



## 第二章 线性表

本章内容：

**2.1 线性表的概念**

**2.2 顺序表**

**2.3 单链表**

**2.4 单链表的变形：循环链表和双向链表**

**2.5 单链表的应用：多项式**



# 数据结构关注的三个方面

## 逻辑结构

数据元素之  
间的逻辑关  
系

## 存储结构

数据逻辑结  
构的物理存  
储映象

## 运算

数据结构上  
的运算，及  
实现方法  
(查找、插  
入、删除、  
更新等)



# 线性表的逻辑结构

## ● 线性表的定义

由 $n(n \geq 0)$ 个数据元素(结点)  $a_1, a_2, \dots, a_n$ 组成的**有限序列**。其中数据元素个数 $n$ 定义为表的长度。当 $n=0$ 时称为空表，非空线性表( $n>0$ )记作：

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

- $a_i$ 是表项， $n$ 是表长度。
- 第一个表项是表头，最后一个表尾

**例1** 26个英文字母组成的字母表  
(A, B, C、...、Z)

**例2** 某校从2010年到2016年各种型号的计算机拥有量变化情况。  
(2000,3000,3800,4000,4500,5000,5600)



# 线性表的逻辑结构

注：一个**数据元素**可以由若干**数据项（Item）**组成，在这种情况下，通常把数据元素称为**记录（Record）**，含有大量记录的线性表又称为**文件（File）**。

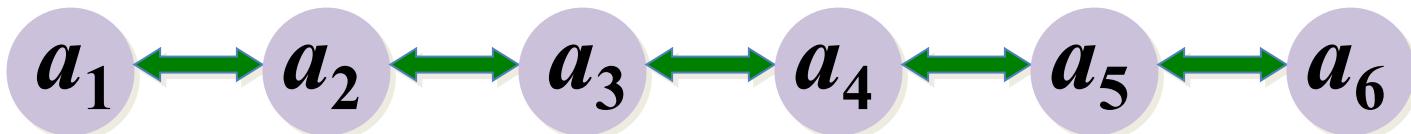
**例3**某高校学生的基本信息表，每个学生的基本信息为一个记录，该记录由学号、姓名、性别、专业以及住址5个数据项组成。

学号	姓名	性别	专业	住址
04180101	侯亮平	男	计算机科学与技术	北京
04180102	高小琴	女	计算机科学与技术	深圳
04180103	陆亦可	女	计算机科学与技术	珠海
04180104	陈海	男	计算机科学与技术	上海
04180105	李达康	男	计算机科学与技术	杭州
04180106	高育良	男	计算机科学与技术	南京
04180107	赵东来	男	计算机科学与技术	武汉
04180108	陈岩石	男	计算机科学与技术	重庆
04180109	沙瑞金	男	计算机科学与技术	珠海



# 线性表的逻辑特征

- (1) 对非空的线性表，有且仅有一个开始结点 $a_1$ ，它没有直接前趋，而仅有一个直接后继 $a_2$ ；
- (2) 有且仅有一个终端结点 $a_n$ ，它没有直接后继，而仅有一个直接前趋 $a_{n-1}$ ；
- (3) 其余的内部结点 $a_i$  ( $2 \leq i \leq n-1$ ) 都有且仅有一个直接前趋 $a_{i-1}$  和一个直接后继 $a_{i+1}$ 。



注：有序表和无序表



# 线性表的存储方式

- 顺序存储方式 —— 顺序表
- 链表存储方式 —— 链表



## 第二章 线性表

2.1 线性表的概念

2.2 顺序表

2.3 单链表

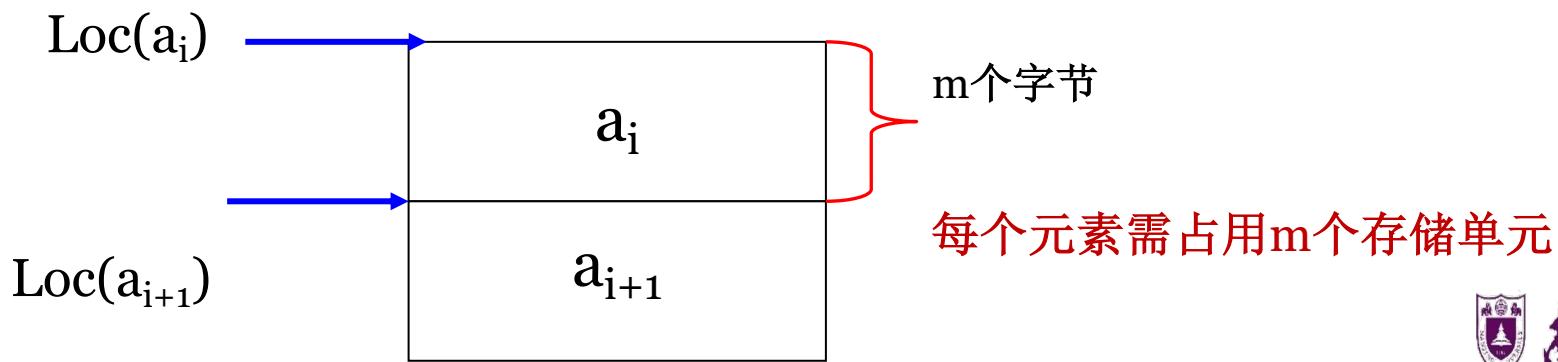
2.4 单链表的变形：循环链表和双向链表

2.5 单链表的应用：多项式



## 2.2 顺序表

- **顺序表：**把线性表的结点按逻辑顺序依次存放在一组**地址连续的存储单元**里。
- 各表项的逻辑顺序与物理顺序一致
- 线性表中第*i+1*个数据元素的存储位置  
 $\text{Loc}(a_{i+1})$ 和第*i*个数据元素的存储位置 $\text{Loc}(a_i)$ 之间满足下列关系：  $\text{Loc}(a_{i+1}) = \text{Loc}(a_i) + m$

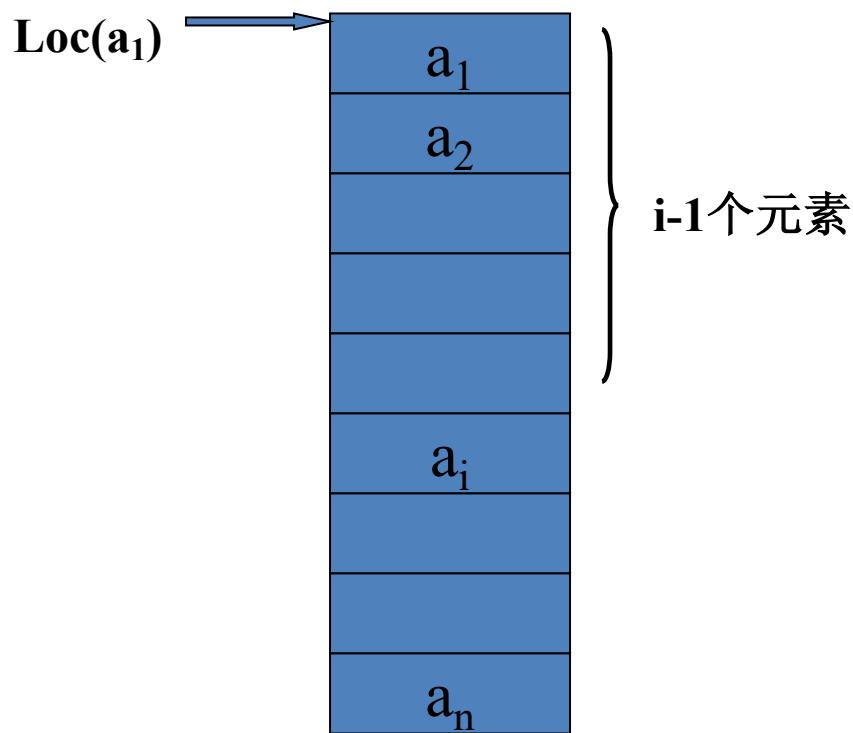




# 顺序表的特点

- 各表项的逻辑顺序与物理顺序一致
- 对各个表项可以顺序访问，也可以随机访问

线性表的第*i*个数据元素 $a_i$ 的存储位置为：



$$\begin{aligned}\text{Loc}(a_i) &= (i-1)*m + \text{Loc}(a_1) \\ &= \text{Loc}(a_1) - m + i*m\end{aligned}$$

由于 $\text{Loc}(a_1)$ 和 $m$ 都是已知的  
所以： $V_0 = \text{Loc}(a_1) - m$

$$\text{Loc}(a_i) = V_0 + i*m$$



# 顺序表的静态存储和动态存储

```
#define maxSize 100
typedef int T;
typedef struct {
    T data[maxSize];
    int n;
} SeqList;
```

//顺序表的静态存储表示

```
typedef int T;
typedef struct {
    T *data;
    int maxSize, n;
} SeqList;
```

//顺序表的动态存储表示



# 顺序表(SeqList)类的定义

```
#include <iostream.h> //定义在 "seqList.h" 中
#include <stdlib.h>
#include "LinearList.h"
const int defaultSize = 100;
template <class T, class E>
class SeqList: public LinearList<T, E> {
protected:
    E *data; //存放数组
    int maxSize; //最大可容纳表项的项数
    int n; //当前已存表项数
    void reSize(int newSize); //改变数组空间大小
```

数据成员



# 顺序表(SeqList)类的定义（续）

**public:**

```
SeqList(int sz = defaultSize);           //构造函数
SeqList(SeqList<T,E>& L);              //复制构造函数
~SeqList() {delete[ ] data;}            //析构函数
int Size() const {return maxSize;}      //求表最大容量
int Length() const {return n;}          //计算表长度
int Search(T x) const;
    //搜索x在表中位置，函数返回表项序号
int Locate(int i) const;
    //定位第 i 个表项，函数返回表项序号
bool Insert(int i, E x);               //插入
bool Remove(int i, E& x);              //删除
};
```



# 顺序表的构造函数

```
#include <stdlib.h>      //操作“exit”存放在此
#include “seqList.h”      //操作实现放在“seqList.cpp”
```

```
template <class T, class E>
SeqList<T, E>::SeqList(int sz) {
    if (sz > 0) {
        maxSize = sz; n = 0;                      //初始化
        data = new E[maxSize];                     //创建表存储数组
        if (data == NULL)                          //动态分配失败
            { cerr << "存储分配错误！" << endl;
              exit(1); }
    }
};
```



# 复制构造函数

```
template <class T, class E>
SeqList<T, E>::SeqList ( SeqList<T, E>& L ) {
    maxSize = L.Size();  n = L.Length();
    data = new E[maxSize]; //创建存储数组
    if (data == NULL)      //动态分配失败
        {cerr << "存储分配错误！" << endl;
         exit(1);}
    for (int i = 1; i <= n; i++) //传送各个表项
        data[i-1] = L.getData(i);
};
```



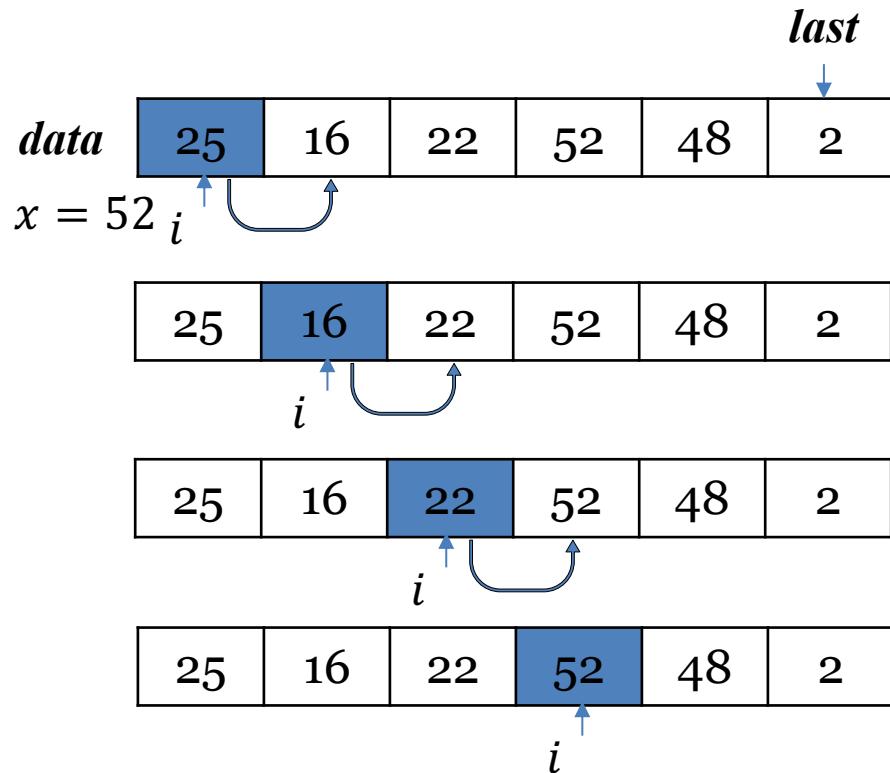
# 顺序表的搜索算法

在表中顺序搜索与给定值  $x$  匹配的表项，找到则函数  
返回该表项是第几个元素，否则函数返回0

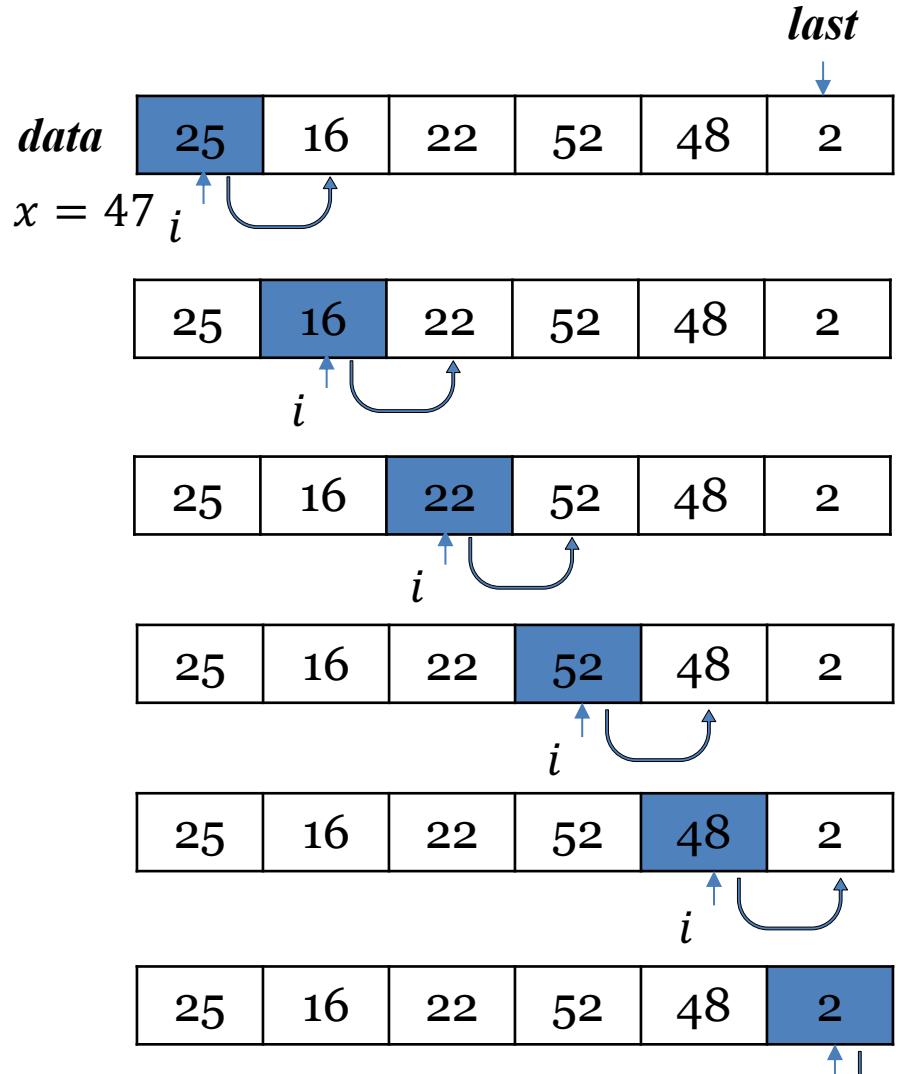
```
template <class T, class E>
int SeqList<T, E>::search(T& x) const {
    for (int i = 1; i <= n; i++)          //顺序搜索
        if ( data[i-1] == x ) return i;
                                            //表项序号和表项位置差1
    return 0;                            //搜索失败
};
```



# 顺序表的查找



$x = 52$  (成功)



$x = 47$  (失败)



# 顺序查找数据的时间代价 (比较次数分析)

ACN(Average Comparing Number)

- 搜索成功: 位置*i*的查找概率*p<sub>i</sub>*, 找到该表项时的数据比较次数*c<sub>i</sub>*

平均比较次数  $ACN = \sum_{i=0}^{n-1} p_i \times c_i$

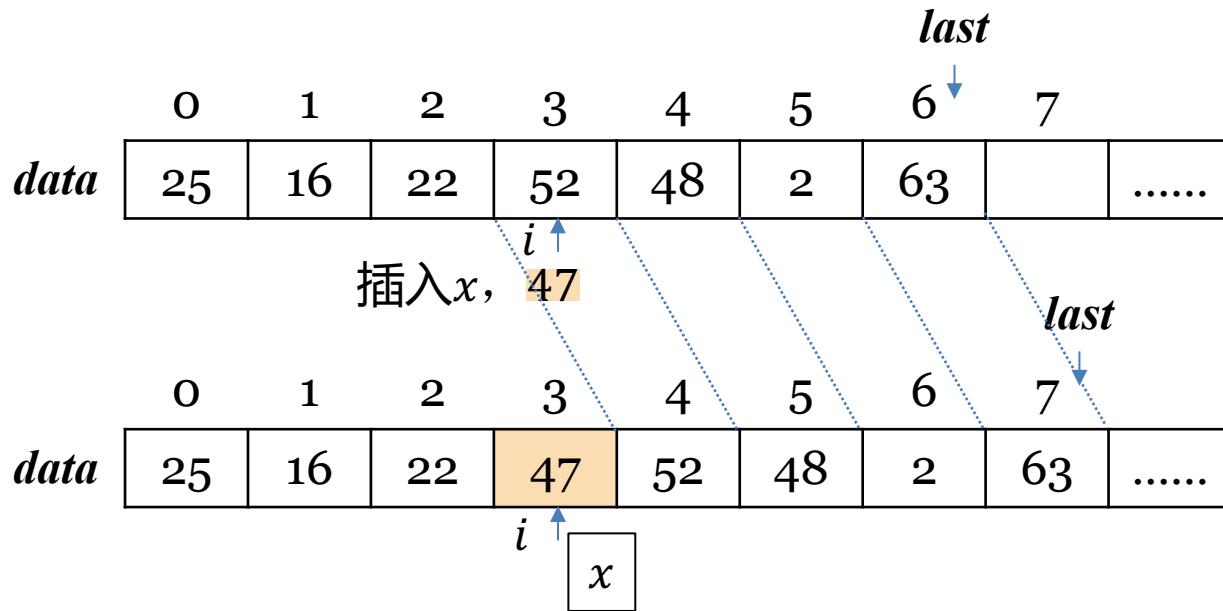
若搜索概率*p<sub>i</sub>*相等, 则

$$\begin{aligned} ACN &= \frac{1}{n} \sum_{i=0}^{n-1} (i+1) = \frac{1}{n} (1+2+\dots+n) = \\ &= \frac{1}{n} * \frac{(1+n)*n}{2} = \frac{1+n}{2} \end{aligned}$$

- 搜索不成功: 数据比较*n*次



# 顺序表的插入





# 表项的插入算法

//将新元素x插入到表中第i ( $1 \leq i \leq n+1$ ) 个表项位  
//置。函数返回插入成功的信息

```
template <class T, class E>
bool SeqList<T, E>::Insert (int i, E x) {
    if (n == maxSize) return false;      //表满
        if (i < 1 || i > n+1) return false; //参数i不合理
        for (int j = n; j >= i; j--)          //依次后移
            data[j] = data[j-1];
        data[i-1] = x;                      //插入第i表项在data[i-1]处
        n++;   return true;                //插入成功
};
```



# 顺序表插入的时间代价（移动次数）

在表中第  $i$  个位置插入，从  $\text{data}[i-1]$  到  $\text{data}[n-1]$  成块后移，移动  $n-1-(i-1)+1 = n-i+1$  项

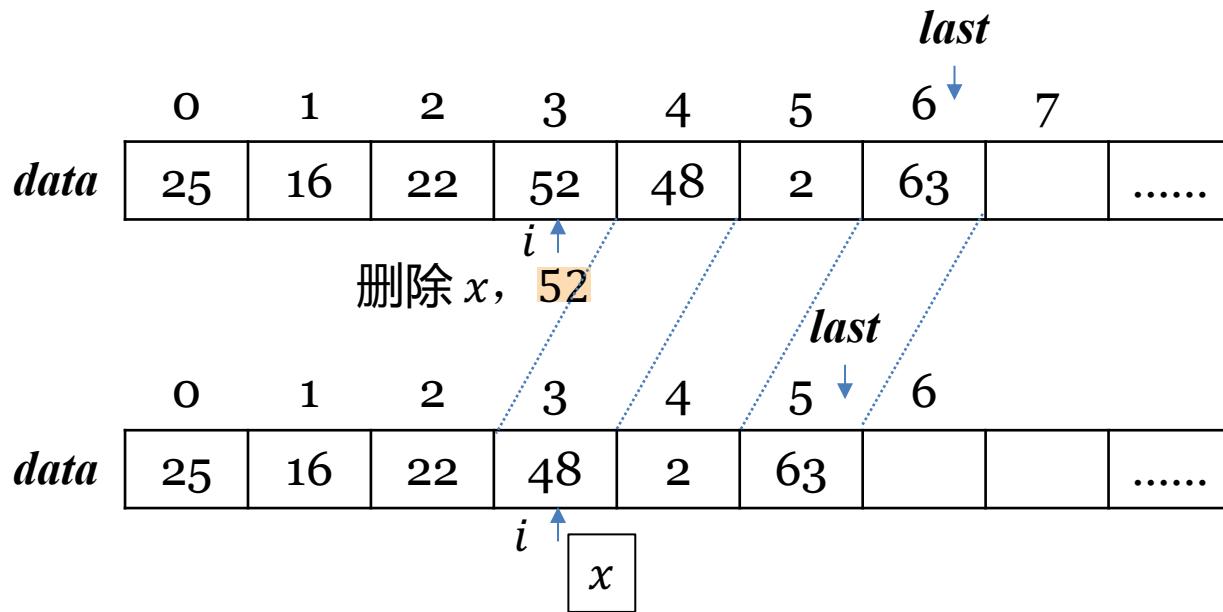
平均数据移动次数AMN(Average Moving Number)在各表项插入概率相等时为

$$\begin{aligned} \text{AMN} &= \frac{1}{n+1} \sum_{i=1}^{n+1} (n-i+1) = \frac{1}{n+1} (n + \dots + 1 + 0) \\ &= \frac{1}{(n+1)} \frac{n(n+1)}{2} = \frac{n}{2} \end{aligned}$$

在插入时有  $n+1$  个插入位置，平均移动  $n/2$  项



# 顺序表的表项删除





# 表项的删除算法

//从表中删除第 i ( $1 \leq i \leq n$ ) 个表项，通过引用型参数 x 返回被删元素。函数返回删除成功信息

```
template <class T, class E>
bool SeqList<T, E>::Remove (int i, E& x) {
    if (n == 0) return false;           //表空
    if (i < 1 || i > n) return false;   //参数i不合理
    x = data[i-1];
    for (int j = i; j <= n-1; j++)    //依次前移，填补
        data[j-1] = data[j];
    n--;
    return true;
};
```



## 顺序表的表项删除的时间代价(移动次数)

删除第  $i$  个表项，需将第  $i+1$  项到第  $n$  项全部前移，需前移的项数为  $n-(i+1)+1 = n-i$

平均数据移动次数AMN (Average Moving Number) 在n个表项删除概率相等时为

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

在删除时有n个删除位置，平均移动 $(n-1)/2$ 项



# 用顺序表实现集合的“并”运算

```
template <class Type>
void Union ( SeqList<Type> & LA,
              SeqList<Type> & LB ) {
    int n = LA.Length ();
    int m = LB.Length ();
    for ( int i = 1; i <= m; i++ ) {
        Type x = LB.getData(i);
                                //在LB中取一元素
        int k = LA.Search (x);   //在LA中搜索它
        if ( k == 0 )           //若未找到插入它
        { LA.Insert (n, x); n++; }
    }
}
```



# 用顺序表实现集合的“交”运算

```
template <class Type>
void Intersection ( SeqList<Type> & LA,
                     SeqList<Type> & LB ) {
    int n = LA.Length ();
    int m = LB.Length (); int i = 1;
    while ( i <= n ) {
        Type x = LA.getData (i); //在LA中取一元素
        int k = LB.Search (x); //在LB中搜索它
        if ( k == 0 ) { LA.Remove (i,x); n--; }
                           //未找到在LA中删除它
        i++;
    }
}
```



**思考：**以上的集合的“交”和“并”运算  
的时间复杂性是多少？



# 顺序表的特点

- 特点：逻辑关系上相邻的两个数据元素在物理位置上也相邻。
- 优点：**节省存储空间。**由于结点之间的相邻逻辑关系可以用物理位置上的相邻关系表示，因此**不需增加额外的存储空间**来表示此关系。**存取速度快。**
- 缺点：插入、删除等操作时需要移动大量数据



## 第二章 线性表

2.1 线性表的概念

2.2 顺序表

2.3 单链表

2.4 单链表的变形：循环链表和双向链表

2.5 单链表的应用：多项式



# 单链表

## 连续存储方式（顺序表）

- 特点：存储利用率高，存取速度快
- 缺点：插入、删除等操作时需要移动大量数据

## 链式存储方式（链表）

- 特点：适应表的动态增长和删除
- 缺点：需要额外的指针存储空间



# 单链表

## ● 单链表的特点

- 每个元素(表项)由结点(*Node*)构成。



- 线性结构

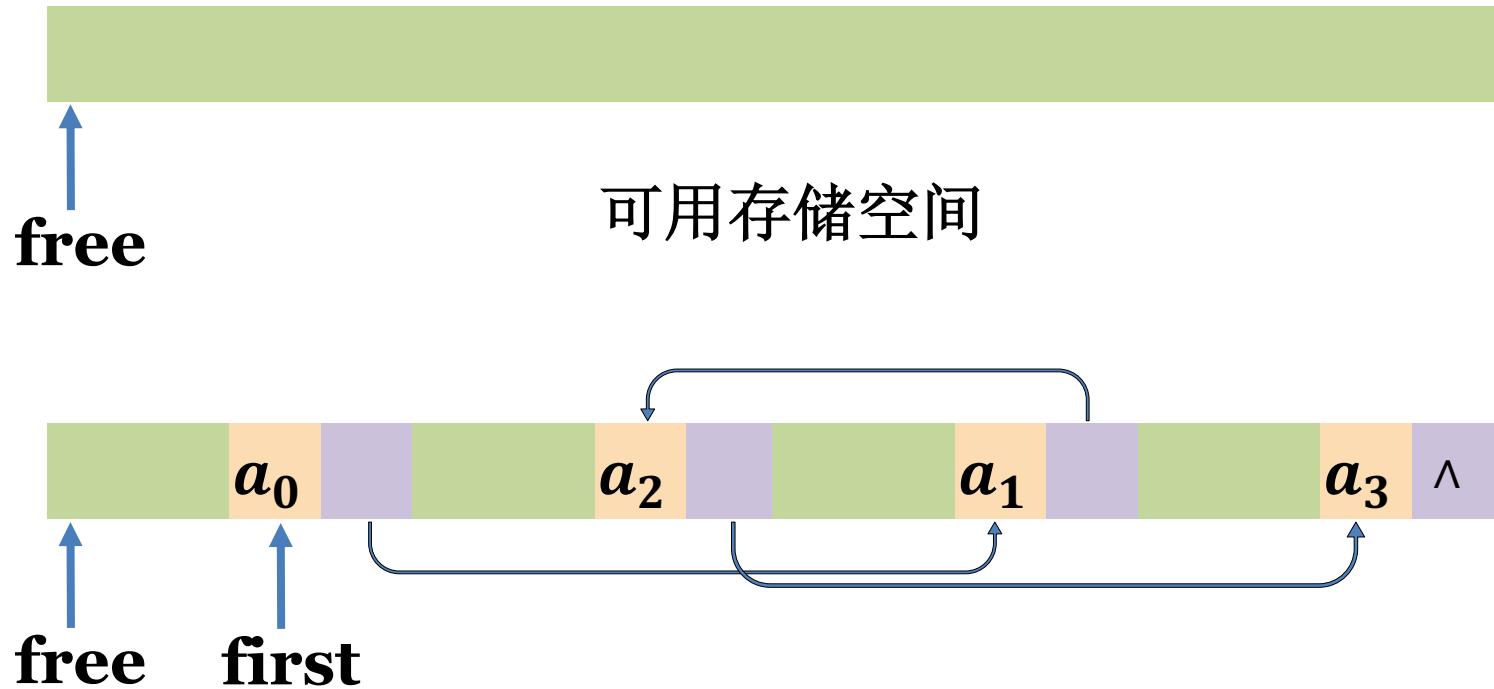


- 结点可以不连续存储
- 表长度可方便地扩充

last



# 单链表的存储映像



经过一段运行后的单链表结构



# 单链表的类定义

- 多个类表达一个概念(单链表)。
  - 链表结点(*ListNode*)类
  - 链表(*List*)类



# 链表类定义

```
class List; //复合方式
```

```
class ListNode { //链表结点类  
friend class List; //链表类为其友元类  
  
private:  
    int data; //结点数据, 整型  
    ListNode * link; //结点指针  
};
```

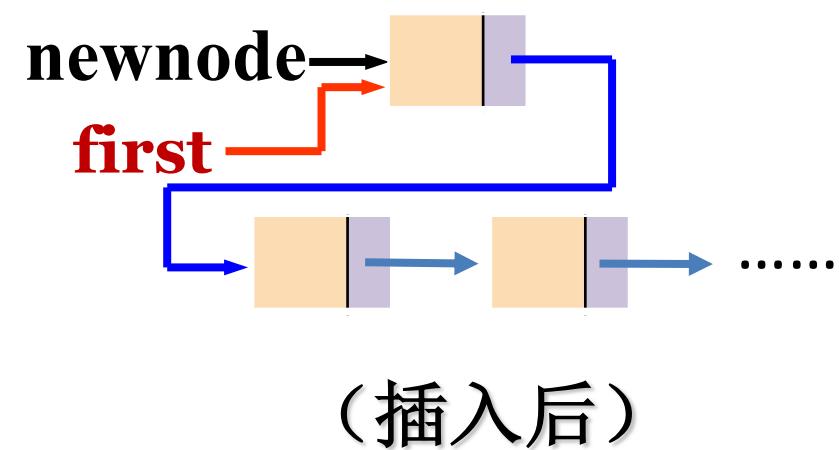
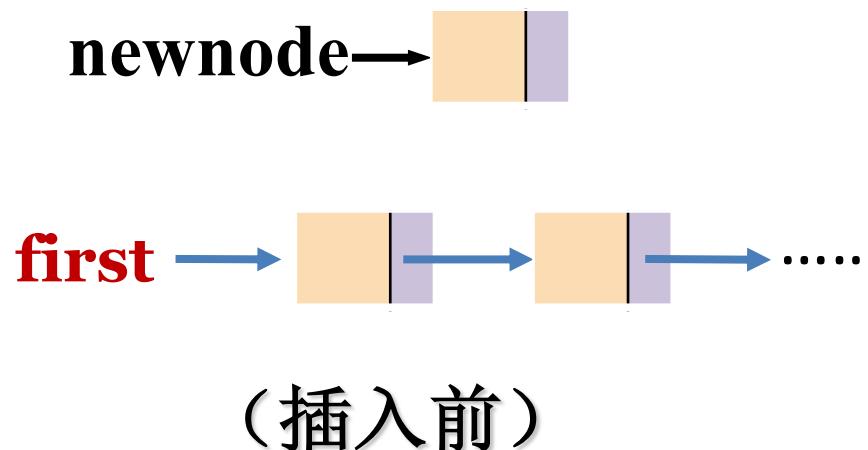
```
class List { //链表类  
private:  
    ListNode *first ; //表头指针  
};
```



# 单链表中的插入

- 第一种情况：在链表**最前端**插入

```
newnode->link = first ;  
first = newnode;
```

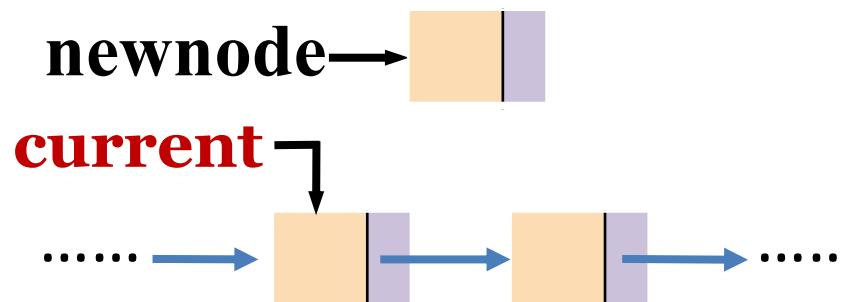




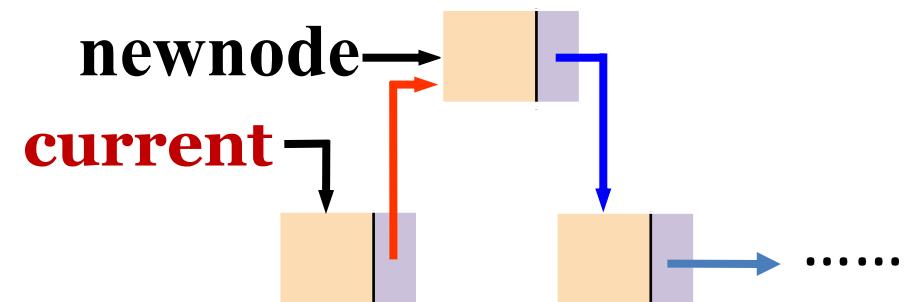
# 单链表中的插入

- ◆ 第二种情况：在链表中间插入

```
newnode->link = current->link;  
current->link = newnode ;
```



(插入前)



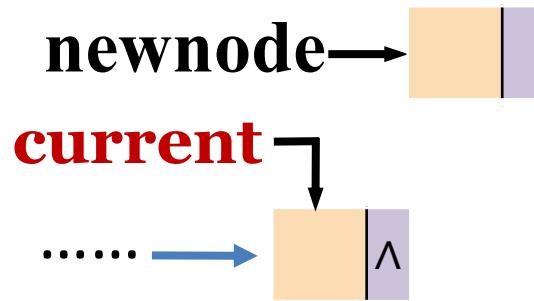
(插入后)



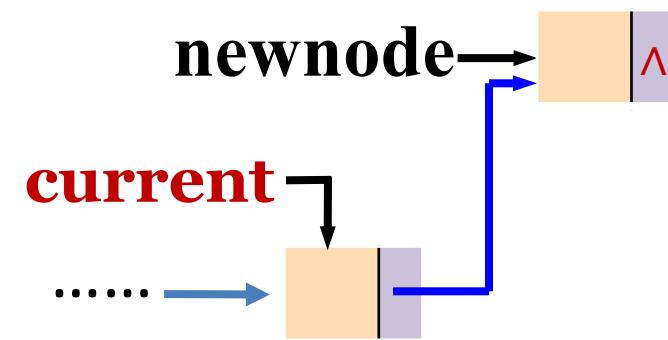
# 单链表中的插入

- ◆ 第三种情况：在链表末尾插入

```
newnode->link = current->link;  
current->link = newnode ;
```



(插入前)



(插入后)



# 单链表的插入算法

```
bool List::Insert(int i, int x) {  
    //将新元素 x 插入到第 i 个结点之后。 i 从1开始，  
    //i = 0 表示插入到首元结点之前。  
    if (first == NULL || i == 0) {    //空表或首元结点前  
        LinkNode *newNode = new LinkNode(x);  
        //建立一个新结点  
        newNode->link = first; first = newNode;  
        //新结点成为首元结点  
    }  
    else {                      //否则，寻找插入位置  
        LinkNode *current = first; int k = 1;  
    }  
}
```

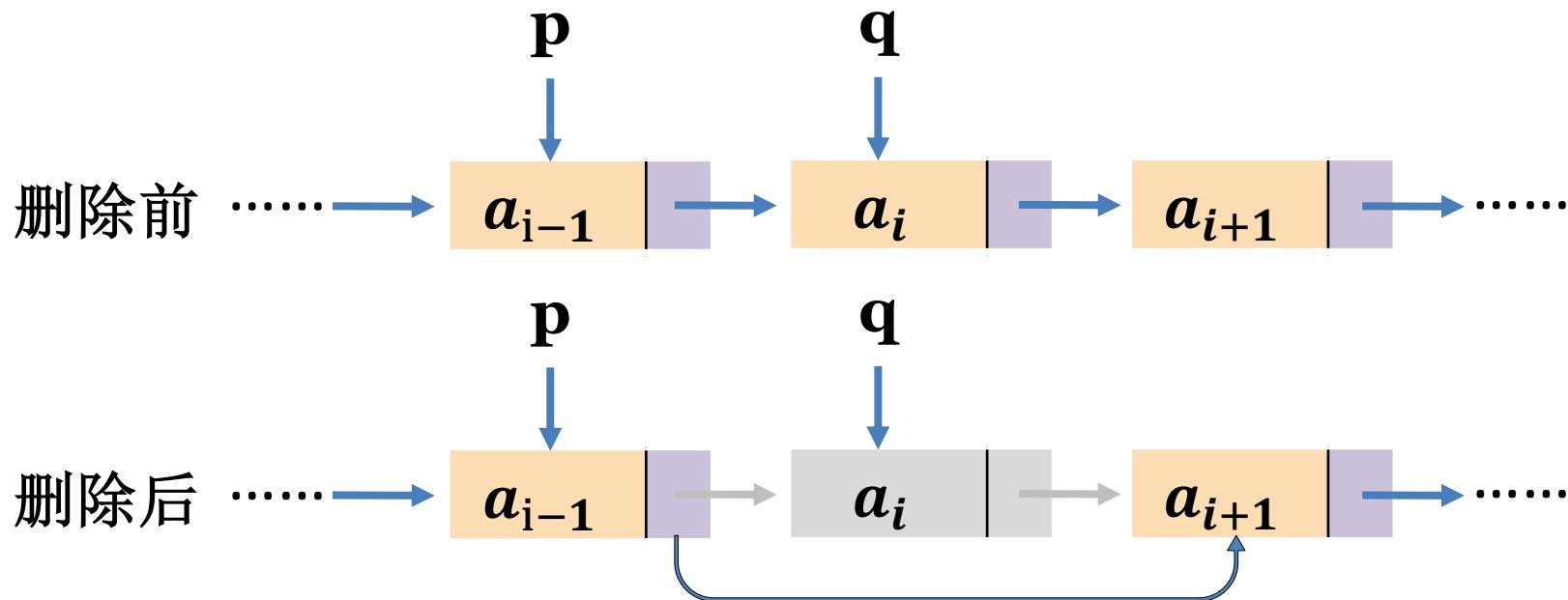


```
while (k < i && current != NULL) //找第i结点
    { current = current->link; k++; }
if (current == NULL && first != NULL) //链短
    {cerr << “无效的插入位置!\n”; return false;}
else { //插入在链表的中间
    LinkNode *newNode = new LinkNode(x);
    newNode->link = current->link;
    current->link = newNode;
}
return true;
};
```



# 单链表的删除算法

- 第一种情况: 删除表中第一个元素
- 第二种情况: 删除表中或表尾元素



在单链表中删除含 $a_i$ 的结点



# 单链表的删除算法

```
bool List::Remove (int i, int& x) {  
    //将链表中的第 i 个元素删去, i 从1开始。  
    LinkNode *del;          //暂存删除结点指针  
    if (i <= 1) { del = first; first = first->link; }  
    else {  
        LinkNode *current = first; k = 1; //找i-1号结点  
        while (k < i-1 && current != NULL)  
        { current = current->link; k++; }  
        if (current == NULL || current->link == NULL)  
        {cout << “无效的删除位置!\n”; return false;}  
    }  
}
```



```
    del = current->link;      //删中间/尾结点
    current->link = del->link;
}
x = del->data; delete del; //取出被删结点数据
return true;
};
```



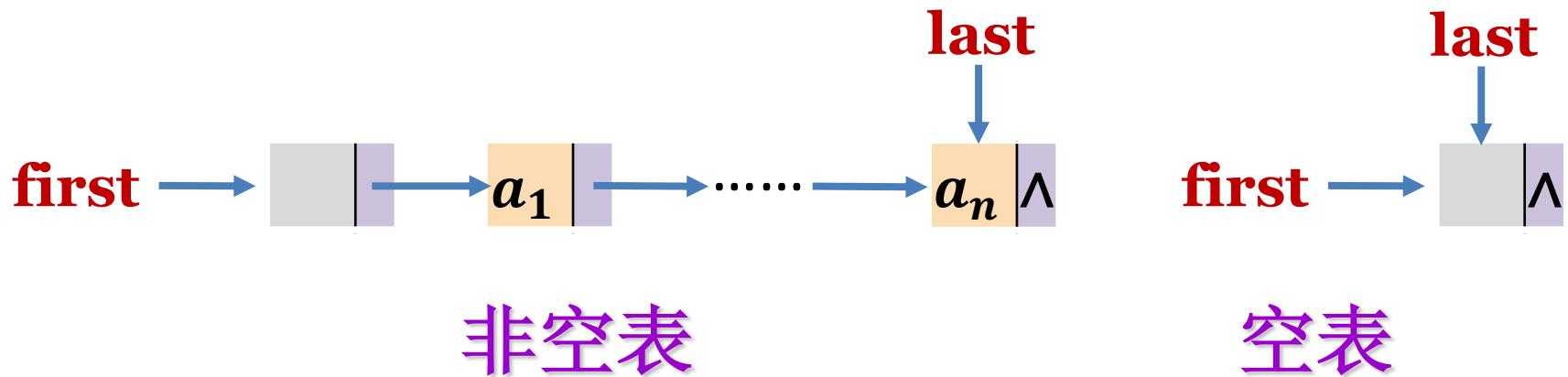
# 单链表的优缺点

- 1) 实现单链表的插入和删除算法，不需要移动元素，只需修改结点指针，比顺序表方便。
- 2) 情况复杂，要专门讨论空表和在表头插入的特殊情形。
- 3) 寻找插入或删除位置只能沿着链顺序检测。



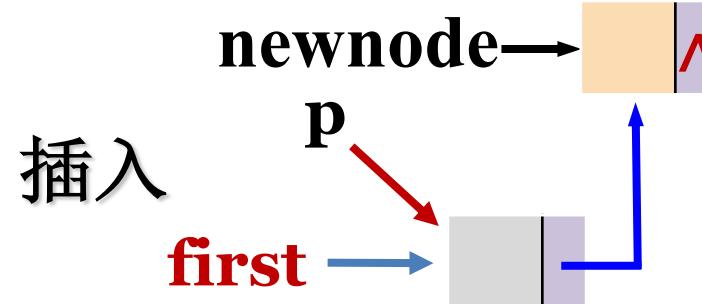
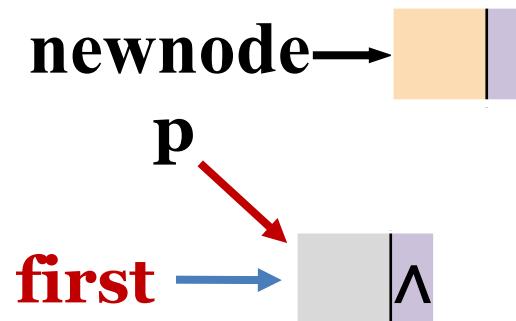
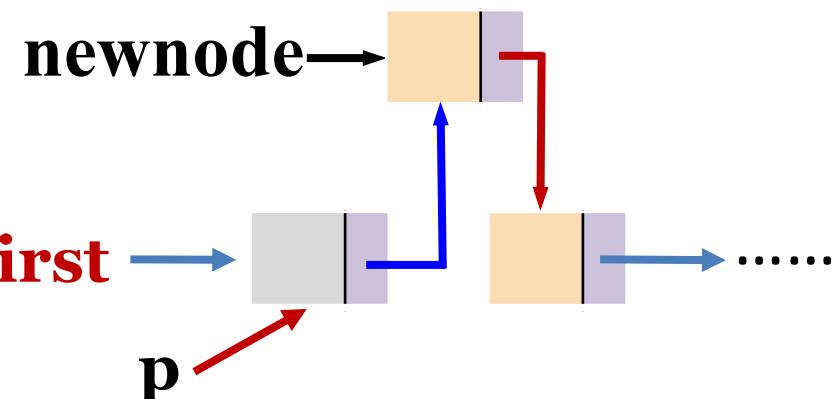
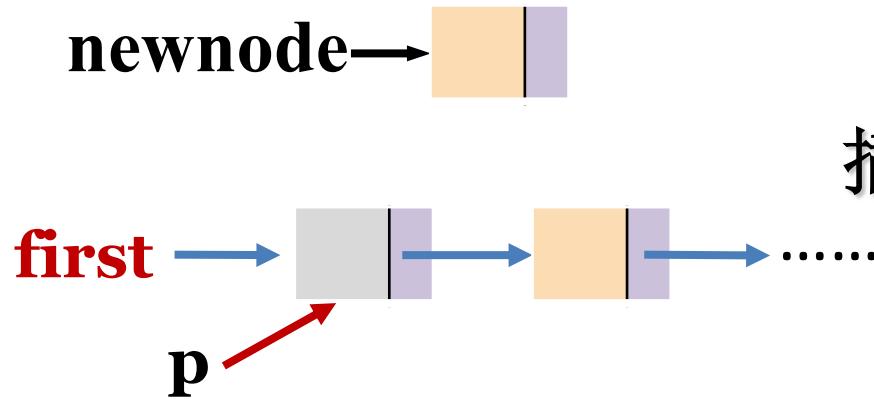
# 带附加头结点（表头结点）的单链表

- 表头结点位于表的最前端，本身**不带数据**，仅标志表头。
- 设置表头结点的目的是**统一空表与非空表的操作**，简化链表操作的实现。





# 在带表头结点的单链表最前端插入新结点



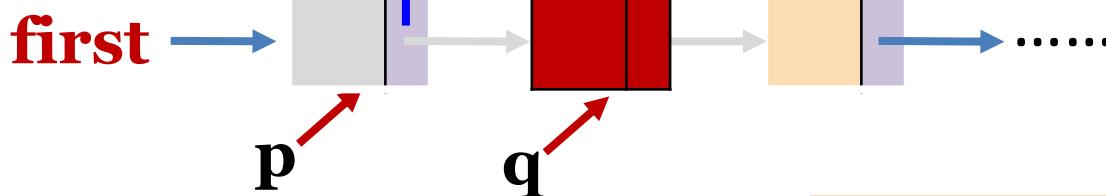
**newnode->link = p->link;**  
**p->link = newnode;**



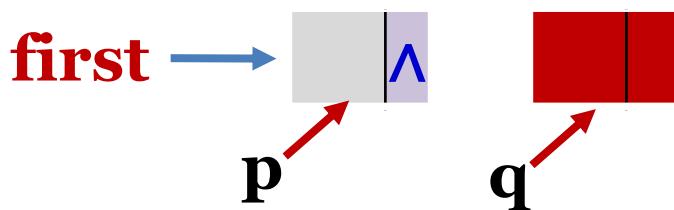
# 从带表头结点的单链表中删除最前端的结点



(非空表)



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



(空表)



# 用模板定义的单链表

```
template <class T> //定义在 "LinkedList.h"
struct LinkNode { //链表结点类的定义
    E data; //数据域
    LinkNode<T> *link; //链指针域
}
数据成员

LinkNode() { link = NULL; } //构造函数
LinkNode(const T& item, LinkNode<T> *ptr
=NULL)
    { data = item; link = ptr; } //构造函数

};
```



**template <class T>**

**class List : public LinearList<T> {**

**//单链表类定义, 不用继承也可实现**

**protected:**

**LinkNode<T> \*first; //表头指针 数据成员**

**public:**

**List( ){ first = new LinkNode<T>; } //构造函数**

**List(const T& x) {first = new LinkNode<T>(x); }**

**List( List<T>& L); //复制构造函数**

**~List( ){makeEmpty( );} //析构函数**

**void makeEmpty( ); //将链表置为空表**

**int Length( ) const; //计算链表的长度**



```
LinkNode<T, E> *Search(T x);      //搜索含x元素
LinkNode<T, E> *Locate(int i);    //定位第i个元素
T *getData(int i);                //取出第i元素值
void setData(int i, T & x);        //更新第i元素值
bool Insert (int i, T & x);        //在第i元素后插入
bool Remove(int i, T & x);        //删除第i个元素
bool IsEmpty() const              //判表空否
{ return first->link == NULL ? true : false; }
LinkNode<T> *getHead( ) const { return first; }
void setHead(LinkNode<T> *p ) { first = p;}
void Sort();                      //排序
};
```



# 单链表的插入算法

```
template <class T>
bool List<T>::Insert (int i, T& x) {
    //将新元素 x 插入在链表中第 i 个结点之后。
    LinkNode<T> *current = Locate(i);
    if (current == NULL) return false;          //无插入位置
    LinkNode<T> *newNode =
        new LinkNode<T>(x);                  //创建新结点
    if (newNode == NULL)                      //动态分配失败
        {cerr << "存储分配错误！" << endl; exit(1);}
    newNode->link = current->link;           //链入
    current->link = newNode;
    return true;                             //插入成功
};
```



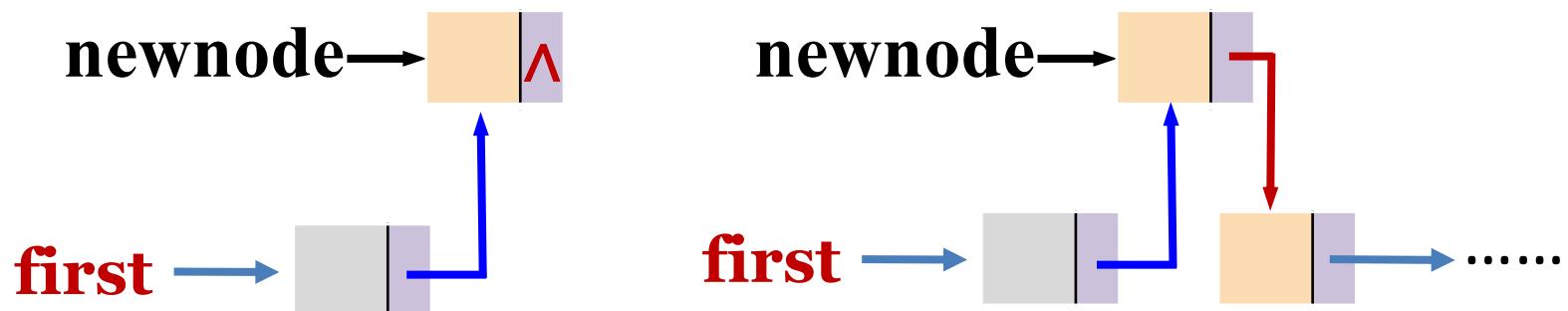
# 单链表的删除算法

```
template <class T>
bool List<T>::Remove (int i, T& x ) {
    //删除链表第i个元素, 通过引用参数x返回元素值
    LinkNode<T> *current = Locate(i-1);
    if ( current == NULL || current->link == NULL)
        return false;          //删除不成功
    LinkNode<T> *del = current->link;
    current->link = del->link;
    x = del->data;           delete del;
    return true;
};
```



# 前插法建立单链表

- 从一个空表开始，重复读入数据：
  - 生成新结点
  - 将读入数据存放到新结点的数据域中
  - 将该新结点插入到链表的前端
- 直到读入结束符为止。



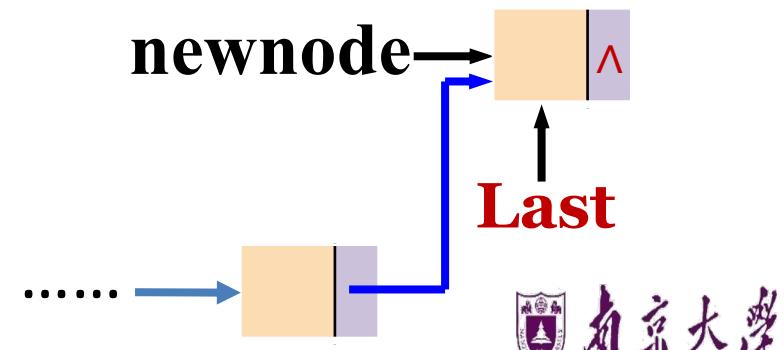
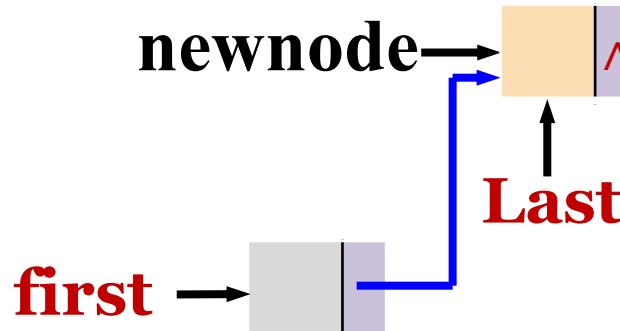
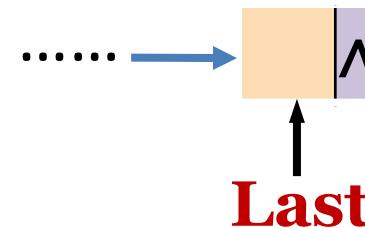
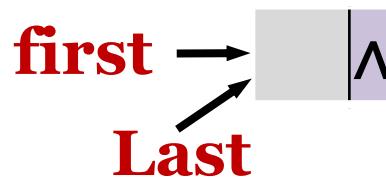


```
template <class T>
void inputFront (T endTag, List<T>& L) {
    LinkNode<T> *newNode, *newF; T val;
    newF = new LinkNode<T>;
    L.setFirst (newF);      //first->link默认值为NULL
    cin >> val;
    while (val != endTag) {
        newNode = new LinkNode<T>(val);
        newNode->link = newF->link;          //插在表前端
        newF->link = newNode;
        cin >> val;
    }
};
```



# 后插法建立单链表

- 每次将新结点加在插到链表的表尾
- 设置一个尾指针 last，总是指向表中最后一个结点，新结点插在它的后面
- 尾指针last初始时置为指向表头结点地址





```
template <class T>
void inputRear ( T endTag, List<T>& L ) {
    LinkNode<T> *newNode, *last; T val;
    last = new LinkNode<T>;           //建立链表的头结点
    L.setFirst(last);                  //为链表L的first赋值
    cin >> val;
    while ( val != endTag ) {          //last指向当前的表尾
        newNode = new LinkNode<T>(val);
        last->link = newNode;   last = newNode;
        cin >> val;             //插入到表末端
    }
    last->link = NULL;                //表收尾
};
```