

# 第一章 线性表

1 线性表及其基本运算

2 顺序存储的线性表

3 顺序存储的栈和队列

4 链接存储的线性表

5 链接存储的栈和队列

6 线性表的其他存储方式

7 线性表的查找

8 广义表

**线性表：**具有**相同类型**的 $n$ 个数据元素 $k_0, k_1, \dots, k_{n-1}$ 的有限序列。  
记为  $(k_0, k_1, \dots, k_{n-1})$ 。


**线性表的长度：**元素个数 $n$ 称为线性表的长度。

称长度为零的线性表为**空的线性表**（简称空表）。

$k_0, k_1, \dots, k_i, k_{i+1}, \dots, k_{n-1}$

  
最前面的一个元素

当 $n \geq 1$ 时，

  
最后一个元素

$k_i$  是 $k_{i+1}$ 的直接**前趋元素**， $k_{i+1}$ 是 $k_i$ 的直接**后续元素**，

**结点（记录）**：线性表中的数据元素也称为结点，或者称为记录。它可以是整数、实数、字符或字符串；也可以是由若干个数据项组成，其中每个数据项可以是一般数据类型，也可以是构造类型。数据项称为**字段**或**域**。

星期	Mon	Tue	Wed	Thu	Fri	Sat	Sun
温度	15.5	16.0	15.7	15.0	15.0	16.1	16.4



结点

每个节点有两个字段：一个是三个字符组成的字符串，另一个是实数。

## 第一章 线性表

### 1

## 线性表及其基本运算

**关键字：**线性表中的结点可由若干个字段组成，我们称能唯一标识结点的字段为关键字，或简称为**键**。

键

↓

表1.1.2 职工工资表

结点 →

职工号	姓名	性别	年龄	工资
001	Wang	male	35	160.50
002	Cai	male	32	150.00
003	Zhang	female	28	130.00
⋮	⋮	⋮	⋮	⋮

### 线性表的基本运算

**查找：** 查找线性表的第 $i$  ( $0 \leq i \leq n-1$ ) 个结点；  
在线性表中查找值为 $x$ 的结点。

**插入：** 把新结点插在线性表的第 $i$  ( $0 \leq i \leq n-1$ ) 个位置上；  
把新结点插在值为 $x$ 的结点的前面或后面。

**删除：** 在线性表中删除第 $i$  ( $0 \leq i \leq n-1$ ) 个结点；  
在线性表中删除值为 $x$ 的结点。

**其他运算：** 遍历，排序，分解，合并

## 第一章 线性表

### 2

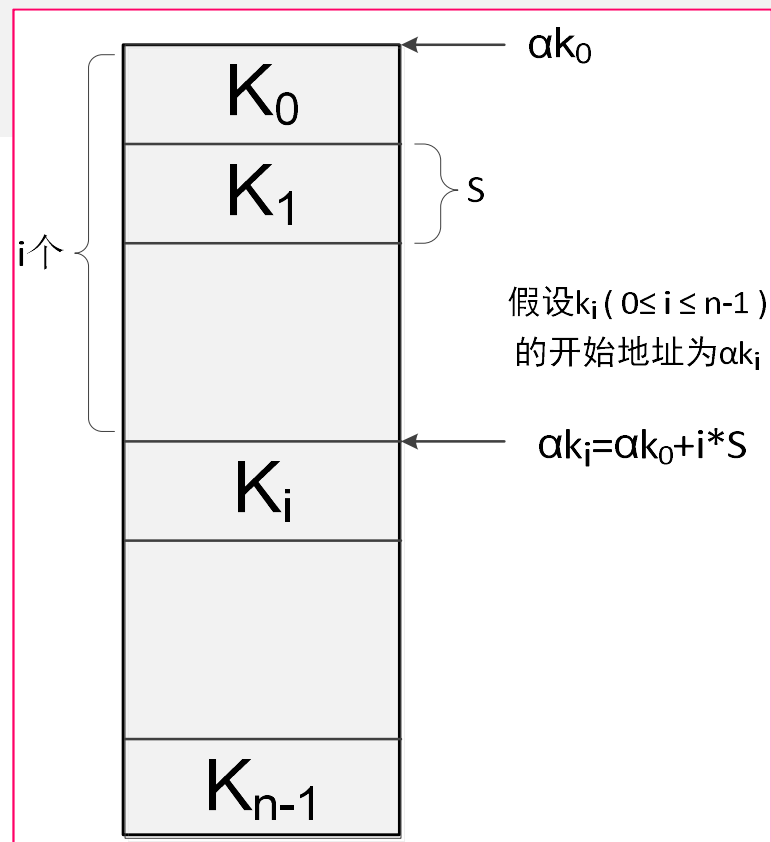
## 顺序存储的线性表

**线性表的顺序存储：**用一组连续的存储单元依次存储线性表中的结点。

顺序存储：可以直接计算出 $k_i$ 的开始地址。  
对于存取第 $i$  ( $0 \leq i \leq n-1$ ) 个结点特别方便。

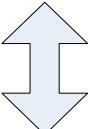
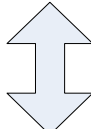
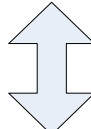
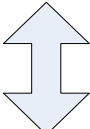
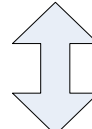
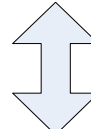
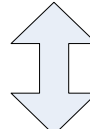
存放结点 $k_i$  ( $0 \leq i \leq n-1$ ) 的开始地址为  $\alpha k_i$

$$\alpha k_i = \alpha k_0 + i * S$$

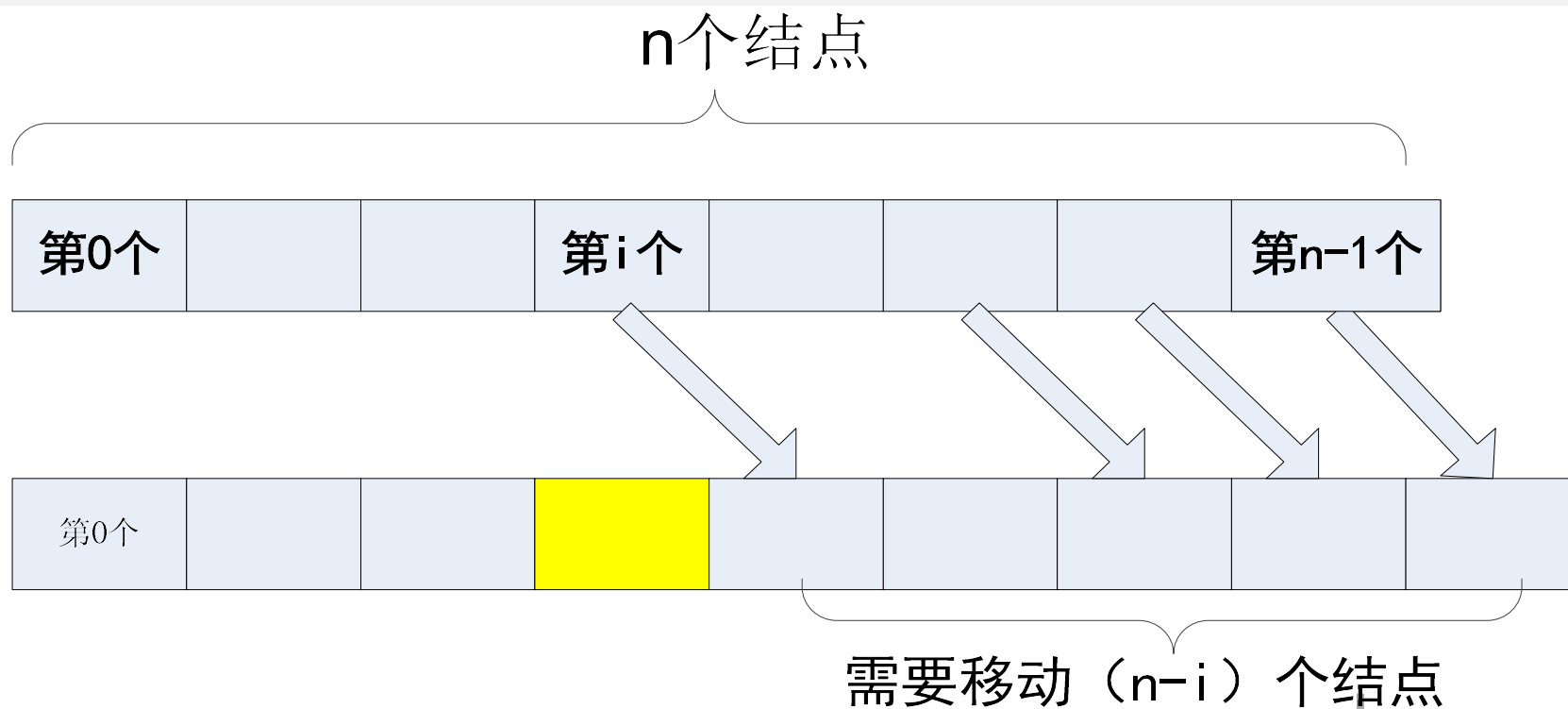


用C语言的数组表示线性表 (  $k_0, k_1, \dots, k_{n-1}$  ) 。

```
#define MAXSIZE 100
int list[MAXSIZE];
int n;
```

$\alpha k_0$	$\alpha k_1$	.....	$\alpha k_{i-1}$	$\alpha k_i$	.....	$\alpha k_{n-1}$
						
&list[0]	&list[1]	.....	&list[i-1]	&list[i]	.....	&list[n-1]

线性表的插入：在具有 $n$ 个结点的线性表中，把新结点插在线性表的第 $i$ （ $0 \leq i \leq n-1$ ）个位置上，使原来长度为 $n$ 的线性表变成长度为 $(n+1)$ 的线性表。





### 线性表的插入

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

```
#define MAXSIZE 100
```

```
int list[MAXSIZE];
```

```
int sq_insert(int list[], int *p_n, int i, int x)
```

```
{    int j;
```

```
    if( i<0 || i>*p_n) return (1);
```

非法情况

```
    if( *p_n == MAXSIZE ) return (2);
```

```
    for( j= *p_n; j>i; j--)
```

循环执行移动操作

```
        list[j] = list[j-1];
```

```
    list[i] = x;
```

```
    (*p_n)++;
```

```
    return(0);
```

```
}
```

### 线性表的插入——效率分析

执行时间主要花费在移动节点。假设把新结点插在第 $i$  ( $0 \leq i \leq n-1$ ) 个位置上的概率为 $P_i$

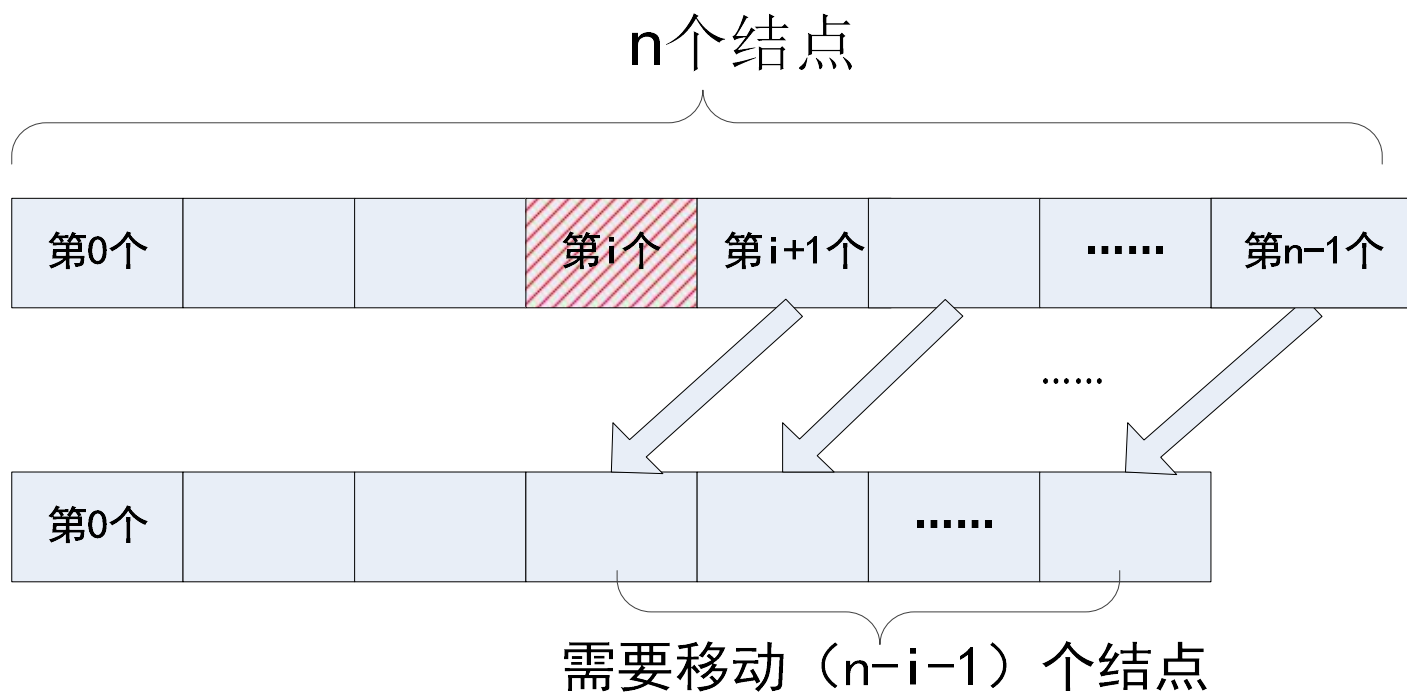
移动结点的平均次数:  $\sum_{i=0}^n p_i (n - i)$

如果线性表中每个可插入位置的插入概率相同, 则移动平均次数为 $\frac{n}{2}$

当线性表的结点很多, 且每个结点的数据量相当大的时候, 花费在移动结点上的执行时间就会很长。



线性表的删除：在具有 $n$ 个结点的线性表中，删除第 $i$  ( $0 \leq i \leq n-1$ ) 个位置上的结点，使原来长度为 $n$ 的线性表变成长度为 $(n-1)$ 的线性表。



线性表的删除：在具有 $n$ 个结点的线性表中，删除第 $i$  ( $0 \leq i \leq n-1$ ) 个位置上的结点，使原来长度为 $n$ 的线性表变成长度为 $(n-1)$ 的线性表。

线性表的删除

```
int sq_delete(int list[], int *p_n, int i) //删除元素
```

```
{    int j;  
    if( i<0 || i>=*p_n) return (1);
```

非法情况

```
        for( j=i+1; j<*p_n; j++)  
            list[j-1] = list[j];  
        (*p_n)--;  
        return (0);
```

循环执行移动操作

```
}
```

## 线性表的删除——效率分析

执行时间主要花费在移动节点。假设删除第 $i$  ( $0 \leq i \leq n-1$ ) 个位置上的结点的概率为 $P_i$ ，因为删除第 $i$  ( $0 \leq i \leq n-1$ ) 个位置上的结点需要移动  $(n-i-1)$  个结点。所以移动结点的平均次数：

$$\sum_{i=0}^n p_i(n-i-1)$$

如果假设删除线性表中任何一个结点的概率相同，则移动平均次数约为 $\frac{n}{2}$

当线性表的结点很多，且每个结点的数据量相当大的时候，花费在移动结点上的执行时间就会很长。



用顺序存储的线性表表示多项式

一般代数多项式可以写成：

$$A(x) = a_m x^{e_m} + a_{m-1} x^{e_{m-1}} + \dots + a_1 x^{e_1} + a_0 x^{e_0}$$

我们可以用包含coef和exp两个字段的结点表示多项式的非零项。  
coef给出系数，exp给出次数（指数）

用数组poly[MAXN]表示多项式。

```
#define MAXN 100
```

```
typedef struct term { float coef;  
                      int exp;  
                      } TERM;
```

```
TERM poly[MAXN];
```

用顺序存储的线性表表示多项式

例如，对于下面两个多项式：

$$A(x) = 8x^{60} + 6x^{50} + 4x^{25} + 2x^{10} + 1$$

$$B(x) = 7x^{60} - 6x^{50} + 3x^{10}$$

用C函数实现多项式的相加。（教材P6~8）

提问：

- 1) 需要考虑哪些非法情况
- 2) P8 的两个while程序，会在某一次多项式相加中会依次被执行到吗？

### 用顺序存储的线性表表示多项式

#### 时间复杂度分析：

执行时间主要花费在三个循环上。

设 $A(x)$ 和 $B(x)$ 非零项的个数分别为 $m$ 和 $n$ ，对于第一个循环，当 $A(x)$ 和 $B(x)$ 的各项次数都不相同时，执行循环的次数最多，共执行 $(m+n)$ 次，故其执行时间最多为 $O(m+n)$ ；

第二和第三个循环最多分别执行 $m$ 次和 $n$ 次，其执行时间最多分别为 $O(m)$ 和 $O(n)$ 。

执行`polyadd()`的时间为： $O(m+n)$



## 第一章 线性表

### 2

## 顺序存储的线性表

### 上机练习

1001, 1002, 1003,  
1004: 线性表插入删除操作  
1005: 线性表的比较  
1023: 一元多项式乘法

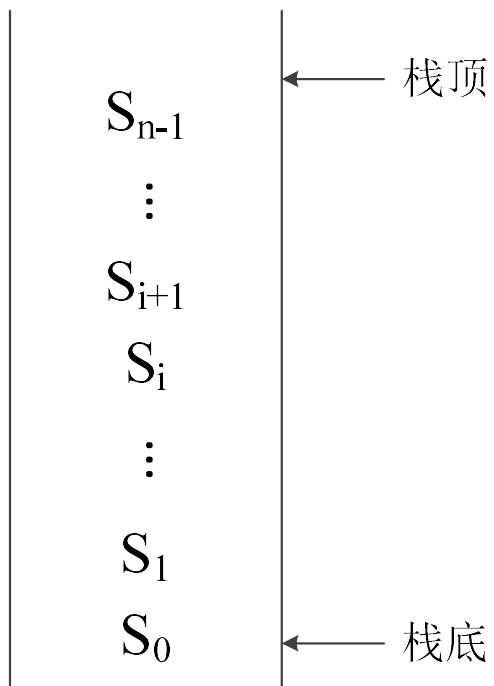
栈和队列都是特殊的线性表。



汉诺塔： 如果要拿出最底下的盘子，一定要先拿走上面的盘子。  
最底下的盘子肯定是最早放进去的。

栈和队列都是特殊的线性表。

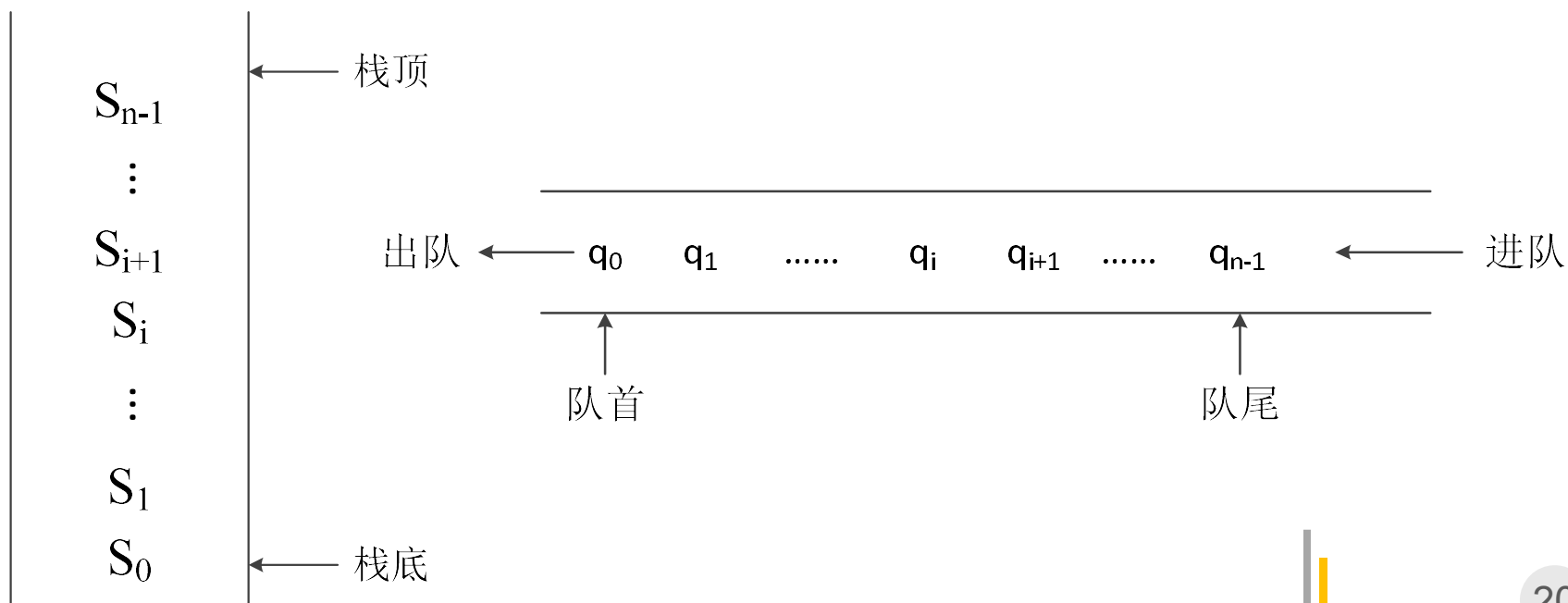
栈：只允许在一端进行插入和删除的线性表，**后进先出**。



栈和队列都是特殊的线性表。

**栈**：只允许在一端进行插入和删除的线性表。**后进先出**。

**队列**：只允许在一段进行插入，并且只允许在另一端进行删除的线性表。**先进先出**。



### 栈及其基本运算

栈：只允许在一端进行插入和删除的线性表。

栈顶：允许插入和删除的一端。

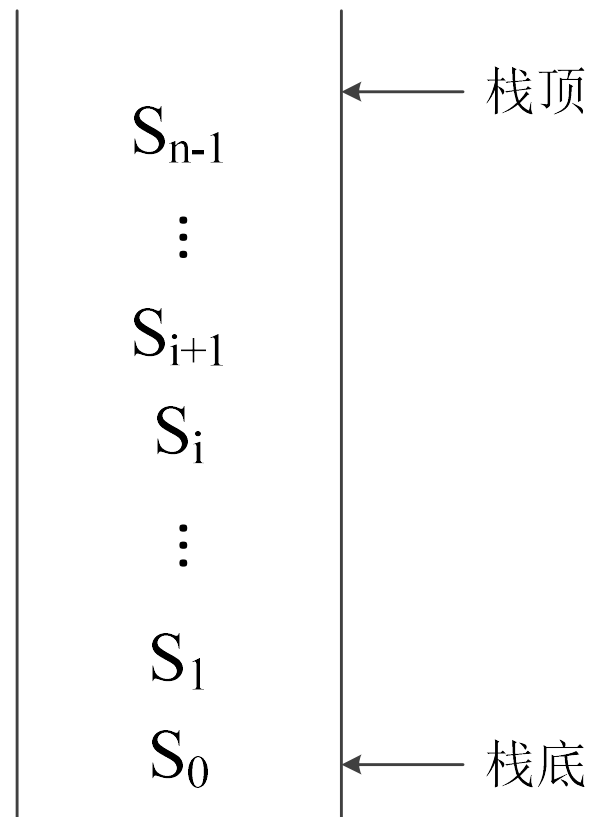
栈底：不允许插入和删除的一端。

进栈：栈的插入。

出栈：栈的删除。

最后进栈的结点必定最先出栈，  
所以栈具有**后进先出**的特性。

简称**LIFO (Last In First Out)**表。

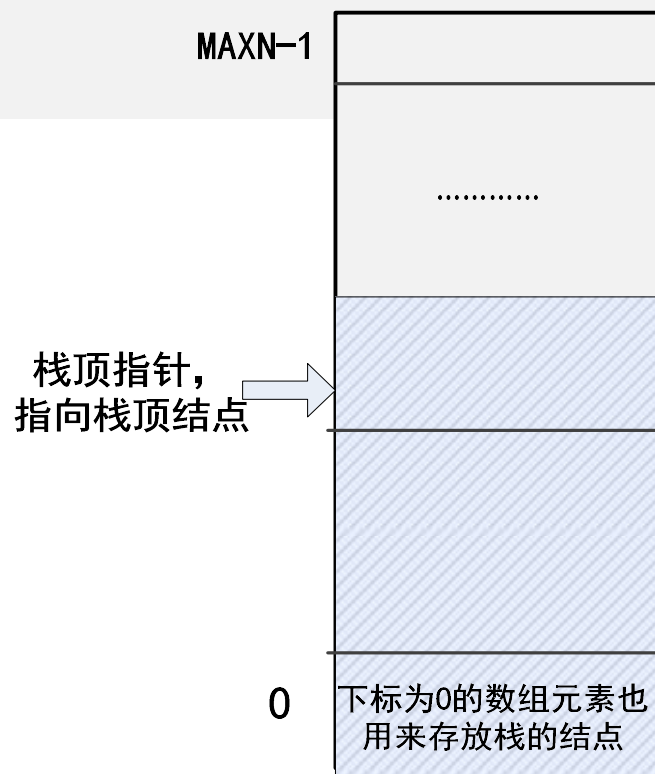


### 栈及其基本运算

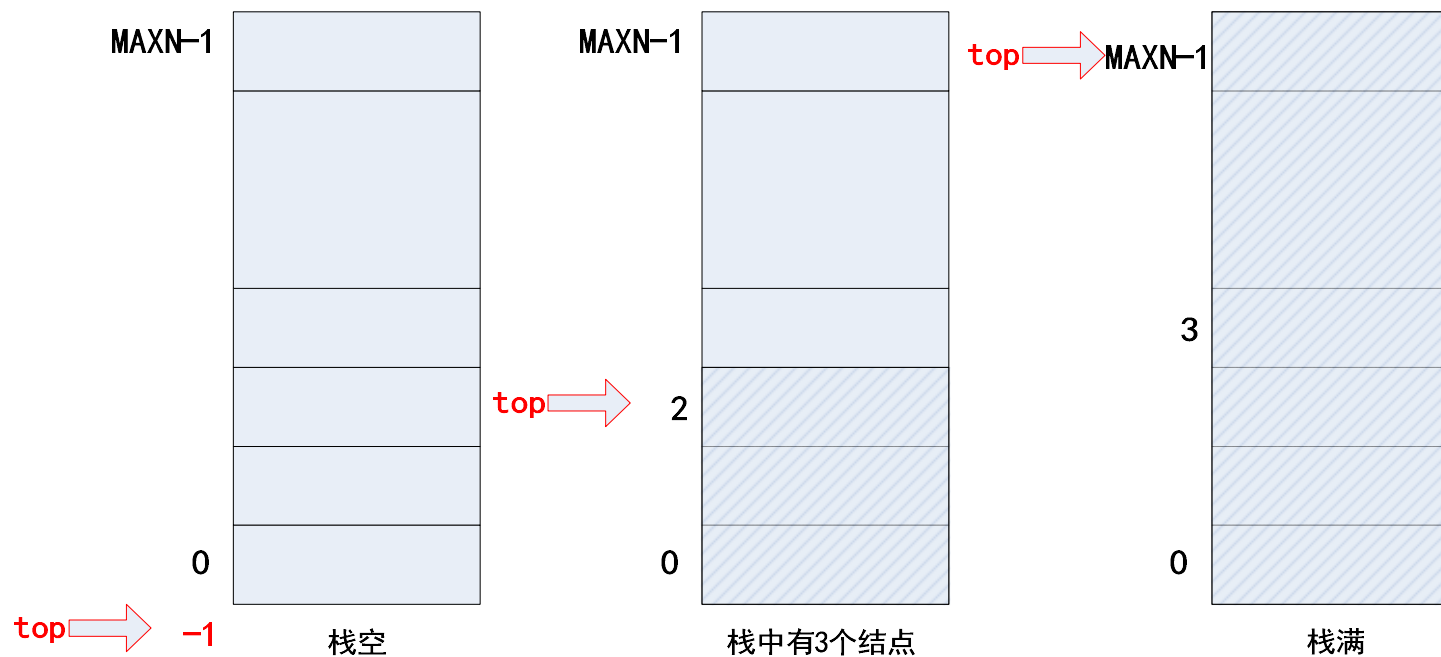
顺序存储：用数组表示栈。栈顶指针 $top$ （在顺序存储中，是数组的一个下标）指向栈顶结点在数组的存放位置。

`push()`：向栈内压入一个成员；  
`pop()`：从栈顶弹出一个成员；  
`empty()`：如果栈为空返回`true`，否则返回`false`；  
`top()`：返回栈顶，但不删除成员；  
`size()`：返回栈内元素的个数；

进栈：首先判断是否栈满，不满，可以进栈。  
出栈：首先判断是否栈空，不空，可以出栈。



### 栈及其基本运算



方法一：  $top$  指针指向栈顶结点在数组的存放位置

### 栈及其基本运算



方法二：  $top$  指针指向下一次进栈结点的存放位置



## 栈及其基本运算

栈的表示：可用数组表示栈。

### 用C语言的数组表示栈的两种方法

	指针top指向栈顶结点 在数组的存放位置	指针top指向下一次进栈结点的 存放位置
栈满	$Top = MAXN - 1$	$Top = MAXN$
栈空	$Top = -1$	$Top = 0$
进栈	首先执行 $top + 1$ ，然后把进栈结点送到 $top$ 当前位置	首先把进栈结点送到 $top$ 所指的数组元 素中，然后执行 $top + 1$
出栈	首先把 $top$ 所指向的栈顶结点送到接受 结点的变量中， 然后执行 $top - 1$	首先执行 $top - 1$ ， 然后把 $top$ 所指向的栈顶结点送到接 受结点的变量中
注意	栈满时不能进栈，栈空时不能出栈	

### 栈及其基本运算

栈的基本运算：进栈及出栈

```
#include <stdio.h>
#define MAXN 26
char stack[MAXN];
int top = 0;           //把栈设置成空的初态
int push(char x) //进栈
{
    if (top >= MAXN)
        return (1); //栈满，进栈失败，返回1
    stack[top++] = x; //进栈，然后top加1
    return (0);
}
```

### 栈及其基本运算

栈的基本运算：进栈及出栈

```
#include <stdio.h>
#define MAXN 26
char stack[MAXN];
int top = 0;           //把栈设置成空的初态

int pop(char *p_y) {   //pop() 函数实现出栈
    if(top == 0)
        return 1;     //栈空，出栈失败，返回1
    *p_y = stack[--top];
    return 0;         //出栈成功，返回0
}
```

### 栈及其基本运算

作业：

1. 在顺序存储的线性表基础上实现栈的基本操作。
2. 完成上机练习 1020, 1019, 1021

### 上机作业 1021

一个铁路调度站，为栈式结构，所有的火车必须右端进去并且从左端离开，现在有 $n(0 < n < 10)$ 列火车要进行调度，按照进入的顺序从1到 $n$ 进行编号。对于一个给定的出站序列，你需要判断是否是一个合法的序列。

例如： $n = 4$ , 出站序列为 4321，这是合法的，1234依次进栈，再依次出栈得到4321。

$n=3$ , 312 就是非法的。

### 栈及其基本运算

#### 上机作业 1021

```
读入要判断的序列的个数k,
for (j=0; j<k; j++) // 执行以下循环k次
{
    读入待判断的序列。
    n表示某一行序列中数字的个数 ; //需进栈的数字 是 1, 2, 3, .....n
    建立一个空栈;
    for (i = 1; i <= n; i++) //from 1 to n 依次进栈, 并判断是否需要出栈
    {
        i进栈;
        while(栈不空 && 栈顶元素== “待判断的序列” 的首元素)
        {
            出栈;
            序列去掉一个元素;
        }
    }
    if 序列中不再有元素 序列合法
}
```

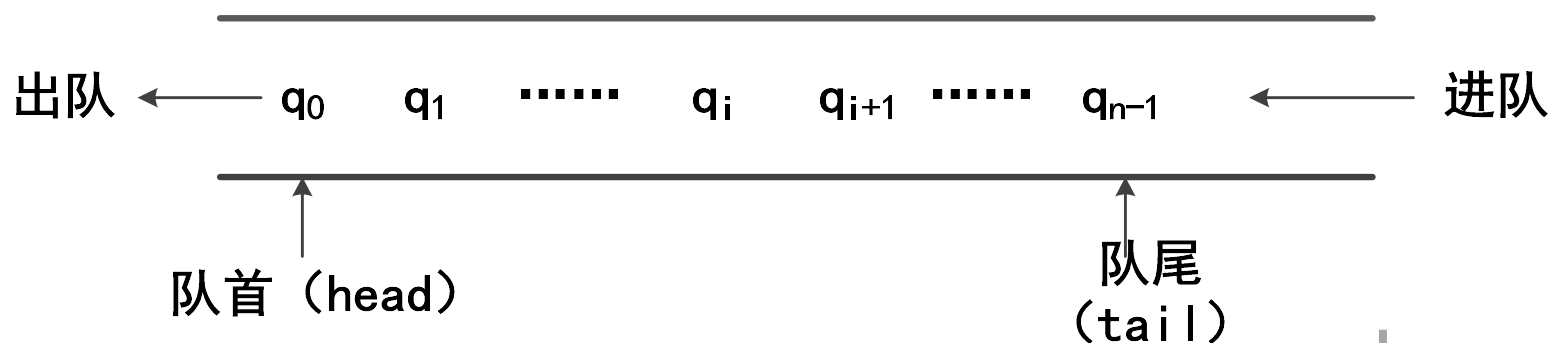
### 队列及其基本运算

队列：只允许在一端进行插入，且只允许在另一端进行删除的线性表。

队首：允许删除的一端。称删除为出队。

队尾：允许插入的一端。称插入为进队。

最先进入队列的结点必定最先出队，所以队列具有先进先出的特性。  
简称为**FIFO (First In First Out)** 表。

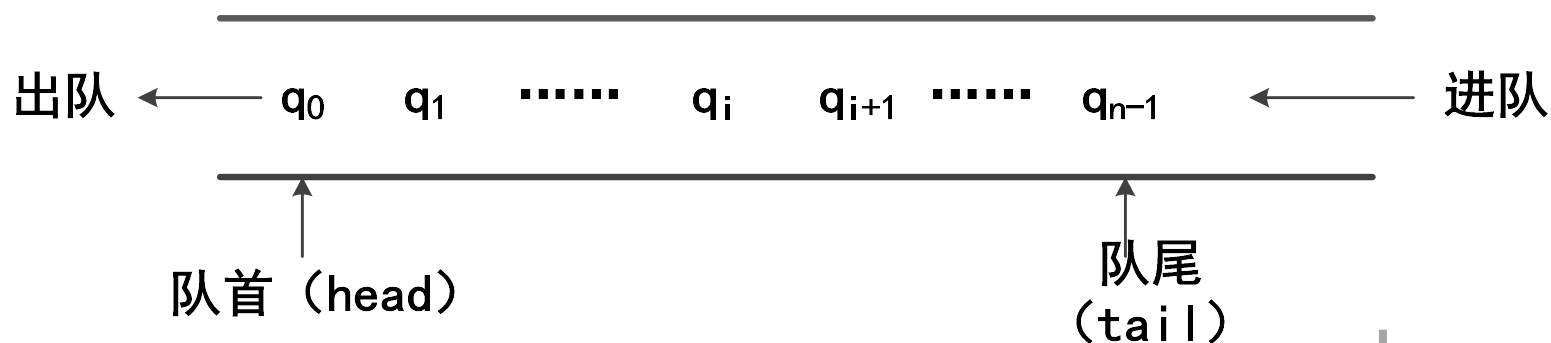


### 队列及其基本运算

顺序存储：使用数组表示队列。

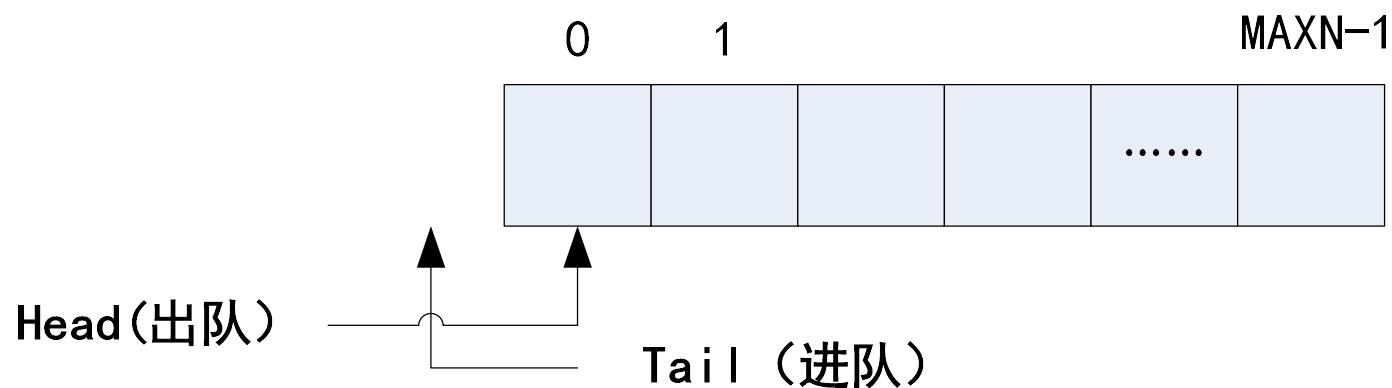
通常用一个指针（数组下标） $head$ 指向首结点在数组的存放位置，称 $head$ 为头指针；

用另一个指针（数组下标） $tail$ 指向队尾结点在数组的存放位置，称 $tail$ 为尾指针。



## 队列及其基本运算

顺序存储——使用数组表示队列。

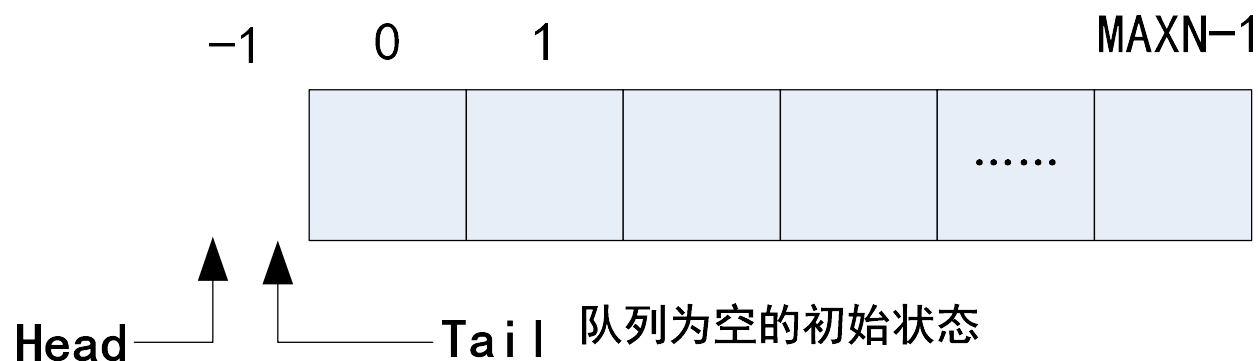


方法一：head指向队首结点在数组的存放位置，  
tail指向队尾结点在数组中的存放位置



## 队列及其基本运算

可以使用数组表示队列。



方法二：head指向存放队首结点的数组元素的前一个数组元素，  
tail指向队尾结点在数组中的存放位置

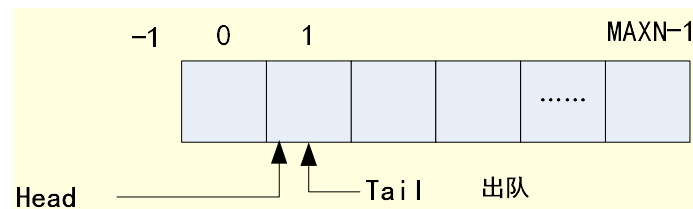
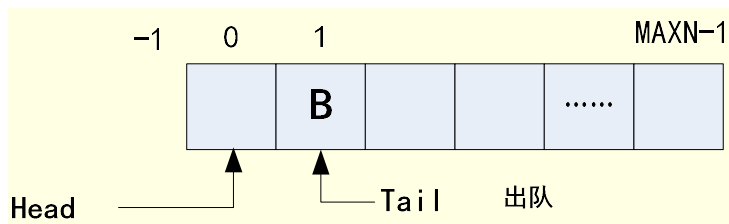
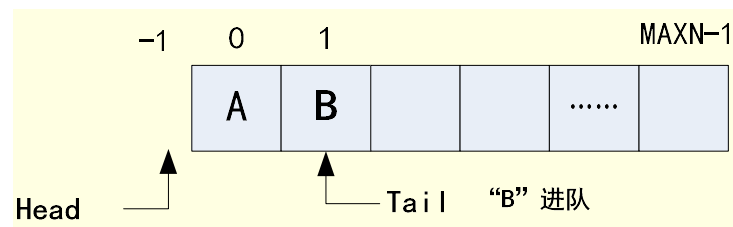
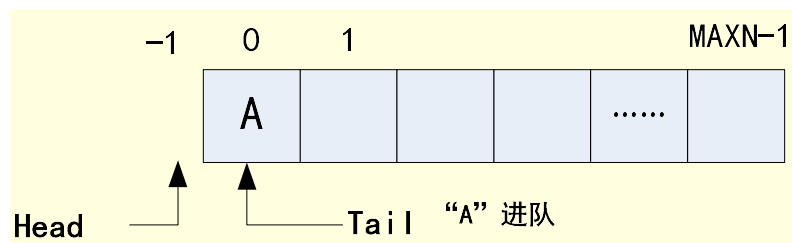
## 第一章 线性表

### 3

## 顺序存储的栈和队列

### 队列及其基本运算

用数组表示队列。



方法二：head指向存放队首结点的数组元素的前一个数组元素，  
tail指向队尾结点在数组中的存放位置

### 队列及其基本运算

```
int de_queue(char *p_y) //出队
{
    if (_____) //队空, 不能出队, 返回1
        return 1;

    *p_y = q[++ head]; //头指针+1, 把队首结点送到
    指针p_y所指的变量中
    return 0;          //出队成功, 返回0
}
```

## 队列及其基本运算

### 用C语言的数组表示队列的两种方法

	指针head指向队首结点在数组的存放位置；指针tail指向队尾结点在数组的存放位置。	head指向存放队首结点的数组元素的前一个数组元素；指针tail指向队尾结点在数组的存放位置。
队列为空的初始状态		Head=tail=-1
队满	tail=MAXN-1	tail=MAXN-1
进队	队列不满时，可进队。先执行tail=tail+1，再把新结点送到由tail所指的数组元素中。	首先tail+1，然后把新结点存放在由tail所指的数组元素中
出队	队列非空，可出队。先把head所指的队首结点送到接受结点的变量中，然后执行head+1。	head=Head+1，然后将由head指出的数组元素出队。
注意	队列满时不能进队，空时不能出队	

### 队列及其基本运算

作业：在顺序存储的线性表基础上，实现下述队列基本运算

back()返回最后一个元素  
empty()如果队列空则返回真  
front()返回第一个元素  
pop()出队一个元素  
push()入队一个元素  
size()返回队列中元素的个数

## 队列及其基本运算

### 队列的假满问题:

当 $\text{tail} = \text{MAXN} - 1$  且  $\text{head} = -1$ 时, 队列真满。

其余情况, 数组中有空着的元素, 此时即使 $\text{tail} = \text{MAXN} - 1$ , 队列也没有真正的满。如何充分利用已经空着的数组元素?

首先, 当队列出现队空时, 置  $\text{head} = \text{tail} = -1$

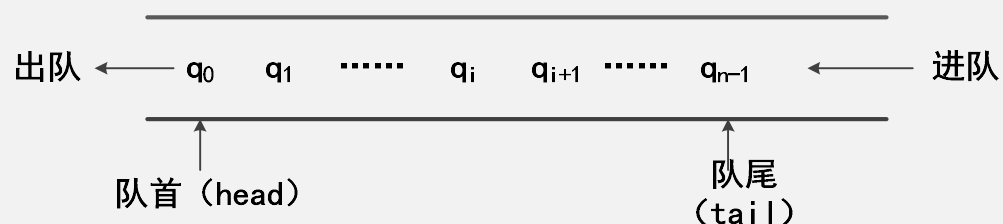
另外, 可采用两种方法:

- 1) 当一个结点出队时, 就把队列中所有的结点都依次向队首方向移动一个位置, 并修改头指针和尾指针。
- 2) 当 $\text{tail} = \text{MAXN} - 1$ 且队列不是真正的满时, 把队列的所有结点都依次移到数组的最前端, 并修改头指针和尾指针。

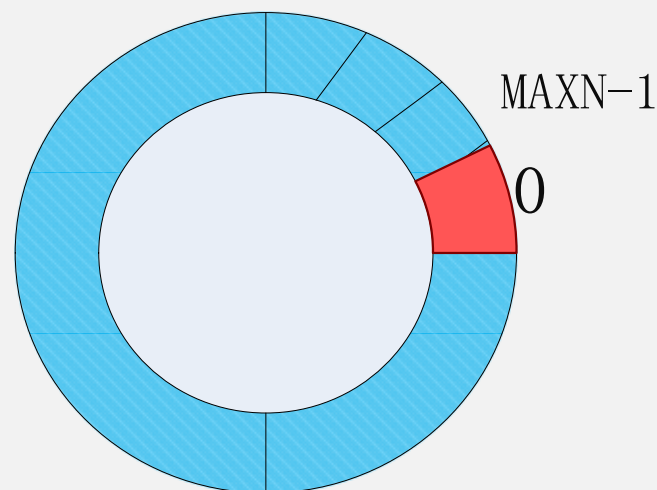
缺点: 移动结点, 影响效率

### 环形队列

如果我们把表示队列的数组的元素 $q[0]$ 和 $q[\text{MAXN}-1]$  “**连接**” 起来，形成一个环形的表，称这样的环形表为环形队列。



$$(\quad + 1) \% \text{MAXN}$$



## 环形队列

如果我们把表示队列的数组的元素 $q[0]$ 和 $q[\text{MAXN}-1]$ 连接起来，形成一个环形的表，称这样的环形表为环形队列。

head：指向存放队首结点的数组元素的前一个元素。

tail：指向队尾结点的数组元素。

初始时，置队列空的初始状态为： $\text{head}=\text{tail}=0$ ；

环形队列的队满和队空都有： $\text{head}=\text{tail}$ ；不能只用 $\text{head}=\text{tail}$ 来判断队列满或队列空。

解决方法：增加一个标志tag，在 $\text{head}=\text{tail}$ 的情况下，由tag为0（或1）来表示队列空（或满）。

进队及出队的函数 P14



## 环形队列

进队及出队的函数：

进队：

- a) 在执行进队操作**之前**，必须保证队列不满；
- b)  $\text{tail} = (\text{tail} + 1) \% \text{MAXN}$ ; 存入新元素；
- c) 执行**之后**，判断是否队满，如果满，置  $\text{tag} = 1$ ；

出队：

- a) 在执行出队操作**之前**，必须保证队列不空。
- b)  $\text{head} = (\text{head} + 1) \% \text{MAXN}$ ；
- c) 执行**之后**，判断是否队空，如果空，置  $\text{tag} = 0$ ；

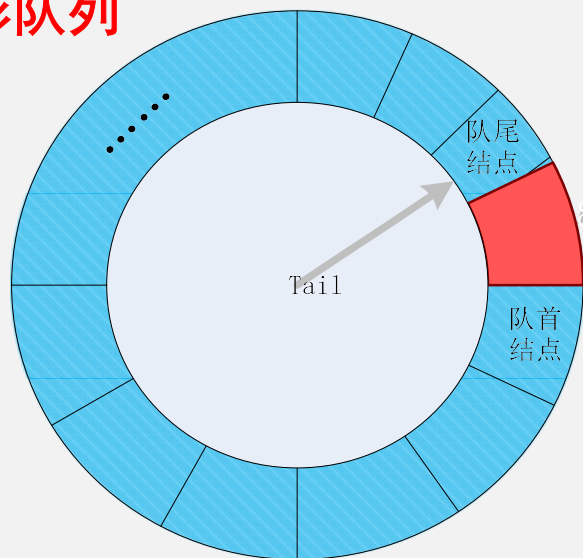
## 环形队列

标志tag：进队和出队都需要判断；在频繁进队和出队的情况下，将增加运行时间。

解决方案：我们可以以一个结点所占用的存储空间为代价，来节省执行时间。

进队时：先执行 $\text{tail} = (\text{tail} + 1) \% \text{MAXN}$ 。然后，判断tail是否已经赶上head。如果没有赶上head，则进队。如果赶上，则不允许进队（此时其实有一个空的数组元素），tail退回原来的位置，报告队满。

### 环形队列



Head: 指向存放队首结点的  
数组元素的前一个元素

此时虽然有一个空的数组元素，  
但是不允许进队

修改过之后的函数：

进队：先完成指针的移动，再判断是否队满。

出队：判断是否队空，不空执行出队操作。

注意：判断条件的变化。

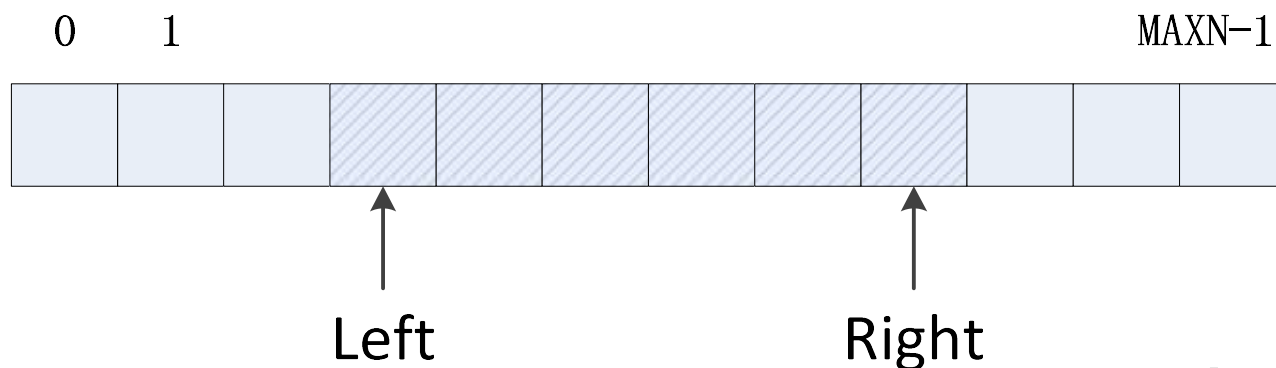
修改过之后的函数P 1 5（如果进队后导致队真的满，则不允许进队）

## 双向队列

我们称 **允许在两端进行插入和删除的线性表** 为双向队列。

双向队列没有队首和队尾之分，可用两个指针left和right分别指向双向队列中左端和右端结点。

用数组表示双向队列的结构，如下图：



## 双向队列

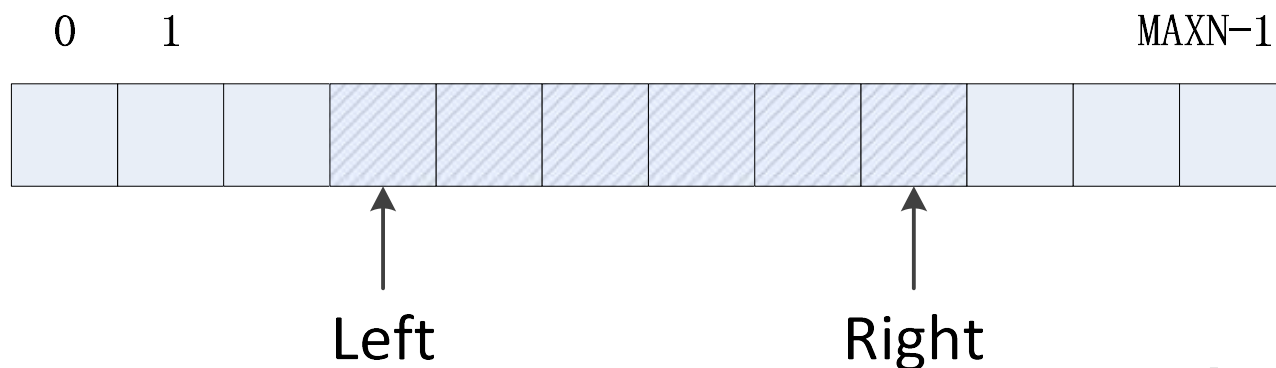
队列空的标志:  $\text{left} > \text{right}$

初始状态:

中间位置:  $m = (\text{MAXN}-1) / 2$ ,  $\text{left} = m + 1, \text{right} = m$

左端满的标志为:  $\text{left} = 0$

右端满的标志为:  $\text{right} = \text{MAXN}-1$



### 作业:

1. 在顺序存储的线性表基础上实现自己的队列

上机练习

1012 环形队列基本操作

1013, 医疗调度系统。分析题意之后可以发现可以用静态数组。

1014, 循环打印

1015, 查字典

1016, 欣赏书法

1017, 座位分配

1018, 玩具谜题

## 栈的应用

- 实现递归程序的执行;
- 执行算术表达式的计算;
- 用于需要用回溯法解决问题的场合;

### 栈的应用

例1. 用栈求算术表达式的值。

中缀表达式——二目运算符位于参与运算的两个操作数的中间

后缀表达式——把运算符放在参与运算的两个操作数的后面。

中缀表达式:  $(a+b)*c-d/(e*f)$

后缀表达式:  $ab+c*def*/-$





# 第一章 线性表

## 3

## 顺序存储的栈和队列

### 栈的应用

例1. 用栈求算术表达式的值。

中缀:  $5*(7-2*3)+8/2$

后缀:  $5\ 7\ 2\ 3\ \times\ -\ \times\ 8\ 2\ /\ +$



步骤	读剩的后缀式	栈中内容
1	5 7 2 3 × - × 8 2 / +	
2	7 2 3 × - × 8 2 / +	5
3	2 3 × - × 8 2 / +	5 7
4	3 × - × 8 2 / +	5 7 2
5	× - × 8 2 / +	5 7 2 3
6	- × 8 2 / +	5 7 6
7	× 8 2 / +	5 1
8	8 2 / +	5
9	2 / +	5 8
10	/ +	5 8 2
11	+	5 4
12		9

### 栈的应用

例1. 用栈求算术表达式的值。

后缀表达式的优点：不使用括号，不用考虑运算符的优先级别。

后缀表达式的求值：利用栈。

先从左到右扫描表达式，**每遇到一个操作数，就让操作数进栈；**

**每遇到一个运算符，就从栈中取出最顶上的两个操作数进行运算，然后将计算结果放进栈中。**

如此继续扫描，直到表达式最后一个运算符执行完毕，这时送入栈顶的值就是该后缀表达式的值。

非法情况：除法，幂运算，数太大溢出.....

### 栈的应用

//栈——用于计算，exp[]——存放后缀表达式

.....

```
while(exp[i] != '\0')
```

```
{
```

```
    if(是操作数)
```

```
    {
```

```
        s.push(运算数);
```

```
    }
```

```
        else
```

```
            switch (exp[i]) .....//出栈，按运算符进行运算，入栈
```

```
        i++;
```

```
    }
```

## 栈的应用

例1. 用栈求算术表达式的值。

如何把中缀表达式转换成等价的后缀表达式？

中缀和后缀：操作数出现的次序相同。

后缀：运算符出现的次序就是实际执行的次序；

中缀：不按运算符出现的次序执行，按优先级；

转换方法？

扫描中缀表达式，遇到操作数，输出。

遇到运算符，比较优先级，分情况输出。

代码段：P22-中缀表达式转换为后缀表达式

### 栈的应用

**例2.求解迷宫问题：**在给定的迷宫中找出一条从入口到出口的路径。

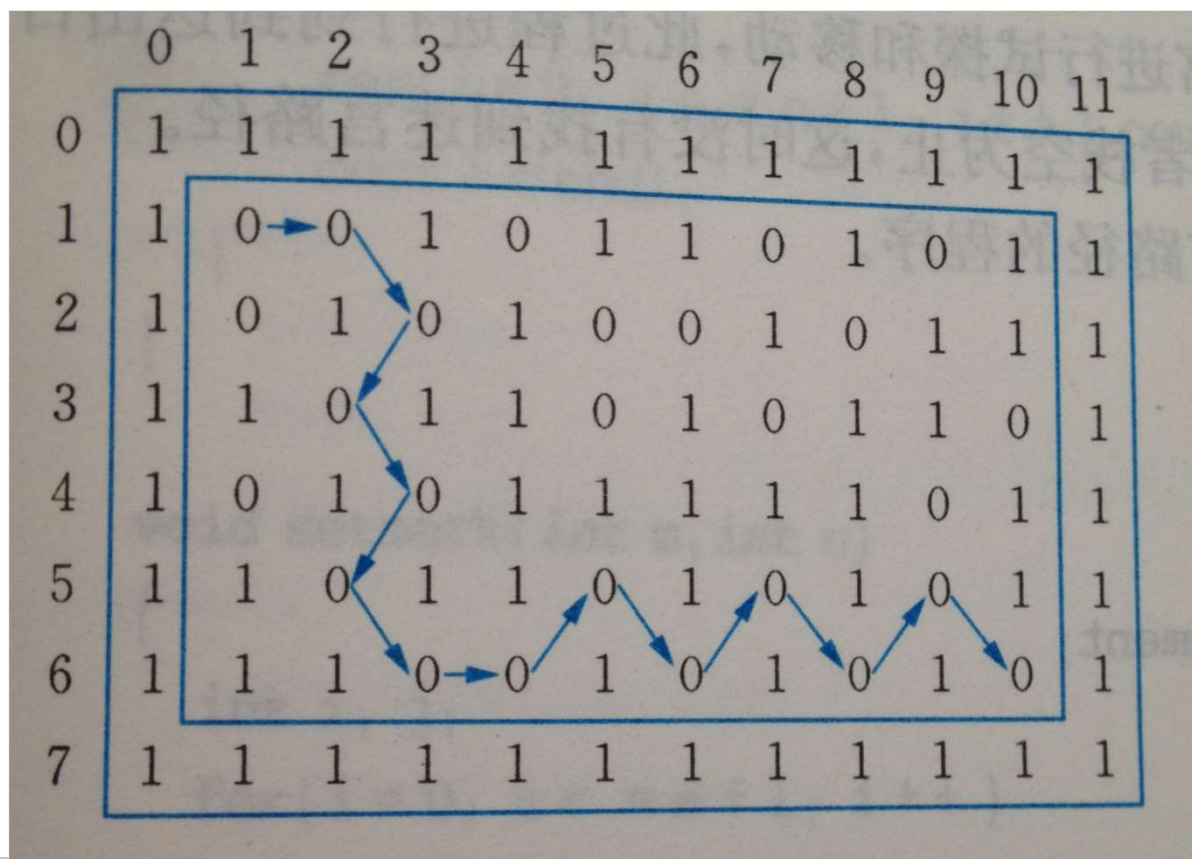
用矩阵表示迷宫。用0表示该位置可以通过，用1表示不能通过。在四周填上1，并假设入口为 $\text{maze}[1][1]$ ，出口为 $\text{maze}[m][n]$ 。

在迷宫某个位置  $(i, j)$  上，共有8个试探方向。如图1.3.15。

使用栈记录已经到达过的路径，栈中每个结点表示路径中一个到达位置  $(x, y)$ ，并指出沿着 $d$ 方向到达下一个位置。

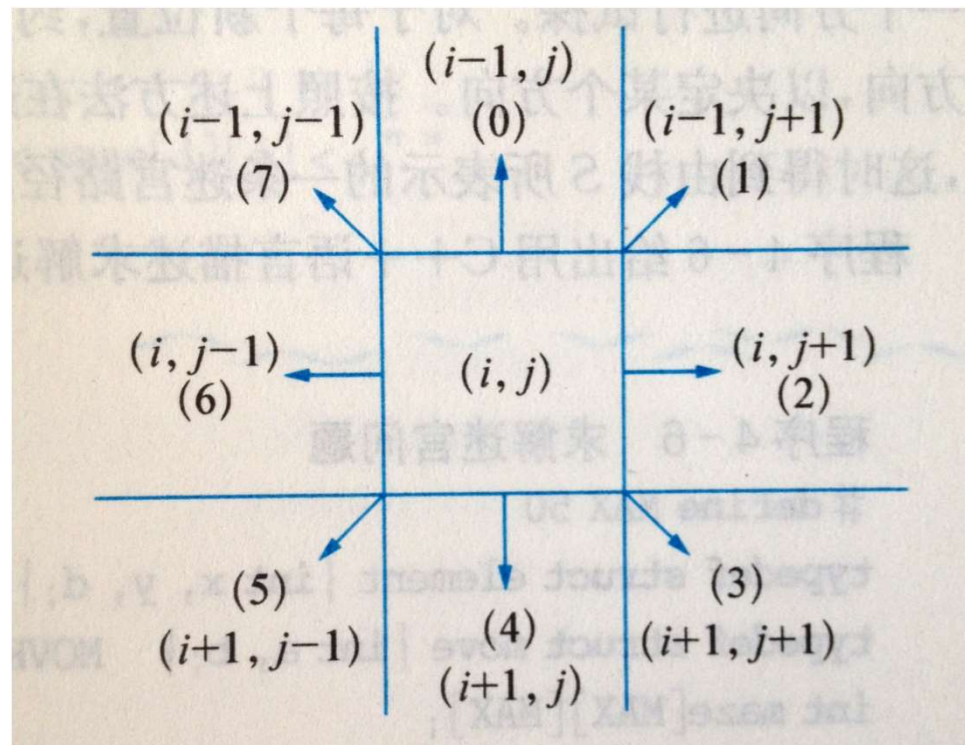
### 栈的应用

例2.求解迷宫问题：在给定的迷宫中找出一条从入口到出口的路径。



### 栈的应用

例2.求解迷宫问题：在给定的迷宫中找出一条从入口到出口的路径。



### 栈的应用

**例2.求解迷宫问题：**在给定的迷宫中找出一条从入口到出口的路径。

如果从位置  $(i, j)$  沿着 $k$ 方向到达了新位置  $(g, h)$  , 那么让  $(i, j, k)$  进栈, 再从  $(g, h)$  出发, 试着前进; 如果所取的方向受阻, 那么继续取八个方向中还没有试过的方向 (从0方向开始顺时针) 。

如果所有八个方向都受阻, 退到栈顶元素所指的位置, 沿着该位置的下一个方向进行试探。

此过程进行到到达出口为止。

代码段: P27-用栈求解迷宫



## 第一章 线性表

### 3

## 顺序存储的栈和队列

### 作业

综合上机练习：（1022， 1024， 1025， 1026,1027）

顺序存储的线性表的优点：

可以使用一个简单的公式计算出线性表中第 $i$ 个节点的存放位置，存取第 $i$ 个结点非常方便。

缺点：

- (1) 因为数组元素的个数是固定的，所以线性表的容量不易扩充。
- (2) 插入或删除一个结点平均需要移动一半左右的结点，效率低。

为了克服上述缺点，可以使用线性表的另一种存储方式——链接存储。

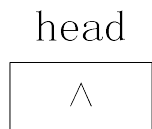
## 线性链表的逻辑结构和建立

**线性链表：**采用链接存储方式存储的线性表，也称单链表或链表。

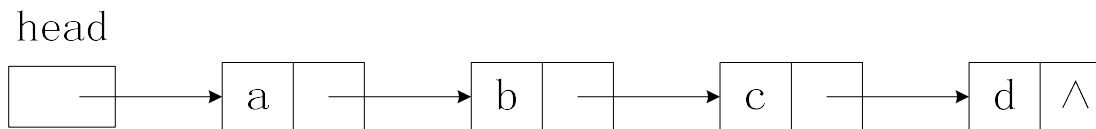
线性链表的每个结点除了有一个字段存放结点值外，还需要有一个字段用来存放其后续结点的地址，称这样的字段为**链接指针**。

**头指针：**指向第一个节点。如果线性链表为空，则头指针为空。当线性链表非空时，需将最后一个结点的链接指针置为空，表示该结点没有后续结点。

**优点：**插入和删除不需要移动后面的结点（以空间为代价）。



(a) 空链表



(b) 具有四个节点的链表

## 第一章 线性表

### 4

## 链接存储的线性表

### 线性链表的逻辑结构和建立

动态建立新的链表结点：

```
struct 结构体名称 * newNode;
```

```
newNode=(struct 结构体名称 *) malloc(sizeof(struct 结构体名称))
```

```
NODE * newNode;
```

```
newNode=(NODE *) malloc(sizeof(NODE))
```

```
struct 结构体名称  
{  
    结点成员;  
    .....  
    struct 结构体名称 *next;  
}
```

```
typedef struct 结构体名称  
{  
    结点成员;  
    .....  
    struct 结构体名称 *next;  
} 新定义的结构类型名称;
```

## 线性链表的逻辑结构和建立

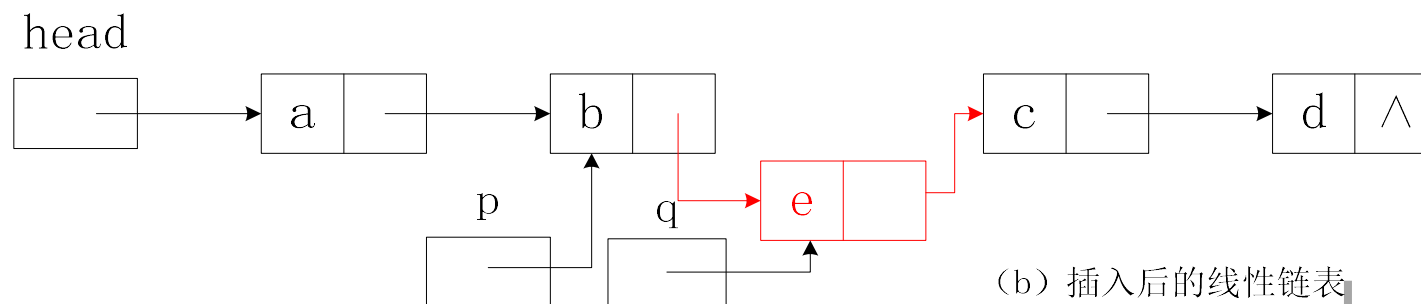
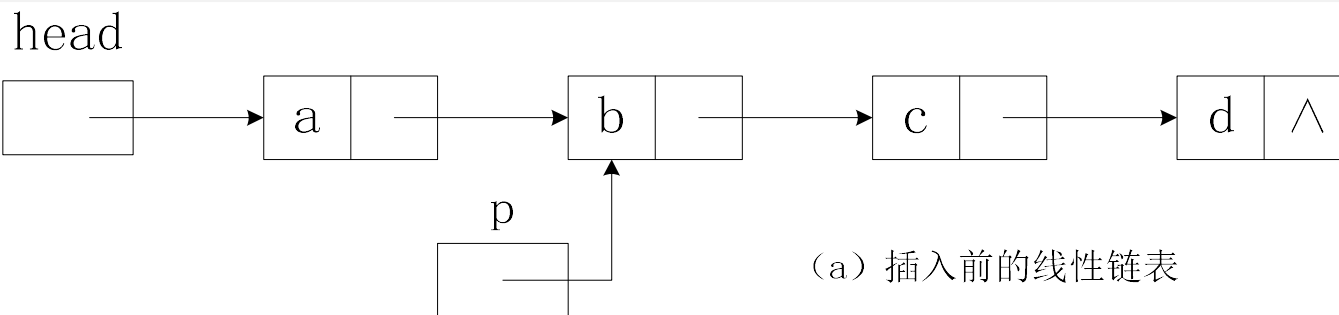
动态建立新的链表结点：

- ❑ 链表为空的情况如何表示
- ❑ 单链表最后一个结点的指针字段必须是NULL；
- ❑ 维护好head

代码段： P28-创建链表

### 线性链表的插入和删除

如何在指定节点后面插入一个新结点？



### 线性链表的插入和删除

如何在指定节点后面插入一个新结点？

void insert(NODE \*\*p\_head, char a, char b) //插入结点

```
{
    NODE *p, *q;
    q=(NODE*)malloc(sizeof(NODE));
    q->data=b;
    q->link=NULL;
    if(*p_head==NULL) //若链表为空，令头指针指向q
        *p_head=q;
    else
        {
            p=*p_head;
            while (p->data!=a&& p->link!=NULL) //寻找结点a
                p=p->link;
            q->link=p->link;
            p->link=q;
        }
}
```

完成结点插入

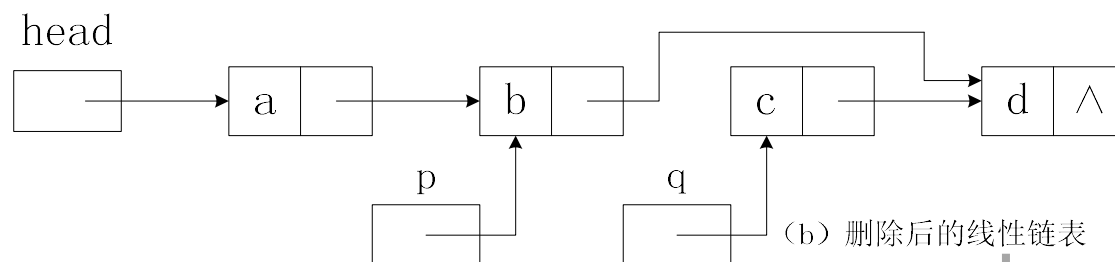
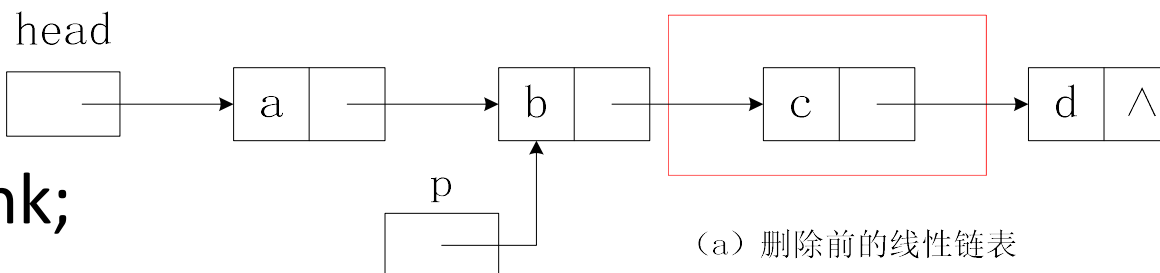
### 线性链表的插入和删除

如何在给定的线性链表中，删除指定结点的后续结点？

`q = p->link;`

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

`Free(q);`





### 线性链表的插入和删除

```
int delete(NODE **p_head, char a) //删除结点内容为“a”的结点
{
    NODE *p, *q;
    q = *p_head;
    if(q == NULL) return 1; //链表为空，返回1
    if(q->data == a){ //若第一个结点的data为a
        *p_head = q->link; //令头指针指向下一个结点
        free(q);
        return 0; //删除成功，返回0
    }
    else{
        while(q->data != a && q->link != NULL){
            p = q;
            q = q->link; //遍历寻找data为a的结点
        }
        if(q->data == a){ //如果q所指结点data为a
            p->link = q->link; //删除q所指结点
            free(q);
            return 0; //删除成功，返回0
        }
    }
}
```

### 用线性链表表示多项式

多项式需要记录：非零项的系数和次数。结点形式：

coef	exp	link
------	-----	------

多项式相加

- ❑ NULL?
- ❑ 链表的插入操作要保证插入完成后指数从大到小

代码段：P33-多项式相加By单链表

几种变形的线性链表

单链表在什么情况下处理不够方便？

### 几种变形的线性链表

环形链表：让线性链表的最后一个结点的指针指向第一个结点。  
优点：可以从其中的某个结点出发访问到表中所有的其他结点。

带表头结点的链表：在线性表中增加一个附加结点（也称为表头结点），它不是链表中的结点，数据字段用于存放一些辅助信息。

带表头结点的环形链表：



请画出上面三种链表的结构

### 几种变形的线性链表

利用带表头结点的环形链表表示多项式，实现多项式相加。

**作业：对比P33页的程序，说明利用带表头结点的环形链表使得程序段发生了什么变化，为什么？**

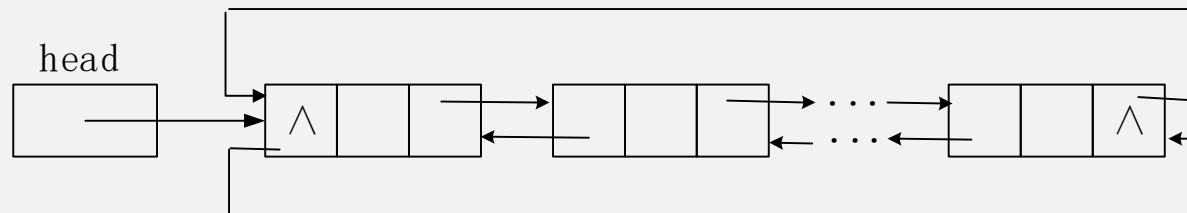
## 双向链表

之前讨论的线性链表的共同的缺点：要访问某个结点的前驱结点很不方便。

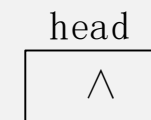
双向链表：每个结点带有两个指针：左指针指向该结点的前趋结点。右指针指向该结点的后续结点。

Llink	Data	Rlink
-------	------	-------

### 环形双向链表



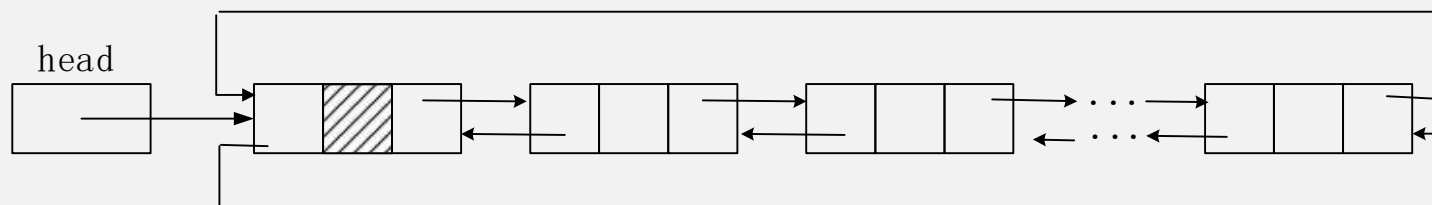
(a) 非空表



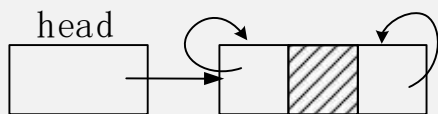
(b) 空表

图1.4.10 环形双向链表

### 带表头结点的双向链表 以及 带表头结点的环形双向链表



(a) 非空表



(b) 空表

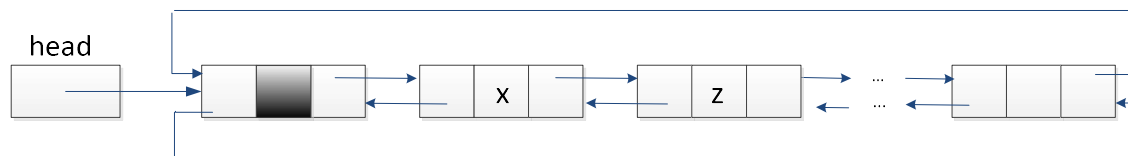
判断链表为空?  
判断已经遍历完一遍?

图1.4.12 带表头节点的环形双向链表

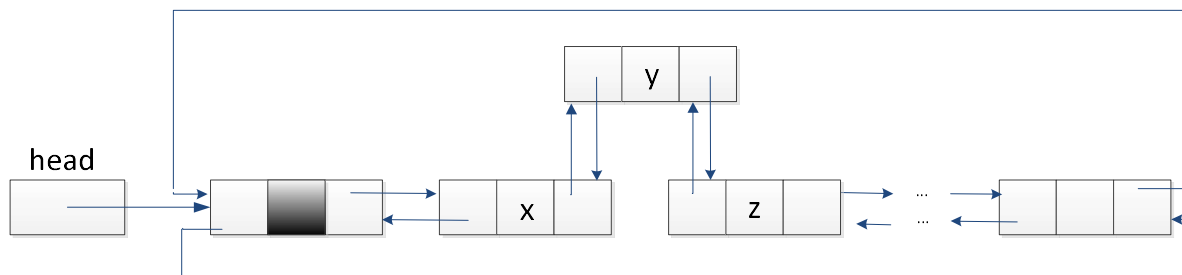


带表头结点的双向链表 以及 带表头结点的环形双向链表

带表头结点的环形双向链表的插入和删除都十分方便



(a) 插入前的双向链表



(b) 插入后的双向链表

### 带表头结点的双向链表 以及 带表头结点的环形双向链表

int insert\_d\_l(head, x, y) //将值为y的结点插在值为x的结点的右面。

    NODE \*head;

    char x,y;

```
    {
        NODE *p,*q;
        p=head->rlink;
        while(p!=head&& p->data!=x) p=p->rlink;
        if(p==head) return (1); //找不到结点x
        q=(NODE*)malloc(sizeof(NODE));
```

        q->data=y;

        q->rlink=p->rlink; 完成双向链表的指针变化

        p->rlink=q;

        q->rlink->link=q;

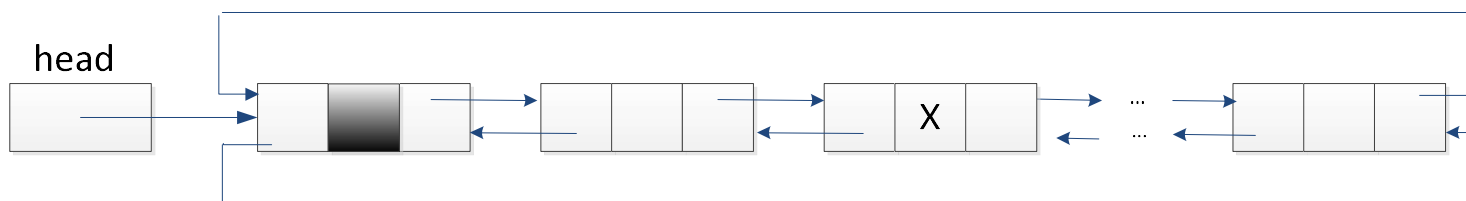
        q->llink=p;

        return(0);

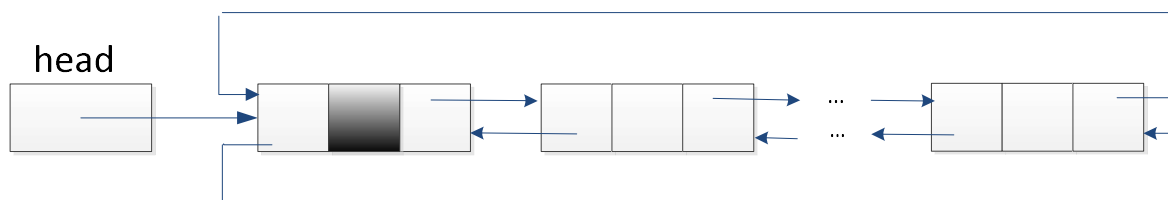
}

带表头结点的双向链表 以及 带表头结点的环形双向链表

带表头结点的环形双向链表的插入和删除都十分方便。



(a) 删除前的双向链表



(b) 删除后的双向链表

带表头结点的双向链表 以及 带表头结点的环形双向链表

```
int delete_d_l(NODE * head, char x)
{
    NODE *p;
    p=head->rlink;
    while(p!=head&& p->head!=x)
        p=p->rlink;
    if(p==head) return (1);
    p->llink->rlink=p->rlink;
    p->rlink->llink=p->llink;
    free(p);
    return(0);
}
```

## 第一章 线性表

### 4

## 链接存储的线性表

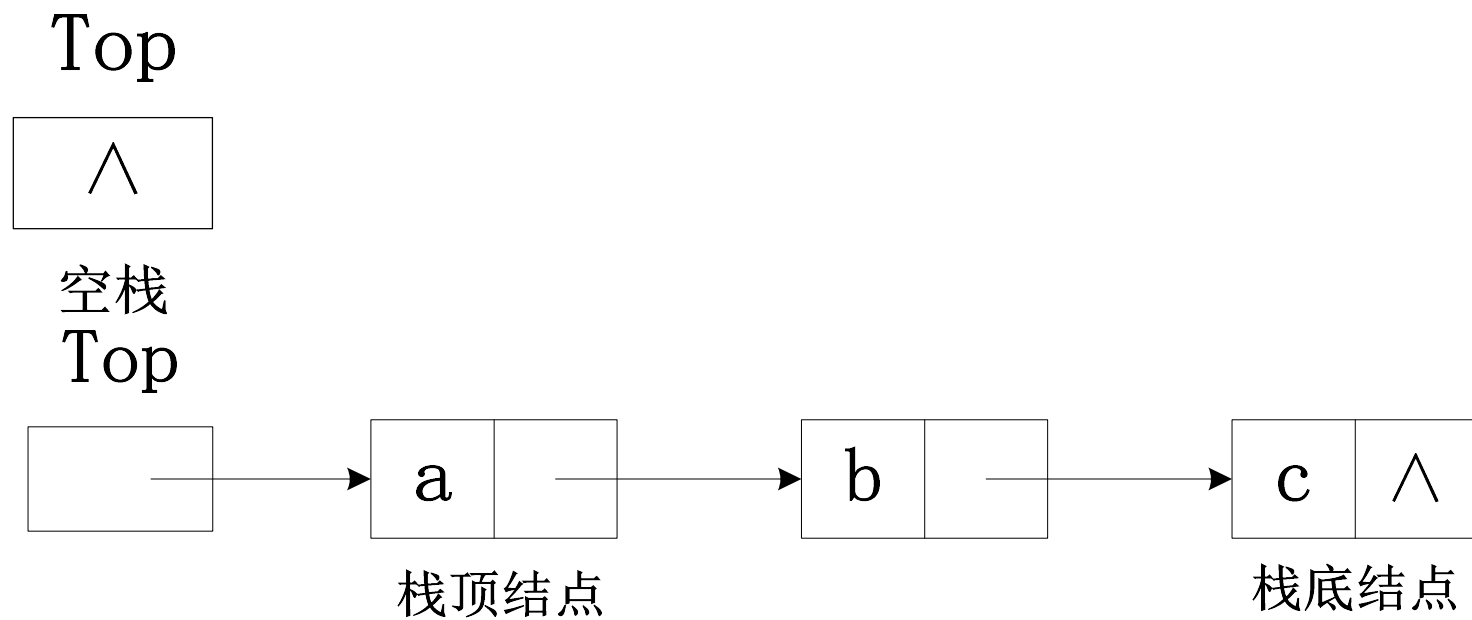
上机作业:

1006

1007

## 链栈

链栈：用链表表示栈。



## 第一章 线性表

### 链栈 进栈和出栈程序

**NODE\* top=NULL;**

void push\_l( char x)

```
{  
    NODE *p;  
    p=(NODE*)malloc (sizeof(NODE));  
    if (NULL == p) // 内存分配失败  
    {  
        fprintf(stderr, "Malloc failed.\n");  
        return ;  
    }  
    p->data=x;  
    p->link=top;  
    top=p;  
}
```

## 5

## 链接存储的栈和队列

```
Int push (x) //进栈  
char x;  
{ if (top>=MAXN)  
    return (1); //栈满, 进栈失  
败, 返回1  
    stack[top++]=x; //进栈, 然后  
top加1  
    return (0);  
}
```

链栈 进栈和出栈程序

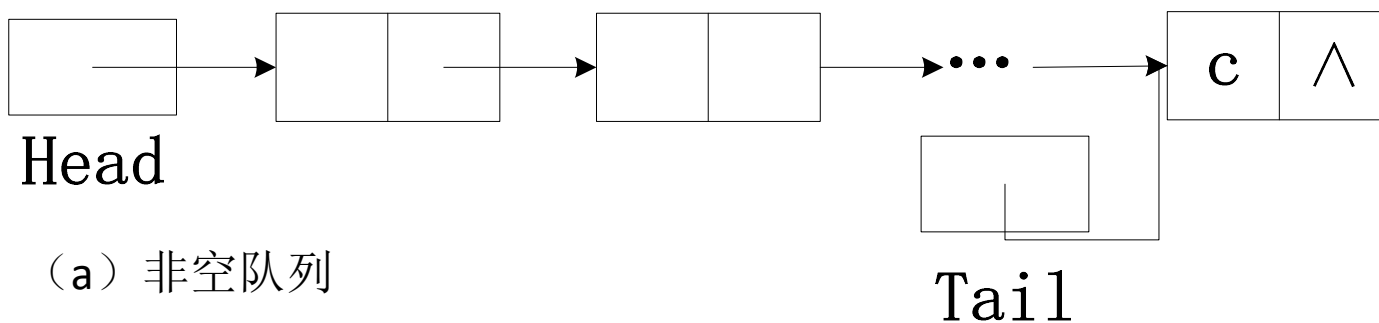
```
int pop_l(char *p_y)
{
    NODE *p;
    if(NULL==top) return (1);
    *p_y=top->data;
    p=top;
    top=top->link;
    free(p);
    return(0);
}
```

如何调用以取得pop的值



链接队列：用 链表 表示队列。

```
NODE * head, *tail;
```



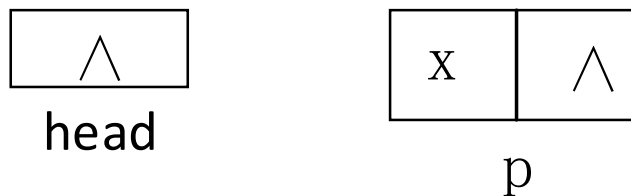
空队列?

```
If ( head == NULL ) .....
```

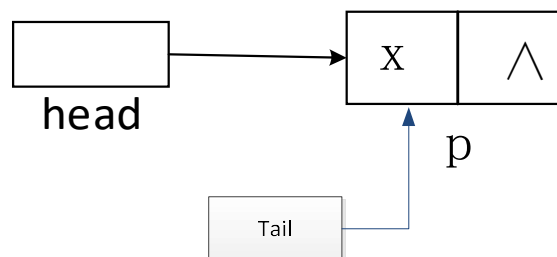
## 链接队列 进队和出队程序

### 队列空时进队

进队前:

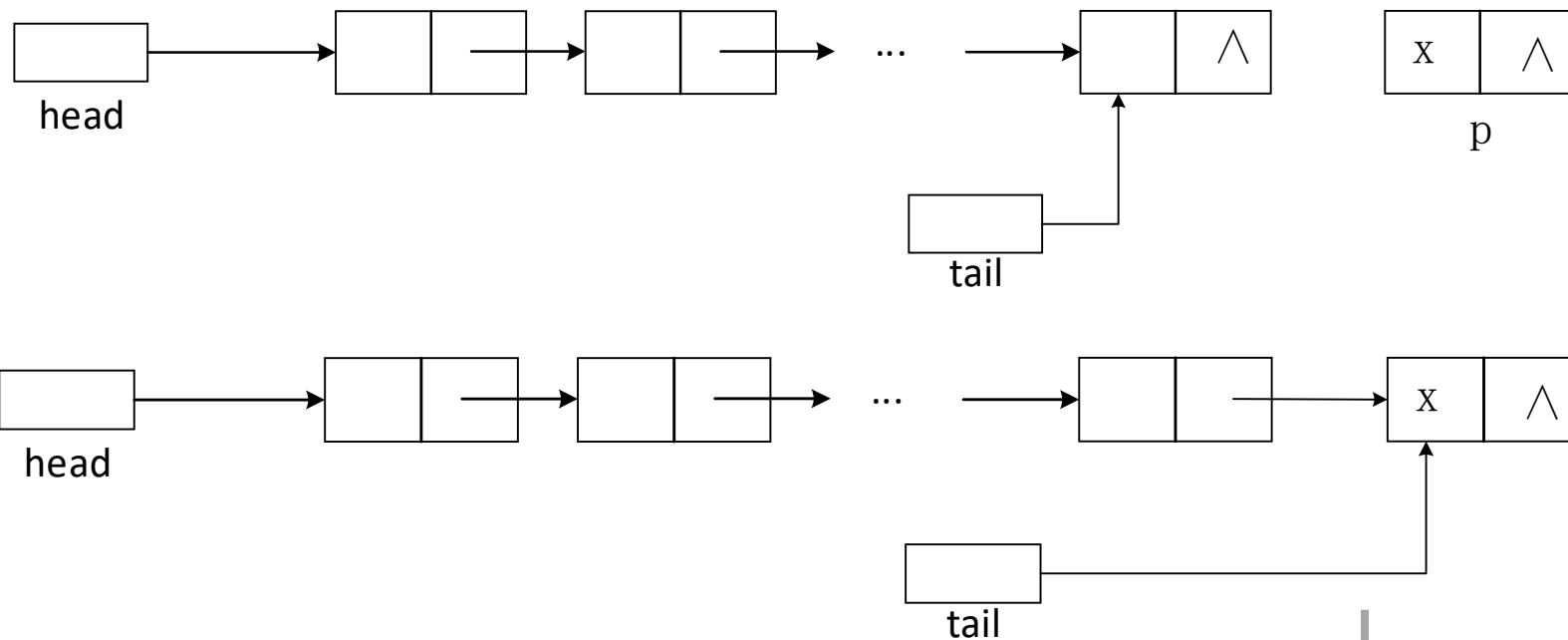


进队后:

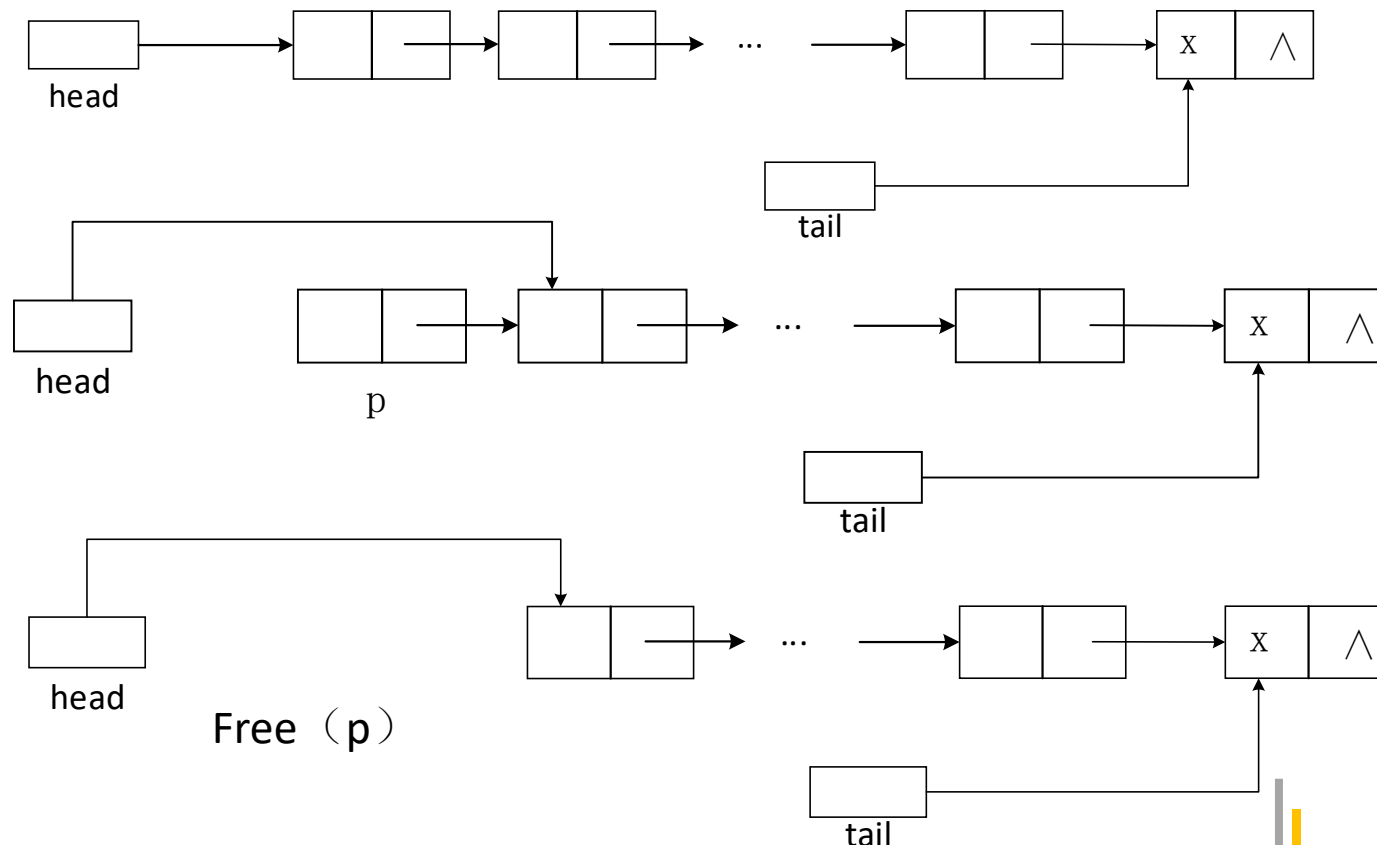


### 链接队列 进队和出队程序

#### 队列不空时进队



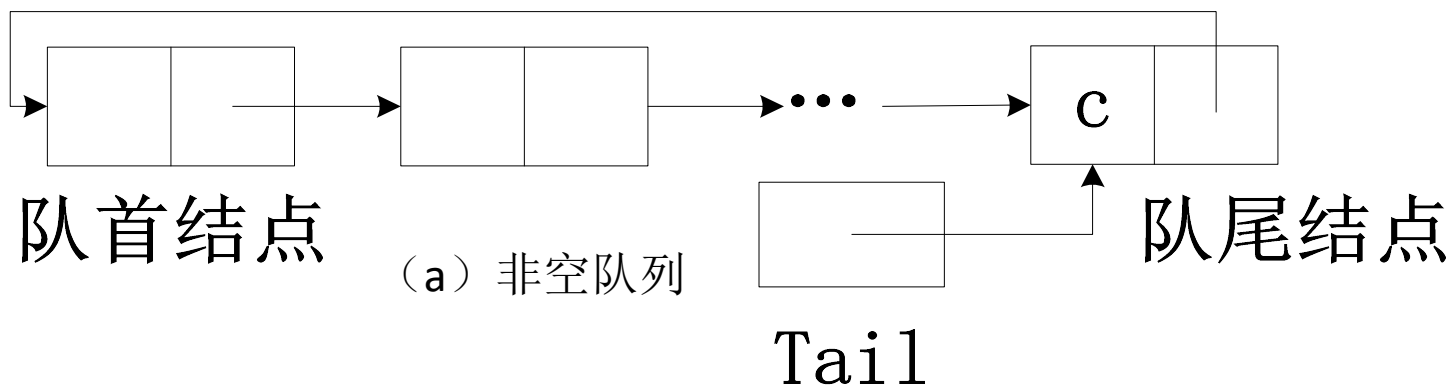
### 链接队列 进队和出队程序



### 环形链接队列

环形链接队列：让链接队列的队尾结点的队首链接指针指向队首结点，这样就构成一个环形链接队列。

通过尾指针tail可以很容易的找到队首结点，可以省去头指针。



空队列?

**If ( tail == NULL ) .....**

**压缩存储：节省存储空间。**

**索引存储：方便查找。**

**散列存储：方便从一个结点集合中找出具有给定键值的结点。**

**压缩存储：节省存储空间。**

**有部分结点含有相同的值 $v$ ，不存储这些含有相同值 $v$ 的结点，并断定不在结构中出现的结点的值为 $v$ ，这就是压缩存储的基本概念。**

**假设线性表 $F = (k_0, k_1, \dots, k_{n-1})$ 中，有若干个结点，它们的值都是 $v$ ，用结点 $k_i'$ 替换 $k_i$ ， $k_i'$ 的值含有两个部分，除 $k_i$ 原来值外，还包含 $k_i$ 在线性表 $F$ 中的序号 $i$ ，记作  $k_i' = (i, k_i)$ 。**

$F' = (k_0', k_1', \dots, k_{n-1}')$

**从 $F'$ 中删除所有 $(j, v)$ 的结点 $k_j'$ ，得到一个新的线性表 $F''$**

**任何一种存储 $F''$ 的方法都称为 $F$ 的压缩存储**

## 压缩存储

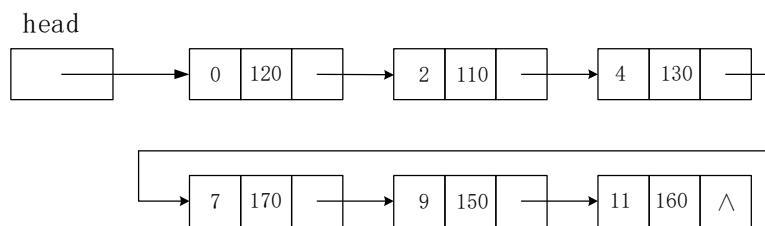
例: 对于含有若干个零的线性表

$F = (120, 0, 110, 0, 130, 0, 0, 170, 0, 150, 0, 160)$

可分别采用如下图所示的顺序压缩存储和链接压缩存储。

	i	ki
0	0	120
1	2	110
2	4	130
3	7	170
4	9	150
5	11	160

(a) 顺序压缩存储



(b) 链接压缩存储

图1.6.1 压缩存储



## 压缩存储

**查找第 $i$ 个结点的值：**对于链接压缩存储，几乎需要查遍所结点。  
对于顺序压缩，排序后可采用后面将介绍的二分查找法。

中

## 压缩存储

应用实例：设X，Y是具有10000个分量的向量，其中X中不为零的分量不到8%，我们要计算向量X和Y的内积。

$$sum = \sum_{i=1}^{9999} x_i y_i$$

中  
首先将X进行顺序压缩存储，然后依次输入Y中的元素，并算出sum之值。

示例代码段 P43-innerproduct

**索引存储，目的：方便查找。**

**用若干个子线性表代替线性表F，子线性表的全体正好是F的全部结点。**

**为了方便查找，另外设立一个索引线性表（简称索引表），它有m个结点，每个节点分别给出每个子线性表的首结点的地址。**

**如何把F中的结点划分成子线性表？**

**例：线性表F= (  $k_0, k_1, \dots, k_7$  ) 中各个结点的值如下：**

$k_0 = (18, \text{book})$	$k_1 = (25, \text{box})$	$k_2 = (92, \text{pen})$
$k_3 = (153, \text{paper})$	$k_4 = (198, \text{paper})$	$k_5 = (245, \text{pencil})$
$k_6 = (278, \text{rule})$	$k_7 = (300, \text{bag})$	

**其中第一个分量是关键词。**

索引存储，方便查找。

如何把F中的结点划分成子线性表？

k0= (18, book)	k1= (25, box)	k2= (92, pen)
k3= (153, paper)	k4= (198, paper)	k5= (245, pencil)
k6= (278, rule)	k7= (300, bag)	

如果取  $\text{index}(\text{key}) = \lfloor \text{key}/100 \rfloor$ , 则 F划分为:

索引函数：用它计算F中每个结点的索引。

F0= (18, book) , (25, box) , (92, pen)  
F1= (153, paper) , (198, paper)  
F2= (245, pencil) , (278, rule)  
F3= (300, bag)

例：查找键值为278的结点。则可以先计算  $\text{index}(278) = \lfloor 278/100 \rfloor = 2$   
只需要在F2 中查找。

## 索引存储

如何存储子线性表及索引表？

如果存储索引表用存储方法A，存储子线性表用存储方法B，我们就把这种索引存储方法称为“A-索引-B”

“顺序-索引- 链接”：子线性表使用链接存储，便于插入和删除操作；

### 索引存储

界限字段：子线性表 $F_i$ 中的任何一结点的键值都小于 $X_i$ 中的界限值。

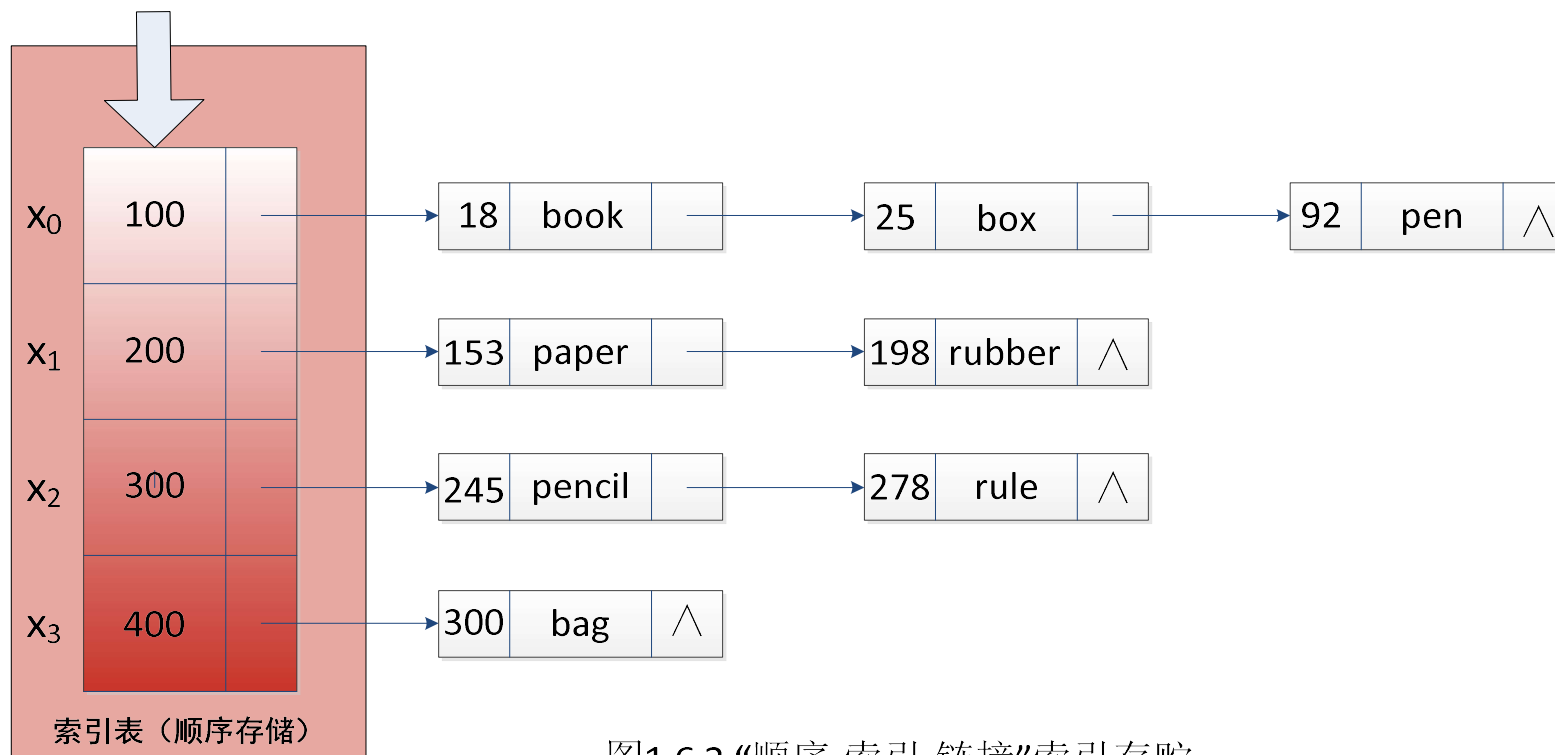


图1.6.2 “顺序-索引-链接”索引存储

**散列 (Hash) 存储——目的：加快查找**（从结点集合中找出具有给定键值的结点）

**散列存储是通过对结点的键值做某种运算来确定具有此键值的结点的存放位置。**

**设有线性表** $F = (k_0, k_1, \dots, k_{n-1})$ **和数组**  $T[m]$ **，结点** $k_i$ **的键值为** $key_i$ **，记** $F$ **中所有结点的键值的集合为** $S$ **。**

$h(x)$ ：**从** $S$ **到整数区间** $[0, m-1]$ **上的一个一一对应函数。**

$k_i$ **存放在数组** $T[m]$ **中的位置由**  $h(key_i)$  **决定。**

**这种存储方法称为散列存储。** $h(x)$  **称为** **散列函数**。  
**存放结点的数组** $T[m]$ **为散列表。**

**问题：** 查找效率的提高以什么为代价？

### 散列 (Hash) 存储

例：F是含有六个结点的线性表

k0= (9, E)

k1= (12, B)

k2= (20, C)

k3= (26, A)

k4= (34, D)

k5= (48, F)

取  $h(x) = x \bmod 10$  ( $x$ 为整数, 则值域区间在 $[0,9]$ )

使用能存放十个结点的数组T[10]作为Hash表。

存储情况如图。

如果想找key4=34 的结点,  
 $34 \bmod 10 = 4$ , 就能在数组元素T[4]中找到它。

存在什么风险?

线性表结点的键值K<sub>i</sub>

↓

0	20	C
1		
2	12	B
3		
4	34	D
5		
6	26	A
7		
8	48	F
9	9	E



### 散列 (Hash) 存储

一般来说，即使把存放结点的Hash表的容量取得很大，有时也难以找到一个足够简单的一一对应函数。

如果不一一对应，则会出现冲突。

因此，实在散列存储的两个主要问题：

- (1) 选取散列函数
- (2) 选取解决冲突的办法

## 线性表的查找

**假设：**具有 $n$ 个结点的线性表 $F = (k_0, k_1, \dots, k_{n-1})$ 的结点的键值为正整数。

**查找：**对给定的线性表和某一键值 $v$ ，从线性表中找一个键值为 $v$ 的结点。

**MAX (A)** = 为找到具有给定键值对结点所需要的比较次数的**最大值**。

**AVG (A)** = 为找到具有给定键值对结点所需要的比较次数的**平均值**。

如何求**AVG (A)** ?

设结点 $k_i$ 被查找的概率是 $p_i$ ， $\sum_{i=1}^n p_i = 1$

$P_i$ ：也叫做结点 $k_i$ 的相对使用概率。它表示在对给定线性表进行 $t$ 次查找中，查找结点 $k_i$ 的次数近似地等于 $t * P_i$

### 线性表的查找

如果用 $S_i$ 表示用方法A查找结点 $k_i$ 所需要的比较次数，那么有

$$\text{MAX}(A) = \max \{S_i \mid i = 0, 1, \dots, n-1\}$$

$$\text{AVG}(A) = \sum_{i=0}^{n-1} p_i S_i$$

### 线性表的查找

- 顺序查找法
- 二分查找法
- 分块查找法
- Hash查找法

### 线性表的查找——顺序查找法

把线性表中的 **结点的键值** 依次同 **给定的键值v** 作比较。如果找到所需结点，那么查找成功。如果整个表都查遍了仍未找到所需的结点，那么查找失败。

MAX（顺序查找法）=  $n$

$$\text{AVG（顺序查找法）} = \sum_{i=0}^{n-1} s_i p_i = \sum_{i=0}^{n-1} (i+1) p_i = \sum_{i=1}^n i p_{(i-1)}$$

可以看出，在使用顺序查找法时，把使用概率高的结点排在前面，可以降低平均比较次数。

即 把线性表中的结点按  $p_i \geq p_{i+1}$  ( $i=0, \dots, n-2$ ) 排序。

### 线性表的查找——顺序查找法

通常，可以假定所有的结点都具有相同的使用概率。此时可以得到：

$$\text{AVG (顺序查找法)} = \sum_{i=1}^n i \frac{1}{n} = \frac{1}{n} * \frac{n(n+1)}{2} = \frac{n+1}{2} \cong \frac{n}{2}$$

如果只知道相对使用概率相差很大，但不知道具体数值，如何优化？

在每次查找结点时，总是把找到的节点移近线性表的始端，而其他结点仍按原来的次序存放。这样，常用的结点将具有向线性表始端移动的趋势。

两种方法：

- 1) 在找到结点时把它移到线性表的最前面。
- 2) 把找到的结点向线性表的始端移动一个位置。（做一次交换）

**（自组织，自适应）**

## 线性表的查找——顺序查找法

**查找函数** 顺序存储的线性表: `seque1()`, `seque2()`.

链接存储的线性表: `seque3()`,

**代码示例: P47. 比较**`seque1()`, `seque2()` 的不同

### 线性表的查找——二分查找法

猜数游戏。

例：1~64之间的一个数。



### 线性表的查找——二分查找法

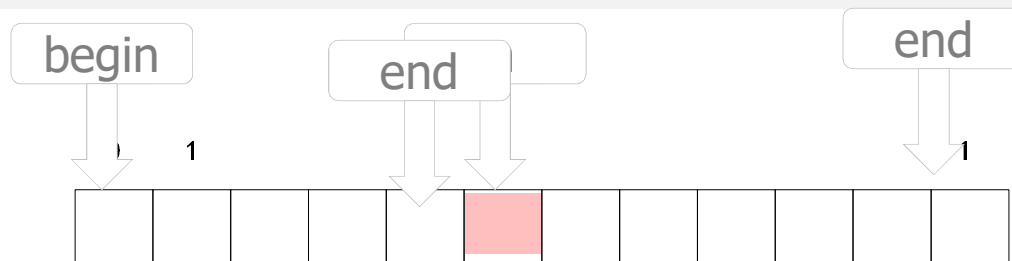
假设具有 $n$ 个结点的线性表**按键值的递增次序排好序**，且按照顺序存储方式把线性表中的结点依次存放在数组元素 $a[0]$ ,  $a[1]$ ,  $\dots$ ,  $a[n-1]$ 中。

算法如下：

- (1) 置 $i=0, j=n-1$ ;
- (2) 若 $i>j$ , 则查找失败, 算法结束; 否则, 转 (3) ;
- (3) 置 $m=(i+j)/2$ ;
- (4) 若 $v=a[m]$ , 则查找成功, 算法结束; 否则, 转 (5) ;
- (5) 若 $v<a[m]$ , 则置 $j=m-1$ , 转 (2) ; 否则, 置 $i=m+1$ , 转 (2)

## 线性表的查找——二分查找法

假设具有 $n$ 个结点的线性表**按键值的递增次序排好序**，且按照顺序存储方式把线性表中的结点依次存放在数组元素 $a[0]$ ,  $a[1]$ ,  $\dots$ ,  $a[n-1]$ 中。



Begin=0; end=n-1;

While ( \_\_\_\_\_ ) {

$m = (\text{begin} + \text{end}) / 2$ ;

    If  $V == a[m]$ , 查找成功, 算法结束 return(m)。

    Else if  $V < a[m]$  \_\_\_\_\_;

        else \_\_\_\_\_;

}

Return (-1) ;

### 线性表的查找——二分查找法

假设各结点的相对使用概率相同，并且只考虑  $n = 2^t - 1 (t \geq 0)$

计算需要执行平分比较的次数。平时的一次比较可产生三种结果。

如果  $t=1$ ，执行一次平分比较。否则， $m = 2^{t-1} - 1$ 。

如果  $k[m] == v$ ，则过程结束。

否则，继续对结点进行二分。

因此，最多平分  $t$  次必定结束。由此可得：

$$\text{MAX (二分查找法)} = t = \log_2(n + 1) \cong \log_2 n$$

另外，可以证明  $\text{AVG (二分查找法)} \cong \log_2 n$

### 线性表的查找——二分查找法

二分查找法要比顺序查找法快得多。但是，使用二分查找法时，线性表必须预先按键值排好序。而且还要求用顺序存储方法存储给定的线性表。

思考：适用的情况？

### 线性表的查找——分块查找法

假设线性表 $F$ 已按键值排好序，把 $F$ 分成 $m$ 个块： $F_0, F_1, \dots, F_{m-1}$ 。当  $i \leq j$  时， $F_i$ 中的结点都小于 $F_j$ 中的结点。

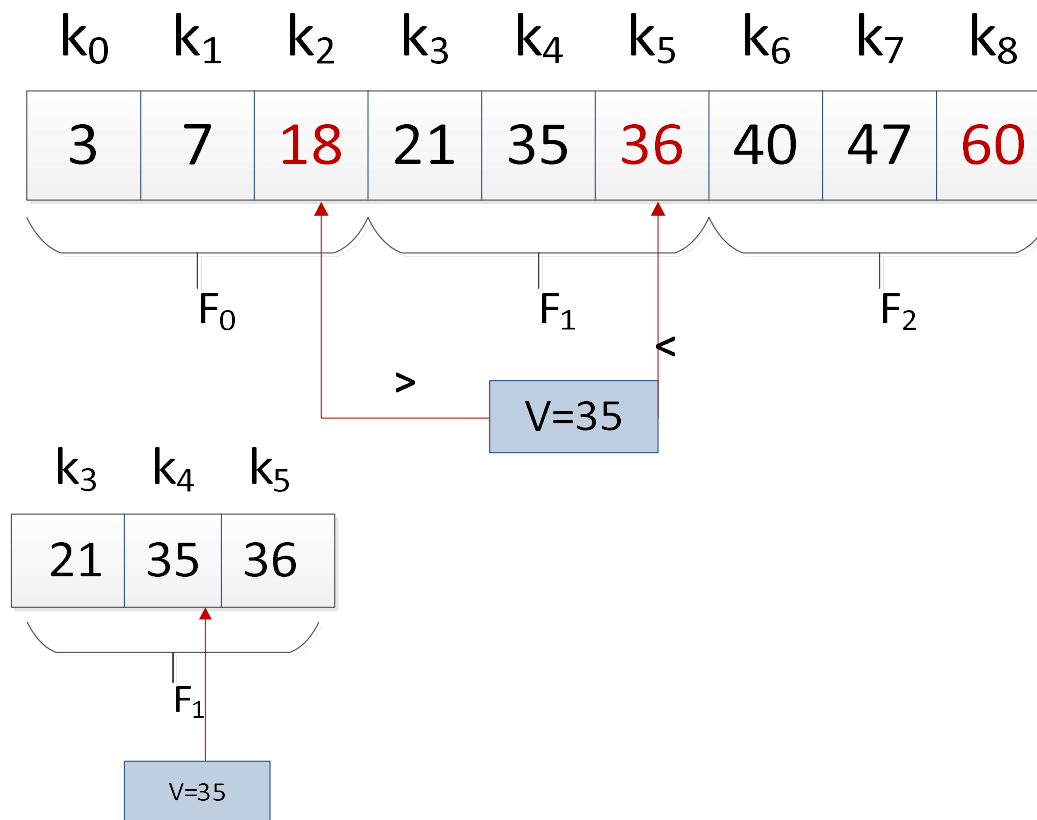
为了找出键值为 $v$ 的结点，依次将 $F_j$ 的最后一个节点的键值与 $v$ 比较。直到某个 $j$ 的最后结点 $k$ ，有 $v \leq \text{key}(k)$

这时可以推知要找到结点应落在  $F_j$ 中，然后用顺序查找法在 $F_j$ 中查找。

由于 $F_j$ 是被顺序查找的，给定的线性表 $F$ 可以不必完全排好序，只需要满足：当  $i \leq j$  时， $F_i$ 中的结点都小于 $F_j$ 中的结点，且提供每块中最大的键值的信息，以便确定要查找的结点在哪一块中。

适用的存储：即适用于顺序存储，也适用于链接存储。

### 线性表的查找——分块查找法



## 线性表的查找——分块查找法

效率分析（假定各节点的相对使用于概率相同）

假定要找的结点在第*i*块中，在查找时，先通过（*i*+1）次比较找到所在的块，在此块中做最多*n* 次的比较。因此

$$\text{MAX（分块查找法）} = \max\{(i+1) + n_i | i = 0, 1, \dots, m-1\}$$

$$\begin{aligned} \text{AVG（分块查找法）} &= \frac{1}{n} \sum_{k=0}^{n-1} S_k = \frac{1}{n} \sum_{i=0}^{m-1} \sum_{j=0}^{n_i-1} [(i+1) + (j+1)] \\ &= \frac{1}{n} \sum_{i=0}^{m-1} \sum_{j=1}^{n_i} [(i+1) + j] \end{aligned}$$

如果分块时用大小相同的块， $f=n/m$ ,  $n_0=n_1=\dots=f$

$$\text{MAX（分块查找法）} = m+f$$

$$\text{AVG（分块查找法）} = (m+f)/2$$

## 线性表的查找——Hash查找法

其他方法：通过键值的比较来确定被查找的键值的存放地址

Hash查找：通过键值做某种运算来确定键值的存放地址。

在Hash存储的基础上，需要解决两个问题：

- (一) Hash函数
- (二) 解决冲突



## 线性表的查找——Hash查找法

### (一) Hash函数

设有线性表  $F = (k_0, k_1, \dots, k_{n-1})$  和数组  $T[m]$ , 结点  $k_i$  的键值为  $key_i$ , 记  $F$  中所有结点的键值的集合为  $S$ 。

$h(x)$  是从  $S$  到整数区间  $[0, m-1]$  上的一个一一对应函数。

$k_i$  存放在数组  $T[m]$  中的位置由  $h(key_i)$  决定。

这种存储方法称为散列存储。 $h(x)$  称为散列函数。存放结点的数组  $T[m]$  为散列表。

### 线性表的查找——Hash查找法

#### (一) Hash函数

1. 数字分析法
2. 移位法
3. 平方取中法
4. 除法
5. 基数转换法

出现冲突，该如何处理？

.....

## 线性表的查找——Hash查找法

### （二）解决冲突的方法

1. 用开址寻址法解决冲突
2. 用拉链法解决冲突

## 线性表的查找——Hash查找法

### （二）解决冲突的方法——用开址寻址法解决冲突

Hash表初始化：把某个不作键值的值送到Hash表的每个位置上。

所选用的Hash函数是 $h(x)$ ，并用数组 $t[m]$ 作为Hash表，这个表有 $m$ 个位置。

Hash表的插入：为了把值 $k$ 存入Hash表，先计算出 $h(k) = i$ 。如果 $t[i]$ 是一个空位置，那么把 $k$ 存入 $t[i]$ ，插入过程结束；

如果 $t[i]$ 不是空位置，那么再判断 $t[i]$ 是否等于 $k$ ，如果 $t[i] = k$ ，则键值 $k$ 已在Hash表中，插入过程结束；否则， $t[i]$ 已被另一个键值所占用（发生冲突），此时必须为 $k$ 找另一个空位置。

办法：线性探索。

## 线性表的查找——Hash查找法

### （二）解决冲突的方法——用开址寻址法解决冲突

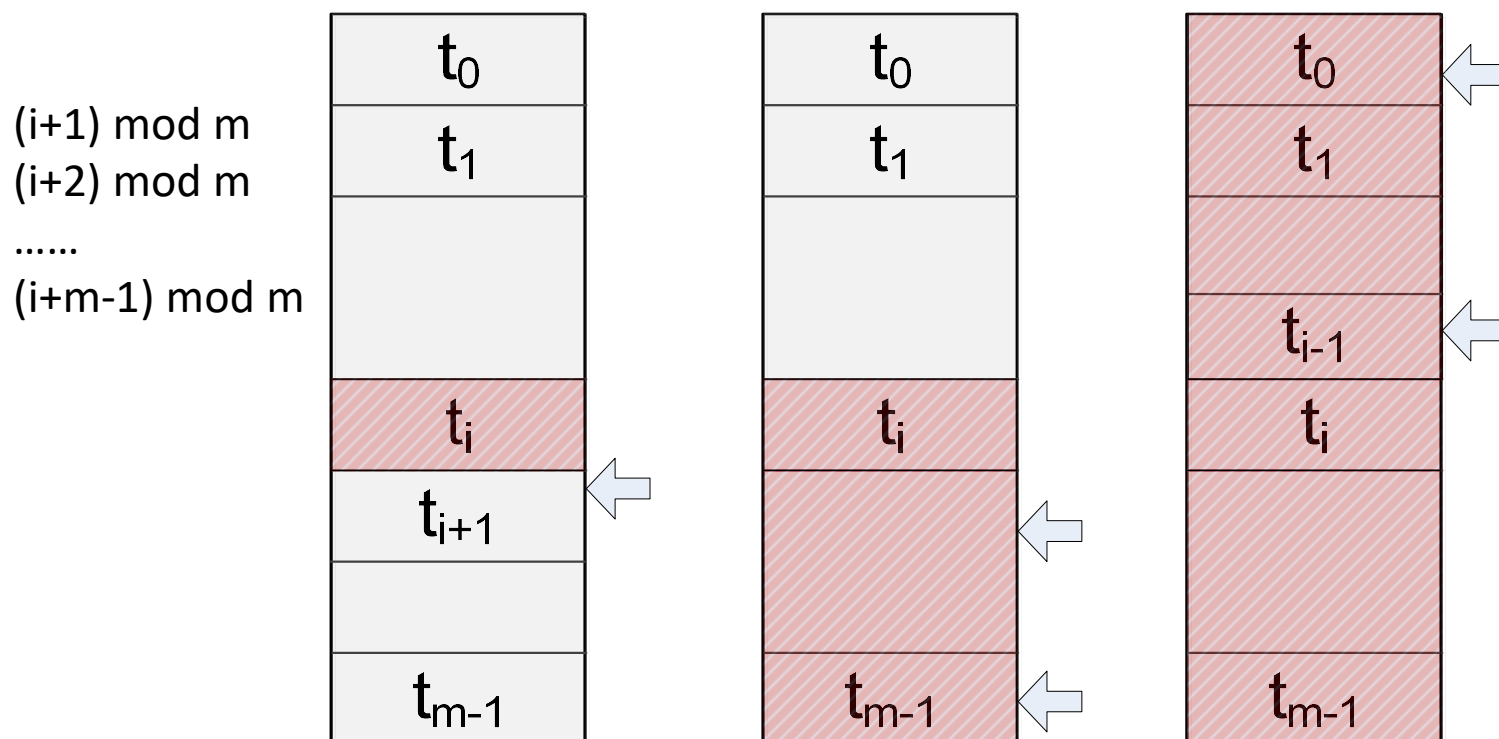
办法：线性探测

一旦找到一个空位置，就把k存入，插入过程结束。如果用完整个探测地址序列还没有找到空位置，那么Hash表满，插入失败。

问题：“探索”的范围？

### 线性表的查找——Hash查找法

#### (二) 解决冲突的方法——用开址寻址法解决冲突



## 线性表的查找——Hash查找法

### （二）解决冲突的方法——用开址寻址法解决冲突

办法：线性探测

查找：首先计算出 $i=h(k)$ 。如果 $t[i]=k$ ，则查找成功，查找过程结束。如果不等，那么必须循环探测地址序列进行查找。查找过程一直进行到下面三种情况之一出现为止。

- （1）当前位置上的键值等于 $k$ ，查找成功
- （2）找到一个空位置，查找失败
- （3）用完循环探测地址序列还没有找到 $k$ ，则查找失败。

## 线性表的查找——Hash查找法

(二) 解决冲突的方法——用开址寻址法解决冲突

办法：线性探测

查找：int search(int t[M], int k)

//t:存储空间, i:t的地址下标, j:向下探测的游标

```
i = hash(k);
```

```
For (j=0; j<M && t[(i+j)%M]!=k && t[(i+j)%M]!=0; j++) ;
```

```
i = (i+j)%M;
```

```
if (t[i]==k) return (i);
```

```
Return (-1);
```



### 线性表的查找——Hash查找法

#### (二) 解决冲突的方法——用开址寻址法解决冲突

删除：先找到，然后删除。

注意：不能把 $t[j]$ 置成空，而只能标上已被删除的标记，否则将会影响以后的查找。为什么？

因此，在插入时，凡遇到标有删除标记的位置，都可以插入。而在查找时，凡遇到标有删除标记的位置，还要继续查下去。

## 线性表的查找——Hash查找法

(二) 解决冲突的方法——用开址寻址法解决冲突

删除: `if (t[i]==k) t[i]=-1;` //不能置空, 而是设置为已删除

插入: `For (j=0; j<M && t[(i+j)%M]!=k && t[(i+j)%M]>0; j++ ) ;`

## 线性表的查找——Hash查找法

### （二）解决冲突的方法——用拉链法解决冲突

把具有相同Hash地址的键值都存放在同一个链表中，有 $m$ 个Hash地址就有 $M$ 个链表，同时用数组 $t[m]$ 存放各个链表的头指针。

## 第一章 线性表

### 7

## 线性表的查找

作业

1009 1010 1011

### 广义表的概念和存储结构

广义表的定义：广义表A是n个元素  $a_0, a_1, \dots, a_{n-1}$  的有限序列，记为  $A = (a_0, a_1, \dots, a_{n-1})$ 。其中A是广义表的名字，n是广义表的长度， $a_i$  ( $0 \leq i \leq n-1$ ) 是数据元素或者是广义表。

一个递归的定义

若  $a_i$  是数据元素，则称  $a_i$  为广义表A的原子；

若  $a_i$  是广义表，则称  $a_i$  为广义表A的子表。称  $a_0$  为广义表A的表头；称  $(a_1, a_2, \dots, a_{n-1})$  为广义表A的表尾，分别记为  $\text{head}(A) = a_0$ ， $\text{tail}(A) = (a_1, a_2, \dots, a_{n-1})$ 。

我们通常用大写字母表示广义表的名字，用小写字母表示广义表的原子。

## 广义表的概念和存储结构

几个例子

$D = ()$

$A = (a, (b, c))$

$B = (A, A, ())$

$C = (a, C) \quad // \text{ " 递归表" }$

## 广义表的概念和存储结构

广义表的存储：用链接的存储结构表示广义表。可以用如下结点形式表示广义表的一个元素。

Tag	Dlink/data	link
-----	------------	------

标识字段

指向下一个结点的指针

若没有下一个结点，则置link为空。

若结点表示一个原子，则tag=0，且第二个字段为data

若结点表示一个广义表，则tag=1，且第二个字段为dlink，是一个指向子表的指针。

## 广义表的概念和存储结构

广义表的存储：用链接的存储结构表示广义表。可以用如下结点形式表示广义表的一个元素。

```
struct node
{
    int tag;
    union
    {
        struct node * dlink;
        char data;
    } dd;
    struct node * link ;
};
typedef struct node NODE;
```