



第三章 链表



顺序存储结构的优缺点

■ 优点

- 逻辑相邻，物理相邻
- 可随机存取任一元素
- 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
- 表容量难以扩充

A decorative graphic on the left side of the slide, consisting of overlapping yellow, red, and blue squares with a black crosshair.

链表 – Overview

- 单向链表
- 循环链表
- 双向链表
- 稀疏矩阵的表示

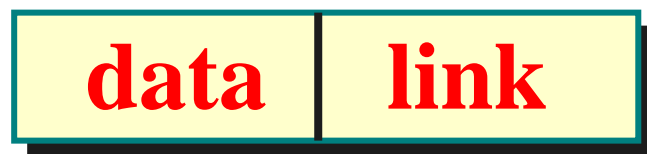


单向链表

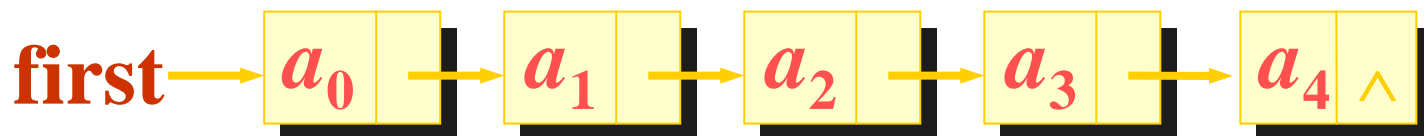
单向链表

■ 特点

- ◆ 每个元素(表项)由结点(Node)构成。

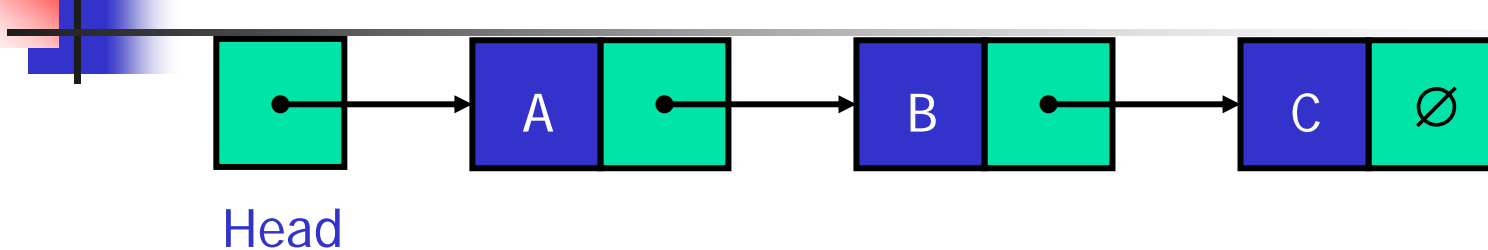


- ◆ 线性结构

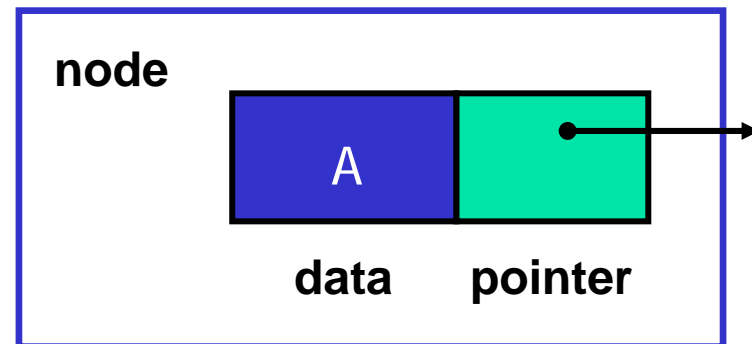


- ◆ 结点可以不连续存储
- ◆ 表可扩充

Linked Lists



- A *linked list* is a series of connected *nodes*
- Each node contains at least
 - A piece of data (any type)
 - Pointer to the next node in the list
- *Head*: pointer to the first node
- The last node points to NULL

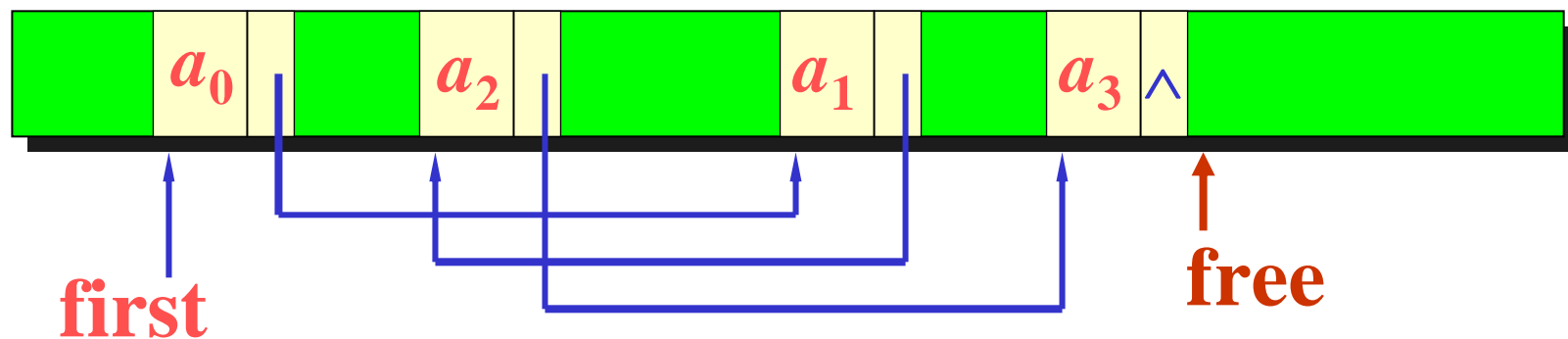


单向链表的存贮映像



↑
free

(a) 可利用存储空间



(b) 经过一段运行后的单链表结构

单链表的类定义

- 多个类表达一个概念(单链表)
 - ◆ 链表结点(ListNode)类
 - ◆ 链表(List)类
- 定义方式
 - ◆ 复合方式
 - ◆ 嵌套方式
 - ◆ 继承方式



class List; //链表类定义（复合方式）

class ListNode {

//链表结点类

friend class List;

//链表类为其友元类

private:

int data;

//结点数据, **整型**

ListNode *link;

//结点指针

};

class List {

//链表类

private:

ListNode *first;

//表头指针

};



class List { //链表类定义(嵌套方式)

private:

class ListNode { //嵌套链表结点类

public:

int data;

ListNode *link;

};

ListNode *first, *current; //表头指针

public:

//链表操作.....

};



链表类和链表结点类定义(继承方式)

```
class ListNode {           //链表结点类
```

```
protected:
```

```
    int data;
```

```
    ListNode * link;
```

```
};
```

```
class List : public class ListNode {
```

```
//链表类， 继承链表结点类的数据和操作
```

```
private:
```

```
    ListNode *first, *current;    //表头指针
```

```
};
```



class List



A Simple Linked List Class

- We use two classes: **CNode** and **CList**
- Declare CNode class for the nodes
 - data: **double**-type data in this example
 - next: a pointer to the next node in the list

```
class CNode {  
public:  
    double      m_dData;      // data  
    CNode*      m_pNext;      // pointer to next  
};
```



Methods of CNode

- `int SetValue();`
- `double GetValue();`
- `CNode *GetNext();`
- `etc.`



A Simple Linked List Class

```
class CList {
public:
    CList(void) { head = NULL; }           // constructor
    ~CList(void);                          // destructor

    bool IsEmpty() { return head == NULL; }
    Node* InsertNode(int index, double x);
    int FindNode(double x);
    int DeleteNode(double x);
    void DisplayList(void);
private:
    Node* head;
};
```



Methods of CList

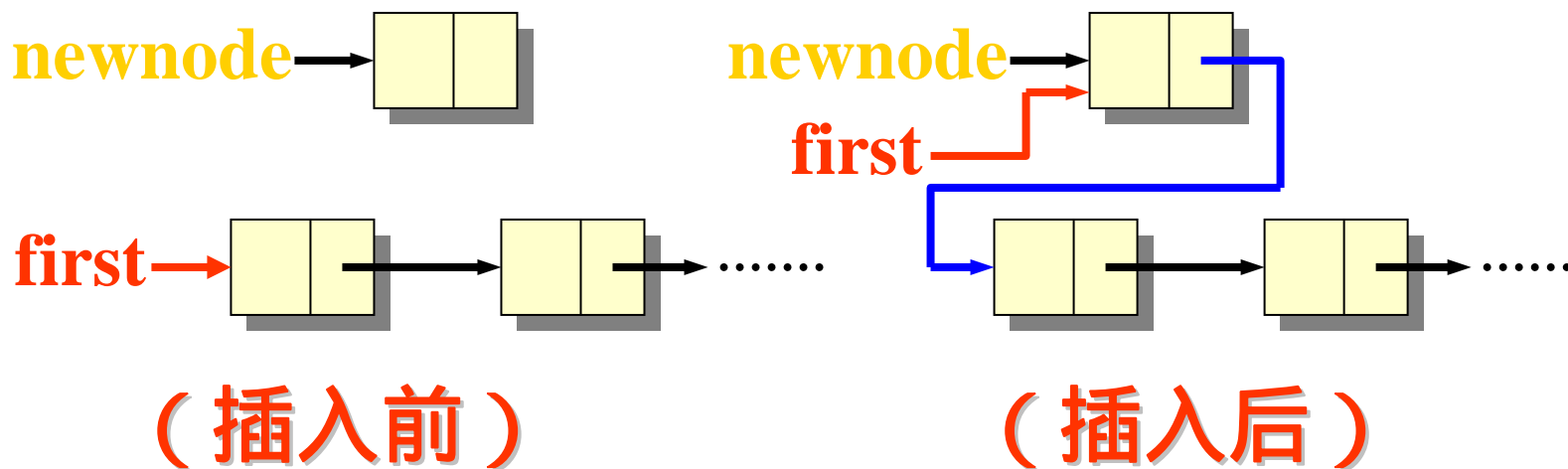
- List();
- ~List();
- int IsEmpty();
- int Free();
- int InsertAt(int nPos);
- int InsertAfter(CNode *pNode);
- int Locate(CNode *pNode);
- CNode *GetAt(int nPos);
- int DeleteAt(int nPos);
- int DeleteAfter(CNode *pNode);
- int Size();
- void Print();

单链表中的插入与删除

插入

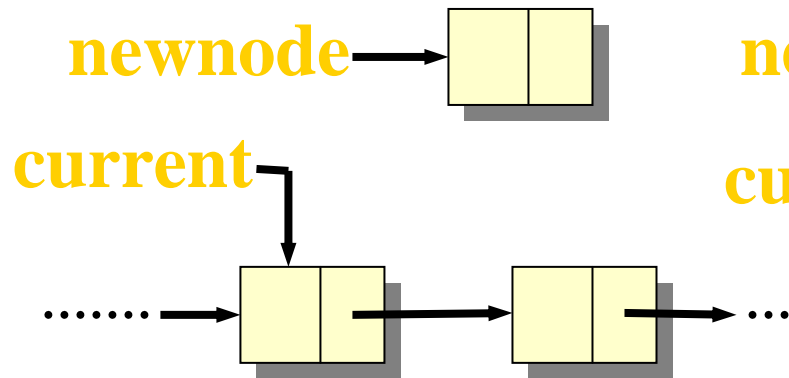
- ◆ 第一种情况：在第一个结点前插入

`newnode->link = first ;`
`first = newnode;`

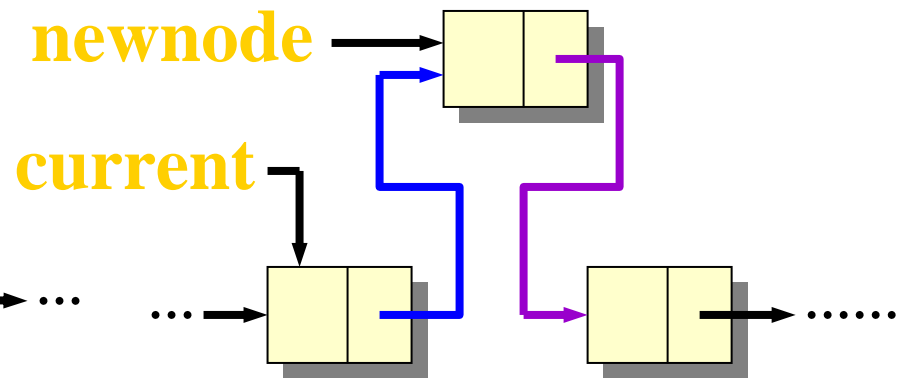


◆ 第二种情况：在链表中间插入

newnode->link = current->link;
current->link = newnode ;



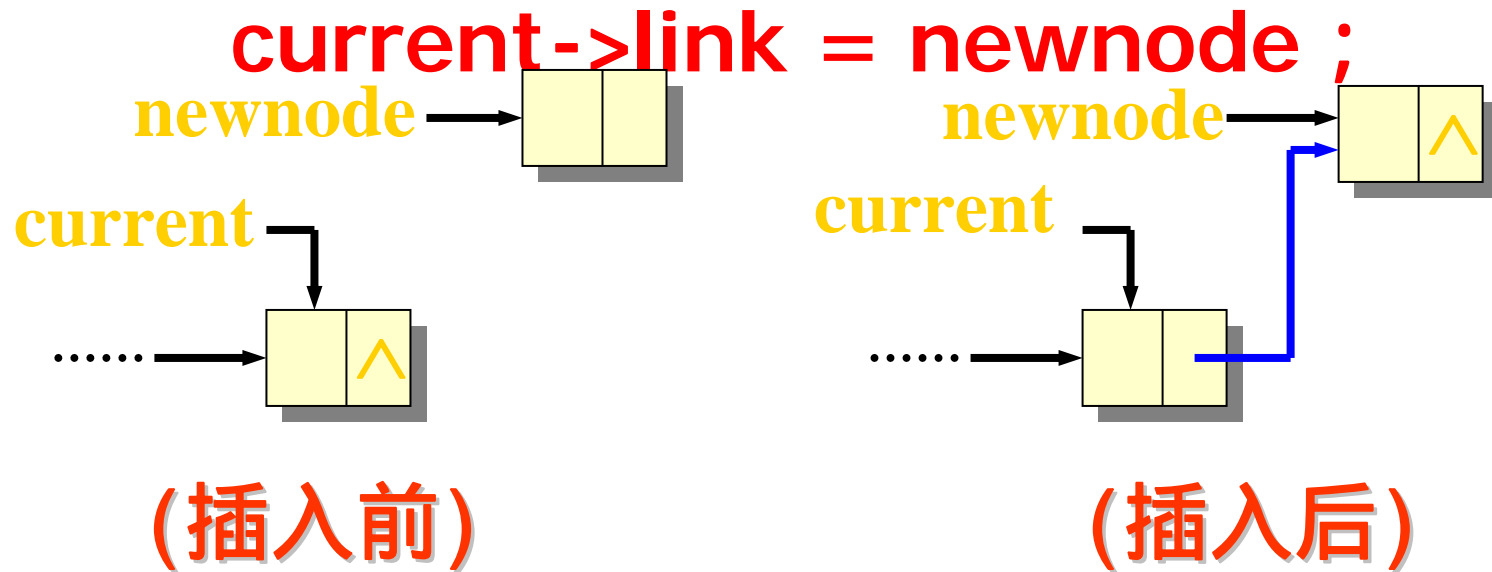
(插入前)



(插入后)

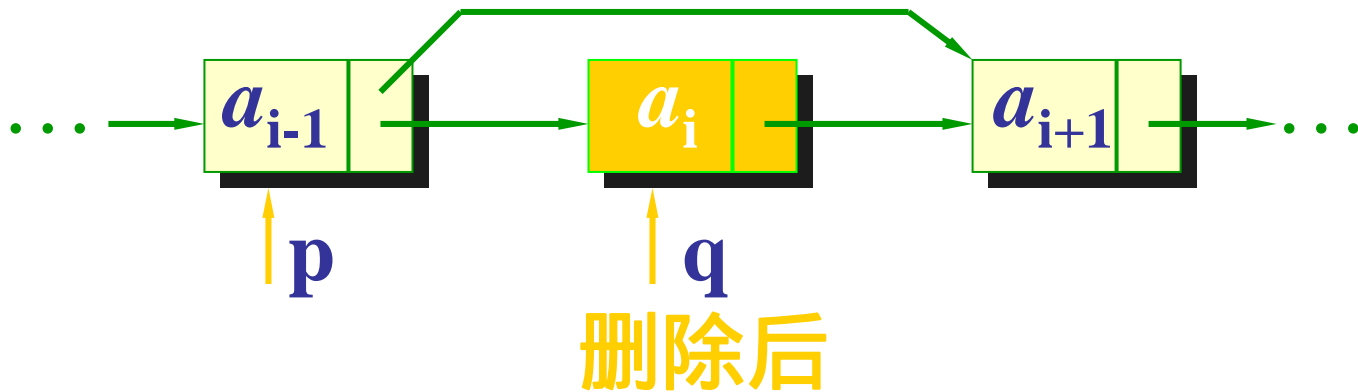
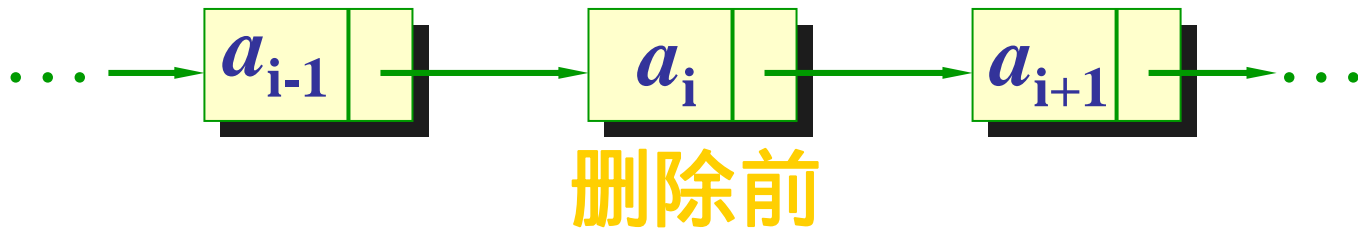
◆ 第三种情况：在链表末尾插入

newnode->link = current->link;



删除

- ◆ 第一种情况：删除表中第一个元素
- ◆ 第二种情况：删除表中或表尾元素



在单链表中删除含 a_i 的结点

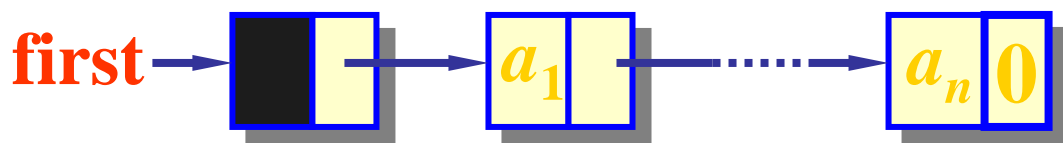
A decorative graphic on the left side of the slide, consisting of overlapping blue, red, and yellow squares with a black crosshair.

带表头结点的单链表

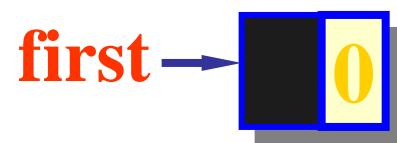
带表头结点的单链表

表头结点位于表的最前端，本身不带数据，仅标志表头。

- 设置表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。

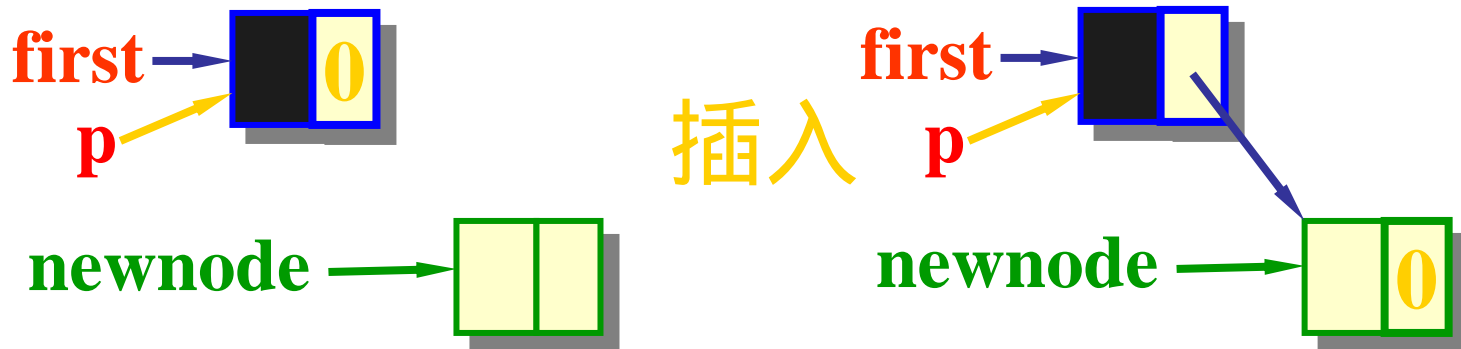


非空表



空表

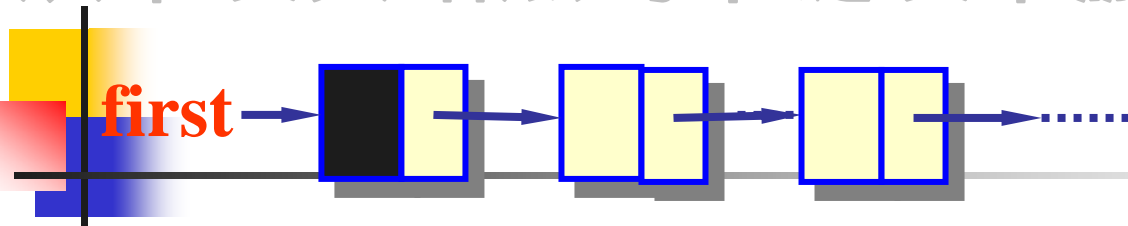
在带表头结点的单链表 第一个结点前插入新结点



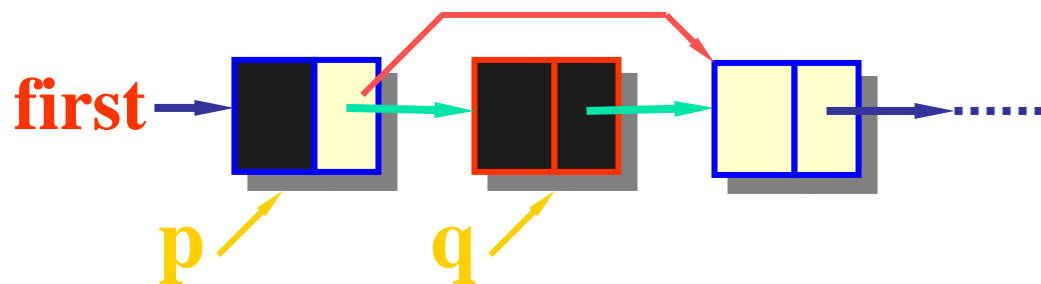
$\text{newnode} \rightarrow \text{link} = \text{p} \rightarrow \text{link}; \text{p} \rightarrow \text{link} = \text{newnode};$



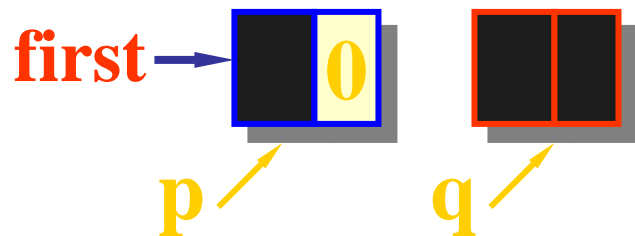
从带表头结点的单链表中删除第一个结点



(非空表)



$q = p \rightarrow \text{link};$
 $p \rightarrow \text{link} = q \rightarrow \text{link};$
 $\text{delete } q;$



(空表)



循环链表

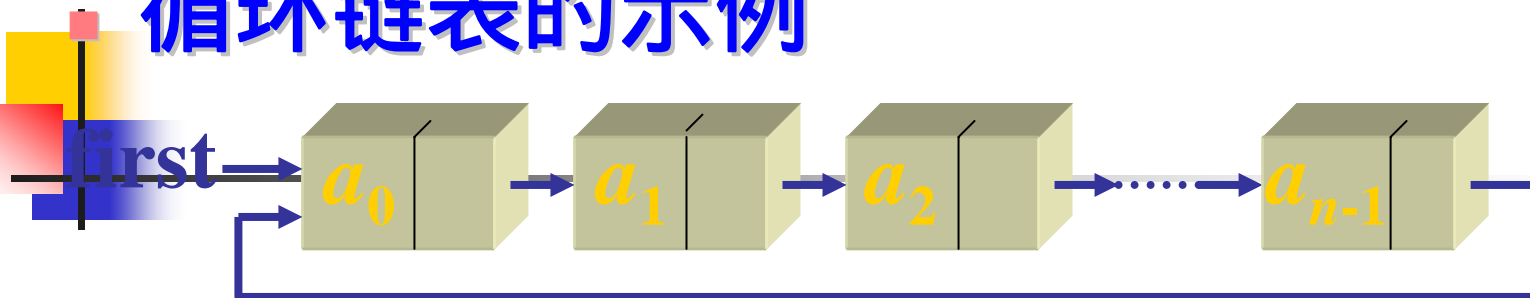


循环链表

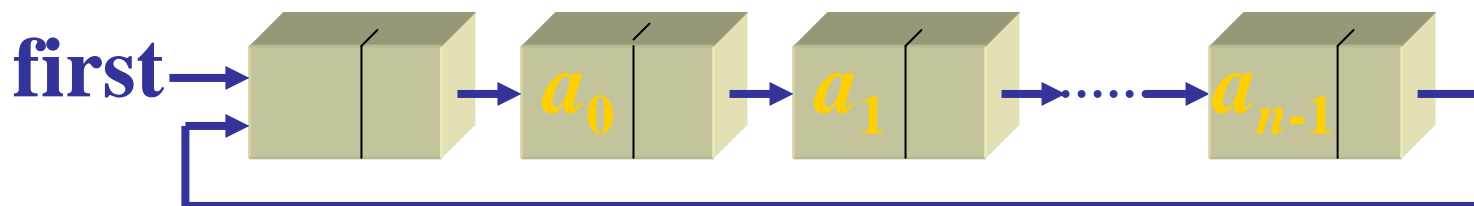
循环链表是单链表的变形。

- 循环链表最后一个结点的 **link** 指针不为 **NULL**，而是指向了表的前端。
- 为简化操作，在循环链表中往往加入表头结点。
- 循环链表的特点是：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。

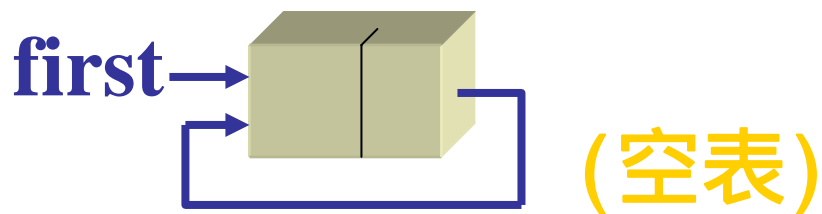
循环链表的示例



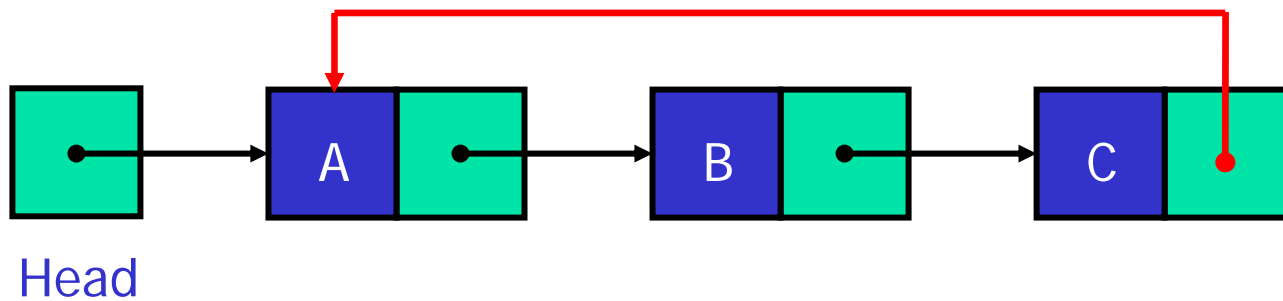
带表头结点的循环链表



(非空表)



带表头的循环链表

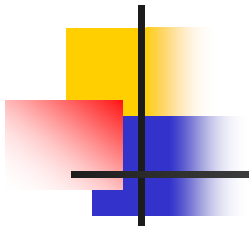




练习：多项式



多项式 (Polynomial)


$$P_n(x) = c_0 + c_1x + c_2x^2 + \cdots + c_nx^n$$

$$= \sum_{i=0}^n c_i x^i$$

- n 阶多项式 $P_n(x)$ 有 $n+1$ 项。
 - ◆ 系数 $c_0, c_1, c_2, \dots, c_n$
 - ◆ 指数 $0, 1, 2, \dots, n$ 。按升幂排列



多项式(Polynomial)的抽象数据类型

```
class Polynomial {
```

```
public:
```

```
    Polynomial ( );
```

//构造函数

```
    int operator ! ( );
```

//判是否零多项式

```
    float Coef ( int e);
```

```
    int LeadExp ( );
```

//返回最大指数

```
    Polynomial Add (Polynomial poly);
```

```
    Polynomial Mult (Polynomial poly);
```

```
    float Eval ( float x);
```

//求值

```
}
```



多项式的存储表示

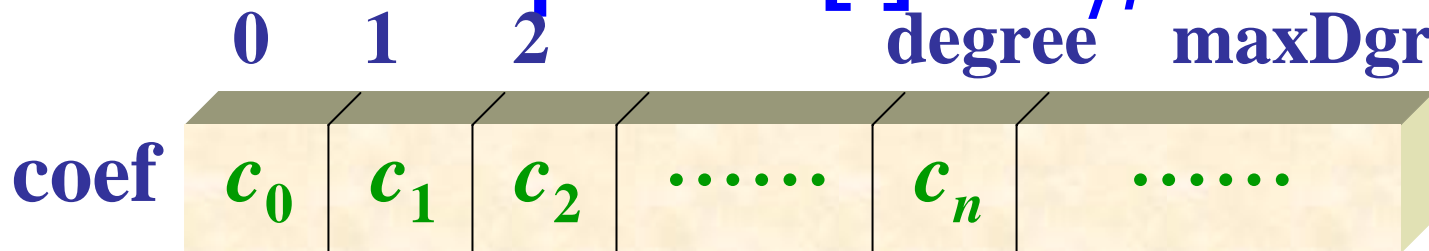
第一种：private:

(静态数组表示) `int degree;`
 `float coef`
 `[maxDegree+1];`

$P_n(x)$ 可以表示为：

`pl.degree = n`

`pl.coef[i] = c_i , $0 \leq i \leq n$`





第二种：private:

(动态数组表示)

```
int degree;
```

```
float *coef;
```

构造函数 Polynomial (int sz) {

```
degree = sz;
```

```
coef = new float [degree + 1];
```

```
}
```

以上两种存储表示适用于指数连续排列的多项式。但对于指数不全的多项式如 $P_{101}(x) = 3 + 5x^{50} - 14x^{101}$, 不经济。



第三种：多项式的链表表示

在多项式的链表表示中每个结点三个数据成员：

Data \equiv Term



- 优点是：
 - ◆ 多项式的项数可以动态地增长，不存在存储溢出问题。
 - ◆ 插入、删除方便，不移动元素。



多项式(polynomial)类的链表定义

```
struct Term {                                //多项式结点定义
    float coef;                             //系数
    int exp;                                //指数
    Term ( float c, int e ) { coef = c; exp = e; }
};

class Polynomial    {                       //多项式类的定义
    List<Term> poly;                         //构造函数
    friend Polynomial & operator +
        ( Polynomial &, Polynomial &); //加法
};
```



作业：多项式类

- 多项式的各种运算
 - 加，减，乘，除^{*}
- 简单多项式的其他特殊运算
 - 二项式的矩阵表示
 - 二次多项式的求根计算
 - 其他
- Deadline: Monday, Oct. 10



Interfaces

```
class Polynomial
{
public:
    Polynomial ( );           //构造函数
    int operator ! ( );       //判是否零多项式
    float Coef ( int e);      // 返回某次数的系数
    int LeadExp ( );          //返回最大指数
    Polynomial Add (Polynomial poly); // 加法
    Polynomial Minus(Polynomial poly); // 減法
    Polynomial Mult (Polynomial poly); // 乘法
    float Eval ( float x);    //求值
}
```



要求

- 写文档
- 测试说明
- 占成绩40%



讨论

- 链表的表头可存放其他信息
 - 链表（非节点）的其他信息
 - 是否需存放链表元素个数



Q&A
