



## 第二章 线性表

机电工程与自动化学院 L栋301

任卫红 助理教授

[renweihong@hit.edu.cn](mailto:renweihong@hit.edu.cn)

<http://faculty.hitsz.edu.cn/renweihong>



## 2.1 线性表的类型定义

# C语言和C++

## 一、内存空间申请

### 1. 静态数组 (C语言)

DataType A[N]—N为常量

```
int A[10]; char A[100]...
```

```
int A[n] //n为变量, 编译出错
```

静态数组位于栈区, 定义时就已分配固定空间, 在运行时不能改变, 使用后自动释放。

### 2. 动态数组 (C语言)

```
int* p = (int *) malloc ( sizeof(int) * 100 )
```

```
int* p = (int *) malloc ( sizeof(int) * n )
```

动态数组位于堆区, 运行时进行空间分配, 可通过**realloc**改变, 使用后需手动释放 (**free**)。



### 1. 静态数组 (C++)

可以认为和C语言一样, 初始化等稍有区别。

### 2. 动态数组 (C++)

```
int* p = new int[100]
```

```
int* p = new int[n]
```

```
int* p = new int(100)
```

```
delete p; delete []p
```

动态数组位于自由存储, 默认使用堆来实现自由存储, 堆与自由存储区并不等价。不可以改变大小, 使用后需手动释放 (**delete**)。

# C语言和C++

## 一、内存空间申请

### 1. 静态数组 (C语言)

DataType A[N]—N为常量

int A[10]; char A[100]...

int A[n] //n为变量，编译出错

静态数组位于栈区，定义时就已分配固定空间，在运行时不能改变，使用后自动释放。

### 2. 动态数组 (C语言)

int\* p = (int \*) malloc ( sizeof(int) \* 100 )

int\* p = (int \*) malloc ( sizeof(int) \* n )

动态数组位于堆区，运行时进行空间分配，可通过realloc改变，使用后需手动释放 (free)。

### 3. 类模板-容器

Vector<DataType> A(n, value)

Vector<int> A(10); Vector<int> A(10, 1)

Vector<float> A(n); Vector<int> A

- ✓ 严格的线性顺序排列,可通过元素在序列中的位置访问元素
- ✓ 可以看成一种数据结构，或者ADT抽象数据类型
- ✓ 能够感知内存分配器，进行动态扩容
- ✓ 存在作用域，可以自动销毁
- ✓ 不同的容器运行时数据存储的位置是不一样的，Vector运行在堆上

# C语言和C++

## 二、输入输出

### 1. C语言

scanf(格式化字符串,参数列表)

```
char name[21]; int age=0;  
char sex=0; double weight=0;
```

```
scanf("%s ", name);
```

%s遇到空格符、回车符、制表符 (tab) 结束

```
scanf("%c ", &sex);
```

```
scanf("%d ", &age);
```

```
scanf("%lf ", &weight);
```

```
scanf("%s %c %d %lf", name,&sex,  
&age,&weight);
```

printf("输出控制符 非输出控制符", 输出参数)

```
# include <stdio.h>
```

```
int main()
```

```
{
```

```
    int i = 10;
```

```
    int j = 3;
```

```
    printf("i = %d, j = %d\n", i, j);
```

```
    return 0;
```

```
}
```

# C语言和C++

## 二、输入输出

### 1. C语言

scanf(格式化字符串,参数列表)

```
char name[21]; int age=0;  
char sex=0; double weight=0;
```

```
scanf("%s ", name);
```

%s遇到空格符、回车符、制表符 (tab) 结束

```
scanf("%c ", &sex);
```

```
scanf("%d ", &age);
```

```
scanf("%lf ", &weight);
```

```
scanf("%s %c %d %lf", name,&sex,  
&age,&weight);
```

```
char c;  
c=getchar();  
getchar(); //清空缓冲区
```

在输入缓冲区顺序读入一个字符(包括空格、回车和Tab)

```
char s[100];  
gets("%s", s);  
printf("%s", s);
```

gets()主要用来接收字符串，可以接收空格，遇到 ‘\n’时结束，但不接收 ‘\n’，把 ‘\n’留存输入缓冲区

输入 “**abc def**”，输出为 “**abc def**”

# C语言和C++

## 二、输入输出

### 2. C++(cin, cout)

```
#include<iostream>
using namespace std;
int main(){
    int x;
    float y;
    cout<<"Please input an int number:"<<endl;
    cin>>x;                                // 空格、制表符和换行符忽略
    cin>>x>>y;
    cout<<"The int number is x= "<<x<<endl;
    cout<<"Please input a float number:"<<endl;
    cin>>y;
    cout<<"The float number is y= "<<y<<endl;
    return 0;
}
```

# C语言和C++

## 二、输入输出

### 2. C++ (cin.getline)

cin.getline(接收字符串的变量, 接收字符个数, 结束字符)

```
#include <iostream>
using namespace std;
main ()
{
    char m[20];
    cin.getline(m,5);
    cout<<m<<endl;
}
```

输入: jkljkljkl  
输出: jklj

接收5个字符到m中, 其中最后一个为'\0', 所以只看到4个字符输出;

如果把5改成20:  
输入: jkljkljkl  
输出: jkljkljkl

输入: jklf fjlsjf fj sdklf  
输出: jklf fjlsjf fj sdklf



# C语言和C++

## 二、输入输出

### 2. C++ (cin.getline)

```
#include <iostream>
using namespace std;
main ()
{
    char m[100];
    cin.getline(c,8,'e'); //abcdefghijlmn
    cout<<c<<endl;      //abcd
    return 0;
}
```

# C语言和C++

## 二、输入输出

### 2. C++ (getline)

用法：接收一个字符串，可以接收空格并输出，需包含 “`#include<string>`”

```
#include<iostream>
#include<string>
using namespace std;
main ()
{
    string str;
    getline(cin, str);
    cout<<str<<endl;
}
```

输入： jkljkljkl

输出： jkljkljkl

输入： jkl jfksldfj jklsjfl

输出： jkl jfksldfj jklsjfl

# C语言和C++

## 三、面向过程和面向对象

### 1. 面向过程(C语言)

```
#define LIST_INIT_SIZE 100 // 顺序表存储空间的初始分配量
```

```
typedef struct {
```

```
    int* elem;    // 顺序表存储空间的基址（指向顺序表所占内存的  
    起始位置）
```

```
    int length; // 当前顺序表长度（包含多少元素）
```

```
    int listsize; // 当前分配的存储容量（可以存储多少元素）
```

```
} SqList;
```

```
int ListInsert(SqList* L, int i, int e);
```

```
int ListDelete(SqList* L, int i, int *e);
```

```
void ListDisplay(SqList* L);
```

```
int InitList(SqList* L);
```

```
int DestroyList(SqList* L);
```

# C语言和C++

## 三、面向过程和面向对象

### 1. 面向过程(C语言)

```
int main() {  
    int i, e, n=10;  
    SqList L;  
    printf("InitList \n");  
    InitList(&L);  
    ListInsert(&L, i, 2 * i);  
    ListDisplay(&L);  
    ListDelete(&L, 6, &e)  
    return 1;  
}
```

# C语言和C++

## 三、面向过程和面向对象

### 2. 面向对象 (C++)

`#define LIST_INIT_SIZE 100` // 顺序表存储空间的初始分配量

`class SqList{`

`private:`

`int *elem;`

`int length;`

`int listsize;`

`public:`

`SqList();` //构造函数

`int ListInsert(int i, int e);`

`int ListDelete(int i, int &e);`

`void ListDisplay();`

`~SqList();` //析构函数

`};`

# C语言和C++

## 三、面向过程和面向对象

### 2. 面向对象(C++)

```
int main() {  
    int n=10, e;  
    SqList list;  
    for (int i=0; i<n; i++)  
    {  
        cout << "Input a list element: " << endl;  
        cin>>e;  
        list.ListInsert(1, e);  
    }  
    list.ListDisplay();  
    list.ListDelete(2, e);  
    list.ListDisplay();  
    return 0;  
}
```

# C语言和C++

## 四、模板 `template <typename T>`

### 1. 函数模板

```
int Max(int a, int b)
{
    return a > b ? a : b;
}
```

```
char Max(char a, char b)
{
    return a > b ? a : b;
}
```

```
float Max(float a, float b)
{
    return a > b ? a : b;
}
```

//template 关键字告诉C++编译器，需要开始泛型编程；T - 参数化数据类型

```
template <typename T>
```

```
T Max(T a, T b) {
    return a > b ? a : b;}
```

```
void main(){
```

```
    int x = 1;
```

```
    int y = 2;
```

```
    cout << "max(1, 2) = " << Max(x, y) << endl;
```

```
    cout << "max(1, 2) = " << Max<int>(x, y) << endl;
```

```
    float a = 2.0;
```

```
    float b = 3.0;
```

```
    cout << "max(2.0, 3.0) = " << Max(a, b) << endl;
```

```
    system("pause");
```

```
    return;
```

```
}
```

# C语言和C++

## 四、模板 `template <class T>`

### 2. 类模板

```
class Compare_int
```

```
{
```

```
public:
```

```
    Compare(int a, int b)
```

```
    {    x=a;
```

```
        y=b;
```

```
    }
```

```
    int max( )
```

```
    {
```

```
        return (x>y)?x:y;
```

```
    }
```

```
    int min( )
```

```
    {
```

```
        return (x<y)?x:y;
```

```
    }
```

```
private :   int x, y;};
```

```
class Compare_float
```

```
{
```

```
public:
```

```
    Compare(float a, float b)
```

```
    {    x=a;
```

```
        y=b;
```

```
    }
```

```
    float max( )
```

```
    {
```

```
        return (x>y)?x:y;
```

```
    }
```

```
    float min( )
```

```
    {
```

```
        return (x<y)?x:y;
```

```
    }
```

```
private :   float x, y;};
```



# C语言和C++

## 四、模板 `template <class T>`

### 2. 类模板

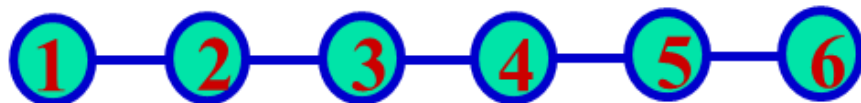
```
template <class numtype>  
//声明一个模板，虚拟类型名为  
numtype
```

```
class Compare  
{  
public:  
    Compare(numtype a, numtype b)  
    {  
        x=a;  
        y=b;  
    }  
    numtype max( )  
    {  
        return (x>y)?x:y;  
    }  
    numtype min( )  
    {  
        return (x<y)?x:y;  
    }  
private : numtype x, y;};
```

# 第一节 线性表

## 一、线性表

- 线性表是最简单的一类线性数据结构



- 线性表是由n个数据元素组成的有限序列，相邻数据元素之间存在着序偶关系，可以写为：

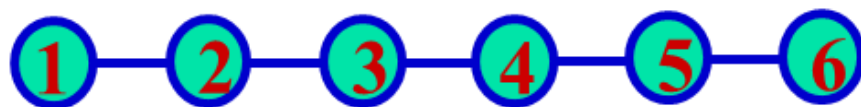
$$(a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_{n-1}, a_n)$$

其中,  $a_i$  是表中元素,  $i$  表示元素  $a_i$  的位置,  $n$  是表的长度

# 第一节 线性表

## 二、线性表的特点

- 在数据元素的非空有限集中



1. 存在惟一的一个被称作“第一个”的数据元素(1)
2. 存在惟一的一个被称作“最后一个”的数据元素(6)
3. 除第一个元素外，每个数据元素均只有一个前驱
4. 除最后一个元素外，每个数据元素均只有一个后继  
(1的后继是2，2的后继是3)

# 第一节 线性表

## 二、线性表的特点

- 线性表中的元素具有相同的特性，属于同一数据对象，如：
  1. 26个字母的字母表：(A, B, C, D, ..., Z)
  2. 近期每天的平均温度：(30°C, 28°C, 29°C, ...)

# 第一节 线性表

## 二、线性表的特点

- 复杂线性表中，数据元素可以由若干个数据项组成
- 此时，数据元素可以成为记录
- 含有大量记录的线性表称为文件

	姓名	学号	班级	...
1	李力	200513001	计B	...
2	张明	200513002	计B	...
...	...	...	...	...
60	吴丽	200513060	计A	...



## 2.2 顺序表

## 第二节 顺序表

### 一、顺序表

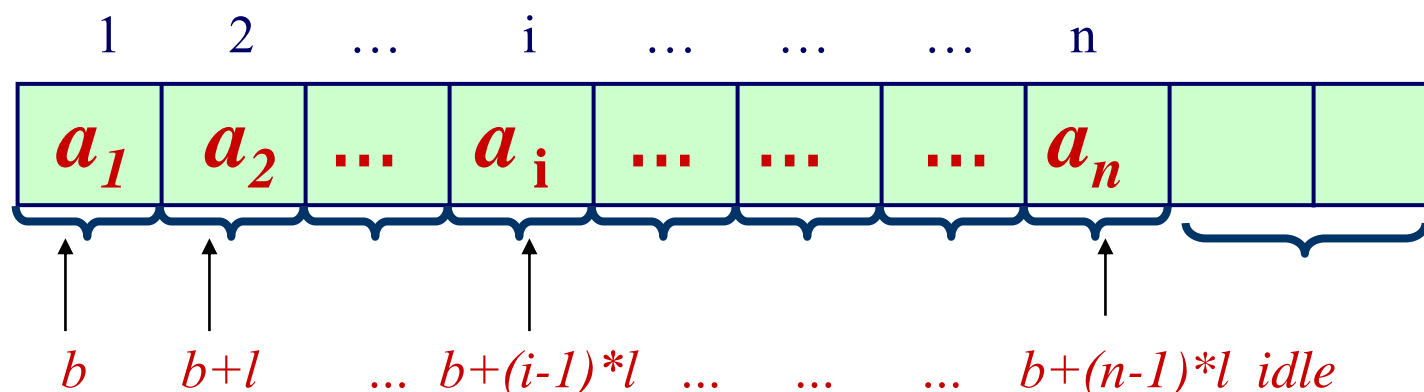
- 顺序表是线性表的顺序表示
- 用一组地址连续的存储单元依次存储线性表的数据元素

b	b+1	b+2	b+3	b+4	...	b+24	b+25
A	B	C	D	E	...	Y	Z

## 第二节 顺序表

### 一、顺序表（元素位置）

#### ■ 顺序表数据元素的位置：



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

$$\text{LOC}(a_i) = \text{LOC}(a_1) + (i-1)*l \quad l \text{ 表示元素占用的内存单元数}$$

只要确定了起始位置，则其他元素的地址均可知，均可随机存取！



## 第二节 顺序表

### 二、顺序表的定义

- 采用动态分配的一维数组表示顺序表

```
#define LIST_INIT_SIZE    100    // 线性表存储空间的初始分配量  
  
class SqList{  
    int    *elem;        // 存储空间基址  
    int    length;        // 当前长度  
    int    listsize;      // 当前分配的存储容量(元素数)  
  
};
```

## 第二节 顺序表

### 三、顺序表的初始化

#### ■ 创建顺序表

```
#define LIST_INIT_SIZE    100    // 线性表存储空间的初始分配量

class SqList{
    int    *elem;        // 存储空间基址
    int    length;        // 当前长度
    int    listsize;      // 当前分配的存储容量(元素数)
public:
    SqList();
    ~SqList();
};
```

## 第二节 顺序表

### ■ 创建顺序表

```
□ SqList::SqList()    //构造函数创建顺序表
{
    elem = new int[LIST_INIT_SIZE];
    if(!elem)
        exit(-1);
    listsize = LIST_INIT_SIZE;
    length = 0;
}
```

## 第二节 顺序表

### 三、顺序表的插入

- 顺序表的插入操作是指在顺序表的第 $i-1$ 个数据元素和第 $i$ 个数据元素之间插入一个新的数据元素，即将长度为 $n$ 的顺序表：

$$(a_1, \dots, a_{i-1}, a_i, \dots, a_n)$$

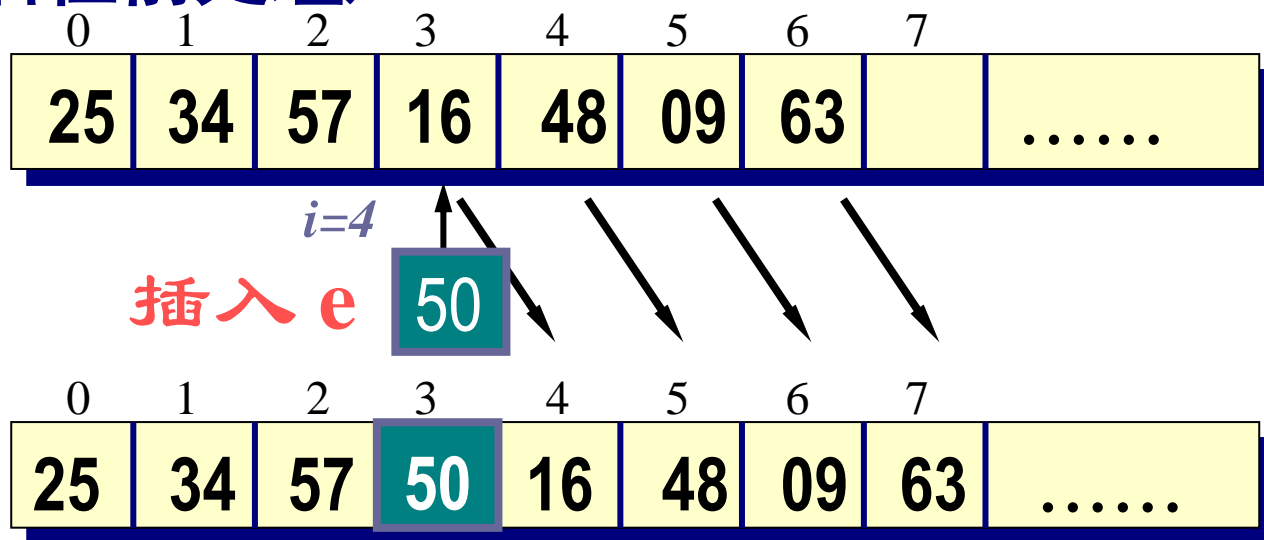
变成长度为 $n+1$ 的顺序表：

$$(a_1, \dots, a_{i-1}, e, a_i, \dots, a_n)$$

## 第二节 顺序表

### 三、顺序表的插入

- 在第3个元素与第4个元素之间插入新元素b
- 需要将最后元素n至第4元素 (共 $7-4+1$ ) 都向后移一位 (从后往前处理)



## 第二节 顺序表

### 三、顺序表的插入

类**SqList**增加插入方法: **int ListInsert(int i,int e);**

```
int SqList::ListInsert(int i,int e) //在第i个位置插入元素e, i从1开始
{
    int k;

    if(i<1 || i>length+1) //插入位置不正确
        return 0;

    for(k=length-1; k>=i-1; k--) //元素后移
        elem[k+1] = elem[k];
    elem[k+1] = e; //插入元素
    length++; //长度加1
    return 1;
}
```

## 第二节 顺序表

### 三、顺序表的插入

- 在顺序表中第*i*个位置插入一个元素，需要向后移动元素个数为： $n-i+1$
- 平均移动元素数为（假设在第*i*个元素之前插入的概率为  $p_i$ ）：

$$E_{is} = \sum_{i=1}^{n+1} p_i \times (n-i+1)$$

## 第二节 顺序表

### 三、顺序表的插入

- 当插入位置等概率时,  $p_i = 1/(n+1)$ , 因此:

$$E_{is} = \sum_{i=1}^{n+1} [1/(n+1)] \times (n-i+1) = n/2$$

- 顺序表插入操作的时间复杂度为  $O(n)$



## 第二节 顺序表

### 四、顺序表的删除

- 顺序表的删除操作是指将顺序表的第*i*个数据元素删除，即将长度为*n*的顺序表：

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

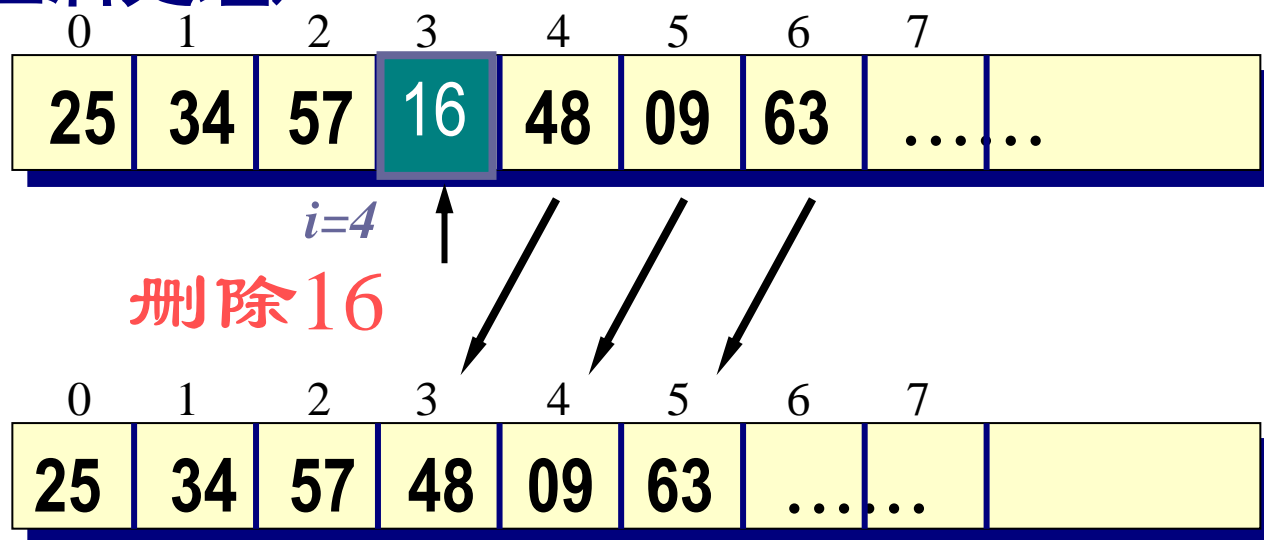
变成长度为*n-1*的顺序表：

$$(a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n)$$

## 第二节 顺序表

### 四、顺序表的删除

- 将第4个元素删除
- 需要将第5元素至最后元素 $n$  (共 $7-4$ ) 都向前移一位位置 (从前往后处理)



## 第二节 顺序表

### 四、顺序表的删除

类**SqList**增加删除方法: **int ListDelete(int i,int &e);**

```
int SqList::ListDelete(int i,int &e) //删除第i个元素, 用e返回该元素值
{
    if(i<1 || i>length)           //删除位置不正确, 返回0
        return 0;

    e = elem[i-1];                 //保存删除元素
    for(k=i; k<length; k++)        //第i+1之后的元素前移
        elem[k-1] = elem[k];
    length--;                      //长度减1
    return 1;                      //删除操作成功, 返回1
}
```

## 第二节 顺序表

### 四、顺序表的删除

- 在顺序表中删除一个元素，需要向前移动元素个数为： $n-i$
- 平均移动元素数为：

$$E_{dl} = \sum_{i=1}^n q_i \times (n-i)$$

## 第二节 顺序表

### 四、顺序表的删除

- 当删除位置等概率时,  $q_i=1/n$ , 因此:

$$E_{dl} = \sum_{i=1}^n [1/n] \times (n-i) = (n-1)/2$$

- 顺序表删除操作的时间复杂度为  $O(n)$

## 第二节 顺序表

### 五、顺序表的其它操作

- 查找第*i*个位置的元素值
- 查找元素所在位置
- 得到表长
- 置空表
- 销毁表(析构函数`~Sqlist()`)

## 第二节 顺序表

### 六、顺序表的优缺点

#### ■ 优点：

- ✓ 元素可以随机存取
- ✓ 元素位置可用一个简单、直观的公式表示并求取

#### ■ 缺点：

- ✓ 在作插入或删除操作时，需要移动大量元素

## 第二节 线性链表

### 一、链表

- 链表是线性表的链式存储表示
- 链表中逻辑关系相邻的元素不一定在存储位置上相连，用指针表示元素之间的邻接关系

线性表的链式存储表示主要有三种形式：

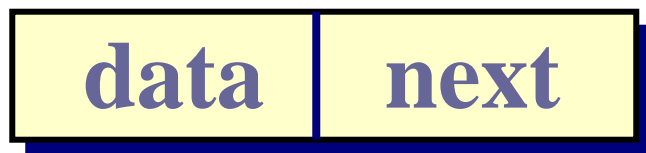
- 线性链表
- 循环链表
- 双向链表



## 第二节 线性链表

### 二、线性链表

- 线性链表的元素称为结点 (node)
- 结点除包含数据元素信息的数据域外，还包含指示直接后继的指针域

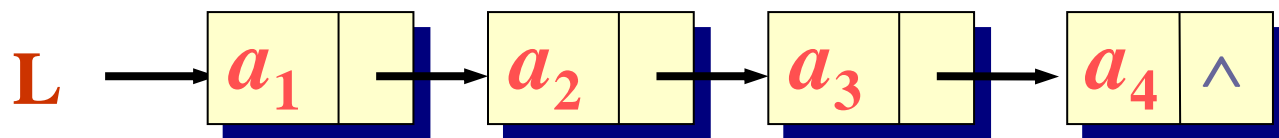


- 每个结点，在需要时动态生成，在删除时释放

## 第二节 线性链表

### 二、线性链表

- $N$ 个结点 ( $a_i$  ( $1 \leq i \leq n$ ) 的存储映像) 链结成一个链表, 即为线性表的链式存储结构.
- 链表的每个结点中只包含一个指针域, 故又称线性链表或**单链表**
- 线性链表可由头指针惟一确定



## 第二节 线性链表

### 三、线性链表的定义

data	next
------	------

#### ■ 定义结点类

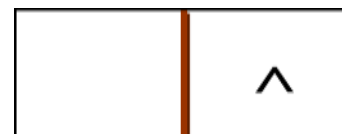
```
class LNode           //结点类
{
private:
    int          data;    //数据域
    LNode        *next;   //指针域

public:
    //构造函数
    LNode(int e=0,LNode *ptrNext=NULL):data(e),next(ptrNext){ }
    //析构函数
    ~LNode() {}
};
```

## 第二节 线性链表

### 三、线性链表的定义

头指针head

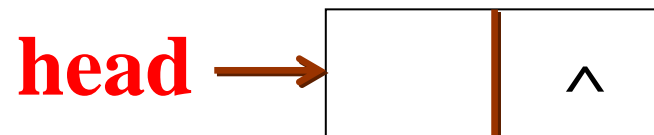


#### ■ 定义链表类

```
class LinkList          //单链表
{
private:
    LNode   *head;       //头指针
    int GetLen();        //计算表长
    LNode   *GetElem(int i); //查找第i个结点并返回指针
public:
    LinkList();          //构造函数
    int ListInsert(int i, int e); //插入
    int ListDelete(int i, int &e); //删除
    void ListDisplay();   //输出
    ~LinkList();         //析构释放结点
};
```

## 第二节 线性链表

### 四、构造函数初始化

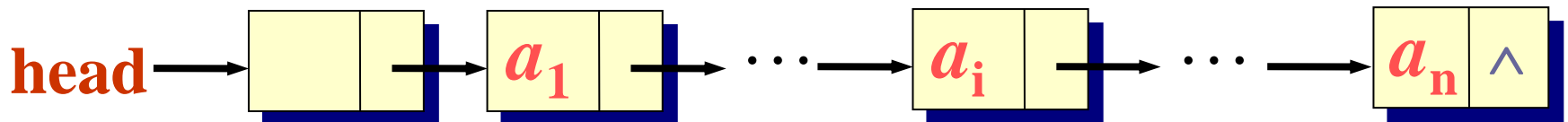


```
LinkList::LinkList()    //生成头结点
{
    head = new LNode;
}
```

## 第二节 线性链表

### 五、1. 找第 $i$ 个元素

- 在线性链表中找第 $i$ 个元素，并返回指针
- 从头结点开始，顺链一步步查找
- 查找第 $i$ 个数据元素的基本操作为：移动指针，比较 $k$ 和 $i$ 。（ $k$ 为当前指针所指向的结点序号）



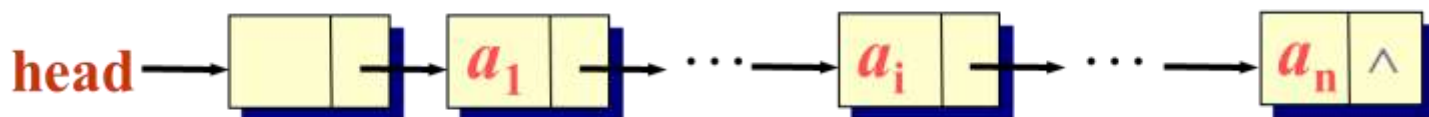
## 第二节 线性链表

### 五、1. 找第*i*个元素

```
ListNode *LinkList::GetElem(int i)
{
    ListNode *p;
    int k;

    for(p=head, k=0; k<i&& p->next; k++) //初始p指向头结点, 第0个结点
        p = p->next;                    //p的后继存在, p后移, k加1

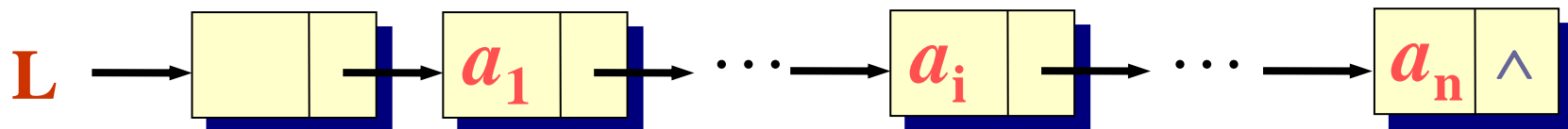
    if(k==i)                             //存在第i个结点, 返回指针
        return p;
    return NULL;                         //不存在第i个结点, 返回空
}
```



## 第二节 线性链表

### 五、1. 找第*i*个元素

- 算法时间复杂度主要取决于for循环中的语句频度
- 频度与被查找元素在单链表中的位置有关
- 若 $1 \leq i \leq n$ ，则频度为 $i-1$ ，否则为 $n$
- 因此时间复杂度为 $O(n)$



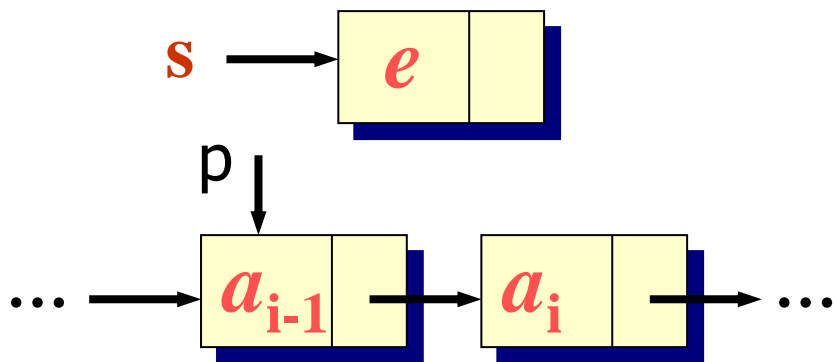


## 第二节 线性链表

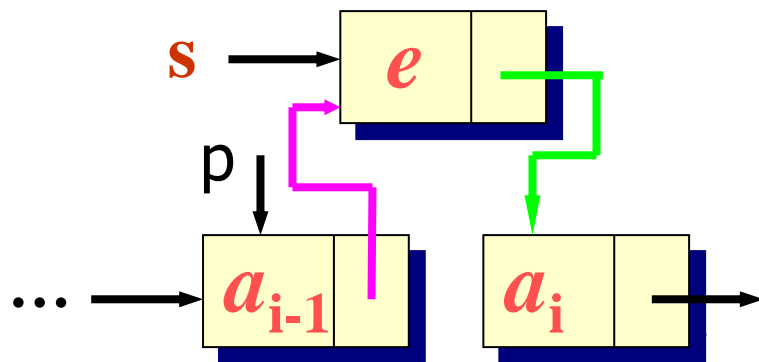
### 五、2. 线性链表的插入

- 在线性链表的第 $i-1$ 元素与第 $i$ 元素之间插入一个新元素

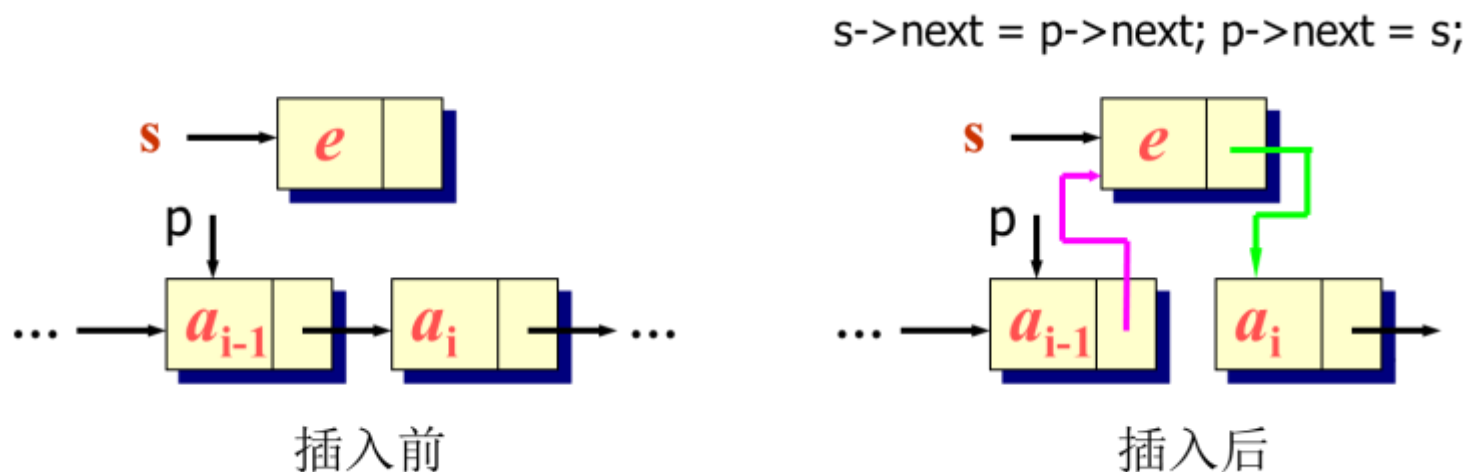
$s \rightarrow \text{next} = p \rightarrow \text{next}; p \rightarrow \text{next} = s;$



插入前



插入后



可见，在链表中插入结点只需要修改指针。若要在第  $i$  个结点之前插入元素，修改的是第  $i-1$  个结点的指针。

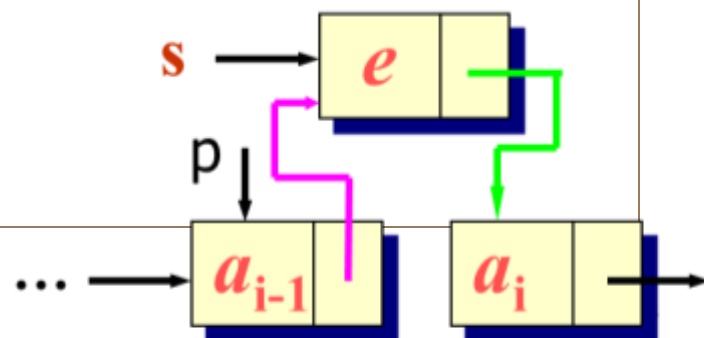
因此，在单链表中第  $i$  个结点之前进行插入的**基本操作为**：找到线性表中第  $i-1$  个结点，创建新结点  $s$ ，然后修改第  $i-1$  个结点和  $s$  结点的后继指针。

## 第二节 线性链表

### 五、2. 线性链表的插入

```
int LinkList::ListInsert(int i, int e)    //第i个位置插入e
{
    LNode *p = GetElem(i-1);             //p指向第i-1结点
    if(!p)                                 //若没有第i-1结点, 即p为空, 则不能插入
        return 0;

    LNode *s = new LNode(e, p->next);    //生成新结点s, 插入p结点之后
    p->next = s;
    return 1;
}
```



插入后

## 第二节 线性链表

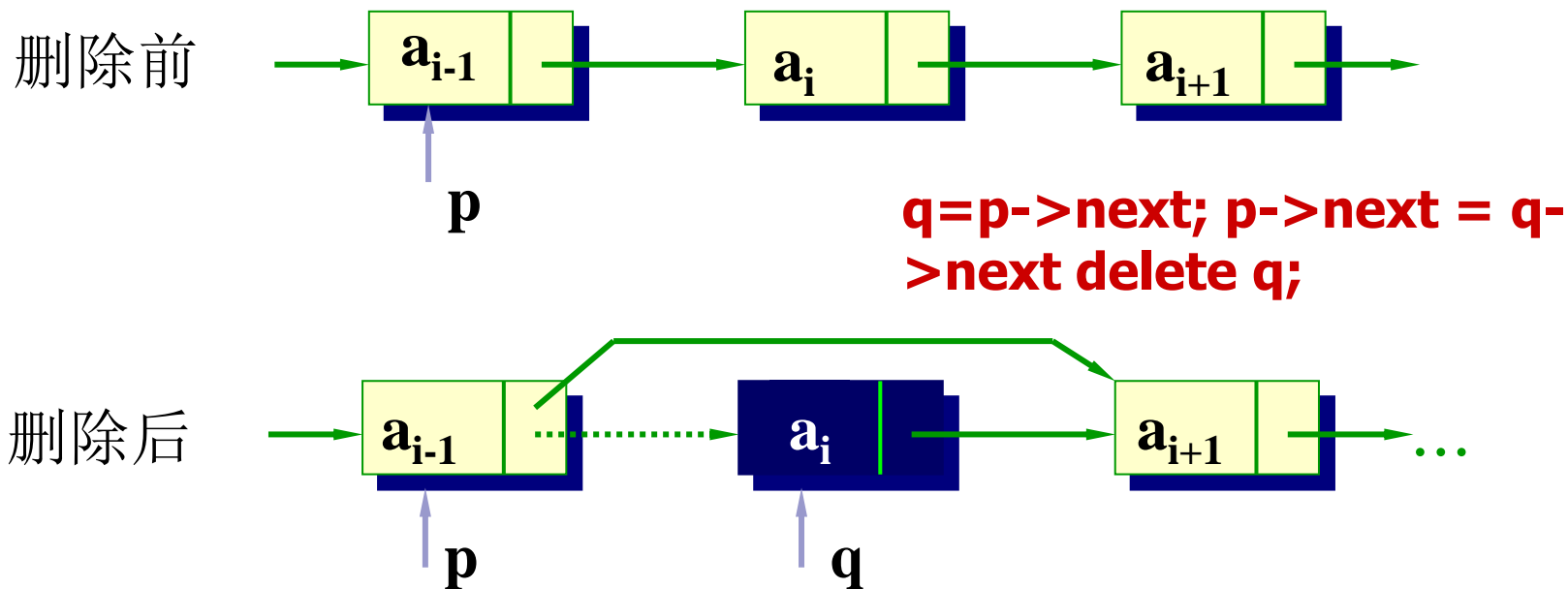
### 五、2. 线性链表的插入

- 算法时间复杂度主要取决于GetElem的时间复杂度
- GetElem的时间复杂度为 $O(n)$
- 因此线性链表插入的时间复杂度为 $O(n)$

## 第二节 线性链表

### 五、3. 线性链表的删除

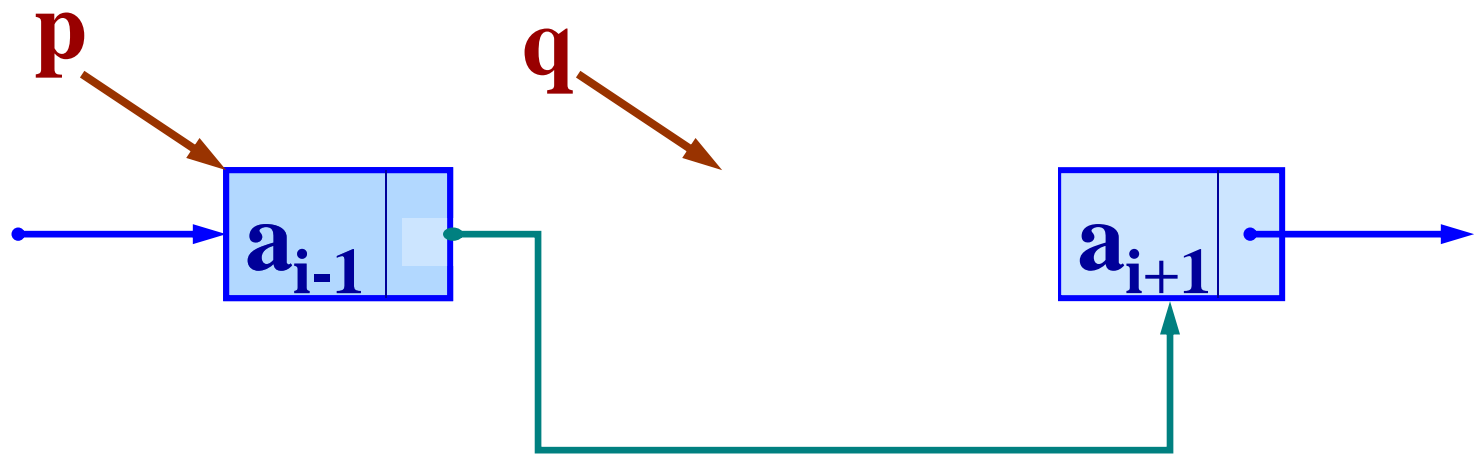
#### ■ 将线性链表的第 $i$ 元素删除



在单链表中删除第  $i$  个结点的基本操作为:找到线性表中第 $i-1$ 个结点, 修改其指向后继的指针。

$q = p \rightarrow \text{next};$   $p \rightarrow \text{next} = q \rightarrow \text{next};$

$e = q \rightarrow \text{data};$   $\text{delete } q;$

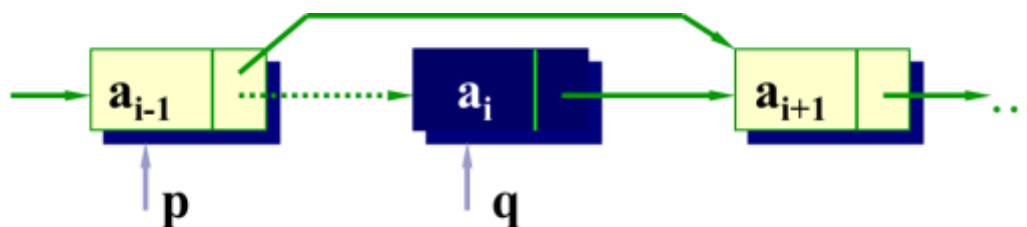


## 第二节 线性链表

### 五、3. 线性链表的删除

```
int LinkList::ListDelete(int i, int &e) //删除
{
    LNode *p = GetElem(i-1);
    if(!p || !p->next)
        return 0;

    LNode *q;
    q = p->next;
    e = q->data;
    p->next = q->next;
    delete q;
    return 1;
}
```



## 第二节 线性链表

### 五、3. 线性链表的删除

- 同样，算法时间复杂度主要取决于GetElem的时间复杂度
- 因此线性链表删除元素的时间复杂度为 $O(n)$



## 第二节 线性链表

### 五、4. 线性链表的删除

- 计算表长
- 输出表
- 置空表
- 析构撤销表

## 第二节 线性链表

### 五、4. 线性链表的删除

#### ■ 析构撤销表

```
~LinkList::LinkList()    //析构释放各结点
{
    LNode *p,*q;

    for(p=head; p; )    //循环单链表
    {
        q=p, p=p->next;    //q指向p结点, p后移
        delete q;          //释放q
    }
}
```

## 第二节 线性链表

### 六. 线性链表的创建

```
int LinkList::ListInsert(int i,int e)    //第i个位置插入e
{
    LNode *p = GetElem(i-1);            //p指向第i-1结点
    if(!p)                                //若没有第i-1结点，即p为空，则不能插入
        return 0;

    LNode *s = new LNode(e,p->next);    //生成新结点s，插入p结点之后
    p->next = s;
    return 1;
}
```

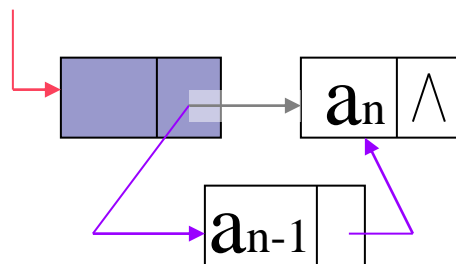
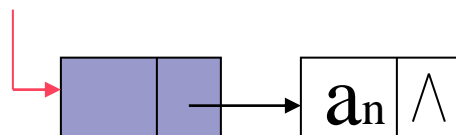
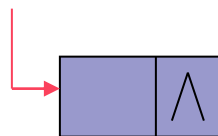
生成

插入n个元素的时间复杂度 $O(n^2)$

## 第二节 线性链表

### 六. 线性链表的创建—头插法

头插法，即表头不断插入新结点。逆序输入数据值。



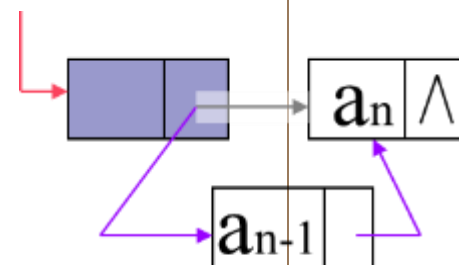
## 第二节 线性链表

### 六. 线性链表的创建—头插法

```
void LinkList::CreateListInHead(int n)    //头插法
{
    int    i,e;
    LNode  *s;

    for(i=0; i<n; i++)
    {
        cin>>e;

        s = new LNode(e, head->next); //生成新结点s
        head->next = s;                //s链在头结点之后
    }
}
```



插入 $n$ 个元素的时间复杂度 $O(n)$

## 第二节 线性链表

### 六. 线性链表的创建—尾插法

在表尾不断插入新结点。按链表序输入数据值。  
为记录尾结点，增加一个尾指针tail，  
指向最后一个结点。

## 第二节 线性链表

### 六. 线性链表的创建—尾插法

```
void LinkList::CreateListInTail(int n)  //尾插法
{
    int i, e;
    LNode *tail = head, *s;           //初始，尾指针指向头结点，表中无数据

    for(i=0; i<n; i++)
    {
        cin>>e;

        s = new LNode(e);             //创建结点s, next为空
        tail->next = s;                //s链在尾结点之后
        tail = s;                     //新的尾结点是s
    }
}
```

插入n个元素的时间复杂度 $O(n)$

# 单链表的合并

## ■ 将两个有序链表合并为一个有序链表

```
LinkList &LinkList::MergeList(LinkList &LB)    //this和LB两个有序表合并，并返回
{
    LinkList    *LC = new LinkList;           //结果表

    LNode *pa = head->next;                    //pa,pb分别指向this和LB的第一个结点
    LNode *pb = LB.head->next;
    LNode *pc = LC->head;                      //pc指向LC的头结点

    while (pa&&pb)                             //pa,pb不空
    {
        if (pa->data <= pb->data)               //pc之后链接pa
        {
            pc->next = pa;
            pa = pa->next;                      //pa后移
        }
        else if (pa->data > pb->data)           //pc之后链接pb
        {
            pc->next = pb;
            pb = pb->next;                      //pb后移
        }
        pc = pc->next;                          //pc后移
    }
    pc->next = pa?pa:pb?pb:NULL;                //链接pa、pb剩余链表
    head->next = NULL;                          //this置空表
    LB.head->next = NULL;
    return *LC;
}
```



# 单链表的合并

- 将两个有序链表合并为一个有序链表



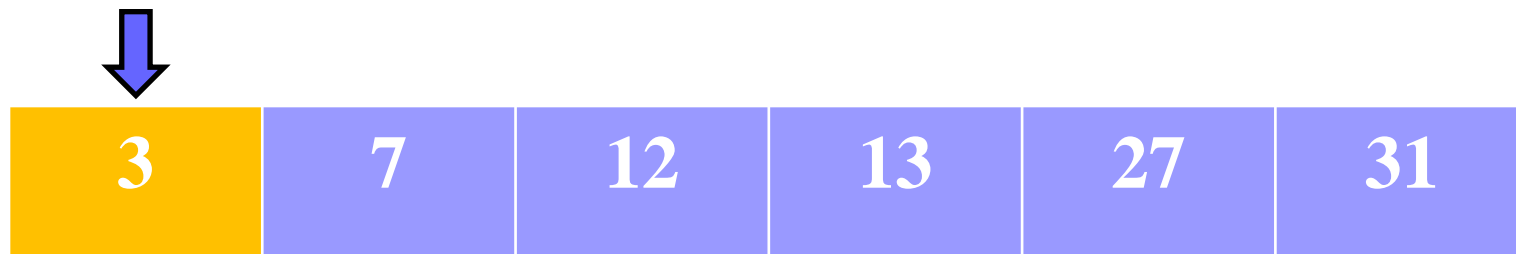
3	7	12	13	27	31
---	---	----	----	----	----

5	9	10	17	26
---	---	----	----	----



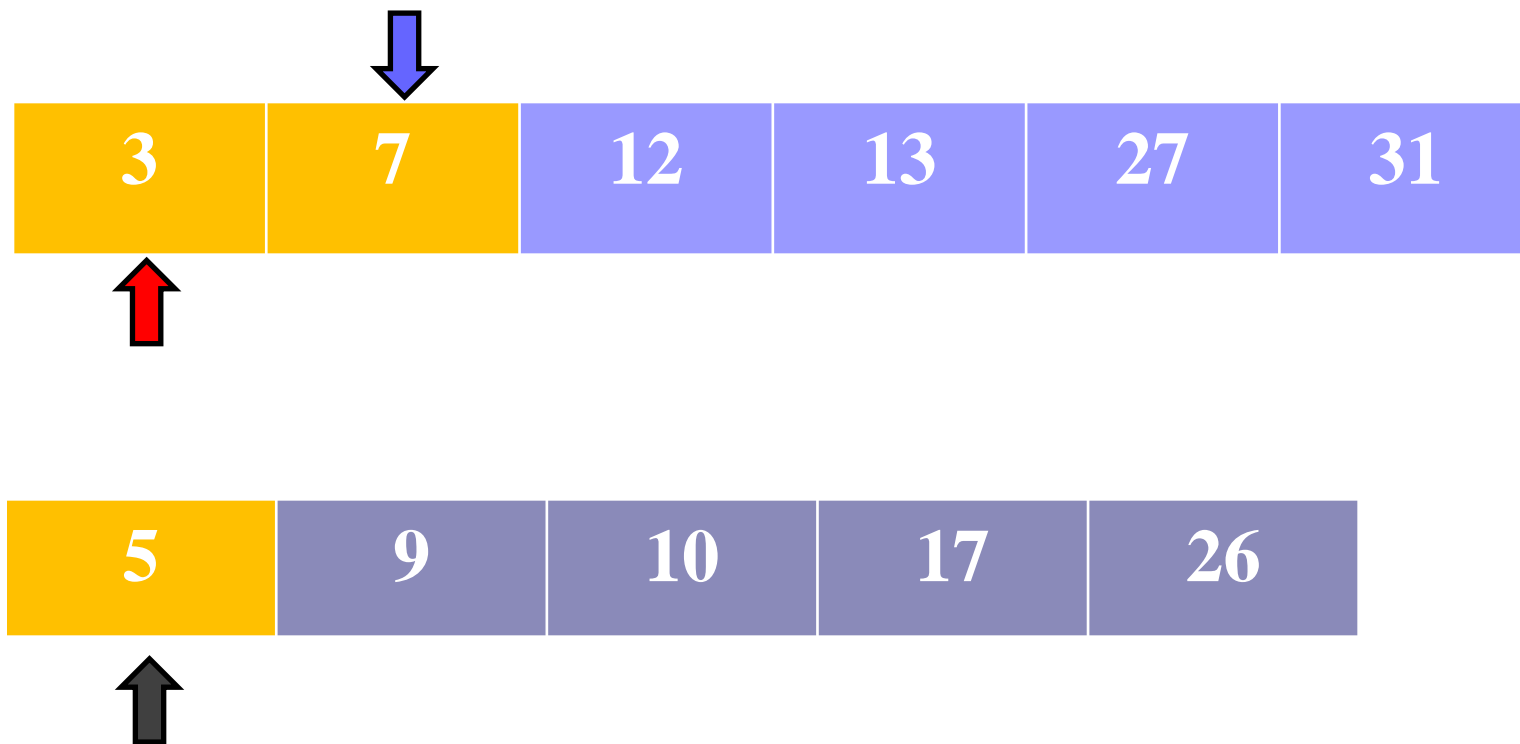
## 单链表的合并

- 将两个有序链表合并为一个有序链表



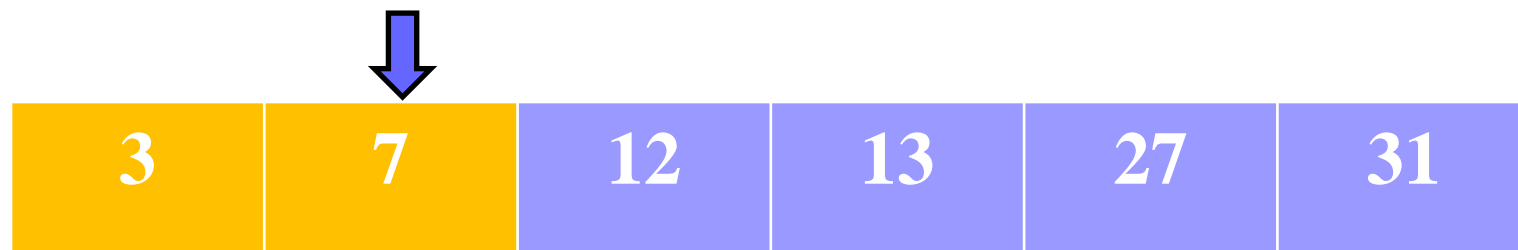
# 单链表的合并

- 将两个有序链表合并为一个有序链表



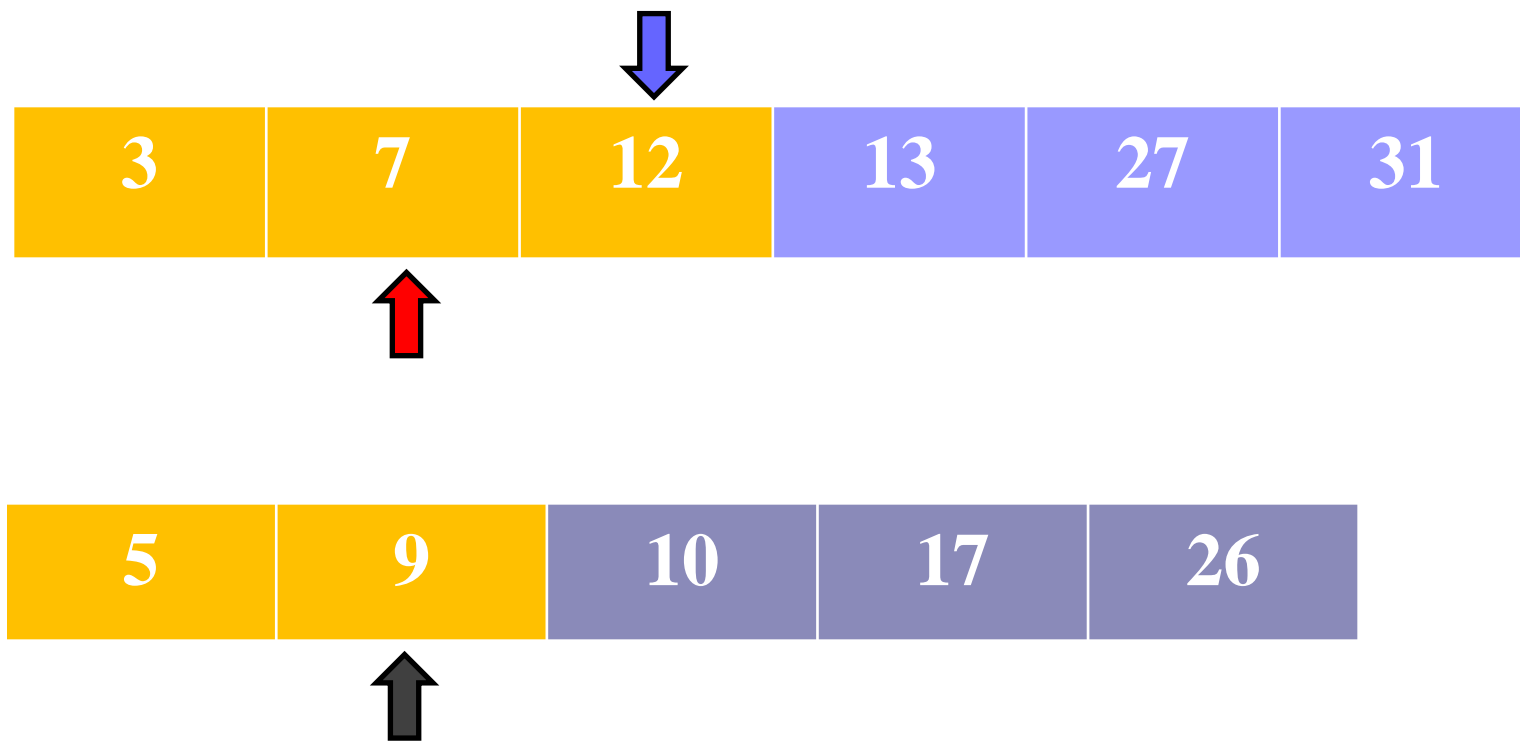
# 单链表的合并

- 将两个有序链表合并为一个有序链表



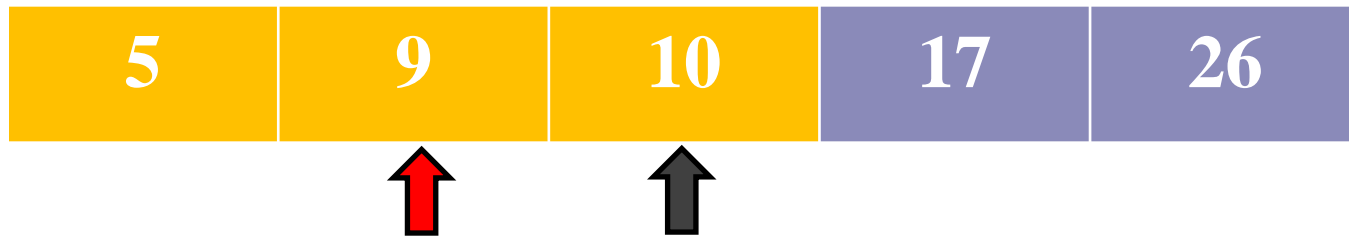
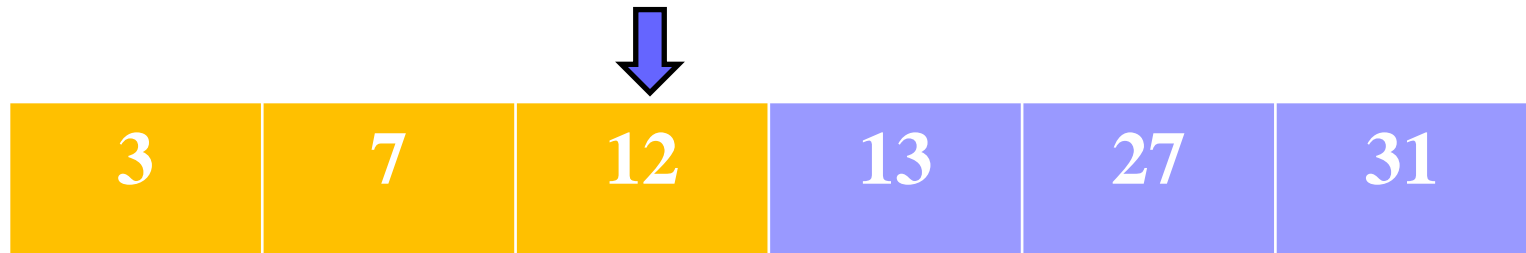
# 单链表的合并

- 将两个有序链表合并为一个有序链表



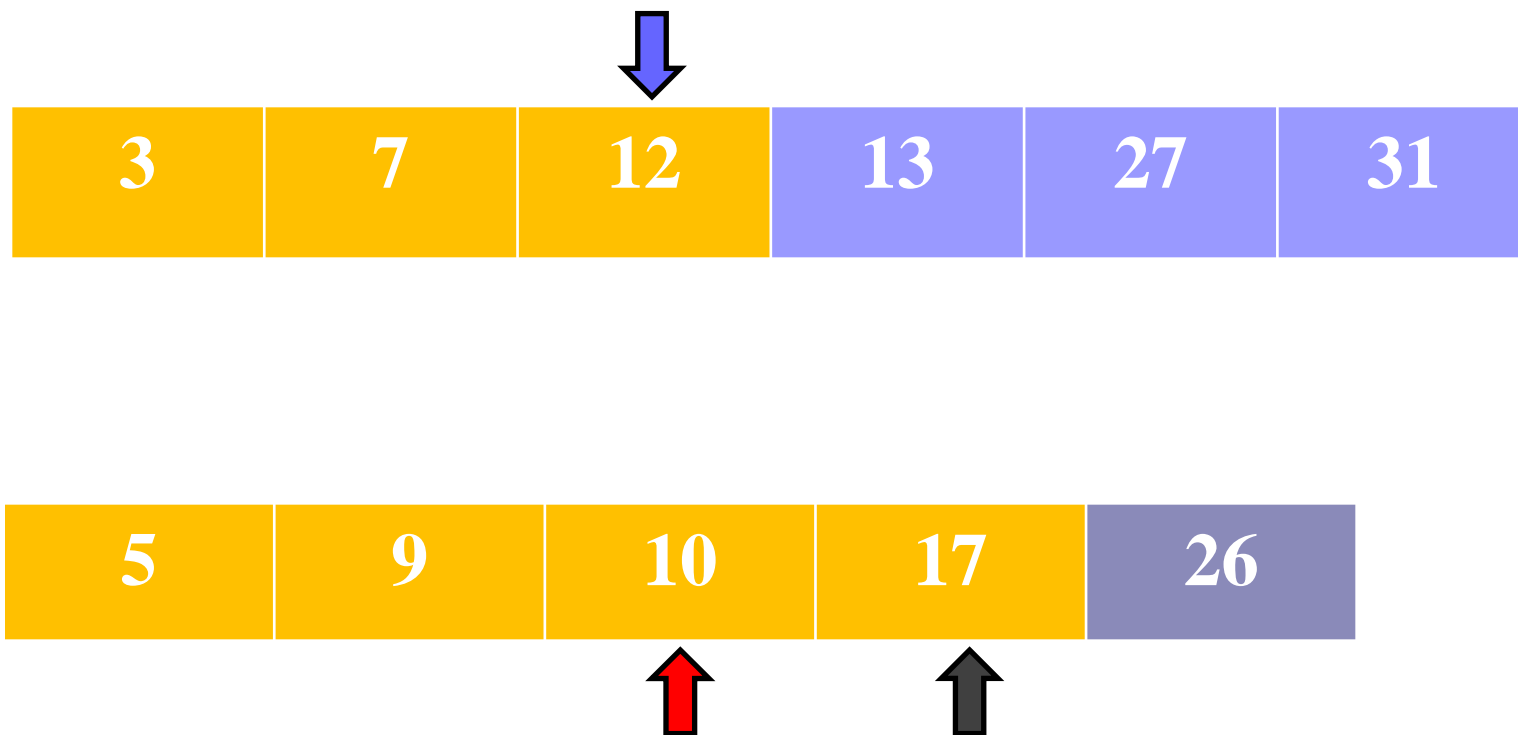
# 单链表的合并

- 将两个有序链表合并为一个有序链表



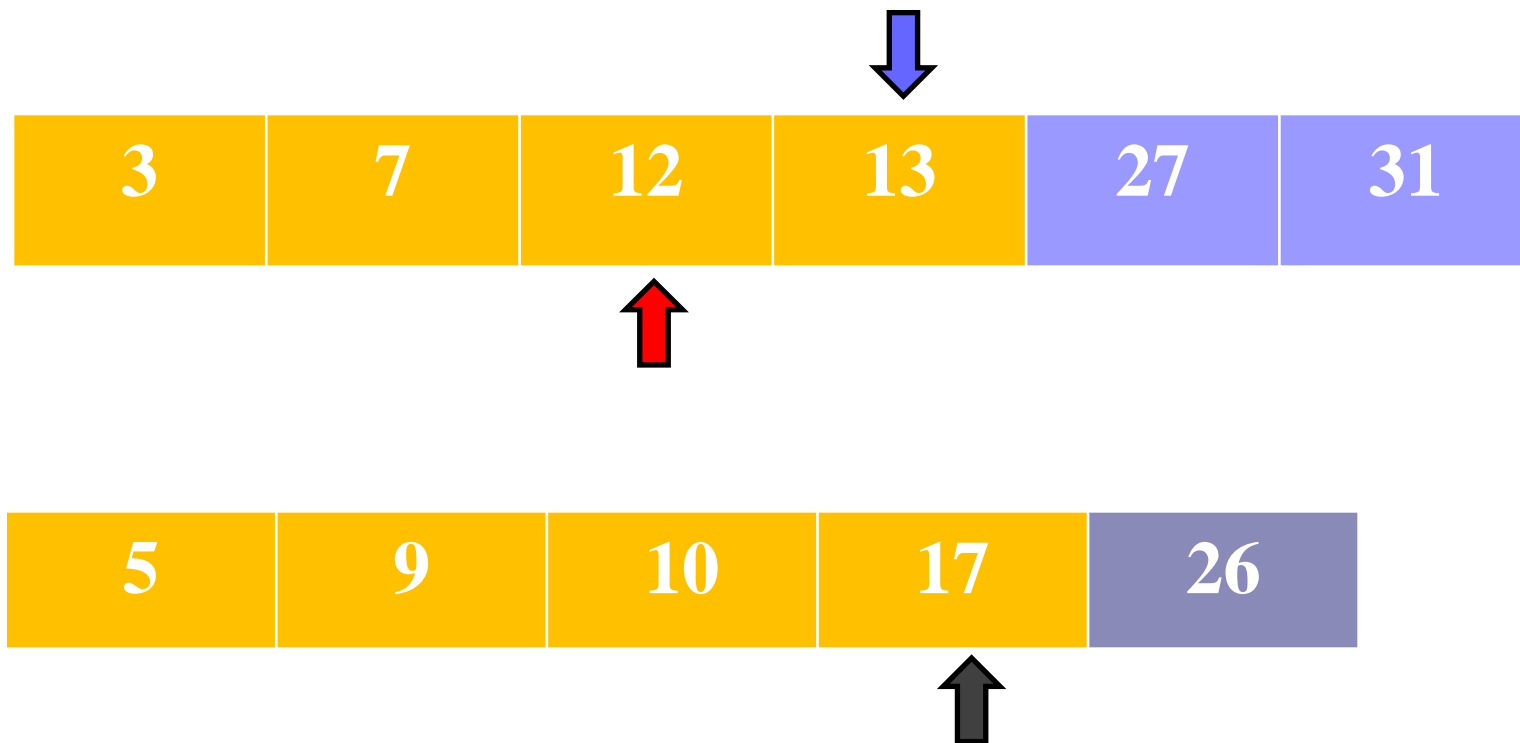
# 单链表的合并

- 将两个有序链表合并为一个有序链表



# 单链表的合并

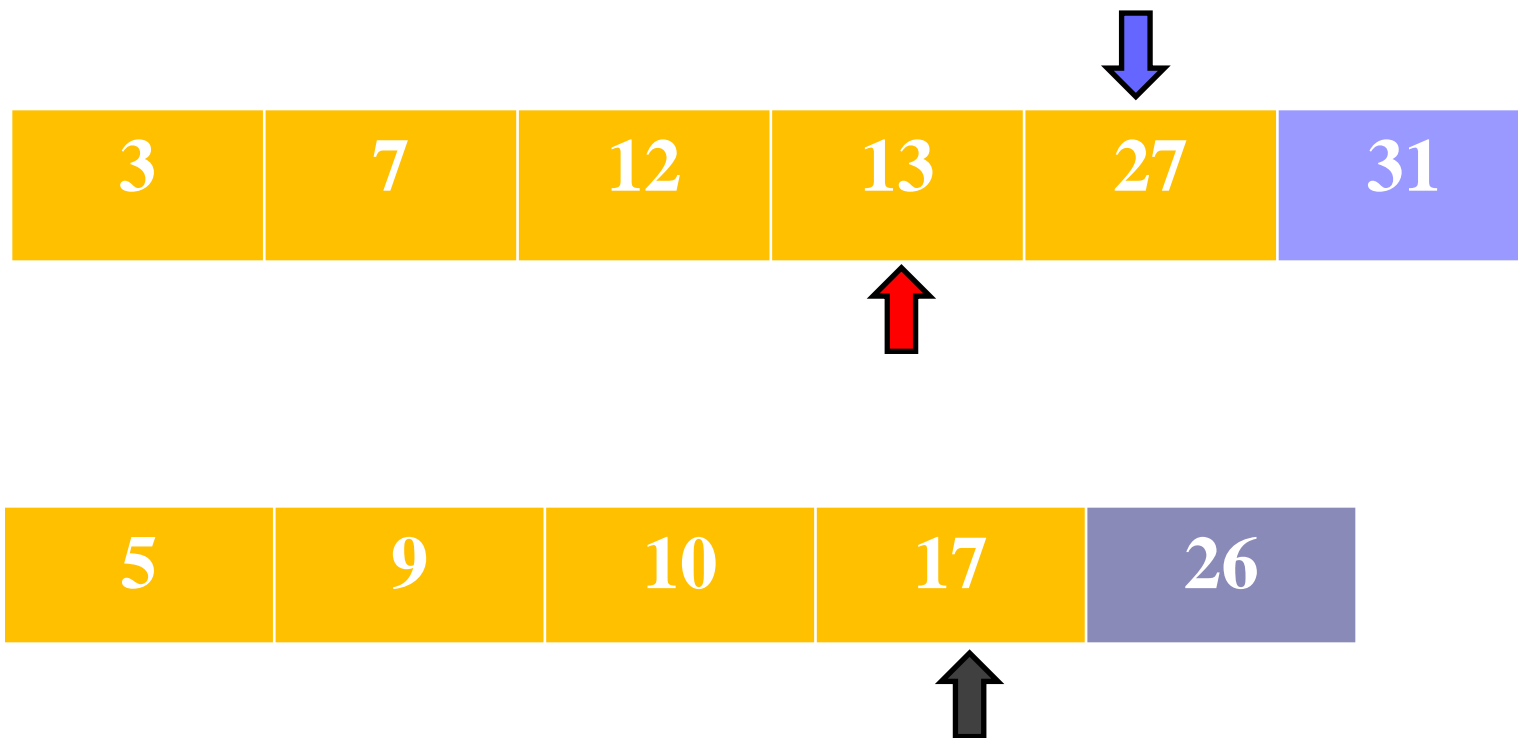
- 将两个有序链表合并为一个有序链表





## 单链表的合并

- 将两个有序链表合并为一个有序链表



# 单链表的合并

- 将两个有序链表合并为一个有序链表



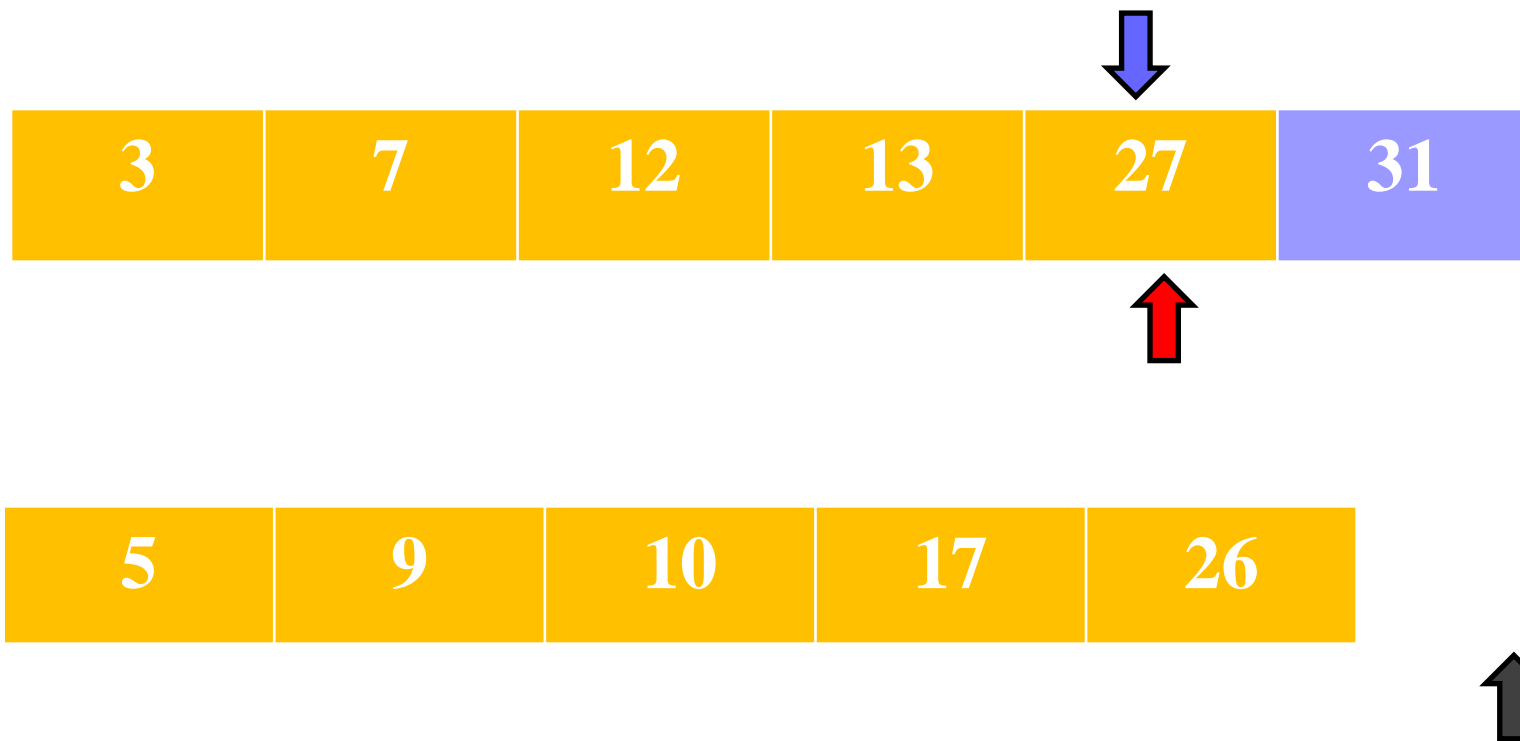
# 单链表的合并

- 将两个有序链表合并为一个有序链表



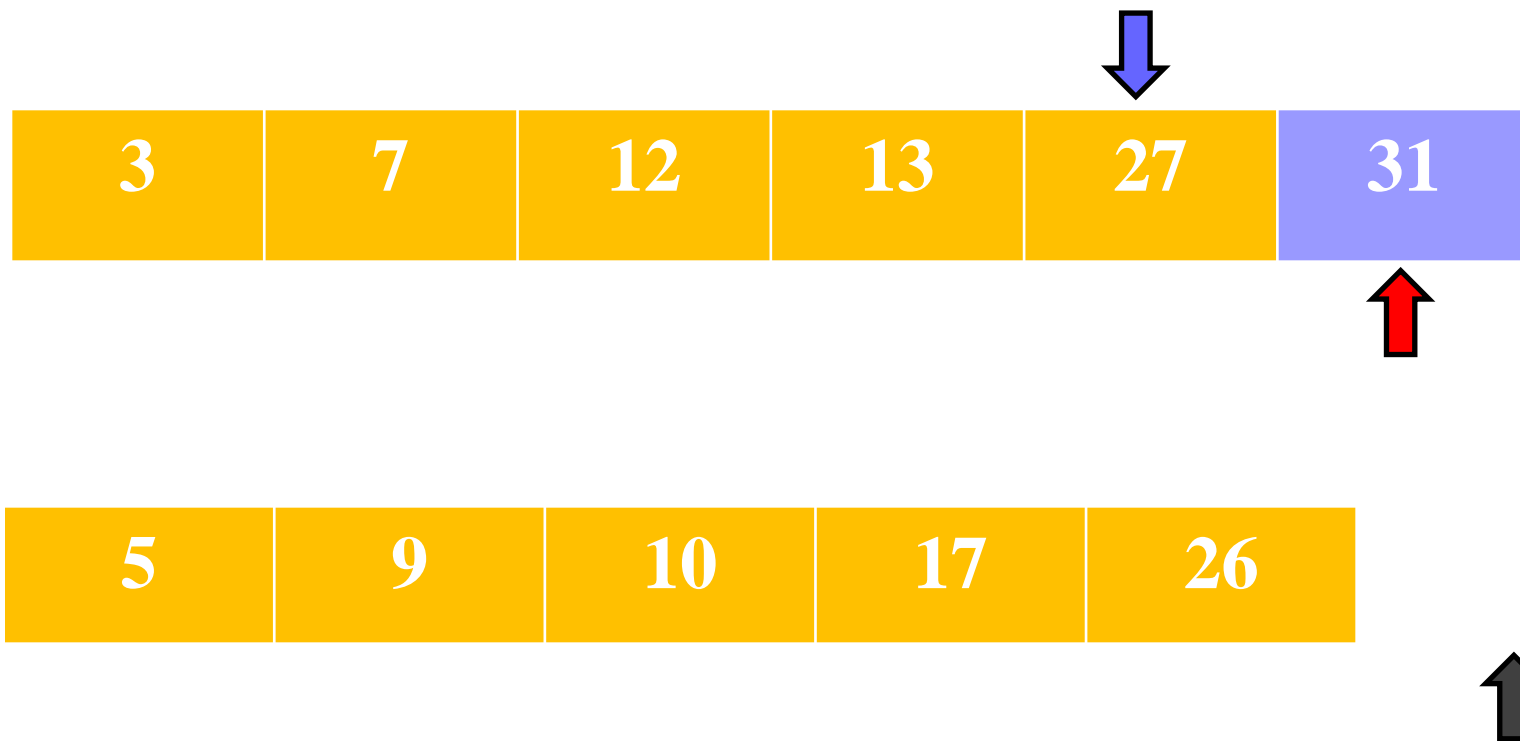
## 单链表的合并

- 将两个有序链表合并为一个有序链表



# 单链表的合并

- 将两个有序链表合并为一个有序链表



# 单链表的合并

## ■ 将两个有序链表合并为一个有序链表

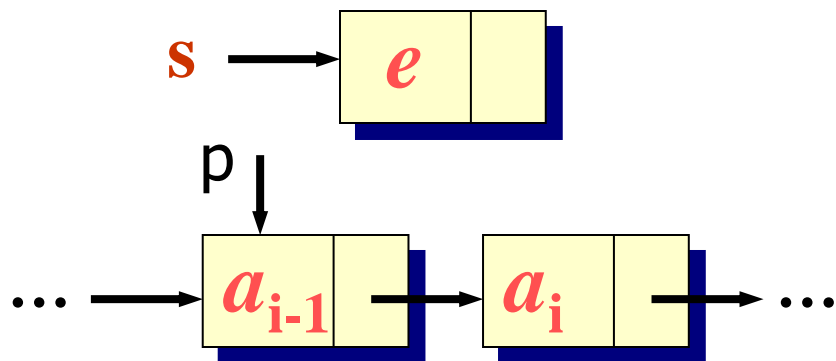
```
LinkList &LinkList::MergeList(LinkList &LB)    //this和LB两个有序表合并，并返回
{
    LinkList    *LC = new LinkList;           //结果表

    LNode *pa = head->next;                     //pa,pb分别指向this和LB的第一个结点
    LNode *pb = LB.head->next;
    LNode *pc = LC->head;                       //pc指向LC的头结点

    while (pa&&pb)                               //pa,pb不空
    {
        if (pa->data <= pb->data)                 //pc之后链接pa
        {
            pc->next = pa;
            pa = pa->next;                         //pa后移
        }
        else if (pa->data > pb->data)             //pc之后链接pb
        {
            pc->next = pb;
            pb = pb->next;                         //pb后移
        }
        pc = pc->next;                             //pc后移
    }
    pc->next = pa?pa:pb?pb:NULL;                 //链接pa、pb剩余链表
    head->next = NULL;                          //this置空表
    LB.head->next = NULL;
    return *LC;
}
```

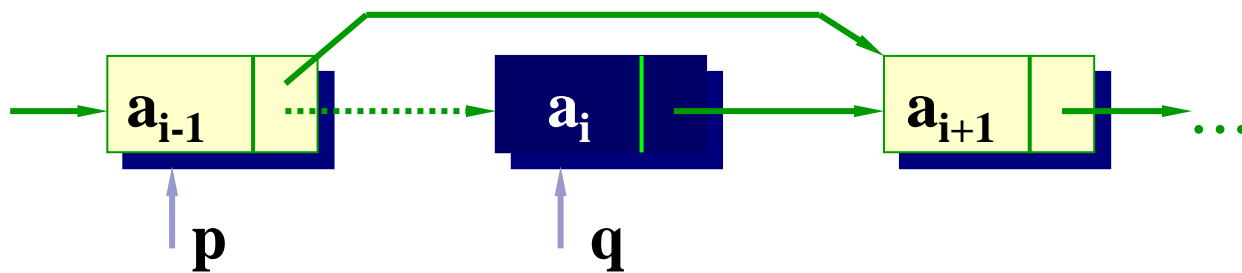
# 练习

## 1. 线性链表的插入



插入前

## 2. 线性链表的删除



## 第二节 线性链表

### 七、静态链表

- 线性链表也可以采用静态数组实现
- 与顺序表有两点不同：
  - 1、每个元素包括数据域和指针域
  - 2、元素的逻辑关系由指针确定

	0	1	2	3	4	5	6	7	8	9	10	...
数据			D	C	F		B	A		E		...
指针	7	5	9	2	^	8	3	6	10	4	11	



## 静态链表

- 可借用一维数组来描述线性链表
- 类型说明

```
#define MAXSIZE 100
```

```
class node{
```

```
    ElemType data;
```

```
    int cur; //游标，替代next指针，指示下一个结点的位置
```

```
};
```

```
class slist{
```

```
    node list[MAXSIZE];
```

```
public:
```

```
    ...
```

```
};
```

## 静态链表示例

0		1
1	Zhao	2
2	Qian	3
3	Sun	4
4	Li	5
5	Zhou	6
6	Wu	0
7		

修改前

0		1
1	Zhao	2
2	Qian	3
<b>3</b>	Sun	7
4	Li	<b>5</b>
5	Zhou	6
6	Wu	0
7	Shi	4

在第3个结点后  
增加结点“shi”

0		1
1	Zhao	3
<b>2</b>	Qian	<b>3</b>
3	Sun	7
4	Li	5
5	Zhou	6
6	Wu	0
7	Shi	4

删除结点“qian”

## 第二节 线性链表

### 七、静态链表

- 与单链表区别：

- 1、静态链表暂时不用的结点，链成一个备用链表
- 2、插入时，从备用链表中申请结点
- 3、删除结点时，将结点放入备用链表

## 静态链表实现策略

- 备用链表
  - 为了辨明数组中哪些分量未被使用，将所有未被使用过以及被删除的分量用游标链成备用链。
- 一种策略（浪费两个分量）
  - 将数组的第一个分量用来做备用链表的头结点。串起整个备用分量。
  - 将数组的第二个分量用来做静态链表的头结点。

# 静态链表实现示例（初始化备用链）

0		1
1		2
2		3
3		4
4		5
5		6
6		7
7		0

**For (i=0;i<MAXSIZE-1;i++)**

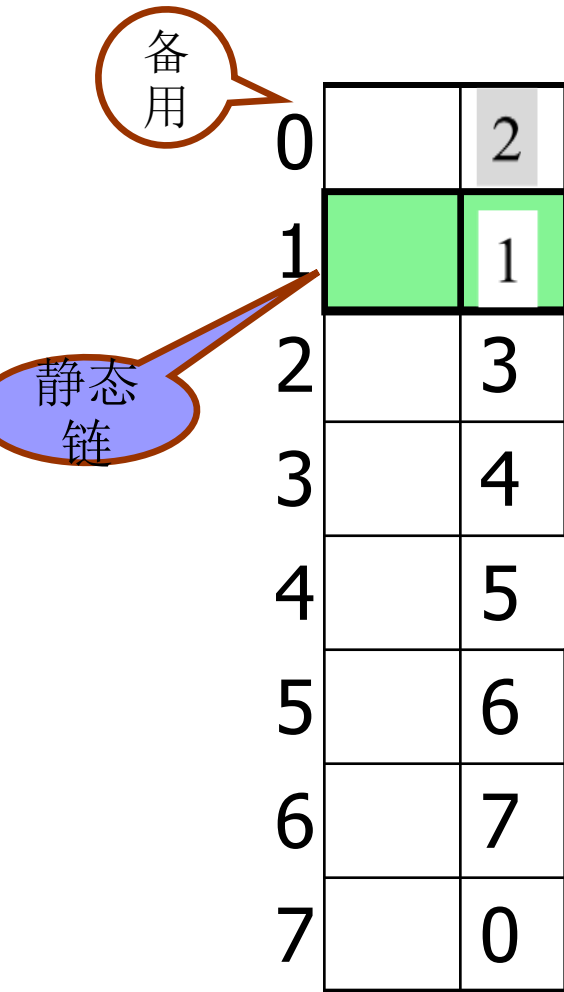
**list[i].cur =i+1;**

**list[MAXSIZE-1]=0**

**//建立备用链表,**

**//list[0]为其头结点**

# 静态链表实现示例（分配头结点）



0		2
1		1
2		3
3		4
4		5
5		6
6		7
7		0

$k = \text{List}[0].\text{cur};$        $// k=1$

$//$  从备用链上取得一个备用结点的下标;

$\text{List}[0].\text{cur} = \text{list}[k].\text{cur};$

$//$  修改备用链头结点的游标指向下一个  
 $//$  备用结点。

$\text{List}[k].\text{cur} = 1;$

$//k$  为当前获得的结点的下标，在此为静态链表分配头结点，并将链表末尾的游标初始化为头结点下标。

## 静态链表实现示例（头插法）

0		3
1		2
2	Z	1
3		4
4		5
5		6
6		7
7		0

`k = list[0].cur;`

`//k=2,获得一个备用结点,`

`List[0].cur=list[k].cur;`

`// 并修改备用链头结点的游标`

`List[k].data = 'z';` `//data='z';`

`List[k].cur=list[1].cur;`

`List[1].cur=k;`

`// 在静态链表中插入第一个元素`

## 静态链表实现示例（删除结点）

0		5
1		4
2	z	1
3	h	2
4	a	3
5		6
6		7
7		0

// 删除静态链表中的第i个结点,假设它在数组中的下标为p,前一个结点的下标为q。

List[q].cur=list[p].cur;

//修改q的cur指向p的cur。

List[p].cur=list[0].cur;

//记录下一个可用节点

List[0].cur=p;

//将p结点插入到备用链表的头结点之后。

0		3
1		4
2	z	1
3	h	5
4	a	2
5		6
6		7
7		0

← p

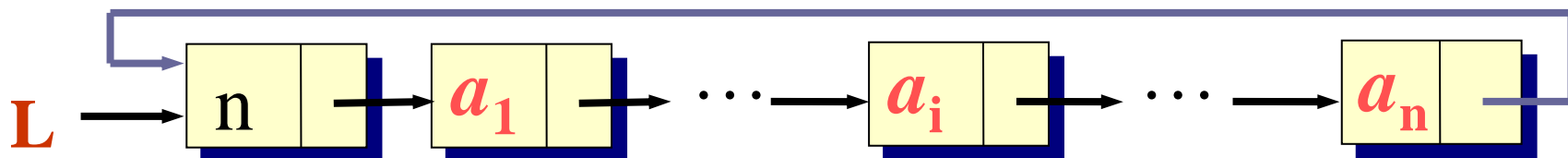
← q



## 第三节 循环链表

### 一、循环链表

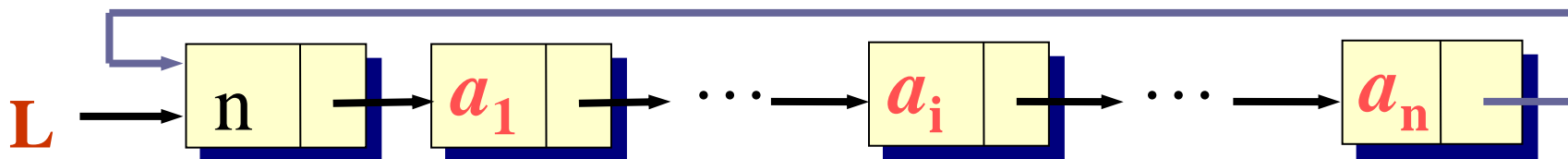
- 循环链表是一种特殊的线性链表
- 循环链表中最后一个结点的指针域指向头结点，整个链表形成一个环。



## 第三节 循环链表

### 二、查找、插入和删除

- 在循环链表中查找指定元素，插入一个结点或删除一个结点的操作与线性链表基本一致
- 差别仅在于算法中的循环条件不是 $p \rightarrow \text{next}$ 或 $p$ 是否为空( $\wedge$ )，而是它们是否等于头指针( $L$ )



## 第四节 双向链表

### 一、双向链表

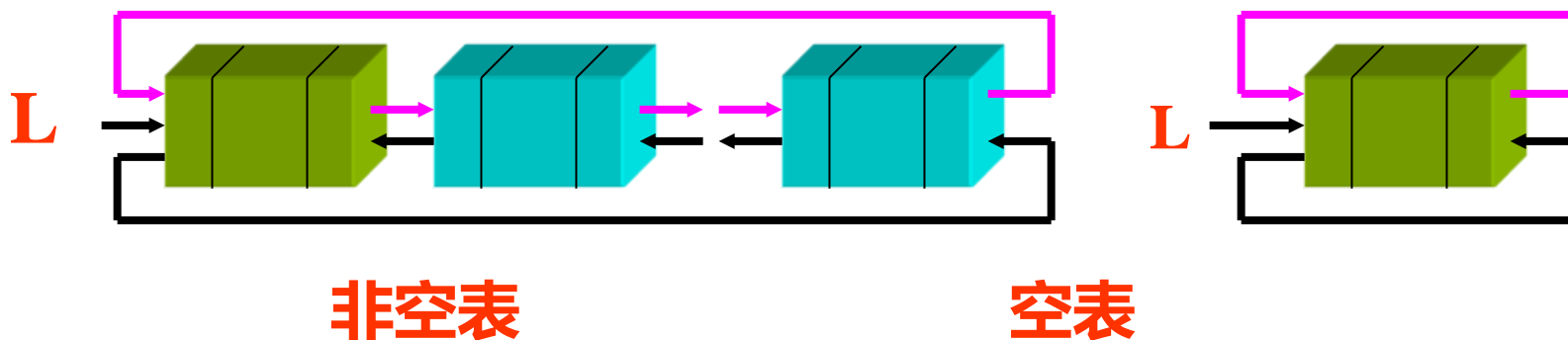
- 双向链表也是一种特殊的线性链表
- 双向链表中每个结点有两个指针，一个指针指向直接后继(next)，另一个指向直接前驱(prior)



## 第四节 双向链表

### 二、双向循环链表

- 双向循环链表中存在两个环(一个是直接后继环(红), 另一个是直接前驱环(蓝))



## 第四节 双向链表

### 三、双向链表的定义

#### ■ 定义一个双向链表的结点

```
class DuLNode {  
    ElemType    data;  
    DuLNode    *prior;  
    DuLNode    *next;  
};
```

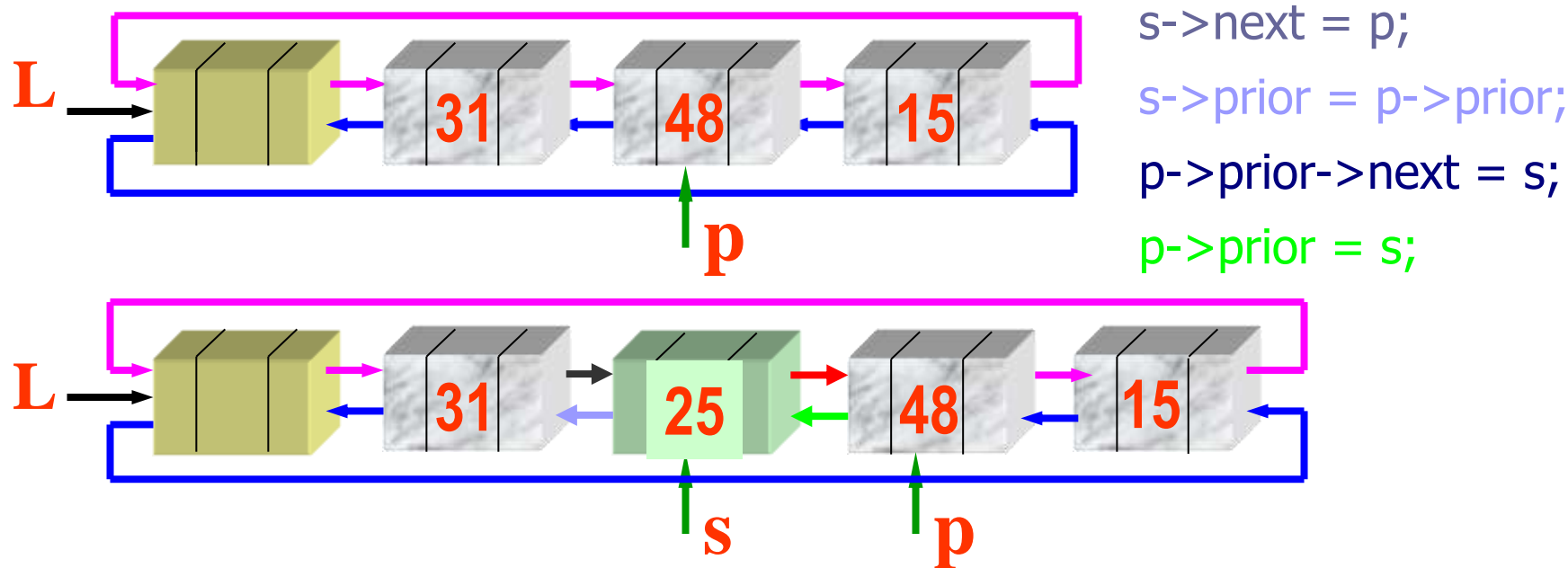
对于任何一个中间结点有：

$$p = p \rightarrow \text{next} \rightarrow \text{prior}$$
$$p = p \rightarrow \text{prior} \rightarrow \text{next}$$

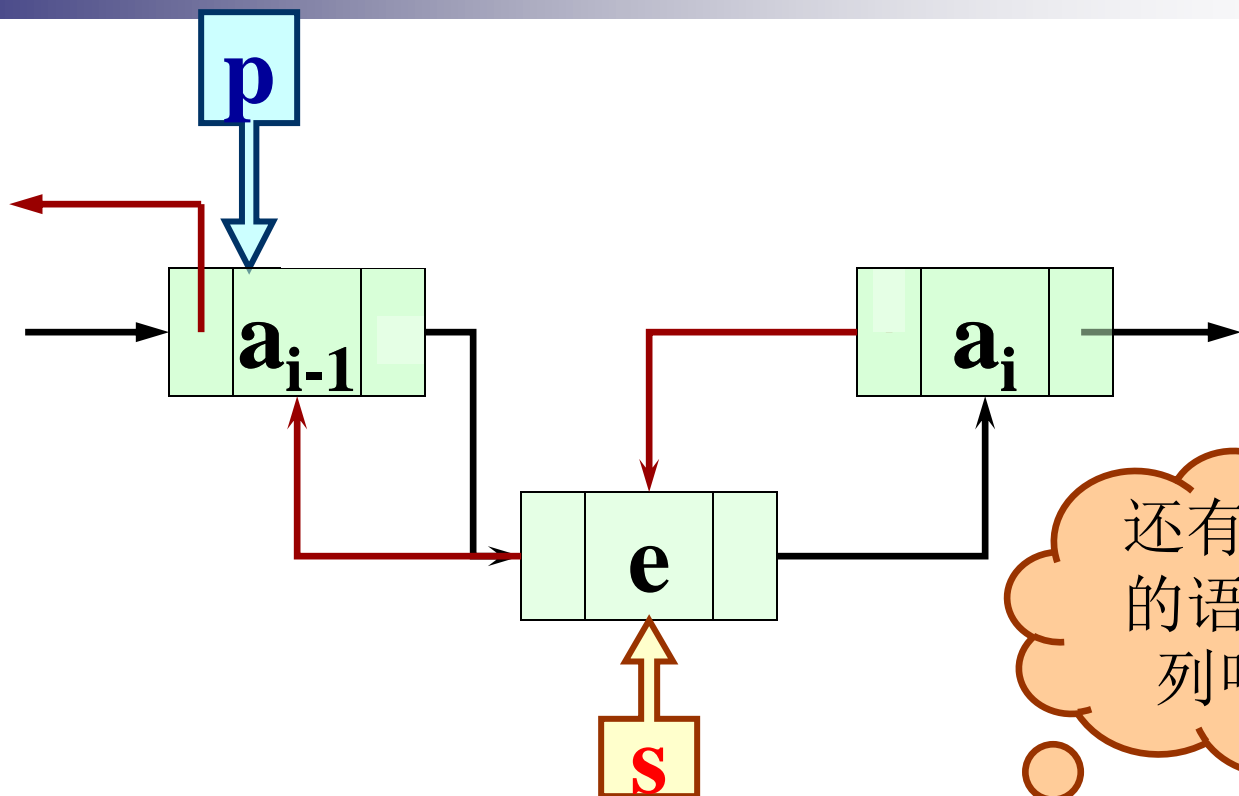

## 第四节 双向链表

### 四、双向链表的插入

- 双向链表的插入操作需要改变两个方向的指针



# 插入



还有其他的  
语句序列吗？

$s \rightarrow \text{next} = p \rightarrow \text{next};$

$p \rightarrow \text{next} = s;$

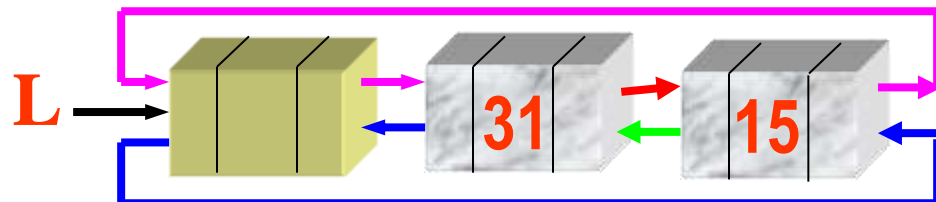
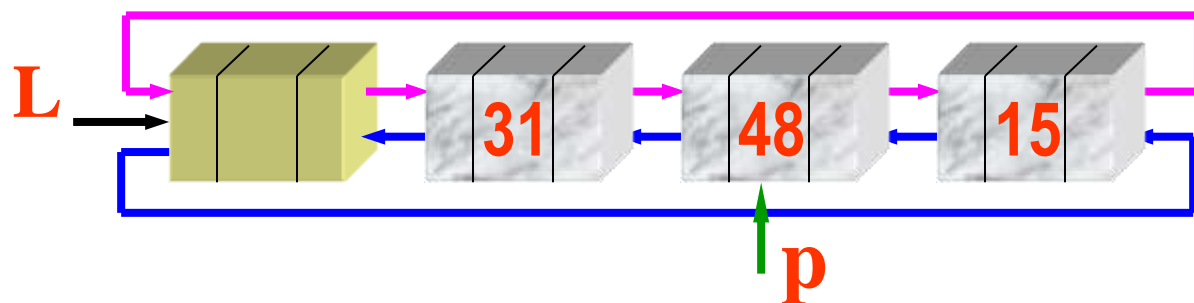
$s \rightarrow \text{next} \rightarrow \text{prior} = s;$

$s \rightarrow \text{prior} = p;$

## 第四节 双向链表

### 四、双向链表的删除

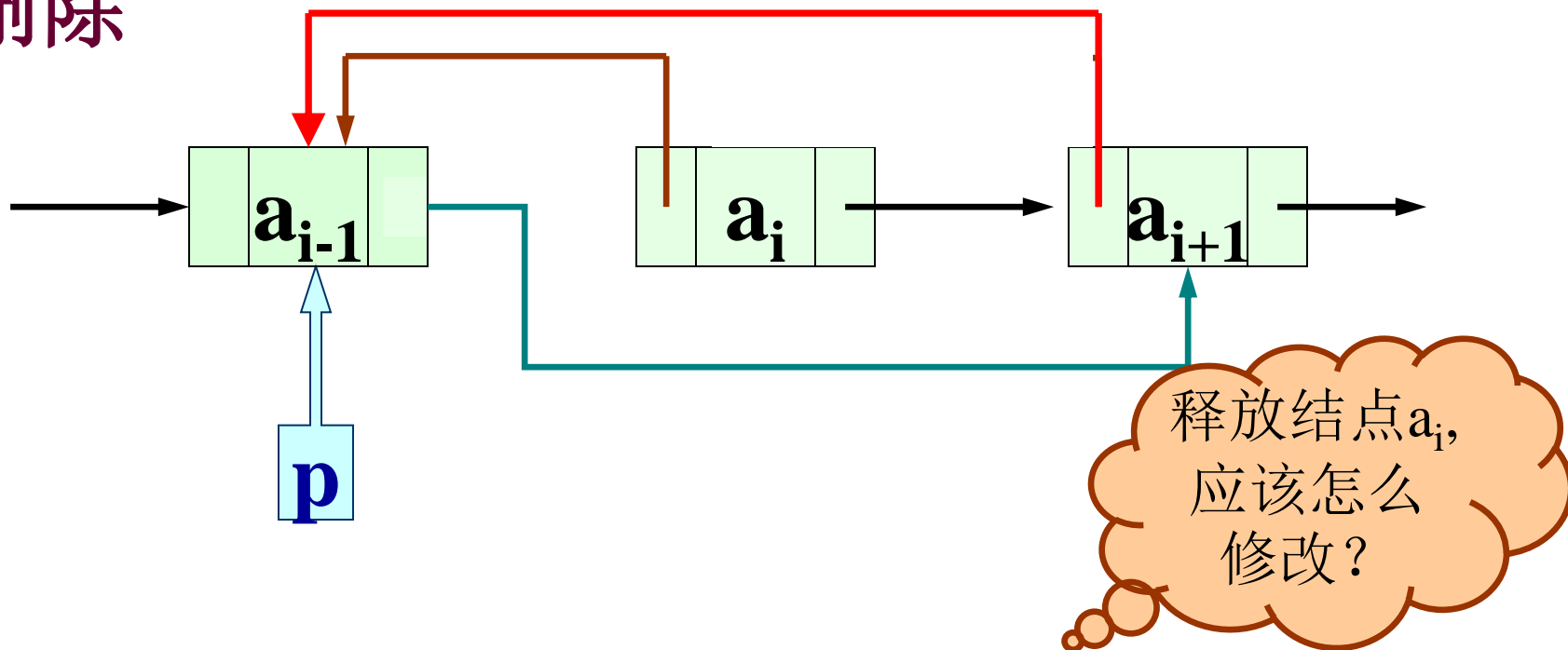
- 双向链表的删除操作需要改变两个方向的指针



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



# 删除



$p \rightarrow \text{next} = p \rightarrow \text{next} \rightarrow \text{next};$

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





## 2.4 一元多项式的表示

# 一元多项式

$$p_n(x) = p_0 + p_1x + p_2x^2 + \dots + p_nx^n$$

在计算机中，可以用一个线性表来表示：

$$P = (p_0, p_1, \dots, p_n)$$

但是对于形如

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

的多项式，上述表示方法是否合适？

一般情况下的**一元稀疏多项式**可写成

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

其中： $p_i$  是指数为 $e_i$  的项的非零系数，

$$0 \leq e_1 < e_2 < \dots < e_m = n$$

可以下列线性表表示：

$$\left( (p_1, e_1), (p_2, e_2), \dots, (p_m, e_m) \right)$$

例如:

$$P_{999}(x) = 7x^3 - 2x^{12} - 8x^{999}$$

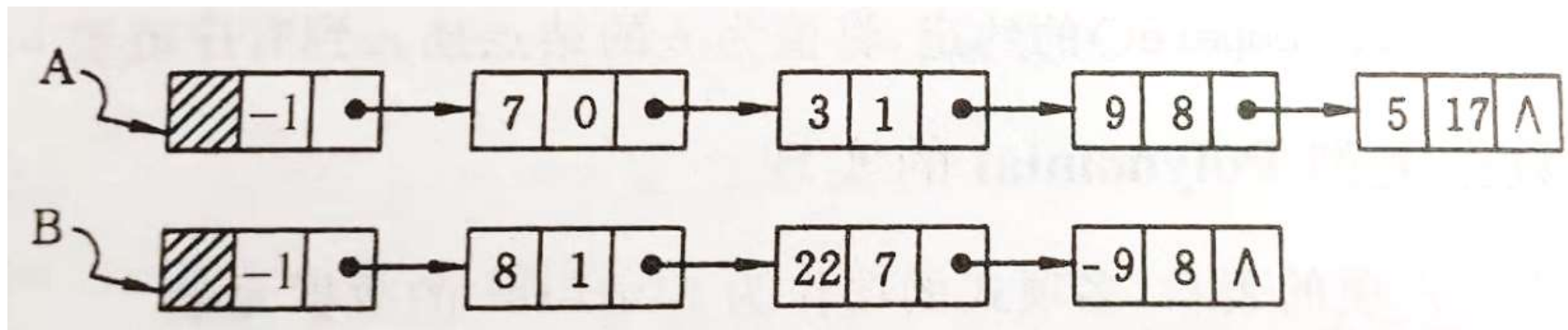
可用线性表

$((7, 3), (-2, 12), (-8, 999))$

表示

$$A(x) = 7 + 3x + 9x^8 + 5x^{17}$$

$$B(x) = 8x + 22x^7 - 9x^8$$



# 一元多项式的实现：

**P42的LinkedList链表类中方法修改：**

**void CreatePolyn(polynomial &P,int m);**

# 一元多项式创建和输出的实现：

**P43的LinkedList链表类中方法修改：**

```
LinkedList &AddPoly(LinkedList &PolyA,LinkedList  
&PolyB);
```



## 第五节 顺序表与链表的比较

### 一、基于空间的比较

- 存储分配的方式
  - ◆ 顺序表的存储空间是静态分配的
  - ◆ 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

## 第五节 顺序表与链表的比较

### 二、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
  - ◆ 链表必须顺序存取
- 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

## 第五节 顺序表与链表的比较

### 三、基于应用的比较

- 如果线性表主要是存储大量的数据，并主要用于查找时，采用顺序表较好，如数据库
- 如果线性表存储的数据元素经常需要做插入与删除操作，则采用链表较好，如操作系统中进程控制块(PCB)的管理，内存空间的管理等

# 作业

1. 数据结构按逻辑结构可分为两大类，它们是：\_\_\_\_结构\_\_\_\_结构。

2. 下面程序段的时间复杂度为\_\_\_\_\_：

```
for (i=0; i<n; i++)  
    for (j=0; j<m; j++)  
        A[i][j]
```

# 作业

3. 若经常需要对线性表进行插入和删除运算，最好采用\_\_\_\_\_存储结构，若经常需要对线性表进行查找运算，最好采用\_\_\_\_\_存储结构。

4. 对于一个单链表，在表头插入结点的时间复杂度为\_\_\_\_\_，在表尾插入结点的时间复杂度为\_\_\_\_\_。

5. 双向链表指针**p**结点之后插入指针为**s**的结点，执行语句：\_\_\_\_\_。

# 作业

**6. 根据二元组关系，指出它们属于何种数据结构。**

**a)  $A = (D, R)$  ,  $D = \{a, b, c, d, e, \}$ ,**

**$R = \{ \langle a, b \rangle, \langle b, c \rangle, \langle c, d \rangle, \langle d, e \rangle, \langle e, f \rangle \}$**

**b)  $B = (D, R)$  ,  $D = \{1, 2, 3, 4, 5, 6\}$ ,**

**$R = \{ (1,2) , (2,3) , (2,4) , (3,4) , (3,5) , (3,6) , (4,5) , (4,6) \}$**

**c)  $C = (D, R)$  ,  $D = \{a, b, c, d, e, f, g, h\}$ ,**

**$R = \{ \langle d, b \rangle, \langle d, g \rangle, \langle d, a \rangle, \langle b, c \rangle, \langle g, e \rangle, \langle g, h \rangle, \langle e, f \rangle \}$**

# 作业

7. 线性表是具有 $n$ 个\_\_\_\_\_的有限序列 ( $n>0$ )。

**A** 表元素

**B** 字符

**C** 数据元素

**D** 数据项

**E** 信息项

# 作业

8. 某线性表中最常用的操作是在最后一个元素之后插入一个元素和删除第一个元素，则采用\_\_\_\_\_存储方式最节省运算时间。

- A.** 单链表
- B.** 单循环链表
- C.** 带尾指针的单循环链表
- D.** 带头结点的双循环链表



# 作业

9. 静态链表中指针表示的是\_\_\_\_\_。

- A.** 内存地址
- B.** 数组下标
- C.** 下一元素地址
- D.** 左、右孩子地址

# 作业

10. 完成在双循环链表结点**p**之后插入**s**的操作是\_\_。
11. 假设一个头指针为**head**的带头结点的单循环链表，判断该表为空表的语句是\_\_\_\_\_。
12. (判对错) 循环链表不是线性表。
13. (判对错) 顺序存储方式的优点是存储密度大，且插入、删除运算效率高。
14. (判对错) 链表中的头结点仅起到标识的作用。

# 作业

15. 在一个以**h**为头的单循环链中，**p** 指针指向链尾的条件是\_\_\_\_\_。

A.  $p \rightarrow next = h$

B.  $p \rightarrow next = NULL$

C.  $p \rightarrow next \rightarrow next = h$

D.  $p \rightarrow data = -1$

# 作业

16. 假设静态链表**A**，其结点结构为（**data**，**next**），**A[0]**是备用链表头结点，**A[1]**是线性表头结点。在线性表头插入元素**e**的语句是\_\_\_\_\_。