

计算机科学基础II

第七章 动态内存分配

曹鹏

Email: caopeng@seu.edu.cn

Tel: 13851945861

本章提纲

- ◆ 动态内存分配的基本概念 (7.1.1, 7.1.2节)
- ◆ 浅复制与深复制 (7.1.3节)
- ◆ 线性表之链表 (7.2节)
- ◆ 线性表之栈/队 (7.3.1, 7.3.2节)

数组实现顺序表的缺陷

◆ 线性表之顺序表：大小不定→大开小用

◆ 合法的数组定义

```
int a[10]; //①整型常量定义数组长度
```

```
const int size=20;
```

```
int b[size]; //②整型常量变量定义数组长度
```

◆ 希望的数组定义（但不合法）

```
int size;
```

```
size= 5; int c[size]; //变量定义数组长度
```

```
size=10; int c[size]; //数组定义后可再一次改变长度
```

数组实现顺序表的缺陷

◆ 问题症结：静态内存管理

程序在编译时，根据数组元素类型和个数分配所需内存大小，在程序运行时无法改变

◆ 解决方法：动态内存管理

程序在运行时确定所需内存大小

内存空间分布

§3.3.1 图3.3

内存空间	存储数据	数据初值
栈区 (stack)	函数（含主函数）局部变量 函数参数	随机值
全局数据区	全局变量和静态变量	全0
代码区	程序代码	-

内存空间分布

§3.3.1 图3.3

内存空间	存储数据	数据初值
自由存储区 (堆区 heap)	动态管理的数据	随机值
栈区 (stack)	函数 (含主函数) 局部变量 函数参数	随机值
全局数据区	全局变量和静态变量	全0
代码区	程序代码	-

动态内存管理

◆ 动态内存分配(new)

程序运行中，程序员在堆区申请一块空间，用于存储变量（对象）

◆ 动态内存释放(delete)

程序运行中，程序员将堆区申请的空间释放

动态内存管理vs.静态内存管理

静态内存管理

栈stack

宿舍

分配空间

在学校统一分配宿舍（编译器），在使用过程中不能改变位置，也不能改变大小（栈空间）

释放空间

离校时学校统一收回

动态内存管理

堆heap

宾馆

分配空间

外出入住宾馆（程序员申请堆空间），可以申请多个房间

释放空间

退房

动态内存分配

◆ 操作符new

◆ 动态分配变量（对象）

指针变量=new 变量类型;

指针变量=new 变量类型（初始化式）;

◆ 指针变量的与动态分配(new)的变量类型相同

◆ 指针变量的值为动态分配的变量的地址

◆ 分配对象空间时，调用相应的构造函数

◆ 不初始化：无参

◆ 初始化：根据初始化参数个数和类型

```
int *pi=new int;
```

```
int *pi=new int(10);
```

```
Complex *pComplex=new Complex; //调用构造函数
```

```
Complex *pComplex=new Complex(); //同上式
```

```
Complex *pComplex=new Complex(1.0, 2.0);
```

动态内存分配

◆ 操作符new

◆ 动态分配变量数组（对象数组）

指针变量=new 变量类型[变量表达式];

◆ 指针变量的值为动态分配的数组的首地址

◆ 分配对象空间时，调用无参数的构造函数，调用次数与数组长度相等

```
int *pi=new int[size];
```

```
Complex *pComplex=new Complex[cnt];
```

```
//不带初始化参数，必须有无参数构造函数
```

动态内存分配

◆ 操作符new

	功能	静态内存分配	动态内存分配
变量	分配空间	<code>int i;</code>	<code>int *p=new int;</code>
	分配空间并初始化	<code>int i=10;</code>	<code>int *p=new int(10);</code>
数组	分配定长空间	<code>int i[5];</code>	<code>int *p=new int[5];</code>
	分配变长空间	无法实现	<code>int size;</code> <code>size=5;</code> <code>int *p=new int[size];</code>

动态内存分配

- ◆ 动态分配数组空间，数组的大小是个变量，彻底解决变长数组的问题
- ◆ **new**的返回值是变量类型的指针，指向变量/对象（数组）的（首）地址
- ◆ 分配的数组空间是无名对象，只能通过指针间接访问
- ◆ **new**分配空间可能不成功（返回**NULL**），因此返回的指针访问该空间前需要进行**分配确认**

动态内存释放

◆ 操作符 `delete`

◆ 动态释放变量（对象）

◆ 如果是类对象指针，则调用析构函数

`delete` 指针变量;

//释放(`delete`)指针变量指向的空间，不是`delete`指针变量

◆ 动态释放数组

◆ 如果是类对象指针，则调用析构函数，调用次数与数组长度相等

`delete []` 指针变量;

//如果是类对象数组指针，则调用多次析构函数

//[]中不用写数组元素个数，系统会自动判断

`Complex *p=new Complex[10];`

A) `delete p;`  //只删除第一个元素`p[0]`

B) `delete []p;`  //删除整个数组`p[0]~p[9]`

【例7.1】动态数组的建立与撤销

```
#include <iostream>
#include <cstring>
using namespace std;
int main(){
    int n;
    char *pc;
    cout<<"请输入动态数组的元素个数"<<endl;
    cin>>n;
    pc=new char[n]; //申请包含25个字符元素的内存空间
    strcpy(pc, "自由存储区内存的动态分配"); //12个字+' \0'
    cout<<pc<<endl;
    delete []pc; // 撤销并释放pc所指向的n个字符的内存空间
    return 0;
}
```

【例7.3】自由存储区对象分配和释放

```
class CGoods{
    string Name;
    int Amount;
    float Price;
    float Total_value;
public:
    CGoods(){cout<<"调用默认构造函数"<<endl;};
    CGoods(string name,int amount ,float price){
        cout<<"调用三参数构造函数"<<endl;
        Name=name; Amount=amount;
        Price=price; Total_value=price*amount;
    }
    ~CGoods(){ cout<<"调用析构函数"<<endl;};
};
```


【例7.3】自由存储区对象分配和释放

```
int main(){  
    int n;  
    CGoods *pc,*pc1,*pc2;  
    pc=new CGoods("夏利2000",10,118000); //调用三参数构造  
    构造函数  
    pc1=new CGoods(); //调用默认构造函数  
    cout<<"输入商品类数组元素数"<<endl;  
    cin>>n;  
    pc2=new CGoods[n]; //调用默认构造函数，共调n次  
    delete pc;  
    delete pc1;  
    delete []pc2;  
    return 0;  
}
```


动态内存释放

- ◆ 动态释放后，指针变量还能用来访问空间吗？
 - ◆ 可以，称为**空悬指针**，但会产生不可预知的后果
 - ◆ 如何避免？动态释放后立即将指针变量置为空值
指针变量名=NULL;

动态内存管理

- ◆ 操作符`new`和`delete`必须配对使用
 - ◆ 少了：没有`delete`
 - ◆ 导致动态分配的空间没有释放，也无法被再次分配，称为**内存泄漏**
 - ◆ 多了：重复`delete`
 - ◆ 可能这部分空间已经被再次分配
 - ◆ 再次`delete`时导致新分配的空间被误释放，或者其他不可预知的后果，称为**重复释放**

动态内存管理

◆ 动态分配对象的生命期不限于建立时的作用域

```
Complex* A(int len){...  
    Complex* p=new Complex[len];  
    return p;  
...}
```

```
void B(Complex *p){...  
    p[0]=val;  
...}
```

```
void C(Complex *p){...  
    delete p;  
...}
```

```
int main(){  
    int length; Complex *pc;  
    ...  
    pc=A(length); //动态分配空间  
    ...  
    B(pc); //访问（读写）该空间  
    ...  
    if (...) C(pc); //动态释放空间  
    return 0;  
}
```

动态内存管理

- ◆ 分配确认
- ◆ 空悬指针
- ◆ 内存泄漏
- ◆ 重复释放
- ◆ 动态分配对象的生命期

动态内存管理vs.静态内存管理

静态内存管理

栈	宿舍
分配空间	学校统一分配宿舍（编译器），有固定宿舍地址（变量名），在读过程中不能改变位置，也不能改变大小（栈空间）
释放空间	离校时学校统一收回

动态内存管理

堆	宾馆
分配空间	外出入住宾馆（程序员申请堆空间），临时分配了房间号（new返回的指针值，也就是动态分配空间的地址）
释放空间	根据房间号退宿（delete 指针值）
分配确认	如果没有空余房间，申请可能不成功，所以务必确认房间号有效（非NULL）
空悬指针	虽然退宿，但是仍然知道房间号（指针值）。而房间可能已经重新分配。
内存泄漏	忘了退宿，导致房间一直被占用，可用的房间越来越少 不但忘了退宿，还忘了房间号（指针值被修改），导致房间永远被占用
重复释放	多次退宿，房间可能已经重新分配，退的是别人的房间
生命期	A（函数）申请房间(new)，把房间号（指针值）告诉B，由B使用，A或B再把房间号（指针值）告诉C，由C退房(delete)

动态内存管理对类对象复制的影响

◆ 复制

◆ 复制构造函数

- ◆ 初始化的时候复制
- ◆ 参数值传递
- ◆ 返回值值传递

◆ 赋值操作符重载函数

- ◆ 使用的时候复制

◆ 如果类的数据成员所占用空间是动态分配的，会对复制产生怎样的影响？

类对象的复制

静态分配数据成员



浅复制
(默认复制构造函数)

动态分配数据成员



深复制

浅复制

浅复制引发的问题

- ◆ 不同类对象（A和B）使用同一块动态分配的堆空间，浅复制引发问题
 - ◆ 使用阶段（阶段1）
 - ◆ 析构阶段（阶段2,3）

阶段1：
对象AB均未析构

阶段2：
对象A析构

阶段3：
对象B析构

对象A(B)修改动态分配的
空间时，对象B(A)的也被
改了



析构函数释放堆空间：
对象B存在**空悬指针**问
题



析构函数不释放堆空间：
对象B不变



析构函数释放堆空间：
对象B存在**重复释放**问题



析构函数不释放堆空间：
对象B存在**内存泄露**问题



深复制

深复制实现类对象复制，分两步：

- ◆ 第一步，内存分配：对新的类对象，动态分配新的堆空间
- ◆ 第二步，内存初始化：复制原有对象的堆空间的值，用以初始化新分配的堆空间

受深复制影响的类成员函数

- ◆ 复制构造函数
- ◆ 赋值操作符重载函数
- ◆ 析构函数

【例7.4】深复制的实现

```
#include <iostream>
using namespace std;

class student
{
    char *pName;
public:
    student();
    student(char *pname);
    student(student& str); //复制构造函数
    student & operator=(student & str); //赋值操作符重载函数
    ~student(); //析构函数
};
```

【例7.4】深复制的实现

```
student::student(){
    cout<<"Constructor";
    pName=NULL;
    cout<<"默认"<<endl;
}
student::student(char *pname){
    cout<<"Constructor";
    if(pName=new char[strlen(pname)+1])
        //加1不可少，否则串结束符冲了其他信息，析构会出错！
        strcpy(pName,pname);
    cout<<pName<<endl;
}
```

【例7.4】自定义复制构造函数实现深复制

//浅复制，或者采用默认复制构造函数

```
student::student(student& str)
{
    pName=str.pName;
};
```



错误

//深复制

```
student::student(student& str)
{
    cout<<"Copy Constructor";
    if (str.pName){ //待复制堆空间非空
        pName=new char [strlen(str.pName)+1]; //先分配
        if (pName) strcpy(pName, str.pName); //再复制
    }
    else pName=NULL;
};
```

【例7.4】 自定义赋值操作符重载函数实现深复制

//浅复制

```
student & student::operator=(student & str)
{
    pName=str.pName;
    return *this;
};
```

错误

//深复制

```
student & student::operator=(student & str)
{
    cout<<"Copy Assign operator";
    if (pName!=str.pName){ //排除赋值给自己的情况, 如s1=s1;
        delete []pName; pName=NULL; //先释放堆空间并将指针置空
    }
    if (str.pName){ //待复制堆空间非空
        pName=new char [strlen(str.pName)+1]; //先分配
        if (pName) strcpy(pName, str.pName); //后复制
    }
    else pName=NULL;
    return *this;
};
```

【例7.4】自定义析构函数实现深复制

//浅复制，或者采用默认析构函数

```
student::~~student()  
{  
};
```



错误

//深复制




```
student::student()  
{  
    cout << "Destructor";  
    delete []pName;  
};
```

【例7.4】深复制的实现

```
int main(void){  
    student s1("范英明"),s2("沈俊"),s3;  
    student s4=s1;  
    s3=s2;  
    return 0;  
}
```

Constructor	范英明	//建立S1
Constructor	沈俊	//建立S2
Constructor	默认	//建立S3
Copy Constructor	范英明	//建立S4
Copy Assign Operator	沈俊	//用S2赋值S3
Destructor	范英明	//析构S4
Destructor	沈俊	//析构S3
Destructor	沈俊	//析构S2
Destructor	范英明	//析构S1

浅复制与深复制的区别

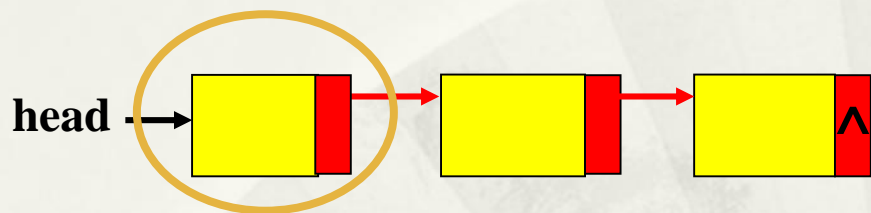
		浅复制	深复制
成员函数	复制构造函数	系统默认	自定义
	赋值操作符重载函数	系统默认	自定义
	析构函数	系统默认	自定义
实现方法	复制对象	按 成员 复制 复制堆空间指针 	按 内容 复制 先分配堆空间， 再复制堆空间的内容
	堆空间使用	多个类对象 共享 	类对象 独享
	堆空间释放	存在 空悬指针 或 重复释放 问题 	类对象释放各自堆空间

线性表之链表

- ◆ 线性表：一种含有 $n(\geq 0)$ 个同类结点的有限序列
 - ◆ 均匀性：各个结点具有相同的数据类型
 - ◆ 有序性：各个结点之间的相对位置是线性的
- ◆ 按访问方式不同，可分为
 - ◆ 直接随机访问：顺序表（数组）
 - ◆ 间接顺序访问：链表（参见7.2节）
 - ◆ 双端单向访问：队列（参见7.3节）
 - ◆ 单端双向访问：栈（参见7.3节）

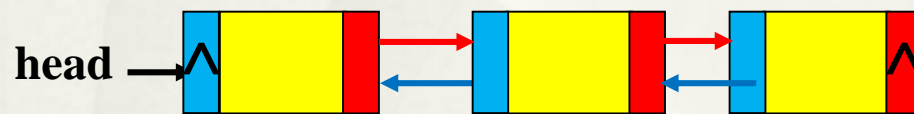
链表的形式

单向链表



结点

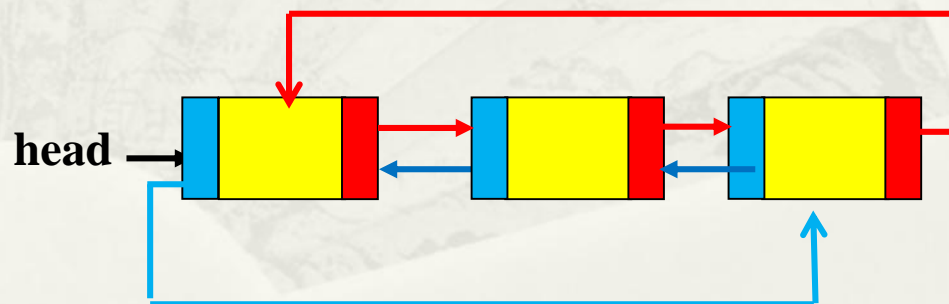
双向链表



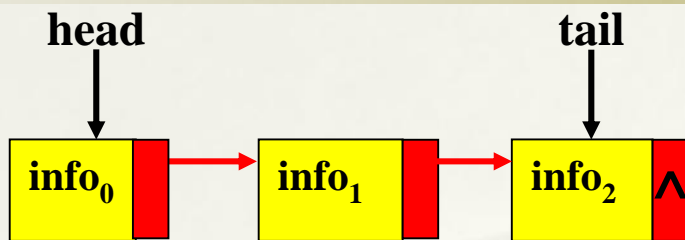
单向循环链表



双向循环链表



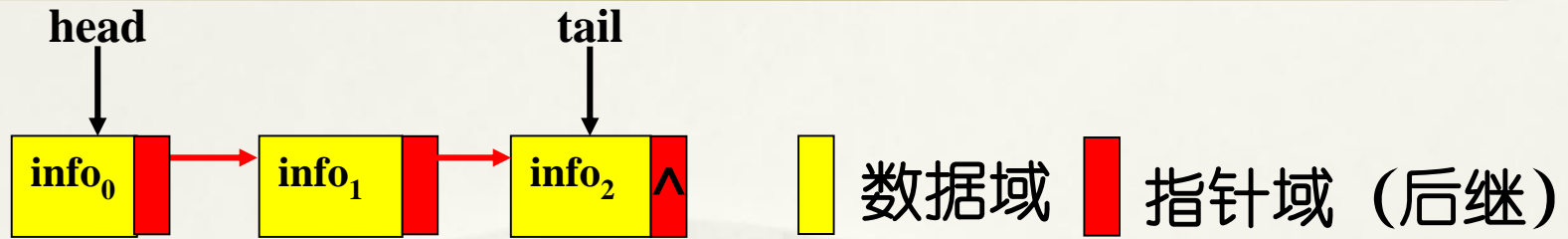
单（向）链表



◆单链表的特征

- ◆间接顺序访问：只能由表头(head)开始，逐个访问各个结点
- ◆只能访问每个结点的后继结点，不能访问前驱结点（不代表没有前驱）

单链表的构成（结点）



◆ 结点（多个）：结构变量/类对象

◆ 数据域（黄色）：存储结点数据

◆ 指针域（红色）：指向结点结构变量/类对象的指针，存储该结点的后继节点的地址

```
typedef int Datatype; //将int类型重命名为Datatype类型
```

```
struct node{
```

```
    Datatype info; //数据域
```

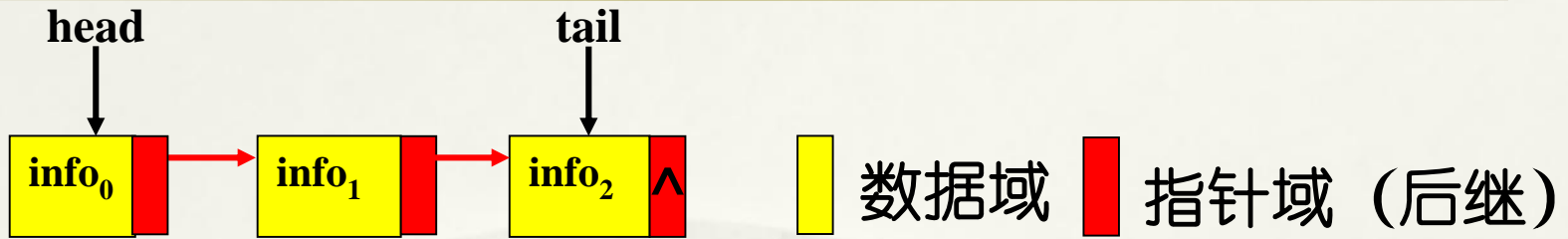
```
    node *link;    //指针域
```

```
};
```

```
node * n1, n2;
```

指针域类型为结点类型的指针，node *
缺少*则为结点类型，导致无穷递归定义的错误

单链表的构成（表头/表尾）



◆表头指针head

◆定义

```
node *head;
```

◆指向链表第一个结点

◆通过head再顺序遍历链表的其他结点

◆如果head的值(表头地址)丢失，则整个链表丢失，内存泄漏

◆表尾指针tail

◆定义

```
node *tail;
```

◆指向链表最后一个结点，指针域的值**为NULL**

◆可由表头指针遍历寻得（因此也可以没有）

单链表操作

- ◆查找结点
- ◆插入结点
- ◆链表生成（前向生成/后向生成）
- ◆删除结点

```
typedef int Datatype; //将int类型重命名为Datatype类型
struct node{
    Datatype info; //数据域
    node *link;    //指针域
};
```

单链表操作

- ◆查找结点
- ◆插入结点
- ◆链表生成（前向生成/后向生成）
- ◆删除结点

查找结点

```
node *traversal(node *head, Datatype data){  
    node *p=head;  
    //从表头起逐个访问每个结点的后继  
    while(p!=NULL&& p->info!=data) p=p->link;  
    return p;  
}
```

- ◆表头head不能动，用p指针从表头起访问每个结点
- ◆查找终止条件
 - ◆找到了：p指向的结点就是要找的结点
p->info==data
 - ◆找完了：p指向表尾的后继，也就是NULL
p==NULL
- ◆类似方式实现：打印链表，计算链表长度

单链表操作

- ◆ 查找结点
- ◆ 插入结点
- ◆ 链表生成（前向生成/后向生成）
- ◆ 删除结点

插入结点

- ◆ 在链表中插入一个结点，对前后结点有什么影响？
 - ◆ 类比：排队时插队，影响的是前一个人还是后一个人？
 - ◆ 后一个人，因为后一个人单向指向前一个人
- ◆ 本质：排队和单链表类似，具有方向性
 - ◆ 排队：向**前**排，**插入**影响**后**一个人（的前驱）
 - ◆ 单链表：向**后**排，**插入**影响**前**一个结点（的后继）

插入结点

◆插入新结点

- ◆ 找到前一个结点q（插在q结点之后）

- ① 修改新结点的后继，指向后一个结点p

- ② 修改前一个结点p的后继，指向新结点

- ③ 必要的话，更新表头/表尾

◆在指定位置插入新结点newnode

- ◆中间 ($q \rightarrow \text{newnode} \rightarrow p$)

- ◆表尾tail

- ◆表头head

插入结点

◆在中间 ($q \rightarrow p$, q 结点之后, p 结点之前) 插入新结点newnode

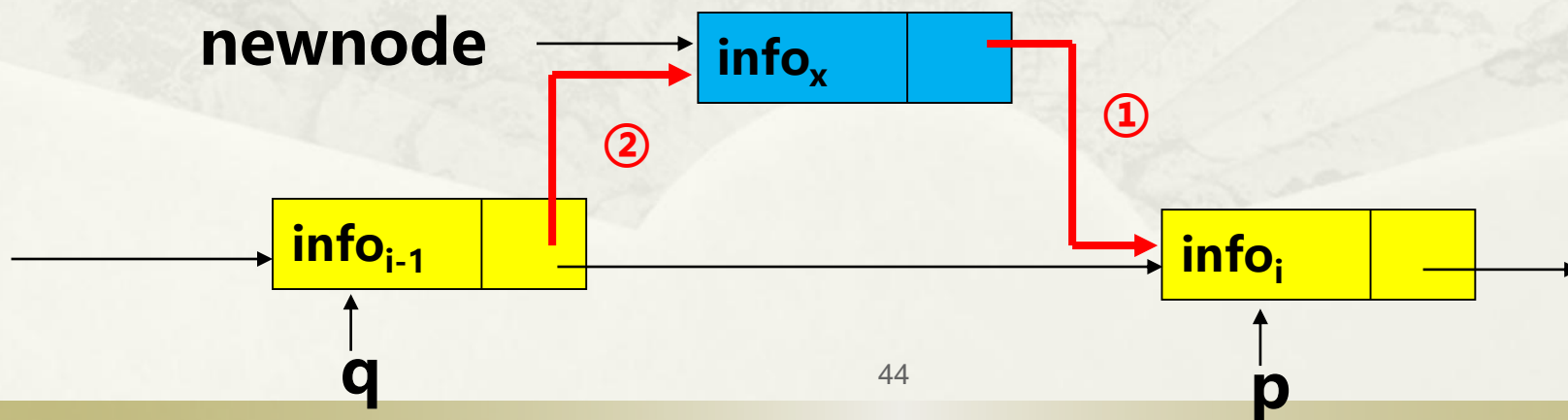
//①修改newnode的后继, 指向 p 结点

newnode \rightarrow link = p ; //或newnode \rightarrow link = $q \rightarrow$ link;

//②修改 q 结点的后继, 指向newnode

$q \rightarrow$ link = newnode;

只有 q 指针, 没有 p 指针可不可以? 可以, p 就是 $q \rightarrow$ link
反之呢? 不可以



插入结点

◆在表尾插入新结点newnode

//①修改newnode的后继, 指向NULL

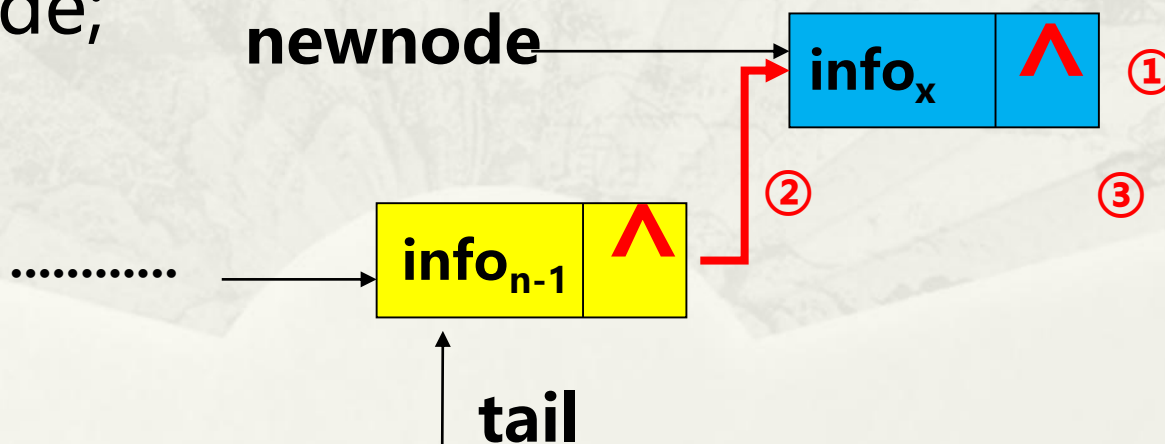
newnode→link=NULL;

//②修改tail的后继, 指向newnode

tail→link=newnode;

//③修改tail, 指向newnode

tail=newnode;



插入结点

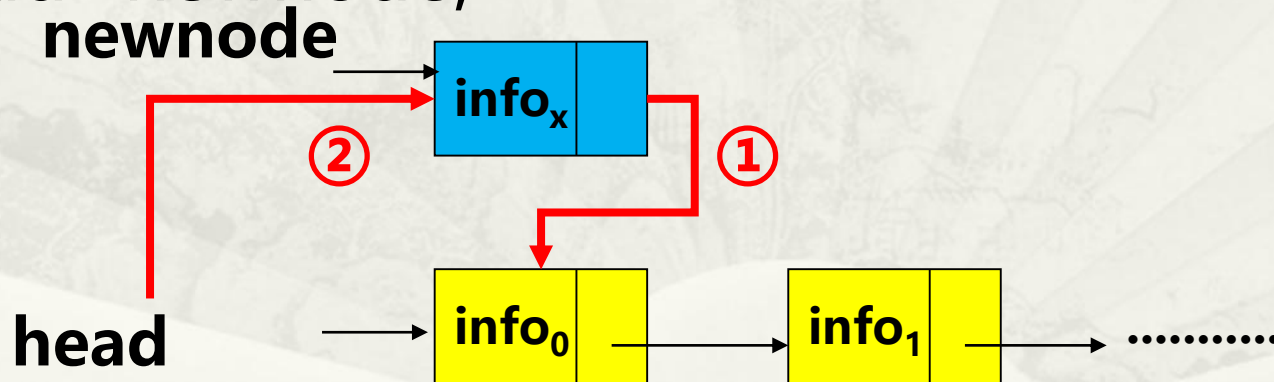
◆在表头插入新结点newnode

//①修改newnode的后继, 指向head

newnode→link=head;

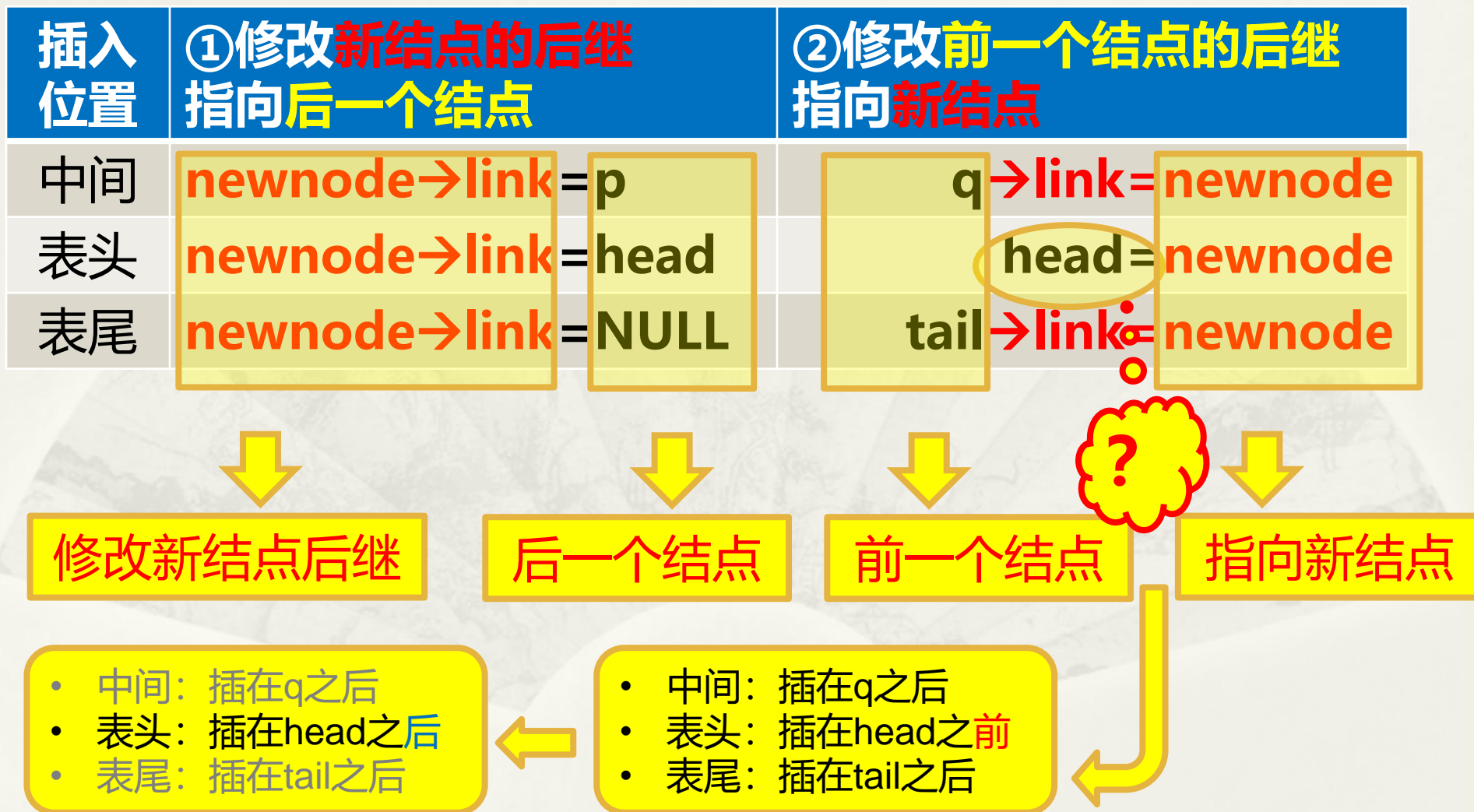
//②修改head, 指向newnode

head=newnode;



插入结点

◆三种位置的插入算法能否统一？



插入结点

◆统一算法：实现某结点后插入新结点

◆在链表前额外加上一个空表头

◆空表头的数据域是空的

◆插入结点只能在空表头之后



◆链表为空时，仍有空表头

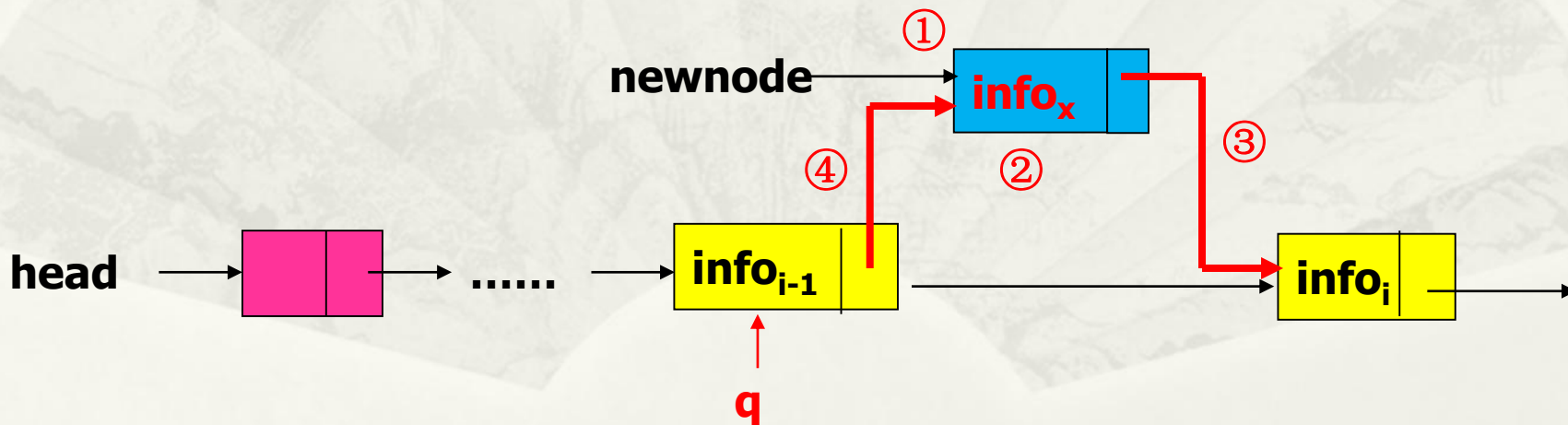


- 收益：统一算法，便于维护
- 代价：浪费1个结点（空表头）的存储空间

插入结点

//在q结点后插入新结点

```
void insert(node *q, Datatype x){  
    node *newnode=new node; //①新建结点newnode  
    newnode->info=x;          //②newnode数据域赋值  
    newnode->link=q->link; //③newnode后继指向q结点的  
    后继  
    q->link=newnode; //④q结点后继指向newnode  
}
```



单链表操作

- ◆查找结点
- ◆插入结点
- ◆链表生成 (前向生成/后向生成)
- ◆删除结点

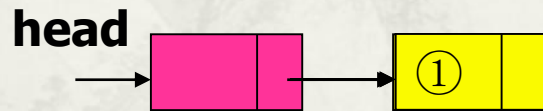
生成链表

◆在插入结点的算法基础上，实现生成链表的算法

◆向前生成链表

由表尾**向表头**逐步生成链表

将每个新结点**插入表头**结点之后



◆向后生成链表

由表头**向表尾**逐步生成链表

将每个新结点插入**表尾**结点之后



向前生成链表

```
node *createup(){
    node *head,*p;
    Datatype data;
    //step1:建立空表头
    head=new node;
    head->link=NULL;
```

```
//step2:将每个新结点作为表头结点的后继
```

```
while(cin > > data){
```

```
p=new node; //建立新结点
```

p->info=data; //新结点写入数据

```
p->link= head->link; //表头结点的后继作为新结点的后继
```

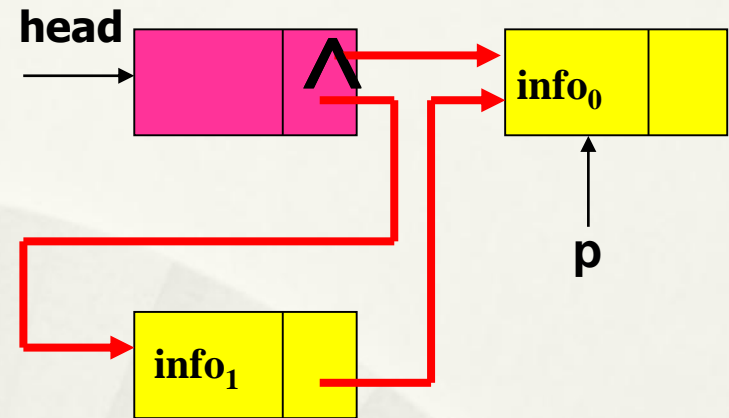
```
head->link=p; //新结点作为表头结点的后继
```

}

```
//step3:返回表头结点
```

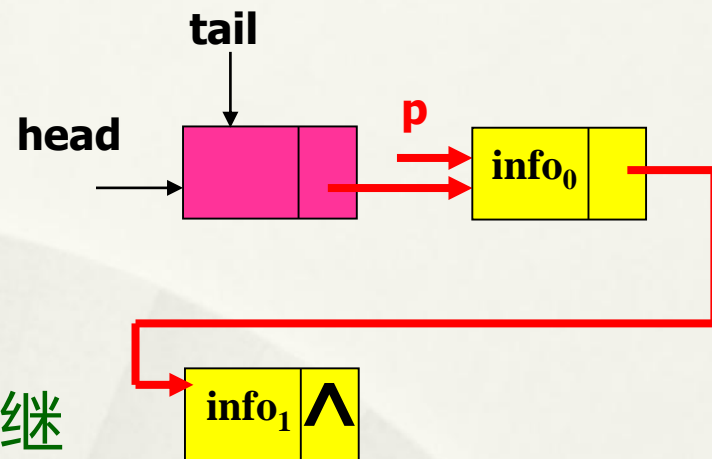
```
return head;
```

}



向后生成链表

```
node *createdown(){
    node*head,*tail,*p;
    Datatype data;
    //step1:建立表尾(头)结点
    tail=head=new node;
    //step2:将每个新结点作为表尾结点的后继
    while(cin>>data){
        p=new node; //建立新结点
        p->info=data; //新结点写入数据
        tail->link=p; //新结点作为表尾结点的后继
        tail=p; //新结点作为表尾
    }
    //step3:表尾结点的后继置为NULL
    tail->link=NULL;
    //step4:返回表头结点
    return head;}
```



单链表操作

- ◆查找结点
- ◆插入结点
- ◆链表生成（前向生成/后向生成）
- ◆删除结点

删除结点

◆插入新结点

- ◆找到前一个结点
- ◆修改新结点的后继，指向后一个结点
- ◆修改前一个结点的后继，指向新结点

◆删除结点

- ◆找到前一个结点
- ◆修改前一个结点的后继，指向该结点的后继
- ◆释放该结点

删除结点

//删除q结点之后的结点

```
void del (node *q){
```

```
    node *delnode=q->link; //①标记被删结点为delnode
```

```
    q->link=delnode->link; //②q结点后继指向delnode的后继
```

```
    delete delnode; //③释放delnode
```

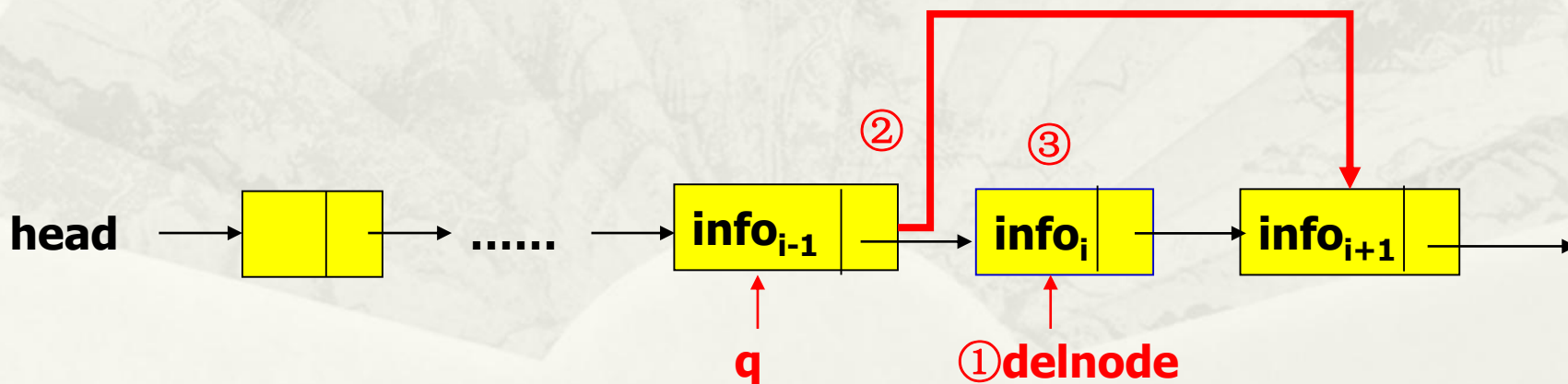
```
}
```

//②还可以写成 $q->link=q->link->link$;

是否可以删除q结点? 不可以, 除非有q的前驱

是否可以不定义局部变量delnode?

不可以, 删除前 $q->link$
已不再指向delnode



单链表操作小结

- ◆ 核心是指针的读写 $q \rightarrow \text{link} = \text{delnode} \rightarrow \text{link}$;
 - ◆ 赋值号左边($q \rightarrow \text{link}$): 被修改的指针, q结点 (q指针指向的结点) 的link指针
 - ◆ 赋值号右边 ($\text{delnode} \rightarrow \text{link}$) : $q \rightarrow \text{link}$ 指针被修改为指向 $\text{delnode} \rightarrow \text{link}$; 或者说, $q \rightarrow \text{link}$ 指针的值 (是一个地址) 被修改为 $\text{delnode} \rightarrow \text{link}$ 指针的值
- ◆ 每个结点通过指针域(link)连接下一个结点, 构成链表。链表操作的关键是**结点指针域的修改**
 - ◆ 插入/删除结点, 修改**前驱**结点的指针域(link)
- ◆ 链表结点的个数运行时确定, 每个**结点也是动态分配**, 结点从链表删除时, 注意结点内存的动态释放

类模板实现单链表

◆ 结点类模板Node

- ◆ 处理的对象是结点
- ◆ 封装结点的属性（数据域；指针域）和基本操作
- ◆ 将链表类List作为友元类，这样可以在List类成员函数中直接访问Node的私有成员（数据域、指针域）

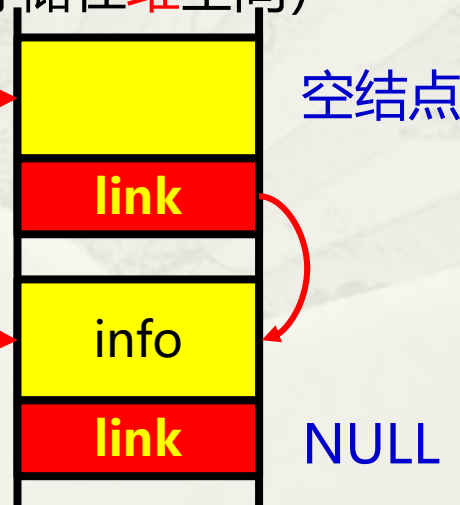
◆ 链表类模板List

- ◆ 处理的对象是链表
- ◆ 封装链表的属性（表头指针；表尾指针）和基本操作

List类对象
(一般定义对象,
分配在栈空间)



Node类对象
(一般动态分配,
存储在堆空间)



空结点

NULL

类模板实现单链表

知识点

- ◆友元类
- ◆类模板
- ◆动态内存管理
- ◆链表

类模板实现单链表

◆ 结点类模板 Node

- ◆ 在结点自身后插入结点 InsertAfter
- ◆ 删除结点自身后的结点 RemoveAfter

◆ 链表类模板 List

- ◆ 清空链表（只留表头） MakeEmpty
- ◆ 查找链表中的结点 Find
- ◆ 计算链表长度 Length
- ◆ 显示链表 PrintList
- ◆ 向前生成链表 InsertFront
- ◆ 向后生成链表 InsertRear
- ◆ 升序生成链表 InsertOrder
- ◆ 创建结点（孤立结点） CreateNode
- ◆ 删除链表中的结点 DeleteNode

结点类模板

//结点类

template<typename T> class List; //引用声明

template<typename T> class Node{

T info; //数据域

Node<T> *link; //指针域

public:

Node(); //构造函数（空表头结点）

Node(const T & data); //构造函数（非表头结点）

void InsertAfter(Node<T>* p); //在结点自身后插入结点

Node<T>* RemoveAfter(); //在结点自身后删除结点

friend class List<T>;

//以List类为友元类，使得List类可访问Node类的私有成员
 (info和link)

};

结点类模板

//构造表头结点

```
template <typename T> Node<T>::Node(){  
    link=NULL;  
}
```

//构造结点 (非表头)

```
template <typename T> Node<T>::Node(const T & data){  
    info=data;  
    link=NULL;  
}
```

结点类模板

//在结点自身后插入结点

```
template<typename T>
```

```
void Node<T>::InsertAfter(Node<T>* p){
```

```
    p->link=link; //本结点的后继作为新结点的后继
```

```
    link=p; //新结点作为本结点的后继
```

```
}
```

//在结点自身后删除结点

```
template<typename T>
```

```
Node<T>* Node<T>::RemoveAfter(){
```

```
    Node<T>* tempP=link; //被删结点指向本结点的后继
```

```
    if(link) link=link->link; //如非链尾，后继的后继作为新
```

```
    的后继
```

```
    return tempP; //返回被删结点
```

```
}
```


单链表类模板

//链表类

```
template<typename T> class List{  
    Node<T> *head,*tail; //链表头指针和尾指针  
public:  
    List(); //构造函数, 生成空链表  
    ~List(); //析构函数  
    void MakeEmpty(); //清空链表, 只余表头结点  
    Node<T>* Find(T data); //查找结点  
    int Length(); //计算链表长度  
    void PrintList(); //显示链表  
    void InsertFront(Node<T>* p); //向前生成链表  
    void InsertRear(Node<T>* p); //向后生成链表  
    void InsertOrder(Node<T>* p); //升序生成链表  
    Node<T>* CreatNode(T data); //创建结点(孤立结点)  
    Node<T>* DeleteNode(Node<T>* p); }; //删除结点
```


链表类模板

//构造函数, 生成空链表 (只有表头结点)

```
template<typename T> List<T>::List(){  
    head=tail=new Node<T>();  
}
```

```
template <typename T> Node<T>::Node(){  
    link=NULL;}
```

//析构函数

```
template<typename T> List<T>::~~List(){  
    MakeEmpty(); //删除除了表头结点外的所有节点  
    delete head;  
}
```

链表类模板

Node<T>类私有成员

//清空链表（只剩表头结点）

```
template<typename T> void List<T>::MakeEmpty(){
    Node<T> *tempP; //标记被删结点为tempP
    while(head->link!=NULL){
        tempP=head->link; //tempP指向表头的后继结点
        head->link=tempP->link; //表头的后继指向tempP的后继
        //head->link=head->link->link;
        delete tempP;
        tempP=head->RemoveAfter();
    }
    tail=head; //只剩空表头，表头与表尾相同，即为空链
}
```

属于向后删除，能不能向前删除？

不能，因为只能访问后继结点

链表类模板

//查找结点

```
template<typename T>
```

```
Node<T>* List<T>::Find(T data){
```

```
    Node<T> *tempP=head->link;
```

```
    while( tempP!=NULL&&tempP->info!=data)
```

```
        tempP=tempP->link;
```

```
    return tempP;
```

```
}
```

//计算链表长度

```
template<typename T>
```

```
int List<T>::Length(){
```

```
    Node<T>* tempP=head->link;
```

```
    int count=0;
```

```
    while(tempP!=NULL){
```

```
        count++;
```

```
        tempP=tempP->link; }
```

```
    return count;}
```

1. 从表头起

2. 逐个查看/处理

3. 直至表尾

//打印链表

```
template<typename T>
```

```
void List<T>::PrintList(){
```

```
    Node<T>* tempP=head->link;
```

```
    while(tempP!=NULL){
```

```
        cout<<tempP->info<<"\t";
```

```
        tempP=tempP->link; }
```

```
    cout<<endl;}
```

链表类模板

//向前生成链表

```
template<typename T>
```

```
void List<T>::InsertFront(Node<T> *p){
```

```
    p->link=head->link;
```

```
    head->link=p;
```

```
    if(tail==head) tail=p;
```

```
}
```

head->InsertAfter(p)

//向后生成链表

```
template<typename T>
```

```
void List<T>::InsertRear(Node<T> *p){
```

```
    p->link=tail->link; //p->link=NULL;
```

```
    tail->link=p;
```

```
    tail=p;
```

```
}
```

tail->InsertAfter(p)

链表类模板

//升序生成链表

template<typename T>

void List<T>::InsertOrder(Node<T> *p){

Node<T> *tempQ=head, *tempP=head->link;

//tempP是tempQ的后继

while(tempP!=NULL){

//找到正确位置(tempQ→p→tempP)或到表尾

if(p->info<tempP->info) break; //一旦发现第一个更大的结点, 即表示找到插入点

tempQ=tempP;

tempP=tempP->link;

}

tempQ->InsertAfter(p); //将p结点插在tempQ之后

if(tempQ==tempQ->link)

tempQ->link=p; //tail=p;

}

1. 找到插入点

2. 插入

链表类模板

//生成结点 (孤立结点)

```
template<typename T>
```

```
Node<T>* List<T>::CreatNode(T data){
```

```
    Node<T>*tempP=new Node<T>(data);
```

```
    return tempP;
```

```
}
```

```
template <typename T>
```

```
Node<T>::Node(const T & data){
```

```
    info=data; link=NULL; }
```

链表类模板

//删除结点p

```
template<typename T>
```

```
Node<T>* List<T>::DeleteNode(Node<T>* p){  
    Node<T>* tempP=head; //tempP作为p的前驱  
    while(tempP->link!=NULL&&tempP->link!=p)  
        tempP=tempP->link;  
    if(tempP->link==tail) tail=tempP;  
    return tempP->RemoveAfter();  
}
```

特例1：如果结点p恰好是表尾(p==tail)?

tempP->link==p

特例2：如果结点p不在链表中?

tempP->link==NULL

```
template<typename T> Node<T>* Node<T>::RemoveAfter(){
```

```
    Node<T>* tempP=link; //被删结点指向本结点的后继
```

```
    if (link) link=link->link; //如非链尾，后继的后继作为新的后继
```

```
    return tempP; //返回被删结点
```

```
}
```


【例7.5】链表类模板

- ◆ 由键盘输入16个整数
- ◆ 以这些整数作为结点数据，生成两个链表，一个向前生成，一个向后生成，输出两个表。
- ◆ 然后给出一个整数在一个链表中查找，找到后删除它，再输出该表。
- ◆ 清空该表，再按升序生成链表并输出。

【例7.5】链表类模板

```
int main(){
    Node<int> * P1;
    List<int> list1,list2;
    int a[16],i,j;
    cout<<"请输入16个整数"<<endl;
    for(i=0;i<16;i++) cin>>a[i];           //随机输入16个整数
    for(i=0;i<16;i++){
        P1=list1.CreatNode(a[i]);
        list1.InsertFront(P1);              //向前生成list1
        P1=list2.CreatNode(a[i]);
        list2.InsertRear(P1);                //向后生成list2
    }
    list1.PrintList();
    cout<<"list1长度: "<<list1.Length()<<endl;
    list2.PrintList();
}
```

【例7.5】链表类模板

```
cout<<"请输入一个要求删除的整数"<<endl;
cin>>j;
P1=list1.Find(j);
if(P1!=NULL){
    P1=list1.DeleteNode(P1);
    delete P1;
    list1.PrintList();
    cout<<"list1 长度: "<<list1.Length()<<endl;
}
else cout<<"未找到"<<endl;
list1.MakeEmpty();                                //清空list1
for(i=0;i<16;i++){
    P1=list1.CreatNode(a[i]);
    list1.InsertOrder(P1);                            //升序创建list1
}
list1.PrintList();
return 0;
}
```

顺序表vs.链表

顺序表(数组)	链表
一次建立, 连续, 有界	分次建立, 离散, 无界
直接随机访问 (通过下标访问成员)	间接顺序访问 (通过前驱/后继访问成员)
访问表成员的时间开销 与位置无关	访问表成员的时间开销与 位置有关
插入/删除表成员的时间 开销与位置有关	插入/删除表成员的时间开 销与位置无关

线性表之链表

- ◆ 线性表：一种含有 $n(\geq 0)$ 个同类结点的有限序列
 - ◆ 均匀性：各个结点具有相同的数据类型
 - ◆ 有序性：各个结点之间的相对位置是线性的
- ◆ 按访问方式不同，可分为
 - ◆ 直接随机访问：顺序表（数组）
 - ◆ 间接顺序访问：链表（参见7.2节）
 - ◆ 双端单向访问：队列（参见7.3节）
 - ◆ 单端双向访问：栈（参见7.3节）

栈(stack)&队(queue)

- ◆特殊的线性表：可由顺序表或链表实现
- ◆与顺序表/链表区别
 - ◆栈/队：操作（插入/删除）结点位置受限

栈

- ◆ 类比：筒子里的薯片
 - ◆ 操作位置固定
 - ◆ 从筒子口取出薯片
 - ◆ 从筒子口放入薯片
 - ◆ 后进先出
 - ◆ 先放入的薯片后取出



栈

◆之前提到过栈(stack)的概念

不同点

一种数据结构	一段内存空间
相对队(queue)而言	◆相对堆(heap)（自由存储区）而言 ◆也称为局部变量区

相同点

后进先出

栈

◆类比：筒子里的薯片

筒子	栈
最底下那片（固定）	栈底（固定）
最上面那片（在“最上面”加入/取出薯片，“最上面”的位置随薯片数量变化）	栈顶（插入、删除结点的位置，该位置不断变化着）
加入薯片	入栈（插入结点）
取出薯片	出栈（删除结点）
筒子中薯片的片数	栈大小
筒子满了	栈满
筒子空了	栈空

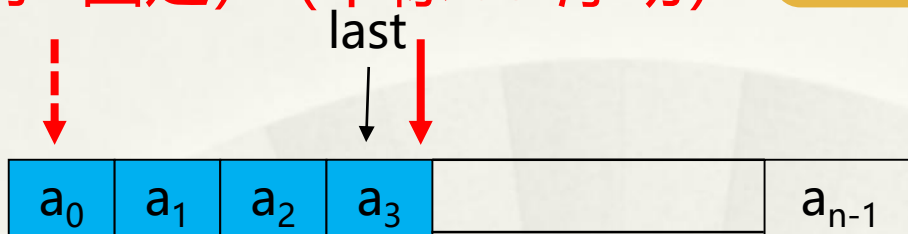
栈的实现方式

栈底
(下标0固定)

栈顶
(下标last浮动)

哪头栈顶？哪头栈底？

顺序栈

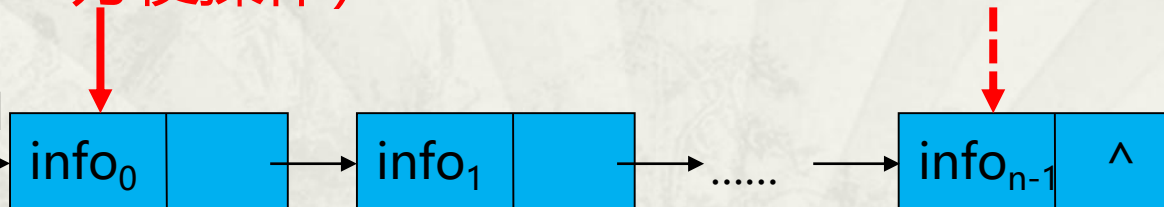


- ◆ 栈底：固定
- ◆ 栈顶：浮动，操作位置

链栈

head

栈顶
(head方便操作)



◆ 基本属性：栈顶

◆ 基本操作：入栈（插入结点）/出栈（删除结点）

链栈没有空表头，为什么？

不需要统一位置，只有一种插入位置

顺序栈类模板

//顺序表

```
template <typename T, int size>
```

```
class SeqList{
```

```
    T list [size];
```

```
    int max_size;
```

```
    int last;
```

```
public:
```

```
    SeqList();
```

```
    bool IsEmpty()const;
```

```
    bool IsFull()const;
```

```
    bool Insert(T & x, int i);
```

```
    bool Remove(T & x);
```

```
    void MakeEmpty();
```

```
}
```

//顺序栈

```
template <typename T, int size>
```

```
class Stack{
```

```
    T list [size];
```

```
    int max_size;
```

```
    int top;
```

```
public:
```

```
    Stack();
```

```
    bool IsEmpty( )const;
```

```
    bool IsFull( )const;
```

```
    void Push(const T & data);
```

```
    T Pop();
```

```
    void MakeEmpty();
```

```
}
```

顺序栈类模板

//顺序表

```
template <typename T, int size>
```

```
SeqList<T, size>::SeqList()
```

```
{  
    last=-1;  
    max_size=size;  
}
```

//顺序栈

```
template <typename T, int size>
```

```
Stack<T, size>::Stack()
```

```
{  
    top=-1;  
    max_size=size;  
}
```

空满判断

//顺序表

```
template <typename T, int size>
bool SeqList<T, size>::IsEmpty( ) const // 判断表是否空
{ return last == -1; }
```

```
template <typename T, int size>
bool SeqList<T, size>::IsFull( ) const // 判断表是否满
{ return last == max_size - 1; }
```

//顺序栈

```
template <typename T, int size>
bool Stack<T, size>::IsEmpty( ) const // 判断栈是否空
{ return top == -1; }
```

```
template <typename T, int size>
bool Stack<T, size>::IsFull( ) const // 判断栈是否满
{ return top == max_size - 1; }
```

压栈

//顺序表

```
template <typename T, int size>
bool SeqList<T, size>::Insert(T & x, int i)
{
    if (...) return false;
    else {...; return true;}
}
```

//顺序栈

```
template<typename T, >
void Stack<T>::Push(const T &data){
    assert(!IsFull()); //栈满则不能继续压栈，退出程序
    list[++top]=data; //栈顶指针先加1，元素再进栈
}
```

断言(assert)

#include <cassert>

void assert(int expression);

相当于: if(expression==FALSE) 退出程序;

功能: 对断定为真的条件进行核实, 当不满足时暴露问题

出栈

//顺序表

```
template <typename T, int size>
bool SeqList<T,size>::Remove(T & x){
    ....
}
```

//顺序栈

```
template<typename T, int size>
T Stack<T, size>::Pop(){
    assert(!IsEmpty()); //栈空则不能继续出栈，退出程序
    return list[top--]; //返回栈顶元素，同时栈顶指针退1
}
```


清空栈

//顺序表

```
template<typename T, int size>
void SeqList<T, size>::MakeEmpty(){
    last=-1;
}
```

//顺序栈

```
template<typename T, int size>
void Stack<T, size>::MakeEmpty(){
    top=-1;
}
```

【例7.8】顺序栈

```
int main(){
    int i,a[10]={0,1,2,3,4,5,6,7,8,9},b[10];
    Stack<int> istack(10);
    for(i=0;i<10;i++) istack.Push(a[i]);
    if(istack.IsFull()) cout<<"栈满"<<endl;
    istack.PrintStack();
    for(i=0;i<10;i++) b[i]=istack.Pop();
    if(istack.IsEmpty()) cout<<"栈空"<<endl;
    for(i=0;i<10;i++) cout<<b[i]<<"\t";           //注意先进后出
    cout<<endl;
    istack.Pop();                                   //下溢出，断言(assert)失败
    return 0;
}
```

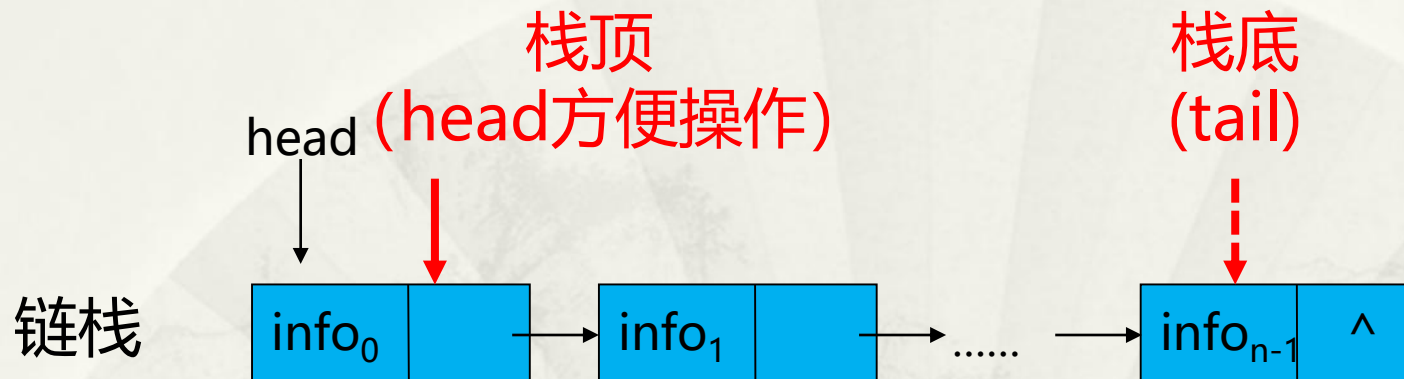
栈满

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

栈空

9	8	7	6	5	4	3	2	1	0
---	---	---	---	---	---	---	---	---	---

链栈



- ◆入栈：插入结点作为新表头
- ◆出栈：弹出表头
- ◆实现栈时，不再需要空表头

链栈类模板

//链栈结点

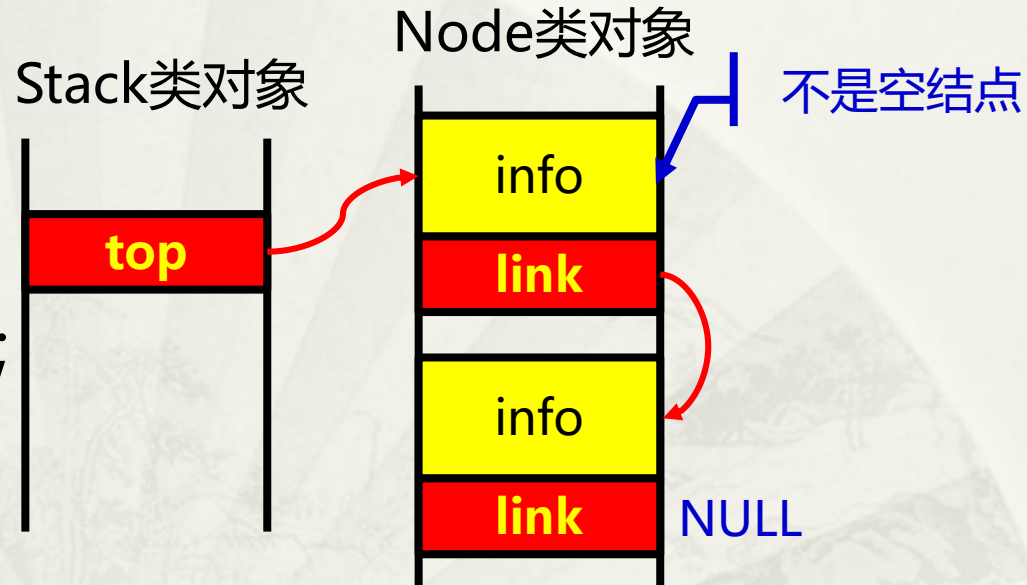
```
template<typename T>class Stack;
template<typename T>class Node
{
    T info;
    Node<T> *link;
public:
    Node(T data=0, Node<T> *next=NULL)
    {
        info=data;
        link=next;
    }
    friend class Stack<T>;
};

//用于链表类的结点类的构造函数
template <typename T>
Node<T>::Node(const T & data){
    info=data;
    link=NULL;
}
```

链栈类模板

//链栈

```
template<typename T>class Stack{  
    Node<T> *top; //栈顶指针  
public:  
    Stack();  
    ~Stack();  
    void MakeEmpty();  
    bool IsEmpty();  
    void Push(const T &data);  
    T Pop();  
    T GetTop();  
}
```



构造/析构函数

//构造函数

```
template<typename T>
Stack<T>::Stack(){
    top=NULL;
}
```

链表类的构造函数：
head=tail=new Node<T>();

//析构函数

```
template<typename T>
Stack<T>::~~Stack(){
    MakeEmpty();
}
```

链表类的析构函数：
MakeEmpty();
delete head;

清空栈/判断栈空

//清空栈

```
template<typename T>
void Stack<T>::MakeEmpty()
{
    Node<T> *temp;
    while(top!=NULL){
        temp=top;
        top=top->link;
        delete temp;
    }
}
```

//判断栈空

```
template<typename T>
bool Stack<T>::IsEmpty(){
    return top==NULL;
}
```

//判断栈满

```
template<typename T>
bool Stack<T>::IsFull(){
```

链表会“满”吗？

链栈中没有IsFull()函数

压栈/出栈

//压栈

```
template<typename T>
void Stack<T>::Push(const T &data){
    top=new Node<T>(data,top);
}
```

//出栈

```
template<typename T>
T Stack<T>::Pop(){
    assert(!IsEmpty());
    Node<T> *temp=top;
    T data=temp->info;
    top=top->link;
    delete temp;
    return data;
}
```

结点类的构造函数

```
Node(T data=0, Node<T> *next=NULL) {
    info=data;
    link=next;}

```

为什么出栈要判断是否为空栈，而压栈不需要判断是否为满栈？

取栈顶

```
template<typename T>
T Stack<T>::GetTop()
{
    assert(!IsEmpty());
    return top->info;
}
```

顺序栈vs.链栈

- ◆本质上是顺序表和链表的区别
- ◆逻辑功能一致，物理结构不同

顺序栈	链表栈
空间预先开辟 存在浪费	空间随用随开 节省空间
压栈/出栈使用已开辟的空间， 速度快	压栈/出栈→空间分配/释放， 速度慢
可能满	不可能满
可能空	可能空

队

- ◆ 类比：排队，而且不允许插队
 - ◆ 操作位置固定
 - ◆ 入队的位置：队尾
 - ◆ 出队的位置：队首
 - ◆ 先进先出

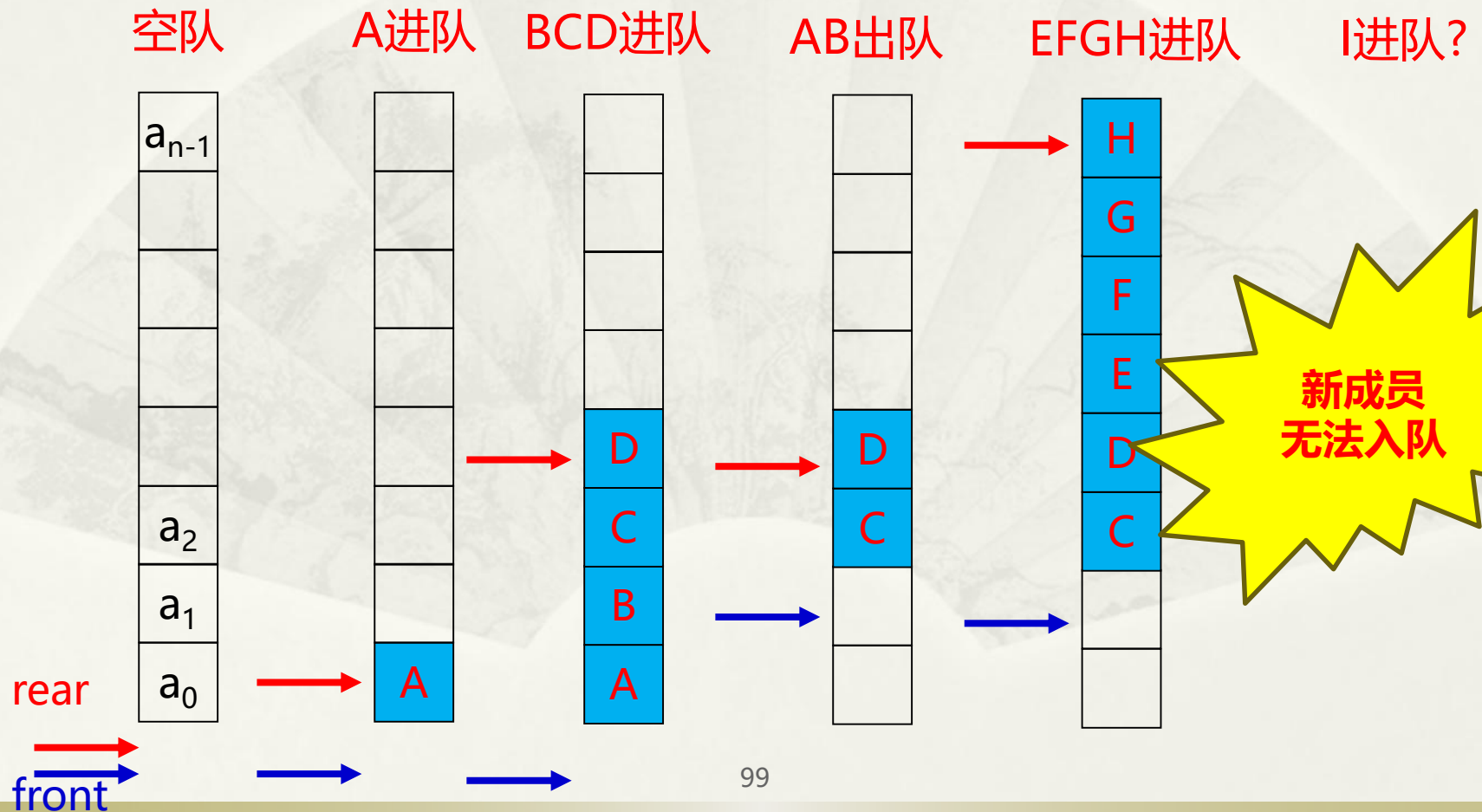
栈vs.队

		栈	队
相同点	数据结构类型	操作（插入/删除结点）位置受限的线性表（顺序表/链表）	
不同点	操作位置	单端双向 ◆栈顶：压栈/出栈	双端单向 ◆队尾：入队 ◆队首：出队
	操作顺序	后进先出	先进先出

顺序表队列

rear: 队尾下标

front: 队首下标

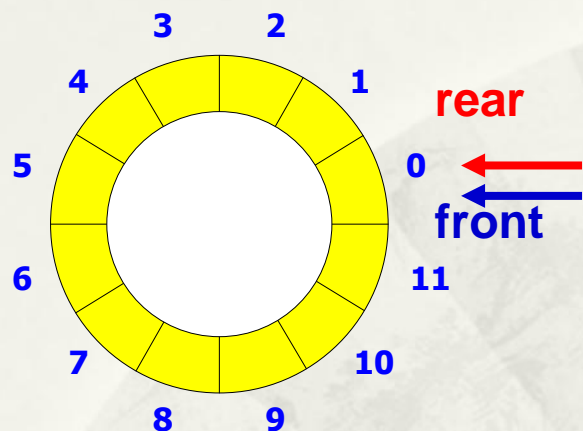


循环队列

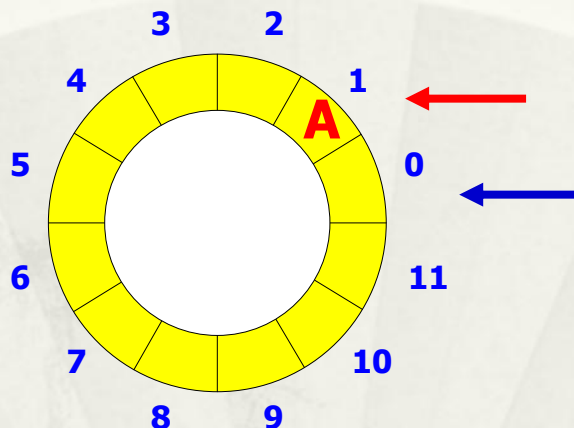
rear: 队尾下标

front: 队首下标

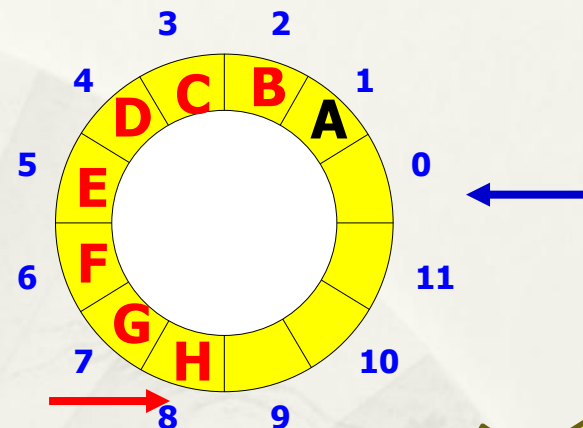
空队



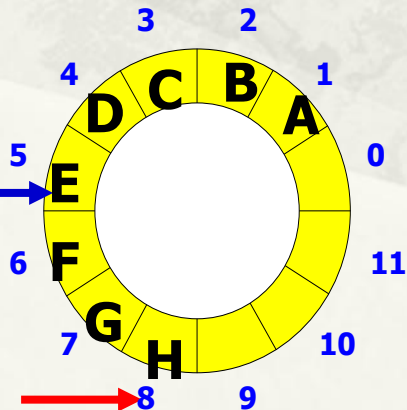
A进队



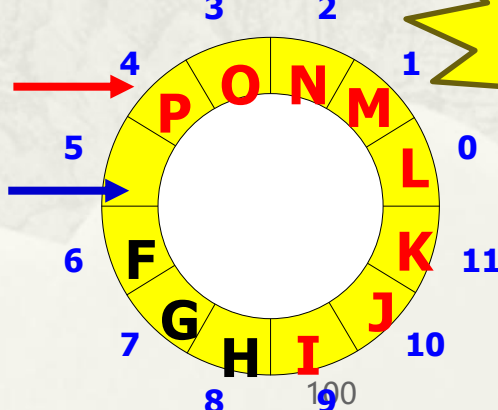
BCDEFGH进队



ABCDE 出队

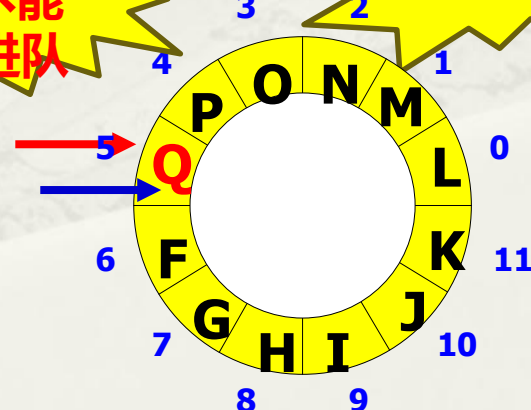


IJKLMNOP进队



满队,
Q不能
再进队

Q进队



无法分辨
空or满

循环队列类模板

```
template<typename T>class Queue{
    int rear,front;        //队尾/队首
    T *elements;           //动态数组指针
    int MaxSize;           //最多只能容纳MaxSize-1个
public:
    Queue(int ms);
    ~Queue();
    void MakeEmpty();      //清空
    bool IsEmpty() const;  //是否满
    bool IsFull() const;   //是否空
    int Length() const;    //长度
    void EnQueue(const T &data); //入队
    T DeQueue();           //出队
    T GetFront();          //取队首数据
}
```

构造/析构函数

//构造函数

```
template<typename T>
Queue::Queue(int ms){
    MaxSize=ms;
    rear=front=0;
    elements=new T[MaxSize];
    assert(elements!=NULL); //断言分配成功
}
```

//析构函数

```
template<typename T>
Queue::~~Queue(){
    delete[] elements;
}
```

清空队列

```
template<typename T>
void T Queue<T>::MakeEmpty()
{
    front=rear=0;
}
```

判断空满/计算长度

//是否空

```
template<typename T>  
bool Queue::IsEmpty() const  
{return rear==front;}
```

//是否满

```
template<typename T>  
bool Queue::IsFull() const  
{return (rear+1)%MaxSize==front;}
```

//队列长度

```
template<typename T>  
int Queue::Length() const  
{return (rear-front+MaxSize)%Max_size;}
```

入队/出队

//进队

```
template<typename T>
void Queue<T>::EnQue(const T&data){
    assert(!IsFull());           //断言队列不满
    rear=(rear+1)%MaxSize;       //更新队尾下标
    elements[rear]=data;         //从队尾入队
}
```

//出队

```
template<typename T>
T Queue<T>::DeQue(){
    assert(!IsEmpty());          //断言队列不空
    front=(front+1)%MaxSize;      //更新队首下标
    return elements[front];       //从队首出队
}
```

取队首数据

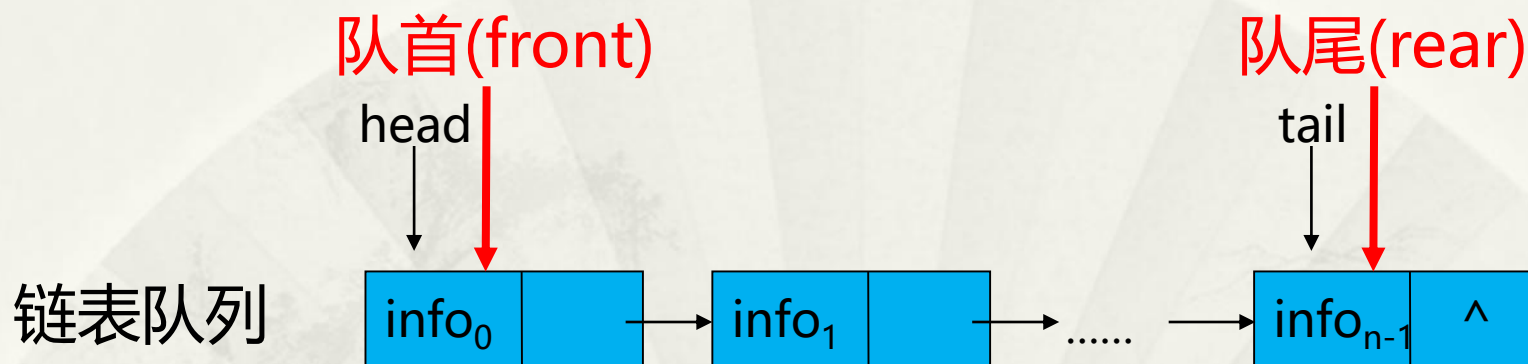
```
template<typename T>
T Queue<T>::GetFront()
{
    assert(!IsEmpty());
    return elements[(front+1)%MaxSize];
}
```

【例7.10】顺序队

```
int main(){
    int i;
    Queue<char> que;           //默认为18元素队列，可用17个元素，包括串结束符
    char str1[]="abcdefghijklmnp";
    que.MakeEmpty();
    for(i=0;i<17;i++) que.Enqueue(str1[i]);
    if(que.IsFull()) cout<<"队满";
    cout<<"共有元素: "<<que.Length()<<endl;
    for(i=0;i<17;i++) cout<<que.DeQueue();    //先进先出
    cout<<endl;
    if(que.IsEmpty()) cout<<"队空";
    cout<<"共有元素: "<<que.Length()<<endl;
    return 0;
}
```

队满共有元素: 17
abcdefghijklmnp
队空共有元素: 0

链表队列



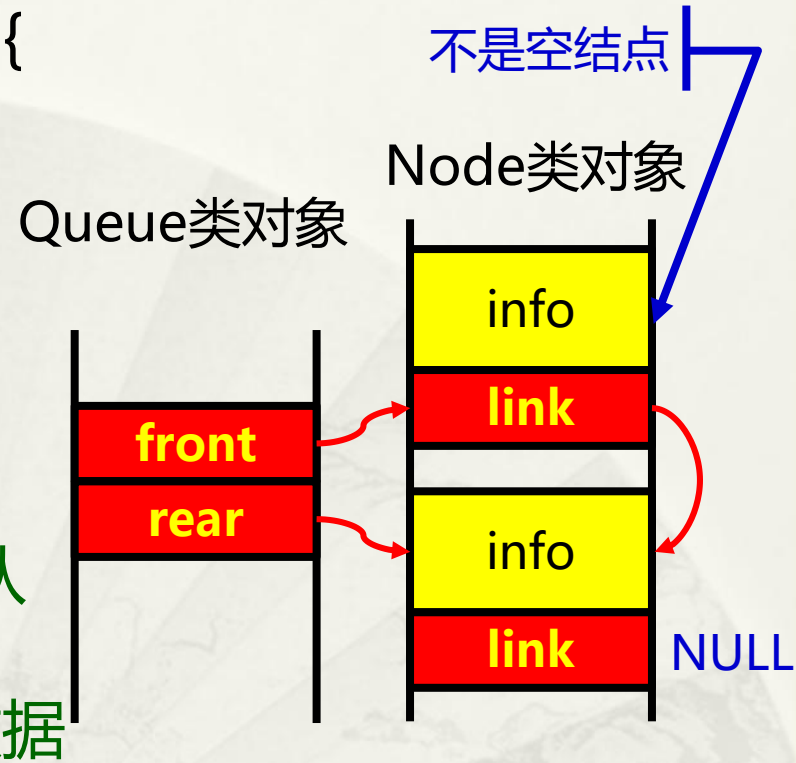
- ◆队首出队：删除表头
- ◆队尾进队：添加表尾
- ◆实现队列时，同样不再需要空表头

链队结点类模板

```
template<typename T>Queue;           //链队类模板
template<typename T>class Node{      //链队结点类模板
    T info;
    Node<T> *link;
public:
    Node(T data=0,Node *next=NULL)
    {info=data; link=next};
    friend class Queue<T>;
};
```

链队类模板

```
template<typename T>class Queue{
    Node<T> *front,*rear;
public:
    Queue();
    ~Queue();
    void MakeEmpty();           //清空队列
    bool IsEmpty()              //是否空
    void EnQueue(const T &data); //进队
    T DeQueue();                //出队
    T GetFront();               //取队首数据
}
```



构造/析构函数

//构造函数

```
template<typename T>
Queue<T>::Queue(){
    rear=front=NULL;
}
```

//析构函数

```
template<typename T>
Queue<T>::~~Queue(){
    MakeEmpty();
}
```

清空队/判断队空

//清空队

```
template<typename T>
void Qunue<T>::MakeEmpty()
{
    Node<T> *temp;
    while(front!=NULL){
        temp=front;
        front=front->link;
        delete temp;
    }
}
```

//判断队空

```
template<typename T>
bool Qunue<T>::IsEmpty(){
    return NULL==front;
}
```

同链栈，链队中没有IsFull()函数

入队/出队

//入队

```
template<typename T> void Queue<T>::EnQue(const T &data){  
    if(NULL==front) //空队, 新结点既是队尾, 也是队首  
        front=rear=new Node<T>(data, NULL);  
    else //非空队, 新结点是队尾  
        rear=rear->link=new Node<T>(data, NULL);  
}
```

//出队

```
template<typename T> T Queue<T>::DeQue(){  
    assert(!IsEmpty());  
    Node<T> *temp=front;  
    T data=temp->info; //取队首结点数据  
    front=front->link; //队首出队  
    delete temp; //释放队首结点空间  
    return data;  
}
```

取队首数据

```
template<typename T>
T Queue<T>::GetFront()
{
    assert(!IsEmpty());
    return front->info;
}
```


顺序表/链表实现栈/队

		栈	队
顺序表	空	$top == -1;$	$front == rear;$
	满	$top == ms - 1;$	$front == (rear + 1) \% ms;$
	添加	$top++;$	$rear = (rear + 1) \% ms;$
	删除	$top--;$	$front = (front + 1) \% ms;$
链表	空	$NULL == top;$	$NULL == rear;$ $NULL == front;$
	添加	表头前添加	表尾后添加
	删除	删除表头	删除表头

本章小结(1)

◆动态内存分配 (7.1.1, 7.1.2节)

◆在哪分配

- ◆堆(heap), 又称自由存储区

- ◆区别于: 栈(stack), 局部变量区; 全局与静态变量区; 代码区

◆如何分配/释放: 操作符new和delete

◆new

- ◆返回指针类型

- ◆可分配变量(对象)并初始化, 也可分配数组, 但二者只能选一种

◆delete

- ◆通过指针变量释放

- ◆如释放数组空间, 要加[]

◆注意问题

- ◆分配失败 (new后可能)

- ◆空悬指针 (delete后一定是)

- ◆内存泄露 (少delete时)

- ◆重复释放 (多delete时)

- ◆动态分配对象的生命期 (new后, delete前, 二者之间)

本章小结(2)

◆浅复制与深复制（7.1.3节）

- ◆讨论场景：类对象的复制

- ◆什么时候会有类对象的复制

 - ◆一个类对象给另一个类对象复制（赋值操作符重载）

 - ◆初始化类对象（复制构造函数）

 - ◆类对象作为参数值传递（复制构造函数）

 - ◆类对象作为返回值值传递（复制构造函数）

- ◆浅复制：仅复制每个类成员变量的值

- ◆深复制

 - ◆对于类成员变量不是指针类型的，同浅复制

 - ◆是指针类型的，先动态分配空间，再复制空间的值

- ◆浅复制的错，在于

 - ◆两个对象使用时

 - ◆一个对象析构，而另一个对象


 - ◆析构前（仍在使用时）

 - ◆析构后

- ◆深复制要做哪些事：复制构造函数；赋值操作符重载函数；析构函数

本章小结(3)

◆链表 (7.2 节)

- ◆用处：每新增一个表成员，分配一个新结点；反之亦然。节省空间。
- ◆结点构成：数据域；指针域（单链表中有且仅有一个后继指针）
- ◆操作要点
 - ◆表头指针head不能丢
 - ◆插入/删除结点时，关键在于修改前一个结点的指针域
 - ◆增加时，动态分配新结点；删除时，动态释放原有结点（可选）
- ◆链表操作的关键：
- ◆查找/打印/数个数：表头起→向后逐个结点→直至表尾
- ◆插入结点：关键找到前一个结点；为什么要有空表头
 - ◆先改新结点后继
 - ◆再改前一个结点后继
- ◆插入结点基础上：前向生成（插在表头后）；后向生成（插在表尾后）；
升序生成
- ◆删除结点：关键找到前一个结点
 - ◆先存待删结点
 - ◆再改前一个结点后继

本章小结(4)

◆栈 (7.3.1节)

- ◆特征：单端双向；后进先出

- ◆顺序栈

 - ◆栈顶在表尾

 - ◆压栈：简化的插入结点；出栈：简化的删除结点

- ◆链栈

 - ◆栈顶在表头（无空表头）

 - ◆不会满

- ◆压栈：简化的插入结点；出栈：简化的删除结点（位置明确）

◆队 (7.3.2节)

- ◆特征：双端单向；先进先出

- ◆顺序队

 - ◆必须用循环队（为什么栈的时候不存在）

 - ◆注意队首/队尾间的关系（空/满/数个数/入队/出队）

- ◆链队

 - ◆队首即表头（无空表头），队尾即表尾

 - ◆不会满

- ◆入队：简化的插入插入结点；出队：简化的删除结点（位置明确）



End