第2章 线性表

郑新、徐鹏飞、李健
北京师范大学人工智能学院
2025-2026学年 第一学期

考核要点

- **考核大纲**

  线性表的基本概念、线性表的实现（顺序存储、链式存储）、线性表的应用

- **复习要点**
  - 理解线性表的基本概念（A）
  - 熟练掌握线性表的顺序实现和链接实现（A）
  - 灵活运用线性表求解经典问题（B）
  - 理解广义表的基本概念（B）

学习方式

- 理论学习
  - 《数据结构》俞勇等著，101计划教材
  - 《数据结构》101计划工作平台
  - Data Structure Visualizations（数据结构可视化）
  - VisuAlgo（数据结构与算法可视化）
- 代码实践
  - GeeksforGeeks
  - LeetCode、力扣

提纲

- 2.1 问题引入：一元多项式
- 2.2 线性表的定义与结构
- 2.3 线性表的顺序存储实现
- 2.4 线性表的链式存储实现
- 2.5 线性表的应用
- 2.6 拓展延伸
- 2.7 应用场景：内存管理
- 2.8 小结

2.1 问题导入

- **例1： 一元多项式的表示及运算**
  - 一元多项式 $f(x) = a_0 + a_1x + \ldots + a_{n-1}x^{n-1} + a_nx^n$
  - 主要运算：多项式相加、相减、相乘等
- **【问题】如何在计算机中表示一元多项式并实现相关的运算？**
  - **关键数据一：** 多项式的项数n
  - **关键数据二：** 各项系数 a_i 及指数i

2.1 问题导入

- **方法1： 使用顺序存储结构直接表示**

  使用数组的各个分量表示多项式的各个系数项，如$a[i]$表示项x_i的系数a_i。

  例如： $f(x)=4x^5-3x^2+2x+1$

  表示为：

	$1\cdot x^0$	$2\cdot x^1$	$-3\cdot x^2$	0	0	$4\cdot x^5$	…
$a[i]$	1	2	-3	0	0	4	…
下标i	0	1	2	3	4	5	…

  - **两个多项式相加**： 两个数组对应的分量相加
  - **问题**： 如何表示高阶稀疏多项式：$f(x)=3x^2-4x^{8000}$

2.1 问题导入

- **方法2： 使用顺序存储结构表示非零项**
  - 每个非零项a_ix^i包含两个信息分别是系数a_i和指数i
  - 将多项式中非零项视为二元组(a_i, i)的集合
  - 如$f_1(x) = 9x^{15} - 15x^7 + 2x$　　$f_2(x) = 77x^{28} - 5x^7 + 12x + 99$

系数a_i	9	-15	2	–
指数i	15	7	1	–
数组下标	0	1	2	……

系数a_i	77	-5	12	99	–
指数i	28	7	1	0	–
数组下标	0	1	2	3	……

  - 这种表示的优点和缺点?

2.1 问题导入

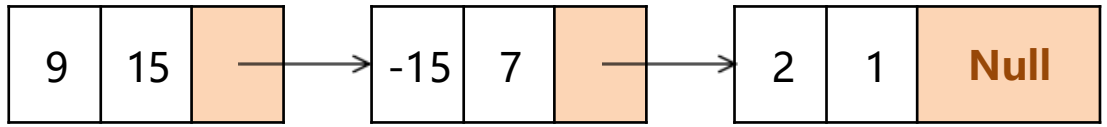
- **方法2：采用链表结构来存储多项式的非零项**

链表中每个结点存储一个非零项，包括两个数据域（系数和指数）和一个指针域。

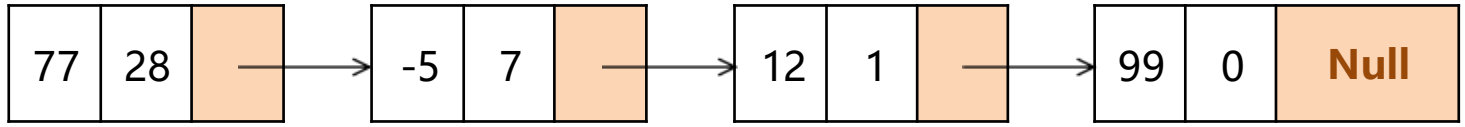
数据域


```
typedef struct PolyNode *Polynomial;
struct PolyNode{
    int coef;
    int expon;
    Polynomial link;
}
```

$f_1(x) = 9x^{15} - 15x^7 + 2x$



$f_2(x) = 77x^{28} - 5x^7 + 12x + 99$



2.1 问题导入

- 启示：数据结构的设计
  - 往往需要在算法简洁、可理解性与时间、空间效率之间权衡
  - 针对具体问题选择合适的数据结构及设计相应的算法
- 本章介绍：
  - 线性表的抽象定义
  - 基于顺序存储线性表实现方法
  - 基于链接存储的线性表实现方法
  - 线性表的基本操作，包括插入、删除等。

提纲

- 2.1 问题引入：一元多项式
- 2.2 线性表的定义与结构
- 2.3 线性表的顺序存储实现
- 2.4 线性表的链式存储实现
- 2.5 线性表的应用
- 2.6 拓展延伸
- 2.7 应用场景：内存管理
- 2.8 小结

2.2.1 线性表的定义

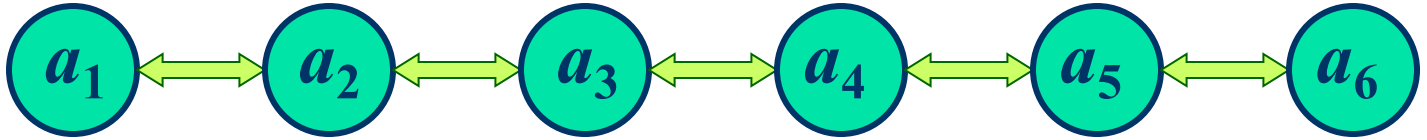
- 线性表的定义和特点
  - 定义 n（≥0）个相同数据类型数据元素的有限序列，记作

$$(a_1, a_2, \ldots, a_n)$$

a_i 是表中数据元素，n是表长度。

  - ✓ 线性表中的元素个数n称为线性表的长度。
  - ✓ 没有元素（n＝0） 的线性表称为空表。
  - ✓ a_1是唯一的起始位置，称为表头；a_n是唯一的结束位置，称为表尾。
  - 特点　线性排列
    - ✓ 除第一个元素外，其他每一个元素有一个且仅有一个直接前趋。
    - ✓ 除最后一个元素外，其他每一个元素有一个且仅有一个直接后继。

2.2.1 线性表的定义



- 理解线性表的要点是
  - a) 表中元素具有逻辑上的顺序性，在序列中各元素排列有其先后次序，有唯一的首元素和尾元素。
  - b) 表中元素个数有限。
  - c) 表中元素都是数据元素。即每一表元素都是原子数据，不允许“表中套表”。
  - d) 表中元素的数据类型都相同。这意味着每一表元素占有相同数量的存储空间。

2.2.1 线性表的定义

ADT List {

数据对象：线性表是n个数据元素的有限序列，$D = \{a_i | a_i \in ElemSet, i = 1, 2, 3, \ldots, n\}$

数据关系：$R = \{< a_{i-1}, a_i > a_{i-1}, a_i \in D, i = 1, 2, 3, \ldots, n\}$

基本操作：线性表 L ∈ List，整数 i 表示位置，元素 e ∈ ElementType

InitList (&L)：**初始化**操作。构造一个空的线性表L。

GetLength (L)：**求表长**操作。返回线性表L的长度，即L中元素的个数。

GetElem (L, i, &e)：按**位查找**操作。获取线性表L中第i个位置的元素的值，并返回元素a_i。

LocateElem (L, e)：按**值查找**操作。在线性表L中查找具有给定关键字值e的元素。

ClearList (&L)：**销毁**操作。销毁线性表L，并释放所占用的内存空间。

isEmpty(L)：**判空**操作。判断线性表是否为空，若为空，则返回ture，否则返回false。

ListInsert (&L, i, e)：**插入**操作。在线性表L中的第i个位置上插入指定元素e。

ListDelete (&L, i)：**删除**操作。删除线性表L中第i个位置的元素。

ListUpdate (L, i, e)：**更新**操作。将线性表L中第i个位置的元素修改为e。

PrintList (L)：**输出**操作。按先后顺序输出线性表L中的所有元素。

}

2.2.2 线性表的结构

- **线性表的逻辑结构：**

  数据元素之间线性的序列关系，即数据元素之间的前驱和后继关系。

- **线性表的物理结构：**

  线性表在计算机中的存储方式，又称为存储结构，即从程序实现的角度将逻辑结构映射到计算机存储单元中。

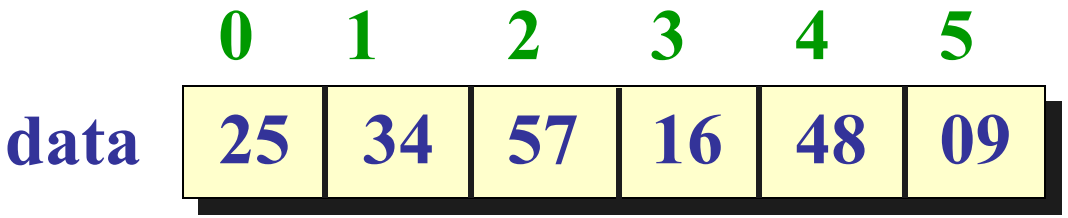
- **存储结构主要有两种形式：顺序存储结构和链式存储结构。**
  - **顺序存储结构**：数据元素被顺序地存储在连续的内存空间中，前驱和后继元素在物理空间上是相邻的。
  - **链式存储结构**：可以动态地申请存储数据的结点空间，并使用类似指针这样的手段将结点按顺序前后链接起来。

提纲

- 2.1 问题引入：一元多项式
- 2.2 线性表的定义与结构
- 2.3 线性表的顺序存储实现
- 2.4 线性表的链式存储实现
- 2.5 线性表的应用
- 2.6 拓展延伸
- 2.7 应用场景：内存管理
- 2.8 小结

2.3 线性表的顺序存储实现

- 顺序表的定义和特点
  - 定义 将线性表中的元素相继存放在一个连续的存储空间中，即构成顺序表。
  - 存储 它是线性表的顺序存储表示，可利用一维数组描述存储结构。
  - 特点 元素的逻辑顺序与物理顺序一致。
  - 访问方式 可顺序存取，可按下标直接存取。



2.3 线性表的顺序存储实现

- 顺序表定义

数组list.data[kMaxSize]: 从list.data[0]开始依次顺序存放，kMaxSize代表数组最大容量，list.last记录当前线性表中最后一个元素在数组中的位置，表空时list.last= -1。

last

data	a_1	a_2	……	a_i	a_{i+1}	……	a_n	……	-
i	0	1	……	*i*-1	*i*	……	*n*-1	……	kMaxSize-1

2.3 线性表的顺序存储实现

- 顺序表定义

数组list.data[kMaxSize]: 从list.data[0]开始依次顺序存放，kMaxSize代表数组最大容量，list.last记录当前线性表中最后一个元素在数组中的位置，表空时list.last= -1。

```
typedef int Position; /* 整型下标，表示元素的位置 */
typedef struct ListNode *List;
struct ListNode {
    ElemSet data[kMaxSize]; /* 存储数据的数组 */
    Position last; /* 线性表中最后一个元素在数组中的位置 */
};
```

2.3.2 顺序表的基本操作

- 初始化：顺序表的初始化即构造一个空表
  - 动态分配表结构所需要的存储空间
  - 将表中 list.last 指针置为-1，表示表中没有数据元素。

代码 2-2 产生一个初始空顺序表 InitList(*list*)

输入： 顺序表 *list*

输出： 完成了初始化的空顺序表 *list*

```
1   list.data ← new ElemSet[kMaxSize]   //申请顺序表空间
2   list.last ← -1;                     //空表中 list.last 值为-1
```

```
list = (List)malloc(sizeof(struct ListNode));
list->last = -1;
```

2.3.2 顺序表的基本操作

- 查找
  - 按下标查找：返回 list[i]，时间复杂度 O(1)
  - 按值查找：在线性表中查找与给定值x相等的数据元素
  - 方法：从第一个元素起依次和x比较，直到找到一个与x相等的数据元素，返回它在顺序表中的存储下标；或查遍整个表都没有找到与x相等的元素，则返回NIL。

算法 2-1 在顺序表 *list* 中查找元素 *x*　Search(*list*, *x*)

输入：顺序表 *list*，*x*∈ ElemSet

输出：元素 *x* 在顺序表 *list* 中的位置 *i* (由于顺序表中的位置从 0 开始，*x* 的实际位序是 *i*+1)；
如果 *x* 不在顺序表中返回 NIL

```
1  i ← 0;
2  while i ≤ list.last 并且 list.data[i] ≠ x do
3  |  i←i+1
4  end
5  if  i > list.last then    // 如果没找到，返回 NIL
6      i ← NIL
7  end
8  return i
```

时间复杂性：
- 1~(n + 1)/2
- 平均O(n)

2.3.2 顺序表的基本操作

- 查找实现

```
Position Search(List list, ElemSet x)
{
   Position i = 0;
   while ((i <= list->last) && (list->data[i] != x)) {
      i++;
   }
   if (i > list->last) {
      i = NIL;
   }
   return i;
}
```

2.3.2 顺序表的基本操作

- 查找算法性能分析
  - 查找成功的平均比较次数

$$ACN = \sum_{i=0}^{n-1} p_i \times c_i$$

  - 若查找概率相等，则

$$ACN = \frac{1}{n}\sum_{i=0}^{n-1}(i+1) = \frac{1}{n}(1+2+\cdots+n) =$$

$$= \frac{1}{n} * \frac{(1+n)*n}{2} = \frac{1+n}{2}$$

  - 查找不成功　数据比较 n 次。

时间复杂度为O(n)

2.3.2 顺序表的基本操作

- 插入
  - 在表的第i个位序上插入一个值为x的新元素

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

  - 执行步骤：
    - 将a_i～a_n顺序向后移动（移动次序从后到前），为新元素让出位置；
    - 将x置入空出的第i个位序；
    - 修改 list.last 指针(相当于修改表长)，使之仍指向最后一个元素。

算法 2-2 在顺序表 *list* 的第 *i* 个位置上插入元素 *x*　Insert (*list*, *i*, *x*)

输入： 顺序表 *list*，*i* 是插入位置的序号（从 1 开始），*x*∈ ElemSet

输出： 完成插入后的顺序表 *list*

```
1  if  list.last = kMaxSize−1 then    // 表空间已满，不能插入
2  |  表满不能插入，退出
3  end
4  if  i<1 或者 i>list.last+2 then   //检查 i 的合法性。注意 i 代表位序，不是数组下标
5  |  插入位置不合法，退出
6  end
7  for  j ← list.last downto i-1 do
8  |  list.data[j+1] ← list.data[j]   //将 ai~an 顺序向后移动
9  end
10 list.data[i-1] ← x   // 新元素插入
11 list.last ← list.last + 1 // list.last 仍指向最后元素
```

时间复杂度为O(n)

2.3.2 顺序表的基本操作

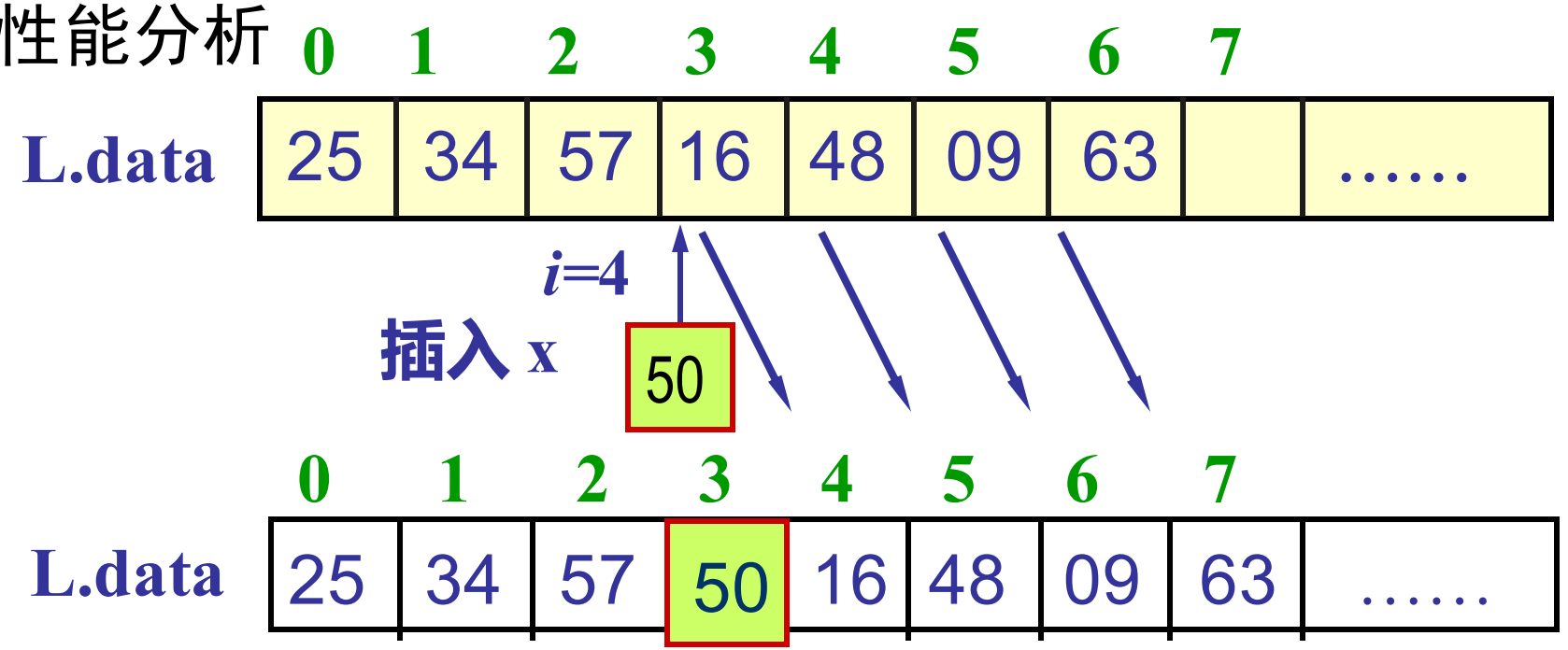
- 插入实现

```
void Insert(List list, Position i, ElemSet x)
{ /* 注意i代表位序，从1开始，不是数组下标 */
   int j;

   if (list->last == (kMaxSize-1)) {  /* 表满不能插入，退出 */
      printf("错误：表满不能插入。\n");
      return;
   }
   if ((i<1) || (i>(list->last+2))) { /* 插入位置不合法，退出 */
      printf("错误：插入位置不合法。\n");
      return;
   }
   for (j=list->last; j>=i-1; j--) {
      list->data[j+1] = list->data[j]; /* 将第i个到第n个元素顺序向后移动 */
   }
   list->data[i-1] = x; /* 新元素插入，数组下标i-1对应第i个位置 */
   list->last++;  /* list->last仍指向最后元素 */
}
```

2.3.2 顺序表的基本操作

- 插入算法性能分析



  - 插入时平均移动元素个数AMN

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

2.3.2 顺序表的基本操作

- 删除
  - 将线性表中的第i个位序上的元素删除

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

    $\rightarrow (a_1, a_2, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n)$
  - 执行步骤：
    - (1)将～顺序向前移动，元素被覆盖；
    - (2)修改list.last指针(相当于修改表长)使之仍指向最后一个元素。

时间复杂度为O(n)

算法 2-3 从顺序表 *list* 中删除第 *i* 个元素　Remove (*list*, *i*)

输入： 顺序表 *list*，*i* 是删除元素的位置序号（从 1 开始）

输出： 完成删除后的顺序表 *list*

```
1  if  i<1 或 i>list.last+1 then  // 检查空表及删除位置的合法性
2  |   不存在这个元素，退出
3  end
4  for  j←i to list.last do
5  |   list.data[j-1]← list.data[j]  // 将 a_{i+1}～a_n 顺序向前移动
6  end
7  list.last ←list.last – 1  // list.last 仍指向最后元素
```

2.3.2 顺序表的基本操作

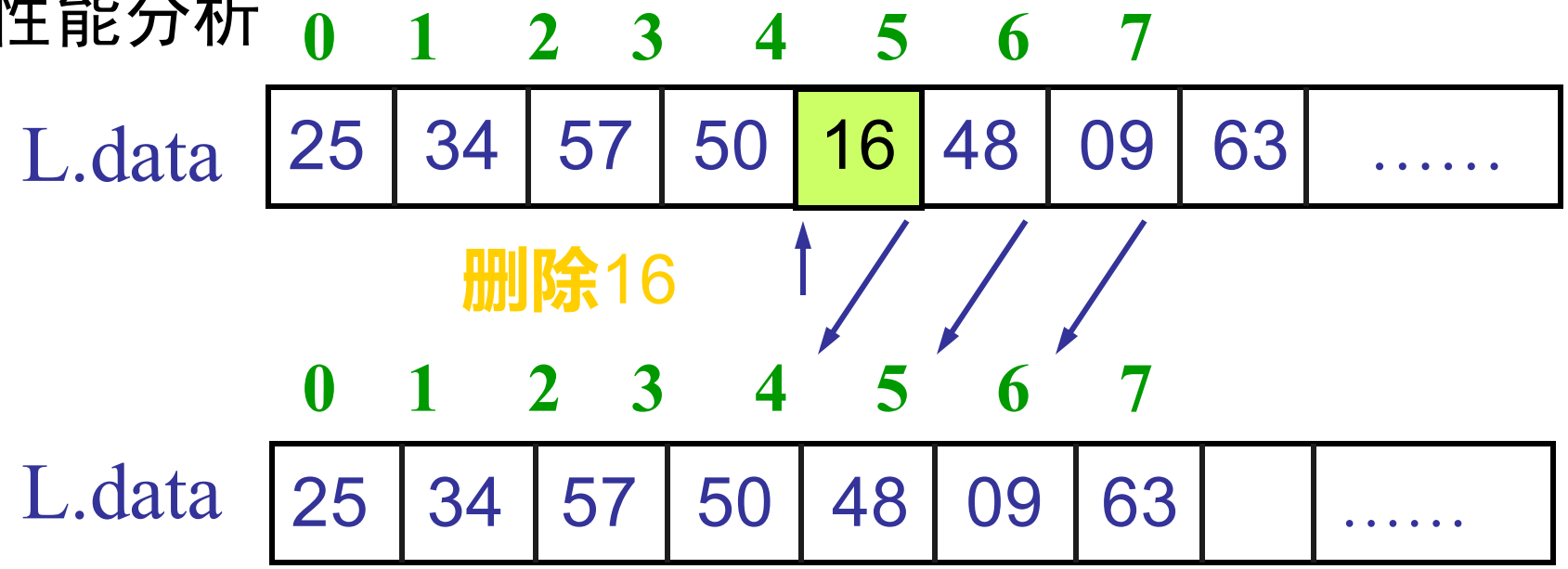
- 删除实现

```
void Remove( List list, Position i )
{ /* 注意i代表位序，从1开始，不是数组下标 */
   int j;

   if ((i<1) || (i>(list->last+1))) { /* 检查空表及删除位置的合法性 */
      printf("错误：不存在这个元素。\n");
      return;
   }
   for (j=i; j<=list->last; j++) {
      list->data[j-1] = list->data[j]; /* 将第i+1个到第n个元素顺序向前移动 */
   }
   list->last--;  /* list->last仍指向最后元素 */
}
```

2.3.2 顺序表的基本操作

- 删除算法性能分析



  - 删除时平均移动元素个数AMN

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

时间复杂度为O(n)

提纲

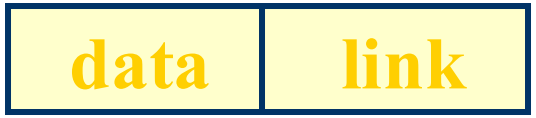
- 2.1 问题引入：一元多项式
- 2.2 线性表的定义与结构
- 2.3 线性表的顺序存储实现
- 2.4 线性表的链式存储实现
- 2.5 线性表的应用
- 2.6 拓展延伸
- 2.7 应用场景：内存管理
- 2.8 小结

2.4 线性表的链接存储实现

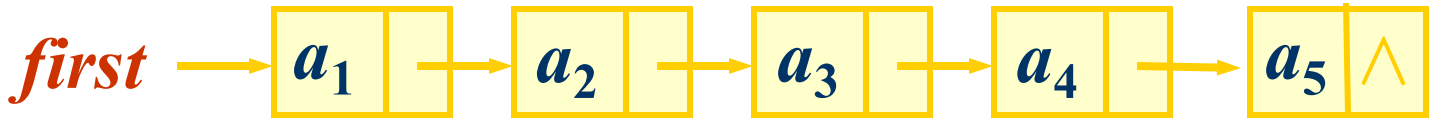
- 线性链表是线性表的链接存储表示。元素之间的逻辑顺序是通过各结点中的**链接指针**来指示的。
- 线性链表分类：
  - 单链表
  - 循环链表
  - 双向链表
- 链表中第一个元素结点称为首元结点，最后一个元素称为尾结点。首元结点不是头结点。

2.4.1 单链表

- 单向链表特点
  - 每个元素(表项)由结点(Node)构成。



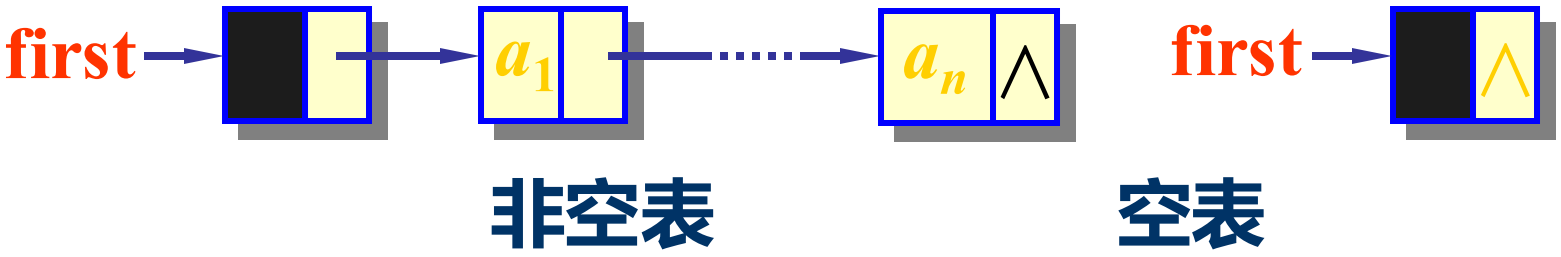
  - 线性结构



  - 结点可以连续，可以不连续存储
  - 结点的逻辑顺序与物理顺序可以不一致
  - 表可扩充

2.4.2 单链表的基本操作

- 带头结点的单链表
  - 头结点位于表的最前端，本身不带数据，仅标志表头。
  - 设置头结点的目的是
    - 统一空表与非空表的操作
    - 简化链表操作的实现。



2.4.1 单链表

- 单链表的结构定义

```
typedef int ElemSet;
typedef struct ListNode *Position; /* 指针即结点位置 */
struct ListNode {
    ElemSet data; /* 存储数据*/
    Position next; /* 线性表中下一个元素的位置 */
};
typedef struct HeadNode *List;
struct HeadNode {
    Position head; /* 单链表头指针 */
    int length; /* 表长 */
};
```

2.3.2 单链表的基本操作

- **求表长**：求单链表元素个数
  - 设一个移动指针p和计数器counter，初始化
  - p逐步往后移，同时计数器counter加1
  - 当后面不再有结点时，counter的值就是结点个数，即表长。

算法 2-4 求单链表 *list* 中的元素个数，即表长　Length (*list*)

输入： 单链表 *list*，其中 *list.head* 指向链表头结点

输出： 单链表长度

```
1  p ← list.head
2  counter ← 0
3  while  p ≠ NIL  do
4  |   counter ← counter + 1
5  |   p ← p.next
6  end
7  return counter
```

时间复杂性O(n)

```
int Length( List list )
{
   Position p;
   int counter;

   p = list->head;  /* 从第1个结点开始 */
   counter = 0;
   while (p != NULL) {
      counter++;   /* 结点计数 */
      p = p->next; /* 指针后移 */
   }
   return counter;
}
```

2.3.2 单链表的基本操作

- **查找**：按序号查找Get (list, i)、按值查找Search(list, x)
  - 按序号查找步骤：
    - 从链表的第一个元素结点起，判断当前结点是否是第i个；
    - 若是，则返回该结点的值，否则继续后一个，直到表结束为止。
    - 如果没有第i个结点则返回错误码(ErrorCode)。

时间复杂性：O(n)

算法 2-5 返回单链表 *list* 中第 *i* 个元素值　Get (*list*, *i*)

输入： 单链表 *list*，其中 *list.head* 指向链表头结点；*i* 是待查找元素在链表中的位序（从 1 开始）

输出： 第 *i* 个元素值；如果不存在则返回错误码（ErrorCode）

```
1   if list.head = NIL 或 i = 0 then //空表或查找位置不合法
2   |  return ErrorCode
3   end
4   p ← list.head
5   counter ← 1
6   while  p ≠ NIL 且 counter < i  do
7   |  p ← p.next
8   |  counter ← counter +1
9   end
10  if  p ≠ NIL  then
11  |  return p.data
12  else
13  |  return ErrorCode //不存在第 i 个元素
14  end
```

2.3.2 单链表的基本操作

- **查找：分按序号查找Get (list, i)**

```
ElemSet Get(List list, int i)
{ /* i是待查找元素在链表中的位序，从1开始 */
   Position p;
   int counter;
   if ((list->head == NULL) || (i == 0)) {
      return ErrorCode; /* 空表或查找位置不合法 */
   }
   p = list->head;
   counter = 1;
   while ((p != NULL) && (counter < i)) {
      p = p->next; /* 指针后移 */
      counter++;  /* 结点计数 */
   }
   if (p != NULL) { /* 以counter等于i退出while循环 */
      return p->data; /* 返回第i个元素值 */
   }
   else { /* 不存在第 i 个元素 */
      return ErrorCode;}
}
```

2.3.2 单链表的基本操作

- **查找**
  - 按值查找步骤：
    - 从链表的第一个元素结点起，判断当前结点其值是否等于x；
    - 若是，返回该结点的位置（即指向该结点的指针），否则继续后一个，直到表结束为止。
    - 找不到时返回空（NIL）。

算法 2-6 在单链表 *list* 中查找元素 *x* 所在结点　Search(*list*, *x*)

输入： 单链表 *list*，其中 *list.head* 指向链表头结点；*x*∈ ElemSet

输出： 元素 *x* 在单链表 *list* 中的位置，即指向该结点的指针；如果 *x* 不在单链表中返回 NIL

```
1   p ← list.head
2   while  p ≠ NIL 且 p.data ≠ x  do
3       p ← p.next
4   end
5   return p
```

时间复杂性：O(n)

2.3.2 单链表的基本操作

- **查找：**按值查找Search(list, x)

```
Position Search( List list, ElemSet x )
{
   Position p;

   p = list->head;
   while ((p != NULL) && (p->data != x)) {
      p = p->next;
   }
   return p;
}
```

  - 查找成功返回结点地址；查找不成功返回空。
  - 注意，while循环条件p != NULL 和 p->data != x不能错位，考虑为什么。

2.4.2 单链表的基本操作

- **插入：**在list的第i个位置上插入元素x

  步骤

  - **找到第i-1个结点（要点）**；
  - 若存在，则申请一个新结点的空间并填上相应值x，然后将新结点插到第i-1个结点之后；
  - 如果不存在则直接退出：

注意：表空和不空时候的不同处理方式

时间复杂度为O(n)

算法 2-7 在单链表 *list* 的第 *i* 个位置上插入元素 *x*　Insert (*list*, *i*, *x*)

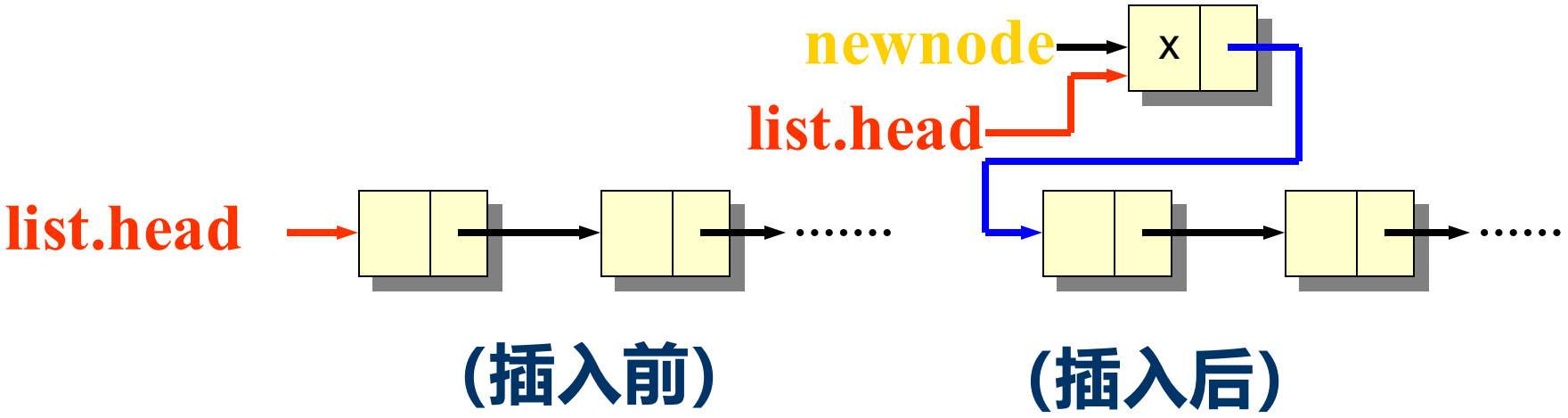
输入：单链表 *list*，*i* 是插入位置的序号（从 1 开始），*x* ∈ ElemSet

输出：完成插入后的单链表 *list*

```
1  if i < 1 then
2  |  插入位置不合法，退出
3  end
4  if i =1 then  //插入第 1 个结点
5  |  new_node ← new ListNode //创建新的结点
6  |  new_node.data ← x
7  |  new_node.next ← list.head //插入表头
8  |  list.head ← new_node
9  else  // i > 1，寻找第 i-1 个结点并插入其后
10 |  p ← list.head
11 |  counter ← 1
12 |  while p ≠ NIL 且 counter < (i-1) do
13 |  |  p ← p.next
14 |  |  counter ← counter +1
15 |  end
16 |  if p ≠ NIL then //p 指向第 (i-1)个结点
17 |  |  new_node ← new ListNode //创建新的结点
18 |  |  new_node.data ← x
19 |  |  new_node.next ← p.next
20 |  |  p.next ← new_node
21 |  else
22 |  |  插入位置不合法，退出
23 |  end
24 end
```

2.4.2 单链表的基本操作

- 插入
  - 第一种情况（前插法）：在第 1 个结点前插入



2.4.2 单链表的基本操作

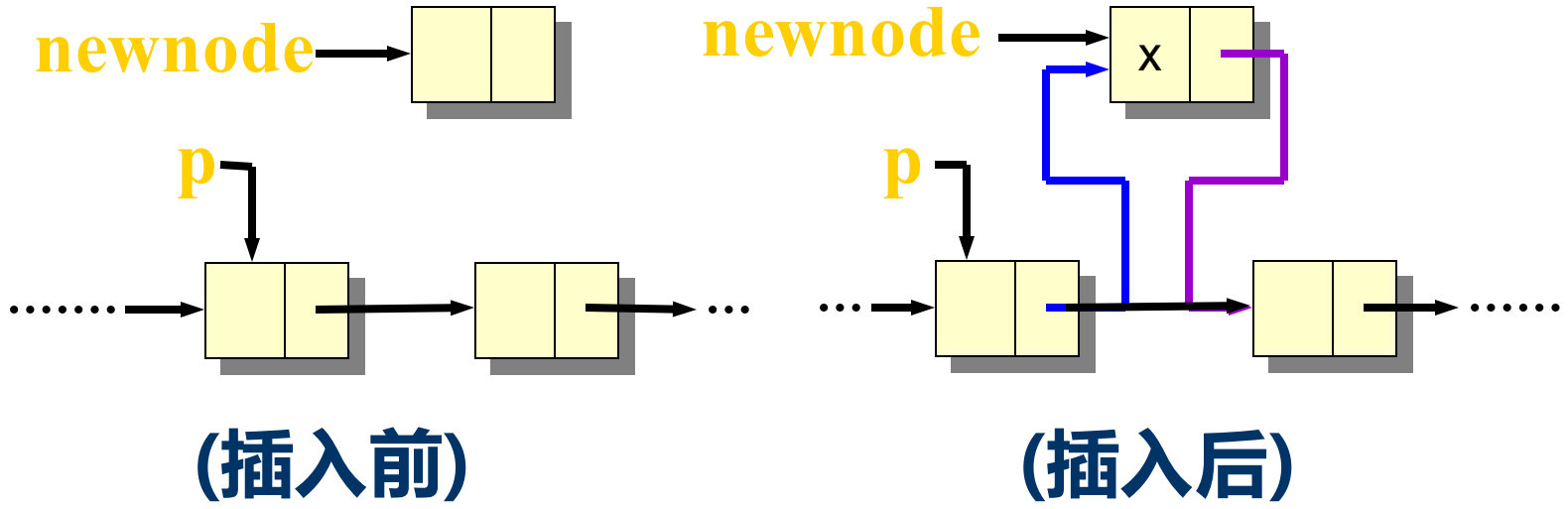
- 插入
  - 第一种情况（前插法）：在第 1 个结点前插入

```
if (i == 1) { /* 插入第1个结点 */
        new_node = (Position)malloc(sizeof(struct ListNode)); /* 创建新的结点 */
        new_node->data = x;
        new_node->next = list->head; /* 将新结点插入链表头 */
        list->head = new_node;
        list->length++;
}
```

2.4.2 单链表的基本操作

- 第二种情况：在链表中间插入

  首先定位指针 p 到插入位置，再将新结点插在其后：



- 注意，需要先更新新节点next指针,再更新当前节点next指针，不能错位，考虑为什么。

  newnode->next= p->next

  p->next= newnode

2.4.2 单链表的基本操作

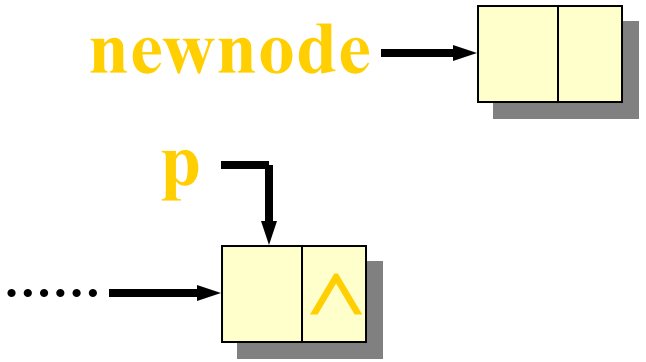
- 第二种情况：在链表中间插入：

```
else { /* i>1，寻找第 i-1 个结点并插入其后 */
        p = list->head;
        counter = 1;
        while ((p != NULL) && (counter < (i-1))) { /* 找第 i-1 个结点 */
                p = p->next;
                counter++;
        }
        if (p != NULL) { /* p指向第 (i-1)个结点 */
                new_node = (Position)malloc(sizeof(struct ListNode)); /* 创建新的结点 */
                new_node->data = x;
                new_node->next = p->next; /* 将新结点插在p后面 */
                p->next = new_node;
                list->length++;
        }
        else { /* 插入位置不合法，退出 */
                printf("错误：插入位置不合法。\n");
        }
}
```

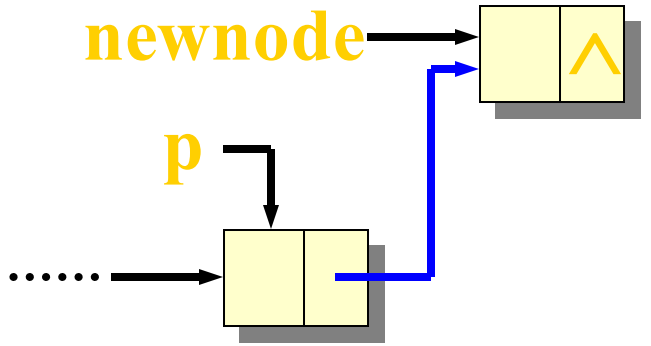
2.4.2 单链表的基本操作

- 第三种情况（尾插法）：在链表末尾插入

  首先定义指针 p 到尾结点位置，再将新结点插在其后，新结点成为新的尾结点。



(插入前)



(插入后)

2.4.2 单链表的基本操作

- 第三种情况（尾插法）：在链表末尾插入（不带尾指针）

```
p = list->head;
while ((p!= NULL) && (counter < (i-1))) { /* 找第 i-1 个结点 */
while (p->next != NULL) { /* 找最后一个节点*/
        p = p->next;
}

new_node = (Position)malloc(sizeof(struct ListNode)); /* 创建新的结点 */
new_node->data = x;
new_node->next = p->next; /* 将新结点插在p后面 */
p->next = new_node;
list->length++;
```

2.4.2 单链表的基本操作

- 第三种情况（尾插法）：在链表末尾插入（带尾指针）

```
typedef struct HeadNode *List;
        struct HeadNode {
        Position head; /* 单链表头指针 */
        Position tail; /* 单链表尾指针 */
        int length; /* 表长 */
};
```

```
new_node = (Position)malloc(sizeof(struct ListNode)); /* 创建新的结点 */
new_node->data = x;
new_node->next=NULL
if(list.head == NULL){
        list.tail=new_node
        list.head=new_node
}else{
        list.tail->next=new_node
        list.tail=new_node
}
list->length++;
```

2.4.2 单链表的基本操作

- **插入：**在list的第i个位置上插入元素x

```
void Insert (List list, int i, ElemSet x)
{ /* i是插入位置的序号，从1开始 */
   Position new_node, p;
   int counter;

   if (i < 1) { /* 插入位置不合法，退出 */
      printf("错误：插入位置不合法。\n");
      return;
   }
   if (i == 1) { /* 插入第1个结点 */
      new_node = (Position)malloc(sizeof(struct ListNode)); /* 创建新的结点 */
      new_node->data = x;
      new_node->next = list->head; /* 将新结点插入链表头 */
      list->head = new_node;
      list->length++;
   }
```

2.4.2 单链表的基本操作

```
    else { /*  i>1，寻找第 i-1 个结点并插入其后 */
        p = list->head;
        counter = 1;
        while ((p != NULL) && (counter < (i-1))) { /* 找第 i-1 个结点 */
            p = p->next;
            counter++;
        }
        if (p != NULL) { /* p指向第 (i-1)个结点 */
            new_node = (Position)malloc(sizeof(struct ListNode)); /* 创建新的结点 */
            new_node->data = x;
            new_node->next = p->next; /* 将新结点插在p后面 */
            p->next = new_node;
            list->length++;
        }
        else { /* 插入位置不合法，退出 */
            printf("错误：插入位置不合法。\n");
        }
    }
}
```

2.4.2 单链表的基本操作

- **删除**：在单链表中删除指定位序i的元素
  - 找到被删除结点的前一个元素的指针（要点）
  - 再删除结点并释放空间。

时间复杂度为O(n)

算法 2-8 从单链表 *list* 中删除第 *i* 个元素 Remove (*list*, *i*)

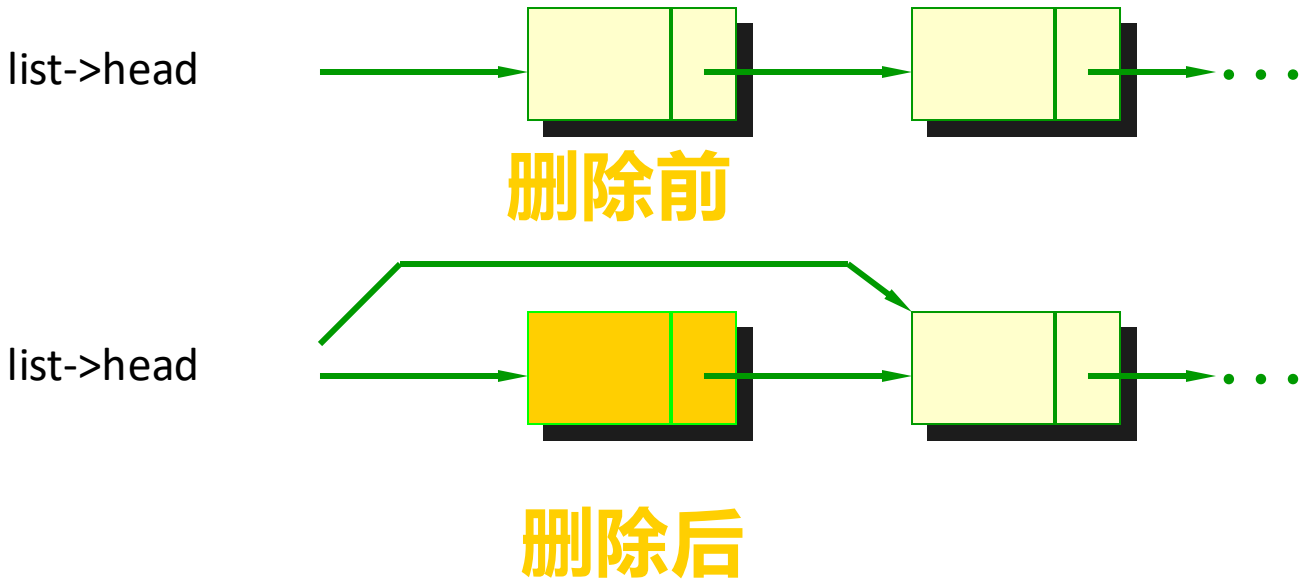
输入： 单链表 *list*，*i* 是删除元素的位置序号（从 1 开始）

输出： 完成删除后的单链表 *list*

```
1  if i < 1 then
2  |  删除位置不合法，退出
3  end
4  p ← list.head
5  if p ≠ NIL 且 i =1 then //删除第 1 个结点
6  |  list.head ← p.next
7  |  delete p
8  else // i > 1，寻找第 i-1 个结点
9  |  counter ← 1
10 |  while p ≠ NIL 且 counter < (i-1) do
11 |  |  p ← p.next
12 |  |  counter ← counter +1
13 |  end
14 |  if p ≠ NIL 且 p.next ≠ NIL then //p 指向第 (i-1)个结点，且待删除结点存在
15 |  |  deleted_node ← p.next
16 |  |  p.next ← deleted_node.next
17 |  |  delete deleted_node
18 |  else
19 |  |  删除位置不合法，退出
20 |  end
21 end
```

2.4.2 单链表的基本操作

- **删除**
  - 第一种情况: 删除表头（表中第 1 个元素）



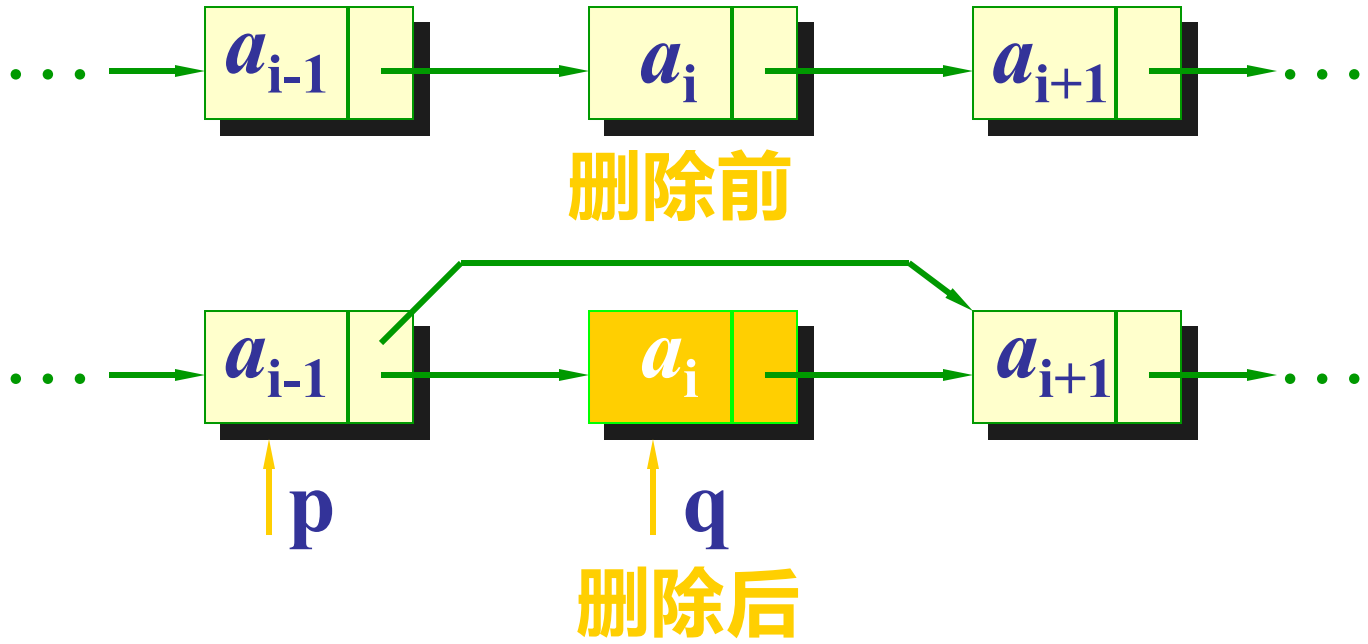
2.4.2 单链表的基本操作

- **删除**
  - 第一种情况: 删除表中第 1 个元素

```
p = list->head;
if ((p != NULL) && (i == 1)) { /* 删除第1个结点 */
        list->head = p->next;
        free(p);
        list->length--;
}
```

2.4.2 单链表的基本操作

- 删除
  - 第二种情况: 删除表中第i个节点



在单链表中第i个 的节点

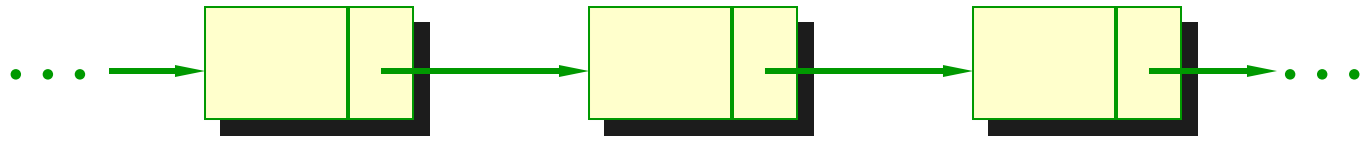
2.4.2 单链表的基本操作

- 删除
  - 第二种情况: 删除表中第i个节点

```
counter = 1;
while ((p != NULL) && (counter < (i-1))) {
        p = p->next;
        counter++;
}
if ((p != NULL) && (p->next != NULL)) { /* p指向第 (i-1)个结点，且待删除结点存在 */
        deleted_node = p->next;
        p->next = deleted_node->next;
        free(deleted_node);
        list->length--;
}
else { /* 删除位置不合法，退出 */
            printf("错误：删除位置不合法。\n");
}
```

2.4.2 单链表的基本操作

- 删除
  - 第三种情况: 删除表尾节点



删除前

删除后

2.4.2 单链表的基本操作

- 删除
  - 第三种情况: 删除表尾节点

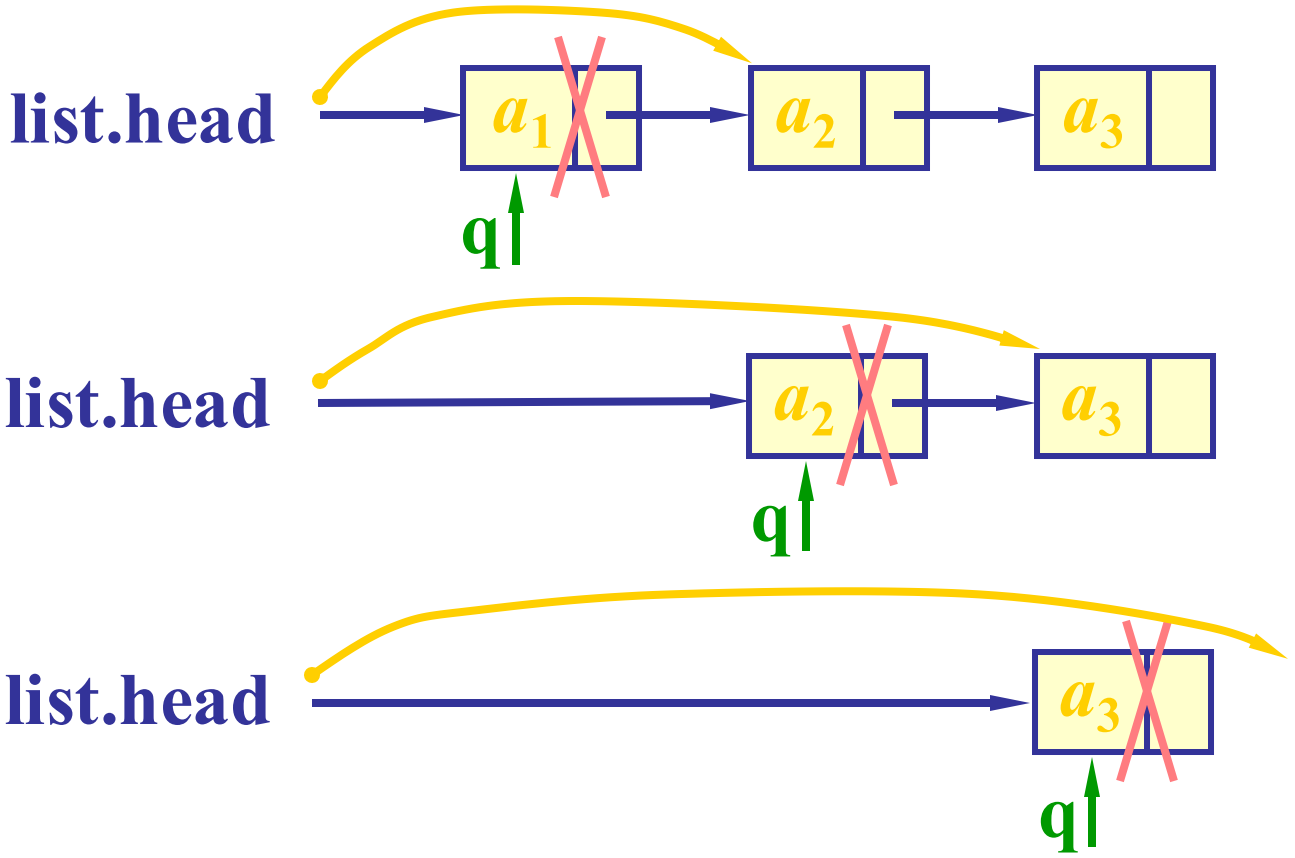
```
p = list->head;

while ((p != NULL) && (p->next != NULL) && (p->next->next != NULL)) {
        p = p->next;
}

if ((p != NULL) && (p->next != NULL)) { /* p指向尾节点上一个节点，且待删除结点存在 */
        free(p->next);
        p->next = NULL;
}
else { /* 删除位置不合法，退出 */
                printf("错误：删除位置不合法。\n");
}
```

2.4.2 单链表的基本操作

- 单链表清空



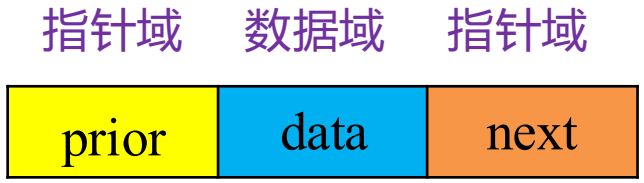
2.4.2 单链表的基本操作

- 单链表清空

```
void makeEmpty(List list) {
//删去链表中除表头结点外的所有其他结点
    Position q;
    while(list.head != NULL) { //链不空时
        q = list.head; //q指示首元结点
        list.head = q->next; //摘下q指示结点
        free(q); //释放它
    }
}
```

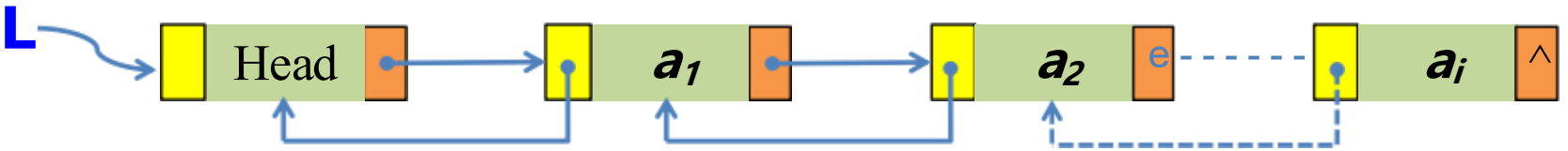
2.4.3 双向链表

- 双向链表的定义和实现
  - 在前后两个方向上都有指针的链表称为**双向链表（Double Linked List）**。双向链表包含一个数据域和两个指针域，其中**数据域（data）**用于存储信息，两个**指针域prior**和**next**分别指向直接前 驱和直接后继。



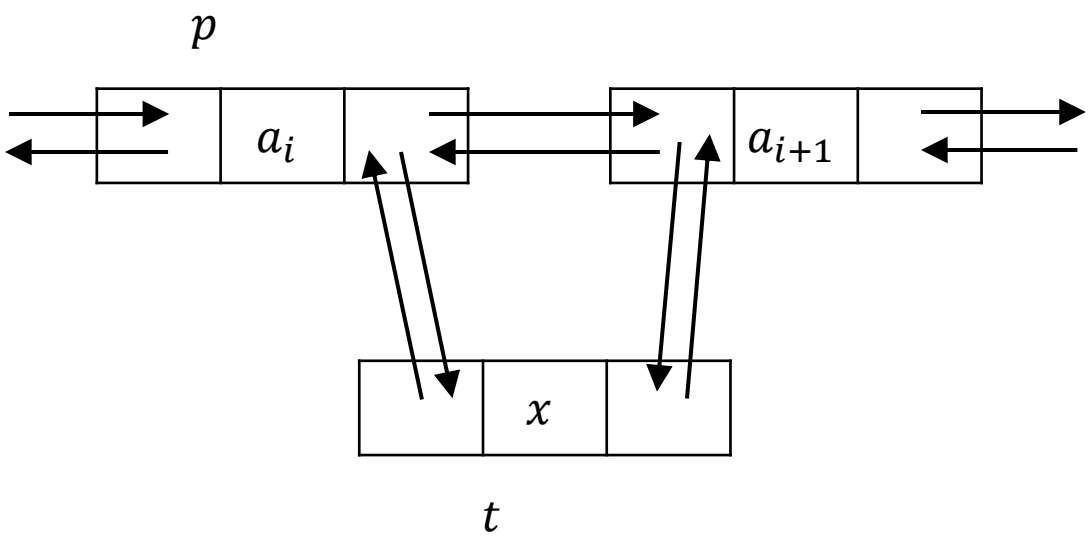
双向链表中结点类型的定义

```
typedef struct DLNode {
    ElemType data;          // 数据域
    struct DLNode *prior;   // 直接前驱指针域
    struct DLNode *next     // 直接后继指针域
} DLNode, *DLinkList        // LinkList为指向结构体LNode的指针类型
```



2.4.3 双向链表

- 插入



代码 2-3 在双向链表中 p 所指结点后面插入 t 所指的结点 DLLInsert(p, t)

输入： 双向链表中某结点指针 p，待插入结点的指针 t

输出： 插入完成后的双向链表 *list*

```
1  p.next.prior ← t
2  t.next ← p.next
3  p.next ← t
4  t.prior ← p
```

思考：2、3步的顺序不能换，为什么？

2.4.3 双向链表

- 插入

```
Status DListInsert (LinkList &L, int i, ElemType &e){
   p = L;j = 0;
   t = new DLNode;
   t -> data = x;
   while (p && j < i-2)
       p = p -> next; ++j;
   if (!p || j > i-2) return ERROR;
   p-> next -> prior = t;
   t -> next = p->next;
   p -> next = t;
   t-> prior = p;
   return OK;
} // DListInsert
```

0. 初始化指针和链表， p指向第一个结点
1.1 构造一个新结点 t
1.2 将结点的数据域置为 x
2. 找到链表第a_{i-1}个结点， 用指针p指向

   若结点不存在， 则返回Error
3. 将结点a_i的prior指针域指向新结点t
4. 将新结点t的next指针域指向结点p的直接后驱
5. 将p的next指针域指向新节点t
6. 将新结点t的prior指针域指向结点p

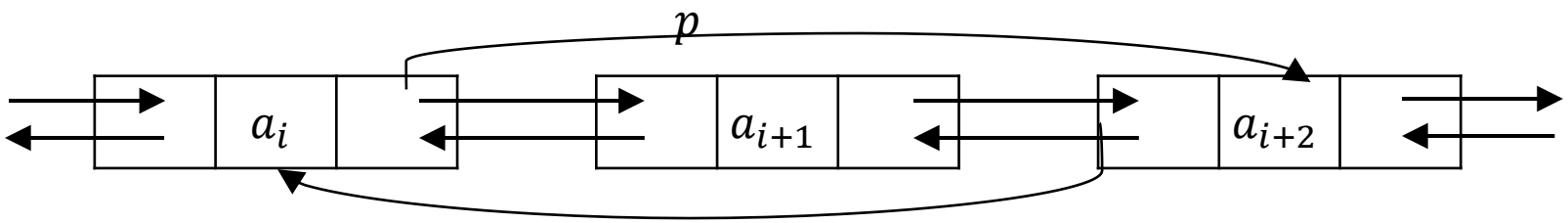
最好情况：（假设从表头开始遍历链表）在表头插入元素，时间复杂度为 O(1)

最坏情况：（假设从表头开始遍历链表）在表尾插入元素，时间复杂度为 O(n)

平均情况：$\sum_{i=1}^{n} p_i \times i = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2}$ **时间复杂度**：O(n)

2.4.3 双向链表

- 删除



  - p. next. prior ← p. prior
  - p.prior.next← p. next
  - delete p

2.4.3 双向链表

- 删除

```
Status DListDelete (LinkList &L, int i, ElemType &e){
   p = L;j = 0;
   while (p && j < i-1)
       p = p -> next; ++j;
   if (!(p -> next) ||j > i-1) return ERROR;
   p -> prior -> next = p -> next;

   p -> next -> prior =  p->prior;

   free(p);
   return OK;
} // DListDelete
```

0. 初始化指针和链表， p指向第一个结点
1. 寻找链表的第 *i*个结点， 并用指针p指向

   若结点不存在， 则返回Error
2. 将结点a_i的prior指针域的直接后继**指向**结点a_i的直接后继， 即跳过待删除结点a_i
3. 将结点a_i的next指针域的直接前驱**指向**结点a_i的直接前继， 即反向也跳过待删除结点a_i
4. 释放待删除结点s所指的空间

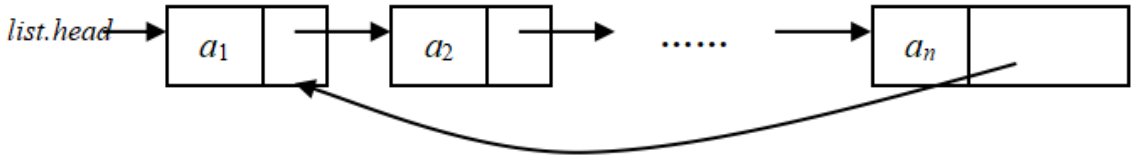
最好情况：（假设从表头开始遍历链表）在表头删除元素，时间复杂度为 O(1)

最坏情况：（假设从表头开始遍历链表）在表尾删除元素，时间复杂度为 O(n)

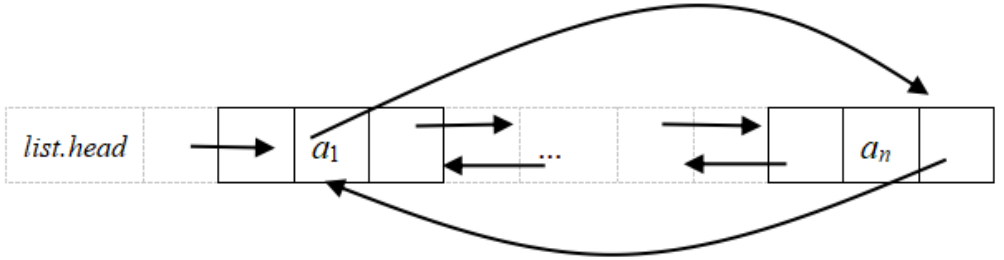
平均情况：$\sum_{i=1}^{n} p_i \times i = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2}$ **时间复杂度**：O(n)

2.4.4 循环链表

- **循环单链表**：链表终端结点的指针指向链表的起始结点



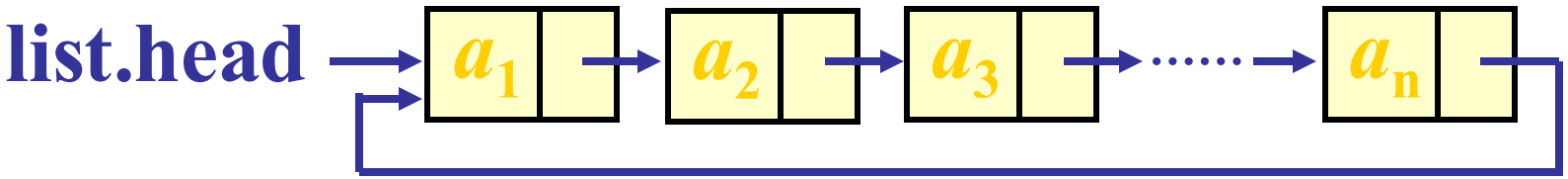
- **循环双向链表**：让终端结点的后继指针指向链表的起始结点，同时让起始结点的前驱指针指向链表的终端结点



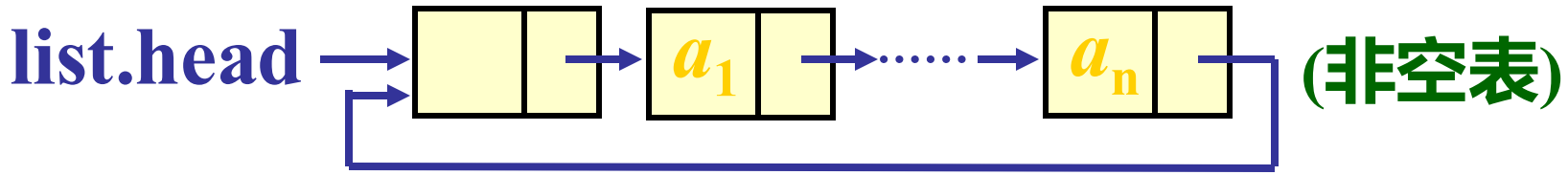
✓ 在循环链表中，从任何一个结点出发都可以到达链表的其他结点。
✓ 除表尾结点，（双向）循环链表的插入、删除和其他基本操作与（双向链表）单链表完全相同。

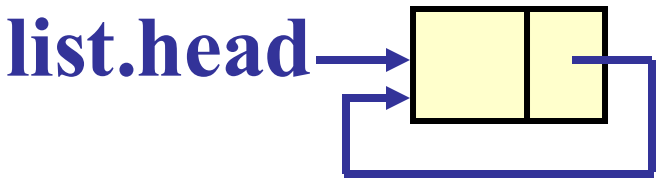
2.4.4 循环链表

- 循环单链表
  - 循环单链表是单链表的变形。链表尾结点的 link 指针不是 NULL，而是指向了表的前端。



  - 为简化操作，在循环单链表中往往加入头结点。



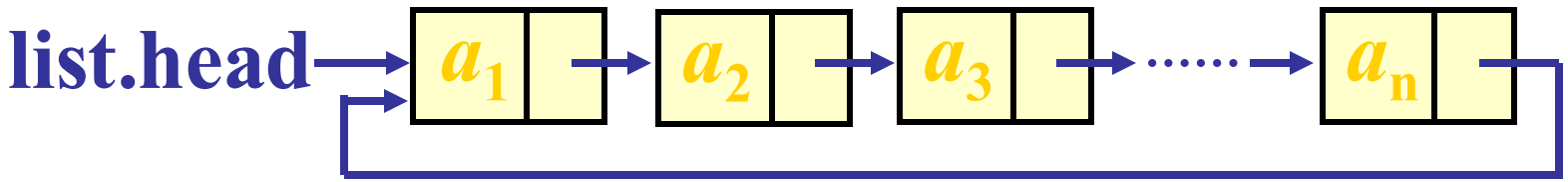


2.4.4 循环链表

- 循环单链表
  - 循环单链表的特点是：只要知道表中某一结点的地址，就可搜寻到所有其他结点的地址。
  - 在搜寻过程中，没有一个结点的 next 域为空。

```
for ( p =  list.head->next; p !=  list.head; p = p->next)
    do S;
```

  - 循环单链表的所有操作的实现类似于单链表，差别在于检测到链尾，指针不为NULL，而是回到链头。



2.4.4 循环链表

- 循环单链表
  - 结构定义

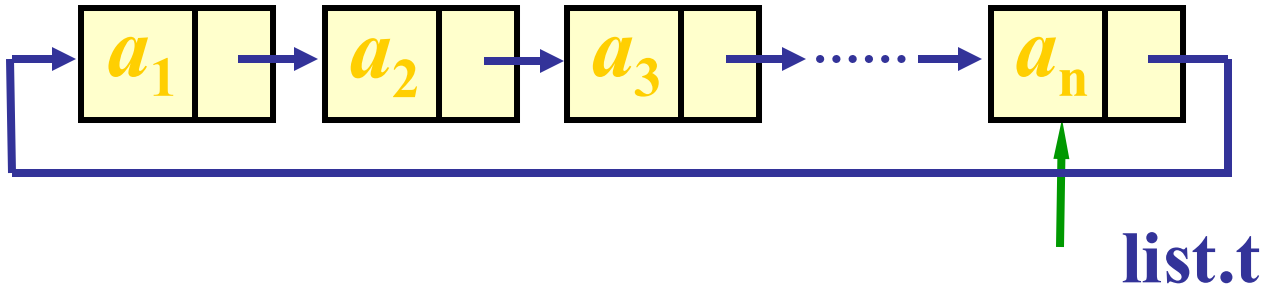
```
typedef int DataType;
typedef struct node {//循环链表定义
    DataType data;//结点数据
    struct node *link; //后继结点指针
} CircNode, *CircList;
```

  - 在链表中将指针 p 定位于第 i 个结点的操作

```
CircNode *p = first;  int k = 0;//first是头结点
while ( p->link != first && k < i )    //回到头结点失败
    { p = p->link;  k++; }//否则 p 指到目标
```

2.4.4 循环链表

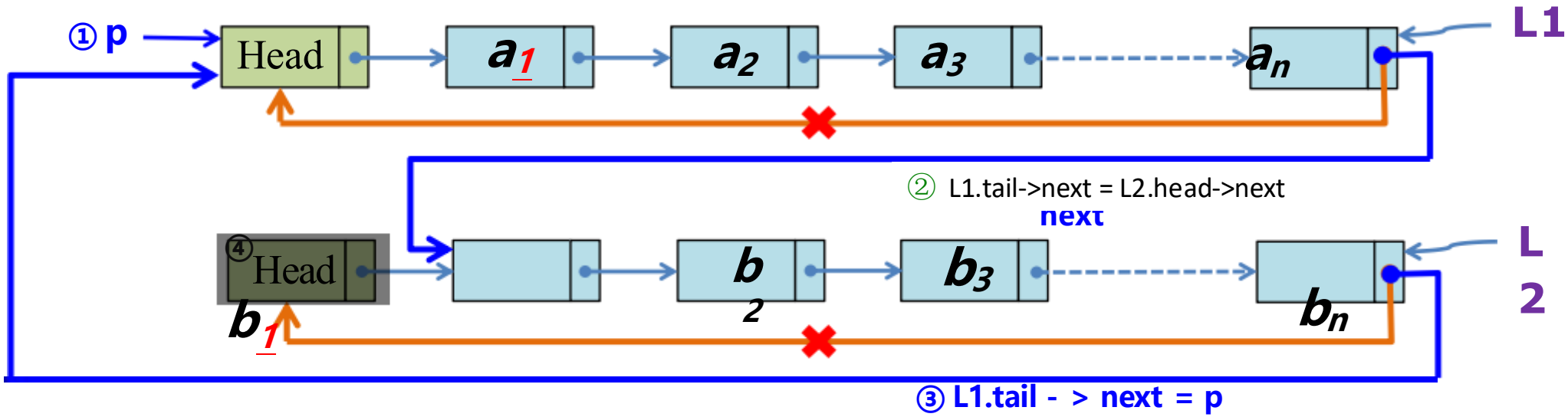
- 带尾指针的循环单链表



  - 如果插入与删除仅在链表的两端发生，可采用带表尾指针的循环链表结构。
    - 在表尾可直接插入新结点，时间复杂性 O(1);
    - 在表尾删除时要找前趋，时间复杂性 O(n);
    - 在表头插入相当于在表尾插入;
    - 在表头可直接删除，时间复杂性 O(1)。

2.4.4 循环链表

- 循环链表的合并



- 将指针p指向链表L1的头结点作为新链表的头结点
- 将链表L1的尾结点的指针next指向链表L2的首元结点
- 将链表L2的尾结点的指针next指向链表L1的头结点
- 释放链表L2的头结点

```
List Connect (List L1, List L2) {
        p = L1.head; // ①
        L1.tail -> next = L2.head->next // ②
        L2.tail -> next = p // ③
        free(L2) // ④
    return L1;
}
```

2.4.4 静态链表

- 静态链表定义

  用数组存放线性表中的元素，但并不按照元素顺序在数组中依序存放，而是给每个数组元素增加一个域，用于指示线性表中下一个元素的位置（即数组的下标）

  - 物理存储空间上依赖于数组
  - 元素的逻辑链接关系采用链表思想

data		g	d	a	e	b	f		h	c	
next	**3**	8	4	5	6	9	1		0	2	
下标	**0**	1	2	3	4	5	6	7	8	9	10

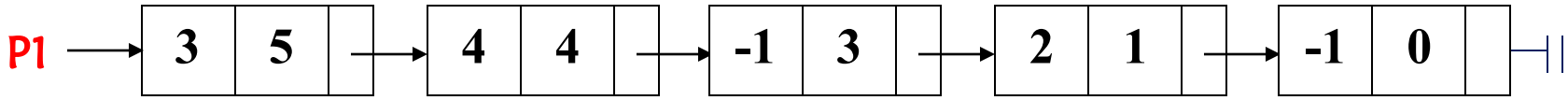
(a, b, c, d, e, f, g, h)

提纲

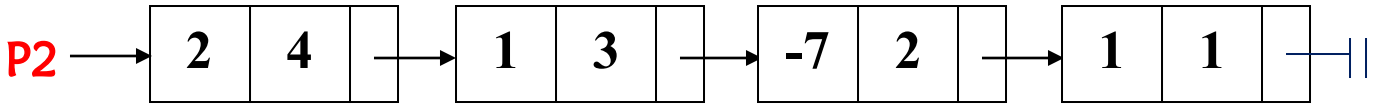
- 2.1 问题引入：一元多项式
- 2.2 线性表的定义与结构
- 2.3 线性表的顺序存储实现
- 2.4 线性表的链式存储实现
- 2.5 线性表的应用
- 2.6 拓展延伸
- 2.7 应用场景：内存管理
- 2.8 小结

2.5.1 一元多项式的加法

- 如何将两个一元多项式相加
  - $P_1(x) = 3x^5 + 4x^4 - x^3 + 2x - 1$



  - $P_2(x) = 2x^4 + x^3 - 7x^2 + x$



2.5.1 一元多项式的加法

- 多项式的链式表示
  - 用数组定义的多项式存储结构不易扩充，在执行多项式运算时可能会增减许多项，采用链表表示将会提高运算的效率。
  - 在多项式的链表表示中每个结点 data 的构成为： Term = { coef, expon }
  - 每个结点实际上有三个域，如图：

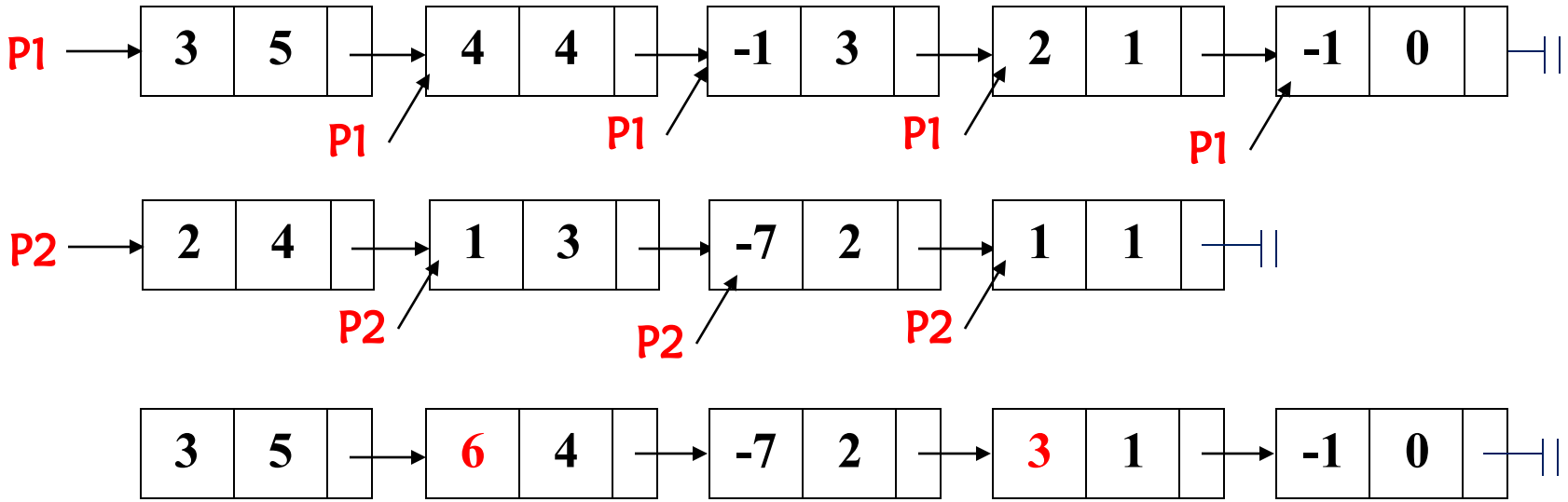

```
typedef double CoefSet; /* 定义多项式系数为double型 */
#define eps (1e-9)
typedef struct PolyNode *Position; /* 指针即结点位置 */
struct PolyNode {
    CoefSet coef; /* 系数 */
    int expon; /* 指数 */
    Position next; /* 线性表中下一个元素的位置 */
};
typedef Position Polynomial;
```

2.5.1 一元多项式的加法

- 链表实现

  p1和p2分别指向两个多项式第一个结点，不断循环比较p1和p2所指的各结点，做不同处理

  - P1.expon==P2.expon，插入求和结点，P1=P1.next；P2=P2.next
  - P1.expon>P2.expon，　插入P1，P1=P1.next;
  - P1.expon<P2.expon，　插入P2，P2=P.next;



2.5.1 一元多项式的加法

- 链表实现
  - 定义节点合并函数

```
Position Attach(CoefSet coef, int expon, Position rear)
{
    rear->next = (Position)malloc(sizeof(struct PolyNode)); /* 在rear后面新建一个结点 */
    rear->next->coef = coef;
    rear->next->expon = expon;
    rear->next->next = NULL;
    rear = rear->next; /* rear指向新的最后一个结点 */
    return rear;
}
```

2.5.1 一元多项式的加法

- 链表实现
  - 创建新链表头节点p

```
Polynomial PolynomialAdd(Polynomial p1, Polynomial p2)
{
        Position rear, temp;
        Polynomial p;
        CoefSet sum;
        p = (Polynomial)malloc(sizeof(struct PolyNode)); /* 创建临时空头结点 */
        rear = p; /* rear指向多项式最后一个结点，初始化指向空头结点 */
        rear->next = NULL;
```

2.5.1 一元多项式的加法

- 链表实现
  - 从大到小遍历数据项指数

```
while ((p1 != NULL) && (p2 != NULL)) {
    if (p1->expon > p2->expon) { /* p1中的数据项指数较大 */
        rear = Attach(p1->coef, p1->expon, rear);
        p1 = p1->next;}
    else if (p1->expon < p2->expon) { /* p2中的数据项指数较大 */
        rear = Attach(p2->coef, p2->expon, rear);
        p2 = p2->next;}
    else { /* 两数据项指数相等 */
        sum = p1->coef + p2->coef;
        if (sum >= eps || sum <= -eps) { /* 如果和不为0 */
            rear = Attach(sum, p1->expon, rear);
        }
        p1 = p1->next;
        p2 = p2->next;
    }
}
```

2.5.1 一元多项式的加法

- 链表实现
  - 处理尚有内容的链表

```
/* 将未处理完的另一个多项式的所有结点依次复制到结果多项式中去 */
while (p1 != NULL) {
        rear = Attach(p1->coef, p1->expon, rear);
        p1 = p1->next;
}
while (p2 != NULL) {
        rear = Attach(p2->coef, p2->expon, rear);
        p2 = p2->next;
}
temp = p;
p = p->next; /* 令p指向结果多项式第一个非零项 */
free(temp); /* 清除临时空头结点 */
return p;
}
```

2.5.2 大整数处理

- **大整数表示**
  - 用顺序表中的元素依次表示该大整数的个位数、十位数、百位数……
  - n = -23456所对应的digits[] = {6, 5, 4, 3, 2}，length = 5，sign = -1

```
#define kMaxSize 1000
#define ErrorCode -1
typedef struct BigIntNode *BigInt;
struct BigIntNode {
        int digits[kMaxSize]; /* 各位数字从低位到高位顺次存储 */
        int length; /* 位数 */
        int sign; /* 正负 */
};
```

2.5.2 大整数处理

- **加法运算：**假设a+b>0,基本过程是：
  - 首先将这两个大整数的位数对齐（位数较少的大整数的高位补0）
  - 将两个大整数对应的位数依次相加或相减，同时处理进位或借位，并将结果存入一个新的大整数c中。
  - 处理加法导致的最高位进位，或减法导致的前导0问题。
  - 思考，a+b<0如何处理?

时间复杂度是O(*n*)

2.5.2 大整数处理

- **加法运算：92789-6321**
  - **初始化及位数对齐: {9, 8, 7, 2, 9}、{1, 2, 3, 6, 0}**

```
BigInt BigIntAdd( BigInt a, BigInt b )
{ /* 注意：此函数只处理 a+b>0 的情况 */
    BigInt c;
    int i, x, y, carry;
    c = (BigInt)malloc(sizeof(struct BigIntNode));
    c->sign = 1; /* 需要保证a+b>0 */
    c->length = max(a->length, b->length); /* 位数对齐 */
    carry = 0; /* 进位/借位值初始化为0 */
```

2.5.2 大整数处理

- 加法运算：92789-6321
  - 遍历及赋符号: {9, 8, 7, 2, 9}、{-1, -2, -3, -6, 0}

```
for (i=0; i<c->length; i++) {
    if (i < a->length) {
        x = a->sign * a->digits[i];
    }
    else {
        x = 0; /* 最高位补0 */
    }
    if (i < b->length) {
        y = b->sign * b->digits[i];
    }
    else {
        y = 0; /* 最高位补0 */
    }
```

2.5.2 大整数处理

- **加法运算**
  - 对位运算：**{9, 8, 7, 2, 9}、{-1, -2, -3, -6, 0} => {8, 6, 4, 6, 8}**

```
        c->digits[i] = x + y + carry; /* 计算第i位加法 */
        if (c->digits[i] >= 10) { /* 处理进位 */
                carry = 1;
                c->digits[i] -= 10;
        }
        else if (c->digits[i] < 0) { /* 处理借位 */
                carry = -1;
                c->digits[i] += 10;
        }
        else {
                carry = 0;
        }
}
```

2.5.2 大整数处理

- **加法运算：**
  - **处理高位进位或前导0**

```
if (carry > 0) { /* 加法导致最高位产生进位 */
        if (c->length == kMaxSize) {
                printf("错误：位数超限。\n");
                c->length = ErrorCode;
        }
        else {
                c->digits[c->length] = carry;
                c->length++;
        }
}
/* 消除减法导致高位出现的前导0 */
while ((c->length > 0) && (c->digits[c->length-1]==0)) {
   c->length--;
}
return c;
```

2.5.2 大整数处理

- 乘法运算：两个正的大整数a和b相乘
  - 用i和j的二重循环分别枚举大整数a和b的每一位（数组下标从0开始）
  - 对于a的第i位数字和b的第j位数字（按从低位到高位的顺序），将它们相乘并累加至表示计算结果的大整数c的第i+j位上。
  - 对于得到的大整数c，从最低位到最高位依次处理进位问题，得到最终的计算结果。

时间复杂度是$O(n^2)$

5678×1234

8	7	6	5

4	3	2	1

			7*2+…				

2.5.2 大整数处理

- 乘法运算：两个正的大整数a和b相乘

```
BigInt BigIntMultiply( BigInt a, BigInt b )
{
        BigInt c;
        int i, j, carry, temp;
        c = (BigInt)malloc(sizeof(struct BigIntNode));
        if ((a->length == 0) || (b->length == 0)) { /* 处理结果为0的特殊情况 */
                c->sign = 1;
                c->length = 0;
        }
```

2.5.2 大整数处理

- 乘法运算：两个正的大整数a和b相乘
  - 分配空间及0值情况处理

```
BigInt BigIntMultiply( BigInt a, BigInt b )
{
        BigInt c;
        int i, j, carry, temp;
        c = (BigInt)malloc(sizeof(struct BigIntNode));
        if ((a->length == 0) || (b->length == 0)) { /* 处理结果为0的特殊情况 */
                c->sign = 1;
                c->length = 0;
        }
```

2.5.2 大整数处理

- 乘法运算：两个正的大整数a和b相乘
  - 判读符号及初始化

```
else { /* a和b均不为0 */
        if (a->sign == b->sign) { /* 判断结果的符号位 */
                c->sign = 1;
        }
        else {
                c->sign = -1;
        }
        c->length = a->length + b->length - 1; /* 确定结果的位数 */
        if (c->length > kMaxSize) {
                printf("错误：位数超限。\n");
                c->length = ErrorCode;
                return c;
        }
        for (i=0; i<c->length; i++) { /* 初始化c */
                c->digits[i] = 0;
        }
```

2.5.2 大整数处理

- 乘法运算：两个正的大整数a和b相乘
  - 计算并累加结果

```
for (i=0; i<a->length; i++) { /* 计算并累加结果 */
        for (j=0; j<b->length; j++) {
                c->digits[i+j] += (a->digits[i] * b->digits[j]);
        }
}
```

2.5.2 大整数处理

- 乘法运算：两个正的大整数a和b相乘
  - 处理进位（保证每个元素值处于0-10整数）

```
carry = 0; /* 初始化进位值 */
for (i=0; i<c->length; i++) { /* 从最低位到最高位依次处理进位问题 */
        temp = c->digits[i] + carry;
        c->digits[i] = temp % 10;
        carry = temp / 10;
}
```

2.5.2 大整数处理

- 乘法运算：两个正的大整数a和b相乘
  - 最高位进位处理

```
if (carry > 0) { /* 最高位产生进位 */
        if (c->length == kMaxSize) {
                printf("错误：位数超限。\n");
                c->length = ErrorCode;
        }
        else {
                c->digits[c->length] = carry;
                c->length++;
        }
}
```

提纲

- 2.1 问题引入：一元多项式
- 2.2 线性表的定义与结构
- 2.3 线性表的顺序存储实现
- 2.4 线性表的链式存储实现
- 2.5 线性表的应用
- 2.6 拓展延伸
- 2.7 应用场景：内存管理
- 2.8 小结

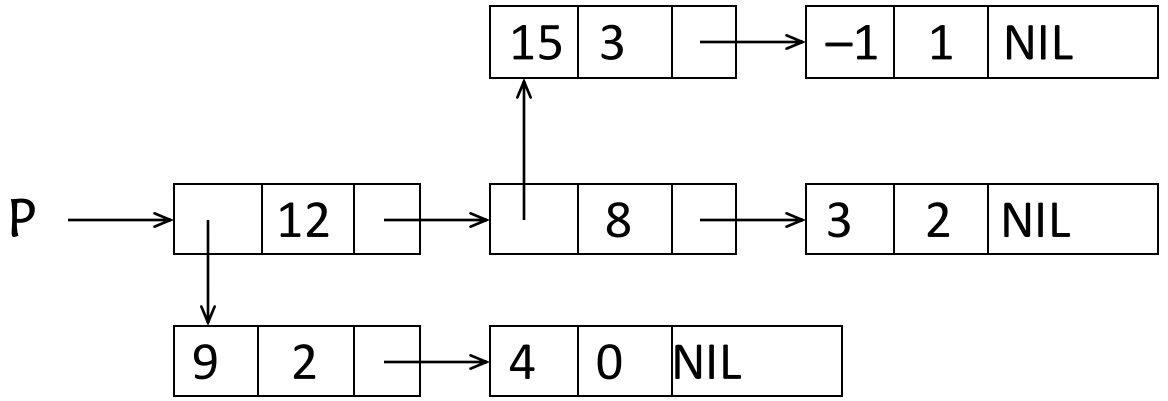
2.6.1 广义表

- 广义表
  - 广义表是线性表的推广
  - 对于线性表而言， n个元素都是基本的单元素；
  - 广义表中，这些元素不仅可以是单元素也可以是另一个广义表

tag	data	next
	sub_list	

- 例如：二元多项式的表示

$$P(x,y) = 9x^{12}y^2 + 4x^{12} + 15x^8y^3 - x^8y + 3x^2 \longrightarrow P(x,y) = (9y^2+4)x^{12} + (15y^3 - y)x^8 + 3x^2$$



2.6.1 广义表

- 广义表的特性
  - 有次序性：元素按顺序排列，位置不同则广义表不同。
  - 有长度：广义表中**直接元素**（原子或子表）的个数。
  - 有深度：最大嵌套层数。原子深度为0，子表深度为“所有元素的深度的最大值＋1”。
  - 可递归：广义表可
  - 可共享：多个广义

A( )	A长度为0，深度为1
B(6, 2)	B长度为2，深度为1
C(‘*a*’, (5, 3, ‘*x*’))	C长度为2，深度为2
D(B, C, A)	D长度为3，深度为3
E(B, D)	E长度为? 深度为?
F(4, F)	F长度为? 深度为?

2.6.2 多维数组和特殊矩阵

1. 多维数组的顺序存储：按照行优先的顺序存放，即先存放第0行的元素，再存放第1行的元素，……，其中每一行中的元素再按照列的顺序存放。

例如：二维整形数组 a[3][4]的存放顺序：

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[2][0]	a[2][1]	a[2][2]	a[2][3]

一般来说，二维数组元素$a[i][j]$的存储位置（地址）l_{ij}计算方法是：

设n维数组各维大小是（s_1，s_2，……，s_n），第一个元素的地址是$l_{(0,0,\ldots,0)}$，元素占用空间为$size$个字节，则下标为（i_1,i_2，...,i_n）的元素位置是：

$$l_{(i1,i2,\ldots in)} = l_{(0,0,\ldots,0)} + (i_1 \times s_2 \times s_3 \times \ldots \times s_n + i_2 \times s_3 \times \ldots \times s_n + \ldots + i_{n-1} \times s_n + i_n) \times size$$

2.6.2 多维数组和特殊矩阵

- 特殊矩阵：上三角矩阵和下三角矩阵

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix} \qquad \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

压缩空间存储：如果只存储上三角（或者下三角）元素，则将近减少了一半的存储空间

对于下三角矩阵，设单个元素所占空间为*size*，则$a[i][j]$的存储位置l_{ij}与矩阵首个元素$a[0][0]$的地址l_{00}的关系是：

$l_{ij} = l_{00} + (i(i+1)/2+j) \times size$，$i \geq j$

2.6.2 多维数组和特殊矩阵

- **稀疏矩阵**
  - 数值为0的元素数目远远多于非0元素的数目，并且非0元素分布没有规律
  - 存储：非0项

1) 三元组表的顺序存储

row	0	0	1	2	3	3	3
col	0	3	1	3	0	1	4
value	18	2	27	-4	23	-1	12
	0	1	2	3	4	5	6

2) 三元组表的链式存储：十字链表

采用一种典型的多重链表——十字链表来存储稀疏矩阵

- 只存储矩阵非0元素项

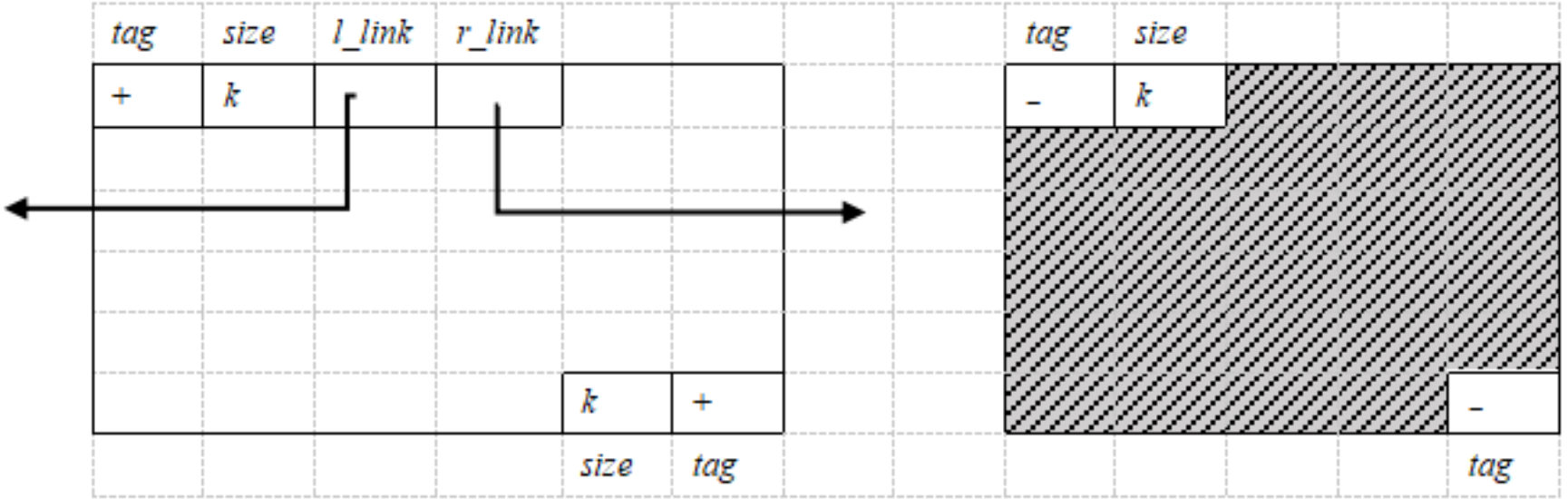
  结点的数据域：行坐标Row、列坐标Col、数值Value
- 每个结点通过两个指针域，把同行、同列串起来;
  - 行指针(或称为向右指针)Right
  - 列指针（或称为向下指针）Down

提纲

- 2.1 问题引入：一元多项式
- 2.2 线性表的定义与结构
- 2.3 线性表的顺序存储实现
- 2.4 线性表的链式存储实现
- 2.5 线性表的应用
- 2.6 拓展延伸
- 2.7 应用场景：内存管理
- 2.8 小结

2.7 应用场景：内存管理

- 操作系统空闲内存管理：空闲块的管理，处理申请和回收；
- 内存空闲块组织成双向链表
- 空闲块分配策略：首次适配、最佳适配
- 空闲块回收：空闲块合并



(a) 空闲块结构　　（b）已分配块结构

提纲

- 2.1 问题引入：一元多项式
- 2.2 线性表的定义与结构
- 2.3 线性表的顺序存储实现
- 2.4 线性表的链式存储实现
- 2.5 线性表的应用
- 2.6 拓展延伸
- 2.7 应用场景：内存管理
- 2.8 小结

2.8 小结

- 线性表的逻辑结构特性是指数据元素之间存在着线性关系
- 顺序存储结构（顺序表）和链式存储结构（链表）在逻辑上都是线性结构
- **顺序表**在的元素存储直接反应了其逻辑上的线性关系，可借助于数组表示；**链表**依靠指针来反映其线性逻辑关系
- 顺序表是随机存取（access）结构，链表是顺序存取结构
- 关于顺序表的三段式命题结构：
  - 给出算法的设计思想
  - 采用**类**C或C++语言描述算法，并给出注释
  - 分析所设计算法的时间复杂度和空间复杂度

2.8 小结

- 顺序表与链表基于空间的比较
  - 存储分配的方式
    - 顺序表的存储空间可以是静态分配的，也可以是动态分配的。
    - 链表的存储空间是动态分配的。
  - 存储密度 = 结点数据本身所占的存储量/结点结构所占的存储总量
    - 顺序表的存储密度 = 1
    - 链表的存储密度 < 1

2.8 小结

- 顺序表与链表基于时间的比较
  - 存取方式
    - 顺序表可以随机存取，也可以顺序存取。
    - 链表只能顺序存取。
  - 插入/删除时移动元素个数
    - 顺序表平均需要移动近一半元素。
    - 链表不需要移动元素，只需要修改指针。
    - 若插入/删除仅发生在表的两端，宜采用带尾指针的循环链表。

2.8 小结

		顺序表	链表
空间	存储空间	预先分配，会导致空间闲置或溢出	动态分配，不会出现空间闲置或溢出
	逻辑结构与物理结构	逻辑上相邻的元素，物理上也相邻	逻辑上相邻的元素，物理上不一定相邻，通过指针进行连接。
	存储密度	存储密度高（=1），不需要额外空间来表示节点间的逻辑关系	存储密度低（<1），需要借助指针来表示元素间的逻辑关系
时间	存取方式	随机存取。按值查找，列表无序 T=O(n)，列表有序 T=O(log2n)；按位置查找，T=O(1)	顺序存取。按位置查找，也需要顺序访问元素，T=O(n)
	插入、删除	平均移动半个表长的元素，T=O(n)	不需移动元素，确定插入、删除位置后，T=O(1)。但查找位置需顺序遍历链表。
适用情况		① 表长固定 ② 很少进行插入或删除操作，经常按元素位置序号访问数据元素	① 表长变化较大 ③ 稀疏序列 ② 需要频繁进行插入或删除操作

2.8 小结

链表名称	操作名称		
	查找表头结点	查找表尾结点	查找结点*p的前驱结点
带头结点的单链表L	L -> next 时间复杂度O(1)	从L -> next依次向后遍历 时间复杂度O(n)	通过 p -> next 无法找到其前驱
带头结点仅设头指针L的循环单链表	L -> next 时间复杂度O(1)	从L -> next依次向后遍历 时间复杂度O(n)	通过 p -> next 可以找到其前驱 时间复杂度O(n)
带头结点仅设尾指针R的循环单链表	R -> next 时间复杂度O(1)	R 时间复杂度O(1)	通过 p -> next 可以找到其前驱 时间复杂度O(n)
带头结点的双向循环链表L	L -> next 时间复杂度O(1)	L -> prior 时间复杂度O(n)	L -> prior 时间复杂度O(1)

单选题 1分

设置

以下哪项描述了线性表的数据对象？

A 一个无限序列的数据元素集合

B 一个有限序列的数据元素集合

C 一个无序的数据元素集合

D 一个固定大小的数据元素集合

提交

多选题 1分

设置

以下描述，哪些属于线性表的特点。

A 具有无穷多个元素

B 同一个线性表中，所有元素都具有相同的数据类型

C 所有的线性表都采用顺序表进行存储

D 线性表中的元素称为数据项

E 线性表中的元素之间具有先后顺序

提交

单选题 1分

设置

线性表的顺序存储实现通常使用以下哪种数据结构?

A 链表

B 数组

C 哈希表

D 树

提交

单选题 1分

设置

线性表的顺序存储结构是一种（ ）

A 随机存取的存储结构

B 顺序存取的存储结构

C 索引存取的存储结构

D 散列存取的存储结构

提交

多选题 1分

对于多项式 $f(x) = 4x^{7000} - 2x^{500} + 1$，我们可以使用（）方式进行表示。

对于多项式 $f(x) = 4x^{7000} - 2x^{500} + 1$，我们可以使用（）方式进行表示。

A 顺序存储

B 逆序存储

C 链式存储

D 以上都不行

提交

单选题 1分

设置

线性表的特点是每个元素都有一个前驱和一个后继。

A 正确

B 错误

提交

单选题 1分

设置

对于多项式 $f(x) = 3x^{5000} - 2x^{500} + 1$，如果我们使用顺序存储结构的直接表示法，数组大小需要多少？

A 3

B 500

C 5000

D 5001

提交

多选题 1分

设置

顺序表进行插入和删除操作时，以下哪些说法是正确的？

A 插入操作的时间复杂度为O(n)

B 删除操作的时间复杂度为O(n)

C 插入和删除操作都需要移动元素

D 插入和删除操作的时间复杂度相同

提交

单选题 1分

设置

1.一个线性表第一个元素的存储地址是100,每个元素的长度为2,则第5个元素的地址是( )答案

A 110

B 108

C 100

D 120

提交

单选题 1分

设置

3. 向一个有127个元素的顺序表中插入一个新元素并保持原来顺序不变,平均要移动( )个元素。答案

A 64

B 63

C 63.5

D 7

提交

单选题 1分

设置

5. 在一个单链表中,若p所指结点不是最后结点,在p之后插入s所指结点,则执行( )答案

A s->next=p;p->next=s;

B s->next=p->next;p->next=s;

C s->next=p->next;p=s;

D p->next=s;s->next=p;

提交

单选题 1分

设置

6.在一个单链表中,若删除p所指结点的后续结点,则执行( )答案

A p->next=p->next->next;

B p=p->next; p->next=p->next->next;

C p->next=p->next;

D p =p->next->next;

提交

单选题 1分

设置

线性表采用链式存储结构所具有的特点是

A 所需空间地址必须不连

B 需要事先估计所需存储空间

C 可随机存取

D 插入、删除操作不必移动元素

提交

单选题 1分

设置

关于线性表的顺序存储结构和链式存储结构的描述中，正确的是（ ）。

Ⅰ.线性表的顺序存储结构优于链式存储结构

Ⅱ.顺序存储结构比链式存储结构的存储密度高

Ⅲ.如需要频繁插入和删除元素，最好采用顺序存储结构

Ⅳ.如需要频繁插入和删除元素，最好采用链式存储结构

A Ⅲ

B Ⅱ、Ⅳ

C Ⅱ、Ⅲ

D Ⅲ、Ⅳ

提交

单选题 1分

设置

某算法在含有n（n≥1）个节点的单链表中查找值为x节点，其时间复杂度是

A $O(\log_2 n)$

B $O(1)$

C $O(n^2)$

D $O(n)$

提交

单选题 1分

设置

通过含有n（n ≥ 1）个元素的数组a，采用头插法建立一个单链表L，则 L 中节点值的次序（ ）。

A 与数组a的元素次序相同

B 与数组a的元素次序相反

C 与数组a的元素次序无关

D 以上都不对

提交

单选题 1分

设置

下面关于线性表的一些说法中，正确的是（）。

A 对一个设有头指针和尾指针的单链表执行删除最后一个元素的操作与链表长度无关

B 线性表中每个元素都有一个直接前驱和一个直接后继

C 为了方便插入和删除数据，可以使用双链表存放数据

D 取线性表第 i 个元素的时间与 i 的大小有关

提交

单选题 1分

设置

在双链表中向**p**所指的结点之前插入一个结点**q**的操作为（）。

A p->prior = q; q->next = p; p->prior->next = q; q->prior = p->prior;

B q->prior = p->prior; p->prior->next = q; q->next = p; p->prior = q->next;

C q->next = p; p->next = q; q->prior->next = q; q->next = p;

D p->prior ->next = q; q->next = p; q->prior = p->prior; p->prior = q;

提交

单选题 1分

设置

【2016统考真题】已知一个带有表头结点的双向循环链表L，节点结构为[prev|data|next]，其中prev和next分别指向其直接前驱和直接后继结点的指针，先要删除指针p所指的结点，正确的语句是：（）。

A p->next->prev = p->prev; p->prev->next = p->prev; free(p);

B p->next->prev = p->next; p->prev->next = p->next; free(p);

C p->next->prev = p->next; p->prev->next = p->prev; free(p);

D p->next->prev = p->prev; p->prev->next = p->next; free(p);

提交

单选题 1分

设置

【2021统考真题】已知头指针 h 指向一个带头结点的非空单循环链表，结点结构为 [data|next]，其中next是指向直接后继结点的指针，p是尾指针，q是临时指针。现要删除该链表的第一个元素，正确的语句序列是（）。

A h->next = h->next->next; q = h->next; free(q);

B q = h->next; h->next = h->next->next; free(q);

C q = h->next; h->next = q->next; if(p!=q) p = h; free(q);

D q = h->next; h->next = q->next; if(p==q) p = h; free(q);

提交