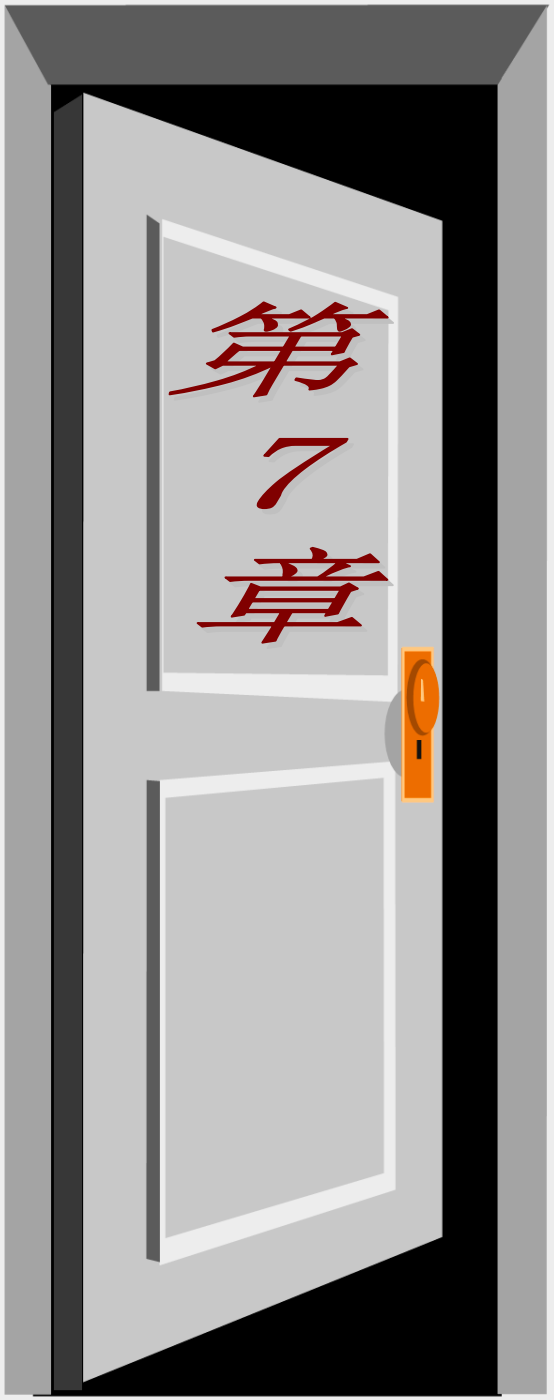

尚德机构

# 数据结构

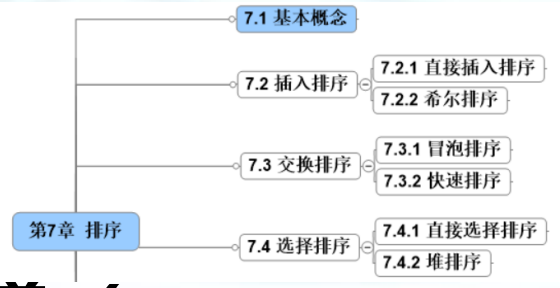
主讲：王老师

学习是一种信仰！ IN LEARING WE TRUST




# 排 序

# 7.1　基本概念




▲*数据排序*——将一个文件的记录按关键字递增（或递减）次序排列，使文件成为有序文件，此过程称为排序。

▲*稳定排序*——如果待排序文件中存在多个关键字相同的记录，经过排序后，这些具有相同关键字的记录之间的相对次序保持不变。

▲*不稳定排序*——反之，则是不稳定的。

▲排序类型——
- *内部排序*：全部数据存于内存；
- *外部排序*:需要对外存进行访问的排序过程


内部排序方法又可以分为五类：**插入、选择、交换、归并和分配排序**。

待排序记录的存储方式一般有三种：**顺序结构、链式结构和辅助表形式**。

评价排序算法的标准主要有两条：**执行算法需要的时间，以及算法所需要的附加空间。**

在本章的讨论中，若无特别声明，都假定排序操作是按递增要求的，排序文件以顺序表作为存储结构，并假定关键字为整数。


定义待排序记录的数据类型如下：

```
#define MAXSIZE 100
typedef int KeyType;
typedef struct {
    KeyType key;
    InfoType other info;
} RecType;
typedef RecType SeqList[MAXSIZE+1];  //表中0元素空着或用作哨兵单元
SeqList R;                           //R为待排序的记录文件
```


每次将一个待排序的记录按其关键字的大小插入到前面已排好序的文件中的适当位置，直到全部记录插入完为止。插入排序主要包括**直接插入排序和希尔排序**两种。

# 7.2.1 直接插入排序


```
void InsertSort(SeqList R, int n)
{   //对顺序表R做直接插入排序
    int i, j;
    for(i=2; i<=n; i++)
        if(R[i].key<R[i-1].key) {          //若R[i].key>=有序区中所有的key，则R[i]不动
            R[0]=R[i];                     //当前记录复制为哨兵
            for(j=i-1; R[0].key<R[j].key; j--)  //找插入位置
                R[j+1]=R[j];               //记录后移
            R[j+1]=R[0];                   //R[i]插入到正确位置
        }
}
```

算法中的R[0]有两个作用，一个作用是在进入查找循环之前保存R[i]的副本；但主要还是用来在查找循环中“监视”数组下标变量j是否越界，一旦越界（即j=0），R[0].key自比较，使循环条件不成立而结束循环。因此，常把R[0]称为哨兵。


直接插入排序的时间复杂性有很大差别。最好的情况是文件初始为正序，此时的时间复杂度是O(n)。最坏的情况是文件初始状态为反序，相应的时间复杂度为$O(n^2)$。容易证明，该算法的平均时间复杂度也是**$O(n^2)$**。

直接插入排序算法是**稳定**的。

# 7.2.1 直接插入排序


| | |
|---|---|
| 初始关键字： | [46] [39, 17, 23, 28, 55, 18, 46] |
| i=2 R[0].key=39 | [39, 46] [17, 23, 28, 55, 18, 46] |
| i=3 17 | [17, 39, 46] [23, 28, 55, 18, 46] |
| i=4 23 | [17, 23, 39, 46] [28, 55, 18, 46] |
| i=5 28 | [17, 23, 28, 39, 46] [55, 18, 46] |
| i=6 55 | [17, 23, 28, 39, 46, 55] [18, 46] |
| i=7 18 | [17, 18, 23, 28, 39, 46, 55] [46] |
| i=8 46 | [17, 18, 23, 28, 39, 46, 46, 55] |

# 7.2.2二、希尔排序



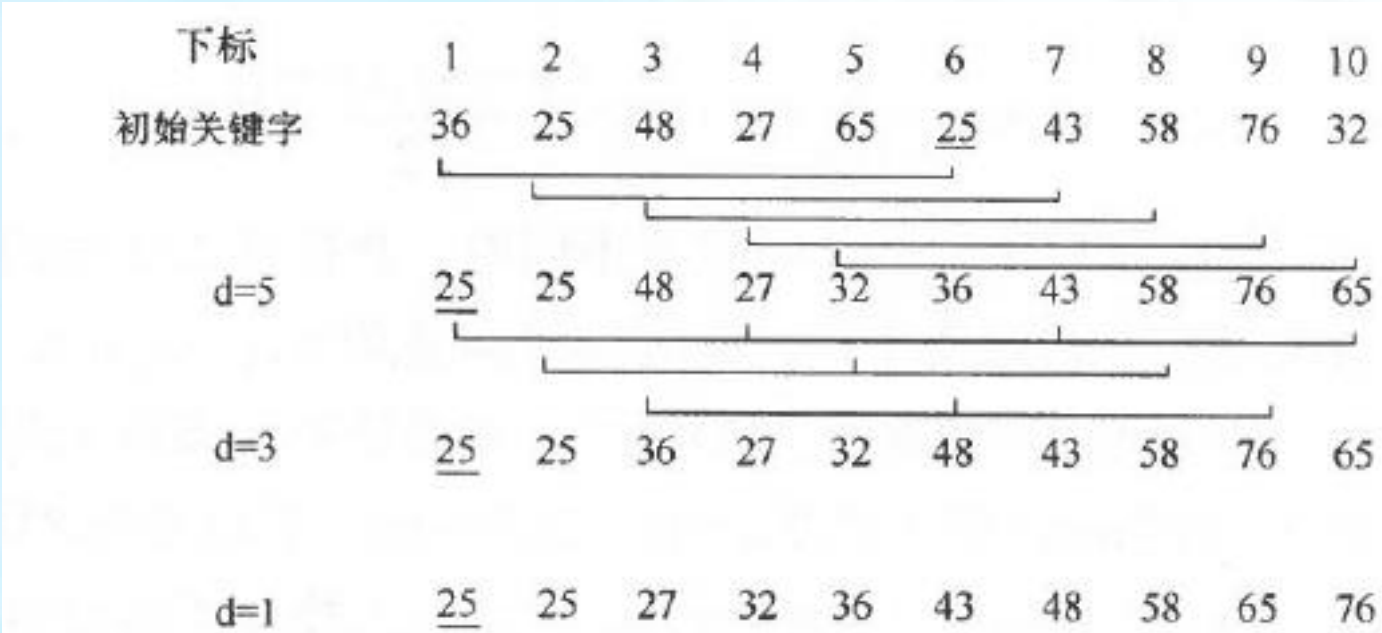
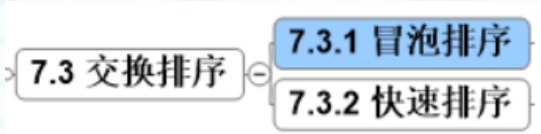



图 7.1 希尔排序示例

希尔排序是**不稳定**的。

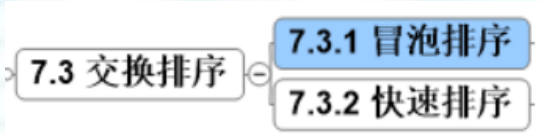



交换排序的基本思想是：两两比较待排序记录的关键字，如果发现两个记录的次序相反时即进行交换，直到所有记录都没有反序时为止。

本节将介绍两种交换排序方法：**冒泡排序和快速排序。**

## 7.3.1 冒泡排序

冒泡排序（Bubble Sort）是一种简单的排序方法，其基本思想是：通过相邻元素之间的比较和交换，使关键字较小的元素逐渐从底部移向顶部，就像水底下的气泡一样逐渐向上冒泡，所以使用该方法的排序称为“冒泡”排序。当然，随着排序关键字较小的元素逐渐上移（前移），排序关键字备大的元素也逐渐下移（后移），小的上浮，大的下沉，所以冒泡排序又被称为“起泡”排序。

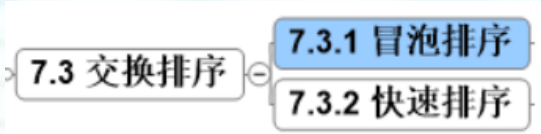



交换排序的基本思想是：两两比较待排序记录的关键字，如果发现两个记录的次序相反时即进行交换，直到所有记录都没有反序时为止。

本节将介绍两种交换排序方法：**冒泡排序和快速排序。**

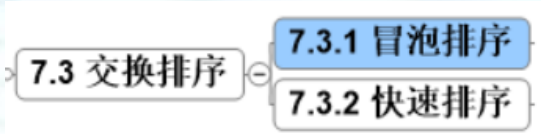
## 7.3.1 冒泡排序

冒泡排序（Bubble Sort）是一种简单的排序方法，其基本思想是：通过相邻元素之间的比较和交换，使关键字较小的元素逐渐从底部移向顶部，就像水底下的气泡一样逐渐向上冒泡，所以使用该方法的排序称为“冒泡”排序。当然，随着排序关键字较小的元素逐渐上移（前移），排序关键字备大的元素也逐渐下移（后移），小的上浮，大的下沉，所以冒泡排序又被称为“起泡”排序。




| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 初始关键字 | [36 | 28 | 45 | 13 | 67 | 36 | 18 | 56] |
| 第一趟 | 13 | [36 | 28 | 45 | 18 | 67 | 36 | 56] |
| 第二趟 | 13 | 18 | [36 | 28 | 45 | 36 | 67 | 56] |
| 第三趟 | 13 | 18 | 28 | [36 | 36 | 45 | 56 | 67] |
| 第四趟 | 13 | 18 | 28 | 36 | [36 | 45 | 56 | 67] |

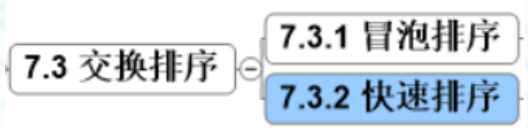
图7.2 自后向前扫描的冒泡排序示例




```
void BubbleSort(SeqList R, int n)
{  //采用自底向上扫描数组R[1..n]做冒泡排序
    int i, j, flag;
    for(i=1; i<n; i++) {                //最多做n-1趟排序
        flag=0;                         //flag表示每一趟是否有交换，先置0
        for(j=n; j>=i+1; j--)           //进行第i趟排序
        if(R[j].key<R[j-1].key) {
            R[0]=R[j-1];                //R[0]作为交换时的暂存单元
            R[j-1]=R[j];
            R[j]=R[0];
                flag=1;                 //有交换，flag置1
        }
        if(flag==0) return;
    }
}
```

若待排序记录为有序（最好情况），则一趟扫描完成，关键比较次数为n-1次且没有移动，比较的时间复杂度为O(n)，冒泡排序算法的时间复杂度为$O(n^2)$。稳定排序。

# 7.3.2二、快速排序




快速排序（Quick sort）又称为划分交换排序。快速排序是对冒泡排序的一种改进方法，在冒泡排序中，进行记录关键字的比较和交换是在相邻记录之间进行，记录每次交换只能上移或下移一个相邻位置，因而总的比较和移动次数较多。在快速排序中，记录关键字的比较和记录的交换是从两端向中间进行的，待排序关键字较大的记录一次就能够交换到后面单元中，而关键字较小的记录一次就能够交换到前面单元中，记录每次移动的距离较远，因此总的比较和移动次数较少，速度较快，故称为“快速排序”。

# 7.3.2 快速排序



初始关键字 [45 53 18 49 36 76 13 97 36 32]

x.key=45 i j

[32 53 18 49 36 76 13 97 36 32]

i j

[32 53 18 49 36 76 13 97 36 53]

i j

[32 36 18 49 36 76 13 97 36 53]

i j

[32 36 18 49 36 76 13 97 36 53]

i j

[32 36 18 49 36 76 13 97 49 53]

i j

[32 36 18 49 36 76 13 97 49 53]

i j

[32 36 18 13 36 76 13 97 49 53]

i j

[32 36 18 13 36 76 13 97 49 53]

i j

[32 36 18 13 36 76 76 97 49 53]

i=j

[32 36 18 13 36] 45 [76 97 49 53]

（a）一趟划分排序过程

依次可得各趟排序结果如下：

| | |
|---|---|
| 初始关键字 | [45 53 18 49 36 76 13 97 36 32] |
| 一次划分后 | [32 36 18 13 36] 45 [76 97 49 53] |
| 二次划分后 | [13 18] 32 [36 36] 45 [53 49] 76 [97] |
| 三次划分后 | 13 [18] 32 36 [36] 45 [49] 53 76 97 |
| 最后的结果 | 13 18 32 36 36 45 49 53 76 97 |

（b）快速排序的每趟结果

## 算法分析：

- 空间：$\log_2 n$；（$\log_2 n$为附加空间—栈）
- 时间：$O(n\log_2 n)$

  注：若初始记录表有序或基本有序，则快速排序将蜕化为冒泡排序,其时间复杂度为$O(n^2)$；

  即：快速排序在表基本有序时，最不利于其发挥效率。
- 稳定性：不稳定排序。


每一趟在待排序的记录中选出关键字最小的记录，依次存放在已排好序的记录序列的最后，直到全部记录排序完为止。本节主要介绍直接选择排序和堆排序两种选择排序方法。

# 7.4.1 直接选择排序



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 初始关键字： | [38 | 33 | 65 | 82 | 76 | 38 | 24 | 11] |
| 一趟排序后： | 11 | [33 | 65 | 82 | 76 | 38 | 24 | 38] |
| 二趟排序后： | 11 | 24 | [65 | 82 | 76 | 38 | 33 | 38] |
| 三趟排序后： | 11 | 24 | 33 | [82 | 76 | 38 | 65 | 38] |
| 四趟排序后： | 11 | 24 | 33 | 38 | [76 | 82 | 65 | 38] |
| 五趟排序后： | 11 | 24 | 33 | 38 | 38 | [82 | 65 | 76] |
| 六趟排序后： | 11 | 24 | 33 | 38 | 38 | 65 | [82 | 76] |
| 七趟排序后： | 11 | 24 | 33 | 38 | 38 | 65 | 76 | [82] |

图 7.4　直接选择排序过程示例

# 7.4.1 直接选择排序


```
void SelectSort(SeqList R, int n)
{   //对R作直接选择排序
    int i, j, k;
    for(i=1 ;i<n; i++) {              //做n-1趟排序
        k=i;                          //设k为第i趟排序中关键字最小的记录位置
        for(j=i+1; j<=n; j++)         //在[i..n]选择关键字最小的记录
            if(R[j].key<R[k].key)
                k=j;                  //若有比R[k].key小的记录，记住该位置
        if(k!=i) {                    //与第i个记录交换
            R[0]=R[i]; R[i]=R[k]; R[k]=R[0];
        }
    }
}
```

直接选择排序的平均时间复杂度为$O(n^2)$

排序法是不稳定的

# 7.4.2二、堆排序



堆的定义如下：n个记录的关键字序列$k_1$，$k_2$，…，$k_n$称为堆，当且仅当满足以下关系：

$$k_i \le k_{2i} 且 k_i \le k_{2i+1}, 或 k_i \ge k_{2i} 且 k_i \ge k_{2i+1} \qquad (1 \le i \le \lfloor n/2 \rfloor)$$

前者称为小根堆，后者称为大根堆。例如关键字序列（76, 38, 59, 27, 15, 44）就是一个大根堆，还可以将此调整为小根堆（15, 27, 44, 76, 38, 59），它们对应的完全二叉树如图7.5所示。

堆排序正是利用大根堆（或小根堆）来选取当前无序区中关键字最大（或最小）的记录实现排序的。每一趟排序的操作是：将当前无序区调整为一个大根堆，选取关键字最大的堆顶记录，将它和无序区中最后一个记录交换，这正好与选择排序相反。堆排序就是一个不断建堆的过程。


（a）大根堆　　（b）小根堆

图 7.5　堆的完全二叉树表示

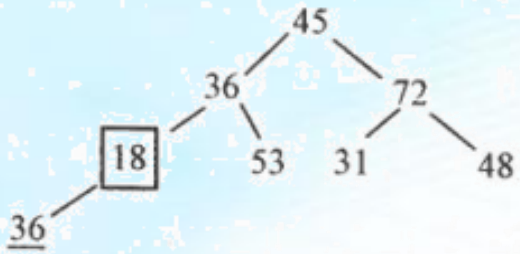
# 7.4.2 堆排序



1 2 3 4 5 6 7 8

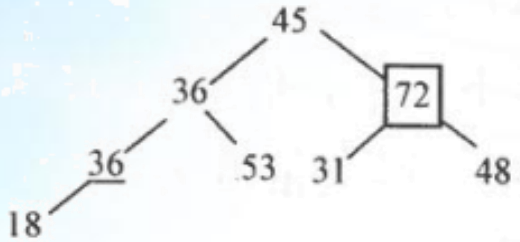
45 36 72 18 53 31 48 36

*i*=4 需要调整，因左孩子36>18



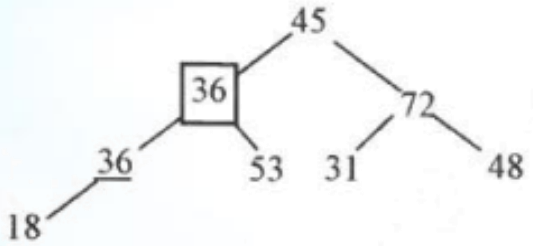

45 36 72 36 53 31 48 18

*i*=3 无需调整，因以72为根的子树已经足堆



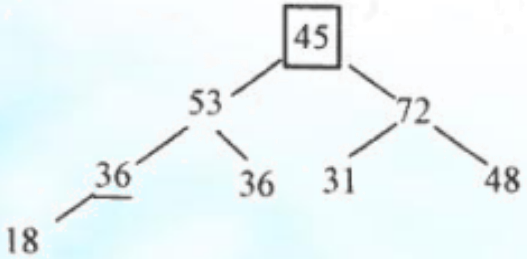

45 36 72 36 53 31 48 18

*i*=2 需要调整，因右孩子53>36



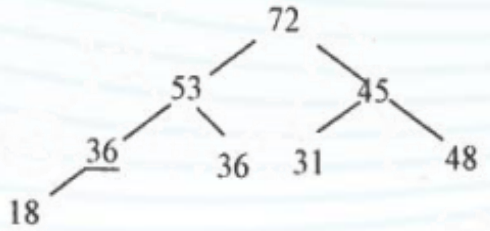

45 53 72 36 36 31 48 18

*i*=1 需要调整，因右孩子72>45




72 53 45 36 36 31 48 18

当45和75交换后，使得45为根的子树不成堆，需要调整




72 53 48 36 36 31 45 18

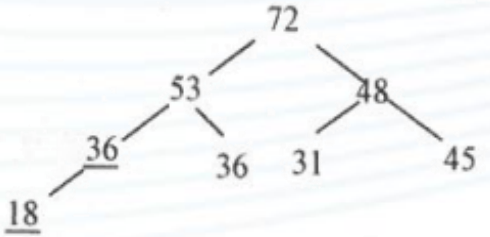



图 7.6 初始堆的建堆过程

最后得到的大根堆为：72 53 48 36 36 31 45 18

# 7.4.2 堆排序



【例7.4】已知关键字序列为（47, 33, 11, 56, 72, 61, 25, 47），采用堆排序方法对该序列进行堆排序，画出建初始堆的过程和每一趟排序结果。

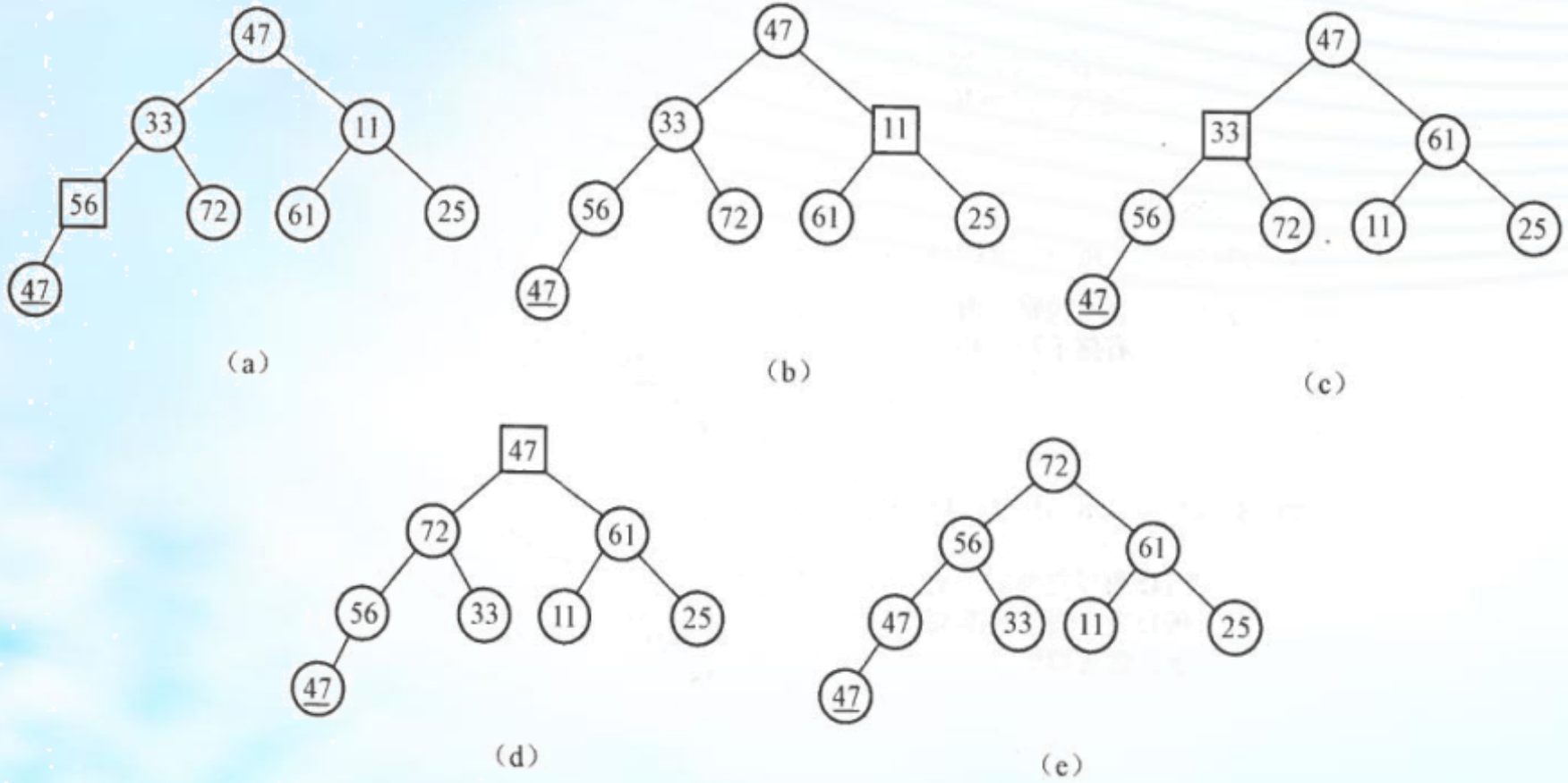



图 7.7　建堆过程

【例7.4】已知关键字序列为（47, 33, 11, 56, 72, 61, 25, 47），采用堆排序方法对该序列进行堆排序，画出建初始堆的过程和每一趟排序结果。

有了初始堆之后，就可以进行堆排序了。首先将关键字最大的堆顶记录R[1]和最后一个记录R[n](n=8)交换，这时得到的关键字序列为[47 56 61 47 33 11 25] [72]，前面方括号内是无序区，后面方括号内是有序区，也就是第一趟排序结果。因为无序区的7个关键字不为堆，因此需要将其调整为堆，调整结果为[61 56 47 47 33 11 25] [72]，由此可得到第二趟排序结果：

[25 56 47 47 33 11] [61 72]。依此类推，可得如下各趟排序结果：

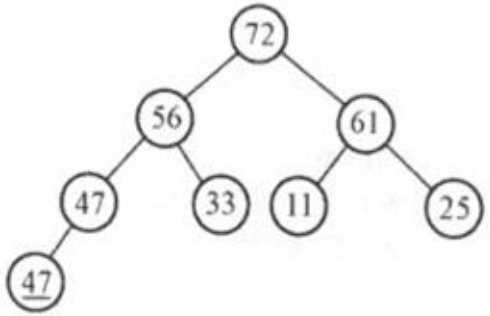
第三趟排序结果：[11 47 47 25 33] [56 61 72]

第四趟排序结果：[11 33 47 25] [47 56 61 72]

第五趟排序结果：[25 33 11] [47 47 56 61 72]

第六趟排序结果：[11 25] [33 47 47 56 61 72]

第七趟排序结果：[11] [25 33 47 47 56 61 72]




- 时间复杂度为$O(n\log_2 n)$。堆排序比直接选择排序的速度快得多。
- 堆排序和直接选择排序都是不稳定的。

# 7.5第五节 归并排序


归并排序（Merge Sor）的基本思想是：首先将待排序文件看成n个长度为1的有序子文件，把这些子文件两两归并，得到[n/2]个长度为2的有序子文件；然后再把这个有序的子文件两两并，如此反复，直到最后得到一个长度为n的有序文件为止，这种排序方法称为二路归并排序。例如，有初始关键字序列（72, 18, 53, 36, 48, 31, 36），其二路归并排序过程如图7.8所示。

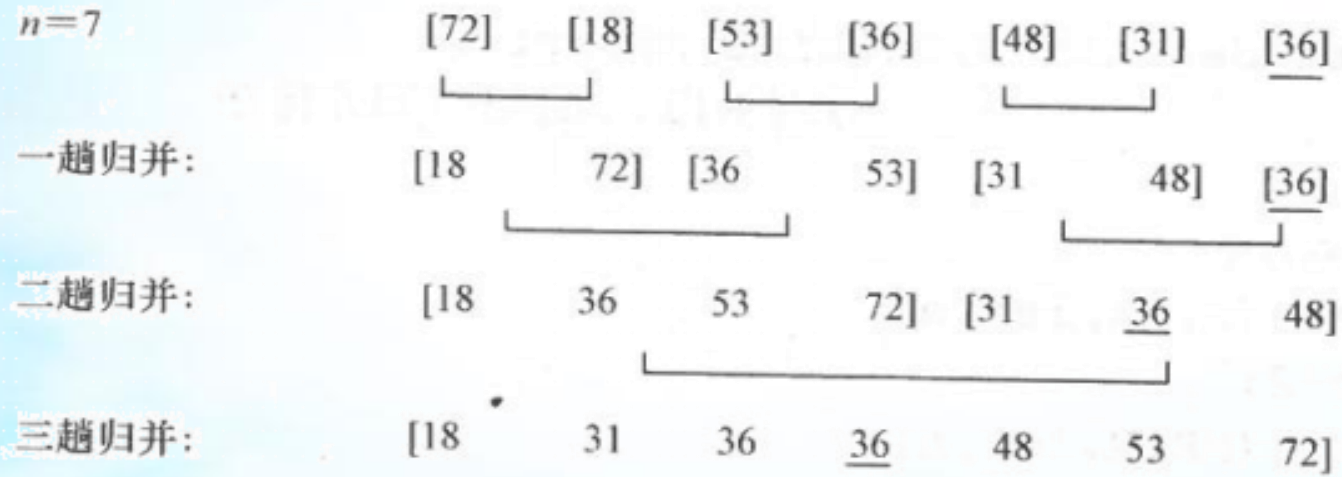
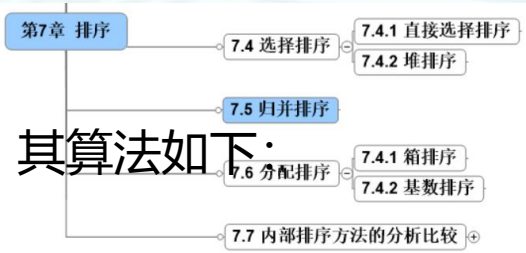



图 7.8 二路归并排序示例

# 7.5 归并排序




二路归并排序中的核心操作是将数组中前后相邻的两个有序序列归并为一个有序序列，其算法如下：

```
void Merge(SeqList R, SeqList MR, int low, int m, int high)
{ //对有序的R[low..m]和R[m+1..high]归并为有序的MR[low..high]
    int i, j, k;
    i=low; j=m+1; k=low;                    //初始化
    while(i<=m && j<=high)
        if(R[i].key<=R[j].key)
            MR[k++]=R[i++];
        else
            MR[k++]=R[j++];
    while(i<=m)
        MR[k++]=R[i++];                     //将R[low..m]中剩余的复制到MR中
    while(j<=high)
        MR[k++]=R[j++];                     //将R[m+1..high]中剩余的复制到MR中
}
```

- 二路归并排序的时间复杂度为$O(n\log_2 n)$
- 稳定的排序


**例：**

**试对下列待排序序列用归并排序法进行排序,给出每趟结果:**

**{ 475，137，481，219，382，674，350，326，815，506 }**

**第一趟：[137 475] [219 481] [382 674] [326 350] [506 815]**

**第二趟：[137 219 475 481] [326 350 382 674] [506 815]**

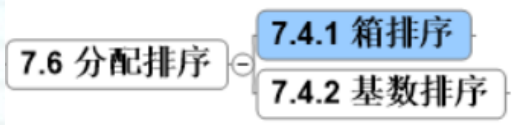
**第三趟：[137 219 326 350 382 475 481 674] [506 815]**

**第四趟：[137 219 326 350 382 475 481 506 674 815]**

**算法分析：**

- **空间：n；（需n个附加空间）**
- **时间：$O(n\log_2 n)$**
- **稳定性：稳定排序**

# 7.6第六节 分配排序 7.6.1一、箱排序




```
void BinSort(SeqList R,int n)
{                                          //对R[0..n-1]进行箱排序
    for(i=0;i<m;i++)
        SetQueue(B[i]);                    //置空所有的链队列
    for(i=0;i<n;i++){
        k=R[i].key;                        //分配
        EnQueue(B[k],R[i]);                //装箱
     }
    i=0;
    while(Empty(B[i]))                     //找第一个非空箱子
        i++;
    p=B[i].f;                              //p指向排序后的第一个记录
    for(j=i+1;j<m;j++)
        if(!Empty(B[i])) {
            将所指向记录链接到上一个非空箱子的尾指针所指向的结点之后
        }
}
```

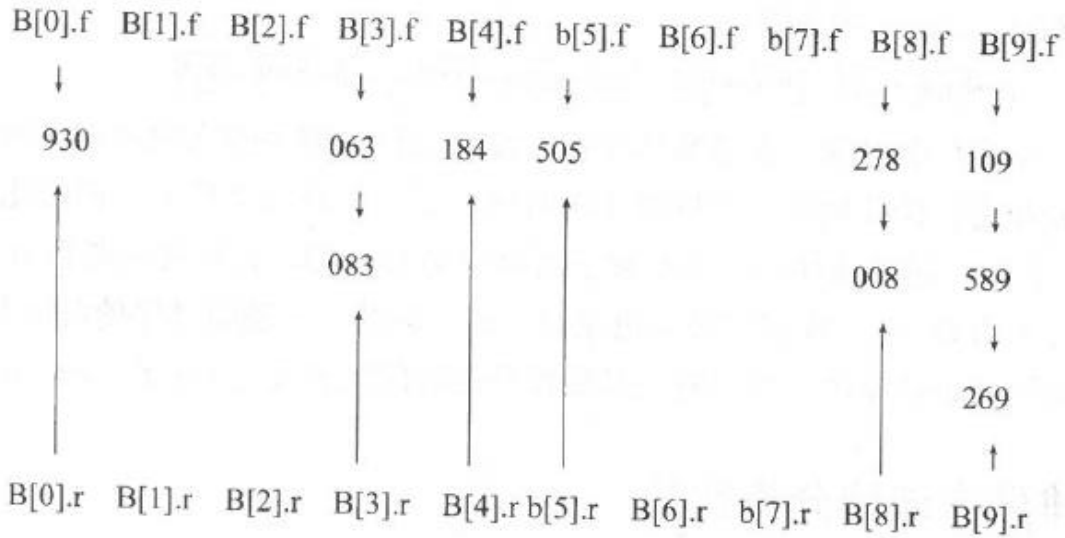
# 7.6.2二、基数排序



【例 7.5】 已知关键字序列{278，109，063，930，589，184，505，269，008，083}，写出基数排序（升序）的排序过程。

初始状态： p→278→109→063→930→589→184→505→269→008→083

第一趟分配，即按个位装箱后状态如下（按尾插法进行装入，即尾指针总是指向新插入的记录，下同）：



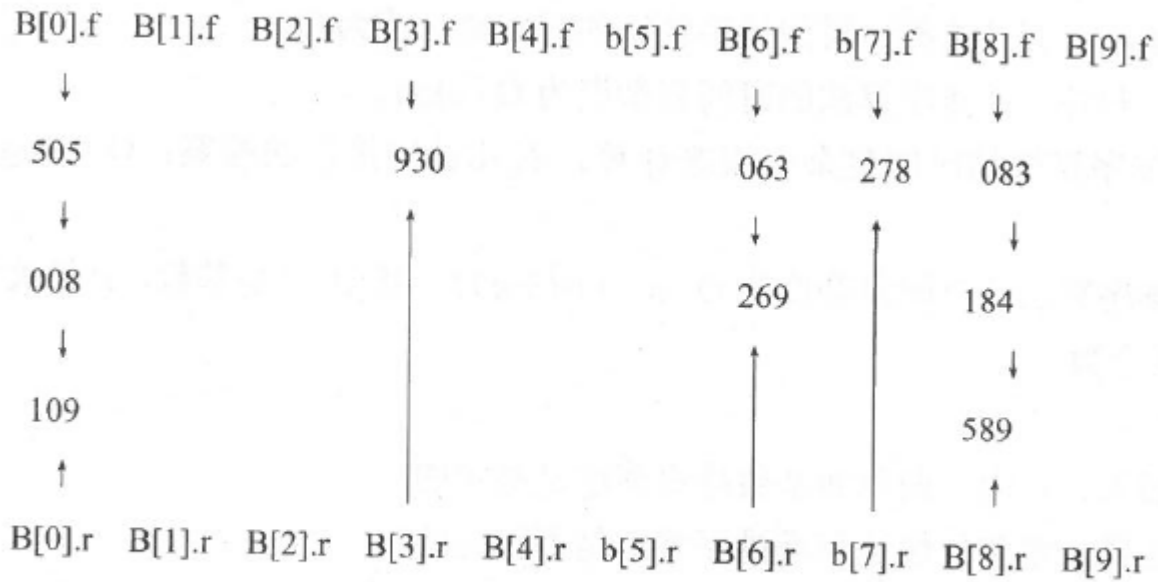

第一趟收集后，已按个位有序：

p→930→063→083→184→505→278→008→109→589→269

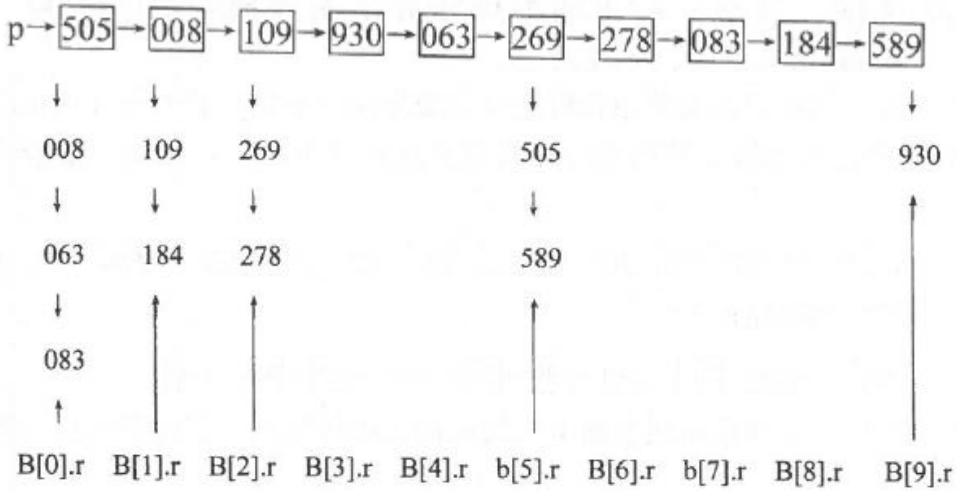
# 7.6.2二、基数排序



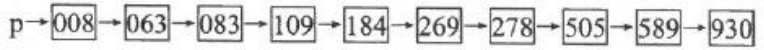
第二趟分配，即按十位装箱后的状态如下：




第二趟收集后，已按十位和个位有序：

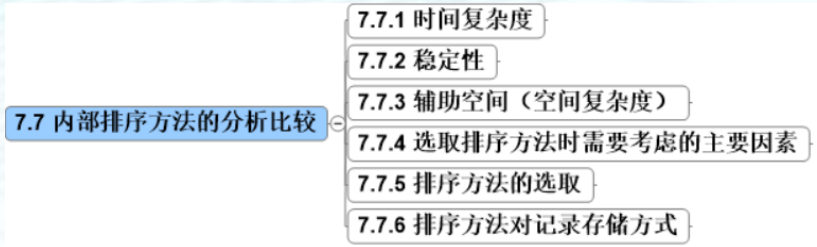



第三趟收集后，已按三位有序：



# 7.7 内部排序方法的分析比较

| 排序方法 | 时间复杂度 | 最坏情况 | 辅助存储 | 稳定性 |
|---|---|---|---|---|
| 直接插入<br>直接选择<br>冒泡排序 | $O(n^2)$<br>直接插入、冒泡最好<br>$O(n)$ | $O(n^2)$ | $O(1)$ | 插入冒泡稳定排序<br>直接选择不稳定 |
| 快速排序 | $O(n\log_2 n)$ | $O(n^2)$ | $O(\log_2 n)$ | 不稳定排序 |
| 堆排序 | $O(n\log_2 n)$ | $O(n\log_2 n)$ | $O(1)$ | 不稳定排序 |
| 归并排序 | $O(n\log_2 n)$ | $O(n\log_2 n)$ | $O(n)$ | 稳定排序 |
| 希尔排序 | 近似$O(n\log_2 n)$ | | $O(1)$ | 不稳定排序 |
| 基数排序 | | | $O(n+rd)$ | 稳定排序 |




## 5. 排序方法的选取

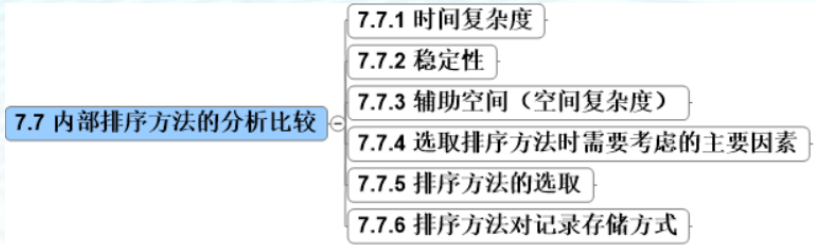
(1)若待排序的一组记录数目n较小（如n≤50）时，可采用插入排序或选择排序。

(2)若n较大时，则应采用快速排序、堆排序或归并排序。

(3)若待排序记录按关键字基本有序时，则适宜选用直接插入排序或冒泡排序。

(4)当n很大，而且关键字位数较少时，采用链式基数排序较好。

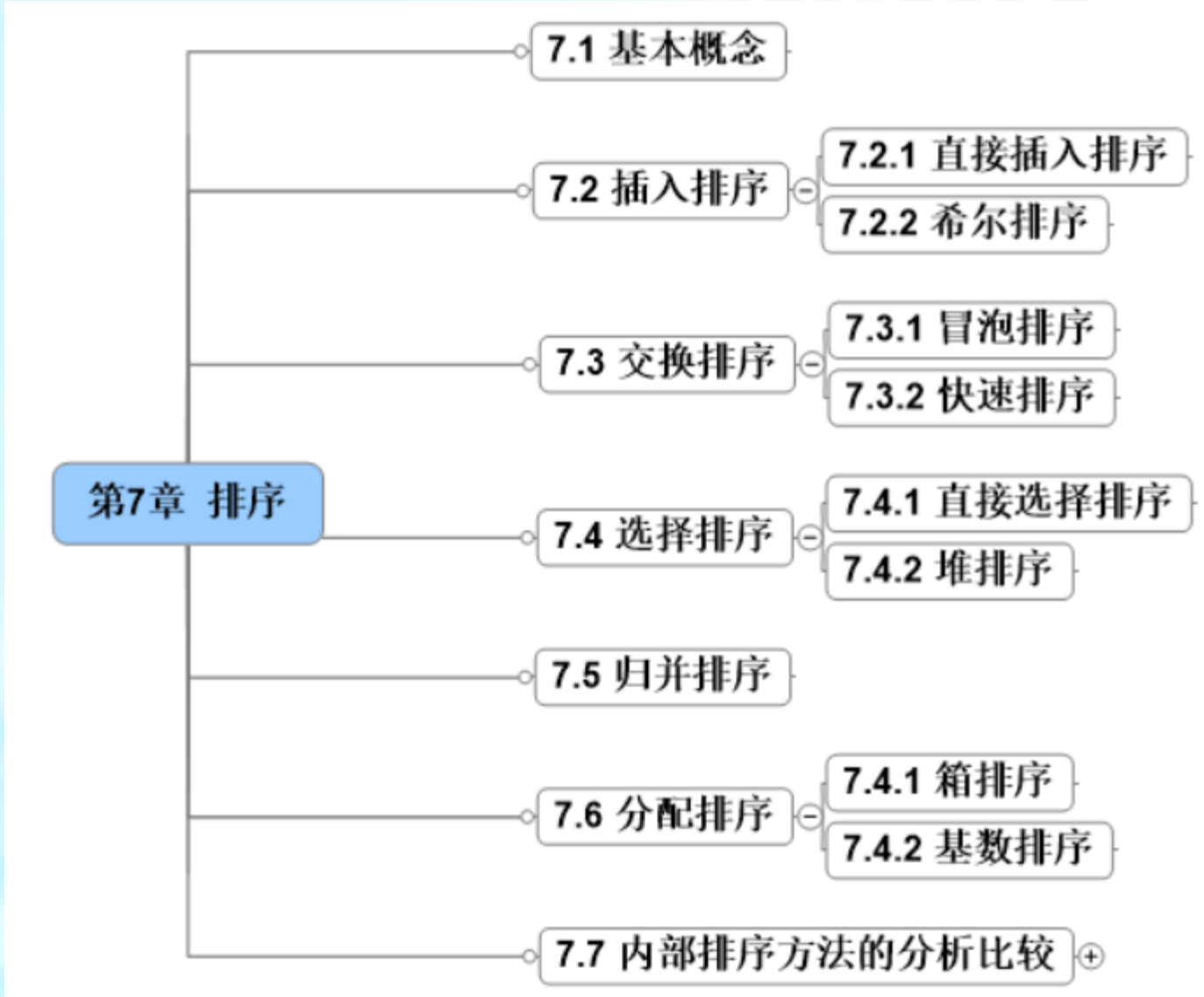
(5)关键字比较次数与记录的初始排列顺序无关的排序方法是选择排序。




## 6. 排序方法对记录存储方式的要求

一般的排序方法都可以在顺序结构（一维数组）上实现。当记录本身信息量较大时，为了避免移动记录耗费大量的时间，可以采用链式存储结构。例如插入排序、归并排序、基数排序易于在链表上实现，使之减少记录的移动次数，但有的排序方法，如快速排序、堆排序在链表上却难于实现，在这种请况下，可以提取关键字建立索引表，然后对索引表进行排序。

# 本章总结


第7章 排序
7.1 基本概念
7.2 插入排序
7.2.1 直接插入排序
7.2.2 希尔排序
7.3 交换排序
7.3.1 冒泡排序
7.3.2 快速排序
7.4 选择排序
7.4.1 直接选择排序
7.4.2 堆排序
7.5 归并排序
7.6 分配排序
7.4.1 箱排序
7.4.2 基数排序
7.7 内部排序方法的分析比较

# 祝大家顺利通过考试！