



Data Structures

线性表 Linear Lists

2024年9月13日

学而不厌 诲人不倦

- ➡ 2.1 引言
- ➡ 2.2 线性表的逻辑结构
- ➡ 2.3 线性表的顺序存储结构及实现
- ➡ 2.4 线性表的链接存储结构及实现
- ➡ 2.5 顺序表和链表的比较
- ➡ 2.6 约瑟夫环与一元多项式求和

第二章 线性表

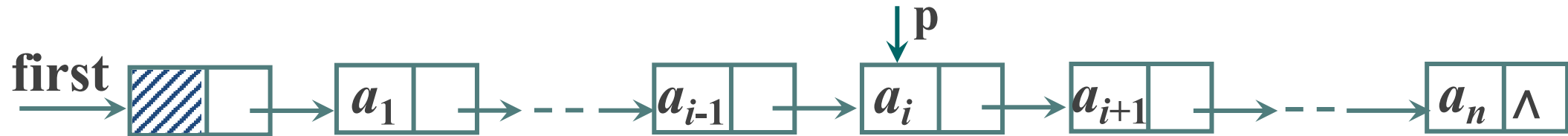
2.4 线性表的链接存储结构及实现

2-4-4 双链表

2.4 线性表的链接存储结构及实现

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



2.4 线性表的链接存储结构及实现

双链表的存储结构定义



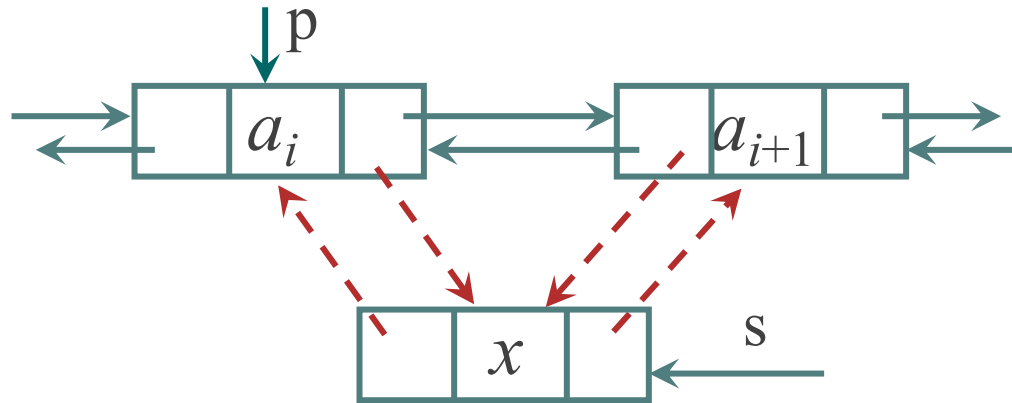
启示? \Rightarrow 时空权衡——空间换取时间 \Rightarrow 数据表示

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



2.4 线性表的链接存储结构及实现

双链表的操作——插入



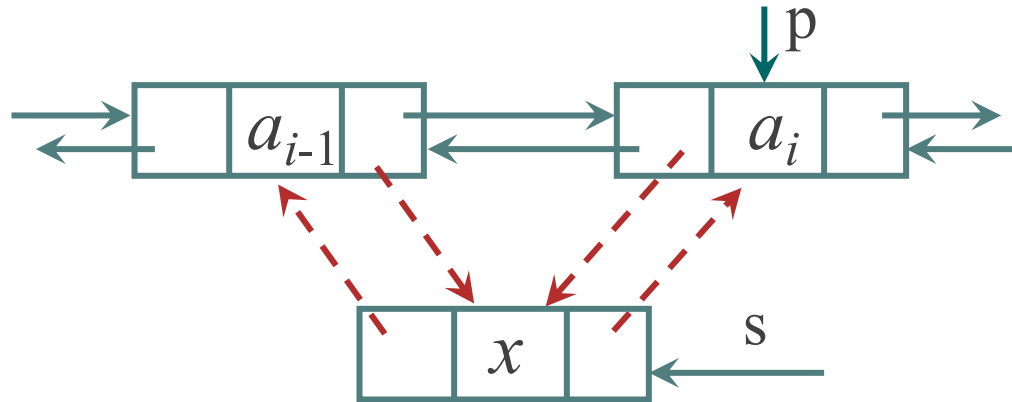
```
s = new DulNode;  
s->prior = p;  
s->next = p->next;  
p->next->prior = s;  
p->next = s;
```



注意指针修改的**相对**顺序

2.4 线性表的链接存储结构及实现

双链表的操作——插入



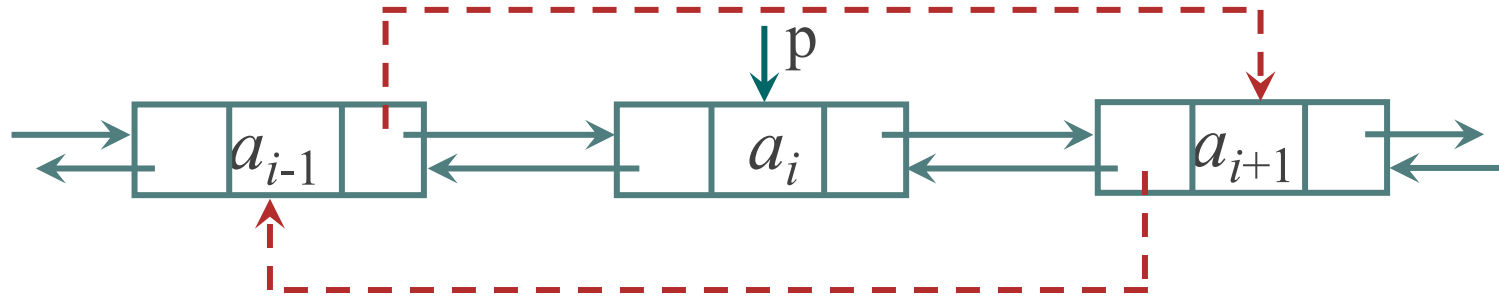
```
s = new DulNode;  
s->prior = p->prior;  
s->next = p;  
p->prior->next = s;  
p->prior = s;
```



双链表的操作更加**灵活**

2.4 线性表的链接存储结构及实现

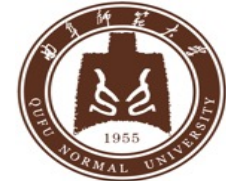
双链表的操作——删除



$(p \rightarrow \text{prior}) \rightarrow \text{next} = p \rightarrow \text{next};$

$(p \rightarrow \text{next}) \rightarrow \text{prior} = p \rightarrow \text{prior};$

🕒 结点 p 的指针是否需要修改? \Rightarrow delete(p);



第二章 线性表

2.4 线性表的链接存储结构及实现

2-4-5 循环链表

2.4 线性表的链接存储结构及实现

循环链表的引入

🕒 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？

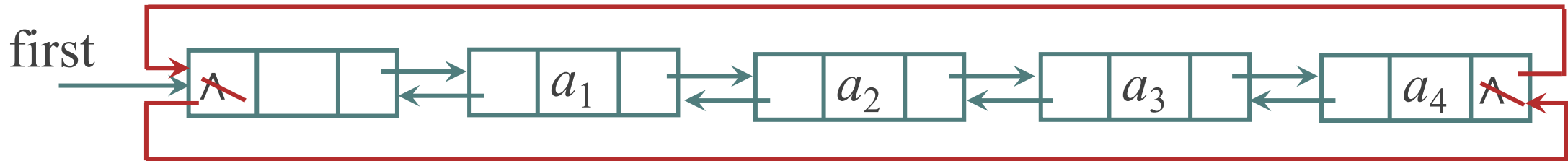
📌 循环链表：将链表的首尾相接

📌 循环单链表：将终端结点的指针由空指针改为指向头结点

2.4 线性表的链接存储结构及实现

循环链表的引入

🕒 双链表可以循环吗？




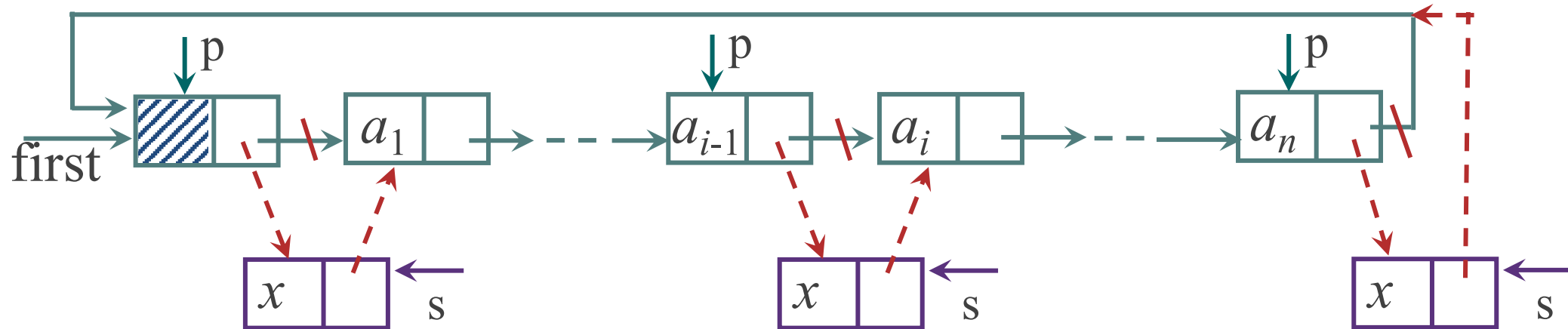
📌 **循环链表**：将链表的首尾相接

📌 **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

2.4 线性表的链接存储结构及实现

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;
```

2.4 线性表的链接存储结构及实现

循环单链表的实现

```
int Insert(int i, DataType x)
{
    Node *p = first; int count = 0;
    while (p != 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;
    }
}
```



与单链表插入操作的区别?

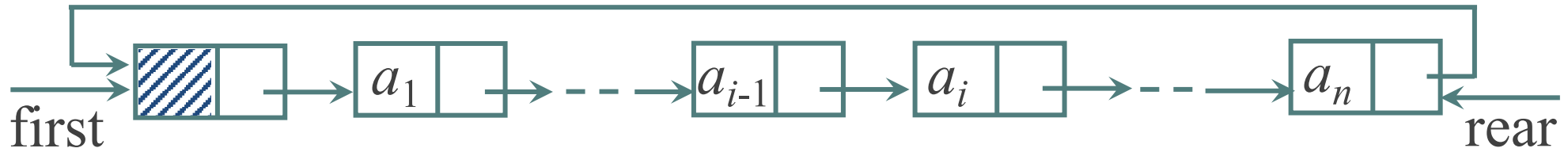
$p \neq \text{nullptr} \rightarrow p \neq \text{first}$

$p \rightarrow \text{next} \neq \text{nullptr} \rightarrow p \rightarrow \text{next} \neq \text{first}$

2.4 线性表的链接存储结构及实现

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: first->next

终端结点: $O(n)$

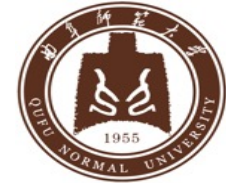


开始结点: rear->next->next

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



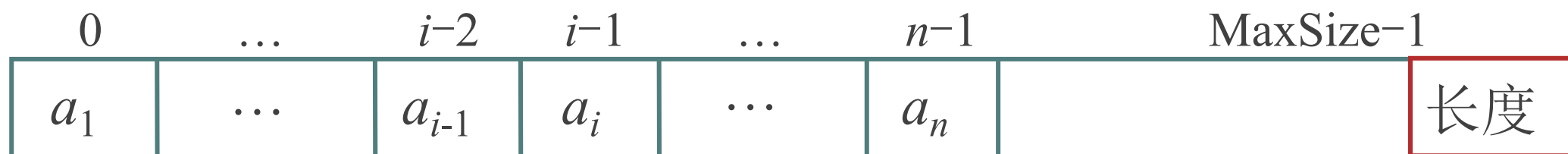
第二章 线性表

2.5 顺序表与链表的比较

2.5 顺序表与链表的比较

存储分配方式

📌 顺序表：采用**顺序**存储结构——**静态**存储分配，即用一段地址**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**存储位置**（下标）来实现



📌 链表：采用**链接**存储结构——**动态**存储分配，即用一组**任意**的存储单元存放线性表的元素，用**指针**来反映数据元素之间的逻辑关系

2.5 顺序表与链表的比较

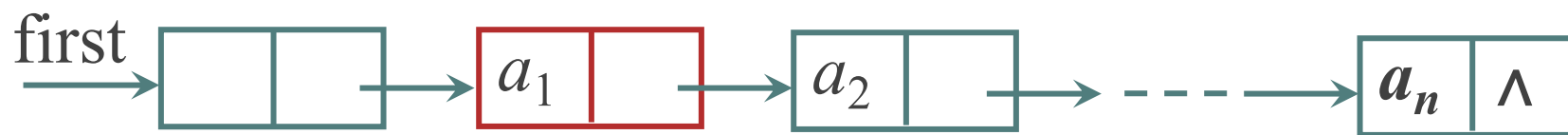
空间性能比较



结点的存储密度比较

顺序表：只存储数据元素

链表：指针的结构性开销



结构的存储密度比较

顺序表：预分配存储空间

链表：链表中的元素个数没有限制

2.5 顺序表与链表的比较

时间性能比较

✚ 按位查找

顺序表: $O(1)$, 随机存取

链表: $O(n)$, 顺序存取



✚ 插入和删除

顺序表: $O(n)$, 平均移动表长一半的元素

链表: 不用移动元素, 合适位置的指针—— $O(1)$

2.5 顺序表与链表的比较

结论

- ✦ 从空间上讲，若线性表中元素**个数变化**较大或者未知，最好使用链表实现；如果用户事先知道线性表的大致长度，使用顺序表的空间效率会更高
- ✦ 从时间上讲，若线性表**频繁查找**却很少进行插入和删除操作，或其操作和元素在表中的位置密切相关时，宜采用顺序表作为存储结构；若线性表需**频繁插入和删除**时，则宜采用链表做存储结构

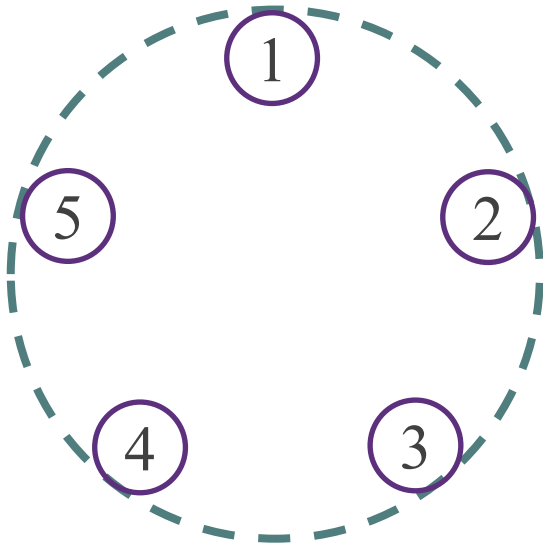
各有优缺点，应根据实际问题进行综合考虑，选定合适的实现方法

第二章 线性表

2.6/2.7 扩展、提高和应用

扩展：约瑟夫环问题

【约瑟夫环问题】 设 $n(n>0)$ 个人围成一个环， n 个人的编号分别为 $1, 2, \dots, n$ ，从第 1 个人开始报数，报到 m 时停止报数，报 m 的人出环，再从他的下一个人起重新报数，报到 m 时停止报数，报 m 的人出环，……，如此下去，直到所有人全部出环为止。对于任意给定 n 和 m ，求 n 个人出环的次序。



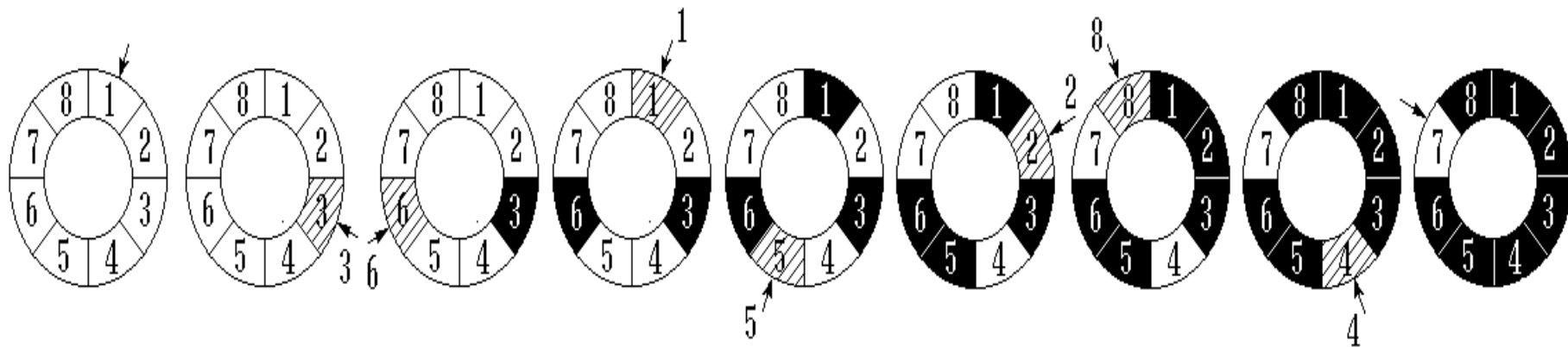
例如， $n = 5, m = 3$ ，则

出环的顺序是：3 1 5 2 4



如何存储这种环状线性结构，并求解约瑟夫环的出环次序呢？

扩展：约瑟夫环问题



```
struct Node
{
    int data;
    struct Node *next;
};
```

```
class JosephRing
{
public:
    JosephRing(int n);
    ~JosephRing();
    void Joseph(int m);
private:
    Node * rear;
};
```

```
JosephRing::JosephRing(int n)
{
    Node *s = nullptr;
    rear = new Node;
    rear->data = 1;
    rear->next = rear;
    for (int i = 2; i <= n; i++)
    {
        s = new Node;
        s->data = i;
        s->next = rear->next;
        rear->next = s;
        rear = s;
    }
}
```

```
int main()
{
    int n = 8, m = 3;
    JosephRing R(n);
    R.Joseph(m);
    return 0;
}
```

Output:

```
3  6  1  5
2  8  4  7
```

```
void JosephRing::Joseph(int m)
{
    Node *pre = rear;
    Node *p = rear->next;
    int count = 1;
    while (p->next != p)
    {
        if (count < m)
        {
            pre = p;
            p = p->next;
            count++;
        }
        else
        {
            cout << p->data << "\t";
            pre->next = p->next;
            delete p;
            p = pre->next;
            count = 1;
        }
    }
    cout << p->data << "\t";
    delete p;
}
```

扩展：一元多项式求和

数学表示： $P_n(x) = p_0 + p_1x + p_2x^2 + \dots + p_nx^n$

计算机表示： $P = (p_0, p_1, p_2, \dots, p_n)$

可以描述为一个由 **$n+1$** 个系数构成的**线性表**。

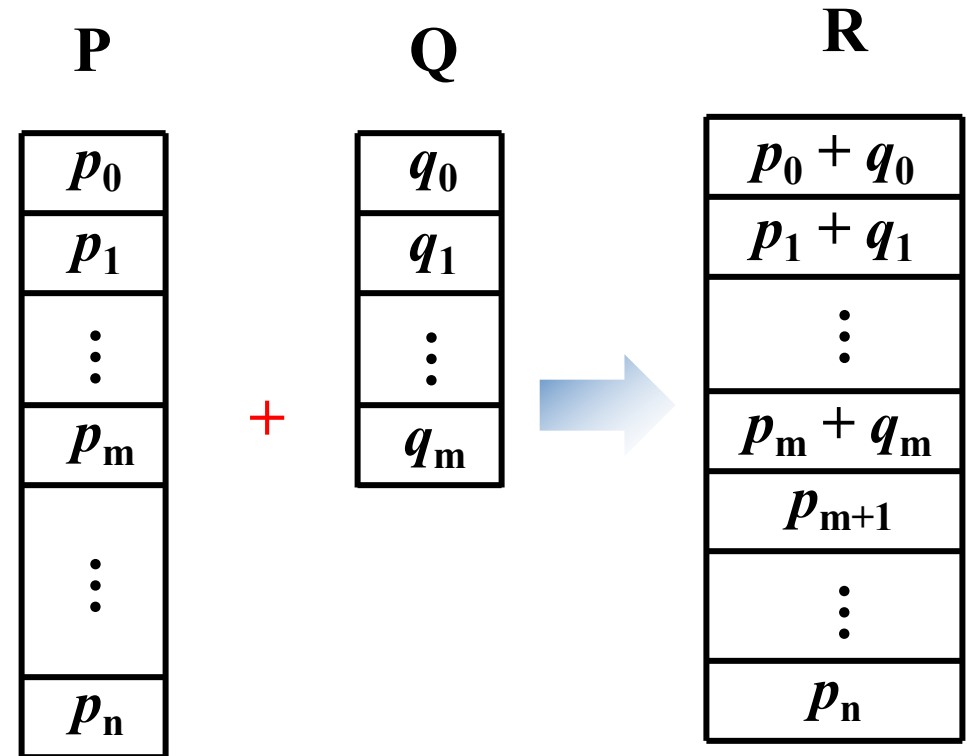
多项式相加：

设 $P_n(x)$: $P = (p_0, p_1, p_2, \dots, p_n)$

$Q_m(x)$: $Q = (q_0, q_1, q_2, \dots, q_m)$ $m < n$

则 $R_n(x) = P_n(x) + Q_m(x)$

$R = (p_0 + q_0, p_1 + q_1, p_2 + q_2, \dots, p_m + q_m, p_{m+1}, \dots, p_n)$



显然，采用**顺序存储**结构实现方便。

扩展：一元多项式求和

然而实际应用中，多项式的次数往往很高，且可能存在很多缺项。

例， $S(x) = 1 + 3x^{10000} + 2x^{20000}$

通常情况下，一元 n 次多项式写成：

$$P_n(x) = p_1x^{e_1} + p_2x^{e_2} + \dots + p_mx^{e_m}$$

$$p_i \neq 0; \quad 0 \leq e_1 < e_2 < \dots < e_m \leq n$$

计算机表示： $P = ((p_1, e_1), (p_2, e_2), \dots, (p_m, e_m))$

实现“求值”、“求项数”这样的操作，采用顺序存储结构。

p_1	e_1
p_2	e_2
\vdots	\vdots
p_m	e_m

？ 采用那种存储结构？

“多项式相加”等运算则采用链式存储结构

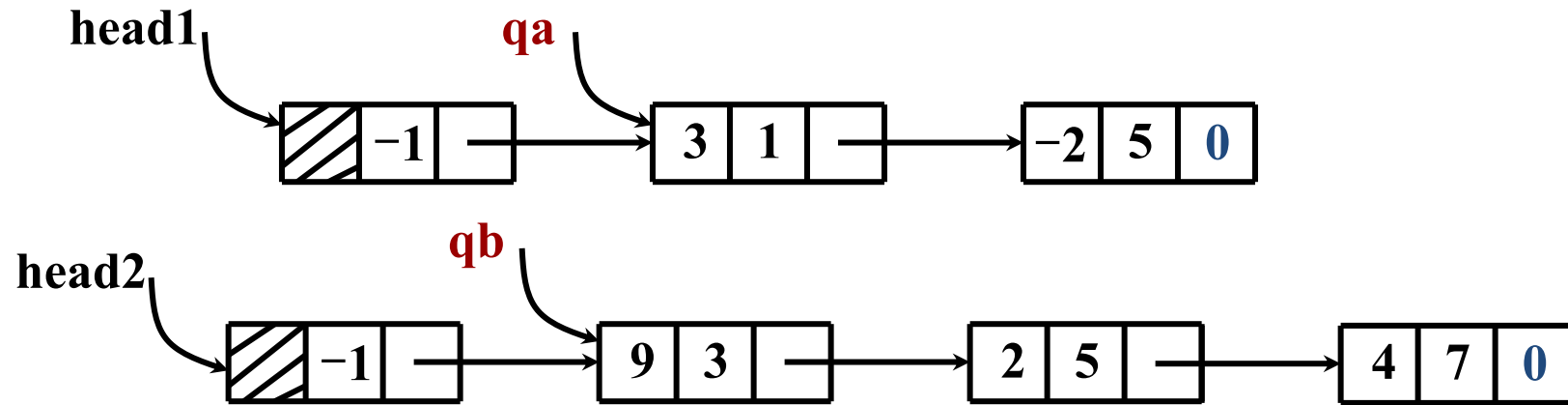
coef	expo	next
------	------	------

系数 指数 下一个结点

扩展：一元多项式求和

【算法】 一元多项式相加 $\text{AddPolyn}(\&\text{Pa}, \&\text{Pb})$ 要求: $\text{Pa} = \text{Pa} + \text{Pb}$

```
struct LNode
{
    DataType coef;
    DataType expo;
    struct LNode *next;
};
```



思想:

依据归并两个有序表的过程,
分三种情况考虑:

- 令 qa 和 qb 分别指向多项式 A 和 B 中当前进行比较的结点;
- 1: $qa \rightarrow \text{expo} < qb \rightarrow \text{expo}$, qa 所指向的结点应插入和多项式中
- 2: $qa \rightarrow \text{expo} > qb \rightarrow \text{expo}$, qb 所指向的结点应插入和多项式中
- 3: $qa \rightarrow \text{expo} = qb \rightarrow \text{expo}$, 求和 $qa \rightarrow \text{coef} + qb \rightarrow \text{coef}$

和 = 0, 释放 qa 和 qb 所指结点;

和 $\neq 0$, 修改 qa 所指结点的系数值, 释放 qb 所指结点;

本章小结

- ❧ 掌握**顺序表**与**链表**的**逻辑结构**和**存储结构**特点
- ❧ 熟练掌握线性表**两类存储结构的定义和描述方法**，以及线性表的各种**基本操作的实现**。
- ❧ 能够从**时间和空间复杂度**的角度综合比较线性表两种存储结构的不同特点及其适用场合。

实验一、顺序存储结构线性表的建立及操作

一、实验目的

1. 掌握线性表的存储结构的特点，理解顺序存储结构表示线性表的方法。
2. 掌握顺序存储结构线性表数据元素类型定义的格式与方法。
3. 掌握对线性表的元素进行删除和插入的原理与方法。
4. 用C++语言实现**顺序结构存储线性表的建立，元素的删除与插入算法**，并上机调试。

二、实验内容

1. 设计C++类及相关方法，用于维护学生成绩表：
基础信息：学号姓名分数：`long num; char name[10]; float score;`
2. 写出建立线性表，并向线性表中输入数据的函数。
3. 写出删除指定学号的学生信息，及按学生的成绩顺序插入新的学生信息的函数。（假定学生的成绩已有序排列）
4. 写出输入及输出的内容。
5. 合并两张有序表（**扩展内容**）

实验时间： 第2周周四晚 19:00-21:00
实验地点： 格物楼A216

实验二、链式存储结构线性表的建立及操作

一、实验目的

1. 掌握线性表的链式存储结构的特点，理解链式存储结构表示线性表的方法。
2. 掌握链式存储结构线性表数据元素类型定义的格式与方法。
3. 掌握单链表建立、遍历、查找、新元素插入、及元素删除的原理与方法。
4. 用C++语言实现单链表，并上机调试。

二、实验内容

1. 设计C++类及相关方法，用于维护单链表。
2. 写出建立单链表，并向单链表中输入数据的函数。
3. 实现单链表的建立、遍历、查找、新元素插入、及元素删除，写出输入及输出的内容。
4. 将两个有序单链表合并为一个有序单链表（**扩展内容**）
5. 双链表及循环链表的实现（**扩展内容**）

实验时间： 第4周周四晚 19:00-21:00
实验地点： 格物楼A216

1. 试总结单链表引入头结点的原因？
2. 编程：如何逆置一个单链表为一个新表？
3. 教材P66, 2(1)题：请说明顺序表和单链表有何优缺点？并分析不同情况下采用何种存储结构更合适？
4. 算法设计：在顺序表中删除所有元素值为 x 的元素，要求空间复杂度为 $O(1)$ ，给出算法伪代码和源代码。
5. 算法设计：已知单链表中各结点的元素值为整型且递增有序，设计算法删除链表中大于 mink 且小于 maxk 的所有元素，并释放被删结点的存储空间，给出算法伪代码和源代码。



Thank You !

Q & A