

线性表

吉林大学计算机学院
谷方明

fmgu2002@sina.com





学习目标

- 熟悉线性表的定义、特性、基本操作
- 掌握顺序表
- 掌握单链表、循环链表、双向链表
- 掌握跳舞链（拓展）
- 掌握静态链表（拓展）



线性表的例子

- 数 列 (1, 2, 4, 8, 16)
- 字符串 **“You cannot improve your past,
but you can improve your future.”**
- 栈和队列



线性表的定义

- 线性表是由零个或多个具有**相同类型**的结点组成的**有序集合**。

$$\text{线性表记为} \left\{ \begin{array}{ll} (a_0, a_1, \dots, a_{n-1}) & n > 0 \\ () & n = 0 \quad \text{空表} \end{array} \right.$$



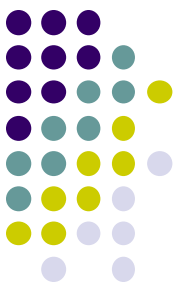
术语和特性

- 称 a_0 为线性表的**头结点**(简称表头), 称 a_{n-1} 为线性表的**尾结点**(简称表尾)
- a_i 为 a_{i+1} 的**前驱结点**(简称前驱), a_{i+1} 为 a_i 的**后继结点**(简称后继)。
- **线性表特性**: 除表头和表尾外, 每个结点都有唯一的前驱和后继。



线性表的基本操作

1. 创建一个线性表;
 2. 确定线性表的长度;
 3. 判断线性表是否为空;
 4. 在表中第 k 个结点后**插入**一个新结点;
 5. **删除**表中第 k 个结点;
 6. **存取**线性表中第 k 个结点的字段值;
 7. **查找**指定字段值在表中的位置;
 8.
- } 基于位置
- } 基于关键词



线性表的顺序存储

- 将线性表的结点按**逻辑顺序**依次存放在**一块地址连续的**内存空间中。顺序存储的线性表也称为**顺序表**。
- 实现顺序存储的最便捷的方法是使用一维数组。
 - ✓ 存储顺序表的数组： **$T * \text{element}$** ;
 - ✓ 顺序表的实际长度： **int length** ;
 - ✓ 顺序表的最大长度： **int maxSize** ;



顺序表结构的声明

```
template<class T>
struct LinearList{
    T * element;
    int length;
    int maxSize;
    LinearList(int size); //init(int size)
    ~LinearList(); //clear
    void ins(int k, int x);
    void del(int k, int* x);
};
```




顺序表的初始化和释放

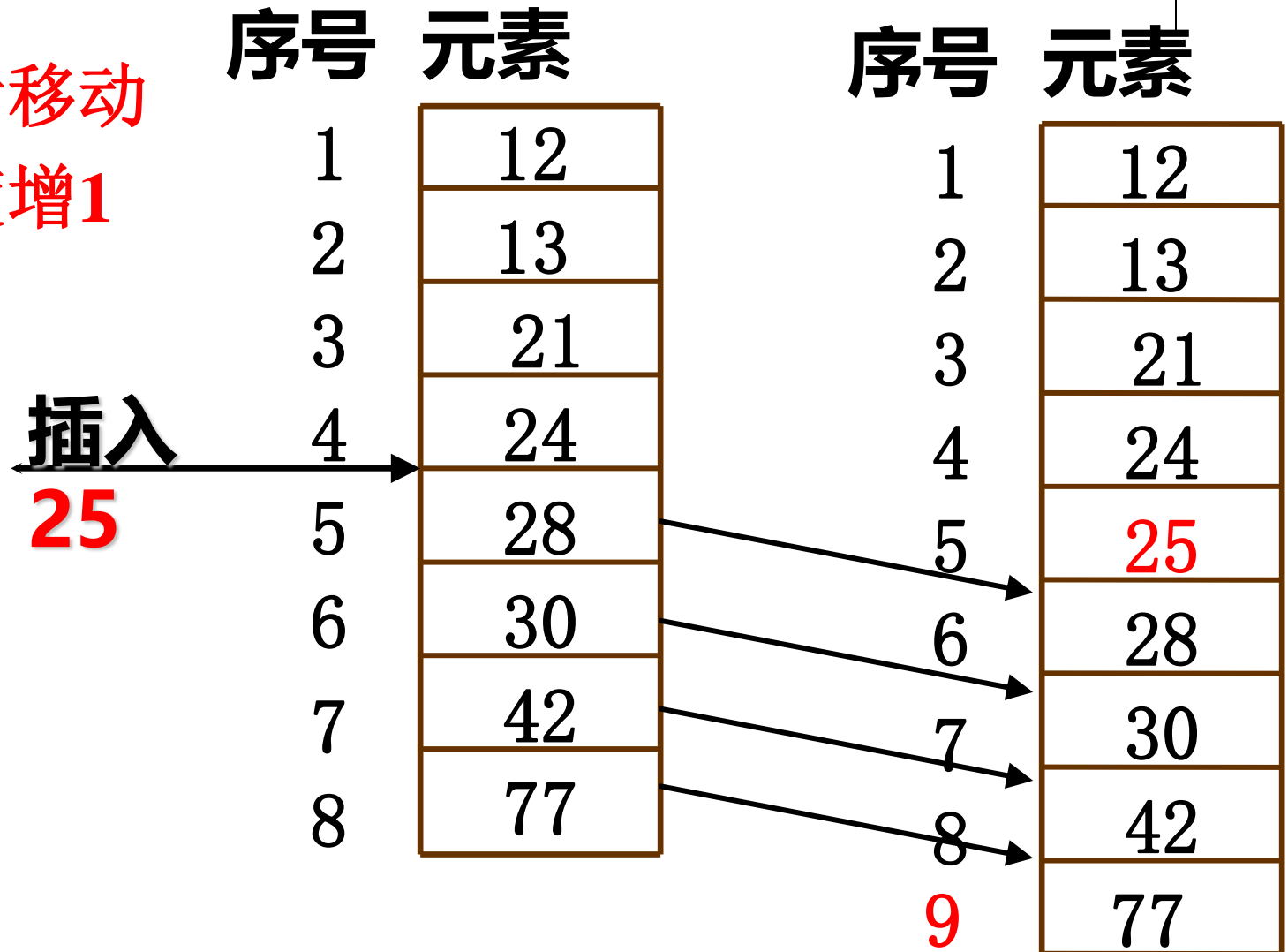
```
template<class T>
LinkedList<T>::LinkedList(int size){
    maxSize = size;
    element = (T*)malloc(size*sizeof(T));//new
    length = 0;
}

template<class T>
LinkedList<T>::~~LinkedList(){
    free(element);//delete
}
```



顺序表的插入

- 元素移动
- 长度增1





插入算法ADL描述

算法 **Insert**(A, k, item)

/* 在第 k 个结点**后**插入值为 item 的结点 */

I1.[**插入合法?**]

IF (k<**0** OR k>**length** OR **length=maxSize**)

THEN (PRINT “**插入不合法**” . RETURN).

I2.[**插入**]

FOR i=length TO k+1 STEP -1 DO **A[i+1]←A[i]**.

A[k] ← item.

length ← length+1. ■



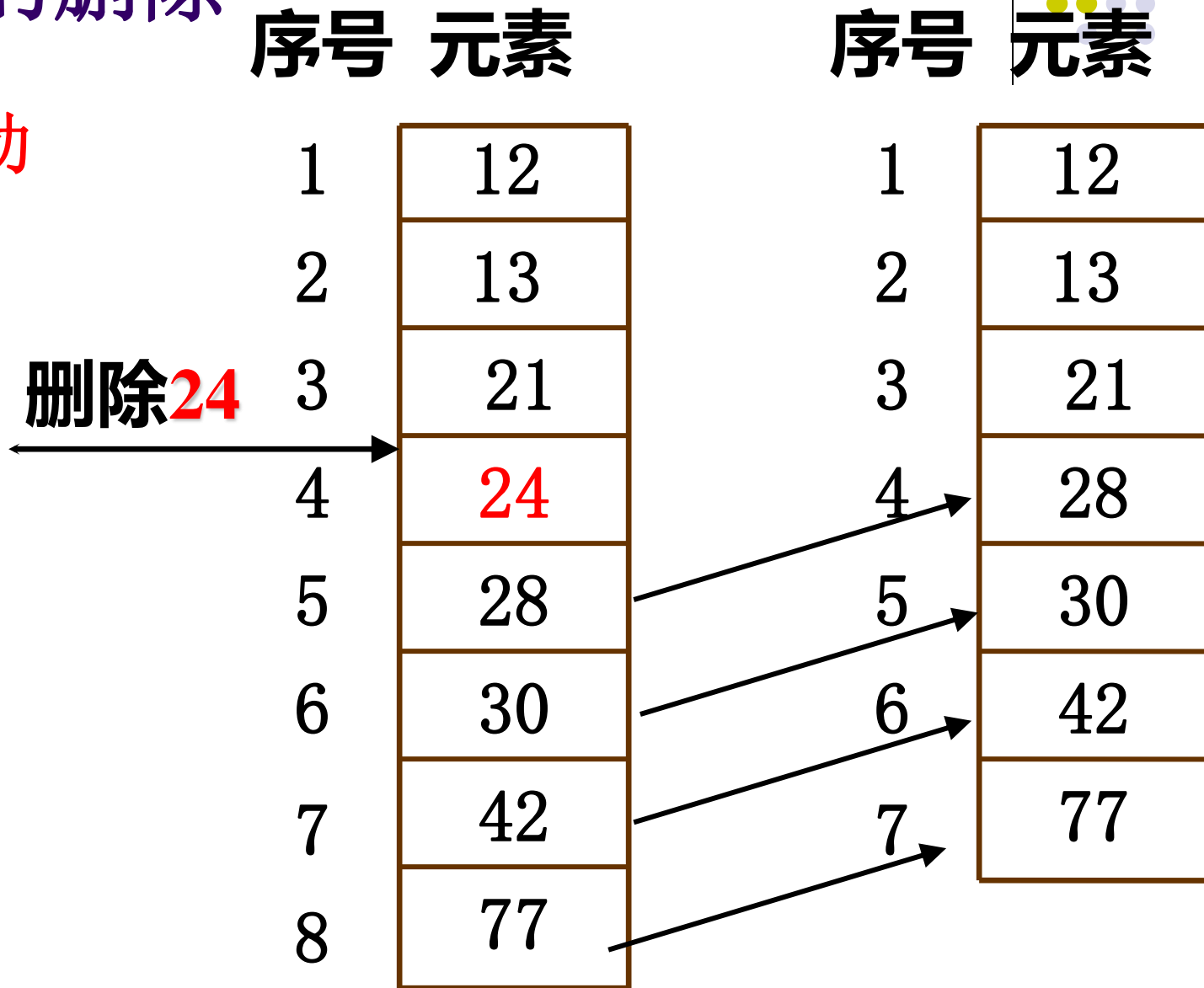
插入操作时间复杂度

- 基本运算：元素移动（表现为元素赋值）
- 最坏时间复杂度
 - ✓ $W(n) = n$ // 若包含 $A[k] \leftarrow \text{item}$. 计数加1
- 期望时间复杂度
 - ✓ $n+1$ 个位置可以发生插入，设插入成功且插入到各位置的概率相同： $1/(n+1)$
 - ✓ $E(n) = (n + (n-1) + \dots + 1 + 0) / (n+1) = n/2$
- $O(n)$



顺序表的删除

- 元素移动
- 长度减1





删除算法ADL描述

算法 **Delete**(A, k)

/* 删除顺序表 A 中第 k 个结点 */

D1.[**k合法?**]

IF (**k<1** OR **k>length** OR **length=0**)

THEN (PRINT“删除不合法” . RETURN).

D2.[**删除**]

FOR i=k+1 TO **length** DO **A[i-1]←A[i]**.

length ← length-1. ■



删除操作时间复杂度

- 基本运算：元素移动
- 最坏时间复杂度
 - ✓ $W(n) = n - 1$
- 期望时间复杂度
 - ✓ n 个位置可以发生删除，设删除成功且各位置被删除的概率相等： $1 / n$
 - ✓ $E(n) = ((n-1) + \dots + 1 + 0) / n = (n-1)/2$
- $O(n)$



存取和查找

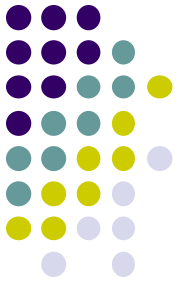
□ 存取/查找第 k 个结点

- ✓ 下标运算
- ✓ 一次命中, $O(1)$

□ 查找元素出现的位置

- ✓ 从前向后逐个比较
- ✓ $O(n)$

测试



```
int main(){
    int i,x;
    LinearList<int> li(10);
    for(i=0;i<5;i++)
        li.ins(i,i+1);
    li.del(2,&x);
    //li.pri()

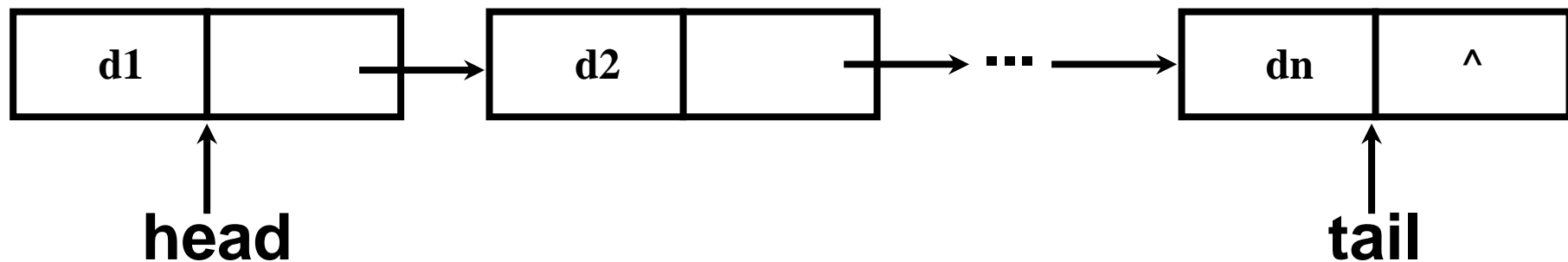
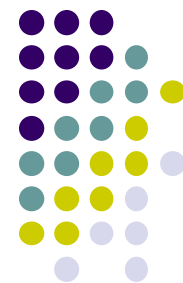
    return 0;
}
```



线性表的链接存储

- 每个存储单元包含结点的值和逻辑相邻结点的地址信息（指针域）。链接存储的线性表也叫**链表**。
- 链表结点的指针域根据需要进行设计。**最常见的是单链表，通常也简称为链表。**

单链表



- ❑ 链表的第一个结点被称为**头结点**(也称为表头), 指向头结点的指针被称为**头指针(head)**.
- ❑ 链表的最后一个结点被称为**尾结点**(也称为表尾), 指向尾结点的指针被称为**尾指针(tail)**.



单链表结构的声明

```
template<class T>
struct SLNode{
    T data;
    SLNode* next;
};
```



```
template<class T>  
struct SLList{  
    SLLNode<T>* head;  
    SLList(); //init  
    ~SLList(); //clear  
    SLLNode<T>* find(int k);  
    void ins(int k,T x);  
    void del(int k,T& x);  
    //void pri();  
};
```



单链表的初始化和释放

```
template<class T>
SLList<T>::SLList(){
    head = new SLNode<T>; //哨位结点
    head->next = NULL;
}
template<class T>
SLList<T>::~~SLList(){
    for(SLNode<T>* p=head;head;p=head){
        head = head->next;
        delete p;
    }
}
```



单链表:查找/存取第k个元素

算法 **Find** (*head*, *k*. *p*)

// 查找链表第*k*个结点, 找到用*p*指向, 其它情况 *p*为NULL

F1. [*k*合法?]

IF (*k* < 0) THEN (*p* ← NULL. RETURN .)

F2. [初始化]

***p* ← head . *i* ← 0.** //有哨兵变量

F3. [找第*k*个结点]

WHILE (*p* ≠ NULL AND *i* < *k*) DO (*p* ← next(*p*) . *i* ← *i* + 1.)



存取/查找第k个的时间复杂度

- 基本运算： 比较（定位）
- 最坏情况下的时间复杂度为 $O(n)$;
- 平均情况下，假设 $k < 1$, $k = 1, \dots, k = n$, $k > n$ 的概率相同，即每种情况的发生概率为 $1/(n+2)$ ，则**WHILE**循环的执行次数平均为

$$\frac{0 + 1 + \dots + n + n}{n + 2} = \frac{1}{n + 2} \left(\sum_{k=1 \dots n} k + n \right) = \frac{1}{n + 2} \left(n(n + 1)/2 + n \right) = \frac{n(n + 3)}{2(n + 2)} = O(n)$$

□ $O(n)$.



单链表:查找元素

算法 **Search** (*head*, *item*. *p*)

/*在链表中查找值为*item*的结点用*p*指向*/

S1. [初始化]

$p \leftarrow \text{next}(\text{head})$. //有哨兵变量

S2. [逐点访问]

WHILE ($p \neq \text{NULL}$ AND $\text{data}(p) \neq \text{item}$)

DO ($p \leftarrow \text{next}(p)$) ■



查找元素操作的时间复杂度

- 基本元算：元素比较

- 最坏时间复杂度

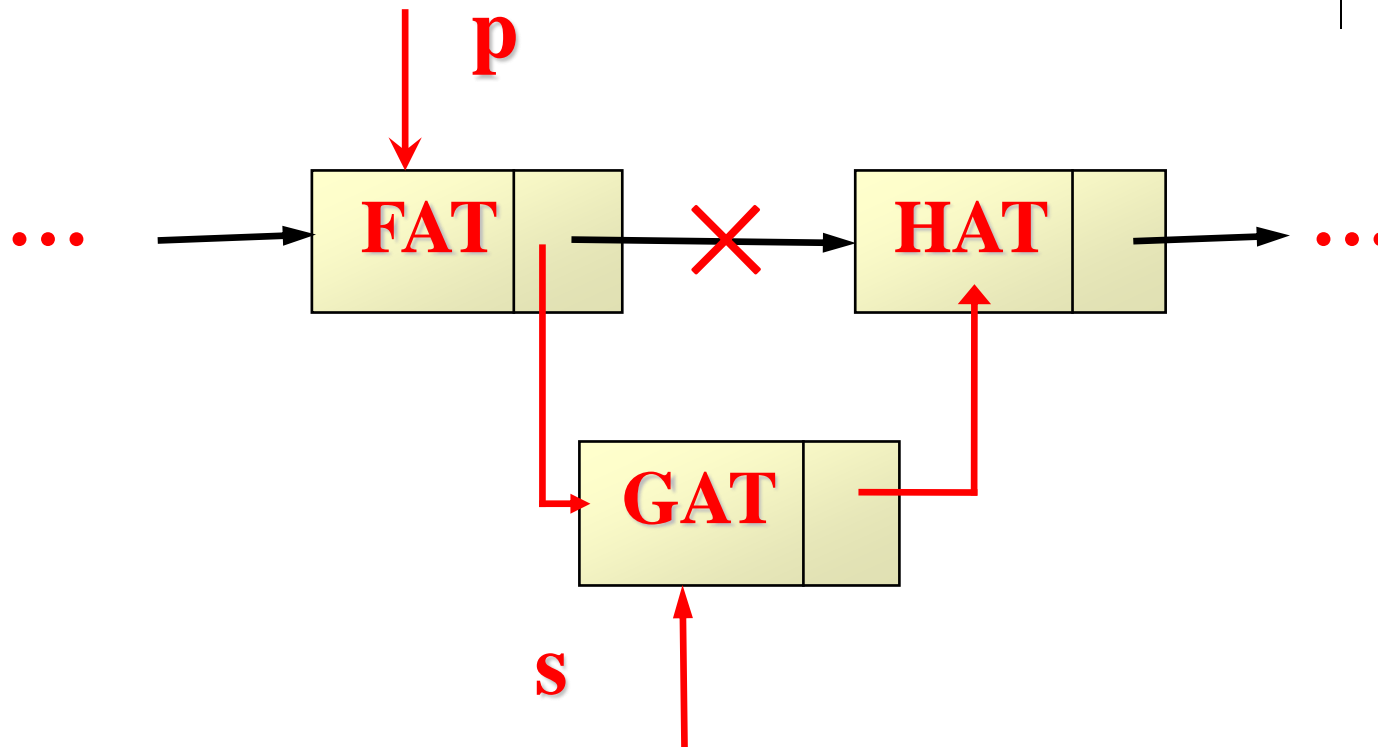
- ✓ $W(n) = n$

- 平均时间复杂度

- ✓ $E(n) = O(n)$

- $O(n)$

单链表的插入



$\text{next}(s) \leftarrow \text{next}(p)$

$\text{next}(p) \leftarrow s$



插入算法ADL描述

算法**Insert** (*head, k, item*)

// 在链表中第*k*个结点 后 插入字段值为*item*的结点

I1. [查找第*k*个结点]

Find(head,k,p).

IF $p = \text{NULL}$ THEN (PRINT “插入不合法” . RETURN.)

I2. [插入]

$s \leftarrow \text{AVAIL}$. $\text{data}(s) \leftarrow \text{item}$. $\text{next}(s) \leftarrow \text{next}(p)$.

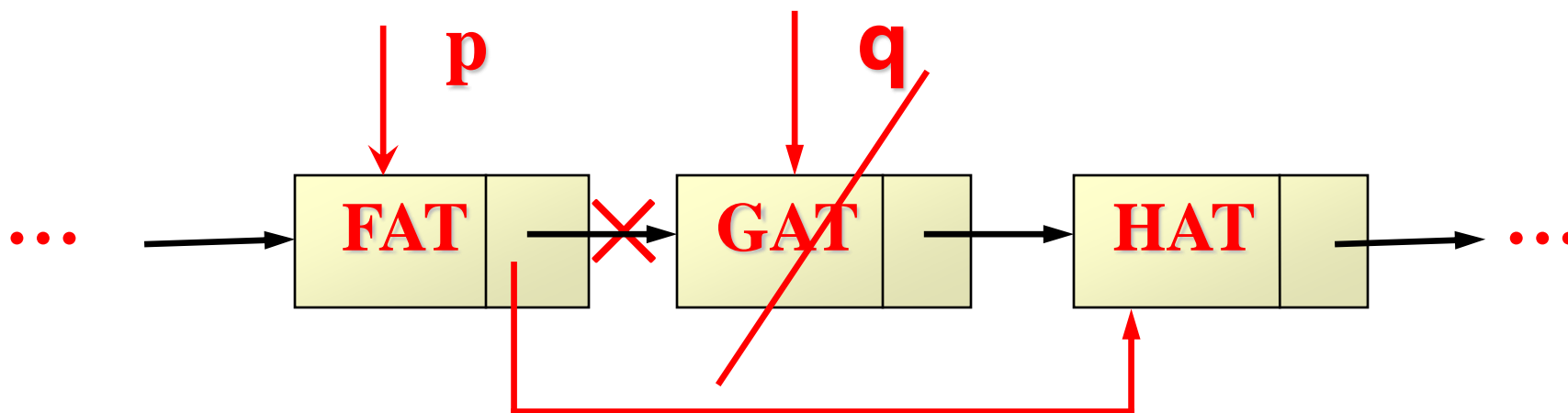
$\text{next}(p) \leftarrow s$. ■



插入操作时间复杂度

- 基本运算：比较（定位）
- 最坏时间复杂度
 - ✓ $W(n) = n$
- 期望时间复杂度
 - ✓ $n+1$ 个位置可以发生插入，设插入成功且插入到各位置的概率相同： $1/(n+1)$
 - ✓ $E(n) = (n + (n-1) + \dots + 1 + 0) / (n+1) = n/2$
- $O(n)$

单链表的删除



$q \leftarrow \text{next}(p). \text{next}(p) \leftarrow \text{next}(q) .$

$\text{AVAIL} \leftarrow q .$



删除算法ADL描述

算法**Delete** (*head*, *k*)

// 删除链表中第*k*个结点

D1. [找第*k-1*结点]

Find(head,k-1,p).

IF $p = \text{NULL}$ OR **next(p)** = NULL

THEN (PRINT “无此结点” . RETURN.)

D2. [删除]

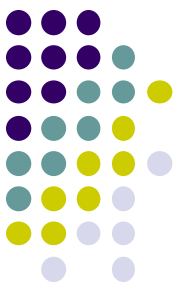
$q \leftarrow \text{next}(p)$. $\text{next}(p) \leftarrow \text{next}(q)$. // 修改p的next指针

AVAIL $\leftarrow q$. ■



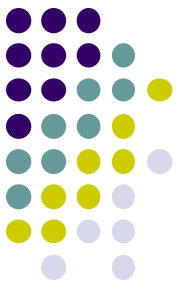
删除操作时间复杂度

- 基本运算：比较（定位元素）
- 最坏时间复杂度
 - ✓ $W(n) = n - 1$
- 期望时间复杂度
 - ✓ n 个位置可以发生删除，设删除成功且各位置被删除的概率相等： $1 / n$
 - ✓ $E(n) = ((n-1) + \dots + 1 + 0) / n = (n-1)/2$
- $O(n)$



顺序表VS链表：时间复杂度

- 线性表的基本操作是存取(第k个)、插入和删除。对于顺序表，随机存取非常容易，但插入或删除元素需要移动若干元素；对于链表，随机存取必须从表头开始遍历链表，但链表的插入和删除操作非常简便，只需修改一个或者两个指针值。
- 对于基于位置的插入和删除，两者时间复杂度都是 $O(n)$ ，顺序表的基本运算是元素移动，链表的基本运算是比较。一般来说，元素移动要大于比较。



- ❑ 对于基于关键词的操作，顺序表要增加查找的时间，耗时更多。
- ❑ 链表的插入和删除涉及到内存的申请和释放，需要调用系统函数，耗时更多。
- ❑ 当线性表要经常进行插入、删除操作时，链表的时间效率较高（动态表）；当线性表基本不需要插入和删除时，顺序表的时间效率较高（静态表）。



思考

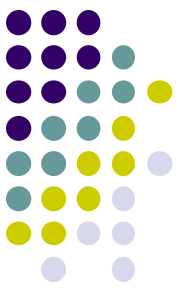
□ 栈和队列的插入删除的操作为何都是 $O(1)$ 的？

□ 栈

- ✓ 顺序表在表尾增删高效
- ✓ 链表在表头增删高效

□ 队列

- ✓ 删除用表头指针加速
- ✓ 插入用表尾指针加速



顺序表VS链表：空间复杂度

- 顺序表所占用的空间来自于申请的数组空间，数组大小是事先确定的。当表中的元素较少时，顺序表中的很多空间处于闲置状态，造成了空间的浪费；
- 链表所占用的空间是根据需要动态申请的，不存在空间浪费的问题，但是链表需要在每个结点上附加一个指针，产生一定的结构性开销



设 N ：线性表的长度；

D ：顺序表可存储结点个数；

s ：存储结点数据域所占字节数；

p ：存储指针所需字节数；

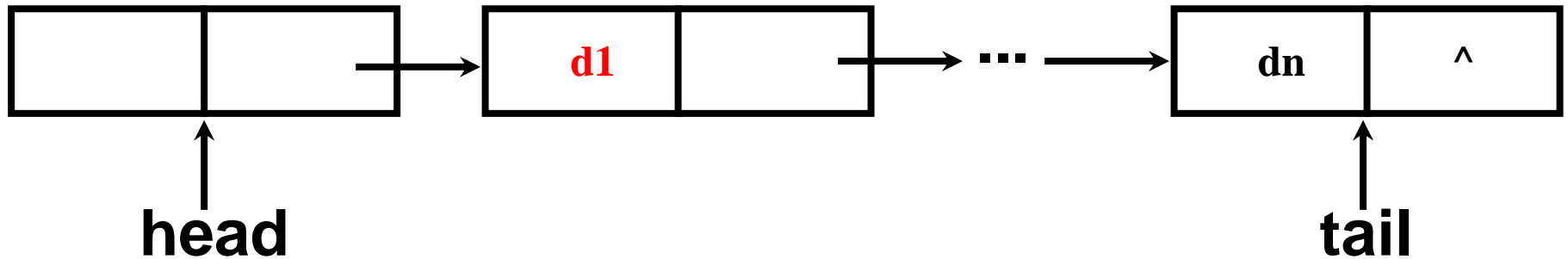
顺序表的空间需求为 $D \times s$ ，链表的空间需求为 $N \times (s + p)$ 。临界点为 $N = D \times s / (s + p)$

- 线性表长度 N 较大时，顺序表的空间效率较高，线性表长度 N 较小时，链表的空间效率较高



哨位结点

- 为了对表头结点插入、删除等操作的实现方便，通常在表的前端增加一个特殊的表头结点，称其为哨位结点（哨兵变量）



增加哨位结点的单链表



例： 删除值为x的结点

□ 无哨位结点版本

```
Node* p0,*p;  
if(head==NULL) return;  
if(head->next== NULL && head->data==x){  
    delete head; head=NULL;  
    return;  
}  
for( p=head ; p ; p0=p , p=p->next )  
    if(p->data==x){  
        p0->next=p->next;  
        delete p;  
        p=p0->next;  
    }
```



□ 有哨位结点版本

```
Node* p0=head,*p=p0->next;  
for( ; p ; p0=p, p=p->next)  
    if(p->data==x){  
        p0->next=p->next;  
        delete p;  
        p=p0->next;  
}
```




哨位结点的用处

- 标识第**0**个（逻辑上）结点；
- 避免讨论**head**为空的情况；
- 易于编程；插入、删除等操作统一规范，



线性表的应用

- 线性表应用极其广泛。
 - ✓ 序列、表格、字符串.....
 - ✓ 数组、矩阵、.....
 - ✓ 环、.....
 - ✓ 多项式、.....

- 线性表是最基本、最简单、也是最常用的一种数据结构。



例：多项式的计算系统

$$p(X) = a_n X^n + a_{n-1} X^{n-1} + \cdots + a_2 X^2 + a_1 X + a_0$$

- 加法、减法、微分、代入求值.....
- 增项、删项

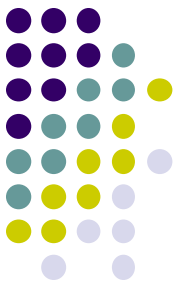


设计方案 I

- 在多项式的链表表示中每个结点增加了一个数据成员**Link**，作为链接指针。

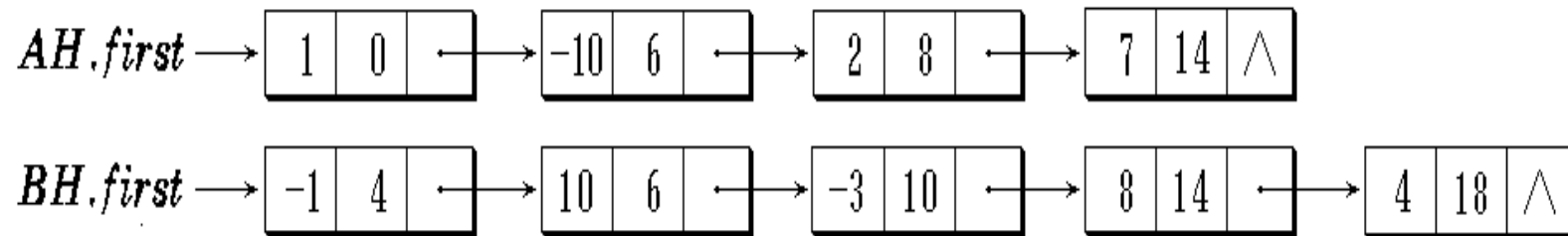
coef	Exp	Link
------	-----	------

- 优点：
 - ✓ 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。



$$AH = 1 - 10x^6 + 2x^8 + 7x^{14}$$

$$BH = -x^4 + 10x^6 - 3x^{10} + 8x^{14} + 4x^{18}$$



(a) 两个相加的多项式



(b) 相加结果的多项式



设计方案 II

□ 顺序存储(每一项保存系数和指数)

- ✓ 优点：实现容易，
- ✓ 缺点：增项、删项效率低

□ 顺序存储(每一项保存系数，指数用下标表示)

- ✓ 优点：处理简单
- ✓ 缺点：可能导致空间浪费严重，适用于指数较小的情况



STL顺序表: `vector`

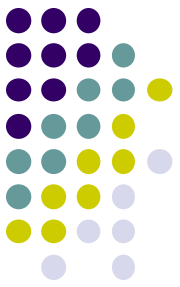
□ 动态数组 / 有序集合

- ✓ `at() // []`
- ✓ `insert()`
- ✓ `erase()`
- ✓ `size() // capacity`
- ✓ `vector<int>::iterator i;`
- ✓ `push_back();` // 末端操作效率高
- ✓ `pop_back();`
- ✓ `back();`
- ✓ `front();`



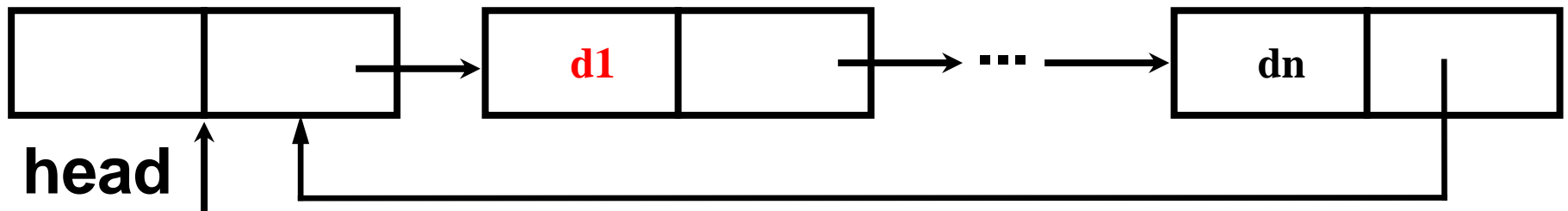
STL链表: list

- 双向链表, 不支持随机存取
 - ✓ insert()
 - ✓ erase()
 - ✓ size() //无capacity
 - ✓ list<int>::iterator i;
 - ✓ push_back();
 - ✓ pop_back();
 - ✓ push_front();
 - ✓ pop_front();
 - ✓ remove, splice, merge, sort,



循环链表

- 链接结构“循环化”，即表尾结点的**next**域存放指向哨位结点的指针，而不是存放空指针**NULL**，这样的单链表被称为循环链表。

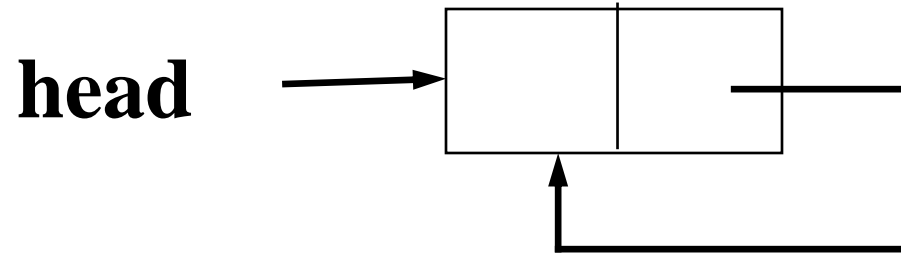


- 循环链表特性: 可**从链表的任何位置开始**，访问链表中的任一结点。



循环链表状态和操作

□ 空表

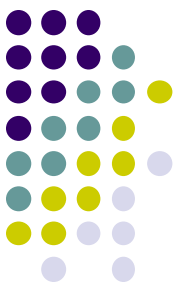


$\text{head} \rightarrow \text{next} == \text{head}$

□ 表尾

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

□ 其操作和单链表类似



例：约瑟夫问题

- 据说著名犹太历史学家 **Josephus** 的故事。
- 罗马人占领乔塔帕特，**39** 个犹太人与 **Josephus** 及其朋友躲到一个洞中。**39** 个犹太人宁愿死也不要被敌人抓到，于是决定了一个自杀方式。**41** 个人排成一个圆圈，由第**1**个人开始报数，每报数到第**3**人该人就必须自杀，然后再由下一个重新报数，直到所有人都自杀身亡为止。
- 然而 **Josephus** 及其朋友并不想遵从。他将朋友与自己安排在第**16**个与第**31**个位置。



简化版本

- **N**个人围成一圈，从第一个开始报数，第**M**个将被杀掉，最后剩下一个。
- 例如**N=6**，**M=5**，被杀掉的人的序号为依次**5**，**4**，**6**，**2**，**3**。最后剩下**1**号。
- 被杀掉的序号顺序？
- 最后剩下的人序号？



设计方案I

□ 模拟法：顺序存储（不真删）

- ✓ 法1：下标表示第*i*个人的编号
- ✓ 法2：第*i*个位置存放编号*i*

□ 效率分析

$$(M/2 + 1) * N + (M/3 + 1) * N + \dots + (M/N + 1) * N \\ < MN * \ln N + N(N-1)$$

- 如果不置0，每次都真删，要移动元素，请同学们分析效率。



设计方案II

- 模拟法：循环链表
- 时间效率： $M*(N-1)$
- 空间效率： $O(N)$

设计方案III

□ 数学公式





双向链表

- 双向链表（**Double-Linked List**）：链表中的结点由**data**域、**left**域(前驱地址)和**right**域(后继地址)构成。



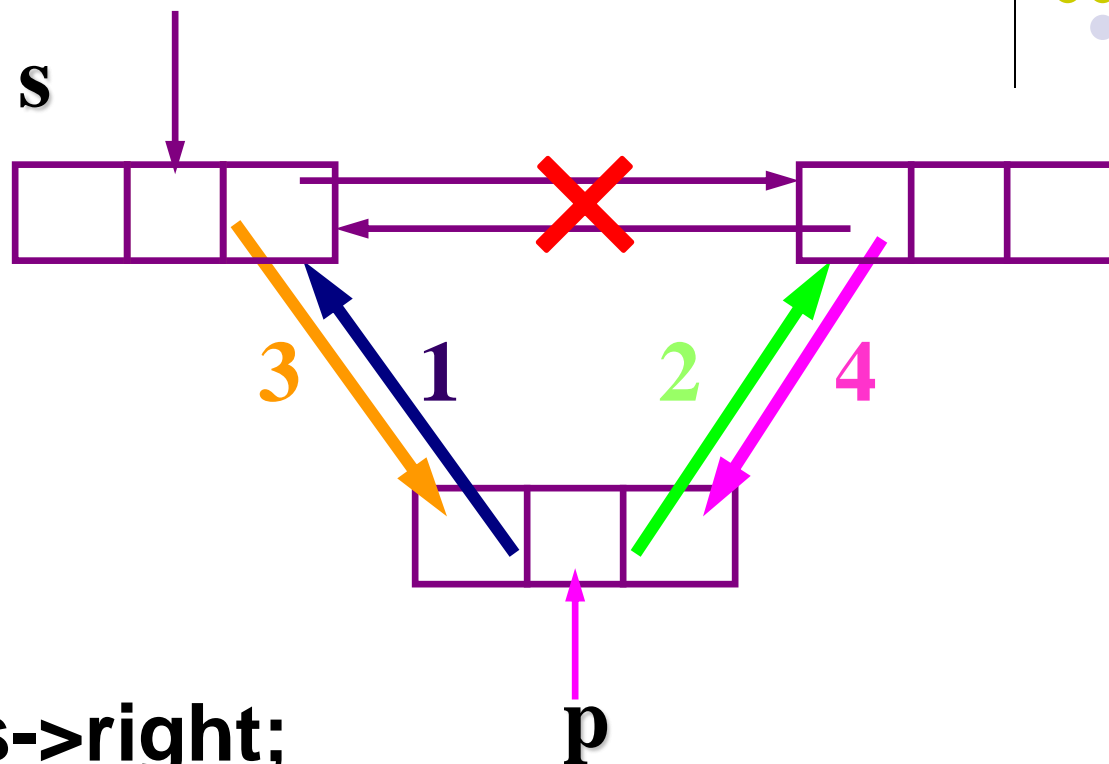
- 双向链表的特性：直接的双向访问能力
- 链表中表头结点的**left**指针和表尾结点的**right**指针均为**NULL**.



双向链表结点的结构

```
template<class T>
struct DLNode{
    T data;
    DLNode* left,*right;
};
```

插入算法



- $p \rightarrow \text{left} = s;$
- $p \rightarrow \text{right} = s \rightarrow \text{right};$
- $p \rightarrow \text{left} \rightarrow \text{right} = p;$
- $p \rightarrow \text{right} \rightarrow \text{left} = p;$



插入算法的ADL描述

算法**DLInsert** ($head, s, p.head$)

// 结点 s 的右边插入结点 p ,有哨位

DLI1.[特判: 空 和 s 是尾结点]

IF $s = \Lambda$ **OR** $p = \Lambda$ **THEN RETURN.**

IF $\text{right}(s) = \Lambda$ **THEN** ($\text{left}(p) \leftarrow s$. $\text{right}(p) \leftarrow \Lambda$. $\text{right}(\text{left}(p)) \leftarrow p$. **RETURN.**)

DLI2. [插入]

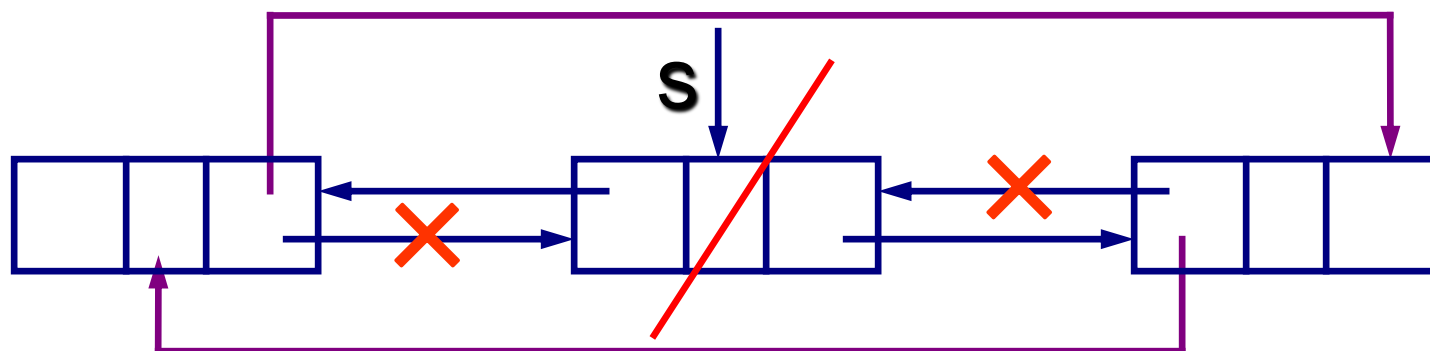
$\text{left}(p) \leftarrow s$.

$\text{right}(p) \leftarrow \text{right}(s)$.

$\text{right}(\text{left}(p)) \leftarrow p$.

$\text{left}(\text{right}(p)) \leftarrow p$. ■

删除算法



- $s \rightarrow \text{left} \rightarrow \text{right} = s \rightarrow \text{right};$
- $s \rightarrow \text{right} \rightarrow \text{left} = s \rightarrow \text{left};$
- $\text{free}(s);$



删除算法的ADL描述

算法 **DLDelete** ($head, p . head$) // 删除结点 p , 有哨位

DLD1. [特判: 若结点 p 为空或尾结点]

IF $p = \Lambda$ THEN RETURN;

IF $\text{right}(p) = \Lambda$ THEN ($\text{right}(\text{left}(p)) \leftarrow \Lambda$. **AVAIL** $\leftarrow p$.
RETURN.)

DLD2. [删除]

$\text{right}(\text{left}(p)) \leftarrow \text{right}(p)$.

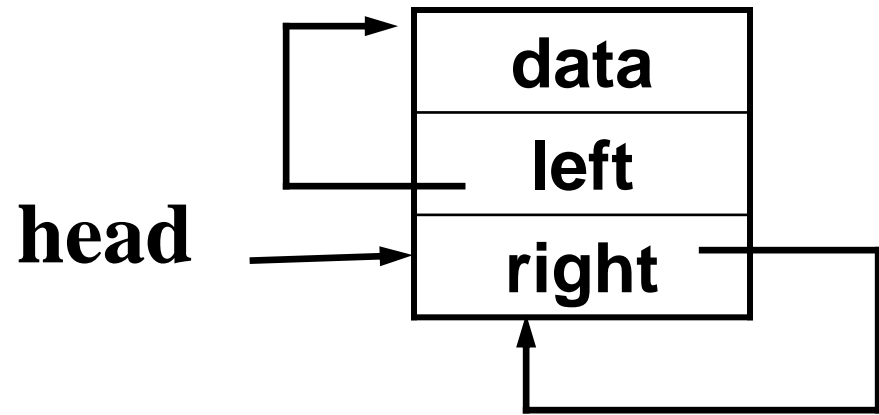
$\text{left}(\text{right}(p)) \leftarrow \text{left}(p)$.

AVAIL $\leftarrow p$. ■



跳舞链

- 带哨兵变量的双向循环链表：跳舞链
- 表头： **head->right**
- 表尾： **head->left**



- 空表： **head->left == head**
或者 **head->right == head**



插入

```
template<class T>
void DLinkedList<T>::ins(int k,T x) { //k个结点后插入x
    DLNode<T>* p,*q;
    p=find(k);
    if(p==NULL) return;
    q=new DLNode<T>;
    q->data=x;
    q->left=p,q->right=p->right;
    q->left->right=q,q->right->left=q;
}
```



删除

```
template<class T>
void DLinkedList<T>::del(int k,T& x) {//删除 第k个结点
    DLNode<T>* p;
    if(k<=0) return;
    p=find(k); if(p==NULL) return;
    p->right->left=p->left;
    p->left->right=p->right;
    x=p->data;
    delete p;
}
```




静态链表

- 前面学习的各种链表都是由指针实现的；链表中结点的分配和回收都是由系统提供的标准函数动态实现的，故称之为动态链表；
- 但是，有的高级语言，如**BASIC**、**FORTRAN**等，没有提供”指针”这种数据类型；链表的插入和删除效率较高，但实现不如顺序表简单；
- 用数组实现的链表，称为静态链表。



静态链表原理

- ❑ 每个结点应含有两个域：**data**域和**next**域；**data**域用来存放结点的数据信息，**next**域指示其后继结点在数组中的相对位置(即数组下标)。
- ❑ 定义一个较大的结构数组作为待分配结点空间(即存储池)。
- ❑ 结点的分配和回收都针对于存储池。简单的，分配时使用未用的数组元素；回收时不处理或置删除标记；



静态单链表： 结构体数组

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

```
Node link[MAXN]; // link[0]是哨兵结点  
int tot=0; //用于结点分配
```



头插法创建链表

```
void insHead(int x)  
{  
    tot++;  
    link[tot].data=x;  
    link[tot].next=link[0].next;  
    link[0].next=tot;  
}
```



链表的遍历

```
void print()  
{  
    for( int p=link[0].next ; p ; p=link[p].next )  
        printf("%d\n",link[p].data);  
}
```



删除头结点

//不回收，空间有浪费

void delHead()

{

int p=link[0].next;

if(p) link[0].next=link[p].next;

}



验证

```
int main()
{
    int n,i;
    scanf("%d",&n);
    for(i=1;i<=n;i++)
        insHead(i);
    print();
    delHead();
    print();
}
```



静态单链表：平行成员数组

```
int data[MAXN];
```

```
//下标相同成员属于同一结点
```

```
int next[MAXN];
```

```
//下标0是哨兵变量， next[0]是头指针
```

```
int tot=0; //用于结点分配
```




//头插法创建链表

void insHead(int x)

{

tot++;

data[tot]=x;

next[tot]=next[0];

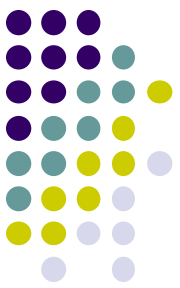
next[0]=tot;

}



```
void print()  
{  
    for(int p=next[0];p;p=next[p])  
        printf("%d\n",data[p]);  
}
```

```
void delHead()  
{  
    int p=next[0];  
    if(p) next[0]=next[p];  
}
```



例：移动小球

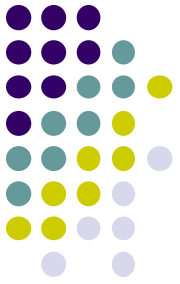
- 描述： n 个小球，从左到右编号依次为1,2,3,4,5,6..... n ，并规定小球1的左边的球号为 n ，小球 n 的右边的球号为1.
- 现在有有以下3种操作：
 - ✓ A $x\ y$:把编号为 x 小球移动到编号为 y 的小球的左边，
 - ✓ B $x\ y$:把编号为 x 小球移动到编号为 y 的小球的右边，
 - ✓ Q 1 m 为询问编号为 m 的小球右边的球号，
Q 0 m 为询问编号为 m 的小球左边的球号。



□ 输入

- ✓ 第一行有一个整数 n ($0 < n < 10000$),表示有 n 组测试数据,
- ✓ 随后每一组测试数据第一行是两个整数 N, M , 其中 N 表示球的个数($1 < N < 10000$), M 表示操作的次数($0 < M < 10000$); 随后的 M 行, 每行有三个数 $s \ x \ y$, s 表示操作的类型, x, y 为小球号。当 s 为 Q 时, 若 x 为 1 , 则询问小球 y 右边的球号, x 为 0 , 则询问小球 y 左边的球号。

□ 输出： 每次询问的球号



□ 样例输入

✓ 1

✓ 6 3

✓ A 1 4

✓ B 3 5

✓ Q 1 5

□ 样例输出

✓ 3



静态链表版跳舞链

- **int D[MAXN];**
- **int L[MAXN];**
- **int R[MAXN];**



创建

```
inline void create()
{
    for( int i = 1; i < n; i++ ){
        D[i] = i;
        L[i] = i - 1;
        R[i] = i + 1;
    }
    L[n] = n-1 ; R[n] = 0 ;
    L[0]= n ; R[0] = 1 ;
}
```



跳舞链用于表上高效插入和删除

- 元素固定、只调整链接
- 查找（第 k 个）： $O(1)$
- 插入（链上）： $O(1)$
- 删除（摘下）： $O(1)$



总结

- 线性表的定义、特性
- 线性表的基本操作
- 顺序表
- 单链表
- 双向链表
- 跳舞链
- 静态链表