数据结构与程序设计（信息类）

Data Structure and Programming

北航软件学院 谭火彬

数据结构与程序设计（信息类）

Data Structure and Programming

第 2 章

线 性 表(Linear List)

目录

CONTENTS

数据结构与程序设计（信息类）

Data Structure and Programming

第 2 章

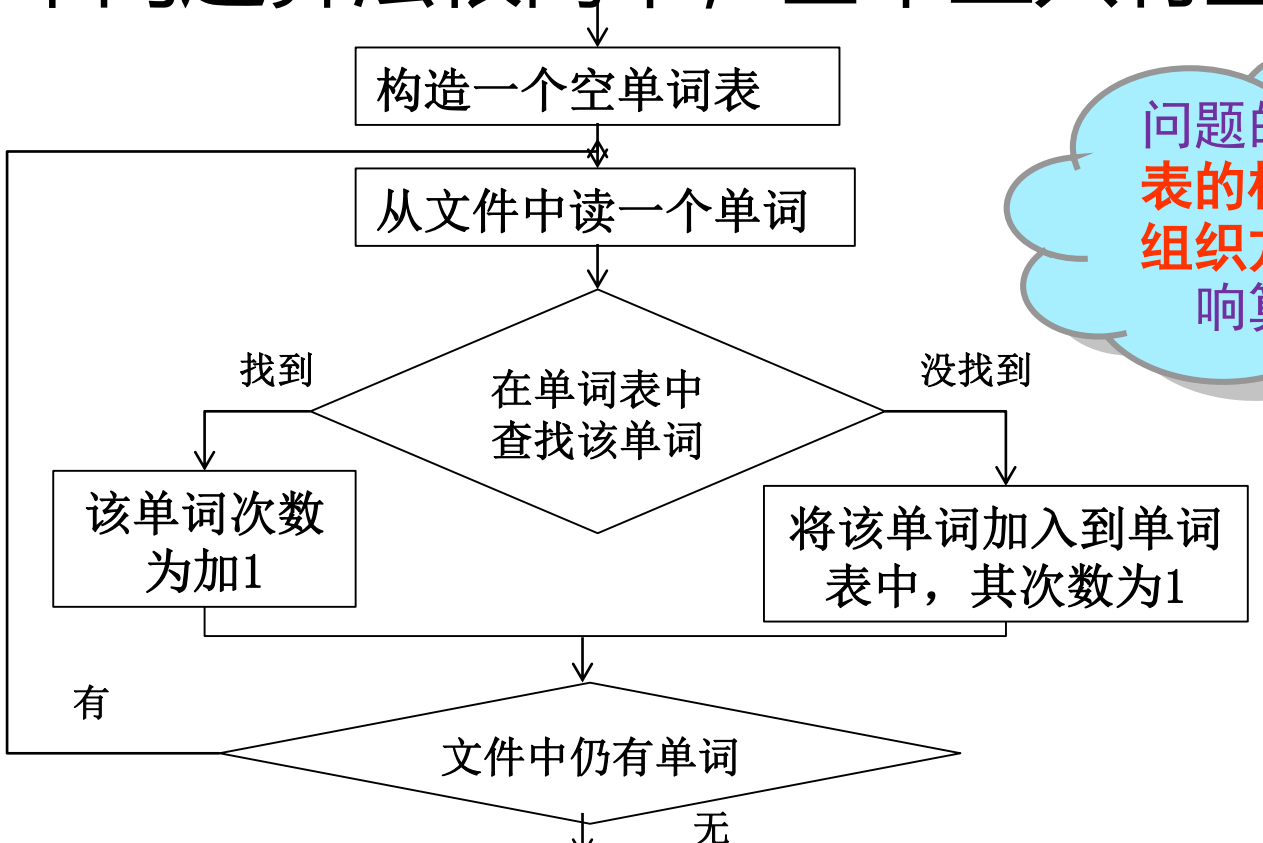
线 性 表(Linear List)

目录

CONTENTS

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?

用顺序存储？链式存储？
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

	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	……

- 线性表：$A=(a_1, a_2, \ldots, a_n)$
- 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
- 线性表
  - 数据元素之间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

示例：线性表

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

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

一个数据元素为一个字母

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

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

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≤i≤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。

……

线性表的基本操作（函数原型）

- 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
- ……

数据结构与程序设计（信息类）

Data Structure and Programming

第 2 章

线 性 表(Linear List)

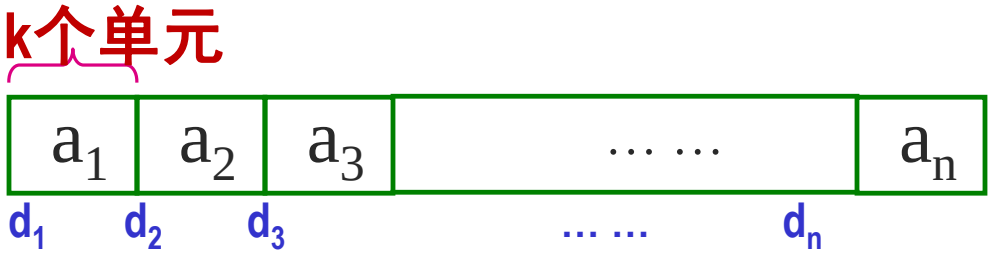
目录

CONTENTS

2.2 线性表的顺序存储

- 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

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



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

LOC(a_i)

顺序存储结构

结论

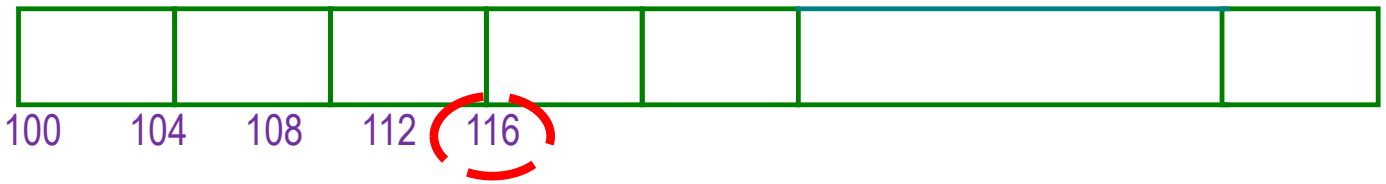
若假设每个数据元素占用k个存储单元，并且已知第一个元素的存储位置$LOC(a_1)$，则有

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

提问

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？

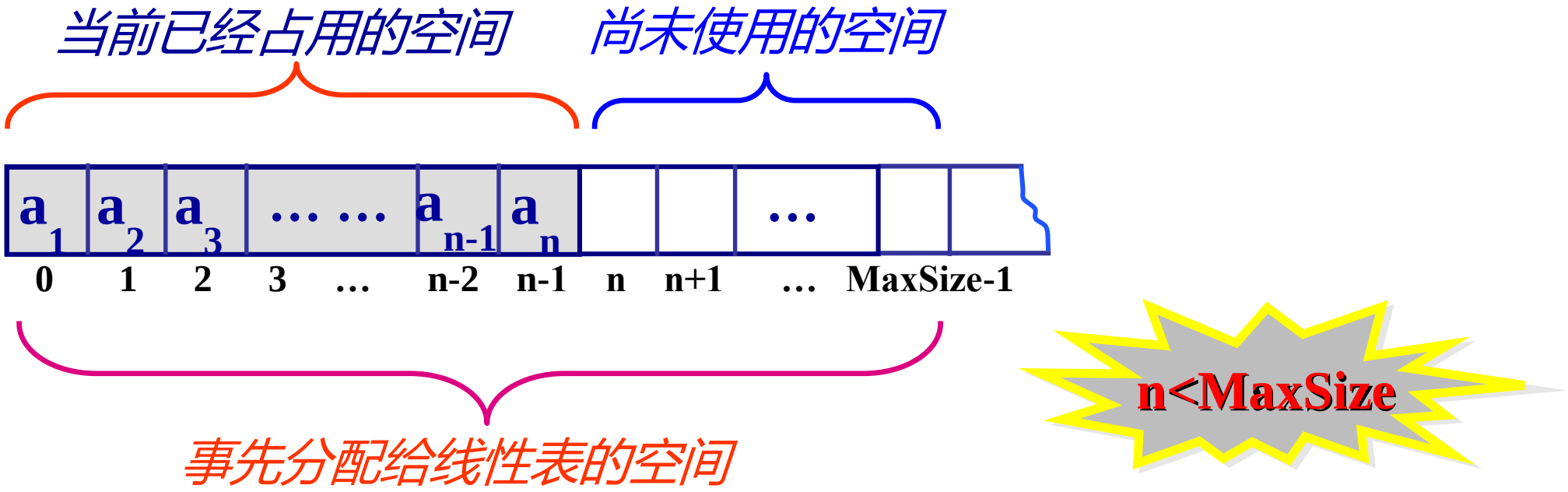
例：$LOC(a_1)=100$　$k=4$　求$LOC(a_5)=?$



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

顺序存储结构示意图

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



C语言实现顺序存储结构

$$A = (a_1, a_2, a_3, \cdots \cdots , a_{n-1}, a_n)$$

```
typedef int ElemType;
#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

基本算法：查找

1.查找：确定元素item在长度为n的顺序表list中的位置

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

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)

注意逻辑位置和物理位置的区别：C语言中第一个元素的下标是0

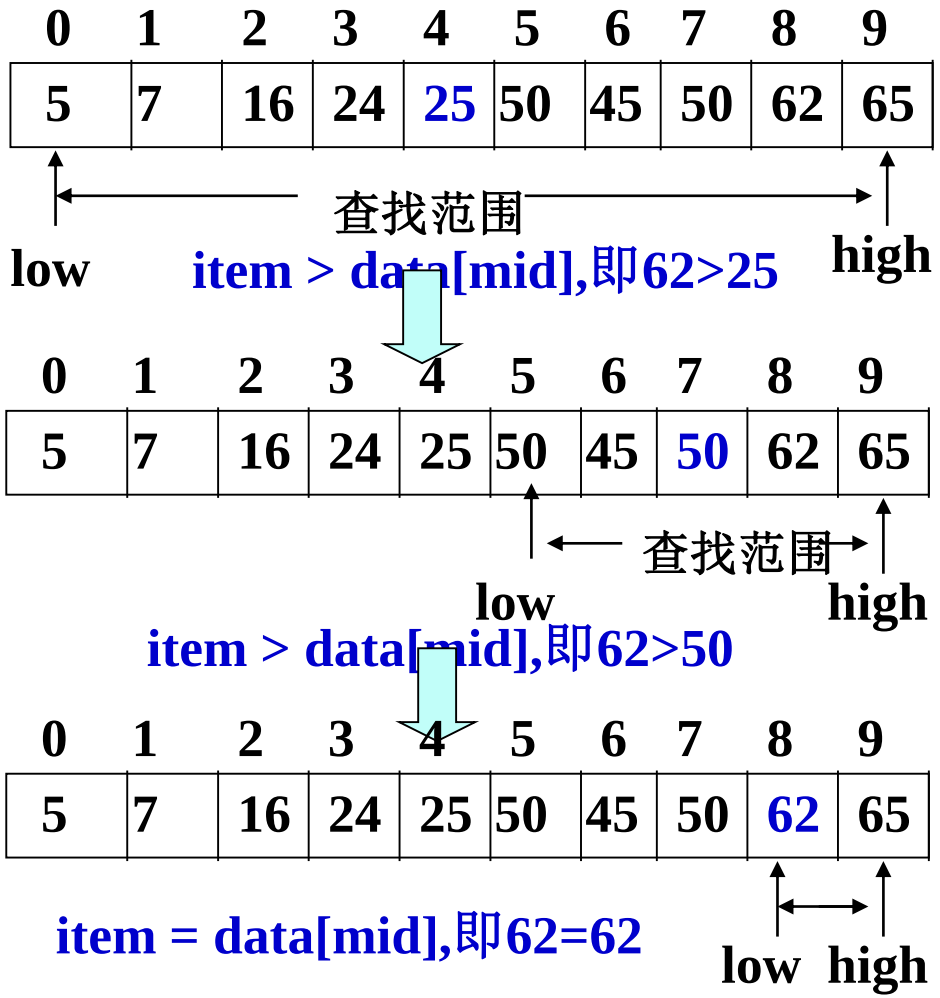
更高效的查找算法：折半查找

如何在有序顺序表中查找元素？

- 假设数据集按由小到大排列，折半查找算法的核心思想是：
  - 将要查找的有序数据集的中间元素与指定数据项相比较；
  - 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；
  - 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；
  - 如果指定数据项等于中间元素，则查找成功结束。
  - 最后如果数据集中没有元素再可进行查找，则查找失败。

示例：折半查找过程

例：在下面有序数据集中查找数据项62



item = 62(查找顶)
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

折半查找算法的实现

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。

基本算法：插入

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

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

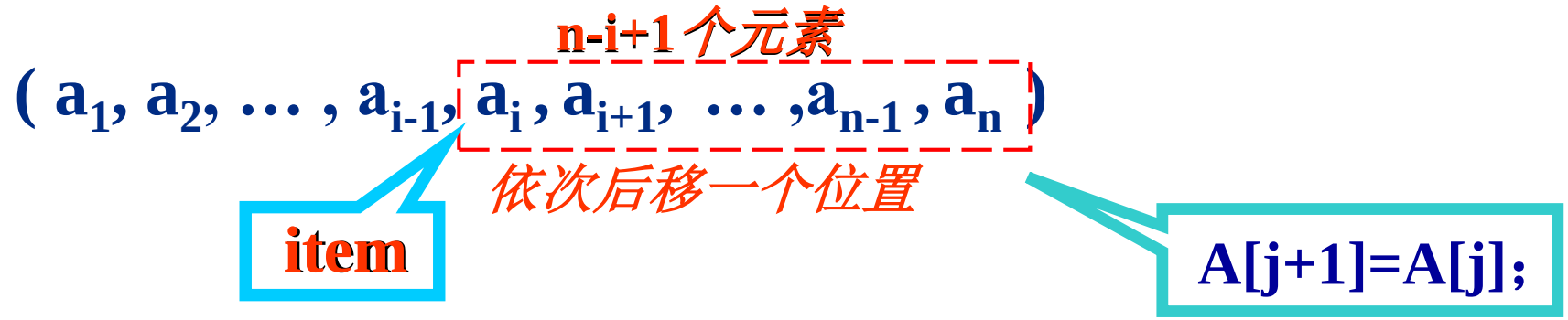
n个数据元素

转换成长度为n+1的线性表

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

n+1个数据元素

插入算法：过程解析



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==MaxSize)

(2) 插入位置是否合适？(正常位置:0≤i≤n)

插入算法：实现

```
/* 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;
      if (N==MaxSize || i<0 || i>N)
            return -1;                    /* 插入失败，约定返回-1 */
      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，约定返回1*/
}
```

测试空间满否

测试插入位置合适否

该算法的时间复杂度是：O(n)

算法分析：元素的移动次数的平均值

- 元素移动次数的平均值：衡量插入和删除算法时间效率的另一个重要指标
  - 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

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

基本算法：删除

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

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

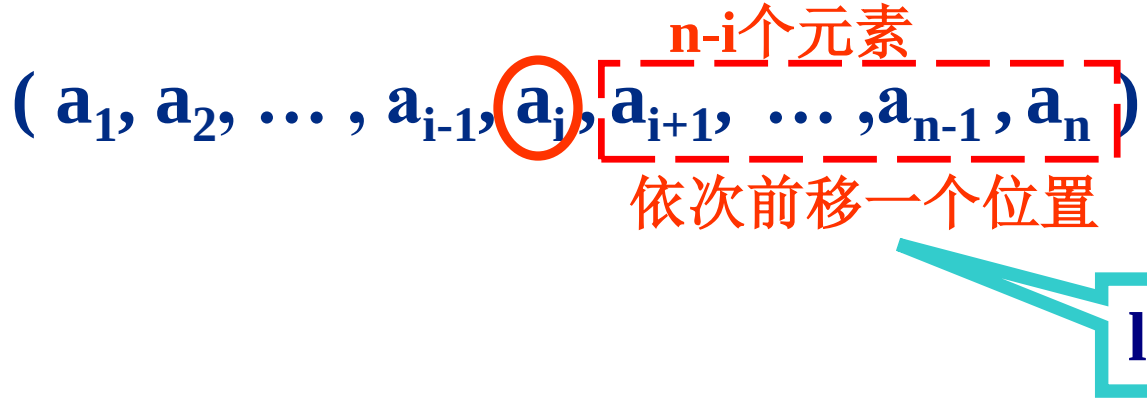
n个数据元素

转换成长度为 n-1 的线性表

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

n-1个数据元素

删除算法：过程解析



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)

删除算法：实现

```
/* 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;                    /* 删除失败 */
      for( k=i+1; k<N; k++ )
          list[k-1]=list[k];            /* 元素依次前移一个位置 */
      N--;                              /* 线性表的长度减1 */
      return 1;                         /* 删除成功 */
}
```

测试表空和位置合适与否

该算法的时间复杂度是：O(n)

元素的平均移动次数：$T_{ds}= \sum p_i(n-i) = \sum (n-i)/n = (n-1)/2$

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。

2.缺点

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。

(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)

数据结构与程序设计（信息类）

Data Structure and Programming

第 2 章

线 性 表(Linear List)

目录

CONTENTS

实例：有序表的插入

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？

$a_i \le a_{i+1}$ $1 \le i \le n-1$

item

$a_1, a_2, a_3, \ldots, a_i, a_{i+1}, a_{i+2}, \ldots, a_n$

依次后移一个位置

插入

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

过程分析

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

算法实现

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

```
/*假设 N是表长度，是一个全局变量 */
int insertElem(ElemType list[ ], ElemType item)
{
    int i,j;

    if(N == MAXSIZE)  return -1;
    /* 寻找item的合适位置 */
    for(i=0; i<N&&item>=list[i]; i++) ;
    for(j=N-1; j>=i; j--)
        list[j+1]=list[j];

    list[i]=item;          /* 将item插入表中 */
    N++;
    return 1;
}
```

能处理吗？

确定插入位置

表长加1

O(n)

特殊情况

```
#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;
}
```

算法改进：利用折半查找确定元素位置

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);      /* 寻找item的合适位置 */

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

     list[i]=item;      /* 将item插入表中 */
     N++;
     return 1;
}
```

折半查找确定插入位置

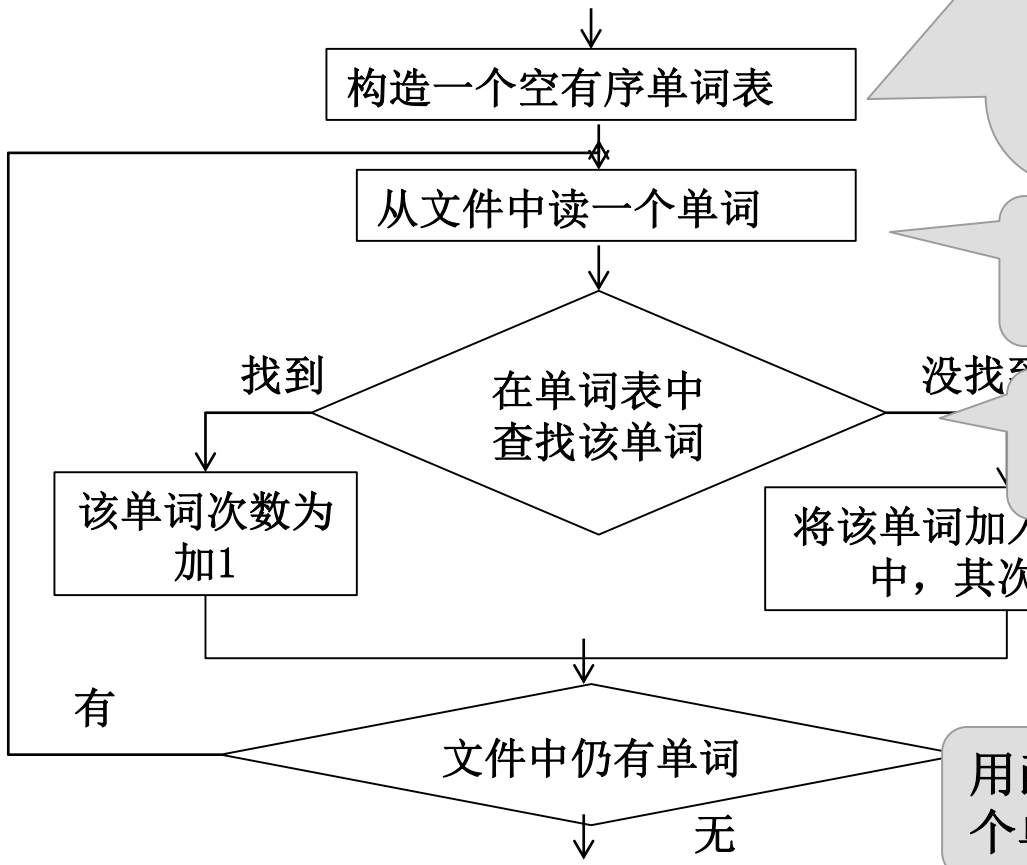
折半查找算法如下:

```
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




```
/*用数组构造一个顺序表，表中单词按字典序组织*/
struct lnode {
    char  word[MAXWORD];
    int count;
};
struct lnode 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

代码实现

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAXWORD  32
#define MAXSIZE  1024
struct lnode {
    char word[MAXWORD];
    int count;
};
int getWord(FILE *bfp,char *w);
int searchWord(struct lnode list[],
               char *w);
int insertWord(struct lnode list[ ],
        int pos, char *w);
int N=0; //全局变量，单词表实际单词个数
```

```c
int main(){
    struct lnode 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;
}
```

代码实现：读单词

```
//从文件中读入一个单词（仅由字母组成的串），并转换成小写字母
int getWord(FILE *fp, char *w)
{
      int c;

    while(!isalpha(c=fgetc(fp)))
        if(c == EOF) return c;
       else continue;
    do  {
         *w++ = tolower(c);
    } while(isalpha(c=fgetc(fp)));
    *w='\0';
   return 1;
}
```

代码实现：查找单词位置和插入单词

```
/*在表中查找一单词，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(struct lnode list[], char *w)
{
    int low = 0, high = N - 1, mid = 0;
    while (low <= high)
    {
        mid = (high + low) / 2;
        if (strcmp(w, list[mid].word) < 0)
            high = mid - 1;
        else if (strcmp(w, list[mid].word) > 0)
            low = mid + 1;
        else
        {
            list[mid].count++;
            return 0;
        }
    }
    return insertWord(list, low, w);
}
```

```
/*在表中相应位置插入一个单词，同时置次数为1*/
int insertWord(struct lnode list[],
        int pos, char *w){
    int i;
    if (N == MAXSIZE)
        return -1;
    for (i = N - 1; i >= pos; i--)
    {
        strcpy(list[i + 1].word, list[i].word);
        list[i + 1].count = list[i].count;
    }
    strcpy(list[pos].word, w);
    list[pos].count = 1;
    N++;
    return 1;
}
```

在本程序中：
查找算法复杂度为$O(\log_2 n)$
插入算法复杂度为 $O(n)$

复习：全局（外部）变量

- 全局变量（global variable）：在函数外面定义的变量，又称外部变量
  - 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
  - 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）

外部变量的定义和声明

- 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()
{ …
```

谨慎使用外部变量

- 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
  - 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：词频统计程序分析

- 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？（考虑怎么实现？）

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$O(N^2)$；

数据结构与程序设计（信息类）

Data Structure and Programming

第 2 章

线 性 表(Linear List)

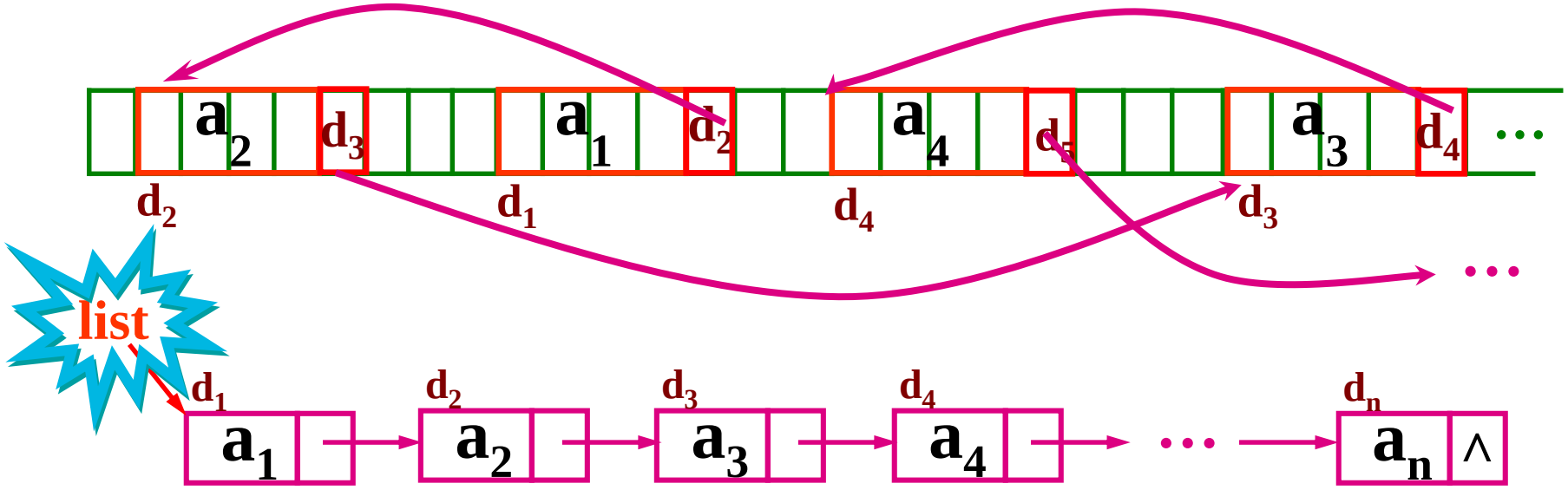
目录

CONTENTS

2.3 线性表的链式存储

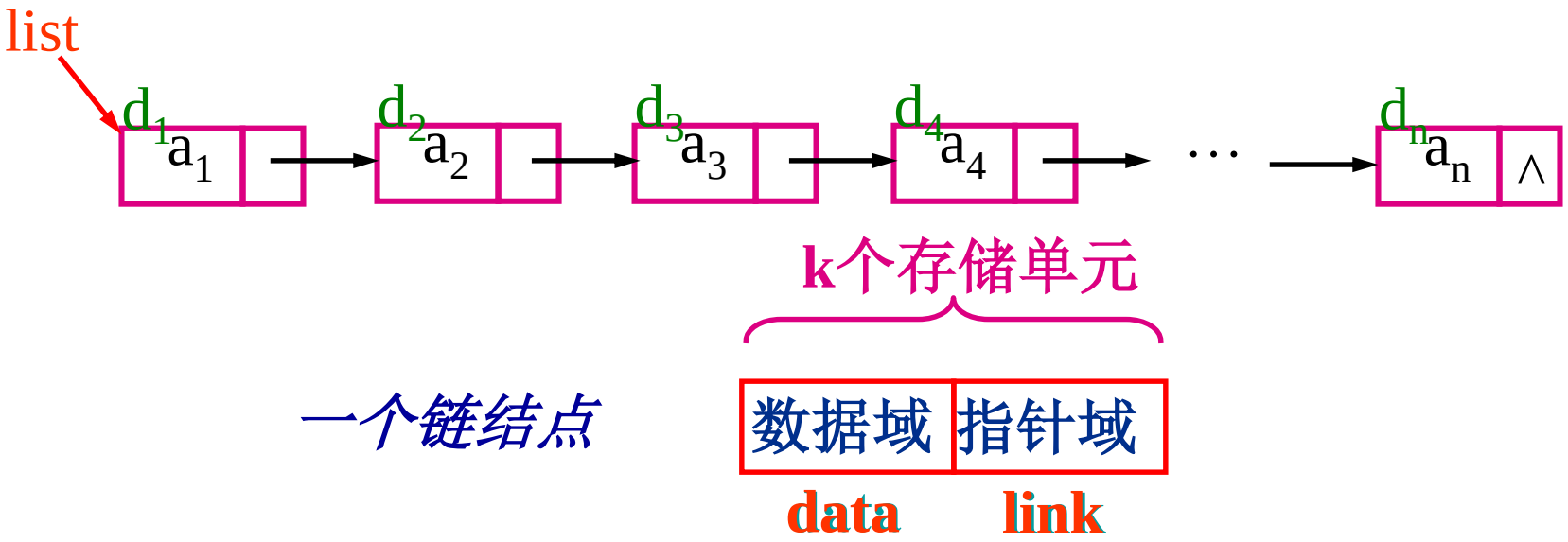
用一组地址任意的存储单元(连续的或不连续的)依次存储表中各个数据元素，数据元素之间的逻辑关系通过指针间接地反映出来。

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



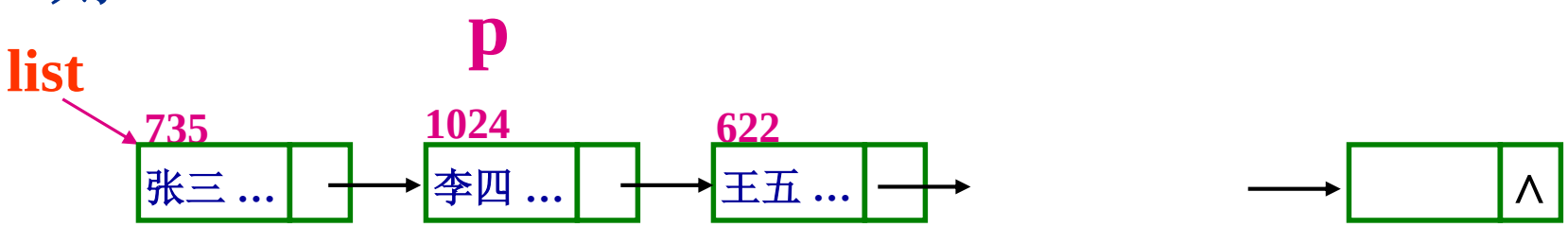
线性链表

- 线性表的这种存储结构称为线性链表，又称单（向）链表
- 一般形式



线性链表（续）

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



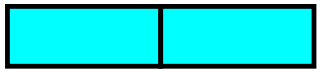
p->data 表示由p指向的链结点的数据域

X = p->data； X = “李四” ；

p->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）

线性链表的定义：自引用结构

链结点：



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;
```

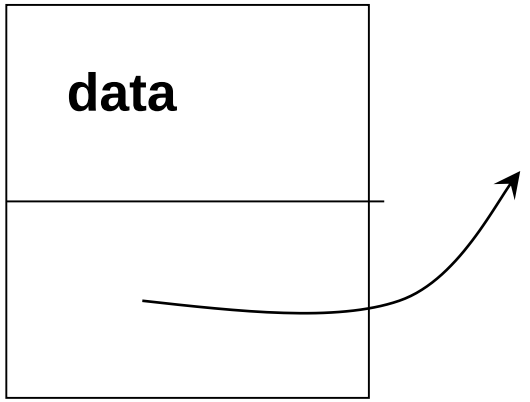
	Num	Name	Age
a_1	60101	张三	17
a_2	60102	李四	16
a_3	60103	王五	20
a_{30}			

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

自引用结构

- 自引用结构其成员分为两部分：
  - 1. 各种实际数据成员
  - 2. 一个或几个指向自身结构的指针

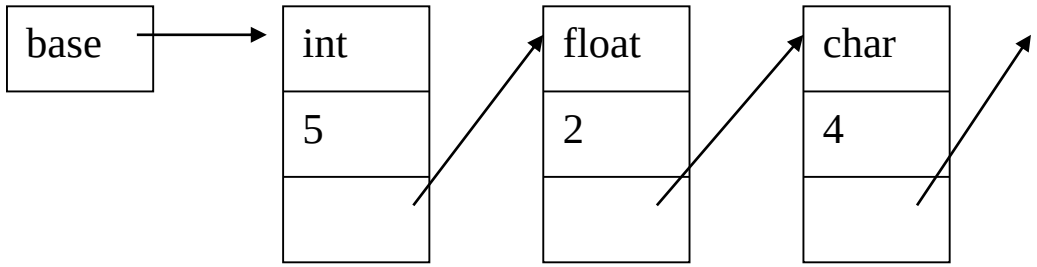
```
struct Type {
    data_member;  // 如 int n;
    struct Type *link;
};
```



自引用结构（续）

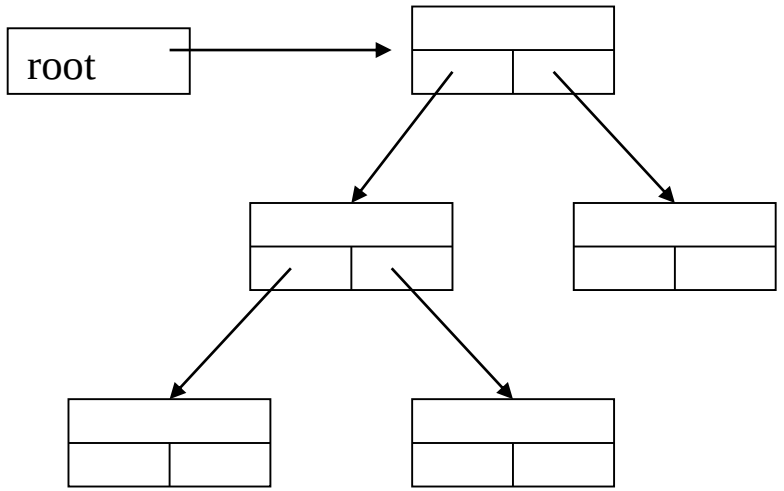
- 链表结构

```
struct  word {
        char *name;
        int count;
        struct word *next;
} *base;
```



- 二叉树结构

```
struct tnode {
        char *word;
        int count;
        struct  tnode *left;
        struct  tnode *right;
} *root;
```



数据结构与程序设计（信息类）

Data Structure and Programming

第 2 章

线 性 表(Linear List)

目录

CONTENTS

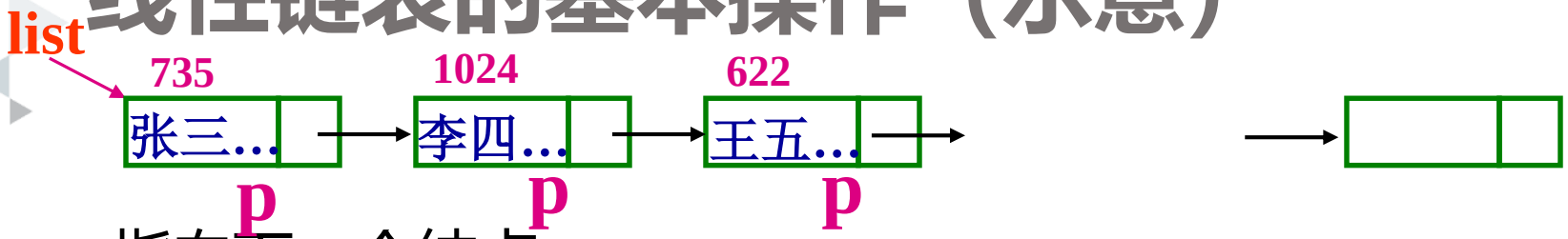
线性链表的基本操作

- 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
- ……

线性链表的基本操作（函数）

- 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

线性链表的基本操作（示意）



- 指向下一个结点：

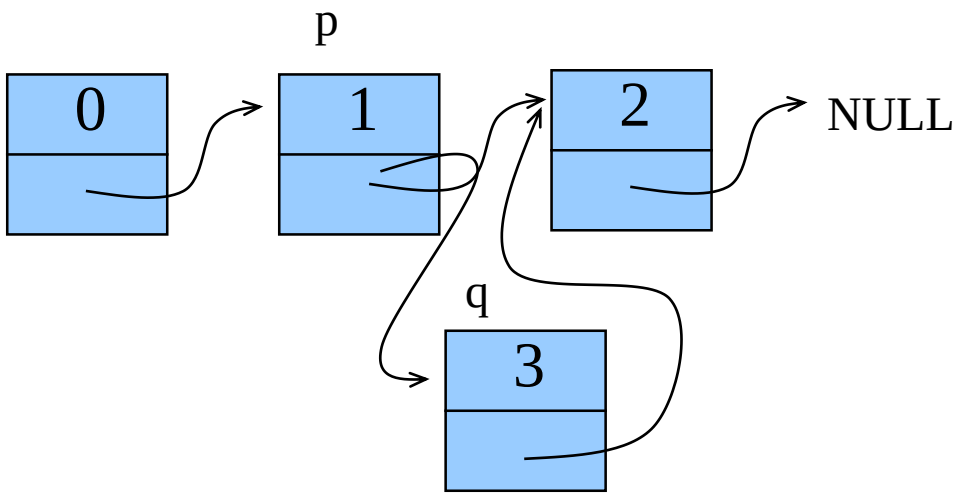
  p = p->link;

- 插入一个结点：

  q->link = p->link;

  p->link = q;

  (在p后插入q)



- 删除一个结点：

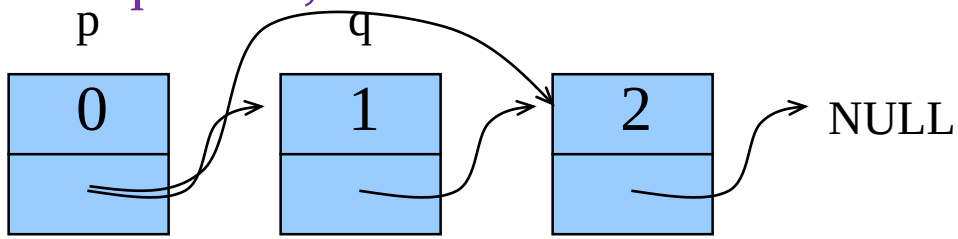
  q = p->link;

  p->link = p->link->link; 或 p->link = q->link;

  free(q);

  (删除p的下一结点)

为什么首先要执行：q = p->link?

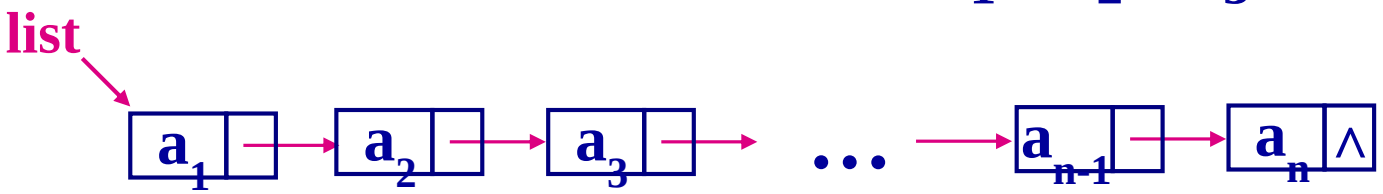


- 遍历一个链表：

  for(p=list; p != NULL; p=p->link)

  ….

1. 建立一个链表 $(a_1, a_2, a_3, \cdots, a_{n-1}, a_n)$



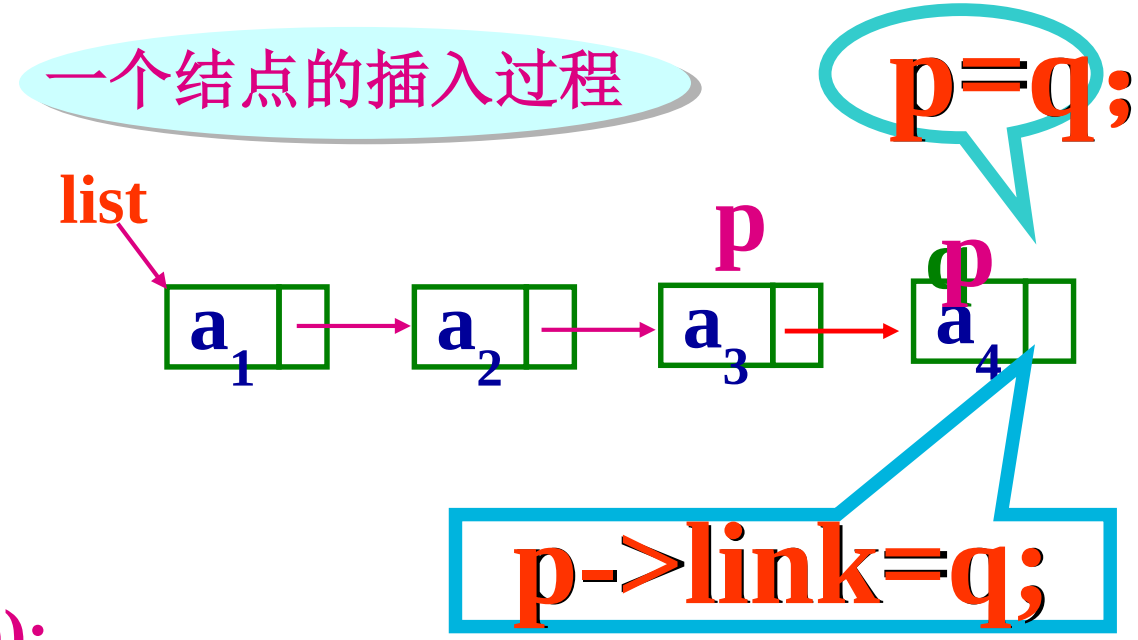

```
struct node{
        ElemType   data;
        struct node   *link;
} ;
typedef  struct node *Nodeptr;
typedef  struct node Node;
Nodeptr   list, p;
```

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);`

头文件：#include <stdlib.h>

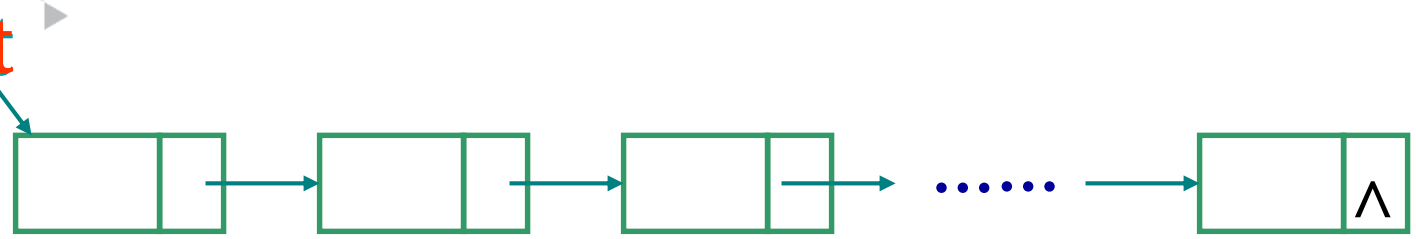


建立链表：C代码实现

```
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;
}
```

时间复杂度O(n)

2. 求链表的长度



初始：

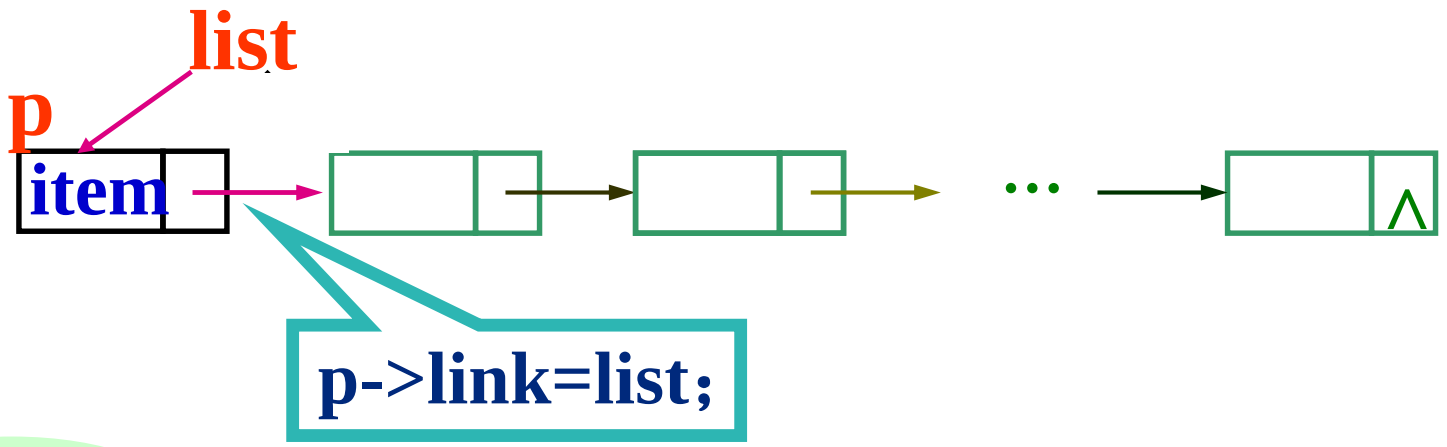
```
n=0;
p=list;

p=p->link;
 n++;
```

链表长度

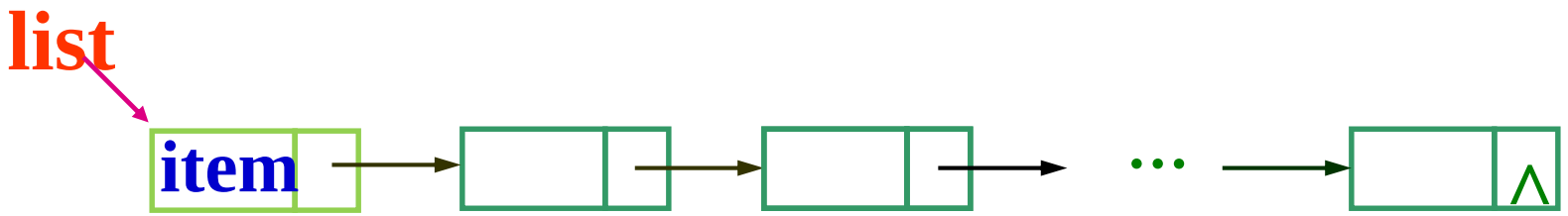
```
int getLength( Nodeptr list )
{
    Nodeptr p;      /* p为遍历链表结点的指针 */
    int n=0;                /* 链表的长度置初值0 */
    for(p=list; p!=NULL; p=p->link)
            /* p依次指向链表的下一结点 */
        n++;                /* 对链表结点累计计数 */

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

时间复杂度O(n)

3. 插入：在非空线性链表第一个结点前插入数据项



插入后



代码实现

```
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;          /* 将item赋给新结点数据域 */
    p->link=list;        /* 将新结点指向原链表第一个结点*/
    return  p;           /* 将链表新头指针返回 */
}
```

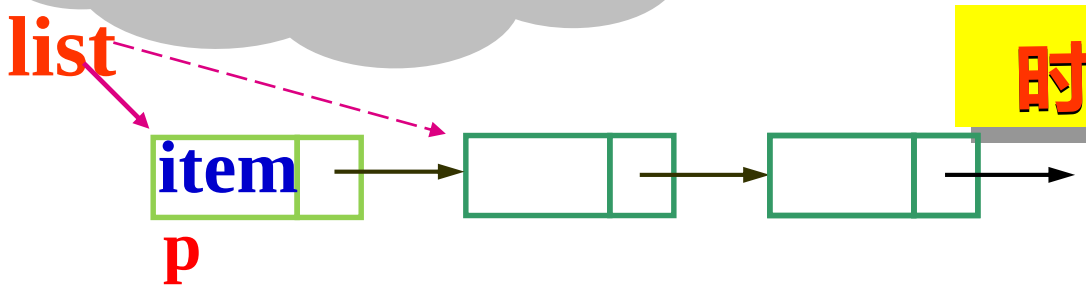
申请一个新结点

请思考一下为什么要返回p，而不用：list = p?

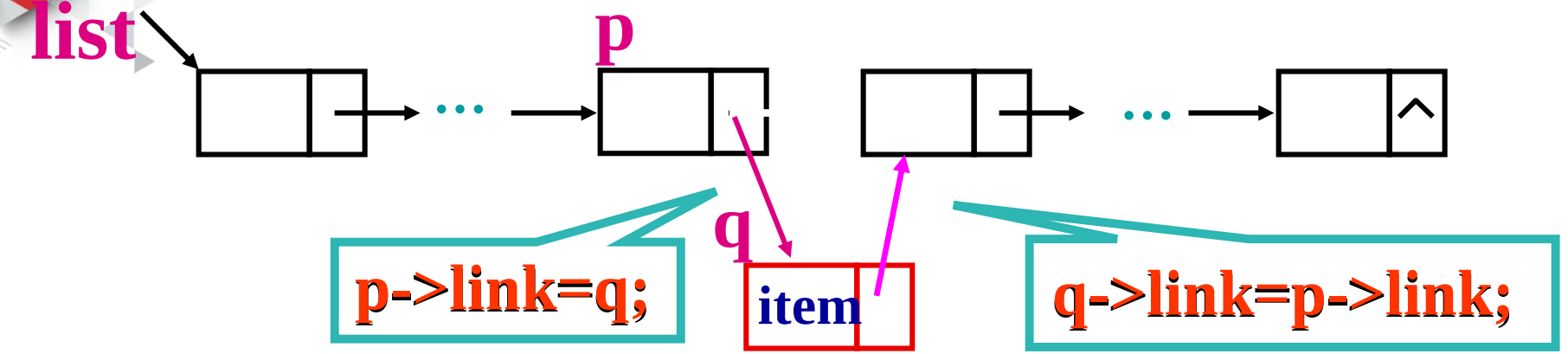
应使用如下方式调用insertFisrt函数：

list = insertFirst(list, item);

时间复杂度：O(1)



4.插入：在链表中由指针p指的链结点之后插入



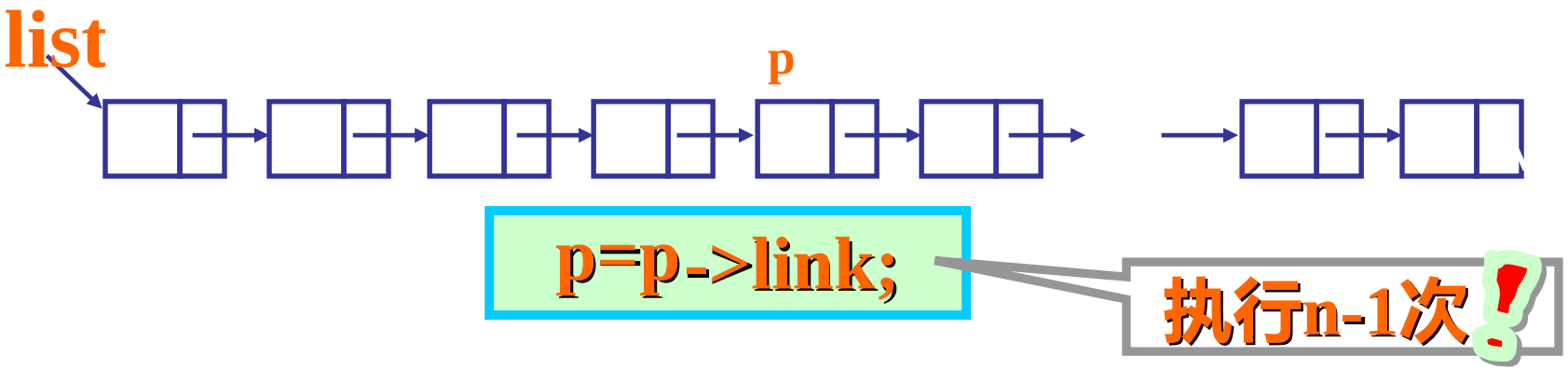

```
void  insertNode(Nodeptr p, ElemType item)
{
        Nodeptr q;

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

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

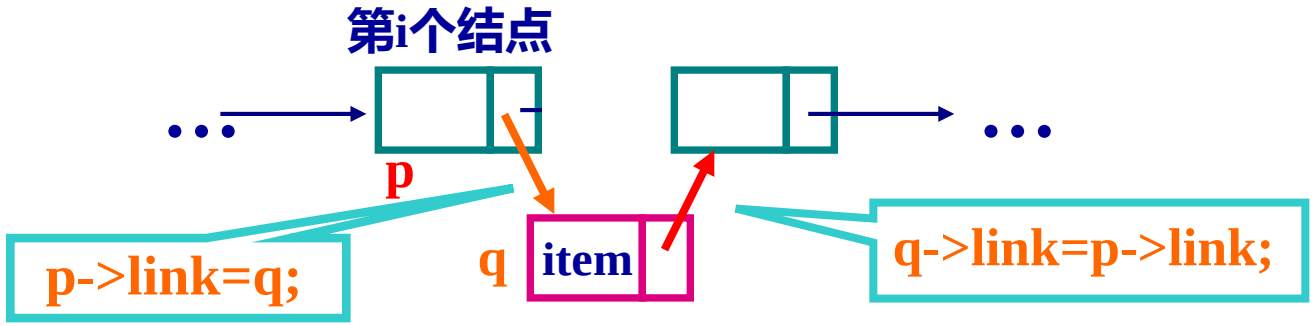
时间复杂度O(1)

5.插入：在链表中第n(n>0)个结点后面插入

寻找第n个结点

如何找到第i个结点？





算法实现

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

时间复杂度O(n)

5a.插入：在有序链表中相应结点后面插入

```
/* 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->data = elem;
    r->link = NULL;
    if (list == NULL) /* list是一个空表 */
        return r;
    for (p = list; p != NULL && elem > p->data; q = p, p = p->link) /* 找到插入位置 */
        ;
    if (p == list)    { /* 在头指针前插入 */
        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;
}
```

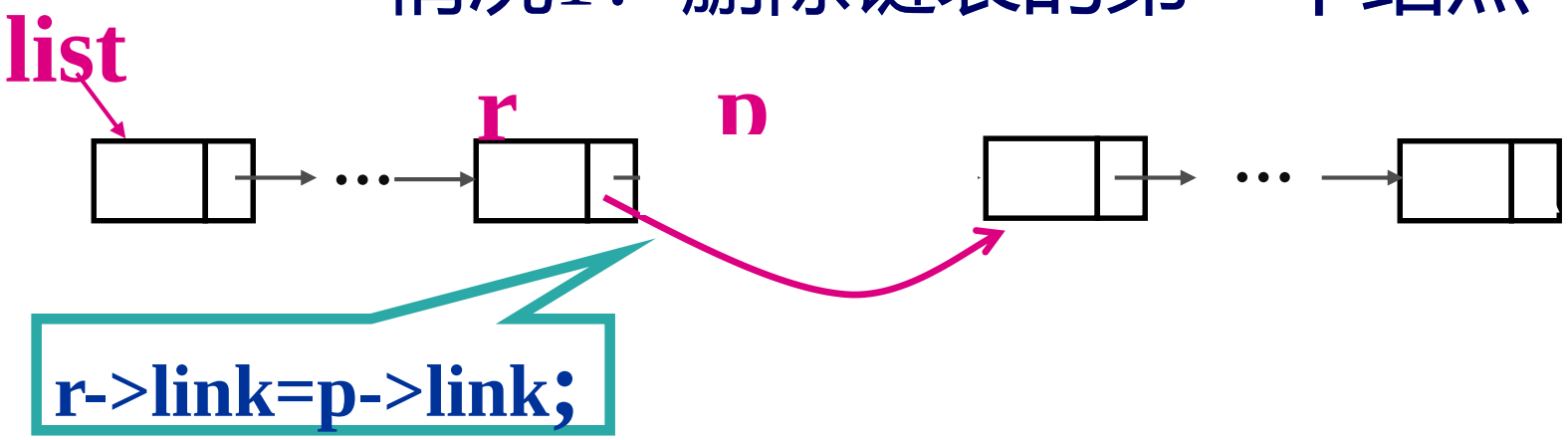
1.在结点p前插入一个结点，必须要知道该结点的前序结点指针，在本程序中，q为p的前序结点指针；
2. 应使用如下方式调用insertNode函数：
list = insertNode(list, item);

时间复杂度O(n)

6.删除：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

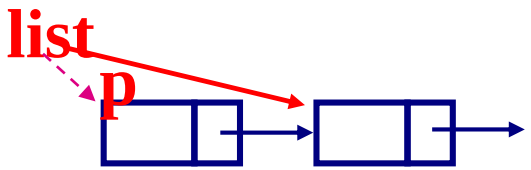
情况1：删除链表的第一个结点



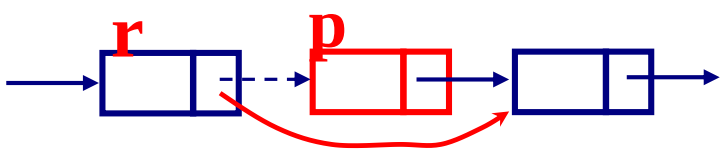
情况2：删除链表中非第一个结点

算法实现

```
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;
}
```



(情况1)

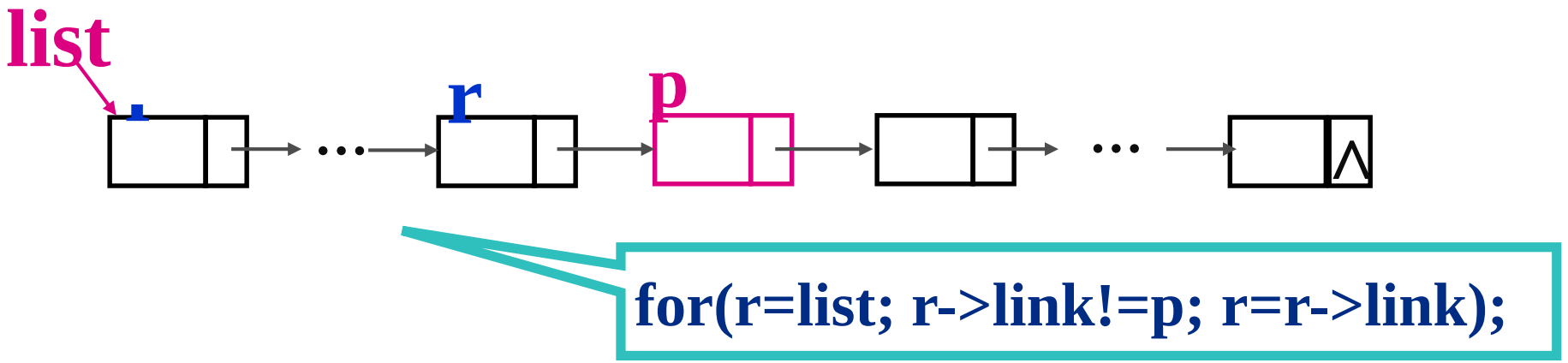


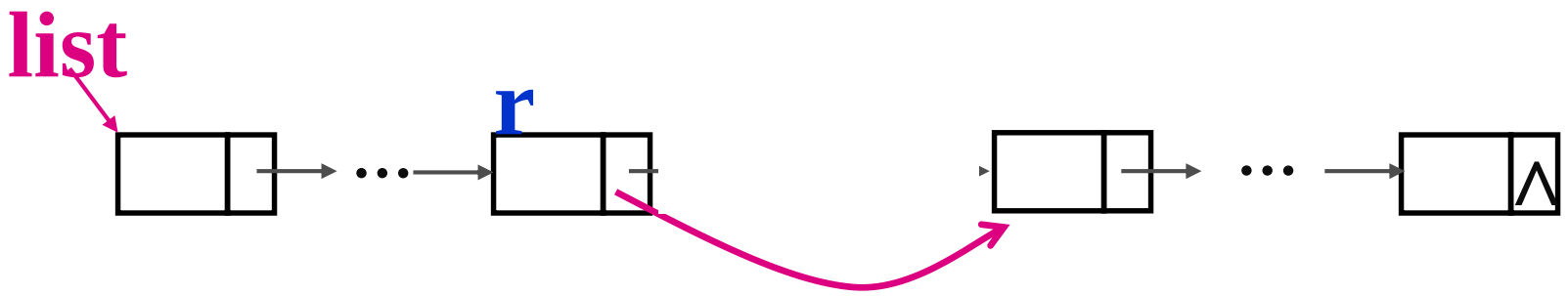
(情况2)

时间复杂度O(1)

7. 删除：从非空链表中删除p指向的结点

设r是p的直接前驱结点指针





代码实现

```
Nodeptr deleteNode2( Nodeptr list, Nodeptr p )
{
    Nodeptr r;
    if(p==list){                          /*当删除链表第一个结点*/
        list=list->link;
        free(p);                          /*释放被删除结点的空间*/
     }
    else{
        for(r=list; r->link!=p  &&  r->link!=NULL; r=r->link)
            ;                             /*移向下一个链结点*/
        if(r->link!=NULL){
            r->link=p->link;
            free(p);
         }
    }
    return list;
}
```

时间复杂度O(n)

寻找p结点的直接前驱r

8. 删除：从链表中删除包含给定元素的结点

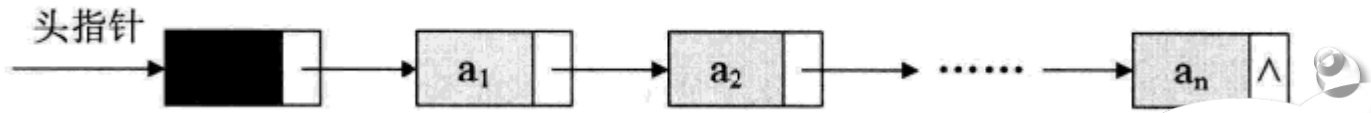
删除链表中值等于给定值的结点（仅删除第一个）

```
Nodeptr deleteNode(Nodeptr list, ElemType elem){
    Nodeptr p, q; // p指向删除的结点，q为p的前一个结点
    if (list == NULL)
        return list; //链表为空
    if (p = list; p != NULL; q = p, p = p->link)   {
        if (p->elem == elem)
            break;
    }
    if (p == list)   { //删除第一个结点
        list = list->link;
        free(p);
    }
    else if (q->link != NULL)  { //删除p指向的结点
        q->link = p->link;
        free(p);
    }
    return list;
}
```

补充问题：头结点

- 由于第一个结点标识着整个链表，在链表操作时， 针对第一个结点的操作意味着一定要修改链表的头指针
  - 必须要考虑是否是第一个结点，而进行不同的处理
- 可以在链表中增加一个单独的头结点，头指针指向头结点，头结点一旦建立后，不再进行任何修改操作。

带头结点的非空链表



带头结点的空链表

请思考这样做有什么好处？

设置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（不用单独考虑是否需要修改头指针）。其数据域一般无意义（也可存放链表的长度）

数据结构与程序设计（信息类）

Data Structure and Programming

第 2 章

线 性 表(Linear List)

目录

CONTENTS

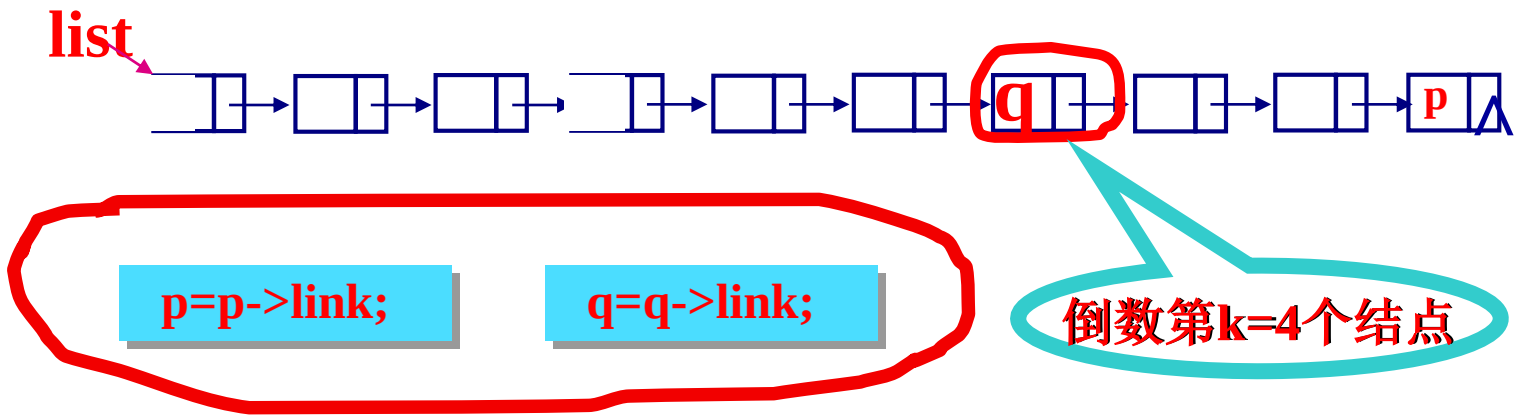
实例：2009年硕士研究生入学考试题

请写一算法，该算法用尽可能高的时间效率找到由list所指的线性链表的倒数第k个结点。若找到这样的结点，算法给出该结点的地址，否则，给出NULL

限制

1. 算法中不得求出链表长度；

2.不允许使用除指针变量和控制变量以外的其他辅助空间。



实现思路

1. 设置一个指针变量p，初始时指向链表的第1个结点；

2. 然后令p后移指向链表的第k个结点；

3. 再设置另一个指针变量 q，初始时指向链表的第1个结点；

4. 利用一个循环让p与q同步沿链表向后移动；当p指向链表最后那个结点时，q指向链表的倒数第k个结点

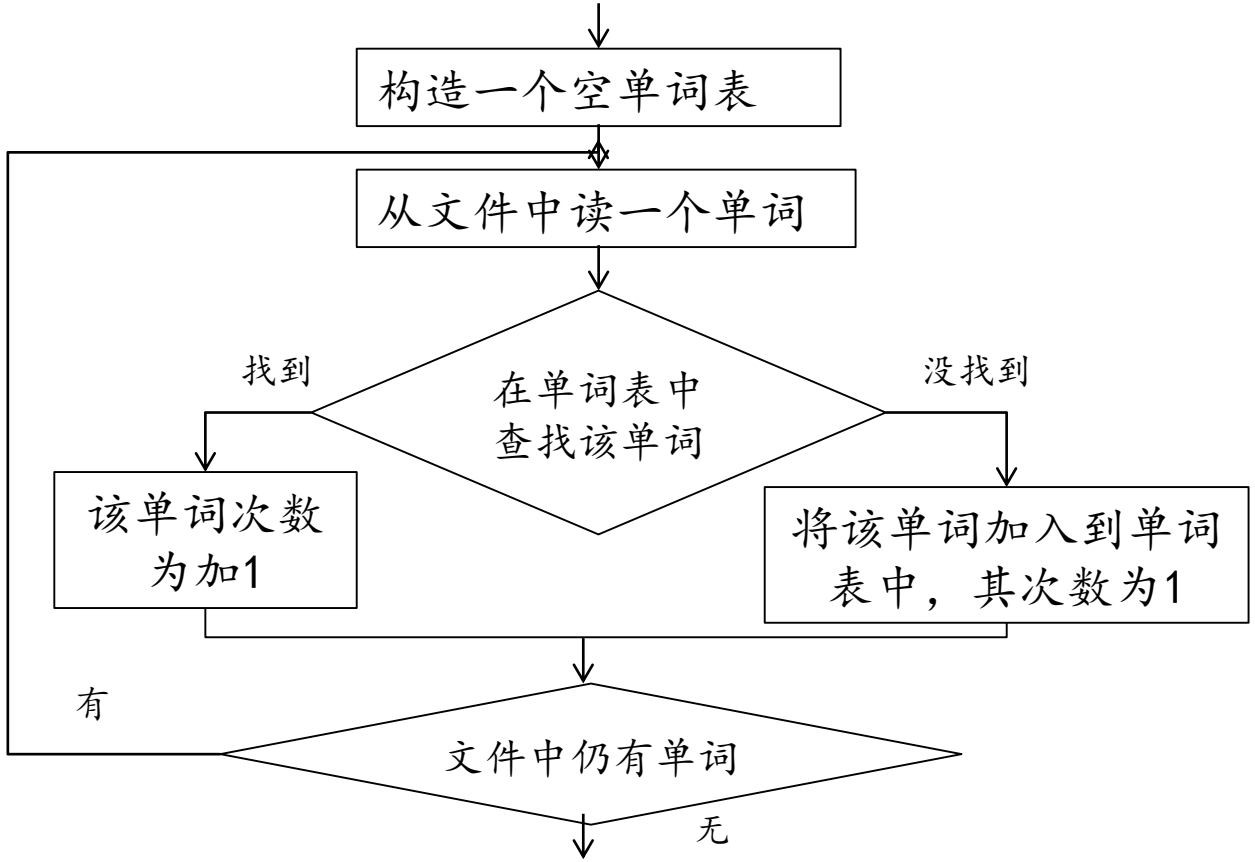
算法实现

```
Nodeptr searchNode(Nodeptr list,int k) {
      Nodeptr p,q;
      int i;
      if(list!=NULL && k>0){
      p=list;
      for(i=1;i<k;i++){ /*  循环结束时，p指向链表的第k个结点*/
            p=p->link;
            if(p==NULL){
                  printf("链表中不存在倒数第k个结点！")
                  return NULL;
            }
      }

     /*p指向链表最后那个结点，q指向倒数第k个结点*/
     for(q=list; p->link!=NULL; p=p->link,q=q->link);

    return q; /*给出链表倒数第k个结点(q指向的那个结点)的地址 */
}
```

O(n)

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）

代码实现

```c
#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
//单词表头指针，使用全局变量
struct node *Wordlist = NULL;
int getWord(FILE *bfp,char *w);
int searchWord(char *w);
int insertWord( struct node *p, char *w);
```

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

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

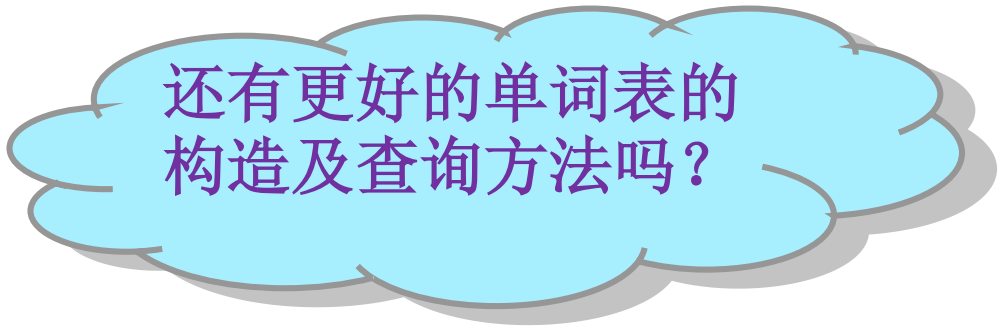
代码实现

```c
/*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;
}
```

```c
/*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)
{
    //q为p的前序结点指针
    struct node *p, *q = NULL;
    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);
}
```

特点分析

- 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



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 $a^nx^n + a^{n-1}x^{n-1} + \ldots + a^1x^1 + a^0x^0$ 的输入方法如下： a^n n a^{n-1} n-1 … 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：

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

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

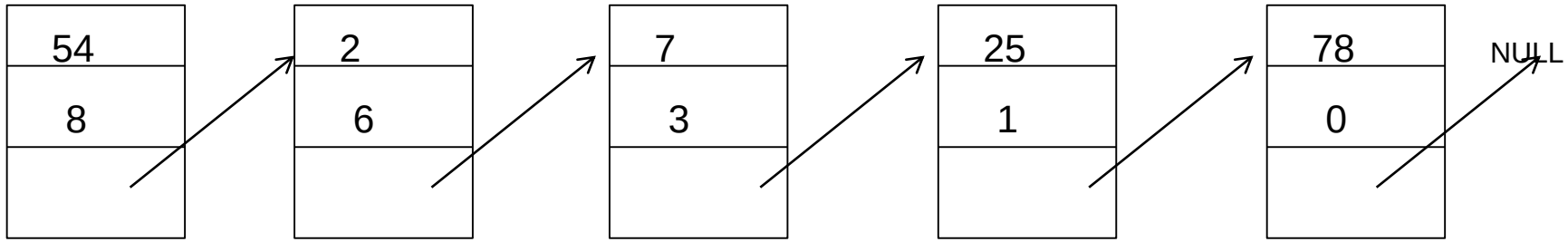
  其和为： $54x^8 + 43x^7 + 2x^6 + 7x^3 + 4x^2 + 33x + 78$

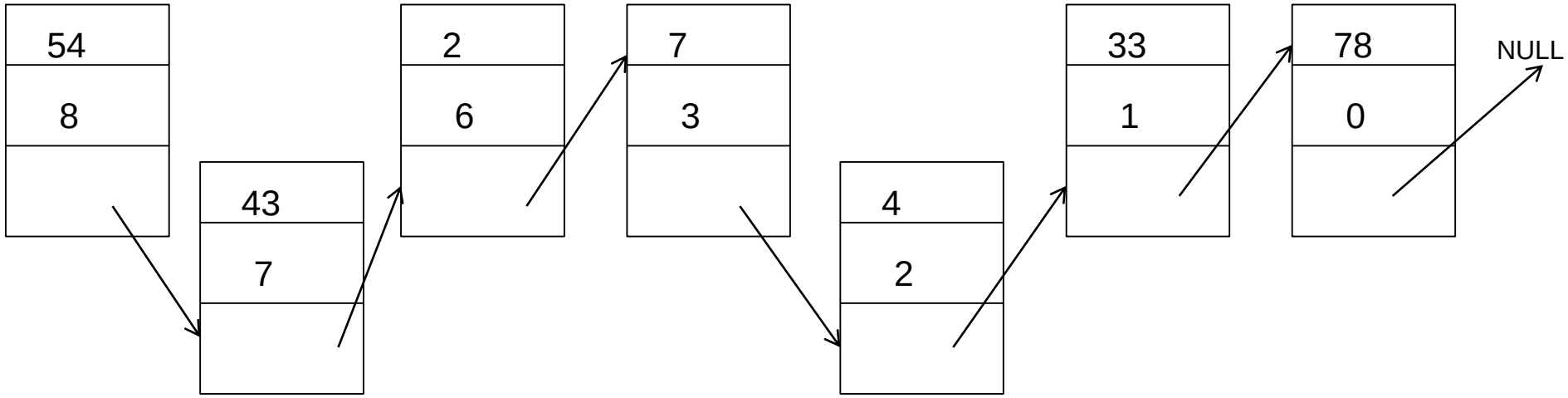
算法设计

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

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

- 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





算法设计

- 将第二个多项式一个结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结点：系数直接相加
  - 第一个多项式中不存在相同指数的结点：插入到某个结点前
  - 第一个多项式中不存在相同指数的结点：插入到头结点前，或尾结点后

算法实现

```c
#include <stdio.h>
#include <stdlib.h>
struct  Node { //一个多项式结点结构
    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');
```

```c
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);
    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;
            } //插入到头结点前
            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插入到尾结点后
} while (c != '\n');
for (p = head; p != NULL; p = p->next)
    printf("%d  %d  ", p->coe, p->pow);
return 0;
}
```

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：时间复杂度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：时间复杂度O(n)

顺序结构和链式结构的比较

存储分配方式	时间性能	空间性能

• 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
	• 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)
		• 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（如单词表），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，用顺序结构次效率会高些

数据结构与程序设计（信息类）

Data Structure and Programming

第 2 章

线 性 表(Linear List)

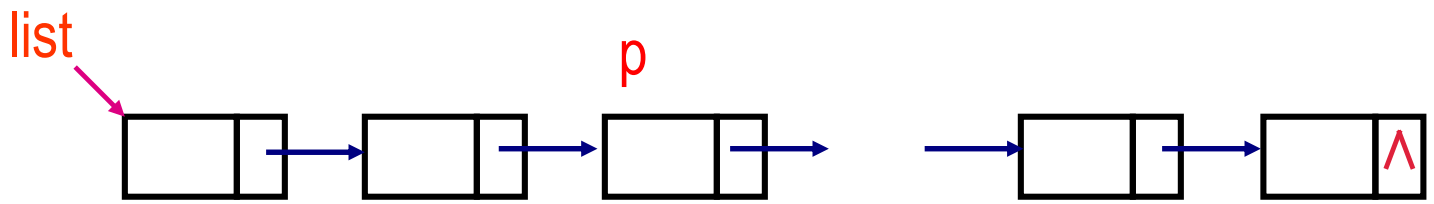
目录

CONTENTS

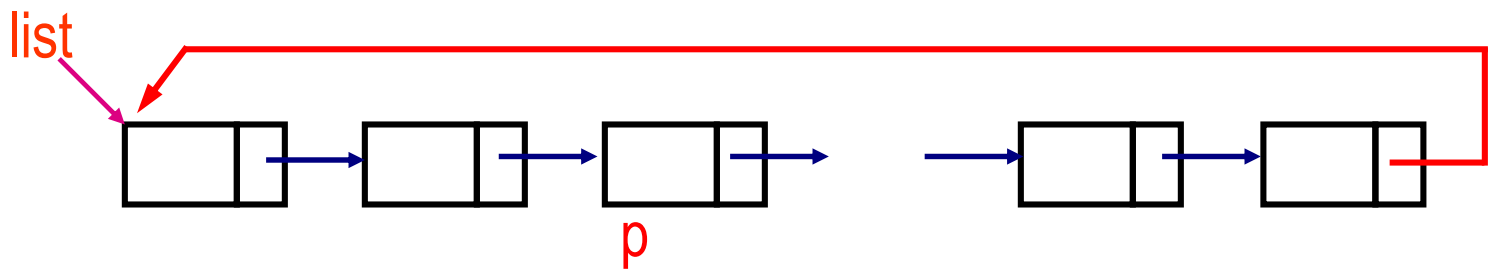
2.4 循环链表

- 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

线性链表



循环链表



求链表的长度：普通链表和循环链表

非循环链表

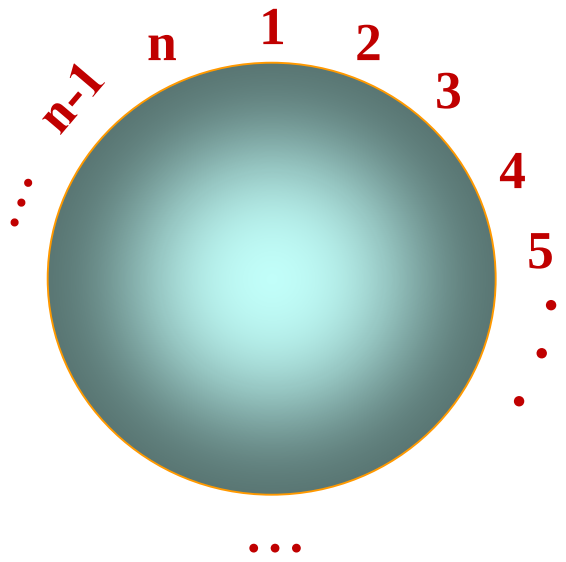
```
int  length( NodePtr list )
{
    Nodeptr p;
    int n;                          /* 链表的长度置初值0 */
    for(p=list,n=0; p!=NULL; p=p->link,n++)
        ;
    return n;                       /* 返回链表的长度n */
}
```

循环链表

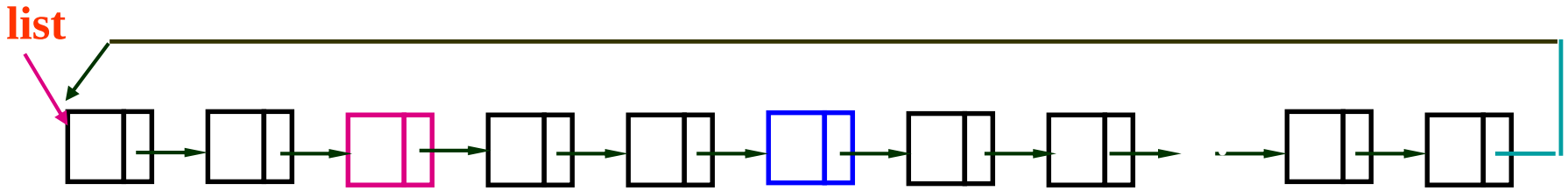
```
int  length( Nodeptr list )
{
    Nodeptr p=list;
    int n=0;                        /* 链表的长度置初值0 */
    if(list == NULL) return 0;
    do{
        p=p->link;
        n++;
    }while(p!=list);
    return n;                       /* 返回链表的长度n */
}
```

例：约瑟夫（Josephu）问题（圆桌问题）

已知n个人(不妨分别以编号1,2,3,…,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,…,依此重复下去，直到圆桌周围的人全部出列



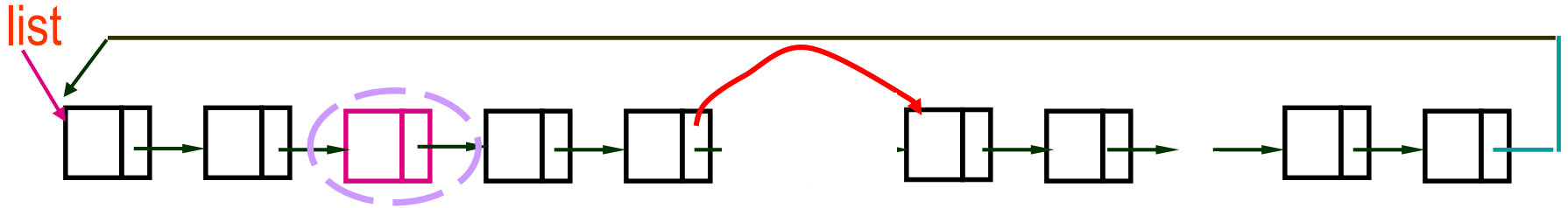
算法设计：使用不带头结点的循环链表



n：链表中链结点的个数（n>0）
k：第一个出发点
m：报数

需要完成的工作：
（1）根据n值，建立循环链表
（2）找到第一个出发点
（3）反复删除第m个链结点

若假设 k=3，m=4

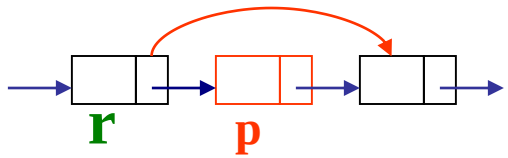


p=p->link;

k-1次　m-1次

代码实现

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



问题2.3：显示文件最后n行

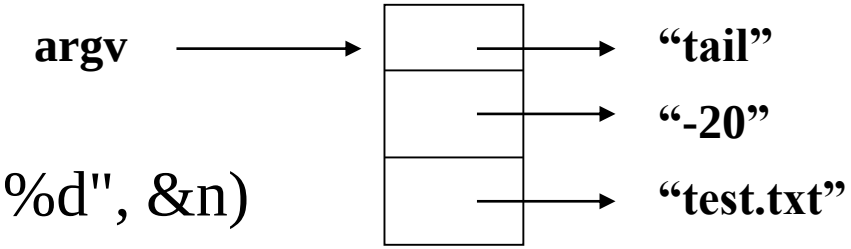
- 问题：命令tail用来显示一个文件的最后n行。其格式为：

  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
  - 说明：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
  - 示例：
    - 若从命令行输入:tail -20 test.txt，以显示文件test.txt的最后20行

问题分析

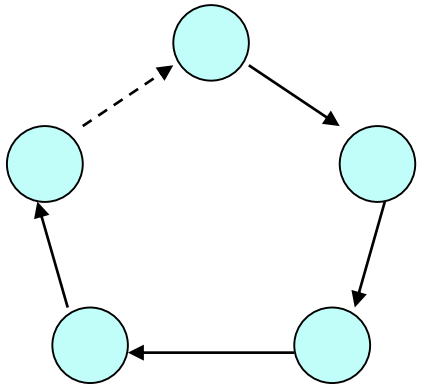
- 如何得到需要显示的行数和文件名？
  - 使用命令行参数
    - int main(int argc, char *argv[])
    - 行数 n = atoi(argv[1]+1) 或 sscanf(argv[1]+1, "%d", &n)
    - 文件名 filename = argv[2]



- 如何得到最后n行？
  - 文本文件每行的字符数是不确定的，只能从前往后顺序读写
  - 同时，一次读写完成后，并不能退回到指定的行重新读写

算法设计

- 方法一：使用n个结点的循环链表。链表中始终存放最近读入的n行
  - 首先创建一个空的循环链表；
  - 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



代码实现

```
#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

代码实现（续）

```
if((fp = fopen(filename, "r")) == NULL){
    printf(" Cann'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

将链表的最后一个结点指向头结点，以构成一个循环链表。

代码实现（续）

```
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)

其他方法

- 方法：两次扫描文件
  - 第一遍扫描文件，用于统计文件的总行数N；
  - 第二遍扫描文件时，首先跳过前面N-n行，只读取最后n行。
  - 如何开始第二遍扫描？
    - fseek(fp, 0, SEEK_SET); -- 将文件读写位置移至文件头
    - 或（关闭后）再次打开同一个文件

请同学们考虑是否还有其它方法？甚至更好的方法？

如：我们可设计这样两个函数：

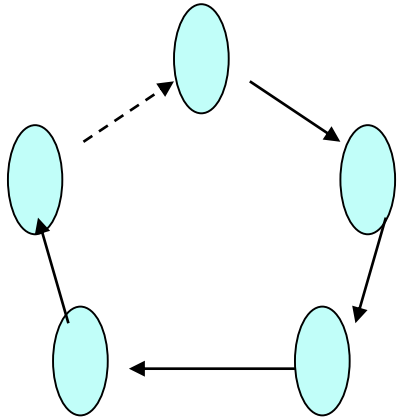
```
fbgetc(fp); //从文件尾开始，每次倒读一个字符
fbgets(buf,size,fp); //从文件尾开始每次倒读一行
```

问题2.4：谁是幸运者（海盗游戏，Josephu）

- 问题：海盗们曾经玩一种游戏：每当捕获一艘船，船上船员只有一人能幸存。规则如下：
  - 船上船员分别编上号，站成一圈；然后从1号船员开始循环数数，每数到13，该船员将被推到海里，直到剩下一个船员。
  - 谁是最后的幸存者？
- 输入：船员人数
- 输出：幸运者号

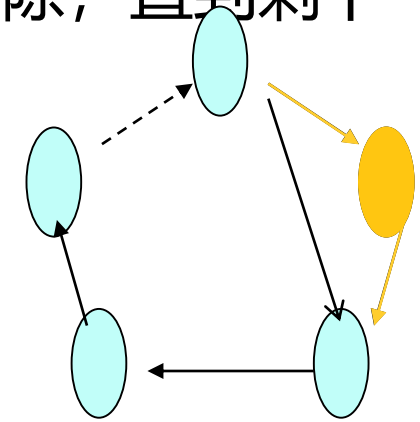
算法设计

- 首先将所有船员按编号构成一个循环链表；



```
first = p = (struct Node *)malloc(sizeof ( struct Node));
first->no = 1;
for(i=2; i<=n; i++){
    p->next = (struct Node *)malloc(sizeof ( struct
Node));
    p = p->next;
    p->no = i;
}
p->next = first;
p= first;
```

- 从循环链表头开始以13为周期循环计数访问，每计数到13，将相应结点删除，直到剩下一个结点；



```
c=1;
while(p != p->next)
   if(c != 13) {
      q = p; /*保存上一结点*/
      p = p->next;
      c++;
   }
   else {
      q->next = p->next; free(p);
      p = q->next;  c = 1;
   }
```

删除一个结点的要点是要保存被删结点的上一个结点

代码实现

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
  int no;
  struct Node *next;
};
int  main(){
    struct Node *first,*p,*q;
    int i,n,c=1;
    scanf(“%d”, &n);

    /*构造一个n个结点的循环链表*/
    /*以13为周期循环计数访问、删除

    printf(“%d\n”, p->no);
    return 0;

}
```

```
first = p = (struct Node *)malloc(sizeof ( struct Node));
first->no = 1;
for(i=2; i<=n; i++){
      p->next = (struct Node *)malloc(sizeof ( struct Node));
      p = p->next;
      p->no = i;
}
p->next = first;
p= first;
```

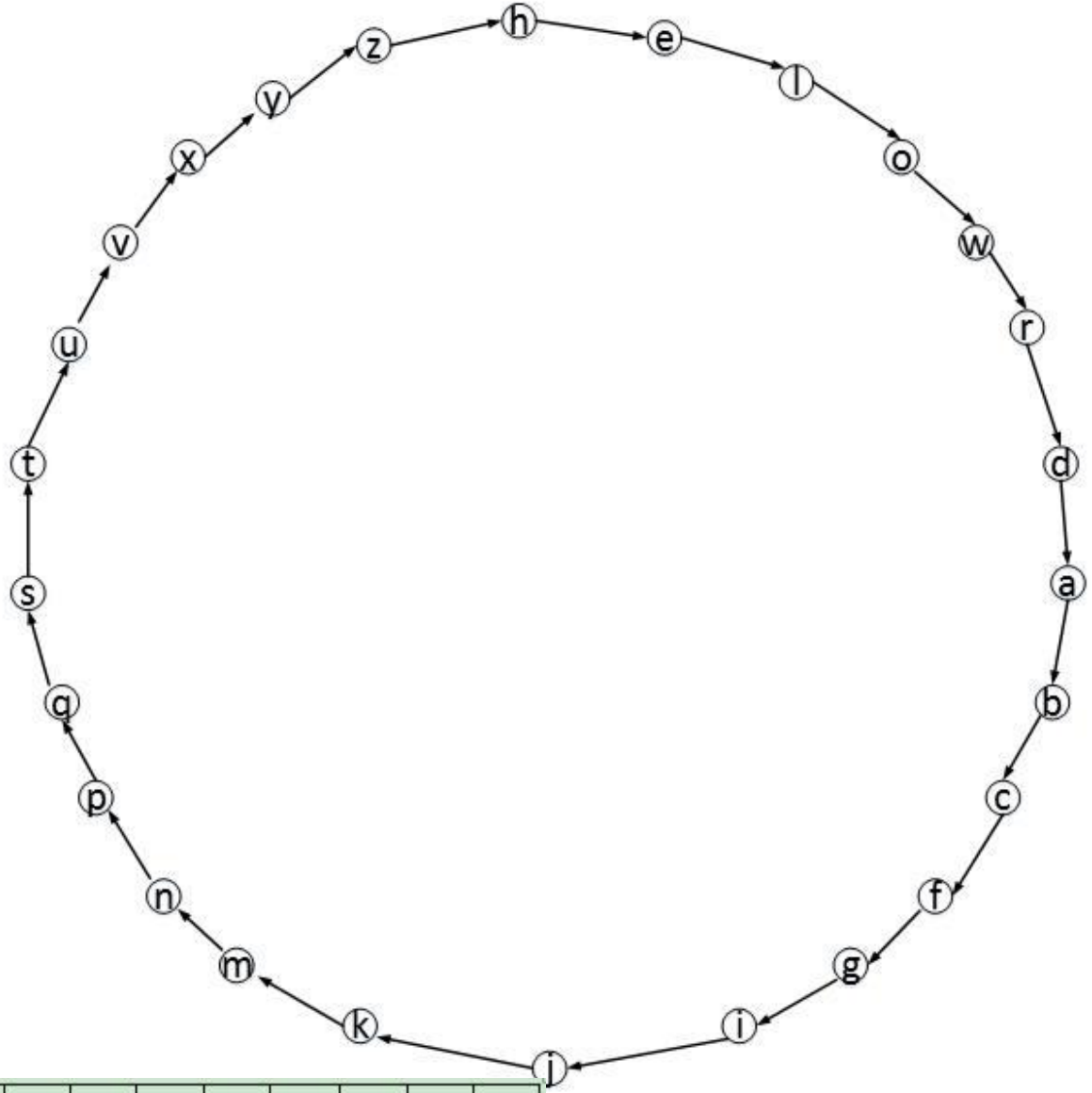
```
while(p != p->next)
    if(c != 13) {
        q = p;
        p = p->next;
        c++;
    }
    else {
        q->next = p->next;
        free(p);
        p = q->next;
        c = 1;
    }
```

作业题4：文件加密（环）

- 【问题描述】有一种文本文件加密方法，其方法如下：
  - 1. 密钥由所有ASCII码可见字符（ASCII码编码值32-126为可见字符）组成，长度不超过32个字符；
  - 2. 先将密钥中的重复字符去掉，即：只保留最先出现的字符，其后出现的相同字符都去掉；
  - 3. 将不含重复字符的密钥和其它不在密钥中的可见字符（按字符升序）连成一个由可见字符组成的环，密钥在前，密钥的头字符为环的起始位置；
  - 4. 设原密钥的第一个字符（即环的起始位置）作为环的开始位置标识，先从环中删除第一个字符（位置标识则移至下一个字符），再沿着环从下一个字符开始顺时针以第一个字符的ASCII码值移动该位置标识至某个字符，则该字符成为第一个字符的密文字符；然后从环中删除该字符，再从下一个字符开始顺时针以该字符的ASCII码值移动位置标识至某个字符，找到该字符的密文字符；依次按照同样方法找到其它字符的密文字符。当环中只剩一个字符时，则该剩下的最后一个字符的密文为原密钥的第一个字符
- 【输入形式】
  - 密钥是从标准输入读取的一行字符串，可以包含任意ASCII码可见字符（ASCII码编码值32-126为可见字符），长度不超过32个字符
- 【输出形式】
  - 加密后生成的密文文件为当前目录下的in_crpyt.txt

环加密

- 示例
  - ◆如果密钥为：helloworld，将密钥中重复字符去掉后为：helowrd，将不在密钥中的小写字母按照升序添加在密钥后，即形成字符串：helowrdabcfgijkmnpqstuvxyz，形成的环如图
  - ◆明码的第一个字母为h，h也是环的起始位置。h的ASCII码制为104，先把h从环中删除，再从下一个字母e开始顺时针沿着环按其ASCII值移动位置标识（即：在字母e为移动第1次，共移动位置标识104次）至字母w，则h的密文字符为w



a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
q	g	k	c	d	r	o	w	x	u	t	s	f	i	m	b	z	a	y	v	n	j	l	h	p	e

算法设计要点

- 1. 去除重复字符
  - 一个字符一个字符读入，如果已经存在，则不保存
- 2. 建立循环链表
  - 逐一申请新结点，添加到链表末尾
  - 最后一个结点链接到第一个结点
- 3. 删除链结点
  - 需要保留记录前一个节点信息
  - 通过前一个结点直接链接到后一个结点，删除当前结点
- 4. 明文和密文对应关系
  - 采用数组，明文字符的ASCII作为数组下标，内容为密文

总体设计

1. 定义结构体对象，存储字符串环；
2. 读入密钥，并去重；（函数：removeDup）
3. 根据密钥生成字符串表，补充不在密钥中的字符；
4. 根据字符串表，构建字符串环（循环链表）；（函数：buildCircle）
5. 根据字符串环，依次删除元素，生成密钥对；（函数：getPasswordTab）
6. 利用密钥对，加密原始文件；（函数：encodeFile）

```
//定义结构体，存储数据
typedef struct _Node{
        char ch;
        struct _Node *link;
}Node, *PNode;
```

```
void removeDup(char *key);//字符串去重
PNode buildCircle(char key[], char allchar[]);//根据密钥生成字符串
void getPasswordTab(PNode list, char passtab[]);//计算密钥对
void encodeFile(char tab[],char *src, char *dest);//加密文件
```

参考代码

```
int main(){
    char key[40], passtab[95];
    int i;
    PNode list = NULL;
    gets(key);
    removeDup(key); //去除重复字符
    for (i = 0; i < 95; i++)
        passtab[i] = i + ' '; //生成全部字符串表
    for (i = 0; key[i] != '\0'; i++)
        passtab[key[i] - ' '] = '\0'; //删除密钥中的字符
    list = buildCircle(key, passtab); //根据密钥和剩余的字符串构造字符串环

    getPasswordTab(list, passtab); //生成密钥对

    encodeFile(passtab, "in.txt", "in_crpyt.txt"); //利用密钥对加密文件

    return 0;
}
```

参考代码（续

```
//去除密钥中的重复字符
void removeDup(char *key) {
    char temp[40];
    int i, j, len=0;
    for(i=0;key[i]!='\0';i++){
        for(j=0;j<len;j++){
            if(temp[j]==key[i])
               break;
        }
        if(j==len){
            temp[len]=key[i];
            len++;
        }
    }
    temp[len]='\0';
    strcpy(key, temp);
}
```

```
//根据密钥和剩余的字符构建字符串环，返回环中最后一个地址
PNode buildCircle(char key[], char allchar[]){
    PNode list=NULL;//第一个结点
    //p当前结点，q前一个节点
    PNode p, q=NULL;
    int i;
    //生成密钥结点
    for(i=0;key[i]!='\0';i++){
        p = (PNode)malloc(sizeof(Node));
        p->ch = key[i];
        p->link = NULL;
        if(list){
            q->link=p;
            q=p;
        }else{
            q=list=p;
        }
    }
```

```
    for(i=0;i<95;i++){//生成剩余字符结点
        if(allchar[i]!='\0'){
            p=(PNode)malloc(
                        sizeof(Node));
            p->ch = allchar[i];
            p->link = NULL;
            if(list){
                q->link=p; q=p;
            }else{  q=list=p;}

        }
    }
    q->link = list; //链接成环
    list = q; //返回最后一个结点的地址
}
```

参考代码（续）

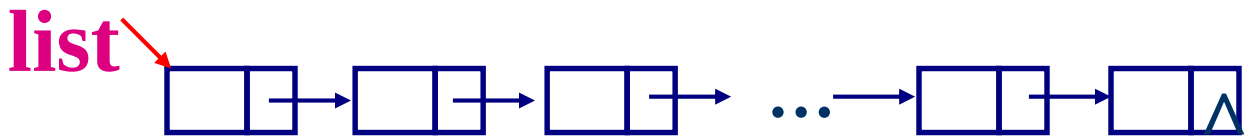
```
//根据字符串环，生成密钥对，保存到passtab中（注意：list指向最后一个结点）
void getPasswordTab(PNode list, char passtab[]){
    char src, first;
    PNode p, q; //p为当前节点，q为前一个结点
    int i=0;
    q = list; //list指向最后一个结点
    p = q->link;
    src = p->ch; //记录要删除的字符
    //第一个字符需保存，以作为最后一个字符密文
    first = src;
    while(p!=p->link){
        q->link = p->link;    free(p); //删除当前结点
        p = q->link; //指向下一个结点
        for(i=src-1;i>0;i--) {//求下一结点，注意循环次数
            q = p;       p = p->link;         }
        passtab[src-32] = p->ch; //删除结点的密文对应找到的结点
        src = p->ch; //记录新的结点字符
    }
    passtab[src - 32] = first; //最后一个字符的密文为第一个字符
}
```

参考代码（续）

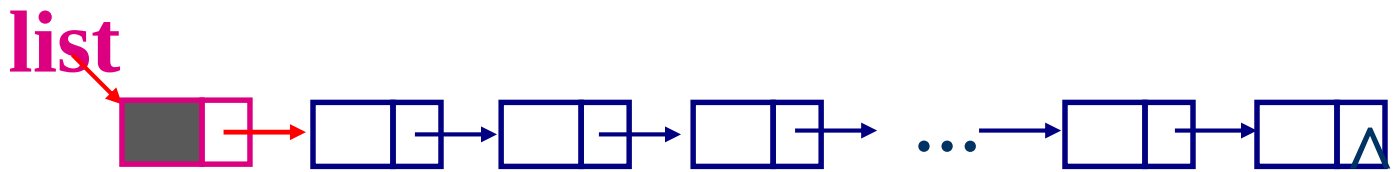
```
//利用密码表加密文件
void encodeFile(char tab[],char *src, char *dest) {
    FILE *in, *out;
    char ch, password; //原始字符和对应的密文
    in=fopen(src, "r");
    out=fopen(dest, "w");
    if(!in){
        printf("Can't open the file: %s", src);         return;    }
    if(!out){
        printf("Can't open the file: %s", dest);         return;    }
    while( (ch=fgetc(in) )!=EOF){   //读文件
        if(ch>=32 && ch<=126)   //只加密可见字符
            password = tab[ch - 32];
        else password = ch;
        fputc(password, out);   //写文件
    }
    fclose(in); //关闭文件
    fclose(out);
}
```

小结：线性表的链式存储

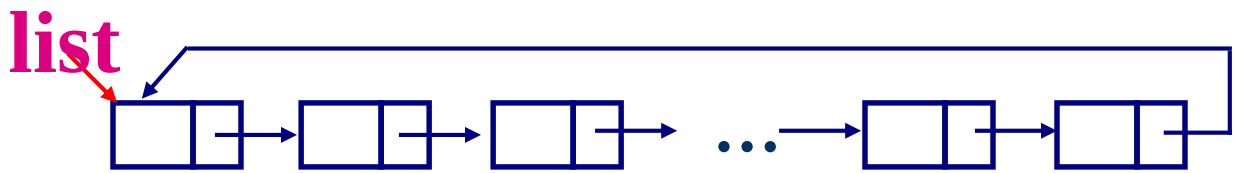
线性链表(单链表)



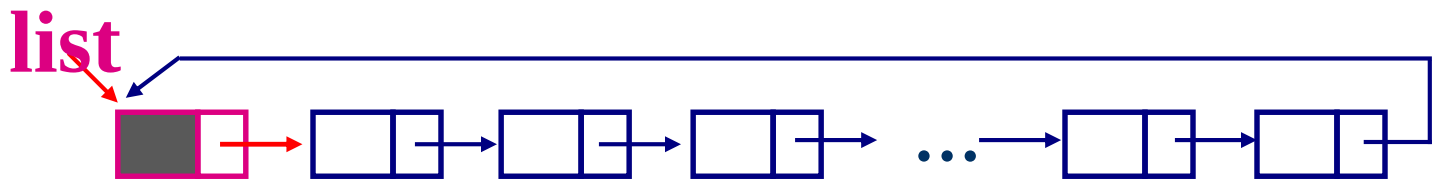
带头结点的线性链表



循环链表



带头结点的循环链表



数据结构与程序设计（信息类）

Data Structure and Programming

第 2 章

线 性 表(Linear List)

目录

CONTENTS

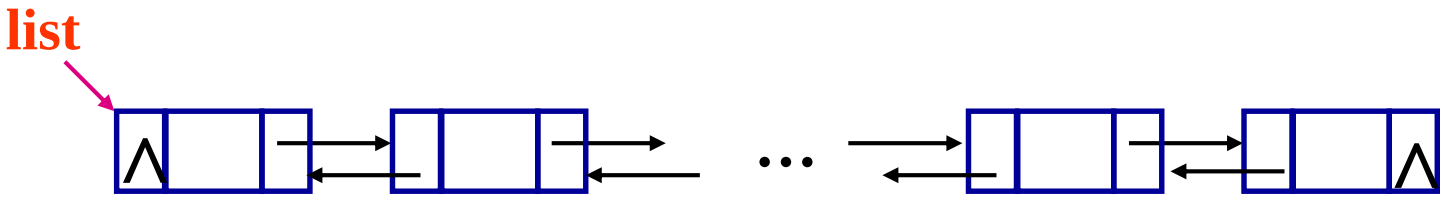
2.5 双向链表

- 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

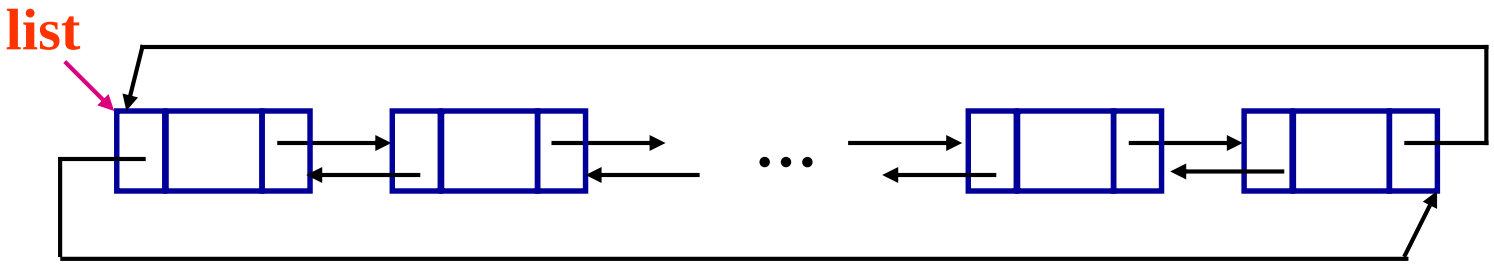

```
struct  node {
      ElemType  data;
      struct  node  *rlink, *llink;
} ;
typedef  struct node  *DNodeptr;
typedef  struct node DNode;
```

- 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（右指针和左指针）

双向链表的几种形式



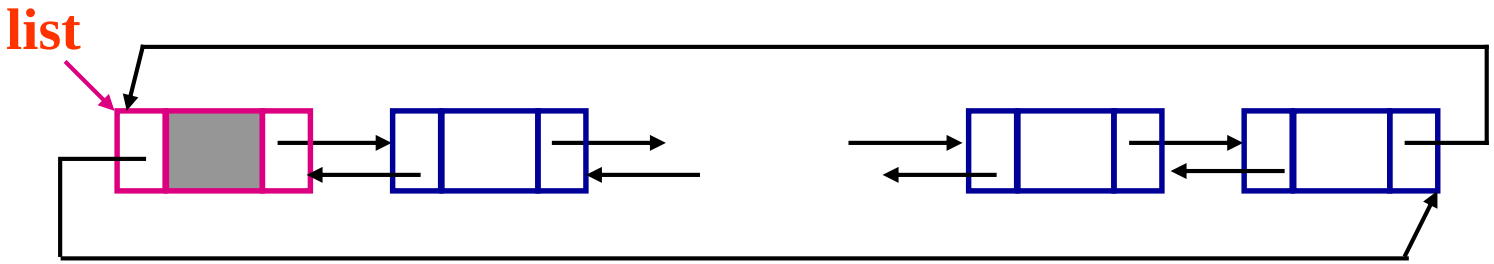
双向链表



双向循环链表

双向链表的插入

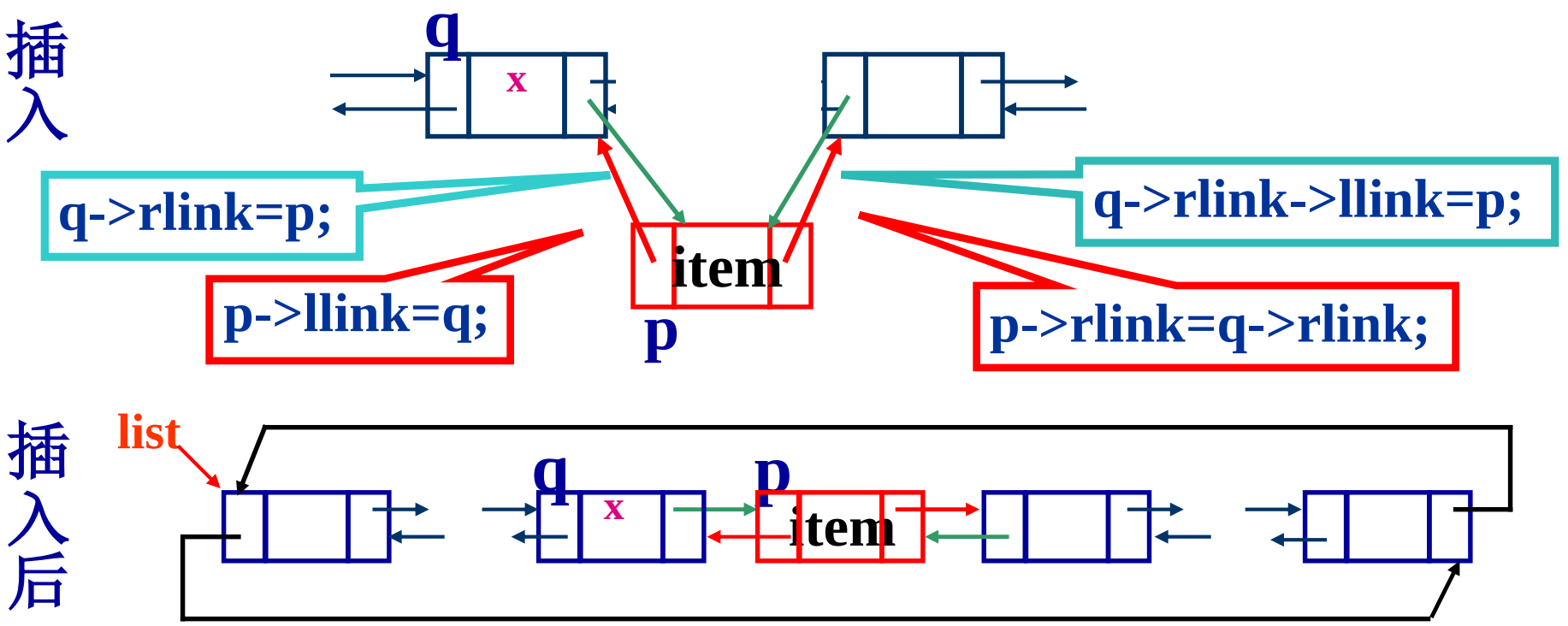
在非空双向循环链表中某个数据域的内容为x的链结点右边插入一个数据信息为item的新结点



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。

双向链表的插入



注意：在头(第一个)指针前插入一个结点时，步骤如下：

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

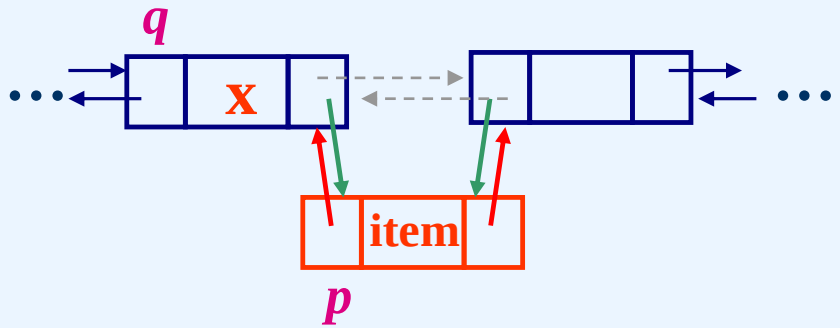
list->llink->rlink = p; list->llink = p; list = p;

算法实现

```
int  insertDNode(DNodeptr list, ElemType x, ElemType item)
{
     int DNodeptr  p,q;

   for(q=list; q && q->data!=x && q->rlink!=list; q=q->rlink)
        ;
   if(!q || q->data!=x)
        return -1;                          /* 没有找到满足条件的结点 */
     p=(DNoteptr)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 */
}
```

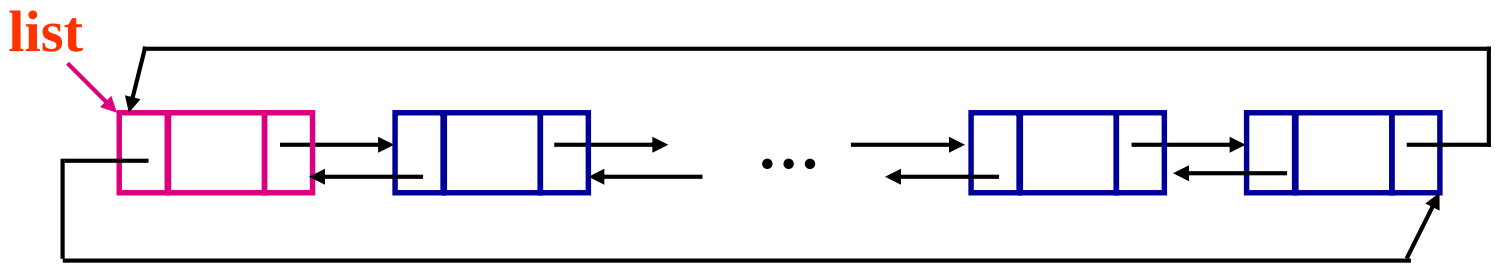
寻找满足条件的结点

O(1)



双向链表的删除

删除非空双向循环链表中数据域的内容为x的链结点

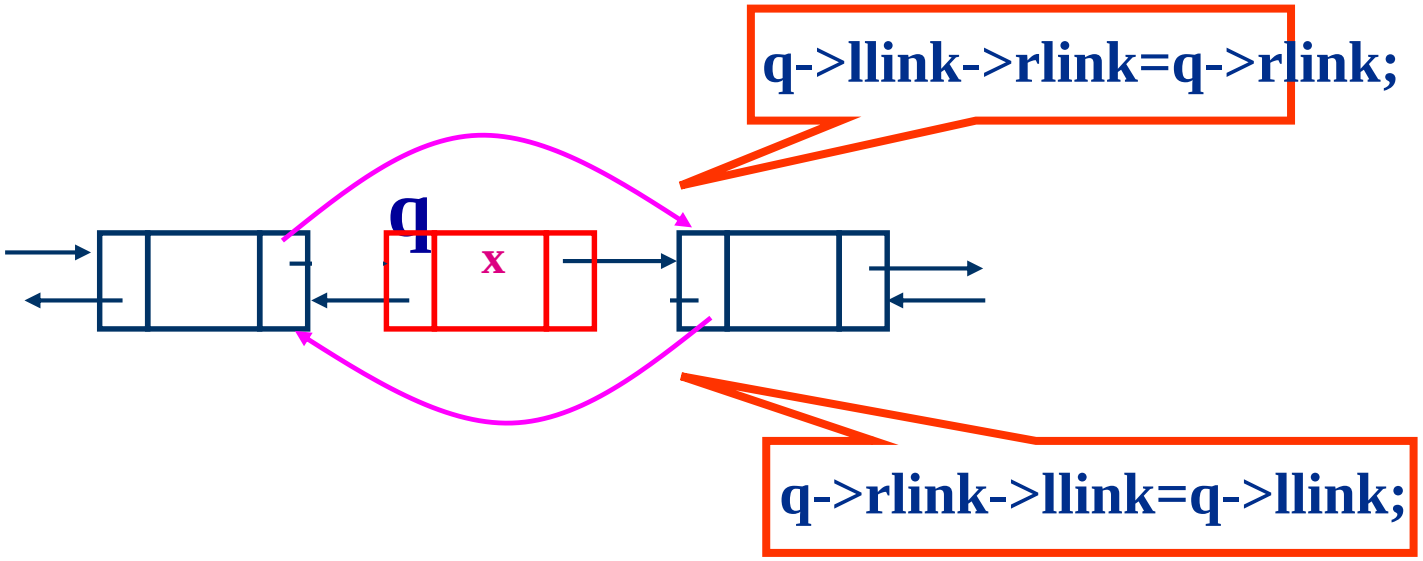


需要做的工作:

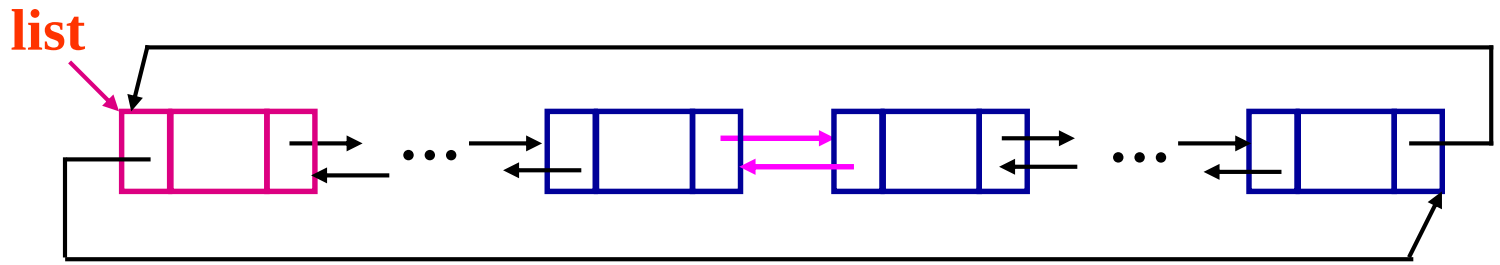
1. 找到满足条件的结点；
2. 若找到，删除(并释放)满足条件的结点。

双向链表的删除

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);
```

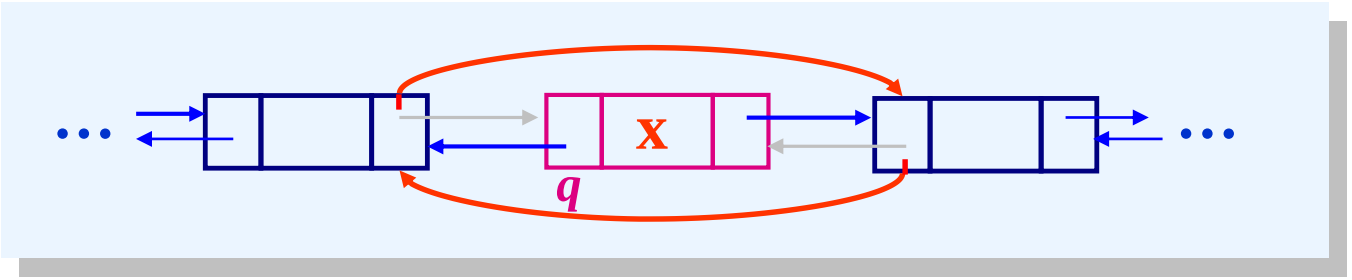
算法实现

非空双向循环链表的删除操作，若按不带头结点处理，需要考虑如下三种情况：

1）删除的是第一个结点
2）删除的是只有唯一一个结点的链表
3）正常的删除

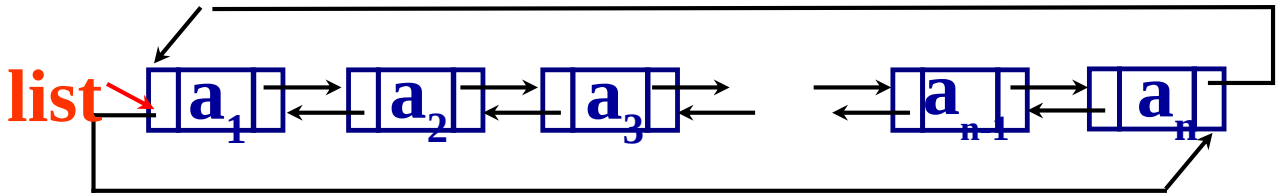
```
int  deleteDNode(DNodeptr list, ElemType x)
{     DNodeptr q;
      /* q初始指向第一个结点，头指针 */
    for(q=list; q && q->data!=x && q->rlink!=list; q=q->rlink)
        ;
    if(!q || q->data!=x)
        return -1;                          /* 没有找到满足条件的结点 */
      q->llink->rlink=q->rlink;
      q->rlink->llink=q->llink;
      free(q);                              /* 释放被删除的结点的存储空间 */
      return 1;                             /* 删除成功 */
}
```

寻找满足条件的结点

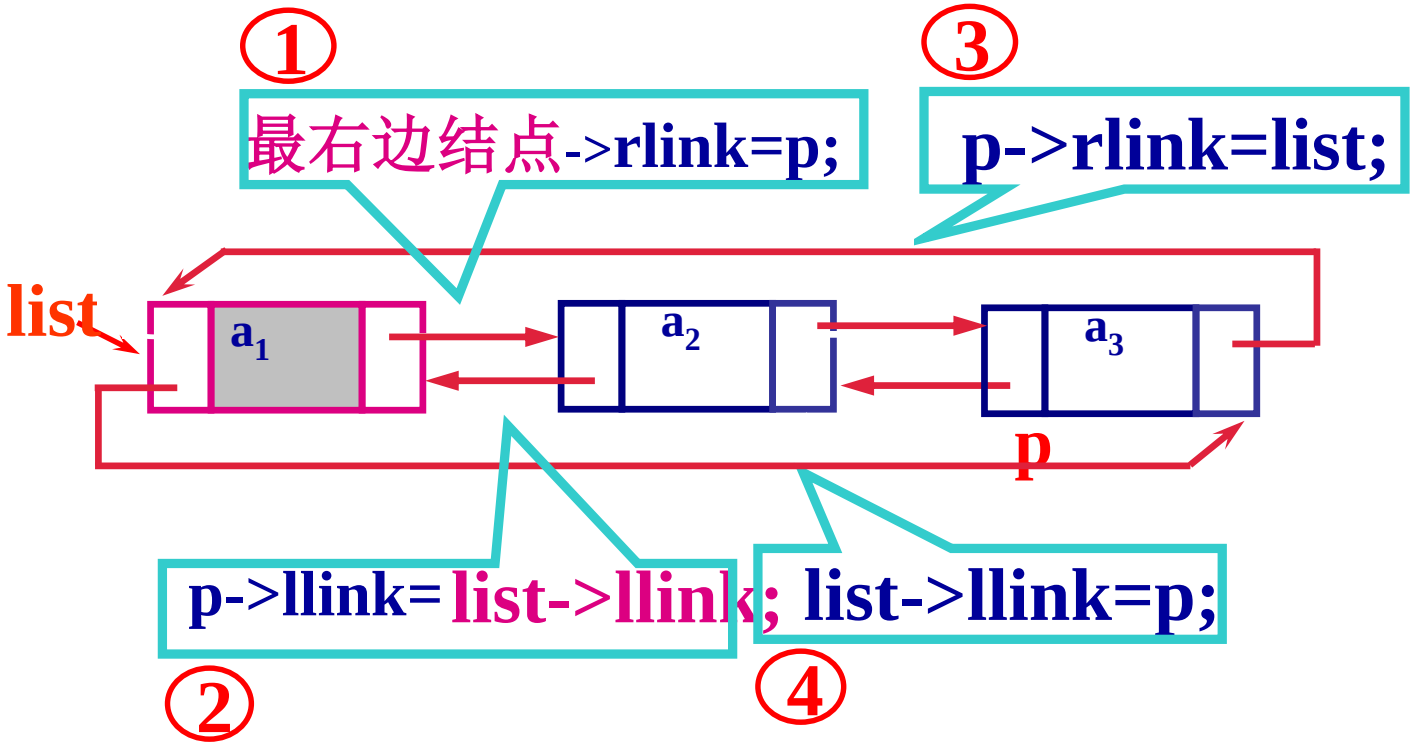


构造一个双向循环链表

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

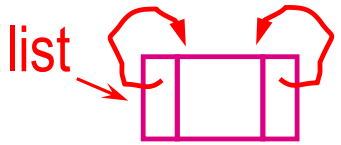


插入



算法实现

```
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;
}
```




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

时间复杂度O(n)

双向链表特点小结

- 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

单选题 1分

设置

若一个链表最常用的操作是在最后一个结点之后插入一个新结点，或删除最后一个结点，则选用下面哪种结构效率最高？

A 带头结点的单链表

B 不带头结点的单链表

C 带头结点的单循环链表

D 带头结点的双向循环链表

提交

线性链表使用的注意事项

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指针，对头指针的移动都应是有意义的（如要在头指针前插入一个结点或要删除链表的第一个结点），否则会造成链表头指针的丢失。

第 2 章

线 性 表

本章小结

Summary

- 线性表的基本概念
  - 线性关系和线性表
  - 线性表的基本操作
- 线性表的顺序存储
  - 构造原理
  - 查找、插入、删除等算法设计和应用
- 线性表的链式存储
  - 线性链表（单链表）
  - 循环链表和双向链表
  - 插入、删除等算法设计