42 |  | **31** | 67 | 70 | A[i]≤A[mid+1] |
|  |  |  |  |  |  |  |  |  |  | 记录移动0次 |
| i=3 | 09 | 12 | 28 | **35** | 42 |  | **31** | 67 | 70 | A[i] > A[mid+1] |
|  |  |  |  |  |  |  |  |  |  | 记录移动4次 |
| i=4 | 09 | 12 | 28 | 31 | **35** | 42 | **42** | 67 | 70 | A[i]≤A[mid+1] |
|  |  |  |  |  |  |  |  |  |  | 记录移动0次 |

(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。

若设前一个表中有n = mid-left+1个元素，后一个表中有m = right-mid个元素。

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n，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，此时，记录排序码比较次数约为



记录移动次数约为

9-33 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**;**

**}**

**}**