第5章 数据结构的应用

CHAPTER

作者寄语：本章是数据结构的应用篇，包括初阶和进阶两个应用篇。初阶应用的排序和查找，主要介绍一些解决查找和排序问题的算法技巧。进阶应用是以旅行商问题为例和当前的人工智能算法相结合，是将数据结构专业基础课知识应用到实际中的一个探索。

本章摘要：本章的第一个部分阐述的是各种排序算法，第二个部分是查找算法，这两个部分都是初阶应用篇；第三个部分是进阶应用篇，是以旅行商问题为例介绍组合优化问题的各种人工智能算法。

重点内容：旅行商问题的人工智能优化算法。

关键词：排序、查找、旅行商问题、人工智能算法。

导读手册：5.1 节阐述简单排序、先进排序以及基数排序算法等；5.2 节介绍静态查找、动态查找以及哈希表算法等；解决旅行商问题的群智能算法以及最新的 AI 算法在 5.3 节中讲述。

5.1 初阶应用：排序

本节首先介绍排序的定义与分类，然后阐述具体的排序方法及相应的算法。

1. 排序的定义

为了便于查找和提高查找效率，对于含有 n 条记录$\{R_1,R_2,\cdots,R_n\}$的序列，记录 $R_1,R_2,\cdots,R_n$ 的值对应的关键字为：$K_1,K_2,\cdots,K_n$，对这些关键字进行排序，使记录的值满足递减或递增的关系，这样就形成了按关键字排列的序列：

$$\{R_{k1},R_{k2},\cdots,R_{kn}\}$$

2. 排序的分类

(1) 根据待排序记录所在位置的不同分类。

内部排序：待排序记录存放在内存。

外部排序：排序过程中需对外存进行访问。

(2) 对于内部排序,依据不同的排序原则分类。

插入排序:直接插入排序、折半插入排序和 2-路插入排序。

交换排序:希尔排序、冒泡排序、快速排序。

选择排序:简单选择排序、堆排序。

归并排序:2-路归并排序。

基数排序:根据位数的排序。

(3) 针对内部排序所需的时间复杂度分类。

简单排序:直接插入排序、希尔(Shell)排序、冒泡排序和直接选择排序。

先进排序:快速排序、堆排序和归并排序。

基数排序:基数排序是这几种排序中效率最高的排序。

(4) 排序关键字可能出现重复,根据重复关键字的排序情况分类。

稳定排序:排序后重复关键字记录的相对次序保持不变。

不稳定排序:与稳定排序相反。

排序数据的物理结构定义:

```
#define MAXSIZE  20;
type int KeyType;
typedef struct
    {   KeyType key;
        InfoType otherinfo;
    } RcdType;
typedef struct
    {   RcdType r[MAXSIZE+1];
        int length;
    } SqList;
```

其中,MAXSIZE 是顺序表最大长度,key 是关键字,otherinfo 为其他数据,length 为顺序表长度,r[MAXSIZE+1]闲置或作为标志位。

5.1.1 简单排序

在简单排序算法中,主要介绍 4 种算法:插入排序,希尔(Shell)排序、冒泡排序和选择排序。插入排序又包括直接插入排序、折半插入排序和 2-路插入排序。简单排序算法在时间复杂度上都属于 $O(n^2)$,且都是稳定的排序算法。

1. 插入排序

插入排序的方法是通过依次插入的过程来完成排序的功能。插入排序包括直接插入排序以及直接插入排序的改进版:折半插入排序和 2-路插入排序。

1) 直接插入排序

n 个记录的直接插入排序的整个排序过程为 $n-1$ 趟插入,即先将序列中第一个记录看成一个有序序列,从第二个记录开始逐个进行有序插入,直至整个数据都在有序序列中为止。例如,一个含有 6 个数的序列{16, 13, 12, 88, 56, 77},要求用直接插入法升序排

列。具体的做法是：首先，建立一个长度为 7 的数组 $r[]$用于存放排序后的有序记录，其中，$r[0]$设置为监视位，存放待插入的数据。在第一趟排序中，待插入数据 16 存入监视位 $r[0]=16$，因为只有一个记录，不用比较则有 $r[1]=16$；在第二趟排序中，待插入数据 13 存入监视位 $r[0]=13$，$r[0]$的值 13 小于 $r[1]$的值 16，记录 16 后移，则有 $r[1]=13$，$r[2]=16$；在第三趟排序中，待插入的监视位 $r[0]=12$，和有序序列里的 $r[1]=13$，$r[2]=16$ 比较后，记录 13 和 16 后移，则有 $r[1]=12$，$r[2]=13$，$r[3]=16$；以此类推，最后插入排序的升序序列为{12,13,16,56,77,88}。

伪代码描述的直接插入排序的算法：

```
void InsertSort ( SqList &L)
{   for ( i=2; i<=L.length; ++i )
    if ( LT( L.r[i].key,L.r[i-1].key ))
    {   L.r[0] = L.r[i];
        L.r[i]=L.r[i-1];
        for ( j=i-2; LT( L.r[0].key, L.r[j].key) ; --j )
            L.r[j+1] = L.r[j];
        L.r[j+1] = L.r[0];
    }
}
```

for (i=2; i<=L.length; ++i)这个语句的意思是从待插入的第二个数据开始比较直至所有数据，即数据的长度。if (LT(L.r[i].key,L.r[i-1].key))的意思是新插入的记录和它前一个插入的记录比较，如果新记录值小，L.r[0] = L.r[i]将新插入的记录放至监视位，前一个插入的记录后移。for (j=i-2; LT(L.r[0].key , L.r[j].key) ; --j)这个循环是用来判断新插入的记录比前一个记录小时，是否比前两个记录甚至更前的记录也小，进而找到要插入的位置。

上述代码包括双层循环，因此直接排序算法的时间复杂度是 $O(n^2)$；只占用了一个监视位空间，其空间复杂度为 $O(1)$。

从上述直接排序算法的执行过程可以看出，如果待插入的数值比较小，查找插入位置要从有序数组的尾找到头，查找的时间会很长，如果每次从有序序列的中间位置开始比较和查找，这样无论要插入数值的大小都能节省一半的查找位置的时间。

2) 折半插入排序

折半插入排序算法就是把监视位 $r[0]$放到有序序列中间位置的排序算法。中间监视位置的计算是记录的个数除以 2。依然用含有 6 个数的序列{16, 13, 12, 88, 56, 77}为例，第一趟排序，有序序列中只有一个记录 16，监视位 $r[0]=16$；第二趟，拟插入数据值为 13，小于监视位 $r[0]=16$，插入 16 的前面(13,16)，监视位(1+1)/2=1，$r[0]=13$；第三趟，拟插入数据值为 12，小于监视位 $r[0]=13$，插入 13 的前面(12,13,16)，监视位(1+1+1)/2≈2，$r[0]=13$；第四趟拟插入的数值是 88，大于监视位 $r[0]$的数值 13，从 13 往后找位置，有序序列为(12,13,16,88)，监视位(1+1+1+1)/2=2，$r[0]$的数值为 13；第五趟拟插入的数值是 56，大于监视位 $r[0]$的数值 13，从 13 往后找位置，有序序列为(12,

13,16,56,88),监视位(1+1+1+1+1)/2≈3,r[0]的数值为 16;第六趟拟插入的数值是 77,大于监视位 r[0]的数值 16,从 16 往后找位置,有序序列为(12,13,16,56,77,88),监视位(1+1+1+1+1+1)/2=3,r[0]的数值为 16。

总之,折半插入排序,每趟都计算中间的监视位 r[0]的值,将拟插入的数据和监视位的值比较,大则从监视位往后找位置插入,小则从监视位向前找位置插入。这样,在插入记录时,比较个数就比直接插入算法减少了一半。

代码描述的折半插入排序算法:

```
void BInsertSort (SqList &L)
{   for ( i=2; i<=L.length; ++i )
    {   L.r[0] = L.r[i];
        low = 1; high = i-1;
        while (low<=high)
        {   m = (low+high)/2;
            if (LT(L.r[0].key,L.r[m].key)) high = m-1;
            else low = m+1;
        }
        for ( j=i-1; j>=high+1; --j )
            L.r[j+1] = L.r[j];
        L.r[high+1] = L.r[0];
    }
}
```

在上述代码中,待插入的数据仍然存入标志位 $L.r[0] = L.r[i]$,折半插入又增加了两个位置信息 low 和 high。初始时:low = 1,指示第一个记录的位置;high = $i-1$,指示有序序列中的最后一个位置(也是待插入数据前一个的位置)。在查找拟插入位置时,if (LT(L.r[0].key,L.r[m].key)) high = m−1; else low = m+1,如果待插入数据小于中间位置记录的值,则 high 的值变小,low 的值变大。直至 low=high,则 high(或者 low,因为此时 high 和 low 的位置相同)的位置就是待插入数据要插入的位置。于是,后移该位置后面的记录 $L.r[j+1] = L.r[j]$;存入待插入的数据 $L.r$[high+1] = $L.r$[0],完毕。

折半插入排序算法的时间复杂度为 $O(n^2)$,空间复杂度为 $O(1)$。值得注意的是,折半插入排序只能减少排序过程中关键字比较的时间,并不能减少记录移动的时间。

3) 2-路插入排序

2-路插入排序算法需要在除了 $L.r$[]之外增加一个数组 d[1],具体的操作过程是:先将 $L.r$[1]赋给 d[1],并将 d[1]作为标志位,$L.r$[2]及其后面的数据 $L.r[i]$都要与 d[1]进行比较,小于 d[1]的 $L.r[i]$的值放到 d[1]前面的有序表,用指针 first 指示 d[1]前面数组的最小值;$L.r[i]$大于 d[1] 的值放在后面的有序表,用指针 final 指示 d[1]后面数组的最大值。

总体来说,2-路插入排序算法是对折半插入排序算法的一个改进,因为增加了两个指针 first 和 final,使得比较次数进一步减少,但是 2-路插入排序算法的时间复杂度仍然

为 $O(n^2)$，空间复杂度为 $O(2)$。

这里仍以 6 个数的序列{16，56，12，88，13，77}为例，进行 2-路插入排序的过程如图 5.1 所示。

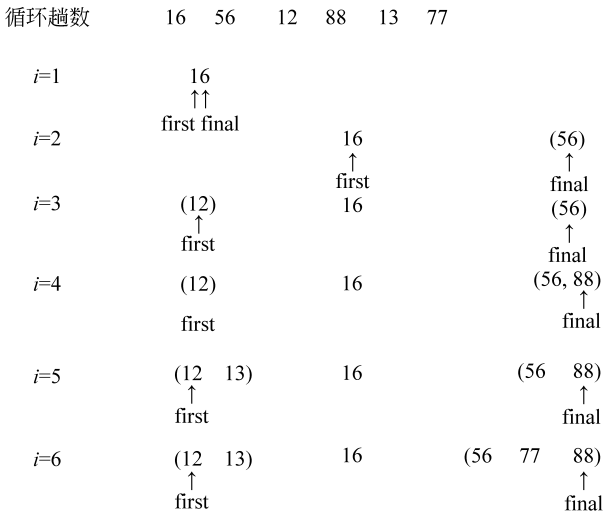

图 5.1　2-路插入排序过程示例

$L.r[1]$的值 16 赋给 $d[1]$，$d[1]$实际上起到一个标志位的作用，所有后面的数据都和 $d[1]=16$ 进行比较，小于 16 的，插入前面的序列，大于 16 的，插入后面的序列，当所有数据插入完毕，最后 2-路插入排序的升序完成。

2. 希尔排序

希尔(Shell)排序与插入排序不同，它是以一种内部交换的形式进行排序。希尔排序也被称为缩小增量排序，其通过对一定距离的两个记录进行比较，根据大小对两个记录进行位置互换。每趟都缩小调整的距离直至距离为 1，最后完成排序的功能。希尔(Shell)排序中距离 d_i 的设定原则是 $d_1<n$，n 是一个正整数，$d_2<d_1$，…，而最后一趟的距离 $d_i=1$。

这里以 6 个数的序列{16，56，12，88，13，77}为例，展示希尔排序的具体实现过程，如图 5.2 所示。

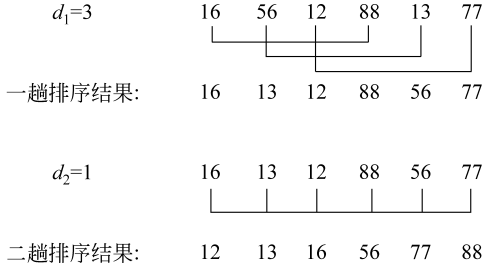

图 5.2　希尔排序过程示例

图 5.2 中的第一趟是间隔距离为 3 的两个记录之间进行比较，小的交换到前面的位置，56 比 13 大，于是 13 和 56 的位置互换，其他两两比较的数据都是前面的小，因此不互换。当间隔距离为 1 的两个记录之间进行比较时，小的数据都换到前面，大的数据都换到后面的位置。

代码描述的希尔排序算法：

```
void ShellInsert ( SqList &L,int dk)
{   for ( i=dk+1; i<=L.length; ++i )
    if (LT(L.r[i].key , L.r[i-dk].key ))
    {   L.r[0] = L.r[i];
        for (j=i-dk; j>0&&LT(L.r[0].key, L.r[j].key); j-=dk )
            L.r[j+dk] = L.r[j];
        L.r[j+dk] = L.r[0];
    }
}
void ShellSort (SqList &L, int dlta[], int t)
{   for (k=0; k<t; ++t)
    ShellInsert (L, dlta[k]);
}
```

ShellSort()函数是负责改变间隔距离值的大小，然后 ShellInsert()函数就是按照间隔距离对数据进行两两比较，小的如果在后面就互换，把小的值放到前面。

希尔排序的时间复杂度和所取间隔增量的值有关。已有学者证明，当增量序列为 $2^{t-k-1}(k=0,1,\cdots,t-1)$时，希尔排序的时间复杂度为 $O(n^{3/2})$，但是一般情况下，其时间复杂度还是 $O(n^2)$。总体来说，希尔排序数值较小的记录会跳跃式前移，但是增量不为 1 时不能保证记录都有序，所以，最后一趟的增量值必须为 1。

3. 冒泡排序

冒泡排序也是一种以内部数据交换位置，数据记录像气泡在水中上浮一般冒出的排序算法。升序排序是小的气泡(记录)在上面，大的气泡(记录)在下面，降序排序则相反。

升序冒泡排序算法的过程是：第一趟，将 n 个记录中的第一个记录的关键字(值)与第二个记录的关键字(值)进行比较，若 $r[1].key>r[2].key$，则**交换**；然后比较第二个记录与第三个记录；以此类推，直至第 $n-1$ 个记录和第 n 个记录比较为止，结果关键字最大的记录被换到最后一个，即第 n 个记录的位置。第二趟，对前 $n-1$ 个记录进行冒泡排序，结果关键字次大的记录被安置在第 $n-1$ 个记录位置。重复上述过程，直到在一趟排序过程中没有记录交换的操作为止。

代码描述的冒泡算法：

```
void BubbleSort (SqList &L)
{   for ( i = 1; i <= L.length; i++ )
    {   for(j = 1; j < L.length - i; j++ )
        {   if (GT(L.r[j].key, L.r[j+1].key))
            {   L.r[0] = L.r[j];
```

```
                L.r[j] = L.r[j+1];
                L.r[j+1] = L.r[0];
            }
        }
    }
}
```

当记录的关键字相同时，记录位置不会发生移动，所以冒泡排序算法是一种稳定的排序算法，其时间复杂度仍然为 $O(n^2)$，空间复杂度为 $O(1)$。

4. 选择排序

选择排序的思想是：第一趟选择未排序序列 n 个记录中最小(或最大)的关键字，将其与第一个记录交换位置；第二趟从剩余的 $n-1$ 个记录中找出关键字次小的记录，将它与第二个记录交换；重复上述操作，共进行 $n-1$ 趟排序后，将序列从小到大排列，选择排序结束。

代码描述的选择排序算法：

```
void SelectSort (SqList &L)
{   for (i=1; i<L.length; ++i)
    {   j = i;
        for ( k=i+1; k<=L.length; k++ )
            if ( LT( L.r[k].key, L.r[j].key )) j =k ;
        if ( i!=j ) L.r[j]↔L.r[i];
    }
}
```

由于当记录的关键字相同时，记录位置不会发生移动，简单选择排序算法属于稳定的排序算法，其时间复杂度仍然为 $O(n^2)$，空间复杂度为 $O(1)$。

5.1.2　先进排序

先进排序，其实就是排序算法的时间复杂度不是 $O(n^2)$ 而是 $O(n\log n)$，算法的运行效率提高了，主要包括快速排序、归并排序和堆排序三种排序算法。

1. 快速排序

快速排序的基本思想和 2-路插入排序类似，都是以第一个数据作为比较标志位，其他数据都和标志位比大小，一趟循环后，第一个数据被放置到近乎中间的位置，在它之前部分的记录的关键字都比它小，在它后面部分的关键字都比它大。以此类推，再分别对前后两个部分快速排序。只不过 2-路插入排序要移动记录时的操作是采取插入的方式，而快速排序则是互换位置。

文字描述的快速排序算法：

有数据序列为 $r[i,\cdots,j]$，指针 i 指示第一个记录，指针 j 指示最后一个记录，标志位 rp 存入序列的第一个记录的值：rp.key= $r[i]$。

步骤 1：从最后一个记录 j 向前找第一个关键字小于第一个记录的标志位 rp.key 的值，并和标志位 rp 交换位置。

步骤 2：再从 $i+1$ 的位置向后找，找到第一个关键字大于标志位 rp.key 的值的记录，并和标志位 rp 交换位置。

步骤 3：重复上述两步，直至 $i==j$ 为止，数据序列被分为前后两个子序。

步骤 4：再分别对两个子序列进行快速排序，直到每个子序列只含有一个记录为止。

注：在一趟排序中①被比较的标志位 rp 的值 rp.key 始终不变，是该序列的第一个记录的值；②比较后互换的位置，始终是和标志位 rp 的位置进行交换。

示例：有数据序列(16,56,12,88,13,77)，分别存储在数组 $r[1]=16,\cdots, r[6]=77$ 中，要求按升序快速排序。

按步骤 1，将第一个记录设置成标志位 $r[0]=16$，从 $r[6]$开始往前逐个与标志位 $r[0]=16$ 比较，找到第一个小于 16 的 $r[5]=13$，于是 13 和 16 的位置互换；再进行步骤 2，从第二个记录往后找第一个比 16 大的记录，于是 $r[2]=56$ 与 16 的位置互换；重复上述步骤 1 和 2，从 $r[5]$往前找第一个比 16 小的记录，于是 $r[3]=12$ 与 16 互换。一趟排序的结果为(13,12,16,88,56,77)，以标志位 16 最后位置来看，16 前面的记录的关键字都比 16 小，后面的记录的关键字都比 16 大，这样就完成了一趟排序。第二趟排序则是分别对一趟排序的结果：子序列(13,12)和(88,56,77)分别排序，直到每个子序列里只有一个记录位置，全部排序完成。第一趟快速排序过程如图 5.3 所示。

	r[0]	r[1]	r[2]	r[3]	r[4]	r[5]	r[6]
第一趟排序:	16	16	56	12	88	13	77
第1次互换:	16	13	56	12	88	16	77
第2次互换:	16	13	16	12	88	56	77
第3次互换:	16	13	12	16	88	56	77

图 5.3　第一趟快速排序过程

代码描述的快速排序算法：

```
int Partition ( SqList &L, int low, int high)
{   L.r[0] = L.r[low];
    pivotkey = L.r[low].key;
    while (low < high)
    {   while (low < high && L.r[high].key >= pivotkey) --high;
        L.r[low++] = L.r[high];
        while (low < high && L.r[low].key <= pivotkey)  ++low;
        L.r[high] = L.r[low];
    }
    L.r[low] = L.r[0];
    return low;
}
```

在算法复杂度上，关键字序列的排序过程是一个近似的完全二叉树，而树的深度即为初始关键字序列的分解次数，为 $\log_2 n$ 次。此外，对于关键字序列而言，不论怎样划分或分解，元素的比较次数都接近于 $n-1$ 次，因此，此种情况下的算法复杂度为 $T(n)=O(n\log_2 n)$。又当记录的关键字相同时，记录位置会发生移动，所以快速排序被认为是较好的不稳定的排序方法。

2. 归并排序

归并排序就是将两个或两个以上的有序表组合成一个新的有序表的过程，常采用 2-路归并排序。

文字描述的 2-路归并排序算法如下。

步骤 1：将有 n 个记录的数据，看成长度为 1 的 n 个有序的子序列。

步骤 2：两两合并排序，得到长度为 2(n 如果是奇数，则包括一个长度为 1 的子序列)的有序子序列，子序列个数是 n/2 个。

步骤 3：再两两合并，直至形成一个长度为 n 的有序序列为止。

代码描述的 2-路归并排序算法：

```
void Merge (RcdType SR[], RcdType &TR[], int i, int m, int n)
{   for (j=m+1, k=i; i<=m && j<=n; ++k)
    {   if ( LQ(SR[i].key, SR[j].key)) TR[k] = SR[i++];
        else TR[k] = SR[j++];
    }
    while (i<=m) TR[k++] = SR[i++];
    while (j<=n) TR[k++] = SR[j++];
}
```

示例：有待排序数据(16,56,12,88,13,77)，要求用 2-路归并算法排序。

首先，将待排序列两个数据一组分成三个子序列，对每个子序列内部进行排序。一趟归并后，每个子序列内部有序。然后，再将两个子序列进行归并排序合并为一个序列。通过三趟归并后，整个序列变为有序序列，排序完成，如图 5.4 所示。

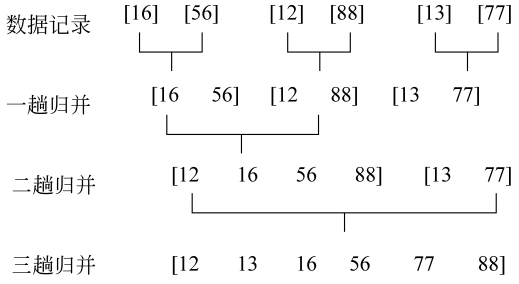

图 5.4　归并排序过程

2-路归并排序的形式也类似一棵二叉树，因此，2-路归并排序算法的时间复杂度也为 $T(n)=O(n\log_2 n)$。因为归并排序采用了 n 个临时空间来存放数据，所以 2-路归并排序算法的空间复杂度为 $S(n)=O(n)$。又由于当记录的关键字相同时，记录位置不会发生

移动，所以归并排序算法属于稳定的排序算法。

3. 堆排序

堆排序是一种每趟选择一个最小(最大)的根结点的排序，其记录分布形似一棵树。堆这棵树的特点是根结点小于(或大于)左右孩子结点，小于左右孩子的根结点的叫小顶堆，反之叫大顶堆。当一棵无序树调整为堆，输出根结点之后，就用树的最后一个结点替代根结点，这时候一趟堆排序完成。堆排序的过程就是每趟输出堆顶结点的最小(或最大)值，堆顶根结点被树的最后一个结点替代的过程，n 趟调整就输出 n 个堆顶结点，进而完成堆排序。

示例：有无序树数据(16,56,12,88,13,77)，如图 5.5(a)所示，要求对该数据用小顶堆排序。

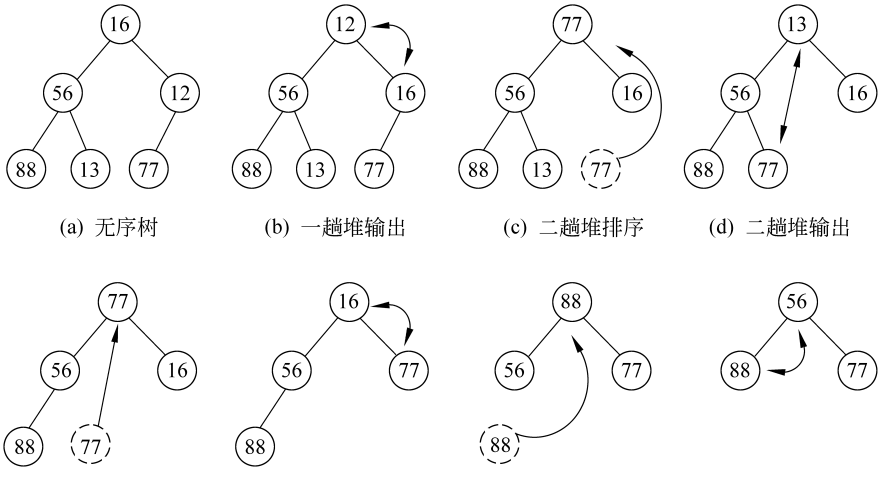

图 5.5 堆排序示例

在图 5.5(a)中，根的右孩子 12 最小，于是根结点 16 与 12 互换，输出新的根结点 12，用树中最后一个结点 77 替换根结点 12，完成一趟堆排序和输出，如图 5.5(b)所示。二趟堆排序如图 5.5(c)所示，结点 13 小于根结点 77，互换并输出根结点 13，如图 5.5(d)所示。三趟排序，结点 16 小于根结点 77，互换并用输出 16，并用最后一个结点 88 替代根结点。四趟排序，结点 56 与根结点 88 互换，输出 56，用最后一个结点 77 替代根结点。五趟排序，输出根结点 77，用最后一个结点 88 替代根结点 77。六趟排序，输出根结点 88，至此，全部结点输出完毕，堆排序结束。堆排序结果为(12,13,16,56,77,88)。

堆排序算法时间复杂度为 $O(n\log n)$，堆排序只需要一个辅助存储空间，因此空间复杂度为 $O(1)$。由于堆排序筛选的调整方法，不可避免地会造成相同关键字结点相对位置发生变化，因此堆排序是一种不稳定的排序算法。

5.1.3 基数排序

5.1.1 节与 5.1.2 节介绍的排序算法时间复杂度分别为 $O(n^2)$ 和 $O(n\log n)$，本节介绍