

# 第二章 线性表

---

张史梁

slzhang.jdl@pku.edu.cn

# 线性表

---

## □ 目的：

- 介绍线性表的逻辑结构和各种存储表示方法，以及定义在逻辑结构上的各种基本运算及其在存储结构上如何实现这些基本运算。
- 能够针对具体应用问题，选择合适的存储结构，设计相应算法，解决与线性表有关的实际问题。

## □ 重点：

- 熟练掌握顺序表和链表上实现的各种基本算法及相关的时间性能分析

# 内容提要

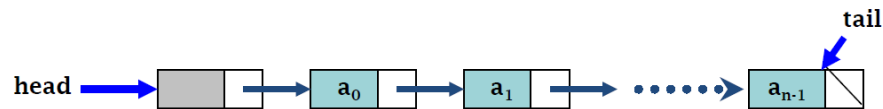
## □ 线性表（逻辑结构）

$$\{a_0, a_1, \dots, a_{n-1}\}$$

## □ 顺序表示和实现（顺序表）



## □ 链式表示和实现（链表）

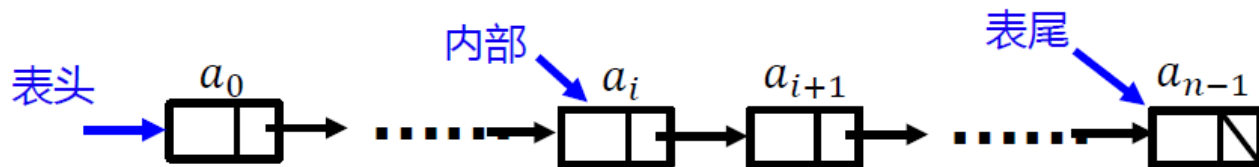


## □ 顺序表和链表的比较

# 线性结构

□ 二元组  $B = \{K, R\}$ ,  $K = \{a_0, a_1, \dots, a_{n-1}\}$   $R = \{r\}$

- 有一个唯一的**开始节点**，它没有前驱，有一个唯一的直接后继
- 有一个唯一的**终止节点**，它有一个唯一的直接前驱，而没有后继
- 其它的结点皆称为**内部结点**，每个内部结点有且仅有一个唯一的直接前驱，有一个唯一的直接后继
- $r = \langle a_i, a_{i+1} \rangle$  是  $a_{i+1}$  的前驱， $a_{i+1}$  是  $a_i$  的后继
- 前驱/后继关系  $r$ ，具有**反对称性**和**传递性**



# 线性结构

---

- 均匀性：虽然不同线性表的数据元素可以是各种各样的，但对于**同一线性表**的各数据元素必定具有**相同的数据类型和长度**
- 有序性：各数据元素在线性表中都有自己的位置，且数据元素之间的**相对位置是线性的**

# 线性表

---

## □ 三个方面

- 线性表的逻辑结构
- 线性表的存储结构
- 线性表的运算

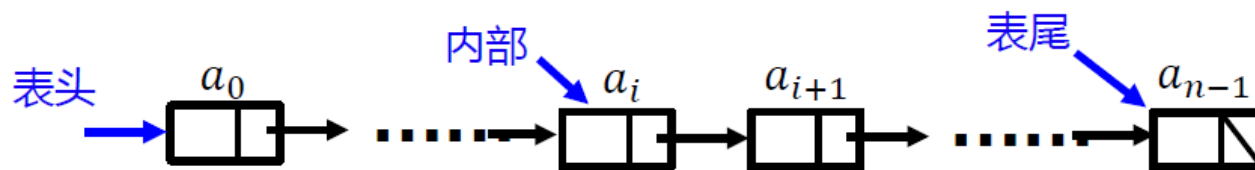
# 线性表的逻辑结构

---

□ 二元组  $B = \{K, R\}$ ,  $K = \{a_0, a_1, \dots, a_{n-1}\}$ ,  $R = \{r\}$

□ 线性表的属性

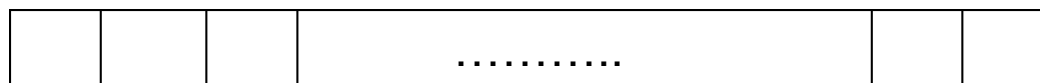
- 线性表的长度
- 表头 (head)
- 表尾 (tail)
- 当前位置 (current position)



# 线性表的存储结构

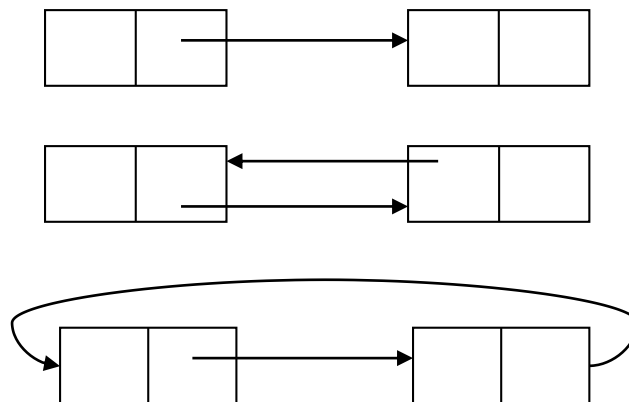
## □ 顺序表

- 按索引值从小到大存放在一片相邻的连续区域
- 紧凑结构，存储密度为1



## □ 链表

- 单链表
- 双链表
- 循环链表
- 存储密度 $<1$



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)



# 线性表的运算

---

- 建立线性表
- 清除线性表
- 插入一个新元素
- 删除某个元素
- 修改某个元素
- 排序
- 检索

# 线性表的抽象数据类型

---

```
□ ADT List {  
    [DATA]  
    [OPERATIONS]  
    PList createNullList( void ) ; //创建空线性表  
    int insert(PList palist, int p, DataType x); //插入一个元素  
    int delete( PList palist, int p ); //删除某个元素  
    int locate( PList palist, DataType x ) ; //查找某个特定元素  
    DataType retrieve( PList palist, int p ); //得到某个下标的元素值  
    int next( PList palist, int p); //查找某个元素的后继元素  
    int previous( PList palist, int p); //查找某个元素的前驱元素  
    int isNullList( PList palist); //判别一个线性表是否为空表  
}
```

# 内容提要

---

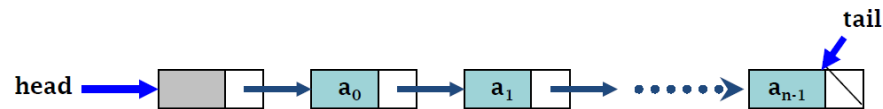
□ 线性表（逻辑结构）

$$\{a_0, a_1, \dots, a_{n-1}\}$$

□ 顺序表示和实现（顺序表）



□ 链式表示和实现（链表）



□ 顺序表和链表的比较

# 顺序表

---

- 也称向量，采用定长的一维数组存储结构
- 主要特性
  - 元素的类型相同
  - 元素顺序地存储在连续存储空间中（数组存储），每一个元素有唯一的索引值
  - 使用常数作为向量长度
- 读写其元素很方便，通过下标即可指定位置
  - 只要确定了首地址，线性表中任意数据元素都可以随机存取
  - 因此，顺序表为“随机存取的存储结构”。

# 顺序表

## □ 元素地址计算：

$$Loc(k_i) = Loc(k_0) + c \times i, \quad c = \text{sizeof}(ELEM)$$

逻辑地址  
(下标)

0	$k_0$
1	$k_1$
...	...
$i$	$k_i$
...	
$n-1$	$k_{n-1}$

存储地址 数据元素

$Loc(k_0)$	$k_0$
$Loc(k_0)+c$	$k_1$
...	...
$Loc(k_0)+i*c$	$k_i$
...	
$Loc(k_0)+(n-1)*c$	$k_{n-1}$

# 顺序表（类型）的定义

---

- 在C语言中可以用以下方式定义一个顺序表：

```
#define    MAXNUM    100 /*最多允许的数据元素个数*/
```

```
DataType  element[MAXNUM];
```

```
int        n;    /*顺序表中当前元素的个数*/
```

- **定义的缺陷：**没有反映出element和n的内在联系。在这个定义中，n和element完全独立，所以程序中完全可以将它们作为独立的自由变量来使用。
- 为此，引进SeqList结构体

# 顺序表（类型）的定义

```
#define MAXNUM 100
struct SeqList{

    DataType element[MAXNUM];
    int n;          /* n < MAXNUM */
};
```

在实际应用中，为了方便，定义一个struct SeqList类型的指针：  
typedef struct SeqList \* PSeqList;

例如：PSeqList palist;

palist->n :顺序表中数据元素个数

palist->element[0], palist->element[1],...

剩余空间：n到MAXNUM-1

# 顺序表的基本运算-1

- 创建
- 插入
- 删除
- 查找
- 取值

□ PSeqList createNullList\_seq( void )

创建空顺序表

□ int insert\_seq( PSeqList palist, int p, DataType x, )

在palist所指顺序表中下标为  $p$  的位置上插入一个值为  $x$  的元素，并返回插入成功与否的标志。

此运算在  $0 \leq p \leq \text{palist} \rightarrow n$  时有意义

□ int delete\_seq(PSeqList palist, int p )

在palist所指顺序表中删除下标为  $p$  的元素，并返回删除成功与否的标志。此运算在  $0 \leq p < \text{palist} \rightarrow n$  时有意义

插入:  $(k_0, k_1, \dots, k_p, k_{p+1}, \dots, k_{n-1})$   $n$   
 $(k_0, k_1, \dots, k_{p-1}, x, k_p, \dots, k_{n-1})$   $n+1$

删除:  $(k_0, k_1, \dots, k_{p-1}, k_p, k_{p+1}, \dots, k_{n-1})$   $n$   
 $(k_0, k_1, \dots, k_{p-1}, k_{p+1}, \dots, k_{n-1})$   $n-1$



# 顺序表的基本运算-2

---

□ `int locate_seq( PSeqList palist, DataType x )`

在palist所指顺序表中寻找值为  $x$  的元素的下标，若palist中不存在值为  $x$  的元素，则返回一个特殊的下标值-1

□ `DataType retrieve_seq( PSeqList palist, int p )`

在palist所指顺序表中，寻找下标为 $p$  ( $0 \leq p < \text{palist} \rightarrow n$ ) 的元素，并将该元素的值作为函数值返回;否则返回一个特殊的值

# 顺序表的基本运算-3

---

- ❑ `int next_seq( PSeqList palist, int p )`  
求palist所指顺序表中下标为 p 的元素的后继元素下标，  
当不存在下标为 p 的元素或虽有该元素但无后继时，返回一个特殊的下标值 -1
- ❑ `int previous_seq( PSeqList palist, int p )`  
求palist所指顺序表中下标为 p 的元素的前驱元素下标，  
当不存在下标为 p 的元素，或虽有该元素但无前驱时，  
本函数返回一个特殊的下标值 -1
- ❑ `int isNullList_seq( PSeqList palist )`  
判别palist所指顺序表是否为空表。若为空则返回1，否则返回 0

# 创建空顺序表

- ❑ 函数名： createNullList\_seq
- ❑ 功能：建立空线性表
- ❑ 程序：

```
PSeqList createNullList_seq (void )
```

```
{
```

```
    PSeqList palist;
```

```
    palist =(PSeqList ) malloc (sizeof(struct SeqList ) );
```

```
    if (palist!= null) palist -> n = 0 ;
```

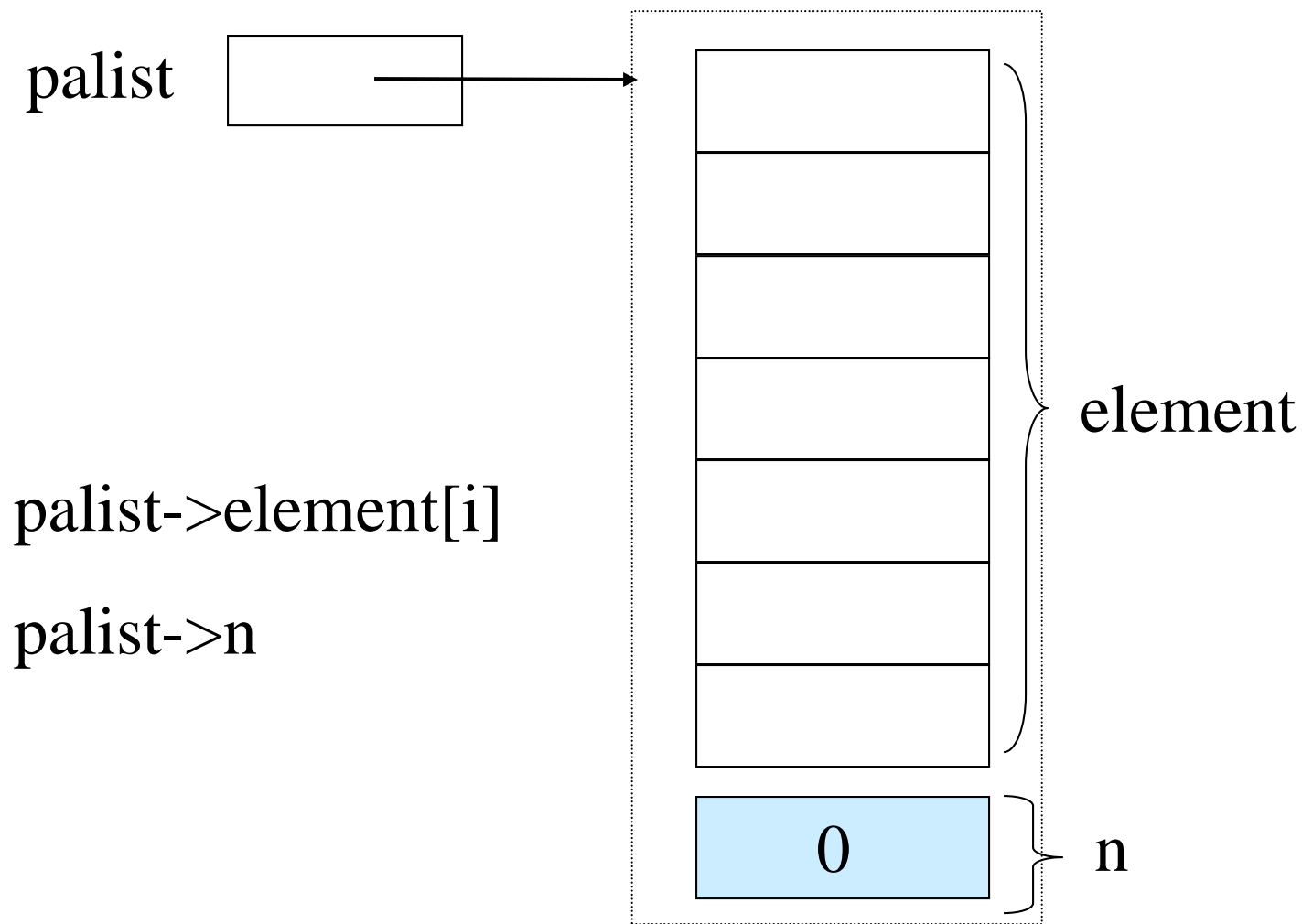
```
    else printf(“out of space!\n”);
```

```
    return (palist);
```

```
}
```

```
struct SeqList  
{  
    DataType element[MAXNUM];  
    int    n;    /* 0=< n < MAXNUM */  
}  
typedef struct SeqList *PSeqList
```

# 创建空顺序表图示



# 创建空顺序表

```
① // #define MAXNUM 100 /*最多允许的元素个数*/  
② struct SeqList {  
③     DataType *element; // DataType element[MAXNUM];  
④     int n;  
⑤ };  
⑥ typedef struct SeqList *PSeqList; //指向struct SeqList类型的指针类型
```

```
① PSeqList createNullList_seq (int m)  
② {  
③     PSeqList palist;  
④     palist =(PSeqList ) malloc (sizeof (struct SeqList ) );  
⑤     if (palist!= null)  
⑥     {  
⑦         palist -> element = (DataType *) malloc (sizeof (DataType ) *m);  
⑧         if (palist -> element != NULL)  
⑨         {  
⑩             palist -> n = 0 ; return ( palist );  
⑪         }  
⑫         else free( palist );  
⑬     }  
⑭     printf(“out of space!\n”);  
⑮     return (NULL);  
⑯ }
```

# 顺序表上的运算

---

## □ 重点讨论：

- 插入元素运算
- 删除元素运算
- 取值运算
- 定位运算

## □ 其他运算请大家思考.....

# 顺序表中的插入

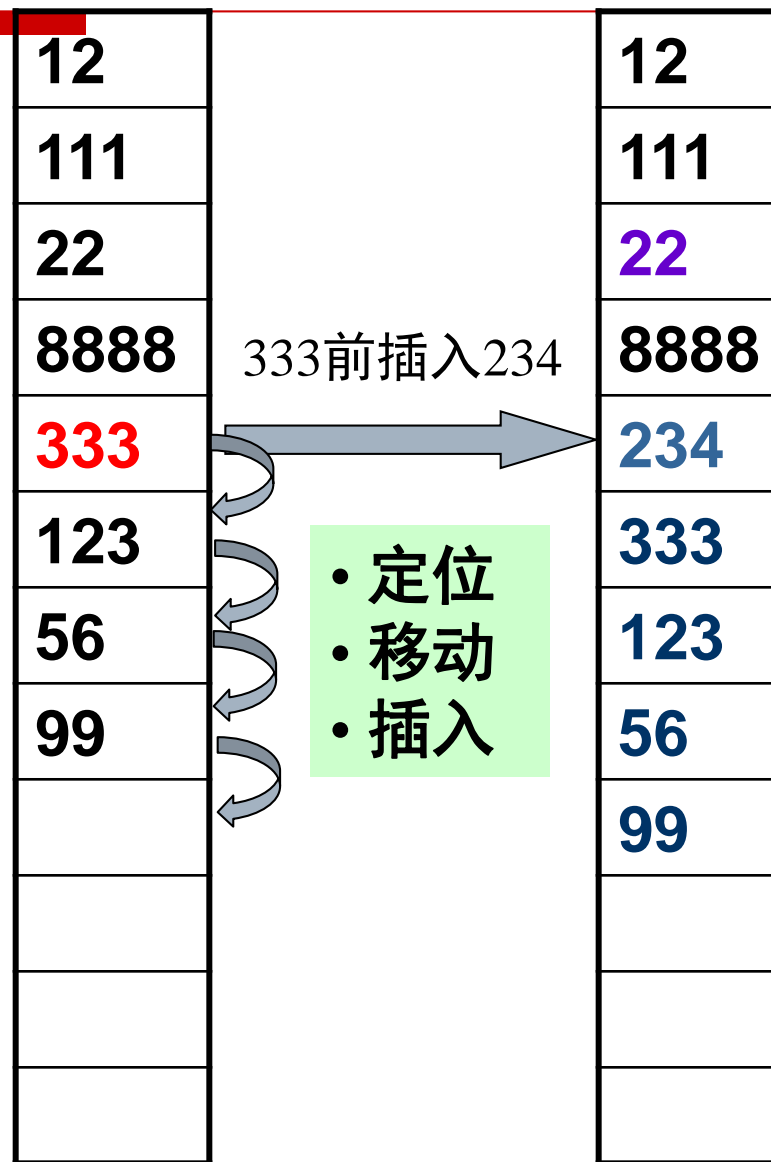
- ❑ 在palist所指顺序表中下标为p的位置上插入一个值为x的元素，并返回插入成功与否的标志。

- ❑ 程序：

```
int insert_seq( PSeqList palist,  
               int p, DataType x)
```

- ❑ 例如

```
succ=insert_seq( palist,4, 234);
```



# 顺序表中插入的实现

---

```
① int insert_seq( PSeqList palist, int p, DataType x )
② { int q;
③  /* if ( palist->n == MAXNUM )      { printf("Overflow!\n"); return (0); }  */
④  if ( p<0 || p > palist->n ) /*不存在下标为 p 的元素 */
⑤    { printf("not exist! \n"); return (0); }

⑥  /* 将 p 及以后的元素后移一个位置 */
⑦  for(q=palist->n - 1; q>=p; q--)
⑧      palist->element[q+1] = palist->element[q];

⑨  palist->element[p] = x;    /* 在p下标位置上放元素x */
⑩  palist->n = palist->n + 1; /* 元素个数加 1 */
⑪  return (1);
⑫ }
```



# 插入算法的时间复杂性

---

- 设 $p_i$ 是在第 $i$ 个元素之前插入一个元素的概率
- 则在长度为 $n$ 的线性结构中在下标为 $i$ 的元素之前插入一个元素需移动元素的次数为 $(n-i)$
- 则插入时的平均移动次数为：

$$E_{is} = \sum_{i=0}^n p_i (n-i)$$

设  $p_i = \frac{1}{n+1}$  则有

$$E_{is} = \frac{1}{n+1} \sum_{i=0}^n (n-i) = \frac{n}{2}$$

时间代价为 $O(n)$ 。

# 顺序表中的删除

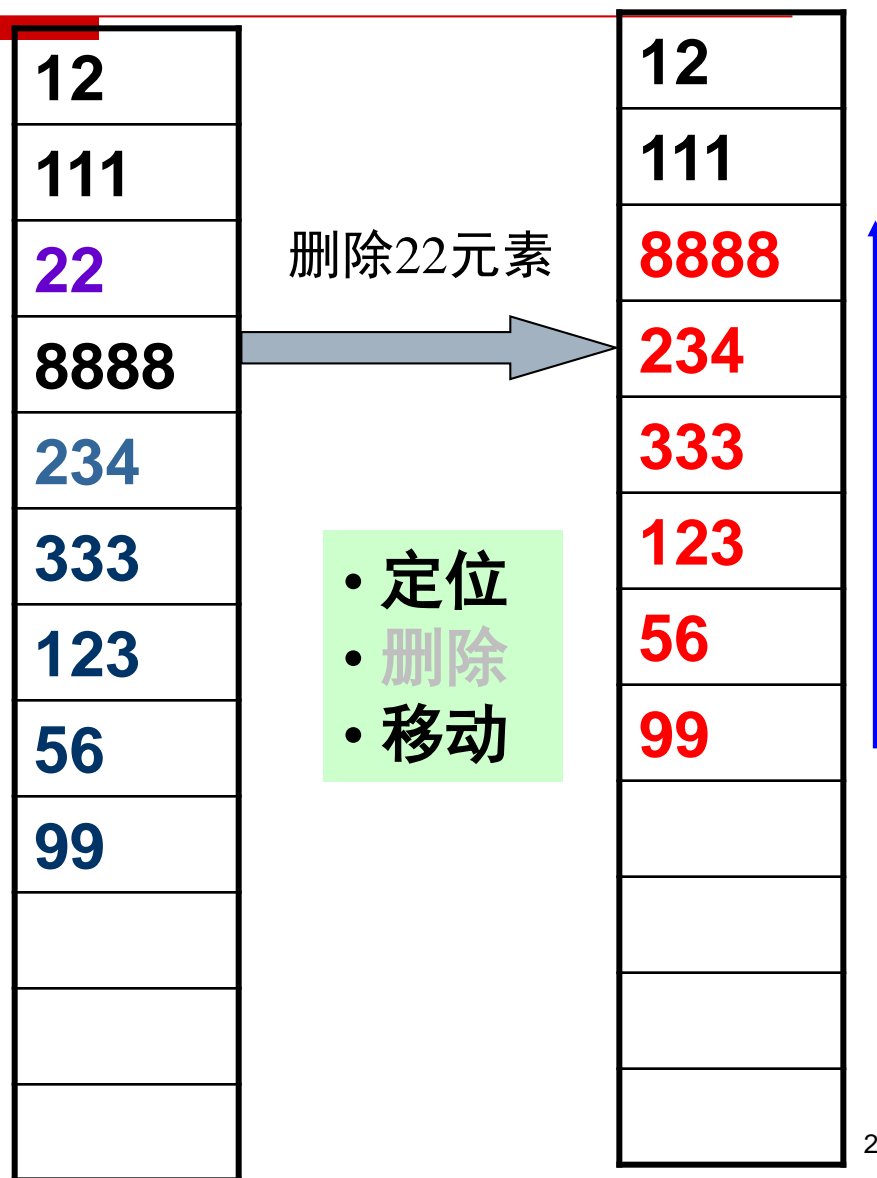
- 在palist所指顺序表中删除下标为  $p$  的元素，并返回删除成功与否的标志。

- 程序：

```
int delete_seq( PSeqList  
                palist, int p )
```

- 举例

```
succ = delete_seq(palist, 2);
```



# 顺序表中删除的实现

---

```
① int delete_seq( PSeqList palist, int p )
② {
③     int q;
④     if (p<0 || p>palist->n-1) //不存在下标为 p 的元素
⑤         { printf("not exist!\n "); return(0); }

⑥     /*将 p 以后的元素前移一个位置 */
⑦     for(q=p; q<palist->n-1; q++)
⑧         palist->element[q] = palist->element[q+1];

⑨     palist->n = palist->n - 1;    /*元素个数减 1 */
⑩     return (1);
⑪ };
```

# 删除算法的时间复杂性

---

- 设 $P_i'$ 是在下标为 $i$ 的位置上删除元素的概率
- 则在长度为 $n$ 的线性结构中把下标为 $i$ （第 $i+1$ 个）位置上的元素删除需要移动 $n-i-1$ 个元素：
- 则删除时的平均移动次数为：

$$M_d = \sum_{i=0}^{n-1} (n-i-1)P_i'$$

设  $P_i' = \frac{1}{n}$  则有

$$M_d = \frac{1}{n} \sum_{i=0}^{n-1} (n-i-1) = \frac{n-1}{2}$$

时间代价为 $O(n)$ 。

# 顺序表取值运算的实现

---

- 功能：在palist所指顺序表中，寻找下标为p的元素，并将该元素的值作为函数值返回，若找不到则返回一个特殊的值。

```
DataType retrieve_seq( PSeqList palist,int p )
{
    if(p>=0 && p<palist->n)
        return (palist->element[p]);
    printf("not exist.\n")
    return(SPECIAL);
}
```

# 顺序表定位运算的实现

---

□ 功能 :求x在palist所指顺序表中的下标

```
int locate_seq (PSeqList palist, Datatype x )
{
    int q;
    for (q=0; q<palist->n; q++)
        if (palist->element[q]==x)
            return (q);
    return(-1);
}
```

# 定位运算的时间复杂性

---

- **定位运算**：通过比较完成，完成一次定位平均需要 $n/2$ 次比较，时间复杂度为 $O(n)$ ；
- 对于有序表的改进：  
采用折半查找方法， $O(\log_2 n)$

# 思考

---

- 顺序表中，插入删除操作需要考虑哪些问题？
- 顺序表有哪些优缺点？



# 补充说明

---

- 为了上述程序执行，需将下面的类型说明、常量说明及类型定义包含在文件中。

```
#define MAXNUM 100 /* 线性表空间大小 */
#define FALSE 0
#define TRUE 1
typedef int DataType;
struct SeqList {
    DataType element[MAXNUM];
    int n;
};
typedef struct SeqList *PSeqList;
PSeqList createNullList( void ) ; //创建空线性表
int insert(PList palist,int p,DataType x); //插入一个元素
int delete( PList palist, int p ); //删除某个元素
int locate( PList palist, DataType x ) ; //查找某个特定元素
```

◦ ◦ ◦ ◦ ◦

# 线性表的应用——顺序表部分

---

## □ 问题

- Josephus（约瑟夫）问题



- 纯集合构造

- 有序表的归并

- 求集合的并集

# Josephus问题

---

弗拉维奥·约瑟夫斯（Josephus）  
是一世纪的一名犹太历史学家。



# 约瑟夫问题的由来

---

- Josephus在自己的日记中写道：在罗马人占领乔塔帕特后，39 个犹太人与Josephus及他的朋友躲到一个洞中
- 39个犹太人决定宁愿死也不要被敌人抓到，于是决定了一个自杀方式



# 约瑟夫问题的由来

---

## □ 自杀方式的约定：

- 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。
- Josephus要他的朋友先假装遵从，他将朋友与自己安排在第 个与第 个位置，于是逃过了这场死亡游戏。

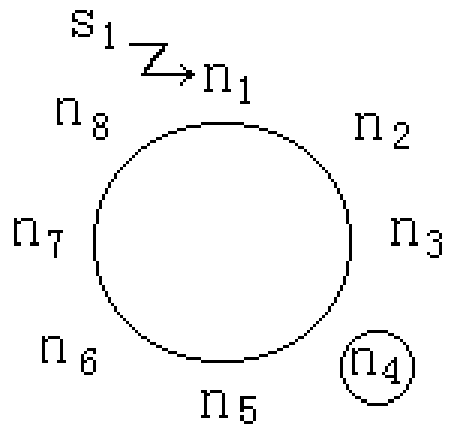
## □ 后传：

- 约瑟夫斯说服了他的朋友，他们将向罗马军队投降，不再自杀。

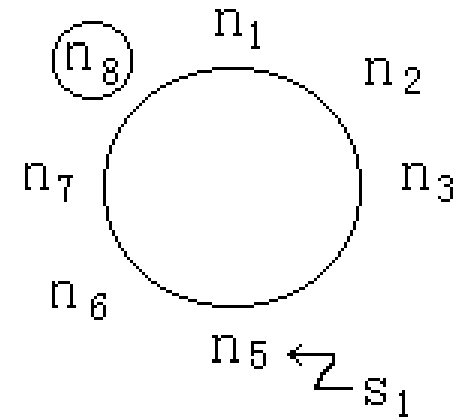
# 约瑟夫问题的描述

---

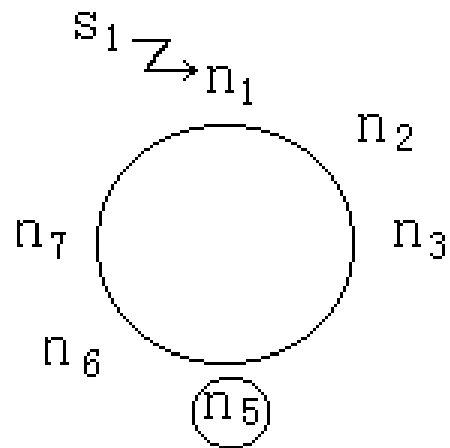
- $n$ 个人围坐在一圆桌周围，现从第 $s$ 个人开始报数，数到第 $m$ 的人出列，然后从出列的下一个入重新开始报数，数到第 $m$ 的人又出列，如此反复直到所有的人全部出列为止
- **问题描述：**对于任意给定的 $n$ ,  $s$ 和 $m$ ，求按出列次序得到的人员序列。
  - $n$  : 参与游戏的人数，每个人的信息
  - $s$  : 开始的人
  - $m$  : 单次计数



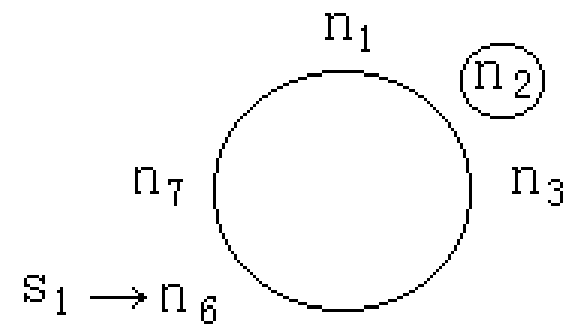
**(a) n4**



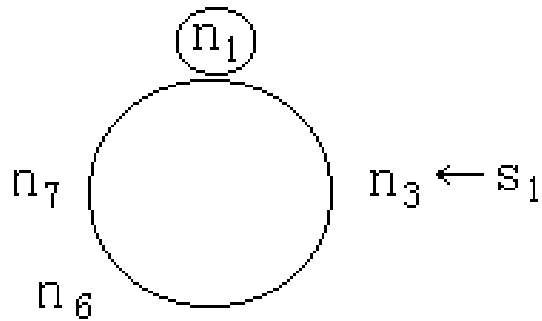
**(b) n4 n8**



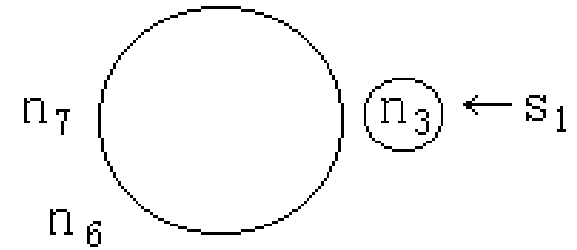
**(c) n4 n8 n5**



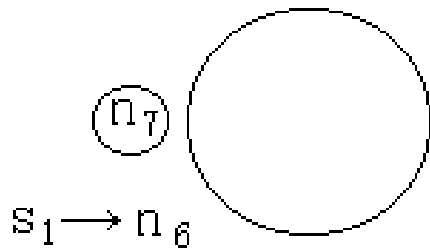
**(d) n4 n8 n5 n2**



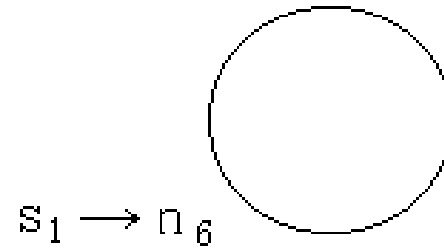
**(e)  $n_4$   $n_8$   $n_5$   $n_2$   $n_1$**



**(f)  $n_4$   $n_8$   $n_5$   $n_2$   $n_1$   $n_3$**



**(g)  $n_4$   $n_8$   $n_5$   $n_2$   $n_1$   $n_3$   $n_7$**



**(h)  $n_4$   $n_8$   $n_5$   $n_2$   $n_1$   $n_3$   $n_7$   $n_6$**



# 顺序表方式实现

---

## □ 步骤：

- 1: 建立顺序表；
- 2: 出列算法；
- 3: 主程序

# 约瑟夫问题解答算法-1

```
① #include <stdio.h>
② #include <stdlib.h>
③ #define MAXNUM 100
④ typedef int DataType;
```

```
⑤ //顺序表的定义
⑥ struct SeqList
⑦ {
⑧     DataType element[MAXNUM];
⑨     int n;
⑩ };
⑪ typedef struct SeqList *PSeqList;
```

```
⑫ PSeqList createNullList_seq( ) {.....}
⑬ int insert_seq(PSeqList palist, DataType x, int p) {.....}
⑭ int delete_seq(PSeqList palist, int p) {.....}
⑮ int locate_seq(PSeqList palist, DataType x) {.....}
⑯ .....
```

# 约瑟夫问题解答算法-2

```
① josephus_seq( PSeqList palist, int s, int m)
② {
③     int del, i, w;
④     del = s;
⑤     for( i = palist->n; i>0; i--) /* 找出列的元素 */
⑥     {
⑦         del = ( del + m - 1 ) % i ;
⑧         if ( del == 0 ) del = i;
⑨         w = retrieve_seq(palist,del); /* 求要出列元素值 */
⑩         printf("Out element %d \n",w); /* 元素出列 */
⑪         delete_seq(palist,del-1); /* 删除出列的元素 */
⑫     }
⑬ };
```

# 约瑟夫问题解答算法-3

```
① int main( )
② {
③     PSeqList jos_alist;    int n, s, m, i ;
④     printf("\n please input the values(<100) of n = ");
⑤     scanf("%d",&n);
⑥     printf(" please input the values of s = ");
⑦     scanf("%d",&s);
⑧     printf(" please input the values of m = ");
⑨     scanf("%d",&m);

⑩     jos_alist=createNullList_seq( ); /* 创建空顺序表 */
⑪     if (jos_alist==NULL) return (FALSE);
⑫     for( i = 0; i < n; i++ ) insert_seq( jos_alist,i+1, i ); /* 线性表赋值 */

⑬     josephus_seq (jos_alist, s, m);
⑭     return (TRUE);
⑮ }
```

# 顺序表方式实现

---

## □ 步骤：

- 1: 建立顺序表；
- 2: 出列算法；
- 3: 主程序

## □ 时间复杂度分析：

- 运行时间主要是出列元素的删除（数组的移动），  
每次最多移动 $i-1$ 个元素，总计个数不超过

$$(n-1)+(n-2)+\dots+1 = n(n-1)/2 \Rightarrow O(n^2)$$

# 线性表的应用

---

## □ 问题

- Josephus问题
- 纯集合构造
- 有序表的归并
- 求集合的并集



# 纯集合构造

---

- 已知一个非纯集合B，试构造一个纯集合A，使A中包含B中各不相同的元素。
- 已知线性表pblast中包含集合B中所有元素，试构造线性表palist使palist中只包含pblast中不同的元素。

2	3	5	6	7	8	11	5	8	9
---	---	---	---	---	---	----	---	---	---

2	3	5	6	7	8	11	9
---	---	---	---	---	---	----	---

# 顺序表实现的操作步骤

---

- 从线性表B中依次取得每个元素
  - $\text{retrieve\_seq}(\text{pblist}, p) \longrightarrow x$
  
- 在线性表A中查找x
  - $\text{locate\_seq}(\text{palist}, x)$
  
- 若不存在，则插入，插入到表最后
  - $\text{insert\_seq}(\text{palist}, \text{palist} \rightarrow n, x)$



pblist

2	3	5	6	7	8	11	5	8	9
---	---	---	---	---	---	----	---	---	---

palist

2	3	5	6	7	8	11	9
---	---	---	---	---	---	----	---

```
void purge(PSeqList palist,PSeqList pblist) {  
    int i=0;  
    DataType x;  
    for ( i=0;i<pblist->n;i++)  
    {   x= retrieve_seq(pblist, i) ;  
        //取pblist中序号为i的元素赋给x  
        if ( locate_seq( palist, x )== -1)  
            insert_seq(palist, palist->n, x);  
        //palist 中不存在和x相同的元素，插入之  
    } //for  
} // purge ,  
算法的时间复杂度为 $O(n^2)$ 
```

# 线性表的应用

---

## □ 问题

- Josephus问题
- 纯集合构造
- 有序表的归并
- 求集合的并集

# 应用举例—有序表的归并

---

- 利用线性表抽象数据类型提供的操作,可以完成其他更复杂的操作.
- 归并两个数据元素按非递减有序排列的线性表A和B,求得线性表C也具有同样的特性
- 上述问题可以演绎为:
  - 创建线性表pclist, 将存在于线性表palist和线性表pblist中的元素按非递减的顺序插入到线性表pclist中去。

# 线性表实现的操作步骤

---

- 从线性表A、B中依次取得当前每个元素
  - $\text{retrieve\_seq}(\text{pablist}, i) \longrightarrow x a_i$
  - $\text{retrieve\_seq}(\text{pblist}, j) \longrightarrow x b_j$
- 若  $x a_i \leq x b_j$ ，则把  $x a_i$  插入到中，否则把  $x b_j$  插入到表中
  - $\text{insert\_seq}(\text{pclist}, \text{pclist} \rightarrow n, x)$
- 当只有一个表中还有没有处理的元素，把剩下的元素插入到C表中。

# 线性表的应用

---

## □ 问题

- Josephus问题
- 纯集合构造
- 有序表的归并
- 求集合的并集

# 应用举例—求集合的并集

## □ 求集合的并集

- 假设有两个集合A和B分别用两个线性表来表示palist和pblast, (线性表中的数据元素即为集合的元素).
- 求一个新集合 $A=A \cup B$

## □ 上述问题可以演绎为:

- 扩大线性表palist, 将存在于线性表pblast中而不存在于线性表palist中的元素插入到线性表palist中去。

2	3	4	6	7	1	11	13
---	---	---	---	---	---	----	----

5	6	7	8	9
---	---	---	---	---

2	3	4	6	7	1	11	13	5	8	9
---	---	---	---	---	---	----	----	---	---	---

# 顺序表实现的操作步骤

---

- 从线性表B中依次取得每个元素
  - $\text{retrieve\_seq}(\text{pblist}, p) \longrightarrow x$
  
- 在线性表A中查找x
  - $\text{locate\_seq}(\text{palist}, x)$
  
- 若不存在，则插入，插入到表最后
  - $\text{insert\_seq}(\text{palist}, \text{palist} \rightarrow n, x)$

```
void unionlist(PSeqList palist,PSeqList pblist)
```

```
{ int i; DataType x;
```

```
  for ( i=0; i<pblist->n; i++)
```

```
  { x= retrieve_seq(pblist, i);
```

```
    //取pblist中序号为i的元素赋给x
```

```
    if ( locate_seq(palist,x)==-1)
```

```
      insert_seq(palist, palist->n, x);
```

```
    //palist 中不存在和x相同的元素，插入之
```

```
  }
```

```
}// unionlist
```

❑ 算法的时间复杂度为 $O(n^2)$

2	3	4	6	7	1	11	13
---	---	---	---	---	---	----	----

5	6	7	8	9
---	---	---	---	---

2	3	4	6	7	1	11	13	5	8	9
---	---	---	---	---	---	----	----	---	---	---



# 内容提要

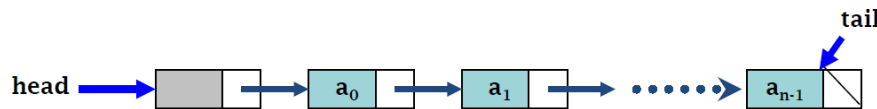
□ 线性表（逻辑结构）

$$\{a_0, a_1, \dots, a_{n-1}\}$$

□ 顺序表示和实现（顺序表）



□ 链式表示和实现（链表）



□ 顺序表和链表的比较

# 顺序表的优缺点

---

## □ 优点：

- 不需要附加空间；**随机存取**任一元素。

## □ 缺点

- 一开始就要分配**足够大**的一片**连续**的内存空间，很难估计所需空间的大小。
- 更新操作代价大，插入与删除运算要**移动大量的元素**，效率较低。

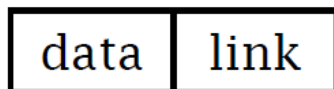
## □ 改进：

- 使用链接存储结构（不要求逻辑相邻物理也相邻，通过指针表达逻辑关系），克服顺序存储的不足，但失去了随机存取的优点。

# 链表

---

- 通过指针把一串存储结点链接成一个链
- 存储结点由两部分组成：
  - 数据域+ 指针域（后继地址）



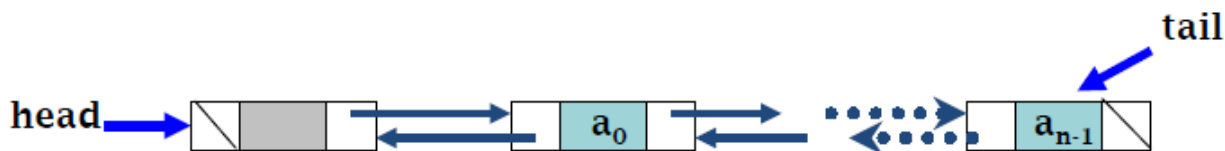
# 链表

## □ 分类（根据链接方式和指针多寡）

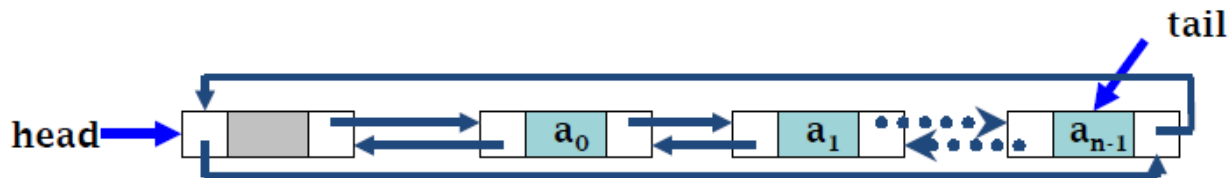
### ■ 单链



### ■ 双链



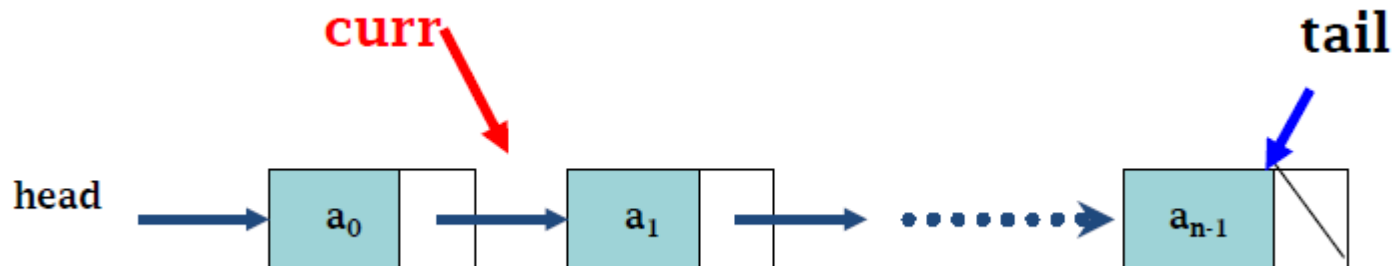
### ■ 循环链



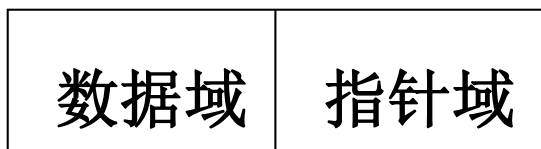
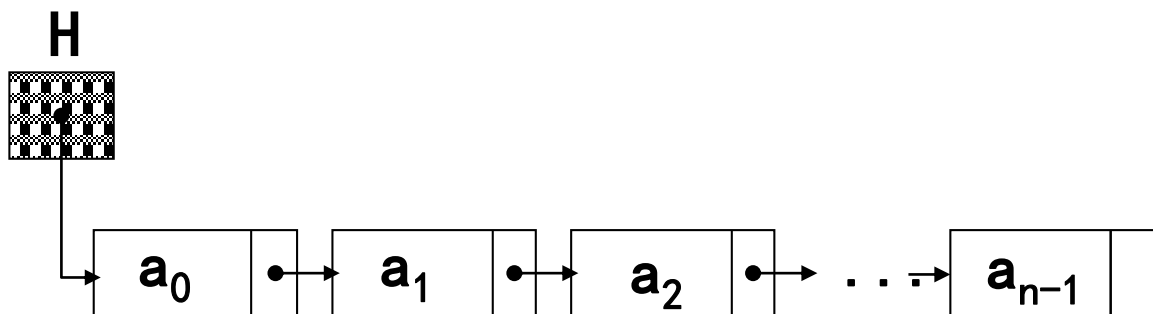
# 单链表

## □ 简单的单链表

- 整个单链表的表示: head
- 第一个结点: head
- 空表判断:  $\text{head} == \text{NULL}$
- 当前结点: curr
- 最后一个节点?



# 举例

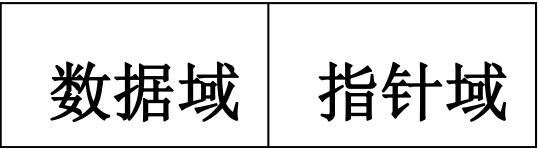


头指针H

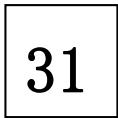
31

存储地址	数据域	指针域
1	Li	43
7	Qian	13
13	Sun	1
19	Wang	NULL
25	Wu	37
31	Zhao	7
37	Zheng	19
43	Zhou	25

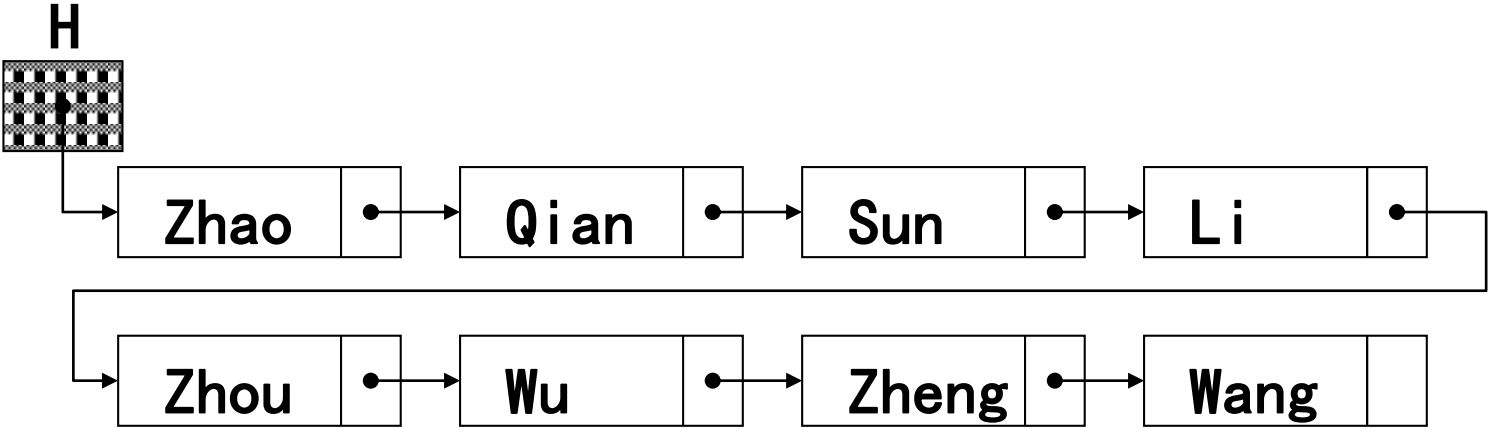
# 举例



头指针H



存储地址	数据域	指针域
1	Li	43
7	Qian	13
13	Sun	1
19	Wang	NULL
25	Wu	37
31	Zhao	7
37	Zheng	19
43	Zhou	25



线性链表的逻辑状态

# 单链表定义

---

- ① `struct Node;`                    `/* 单链表结点类型 */`
- ② `typedef struct Node *PNode;`   `/* 结点指针类型 */`
- ③ `struct Node`                    `/* 单链表结点结构 */`
- ④ `{`
- ⑤     `DataType info;`
- ⑥     `PNode link;`
- ⑦ `};`
- ⑧ `typedef struct Node *PLinkList ;` `/* 单链表指针类型 */`
- ⑨ `PLinkList plist;`            `/* 指向单链表*/`

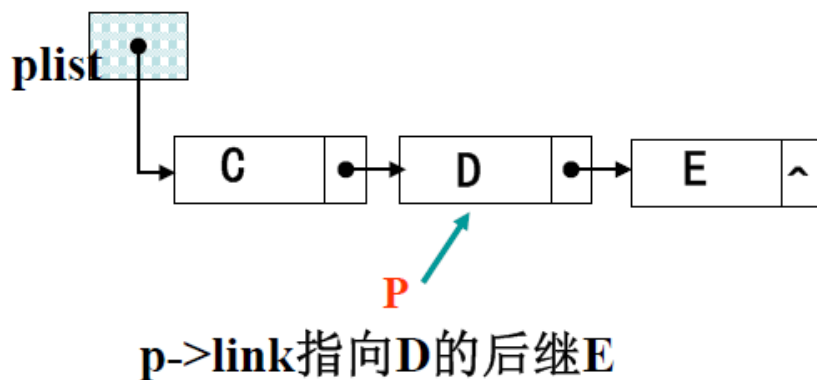
注意：PLinkList和PNode本质上是相同的，都是指向Node结构体的指针，但是逻辑上，我们用PNode指向一个中间节点，PLinkList指向链表（头节点），防止混淆。



# 单链表定义

## □ 简单的单链表

- 整个单链表: `plist`
- 第一个结点: `plist`
- 第二个节点: `plist->link`
- 空表判断: `plist == NULL`



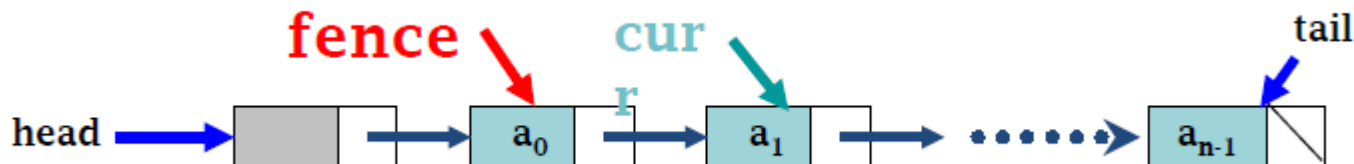
# 单链表

## □ 头节点

- 可存放链表数据（长度），也可不存储任何信息

## □ 带头结点的单链表

- 整个单链表：head
- 第一个结点：head->next, **head ≠ NULL**
- 空表判断：head->next == NULL
- 当前结点a1：fence->next (curr隐含)



# 单链表基本运算及其实现

---

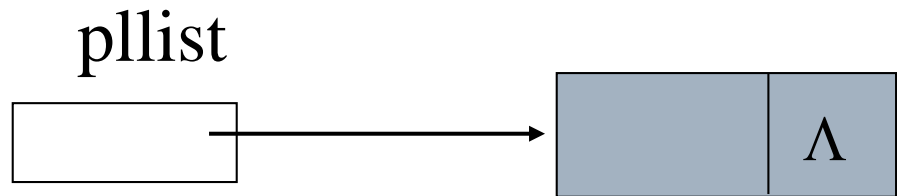
- 创建空的单链表 `PLinkList createNull_link(void)`
- 插入结点 `int insert_link( PLinkList plist, int i, DataType x)`
- 删除结点 `int delete_link(PLinkList plist, DataType x)`
- 定位运算 `PNode locate_link(PlinkList plist, DataType x)`
- 求存储地址运算 `PNode find_link(PLinkList plist, int i)`
- 判空运算 `int isNull_link(PLinkList plist)`

# 建立空链表

- ❑ 函数名：createNulllist\_link
- ❑ 功能：建立带有**头节点**的空链表
- ❑ 程序：

```
PLinkList createNulllist_link (void)
{
    PLinkList pllist;
    pllist =(PLinkList) malloc (sizeof(struct Node) );
    if(pllist != NULL)
        pllist -> link = NULL;
    return (pllist);
}
```

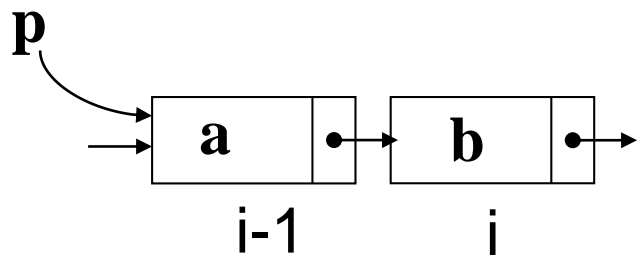
注意： 不再设置顺序表  
中的长度n



# 单链表插入运算

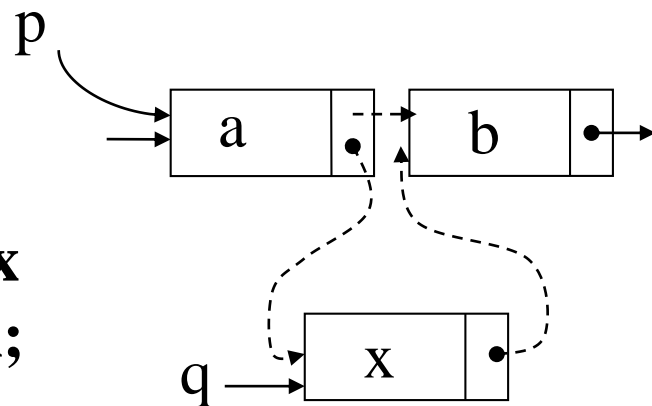
- ❑ 函数名: `insert_link`
- ❑ 功能: 在 `pplist` **带头结点** 的单链表中下标为 `i` (**第** 个) 的结点前插入一个值为 `x` 的元素, 并返回插入成功与否的标志。
- ❑ 程序:

```
int insert_link(PLinkList plist, int i, DataType x)
```



单链表

建立节点 `q`, 赋值 `x`  
`q->link = p->link;`  
`p->link = q;`  
注: 顺序不能反



```

① int insert_link( PLinkList plist, int i, DataType x)
② {
③     PNode p,q;
④     p= plist;
⑤     int j=0;
⑥     while(                ) /*查找i的前驱*/
⑦     { p=p->link; j++; }
⑧     if(j!=i) /*查找失败*/
⑨     { printf("i=%d is not reasonable.\n",i); return 0; }

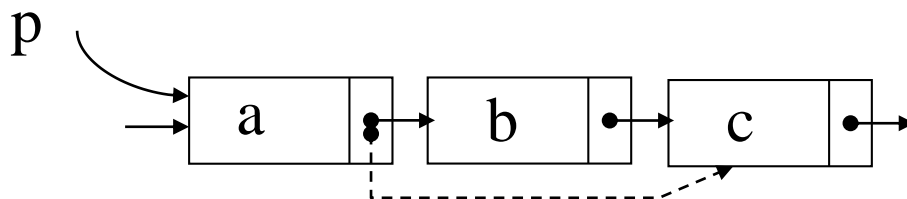
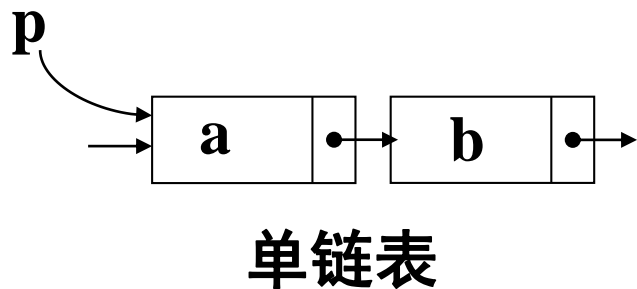
⑩     q = (PNode)malloc( sizeof( struct Node ) ); /*申请新节点*/
⑪     if ( q == null ) /*申请失败*/
⑫     { printf( "Out of space!!!\n" ); return 0; }
⑬     else /*插入链表中*/
⑭     { q->info = x;
⑮         return 1;
⑯     }
⑰ }

```

# 单链表删除运算

- ❑ 函数名: delete\_link
- ❑ 功能: 在pplist带头结点的单链表中删除第一个元素值为 x 的元素, 并返回删除成功与否的标志。
- ❑ 程序:

```
int delete_link( PLinkList pllist, DataType x)
```



**p-> link = p-> link -> link;**

释放空间, 只有malloc, 没有free 造成内存泄漏

```

① int delete_link( PLinkList plist, DataType x )
② /* 在plist带头结点的单链表中删除第一个元素值为 x 的元素*/
③ {
④     PNode p,q;
⑤     p = plist;
⑥     while(                                ) /* 查找x结点的前驱 */
⑦         p = p->link;
⑧     if ( p->link == null ) /* 没找到元素为x的结点 */
⑨     {
⑩         printf("Not exist!\n "); return (0);
⑪     }
⑫     else /* 删除该结点 */
⑬     {
⑭         q = p->link; p->link = q->link;
⑮         return (1);
⑯     }
⑰ }

```



# 算法时间复杂度分析

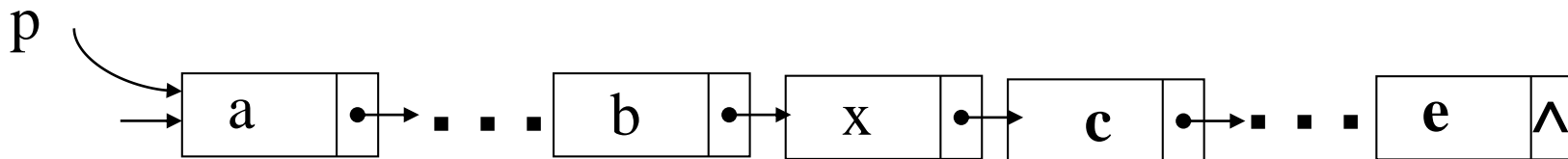
---

- 由于“插入/删除运算”首先需要定位，而定位必须从头结点开始顺链一个一个结点进行比较，因此其“基本操作”为比较运算。
- 比较运算的执行次数与结点的位置有关
  - 最好情况时为1次（第一个结点），最坏为 $n$ 次（所有结点皆比较一次）时间复杂度为 $O(n)$
  - 平均情况下需要查找 $n/2$ 个元素
  - 所以，查找算法的时间复杂度为 $O(n)$
- 插入与删除元素时所花费时间只是申请结点和修改指针的时间，时间复杂度为 $O(1)$

# 单链表定位运算

- ❑ 函数名： `locate_link`
- ❑ 功能：在 `pplist` 带头结点的单链表中求第一个值为 `x` 的节点存储位置。
- ❑ 程序：

**`PNode locate_link(PLinkList plist, DataType x)`**



---

```
PNode locate_link(PlinkList plist, DataType x)
```

```
/* 在plist带头结点的单链表中定位第一个元素值为 x 的元素*/
```

```
{ PNode p; p = plist;
```

```
/*在plist带头结点的单链表中找元素为x的结点的前驱 */
```

```
while( p->link != NULL && p->link->info != x )
```

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

```
/* 返回元素为x的结点的地址*/
```

```
return (p->link);
```

```
}
```

# 其他运算

---

## □ 求存储地址运算

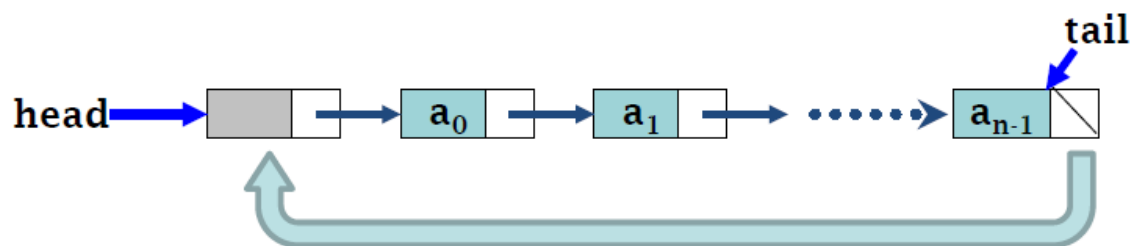
`PNode find_link(PLinkList plist, int i)`

## □ 判空运算

`int isNull_link(PLinkList plist)`

# 单链表存在的问题和改进

- ❑ 单链表的表长是一个隐含的值（后继为空）
  - 可以增加头节点，纪录表长
- ❑ 在单链表最后插入一个元素时，需要遍历整个链表
  - 改为循环链表

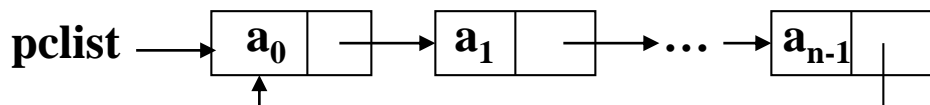


**建立：**将最后一个结点的指针项指向第一个结点（或头结点），构成一个循环链。

**优点：**从任一个结点出发，可访问表中任何一个结点元素。

# 单循环链表

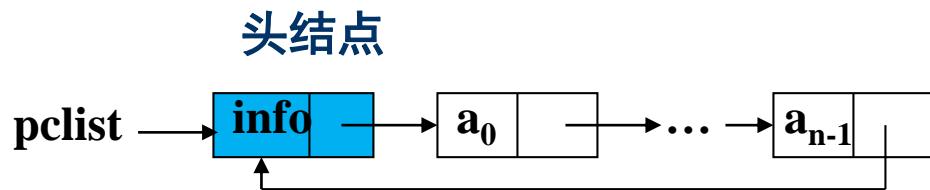
建立：将最后一个结点的指针项指向第一个结点（或头结点），构成一个循环链。



**无头结点：**

循环条件： `p->link == pclist`

表空条件： `pclist == NULL`



**带头结点：**

循环条件： `p->link == pclist`

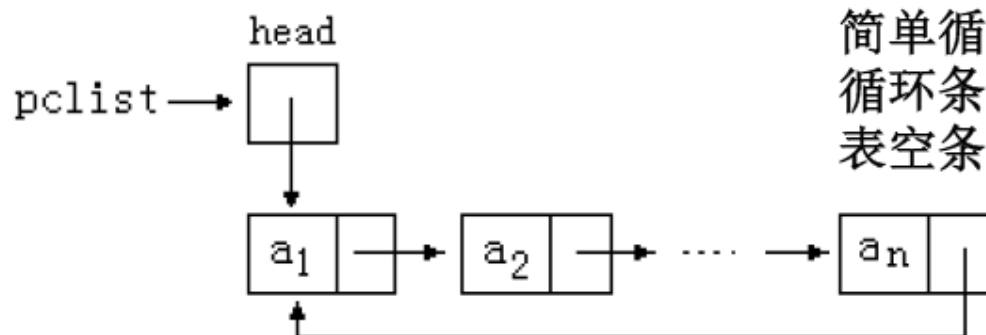
表空条件： `pclist->link == NULL`

# 单循环链表的另一种定义

- ❑ `struct Node; /* 单链表结点类型 */`
- ❑ `typedef struct Node *PNode; /* 结点指针类型 */`
- ❑ `struct Node` `/* 单链表结点结构 */`
  - `{`
  - `DataType info;`
  - `PNode link;`
  - `};`
- ❑ `struct LinkList` `/* 单链表类型定义 */`
  - `{`
  - `PNode head;` `/* 指向单链表中的第一个结点 */`
  - `};`
- ❑ `typedef struct LinkList *PLinkList; /* 单链表类型的指针类型 */`
- ❑ `PLinkList pclist;`

注意：PLinkList和PNode不再相同，更方便我们区分  
我们用PNode指向一个中间节点，PLinkList->head指向链表头。

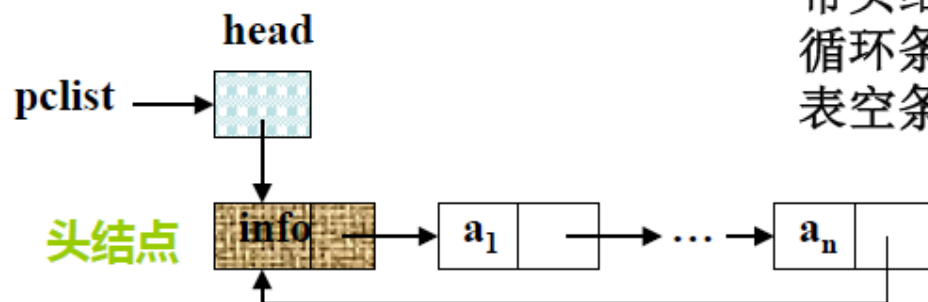
# 单循环链表示意图



简单循环链表:

循环条件:  $p \rightarrow \text{link} == \text{pclist} \rightarrow \text{head}$

表空条件:  $\text{pclist} \rightarrow \text{head} == \text{NULL}$



带头结点的单循环链表:

循环条件:  $p \rightarrow \text{link} == \text{pclist} \rightarrow \text{head}$

表空条件:  $\text{pclist} \rightarrow \text{head} \rightarrow \text{link} == \text{NULL}$



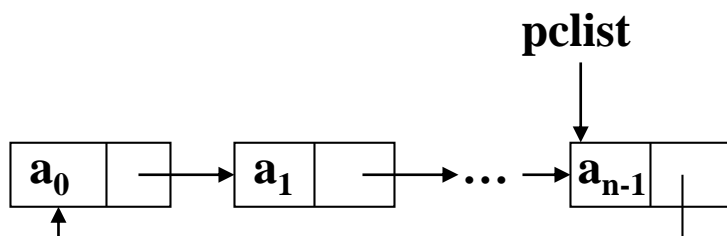
# 改进的单循环链表

上面循环链表的缺点：

如果要访问最后一个结点，仍要访问表中所有的结点。

改进：

修改pclist指向最后一个结点，方便某些操作(如两链表合并)



最后结点： pclist

第一个结点： pclist->link

循环条件： p == pclist

空表判断： pclist == NULL

思考问题1：

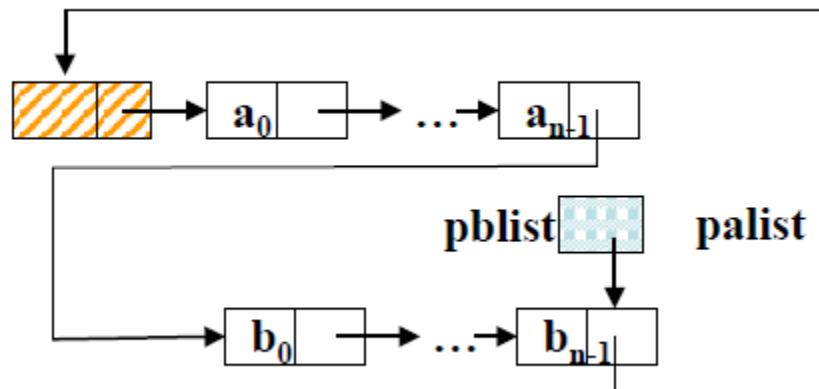
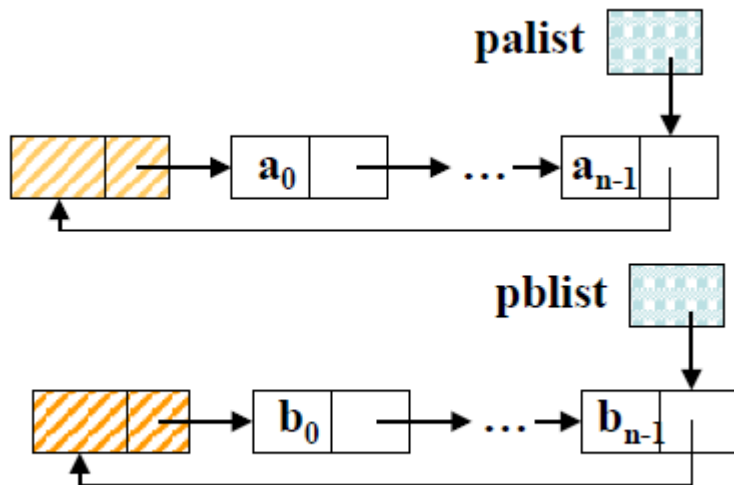
如何将一个单循环链表（无头结点，且pclist指向第一个结点）倒置？

$$(a_0, a_1, \dots, a_{n-1}) \Rightarrow (a_{n-1}, a_{n-2}, \dots, a_0)$$

# 改进的单循环链表

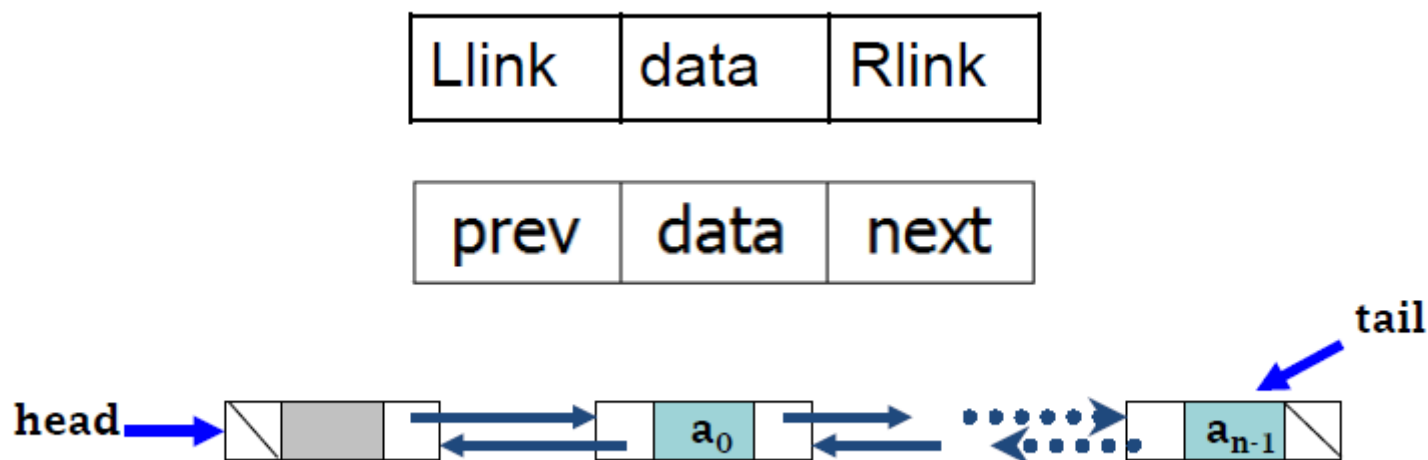
## 思考问题2:

如何将仅设尾指针的两个循环链表合并



# 双向链表

- 为弥补单链表的不足,而产生双向链表
  - 单链表的next 字段仅仅指向后继结点,不能有效地找到前驱,反之亦然
  - 增加一个指向前驱的指针



# 双向链表定义

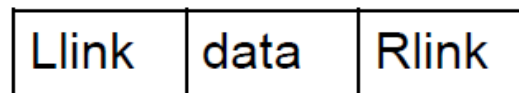
---

//数据说明:

① **struct DoubleNode**    /\* 双链表**结点结构** \*/

② {

③    **DataType**    **info;**



④    **struct DoubleNode \*llink, \*rlink;** //分别等价于**pre, next**

⑤ **};**

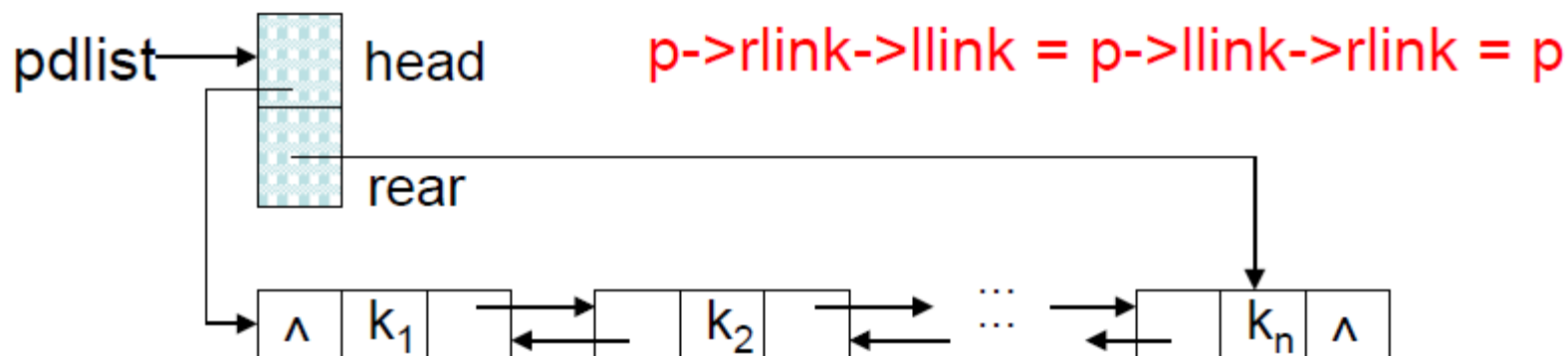
⑥ **typedef struct DoubleNode \*PDoubleNode;** /\* 结点指针类型 \*/

# 双向链表定义

- ① struct DoubleList    /\* 双链表类型 \*/
- ② {
- ③    PDoubleNode head; /\* 指向第一个结点 \*/
- ④    PDoubleNode rear; /\* 指向最后一个结点 \*/
- ⑤ };
- ⑥ typedef struct DoubleList \*PDoubleList;
- ⑦ PDoubleList pdList; /\* pdlist指向双链表 \*/



# 双向链表上的运算



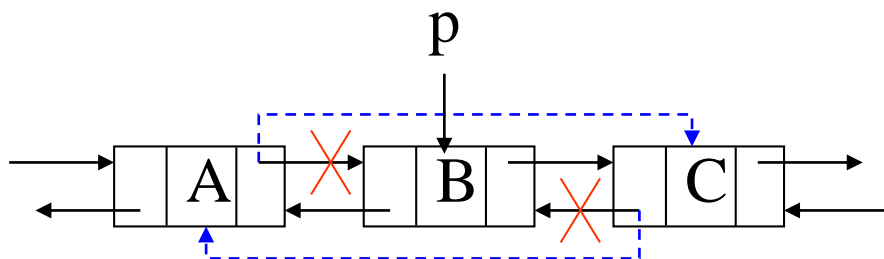
空表判断 :  $pdlist \rightarrow head == NULL$

最后结点判断:  $p \rightarrow rlink == NULL$

第一个结点 :  $pdlist \rightarrow head$

最后结点 :  $pdlist \rightarrow rear$

# 双向链表结点删除图示



删除双向链表的结点

**$p \rightarrow \text{llink} \rightarrow \text{rlink} = p \rightarrow \text{rlink}$**

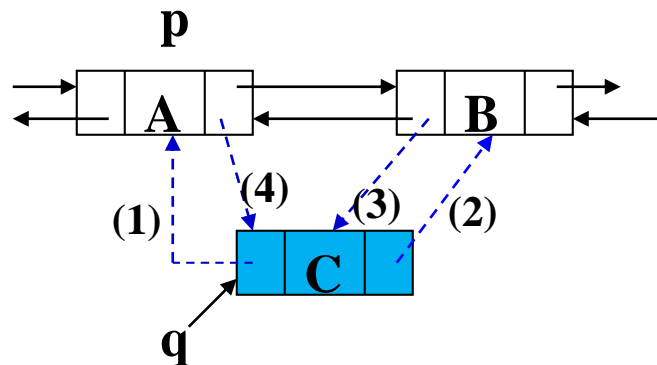
**$p \rightarrow \text{rlink} \rightarrow \text{llink} = p \rightarrow \text{llink}$**

**$\text{free}(p)$**

# 双向链表结点插入图示

## 插入方法1

**`q=(PDoubleNode)malloc(sizeof(struct DoubleNode))`**



**p后插入新结点:**

**(1) `q->llink = p;`**

**(3) `p->rlink->llink = q;`**

**(2) `q->rlink = p->rlink;`**

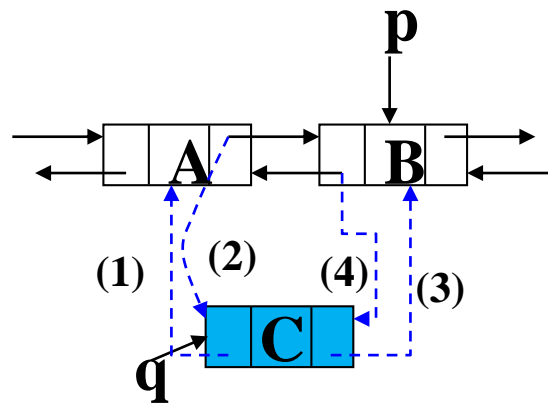
**(4) `p->rlink = q;`**



# 双向链表结点插入图示

插入方法2:

**`q=(PDoubleNode)malloc(sizeof(struct DoubleNode))`**



往双向链表中p前插入结点

(1) **`q->llink = p->llink;`**

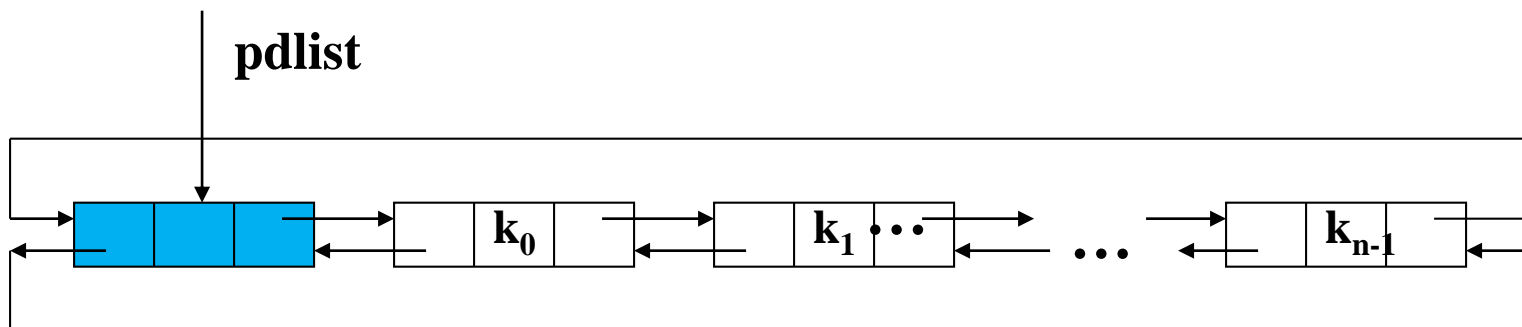
(2) **`p->llink->rlink = q;`**

(3) **`q->rlink = p;`**

(4) **`p->llink = q;`**

# 双向循环链表

- 将单链表或者双链表的头尾结点链接起来，就是一个循环链表
- 不增加额外存储花销，却给不少操作带来了方便
  - 从循环表中任一结点出发，都能访问到表中其他结点



**PDoubleList pdlist; /\* pdlist指向双链表 \*/**

空表判断 : **pdlist->rlink == NULL**

最后结点判断: **p->rlink == pdlist** 或 **p == pdlist->llink**

第一个结点 : **pdlist->rlink**

最后结点 : **pdlist->llink**

# 链表的边界条件

---

## □ 几个特殊点的处理

- 头指针处理
- 非循环链表尾结点的指针域保持为NULL
- 循环链表尾结点的指针回指头结点

## □ 链表处理

- 空链表的特殊处理
- 插入或删除结点时指针勾链的顺序
- 指针移动的正确性
  - ◆ 插入
  - ◆ 查找或遍历

# 线性表的应用——链表部分

---

## □ 问题

- Josephus问题

- 一元多项式表示和运算



# 循环链表方式实现

---

## □ 步骤：

### ■ 建立循环链表；

### ■ 出列算法；

- ① 找循环单链表中的第s个结点放在指针变量p中，
- ② 从p所指结点开始计数寻找第m个结点，输出该结点的元素值；
- ③ 删除该结点，并将该结点的下一个结点放在指针变量p中，转去执行②，直到所有结点被删除为止；

### ■ 主程序

## □ 时间复杂度分析：

- 创建链表，求第s个结点，求n个第m个应出列的元素  
 $O(n) + O(s) + O(m*n)$

# 线性表的应用——链表部分

---

## □ 问题

- Josephus问题
- 一元多项式表示和运算

# 一元多项式表示和运算

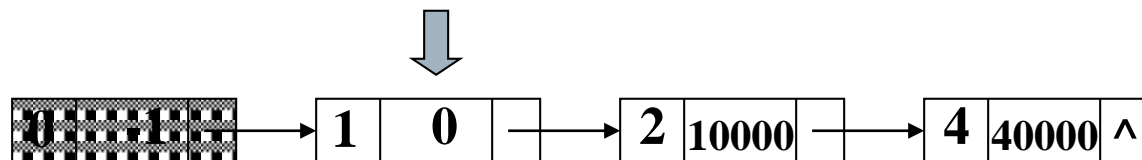
一元多项式:  $P_n(x) = p_0 + p_1x + p_2x^2 + \dots + p_n x^n$

线性表表示:  $P = (p_0, p_1, p_2, \dots, p_n)$

顺序表表示: 只存系数 (第*i*个元素存 $x^i$ 的系数)

特殊问题:  $p(x) = 1 + 2x^{10000} + 4x^{40000}$   浪费空间

链表表示: 每个结点结构



# 一元多项式表示和运算

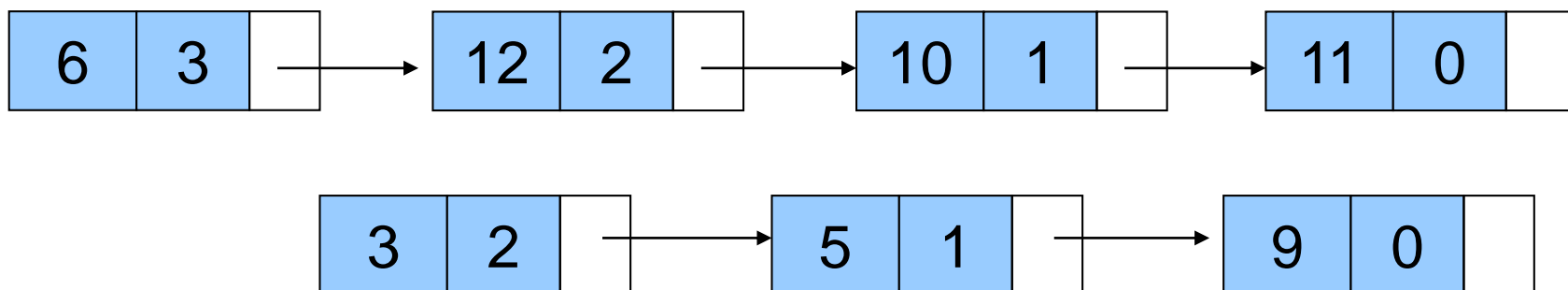
□ 数据：

$$f(x)=6x^3+12x^2+10x+11$$

$$g(x)=3x^2+5x+9;$$

□ 运算：  $f(x)+g(x)$   $f(x)-g(x)$   $f(x)*g(x)$   $f(x)/g(x)$

□ 用线性表表示多项式；用节点表示项；用节点之间的关系表示项之间的降幂关系。如：





# 一元多项式表示和运算

## □ 数据定义：

```
struct linknode
{
    float c; //coefficient, 记录每一项的系数
    int e;    //power, 记录每一项的幂指数
    struct linknode *link; //指向下一个节点的指针
};

typedef struct linknode * Plist;
typedef struct linknode * Pnode;
```

## □ 操作：

- 加法：相同指数对应结点的系数项相加，如和为0，删除结点，否则必定为和链表的一个结点。（实质上就是两个单链表的合并问题）
- 减法、乘法：
- 除法：

# 一元多项式表示和运算

## □ 输入的形式和输入值的范围：

- 从终端读入参数构造一个多项式链表，同时读入一对数字，第一个作为多项式项的系数，第二个作为项的幂指数；当读入0 0时，输入操作结束。
- 输入：3 2； 2 1； 0 0 //表示输入 $3x^2 + 2x$   
4 1； 1 0； 0 0 //表示输入 $4x + 1$   
+ //表示选择加法运算
- 输出的形式：  
如果多项式链表为空，输出空多项式信息，即多项式为0；上述运算输出： $3x^2 + 6x + 1$

# Example: 输入输出设计

系数	指数	指针
----	----	----

- 输入链1: 

	H	→
--	---	---

 → 

3	4	→
---	---	---

 → 

2	2	→
---	---	---

 → 

1	0	
---	---	--
- 输入链2: 

	H	→
--	---	---

 → 

2	2	→
---	---	---

 → 

3	0	
---	---	--
- 输出链: 

	H	→
--	---	---

 → 

3	4	→
---	---	---

 → 

4	2	→
---	---	---

 → 

4	0	
---	---	--
- 多项式的输入
  - 每次建立一个节点并处理其信息: `insert(c, e, Node)`
  - 每次插入到链表的相应位置, 实现降幂排列

# main 函数设计

---

```
• int main()
• {
•     Plist list1,list2,list3,list4;
•     int flag;    //判断运算类型
•     list1=createnualllist();
•     list2=createnualllist();
•     list3=createnualllist();
•     list4=createnualllist();
•
•     printf("请输入第一个多项式:\n");
•     list1=READ(list1);
•
•     printf("请输入第二个多项式:\n");
•     list2=READ(list2);
•
•     printf("请选择运算类型\n");
•     printf("1:加法 2:减法 3:乘法 4:除法\n");
•     printf("请输入数字");
•     scanf("%d",&flag);
•
•     if(flag==1) list3=ADD(list1,list2,list3);
•     else if(flag==2) list3=MINUS(list1,list2,list3);
•     else if(flag==3) list3=MUTIPLY(list1,list2,list3);
•     else if(flag==4)
•     {
•         list3=DIVIDE(list1,list2,list3);
•         list4=YUSHU(list1,list2,list3,list4);
•     }
•
•     //以下略
• }
```

# READ函数设计

---

① //将多项式的数据读入到链表list中，返回list

② Plist READ (Plist list)

③ {

④     float C; int E;

⑤     Pnode p=list;

⑥     printf("请按降幂输入，系数和幂用空格隔开，以“0 0”结束.\n");

⑦     while(1)

⑧     {

⑨         scanf("%f %d",&C,&E);

⑩         insert(C,E,p);

⑪         if(C==0&&E==0) return list;    //判断结束

⑫         if(C!=0) p=p->link;

⑬     }

⑭ }

# Insert函数设计

---

① //将新结点插入到链表中P指向的结点之后

② int insert(float C,int E,Pnode p)

③ {

④ if(C==0) return 1; //不插入系数为0的项

⑤ Pnode q=(Pnode)malloc(sizeof(struct linknode));

⑥ if(q==NULL) return 0;

⑦ q->c=C;

⑧ q->e=E;

⑨ q->link=p->link;

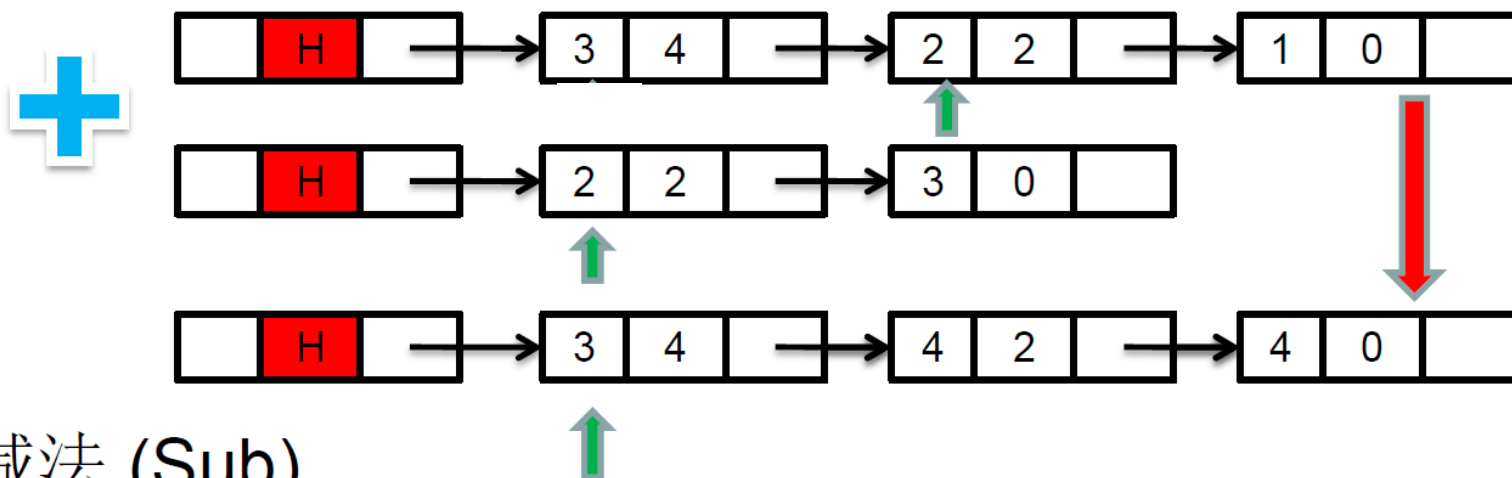
⑩ p->link=q;

⑪ return 1;

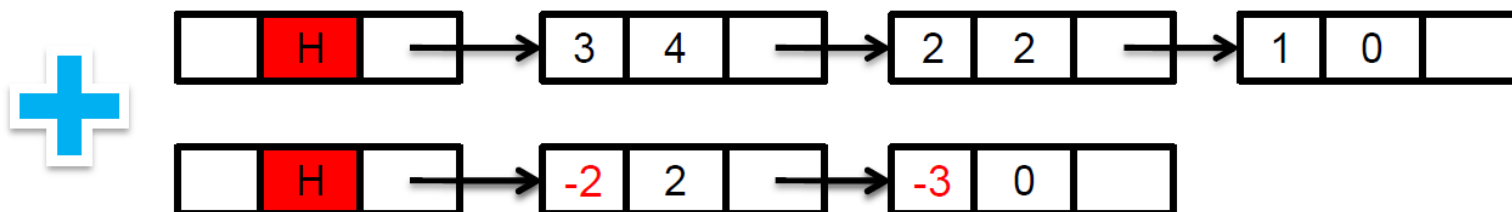
⑫ }

# 加法与减法

- 加法 (Add)


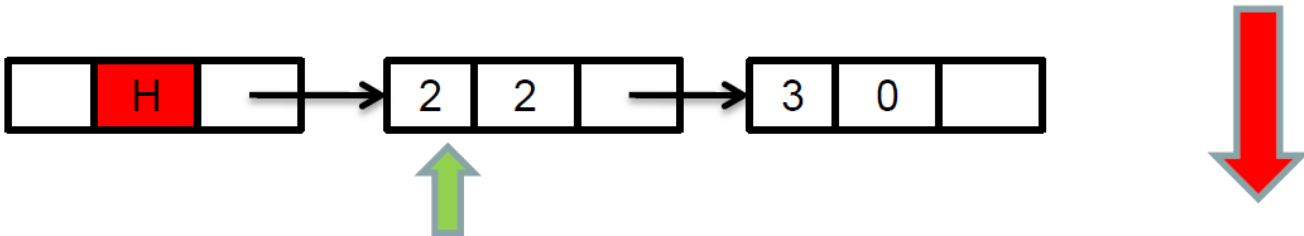



- 减法 (Sub)



# 乘法

---

- 输入1 
- 输入2 
- 临时 



# 内容提要

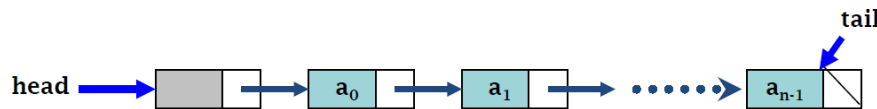
□ 线性表（逻辑结构）

$$\{a_0, a_1, \dots, a_{n-1}\}$$

□ 顺序表示和实现（顺序表）



□ 链式表示和实现（链表）



□ 顺序表和链表的比较

# 线性表实现方法的比较

---

## □ 顺序表的主要优点

- 没有使用指针，不用花费额外开销
- 线性表元素的读访问非常简洁便利

## □ 链表的主要优点

- 无需事先了解线性表的长度
- 允许线性表的长度动态变化
- 能够适应经常插入删除内部元素的情况

## □ 总结

- 顺序表是存储静态数据的不二选择
- 链表是存储动态变化数据的良方

# 顺序表和链表的比较

---

## □ 顺序表

- 插入、删除运算时间代价 $O(n)$ ，查找则可常数时间完成
- 预先申请固定长度的连续空间
- 如果整个数组元素很满，则没有结构性存储开销

## □ 链表

- 插入、删除运算时间代价 $O(1)$ ，但找第 $i$ 个元素运算时间代价 $O(n)$
- 存储利用指针，动态地按照需要为表中新的元素分配存储空间
- 每个元素都有结构性存储开销

# 顺序表和链表存储密度

---

$n$  表示线性表中当前元素的数目

$P$  表示指针的存储单元大小（通常为4 bytes）

$E$  表示数据元素的存储单元大小

$D$  表示可以在数组中存储的线性表元素的最大数目

## □ 空间需求

- 顺序表的空间需求为  $DE$

- 链表的空间需求为  $n(P + E)$

## □ $n$ 的临界值，即 $n > DE / (P + E)$

- $n$  越大，顺序表的空间效率就更高

- 如果  $P = E$ ，则临界值为  $n = D / 2$

# 应用场合的选择

---

## □ 顺序表不适用的场合

- 经常插入删除时，不宜使用顺序表
- 线性表的最大长度也是一个重要因素

## □ 链表不适用的场合

- 当读操作比插入删除操作频率大时，不应选择链表
- 当指针的存储开销，和整个结点内容所占空间相比比例较大时，应该慎重选择

# 小结

---

- 线性表的逻辑结构
- 线性表的顺序存储结构，要求能够综合应用
  - 顺序表上的插入、删除操作及其平均时间性能分析。
  - 利用顺序表设计算法解决简单的应用问题。
- 链式存储结构的单链表，要求能够综合应用
  - 单链表如何表示线性表中元素之间的逻辑关系。
  - 单链表中头指针和头结点的使用。
  - 单链表上实现的建表、查找、插入和删除等基本算法，并分析其时间复杂度。
  - 利用单链表设计算法解决简单的应用问题。

# 小结

---

- 线性表的链式存储结构的循环链表和双链表，要求能够简单应用
  - 循环链表的定义，及其上算法与单链表上相应算法的异同点。
  - 双链表的定义及其相关的算法。
  - 单链表、双链表、循环链表链接方式上的区别。