

数据结构与算法

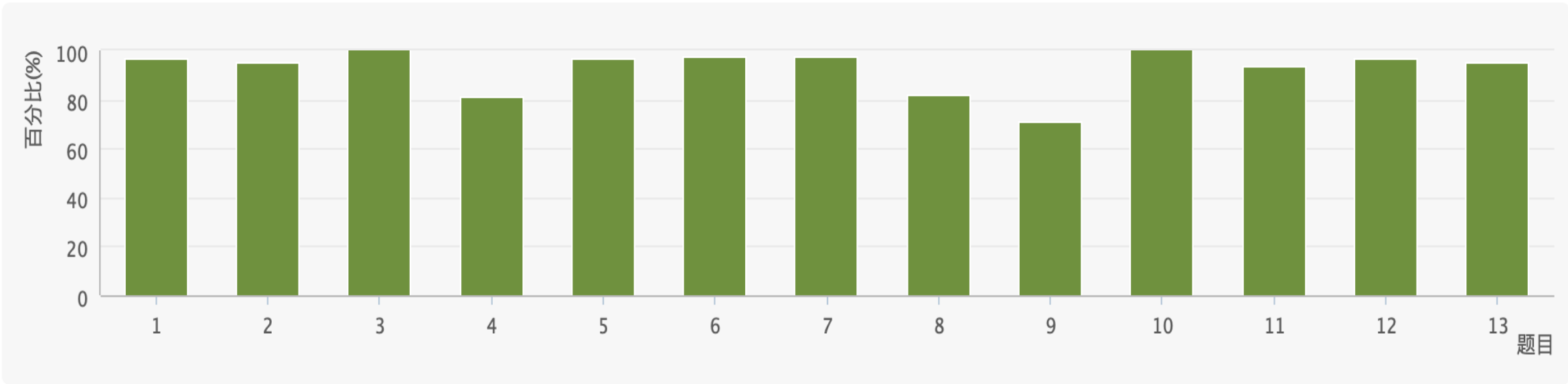


人工智能与大数据学院



统计详情

返回



8 相互之间存在一种或多种特定关系的数据元素的集合称为 ()

正确答案:

第一空: 

正确: 51 人


半对: 0 人

错误: 11 人

正确率:  82%

4 算法的设计要求包括 ()

- A、 正确性
- B、 可读性
- C、 有穷性
- D、 健壮性
- E、 高效率

正确答案: 

正确: 50 人

半对: 0 人

错误: 12 人

正确率:  81%

9 线性表中的数据元素满足（ ）结构

正确答案：

第一空： 线性

正确： 44 人 半对： 0 人 错误： 18 人 正确率： 71%

答题记录
逻辑
线性结构
一对一
顺序存储结构
顺序结构
顺序
线性结构
顺序
一对一
一对一
限行
一对一
线性结构
相同
一对一
相同
链式
逻辑

第2章 线性结构

本章的学习目标：

- ① 理解线性表的基本概念
- ② 掌握线性表的基本操作
- ③ 掌握线性表顺序存储结构的实现方法
- ④ 掌握线性表链式存储结构的实现方法

第2章 线性结构

数据元素之间的关系系统称为逻辑关系或者逻辑结构。

集合结构、线性结构、树形结构、图形结构

线性表

其中线性表是最基本和最常用的数据结构，在线性结构中，数据元素之间是一对一的逻辑关系

第2章 线性结构

本节课的主要内容：

- 1、线性表的定义
- 2、线性表的基本操作
- 3、线性表的顺序存储

第2章 线性结构——线性表

线性表的定义

线性表是 n ($n \geq 0$) 个**具有相同类型**的数据元素的**有限序列**，逻辑结构记为

$$L = (a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$$

线性表的
名称

表头元素

组成线性表的
数据元素， i 是
数据编号

表尾元素，
 n 是表长，
 $n=0$ 时，表示空表

第2章 线性结构——线性表

线性表的定义

线性表

$$L = (a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$$

a_i 的前驱

a_i 的后继

其中， a_{i-1} 称为 a_i 的直接前驱， a_{i+1} 称为 a_i 的直接后继。

注意：表头 a_1 没有直接前驱，表尾 a_n 没有直接后继。线性表的其他数据元素有且仅有一个直接前驱，有且仅有一个后继。
后续在不混淆的情况下将省略“直接”二字

第2章 线性结构——线性表

线性表的定义（示例）

例如，3名学生组成的线性表 $L=(stu1, stu2, stu3)$ 。

在c语言中数据元素可以用结构体表示：

```
struct student{  
    int number;  
    int score;  
    char name[10];  
}stu1,stu2,stu3;
```

第2章 线性结构——线性表

线性表的定义

- 数据元素在线性表中的位置取决于它自身的序号
- 数据元素在线性表中是有序的
- 数据元素之间存在一对一的关系
- 线性表的逻辑结构是线性结构

第2章 线性结构——线性表

线性表的基本操作

- ① 初始化线性表
- ② 销毁线性表
- ③ 线性表清空
- ④ 线性表判空
- ⑤ 求线性表的长度
- ⑥ 取表中的元素
- ⑦ 按值查找
- ⑧ 插入操作
- ⑨ 删除操作
- ⑩ 求前驱操作
- ⑪ 求后继操作

- ◆ 数据结构的操作是定义在逻辑结构层次上的，操作的具体实现是建立在存储结构上
- ◆ 其他操作可以通过基本操作来实现

线性表的基本操作

以下基本操作仅需要一个输入参数，均为线性表

基本操作	线性表初始化	销毁线性表	线性表清空	线性表判空	计算线性表的长度
名称	List_Init	List_Destroy	List_Clear	List_Empty	List_Size
初始条件	线性表不存在	线性表存在			
操作结果	成功返回success，构造一个空的线性表，否则返回fatal	释放线性表所占空间	清除线性表中所有元素，线性表变空	如果线性表为空，返回true，否则返回false	返回线性表中所含数据元素的个数

线性表的基本操作

以下6种基本操作除了需要线性表作为输入参数外，还需要其他参数

基本操作	名称	初始条件	其他输入参数	操作结果
取表中元素	Status List_Retrieve(ListPtr L, int pos, ElemType *elem)	线性表L存在	数据元素位置pos	如果 $1 \leq pos \leq \text{List_Size}(L)$ ，返回success，同时将L中的第pos个元素的值放入elem中，否则返回range_error
按值查找	Status List_Locate(ListPtr L, ElemType *elem, int *pos)		数据元素elem	若找到，返回success，同时将L中首次出现的值为elem的那个元素的序号记入pos，否则，返回fail
插入操作	Status List_Insert(ListPtr L, int pos, ElemType elem)		数据元素位置pos，数据元素elem	判断pos是否合理，表示线性表中还有未用的存储空间，则在第pos个位置上插入一个值为elem的新元素，插入后的表长+1，返回success，否则返回range_error

线性表的基本操作

基本操作	名称	初始条件	其他输入参数	操作结果
删除操作	Status List_Remove(ListPtr L, int pos)	线性表L存在	数据元素位置pos	如果pos合理，删除后使序号为pos+1,pos+2,...,n的元素的序号变为pos,pos+1,...,n-1。表长-1，返回success，否则返回range_error
求前驱操作	Status List_Prior(ListPtr L, int pos, ElemType * elem)		数据元素位置pos	如果第pos个数据元素的直接前驱存在，将其存入elem中，返回success，否则返回fail
求后继操作	Status List_next(ListPtr L, int pos, ElemType * elem)		数据元素位置pos	如果第pos个数据元素的直接后继存在，将其存入elem中，返回success，否则返回fail

线性表的基本操作

在线性表中，在11种基本操作的基础上，可以实现其他一些复杂的操作。下面以两个例子进行说明

【例1】用线性表La和Lb分别表示集合A和集合B，现要求一个新的集合 $A = A \cup B$

线性表的基本操作

【例1】用线性表La和Lb分别表示集合A和集合B，现要求一个新的集合 $A = A \cup B$

分析：将线性表Lb中的数据元素逐个取出，判断其是否在线性表La中，如果不在，就将其插入线性表La

涉及到的线性表的基本操作包括：

- ① 取表中元素
- ② 按值查找
- ③ 插入操作
- ④ 求线性表的长度

第2章 线性表

【例1】 用线性表La和Lb分别表示集合A和集合B，现要求一个新的集合 $A = A \cup B$

【线性表合并算法】

```
1  Status List_Union(ListPtr La, ListPtr Lb){
2      ElemType elem;                                /* elem用于存放从Lb中取出的元素 */
3      Status status;                                /* 状态代码 */
4      int i, j, len = List_Size(Lb);                /* len用于存放从Lb的元素个数 */
5      for(i = 1; i <= len; i++){
6          List_Retrieve(Lb, i, &elem);              /* 取出Lb中第i个数据元素 */
7          status = List_Locate(La, elem, &j);        /* 判断它是否在La中 */
8          if(status != success){                    /* 如果不在 */
9              status = List_Insert(La, 1, elem);    /* 将其插入到La的第1个位置 */
10             if(status != success)break;           /* 插入失败则退出 */
11         }
12     }
13     return status;
14 }
15
```

线性表的基本操作

【思考】用线性表La和Lb分别表示集合A和集合B，现要求一个新的集合 $A = A \cap B$

分析：将线性表La中的数据元素逐个取出，判断其是否在线性表Lb中，如果不在，则将该元素从线性表La中删除

涉及到的线性表的基本操作包括：

- ① 取表中元素
- ② 按值查找
- ③ 删除操作
- ④ 求线性表的长度

线性表的基本操作

【例2】 已知线性表La和Lb中元素分别按非递减顺序排列，现要求将它们合并成一个新的线性表Lc，并使Lc中的元素也按照非递减顺序排列

1, 2, 3, 4, 5, 6

9, 8, 7, 6, 5, 4

1, 2, 2, 3, 4, 4

6, 6, 5, 4, 3, 3

线性表的基本操作

【例2】 已知线性表La和Lb中元素分别按非递减顺序排列，现要求将它们合并成一个新的线性表Lc，并使Lc中的元素也按照非递减顺序排列

La : (1, 3, 3, 6, 7)

Lb : (2, 2, 5, 8)



Lc ?

线性表的基本操作

【例2】 已知线性表La和Lb中元素分别按非递减顺序排列，现要求将它们合并成一个新的线性表Lc，并使得Lc中的元素也按照非递减顺序排列

分析：线性表Lc初始为空。依次扫描La和Lb中的元素，比较当前元素的值，将较小值的元素插入Lc的最后一个元素之后。如此反复，直到一个线性表扫描完毕，然后将未完的那个线性表中余下的元素逐个插入到Lc的表尾。

涉及到的线性表的基本操作包括：

- ① 线性表初始化
- ② 取表中元素
- ③ 线性表插入元素
- ④ 求线性表的长度

```

1 Status List_Merge(ListPtr La, ListPtr Lb, ListPtr Lc){
2     ElemType elem1, elem2;          /* 用于暂存数据元素 */
3     Status status;                  /* 状态代码 */
4     status = List_Init(Lc);          /* 初始化线性表Lc */
5     if(status != success){return status;} /* 初始化失败则退出 */
6     int i=1, j=1, k=1;              /* i,j,k分别用于指示La, Lb, Lc中当前正处理的元素 */
7     int n = List_Size(La), m = List_Size(Lb); /* n,m分别存放La, Lb中的元素个数 */
8     while(i <= n && j <= m) {        /* 两个表都还未处理完 */
9         List_Retrieve(La, i, &elem1);
10        List_Retrieve(Lb, j, &elem2);
11        if(elem1 < elem2){
12            status = List_Insert(Lc, k, elem1);
13            i = i+1;                  /* 处理La中下一个数据元素 */
14        }else{
15            status = List_Insert(Lc, k, elem2);
16            j = j+1;                  /* 处理Lb中下一个数据元素 */
17        }
18        if(status != success) return status; /* 插入失败则退出 */
19        k = k+1;
20    }
21    while(i <= n){                    /* 表La都还未处理完 */
22        List_Retrieve(La, i, &elem1);
23        status = List_Insert(Lc, k, elem1);
24        if(status != success) return status; /* 插入失败则退出 */
25        i = i+1; k = k+1;
26    }
27    while(j <= m){                    /* 表Lb都还未处理完 */
28        List_Retrieve(Lb, j, &elem2);
29        status = List_Insert(Lc, k, elem2);
30        if(status != success) return status; /* 插入失败则退出 */
31        i = i+1; k = k+1;
32    }
33    return status;
34 }

```

有序表的合并算法

线性表的顺序存储

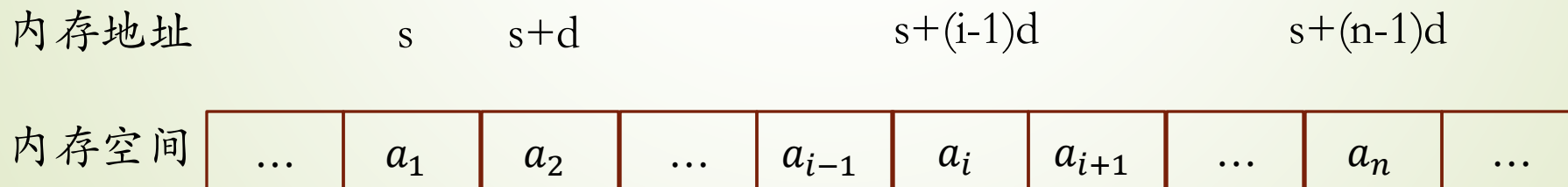
线性表的**顺序存储结构**是指用一组地址连续的内存单元依次存储线性表中的各个数据元素，数据元素之间的线性关系通过存储单元的相邻关系来体现，用这种存储形式存储的线性表称为**顺序表**

线性表的顺序存储

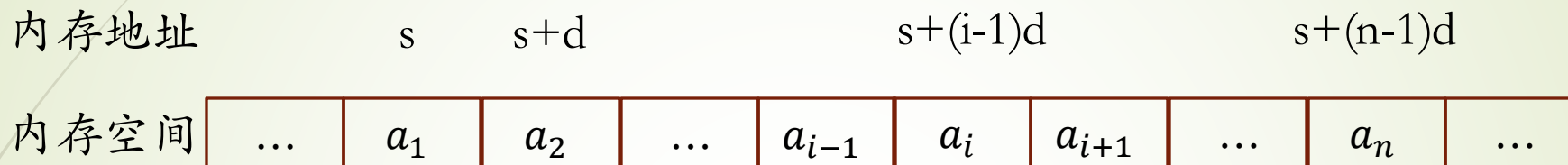
在顺序表的存储结构中，内存中物理地址相邻的结点一定具有线性表中逻辑相邻关系。

对线性表 $L = (a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$ ，由于所有数据元素 a_i 的类型相同，因此每个元素占用的存储空间的大小是相同的。

假设每个数据元素占 d 个存储单元，第一个数据元素的存放地址（基地址）是 s ，则该线性表的顺序存储结构如下图所示：



线性表的顺序存储



由于数据元素 a_i 和 a_{i+1} 存放在两个相邻的存储单元中，记 $loc(a_i)$ 为 a_i 的存储地址，则第 i 个数据元素的地址可用如下公式计算

$$loc(a_i) = loc(a_1) + (i - 1) \times d$$

每个数据元素地址的计算都需要1次减法、1次乘法和1次加法，需要的计算时间是相同的，这表明顺序表具有按数据元素的序号随机存取的特点，因此**顺序表也称为线性表的随机存储结构**。

线性表的顺序存储

顺序存储的优缺点：

优点：

- 随机存取元素容易实现，根据定位公式容易确定表中每个元素的存储位置
- 简单、直观

缺点：

- 插入和删除结点困难
- 扩展不灵活

线性表的顺序存储

线性表的顺序存储结构描述：

```
#define LIST_INIT_SIZE 100
```

```
typedef struct SqList{  
    int *elem;           // 线性表的基地址  
    int length;         // 线性表的当前长度  
    int listsize;        // 顺序表当前分配的存储空间的大小  
}SqList, *ListPtr;
```

```
typedef ListPtr sqlistptr;
```

线性表的顺序存储

线性表的初始化:

```
Status List_Init(sqlistptr L){  
    Status s=success;  
    L->listsize = LIST_INIT_SIZE;  
    L->length = 0;  
    L->elem = (int *)malloc(sizeof(int)*L->listsize);  
    if(L->elem == NULL)  
        s = fatal;  
    return s;  
}
```

算法时间复杂度: $O(1)$

线性表的顺序存储

顺序表的查找操作

查找有时也称定位，查找的要求通常有两种：按位置查找和按值查找。按位置查找是给定数据元素的位置，在线性表中找出相应的数据元素；按值查找是给定数据元素的值（或值的一部分，比如数据元素的某个数据项），在线性表中查找相应数据元素的位置（或其它信息）

线性表的顺序存储

顺序表的查找操作——按位置查找

```
Status List_Retrieve(ListPtr L, int pos, int *elem){  
    Status status = range_error;  
    int len = L->len;  
    if(1 <= pos && pos <= len){  
        *elem = L->elem[pos];  
        status = success;  
    }  
    return status;  
}
```

算法时间复杂度: $O(1)$

线性表的顺序存储

顺序表的查找操作——按值查找

```
Status List_Locate(ListPtr L, int elem, int *pos){
    Status status = range_error;
    int len = L->len;
    int i = 1;
    while (i <= len && L->elem[i] != elem) {
        i++;
    }
    if(i <= len){
        *pos = i;
        status = success;
    }
    return status;
}
```

算法时间复杂度: ?

线性表的顺序存储

(1) 顺序表的按位置查找算法

该算法的算法复杂度为 $O(1)$ 。

这也是顺序表是随机存取结构，查找速度快的原因。

(2) 顺序表的按值查找算法

List_Locate的时间耗费主要在于比较数据元素是否相等，显然，比较的次数和给定的数据元素有关。最好的情况是第1个元素就是要找的元素，只需比较1次。如果该数据元素不在此线性表中，则需要比较 n 次，最坏的时间复杂度为 $O(n)$

线性表的顺序存储

顺序表的插入操作

顺序表的插入操作是指在表的某个位置插入一个新的数据元素。
设原来的顺序表为

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

在第*i*个位置成功插入数据元素*x*后，原顺序表变为表长为*n+1*的顺序表

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

由于是顺序表，结点之间的逻辑顺序和物理顺序一致，为保证插入数据元素后的相对关系，必须先将位置为*i, i+1, …, n*的元素依次后移一个位置

线性表的顺序存储

顺序表的插入操作

具体描述如下：

对给定的顺序表L、给定的数据元素elem和给定的位置pos，若 $1 \leq \text{pos} \leq \text{List_Size}(L) + 1$ ，在顺序表L的第pos个位置上插入一个值为elem的新数据元素，原序号为pos, pos+1, ..., n的数据元素的序号变为pos+1, pos+2, ..., n+1，插入后表长 = 原表长 + 1，返回success；否则，若没有空间则返回overflow，若插入位置不正确则返回range_error

线性表的顺序存储

顺序表的插入操作

具体实现步骤：

- ① 检查插入位置是否合法，如果合法则继续，否则退出。
- ② 判断表是否已占满，因为可能存在所分配存储空间全部被使用的情况，此时也不能实现插入。
- ③ 若前面检查通过，则数据元素依次向后移动一个位置；为避免覆盖原数据，应从最后一个向前依次移动
- ④ 新数据元素放到恰当位置
- ⑤ 表长加1

顺序表的插入操作（算法分析）

顺序表中插入操作的时间主要消耗在数据的移动上。

假设顺序表的长度为 n ，插入的位置为 i ，在第 i 个位置上插入数据元素 $elem$ ，从 a_i 到 a_n 都要向后移动一个位置，共需要移动 $n-i+1$ 个数据元素。因此，最好的情况是在最后一个元素后面插入新元素，此时不需要移动数据，时间复杂度为 $O(1)$ ，最坏的情况是在第一个位置插入新元素，此时需要移动 n 次，时间复杂度为 $O(n)$

线性表的顺序存储

顺序表的删除操作

顺序表的删除操作是指删除顺序表中某个位置的数据元素。设原来的顺序表为

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

删除第*i*个位置的数据元素 a_i ，原顺序表变为表长为*n*-1的顺序表

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

由于是顺序表，结点之间的逻辑顺序和物理顺序一致，为保证删除操作后数据元素之间的相对关系，必须将 a_i 后面的数据元素依次前移一个位置

线性表的顺序存储

顺序表的删除操作

具体描述如下：

对给定的顺序表L和给定的位置pos，若 $1 \leq \text{pos} \leq \text{List_Size}(L)$ ，在顺序表L中删除序号为pos的数据元素，原序号为pos+1, pos+2, ..., n的数据元素的序号变为pos, pos+1, ..., n-1，删除后表长 = 原表长 - 1，返回success；否则返回range_error

线性表的顺序存储

顺序表的删除操作

具体实现步骤：

- ① 检查删除位置是否合法，如果合法则继续，否则退出。
- ② 若前面检查通过，删除指定位置的元素，其后的数据元素依次向前移动一个位置
- ③ 表长减1

顺序表的删除操作（算法实现）

```
1  Status List_Remove(ListPtr L, int pos){
2      Status status = range_error;    /* 初始化状态变量为range_error */
3      int len = L -> length;
4      int i;
5      if(1 <= pos && pos <= len){
6          for(i=pos; i < len; i++){
7              L -> elem[i] = L -> elem[i+1]; /* 数据元素前移一个位置 */
8              L -> length--;
9              status = success;
10         }
11         return status;
12     }
```

与顺序表的插入操作类似，顺序表的删除操作的时间主要消耗在移动顺序表中的数据元素上。假设顺序表的长度为 n ，最好的情况是删除最后一个元素，此时不需要移动数据，时间复杂度为 $O(1)$ ，最坏的情况是删除第一个数据元素，此时需要移动 $n-1$ 次，时间复杂度为 $O(n)$

线性表的顺序存储

顺序表的基本操作除查找、插入和删除以外，还包括顺序表的创建、销毁、清空、判空以及前驱后继的求取等。对应的具体的实现算法的时间复杂度均为 $O(1)$

顺序表的优点：

- ① 方法简单，易于实现
- ② 不用为表示结点间的逻辑关系而增加额外的存储开销
- ③ 可按序号随机存取顺序表中的数据元素

顺序表的缺点：

- ① 插入、删除操作时，平均移动大约表中一半的元素，因此对较长的顺序表而言效率低
- ② 需要预先分配存储空间，预先分配存储空间过大，可能会导致顺序表后部大量存储空间闲置，造成空间极度浪费；预先分配空间过小，又会造成数据溢出。

线性表的顺序存储

课后思考题：

使用C语言设计编写功能函数，实现顺序表插入数据结点，结点对应的结构体以及函数定义如下：

```
typedef struct{  
    int data[N]; //顺序表的数据结点  
    int len;      //顺序表最后一个结点的下标值  
}seqlist_t;  
int seqlist_insert(seqlist_t *l, int value);
```