第 10 章 内排序

教材中练习题及参考答案

1. 直接插入排序算法在含有n个元素的初始数据正序、反序和数据全部相等时，时间复杂度各是多少？

答：含有n个元素的初始数据正序时，直接插入排序算法的时间复杂度为$O(n)$。

含有n个元素的初始数据反序时，直接插入排序算法的时间复杂度为$O(n^2)$。

含有n个元素的初始数据全部相等时，直接插入排序算法的时间复杂度为$O(n)$。

2. 回答以下关于直接插入排序和折半插入排序的问题：

（1）使用折半插入排序所要进行的关键字比较次数，是否与待排序的元素的初始状态有关？

（2）在一些特殊情况下，折半插入排序比直接插入排序要执行更多的关键字比较，这句话对吗？

答：（1）使用折半插入排序所要进行的关键字比较次数，与待排序的元素的初始状态无关。无论待排序序列是否有序，已形成的部分子序列是有序的。折半插入排序首先查找插入位置，插入位置是判定树失败的位置（对应外部节点），失败位置只能在判定树的最下两层上。

（2）一些特殊情况下，折半插入排序的确比直接插入排序需要更多的关键字比较，例如，在待排序序列正序的情况下便是如此。

3. 有以下关于排序的算法：

```
void fun(int a[],int n)
{   int i,j,d,tmp;
    d=n/3;
    while (true)
    {   for (i=d;i<n;i++)
        {   tmp=a[i];
            j=i-d;
            while (j>=0 && tmp<a[j])
            {   a[j+d]=a[j];
                j=j-d;
            }
            a[j+d]=tmp;
        }
        if (d==1) break;
        else if (d<3) d=1;
```

```
            else d=d/3;
        }
    }
```

（1）指出fun(*a*，*n*)算法的功能。

（2）当*a*[]={5，1，3，6，2，7，4，8}时，问fun(*a*，8)共执行几趟排序，各趟的排序结果是什么？

答：（1）fun(*a*，*n*)算法的功能是采用增量递减为1/3的希尔排序方法对*a*数组中元素进行递增排序。

（2）当*a*[]={5，1，3，6，2，7，4，8}时，执行fun(*a*，8)的过程如下：

d=2：2 1 3 6 4 7 5 8

d=1：1 2 3 4 5 6 7 8

共有两趟排序。

4. 在实现快速排序的非递归算法时，可根据基准元素，将待排序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($\log_2 n$)。

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。在排序过程中每次划分都把整个待排序序列根据基准元素划分为左、右两个子序列，然后对这两个子序列进行类似的处理。设$S(n)$为对n个记录进行快速排序时平均所需栈的深度，则：

$$S(n)=\frac{1}{n}\sum_{k=1}^{n}(S(k-1)+S(n-k))=\frac{2}{n}\sum_{i=0}^{n-1}S(i)$$

当n=1时，所需栈空间为常量，由此可推出：$S(n)$=O($\log_2 n$)。

实际上，在快速排序中下一趟首先对较短子序列排序，并不会改变所需栈的深度，所以所需栈的深度仍为O($\log_2 n$)。

5. 将快速排序算法改为非递归算法时通常使用一个栈，若把栈换为队列会对最终排序结果有什么影响？

答：在执行快速排序算法时，用栈保存每趟快速排序划分后左、右子区间的首、尾地址，其目的是为了在处理子区间时能够知道其范围，这样才能对该子区间进行排序，但这与处理子区间的先后顺序没什么关系，而仅仅起存储作用（因为左、右子区间的处理是独立的）。因此，用队列同样可以存储子区间的首、尾地址，即可以取代栈的作用。在执行快速排序算法时，把栈换为队列对最终排序结果不会产生任何影响。

6. 在堆排序、快速排序和二路归并排序中：

（1）若只从存储空间考虑，则应首先选取哪种排序方法，其次选取哪种排序方法，最后选取哪种排序方法？

（2）若只从排序结果的稳定性考虑，则应选取哪种排序方法？

（3）若只从最坏情况下的排序时间考虑，则不应选取哪种排序方法？

答：（1）若只从存储空间考虑，则应首先选取堆排序（空间复杂度为O(1)），其次选取快速排序（空间复杂度为O($\log_2 n$)），最后选取二路归并排序（空间复杂度为O(n)）。

（2）若只从排序结果的稳定性考虑，则应选取二路归并排序，其他两种排序方法是不

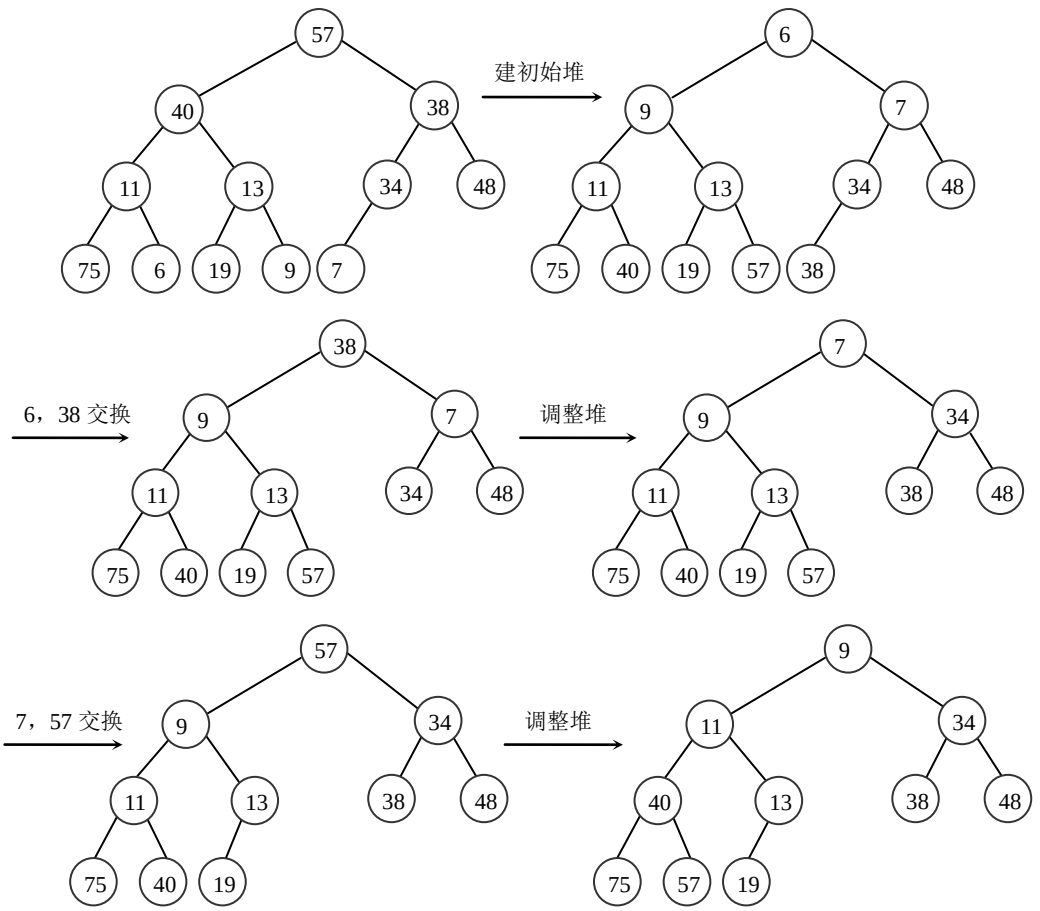
稳定的。

（3）若只从最坏情况下的排序时间考虑，则不应选取快速排序方法。因为快速排序方法最坏情况下的时间复杂度为$O(n^2)$，其他两种排序方法在最坏情况下的时间复杂度为$O(n\log_2 n)$。

7. 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（57，40，38，11，13，34，48，75，6，19，9，7），要得到其第 4 个元素（k=4）之前的部分有序序列，用所选择的算法实现时，要执行多少次比较？

答：采用堆排序最合适，建立初始堆（小根堆）所花时间不超过$4n$，每次选出一个最小元素所花时间为$\log_2 n$，因此得到第k个最小元素之前的部分序列所花时间大约为$4n+k\log_2 n$，而冒泡排序和简单选择排序所花时间为kn。

对于序列（57，40，38，11，13，34，48，75，6，19，9，7），形成初始堆（小根堆）并选最小元素6，需进行18次比较；选次小元素7时，需进行5次比较；再选元素9时，需进行6次比较；选元素11时，需进行4次比较，总共需进行33次比较。整个过程如图10.2所示。



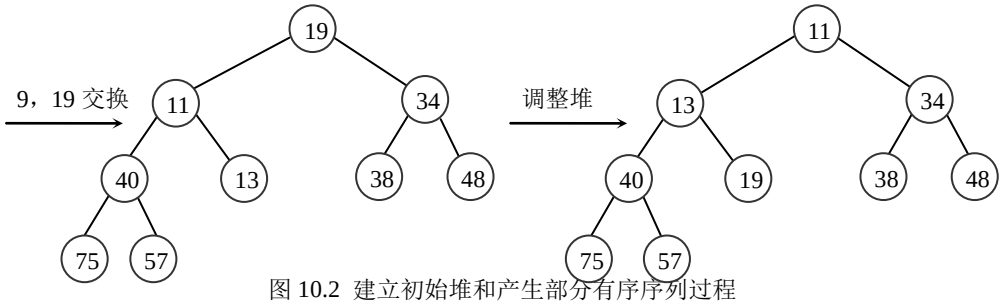


图 10.2 建立初始堆和产生部分有序序列过程

8. 基数排序过程中用队列暂存排序的元素，是否可以用栈来代替队列？为什么？

答：不能用栈来代替队列。

基数排序是一趟一趟进行的，从第2趟开始必须采用稳定的排序方法，否则排序结果可能不正确，若用栈代替队列，这样可能使排序过程变得不稳定。

9. 线性表有顺序表和链表两种存储方式，不同的排序方法适合不同的存储结构。对于常见的内部排序方法，说明哪些更适合于顺序表？哪些更适合于链表？哪些两者都适合？

答：更适合于顺序表的排序方法有希尔排序、折半插入排序、快速排序、堆排序和归并排序。

更适合于链表的排序方法是基数排序。

两者都适合的排序方法有直接插入排序、冒泡排序和简单选择排序。

10. 设一个整数数组 $a[0..n-1]$中存有互不相同的 n 个整数，且每个元素的值均在 1～n 之间。设计一个算法在 O(n)时间内将 a 中元素递增排序，将排序结果放在另一个同样大小的数组 b 中。

解：对应的算法如下：

```
void fun(int a[],int n,int b[])
{   int i;
    for (i=0;i<n;i++)
        b[a[i]-1]=a[i];
}
```

11. 设计一个双向冒泡排序的算法，即在排序过程中交替改变扫描方向。

解：置 i 的初值为 0，先从后向前从无序区 $R[i..n-i-1]$归位一个最小元素 $R[i]$到有序区 $R[0..i-1]$，再从前向后从无序区 $R[i..n-i-1]$归位一个最大元素到有序区 $R[n-i..n-1]$。当某趟没有元素交换时，则结束；否则将 i 增加 1。对应的算法如下：

```
void DBubbleSort(RecType R[],int n)  //对 R[0..n-1]按递增序进行双向冒泡排序
{   int i=0,j;
    bool exchange=true;              //exchange 标识本趟是否进行了元素交换
    while (exchange)
    {   exchange=false;
        for (j=n-i-1;j>i;j--)
            if (R[j].key<R[j-1].key)    //由后向前冒泡小元素
            {   exchange=true;
                swap(R[j],R[j-1]);      //R[j]和 R[j-1]交换
```

```
            }
        for (j=i;j<n-i-1;j++)
            if (R[j].key>R[j+1].key)     //由前向后冒泡大元素
            {   exchange=true;
                swap(R[j],R[j+1]);       //R[j]和 R[j+1]交换
            }
        if (!exchange) return;
        i++;
    }
}
```

12. 假设有 n 个关键字不同的记录存于顺序表中，要求不经过整体排序而从中选出从大到小顺序的前 m（$m<<n$）个元素。试采用简单选择排序算法实现此选择过程。

解：改进后的简单选择排序算法如下：

```
void SelectSort1(RecType R[],int n,int m)
{   int i,j,k;
    for (i=0;i<m;i++)            //做第 i 趟排序
    {   k=i;
        for (j=i+1;j<n;j++)      //在当前无序区 R[i..n-1]中选 key 最大的 R[k]
            if (R[j].key>R[k].key)
                k=j;             //k 记下目前找到的最大关键字所在的位置
        if (k!=i)
            swap(R[i],R[k]);     //交换 R[i]和 R[k]
    }
}
```

13. 对于给定的含有 n 元素的无序数据序列（所有元素的关键字不相同），利用快速排序方法求这个序列中第 k（$1\leqslant k\leqslant n$）小元素的关键字，并分析所设计算法的最好和平均时间复杂度。

解： 采用快速排序思想求解，当划分的基准元素为 $R[i]$时，根据 i 与 k 的大小关系再在相应的子区间中查找。对应的算法如下：

```
KeyType QuickSelect(RecType R[],int s,int t,int k) //在 R[s..t]序列中找第 k 小的元素
{   int i=s,j=t;
    RecType tmp;
    if (s<t)                     //区间内至少存在 2 个元素的情况
    {   tmp=R[s];                //用区间的第 1 个记录作为基准
        while (i!=j)             //从区间两端交替向中间扫描,直至 i=j 为止
        {   while (j>i && R[j].key>=tmp.key)
                j--;             //从右向左扫描,找第 1 个关键字小于 tmp 的 R[j]
            R[i]=R[j];           //将 R[j]前移到 R[i]的位置
            while (i<j && R[i].key<=tmp.key)
                i++;             //从左向右扫描,找第 1 个关键字大于 tmp 的 R[i]
            R[j]=R[i];           //将 R[i]后移到 R[j]的位置
        }
        R[i]=tmp;
        if (k-1==i) return R[i].key;
        else if (k-1<i) return QuickSelect(R,s,i-1,k); //在左区间中递归查找
        else return QuickSelect(R,i+1,t,k);                //在右区间中递归查找
    }
```

```
    else if (s==t && s==k-1)     //区间内只有一个元素且为R[k-1]
        return R[k-1].key;
    else
        return -1;                    //k 错误返回特殊值-1
}
```

对于 QuickSelect(*R*，*s*，*t*，*k*)算法，设序列 *R* 中含有 *n* 个元素，其比较次数的递推式为：

$T(n)=T(n/2)+O(n)$

可以推导出 $T(n)=O(n)$，这是最好的情况，即每次划分的基准恰好是中位数，将一个序列划分为长度大致相等的两个子序列。在最坏情况下，每次划分的基准恰好是序列中的最大值或最小值，则处理区间只比上一次减少 1 个元素，此时比较次数为 $O(n^2)$。在平均情况下该算法的时间复杂度为 $O(n)$。

14. 设 *n* 个记录 *R*[0..*n*−1]的关键字只取 3 个值：0，1，2。采用基数排序方法将这 *n* 个记录排序。并用相关数据进行测试。

解：采用基数排序法，将关键字为 3 个值的记录分别放到 3 个队列中，然后收集起来即可。对应的算法如下：

```
#include "seqlist.cpp"                //顺序表基本运算算法
#include <malloc.h>
#define Max 3
typedef struct node
{   RecType Rec;
    struct node *next;
} NodeType;
void RadixSort1(RecType R[],int n)
{   NodeType *head[Max],*tail[Max],*p,*t;//定义各链队的首尾指针
    int i,k;
    for (i=0;i<Max;i++)               //初始化各链队首、尾指针
        head[i]=tail[i]=NULL;
    for (i=0;i<n;i++)
    {   p=(NodeType *)malloc(sizeof(NodeType));//创建新节点
        p->Rec=R[i];
        p->next=NULL;
        k=R[i].key;                   //找第 k 个链队,k=0,1 或 2
        if (head[k]==NULL)            //进行分配,采用前插法建表
        {   head[k]=p; tail[k]=p; }
        else
        {   tail[k]->next=p; tail[k]=p; }
    }
    p=NULL;
    for (i=0;i<Max;i++)               //对于每一个链队进行循环收集
        if (head[i]!=NULL)            //产生以 p 为首节点指针的单链表
        {   if (p==NULL)
            {   p=head[i]; t=tail[i]; }
            else
            {   t->next=head[i]; t=tail[i]; }
```

```
        }
        i=0;
        while (p!=NULL)              //将排序后的结果放到R[]数组中
        {   R[i++]=p->Rec;
            p=p->next;
        }
}
```

设计如下主函数：

```
int main()
{   int i,n=5;
    RecType R[MAXL]={{1,'A'},{0,'B'},{0,'C'},{2,'D'},{1,'F'}};
    printf("排序前:\n  ");
    for (i=0;i<n;i++)
        printf("[%d,%c] ",R[i].key,R[i].data);
    printf("\n");
    RadixSort1(R,n);
    printf("排序后:\n  ");
    for (i=0;i<n;i++)
        printf("[%d,%c] ",R[i].key,R[i].data);
    printf("\n");
    return 1;
}
```

程序执行结果如下：

```
排序前:
  [1,A]  [0,B]  [0,C]  [2,D]  [1,F]
排序后:
  [0,B]  [0,C]  [1,A]  [1,F]  [2,D]
```

显然，RadixSort1()算法的时间复杂度为 O(n)。