



数据结构

Data Structure

张先宜

QQ群: **275437164** (XC数据结构交流群)

手机: 18056307221 13909696718

邮箱: zxianyi@163.com

QQ: 702190939

第2章 线性表

2.1 线性表的定义和运算

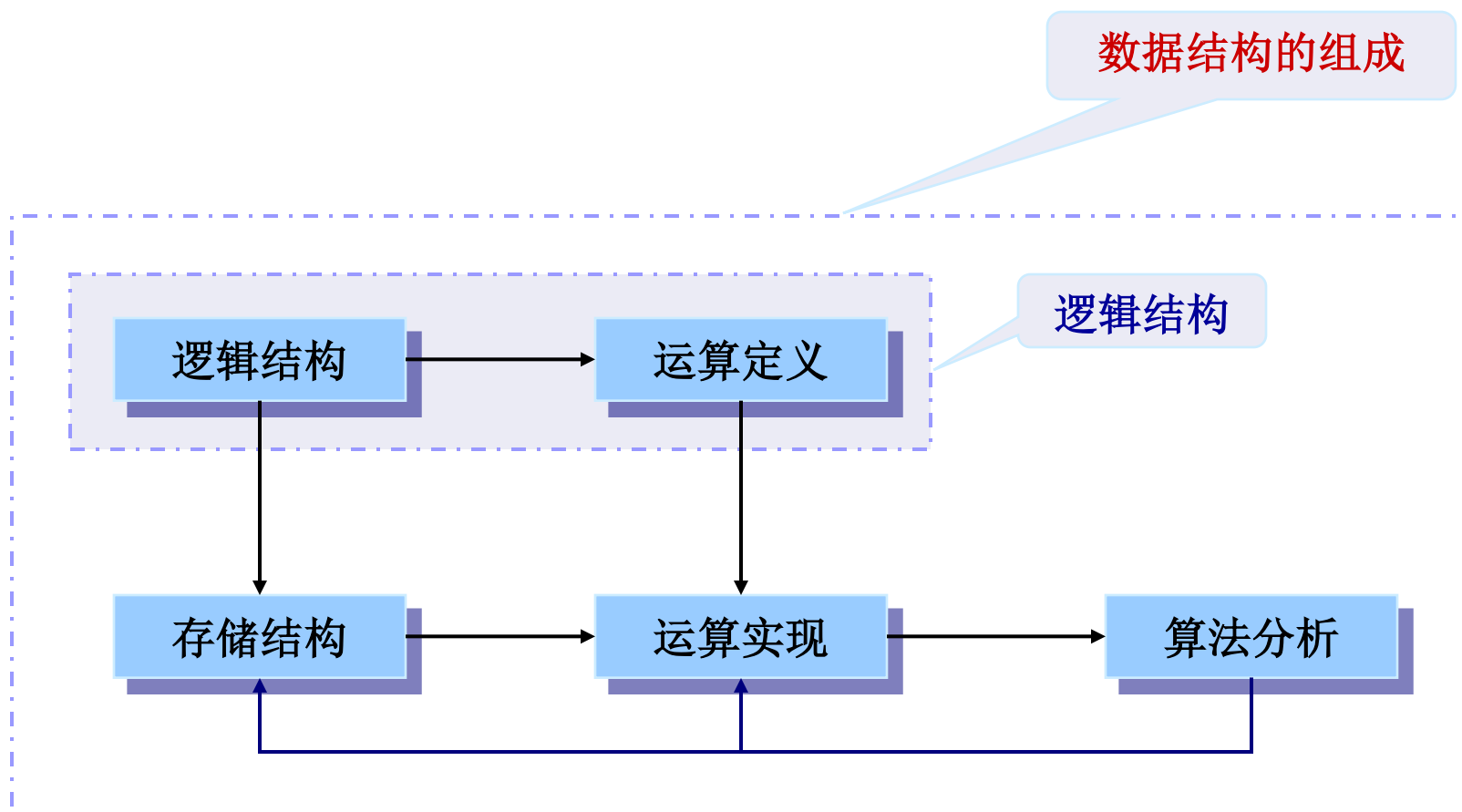
2.2 线性表的顺序存储结构

2.3 线性表的链式存储结构

2.4 其它结构形式的链表

■ 学习线性表结构时，需要掌握哪些方面的内容？

👉 请看下图：



2.1 线性表的定义和运算

- **线性表**是一种最常用、最基本的数据结构；
- **线性表**是一种简单的、应用广泛的数据结构；
- **线性表**是学好其它许多结构的基础；

■ 线性表的实例：

- ➡ 字母表 (A , B , C , D , , Z) ;
- ➡ 数字表 (0 , 1 , 2 , 3 , 4 , , 9) ;
- ➡ 月份表 (1月,2月,...,12月);
- ➡ 季节表 (春 , 夏 , 秋 , 冬) ;
- ➡ 学生成绩表 , 其中每个元素就是一个人的成绩信息。
- ➡ 数据库中table等

2.1.1 线性表的定义

☞ 逻辑结构和运算

- 定义：**线性表**L是由n个元素 a_1, a_2, \dots, a_n 组成的**有限序列**。

记作 $L = (a_1, a_2, \dots, a_n)$

其中 $n \geq 0$ 为**表长度**；

$n=0$ 时L为**空表**，记作 $L = ()$

- 表中元素 a_i 的含义：

☞ 在不同的场合有不同的含义，可以是简单类型数据，也可以是复杂的结构类型或对象。

☞ 但是，在同一表中，元素类型相同。

■ $L = (a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$ 中：

☞ a_{i-1} 叫做 a_i 的**直接前驱**；

☞ a_{i+1} 叫做 a_i 的**直接后继**。

■ **非空线性表的特点：**

☞ 存在一个“第一个（头、首）”数据元素；

☞ 存在一个“最后一个（尾）”数据元素；

☞ 除首元素外其他元素**有且仅有一个**直接前趋；

☞ 除尾元素外其他元素**有且仅有一个**直接后继。

2.1.2 线性表的运算

☞ 对线性表有如下基本运算：

(1)初始化：**initialList(L)**

创建一个空的线性表，使用线性表必经过程。

(2)求表长度：**length()**

返回线性表中的元素个数。

(3)按序号取元素：**getElement(i,x)**

从线性表中取出序号为 i 的数据元素。

前提： $1 \leq i \leq n$ ，即存在该元素。否则，应当如何处理？

(4)按值查找元素：**locate(x)**

在线性表中查找给定值的元素 x 所在的位置。
若不存在，应如何给出相关信息？

(5)插入元素：**listInsert(i, x)**

在线性表中给定的位置 i 处，插入给定值为 x 的元素 x 。

前提： $1 \leq i \leq n+1$ ，即插入位置有效，否则如何处理？还有表空间满如何处理？

(6)删除元素：**listDelete(i)**

删除线性表中指定序号 i 处的元素。

前提： $1 \leq i \leq n$ ，即存在该元素。否则，应当如何处理？还有空表如何处理？

- 这里给出的是线性表的6个基本运算，实际中可根据需要**增减**运算；

☞ 例：**两表合并为一表，一个表拆分为多个表...**

- 借助这些基本运算可以构造出更加复杂的运算。

☞ 例：**删除 x 元素** — 可先用**locate(x)**找出元素x的位置 i，再用**listDelete(i)**进行删除。



子曰：“知之者不如好之者，好之者不如乐之者。”

2.2 线性表的顺序存储结构—顺序表

【本节内容】

2.2.1 顺序存储结构

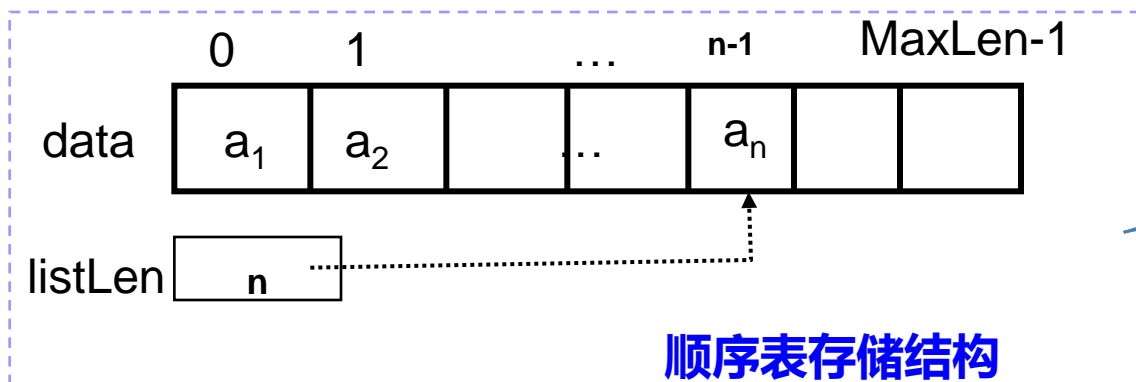
2.2.2 顺序表运算

2.2.3 顺序表应用

2.2.1 顺序存储结构

- 假设有一个足够大的连续存储空间（数组）data, 用于存储线性表的元素。
- 将线性表中的元素依次存储到数组中---顺序存储方式--顺序表。

■ 顺序表结构如下图所示：



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

■ 两个数据成员：

- 👉 **data[]** 数组用来存放线性表的数据元素；
- 👉 **listLen** 记录表中的元素个数，为整型变量。

■ 顺序表类型描述

```
#define MaxLen 100           //最多100个元素
typedef int elementType;     //定义elementType为整型

struct sList
{
    elementType data[MaxLen]; //定义存放元素的数组
    int listLen;              // 定义长度分量
};

typedef struct sList seqList; //定义seqList类型
```

■ 或者为：

```
#define MaxLen 100           //最多100个元素
typedef struct sList
{
    elementType data[MaxLen]; //定义存放元素的数组
    int listLen;              // 定义长度分量
}seqList;
```

【注】 元素的类型这里用elementType，实际中视实际情况用具体的类型来替换，如：int等。
如用 **typedef int elementType;** //定义为整型

- 使用**typedef** 把 **seqList**定义为一种**数据类型**，我们可以使用此类型来定义变量

☞ 例：**seqList L1, *L2;**

■ **seqList**分量的使用方式

☞ 上例中，L1是seqList型的结构变量，分量的引用方法：

L1.listLen ; L1.data[i]

☞ L2为seqList型的结构指针变量，分量的引用方法：

L2->listLen; L2->data[i];

■ 顺序表SeqList类的C++完整描述

```
class SeqList
{
public:
    SeqList();    //初始化空表
    int length(); //求表长度（元素个数）
    bool getElement(int i, elementType &x); //取元素
    int locate(elementType x);              //定位元素
    int listInsert(int i, elementType x); //插入
    int listDelete( int i );              //删除
private:
    elementType data[MaxLen]; //存放表元素的数组
    int listLen;              //记录表中的元素个数
};
```

■ 【说明】顺序表元素下标和数组data的下标相差1。

- ☞ 表元素下标从1开始；
- ☞ 数组data按C语言规范，下标从0开始。
- ☞ 对应关系如下图：

a_1	a_2	...	a_{n-1}	a_n
data[0]	data[1]	...	data[n-2]	data[n-1]

2.2.2 顺序表的运算实现

1. 初始化

建立一个空表，即使得顺序表的listLen=0。

```
void initialList(seqList *L)
{
    L->listLen=0;
}
```

👉 为什么要用 *L？还有其他方式返回顺序表吗？

👉 算法时间复杂度 $O(1)$

2.求顺序表长度：

定义函数返回表L的listLen分量即可。

```
int  listLength( seqList L )  
{  
    return L.listLen ;  
}
```

- ❏ 为什么不用指针呢？可以用指针吗？
- ❏ 表的长度可以用指针或引用返回吗？
- ❏ 时间复杂度 $O(1)$

3.按序号取元素：

给定元素序号 i ，取出第 i 个元素，取出的值用变量 x 返回。如果 i 超出范围，怎样处理？

```
void getElement ( seqList L , int i, elementType &x )  
{  
    //i超出范围的处理, 没有实现  
    if ( i < 1 || i > L.listLen )  
        error(“超出范围” );  
    else  
        x = L.data[i-1];  
}
```

- 👉 注：此段代码只是为了描述，实际编程不能这样处理。
- 👉 用 x 返回取得的元素值，这里使用了 C++ 的引用
- 👉 还有其他返回值的方法吗？

■ 一种实际实现代码示例（有不同实现方法）

```
bool getElement(seqList L, int i, elementType &x)
{
    //序号i超出范围，取元素失败，返回0
    if(i<1 || i>L.listLen)
        return 0;
    else
    {
        x=L.data[i-1]; //取得元素存x，用参数返回
        return 1;      //取元素成功，返回1；
    }
}
```

👉 时间复杂度：O(1)



多少事，从来急；天地转，光阴迫。一万年
太久，只争朝夕。毛泽东

4. 按值查找元素：

将给定的元素x 与 L 中数据元素逐个进行比较，若存在相同的元素，则返回第一个相同元素的位置序号；否则，不存在相同元素，返回值 0。

```
int listLocate ( seqList L , elementType x )
{   int i;
    for( i=0;i<L.listLen;i++ )
        if ( L.data[i]==x )
            return i+1; //数组的下标比元素序号少1
    return 0; //如果找到了x从上个return语句返回；
              //执行到此，说明没有找到x，所以返回0.
}
```

【算法分析】

1) 基本操作：数据比较；

2) 若 L 中存在数据元素 x，位置为 i， $1 \leq i \leq n$ ，
需要比较次数：i 次；

☞ 概率为： $1/n$ ；

☞ 所以平均可能比较次数为：
$$\sum_{i=1}^n i \times \frac{1}{n} = \frac{(n+1) \times n}{2n} = \frac{n+1}{2}$$

3) 若 L 中不存在数据元素 x，比较次数为：n；

■ 所以时间复杂度： $O(n)$

5.插入运算：

在序号 i 位置插入元素 x 。

【分析】

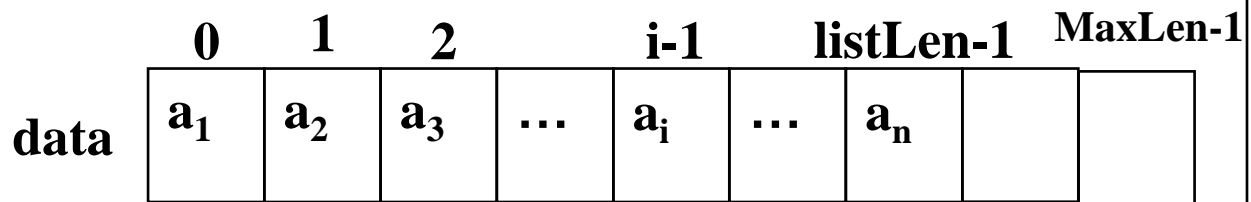
(1) 首先要检查插入条件是否满足：

- ☞ 表空间未滿，即 $\text{listLen} \leq \text{MaxLen} - 1$ ；
- ☞ 序号正确：范围在 $1 \sim n+1$ 之间，即： $1 \leq i \leq n+1$ ；
- ☞ x 的类型与表的数据类型相同。

(2) 插入步骤：

- ☞ a_i, a_{i+1}, \dots, a_n 往后移一位，如何实现？
- ☞ 填入 x ，即 $\text{data}[i-1] = x$ ；
- ☞ 表长度增1，即： $L.\text{listLen}++$ 。

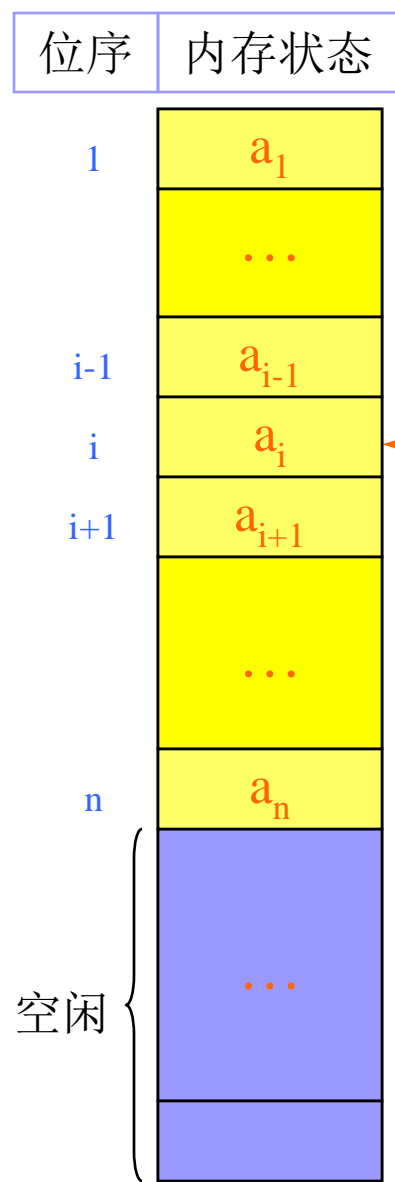
seqList



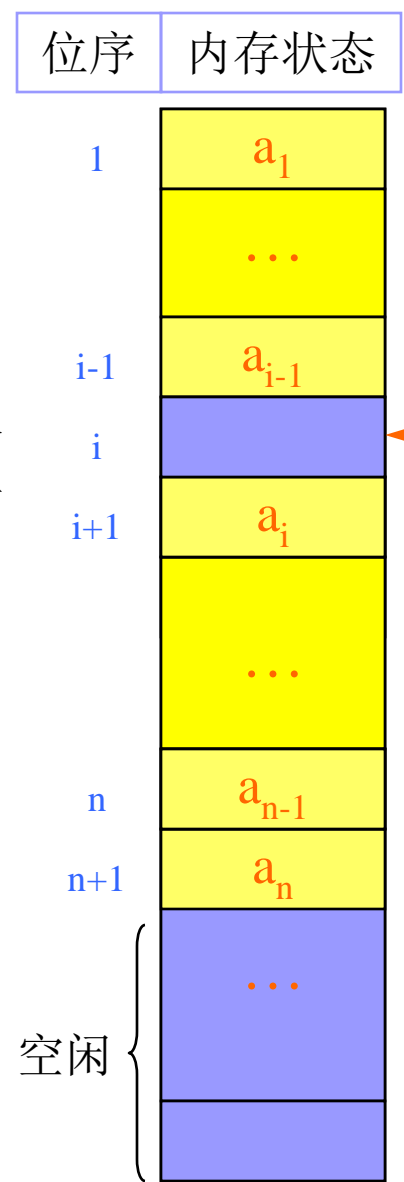
listLen n

x

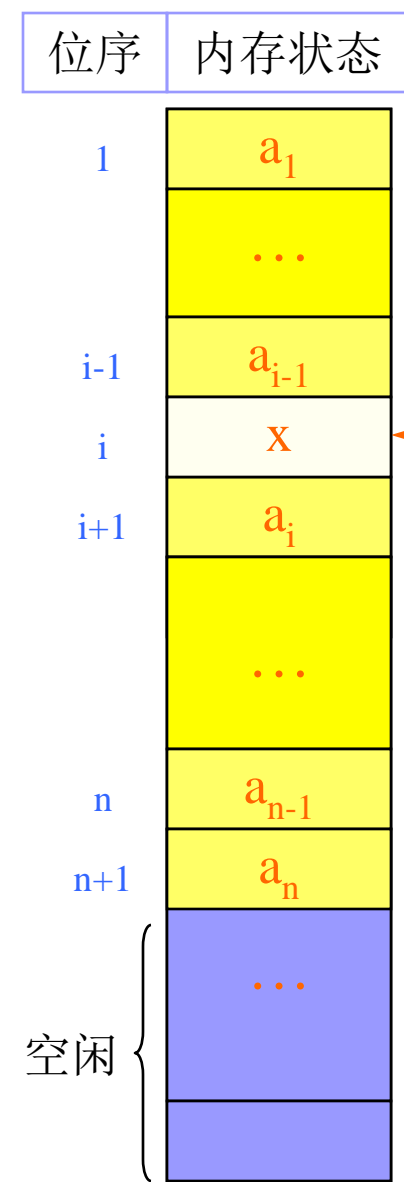
全部向后移一个位置



插入位置



空位



插入新元素

```
void listInsert( seqList *L, elementType x, int i )
{
    //注意i为元素序号，非数组下标
    if      顺序表满了          error(“溢出” );
    else if (  序号错了        error(“序号错误” );
    else
    {
        如何实现 $a_i, a_{i+1}, \dots, a_n$ 往后移?

        L->data[i-1]=x;
        L->listLen++;
    }
}
```

一种实际实现：(空间满返回0；超范围返回1；正确插入返回2)

```
int listInsert( seqList *L, elementType x, int l )
```

```
{ int j;
```

```
  if(L->listLen==MaxLen)
```

```
    return 0; //空间满
```

```
  else if(i<1 || i>L->listLen+1)
```

```
    return 1; //序号超出范围
```

```
  else
```

```
  { for(j=L->listLen-1; j>=i-1; j--)
```

```
    L->data[j+1]=L->data[j]; //后移元素
```

```
  L->data[i-1]=x; //插入元素x
```

```
  L->listLen++; //长度增1
```

```
  return 2;
```

```
}}
```

【算法分析】

- 1) 基本操作：插入前移动数据元素；
- 2) 插入位置为 i ，需要移动次数： $n-i+1$ 次；
- 3) 插入位置概率相同： $1/(n+1)$ ；

4) 平均可能移动次数：
$$\sum_{i=1}^{n+1} (n-i+1) \times \frac{1}{n+1} = \frac{(n+1) \times n}{2(n+1)} = \frac{n}{2}$$

■ 所以时间复杂度： $O(n)$

6.删除元素

删除表中序号 i 位置的元素。

【分析】

(1) 首先要检查删除条件是否满足：

☞ 序号正确：范围在 $1 \sim n$ 之间，即： $1 \leq i \leq n$ ；

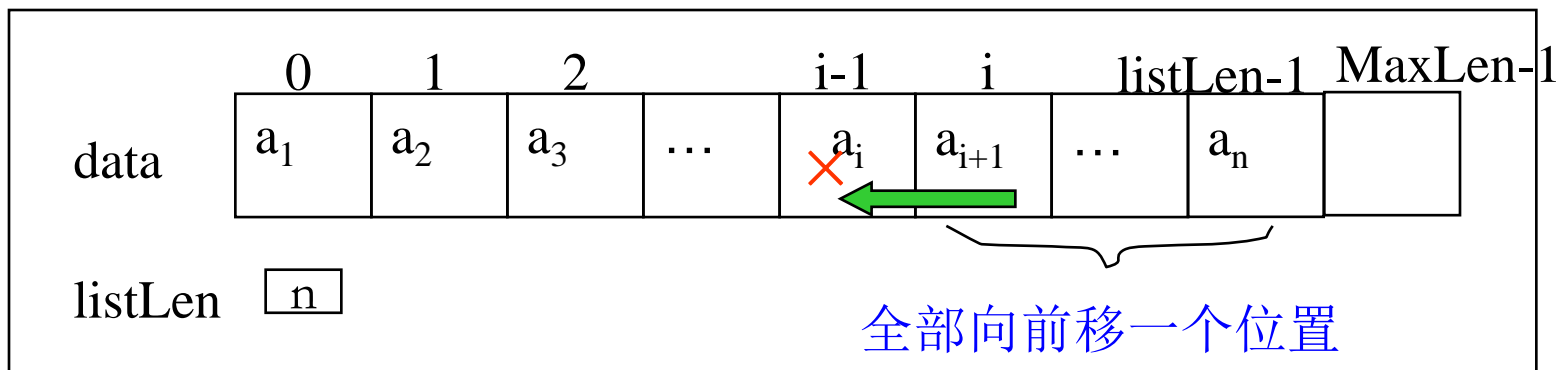
☞ 是否空表？

(2) 删除步骤：

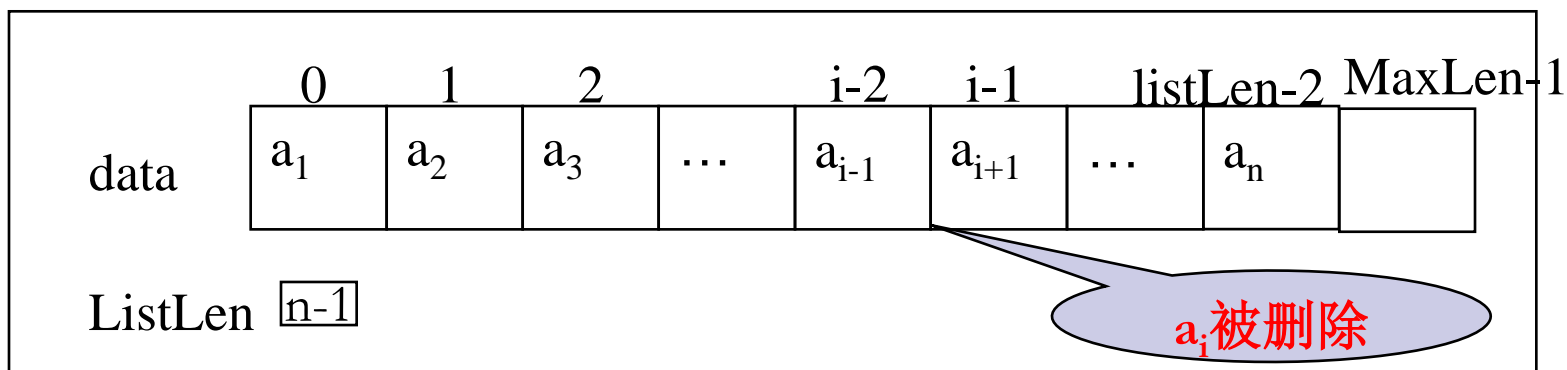
☞ a_{i+1}, \dots, a_n 往前移动一个位置，“挤掉” a_i 元素。

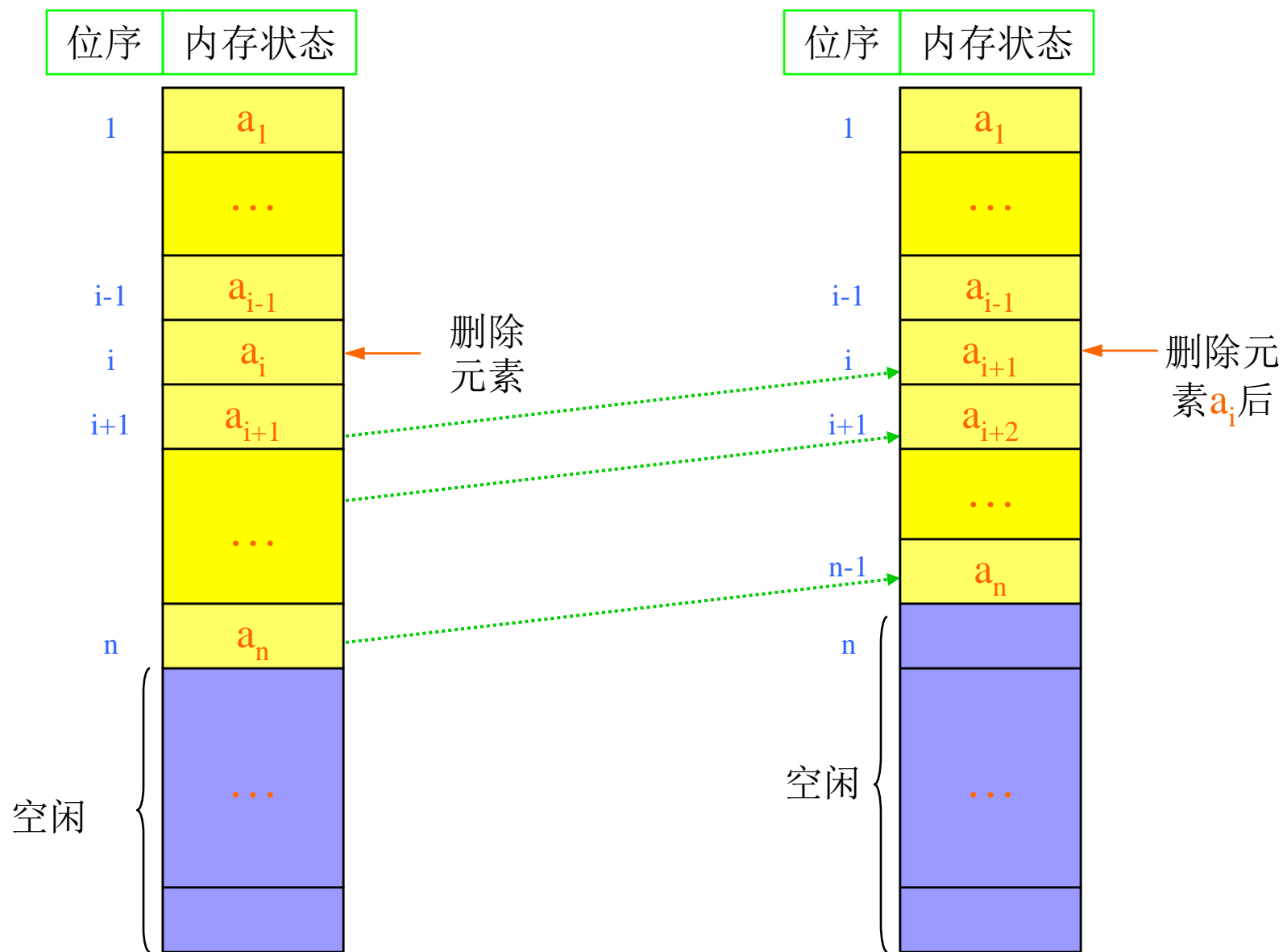
☞ 表长度减1，即 $\text{listLen}--$ 。

seqList



seqList





■ 删除算法描述

```
void listDelete ( seqList *L , int i ) ) //i为元素编号
{
    if (L->listLen==0) error(“下溢出” );
    else if(i<1||i>L->listLen) error(“序号错误” );
    else {
        for(j=i; j<=L->listLen-1; j++)
            L->data[j-1]=L->data[j];
            //循环前移元素。
            //j是移走元素数组下标
        L->listLen--;
    }
}
```

【思考问题】

算法中循环控制变量 **j** 的含义是 ____ :

- A. 要移走的线性表元素的序号 ;
- √ B. 要移走的数组元素的下标 ;
- √ C. 将要移到的线性表元素的序号 ;
- E. 将要移到的数组元素的下标。

一种实际实现：(空表返回0；序号超出范围返回1；正确删除返回2)

```
int listDelete(seqList *L, int i )
{ int j;
  if(L->listLen<=0)    return 0; //空表，返回值0
  else if(i<1 || i>L->listLen)
    return 1; //删除的序号不在有效范围内
  else
  {   for(j=i; j<L->listLen; j++ )
      L->data[j-1]=L->data[j]; //循环前移表元素

      L->listLen--; //修改表长度
      return 2;    //成功删除，返回值2.
  }
}
```

【算法分析】

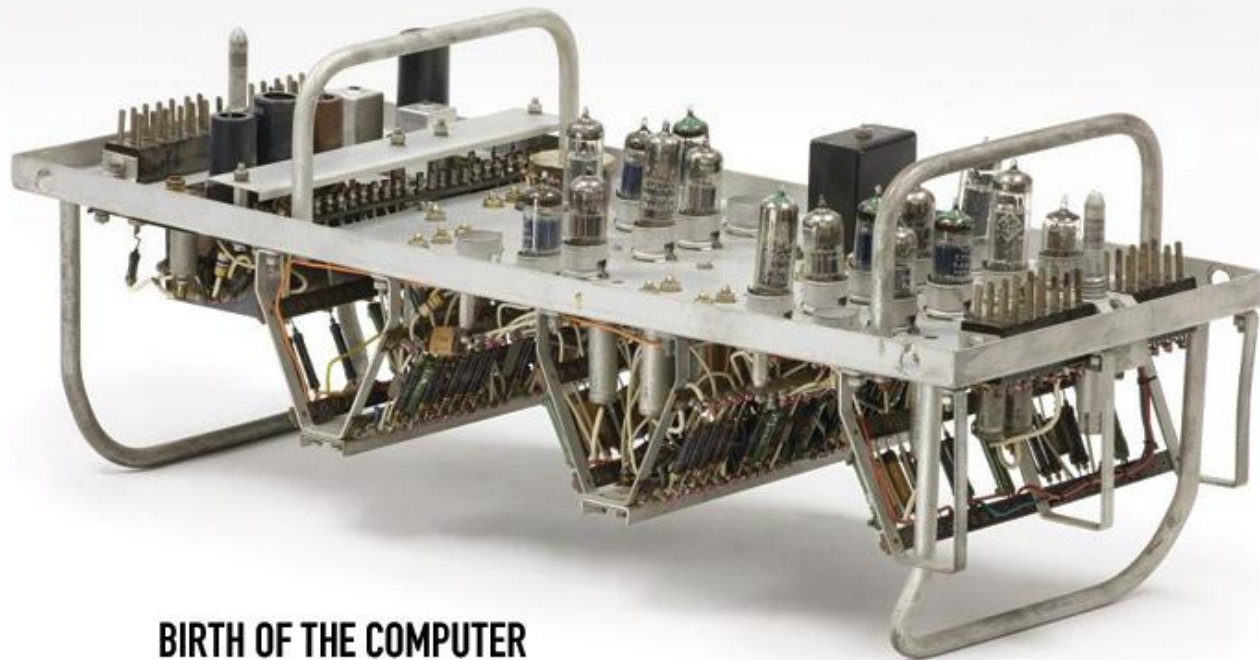
1) 基本操作：向前移动数据元素，覆盖原来值；

2) 删除位置为 i ，需要移动次数： $n-i$ 次；

3) 删除元素位置概率相同： $1/n$ ；

4) 平均可能移动次数：
$$\sum_{i=1}^n (n-i) \times \frac{1}{n} = \frac{(n-1) \times n}{2n} = \frac{n-1}{2}$$

■ 所以时间复杂度： $O(n)$



BIRTH OF THE COMPUTER

程序是调（debug）出来的。

2.2.3 顺序表的应用

【例2.0】 现有2个集合A和B，求一个新集合 $C=A \cup B$ ，3个集合都用顺序表表示。

比如， $A=\{2, 4, 10, 5, 9\}$ ， $B=\{1, 4, 6, 8\}$ ，
则 $C=\{2, 4, 10, 5, 9, 1, 6, 8\}$

■ 算法思想：

这是较简单的顺序表合并问题，要注意的是因为A、B和C都是集合，**合并后C中不能出现重复的元素**。基本步骤如下：

- ① 循环取出A中的元素，直接插入到C的尾部，此元素在C中的序号为 **$C.\text{listLen}+1$** ；
- ② 循环取出B中的元素，判断是否出现在A中，在A中时跳过（**集合元素不能重复**），不在A中，则插入到C的尾部。

■ 算法描述（一种实现，有多种实现，Exam200UnionSet.cpp）：

```
void mergeSet(seqList A, seqList B, seqList &C)
{  int i;
   for(i=0;i<A.listLen;i++)
   {
       listInsert( &C, A.data[i], C.listLen+1 );
   }
   for(i=0;i<B.listLen;i++)
   {
       //检查x是否在A中出现，未出现元素插入C
       if(listLocate(A,B.data[i])==0)
           listInsert( &C, B.data[i], C.listLen+1);
   }
}
```

【算法分析】

假设A的元素个数为 m ，B的元素个数为 n

本例的时间性能主要取决于第二个for循环，事实上这是一个双重循环，因为A.locate(x)中有循环，其时间性能为 $O(m)$ ，

所以总的时间复杂度为 $O(m \times n)$

【思考问题】

- ① 为什么listInsert中的插入位置是pC->listLen+1，而不是pC->listLen呢？
- ② 为什么A、B没用引用，而C使用引用呢？
- ③ 本例使用引用从子函数往主函数传回新表C，如果采用指针回传程序该如何修改？还有其它传回C的方法吗？
- ④ 如果算法中的A和B都用“指针”或“引用”有什么不同呢？

【例2.1】 已知顺序表L递增有序，设计算法，在L中插入元素x，使得L依然递增有序。如：
 $L=(2,5,7,9,12,14,15)$ ， $x=8, 17, 1$

【算法思想】

👉 基本思想：

找到x的插入位置i， $1 \leq i \leq n+1$ ，保持插入后L仍递增，然后插入。

☞ 对此有2种基本方法：

① 从前往后搜索插入位置，然后批量移动其后面的元素。这种方法比较费时：搜索前面 i 个元素；移动后面 $n-i+1$ 个元素。每个元素都会访问到。

② 从后往前搜索，同时移动元素。当元素值大于 x 时后移，重复进行，直到小于等于 x 元素，后退一个单元即 x 的插入位置。

比较次数比移动次数多1（除非第一个元素）。

如： $L=(2,5,7,9,12,14,15)$ ， $x=8$ ，17，1

■ **算法描述(只是描述，不能直接运行)：**

```
void insert ( seqlist *L , elementType x )
{   int i=L->listLen-1 ; //取表中最后元素的数组下标
    if ( i>=MAXLEN-1 ) then error ( "overflow" );
                                //表满
    else
    {   while ( i>=0 && L->data[i]>x )
        {   //往前搜索插入位置，并后移元素
            L->data[i+1]=L->data[i] ;
            i--;
        }
        L->data[i+1]=x ; //为什么是i+1呢？
        L->listLen++ ;
    }
```


【算法实现】

- ☞ 从前往后搜索插入：ex211OrderedList1.cpp
- ☞ 从后往前搜索插入：Exam201InclInsert.cpp

【算法分析】

- 1) 基本操作：插入前比较和移动数据元素；
- 2) 插入位置为 i ，需要移动次数： $n-i+1$ 次；
- 3) 插入位置概率相同： $1/(n+1)$ ；

4) 平均可能移动次数：
$$\sum_{i=1}^{n+1} (n-i+1) \times \frac{1}{n+1} = \frac{(n+1) \times n}{2(n+1)} = \frac{n}{2}$$

- 5) 一般情况下，比较次数比移动次数多1次，取决于插入位置。最好情况：比较1次，移动0次。最差情况：比较 n 次，移动 n 次。

■ 时间复杂度： $O(n)$

【思考问题】

- ① 算法中的while循环结束时，目标空位置（插入位置）的数组下标是 i 还是 $i+1$ ？请模拟在表（4,6,10,15,20）中分别插入25、8和2时的实现过程。
- ② 如果while循环条件 $\text{data}[i] > x$ 改为 $\text{data}[i] \geq x$ ，结果会有何不同？
- ③ 用for循环能否完成相同功能呢？

【例2.2】 假设顺序表A、B分别表示一个集合，设计算法以判断集合A是否是集合B的子集，若是，则返回TRUE，否则返回FALSE，并要求时间尽可能少。

例：A={6,4,2}， B={9,2,6,4,7,1,3}

A={6,8,2}， B={9,2,6,4,7,1,3}

【算法思想】

- 👉 2层循环实现：
- 👉 第一层循环，依次取出A中元素；
- 👉 第二层循环，判断A中取出的元素是否在B中。若在B中，回到第一层循环，继续取A的下一个元素。若不在B中，直接返回false。
- 👉 A的所有元素都在B中，返回true。

■ 【算法描述】（实现代码：Exam202SubSet.cpp）

```
BOOL subset(seqList *A,*B)
{ int ia,ib; elementType x; BOOL suc;
  // ia,ib为A、B元素数组下标
  for ( ia=0; ia<A->listLen; ia++ )
  {
    ib=0; suc=FALSE;           //suc为搜索成功与否的标志
    while ( ib<B->listLen && suc==FALSE )
      if(A->data[ia] ==B->data[ib])
        suc=TRUE; //搜索到指定元素，设置成功标志
      else ib++;    //否则，继续搜索B的下一个元素
    if ( suc==FALSE ) return FALSE;
                        //A表当前元素不在B中，立即结束
  }
  return TRUE;
  //到此处时，第一层循环结束，A一定是B的子集
}
```

【算法分析】

- 由于A中每个元素都要与B中每个元素比较，故该算法的时间复杂度为两表之长度的积的数量级，即为 $O(|A|*|B|)$ 。

【思考问题】

- ① 也可这样求解：将判断指定元素是否在B表中的求解单独写一个函数，在此不再赘述，有兴趣的读者可自己练习。
- 此算法函数参数A和B都是指针，不用指针可以吗？用指针有什么好处？使用C++的“引用”是否可发挥相同作用？

【例2.3】 假设**递增有序**顺序表A、B分别表示一个集合，设计算法以判断集合A是否是集合B的子集，若是，返回TRUE，否则返回FALSE，并要求时间尽可能少。

【算法思想】

👉 本题也可用前例算法求解，但没有用到所给出的**递增有序的条件**，时间性能不是最佳。

■ A是B的子集演示

A

3	5	7
---	---	---

ia

B

1	2	3	4	5	7	9
---	---	---	---	---	---	---

ib

成功！ 返回true

■ A不是B的子集演示

A

3	6	7
---	---	---

ia

B

1	2	3	4	7	8	9
---	---	---	---	---	---	---

ib

失败！ 返回false

☞ 设用ia和ib分别为A和B中元素的编号，当ia和ib均指向各自表中某一元素时，可能会出现如下几种情况：

① **A.data[ia] > B.data[ib]:**

- ★ 即A表中当前元素大于B表中当前元素，因而需要继续在B表中搜索，即要执行ib++以使ib指向B表中下一个元素并继续搜索。

② **A.data[ia]== B.data[ib]:**

- ✦ **A表中当前元素在B表中，即查找成功，因而可继续A表中下一个元素的判断，即要执行ia++以使ia指向A表的下一个元素并继续搜索。与此相对应的是，指示B表中元素的ib应指示到哪儿？**
- ✦ **一种简单的办法是从头开始，但那样的话，还是没有用上递增有序的条件，因而没有改善算法的时间复杂度。仔细分析可得到另一种办法，即ib还是指示原来的位置，或者简单地往后移动一位，即执行ib++也可以。**

③ **A->data[ia] < B->data[ib]:**

- ✦ **即A表中当前元素小于B表中当前元素，因而肯定小于其后面的所有元素，所以该元素肯定不在B表中，即搜索失败。由于只要有一个元素不在B表中，就意味着A表不是B表的子集，故整个算法的求解结束，可返回结果（FALSE）。**

- 重复执行上述判断过程，直到ia和ib中至少有一个指向表尾之后为止，此时，根据不同的情况可以分别得出如下结论：

- 1) **ia >= A.listLen**: 即A表结束，意味着A表中每个元素都已经被判断过了，并且都在B表中（**为什么？**），因而可以返回结论TRUE。
- 2) 否则，A表未结束，B表结束，意味着A表中的当前元素肯定不在B表中，因而返回结论FALSE。

■ **算法描述**：（ **实现**：Exam203OrderedSubset.cpp ）

BOOL OrderedListSubset (seqList *A,*B)

```
{  
    int ia=0, ib=0;  
    while (ia<A->listLen && ib<B->listLen)  
        if (A->data[ia]== B->data[ib]) {ia++; ib++;}  
        else if (A->data[ia]> B->data[ib]) ib++;  
        else return FALSE;  
  
    if (ia>=A->listLen) return TRUE;  
    else return FALSE; //一表结束，另一表未结束  
}
```

【算法分析】

- 由于ia、ib从头开始依次指示A、B表中每个元素一次（严格地说，由于停顿可能使某个元素被比较几次，但每次比较至少要通过一个元素），故算法的时间复杂度为两表长度之和的数量级，即 $O(|A|+|B|)$ 。

【思考问题】

- ① 一般情况下，A、B 中哪个表会先结束？
- ② 什么情况下 A、B 会同时结束？



只有不想会，没有学不会。

【例2.4】 设计算法将**递增有序**顺序表A、B中的元素合并为一个递增有序顺序表C，并要求时间尽可能少。

例： $A=(1,3,4,6,7,8,9)$, $B=(2,4,5,7)$

【算法思想】

- ☞ 每次取A和B中的一个最小元素，依次插入C表即可。
- ☞ 设用ia和ib分别为A、B表元素的数组下标。
- ☞ 当ia和ib均指向各自表中某一元素时，元素大小的比较可能会出现如下几种情况：

① $A.data[ia] < B.data[ib]$:

A表中当前元素较小，插入C表。执行 $ia++$ ，继续取A表的下一个元素，而ib不变；

② $A.data[ia] == B.data[ib]$:

A、B表当前元素相同，同时插入C表。执行 $ia++$ ， $ib++$ ，同时取下一个元素；

③ $A->data[ia] > B->data[ib]$:

B表中当前元素较小，插入C表。执行 $ib++$ ，继续取B表的下一个元素，而ia不变；

■ 重复执行上述判断、插入过程，直到A和B表中有一个结束。

■ 未结束表中剩下元素如何处理？

-- 循环取出依次插入C表尾部。

【算法描述】

```
void mergeList(seqList A, seqList B, seqList &C)
{
    int ia=0,ib=0,ic=0; //A、 B、 C元素数组下标 , 从0开始
    while(ia<A.listLen && ib<B.listLen)
    {
        //情况1 : A.data[ia]<B.data[ib]
        if(A.data[ia]<B.data[ib])
        {
            listInsert(&C,A.data[ia],ic+1);
            ic++;
            ia++;
        }
        //情况2 : A.data[ia]==B.data[ib]
```



```
else if(A.data[ia]==B.data[ib])
```

```
{
```

```
    listInsert(&C,A.data[ia],ic+1);
```

```
    ic++;
```

```
    ia++;
```

```
    listInsert(&C,B.data[ib],ic+1);
```

```
        //或listInsert(&C,A.data[ia],ic);
```

```
    ic++;
```

```
    ib++;
```

```
}
```

```
else
```

```
    //情况3 : A.data[ia]>B.data[ib]
```

```
{
```

```
    listInsert(&C,B.data[ib],ic+1);
```

```
    ic++;
```

```
    ib++;
```

```
}
```

```
}
```

//下面处理一个表结束，另一个表未结束情况

while(ia<A.listLen) //处理A表未结束情况

{

listInsert(&C,A.data[ia],ic+1);

ic++;

ia++;

}

while(ib<B.listLen)

{

listInsert(&C,B.data[ib],ic+1);

ic++;

ib++;

}

}

【算法实现】

👉 Exam204OrderedMergeList.cpp

【算法分析】

👉 一趟循环处理完A、B全部元素，所以时间复杂度：
 $O(|A|+|B|)$

【思考问题】

- ① 上述算法，“情况①”可以并入“情况②”或“情况③”吗？
- ② A、B 什么情况下差不多会同时结束？
- ③ 算法中为什么表C要用“引用”，而A、B未用？用指针行吗？
- ④ 递减表呢？递减并递增、递增并递减？一个递减一个递增并未递增或递减？

■ 【思考问题】

☞ 若数据元素有多个数据项，比如学生成绩表有4项：**学号、姓名、课程、分数**。能不能用顺序表存储呢？如何存储呢？

■ 【方式一】直接在顺序表中定义

```
typedef struct
```

```
{
```

```
    char* sID[MaxLen]; //学号
```

```
    char* sName[MaxLen]; //姓名
```

```
    char* sCourse[MaxLen]; //课程名称
```

```
    int sScore[MaxLen]; //分数
```

```
    int listLen; //表长度
```

```
}seqList;
```

■ 结构成员（分量）引用方式

☞ 例：`seqList L,*L1;`

☞ `L.sID[i]; L.sName[i]; ...`

☞ `L1->sID[i]; L1->sScore[i]; ...`

- **【方式二】先用结构定义元素，顺序表定义不变。**

```
typedef struct element  
{  
    char* sID;  
    char* sName;  
    char* sCourse;  
    int sScore;  
}elementType;
```



```
typedef struct seqlist
```

```
{
```

```
    elementType data[MaxLen];
```

```
    //elementType为上页定义的结构体
```

```
    int listLen;
```

```
} seqList;
```

■ **元素项的引用方法，如：seqList L,*L1;**

☞ **L.data[i].sID; L.data[i].sCourse; ...**

☞ **L1->data[i].sID; L1->data[i].sCourse; ..**

【题型注意】

- ① 线性表表示集合，注意不能有重复元素；
- ② 集合的交、并、差运算；
- ③ 注意 $A=A \cup B$ 与 $C=A \cup B$ 的区别，其它运算同；
- ④ 线性表合并、分解；
- ⑤ 增加有序表（递增、递减）条件。

线性表顺序存储结构小结

■ 1. 顺序存储结构的优点

- ☞ (1) 可随机存取表中任一数据元素，且取任何一个元素的时间相同；
- ☞ (2) 存储空间连续，元素顺序存放，逻辑上相邻的元素，存储位置相邻。
- ☞ (3) 除了元素自身外，不必增加额外的存储空间。

线性表顺序存储结构小结

■ 2. 顺序存储结构的缺点

- ☞ (1) 线性表的容量难以扩充；
- ☞ (2) 在给长度变化较大的线性表预先分配空间时，必须按照最大空间分配，使得存储空间不能充分利用；
- ☞ (3) 插入、删除一个数据元素时，需要对插入点或删除点后面所有元素逐个进行移动，需要花费较多的时间。比如表中有千万条数据。

■ 如何克服以上缺点呢？

- ☞ 考虑不需移动元素、可按需增加空间的存储结构——链表（线性表的链式存储结构）。

【作业布置】

(P51)

 2.3

 2.6

 2.7

 2.8

 2.9



可怜天下父母心。

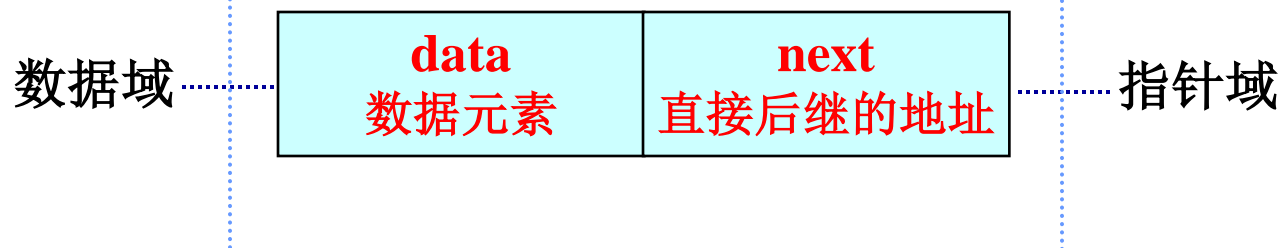
2.3 线性表的链式存储表示及实现

2.3.1 链表的概念

1. 链表的基本结构

- ☞ 用不连续的、或连续的存储单元存储线性表元素；
- ☞ 每个数据元素后，加上一个地址域，此地址为其直接后继的地址；数据元素和地址域组成**结点**（节点）。

■ 结点结构如图：



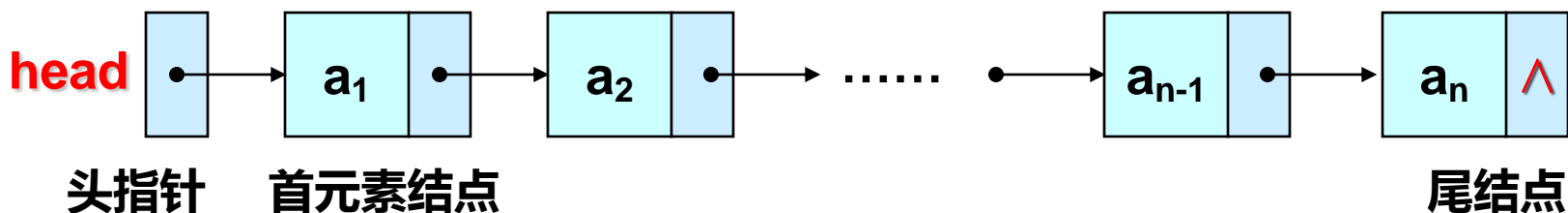
- ➡ **数据域** – 存储数据元素；
- ➡ **指针域** – 存储直接后继元素的地址（没有后继元素指针为空）；
- ➡ **指针、链** – 指针域中存储的信息，即另一个结点的地址；

■ 链表（Linked List）

- ➡ 通过每个结点的指针域将线性表中 n 个元素按其逻辑顺序**链接**在一起的**结点序列**，即线性表的链式存储结构。

■ 头指针(head)

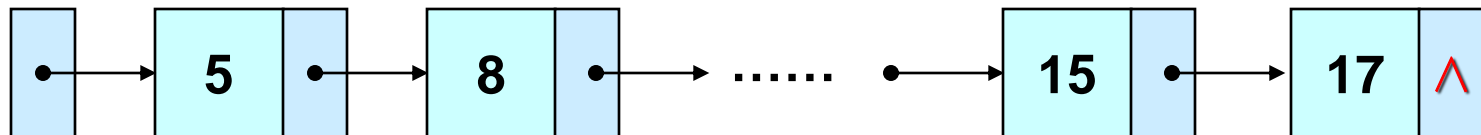
- ☞ 指向链表的第一个结点，是第一个结点的地址，或链表在存储器中的首地址；
- ☞ 头指针 (head) 的类型与其它结点指针域的指针 (next) 类型一致，都是指向**同一类型**结点。
- ☞ **单链表由头指针唯一确定**。因为由头指针即可找到第一个结点，由第一个结点的next指针即可找到第二个结点，如此“**顺藤摸瓜**”，即可找到链表上的所有结点。



■ 单链表 (Single Linked List)

- ❏ 线性链表中每个结点由**数据域**和**一个指针域**构成，指针域存放指向下一个结点的地址；
- ❏ 尾结点的指针域内容为空，表示链表结束，图形中用 \wedge 表示；
- ❏ 也称为：链表、或线性链表。
- ❏ 这些结点在内存中的位置可能是不连续的。
- ❏ 例：L=(5, 8, 9, 21, 4, 19, 15, 17) 的单链表表示：

head



■ 2. 单链表结点的存储描述

```
struct sNode
```

```
{
```

```
    elementType data; // 数据域
```

```
    struct sNode* next; // 指针域
```

```
                        //结构（结点）自身引用
```

```
}; //“;”不能省略
```

```
typedef struct sNode node; //或
```

```
typedef struct sNode node;
```

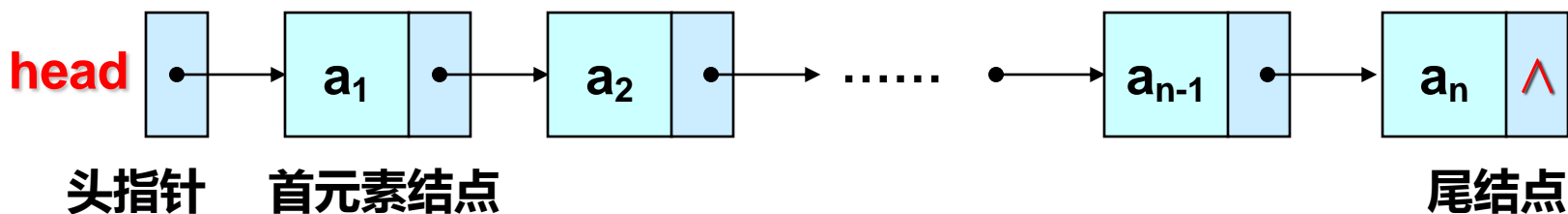
```
//用typedef将node定义为结点类型。
```

【合成定义】

```
typedef struct sNode
{
    elementType data; // 数据域
    struct sNode* next; // 指针域
                        //结构（结点）自身引用
} node;
```

■ 头指针的表示

- ☞ 是链表第一个元素结点的地址，用指针 ***head** 来指出；
- ☞ 由于head指针指向的是链表的结点，所以其类型与next指针相同，皆为node类型；
- ☞ 所以头指针的定义为：**node* head;**
- ☞ 今后实现时，头指针常用：**node* L;**
- ☞ 上述描述看上去好像只和一个结点有关，事实上，**定义了头指针后，即定义了整条链表。**



■ 头指针引用结点元素和指针的方式

假定：`node* head;`

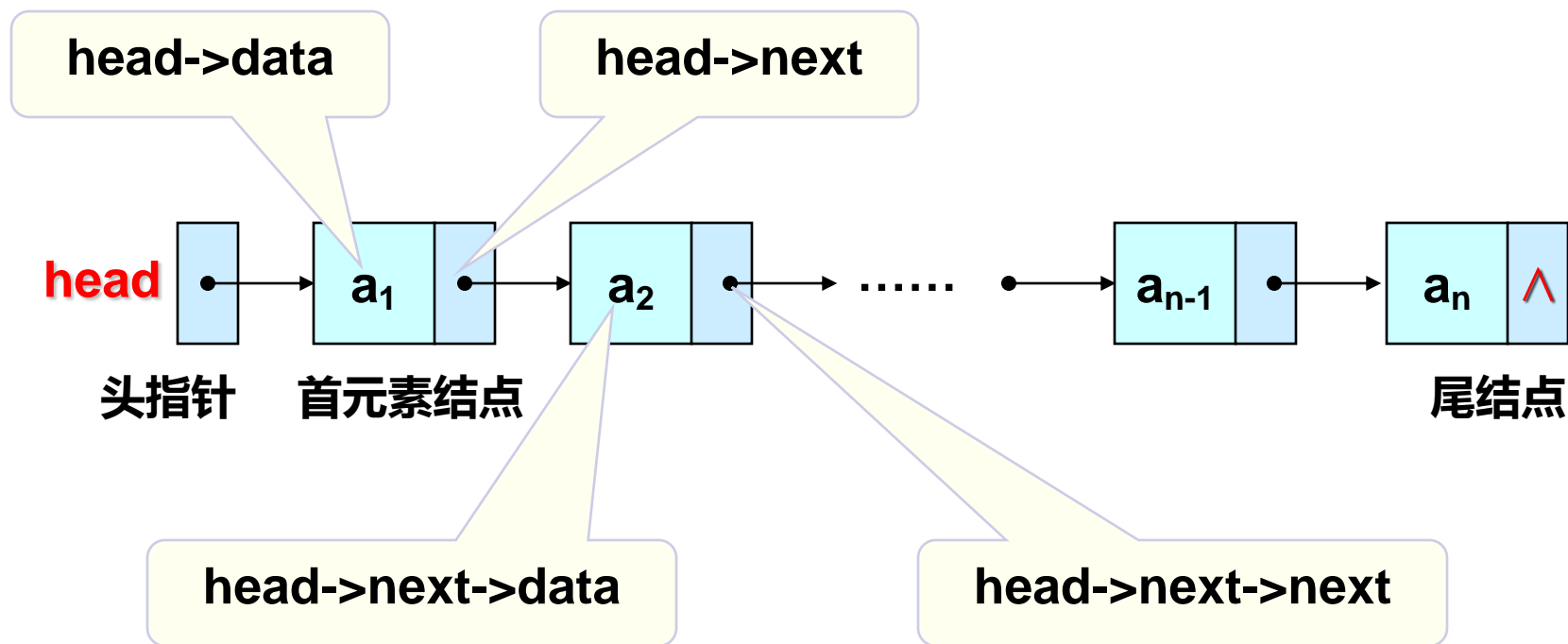
(1) 方式一（常用）

☞ `head->data`为a1; `head->next->data`为a2; ...

(2) 方式二

☞ `(*head).data`为a1; `(*(*head).next).data`为a2; ...

- `(*head)` 为node类型，是一个结构类型；而 `head` 是一个node结构类型的指针。所以成员（分量）的引用方式不同。如下图所示：



【讨论】

- ❏ 结点数量动态变化的，也叫动态链表；
- ❏ 插入新结点时，需要临时申请内存空间：
 - ✦ malloc、free —— C和C++申请和释放内存；
 - ✦ new、delete —— C++ 申请和释放内存。
- ❏ malloc、free 为C语言的库函数
- ❏ new、delete 为C++的操作符
- ❏ 它们必须成对使用，否则造成内存泄漏（memory leak）
- ❏ 顺序表中为什么不需要这样处理呢？

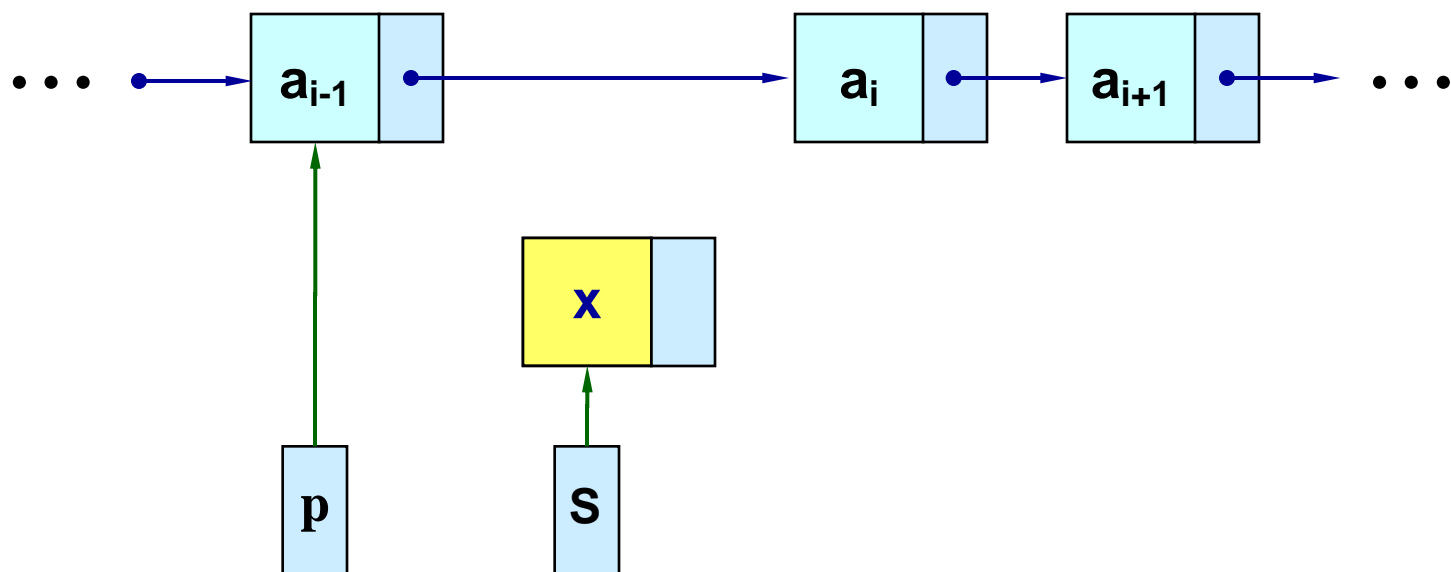
3. 单链表的进一步讨论 → 带头结点的单链表

- ☞ 我们先讨论单链表的插入操作，然后引出“**带头结点的单链表**”概念。
- ☞ 假定：待插入结点的数据元素为**x**，结点指针（地址）为**S**，插入到线性表的第**i**个结点位置。

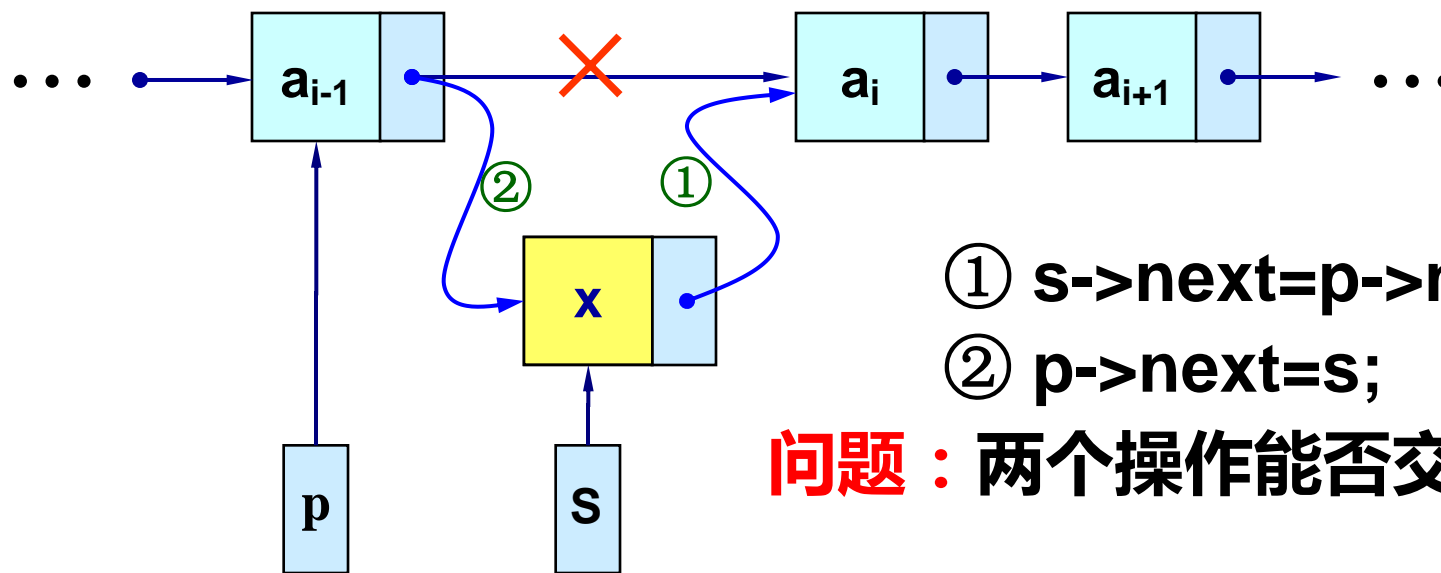
(1) 情况一：

- ☞ $1 < i \leq n$ ，即插入位置不是第一个结点，也不是最尾位置。操作过程如下图所示。

■ 搜索插入位置，申请新结点



- ①修改待插入结点的next指针，使指向原线性表的第 i 个结点。即： $s \rightarrow next = p \rightarrow next$;
- ②修改第 $i-1$ 个结点的指针域，使其指向新结点 s 。即： $p \rightarrow next = s$;



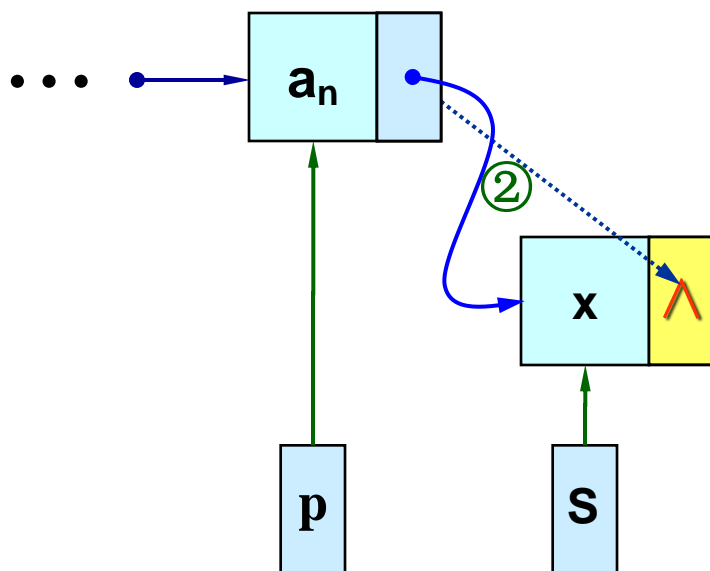
问题：两个操作能否交换次序？

(2) 情况二：插入位置 $i=n+1$ ，即插到最后。

① $s \rightarrow \text{next} = \text{NULL}$;

② $p \rightarrow \text{next} = s$;

■ 是否可以沿用“情况一”的插入操作呢？



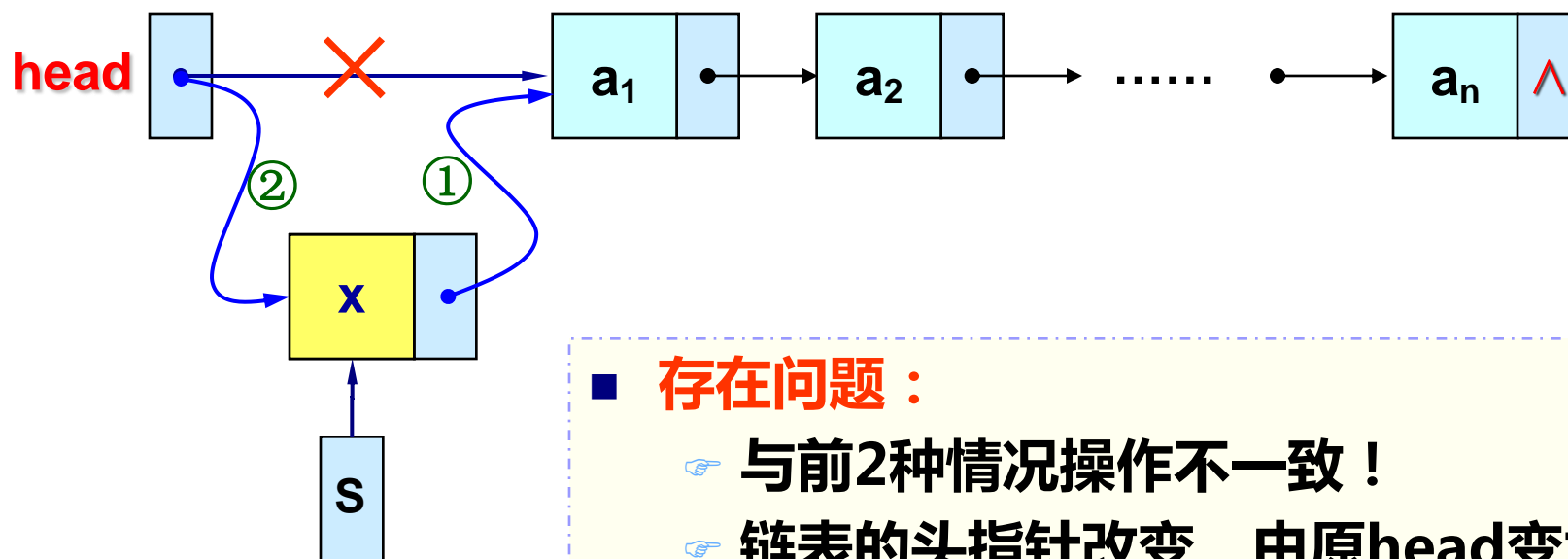
① $s \rightarrow \text{next} = p \rightarrow \text{next}$; 或
 $s \rightarrow \text{next} = \text{NULL}$;

② $p \rightarrow \text{next} = s$;

(3) 情况三：插入位置 $i=1$ ，即插入成为第一个结点。

① $s \rightarrow \text{next} = \text{head}$;

② $\text{head} = s$;



■ 存在问题：

- 与前2种情况操作不一致！
- 链表的头指针改变，由原head变为s。

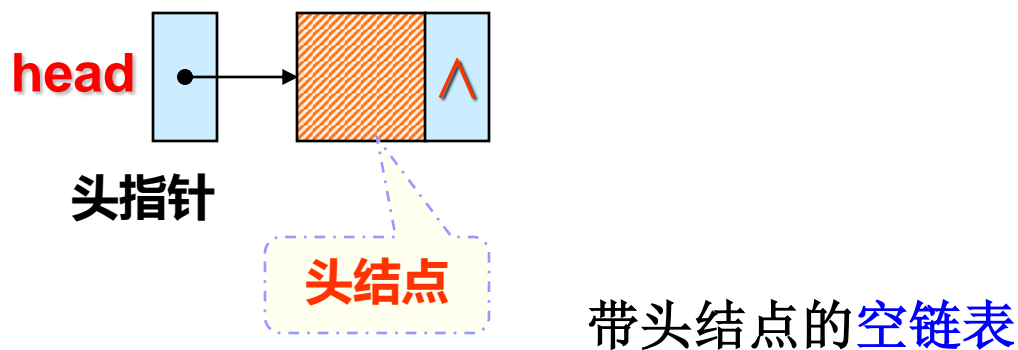
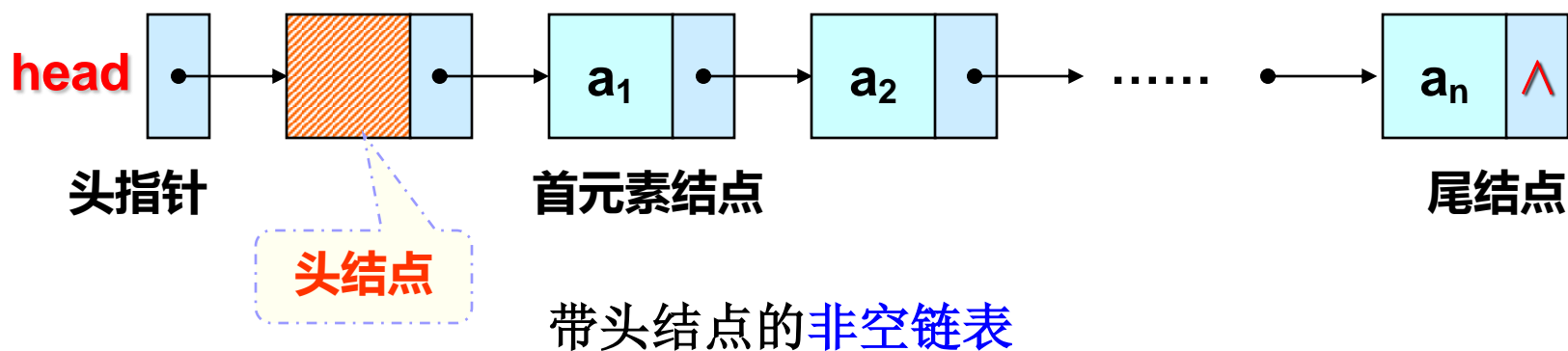
■ 删除结点是否存在同样的问题呢？

--答案是肯定的！

■ 带头结点的单链表

- ☞ 在头指针后，第一个元素结点之前人为附加一个结点，叫做**头结点**；
- ☞ **头结点类型和元素结点类型相同，同为node类型；**
- ☞ head指针指向头结点；
- ☞ **头结点的数据域不用，也可以存储链表长度等信息；**
- ☞ 头结点的指针域（next）指向第一个数据元素结点（首元素结点）
- ☞ **头结点不能删除；**
- ☞ 头结点前不能插入结点，即插入的结点必须位于头结点之后。

■ 带头结点的单链表图示



■ 加了头结点之后

- ☞ 任何位置的插入和删除操作方式相同；
- ☞ 链表一旦建立，头指针（首地址）始终不变。

■ 空链表

- ☞ 有了头结点后，我们还可以创建空链表（见上图）。
- ☞ 没有头结点情况下，没法创建空链表，或说没有头结点的头指针是不确定的。

■ 【说明】后面的内容，如无特殊说明，全部基于带头结点的链表！！！！

■ 单链表LinkedList类的C++完整描述

```
class LinkedList
```

```
{
```

```
public:
```

```
    LinkedList();    //初始化空链表
```

```
    int length();    //求链表长度（结点数）
```

```
    bool getElement(int i, elementType &x); //按序号取元素
```

```
    node* locate(elementType x); //查找元素，返回目标指针
```

```
    bool listInsert(int i, elementType x);
```

```
    bool listDelete(int i);
```

```
    void createListR(); //尾插法创建单链表
```

```
    void createListH(); //头插法创建单链表
```

```
    void print();       //打印单链表元素
```

```
    void destroy();     //销毁所有node结点，否则内存泄漏。  
                        //此函数功能也可以放在析构函数完成。
```

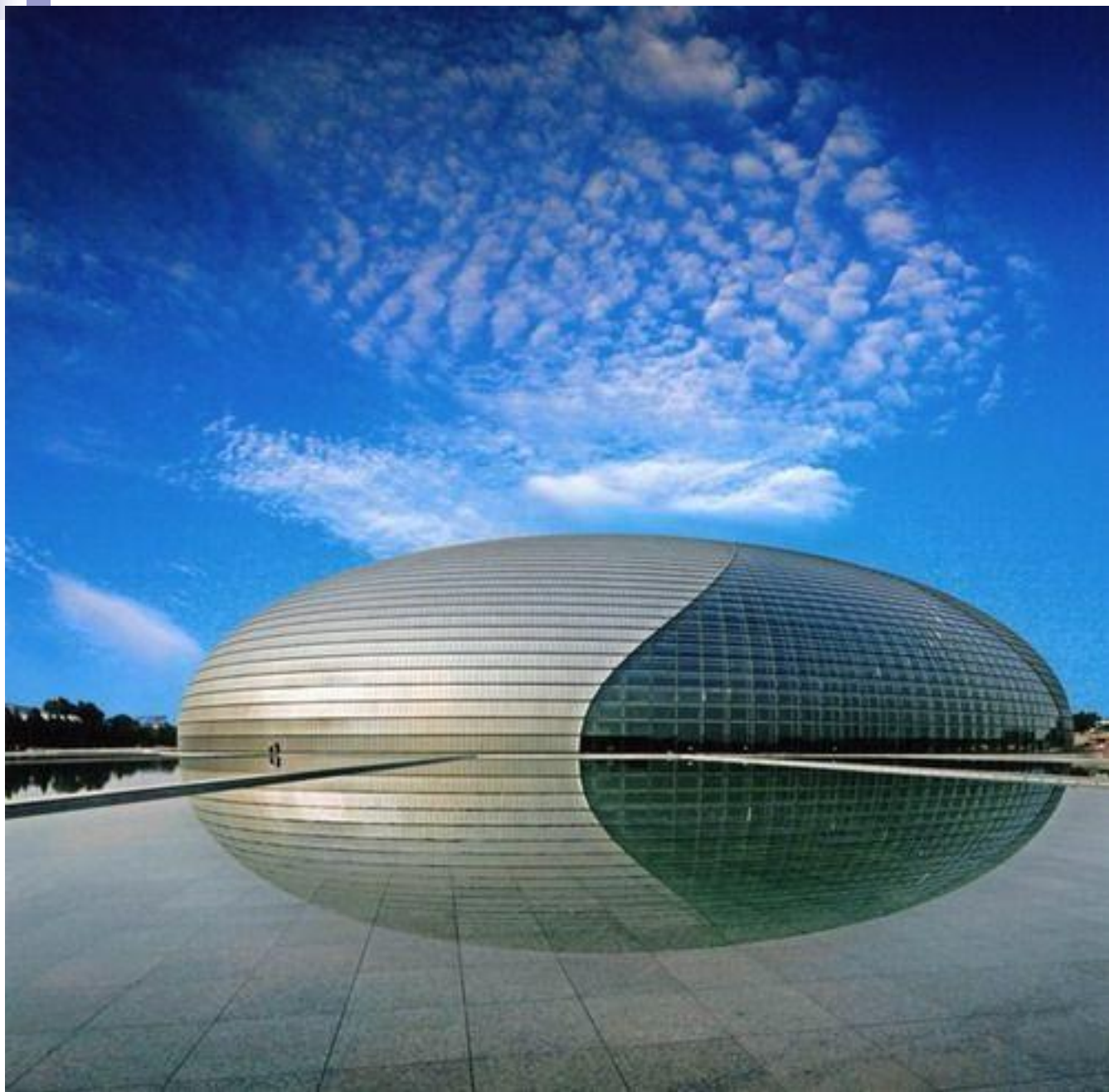
```
private:
```

```
    node * head;        //单链表头指针。node可用类或结构体定义
```

```
};
```

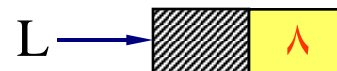
百川东到海，
何时复西归？
少壮不努力，
老大徒伤悲。

长歌行·佚名



2.3.2 单链表运算的实现

1. 初始化链表



👉 创建一个只有头结点，长度 $n=0$ 的链表。

【方法一】用函数返回值传递链表（最易理解）

```
node* initialList( )  
{  
    node* p;  
    p=new node; //动态申请内存，作为头结点。  
    p->next=NULL;  
    return p;    //返回创建的只有头结点的单链表  
}
```

【算法分析】时间复杂度 $O(1)$ 。

【方法二】使用“引用”从子函数往主函数传递创建的链表

```
void initialList(node * &L)
{
    //申请内存产生头结点，OS根据内存使用情况，
    //确定节点在内存中的位置，即：L的值。
    //所以，L在使用new node 前后值会不同
    L=new node;
        //或 L=(node*)malloc(sizeof(node));
    L->nex=NULL; //next 域为空
}
```

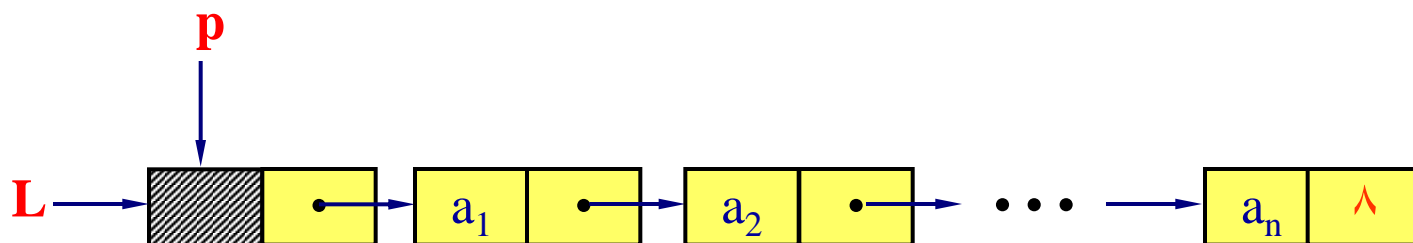
【问题】函数可以定义成 void initialList(node* L) 吗？

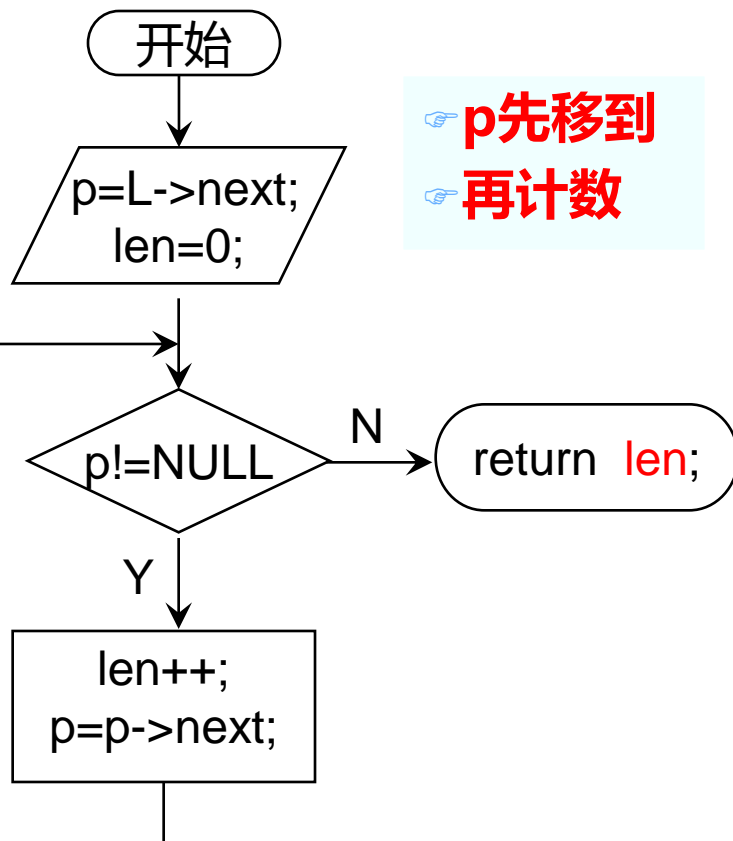
【方法三】用指针的指针传递链表

```
void initialList1( node** A )
{   //A 为指向node型节点的指针的指针 ,
    //即 : 节点指针的地址。
    (*A)=new node;
    //(*A)为指向node型节点的指针 ;
    //( **A) 为node型结点结构体 ;
    //动态在heap上申请内存 , 保存节点 ( 头结点 )
    (*A)->next=NULL;
}
```

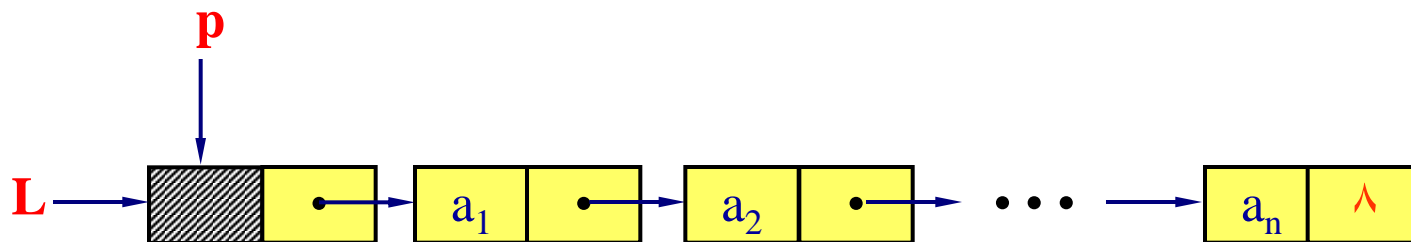
■ 2. 求链表的长度（不包括头结点）

- 与顺序表不同，链表没有listLen分量；
- 通过数出结点个数 n ，即链表长度；
如： $L=(5,2,4,8,1)$
- 用一个指针 p ，初始指向首元素结点， $p=L->next$ ；
- 用一个变量len记录数过的结点数，初始 $len=0$ ；
- 指针 p 每指向一个结点计数len加1，然后 p 移到下一个结点，即： $p=p->next$ ；
- 直到 $p==NULL$ ，数出全部元素个数 n ，即链表长度。





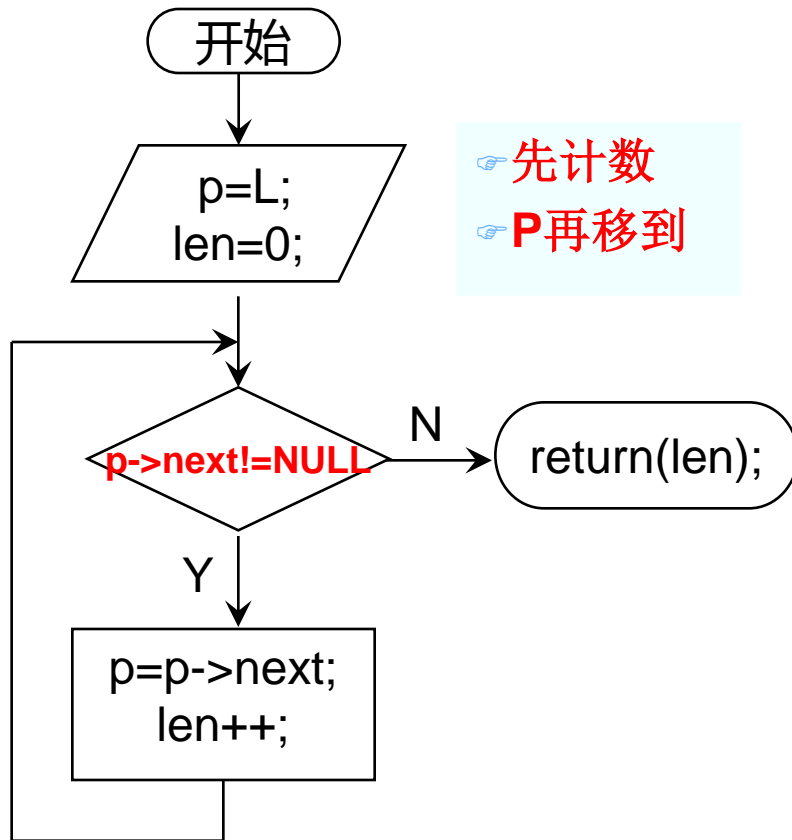
→ **p先移到**
→ **再计数**



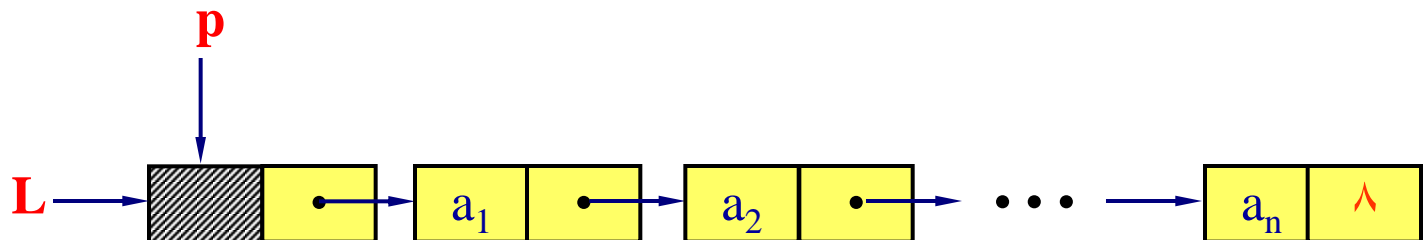
【算法描述】

```
int listLength(node* L)
{ int len=0; //保存长度值，初始化为0
  node* p=L->next; //p指向第一个元素结点
  while(p!=NULL)
  { //先指向、再计数
    len++; //p!=NULL说明存在元素结点，计数加1
    p=p->next; //p后移指向下一个结点
  }
  return len; //返回总计数值，即长度
}
```


■ 2. 求链表的长度（不包括头结点）-- 另一种实现



```
int listLength(node* L)
{ int len=0;    //保存长度值
  node *p=L;   //p指向头结点
  while(p->next!=NULL) //不同点
  {
    len++;      //计数加1
    p=p->next;  //p移到已经计数结点
  }
  return len;   //返回长度
}
```



【算法分析】

👉 时间复杂度： $O(n)$

【思考问题】

- 👉 是否可以用指针和引用来返回长度值呢？如何实现？
- 👉 本函数的形参为什么可以用单指针呢？

■ 3. 按序号取元素

☞ 在链表L中取出第i个元素。成功，返回目标结点指针；失败返回空指针。

☞ 如：L=(5,2,4,8,1)，i=3, 1, 5, 0, 6

■ 问题：怎么知道第i个结点（元素）呢？

☞ 用一个指针p，初始指向首元素结点，p=L->next；

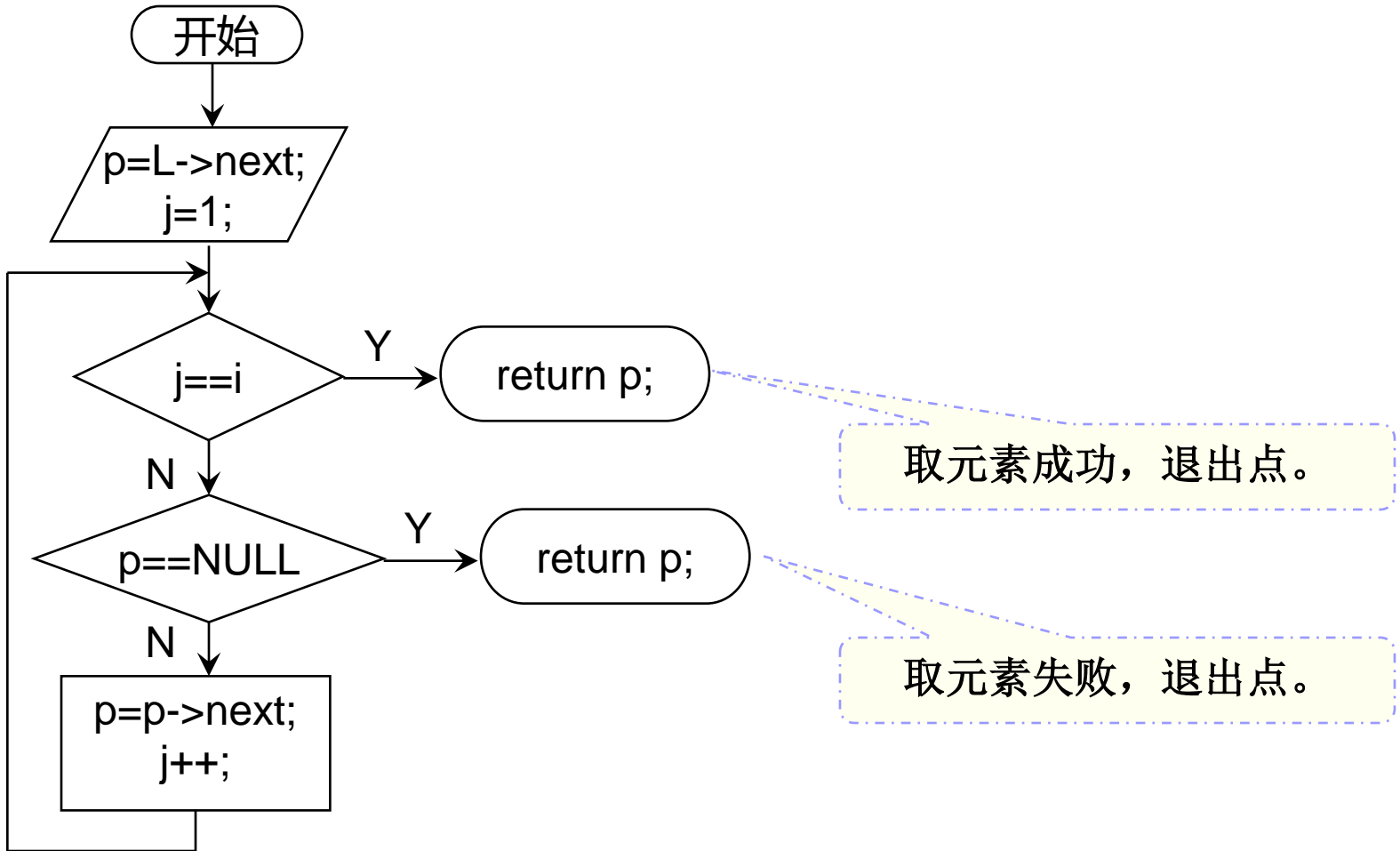
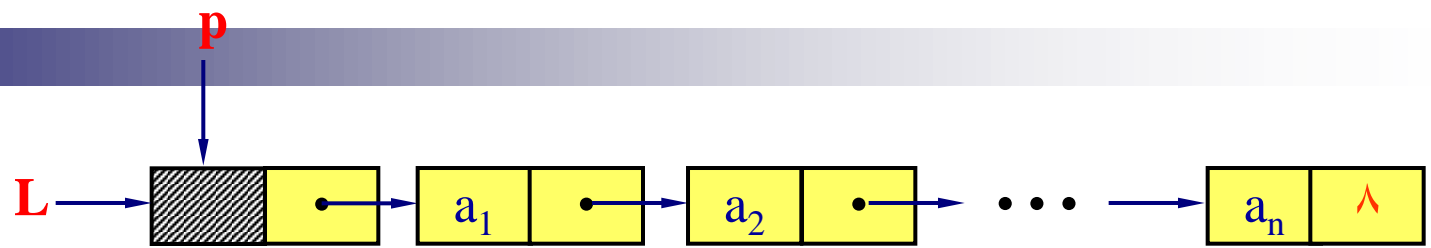
☞ 用一个计数变量j记录数过的结点数，初始j=1；

☞ 指针p每指向一个结点计数j加1，然后p移到下一个结点，即：p=p->next；

☞ 直到j==i 或 p==NULL，退出循环；

☞ 若j==i，p指向a_i，即目标结点；

☞ 若p==NULL，i超出范围，目标结点不存在。



■ 【算法描述】

```
node* getElement(node *L, int i)
```

```
{
```

```
    node* p=L->next;
```

```
    int j=1;    //从1号元素开始搜索。
```

```
    while( (j!=i) && (p!=NULL) ) //当前节点不是目标节点，  
                                   //又不为空，继续处理下一个节点
```

```
    { p=p->next;
```

```
      j++;    }
```

```
    return p; //当j==i时，p为目标节点；
```

```
              //否则p==NULL，序号超出范围，取元素失败。
```

```
}
```

【算法分析】

👉 时间复杂度： $O(n)$

【思考问题】

- ① 分析为什么当 $i < 1$ 及 $i > n$ 两种无效范围，p都为NULL？
- ② 能否用指针和引用返回目标节点呢？如何实现？（见实现代码）
- ③ 形参为什么可以用单指针呢？

4. 按元素值查找元素

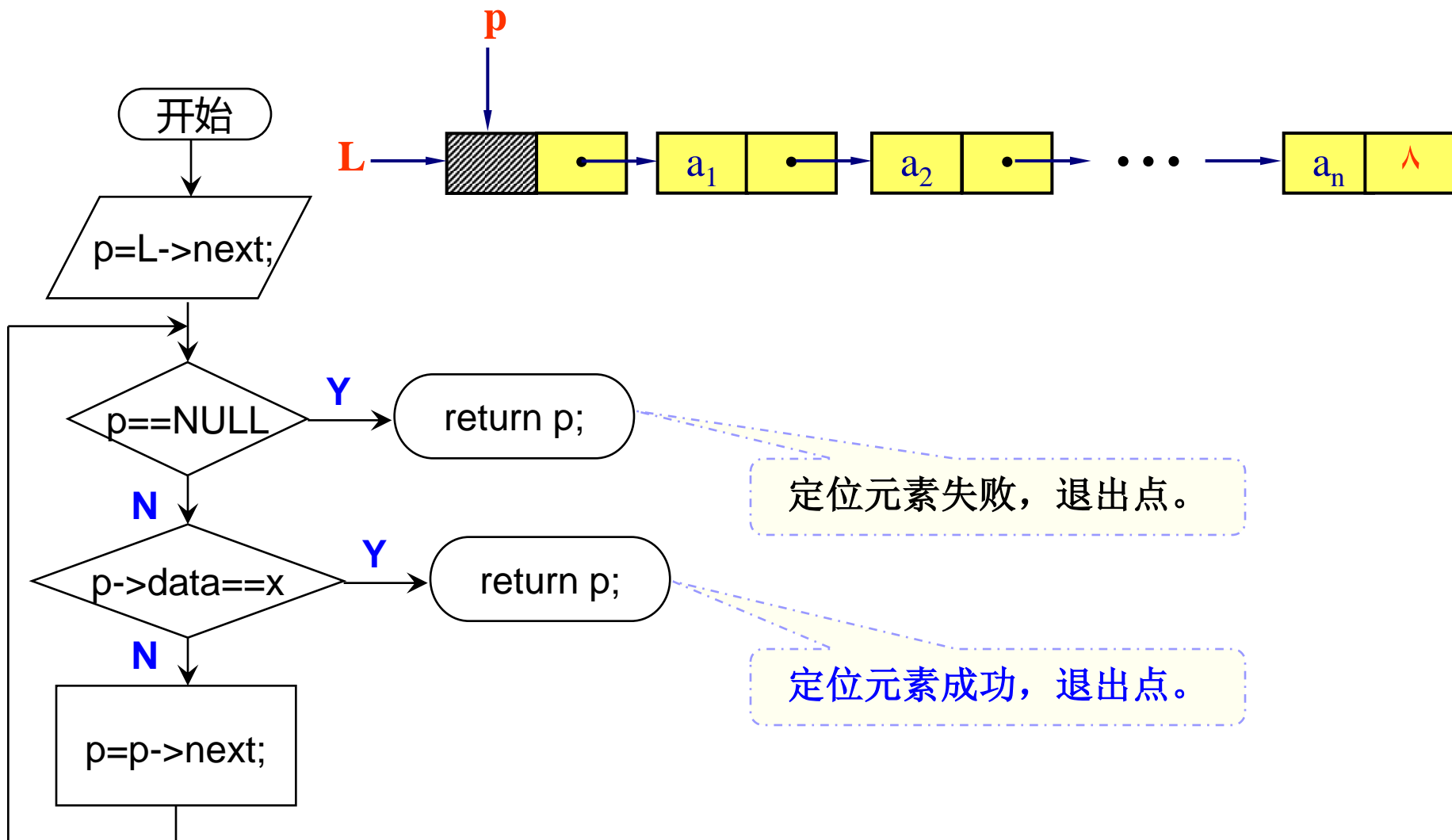
- ☞ 给定元素值 x ，在链表 L 中搜索（定位）此元素结点，**成功**，返回目标结点指针；**否则**返回NULL。

【算法思想】

- ☞ 与按序号取元素类似，用一个指针 p ，逐个指向链表的每个结点；
- ☞ 对每个结点，比较 $p \rightarrow data == x$ ？，若相等 p 指向的即为目标结点，返回 p 即可；
- ☞ 否则，继续搜索下一个结点， $p = p \rightarrow next$ ；
- ☞ 若循环结束，说明 x 不在链表中，返回**NULL**。

4. 按元素值查找元素

成功，返回目标结点指针；否则返回NULL。



【算法描述1】

- 函数返回值返回目标指针。

```
node* locate(node* L, elementType x)
```

```
{
```

```
    node* p=L->next;
```

```
    while(p!=NULL)
```

```
    {
```

```
        if(p->data==x)
```

```
            return p; //查找成功，返回目标结点指针
```

```
        else
```

```
            p=p->next; //指针后移到下一个结点，继续查找
```

```
    }
```

```
    return NULL; //查找失败，返回空指针
```

```
}
```

【算法描述2】

- 函数引用参数返回目标指针。

Bool locate1(node* L, elementType x, node* &p)

```
{
    p=L->next;
    while(p!=NULL)
    {
        if(p->data==x)
            return true; //成功定位，返回true
        else
            p=p->next; //指针后移到下一个结点，继续查找
    }
    return false; //查找失败，返回false
}
```

【算法描述3】

- 函数双重指针参数返回目标指针。

```
Bool locate2(node* L, elementType x, node **A)
{
    (*A)=L->next;
    while((*A)!=NULL)
    {
        if((*A)->data==x)
            return true;           //成功定位，返回true
        else
            (*A)=(*A)->next;      //指针后移，继续查找
    }
    return false;                 //查找失败，返回false
}
```

【算法分析】

👉 时间复杂度： $O(n)$

【思考问题】

- ① 能否同时返回结点序号？如何实现？
- ② 在后两种实现中，为什么不能用单指针实现？



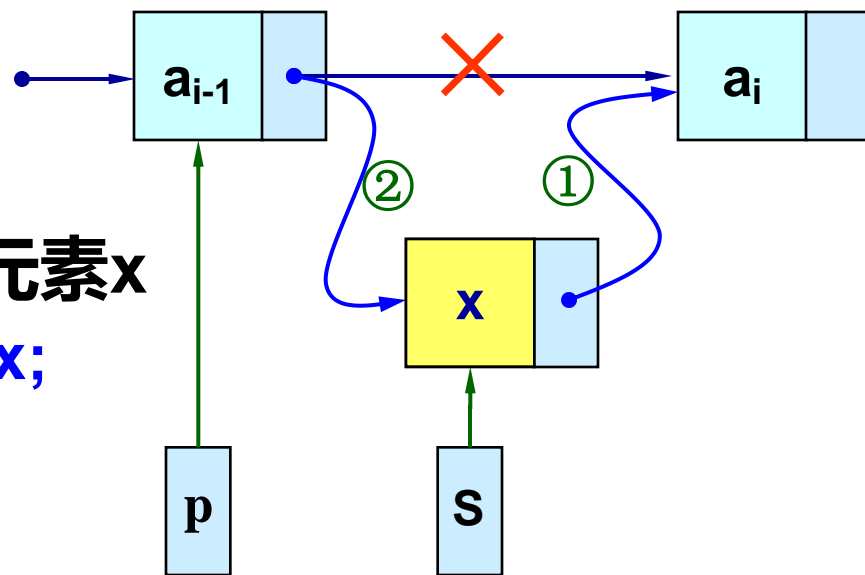
上大学：学知识、长见识、提素质、增能力

5. 插入结点算法

- ➡ 给定 i 位置，插入元素 x 结点。回顾：前面讨论的插入过程。成功：返回true；失败：返回false。
- ➡ 如： $L=(5,2,4,8,1)$ ， $i=3$ ，1, 5, 6, **0, 7**

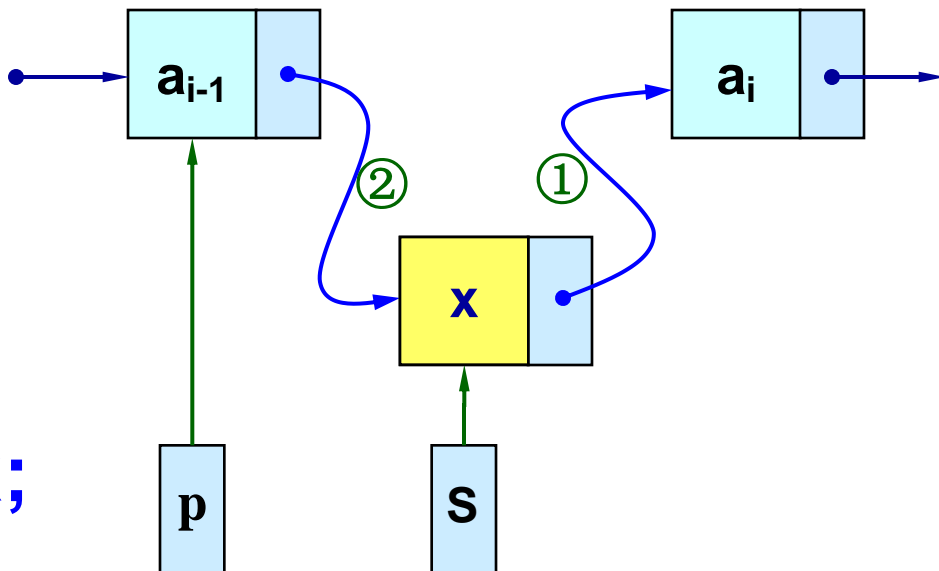
【问题分析】

- ➡ (1) 搜索插入位置
 - ✦ 指针 p 指向结点 a_{i-1}
- ➡ (2) 申请新结点，装入元素 x
 - ✦ $s=\text{new node}; s\rightarrow\text{data}=x;$
- ➡ (3) 插入新结点。
 - ✦ $s\rightarrow\text{next}=p\rightarrow\text{next};$
 - ✦ $p\rightarrow\text{next}=s;$



■ 插入新结点后情况讨论

- ❏ $p \rightarrow \text{next} = s$;
- ❏ $(p \rightarrow \text{next}) \rightarrow \text{data} = x$;
- ❏ $s \rightarrow \text{data} = x$;
- ❏ $(p \rightarrow \text{next} \rightarrow \text{next})$ 指向 a_i ;
- ❏ $(s \rightarrow \text{next})$ 指向 a_i ;
- ❏ $(p \rightarrow \text{next} \rightarrow \text{next}) \rightarrow \text{data}$ 是什么呢?
- ❏ $(p \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{next}) \rightarrow \text{data}$ 呢?



■ 插入算法描述

```
void listInsert( node* L, int i, elementType x )
{
    node* p=L; node* s;   int k=0;
    while( k!=i-1 && p!=NULL ) { //搜索 $a_{i-1}$ 结点指针p ,
        p=p->next;                //p指向下一个节点
        k++;                       //k=i-1或p=NULL退出
    }
    if( p==NULL )    error(“序号错” );
    else { //此时 , k=i-1 , p为 $a_{i-1}$ 节点的指针
        s=new node;           // 创建一个新节点
        s->data=x;             //装入数据
        s->next=p->next;       //插入新节点
        p->next=s;             }
}
```


■ 插入操作的一种实现代码（可有多种实现）

```
bool listInsert(node* L, int i, elementType x)
{ node* p=L; node* S; int k=0;
  while(k!=i-1 && p!=NULL) { //搜索 $a_{i-1}$ 节点指针
    p=p->next; //p指向下一个节点
    k++; }
  if(p==NULL) return false; //i超范围，不能插入，
  else { //此时， $k=i-1$ ，p为 $a_{i-1}$ 节点的指针
    S=new node; //创建一个新节点
    S->data=x; //装入数据
    S->next=p->next; //插入新节点
    p->next=S;
    return true; //正确插入，返回true
  }
```

【算法分析】

☞ 时间复杂度： $O(n)$

【思考问题】

☞ 分析为什么插入位置 $i < 1$ 及 $i > n+1$ 时， p 都为NULL？

☞ 插入算法中为什么初始化时 $p=L$ ，而不是 $p=L \rightarrow next$ ？



6. 删除结点算法

删除序号为 i 的结点。

如： $L=(5,2,4,8,1)$ ， $i=3, 1, 5, 0, 6$

【算法思想】

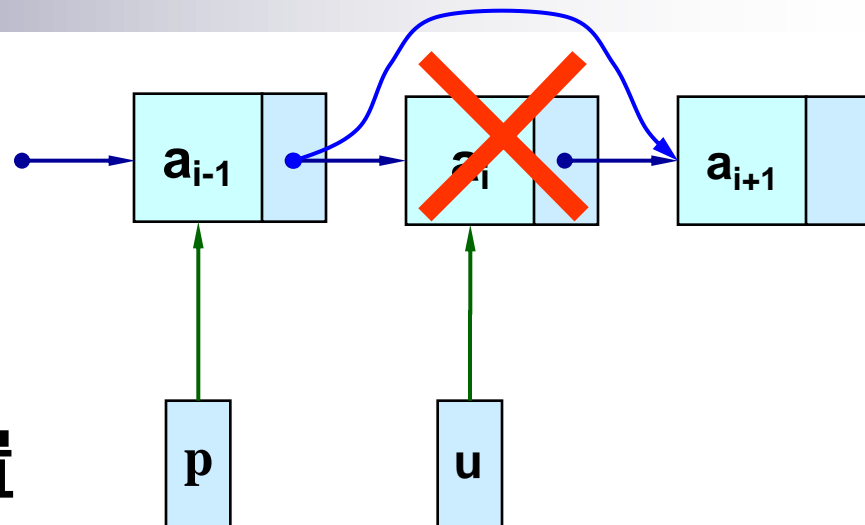
(1) 搜索待删除结点前驱位置

- ➡ 第 $i-1$ 结点，指针 p ；
- ➡ 待删结点，第 i 结点，指针 u 。

(2) 删除目标结点

- ➡ 删除结点仍在内存；
- ➡ $u \rightarrow \text{next}$ 仍指向 a_{i+1} 结点。

(3) 释放结点内存



① $u = p \rightarrow \text{next};$

② $p \rightarrow \text{next} = u \rightarrow \text{next};$ 或

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

③ $\text{delete } u;$ 或

$\text{free}(u);$

■ 删除结点的算法描述

```
void listDelete( node* L, int i )
{ node* u;    node* p=L;    int k=0;
  while( k!=i-1 && p!=NULL ) { //搜索 $a_{i-1}$ 节点指针
    p=p->next;    k++;          }
  if(p==NULL || p->next==NULL)
    error(“删除序号错” ); //删除位置 i 超出范围
  else { //此时, p指向 $a_{i-1}$ 结点
    u=p->next; //u指向待删除节点  $a_i$ 结点
    p->next=u->next; //  $a_{i-1}$ 的next指向 $a_{i+1}$ 结点,
                     //或为空 (  $a_{i-1}$ 为最后结点 )
    delete u; //释放删除节点占据的内存空间, 此句必须,
               //否则这个节点的内存将成为垃圾内存, 泄漏
  }
```

■ 删除结点的一种实现代码

```
bool listDelete( node* L, int i )
{
    node* u;    node* p=L;    int k=0;
    while(k!=i-1 && p!=NULL) { //搜索ai-1节点
        p=p->next; k++;
    }
    if(p==NULL || p->next==NULL)
        return false; //删除位置 i 超出范围，删除失败，返回false
    else { //此时，p指向ai-1
        u=p->next; //u指向待删除节点 ai
        p->next=u->next; //ai-1的next指向ai+1节点，
                        //或为空（ai-1为最后节点）
        delete u; //释放删除节点占据的控件，此句必须，
                //否则这个节点的内存将称为垃圾内存(内存泄漏)
        return true; //删除成功，返回true
    }
}
```

【算法分析】

☞ 时间复杂度 $O(n)$ 。

【思考问题】

- ☞ 分析当 p 指向 a_n 结点时，怎样判断删除失败？
- ☞ 删除算法中为什么初始化时 $p=L$ ，而不是 $p=L \rightarrow next$ ？

7. 链表的构造（创建）

【算法思想】

- ① 申请产生头结点（如果头结点不存在）；
- ② 读入一个元素到变量 x ，可以从键盘、数组或文本文件读入；
- ③ 如果 x 是结束符，结束构造过程；
- ④ 否则，申请产生一个新结点并装入 x ；
- ⑤ 新结点插入链表 L 中；
- ⑥ 转②。



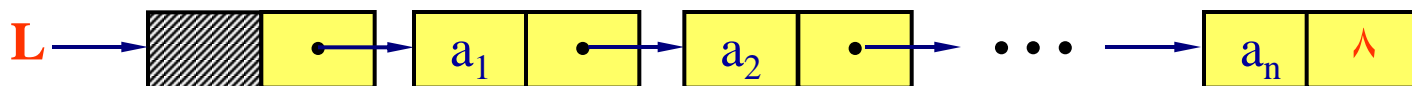
【问题】

① 在什么位置插入新结点？

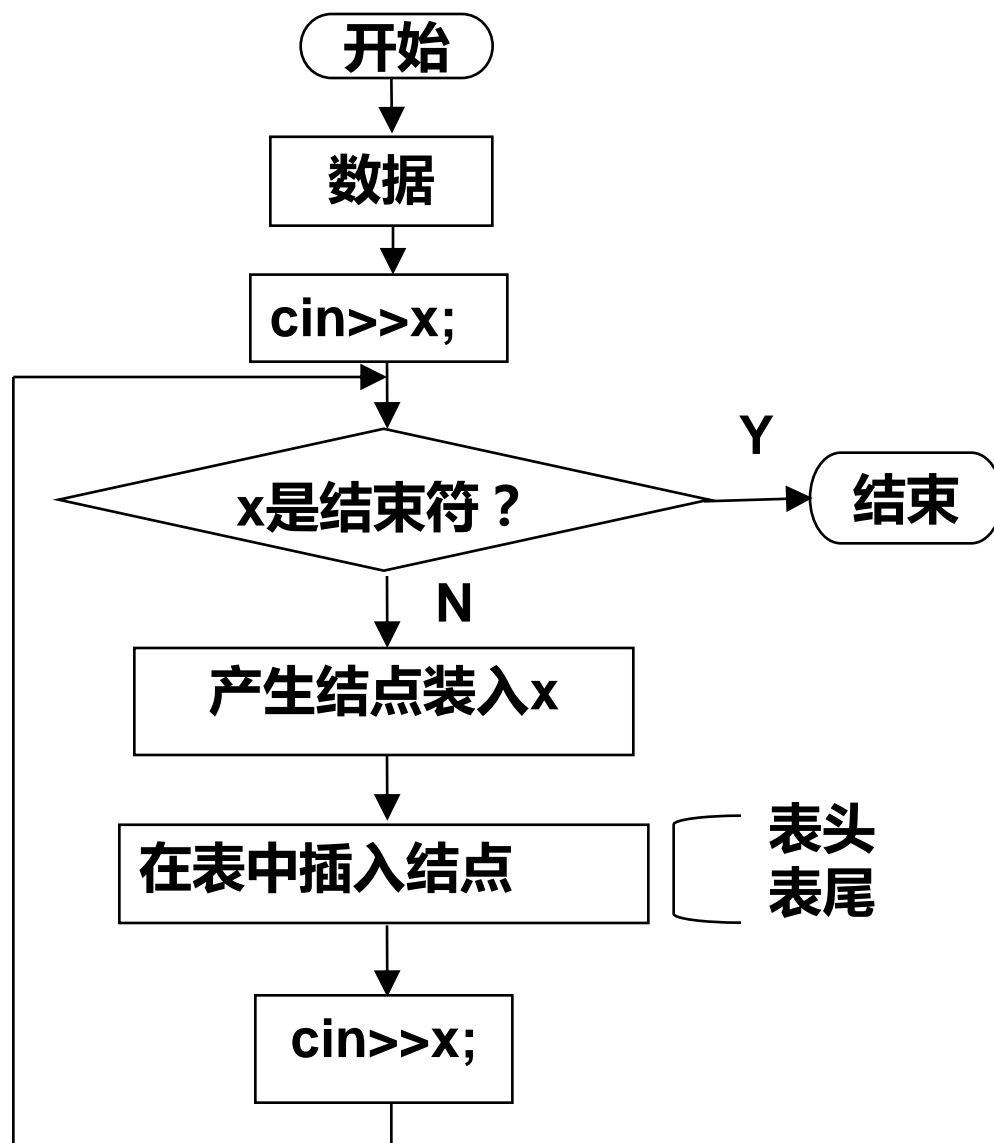
- 👉 **头插法**--新结点每次插入为第一个元素结点（头结点之后）。
- 👉 **尾插法**--新结点每次插入到表的最后，成为最后一个结点，即尾结点。

② 怎样结束插入操作？

- 👉 用特殊符号作为结束标志；
- 👉 规定结点数量，插完结束；
- 👉 从数据文件读入数据控制结束。



■ 键盘输入数据 创建单链表流程图



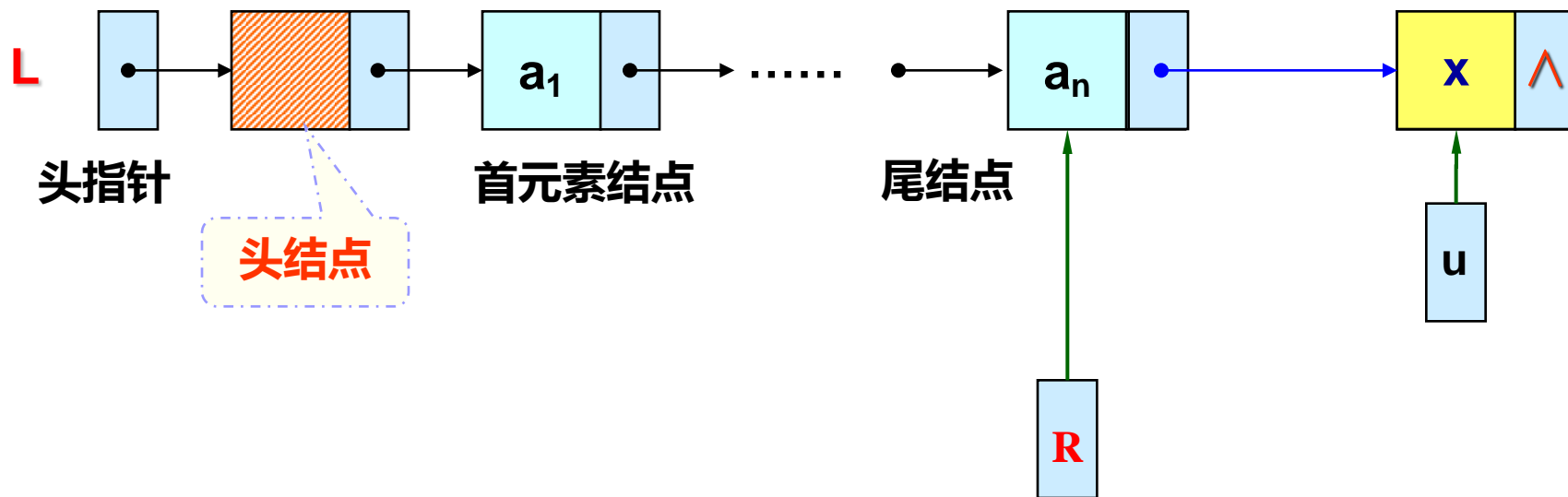
■ (1) 尾插法创建单链表

☞ 尾插法的关键是保留尾结点指针R，使快速定位到表尾。

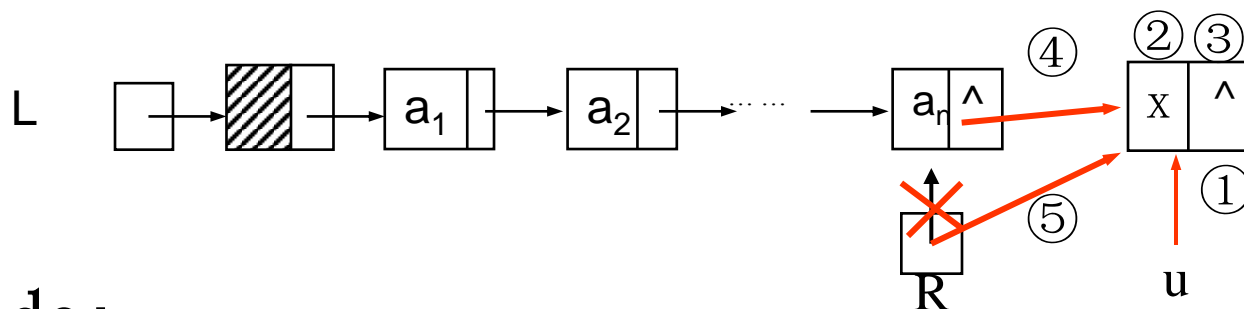
☞ $u = \text{new node}; u \rightarrow \text{data} = x; u \rightarrow \text{next} = \text{NULL};$

☞ $R \rightarrow \text{next} = u;$

☞ $R = u;$



■ (1) 尾插法创建单链表



① $u = \text{new node};$

② $u \rightarrow \text{data} = x;$

③ $u \rightarrow \text{next} = R \rightarrow \text{next};$ //或者 $u \rightarrow \text{next} = \text{NULL};$

④ $R \rightarrow \text{next} = u;$ //新结点接入链表

⑤ $R = u;$ //后移尾指针，u为新的表尾

【算法描述】

```
void createListR (node * & L)
```

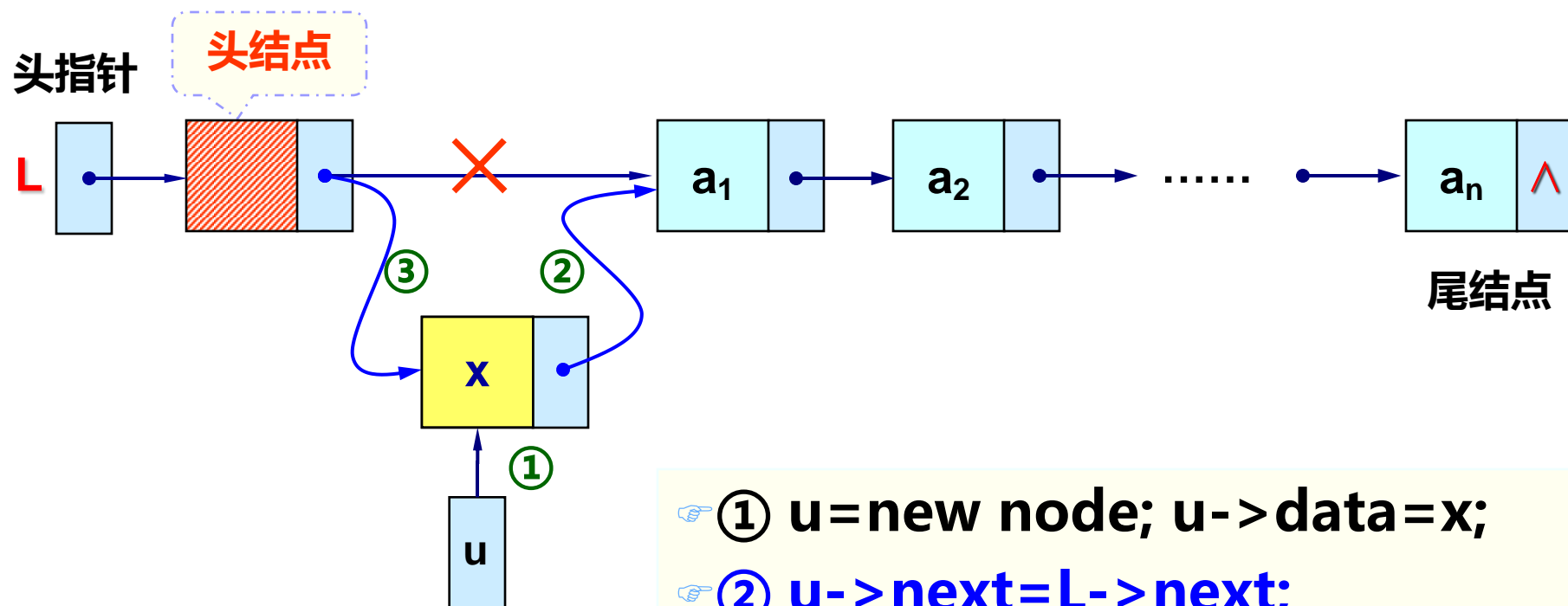
```
{  
    elementType x;  
    node *R,*u;  
    R=L;      //设置尾指针，对空表:头、尾指针相同  
    cout<<"请输入结点元素值（-9999退出）："<<endl;  
    cin>>x;  
    while(x!=-9999)      //假定-9999为结束符号  
    {  
        u=new node;      //申请一个新结点  
        u->data=x;        //元素数据写入新结点  
        R->next=u;        //新结点链接到表尾  
        R=u;              //新结点成为新的尾结点  
        cin>>x;  
    }  
    R->next=NULL;        //尾结点next置空。有无其它方式实现  
}
```

【算法分析】 时间复杂度： $O(n)$ 。

【思考问题】

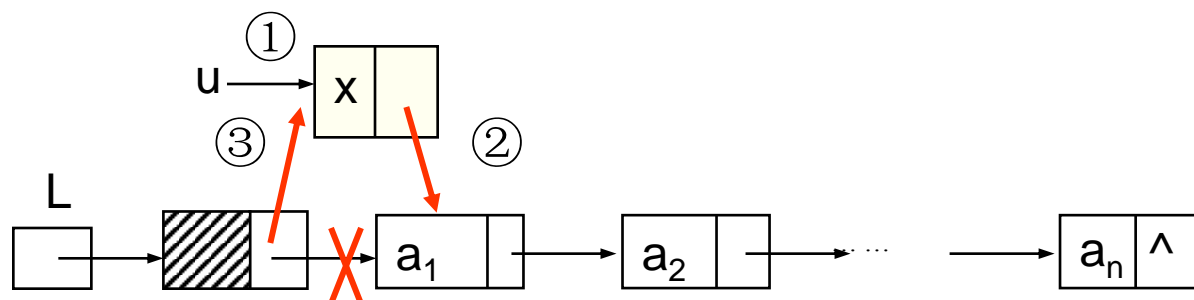
- 👉 可以用指针的指针、函数返回值返回创建的链表吗？
- 👉 有没有其它方式控制创建结束？

■ (2)头插法构造链表



- ① **u = new node; u->data = x;**
- ② **u->next = L->next;**
- ③ **L->next = u;**

■ (2)头插法构造链表



① $u = \text{new node}; u \rightarrow \text{data} = x;$

② $u \rightarrow \text{next} = L \rightarrow \text{next};$

③ $L \rightarrow \text{next} = u$

【算法描述】

```
void createListH( node *& L )
{
    node *u;
    elementType x;
    cout<<"请输入结点元素值 ( -9999退出 ) : "<<endl;
    cin>>x;
    while(x!=-9999) //假定结束符为-9999
    {
        u=new node; //申请一个新结点
        u->data=x;   //元素数据写入新结点

        u->next=L->next; //新结点插入到表头
        L->next=u;

        cin>>x;
    }
}
```

【算法分析】 时间复杂度： $O(n)$ 。

【思考问题】

- ① 有没有其它方式控制创建结束？
- ② 尾插法和头插法构造的链表有何区别？比如键盘输入元素的顺序为1、2、3、4、5，则两种方法创建的链表的结点顺序如何？

 **答案：**

- ✦ **头插法：**5、4、3、2、1
- ✦ **尾插法：**1、2、3、4、5

■ 8. 链表的销毁

- ☞ 用new 或 malloc() 动态申请的内存，用完后，必须显式的用delete 或 free() 释放。
- ☞ 即：new 和delete，malloc()和free()必须成对使用。
- ☞ 否则，OS 将一直不能使用这部分内存空间，造成内存泄漏。
- ☞ 释放单个结点，如指针p指向的结点，用下列语句即可：
 - ✦ delete p; 或 free(P);
- ☞ 要释放整条链表，必须从头结点开始，逐个结点进行释放。

【算法描述】

```
void destroyList( node* & L )
{
    node *p,*u;
    p=L;
    while(p)
    {
        u=p->next;
        delete(p);    //或 : free(p);
        p=u;
    }
    L=NULL;
}
```

//内存泄漏实验



寒雪梅中尽，春风柳上归。 【唐·李白】

2.3.3 链表的应用

【例2.5】设计算法，判断带头结点单链表L是否递增？若递增，则返回true，否则返回false。

【解题分析】

- (1) 链表空，返回true；
- (2) 只有一个元素，返回true；

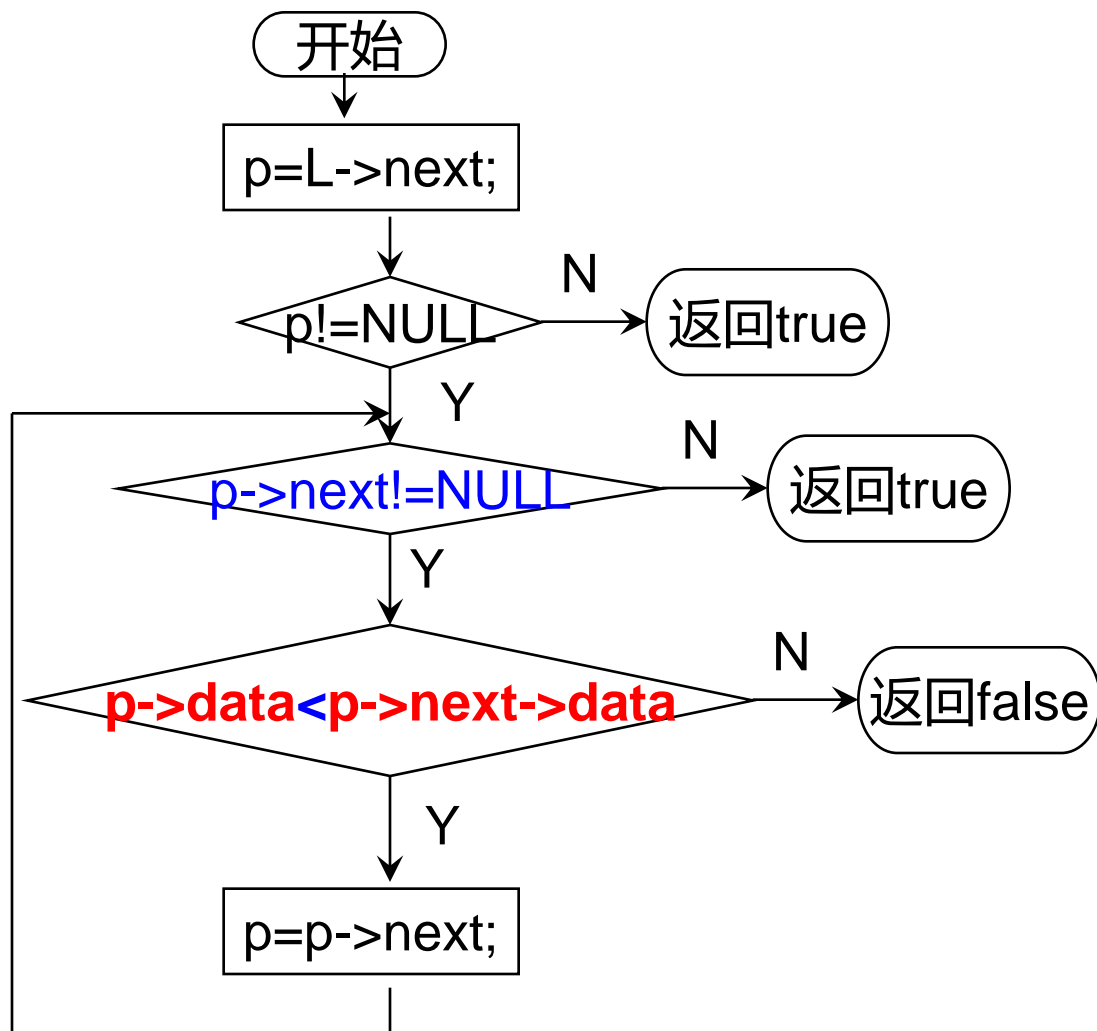
(3) 每个元素都小于其直接后继，返回true；
否则，返回false

☞ 用指针p依次指向每个结点，则p->next指向其直接后继结点。

☞ 若当前结点值小于直接后继结点值，即： $p->data < p->next->data$ ，将p移到下一结点，直到表尾，返回true。 Why?

☞ 否则，当前结点值大于下一结点值，非递增，直接返回false。

■ 由分析得如下流程图：



【算法实现】

👉 Exam205IncLinkedListJudge.cpp

```
bool IncListJudge(LinkedList &L)
{
    node* p=L->next; //p指向L的第一个元素节点
    if(p==NULL)
        return true;           //空表返回 true
    while(p->next!=NULL) //p指向比较的2个结点的前一结点
    {
        if(p->data<p->next->data)
            p=p->next; //当前2个相邻结点递增，P后移一个结点
        else
            return false; //非递增，返回false
    }
    return true; //所有元素都相邻递增，则L递增；
                //一个结点情况也在这返回。
}
```

【算法分析】

👉 时间复杂度： $O(n)$ 。

【思考问题】

- ① 算法中当 $p \rightarrow next == NULL$ 时， p 指向哪个结点？
- ② 本题用顺序表如何实现？

【例2.6】 链表L递增有序，插入元素x后仍保持递增有序。

如：L=(2,5,7,9,12,14,15)，x=8，17，1

【解题分析】

- ☞ 回忆P52页“例2.1”，本题的顺序表实现；
- ☞ 插入算法的变形，搜索合适的插入位置；
- ☞ 用指针p依次指向各个结点，如果 $p \rightarrow data < x$ ，移动p使其指向下一个结点，继续此过程直到第一个 $p \rightarrow data \geq x$ ，则p的位置即新结点的插入位置；
- ☞ 还有一种情况， $p = NULL$ ，说明x比任何结点的元素值都大，要插入链表最后；

❏ 如果按上述做法： $p=NULL$ 时，失去表尾指针。所以实际使用时用 $p \rightarrow next$ 依次指向各个结点，当 x 最大时， $p \rightarrow next$ 为空，但 p 正好指向原来的尾结点。即： p 指向直接前驱结点， $p \rightarrow next$ 指向插入结点位置。

【算法实现】

👉 **Exam206IncLinkedListInsert.cpp**

```
void incListInsert(linkedList &L,elementType x)
{
    node* u;
    node* p=L; //p指向头结点（头指针）
    while(p->next!=NULL && p->next->data<x) //搜索插入位置
    {
        p=p->next; //P后移一个结点
    }
    //循环结束p指向插入位置结点的直接前驱结点
    u=new node; //产生新结点
    u->data=x;
    u->next=p->next;
    p->next=u;
}
```

【算法分析】

👉 时间复杂度： $O(n)$ 。

【例2.7】 复制链表A的内容到新链表B中，结点逻辑顺序同表A。

【解题分析】

- ☞ 创建空链表B；
- ☞ 循环取出A的元素，创建新结点；
- ☞ 尾插法创建B。

【算法实现】

👉 Exam207ListCopy.cpp


```
void ListCopy(linkedList &A, linkedList &B)
{
    node* u;
    node* Pa=A->next; //Pa指向A的首元素结点
    node* Pb=B;        //Pb指向指向B的头结点
    node* Rb=B;        //Rb作为B的尾指针，初始化为B的头指针
    while(Pa!=NULL)
    {
        u=new node;
        u->data=Pa->data; //复制Pa指示结点元素到新结点
        Rb->next=u;       //u插入B表尾
        Rb=u;            //Rb重新指向B表尾
        Pa=Pa->next;      //取A的下一个元素
    }
    Rb->next=NULL;       //B的尾结点的next域置空
}
```


【例2.8】 递增链表A，B表示集合，设计算法求 $C = A \cap B$ ，要求时间性能最好，并分析其时间复杂度。

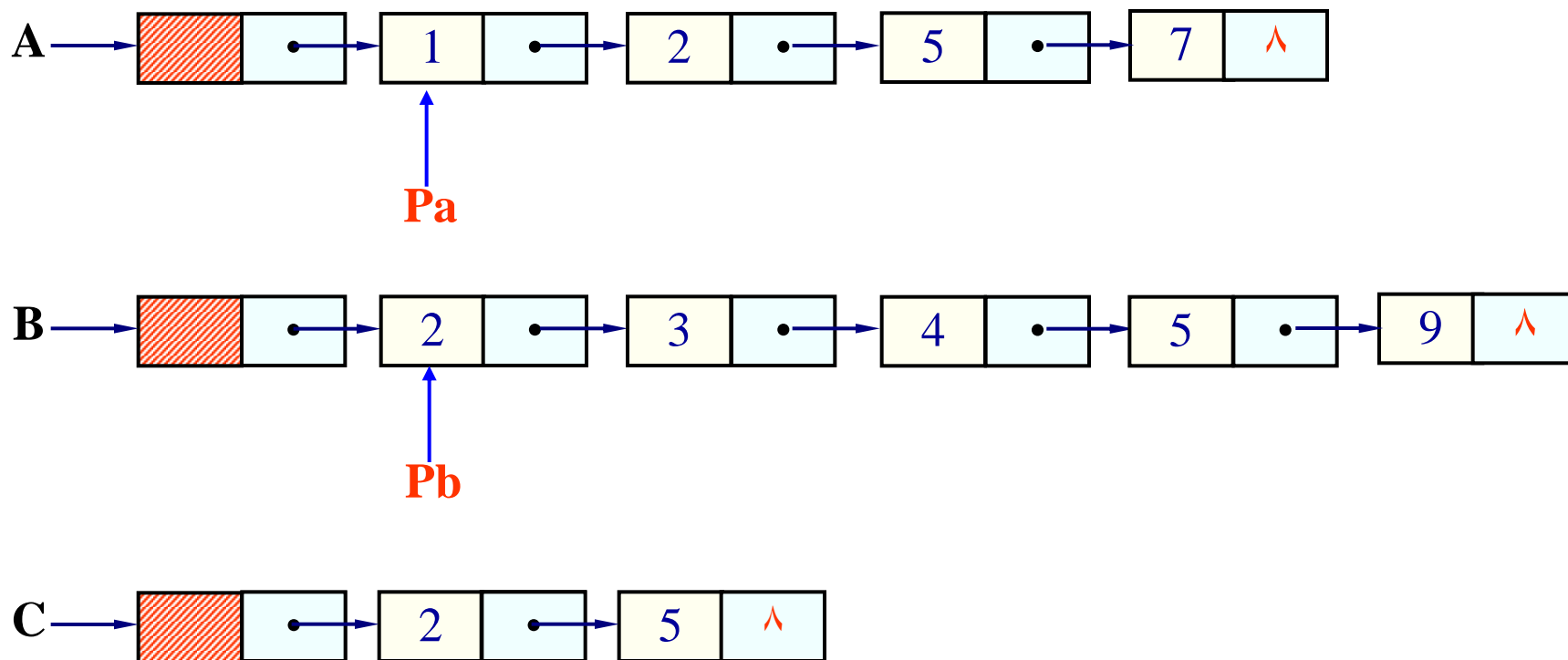
【解题分析】

☞ 容易想到的方法：用2层循环，**第一层**：依次取A的一个元素；**第二层**：将A的这个元素与B的元素依次进行比较——若B中有相同元素，加入C中，继续取A的下一个元素，直到A的元素取完。B每次从第一个结点开始比较。时间复杂度： **$O(|A|*|B|)$** 。

☞ **较好方法**：利用**递增有序**特点。用2个指针Pa和Pb分别指向A和B的结点，比较元素大小，会有3中情况：

- 
- ① **$Pa \rightarrow data == Pb \rightarrow data$** : 当前元素为交集元素，加入C中，Pa、Pb同时后移一个结点，即：
 $Pa = Pa \rightarrow next; Pb = Pb \rightarrow next$ ，继续循环；
 - ② **$Pa \rightarrow data > Pb \rightarrow data$** : A当前元素可能在B中当前结点的后面，移动Pb到下一结点，即：
 $Pb = Pb \rightarrow next$ ，继续循环；
 - ③ **$Pa \rightarrow data < Pb \rightarrow data$** : A当前元素值小于B当前元素值，此A元素不可能在B中，取A的下一个元素，即： **$Pa = Pa \rightarrow next$** ，继续循环；
- 循环直到A，或者B元素取完。

■ 一个实例：



【算法描述】

☞ exam28InterSet.cpp

```
void InterSet(LinkedList &A,LinkedList &B,LinkedList &C)
{
    node* Pa, *Pb, *Rc, *u;
    Rc=C;           //C表尾指针，空表时头尾指针相同
    Pa=A->next; //Pa指向A的第一个元素节点
    Pb=B->next; //Pb指向B的第一个元素节点
    while(Pa!=NULL && Pb!=NULL)
    {
        //A和B只要一个结束，退出循环
        if(Pa->data<Pb->data) // A当前元素不在B中
            Pa=Pa->next; //取A的下一个元素，回去循环
        else if (Pa->data>Pb->data)
            Pb=Pb->next; //A元素值大于B，移动Pb继续搜索
    }
```

else

```
{    //Pa->data==Pb->data , 即为交集元素 ,  
    //在C中产生新节点  
    //Pa , Pb同时后移 , 回去循环  
    u=new node;  
    u->data=Pa->data; //或u->data=Pb->data  
    Rc->next=u;      //尾插法在C中插入u ,  
    Rc=u;            //修改C的尾指针Rc , 指向u  
    Pa=Pa->next;     //Pa和Pb同时后移 ,  
                    //分别取A和B的下一个元素  
    Pb=Pb->next;  
}  
Rc->next=NULL;      //表C结束  
}  
}
```

【算法分析】

- 👉 因为找到的交集元素大于等于当前指针指向的元素，所以搜索从当前指针继续即可，不需从头开始。只需一层循环，**时间复杂度： $O(|A|+|B|)$**

【思考问题】

- 👉 本题用顺序表如何实现？

■ 【思考问题】

☞ 2.2.3 小节的各个例题用链表如何实现？

☞ 本小节的各个例题用顺序表如何实现？

☞ 分别用顺序表、单链表实现集合的并、交、差、判断子集等操作。特别是再加上“递增”或“递减”条件。

线性表链式存储结构（单链表）小结

■ 1. 链式存储结构的优点

- ☞ (1) 动态、按需申请空间，不会有空间闲置和溢出（空间满）问题；
- ☞ (2) 存储空间可以连续、或不连续，逻辑上相邻的元素，存储位置不一定相邻。
- ☞ (3) 插入、删除操作不需移动元素。

线性表链式存储结构（单链表）小结

■ 2. 链式存储结构的缺点

- ☞ (1) 指针域增加额外空间开销；
- ☞ (2) 只能按顺序进行存取，时间复杂度 $O(n)$ ；
- ☞ (3) 有些语言实现需要手工释放空间，处理不好易造成内存泄漏。

■ 链式存储结构适用情况

- ☞ 长度变化较大；
- ☞ 频繁进行插入、删除操作。

■ 单链表算法注意

- 👉 无论是**插入**，还是**删除**操作，都要先取得**目标结点直接前驱的指针**（**p指向 a_{i-1}** ），否则后续结点无法链接到表中。
- 👉 **插入时，先处理新结点指针，再修改原表结点指针；**
- 👉 **删除时，先处理指针，跨过待删除结点，最后再删除结点释放空间，**

【作业布置】

(P51-52)

👉 2.11

👉 2.18

👉 2.21

👉 2.22

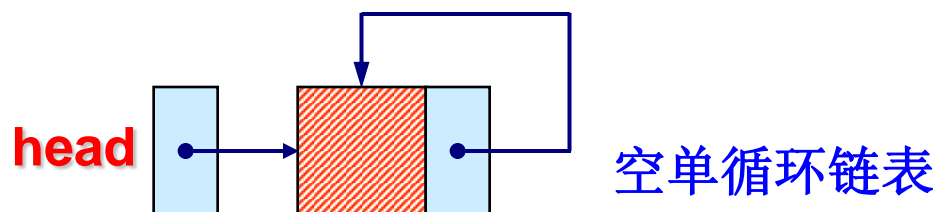
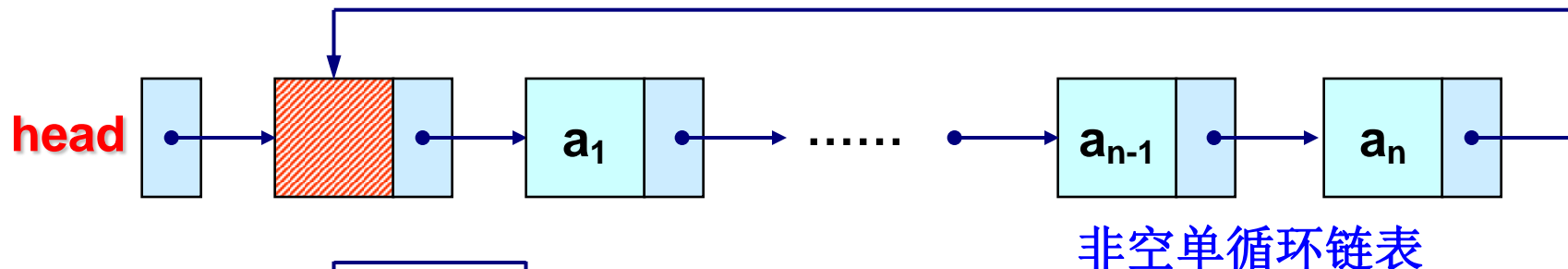


You think you can, you can.

2.4 其它结构形式的链表

■ 2.4.1 单循环链表

👉 尾结点的next指针指向头结点，为头指针，形成闭合环形链。



■ 优点：

👉 单链表访问任何元素（结点）都需要从表首（头指针）开始，循环链表从任一结点出发都可以访问其它任意结点。可在表中反复搜索。

■ 单循环链表基本操作的变化

①初始化

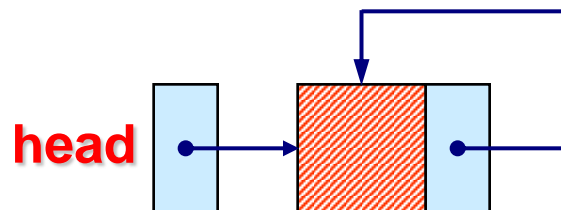
```
void initialList(node *&L)
```

```
{
```

```
    L=new node;    //申请头结点
```

```
    L->next=L;    //置为空表，形成循环
```

```
}
```



②求表长度—注意循环结束条件！

$P=L \rightarrow next; len=0;$

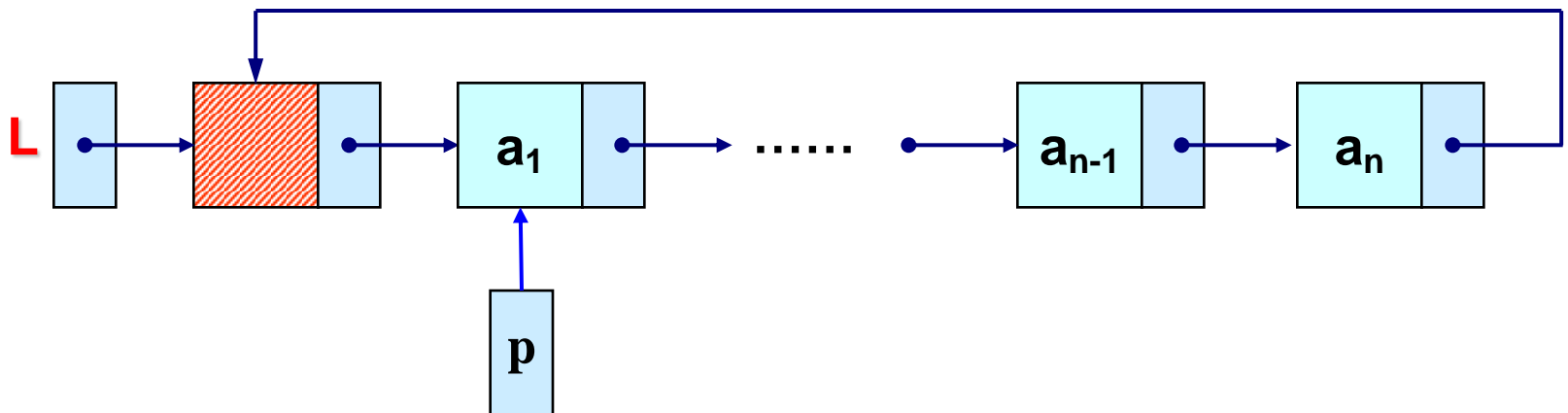
$while(P \neq L)$

{ //单链表循环结束条件 $P \neq NULL$

$P=P \rightarrow next;$

$len++;$

}



③插入—注意插入点判定条件！

```
bool listInsert(node *L, int i, elementType x)
{
    node* p=L;    //p指向头结点
    node* S;
    int k=0;        //k计数前i-1个结点，
                   //控制找到 $a_{i-1}$ 结点，指针为p
    while( k!=i-1 && p->next!=L )
    {
        //搜索 $a_{i-1}$ 结点，指针为p，
        //并取得指向 $a_i$ 的指针p->next
        p=p->next; //p移动到下一个结点
        k++;        //结点计数加1
    }
```


//注意p->next==L , k==i-1时 , 插入位置仍合法 ,
//p指向尾结点 , 新结点插到表尾

if(p->next==L && k!=i-1)

return false; //p指尾结点 , 插入位置 i 不对 , 返回false

else

{

//此时 , k=i-1 , p指向 a_{i-1} 结点 , 目标位置的前一个结点

S=new node; //动态申请内存, 创建一个新结点

S->data=x; //装入数据

S->next=p->next;

p->next=S; //新结点链接入表

return true; //正确插入返回 true

}

}

【思考问题】

- ① 算法中的循环控制条件，如果仍用单链表的 $p \neq \text{NULL}$ ，将怎样？
- ② 除了给出的运算，其它基本运算是否也面临同样问题？
- ③ 特别是**插入**和**销毁**运算循环控制和条件判定？
- ④ 改造单链表的6个基本运算使适应单循环链表。

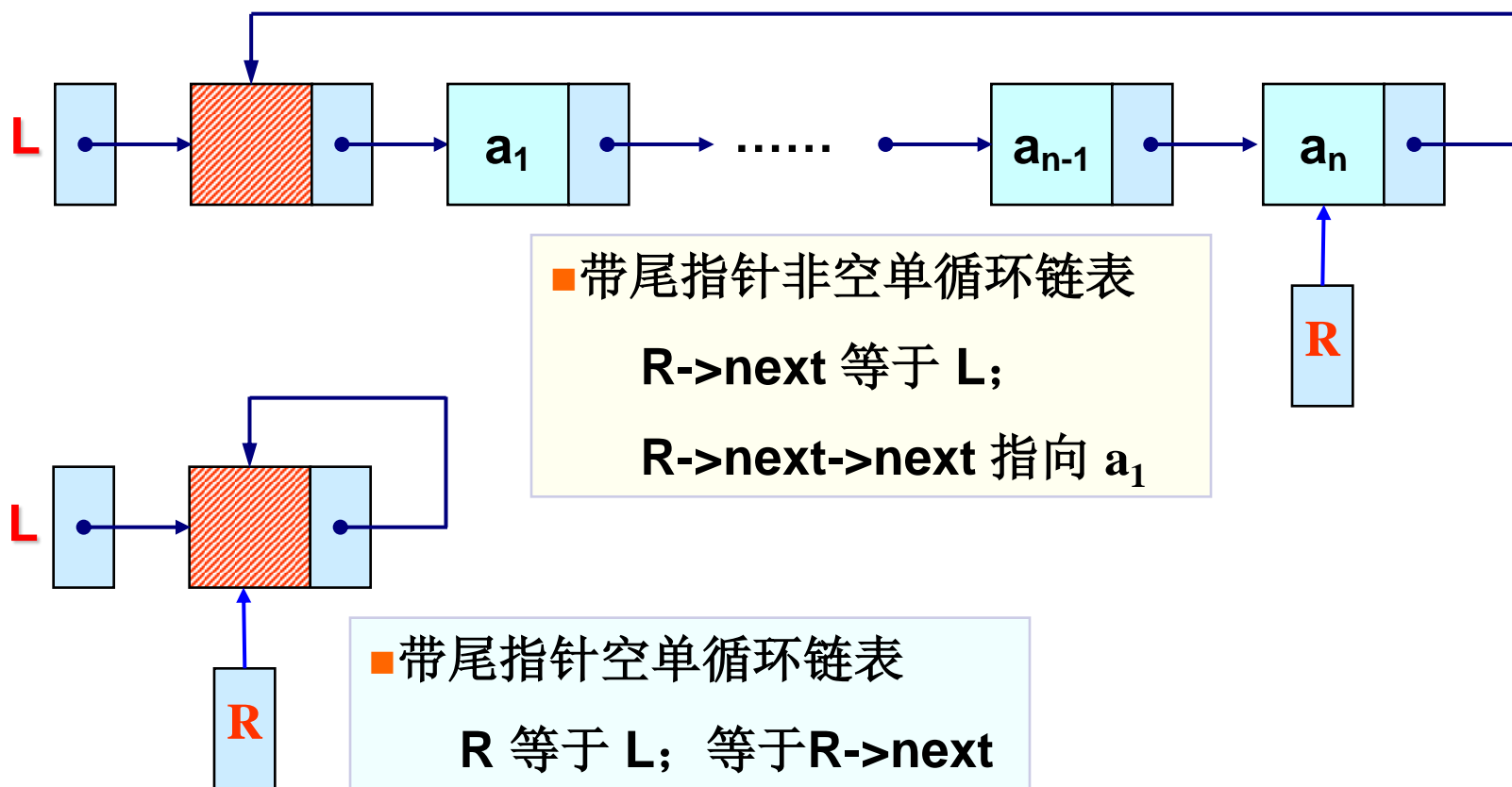
【与单链表的不同点】

☞ **初始化**

☞ 循环控制条件（找 a_{i-1} 结点指针 p ）

2.4.2 带尾指针的单循环链表

- 单循环链表，加尾指针R (Rear)，指向尾结点，则R->next为头指针。可以不要头指针L。



■ 带尾指针单循环链表数据成员的变化

```
class LinkedList
```

```
{
```

```
public:
```

```
    LinkedList();
```

```
    int length();
```

```
    bool getElement(int i, elementType &x);
```

```
    bool locate1(elementType x, node* &p, int &i);
```

```
    bool listInsert(int i, elementType x);
```

```
    bool listDelete(int i);
```

```
private:
```

```
    node* R; //尾指针，取消头指针。此处不同
```

```
};
```

■ 带尾指针单循环链表基本操作的变化

①初始化

```
void initialList(node *&R)
{
    //空表时，R即L，
    //故直接用R申请头结点
    R=new node;
    R->next=R;    //置为空表，形成循环
}
```

②求表长度—注意循环结束条件

```
int listLength(node *R)
```

```
{
```

```
    int len=0;    //保存长度值，初始化为0
```

```
    node* p=R->next->next; //p指向第一个元素结点
```

```
    while(p!=R->next)    //R->next为头指针
```

```
    {
```

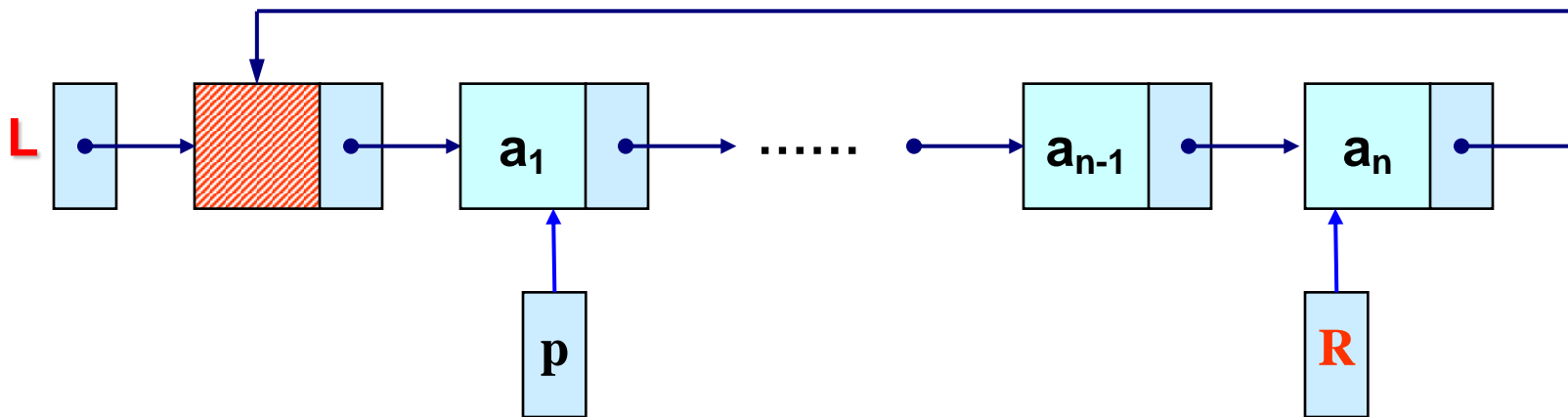
```
        len++;    //p!= R->next，结点存在，结点数加1
```

```
        p=p->next; //p后移一个结点
```

```
    }
```

```
    return len;    //返回长度值
```

```
}
```



③插入操作

☞ 特别注意:

- 1) 如果插入的是第一个元素结点, 或插在尾结点之后 (最后), 移动尾指针指向此结点;
- 2) 插入到表尾的条件判定;

```
bool listInsert(node *&R,int i,elementType x)
{
    node* p=R->next; //p指向头结点
    node* S;
    int k=0;//k计数前i-1个结点, 控制找到 $a_{i-1}$ 结点, 指针为p

    //搜索 $a_{i-1}$ 结点, 指针为p, 并取得指向 $a_i$ 的指针p->next
    while(k!=i-1 && p->next!=R->next)
    {
        p=p->next; //p移动到下一个结点
        k++;        //结点计数加1
    }
}
```

//注意 $p \rightarrow next == R \rightarrow next$, $k == i-1$ 时 , 插入位置仍合法 ,
//p指向尾结点 , 新结点插到表尾

```
if(  $p \rightarrow next == R \rightarrow next$  &&  $k != i-1$  )  
    return false;    //  $p == R$  且  $k != i-1$  , 插入位置  $i$  不对 , 返回 false  
else  
{    //此时 ,  $k = i-1$  , p为 $a_{i-1}$ 结点的指针 , 目标位置的前一个结点  
    S = new node;    //动态申请内存, 创建一个新结点  
    S->data = x;    //装入数据  
    S->next = p->next; //新结点链接入表  
    p->next = S;  
    //特别注意 : 如果插入的是第一个元素结点 , 或插到最后 ,  
    //修改尾指针 , 指向此结点  
    if( $R \rightarrow next == S$ )  
        R = S;  
    return true; //正确插入返回 true  
}
```


④删除操作

☞ 特别注意:

1) 如果删除尾结点, 需要向前移动尾指针, 否则丢失尾指针。

```
bool listDelete(node *&R, int i)
```

```
{
```

```
    node* p=R->next; //p指向头结点
```

```
    node* u;
```

```
    int k=0;
```

```
        //搜索 $a_{i-1}$ 结点, 指针p
```

```
    while(  $k \neq i-1$  &&  $p \rightarrow next \neq R \rightarrow next$  )
```

```
    {
```

```
        p=p->next;
```

```
        k++;
```

```
    }
```

```
if( p->next==R->next )
```

```
    return false; //删除位置 i 超出范围，删除失败，返回false
```

```
else
```

```
{    //此时，k=i-1，p指向 $a_{i-1}$ 结点，p->next指向 $a_i$ 
```

```
    //要删除的目标结点
```

```
    u=p->next;    //u指向待删除结点  $a_i$ 
```

```
    p->next=u->next; //  $a_{i-1}$ 的next指向 $a_{i+1}$ 结点，  
    //或为R->next (  $a_{i-1}$ 为新的尾结点 )
```

```
    //特别注意：如果删除尾结点，需要将尾指针前移，
```

```
    // 否则丢失尾指针
```

```
    if(u==R)
```

```
        R=p;    //p指向结点成为新的尾结点
```

```
    delete u;    //释放待删除结点内存
```

```
    return true; //成功删除，返回 true
```

```
}
```

```
}
```

【思考问题】

- ① 算法中的循环控制条件，如果仍用单链表的 $p \neq \text{NULL}$ ，将怎样？
- ② 除了给出的运算，其它基本运算是否也面临同样问题？
- ③ 特别是插入和删除运算的特殊处理？
- ④ 自行实现带尾指针的单循环链表？

【与单链表的不同点】

- ① 数据成员
- ② 初始化
- ③ 循环控制条件（找 a_{i-1} 结点指针 p ）
- ④ 插入、删除的特殊处理

■ 带尾指针单循环链表优点：

- ☞ 单链表访问任何元素（结点）都需要从表首（头指针）开始，循环链表从任一结点出发都可以访问其它任意结点。可在表中反复搜索。
- ☞ 带尾指针单循环链表既便于求得表尾指针-- R ，也便于求得表头指针-- $R \rightarrow \text{next}$ 。因而方便那些需要同时涉及到表头和表尾的操作。

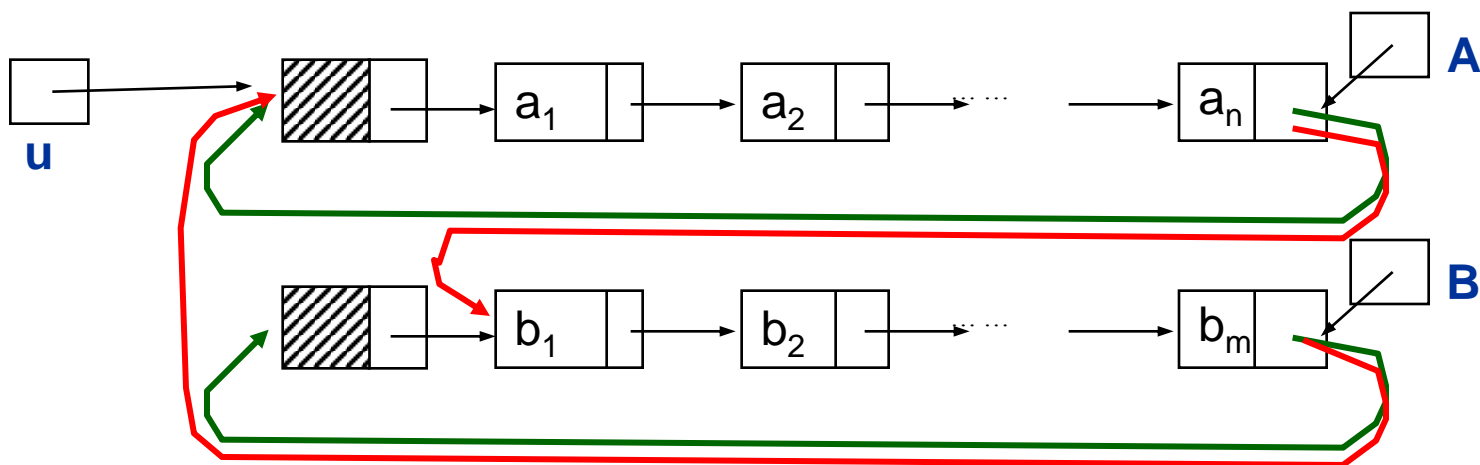
【例2.9】 将A、B两个带尾指针的单循环链表首尾相接，尽量节省时间。

【解题分析】

B的头接到A的尾。相接后以A的头结点为头结点，以B的尾结点为尾结点，仍为带尾指针的单循环链表。操作步骤：

- ① A的尾结点的next指向B的第一个元素结点；B的头结点不需要。A->next=B->next->next;
- ② 释放B的头结点，指针为B->next；
- ③ B的尾指针为新的尾指针，且B->next为新表的头指针，形成循环。

■ 操作过程演示



【关键代码】

`u = A -> next;`

//保留A的头指针

`A -> next = B -> next -> next;`

//A尾接B首

`delete B -> next;`

//删除B的头结点，释放内存

`B -> next = u;`

//形成大的循环链

`A = B;`

//A、B相同，同时指向新的表尾

【算法描述】

☞ 实现代码：exam29scrMergeAB.cpp

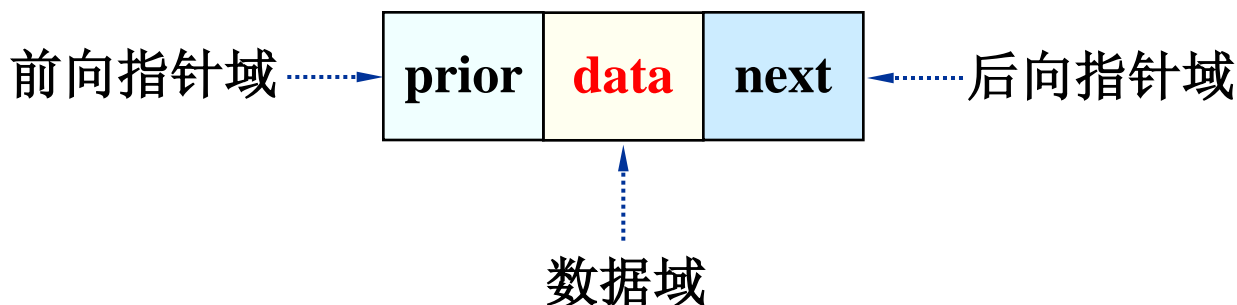
```
void scrMergeAB(LinkedList &A, LinkedList &B)
{
    node* u;
    u=A->next;           //存放A的头指针
    A->next=B->next->next; //B表头链接到A表尾
    delete B->next;      //释放B的头结点
    B->next=u;           //B表尾的next指向A的头结点，
                        //形成大循环
    A=B;                //A、B同时指向新的尾结点，成为尾指针
}
```



2.4.3 双循环链表—双链表 (Double Linked List)

1. 双链表结点结构

- ① 数据域(data)—存放数据元素
- ② 前向指针(prior)—指向直接前驱
- ③ 后向指针(next)—指向直接后继



2. 双链表存储结构c++描述

【双链表结点结构】

```
typedef struct DLNode
{
    elementType    data;           // 数据域
    struct DLNode *prior;          // 前向指针域
    struct DLNode *next;          // 后向指针域
} dNode;
```

【双链表类定义】

```
class dLinkedList
```

```
{
```

```
public:
```

```
    dLinkedList();
```

```
    int length();
```

```
    bool getElement(int i, elementType &x);
```

```
    bool locate(elementType x, dNode* &p, int &i);
```

```
    bool listInsert(int i, elementType x);
```

```
    bool listDelete(int i);
```

```
        //其它函数成员
```

```
private:
```

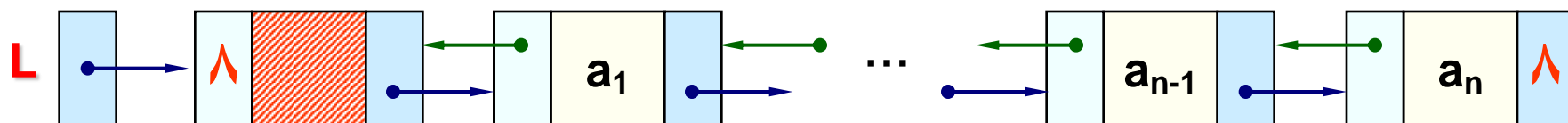
```
    dNode* head; //头指针
```

```
        //其它数据成员
```

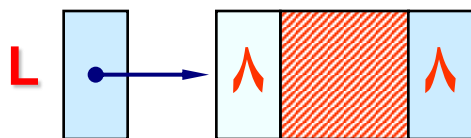
```
};
```

■ 带头结点的双链表

☞ 带头结点的非空双链表

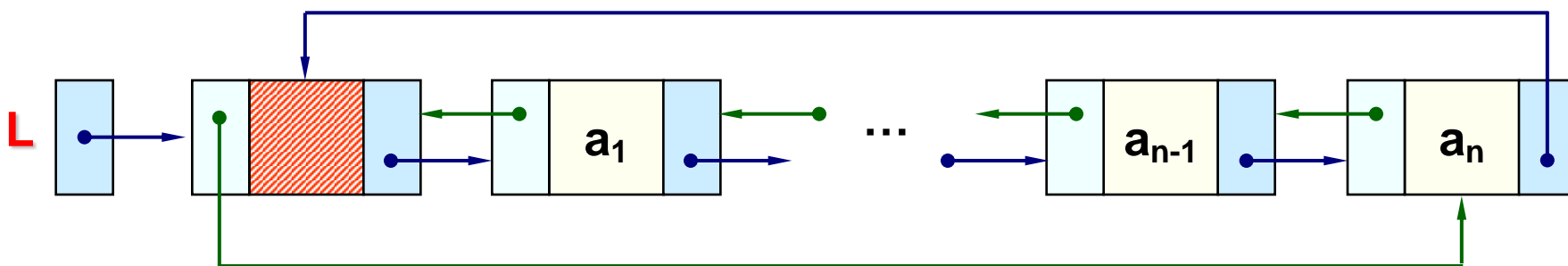


☞ 带头结点的空双链表

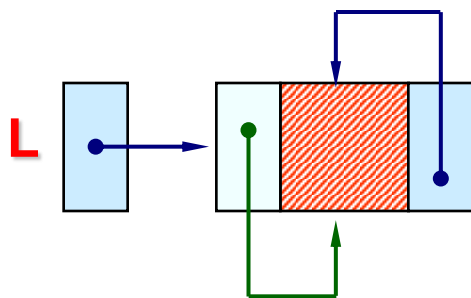


■ 带头结点的双循环链表

☞ 带头结点的非空双循环链表

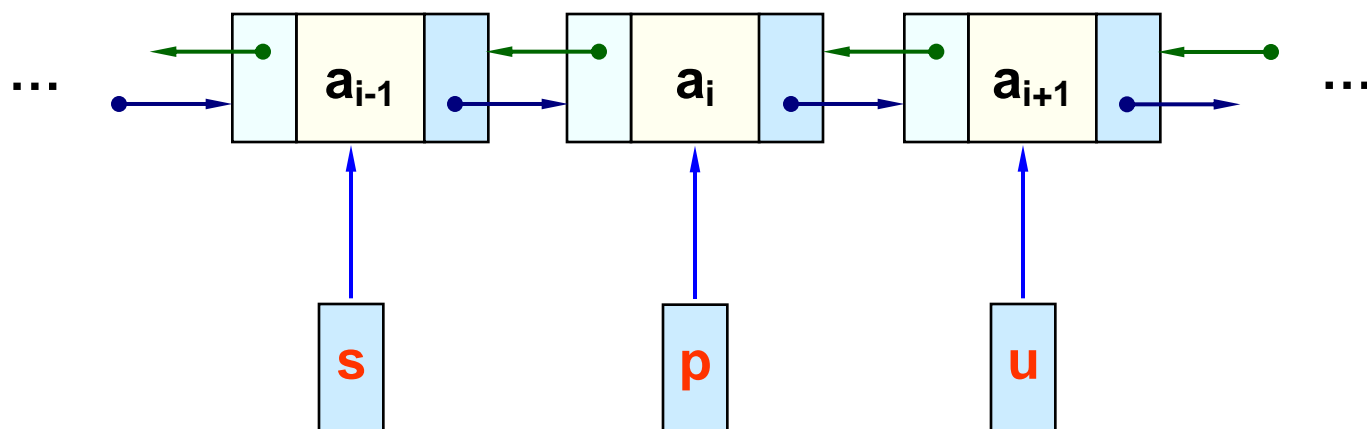


☞ ■ $L \rightarrow \text{prior}$ 指向尾结点。 $L \rightarrow \text{prior} \rightarrow \text{next}$ 指向头结点，即等于 L



■ $L \rightarrow \text{prior} == L \rightarrow \text{next} == L$

■ 双链表结点间的指针关系



- $p \rightarrow \text{next} == u$, 指向 a_{i+1} 结点;
- $p \rightarrow \text{prior} == s$, 指向 a_{i-1} 结点;
- $p \rightarrow \text{next} \rightarrow \text{prior} == p$, 即, $u \rightarrow \text{prior} == p$, 指向 a_i 结点;
- $p \rightarrow \text{prior} \rightarrow \text{next} == p$, 即, $s \rightarrow \text{next} == p$, 指向 a_i 结点;

■ 双链表的优点

- ☞ 访问前驱、后继方便（单链表访问前驱不便）

■ 双链表的缺点

- ☞ 多了一个指针域，占用更多内存

■ 表结构类型

- ☞ 带头结点和不带头结点；
- ☞ 循环和非循环

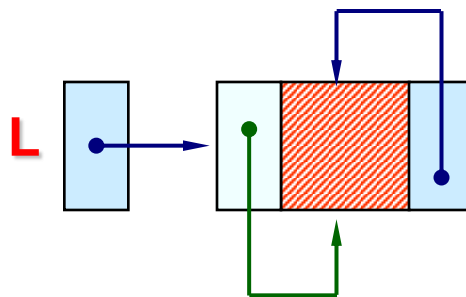
■ 3. 双循环链表的基本操作

☞ (1) 双循环链表的初始化

L=new node;

L->prior=L;

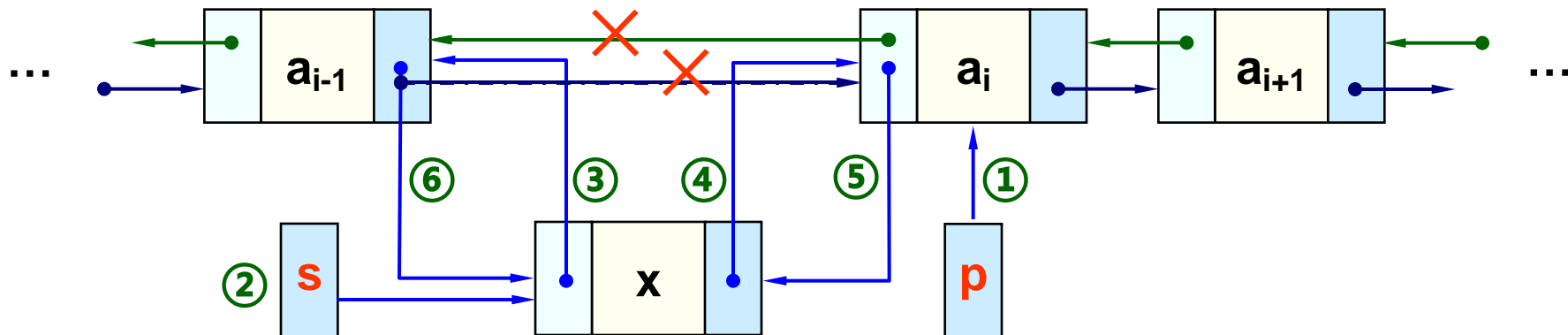
L->next=L;



■ (2)双链表的插入操作

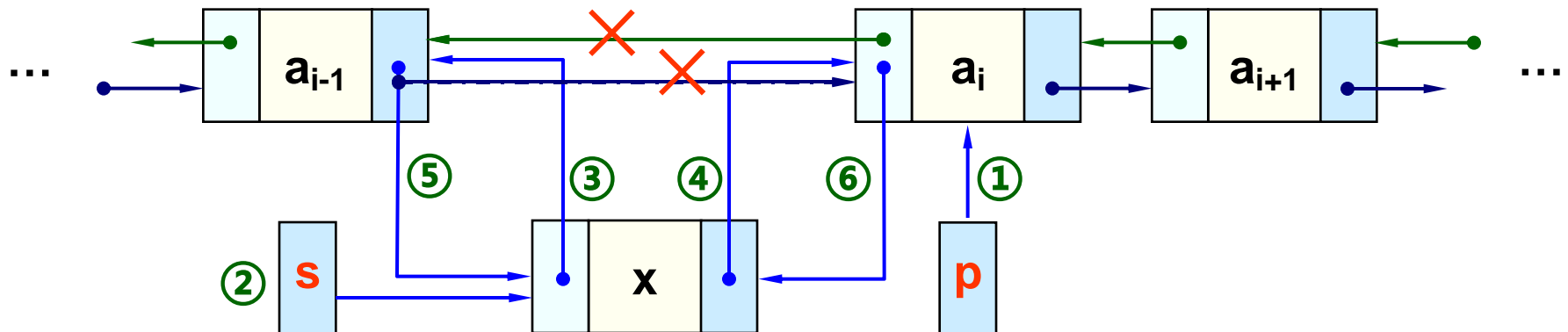
👉 第 i 个元素位置，插入元素值为 x 的新结点。

- ① 搜索插入位置，获取 a_i 结点的指针 p （注意此处与单链表不同）；
- ② 申请新结点，装入数据； $s = \text{new node}; s \rightarrow \text{data} = x$;
- ③ $s \rightarrow \text{prior} = p \rightarrow \text{prior}$;
- ④ $s \rightarrow \text{next} = p$;
- ⑤ $p \rightarrow \text{prior} = s$;
- ⑥ $s \rightarrow \text{prior} \rightarrow \text{next} = s$;



■ 双链表的插入操作的第5步和第6步可改为：

- ① 搜索插入位置，获取 a_i 结点的指针 p ；
- ② 申请新结点，装入数据； $s = \text{new node}; s \rightarrow \text{data} = x$;
- ③ $s \rightarrow \text{prior} = p \rightarrow \text{prior}$;
- ④ $s \rightarrow \text{next} = p$;
- ⑤ $p \rightarrow \text{prior} \rightarrow \text{next} = s$;
- ⑥ $p \rightarrow \text{prior} = s$;



- **双循环链表插入时要找到插入位置 a_i 结点的指针 p** ，这一点与单链表不同，单链表要找 a_{i-1} 结点的指针。

【双循环链表插入算法描述-1】

```
bool listInsert(node *L,int i, elementType x)
```

```
{  
    dNode* p=L->next; //p指向首元素结点  
    dNode* s;  
    int k=1; //从1开始计数，找到 $a_i$ 结点  
    while( k!=i && p!=L )  
    {  
        //搜索 $a_i$ 结点，p=head即又回到头结点  
        p=p->next; //p移到下一个结点  
        k++;  
    }  
}
```

【双循环链表插入算法描述-2】

```
if( p==L && k!=l )  
    return false;
```

//p==L，指向头结点，且k!=i，说明插入位置 i 不对，
返回false

//当 p==L 且 k==i 时，插入位置仍然合法，结点要
插在最后，成为尾结点！！！！

■ 【双循环链表插入算法描述-3】

else

{

//此时, $k=i$, p 为 a_i 结点的指针 (此与单链表不同)
//或者 $p=L, k=i$, 新节点要插入最后, 算法相同

$s = \text{new dNode};$ //动态申请内存, 创建一个新节点
 $s \rightarrow \text{data} = x;$ //装入数据

$s \rightarrow \text{prior} = p \rightarrow \text{prior};$ //s前驱为 a_{i-1}

$s \rightarrow \text{next} = p;$ //s后继为 p

$p \rightarrow \text{prior} = s;$ //p的前驱变为 s

$s \rightarrow \text{prior} \rightarrow \text{next} = s;$ //a_{i-1}的后继为 s

return true; //正确插入, 返回 true

}

■ (3) 双链表的删除操作

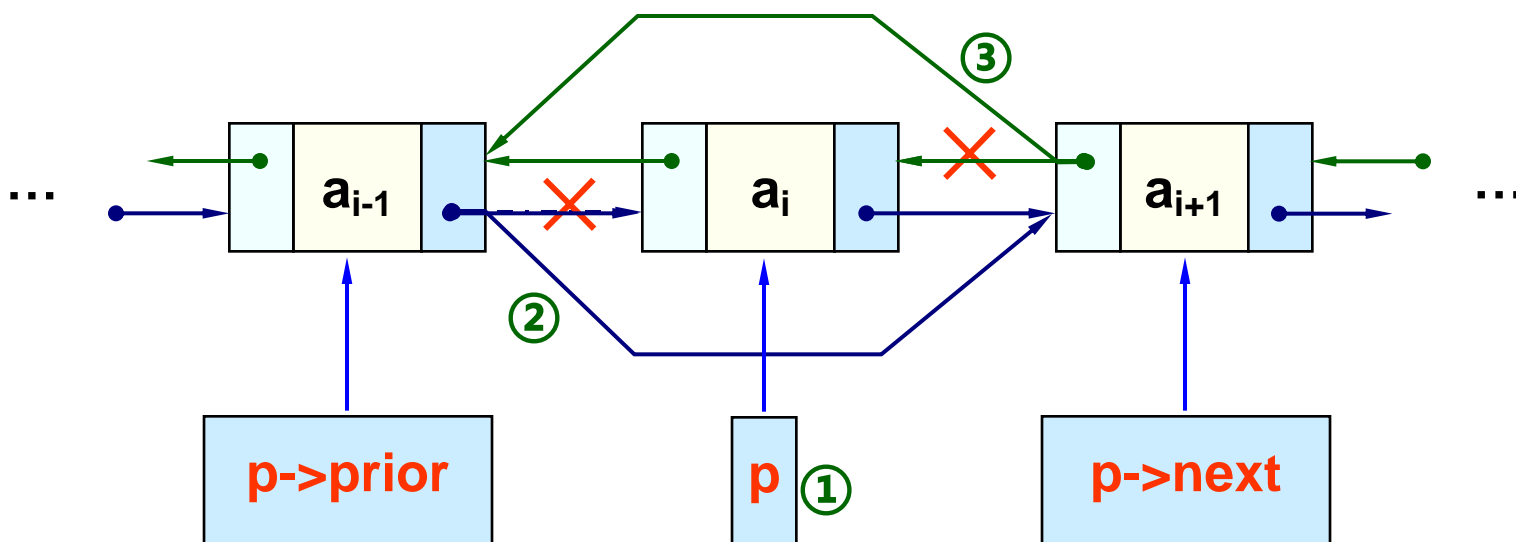
➡ 删除第 i 个元素结点。

➡ ① 搜索 a_i 结点，指针 p ； a_{i-1} 结点指针 $p \rightarrow \text{prior}$ ； a_{i+1} 结点指针 $p \rightarrow \text{next}$ ；

➡ ② $(p \rightarrow \text{prior}) \rightarrow \text{next} = p \rightarrow \text{next}$ ；

➡ ③ $(p \rightarrow \text{next}) \rightarrow \text{prior} = p \rightarrow \text{prior}$ ；

➡ ④ $\text{delete } p; // \text{free}(p)$



【删除操作算法描述-1】

```
bool listDelete(node *L, int i)
```

```
{
```

```
    dNode* p=L->next;    //指向第一个数据结点
```

```
    int k=1;
```

```
    while(k!=i && p!=L)
```

```
    {        //搜索ai节点，p==head说明又回到头结点
```

```
        p=p->next;
```

```
        k++;
```

```
    }
```

```
        //删除位置 i 超出范围，删除失败，返回false
```

```
    if(p==L)
```

```
        return false;
```

【删除操作算法描述-2】

```
else      //此时，p指向 $a_i$ ，执行删除
{
    //p的后继 $a_{i+1}$ 的prior指向p的前驱 $a_{i-1}$ 
    p->next->prior=p->prior;
    //p的前驱 $a_{i-1}$ 的next指向p的后继 $a_{i+1}$ 
    p->prior->next=p->next;
    delete p;      //释放结点
    return true;    //成功删除，返回 true
}
}
```


【思考问题】

- ① 双链表的其他基本运算如何实现？
- ② 带头结点的双循环链表，当前指针p指向 a_i 结点，前向搜索时，怎样判定搜索到了首元素结点？后向搜索时怎样判定搜索到了尾结点？

■ 循环链表使用注意：

- 👉 搜索到表尾的判定条件
- 👉 在表尾插入、删除结点的处理

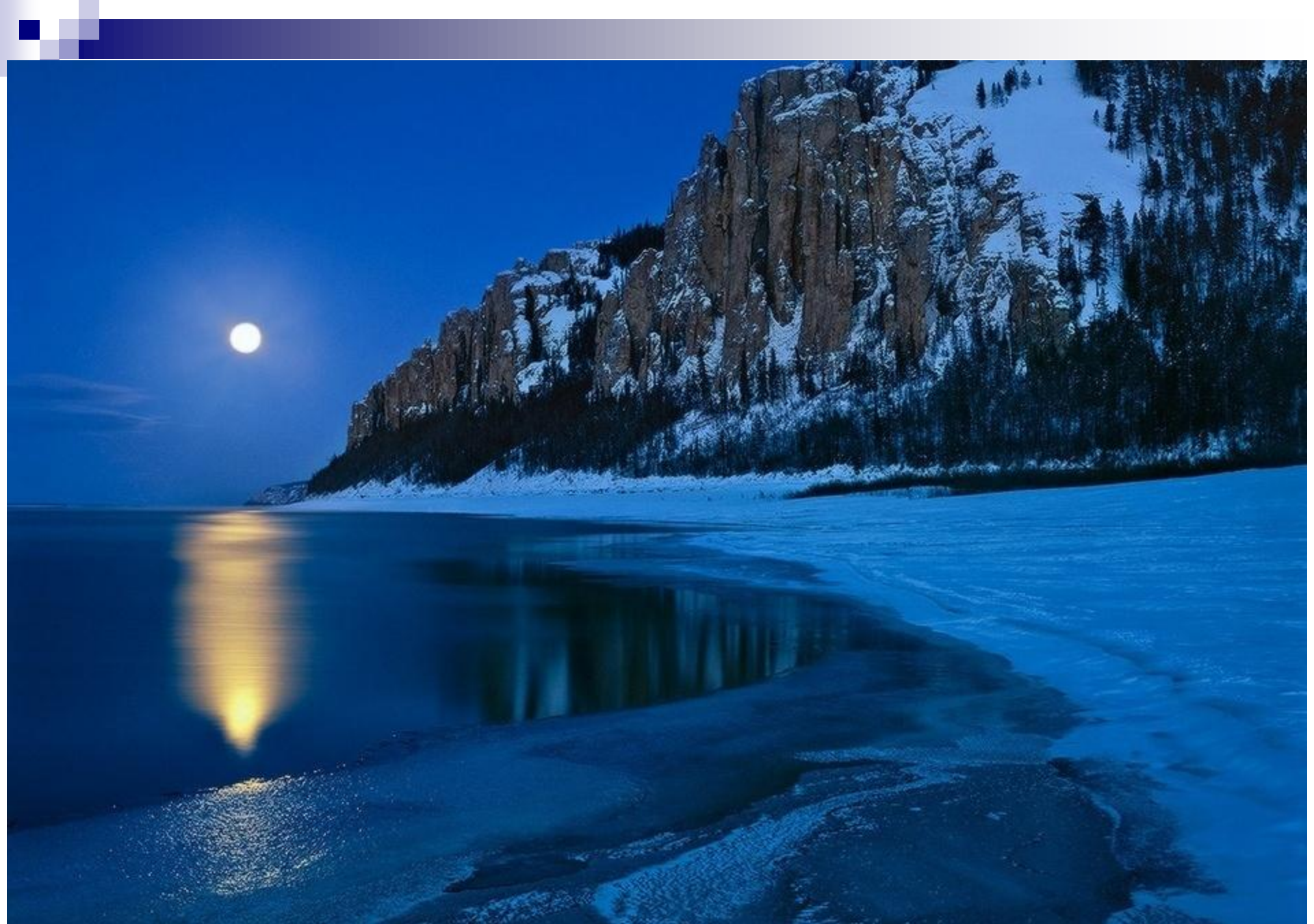
【作业布置】

(P51-52)

👉 2.12

👉 2.23

👉 2.24



2.4.4 静态链表

■ 静态链表 (Static Linked List)

- ☞ 利用数组实现链表的功能，免去了顺序表插入和删除操作时耗时的移动元素操作。
- ☞ 是一个空间换时间的实例。
- ☞ 在像JAVA、C#等没有指针的程序设计语言中，要实现链表就必须采用这种方式。
- ☞ 本节以静态单链表为例来解释静态链表的实现：

- 申请一个较大的一维数组SL[n]。
- 数组的元素由2个成员构成：
 - ☞ **data域**—数据域，保存链表的数据元素；
 - ☞ **next域**—指针域，是指向链表中下一个元素的指针，事实上这里的next是存储下一个元素的数组下标，是一个整数。
- 通过next，形成链式结构。当
- 链表结束，最后一个结点的next域赋给一个特殊的值，比如：-1，表示链表结束。

- 与动态链表不同的是，插入新结点时，首先要判断表空间（数组）是否已满。
- 如果表空间未满，则要获取一个空的单元，即取得空单元的指针（数组下标），那么如何取得空单元的指针呢？
- 一般我们把所有空单元也组成一个链表，根据空单元链表的头指针很容易就能获取一个空单元。
- 删除结点时，则需要把删除元素的单元回收的空单元链表中。

■ 静态链表中事实上存储有2条链表：

➡ 数据链表

➡ 空单元链表

■ 静态链表也可以带有头结点

➡ 一般用数组SL[]头尾2个单元分别作为两条链表的头结点。

■ 比如，

➡ SL[0]为数据链表头结点，头指针为0；

➡ SL[n-1]为空单元链表头结点，头指针为n-1

,

➡ 反之亦可。

【示例】 下图所示为一个静态单链表，

☞ **数据链表**头结点为SL[0]，头指针L=0，保存数据值a、b、c、d、e。

☞ **空单元链表**头结点为SL[11]，头指针S=11

。

数组下标	0	1	2	3	4	5	6	7	8	9	10	11
data			e	b		a		d		c		
next	5	4	-1	9	6	3	-1	2	10	7	1	8

数组下标 0 1 2 3 4 5 6 7 8 9 10 11

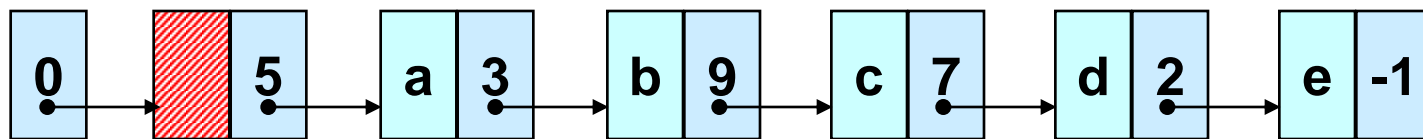
data

next

		e	b		a		d		c		
5	4	-1	9	6	3	-1	2	10	7	1	8

数据链表

L

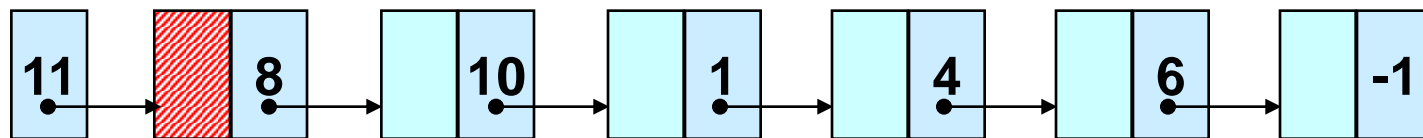


头指针 头结点 首元素结点

尾结点

空单元链表

S



头指针 头结点 首元素结点

尾结点

■ 静态单链表的存储描述

`#define MAXLEN 6` //定义线性表的最大容量

`typedef int elementType;` //定义数据元素的数据类型

`typedef struct sll`

`{`

`elementType data;`

`int next;` //结点指针（数组下标）

`}sNode, SLL[MAXLEN];` //SLL[]为单链表数组类型

■ 静态单链表初始化

//以L[0]为链表头结点

//以L[MAXLEN-1]为空单元链表头结点

//链表结束以-1表示

```
void listInitial(SLL L)
```

```
{
```

```
    int i;
```

```
    L[0].next=-1; //初始化数据链表
```

```
                //初始化空单元链表
```

```
    for(i=MAXLEN-1;i>1;i--)
```

```
        L[i].next=i-1;
```

```
    L[i].next=-1; //空单元链表结束
```

```
}
```

【课外练习】

- ☞ 完成静态单链表的其它运算
- ☞ 完成静态单循环链表的基本运算
- ☞ 完成静态双循环链表的基本运算

【线性表小结】

- 线性表：有限个元素，逻辑有序，一对一关系。
- 顺序表：
 - ☞ 逻辑上相邻的元素物理上也相邻；
 - ☞ 可直接定位，节省搜索时间（随机访问）；
 - ☞ 在插入、删除时，需移动元素，浪费时间；
 - ☞ 存储元素时，无需额外的存储空间；
 - ☞ 【适用情况】表长度变化不大，且能事先估计最大长度；无需经常插入、删除操作，经常读取元素。

■ 链表：

- ☞ 逻辑上相邻的元素物理上不一定相邻；
- ☞ 插入、删除时，不需移动元素；
- ☞ 链式存储结构只能按顺序存、取数据元素，不能随机访问元素。
- ☞ 由于附加指针域，比顺序存储结构多占用存储空间，有效空间利用率降低；但可以充分利用存储器中的碎片空间；
- ☞ 【适用情况】 长度变化幅度大；频繁插入、删除元素；元素很大的线性表。

内容回顾

- **线性表是整个数据结构课程的重要基础。**
- **线性表的相关概念：定义、基本运算**
- **顺序存储结构及其描述，顺序表运算的实现**
- **链表存储结构及其描述，链表中运算的实现**
- **其他形式的链表结构**
- **静态链表的相关概念：定义、存储结构、运算**

研考大纲—线性表部分

■ 一、线性表

☞ (一) 线性表的定义和基本操作

☞ (二) 线性表的实现

✦ 1. 顺序存储

✦ 2. 链式存储

✦ 3. 线性表的应用

42 . (2009) (15分) 已知一个带有表头结点的单链表，结点结构如下图。假设该链表只给出了头指针list。在不改变链表的前提下，请设计一个尽可能高效的算法，查找链表中倒数第k个位置上的结点（k为正整数）。若查找成功，算法输出该结点的data值，并返回1；否则，只返回0。要求：

(1) 描述算法的基本设计思想

(2) 描述算法的详细实现步骤

(3) 根据设计思想和实现步骤，采用程序设计语言描述算法（使用C 或C++或JAVA 语言实现），关键之处请给出简要注释。



42. (2010) (13 分) 设将 $n(n,1)$ 个整数存放到一维数组 R 中 , 试设计一个在时间和空间两方面尽可能有效的算法 , 将 R 中保有的序列循环左移 P ($0 < P < n$) 个位置 , 即将 R 中的数据由 ($X_0 X_1 \dots X_{n-1}$) 变换为 ($X_p X_{p+1} \dots X_{n-1} X_0 X_1 \dots X_{p-1}$) 要求 :

(1) 给出算法的基本设计思想。

(2) 根据设计思想 , 采用C或C++或JAVA语言表述算法 , 关键之处给出注释。

(3) 说明你所设计算法的时间复杂度和空间复杂度。

42 . (2011) (15 分) 一个长度为 L ($L \geq 1$) 的升序序列 S , 处在第 $\lceil L/2 \rceil$ 位置的数称为 S 的中位数。例如, 若序列 $S_1=(11, 13, 15, 17, 19)$, 则 S_1 的中位数是15。两个序列的中位数是含它们所有元素的升序序列的中位数。例如, 若 $S_2=(2, 4, 6, 8, 20)$, 则 S_1 和 S_2 的中位数是11。

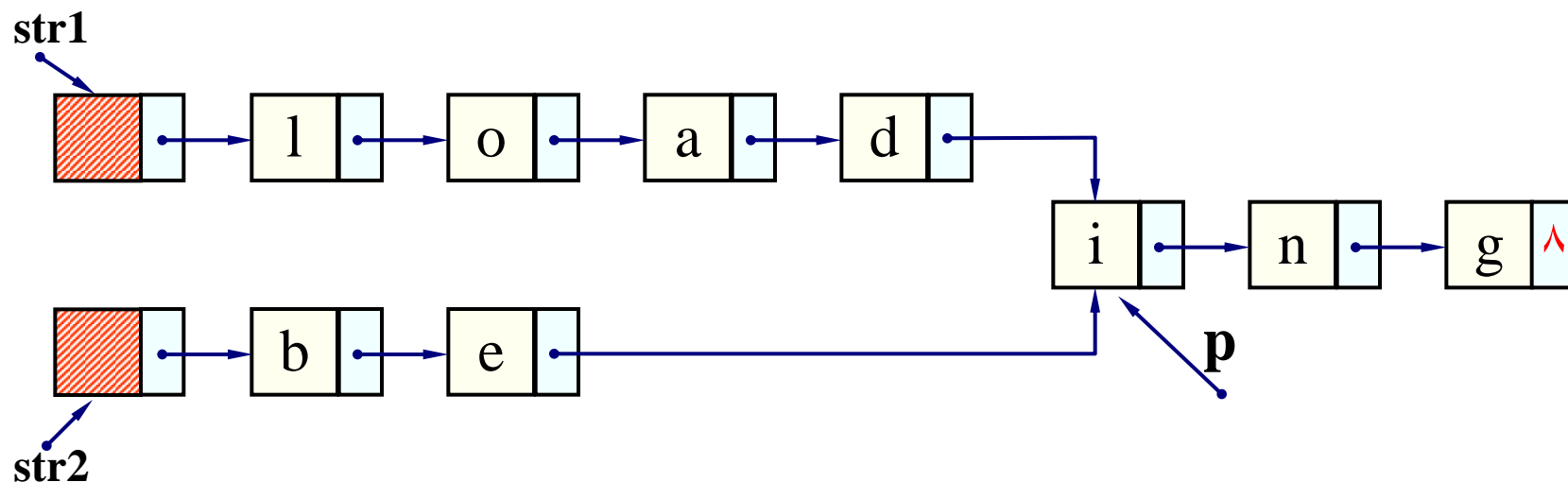
现有两个等长升序序列 A 和 B , 试设计一个在时间和空间两方面都尽可能高效的算法, 找出两个序列 A 和 B 的中位数。要求:

- (1) 给出算法的基本设计思想。
- (2) 根据设计思想, 采用C 或C++或JAVA 语言描述算法, 关键之处给出注释。
- (3) 说明你所设计算法的时间复杂度和空间复杂度。

41. (2012) (10分) 设有6个有序表A、B、C、D、E、F，分别有10、35、40、50、60和200个数据元素，各表中元素按升序排序。要求通过5次两两合并，将6个表最终合并成一个升序表，并在最坏情况下比较的总次数达到最小。请回答下列问题：


- (1) 请写出合并方案，并求出最坏情况下比较的总次数。
- (2) 根据你的合并过程，描述 N ($N \geq 2$) 个不等长升序表的合并策略，并说明理由。

42. (2012) (13分) 假定采用带头结点的单链表存储字符串，如果字符串有相同的后缀，则可共享相同的后缀空间，例如，“loading”和“being”，如下图所示：



设str1和str2分别指向两个单词所在单链表的头结点。结点结构如图：





请设计一个时间上尽可能高效的算法，找出有str1和str2所指向两个链表共同后缀的起始位置（如图中字符i所在结点的位置p）。要求：

- （1）给出算法的基本设计思想。
- （2）根据设计思想，采用C 或C++或JAVA 语言描述算法，关键之处给出注释。
- （3）说明你所设计算法的时间复杂度。

Thank you !

