

数据结构与算法

Data Structure and Algorithms

西安交通大学自动化系

蔡忠闽 周亚东

本课内容

1. 为什么要学习数据结构与算法
2. 什么是数据结构与算法
3. 如何学习数据结构与算法
4. 数据结构基础知识

1.4 数据结构基础知识

1.4.1 基本概念和术语

1.4.2 抽象数据类型的表示与实现

1.4.3 算法和算法分析

1.4.3.1 算法

1.4.3.2 算法设计的要求

1.4.3.3 算法效率的度量

1.4.3.4 算法的存储空间的需求

1.4.3 算法和算法分析

- 事先分析

假定每条语句的执行时间为单位时间。算法的时间复杂度是该算法中所有语句的执行频度之和。

例1：求两个方阵的乘积 $C=A*B$

```
for(i=0;i<n;i++)           //n+1
    for(j=0;j<n;j++)       //n(n+1)
    {
        C[i][j]=0;         //n*n
        for( k=0;k<n;k++)   //n*n*(n+1)
            C[i][j]+=A[i][k]*B[k][j]  //n*n*n
    }
```

$$f(n) = 2n^3 + 3n^2 + 2n + 1$$

1.4.3 算法和算法分析

- **频度**：语句可能重复执行的最大次数。
- **问题的规模**：算法求解问题的输入量，用整数 n 表示。
- **时间复杂度**：一个算法的时间复杂度是该算法的时间耗费，一般地说，时间复杂度是问题规模的函数—— $T(n)$ 。
- **渐近时间复杂度**：通常算法中基本操作重复执行的次数是问题规模 n 的某个函数 $f(n)$ 。若随着 n 的增大， $f(n)$ 是算法执行时间的上界，即

$$\lim_{n \rightarrow \infty} \frac{T(n)}{f(n)} = C, C \geq 0$$

则记作 $T(n) = O(f(n))$ ，称作算法的**渐近时间复杂度**，简称**时间复杂度**。

1.4.3 算法和算法分析

【定理】 若 $f(n) = a_m n^m + a_{m-1} n^{m-1} + \dots + a_1 n + a_0$ 是一个 m 次多项式，则 $f(n) = O(n^m)$ 。

证明：

$$\lim_{n \rightarrow \infty} \frac{f(n)}{n^m} = a_m \neq 0$$

所以有 $f(n) = O(n^m)$ 。

【推论】 时间复杂度由频度的最高阶项来决定。

1.4.3 算法和算法分析

2、一般情况下，对循环语句只考虑循环体语句的执行次数，而忽略该语句中步长加一、终值判别、循环转移等成份。因此，当有若干个循环语句时，**算法的时间复杂度是由嵌套层数最多的循环语句中最内层语句的频度所决定的。**

例2:

x=0;

for (i=1;i<=n;i++)

for (j=1;j<=i;j++)

for (k=1;k<=j;k++)

x++;

$$\sum_{i=1}^n \sum_{j=1}^i \sum_{k=1}^j 1 = \sum_{i=1}^n \sum_{j=1}^i j = \sum_{i=1}^n i(i+1)/2 =$$

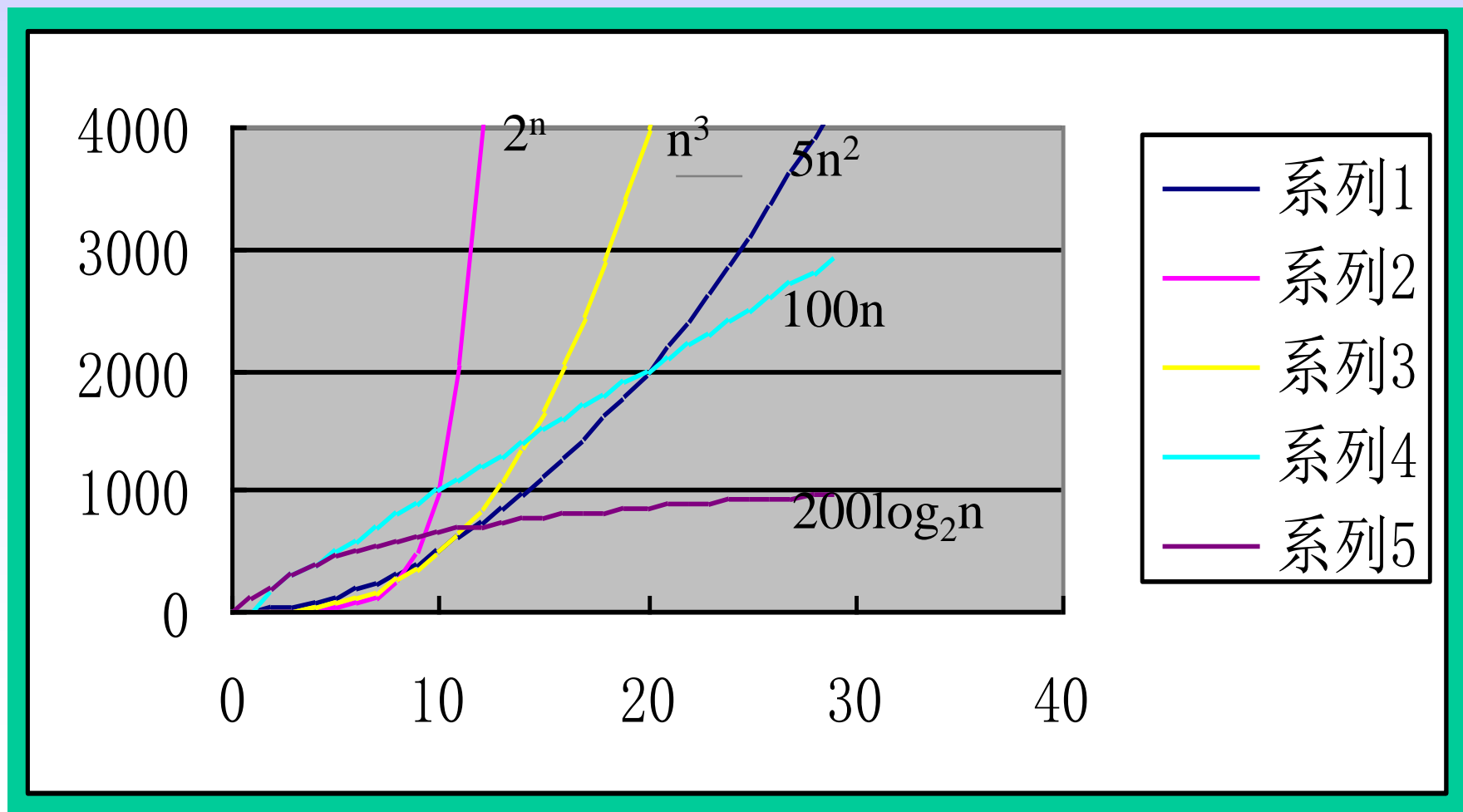
$$[n(n+1)(2n+1)/6 + n(n+1)/2]/2$$

$$T(n) = O(n^3/6 + \dots) = O(n^3)$$

1.4.3 算法和算法分析

- 以下是最常用的计算算法时间的多项式，其关系为：

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$$



1.4.3 算法和算法分析

1.4.3.4 算法的存储空间需求

空间复杂度: 算法所需存储空间的度量, 记作:

$$S(n) = O(f(n))$$

其中 n 为问题的规模(或大小)。

例: 数组逆序问题。

算法1: `for(i=0; i<n; i++) b[n-i-1]=a[i];`
`for(i=0; i<n; i++) a[i]=b[i];`

该算法需额外使用 n 个存储单元, $S(n) = O(n)$

算法2: `for(i=0; i<n/2; i++)`
`{ x=a[i]; a[i]=a[n-i-1]; a[n-i-1]=x; }`

该算法需额外使用1个存储单元, $S(n) = O(1)$

第二章 线性表

第二章 线性表

- 2.1 线性表的类型定义
- 2.2 线性表的顺序表示和实现
- 2.3 线性表的链式表示和实现
 - 2.3.1 线性链表
 - 2.3.2 循环链表
 - 2.3.3 双向链表
- 2.4 一元多项式的表示及相加

2.1 线性结构

在数据元素的非空有限集中：

- ◆ 存在唯一一个被称做“第一个”的数据元素；
- ◆ 存在唯一一个被称做“最后一个”的数据元素；
- ◆ 除第一个元素之外，每个元素都只有一个前驱；
- ◆ 除最后一个元素之外，每个元素都只有一个后继。

线性结构

数据元素之间存在着一个对一的关系



2.1 线性表的类型定义

1. 定义

一个线性表是n个数据元素的有限序列。

例1 英文字母表 (A,B,C,...Z)是一个线性表

例2 学生名单是一个线性表

数据元素(记录)

学号	姓名	性别	年龄
001	王小林	男	18
002	陈 红	女	19
003	刘建平	男	17
...

2.1 线性表的类型定义

2. 线性表的特点

- 元素个数 n : 即表的长度, $n=0$ 时为空表
- $1 < i < n$ 时: a_i 的前驱是 a_{i-1} ,
 a_i 的后继是 a_{i+1} , i 为的 a_i 位序
- a_1 无前驱
 a_n 无后继
- 元素同构,

$(\boxed{a_1}, \boxed{a_2, \dots, a_i, \dots}, \boxed{a_n})$

2.1 线性表的类型定义

3. 线性表的抽象数据类型

ADT List {

数据对象: $D = \{ a_i \mid a_i \in \text{ElemSet}, i=1,2,\dots,n, n \geq 0 \}$

数据关系: $R1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2,\dots,n \}$

基本操作:

(1) 结构初始化 **InitList(&L)**

操作结果: 构造一个空的线性表L。

(2) 销毁结构 **DestroyList(&L)**

初始条件: 线性表L已存在。

操作结果: 销毁线性表L。

2.1 线性表的类型定义

(3) 引用型操作

ListEmpty(L)

初始条件：线性表L已存在。

操作结果：若L为空表，则返回TRUE，否则FALSE。

ListLength(L)

初始条件：线性表L已存在。

操作结果：返回L中元素个数。

GetElem(L, i, &e)

初始条件：线性表L已存在， $1 \leq i \leq \text{LengthList}(L)$

操作结果：用e返回L中第i个元素的值。

2.1 线性表的类型定义

LocateElem(L, e, compare())

初始条件：线性表L已存在，compare()是元素判定函数。

操作结果：返回L中第1个与e满足关系compare()的元素的位序。若这样的元素不存在，则返回值为0。

ListTraverse(L, visit()) 线性表遍历

初始条件：线性表L已存在。

操作结果：依次对L的每个元素调用函数visit()。一旦visit()失败，则操作失败。

2.1 线性表的类型定义

(4) 修改型操作

ClearList(&L)

初始条件：线性表L已存在。

操作结果：将L重置为空表。

PutElem(&L, i, e)

初始条件：线性表L已存在，

$1 \leq i \leq \text{LengthList}(L)$

操作结果：将e的值赋值给L中第i个元素。

2.1 线性表的类型定义

`ListInsert(&L, i, e)`

初始条件: L已存在, $1 \leq i \leq \text{LengthList}(L) + 1$

操作结果: 在L的第i个元素之前插入新的元素e,
L的长度增1。

`ListDelete(&L, i, &e)`

初始条件: L已存在, $1 \leq i \leq \text{LengthList}(L)$

操作结果: 删除L的第i个元素, 并用e返回其值,
L的长度减1。

} `ADT List`

2.1 线性表的类型定义

4. 算法举例

假设有两个集合A和B分别用两个线性表LA和LB表示（即：线性表中的数据元素即为集合中的成员），现要求一个新的集合 $A = A \cup B$ 。

要求对线性表作如下操作：扩大线性表LA，将存在于线性表LB中而不存在于线性表LA中的数据元素插入到线性表LA中去。

(注： \cup —并集，属于A或者属于B)

2.2 线性表的顺序表示和实现

1、顺序表的定义

把线性表的结点按逻辑顺序依次存放在一组地址连续的存储单元里。

	1	2	3	4	5	6
data	25	34	57	16	48	09

用物理位置来表示逻辑结构。

$$\text{LOC}(a_{i+1}) = \text{LOC}(a_i) + l;$$

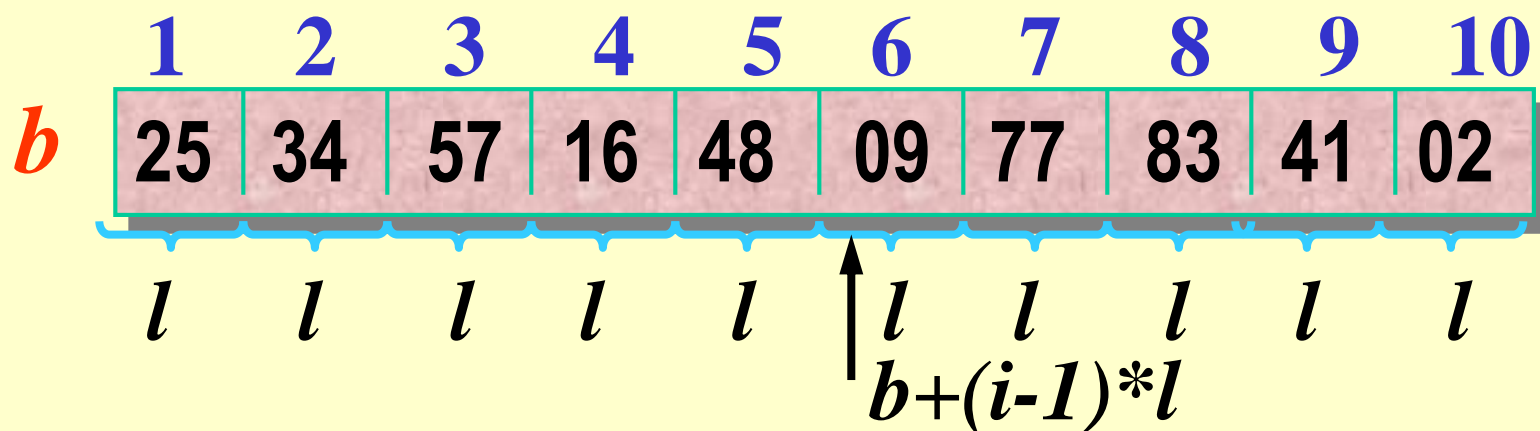
$$\text{LOC}(a_i) = \text{LOC}(a_1) + (i-1) * l$$

特点:

随机存取的存储结构。只要确定了存储线性表的起始位置，线性表中的任一数据元素可随机存取。

2.2 线性表的顺序表示和实现

$$\text{LOC}(a_i) = \text{LOC}(a_{i-1}) + l = b + (i-1) * l$$



2.2 线性表的顺序表示和实现

2、线性表的动态分配顺序存储结构（动态一维数组）

```
#define LIST_INIT_SIZE 100 //初始分配量
```

```
#define LISTINCREMENT 10 //分配增量
```

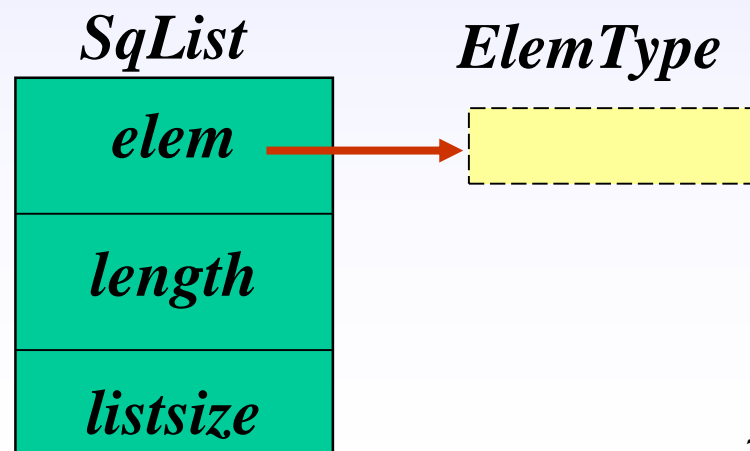
```
typedef struct{
```

```
    ElemType *elem; //存储空间基址
```

```
    int length; //当前长度
```

```
    int listsize; //当前存储容量
```

```
}SqList
```



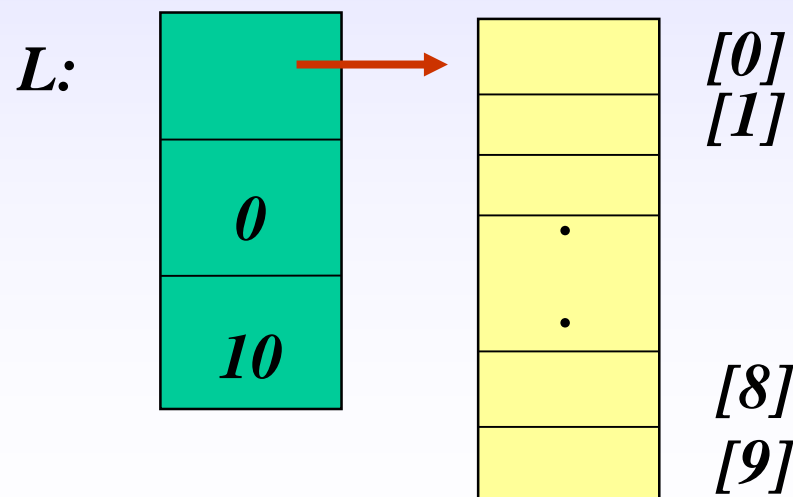
2.2 线性表的顺序表示和实现

3、顺序表的操作

(1) 初始化操作

算法思想：构造一个空表。

设置表的起始位置、表长及可用空间。



2.2 线性表的顺序表示和实现

算法:

```
Status InitList_Sq(SqList &L){ //构造一个空的线性表
    L.elem=(ElemType *)malloc (
        LIST_INIT_SIZE*sizeof(ElemType));
    If (!L.elem) exit(OVERFLOW); //存储分配失败
    L.length=0; //空表长度为0
    L.listsize= LIST_INIT_SIZE; //初始存储容量
    Return OK;
} //InitList_Sq
```

2.2 线性表的顺序表示和实现

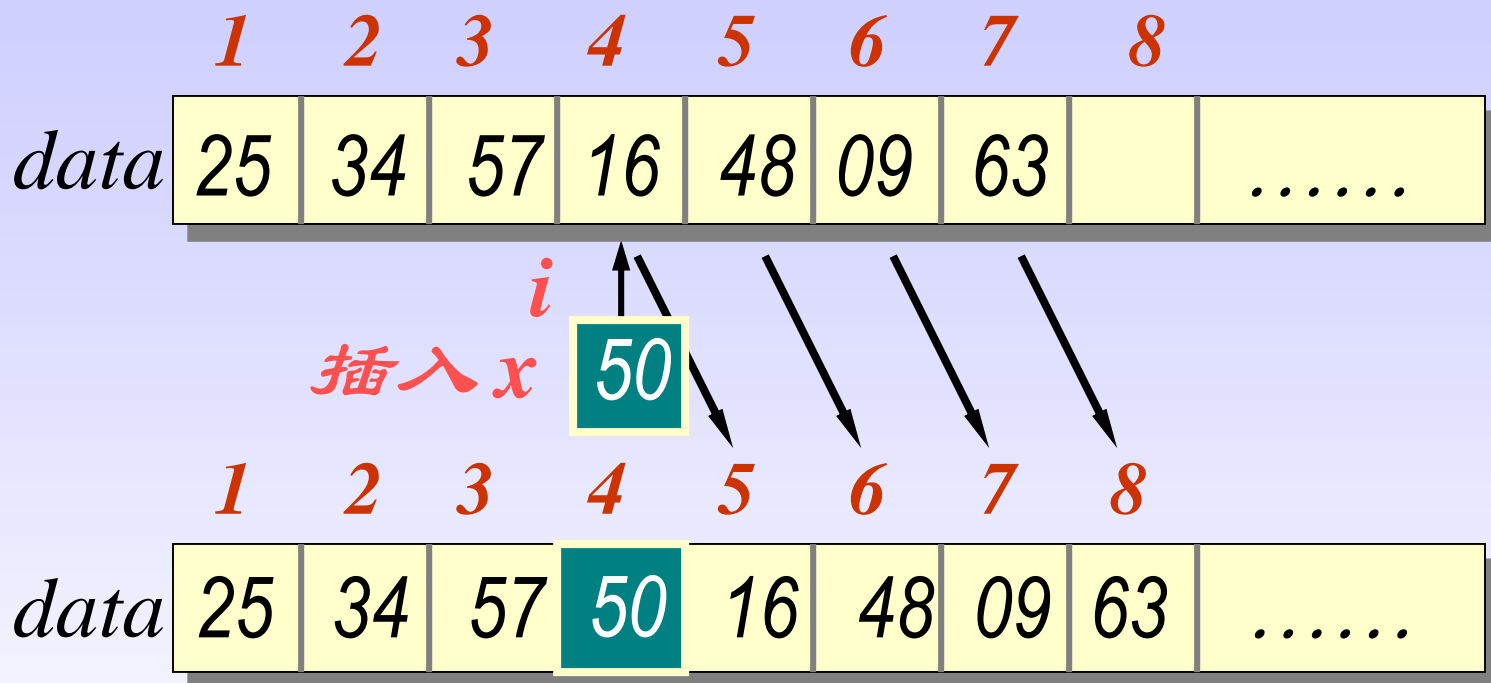
(2) 在线性表中指定位置前插入一个元素

插入第 i 个元素时，线性表的逻辑结构由 $(a_1, \dots, a_{i-1}, a_i, \dots, a_n)$ 改变为 $(a_1, \dots, a_{i-1}, e, a_i, \dots, a_n)$

算法思想：

- 1) 检查 i 值是否超出所允许的范围 ($1 \leq i \leq n+1$)，若超出，则进行“超出范围”错误处理；
- 2) 将线性表的第 i 个元素和它后面的所有元素均向后移动一个位置；
- 3) 将新元素写入到空出的第 i 个位置上；
- 4) 使线性表的长度增1。

2.2 线性表的顺序表示和实现



平均移动次数:

$$\begin{aligned}
 \text{AMN} &= \frac{1}{n+1} \sum_{i=1}^{n+1} (n-i+1) = \frac{1}{n+1} (n + \dots + 1 + 0) \\
 &= \frac{1}{(n+1)} \frac{n(n+1)}{2} = \frac{n}{2}
 \end{aligned}$$

2.2 线性表的顺序表示和实现

```
Status ListInsert_Sq(SqList &L, int i, ElemType e)
{ if (i < 1 || i > L.length+1) return ERROR; // 插入位置不合法
  if (L.length >= L.listsize) { // 当前存储空间已满, 增加分配
    newbase = (ElemType *)realloc
      (L.elem, (L.listsize+LISTINCREMENT)*sizeof (ElemType));
    if (!newbase) exit(OVERFLOW); // 存储分配失败
    L.elem = newbase; // 新基址
    L.listsize += LISTINCREMENT; // 增加存储容量
  }
  q = &(L.elem[i-1]); // q指示插入位置
  for (p = &(L.elem[L.length-1]); p>=q;--p)
    *(p+1)=*p;
  *q=e;
  ++ L.length;
  return OK; }
```


2.2 线性表的顺序表示和实现

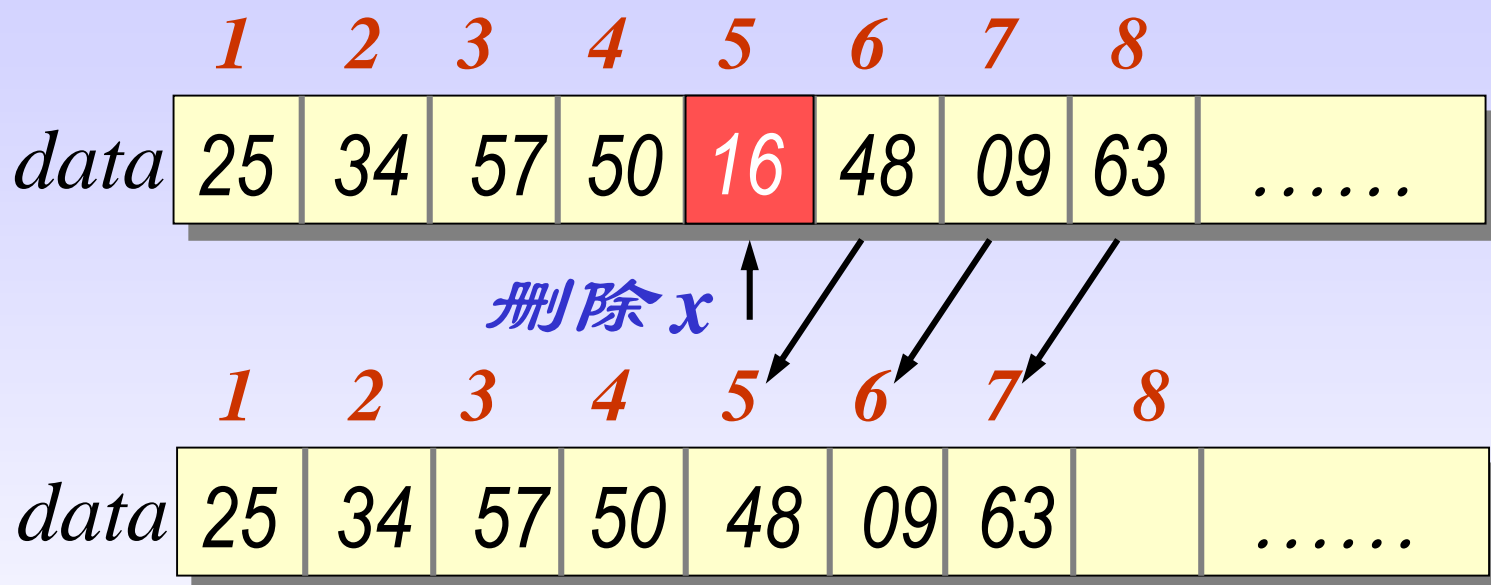
(3) 在线性表中删除第 i 个元素

删除元素时，线性表的逻辑结构由 $(a_1, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$ 改变为 $(a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n)$

算法思想：

- 1) 检查 i 值是否超出所允许的范围 ($1 \leq i \leq n$)，若超出，则进行“超出范围”错误处理；
- 2) 将线性表的第 i 个元素后面的所有元素均向前移动一个位置；
- 3) 使线性表的长度减1。

2.2 线性表的顺序表示和实现



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

2.2 线性表的顺序表示和实现

```
Status ListDelete_Sq(SqList &L, int i, ElemType &e)  
{ if ((i < 1) || (i > L.length)) return ERROR;  
  p = &(L.elem[i-1]);  
  e = *p;  
  q = L.elem+L.length-1;  
  for (++p; p <= q; ++p)  
    *(p-1) = *p;  
  - - L.length;  
  return OK;  
}
```

2.2 线性表的顺序表示和实现

```
Status ListDelete_Sq(SqList &L, int i, ElemType &e)
{ if ((i < 1) || (i > L.length)) return ERROR; // 删除位置不合法
  p = &(L.elem[i-1]); // p为被删除元素的位置
  e = *p; // 被删除元素的值赋给e
  q = L.elem+L.length-1; // 表尾元素的地址
  for (++p; p <= q; ++p)
    *(p-1) = *p; // 被删除元素之后的元素左移
  -- L.length;
  return OK;
}
```

2.2 线性表的顺序表示和实现

– 顺序存储结构的优缺点

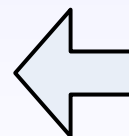
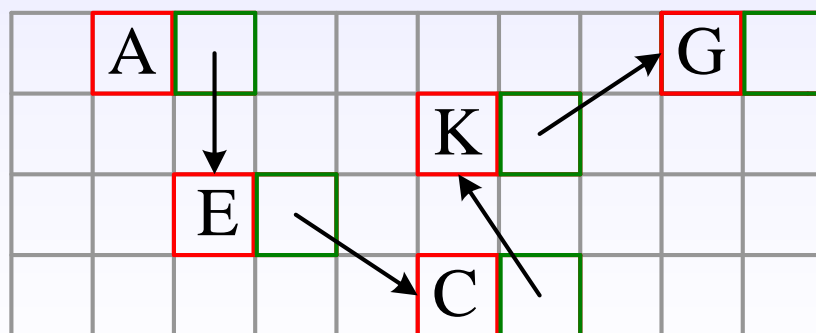
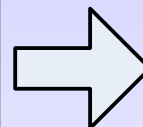
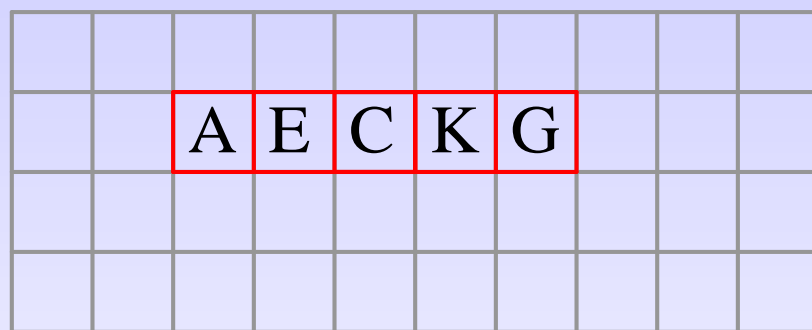
- 优点

- 逻辑相邻，物理相邻
- 可随机（直接）存取任一元素
- 存储空间使用紧凑

- 缺点

- 插入、删除操作需要移动大量的元素
- 预先分配空间需按最大空间分配，利用不充分

2.3 线性表的链式表示与实现



2.3 线性表的链式表示与实现

特点:

- 用一组**任意的存储单元**存储线性表的数据元素
- 利用**指针**实现了用不相邻的存储单元存放逻辑上相邻的元素
- 每个数据元素 a_i ，除存储本身信息外，还需存储其直接后继的信息
- 结点

—**数据域**: 元素本身信息

数据域	指针域
-----	-----

—**指针域**: 指示直接后继的存储位置

2.3 线性表的链式表示与实现

例 线性表 (ZHAO,QIAN,SUN,LI,ZHOU,WU,ZHENG,WANG)



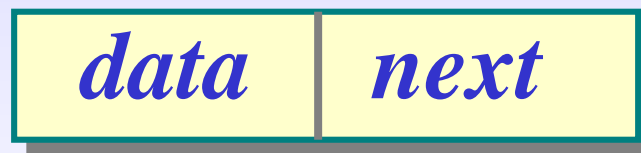
2.3 线性表的链式表示与实现

2.3.1 单链表（或线性链表）

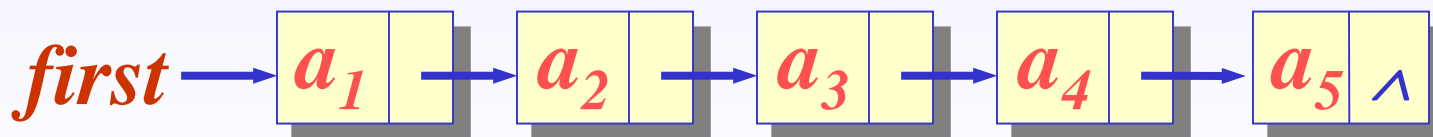
1. 定义：结点中只含一个指针域的链表

2. 特征：

- ◆ 每个元素(表项)由结点(Node)构成。



- ◆ 线性结构



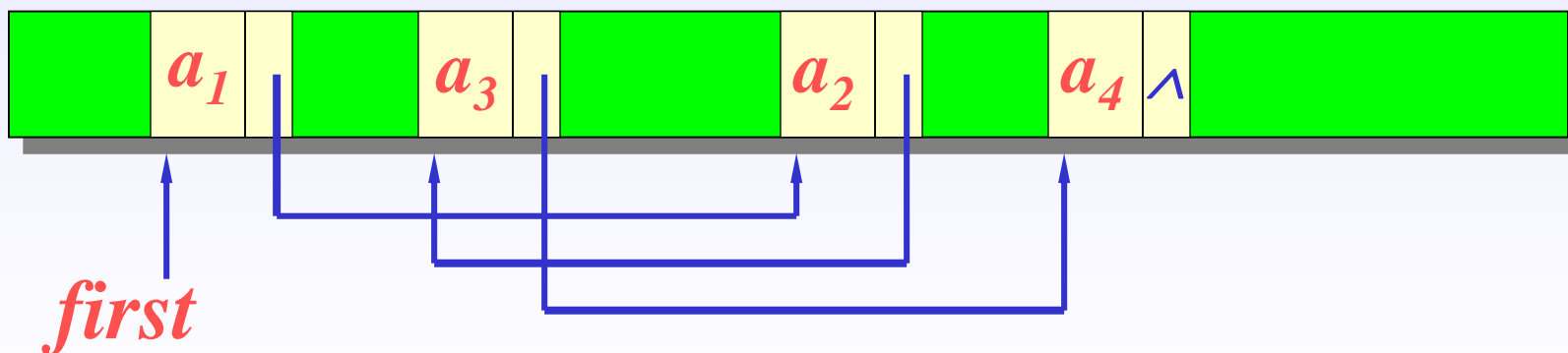
- ◆ 结点可以不连续存储
- ◆ 表可扩充

2.3 线性表的链式表示与实现

3. 单链表的存储映像



(a) 可利用存储空间



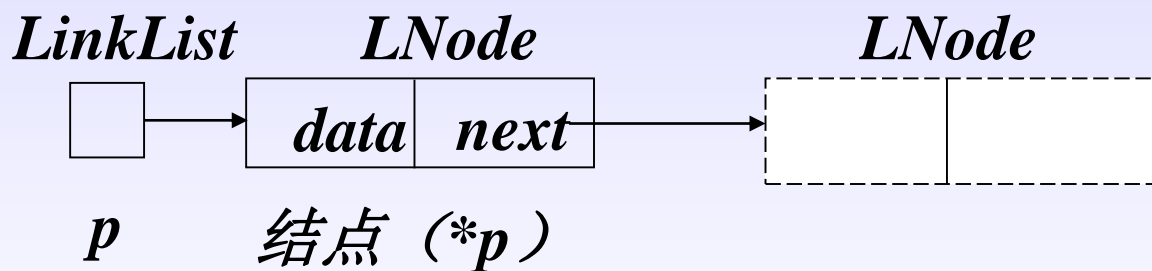
(b) 经过一段运行后的单链表结构

2.3 线性表的链式表示与实现

4. 单链表存储结构的实现

——用C语言中的“结构指针”来描述

```
typedef struct LNode {
    ElemType data;
    struct LNode *next;
} LNode , *LinkList;
```



注意结点 p 与
结点 a_i 的区别

$(*p)$ 表示 p 所指向的结点

$(*p).data \Leftrightarrow p->data$ 表示 p 指向结点的**数据域**

$(*p).next \Leftrightarrow p->next$ 表示 p 指向结点的**指针域**