



北京航空航天大学
BEIHANG UNIVERSITY



数据结构与程序设计

(Data Structure and Programming)

数据结构

线性表

(Linear List)

北航计算机学院 晏海华



本章内容

- 2.1 线性表的基本概念
- 2.2 线性表的顺序存储结构
- 2.3 线性链表及其操作
- 2.4 循环链表及其操作
- 2.5 双向链表及其操作
- 2.6 链表应用举例

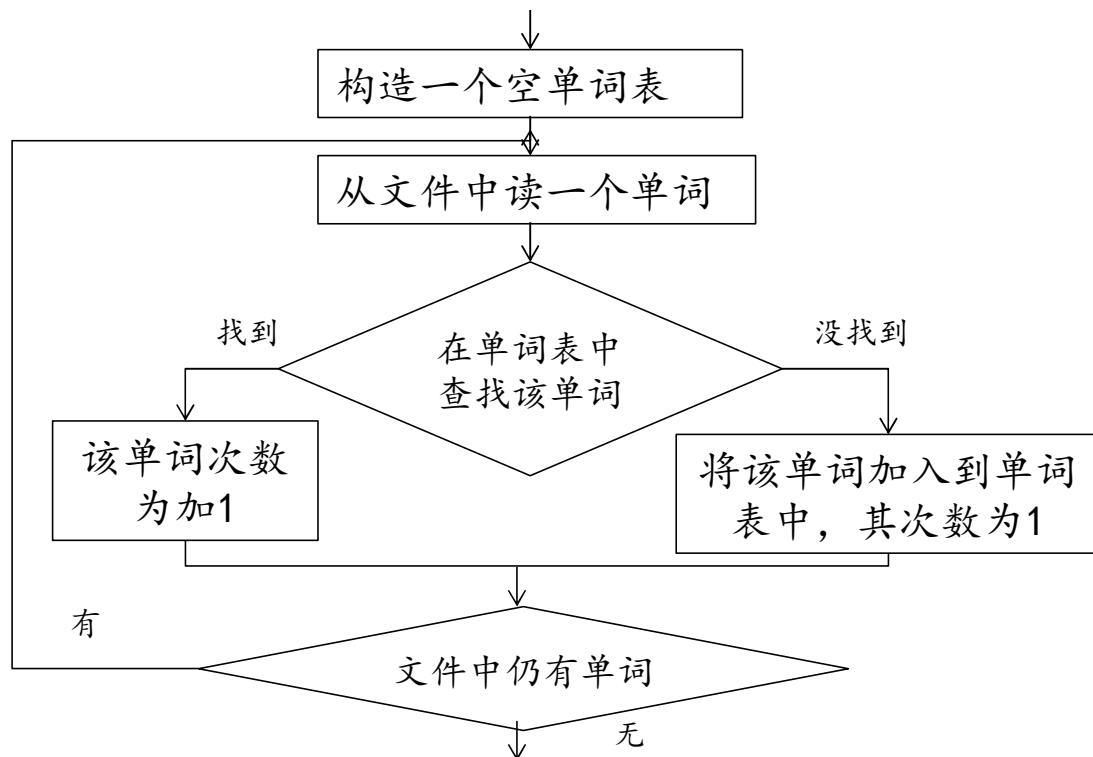
重点

链式存储结构



问题2.1：词频统计

- 问题：编写程序统计一个文本文件中每个单词的出现次数（词频统计），并按字典序输出每个单词及出现次数。
- 算法分析：本问题算法很简单，基本上只有查找和插入操作。



问题的关键是**单词表的构造和单词的组织方式**，它将影响算法的效率



问题2.1：词频统计

●用何种数据结构来构造和组织单词表？



用数组？链表？还是…？
来构造单词表
单词表是有序还是无序？

■不同数据结构构造的单词表如何影响着算法的性能？



当单词表较大时（如一本长篇小说），单词的查找和插入会面临什么问题？



2.1 线性表的基本概念

2.1.1 线性表的定义

$A = (a_1, a_2, a_3, \dots, a_n)$

1. 线性关系

(1) 当 $1 < i < n$ 时,
 a_i 的直接前驱为 a_{i-1} , a_i 的直接后继为 a_{i+1} 。

(2) 除了第一个元素与最后一个元素, 序列中任何一个元素有且仅有一个直接前驱元素, 有且仅有一个直接后继元素。

(3) 数据元素之间的先后顺序为 “一对一” 的关系。

	学 号	姓 名	性 别	年 龄	其 他
a_1	99001	张 华	女	17
a_2	99002	李 军	男	18
a_3	99003	王 明	男	17
\vdots
\vdots
\vdots
a_{50}	99050	刘 东	女	19



2. 线性表的定义

数据元素之间具有的逻辑关系为线性关系的数据元素集合称为线性表，数据元素的个数 n 为线性表的长度，长度为0的线性表称为空表。

线性表的特点：

- (1) 同一性
- (2) 有穷性
- (3) 有序性



几个线性表的例子

$$A = (a_1, a_2, a_3, \dots, a_n)$$

一个数据元素
为一个整数

① 数列: (25, 12, 78, 34, 100, 88)
 a_1 a_2 a_3 a_4 a_5 a_6

② 字母表: ('A', 'B', 'C',, 'Z')
 a_1 a_2 a_3 ... a_{26}

一个数据元素
为一个字母



3 数据表（文件）：

	学 号	姓 名	性 别	年 龄	其 他
a_1	99001	张 华	女	17
a_2	99002	李 军	男	18
a_3	99003	王 明	男	17

.
.
a_{50}	99050	刘 东	女	19

一个数据元素
为一个记录



2.1.2 线性表的基本操作

1. **创建**一个新的线性表。
2. 求线性表的长度。
3. **检索**线性表中第 i 个数据元素。 ($1 \leq i \leq n$)
4. 根据数据元素的某数据项(通常称为关键字)的值求该数据元素在线性表中的位置 (**查找**)。
5. 在线性表的第 i 个位置上**存入**一个新的数据元素。
6. 在线性表的第 i 个位置上**插入**一个新的数据元素。
7. **删除**线性表中第 i 个数据元素。
8. 对线性表中的数据元素按照某一个数据项的值的大小做升序或者降序**排序**。



9. 销毁一个线性表。
10. 复制一个线性表。
11. 按照一定的原则，将两个或两个以上的线性表合并成为一个线性表。
12. 按照一定的原则，将一个线性表分解为两个或两个以上的线性表。
-



2.1.2 线性表的基本操作

- `initList(nodeType *list, int length);` //创建一个空表
- `destroyList (nodeType *list, int length);` //销毁一个表
- `printList(nodeType *list, int length);` //输出一个表
- `getNode (nodeType *list, int pos);` //获取表中指定位置元素
- `searchNode(nodeType *list, Type node);` //在表中查找某一元素
- `insertNode(nodeType *list, int pos, nodeType node);` //在表中指定位置插入一个结点
- `deleteNode(nodeType *list, int pos);` //在表中指定位置删除一个结点



2.2 线性表的顺序存储结构

2.2.1 构造原理

用一组地址连续的存储单元依次存储线性表的数据元素，数据元素之间的逻辑关系通过数据元素的存储位置直接反映。

$(a_1, a_2, a_3, \dots, a_n)$

k个单元



$LOC(a_i)$

所谓一个元素的地址是指该元素占用的k个(连续的)存储单元的第一个单元的地址。



结论

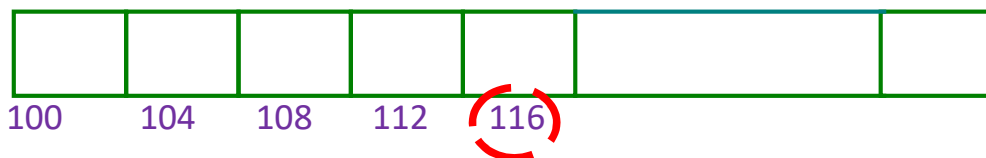
若假设每个数据元素占用 k 个存储单元，并且已知第一个元素的存储位置 $LOC(a_1)$ ，则有

$$LOC(a_i) = LOC(a_1) + (i-1) \times k$$

提问

例：

这个如此简单的公式，说明了顺序存储的线性表具有一个什么样的巨大优势呢？

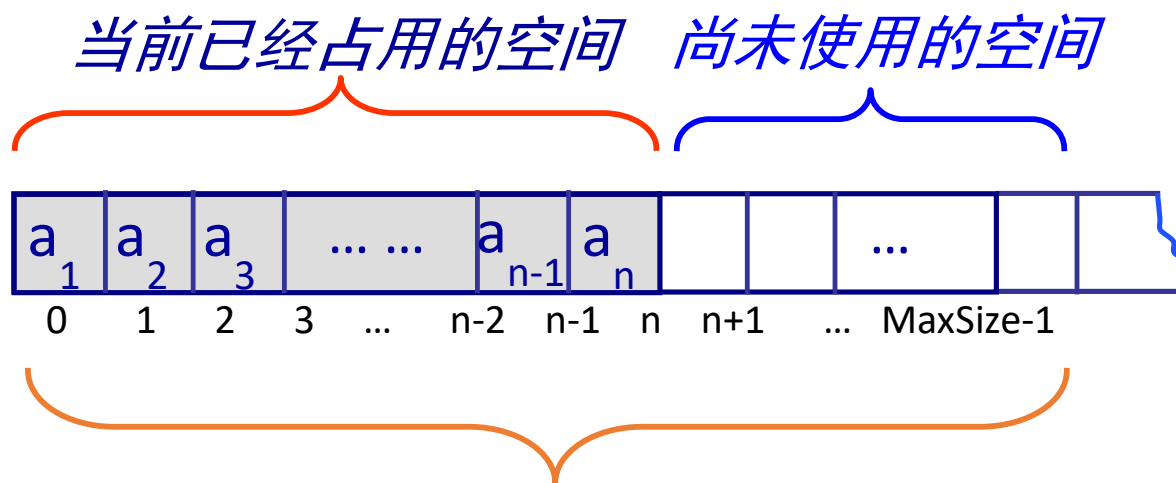


$$LOC(a_5) = 100 + (5-1) \times 4 = 116$$



顺序存储结构示意图

$(a_1, a_2, a_3, \dots, a_{n-1}, a_n)$



$n < \text{MaxSize}$

事先分配给线性表的空间



$(a_1, a_2, a_3, \dots, a_{n-1}, a_n)$

在C语言中

```
#define MaxSize 100  
ElemType A[MaxSize];  
int n;
```

预先分配给线性表的空间大小

表的长度

数组-顺序表



2.2.2 基本算法

1. 查找：确定元素item在长度为n的顺序表list中的位置

($a_1, a_2, a_3, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_{n-1}, a_n$)



算法

顺序查找法

```
int searchElem(ElemType list[ ], int n, ElemType item)
{
    int i;
    for(i=0; i<n; i++)
        if(list[i]==item)
            return i; /* 查找成功，返回在表中位置 */
    return -1;        /* 查找失败，返回信息-1 */
}
```

时间复杂度 $O(n)$



如何在有序顺序表中查找元素？

■ 折半查找算法（The binary search）：

在有序数据集中查找指定数据项（或数据项插入位置）最常用及效率高的算法是折半查找算法。
当数据集较大时，折半查找平均效率高。



折半查找算法 (binary search)

- 假设数据集按由小到大排列，折半查找算法的核心思想是：
 1. 将要查找的有序数据集的中间元素与指定数据项相比较；
 2. 如果指定数据项小于该中间元素，则将数据集的前半部分指定为要查找的数据集，然后转步骤1；
 3. 如果指定数据项大于该中间元素，则将数据集的后半部分指定为要查找的数据集，然后转步骤1；
 4. 如果指定数据项等于中间元素，则查找成功结束。
 5. 最后如果数据集中没有元素再可进行查找，则查找失败。

下面以在一个有序整型数据集中查找给定整数为例来说明折半查找原理。



折半查找算法（续）

例：在下面有序数据集中查找数据项62

0	1	2	3	4	5	6	7	8	9
5	7	16	24	25	50	45	50	62	65

↑ ← 查找范围 → ↑
low high

item > data[mid], 即 62 > 25

0	1	2	3	4	5	6	7	8	9
5	7	16	24	25	50	45	50	62	65

↑ ← 查找范围 → ↑
low high

item > data[mid], 即 62 > 50

0	1	2	3	4	5	6	7	8	9
5	7	16	24	25	50	45	50	62	65

↑ ← 查找范围 → ↑
low high

item = data[mid], 即 62 = 62

item = 62(查找项)

low = 0(查找范围开始)

high = 9(查找范围结束)

mid = (low+high)/2=4(查找范围中间)

low = mid+1=5

high = 9

mid = (low+high)/2=7

low = mid+1=8

high = 9

mid = (low+high)/2=8



折半查找算法（续）

有关查找算法
将在后面章节
作专门介绍

算法

在有序（递增）顺序表list中查找给定元素的折半查找算法如下：

```
int searchElem(ElemType list[ ], int n, ElemType item)
{
    int low=0, high=n-1, mid;
    while(low <= high){
        mid = (high + low) / 2;
        if(( item < list[mid])
            high = mid - 1;
        else if ( item > list[mid])
            low = mid + 1;
        else
            return (mid);
    }
    return -1;
}
```

时间复杂度 $O(\log_2 n)$

也就是说假如一个顺序表
有1024个元素，顺序查找
算法平均查找次数为512次
，而折半查找算法最多 只
须10次。



2. 插入：在长度为n的顺序表list的第i个位置上插入一个新的数据元素item

在线性表的第i-1个数据元素与第i个数据元素之间插入一个由符号item表示的数据元素，使得长度为n的线性表

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

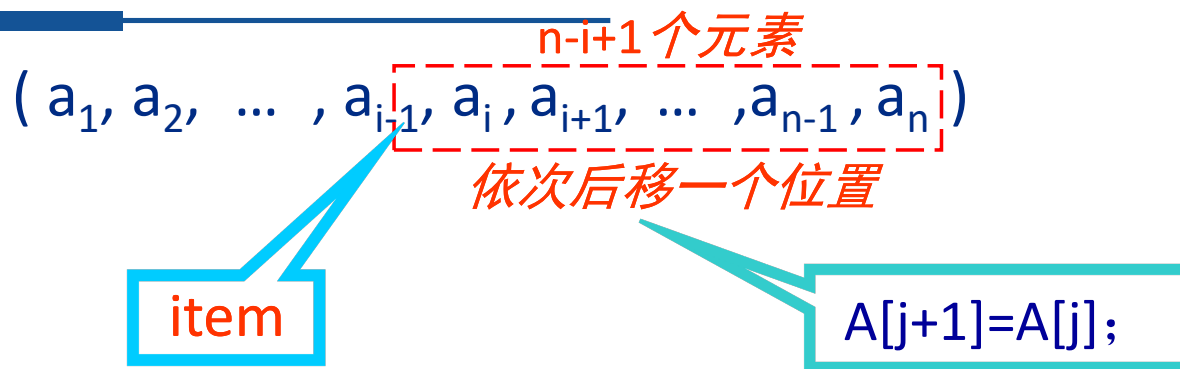
n个数据元素

转换成长度为n+1的线性表

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

n+1个数据元素





正常情况下需要做的工作：

- (1) 将第*i*个元素至第*n*个元素依次后移一个位置；
- (2) 将被插入元素插入表的第*i*个位置；
- (3) 修改表的长度（表长增1）。(*n*++;)

插入操作需要考虑的异常情况：

- (1) 是否表满？ $n = \text{MaxSize}$?
- (2) 插入位置是否合适？ (正常位置: $0 \leq i \leq n$)



约定

若插入成功，算法返回1，
否则，算法返回-1。



算法

/* 假设N是顺序表的长度（元素个数），为一个全局变量*/

int insertElem(ElemType list[], int i, ElemType item)

{

int k;

测试空间满否

if (N==MaxSize || i<0 || i>N)

return -1;

/* 插入失败 */

for(k=N-1; k>=i; k--)

list[k+1]=list[k];

/* 元素依次后移一个位置 */

list[i]=item;

/* 将item插入表的第i个位置 */

N++;

/* 线性表的长度加1 */

return 1;

/* 插入成功 */

}

测试插入位置合适否

该算法的时间复杂度是： **$O(n)$**



衡量插入和删除算法时间效率的另一个重要指标：
元素移动次数的平均值

若设 p_i 为插入一个元素于线性表第 i 个位置的概率 (概率相等)，则在长度为 n 的线性表中插入一个元素需要移动其他的元素的平均次数为

$$T_{is} = \sum_{i=1}^n p_i (n-i+1) = \sum_{i=1}^n (n-i+1) / (n+1) = n/2$$



3. 删除：删除长度为n的顺序表list的某个数据元素

把线性表的第i个数据元素从线性表中去掉，使得长度为n 的线性表

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

n个数据元素

转换成长度为 n-1 的线性表

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

n-1个数据元素





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

n-i个元素

依次前移一个位置

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

正常情况下需要做的工作：

- (1) 将删除元素的下一元素至第n个元素依次前移一个位置；
- (2) 修改表的长度(表长减1)。 (`n--;`)

删除操作需要考虑的异常情况：

- (1) 是否表空？ `n=0` ?
- (2) 删除位置是否合适？(正常位置：`0 ≤ i ≤ n-1`)



约定

若删除成功，算法返回1，
否则，算法返回-1。



算法

```
/* 假设N是表的长度（元素个数），为一个全局变量 */
int deleteElem( ElemType list[], int i )
{
    int k;
    if( N==0 || i<0 || i>N-1 )
        return -1;
    for( k=i+1; k<N; k++ )
        list[k-1]=list[k];
    N--;
    return 1;
}
```

测试表空和位置合适与否

/* 删除失败 */

/* 元素依次前移一个位置 */

/* 线性表的长度减1 */

/* 删除成功 */

该算法的时间复杂度是： $O(n)$



通常采用元素移动次数的
平均值作为衡量删除算法时间
效率的主要指标。

若 p_i 为删除线性表中第 i 个数据元素的概率
(设概率相等)，在长度为 n 的线性表中删除第 i
个数据元素需要移动其他的元素的平均次数为

$$T_{ds} = \sum_{i=1}^n p_i(n-i) = \sum_{i=1}^n (n-i)/n = (n-1)/2$$



2.2.3 顺序存储结构的特点

1. 优点

$$LOC(a_i) = LOC(a_1) + (i-1) \times k$$

- (1) 构造原理简单、直观，易理解。
- (2) 元素的存储地址可以通过一个简单的解析式计算出来。是一种顺序存储结构，读取速度快。
- (3) 由于只需存放数据元素本身的信息，而无其他空间开销，相对链式存储结构而言，存储空间开销小
- (4) 对于有序表，可使用折半查找等快速查找算法，查找效率高。

对于动态表（即需要频繁插入和删除操作的表）往往由于问题规模不知，如果采用顺序结构的话，需要事先分配很大的空间，造成空间浪费或空间不足。

2. 缺点

- (1) 存储分配需要事先进行。
- (2) 需要一块地址连续的存储空间。
- (3) 基本操作（如插入、删除）的时间效率较低。

需要频繁的移动数据

$O(n)$



例

已知长度为 n 的非空线性表 $list$ 采用顺序存储结构, 并且数据元素按值的大小非递减排列(有序), 写一算法, 在该线性表中插入一个数据元素 $item$, 使得线性表仍然保持按值非递减排列。

如何找到插入位置



$item$
 $a_1, a_2, a_3, \dots, \underline{a_i}, \underline{a_{i+1}}, \underline{a_{i+2}}, \dots, a_n$
↑ 依次后移一个位置

$$a_i \leq a_{i+1} \quad 1 \leq i \leq n-1$$

插入

```
for(j=n-1;j>=i;j--)  
    list[j+1]=list[j];  
list[i]=item;  
n++;
```




需要做的工作

1. 寻找插入位置：
从表的第一个元素开始进行比较，若有关系
 $item < a_i$
则找到插入位置为表的第*i*个位置。
2. 将第*i*个元素至第*n*个元素依次后移一个位置；
3. 将item插入表的第*i*个位置；
4. 表的长度增1。

例

1, 3, 5, 5, 8, 10, 13, 15, 20, 25

item=12

1, 3, 5, 5, 8, 10, 12, 13, 15, 20, 25



算法

$a_1, a_2, a_3, \dots, a_i, a_{i+1}, a_{i+2}, \dots, a_n, \text{item}$

↑ 特殊情况

/*假设 N是表长度，是一个全局变量*/

int insertElem(ElemType list[], ElemType item)

{

int i,j;

能处理吗？

if(N == MAXSIZE) return -1;

for(i=0; i<N&&item>=list[i]; i++)

/* 寻找item的合适位置 */

;

for(j=N-1; j>=i; j--)

list[j+1]=list[j];

确定插入位置

list[i]=item;

/* 将item插入表中 */

N++;

return 1;

表长加1

}

$O(n)$



主函数

```
#include <stdio.h>
#define MAXSIZE 1000
typedef int ElemType;
int N=0;
int main()
{
    int i;
    ElemType data,list[MAXSIZE];
    scanf("%d", &N);
    for(i=0; i< N; i++)
        scanf("%d", &list[i]);
    scanf("%d", &data);
    if(insertElem(list, data ) == 1)
        printf("OK\n");
    else
        printf("Fail\n");
    return 0;
}
```



算法

算法总复杂度为:

$O(\log_2 n + n)$

$= O(n)$

```
int insertElem(ElemType list[], ElemType item)
{
    int i=0,j;

    if (N == MAXSIZE) return -1;
    i = searchElem(list, item);

    for(j=N-1; j>=i; j--)
        list[j+1]=list[j];

    list[i]=item;
    N++;
    return 1;
}
```

/* 寻找item的合适位置 */

折半查找确定插入位置

/* 将item插入表中 */

折半查找算法如下:

```
int searchElem(ElemType list[], ElemType item)
{
    int low=0, high=n-1, mid;
    while(low <= high){
        mid = (high + low) / 2;
        if(( item < list[mid]))
            high = mid - 1;
        else if ( item > list[mid])
            low = mid + 1;
        else
            return (mid);
    }
    return low ;
}
```

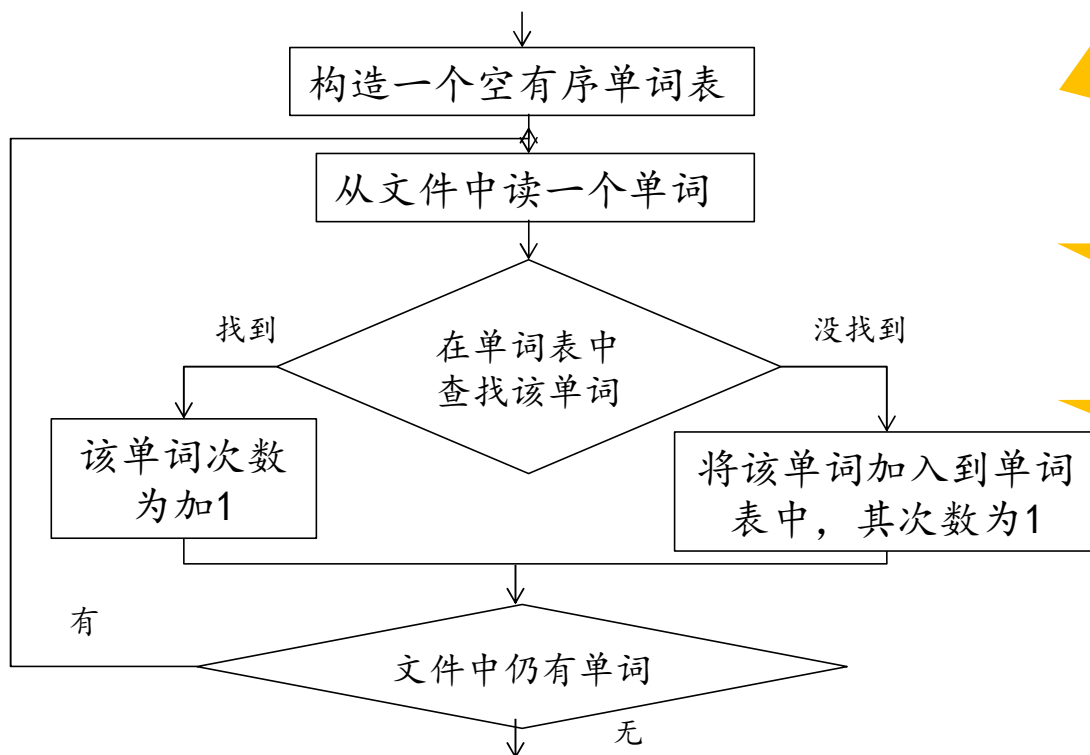


问题2.1：词频统计 – 顺序表

- 问题：编写程序统计一个文件中每个单词的出现次数（词频统计），并按字典序输出每个单词及出现次数。
- 算法分析：
 1. 首先构造一个空的有序（字典序）单词表；
 2. 每次从文件中读入一个单词；
 3. 在单词表中（折半）查找该单词，若找到，则单词次数加1，否则将该单词插入到单词表中相应位置，并设置出现次数为1；
 4. 重复步骤2，直到文件结束。



问题2.1：词频统计 – 顺序表



```
/*用数组构造一个顺序表，  
表中单词按字典序组织*/  
struct Inode {  
    char word[MAXWORD];  
    int count;  
};  
struct Inode  
wordlist[MAXSIZE];  
Int N=0; //全局变量，空表
```

用函数getWord从文件中每次
读入一个单词

用函数searchWord在单词表中
查找一个单词

用函数insertWord在单词表中
插入一个单词



外部（全局）变量

- 外部变量（全局变量，**global variable**）：在函数外面定义的变量。
 - 作用域（**scope**）为整个程序，即可在程序的所有函数中使用。
 - 外部变量有隐含初值0。
 - 生存期（**life cycle**）：外部变量（存储空间）在程序执行过程中始终存在（静态存储分配）。



外部变量说明（**extern**）

- C程序可以分别放在几个文件上，每个文件可作为一个编译单位分别编译。外部变量只需在某个文件上定义一次，其它文件若要引用此变量时，应用**extern**加以说明。（外部变量定义时不必加**extern**关键字。
- 在同一文件中，若前面的函数要引用后面定义的外部（在函数之外）变量时，也应在函数里加以**extern**说明。

```
/* t1.c */  
int N;  
main()  
{  
    ...  
    N = ...  
    ...  
}
```

```
/* t2.c */  
extern int N;  
fun()  
{  
    ...  
    N = ...  
    ...  
}
```

```
extern int N;  
main()  
{  
    ...  
    N = ...  
    ...  
}  
int N=0;  
void fun()  
{ ...
```




外部变量说明（extern）（续）

- 例如，对问题4.2的代码实现中，如果外部变量N不在程序头部定义，则需要用extern加以说明。

```
...  
extern int N;  
int main()  
{  
...  
}  
int N = 0;  
void insertData(int array[], int data)  
{  
...  
}
```

外部变量说明

外部变量定义



外部变量说明（extern）（续）*

- 使用外部变量的原因：
 - 解决函数单独编译的协调；
 - 与变量初始化有关；
 - 外部变量的值是永久的；
 - 解决数据共享；
- 外部变量的副作用：
 - 使用外部变量的函数独立性差，通常不能单独使用在其他的程序中。而且，如果多个函数都使用到某个外部变量，一旦出现差错，就很难发现问题是由哪个函数引起的。在程序中的某个部分引起外部变量的错误，很容易误以为是由另一部分引起的。

风格建议：在程序中应尽量少用或不用外部变量。



问题2.1：词频统计 – 代码实现

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAXWORD 32
#define MAXSIZE 1024
struct Inode {
    char word[MAXWORD];
    int count;
};
int getWord(FILE *bfp, char *w);
int searchWord(struct Inode list[], char *w);
int insertWord(struct Inode list[], int pos, char *w);
int N=0; //单词表中单词的实际个数
```

为什么要将**N**
定义为一个**全局变量**！

```
int main()
{
    struct Inode wordlist[MAXSIZE];    /*单词表*/
    int i;
    char filename[MAXWORD], word[MAXWORD];
    FILE *bfp;

    scanf("%s", filename);
    if((bfp = fopen(bname, "r")) == NULL){
        fprintf(stderr, "%s can't open!\n", filename);
        return -1;
    }
    while( getWord(bfp, word) != EOF)
        if(searchWord(wordlist, word) == -1) {
            fprintf(stderr, "Wordlist is full!\n");
            return -1;
        }
    for(i=0; i<= N-1; i++)
        printf("%s %d\n", wordlist[i].word,
wordlist[i].count);
    return 0;
}
```



问题2.1：词频统计



- 由于顺序表（数组）的大小需要事先确定，用顺序表作为单词表会有什么问题？

由于事先不知道被统计的文件大小（可能是本很厚的书），单词表的大小如何定：

- 1) 太小，对于大文件会造成单词表溢出；
- 2) 太大，对一般文件处理会造成很大的空间浪费。

- 词频统计需要在单词表中频繁的查找和插入单词，有序顺序表结构的单词表有什么特点？

- 1) 单词查找效率很高（可用折半查算法，一次查找算法复杂度为 $O(\log_2 n)$ ）；
- 2) 单词表中单词需要频繁移动（对于一般的英文材料来说，插入操作较多，一次插入算法的复杂度为 $O(n)$ ）；

- 无序（输入序）顺序表结构构造的单词表又有什么特点？

- 1) 单词查找效率低（顺序查找算法，一次查找算法复杂度为 $O(n)$ ）；
- 2) 单词不需要移动（新单词总是放在表尾），但需要对最终的总表要进行排序，简单排序算法的复杂度为 $O(N^2)$ ；



问题2.1：词频统计



- 请用**无序（输入序）**顺序表来重新实现问题2.1，并比较一下两个程序的实际性能。

即单词表中的单词按其读入顺序排放，新加入的单词总是放在表的末尾。

如何获取程序的
实际运行时间？



思考

已知顺序表中的每个数据元素为互不相等的整数。设计算法将其调整为两部分，使得左边所有元素为奇数，右边为偶数。分析算法的时间复杂度。



线性表顺序存储结构的优缺点

● 优点

- 构造原理简单（如用数组）
- 无须为表中元素之间的逻辑关系而增加额外的存储空间
- 可以快速地存取表中任一位置的元素。
- 可将数据组织成有序表，可用折半查找算法查找元素，查找效率高。

■ 缺点

- 存储空间需要事先分配（容易造成表空间不够或浪费）
- 需要一块地址连续的存储空间
- 插入和删除操作需要移动大量元素，效率低。

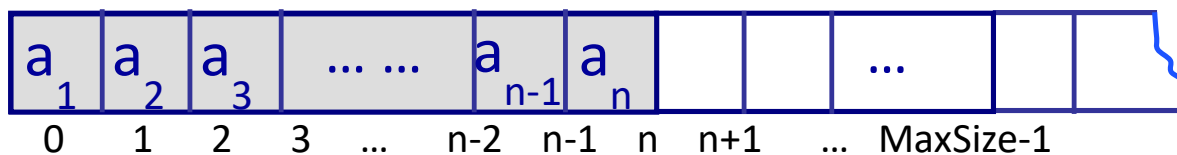


2.3 线性表的链式存储结构

将要讨论的内容

1. 线性链表的构造原理
2. 几个常用符号的说明
3. 线性链表的有关(操作)算法

顺序表



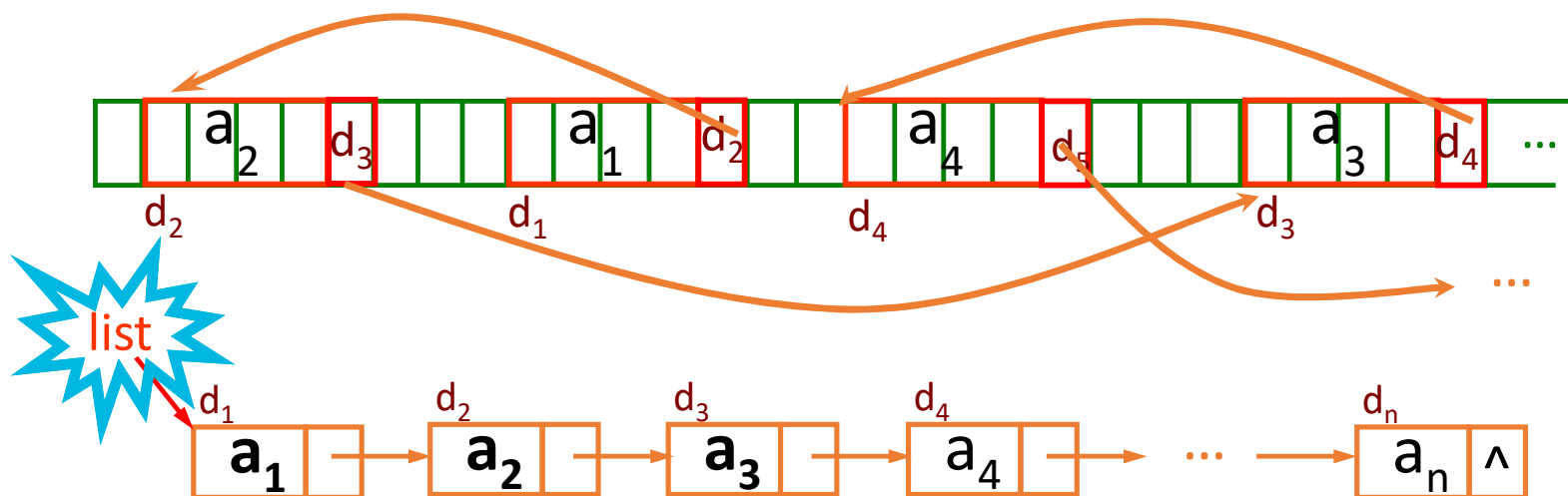
- (1) 一次分配内存
- (2) 需要连续空间
- (3) 插入和删除操作时效较低



一. 线性链表的构造原理

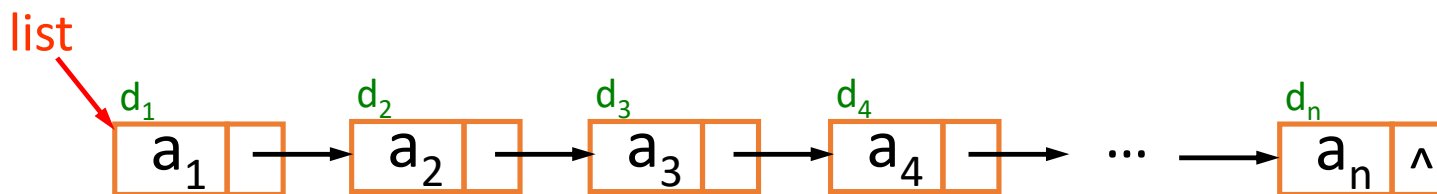
用一组地址任意的存储单元(连续的或不连续的)依次存储表中各个数据元素, 数据元素之间的逻辑关系通过**指针**间接地反映出来。

$(a_1, a_2, a_3, a_4, \dots, a_{n-1}, a_n)$





线性表的这种存储结构称为**线性链表**，或者**单(向)链表**，其一般形式为：



一个链结点

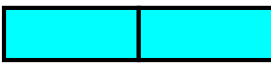
k个存储单元

数据域 指针域

data link



线性链表的定义

链结点: 
data link

链表定义

```
struct node {  
    ElemType data;  
    struct node *link;  
};  
struct node *list, *p;
```

类型定义

```
struct node {  
    ElemType data;  
    struct node *link;  
};  
typedef struct node  
*Nodeptr;  
typedef struct node Node;  
Nodeptr list, p;
```

	Num	Name	Age
a ₁	60101	张三	17
a ₂	60102	李四	16
a ₃	60103	王五	20
a ₃₀			



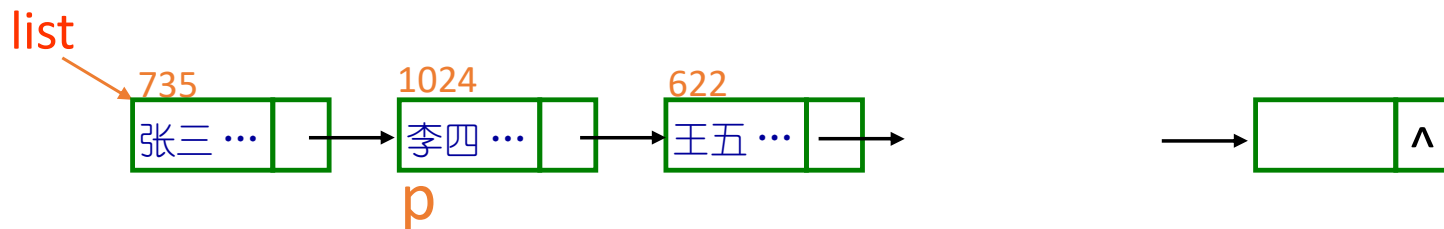
```
typedef struct {  
    int Num;  
    char Name[10];  
    int Age;  
} ElemType;
```



二. 链表结点的基本操作

若指针变量 p 为指向链表中某结点的指针(即 p 的内容为链表中某链结点的地址), 则

$p \rightarrow data$ 表示由 p 指向的链结点的数据域



$p \rightarrow link$ 表示由 p 指向的链结点的指针域, 即 p 所指的链结点的下一个链结点的指针(地址)。



2.3.3 链表的基本操作

- 求线性链表的长度。
- 建立一个线性链表。
- 在非空线性链表的第一个结点前插入一个数据信息为`item`的新结点。
- 在线性链表中由指针`q` 指出的结点之后插入一个数据信息为`item`的链结点。
- 在线性链表中满足某条件的结点后面插入一个数据信息为`item`的链结点。
- 从非空线性链表中删除链结点`q`(`q`为指向被删除链结点的指针)。

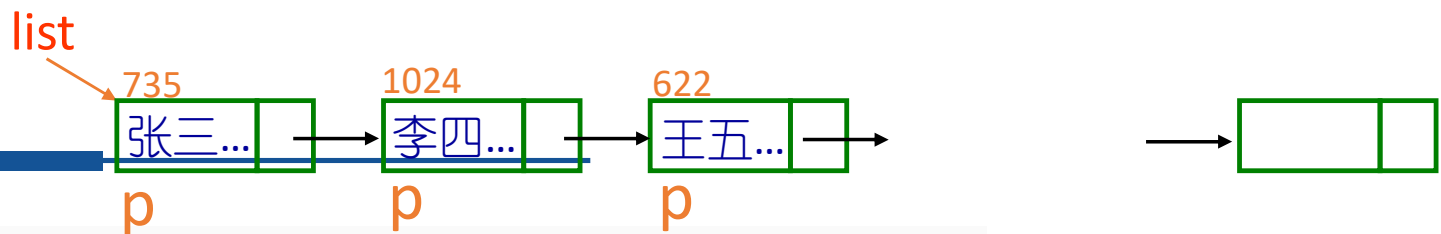


- 删除线性链表中满足某个条件的链结点。
- 线性链表的逆转。
- 将两个线性链表合并为一个线性链表。
- 检索线性链表中的第 i 个链结点。
-



链表的基本操作

- `createList(int n);` //创建一个具有n个结点的链表
- `getLength(Nodeptr list);` //获得链表的长度
- `destroyList (Nodeprt list);` //销毁一个表
- `printList(Nodeptr list);` //输出一个表
- `insertFirst (Nodeptr list, ElemType elem);` //在链表头插入一个元素
- `insertLast(Nodeptr list , ElemType elem);` //在链表尾插入一个元素
- `insertNode(Nodeptr list , Nodeptr p, ElemType elem);` //在链表某一结点后插入包含某一个元素的结点
- `searchNode(Nodeptr list , ElemType elem);` //在链表中查找某一元素
- `deleteNode(Nodeptr list , ElemType elem);` //在链表中删除包含某一元素结点



指向下一个结点:

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

插入一个结点:

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

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

(在p后插入q)

删除一个结点:

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

$p \rightarrow \text{link} = p \rightarrow \text{link} \rightarrow \text{link};$ 或 $p \rightarrow \text{link} = q \rightarrow \text{link};$

$\text{free}(q);$

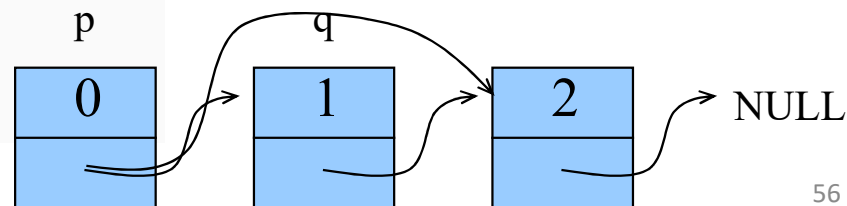
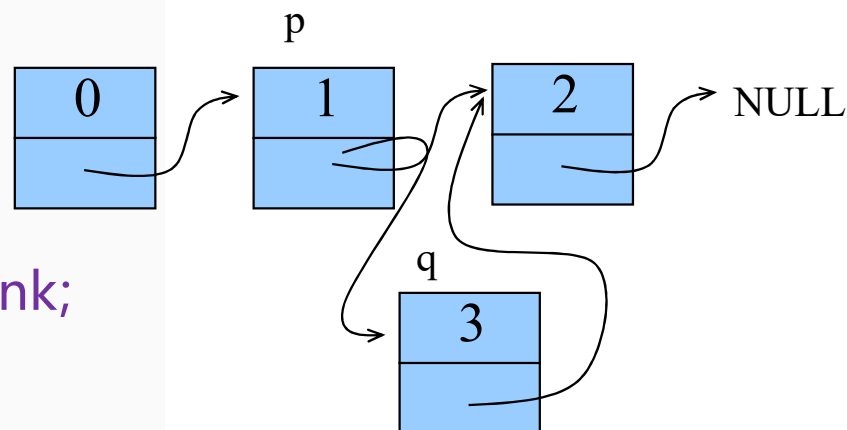
(删除p的下一结点)

遍历一个链表:

$\text{for}(p = \text{list}; p \neq \text{NULL}; p = p \rightarrow \text{link})$

....

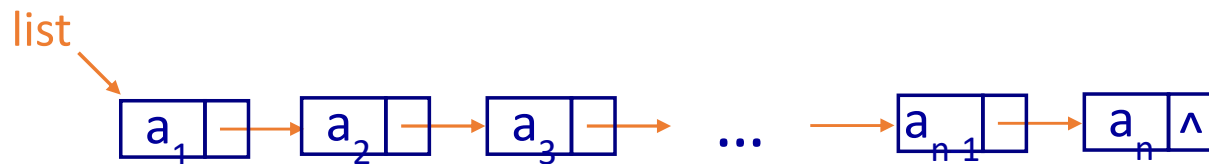
为什么首先要执行:
 $q = p \rightarrow \text{link};$





1. 建立一个线性链表

$(a_1, a_2, a_3, \dots, a_{n-1}, a_n)$



```
struct node{
    ElemType data;
    struct node *link;
};
typedef struct node *Nodeptr;
typedef struct node Node;
Nodeptr list, p;
```

申请一个链结点的空间

```
p=(Nodeptr)malloc(sizeof(Node));
```

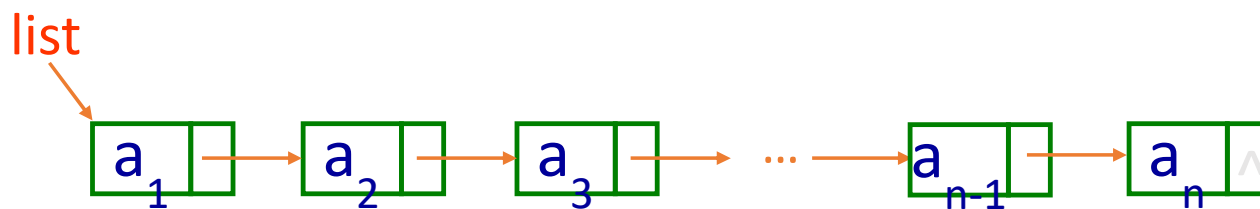
释放一个链结点的空间 `free(p);`

C语言

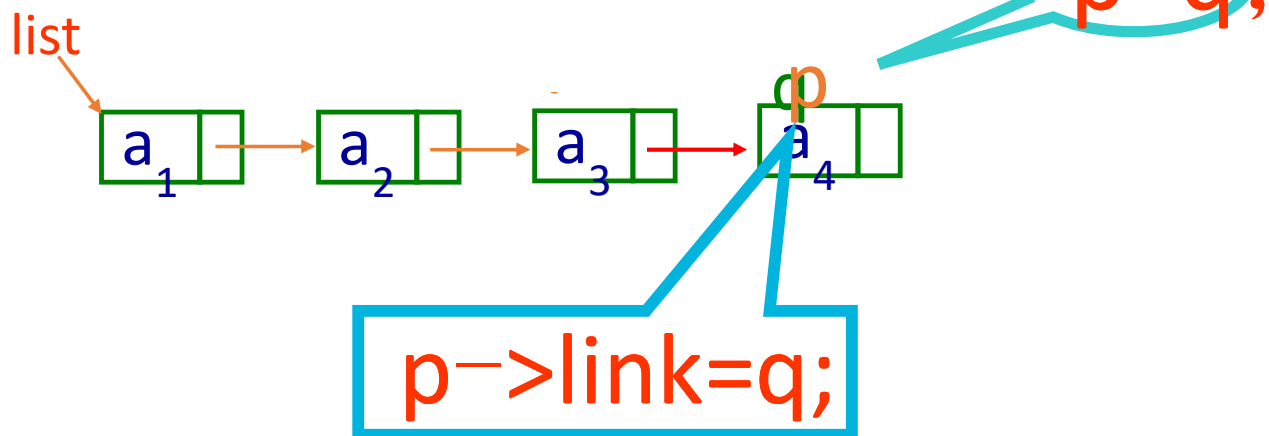
```
头文件: #include <stdlib.h>
```



$(a_1, a_2, a_3, a_4, \dots, a_{n-1}, a_n)$



一个结点的插入过程





算法

```
Nodeptr createList( int n ) /*创建一个具有n个结点的链表*/
{
    /* list是链表头指针, q指向新申请的结点, p指向最后一个结点*/
    Nodeptr p, q, list=NULL;
    int i;
    for(i=0;i<n;i++){
        q=(Nodeptr)malloc(sizeof(Node));
        q->data=read(); /* 取一个数据元素 */
        q->link=NULL;
        if (list==NULL) /*链表为空*/
            list=p=q;
        else
            p->link=q; /* 将新结点链接在链表尾部 */
        p=q;
    }
    return list;
}
```

申请一个新的链结点

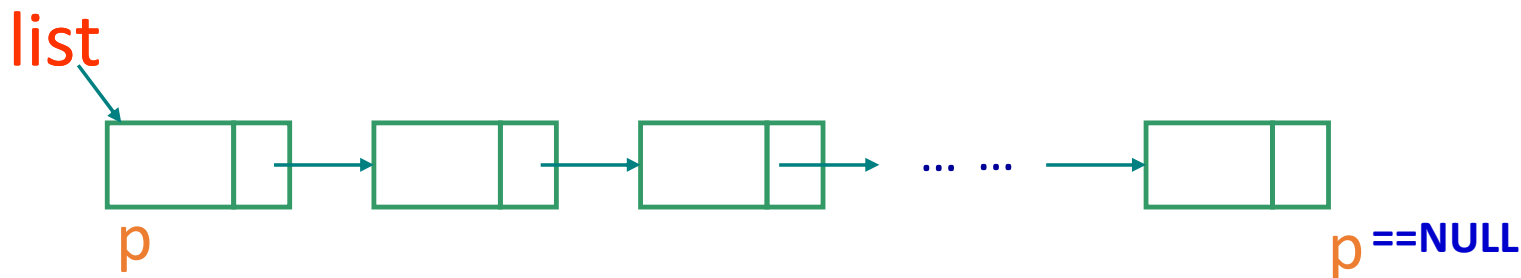
时间复杂度 $O(n)$



2. 求线性链表的长度



list 实际上就是一个指向链表头节点的指针。在实际使用时千万不要移动链表头结点指针！
(有什么后果?)



初始:

$n=0;$
 $p=list;$

$p=p->link;$

$n++;$

链表长度



算法

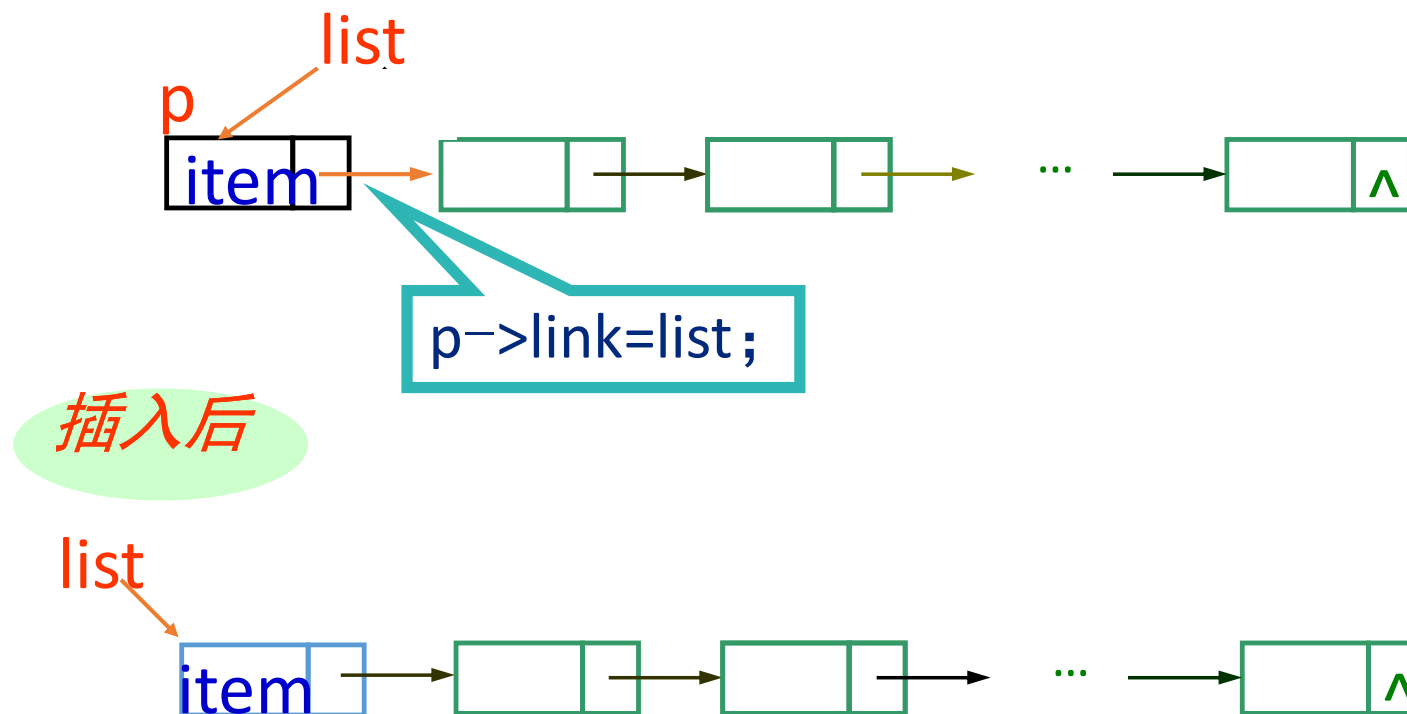
```
int getLength( Nodeptr list )    时间复杂度O(n)
{
    Nodeptr p;    /* p为遍历链表结点的指针 */
    int n=0;      /* 链表的长度置初值0 */
    for(p=list; p!=NULL; p=p->link)
        /* p依次指向链表的下一结点 */
        n++;      /* 对链表结点累计计数 */

    return n;      /* 返回链表的长度n */
}
```

$(a_1, a_2, a_3, \dots, a_{n-1}, a_n)$ 求n



3. 在非空线性链表的第一个结点前插入一个数据项为item的新结点





算法

```
Nodeptr insertFirst( Nodeptr list, ElemType item )
```

```
{
```

```
    /* list指向链表第一个链结点 */
```

```
    p=(Nodeptr)malloc(sizeof(Node));
```

```
    p->data=item;
```

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

```
    return p;
```

```
}
```

申请一个新结点

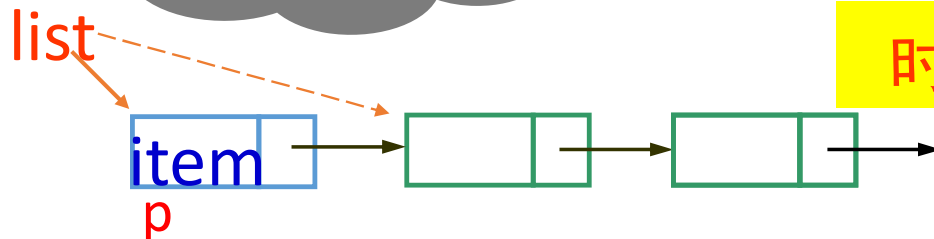
```
    /* 将item赋给新结点数据域 */
```

```
    /* 将新结点指向原链表第一个结点*/
```

```
    /* 将链表新头指针返回 */
```

请思考一下为什么
要返回p，而不用：
list = p?

应使用如下方式调用insertFirst函数：
list = insertFirst(list, item);

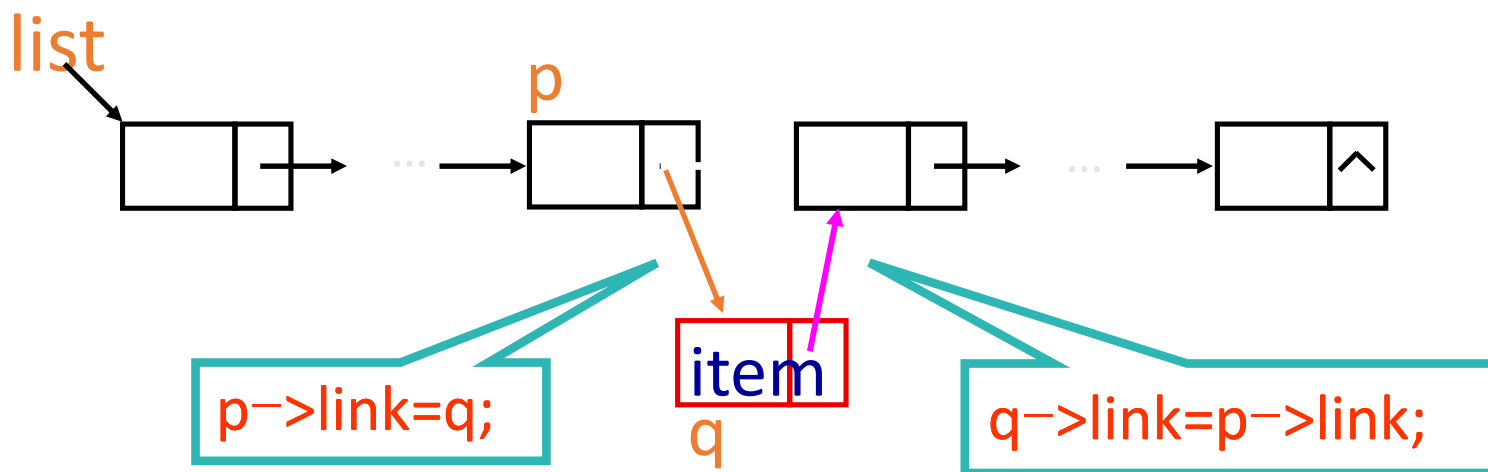


时间复杂度：O(1)



4. 在线性链表中由指针p指的链结点之后插入一个数据项为item 的链结点

插入过程





算法

```
void insertNode(Nodeptr p, ElemType item)
```

```
{
```

```
    Nodeptr q;
```

```
    q=(Nodeptr)malloc(sizeof(Node));
```

```
    q->data=item;
```

/* 将item送新结点数据域 */

```
    q->link = NULL;
```

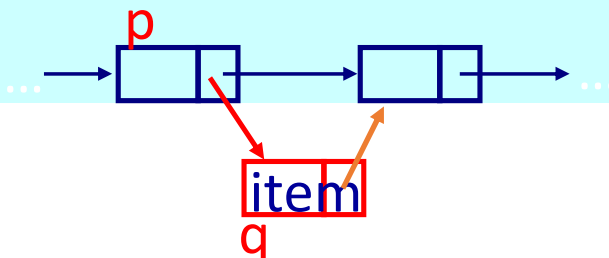
```
    q->link=p->link;
```

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

```
}
```

构造一个新结点

时间复杂度 $O(1)$

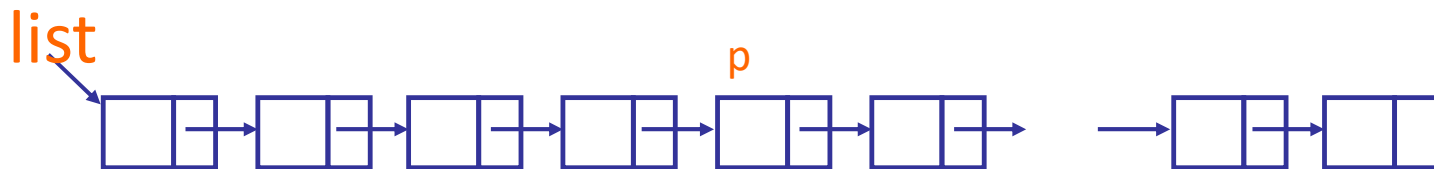




5. 在线性链表中第 $n(n>0)$ 个结点后面插入一个数据项为item的新结点

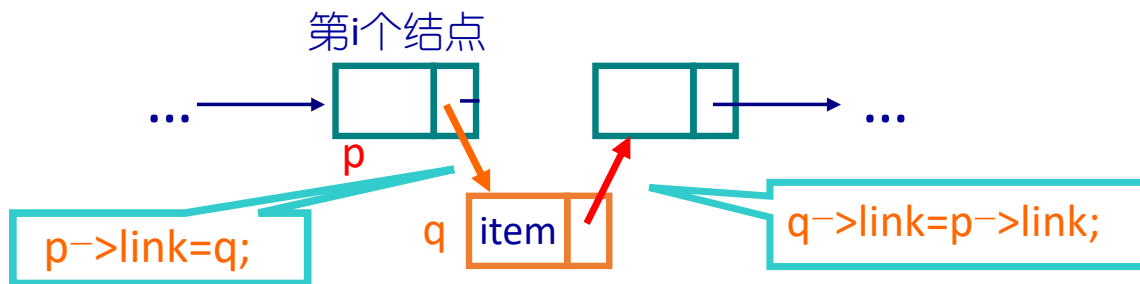
寻找第 n 个结点

如何找到第 i 个结点?



`p = p -> link;`

执行 $n-1$ 次!





```
void insertNode1( Nodeptr list, int n, ElemType item)
{
    Nodeptr p=list, q;
    int i;
    for(i=1;i<=n-1;i++){          /* 寻找第i个结点 */
        if(p->link==NULL)
            break;                /* 不存在第i个结点 */
        p=p->link;
    }

    q=(Nodeptr)malloc(sizeof(Node));
    q->data=item;                  /* 将item送新结点数据域 */
    q->link = NULL;

    q->link=p->link;
    p->link=q;                    /* 将新结点插入到第i个结点之后 */
}

}


```

构造一个新结点

时间复杂度 $O(n)$



5a . 在有序线性链表中相应位置上插入一个数据项为item的新结点



```
/* 设list是一个有序增序链表，将元素elem插入到相应位置上 */
Nodeptr insertNode(Nodeptr list, ElemType elem)
{
    Nodeptr p,q, r;
    r = (Nodeptr)malloc(sizeof(Node)); //创建一个数据项为elem的新结点
    r->elem = elem; r->link = NULL;
    if(list == NULL) /* list是一个空表 */
        return r;
    for(p=list; elem > p->elem && p != NULL; q = p, p = p->link) /* 找到插入位置 */
        ;
    if( p == list){ /* 在头结点前插入 */
        r->link = p;
        return r;
    }
    else { /* 在结点q后插入一个结点 */
        q->link = r;
        r->link = p;
    }
    return list;
}
```

时间复杂度 $O(n)$

1. 对链表进行插入操作时，一定要考虑下面特殊情况（头结点会发生改变）：1）链表为空；2）在头结点前插入。
2. 在结点p前插入一个结点，必须要知道该结点的前序结点指针，否则无法插入。在本程序中，q为p的前序结点指针；
3. 应使用如下方式调用insertNode函数：
`list = insertNode(list, item);`



(单向) 链表插入操作注意事项

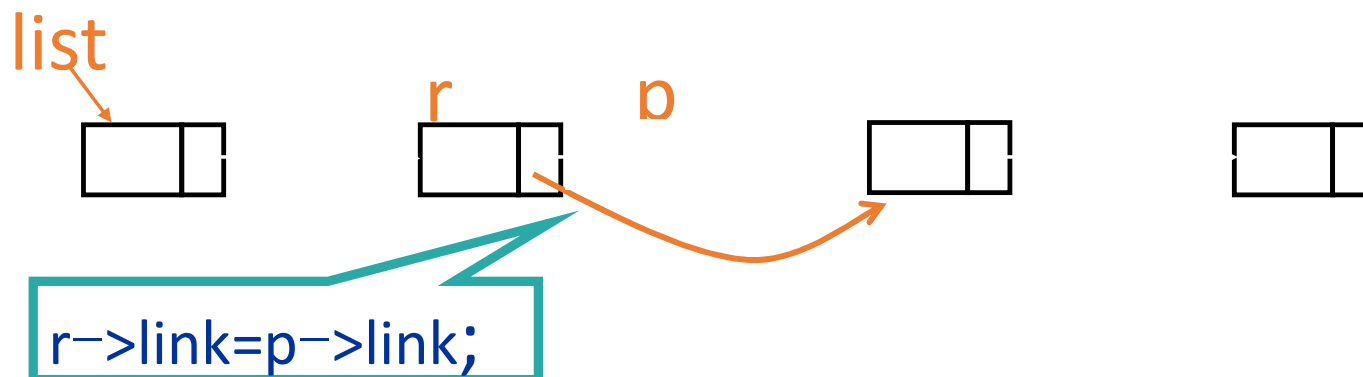
1. 对链表进行插入操作时，一定要考虑下面（头结点会发生改变）特殊情况：
1) 链表为空；2) 在头结点前插入。
2. 在某结点前插入一个结点，必须要知道该结点的前序结点指针，否则无法插入
3. 插入操作（函数）应返回头结点指针，如使用如下方式调用insertNode插入函数，以保确头结点指向正确的结点：
list = insertNode(list, item);



6. 从非空线性链表中删除p指向的链结点, 设p的直接前驱结点由r指出



情况1：删除链表的第一个结点

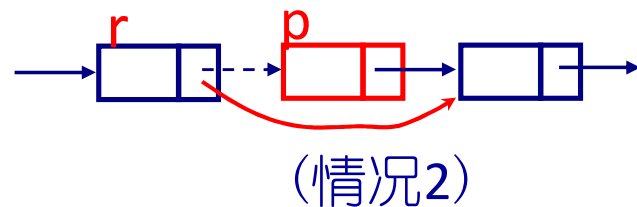
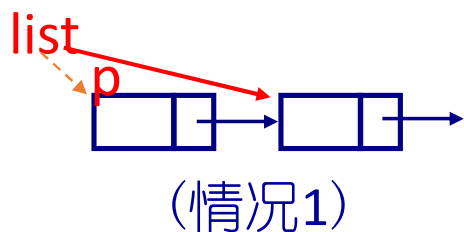


情况2：删除链表中非第一个结点



算法

```
Nodeptr deleteNode1( Nodeptr list, Nodeptr r, Nodeptr
p )
{
    if(p==list)
        list=p->link;      /* 删除链表的第一个链结点*/
    else
        r->link=p->link; /* 删除p指的链结点*/
    free(p);               /* 释放被删除的结点空间*/
    return list
}
```



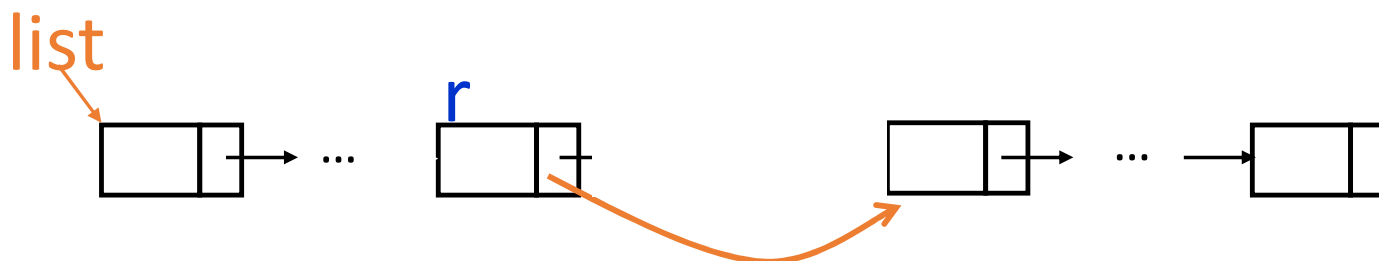
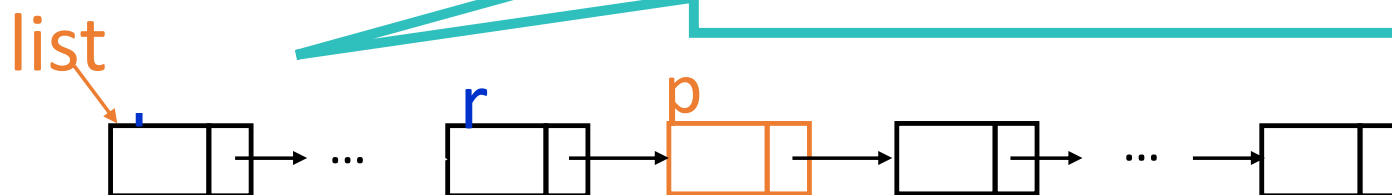
时间复杂度 $O(1)$



7. 从非空线性链表中删除p指向的链结点,

设r是p的直接前驱结点指针

```
for(r=list; r->link!=p; r=r->link)  
;
```





算法

```
Nodeptr deleteNode2( Nodeptr list, Nodeptr p )
{
    Nodeptr r;
    if(p==list){
        list=list->link;
        free(p);
    }
    else{
        for(r=list; r->link!=p && r->link!=NULL; r=r->link)
            ;
        if(r->link!=NULL){
            r->link=p->link;
            free(p);
        }
    }
    return list;
}
```

/*当删除链表第一个结点*/

/*释放被删除结点的空间*/

/*移向下一个链结点*/

时间复杂度 $O(n)$

寻找 p 结点的直接前驱 r



算法

从非空线性链表中删除包含给定元素的结点 一个更常见的操作

```
Nodeptr deleteNode( Nodeptr list, ElemType elem )
{
    Nodeptr p, q;  /*p指向要删除的结点, q为p的前一个结
点*/

    for(p=list; p!=NULL; q=p, p=p->link)
        if(p->elem==elem) /*找到要删除的链结点*/
            break;
    if(p==list) { /* 删除头结点*/
        list = list->link;
        free(p);
    }
    if(q->link != NULL) { /* 删除p指向的结点*/
        q->link = p->link;
        free(p);
    }
    return list;
}
```

思考?
如果要删除的元素不存在，算法会怎么样？

时间复杂度 $O(n)$



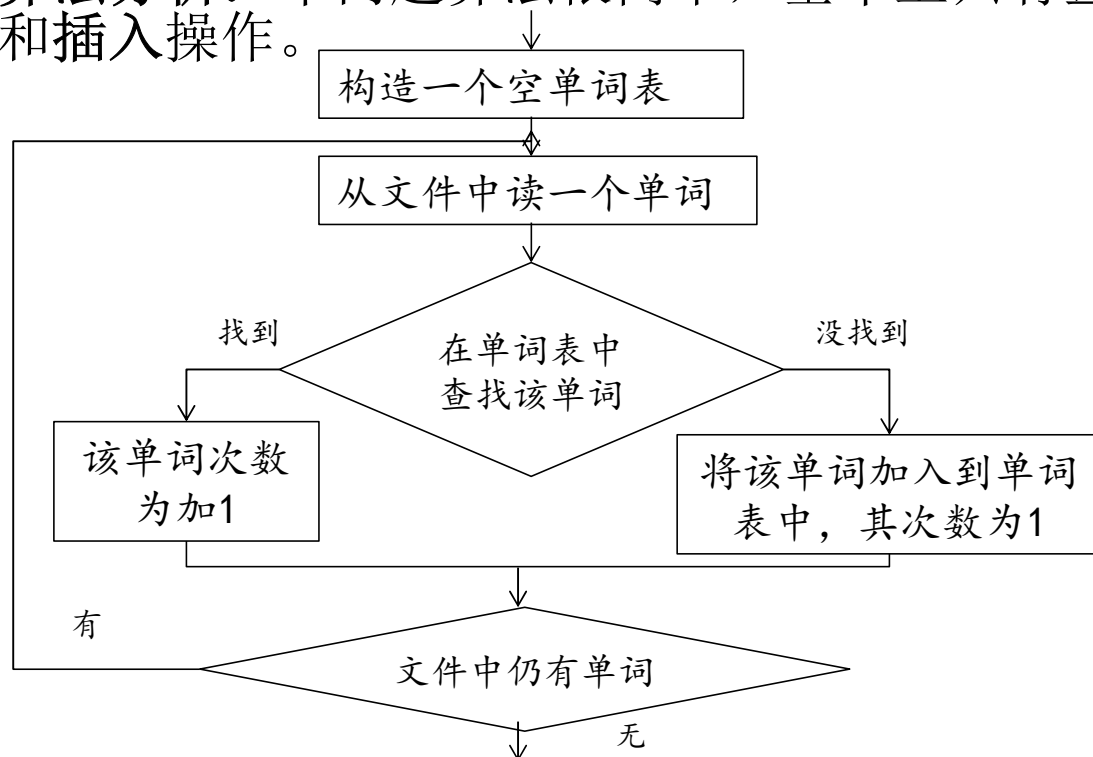
(单向) 链表删除操作注意事项

1. 对链表进行删除操作时，一定要考虑下面特殊情况：1) 链表为空；2) 删除头结点。
2. 在删除某个结点时，必须要知道该结点的前序结点指针，否则无法删除。
3. 结点删除后一定要释放。删除某个结点前，必须要事先保存指向该结点的指针，以便删除后能释放结点。
4. 删除操作（函数）应返回头结点指针，如使用如下方式调用deleteNode删除函数，以保确头结点指向正确的结点：
list = deleteNode(list, item);



问题2.1：词频统计 – 链表

- 问题：编写程序统计一个文件中每个单词的出现次数（词频统计），并按字典序输出每个单词及出现次数。
- 算法分析：本问题算法很简单，基本上只有查找和插入操作。





问题2.1：词频统计 – 链表

由于本问题有如下特点：

1. 问题规模不知（即需要统计的单词数量未知）
2. 单词表需要频繁的执行插入操作

因此，采用顺序表（数组）来构造单词表面临如下问题：

1. 单词表长度太小，容易满，太大，空间浪费
2. 插入操作效率低（经常需要移动大量数据）

而链表具有动态申请结点（空间利用率高），插入和删除结点操作不需要移动结点，插入和删除算法效率高！但单词查找效率低（不能用折半等高效查找算法）



问题2.1：词频统计 – 链表（代码实现）

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAXWORD 32
struct node {
    char word[MAXWORD];
    int count;
    struct node *link;
}; //单词表结构
struct node *Wordlist = NULL; //单词表头指针
int getWord(FILE *bfp, char *w);
int searchWord(char *w);
int insertWord( struct node *p, char *w);
```

Wordlist是一个全局变量！

```
int main()
{
    char filename[32], word[MAXWORD];
    FILE *bfp;
    struct node *p;

    scanf("%s", filename);
    if((bfp = fopen(filename, "r")) == NULL){ //打开一个文件
        fprintf(stderr, "%s can't open!\n", filename);
        return -1;
    }
    while( getWord(bfp, word) != EOF) //从文件中读入一个单词
        if(searchWord(word) == -1) { //在单词表中查找插入单词
            fprintf(stderr, "Memory is full!\n");
            return -1;
        }
    for(p=Wordlist; p != NULL; p=p->link) //遍历输出单词表
        printf("%s %d\n", p->word, p->count);
    return 0;
}
```



问题2.1：词频统计 – 链表（代码实现） *

/*在链表中p结点后插入包含给定单词的结点，同时置次数为1*/

```
int insertWord(struct node *p, char *w)
{
    struct node *q;

    q = (struct node *)malloc(sizeof(struct node));
    if(q == NULL) return -1; //没有内存空间
    strcpy(q->word, w);
    q->count = 1;
    q->link = NULL;
    if(Wordlist == NULL) //空链表
        Wordlist = q;
    else if (p == NULL){ //插入到头结点前
        q->link = Wordlist;
        Wordlist = q;
    }
    else {
        q->link = p->link;
        p->link = q;
    }
    return 0;
}
```

在本程序中：
查找算法复杂度为 $O(n)$
插入算法复杂度为 $O(1)$

/*在链表中查找一单词，若找到，则次数加1；
否则将该单词插入到有序表中相应位置，同
时次数置1*/

```
int searchWord(char *w)
{
    struct node *p, *q=NULL; //q为p的前序结  
点指针
    for(p=Wordlist; p != NULL; q=p, p=p->link){
        if(strcmp(w, p->word) < 0)
            break;
        else if(strcmp(w, p->word) == 0){
            p->count++;
            return 0;
        }
    }
    return insertWord(q, w);
}
```



问题2.1：词频统计 – 链表



●采用链表方式构造单词表具有如下特点：

➤优点

- ✓ 由于采用动态申请结点，能够适应不同规模的问题，空间利用率高
- ✓ 算法简单，插入操作效率高

➤不足

- ✓ 由于采用顺序查找，单词查找效率低

还有更好的单词表的构造及查询方法吗？



问题2.2：多项式相加（链表实现）*

【问题描述】编写一个程序实现任意（最高指数为任意正整数）两个一元多项式相加。

【输入形式】从标准输入中读入两行以空格分隔的整数，每一行代表一个多项式，且该多项式中各项的系数均为0或正整数。对于多项式 $a^n x^n + a^{n-1} x^{n-1} + \dots + a^1 x^1 + a^0 x^0$ 的输入方法如下： $a^n \ n \ a^{n-1} \ n-1 \ \dots \ a^1 \ 1 \ a^0 \ 0$

即相邻两个整数分别表示表达式中一项的系数和指数。在输入中只出现系数不为0的项。

【输出形式】将运算结果输出到屏幕。将系数不为0的项按指数从高到低的顺序输出，每次输出其系数和指数，均以一个空格分隔。最后要求换行。

【样例输入】

54 8 2 6 7 3 25 1 78 0

43 7 4 2 8 1

【样例输出】

54 8 43 7 2 6 7 3 4 2 33 1 78 0

【样例说明】输入的两行分别代表如下表达式：

$54x^8 + 2x^6 + 7x^3 + 25x + 78$

$43x^7 + 4x^2 + 8x$

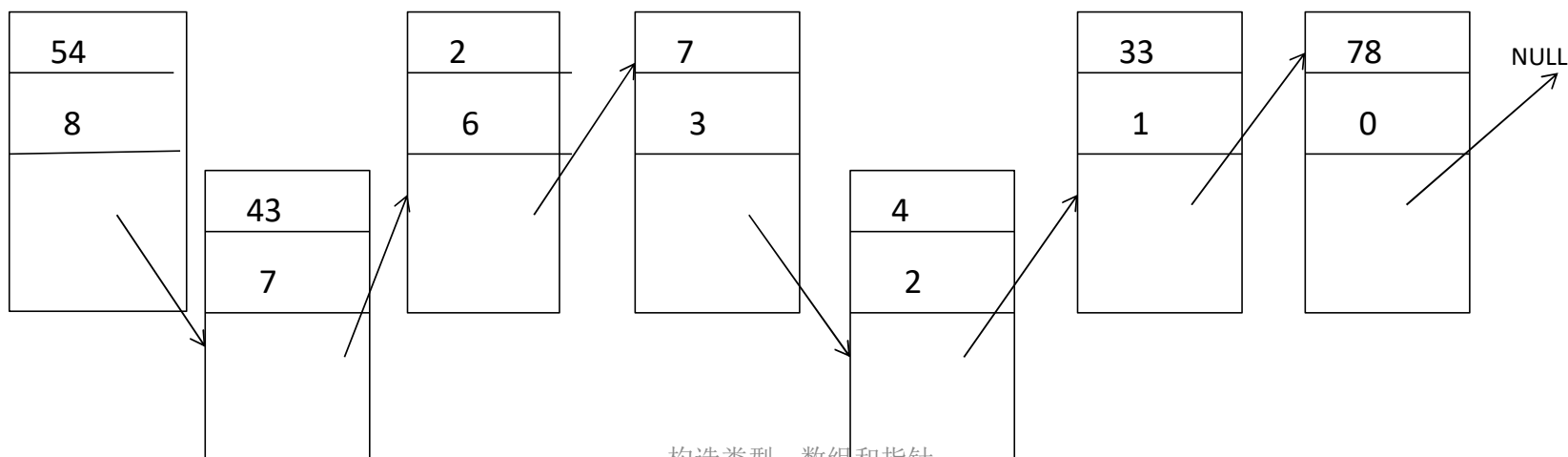
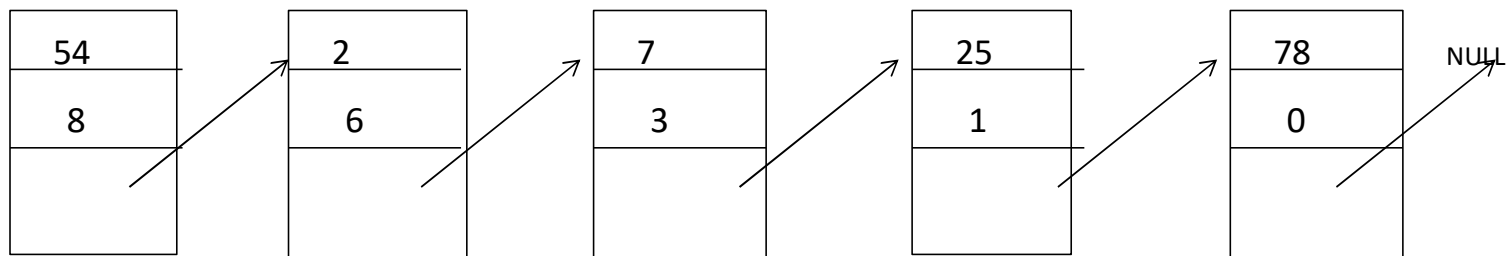
其和为

$54x^8 + 43x^7 + 2x^6 + 7x^3 + 4x^2 + 33x + 78$



问题2.2：算法设计*

- 首先读入第一个多项式，并将其生成一个链表；
- 然后依次将第二个多项式每一项插入（或合并）第一个多项式中



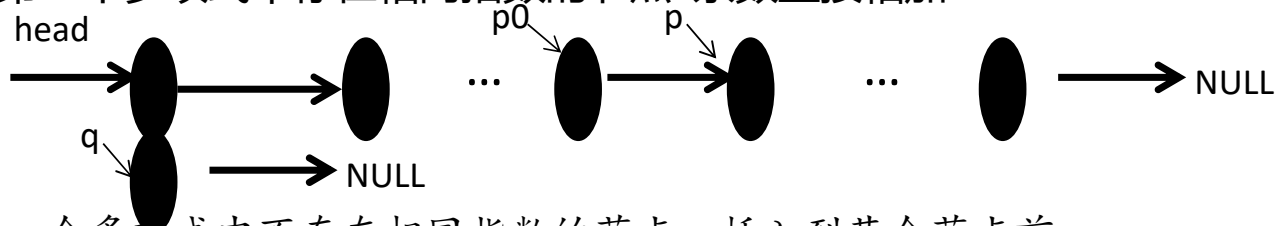
构造类型 - 数组和指针



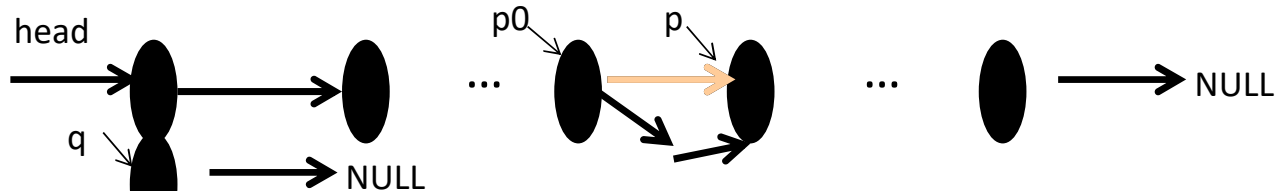
问题2.2：算法设计*

- 将第二个多项式的一个节点加到第一个多项式中有下面几种情况

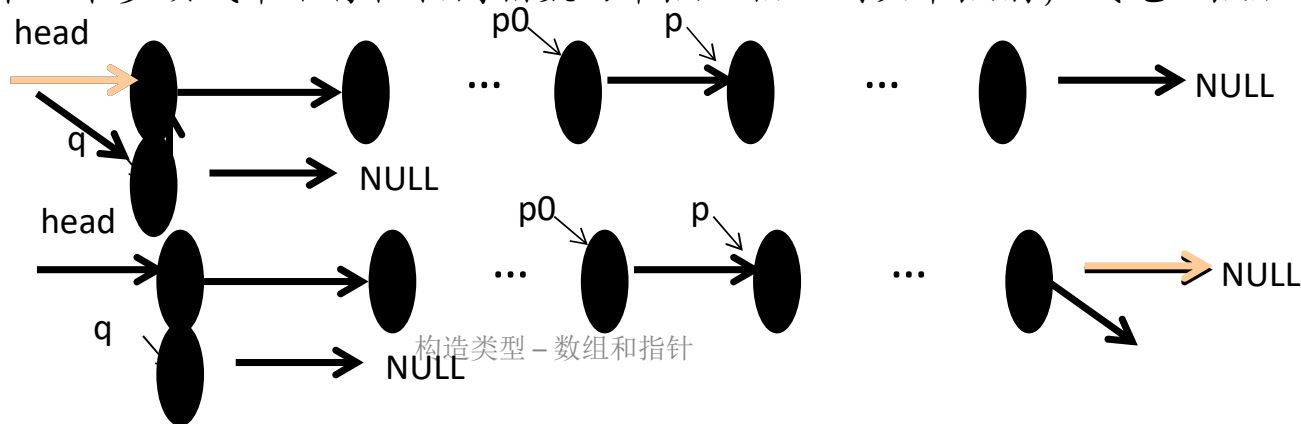
➢ 第一个多项式中存在相同指数的节点: 系数直接相加



- 第一个多项式中不存在相同指数的节点: 插入到某个节点前



- 第一个多项式中不存在相同指数的节点: 插入到头节点前, 或尾结点后





问题2.2： 多项式相加

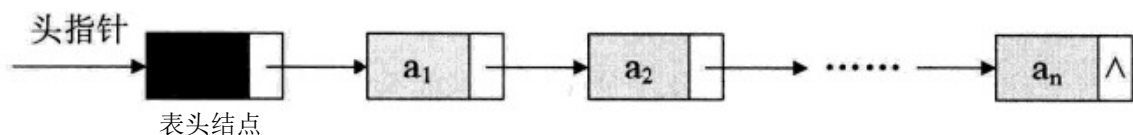
```
//c3_4c.c
#include <stdio.h>
#include <stdlib.h>
struct Node { //一个多项式节点结构
    int coe; //系数
    int pow; //幂
    struct Node *next;
};
int main()
{
    int a,n;
    char c;
    struct Node *head,*p,*q,*p0;
    head = p = NULL;
    do { //创建一个链表存放第一个多项式
        scanf("%d%d%c", &a, &n, &c);
        q = (struct Node *)malloc(sizeof(struct Node));
        q->coe = a; q->pow = n; q->next = NULL;
        if( head == NULL)
            head = p = q;
        else {
            p->next = q;
            p = p->next;
        }
    } while ( c != '\n');
```

```
do { //将第二个多项式的每个项插入到第一个多项式链表中
    scanf("%d%d%c", &a, &n, &c); //生成第二个多项式的一个节点
    q = (struct Node *)malloc(sizeof(struct Node));
    q->coe = a; q->pow = n; q->next = NULL;
    for(p=head; p!=NULL; p0=p,p=p->next) {
        if(q->pow > p->pow) {
            if(p==head) { q->next = head; head = q; break; } //插入到头节点前
            else { q->next = p; p0->next = q; break; } //将q插入到p前
        }
        else if(q->pow == p->pow) { p->coe += q->coe; break; } //指数相等，
        //系数相加
    }
    if(p== NULL) p0->next = q; //将q插入到尾节点后
} while ( c != '\n');
for(p=head; p!=NULL; p=p->next)
    printf("%d %d ", p->coe,p->pow);
return 0;
}
```

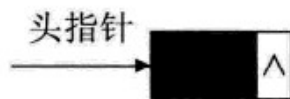


有时在构造链表时会给链表设置一个标志结点，称为**表头结点**（header，又称哑结点dummy node）。如下图所示：

带表头结点的非空链表



带表头结点的空链表



请思考这样做
有什么好处？

设置表头结点的最大好处是对链表结点的插入及删除操作统一了（不用考虑是否是头结点）。其数据域一般无意义（有时也可存放链表的长度）。



2.3.4 链式存储结构的特点

1. 优点

- (1) 存储空间动态分配，可以根据实际需要使用。
- (2) 不需要地址连续的存储空间(不需要大块连续空间)。
- (3) 插入/删除操作只须通过修改指针实现, 不必移动数据元素, 操作的时间效率高。

无论位于链表何处，
无论链表的长度如何，
插入和删除操作的时间都是 $O(1)$ 。

2. 缺点

- (1) 每个链结点需要设置指针域(占用存储空间小)。
- (2) 是一种非连续存储结构，查找、定位等操作要通过顺序遍历链表实现，时间效率较低。

时间为 $O(n)$



思考

1. 线性表可以采用顺序存储结构，也可以采用链存储结构，在实际问题中，应该根据什么原则来选择其中最合适的一种存储结构？



顺序存储结构与链表结构的比较

存储分配方式

- 顺序存储用一段连续的存储单元依次存储线性表的数据元素
- 链表采用链式储存结构, 用一组不连续的存储单元存放线性表的元素

时间性能

- 查找
 - 顺序存储: 无序 $O(n)$, 有序 $O(\log_2 n)$
 - 链表 $O(n)$
- 插入和删除
 - 顺序存储需要平均移动表长一半的元素, 时间为 $O(n)$
 - 链表在给出结点位置后, 插入和删除时间仅为 $O(1)$

空间性能

- 顺序存储需要事先分配存储空间, 分大了浪费, 分小了易发生溢出
- 链表不需要事先分配存储空间, 需要时分配结点, 元素个数不受限制

结论:

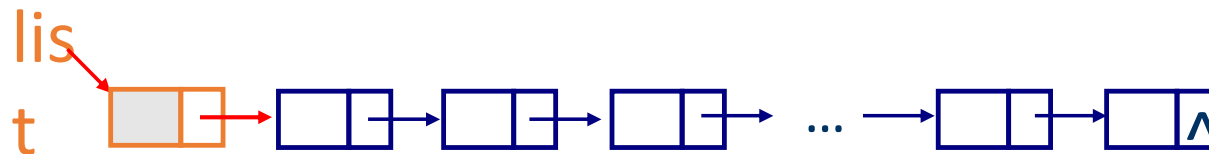
1. 若线性表需要频繁查找 (通讯录), 较少进行插入和删除操作时, 宜采用顺序存储结构。若需要频繁插入和删除时 (如词频统计), 宜采用链表结构。
2. 当线性表中的元素个数变化较大或者根本不知道有多大时 (如词频统计单词表), 最好用链表结构。而如果事先知道线性的大致长度, 用顺序结构效率会高些。



思考题

已知List为没有头结点的单链表中第一个结点的指针, 每个结点数据域存放一个字符, 该字符可能是英文字母或数字字符或其他字符, 编写算法构造三个以带头结点的单循环链表表示的线性表, 使每个表中只含同一类字符。
(要求用尽量少的时间和尽量少的空间)

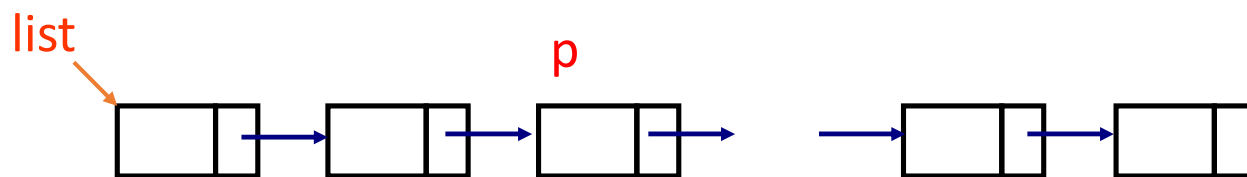
带头结点的线性链表



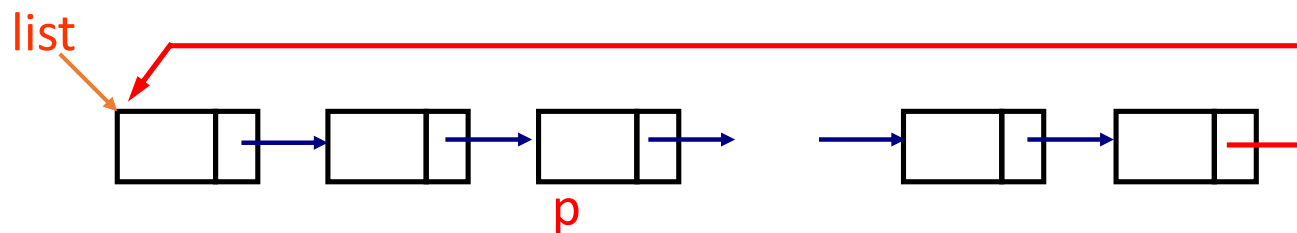


2.4 循环链表

线性链表



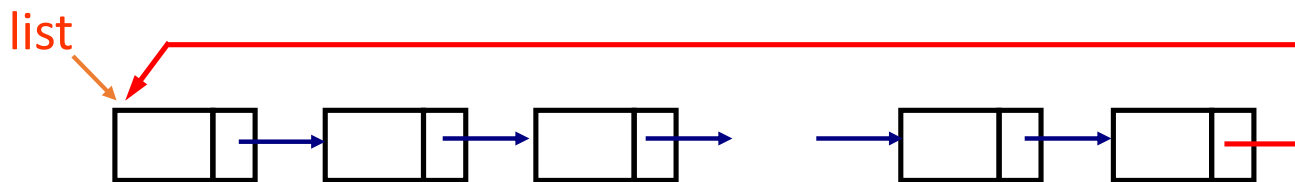
循环链表





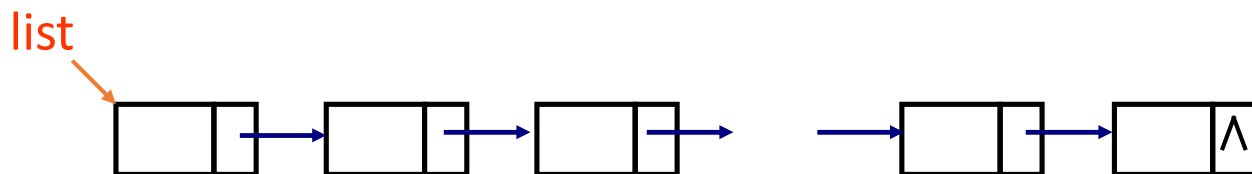
循环链表 是指链表中最后那个链结点的指针域存放指向链表最前面那个结点的指针，整个链表形成一个环。

循环链表

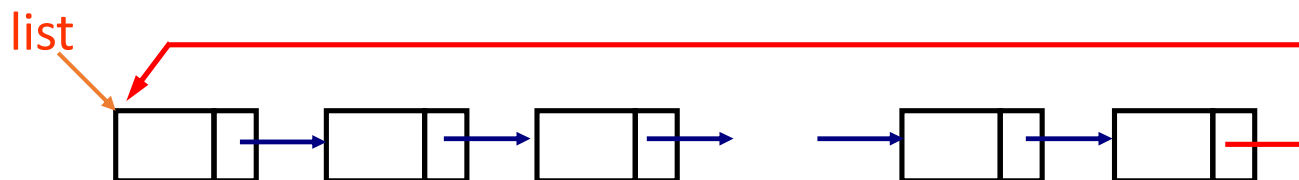




线性链表



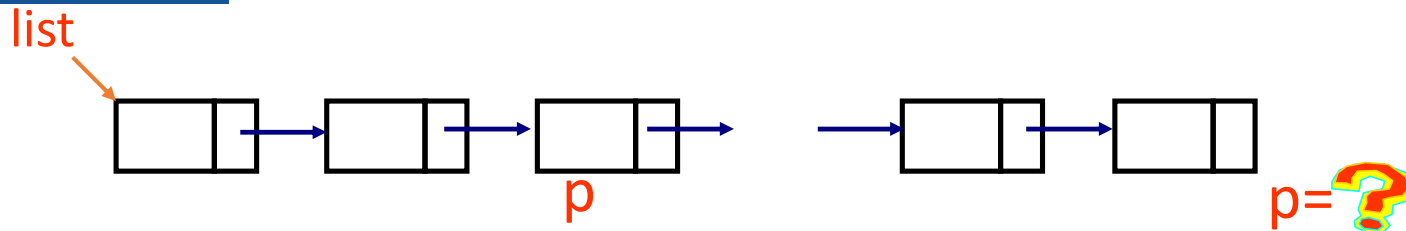
循环链表



对于循环链表，如何判断是否遍历了链表一周？



线性链表



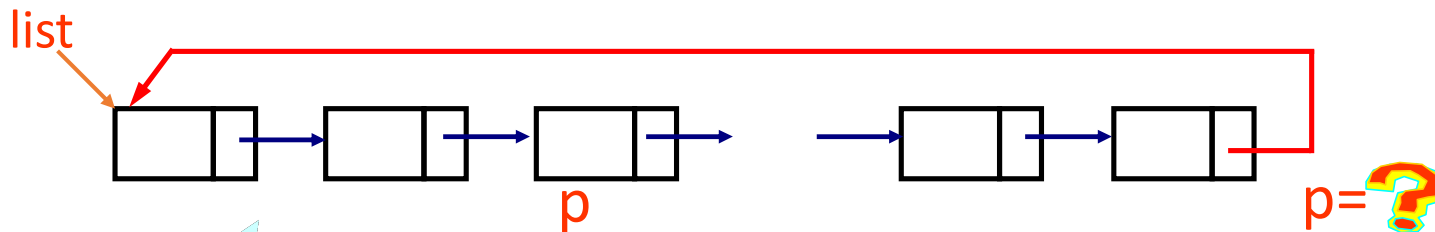
例如

有用的语句:

`p=p->link;`

`p=NULL`

循环链表



`p=list`



求非空线性链表的长度

```
int length( NodePtr list )
{
    Nodeptr p;
    int n;                /* 链表的长度置初值0 */
    for(p=list,n=0; p!=NULL; p=p->link,n++)
        ;
    return n;              /* 返回链表的长度n */
}
```

非
循
环
链
表

```
int length( Nodeptr list )
{
    Nodeptr p=list;
    int n=0;              /* 链表的长度置初值0 */
    if(list == NULL) return 0;
    do{
        p=p->link;
        n++;
    }while(p!=list);
    return n;              /* 返回链表的长度n */
}
```

循
环
链
表

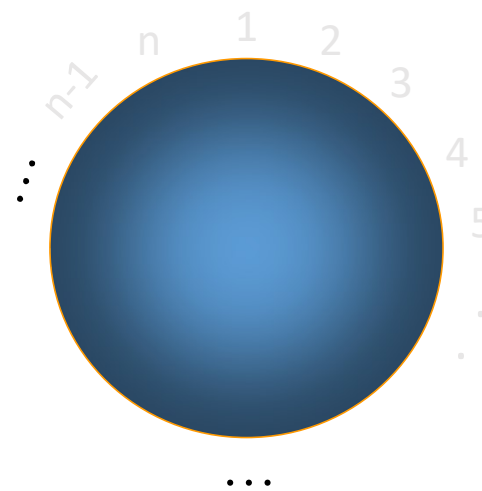


例 约瑟夫(JOSEPHU)问题

已知 n 个人(不妨分别以编号 $1, 2, 3, \dots, n$ 代表)围坐在一张圆桌周围, 编号为 k 的人从 1 开始报数, 数到 m 的那个人出列, 他的下一个人又从 1 开始继续报数, 数到 m 的那个人出列, \dots , 依此重复下去, 直到圆桌周围的人全部出列。

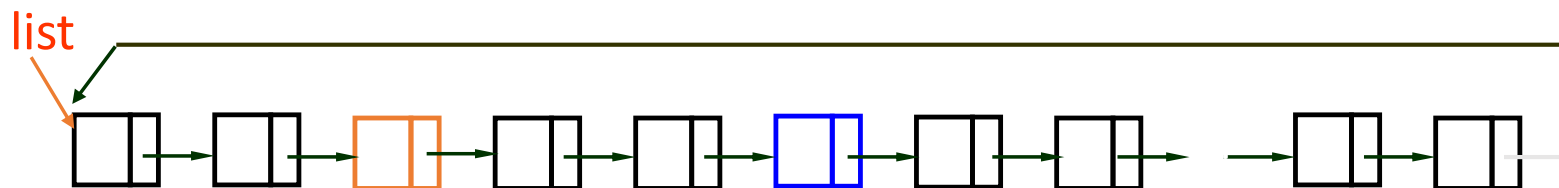
直到圆桌周围只剩一个人。

圆桌问题





利用一个不带表头结点的 循环链表



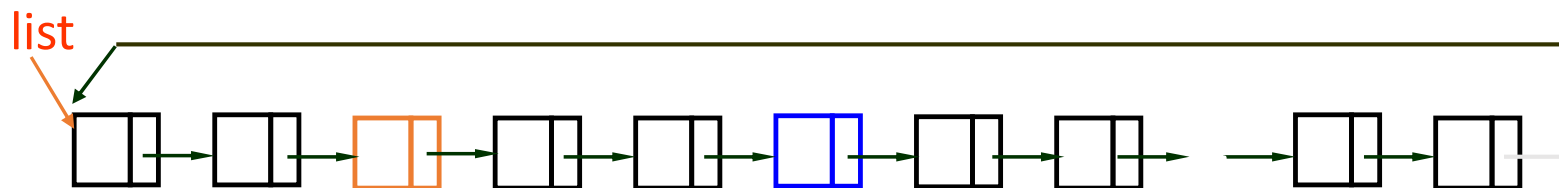
若假设 $k=3$, $m=4$

n : 链表中链结点的个数;
 k : 第一个出发点;
 m : 报数。



需要做的工作：

1. 建立一个循环链表；





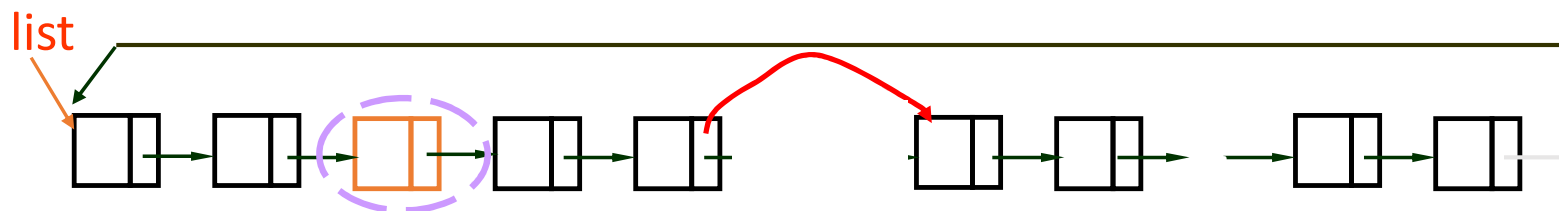
```
Nodeptr createList( int n)
{
    Nodeptr p,r, head = NULL;
    Datatype a;          /* 创建一个空链表 */

    for(i=0;i<n;i++){
        READ(a);          /* 取一个数据元素 */
        r=(NodePtr)malloc(sizeof(Node));
        r->data=a;
        r->link=NULL;
        if (head==NULL)
            head = p = r;
        else {
            p->link=r;      /* 将新结点链接在链表尾部 */
            p=p->link;
        }
    }
    p->link=head;
    return head;
}
```



需要做的工作：

1. 找到第一个出发点；
2. 反复删除第 m 个链结点。



若假设 $k=3$, $m=4$

`p=p->link;`

$k-1$ 次 $m-1$ 次



算法

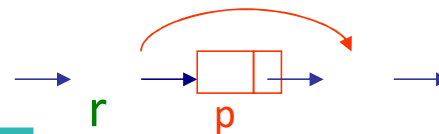
```
void josephu( int n, int k, int m )
{
    Nodeptr list,p,r;
    int i;
    list=NULL;
    for(i=0;i<n;i++) {
        r=(Nodeptr)malloc(sizeof(Node));
        r->data=i;
        if(list==NULL)
            list=p=r;
        else {
            p->link=r;
            p=p->link;
        }
    }
    p->link=list; /* 建立循环链表 */
    r = p;
    for(p=list,i=0;i<k-1;i++,r=p,p=p->link)
        ;
    /* 找到第一个点 */
}
```



当 $k \neq 1, m = 1$ 时

反复寻找并删除结点

```
while(p->link!=p){
    for(i=0;i<m-1;i++){
        r=p;
        p=p->link;
    }
    r->link=p->link;
    printf("%3d",p->data);
    free(p);
    p=r->link;
}
printf("%3d", p->data);
```





主函数

```
#include <stdlib.h>
void josephu(int n, int k, int m);
int main( )
{
    int n, k, m;
    printf("\nInput n, k, m: ");
    scanf("%d %d %d",&n, &k,&m);
    josephu(n,k,m);
    return 0;
}
```

输入链结点总数
n、报数的起始位
置k与报数m。

调用约瑟夫函数



问题2.3：显示文件最后n行

- 问题：命令**tail**用来显示一个文件的最后**n**行。其格式为：

tail [-n] filename

其中：

-n：n表示需要显示的行数，省略时n的值为10。

filename：给定文件名。

如，命令**tail -20 example.txt** 表示显示文件**example.txt**的最后20行。实现该程序，该程序应具有一定的错误处理能力，如能处理非法命令参数和非法文件名。



问题2.3：问题分析

若从命令行输入：

`tail -20 test.txt`

以显示文件`test.txt`的最后20行。

- 如何得到需要显示的行数和文件名？

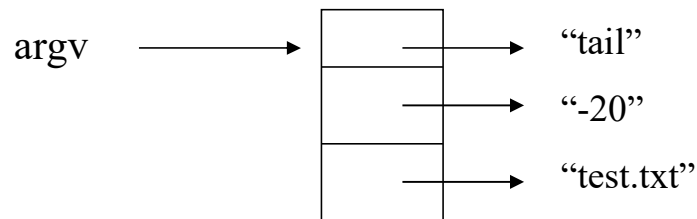
- 使用命令行参数

- `int main(int argc, char *argv[])`

- 行数 `n = atoi(argv[1])+1`

- 文件名 `filename = argv[2]`

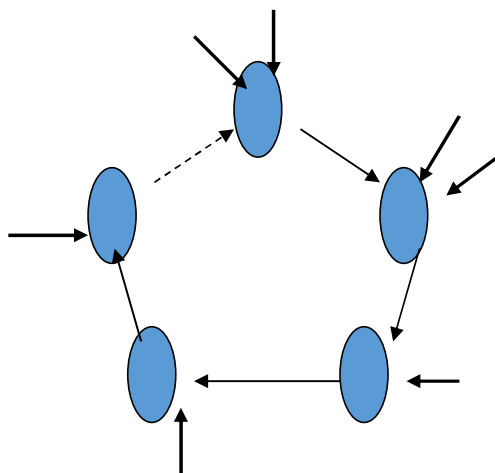
- 如何得到最后n行？





问题2.3：算法设计

- 方法之一：使用 n 个节点的循环链表。链表中始终存放最近读入的 n 行。
 1. 首先创建一个空的循环链表；
 2. 然后再依次读入文件的每一行挂在链表上，最后链表上即为最后 n 行。





问题2.3：代码实现（循环链表）

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define DEFLINES 10
#define MAXLEN 81
struct Node {
    char *line;
    struct Node *next;
};
```

命令行输入中没有指定打印行数时，获取文件名，此时行数为缺省10。如
tail test.txt

```
int main(int argc, char *argv[ ])
{
    char curline[MAXLEN], *filename;
    int n = DEFLINES, i;
    struct Node *first, *ptr;
    FILE *fp;
    if( argc == 3 && argv[1][0] == '-') {
        n = atoi(argv[1]+1);
        filename = argv[2];
    }
    else if( argc == 2)
        filename = argv[1];
    else {
        printf("Usage: tail [-n] filename\n");
        return (1);
    }
}
```

命令行输入中指定打印行数时，获取行数及文件名。
如tail -20 test.txt



问题2.3：代码实现

```
if((fp = fopen(filename, "r")) == NULL){  
    printf(" Can't open file: %s !\n", filename);  
    return (-1);  
}  
first = ptr = (struct Node *)malloc(sizeof ( struct Node));  
first->line = NULL;  
for(i=1; i<n; i++){  
    ptr->next = (struct Node *)malloc(sizeof ( struct Node));  
    ptr = ptr->next;  
    ptr->line = NULL;  
}  
ptr->next = first;  
ptr = first;
```

创建循环链表

将链表的最后一个节点指向头节点，以构成一个循环链表。



问题2.3：代码实现

```
while(fgets(curline, MAXLEN, fp) != NULL){
    if(ptr->line != NULL)    /*链表已经满了，需要释放掉不需要的行*/
        free(ptr->line);
    ptr->line = (char *) malloc ( strlen(curline)+1);
    strcpy(ptr->line, curline);
    ptr = ptr->next;
}
for(i=0; i<n; i++) {
    if(ptr->line != NULL)
        printf("%s",ptr->line);
    ptr = ptr->next;
}
fclose(fp);
return 0;
}
```

测试考虑点：

准备一个包含内容（如11~20行）
的正文文件test.txt

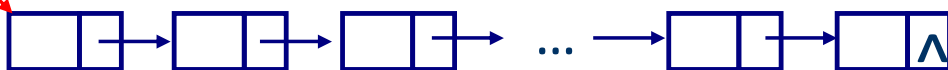
- 1) tail -5 test.txt (正常)
- 2) tail test.txt (正常)
- 3) tail -30 test.txt (非正常)
- 4) tail -0 test.txt (非正常)
- 5) tail -1 test.txt (边界)



线性链表(单链表)

线性表的链式存储结构

list



带头结点的线性链表

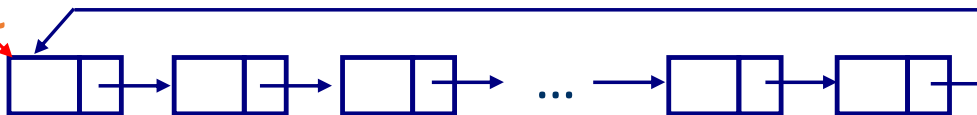
lis

t



循环链表

list



带头结点的循环链表

list



2.5 双向链表及其操作

本节主要内容

1. 双向链表的构造
2. 双向链表的插入与删除



2.5.1 双向链表的构造

所谓**双向链表**是指链表的每一个结点中除了数据域以外设置两个指针域，其中之一指向结点的直接后继结点，另外一个指向结点的直接前驱结点。

链结点的实际构造可以形象地描述如下：

(左指针域) \longleftarrow llink data rlink \longrightarrow (右指针域)

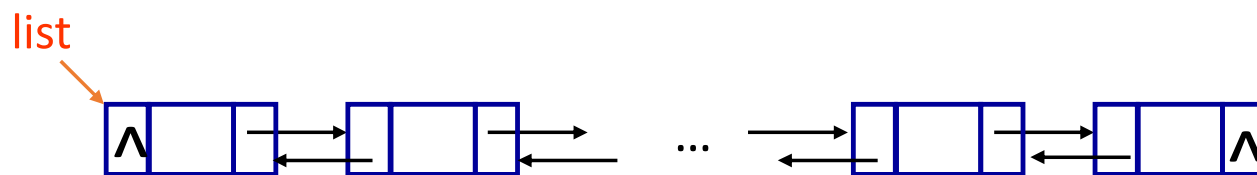
其中，data 为数据域

rlink, llink 分别为指向该结点的直接后继结点与直接前驱结点的指针域

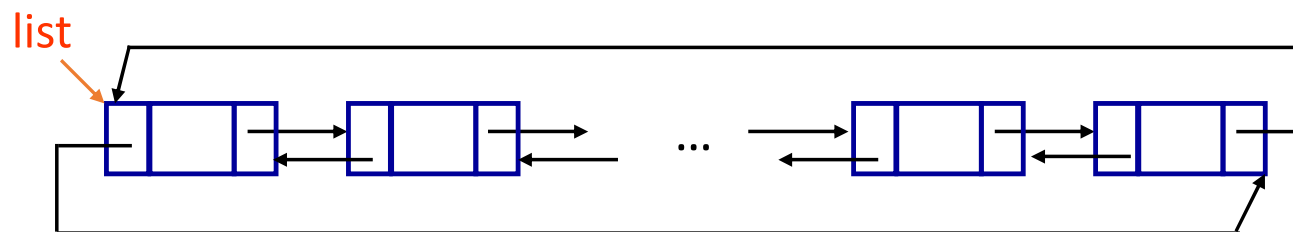
分别称为右指针和左指针



双向链表的几种形式



双向链表



双向循环链表



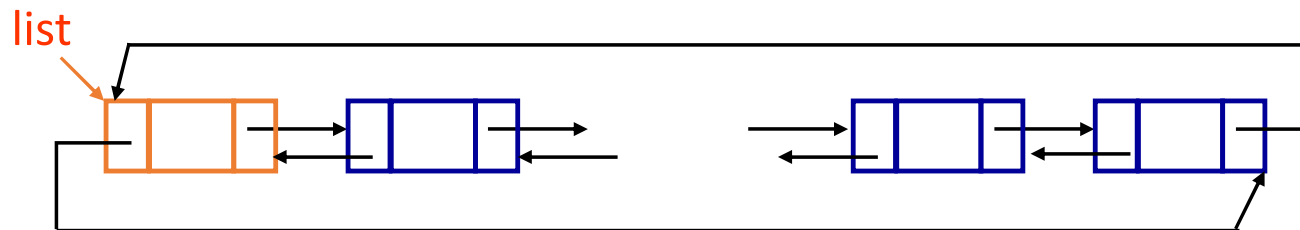
类型定义

```
struct node {  
    ElemType data;  
    struct node *rlink, *llink;  
};  
typedef struct node *DNodeptr;  
typedef struct node DNode;
```




2.5.2 双向链表的插入

功能 在非空双向循环链表中某个数据域的内容为x的链结点右边插入一个数据信息为item的新结点。

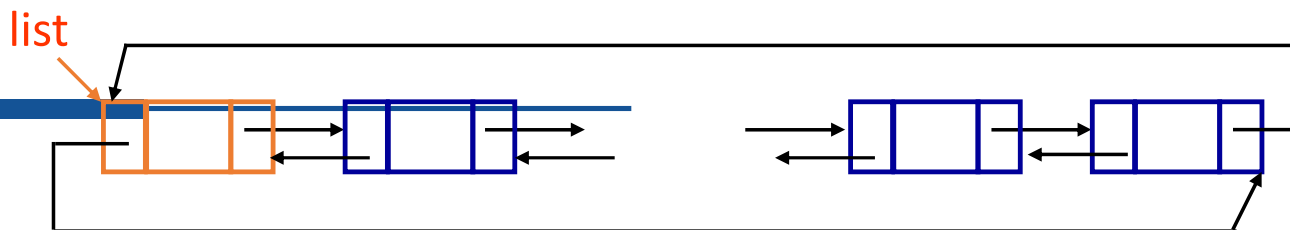


需要做的工作:

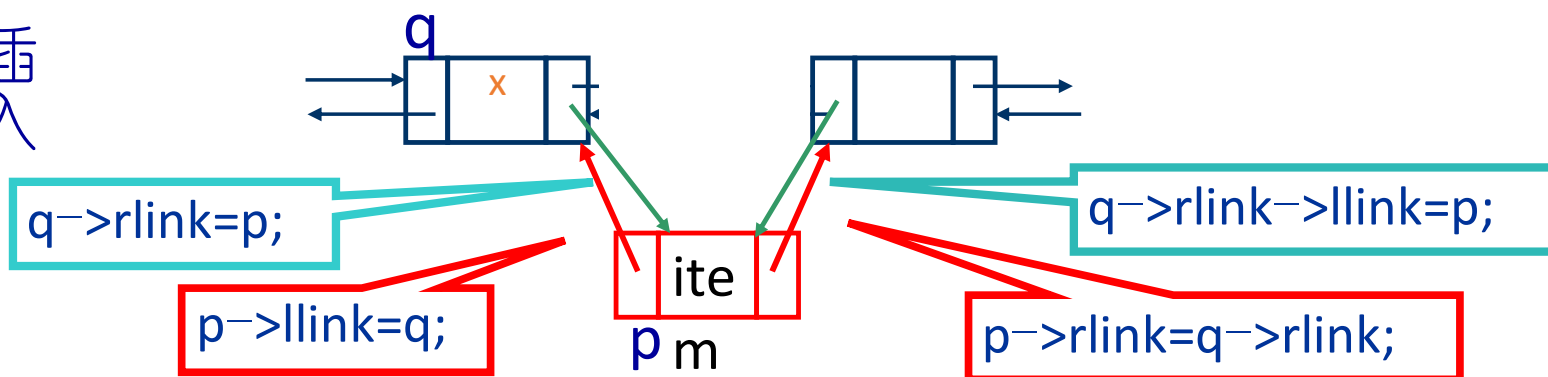
1. 找到满足条件的结点;
2. 若找到, 构造一个新的链结点;
3. 将新结点插到满足条件的结点后面。



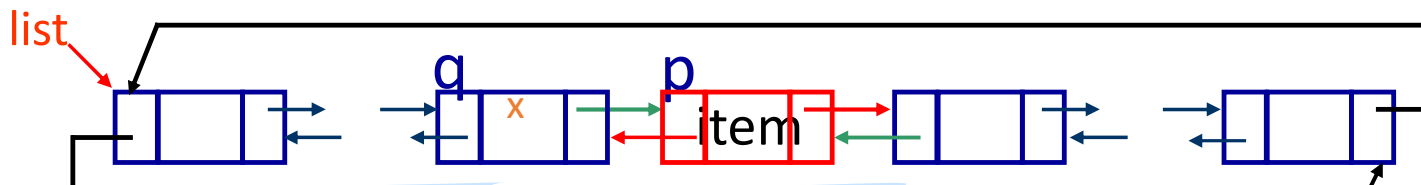
插入前



插入



插入后



注意：在头(第一个)结点前插入一个结点时，步骤如下：

```
p->rlink = list; p->llink = list->llink;  
list->llink->rlink = p; list->llink = p;  
list = p;
```



算法

时间复杂度 $O(n)$

```
int insertDNode(DNodeptr list, ElemType x, ElemType item)
```

```
{
```

```
    int DNodeptr p,q;
```

```
    for(q=list; q->rlink!=list && q->data!=x; q=q->rlink);/* 寻找满足条件的链结点 */
```

```
    if(q->rlink==list&&q->data!=x)
```

```
        return -1;
```

/* 没有找到满足条件的结点 */

```
    p=(DNodeptr)malloc(sizeof(DNode)); /* 申请一个新的结点 */
```

```
    p->data=item;
```

```
    p->llink=q;
```

```
    p->rlink=q->rlink;
```

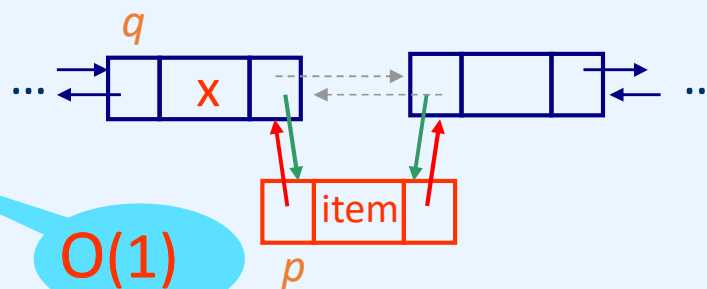
```
    q->rlink->llink=p;
```

```
    q->rlink=p;
```

```
    return 1; /* 插入成功 */
```

```
}
```

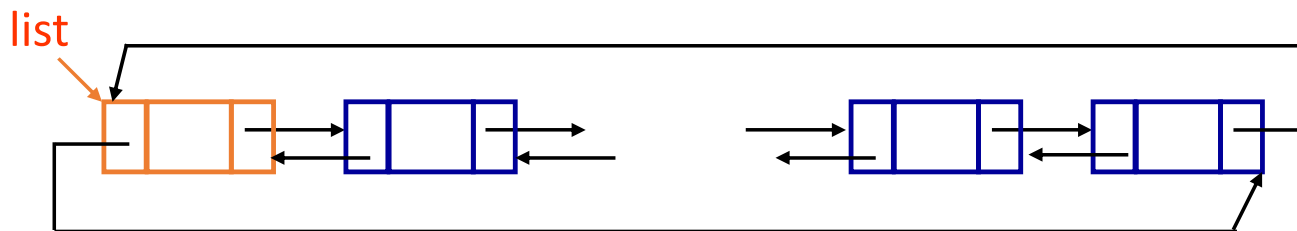
寻找满足条件的结点





2.5.3 双向链表的删除

功能 删除非空双向循环链表中数据域的内容为x的链结点。

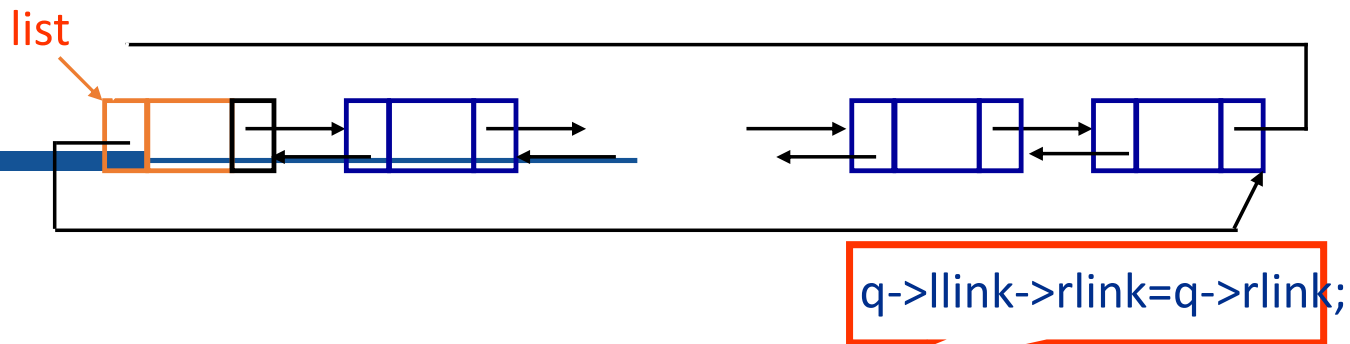


需要做的工作:

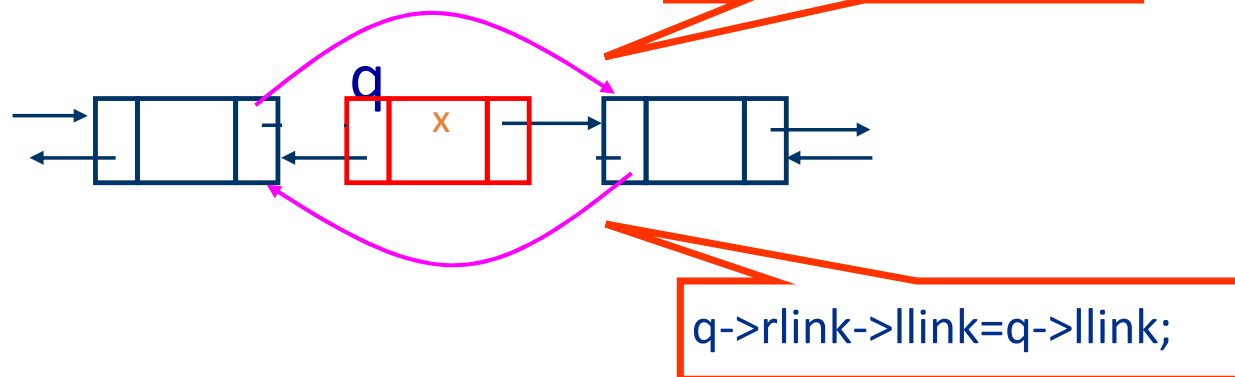
1. 找到满足条件的结点;
2. 若找到, 删除(并释放)满足条件的结点。



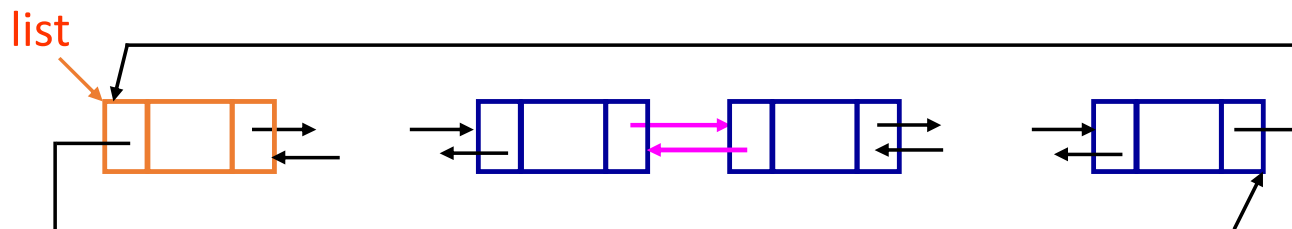
删除前



删除



删除后



注意：删除头(第一个)结点时，步骤如下：

```
list->rlink->llink = list->link;  
list->llink->rlink = list->rlink;  
q = list; list = list->rlink; free(q);
```

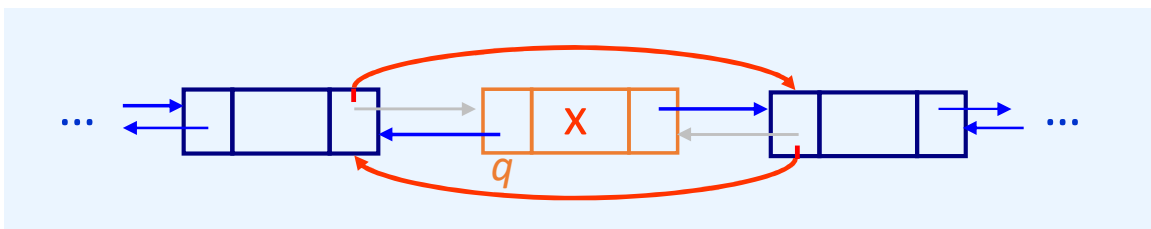


算法

时间复杂度 $O(n)$

```
int deleteDNode(DNodeptr list, ElemType x)
{
    DNodeptr q;
    /* q初始指向头结点的下一个结点 */
    for(q=list; q->rlink!=list && q->data!=x; q=q->rlink) /*找满足条件的链结点 */
    ;
    if(q->rlink==list && q->data!=x)
        return -1; /* 没有找到满足条件的结点 */

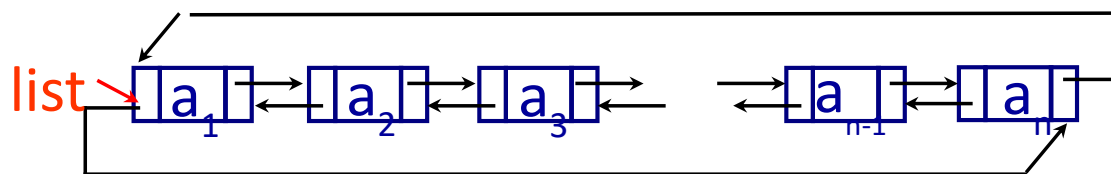
    q->llink->rlink=q->rlink;
    q->rlink->llink=q->llink;
    free(q); /* 释放被删除的结点的存储空间 */
    return 1; /* 删除成功 */
}
```

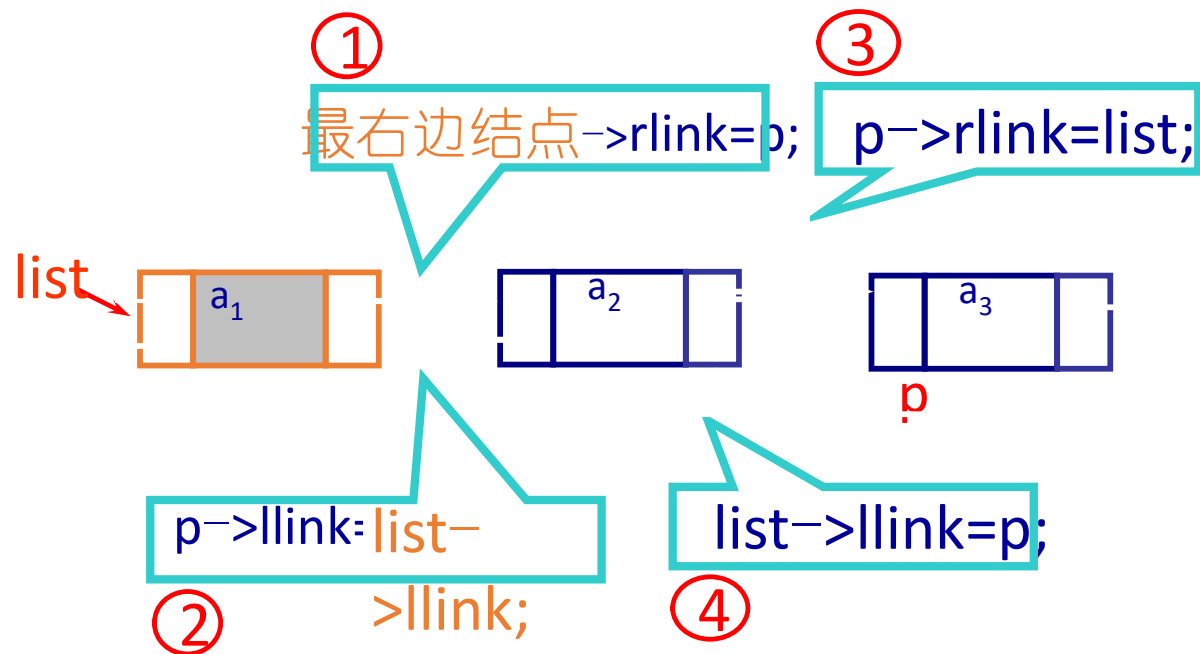




构造一个双向循环链表

$(a_1, a_2, a_3, \dots, a_{n-1}, a_n)$





首先构造链表的第一个结点



```
DNodeptr initDLink(int n)
{
    int i;
    DNodeptr list,p;
    list=(DNodeptr)malloc(sizeof(DNode));
    READ(list->data);
    list->llink=list;
    list->rlink=list;
    for(i=1;i<n;i++){
        p=(DNodeptr)malloc(sizeof(DNode));
        READ(p->data); /* 读入一元素 */
        insertNode(list,p);
    }
    return list;
}
```



算法

```
void insertNode(DNodeptr list, DNodeptr p)
{
    list->llink->rlink=p; ①
    p->llink=list->llink; ②
    p->rlink=list; ③
    list->llink=p; ④
}
```

功能

将指针为p的结点插入到头结点指针为list的双向循环链表中

时间复杂度
 $O(n)$



双向链表

- ✓ 双向链表由于多了一个前驱结点的指针，使得结点的插入和删除时需要做更多的操作，需要小心；
- ✓ 双向链表需要保存前驱和后续结点的指针，要比单向链表多占用一些空间；
- ✓ 双向链表由于很好的对称性，使得对某个结点的前后结点访问带来了方便，简化了算法，可以提高算法的时间性能，以空间换时间。



链表使用的注意事项：

- ① 应确保链表结点指针指向一个合法空间（通常由malloc申请而得），否则结点操作时会出现内存错误（**memory access violation**），如：

```
struct Node *p;  
p->link = q;
```
- ② 单向链表的最后一个结点的p->link指针一定要为NULL, 通常申请一个结点p时及时执行**p->link = NULL;**语句是一个好习惯;
- ③ 当链表结点删除后应及时用free(p)释放。不释放不用的结点会造成内存泄漏（**memory leak**），这是工程应中常见问题；
- ④ 不能随意移动链表的头结点指针，对头结点的移动都应是有意义的（如要在头结点前插入一个结点或要删除链表的第一个结点），否则会造成链表头结点的丢失。



本章内容小结





线性表

线性表的基本概念

- 什么是线性关系？
- 什么是线性表？
- 线性表的基本操作有哪些？其中最主要的有哪些？

线性表的顺序存储结构

- 构造原理。
- 插入、删除操作对应的算法设计。
- 特点(优点、缺点)。

线性表的链式存储结构

- 线性链表、循环链表和双向链表的构造原理。
- 各种链表中进行插入、删除操作对应的算法设计。
- 头结点问题。



结束！