数据结构与程序设计
(Data Structure and Programming)

线性表

(Linear List)

北航计算机学院 林学练

本章内容

2.1 线性表的基本概念

2.2 线性表的顺序存储结构

2.3 线性表的排序（简单排序）

2.4 线性链表及其操作

2.5 循环链表及其操作

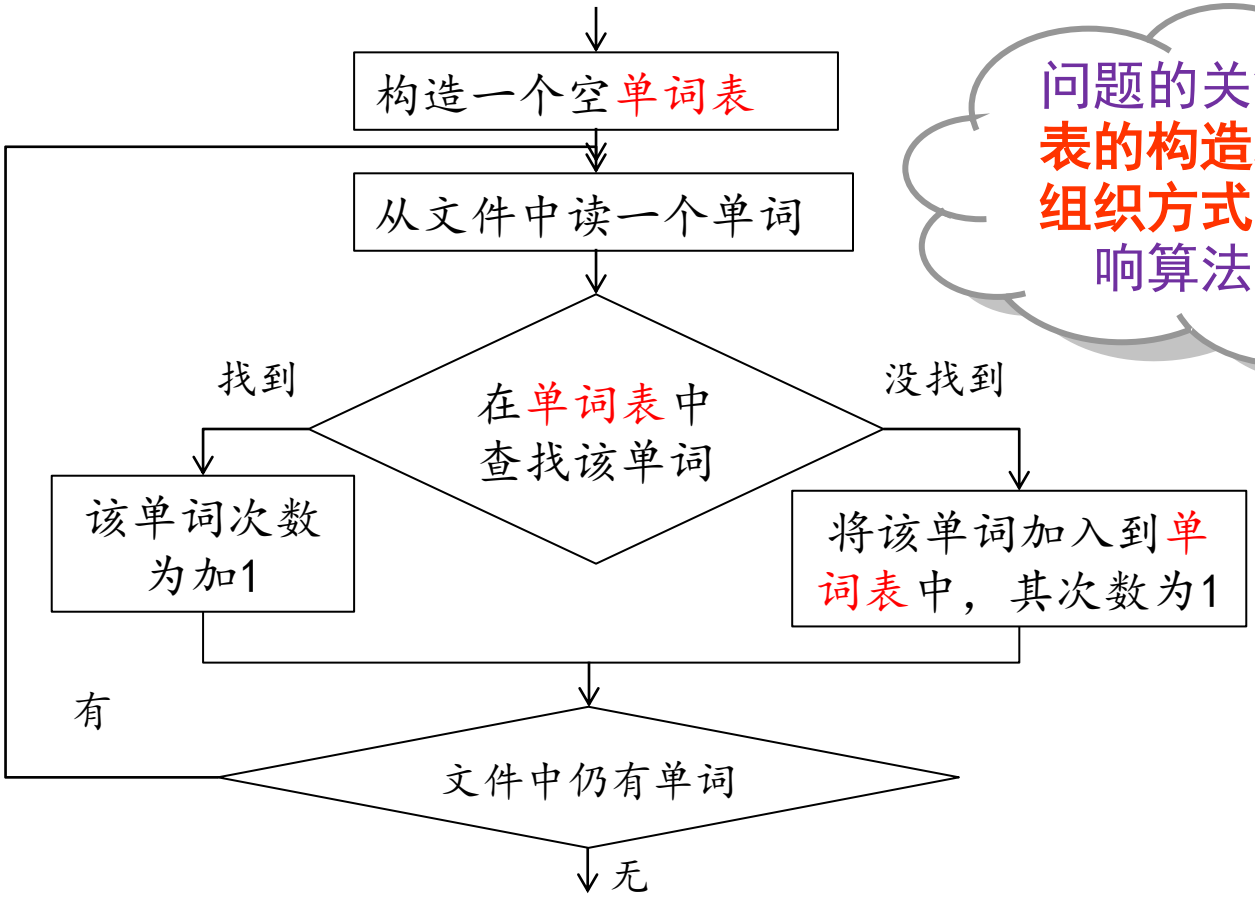
2.6 双向链表及其操作

2.7 链表应用举例

链式存储结构

问题2.1：词频统计

- 问题：编写程序统计一个文本文件中每个单词的出现次数（词频统计），并按字典序输出每个单词及出现次数。
- 算法分析：本问题算法主要是应用**查找**和**插入**操作。



问题的关键是**单词表的构造和单词的组织方式，**它将影响算法的效率

问题2.1：词频统计

- 用何种数据结构来构造和组织单词表?

用数组？链表？还是…?来构造单词表
单词表是有序还是无序？

- 不同数据结构构造的单词表如何影响着算法的性能?

当单词表较大时（如一本长篇小说），单词的查找和插入会面临什么问题？

2.1 线性表（Linear List）的基本概念

2.1.1 线性表的定义

$A=(a_1, a_2, a_3, \ldots\ldots, a_n)$

	学号	姓名	性别	年龄	其他
a_1	99001	张华	女	17	……
a_2	99002	李军	男	18	……
a_3	99003	王明	男	17	……
⋮	…	…	…	…	……
	…	…	…	…	……
	…	…	…	…	……
a_{50}	99050	刘东	女	19	……

1. 线性关系

(1) 当1<i<n时，a_i的直接前驱为a_{i-1}，a_i的直接后继为a_{i+1}。

(2) 除了第一个元素与最后一个元素，序列中任何一个元素有且仅有一个直接前驱元素，有且仅有一个直接后继元素。

(3) 数据元素之间的先后顺序为“*一对一*”的关系。

2. 线性表的定义

数据元素之间具有的逻辑关系为线性关系的数据元素集合称为**线性表**，数据元素的个数n为线性表的长度，长度为0的线性表称为空表。

线性表的特点：

（1）同一性　（2）有穷性　（3）有序性

元素的前后位置关系

常见错误：线性表的有序性是指其中的数据元素是按值由小到大或由大到小的顺序排列的。

2.1.2 线性表的基本操作

1. 创建一个新的线性表。
2. 求线性表的长度。
3. 检索线性表中第i个数据元素。 (1≤i≤n)
4. 根据数据元素的某数据项(通常称为关键字)的值求该数据元素在线性表中的位置（查找）。
5. 在线性表的第i个位置上存入一个新的数据元素。
6. 在线性表的第i个位置上插入一个新的数据元素。
7. 删除线性表中第i个数据元素。
8. 对线性表中的数据元素按照某一个数据项的值的大小做升序或者降序排序。

9. **销毁**一个线性表。
10. **复制**一个线性表。
11. 按照一定的原则，将两个或两个以上的线性表**合并**成为一个线性表。
12. 按照一定的原则，将一个线性表**分解**为两个或两个以上的线性表。

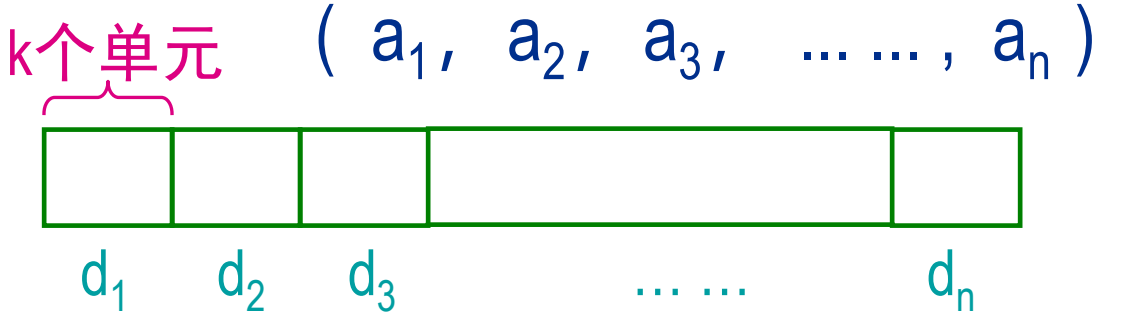
……

- initList(nodeType *list, int length);　　//创建一个空表
- destroyList (nodeType *list, int length); //销毁一个表
- printList(nodeType *list, int length);　　//输出一个表
- getNode (nodeType *list, int pos);　　//获取表中指定位置元素
- searchNode(nodeType *list, Type node); //在表中查找某一元素
- insertNode(nodeType *list, int pos, nodeType node); //在表中指定位置插入一个结点
- deleteNode(nodeType *list, int pos);　　//在表中指定位置删除一个结点

2.2 线性表的顺序存储结构

2.2.1 构造原理

用一组地址连续的存储单元依次存储线性表的数据元素，数据元素之间的逻辑关系通过数据元素的存储位置直接反映。



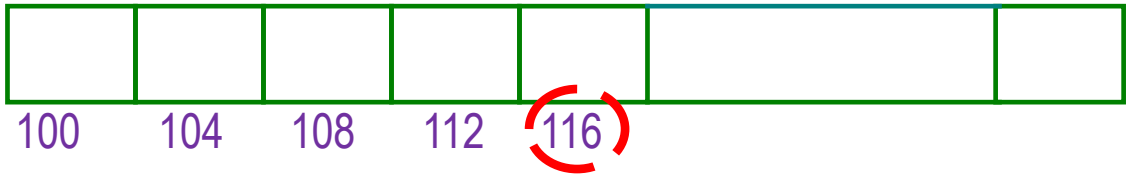
所谓一个元素的*地址*是指该元素占用的k个(连续的)存储单元的第一个单元的地址。

若假设每个数据元素占用k个存储单元，并且已知第一个元素的存储位置$LOC(a_1)$，则有

$$LOC(a_i) = LOC(a_1)+(i-1)\times k$$

这个如此简单的公式，说明了顺序存储的线性表具有一个什么样的巨大优势呢？

例：$LOC(a_1)=100$　$k=4$　求$LOC(a_5)=?$



$$LOC(a_5) = 100+(5-1)\times 4=116$$

$(a_1, a_2, a_3, \ldots\ldots, a_{n-1}, a_n)$

```
#define    MaxSize   100
ElemType  A[MaxSize];
int  n; \\表的长度
```

2.2.2 基本算法

1. 查找：确定元素item在长度为n的顺序表list中的位置

$(a_1, a_2, a_3, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_{n-1}, a_n)$

顺序查找法

```
int searchElem(ElemType list[ ], int n, ElemType item){
    int i;
    for(i=0;i<n;i++)
        if(list[i]==item)
            return  i ;        /* 查找成功，返回在表中位置 */
    return −1;                 /* 查找失败，返回信息−1 */
}
```

时间复杂度O(n)

如何在有序顺序表中查找元素?

- **折半查找算法（The binary search）：**

在有序数据集中查找指定数据项（或数据项插入位置）最常用及效率较高的算法是折半查找算法。当数据集较大时，折半查找平均效率高。

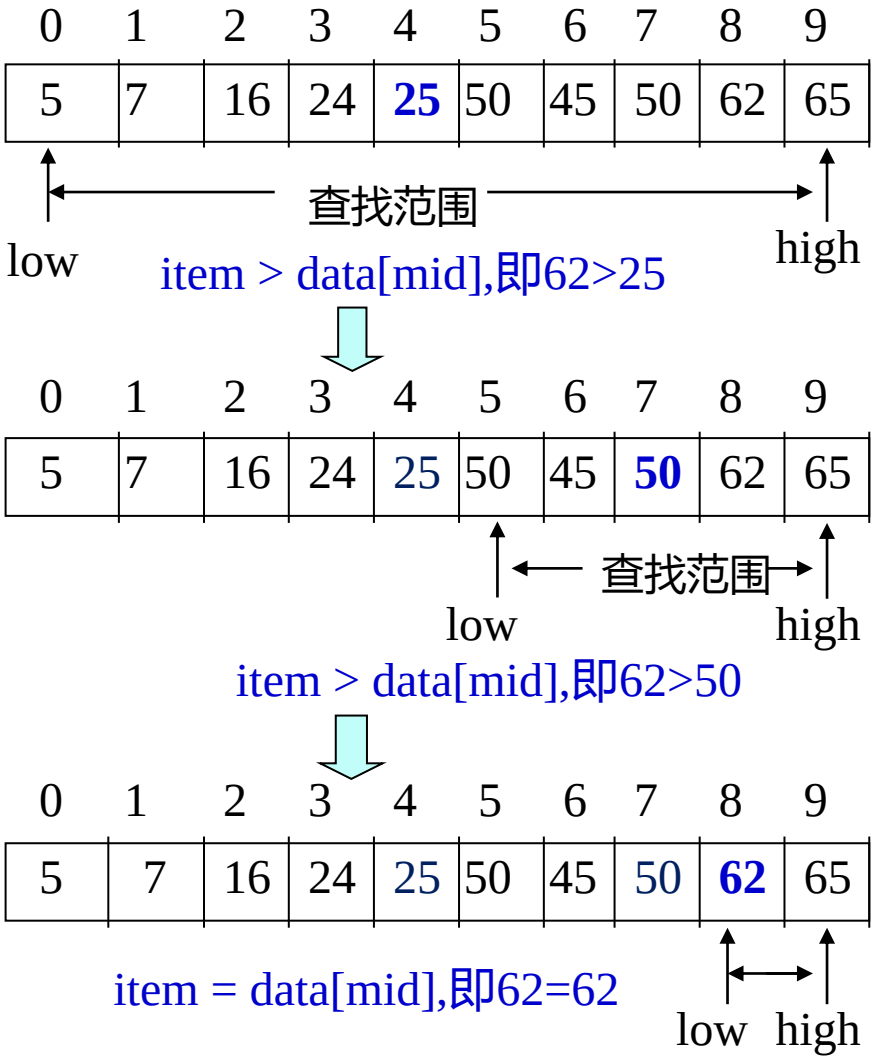
顺序表的折半查找

【算法的核心思想】假设数据集按由小到大排列，如果数据集中没有元素再可进行查找，则查找失败；否则将目标数据项与有序数据集的中间元素相比较：

- 如果指定数据项等于该中间元素，则查找成功结束
- 如果指定数据项小于该中间元素，则将数据集的前半部分指定为要查找的数据集，继续查找
- 如果指定数据项大于该中间元素，则将数据集的后半部分指定为要查找的数据集，继续查找

示例：折半查找过程(C语言)

例：在下面有序数据集中查找数据项 **62**



item = 62(查找顶)
low = 0(查找范围开始)
high = 9(查找范围结束)
mid = (low+high)/2=4(查找范围中间)

(注：取整，数值的下界)

low = mid+1=5
high = 9
mid = (low+high)/2=7

low = mid+1=8
high = 9
mid = (low+high)/2=8

折半查找算法（续）

在有序整数数组中查找给定元素的折半查找算法如下：

```
int bsearch(int item, int array[ ], int len) {
    int low=0, high=len-1, mid;
    while(low <= high){
        mid = (high + low) / 2;
        if(( item < array[mid])
            high = mid – 1;
        else if ( item > array[mid])
            low = mid + 1;
        else
            return (mid);
    }
    return -1;
}
```

练习

在序列(13,38,49,65,76,97)中采用折半查找法查找元素47需要进行 3 次元素间的比较。

13,38,**49**,65,76,97

13,38,49,65,76,97

13,**38**,49,65,76,97

(注意中间元素的准确含义)

2. 插入：在长度为n的顺序表list的第i个位置上插入一个新的数据元素item

在线性表的第i–1个与第i个数据元素之间插入一个由符号item表示的数据元素，使得长度为n的线性表

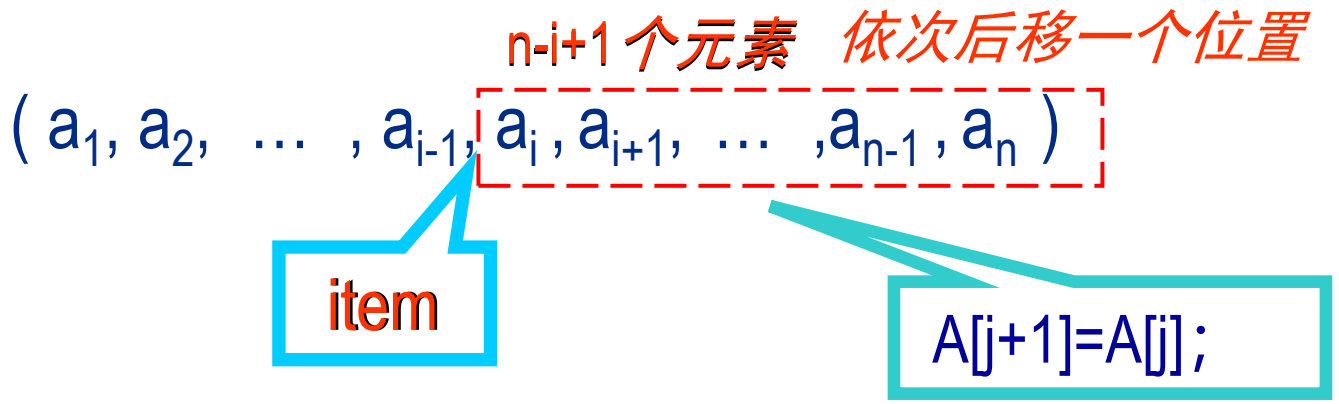
$$(a_1, a_2, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_{n-1}, a_n)$$

n个数据元素

转换成长度为n+1的线性表

$$(a_1, a_2, \ldots, a_{i-1}, \text{item}, a_i, \ldots, a_{n-1}, a_n)$$

n+1个数据元素



正常情况下需要做的工作：

(1) 将第i个元素至第n个元素依次后移一个位置；
(2) 将被插入元素插入表的第i个位置；
(3) 修改表的长度(表长增1)。(n++;)

插入操作需要考虑的异常情况：

(1) 是否表满？ n=MaxSize ?
(2) 插入位置是否合适？ (正常位置：0≤i≤n)

插入算法：在长度为N的顺序表list的第i个位置上插入一个新的数据元素item

```
/* 令N为全局变量*/                    注意，该函数参数i指C里下标为i的元素
int  insertElem(ElemType list[], int i, ElemType item ){
      int k;
      if (N==MaxSize || i<0 || i>N)
          return -1;                  /* 插入失败 */
      for( k=N-1; k>=i; k-- )
          list[k+1]=list[k];          /* 元素依次后移一个位置 */
      list[i]=item;                   /* 将item插入表的第i个位置 */
      N++;                            /* 线性表的长度加1 */
      return 1;                       /* 插入成功 */
}
```

测试空间满否

测试插入位置是否合理

约定：若插入成功，算法返回1；否则返回-1。

该算法的时间复杂度是： **O(n)**

衡量插入和删除算法时间效率的另一个重要指标：元素移动次数的平均值

假设p_i为插入一个元素于线性表第i个位置的概率（概率相等），则在长度为n的线性表中插入一个元素需要移动其他的元素的平均次数为：

$$T_{is} = \sum_{i=1}^{n} p_i (n-i+1) = \sum_{i=1}^{n} (n-i+1)/(n+1) = n/2$$

$a_1, a_2, a_3, \ldots, a_i, a_{i+1}, a_{i+2}, \ldots, a_n$ □

N+1个插入位置

已知长度为n 的非空线性表list采用顺序存储结构,并且数据元素按值的大小非递减排列(有序),写一算法,在该线性表中插入一个数据元素item,使得线性表仍然保持按值非递减排列。

思路1：从尾部开始，比较和移动“同时”进行……

item

$a_1, a_2, a_3, \ldots, a_i, a_{i+1}, a_{i+2}, \ldots, a_n$

依次后移一个位置

哪种方法效率更高?

例

已知长度为n 的非空线性表list采用顺序存储结构，并且数据元素按值的大小非递减排列(有序)，写一算法，在该线性表中插入一个数据元素item，使得线性表仍然保持按值非递减排列。

思路2：首先找到插入位置

$a_i \le a_{i+1}$ $1 \le i \le n-1$

item

$a_1, a_2, a_3, \ldots, a_i, a_{i+1}, a_{i+2}, \ldots, a_n$

依次后移一个位置

插入

```
for(j=n-1;j>=i; j--)
    list[j+1]=list[j];
list[i]=item;
n++;
```

算法

$a_1, a_2, a_3, \ldots, a_i, a_{i+1}, a_{i+2}, \ldots, a_n$, item ← 特殊情况

正确处理否？

```
/*假设 N是表长度，是一个全局变量 */
int insertElem(ElemType list[ ], ElemType item){
      int i,j;

      if(N == MAXSIZE)  return -1;
      for(i=0; i<N&&item>=list[i]; i++)    /* 寻找item的合适位置 */
          ;
      for(j=N-1; j>=i; j--)
          list[j+1]=list[j];

      list[i]=item;      /* 将item插入表中 */
      N++;
      return 1;
}
```

确定插入位置

表长加1

3. 删除：删除长度为n的顺序表list的某个数据元素

把线性表的第i个数据元素从线性表中去掉，使得长度为n 的线性表

$$(a_1, a_2, \ \cdots \ , a_{i-1}, a_i , a_{i+1}, \ \cdots \ , a_{n-1} , a_n)$$

n个数据元素

转换成长度为 n-1 的线性表

$$(a_1, a_2, \ \cdots \ , a_{i-1}, \ a_{i+1}, \cdots \ , a_{n-1}, a_n)$$

n-1个数据元素


```
/* 假设N是表的长度（元素个数），为一个全局变量 */
int  deleteElem( ElemType list[ ],  int i )
{
      int k;
      if(N==0|| i<0 || i>N-1 )
          retutn -1;                  /* 删除失败 */
      for( k=i+1; k<N; k++ )
           list[k-1]=list[k];         /* 元素依次前移一个位置 */
      N--;                            /* 线性表的长度减1 */
      return 1;                       /* 删除成功 */
}
```

测试表空和位置合适与否

该算法的时间复杂度是：**O(n)**

若p_i为删除线性表中第i个数据元素的概率（设概率相等），在长度为n的线性表中删除第i个数据元素需要移动其他的元素的平均次数为

$$T_{ds}=\sum_{i=1}^{n}p_i(n-i)=\sum_{i=1}^{n}(n-i)/n=(n-1)/2$$

2.2.3 顺序存储结构的特点

1. 优点

$LOC(a_i)=LOC(a_1)+(i-1)\times k$

(1) 构造原理简单、直观，易理解。

(2) 元素的存储地址可以通过一个简单的解析式计算出来。是一种顺序存储结构,读取速度快。

(3) 由于只需存放数据元素本身的信息，而无其他空间开销，相对链式存储结构而言，存储空间开销小

(4) 对于有序表，可使用折半查找等快速查找算法。

2. 缺点

对于**动态表**（即需要频繁插入和删除操作的表）往往由于问题规模不知，如果采用顺序结构的话，需要事先分配很大的空间，造成空间浪费或空间不足。

(1) 存储分配需要事先进行。

(2) 需要一块地址连续的存储空间。

(3) 基本操作(如插入、删除)的时间效率较低。

需要频繁的移动数据

O(n)

注：现代编译优化技术可提高插入、删除操作的效率

2.3 线性表的排序（简单排序）

排序：将一个按值任意的数据元素序列转换为一个按值有序的数据元素序列。

简单排序方法：插入排序、选择排序、冒泡排序。

趟 —— 将具有n个数据元素（关键字）的序列转换为一个按照值的大小从小到大排列的序列通常要经过若干 **趟** (Pass)。

稳定性 —— 对于值相同的两个元素，排序前后的先后次序不变，则称该方法为**稳定性排序方法**，否则，称为**非稳定性排序方法**。

说明： 在所有可能的输入实例中，只要有一个实例使得该排序方法不满足稳定性要求，该方法就是非稳定的！

约定

1. 只针对一个数据元素(关键字)序列讨论排序方法。

2. 假设序列中具有n个数据元素(关键字)。

$$(k_1, k_2, k_3, \cdots, k_{n-1}, k_n)$$

存放于数组元素K[1],K[2], …, K[n]中

3. 排序结果按照数据元素(关键字)值的大小从小到大排列。

2.3.1 插入(insert)排序法

核心思想

第i趟排序将无序序列的第i+1个元素插入到一个大小为i且已经按值有序的子序列($k_{i-1,1}$, $k_{i-1,2}$, …, $k_{i-1,i}$)的 **合适位置**，得到一个大小为i+1且仍然按值有序的子序列($k_{i,1}$, $k_{i,2}$, …, $k_{i,i+1}$)。

（1，4，8，12，6，11，7，…）

（1，4，6，8，12，11，7，…）

算法关键步骤示例

49 38 97 76 65 13 27 50

… (若干趟后)

（有序子序列） （无序子序列）

38 49 76 97 65 13 27 50

先查找后插入 或 边比较边移动/交换

38 49 65 76 97 13 27 50

一趟结束

从第2个元素开始

i：1→n-1

j：i-1→0

初 始:	49	38	97	76	65	13	27	50
第1趟:	38	49	97	76	65	13	27	50
第2趟:	38	49	97	76	65	13	27	50
第3趟:	38	49	76	97	65	13	27	50
第4趟:	38	49	65	76	97	13	27	50
第5趟:	13	38	49	65	76	97	27	50
第6趟:	13	27	38	49	65	76	97	50
第7趟:	13	27	38	49	50	65	76	97

一个完整的过程


```
void insertSort(keytype k[ ],int n){
    int i, j;
    keytype  temp;
    for(i=1;i<n; i++){
        temp=k[i];
        for(j=i-1; j>=0 && temp<k[j]; j--)
            k[j+1]=k[j];
        k[j+1]=temp;
    }
}
```

n−1趟排序

思考

1. 排序的时间效率与什么直接有关?

答案

主要与排序过程中元素之间的比较次数直接有关。

2. 若原始序列为一个**按值递增**的序列，则排序过程中一共要经过多少次元素之间的比较?

答案

由于每一趟排序只需要经过一次元素之间的比较就可以找到被插入元素的合适位置，因此，整个n−1趟排序一共要经过n−1次元素之间的比较。

3. 若原始序列为一个**按值递减**的序列，则排序过程中一共要经过多少次元素之间的比较?

答案

由于第i趟排序需要经过i次元素之间的比较才能找到被插入元素的合适位置，因此，整个n−1趟排序一共要经过

$$\sum_{i=1}^{n-1} i = n(n-1)/2$$

次元素之间的比较。

若以最坏的情况考虑，则插入排序算法的时间复杂度为$O(n^2)$。

插入排序法是一种稳定排序方法。

插入排序算法分析

稳定性： 稳定

时间复杂度：
- 最佳情况：n-1次比较，0交换，O(n)
- 最差情况：比较和交换次数为$O(n^2)$
- 平均情况：$O(n^2)$

空间复杂度： O(1)

注：此处仅考虑辅助空间
实际上全量数据必须加载在内存

对于值相同的两个元素，排序前后的先后次序不变，则称该方法为**稳定性排序方法**，否则，称为**非稳定性排序方法**。

插入排序算法优化

用折半查找法
找到插入位置

13 38 49 65 76 97 27

依次右移一个位置

temp

27

13 27 38 49 65 76 97

可减少比较次数，
但不能减少移动次数

折半插入排序法

采用折半查找法确定插入位置

```
void  insertBSort(keytype k[ ], int n){
      int i, j, low, high, mid;
      keytype temp;
      for(i=1;i<n;i++ ){
          temp=k[i];
          low=0;
          high=i–1;
          while(low<=high){
                mid=(low+high)/2;
                if(temp<k[mid])
                      high=mid–1;
                else
                      low=mid+1;
          }
          for(j=i–1; j>=low; j––)
                k[j+1]=k[j];
          k[low]=temp;
      }
}
```

2.3.2 选择(select)排序法

核心思想

第i趟排序从未排序子序列(原始序列的后n−i+1个元素)中**选择**一个值最小的元素，将其置于子序列的最前面。

选择	35	97	38	27	65	13	80	75
交换	13	97	38	27	65	35	80	75
选择	13	97	38	27	65	35	80	75
交换	13	27	38	97	65	35	80	75

n−1趟

i：0→n-2

j：i→n-1

初 始:	49	97	38	76	65	13	27	50
第1趟:	13	97	38	76	65	49	27	50
第2趟:	13	27	38	76	65	49	97	50
第3趟:	13	27	38	76	65	49	97	50
第4趟:	13	27	38	49	65	76	97	50
第5趟:	13	27	38	49	50	76	97	65
第6趟:	13	27	38	49	50	65	97	76
第7趟:	13	27	38	49	50	65	76	97

一个完整的过程

```
void  selectSort(keytype k[ ],int n)
{     int i, j, d;
      keytype  temp;
      for(i=0;i<n−1;i++){
         /*寻找值最小的元素,并记录其位置*/
         d=i;
         for(j=i+1;j<n;j++)
              if(k[j]<k[d])
                   d=j;

         /* 交换最小值元素位置 */
         if(d!=i){
              temp=k[d] ;
              k[d]=k[i];
              k[i]=temp;
         }
      }
}
```

n-1趟排序

思考

若原始序列为一个按值**递增**的序列，则排序过程中一共要经过多少次元素之间的比较？

若原始序列为一个按值**递减**的序列，则排序过程中一共要经过多少次元素之间的比较？

答案： 无论原始序列为什么状态，第i趟排序都需要经过n−i次元素之间的比较，因此，整个排序过程中元素之间的比较次数为$\sum_{i=1}^{n-1}(n-i)=n(n-1)/2$。

结论

选择排序法的元素之间的比较次数与原始序列中元素的分布状态无关。

时间复杂度为$O(n^2)$。

不稳定排序方法

选择排序算法分析

稳定性： 不稳定

时间代价：

- 比较次数：$O(n^2)$
- 交换次数：$n-1$
- 总时间代价：$O(n^2)$

空间代价： $O(1)$

注：此处仅考虑辅助空间

选择最小（最大）元素的效率比较低

2.3.3 冒泡(bubble)排序法

值大的元素往后“沉”
值小的元素向前“浮”

核心思想

第i趟排序对序列的**前n−i+1**个元素从第一个元素开始依次作如下操作：相邻的两个元素比较大小，若前者大于后者，则两个元素交换位置，否则不交换位置。

效果

该n−i+1个元素中最大值元素移到该n−i+1个元素的最后。

i：n-1→1
j：0→i-1

或 i：0→n-2
j：n-1→i+1

初始	49	97	38	13	27	50	76	65
第1趟	49	38	13	27	50	76	65	97
第2趟	38	13	27	49	50	65	76	97
第3趟	13	27	38	49	50	65	76	97
第4趟	13	27	38	49	50	65	76	97

排序总趟数可以小于n-1

设置一标志

$$flag = \begin{cases} 0 & \text{某趟排序过程中无元素交换位置的动作} \\ 1 & \text{某趟排序过程中有元素交换位置的动作} \end{cases}$$

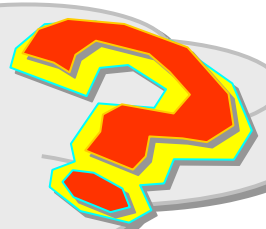
每一趟趟排序前置flag为0；若排序过程中出现元素交换动作，则置flag为1。


```
void  bubbleSort(keytype k[ ],int n){
      int i, j, flag=1;
      keytype temp;
      for(i=n-1; i>0 && flag==1; i--){
            flag=0;                          /* 每趟排序前标志flag置0 */
            for(j=0;j<i;j++)
                  if(k[j]>k[j+1]){
                        temp=k[j];
                        k[j]=k[j+1];
                        k[j+1]=temp;  /* 交换两个元素的位置 */
                        flag=1;               /* 标志flag置1 */
                  }
      }
}
```

泡排序法的排序趟数与原始序列中数据元素的排列有关，因此，排序的趟数为一个范围，即[1..n−1]。

什么情况下只要排序一趟

什么情况下要排序n−1趟



结论

泡排序方法比较适合于参加排序的序列的原始状态基本有序的情况

$O(n^2)$

泡排序法是稳定性排序方法。

（改进）冒泡排序算法分析

稳定性： 稳定

时间代价：

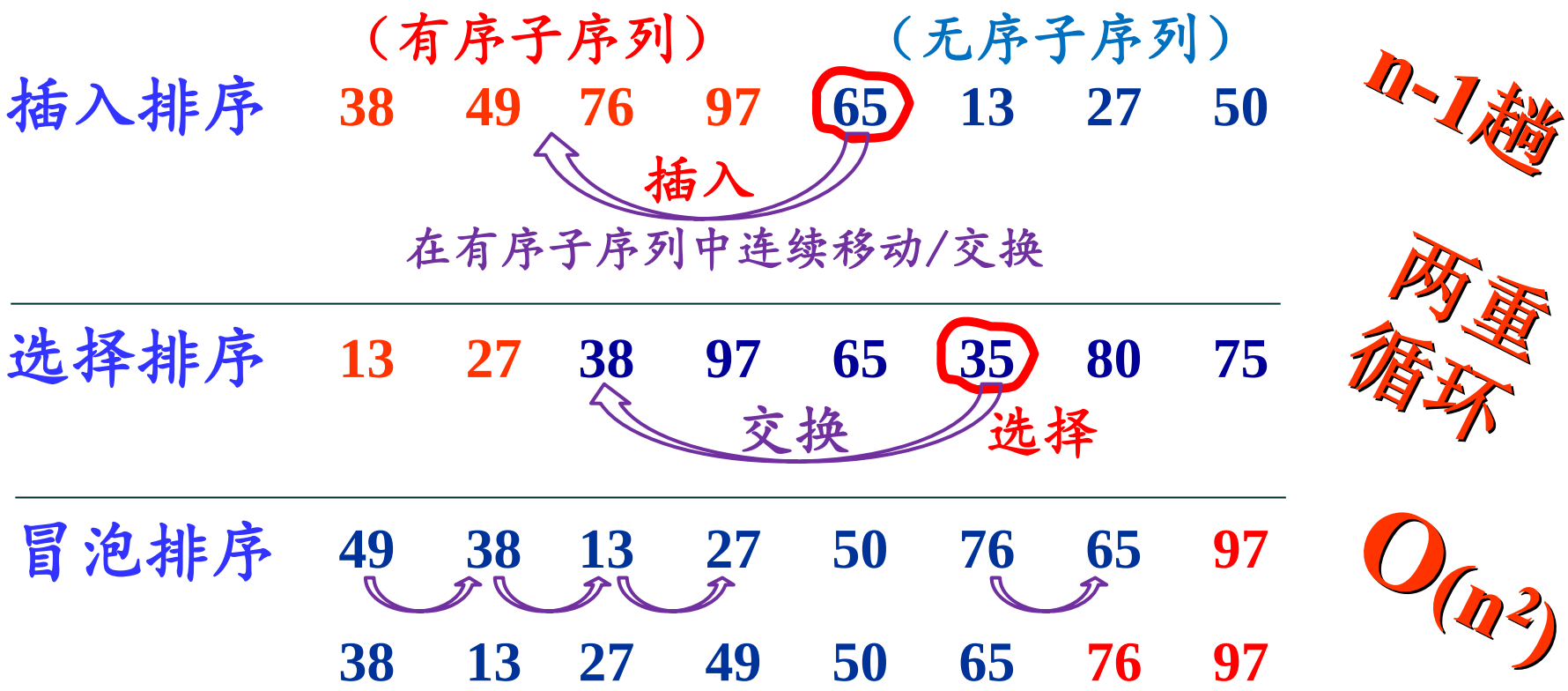
- 时间代价：
  - 最小时间代价为O(n)，最佳情况下只运行第一轮循环
  - 一般情况下为$O(n^2)$

空间代价： O(1)

注：此处仅考虑辅助空间

三种简单排序方法的直观比较

共性：每趟都从无序子序列中找一个元素，将其加入有序子序列



冒泡排序部分融合了“选择”和“插入”排序法：

- 在无序序列中做了“选择”
- 在无序序列中做了“移动”

【2023春季期末考试题】对序列进行从小到大排序，若序列的原始状态为1, 2, 3, 4, 5, 10, 6, 7, 8, 9，要想使得排序过程中元素的比较次数最少，应该采用_____ 方法。

思考：组合key排序

假定要对某班的考试成绩做排序，首先按照分数降序排序，如果分数相同则按学号升序排序。

定义结构体封装学生及成绩信息

```
struct Student {
    int id;
    char Name[MAXWORD];
    int score;
};
```

可以采用任何排序方法

比较两个元素的时候，要额外考虑分数（主key）相等，学号（次key）不同的情况

```
void  bubbleSort(struct Student stu[ ], int n){
    int i, j, flag=1;
    struct Student temp;
    for(i=n-1; i>0 && flag==1; i--){
        flag=0;                    /* 每趟排序前标志flag置0 */
        for(j=0;j<i;j++)
            if( (stu[j].score<stu[j+1].score) ||
                (stu[j].score==stu[j+1].score && stu[j].id>stu[j+1].id)){
                temp=k[j];
                k[j]=k[j+1];
                k[j+1]=temp;  /* 交换两个元素的位置 */
                flag=1;             /* 标志flag置1 */
            }
    }
}
```

回顾：C语言关键字词频统计

- **问题：编写程序统计文件中每个C语言关键字的出现次数。**

定义一个结构说明用以表示关键字与其出现次数：

```
struct Key {
    char *keyword;
    int count;
};
```

使用一个**有序**的**结构数组**来存放关键字表及其出现次数

```
struct Key keytab[ ] = {
    “auto”,   0,
    “break”, 0,
    “case”,   0,
    …
    “while”,  0
};
```

回顾：C语言关键字词频

- 算法主要步骤：

While (仍有新单词读入)

If(在关键字表中查找并找到输入的单词)

相应关键字次数加1;

输出关键字及出现次数;

设函数
char getWord(char word[],int lim)
从标准输入中读入一个长度不超过lim-1的单词，并返回单词类型。
Q：为何不用scanf的%s来读？如，
while(scanf("%s", word) > 0)…

设函数
void printKey(struct Key tab[ ], int n)
输出关键字及出现次数。

设函数
struct Key *binarySearch(char *word, struct Key tab[ ], int n)
在关键字表tab中查找单词word是否存在。如果找到，则返回其出现位置。n为关键字表的长度（关键字个数）。

问题2.1：文本词频统计 – 顺序表

- 问题：编写程序统计一个文件中每个单词的出现次数（词频统计），并按字典序输出每个单词及出现次数。
- 算法分析：

1. 首先构造一个空的有序（字典序）单词表；

2. 每次从文件中读入一个单词；

3. 在单词表中（折半）查找该单词，若找到，则单词次数加1，否则将该单词插入到单词表中相应位置，并设置出现次数为1；

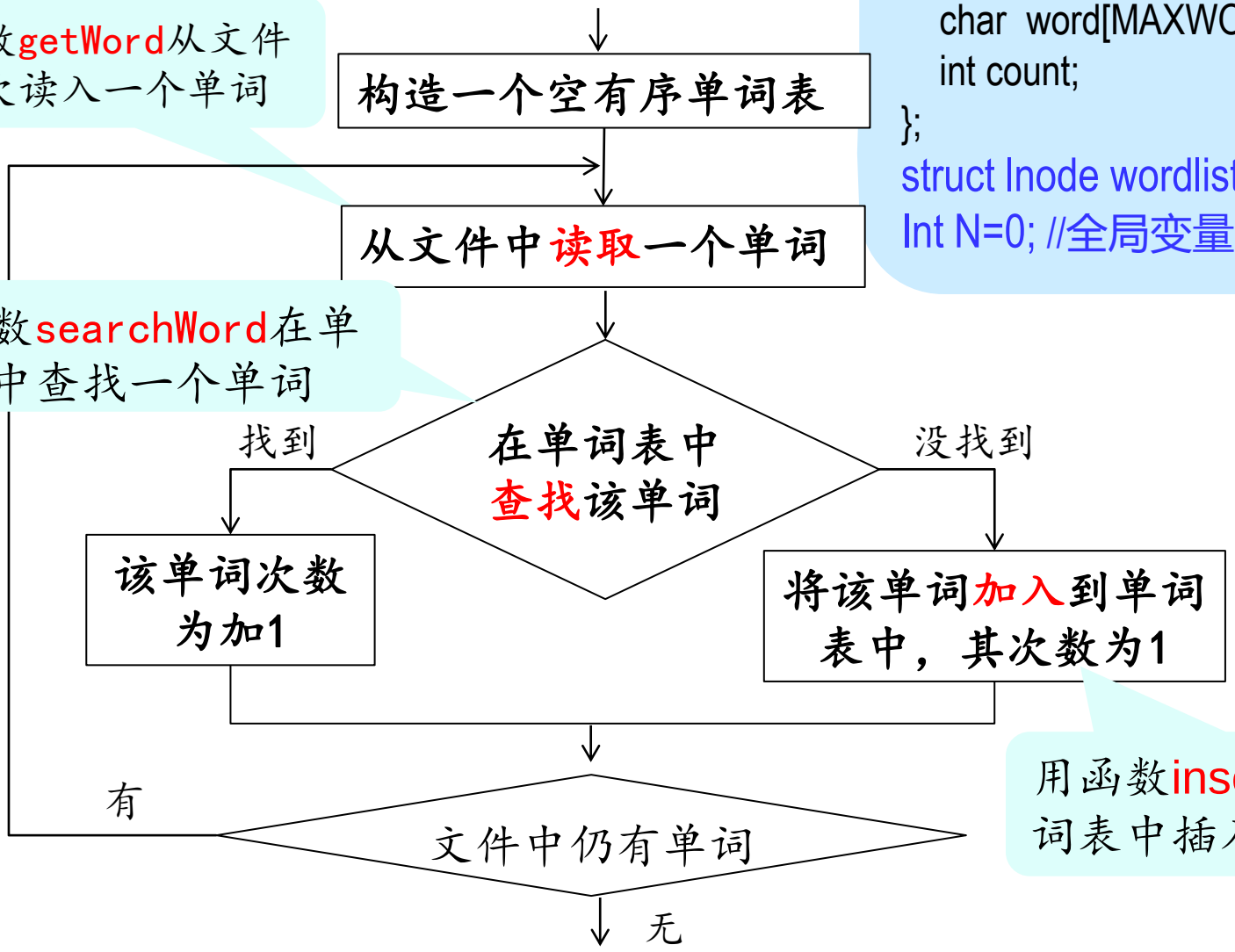
4. 重复步骤2，3，直到文件结束。

问题2.1：文本词频统计

```
/*用数组构造一个顺序表，
表中单词按字典序组织*/
struct Inode {
    char  word[MAXWORD];//字符数组
    int count;
};
struct Inode wordlist[MAXSIZE];
Int N=0; //全局变量，初始为空表
```

用函数getWord从文件中每次读入一个单词

用函数searchWord在单词表中查找一个单词



用函数insertWord在单词表中插入一个单词

问题2.1：文本词频统计 – 参考代码

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAXWORD  32
#define MAXSIZE  1024

struct lnode {
    char word[MAXWORD];
    int count;
};

struct lnode wordlist[MAXSIZE];  /*单词表*/
int N=0; //单词表中单词的实际个数

int getWord(FILE *bfp, char *w);
int searchWord(struct lnode list[], char *w);
int insertWord(struct lnode list[ ],  int pos, char *w);
```

编程知识：外部（全局）变量

- 外部变量（全局变量，global variable）：在函数外面定义的变量。
  - 初始值：外部变量有隐含初值0。
  - 作用域（scope）为整个程序，即可在程序的所有函数中使用。
  - 生存期（life cycle）：外部变量（存储空间）在程序执行过程中始终存在（静态存储分配）。

外部变量说明（extern）

- 在同一文件中，C程序可以分别放在多个文件上，每个文件可作为一个编译单位分别编译。外部变量只需在某个文件上定义一次，其它文件若要引用此变量时，应用extern加以说明。（外部变量定义时不必加extern关键字。
- 若前面的函数要引用后面定义的外部（在函数之外）变量时，也应在函数里加以extern说明。

```
/* t1.c */
int  N;
main(){
      …
      N =  …
      …
}
```

```
/* t2.c */
extern int  N;
fun(){
      …
      N =  …
      …
}
```

```
extern int  N;
main(){
      …
      N =  …
      …
}
int N=0;
void fun(){
 …
```

外部变量说明（extern）

- 使用外部变量的原因：
  - 解决函数单独编译的协调；
  - 与变量初始化有关；
  - 外部变量的值是永久的；
  - 解决数据共享；
- 外部变量的副作用：
  - 使用外部变量的函数独立性差，通常不能单独使用在其他的程序中。而且，如果多个函数都使用到某个外部变量，一旦出现差错，就很难发现问题是由哪个函数引起的。

风格建议：在程序中应尽量少用或不用外部变量。

问题2.1：文本词频统计 – 参考代码

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAXWORD  32
#define MAXSIZE  1024

struct lnode {
    char word[MAXWORD];
    int count;
};

struct lnode wordlist[MAXSIZE];  /*单词表*/
int N=0; //单词表中单词的实际个数

int getWord(FILE *bfp, char *w);
int searchWord(struct lnode list[], char *w);
int insertWord(struct lnode list[ ],  int pos, char *w
```

```
int main(){
    int i;
    char filename[MAXWORD], word[MAXWORD];
    FILE *bfp;
    scanf("%s", filename);
    if((bfp = fopen(bname, "r")) == NULL){
        fprintf(stderr, "%s  can't open!\n",filename);
        return -1;
     }
    while(getWord(bfp,word) != EOF)
        if(searchWord(wordlist, word) == -1) {
            fprintf(stderr, "Wordlist is full!\n");
            fclose(bfp);  return -1;
        }
    for(i=0; i<= N-1; i++)
        printf("%s  %d\n", wordlist[i].word, wordlist[i].count);
    fclose(bfp);
    return 0;
}
```

问题2.1：词频统计 – 代码实现*

```
/*从文件中读入一个单词（仅由字母组成的串），并转换成小写字母*/
int getWord(FILE *fp, char *w){
    int c;

   while(!isalpha(c=fgetc(fp)))
      if(c == EOF) return c;
      else continue;
   do {
       *w++ = tolower(c);
   } while(isalpha(c=fgetc(fp)));
   *w='\0';
  return 1;
}
```

问题2.1：词频统计 – 代码实现*

```
/*在表中相应位置插入一个单词，同时置次数为1*/
int insertWord(struct lnode list[ ], int pos, char *w){
    int i;

    if  (N == MAXSIZE) return -1;

    for(i=N−1; i>=pos; i−−){
        strcpy(list[i+1].word, list[i].word);
        list[i+1].count = list[i].count;
    }
    strcpy(list[pos].word, w);
    list[pos].count = 1;
    N++;
    return 1;
}
```

在本程序中：
查找算法复杂度为$O(\log_2 n)$
插入算法复杂度为 O(n)

```
/*在表中查找一单词，若找到，则次数加1；
否则将该单词插入到有序表中相应位置，同
时次数置1*/
int searchWord(struct lnode list[ ], char *w){
    int low=0, high=N-1, mid=0;
    while(low <= high){
        mid = (high + low) / 2;
        if(strcmp( w, list[mid].word)<0)
            high = mid -1;
        else if(strcmp( w, list[mid].word)>0)
            low = mid + 1;
        else {
            list[mid].count++;
            return 0;
        }
    }
    return insertWord(list, low, w);
}
```

问题2.1：词频统计

- **请用无序顺序表（即单词表中的单词按其读入顺序排放，新加入的单词总是放在表的末尾）来重新实现问题2.1，并比较一下两个程序的实际性能。**

问题2.1：词频统计

- **由于顺序表（数组）的大小需要事先确定，用顺序表作为单词表会有什么问题?**

由于事先不知道被统计的文件大小（可能是本很厚的书），单词表的大小如何定：
1）太小，对于大文件会造成单词表溢出；
2）太大，对一般文件处理会造成很大的空间浪费。

- **词频统计需要在单词表中频繁的查找和插入单词，有序顺序表结构的单词表有什么特点?**

1）单词查找效率很高（可用折半查算法，一次查找算法复杂度为$O(\log_2 n)$）；
2）单词表中单词需要频繁移动（对于一般的英文材料来说，插入操作较多，一次插入算法的复杂度为$O(n)$；

- **无序顺序表结构构造的单词表又有什么特点?**

1）单词查找效率低（顺序查找算法，一次查找算法复杂度为$O(n)$）；
2）单词不需要移动（新单词总是放在表尾），但需要对最终的总表进行排序，简单排序算法的复杂度为$O(N^2)$；