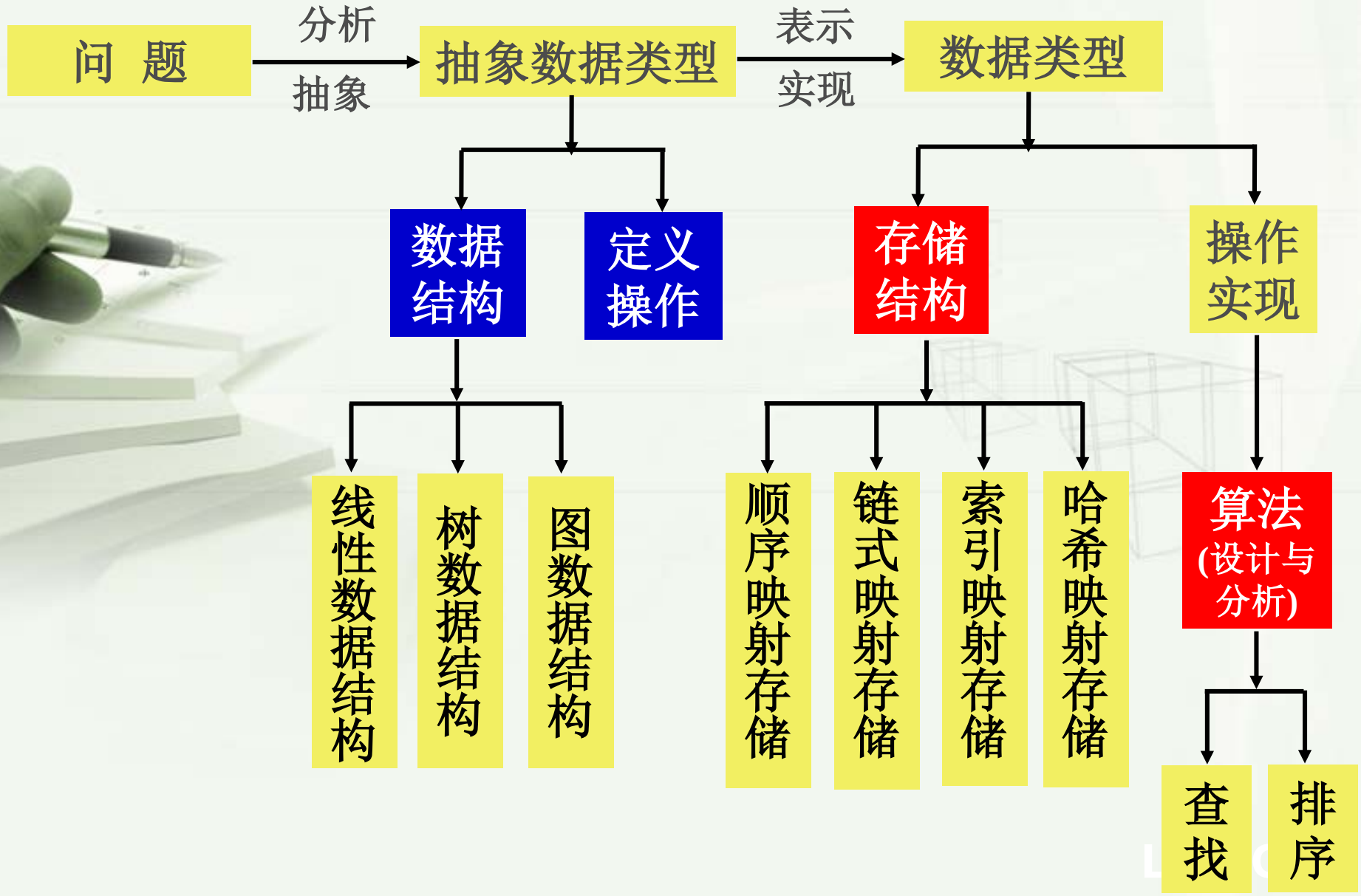

问 题
分析
抽象
抽象数据类型
表示
实现
数据类型
数据结构
定义操作
存储结构
操作实现
线性数据结构
树数据结构
图数据结构
顺序映射存储
链式映射存储
索引映射存储
哈希映射存储
算法(设计与分析)
查找
排序

程序设计语言是问题求解的基本工具

数据结构是问题求解的基础要素

算法设计是问题求解的关键要素

问题求解的本质：

把特定领域中的特定问题的求解过程转换为计算机可执行的程序。

从本章开始，我们将学习现实世界中的三种基本数据结构。对每一种数据结构都是按ADT来介绍和学习，即：

（1）ADT的定义。分析数据的逻辑特性（数据结构），定义常用的操作。

（2）ADT的实现。基于不同的存储结构如何实现ADT(存储结构、算法)

（3）典型应用举例

第二章 线性表

内容提要：

线性数据结构是最简单、应用最广泛和最重要的一种数据结构。本章首先学习最具“一般性”的线性数据结构。

- 一般线性表ADT的定义
  - 数据结构—线性关系
  - 操作定义
- 一般线性表ADT的实现
  - 存储结构
  - 操作实现（算法）
- 典型应用举例

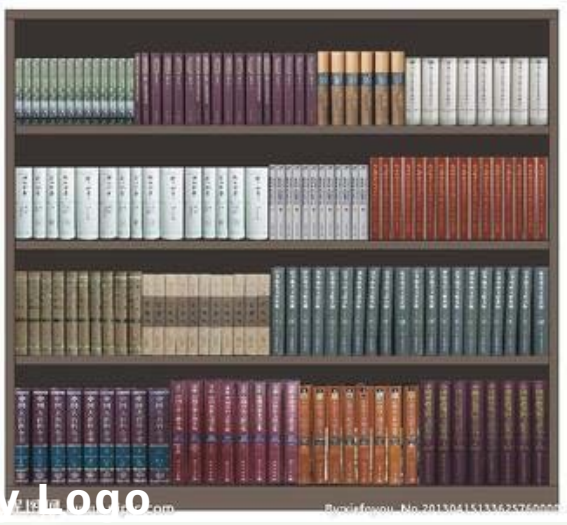
2.1 一般线性表ADT的定义

2.1.1 线性表逻辑结构

[线性表 Linear_List] n(n≥0)个相同特性的数据元素的有限序列，数据元素之间具有线性关系。记作：

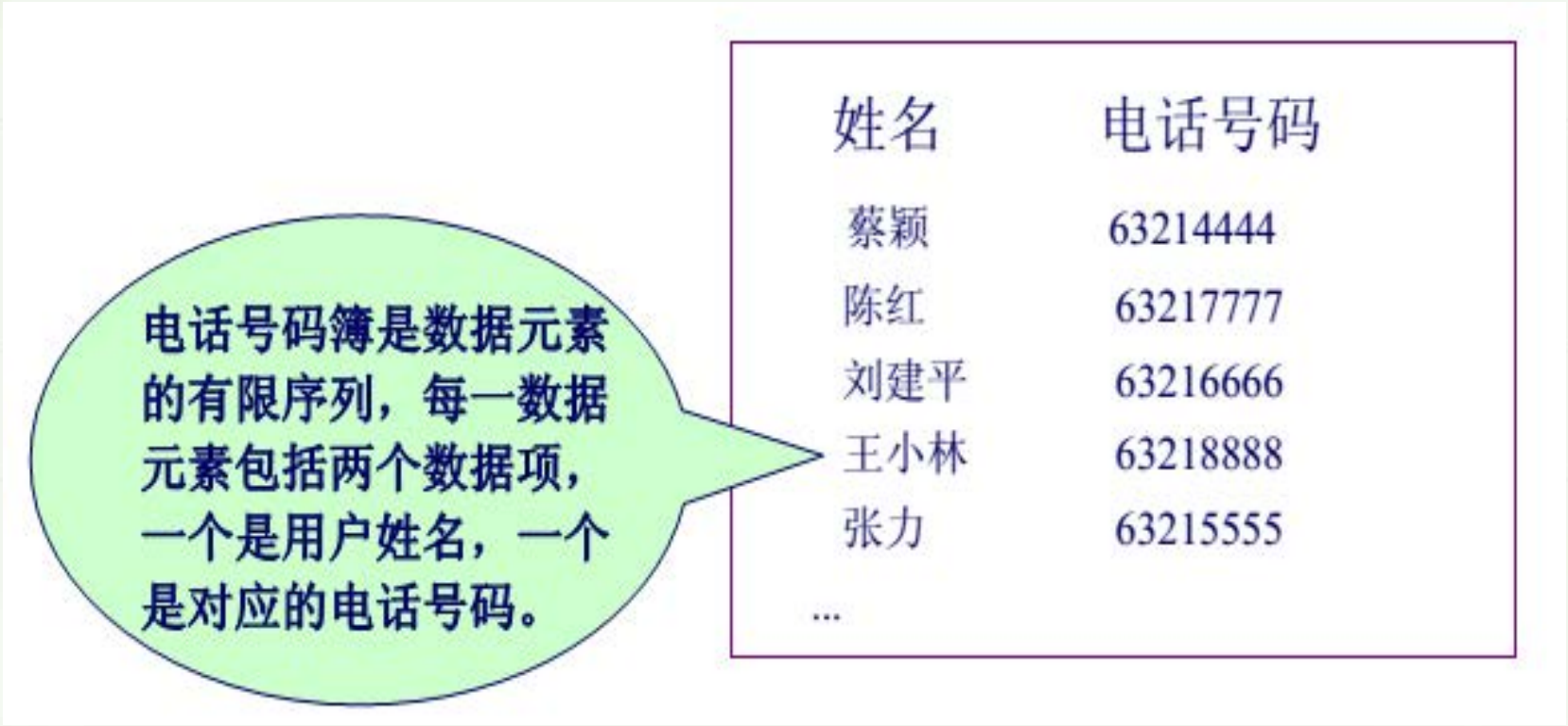
$L=(a_1,a_2,\ldots,a_{i-1},a_i,a_{i+1}\ldots,a_n)$　　其中a_i是数据元素

[线性关系] 除第一个元素外，每个元素有且仅有一个前驱；
除最后一个元素外，每个元素有且仅有一个后继；



2.1 一般线性表ADT的定义

2.1.1 线性表逻辑结构

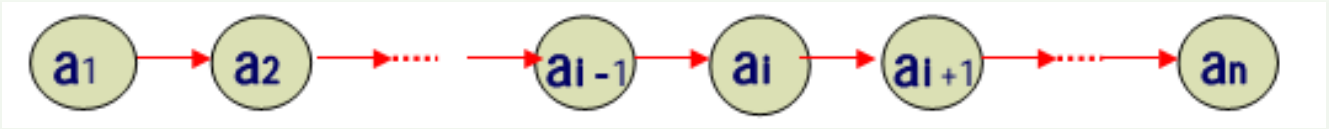


注意：在这几个例子中，“数据元素”是什么？

2.1 一般线性表ADT的定义

2.1.1 线性表逻辑结构

特点： 数据元素之间的关系是它们在数据集合中的相对位置。



术语： 直接前驱 直接后继 空表 长度

一般线性表（数据结构）的形式化表示：

Linear_List=(D,R)

$D=\{a_i \mid a_i \in D_0 \quad i=1,2,\ldots\ldots \quad n\geq 0\}$

$R=\{<a_{i-1},a_i> \mid a_{i-1},a_i \in D_0 \quad i=2,3,4,\ldots\ldots\}$

D_0是某个数据对象（具有相同特性的数据元素的集合）

2.1 一般线性表ADT的定义

2.1.2 线性表上定义的常用操作

在线性表这种数据结构上，经常会有哪些操作呢？

Create(L): 创建一个空表；
Insert(L,i,x): 在第i个元素之后插入元素x
Delete(L,i,x): 删除线性表的第i个元素，删除元素通过x返回
Length(L): 求线性表的长度；
Search(L,x): 在线性表中查找元素x，返回其在表中的位置；
GetData(L,i): 访问线性表的第i个元素；
IsEmpty(L): 判断线性表是否为空；
GetPrior(L,x): 求线性表中元素x的直接前驱；
GetNext(L,x): 求线性表中元素x的直接后继；
PrintList(L): 输出线性表的各个元素；
……

2.1 一般线性表ADT的定义

2.1.3 线性表ADT的定义

ADT描述：

ADT Linear_list is

data structure:

$D=\{a_i \mid a_i \in D_0 \quad i=1,2,\ldots\ldots \quad n \geq 0\}$

$R=\{<a_{i-1},a_i> \mid a_{i-1},a_i \in D_0 \quad i=2,3,4,\ldots\ldots\}$

D_0是某个数据对象，元素类型定义为：ElemType

operations:

Create(L)

Insert(L,i,x)

Delete(L,i,x)

Length(L)

Search(L,x)

GetData(L,i)

IsEmpty(L)

GetPrior(L,x)

GetNext(L,x)

PrintList(L)

END ADT Linear_list

2.1 一般线性表ADT的定义

2.1.3 线性表ADT的定义

假设这个ADT已经实现了，我们就可以像其他数据类型一样来使用它，即说明变量、使用操作。

例1：已知有线性表，要求删除元素 x 的所有出现。

分析：用ADT定义一个线性表，执行查找、删除操作

算法基本思想：

重复：查找、删除，直到线性表中不存在为止。

```
Del_M(Linear_list  &l,ElemType x)
{
        k=Search(l,x);  //查找操作
        while(k!=0)
            {  Delete(l,k,y);   //删除操作
               k=Search (l,x)   //查找操作
            }
}
```

2.1 一般线性表ADT的定义

2.1.3 线性表ADT的定义

例2：设计算法实现集合的“并”运算 。

分析：集合的元素之间没关系的，但是可以用线性表来表示集合。这样集合的运算就可以用线性表的操作来实现。

算法基本思想：

重复：取lb的一个元素，在la表中查找，如果不存在，则在la中插入。直到lb的所有元素处理完为止。

```
Union(Linear_list &la, Linear_list lb)
{       la_len=Length(la);
        lb_len=Length(lb);
        for(i=1;i<=lb_len;i++)
          { x=GetData(lb,i);
            k=Search(la,x);
            if(k==0)  Insert(la, ++la_len,x);
           }
}
```

2.1 一般线性表ADT的定义

2.1.3 线性表ADT的定义

我们学习了C++，线性表ADT也可以用抽象基类表示出来(见教材p44)。

代码2.2

```
Enum bool {false, true}
Template <class T>
class LinearList
{  Public:
        LinearList();
        ~LinearList();
        virtual  int Length() const =0;
        virtual  int Search(T &x) const =0;
        virtual  T *GetData(int i) const =0;
        virtual bool Insert(int i,T& x)=0;
        virtual bool Delete(int i,T& x)=0;
        virtual bool IsEmpty() const =0;
        virtual  T *GetPrior (T& x) const =0;
        virtual  T *GetNext(T& x) const =0;
        virtual  void PrintList()=0;
}
```

2.1 一般线性表ADT的定义

2.1.3 线性表ADT的定义

线性数据结构的分类：

按数据元素分：

- ◆一般线性表—元素没有任何限制
- ◆字符串：—元素限制为字符集
- ◆广义表：—元素又可以是线性表

按实施操作分：

- ◆一般线性表—可以在任何位置插入、删除
- ◆堆栈—只能在一端插入、删除
- ◆队列—插入在一端、删除在另一端
- ◆双端队列—在两端可以插入、删除

按参与关系划分：

- ◆一般线性表—元素只参与一个线性关系（受一个线性关系制约）
- ◆数组—元素可参与多个线性关系 （受多个线性关系制约）

2.2 一般线性表ADT的实现

要虚拟实现（表示）ADT，需要完成：

1.把数据结构表示出来——存储结构

重点：

顺序存储映射

链式存储映射

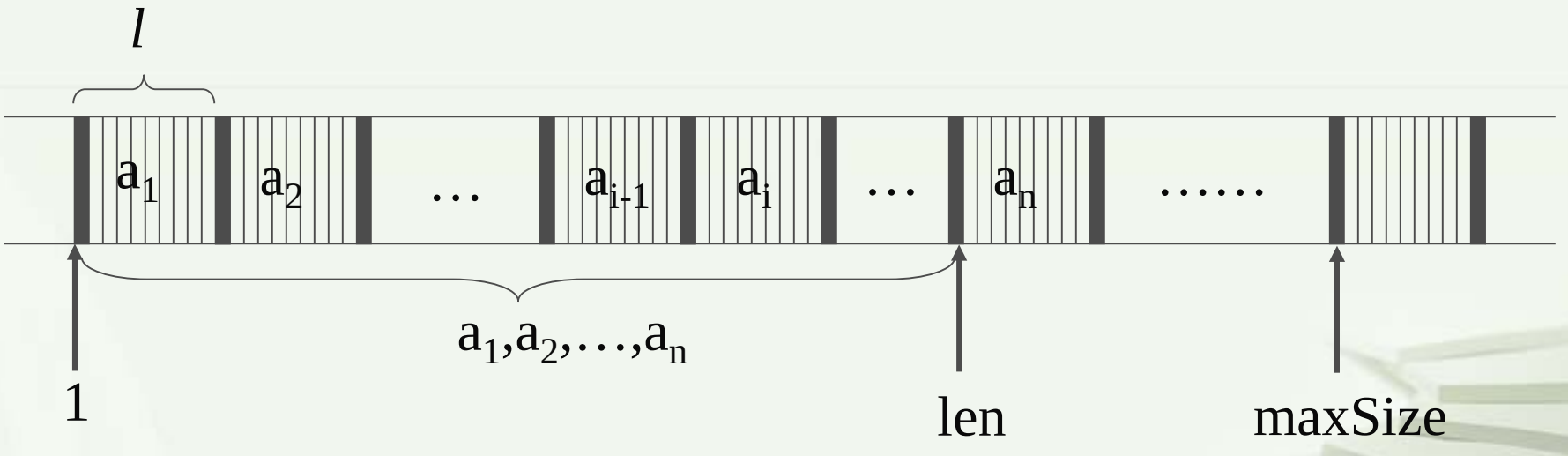
2. 把定义的操作实现出来——设计出算法

2.2 一般线性表ADT的实现

2.2.1 基于顺序映射的线性表ADT实现

2.2.1.1 顺序存储结构

1. 存储方式：用*地址连续*的一组存储单元依次存储线性表的各个元素。具体地：假设存储每个元素占用 *l* 个存储单元，并且以所占的第一个单元的地址作为该元素的存储地址，于是：



l=sizeof(ElemType)　　maxSize>=*n*

2.2 一般线性表ADT的实现

2.2.1 基于顺序映射的线性表ADT实现

2.2.1.1 顺序存储结构

用**LOC(a_i)**表示 **a_i** 的存储地址，则有：

$LOC(a_2)=LOC(a_1)+l$

$LOC(a_3)=LOC(a_2)+l=LOC(a_1)+2*l$

……

$LOC(a_i)=LOC(a_{i-1})+l=LOC(a_1)+(i-1)*l$

LOC(a_1)称为基地址，用BASE表示。

$LOC(a_i)=BASE+(i-1)*l$

2. 特点：

- ♣ 存储空间必须是连续的(可以静态分配，也可以动态分配)
- ♣用物理上的相邻来表示逻辑上的线性关系。逻辑顺序与物理顺序一致。
- ♣ 已知基地址，可以计算出任意元素的存储地址：

$LOC(a_i)=BASE+(i-1)*l$ $\quad\quad$ $2<=l<=n$

2.2 一般线性表ADT的实现

2.2.1 基于顺序映射的线性表ADT实现

2.2.1.1 顺序存储结构

3. 具体实现

用已有的高级语言数据类型来定义顺序存储。因此，凡是可以申请到连续存储空间的数据类型都可以。最简单直接的就是数组（静态或动态）。

```
#define maxSize  允许的最大长度
typedef  数据元素类型  ElemType;
typedef  struct
{ ElemType  data[maxSize];
  int last;
}SeqList;
```

静态顺序存储

```
typedef  数据元素类型  ElemType;
typedef  struct
{ ElemType *data;
  int maxSize;
  int last;
}pSeqList;
```

动态顺序存储

例如： **SeqList *l1*, *l2*;**

pSeqList *l3*;

2.2 一般线性表ADT的实现

2.2.1 基于顺序映射的线性表ADT实现

2.2.1.1 顺序存储结构

特别注意： 无论是静态分配还是动态分配，空间都是一次性得到的、地址连续的，因此空间的大小是有限制的！

——maxSize

静态分配：小了可能会不够；
大了可能会浪费！

动态分配：稍微灵活些，空间不够了可以申请增加空间；
怎么做？

```
void *malloc(size_t  size)
```

```
void *calloc(size_t  num,size_t  size)
```

```
void *realloc(void  *ptr,size_t  size)
```

2.2 一般线性表ADT的实现

2.2.1 基于顺序映射的线性表ADT实现

2.2.1.2 操作的实现

存储结构确定后，前面定义的每个操作就可以设计出算法。

创建空表：Create(*l*)

```
void create(SeqList &l )
{ l.last=-1; }
```

```
void Create(pSeqList &l )
{ l.last=-1;
  l.data=new ElemType[maxSize];
  if(l.data==NULL) {cerr<<“存储分配错
误！ ”<<endl;exit(1);}
}
```

2.2 一般线性表ADT的实现

2.2.1 基于顺序映射的线性表ADT实现

2.2.1.2 操作的实现

求长度：Length(*l*)

```
int  Length(SeqList l )
{ return(l.last+1); }
```

查找： Search(*l*, *x*)

```
int Search(SeqList l, Elemtype x )
{  int i=0;
   while(i <= l.last && l.data[i] != x)
       i++;
   if(i>l.last) return -1
   else return i+1;
}
```

内容回顾

1. 线性表数据结构的逻辑特性

2. 线性表ADT的定义（数据结构+操作）

3. 线性表ADT的实现 之 基于顺序存储结构

（1）如何存储的？ 具有什么优点、缺点？

（2）高级语言如何表示其存储结构？

如何实现其操作？

2.2 一般线性表ADT的实现

2.2.1 基于顺序映射的线性表ADT实现

2.2.1.3 线性表ADT的类定义及实现

C++顺序存储的线性表的类定义

```
#include<iostream.h>
#include<stdlib.h>
typedef  ElemType  T;  //定义数据元素类型为T
class SeqList
{    T *data;            //动态存储的数组存储顺序表
     int MaxSize;        //允许的线性表的最大元素个数
     int last;             //当前最后元素下标
  public:
     SeqList ( int sz );
     ~SeqList ( ) { delete [ ] data; }
```

2.2 一般线性表ADT的实现

2.2.1 基于顺序映射的线性表ADT实现

2.2.1.3 线性表ADT的类定义及实现

```
    int Length() const { return last+1; }   //返回元素的个数
    int Search(T & x) const;   //返回元素x在表中的位置
    void Insert (int i ,T & x);   //在位置i插入元素x
    int Delete(int i,T & x);      //删除值为x的元素
    int IsEmpty () { return last ==-1; } //表空否
    int IsFull () { return last == MaxSize-1; } //判断是否满
    T GetData (int i) {return    data[i-1]   }; //获得第i个元素
    T GetPrior (T& x);   //取x前驱元素
    T GetNext(T& x);     //取x的后继元素
    void PrintList();      //输出线性表
}
```

2.2 一般线性表ADT的实现

2.2.1 基于顺序映射的线性表ADT实现

2.2.1.3 线性表ADT的类定义及实现

下面介绍几个重要成员函数（操作）的实现：

（1）构造函数

算法基本思想：给线性表分配空间，并且置空（长度0）；

```
SeqList :: SeqList ( int sz )
{ //构造函数,通过指定sz，定义数组的长度
    if (sz >0)
   {    data= new T[sz];  //分配连续空间
        if (data != NULL)  //分配成功
          { MaxSize = sz;  last = -1;}
        else  {cerr <<“存储分配错误！”<<endl;exit(1);}
    }
 };
```

2.2 一般线性表ADT的实现

2.2.1 基于顺序映射的线性表ADT实现

2.2.1.3 线性表ADT的类定义及实现

（2）定位（查找）

算法基本思想：逐个判断是否为指定元素—顺序查找方法；

```
int SeqList::Search (T & x) const
 { //搜索函数：在表中从前向后顺序查找 x
    int  i = 0;     //起始位置
    while (i <= last && data[i] != x)
        i++;
    if ( i > last ) return -1;  //没找到
    else return i+1;     //找到
};
```

2.2 一般线性表ADT的实现

2.2.1 基于顺序映射的线性表ADT实现

2.2.1.3 线性表ADT的类定义及实现

（3）插入元素：在第i个元素之前插入元素x（插入到第i个位置）

分析：逻辑上插入后元素x成为第i-1个元素的后继，成为原来第i个元素的前驱。（插入后x为第i个元素）；那么物理上必须要一致（部分元素向后移动）。

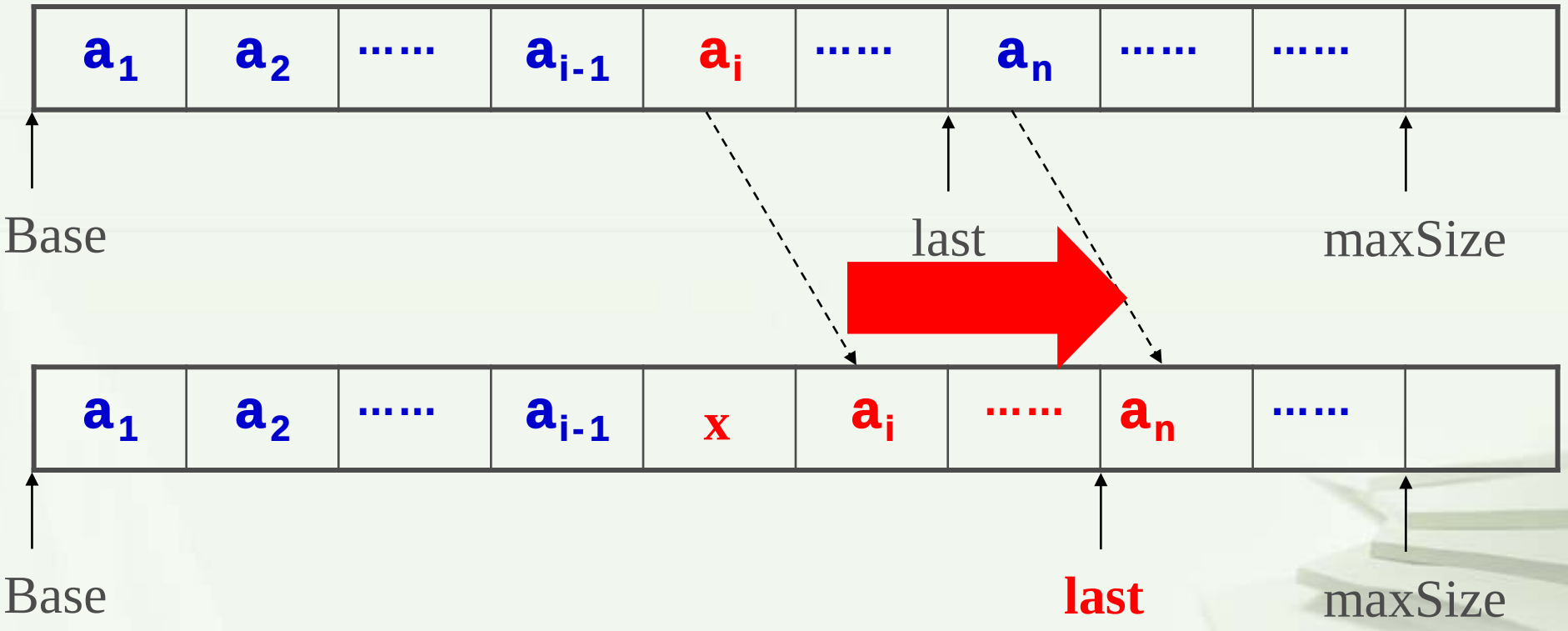
$(a_1, \ldots, a_{i-1}, a_i, \ldots, a_n)$ 改变为 $(a_1, \ldots, a_{i-1}, x, a_i, \ldots, a_n)$

$<a_{i-1}, a_i>$  $<a_{i-1}, x>$, $<x, a_i>$

2.2 一般线性表ADT的实现

2.2.1 基于顺序映射的线性表ADT实现

2.2.1.3 线性表ADT的类定义及实现



2.2 一般线性表ADT的实现

2.2.1 基于顺序映射的线性表ADT实现

2.2.1.3 线性表ADT的类定义及实现

算法基本思想：向后移动部分元素，腾出空间，放置插入元素，长度增加1；

算法如下：

2.2 一般线性表ADT的实现

2.2.1 基于顺序映射的线性表ADT实现

2.2.1.3 线性表ADT的类定义及实现

```
//在指定位置i插入一个数据元素x
void SeqList::Insert   (int i ，  const T& x)
{    //i为下标,不是序号
      if(last == MaxSize-1)
       {  cerr<<“顺序表已满无法插入！ ”<<endl;exit(1);   }
     if(i<0‖i>last+1)
       {   cerr<<"参数i越界出错!"<<endl;    exit(1);   }
     last++;        //当前最后元素下标加1
     for(int j = last ; j > i ; j --)    //移动元素
        data[j]=data[j-1];
    data[i]=x;    //在第i项处插入x
 };
```

时间复杂性？　　最好O(1), 最坏O(n),平均O(n)

2.2 一般线性表ADT的实现

2.2.1 基于顺序映射的线性表ADT实现

2.2.1.3 线性表ADT的类定义及实现

（4）删除元素：删除第i个元素

分析：逻辑上删除第i个元素后，第i-1个元素的后继变为原来的第i+1个元素（原来第i+1个元素的前驱变为第i-1个元素；那么物理上必须要一致（部分元素向前移动）。

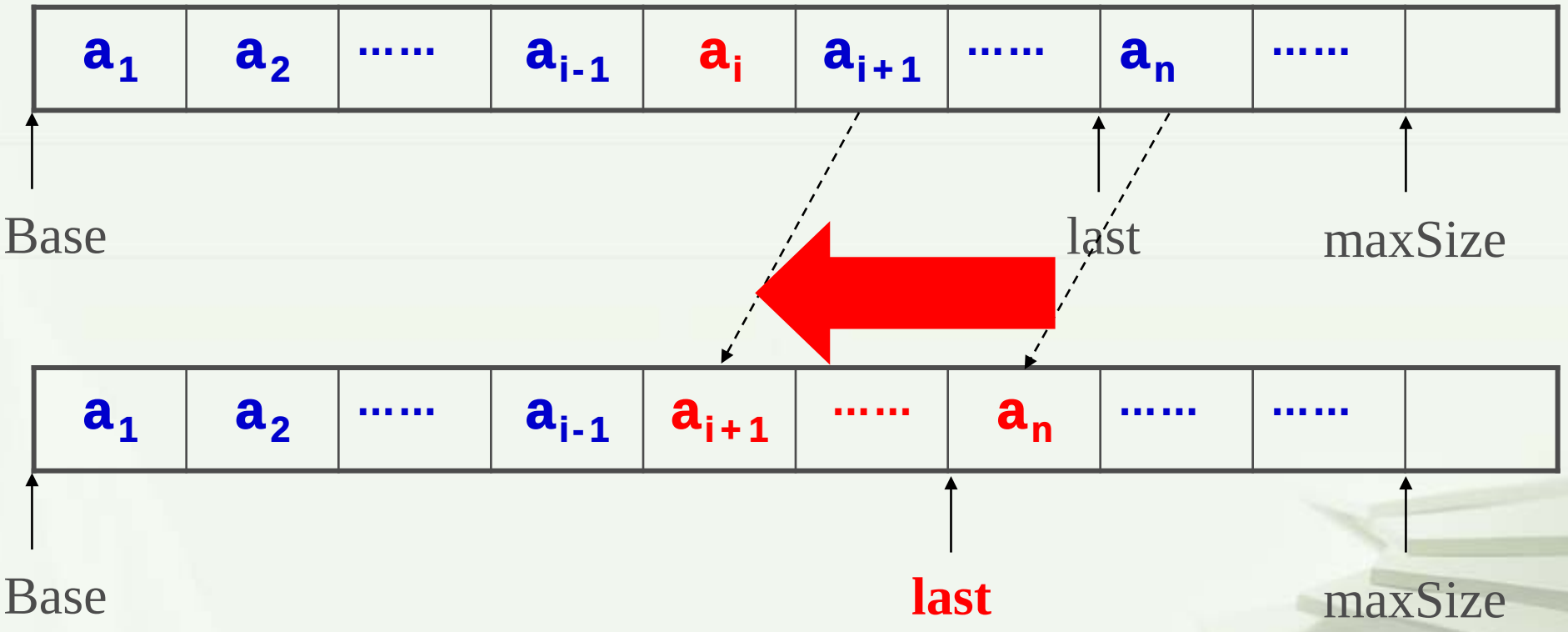
$(a_1, \ldots, a_{i-1}, a_i, \ldots, a_n)$ 改变为 $(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n)$

$\langle a_{i-1}, a_i \rangle, \langle a_i, a_{i+1} \rangle$  $\langle a_{i-1}, a_{i+1} \rangle$

2.2 一般线性表ADT的实现

2.2.1 基于顺序映射的线性表ADT实现

2.2.1.3 线性表ADT的类定义及实现



2.2 一般线性表ADT的实现

2.2.1 基于顺序映射的线性表ADT实现

2.2.1.3 线性表ADT的类定义及实现

```
int SeqList :: Delete ( int i )
{  if ( i >= 0 )
   {  last-- ;          //表长度减1
      for ( int j = i; j <= last; j++ )   //向前移动元素
            data[j] = data[j+1];
       return 1; //成功删除
    }
   return 0; //删除失败
 };
```

时间复杂性？　　最好O(1), 最坏O(n),平均O(n)

思考：要删除指定元素x，算法怎么写？

2.2 一般线性表ADT的实现

2.2.1 基于顺序映射的线性表ADT实现

2.2.1.3 线性表ADT的类定义及实现

更多的其他操作请同学们自己设计写出！

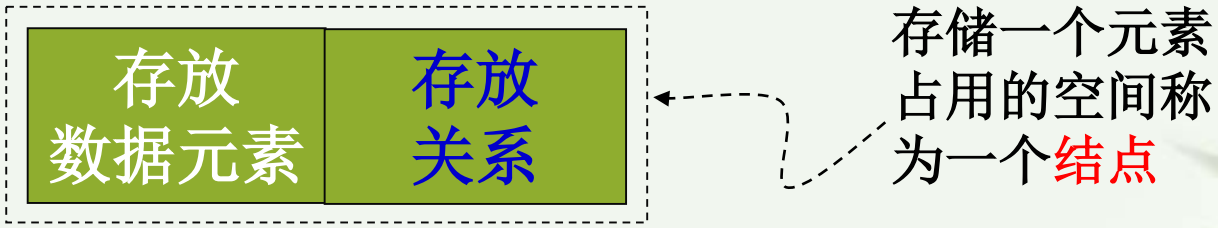
2.2 一般线性表ADT的实现

2.2.2 基于链式映射的线性表ADT实现

2.2.2.1 链式存储结构

1. 存储方式：

用*任意*存储空间单元来存放线性表的各个元素，为了能表示（存储）元素之间的逻辑关系（线性），**在存放每个元素的同时，也存放相关元素的信息（相关元素的存储地址），即用指针来表示元素之间的逻辑关系。** 存放一个数据元素占用的空间为：



2.2 一般线性表ADT的实现

2.2.2 基于链式映射的线性表ADT实现

2.2.2.1 链式存储结构

2. 特点：

♥ 存储线性表的空间不一定连续；

♥ 逻辑关系是由指针（地址）来表示（存储）的；

♥ 元素在逻辑上相邻，但在物理上不一定相邻；

♥ 非随机存取（顺序存取），即访问任何一个元素的时间不同；

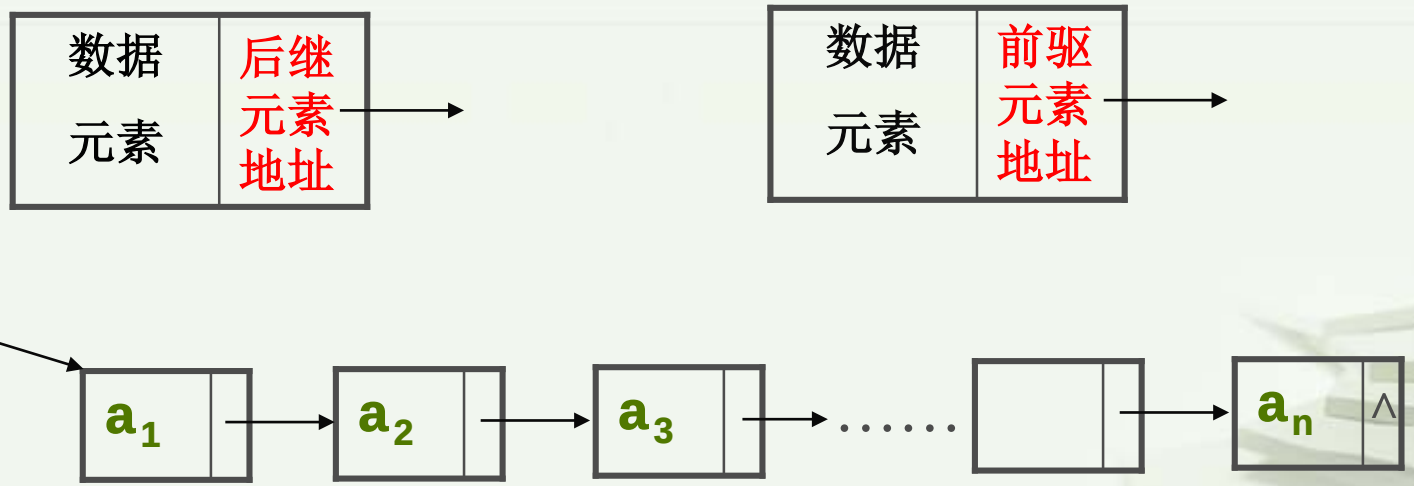
2.2 一般线性表ADT的实现

2.2.2 基于链式映射的线性表ADT实现

2.2.2.1 链式存储结构

3. 链表分类：根据存储相关元素的信息不同，可分为：

单链式存储结构：存放元素的同时，存放其**后继（或前驱）**元素的信息；

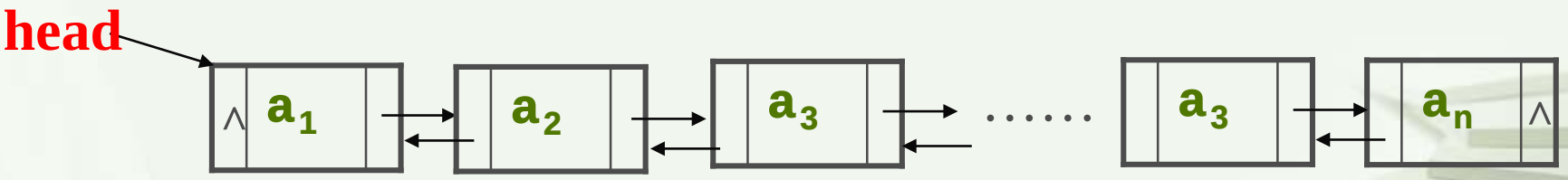


2.2 一般线性表ADT的实现

2.2.2 基于链式映射的线性表ADT实现

2.2.2.1 链式存储结构

双链式存储结构：存放元素的同时，存放其**后继和前驱**元素的信息；

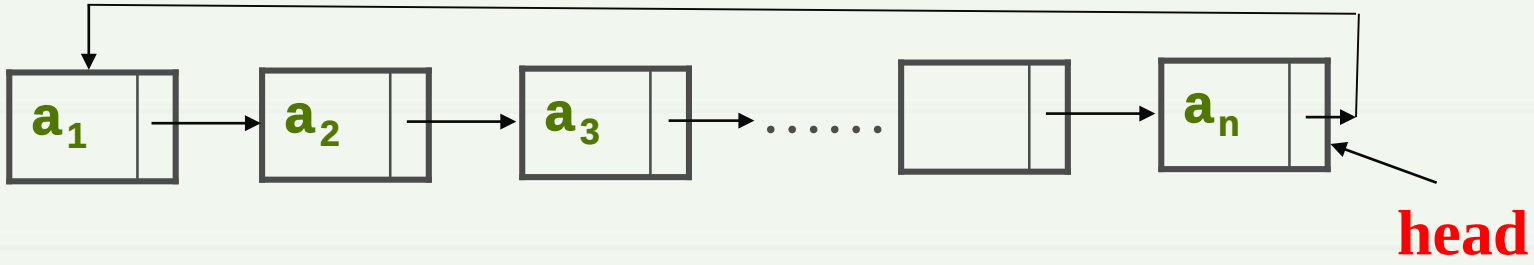


2.2 一般线性表ADT的实现

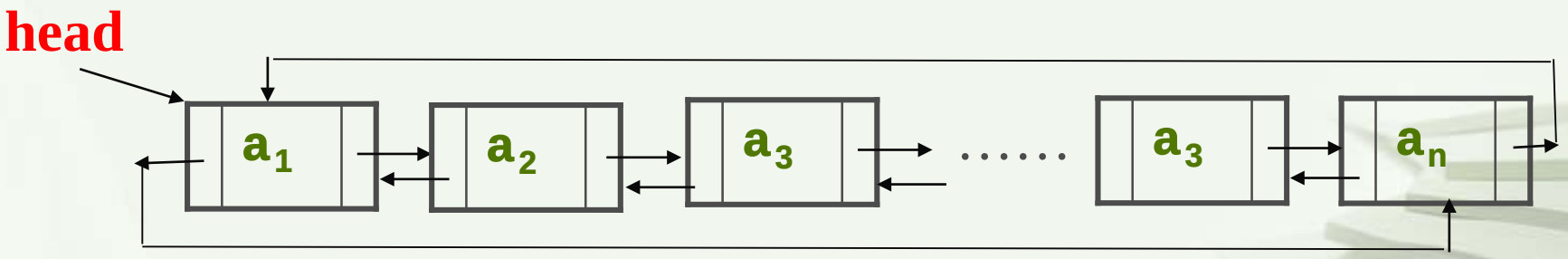
2.2.2 基于链式映射的线性表ADT实现

2.2.2.1 链式存储结构

循环单链：



循环双链：

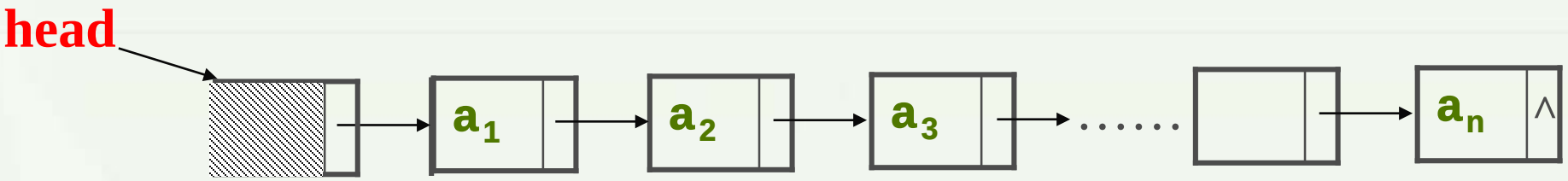


2.2 一般线性表ADT的实现

2.2.2 基于链式映射的线性表ADT实现

2.2.2.1 链式存储结构

有时为了操作方便，在链表的第一个结点之前加一个“头结点”，该结点不存放元素，其指针指向线性表的第一个元素。



2.2 一般线性表ADT的实现

2.2.2 基于链式映射的线性表ADT实现

2. 2. 2. 1 链式存储结构

4. 具体实现：

用高级语言的指针类型实现。

```
typedef 用户数据类型 ElemType;
typedef  struct lnode
{  ElemType  data;
   struct lnode *link;
}lnode,*LinkList;
```

单链式存储

```
typedef 用户数据类型 ElemType;
typedef  struct Dlnode
{  ElemType  data;
   struct Dlnode *llink,*rlink;
}Dlnode,*DLinkList;
```

双链式存储

例如： **LinkList *l1*;**
DLinkList *l2*;

2.2 一般线性表ADT的实现

2.2.2 基于链式映射的线性表ADT实现

2.2.2.2 操作的实现

链式存储结构确定后，前面定义的每个操作就可以设计出算法。

创建空表：**Create(*l*)**

```
void Create(LinkList &l )
{ l=NULL; }
```

```
void Create(LinkList &l )
{ l=(lnode *)malloc(sizeof(lnode);
  l->link=NULL;
}
```

2.2 一般线性表ADT的实现

2.2.2 基于链式映射的线性表ADT实现

2.2.2.2 操作的实现

求长度：**Length(*l*)**

```
int  Length(LinkList l )
{ int i;=0;
   p=l;
   while(p!=NULL)
      { i++; p=p->link;}
   return i;
 }
```

```
int  Length(LinkList l )
{ int i;=0;
   p=l;
   while(p->link!=NULL)
     { i++; p=p->link;}
   return i;
 }
```

2.2 一般线性表ADT的实现

2.2.2 基于链式映射的线性表ADT实现

2.2.2.3 线性表ADT的类定义及实现

C++链式存储的线性表的类定义(定义结点类、链表类)

```
#include <iostream.h>
#include <stdlib.h>
typedef  ElemType  T;  //定义数据元素类型为T
 class LinkList;   //前视定义,否则友元无法定义
 class LinkNode        //链表结点类的定义
{      friend class LinkList;
       private:
          LinkNode *link;
          T data;
        public:
          LinkNode (LinkNode *ptr = NULL)  { link=ptr; }
          LinkNode(const T & item, LinkNode *ptr = NULL)
             { data=item; link=ptr; }
          ~LinkNode(){};
};
```

2.2 一般线性表ADT的实现

2.2.2 基于链式映射的线性表ADT实现

2.2.2.3 线性表ADT的类定义及实现

```
class LinkList
{  private:
       LinkNode *first; //指向链表头的指针
   public:
       LinkList () { first = new LinkNode ();}  //带头结点构造函数
       LinkList ( const T&x )    //不带头结点构造函数
         { first = new LinkNode ( x );  }
       ~LinkList () {MakeEmpty(); delete first;}      //析构函数
       void SetEmpty ( );        //链表置空
        int Length ( ) const;        //求链表长度
        LinkNode *GetHead() const {return first;}
        LinkNode *Search ( T x );
        T GetData ( int i );    //返回第i个元素的值
```

2.2 一般线性表ADT的实现

2.2.2 基于链式映射的线性表ADT实现

2.2.2.3 线性表ADT的类定义及实现

```
    Linknode *Locate(int i) //返回第i个元素的地址（指针）
    void Insert(int i ,T x);    //在第i个元素之后插入元素x
    void Delete(int i,T &x ); //删除第i个元素
    int  IsEmpty()const
      { return(first->link==NULL? 1:0;}
    void InputList(T x);   //建立链表，x是输入结束标志
    void PrintList();    //输出
}
```

2.2 一般线性表ADT的实现

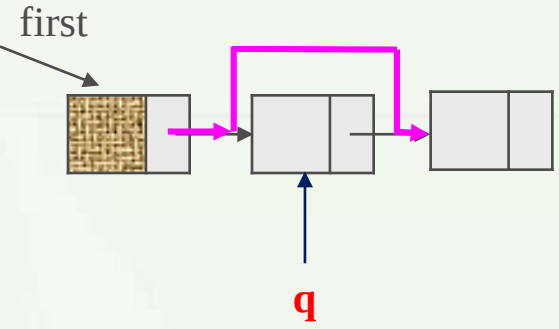
2.2.2 基于链式映射的线性表ADT实现

2.2.2.3 线性表ADT的类定义及实现

下面介绍几个重要成员函数（操作）的实现：

（1）表置空 MakeEmpty()

算法基本思想：把链表的每个结点“摘下”，释放。



q=first->link;

first->link=q->link;

```
void LinkList :: MakeEmpty ( )
 { //删去链表中除表头结点外的所有其它结点
   LinkNode *q;
   while ( first→link != NULL )
   {   q = first→link;     //将一个结点从链中“摘下”
       first→link = q→link;
       delete q;        //释放
   }
};
```

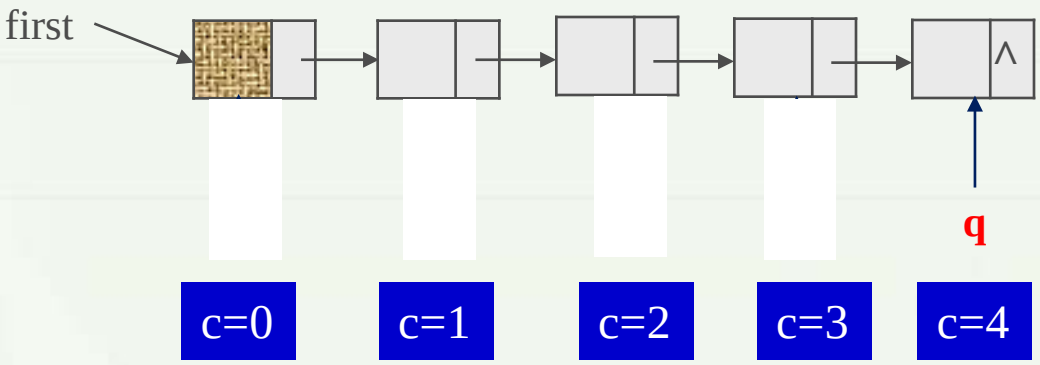
2.2 一般线性表ADT的实现

2.2.2 基于链式映射的线性表ADT实现

2.2.2.3 线性表ADT的类定义及实现

（2）求长度 Length()

算法基本思想：从头开始，边移动指针边计数；




```
int LinkList::Length ( ) const
{  LinkNode *q = first;
   int count = 0;
   while ( q->link != NULL )
   {  count++;
      q=q->link;  }
   return count;
}
```

2.2 一般线性表ADT的实现

2.2.2 基于链式映射的线性表ADT实现

2.2.2.3 线性表ADT的类定义及实现

（3）查找 Search(x)

算法基本思想：从链表头开始，边比较边移动指针；直到找到或到尾。
找到，则返回指向它的指针；
找不到，返回空指针；

```
LinkNode * LinkList::Search( T  x )
{  LinkNode  *p = first→link;    //指针 p 指示第一个结点
   while ( p != NULL && p→data != x )
       p = p→link;
   return p;
}
```

2.2 一般线性表ADT的实现

2.2.2 基于链式映射的线性表ADT实现

2.2.2.3 线性表ADT的类定义及实现

（4）定位 Lcoate(int i)：

算法基本思想：从链表头开始，边计数边移动指针；计数到i停止。

```
LinkNode *LinkList::Locate ( int i )
{   LinkNode  *p=first;
    int j=0;   // j计数
    if ( i < 0 ) return NULL;
     while ( p!= NULL && j<i )          // j = i 停
     {  p=p→link;   j++; }
     return p;
}
```

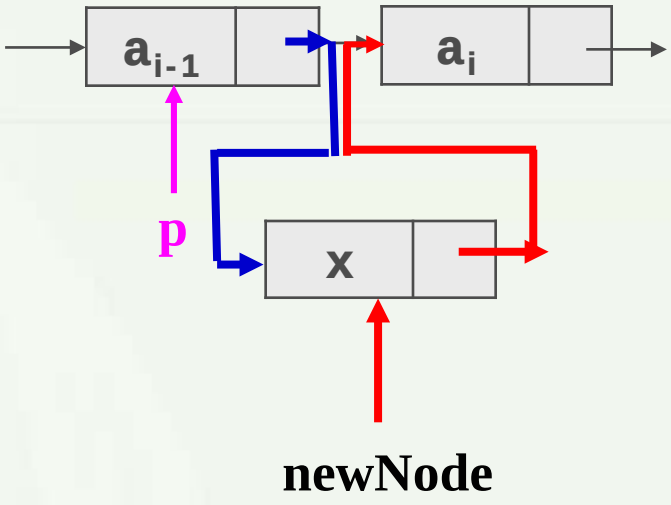
2.2 一般线性表ADT的实现

2.2.2 基于链式映射的线性表ADT实现

2.2.2.3 线性表ADT的类定义及实现

（5）插入 Insert(int i,T x):

算法基本思想：找到第i-1个元素，使x成为其后继，其原来的后继成为x的后继。




```
void LinkList::Insert(int i,T x)
{  LinkNode *p = Locate ( i-1);
   LinkNode * newNode = new LinkNode (x) ;
   newnode->link=p->link;
   p->link=newNode;
}
```

newNode->link=p->link

p->link=newNode

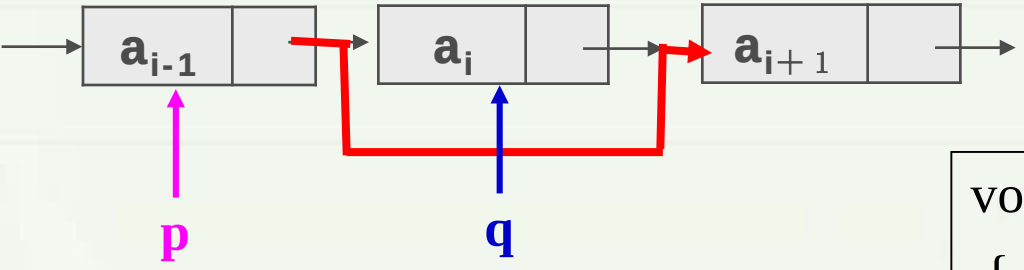
2.2 一般线性表ADT的实现

2.2.2 基于链式映射的线性表ADT实现

2.2.2.3 线性表ADT的类定义及实现

（6）删除 Delete(int i,T &x):

算法基本思想：找到第i-1个元素，使第i+1个元素成为其后继。释放第i个元素。



p=Locate(i-1)

p-.link=p->link->link

```
void LinkList::Delete (int i ,T &x)
 {   LinkNode  *p = Locate (i-1), *q;
     q = p->link;
     p->link = q->link;    //重新链接
     x = q->data;
     delete q;
}
```

2.2 一般线性表ADT的实现

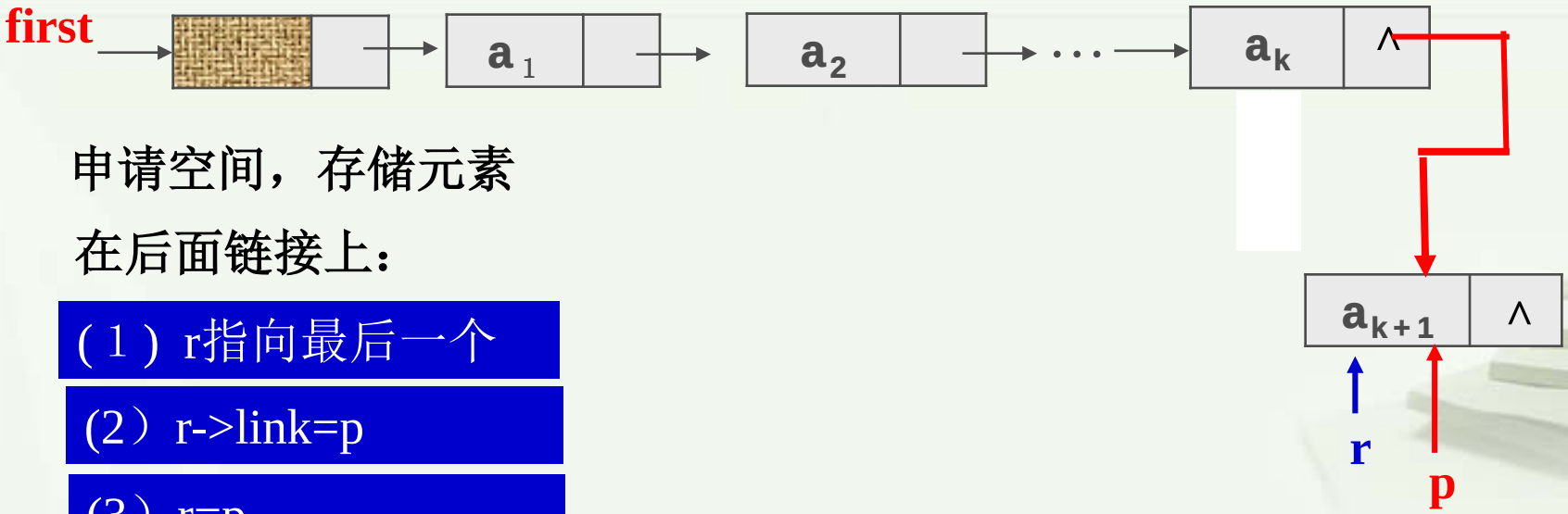
2.2.2 基于链式映射的线性表ADT实现

2.2.2.3 线性表ADT的类定义及实现

（7）输入 InputList()：

算法基本思想：逐个输入数据元素，插入到链表中。

（1）新元素总是链接在后面（一般习惯），元素按自然顺序输入；
（2）新元素总是链接在前面，元素按相反顺序输入；



申请空间，存储元素

在后面链接上：

（1）r指向最后一个

（2）r->link=p

（3）r=p

2.2 一般线性表ADT的实现

2.2.2 基于链式映射的线性表ADT实现

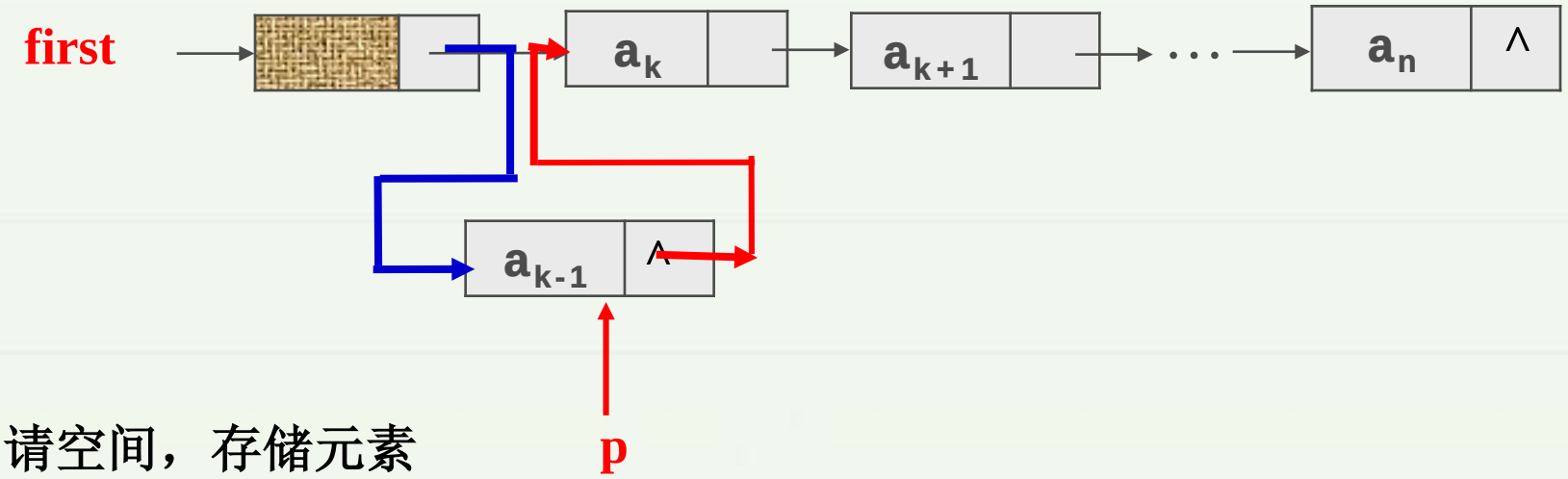
2.2.2.3 线性表ADT的类定义及实现

```
void LinkList :: input (T endTag)
{   LinkNode  *newNode,*r; T val;
    first=new LinkNode ();    //生成头结点
    if (first==NULL) {cerr<<"存储分配错误"<<endl;exit(1);}
    cin>>val;  //读入数据元素
     r=first;  //r总是指向目前链表的最后一个结点
    while(val!=endTag)    //判断是否是结束标志
    {       newNode=new LinkNode (val);  //申请空间，存储读入的元素
            if (newNode==NULL)  {cerr<<"存储分配错误"<<endl;exit(1);}
            r->link=newNode;    //链接到最后面
            r=newNode;    //刚链上的结点成为最后一个结点
             cin>>val; //读入数据元素
    }
}
```

2.2 一般线性表ADT的实现

2.2.2 基于链式映射的线性表ADT实现

2.2.2.3 线性表ADT的类定义及实现



申请空间，存储元素

在前面插入：

```
( 1 )   p->link=first->link
```

```
(2)   first->link=p
```

2.2 一般线性表ADT的实现

2.2.2 基于链式映射的线性表ADT实现

2.2.2.3 线性表ADT的类定义及实现

```
void LinkList :: input (T endTag)
{   LinkNode  *newnode; T val;
    first=new LinkNode ();  //生成头结点
    if (first==NULL) {cerr<<"存储分配错误"<<endl;exit(1);}
    cin>>val;  //按逆序读入元素
    while(val!=endTag)   //判断是否结束标志
    {      newNode=new LinkNode (val);  //申请空间，存放元素
           if (newNode==NULL)  {cerr<<"存储分配错误"<<endl;exit(1);}
           newNode->link=first->link;  //链接到前面
           first->link=newNode;    //成为第一个结点
           cin>>val;  //读入元素
    }
}
```

内容回顾

1. 线性表ADT的实现 之 基于链式存储结构

（1）如何存储的？ 具有什么优点、缺点？

（2）高级语言如何表示其存储结构？

如何实现其操作？

难点与重点：指针概念的理解和操作

2.2 一般线性表ADT的实现

2.2.2 基于链式映射的线性表ADT实现

2.2.2.4 线性表基于链式存储结构的其他形式的实现

双链式存储结构

循环单链式存储结构

循环双链式存储结构

存储形式都是单链的变种；
在操作实现上会不同！

教材P66-73 2.4请同学们自学。

2.3 线性表ADT的应用举例

现实世界中的问题，很多都是具有线性数据结构，这类问题的求解都可以运用学习的线性表知识来解决。

2.3.1 通讯录管理

问题描述：人和人的交流需要联络信息，目前的交流媒介和方式越来越多，联络信息也越来越多庞大，因此方便的管理个人联络信息就十分必要了，这就是通讯录管理的由来。通讯录管理已经成为手机等移动终端不可缺少的工具软件，通过该软件可以完成：新建（增加）、删除、查询、修改、查找、保存、排序等等。

要求：

（1）分析问题，确定问题的数据及关系；

（2）抽象出该问题的ADT;

（3）选择合适的存储结构，实现你定义的ADT;

2.3 线性表ADT的应用举例

2.3.1 通讯录管理

数据： 具体记录联系人的哪些信息，可以自己确定。一般应包括：
序号、姓名、昵称、移动电话、家庭电话、工作电话、EMAIL、QQ、微信……。——确定数据元素

关系： 线性关系。在每个数据项上都存在着线性关系。——为了查询方便

操作： 如何来管理数据。——提供给用户的功能

ADT定义： 与线性表类似（略）

ADT实现：

（1）选择一种存储结构，存储元素和关系；
（2）设计算法，实现每一个操作；

交互界面： 如何把功能提供给用户，如何和用户交互；

2.3 线性表ADT的应用举例

2.3.2 一元n次多项式的处理

问题描述：符号处理是一类非数值性问题，一元多项式就是符号处理的一类实例。一个一元n次多项式的一般形式如下：

$P_n(x)=p_1x^{e1}+p_2x^{e2}+\ldots+p_mx^{em}$

其中 $p_1,p_2,\ldots,p_m$为各项的系数，非零；

$e_1,e_2,\ldots,e_m$ 为各项的指数，满足$0\le e_1 \le e_2 \le \ldots \le e_m$

现要求在计算机中存储这样的多项式，并能对它们进行处理，如：加法、减法、乘法等等。

要求：

（1） 抽象并定义出问题的ADT，即包括：

·数据结构

·常用操作

（2）实现你定义的ADT。即包括：

·选择合适的存储结构（分析存储结构的优缺点）

·对每个操作设计出算法（分析每个算法的时间复杂性）

2.3 线性表ADT的应用举例

2.3.2 一元n次多项式的处理

说明：

（1）多项式的操作至少应包括：

创建：在计算机中存储一个多项式；

输出：以直观的形式输出多项式；

求值：给定一个X_0，求多项式的值；

求导：

加法：两个多项式相加；

减法：两个多项式相减；

乘法：两个多项式相乘；

（2）有能力的同学可以把顺序存储和链式存储都做一下。

（3）有能力的同学可以把交互界面也做一下（比如窗体菜单或文本菜单）

本章小结

重点和难点：

1. 线性数据结构的逻辑特征：线性关系-反映的是元素在集合中的位置

2. 线性数据结构的常用的操作定义：插入、删除、查找、长度

3. 一般线性表ADT的定义：把操作是干什么的定义出来

4. 一般线性表ADT的实现：

（1）基于顺序存储的：

存储方式、特点（优缺点）、具体实现——数组

操作实现：操作具体是怎么做的——算法设计

线性关系是通过物理上相邻来表示的。操作时使逻辑关系发生了变化，那么存储上必须要同时反映出来！——移动

本章小结

重点和难点：

4. 一般线性表ADT的实现：

（2）基于链式存储的：

存储方式、特点（优缺点）、具体实现——指针

操作实现：操作具体是怎么做的——算法设计

线性关系是通过记录有关系的元素的地址来表示的。操作时使逻辑关系发生了变化，那么存储上必须要同时反映出来！——改变指针

注意：链表带头结点和不带头结点的区别！

你在解决问题时应该明确是否带头结点。

本章小结

重点和难点：

线性表上的操作特别多，也变化多样，因此设计和实现操作是线性表部分的重点和难点！

未知存储结构时，设计的算法——抽象算法（基本思想）

已知存储结构时，设计的算法——具体算法（详细描述）

P84-87 练习题：

2.6, 2.9, 2.10, 2.12, 2.14, 2.15, 2.17, 2.21, 2.22, 2.23


END