线性表扩展及应用

学习目标

- 熟知双（向）链表存储和相关操作
- 掌握循环链表存储和相关操作
- 了解线性表的应用

双链表

循环链表

双链表

双链表的引入

从结点 p 出发，如何求得结点 p 的直接前驱?

如何快速求得结点 p 的前驱?

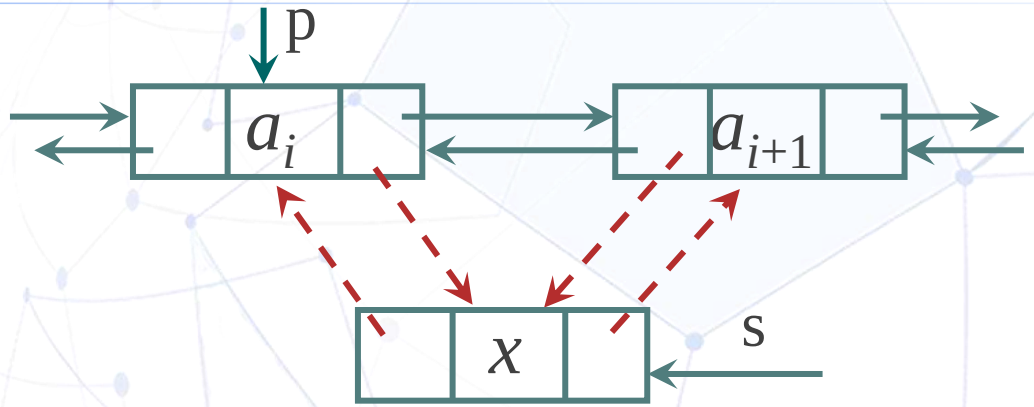
双链表：单链表的每个结点再设置一个指向其前驱结点的指针域

双链表的存储结构定义

启示? ⇨ 时空权衡——空间换取时间 ⇨ 数据表示

```
template <typename DataType>
sruct DulNode
{
    DataType data;
    DulNode< DataType> *prior, *next;
};
```

双链表的操作——插入

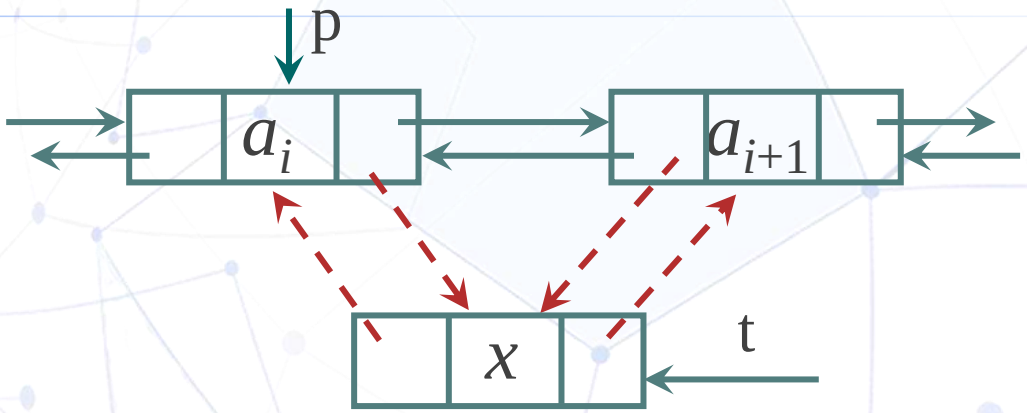



```
s = new DulNode;
s->prior = p;
s->next = p->next;
p->next->prior = s;
p->next = s;
```

注意指针修改的**相对**顺序

双链表的操作——插入

也可以按这种次序插入



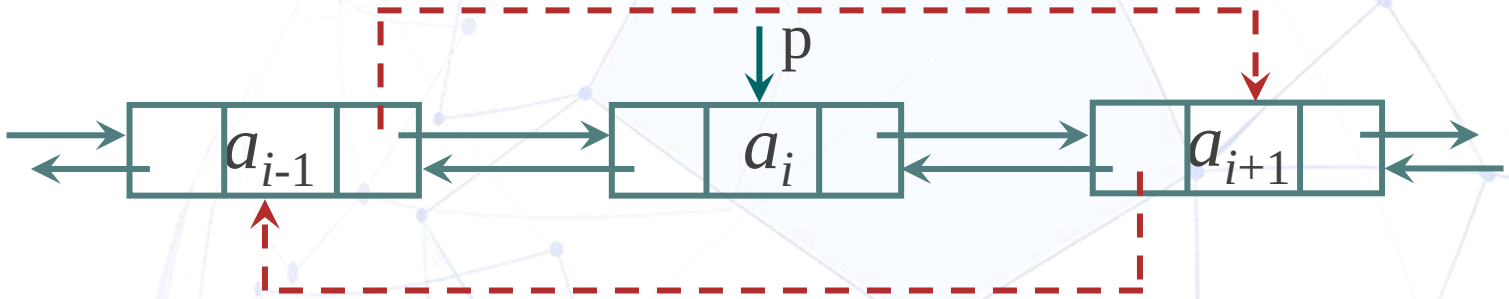
代码 2-3 在双向链表中 *p* 所指结点后面插入 *t* 所指的结点 DLLInsert(*p*, *t*)

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
```

双链表的操作——删除



(p->prior)->next = p->next;

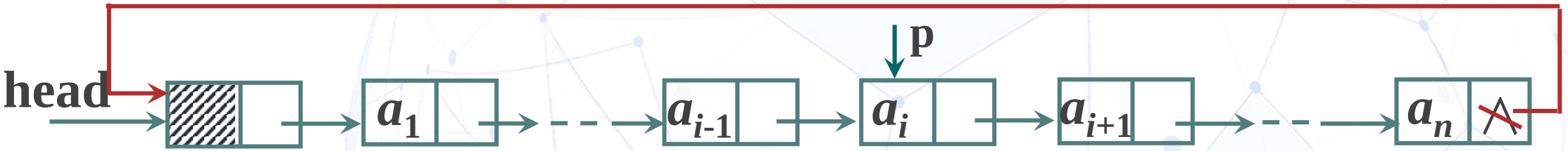
(p->next)->prior = p->prior;

结点 p 的指针是否需要修改? ⇨ delete(p);

循环链表

循环链表的引入

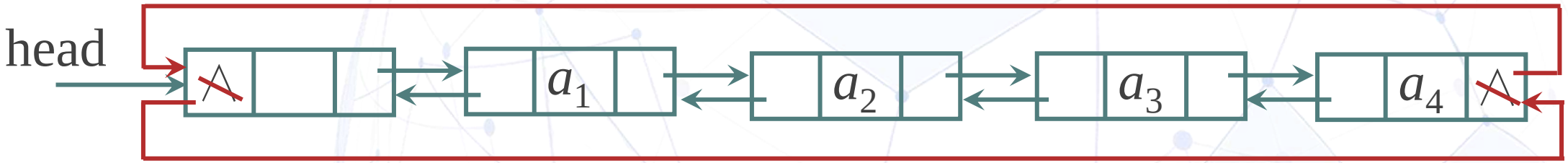
在单链表中，从结点 p 出发通过指针后移，能够求得结点 p 的直接前驱吗?



- 如何求得结点 p 的前驱?
- 循环链表：将链表的首尾相接
- 循环单链表：将终端结点的指针由空指针改为指向头结点

循环链表的引入

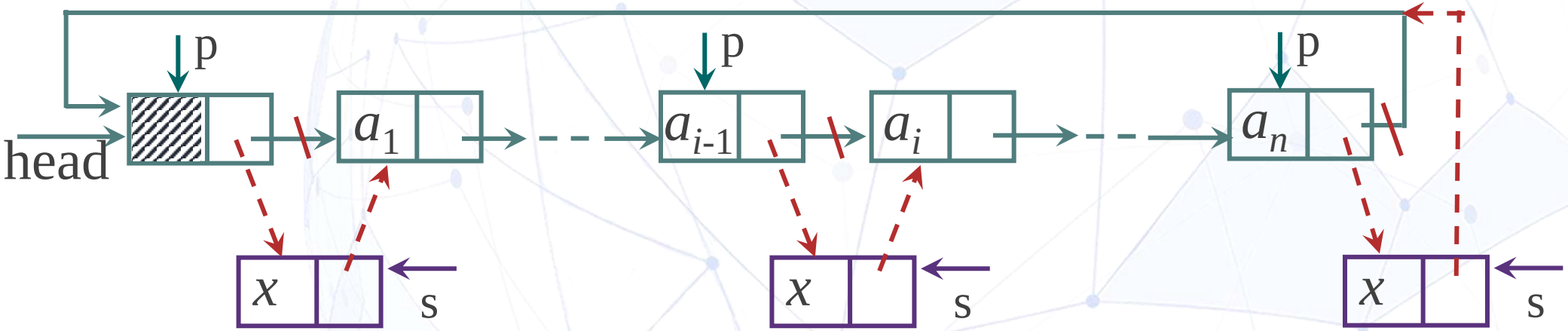
双链表可以循环吗?



- 循环链表：将链表的首尾相接
- 循环双链表：将终端结点的next指针由空指针改为指向头结点，将头结点的prior指针由空指针改为指向终端结点

循环单链表的实现

循环单链表的插入操作——表头、表中间、表尾?




```
s = new Node;
s->data = x;
s->next = p->next;
p->next = s;
```

循环单链表的实现

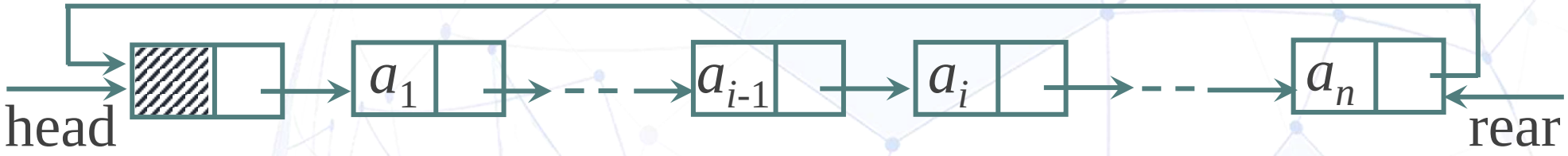
p->next != nullptr ➔ p->next != first

与单链表插入操作的区别?

```
int Insert(int i, DataType x)
{
    Node *p = first;   int count = 0;
    while (p->next!= first && count < i - 1)
    {
        p = p->next;
        count++;
    }
    if (p == nullptr)  throw“位置错误，插入失败”;
    else {
        s = new Node;   s->data = x;
        s->next = p->next; p->next = s;
   }
}
```

循环单链表的变化

如何查找开始结点和终端结点?



开始结点：head->next

终端结点：$O(n)$

开始结点：rear->next->next

终端结点：rear

循环单链表由头指针指示 OR 由**尾指针指示**?

存储结构是否合理，取决于运算是否方便，时间性能是否提高

扩展

静态链表

顺序表的动态分配

静态链表

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（即它在数组中的下标）

- 物理存储空间上依赖于数组
- 元素逻辑链接关系是采用了链表的思想

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）

```
template <typename DataType>
struct  SNode
{    DataType      data;    //数据域
     int next;     //指针域（也称为游标），注意不是
};
```

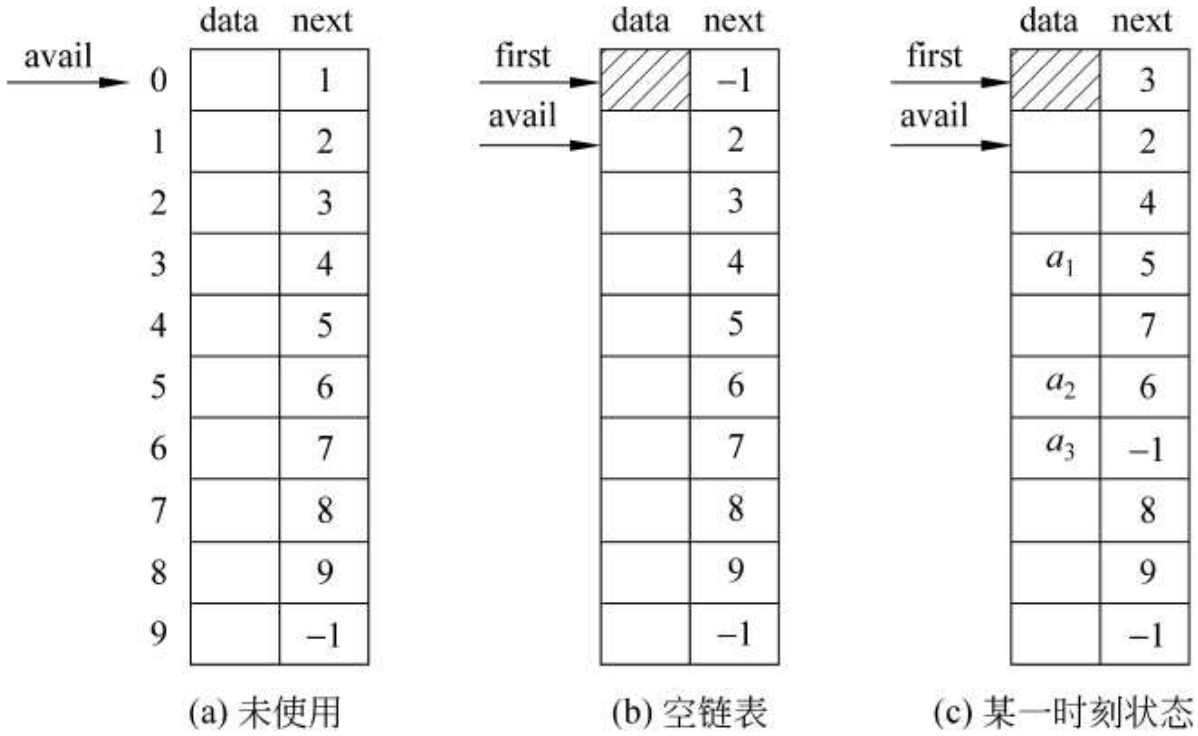


图 2-29　静态链表的存储示意图

```
const int MaxSize = 100;
template <typename DataType>
class StaList
{
     public:
       [methods]
     private:
       SNode SList[MaxSize]; //静态链表数组
       int first，avail；  //游标，头指针和空闲头指针

};
```

顺序表的动态存储

顺序表的静态存储，由于存储空间不能扩充，一旦数组空间占满，再执行插入操作就会发生上溢出；顺序表的**动态分配方式**是在程序执行过程中通过动态存储分配，一旦数组空间占满，另外再分配一块更大空间，用来替换原来的存储空间。

```
const int MaxSize = 100;
template <typename DataType>
class SeqList
{
    public:
      [methods]
    private:
      DataType data[MaxSize];
      int last;
};
```

```
const int InitSize = 100; //顺序表的初始长度
const int IncreSize=10; //每次扩展长度
template <typename DataType>
class SeqList
{
    public:
      SeqList();
    private:
      DataType *data;  //动态申请数组空间的首地址
      int maxSize;  //当前数组的最大长度
      int last;          //线性表最后一个元素位置
};
```

```
const int InitSize = 100;
const int IncreSize=10;
template <typename DataType>
class SeqList
{
      public:
        SeqList();
      private:
        DataType *data;
        int maxSize;
        int last;
};
```

```
template <typename DataType>
SeqList< DataType>::SeqList()
{
     data=new DataType[InitSize];
     maxSize=InitSize;
     last=0;
};
```

```
template <typename DataType>
SeqList< DataType>::insert(int i, DataType x)
{
     if (i<1 || i>last+1) throw("插入位置异常");
     if(last+1==maxSize)  {
          //发生上溢出，扩充存储空间
          DataType *oldData=data;
          maxSize+=IncreSize;
          data=new DataType[maxSize];
          for(int j=0; j<=last;j++)
                    data[j]=oldData[j];
           delete[] oldData;
     }

      for(int j=last;  j>=i; j--)
         data[j]=data[j-1];
     data[i-1]=x;
     last++;
};
```

线性表的应用

一元多项式运算

一元多项式及其运算

- 一元多项式：$f(x)=a_0+a_1x+\cdots+a_{n-1}x^{n-1}+a_nx^n$
- 主要运算：多项式相加、相减、相乘等

$$
\begin{aligned}
P1 &= 3X^5+4X^4-X^3+2X-1\\
+\ P2 &= 2X^4+X^3-7X^2+X\\
\hline
P &= 3X^5+6X^4-7X^2+3X-1
\end{aligned}
$$

主要思路：相同指数的项系数相加，其余部分进行拷贝。

方法1：采用顺序存储结构直接表示一元多项式

用数组a存储多项式的相关数据：数组分量a[i]表示项x^i的系数

$$P1 = 3X^5 + 4X^4 - X^3 + 2X - 1$$

$$P2 = 2X^4 + X^3 - 7X^2 + X$$

$a[i]$	-1	2	0	-1	4	3	……
下标i	0	1	2	3	4	5	……

$a[i]$	0	1	-7	1	2	0	……
下标i	0	1	2	3	4	5	……

优点：每项的指数隐藏在其系数所在的数组下标

多项式相加容易

稀疏多项式?

方法2：采用顺序存储结构表示多项式的非零项

- 每个非零项 $a_i x^i$ 涉及两个信息：指数i和系数a_i
- 可以将一个多项式看成是一个(a_i，i)二元组的集合。

$$\{(a_n，\ n)，(a_{n-1}，\ n-1)，\ \ldots\ldots，(a_0，\ 0)\}$$

$$P_1(x)=9x^{12}+15x^8+3x^2$$

系数	9	15	3	-
指数	12	8	2	-
数组下标i	0	1	2	……

$$P_2(x)=26x^{19}-4x^8-13x^6+82$$

系数	26	-4	-13	82	-
指数	19	8	6	0	-
数组下标i	0	1	2	3	……

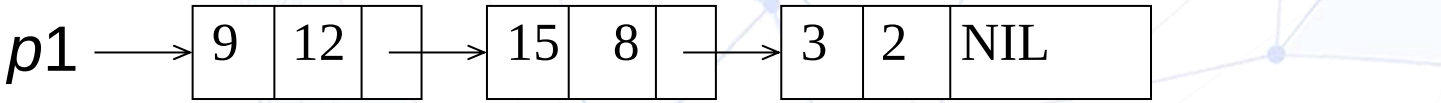
- 克服了稀疏，但是操作不方便，比如在计算时遇到合并系数为0时，需要移动数据

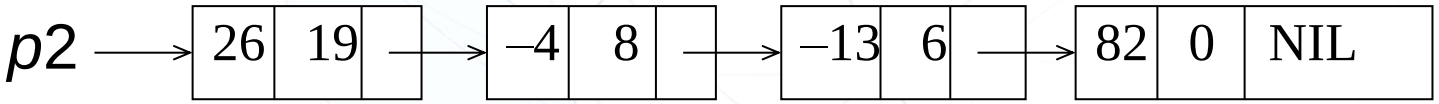
方法3：采用链表结构来存储多项式的非零项

链表表示多项式：链表结点对应一个非零项，包括：系数、指数、指针域

$P_1(x) = 9x^{12} + 15x^8 + 3x^2$

$P_2(x) = 26x^{19} - 4x^8 - 13x^6 + 82$

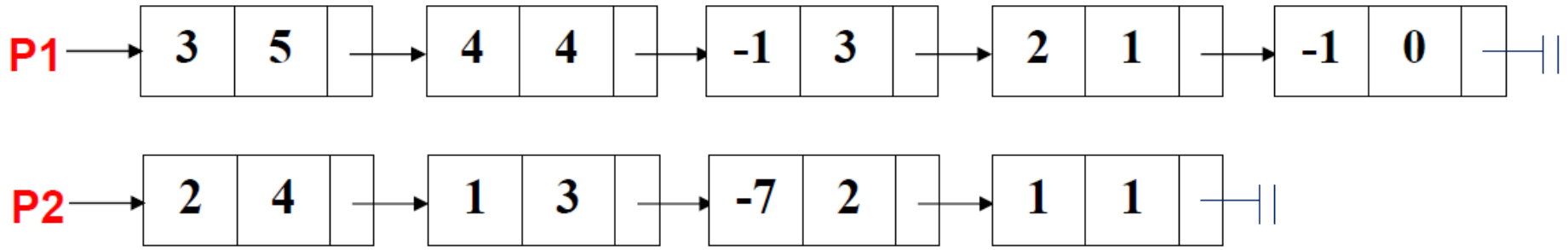




一元多项式加法

$$P1 = 3X^5 + 4X^4 - X^3 + 2X - 1$$
$$P2 = 2X^4 + X^3 - 7X^2 + X$$

采用不带头结点的单向链表，按照指数递减的顺序排列各项

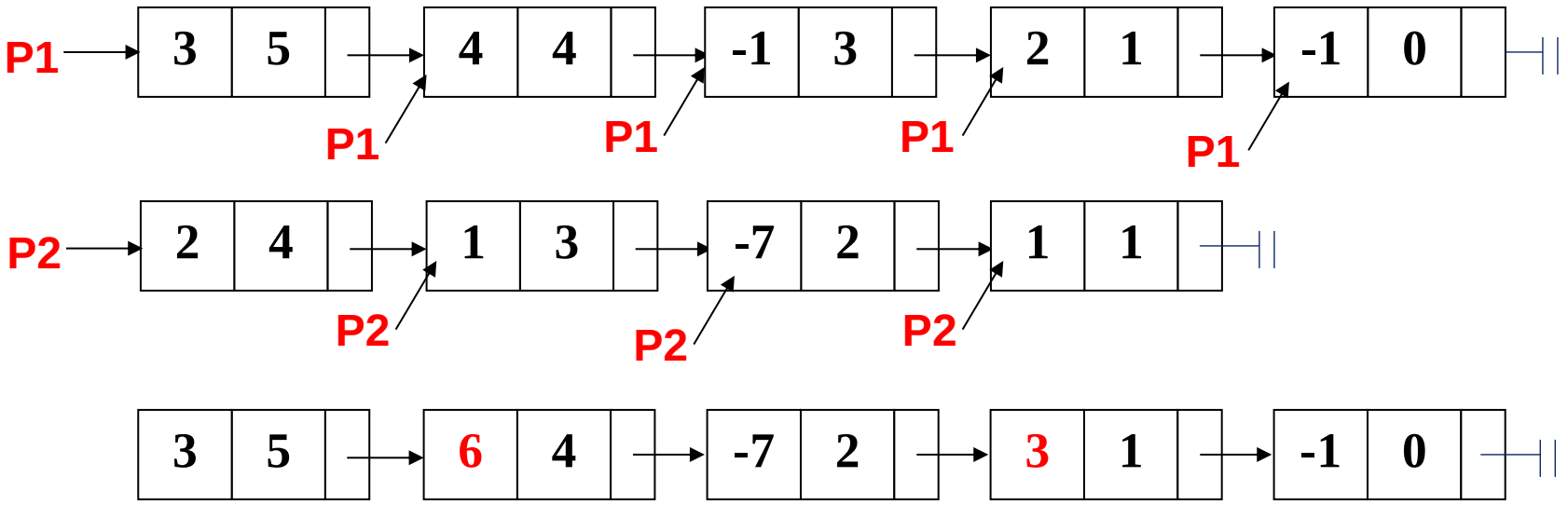



```
struct PolyNode {
    int coef; // 系数
    int expon; // 指数
    PolyNode *next; // 指向下一个节点的指针
};
PolyNode *Polynomial;
Polynomial P1, P2;
```

一元多项式的加法

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

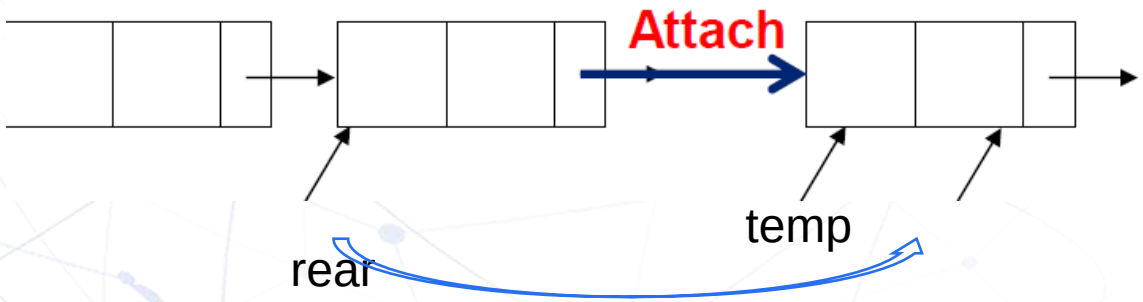
- P1.expon==P2.expon，插入求和结点，P1=P1.next；P2=P2.next
- P1.expon>P2.expon， 插入P1，P1=P1.next;
- P1.expon<P2.expon， 插入P2，P2=P.next;
- 当某一多项式处理完时，将另一个多项式的所有结点依次复制到结果多项式中去



```
void Polynomial::add(Polynomial B,
Polynomial *C)
{
    PolyNode *front, *rear, *temp;

    front = C->first;
    rear = front;//尾插法建立单链表
    int sum;

    PolyNode *p1 = this->first->next;
    PolyNode *p2 = B.first->next;

    while (p1 != nullptr && p2 != nullptr)
    {
        if (p1->expon > p2->expon) {
            //P1 attach_To_C
            p1 = p1->next;  //第1种情况
        }
        else if (p1->expon < p2->expon) {
            //P2 attach_To_C
            p2 = p2->next;  //第2种情况
        }
        else
        {
            sum = p1->coef + p2->coef; //第3种情况
            if (sum != 0) {
                    //sum  attach_To_C
                }
            p1 = p1->next;
            p2 = p2->next;
        }
    }
    while (p1) {//remaining P1  attach_To_C }
    while (p2) {//remaining P2 attach_To_C }
}
```

attach_To_C

```
temp = new PolyNode;
temp->coef = p1->coef;
temp->expon = p1->expon;
temp->next = nullptr;
rear->next = temp;
rear = temp;
```



课堂小结

线性表的链式存储特定

链式存储的常用操作