第二章　线性表

- 2.1 线性表的基本概念（逻辑结构）
- 2.2 线性表的存储结构
  - 2.2.1 顺序存储结构
  - 2.2.2 链式存储结构
- 2.3 线性表的操作算法
  - 2.3.1 顺序表的操作算法
  - 2.3.2 链表的操作算法
- 2.4 线性表的应用

线性结构的特点：

在数据元素的非空有限集中：

（1）存在唯一的一个被称为“第一个”的数据元素（开始结点）；

（2）存在唯一的一个被称为“最后一个”的数据元素（终端结点）；

（3）除第一个之外，集合中的每个数据元素均只有一个前驱；

（4）除最后一个之外，集合中的每个数据元素均只有一 个后继 .

2.1 线性表的基本概念

线性表 —— 由n(n≧0)个数据元素(结点) a_1，a_2，…a_n 组成的有限序列。其中数据元素的个数n定义为表的长度。

- 常常将非空的线性表(n>0)记作：

  $(a_1, a_2, \ldots, a_n)$

- 当 n = 0 时称为空表，

- 线性表中的数据元素在不同的情况下可以有不同的类型。

- 例：26个英文字母组成的字母表
（A，B，C, … , Z）

- 例：一副扑克的点数
（2，3，4，…，J，Q，K，A）

- 例：学生健康情况登记表如下：

姓　名	学　　号	性　别	年龄	健康情况
王小林	790631	男	18	健康
陈　红	790632	女	20	一般
刘建平	790633	男	21	健康
张立立	790634	男	17	神经衰弱
……	……	……	……	……

☞线性表的基本运算

- 数据的运算是定义在逻辑结构上的，
  而运算的具体实现则是在存储结构上进行的。

- 通常线性表有以下几种基本运算：
  1. 创建初始的空线性表(InitList)；
  2. 在线性表中插入一个元素(ListInsert)；
  3. 在线性表中删除某个元素(ListDelete)；
  4. 在线性表中取出某个特定元素(GetElem)；
  5. 在线性表中查找某个元素元素(LocateElem)；
  6. 求线性表的长度(ListLength)；
  7. 判别一个线性表是否为空表(ListEmpty)。
  ……

2.2 线性表的存储结构

- 2.2.1 顺序存储结构
- 2.2.2 链式存储结构

2.2.1 顺序存储结构

线性表的顺序存储结构：

把线性表中的数据元素按逻辑顺序依次存放在一组地址连续的存储单元里。

用这种方法存储的线性表简称顺序表。

线性表的顺序存储结构示意图

逻辑结构

$(a_1, a_2, \ldots, a_i, \ldots, a_n)$

存储地址	数据元素
$Loc(a_1)$	a_1
$Loc(a_1)+c$	a_2
…	…
$Loc(a_1)+(i-1)*c$	a_i

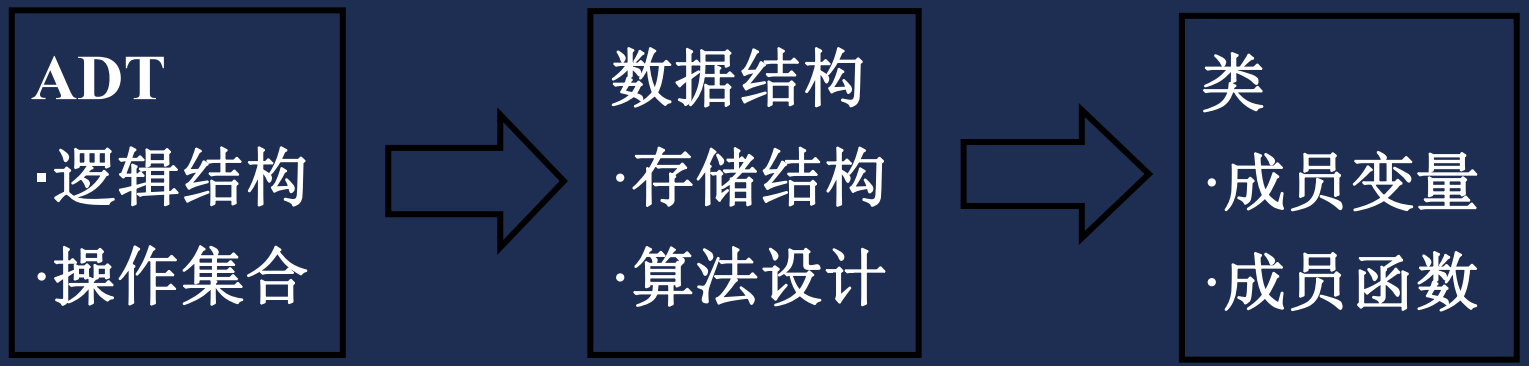
顺序表的特点是：元素的逻辑位置相邻，其物理位置也相邻。

$$Locate(a_{i+1}) = Locate(a_i) + sizeof(ElemType)$$

$$Locate(a_i) = Locate(a_1) + sizeof(ElemType) *(i-1)$$

用C++如何实现顺序表？

ADT的不同视图



(a)使用视图
ADT的定义

(b) 设计视图
ADT的设计

(c) 实现视图
ADT的实现

用C++如何实现顺序表？

- 顺序表类的定义？
  - 成员变量

    一个一维数组、顺序表的长度
  - 成员函数

    顺序表初始化、插入元素、删除元素等

```
template <class T, int MaxSize>
class SeqList{
   T data[MaxSize];          //用于存放数据元素的数组
   int length;               //顺序表中元素的个数
public:
   SeqList( );               //无参构造函数
   SeqList(T a[], int n);    //有参构造函数
   int ListLength();         //求线性表的长度
   T Get(int pos);           //按位置查找
   int Locate(T item);       //按值查找
   void PrintSeqList();      //遍历顺序表
   void Insert(int i, T item); //插入元素
   T Delete(int i);          //删除元素
};
```

如何实现这些操作？

- 顺序表上实现的基本操作
  - 注意：C语言中的数组下标从“0”开始，因此，若L是SeqList类型的顺序表，则表中第i个元素是L.data[i-1]。

1. 初始化操作——构造函数

- 顺序表的无参构造函数

```
template <class T, int MaxSize>
SeqList<T, MaxSize >:: SeqList( )
{  length=0;  }
```

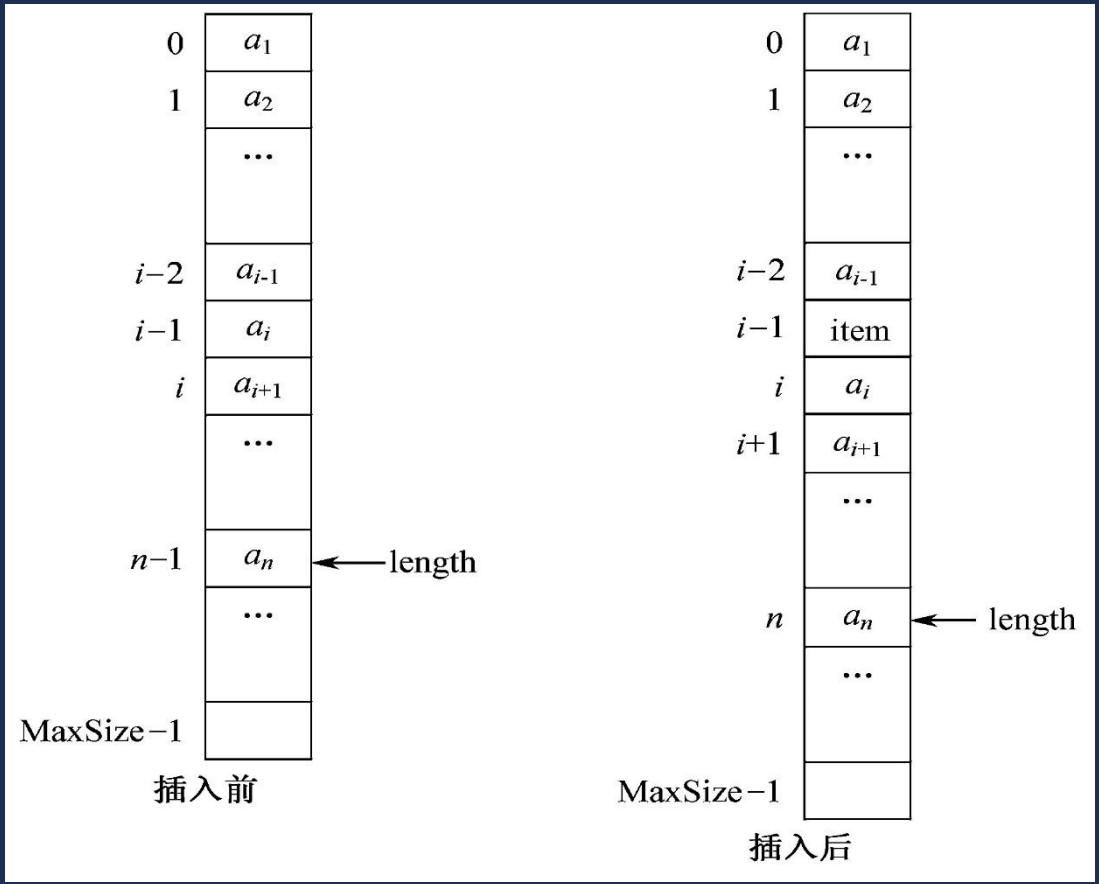
- 顺序表的有参构造函数

```
template <class T, int MaxSize>
SeqList<T, MaxSize>:: SeqList(T a[], int n)
{
    if (n>MaxSize) { cerr<< "参数非法"; exit(1);}
    for (i=0; i<n; i++) data[i]=a[i];
    length=n;
}
```

2. 插入元素操作

- 插入运算

　　是指在表的第 i（1≦i≦n+1）个位置上，插入一个新结点item，使长度为n的线性表变为长度为n+1的线性表。



- 算法设计的过程：

1 清楚问题的输入与输出

2 设计算法的基本步骤

3 给出详细的算法描述

- 算法的输入与输出：

```
template <class T, int MaxSize>
void SeqList<T, MaxSize>::Insert( int i, T item)
```

- 插入算法的基本步骤：

（1）检查参数i的合法性

1 =< i <= L.length + 1

（2）测试当前存储容量

若 L.length >= MaxSize, 则产生溢出错误；

（3）将第i个至第n个元素后移

（4）插入新元素，并将表长加1

插入操作算法：

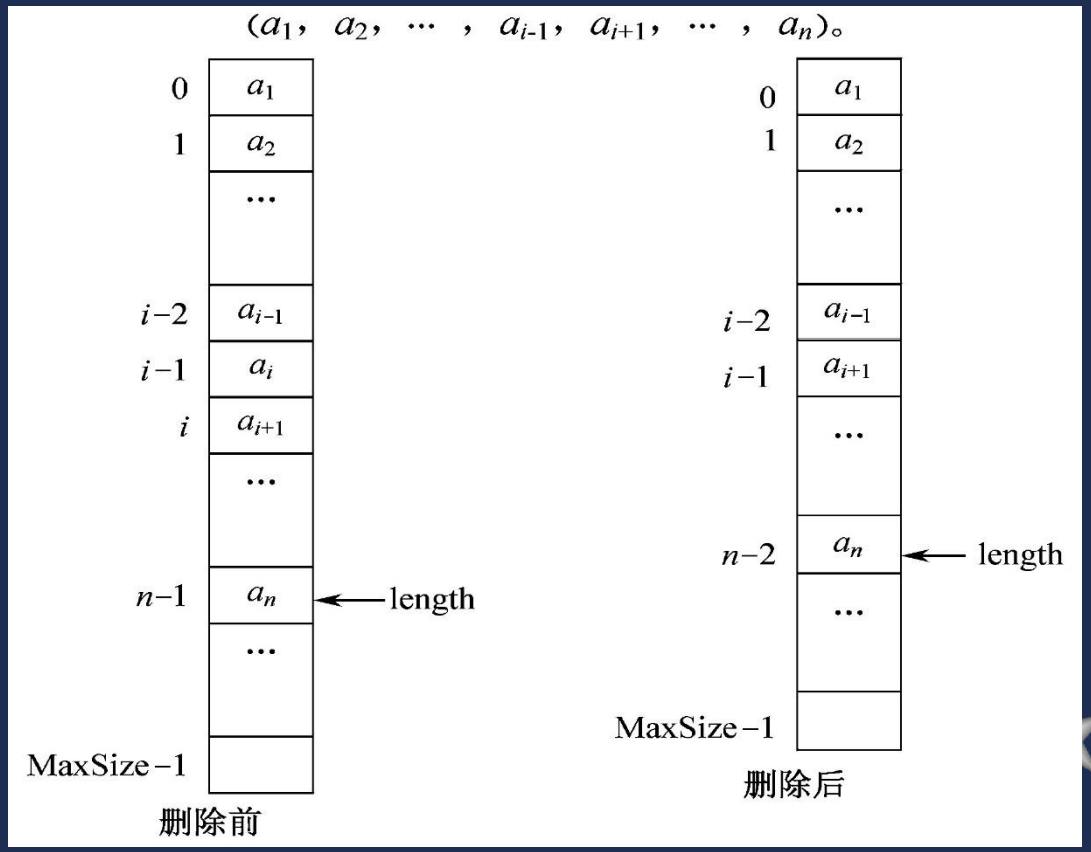
```
template <class T,int MaxSize>
void SeqList<T, MaxSize>::Insert(int i, T item)
{
    if (length>=MaxSize) {cerr<< "上溢"; exit(1);}
    if (i<1 || i>length+1)  {cerr<< "插入位置非法"; exit(1);}
    for (j=length-1; j>=i-1; j--)
        data[j+1]=data[j];
    data[i-1]=item;
    length++;
}
```

如何解决溢出问题？

- 算法时间复杂度的分析：
  - 完成插入所需移动结点的次数不仅依赖于表的长度，而且还与插入位置有关：
    - 最好情况，其时间复杂度$O(1)$；
    - 最坏情况，其时间复杂度为$O(n)$；
    - 由于插入可能在表中任何位置上进行，因此需分析算法的平均复杂度

3. 删除元素操作

--- 将线性表的第i（1<=i<=n)个结点删除，使：长度为n的线性表变为长度为n-1的线性表。



- 算法的输入与输出：

```
template <class T, int MaxSize>
T SeqList<T, MaxSize>::Delete( int i )
```

- 删除算法的基本步骤：

（1）检查参数i

1 =< i <= L.length

（2）测试当前顺序表长度

（3）将第i+1个至第n个元素前移

（4）将表长减1

删除算法：

```
template <class T, int MaxSize>
T SeqList<T, MaxSize>::Delete(int i)
{
    if (length==0)  {cerr<<"下溢"; exit(1);}
    if (i<1 || i>length)  {cerr<<"删除位置非法"; exit(1);}
    x=data[i-1];
    for (j=i; j<length; j++)
        data[j-1]=data[j];
    length--;
    return x;
}
```

4. 按位置查找

```
template <class T, int MaxSize>
T SeqList<T, MaxSize>::Get(int pos)
{
    if (pos<1 || pos>length)
        { cerr<< "查找位置非法";  exit(1); }
    else
        return data[pos-1];
}
```

5. 按值查找

```
template <class T, int MaxSize>
int SeqList<T, MaxSize>::Locate( T item )
{
   for (i=0; i<length; i++)
        if (data[i]==item)
            return i+1 ;
    return 0;
}
```

6. 遍历顺序表

```
template <class T, int MaxSize>
void SeqList<T, MaxSize>::PrintSeqList()
{
    for(i=0; i<length; i++)
        cout<<data[i]<<endl;
}
```

其它操作算法：两个有序顺序表的合并算法

```
void MergeList_Seq( SeqList &La, SeqList &Lb, SeqList &Lc){
    if(La.length+Lb.length> MaxSize)
        { cerr<< "OVERFLOW！ "; exit(1); }
    Lc.length = La.length + Lb.length;
    i=0；  j=0;  k=0;
    while( i<La.length && j<Lb.length )
    {
        if( La.data[i]<Lb.data[j]) Lc[k++] = La[i++];
        else Lc[k++] = Lb[j++];
    }
    while( i<La.length ) Lc[k++] = La[i++];
    while( j<Lb.length ) Lc[k++] = Lb[j++];
}
```

☞ 顺序表的优、缺点：

优点：

（1）无须为表示结点间的逻辑关系而增加额外的存储空间。

（2）可以方便地随机存储表中的任一结点。

缺点：

（1）插入和删除平均须移动一半结点。

2.2.2 链式存储结构

链表

用一组任意的存储单元（可以是无序的）存放线性表的数据元素。

* 无序---可零散地分布在内存中的任何位置上。

例：有线性表：

(bat，cat，eat，fat，hat，jat，lat，mat)

单链表示意图：

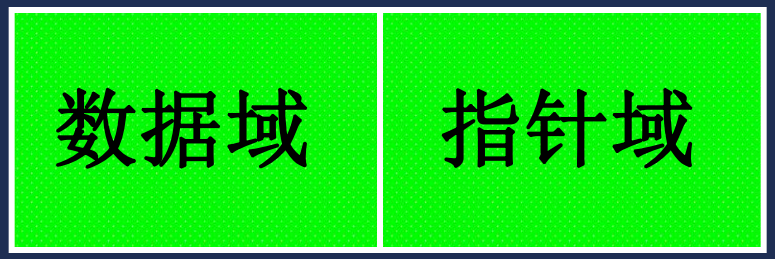
链表的特点：

链表中结点的逻辑次序和物理次序不一定相同。即：逻辑上相邻未必在物理上相邻。

结点之间的相对位置由链表中的指针域指示，

线性表的链式存储结构：

结点的组成：




```
template <class T>
struct Node{
        T data;
        Node<T>* next;
};
```

数据域----表示数据元素自身值。

指针域（链域）----表示与其它结点关系。

通过链域，可将n个结点按其逻辑顺序链接在一起（不论其物理次序如何）。

线性表的链式存储结构：

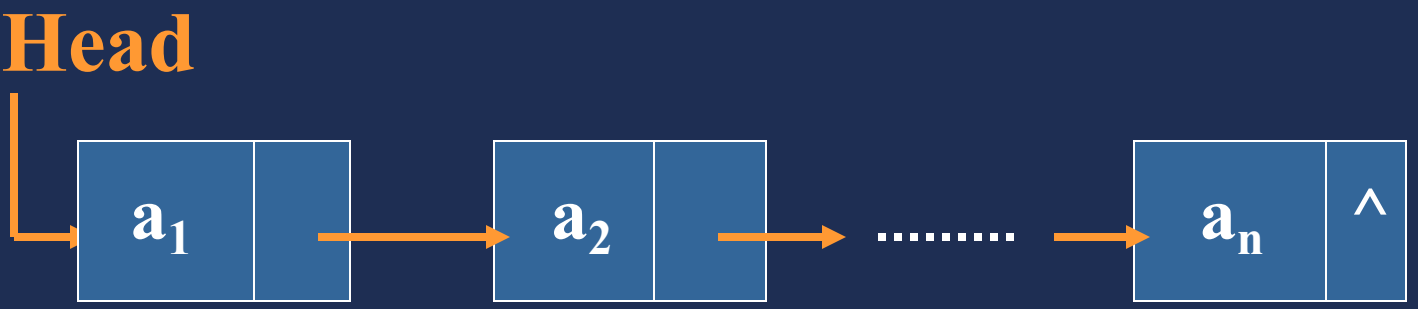
📖**单链表**：----每个结点只有一个链域。

开始结点---（无前驱）用头指针指向之。

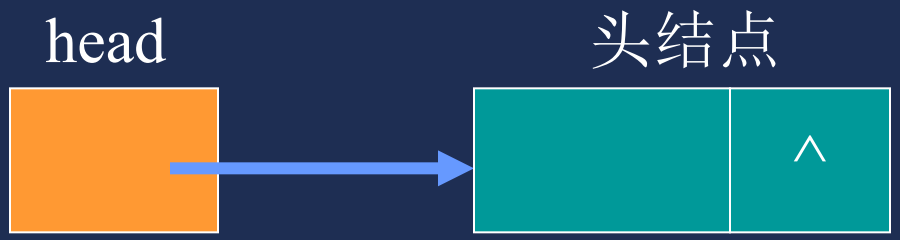
最后一个结点(尾结点)---指针为空（无后继），用^或NULL表示。

表中其它结点---由其前驱结点的指针域指向之。

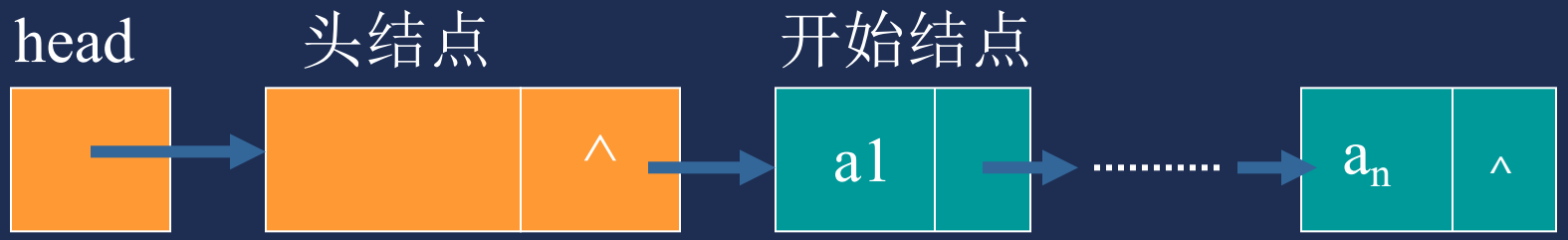
单链表由头指针唯一确定，因此单链表可以用头指针的名字来标识。



☞ 为了给单链表的操作带来方便，一般在单链表的头部增加一个特殊的结点—头结点。



空表



非空表

☞单链表类如何定义？

```
template <class T>
class LinkList{
    Node<T> *head;                  //单链表的头指针
 public:
   LinkList( );                     //建立带头结点的空链表
   LinkList(T a[ ], int n);         //建立有n个元素的单链表
   ~LinkList();                     //析构函数
   int ListLength();                //求单链表的长度
   T Get(int pos);                  //按位查找
   int Locate(T item);              //按值查找
   void PrintLinkList( );           //遍历单链表
   void Insert(int i, T item);      //插入元素
   T Delete(int i);                 //删除结点
};
```

1. 初始化操作—构造函数

- 无参构造函数

```
template <class T>
LinkList<T>:: LinkList( )
{
    head=new Node<T>;
    head->next=NULL;
}
```

1. 初始化操作——构造函数

- 有参构造函数
  - 根据参数提供的n个结点值生成（建立）一个有n个结点的单链表。

☞ 建立单链表的实现

---- ① 头插法建表：

思想：从一个空表开始，不断重复地读入结点数据，生成新结点，将读入数据存放在新结点的数据域中，然后将新结点插入到当前链表的表头上。

注：头插法生成的链表中结点的次序和输入的顺序相反。

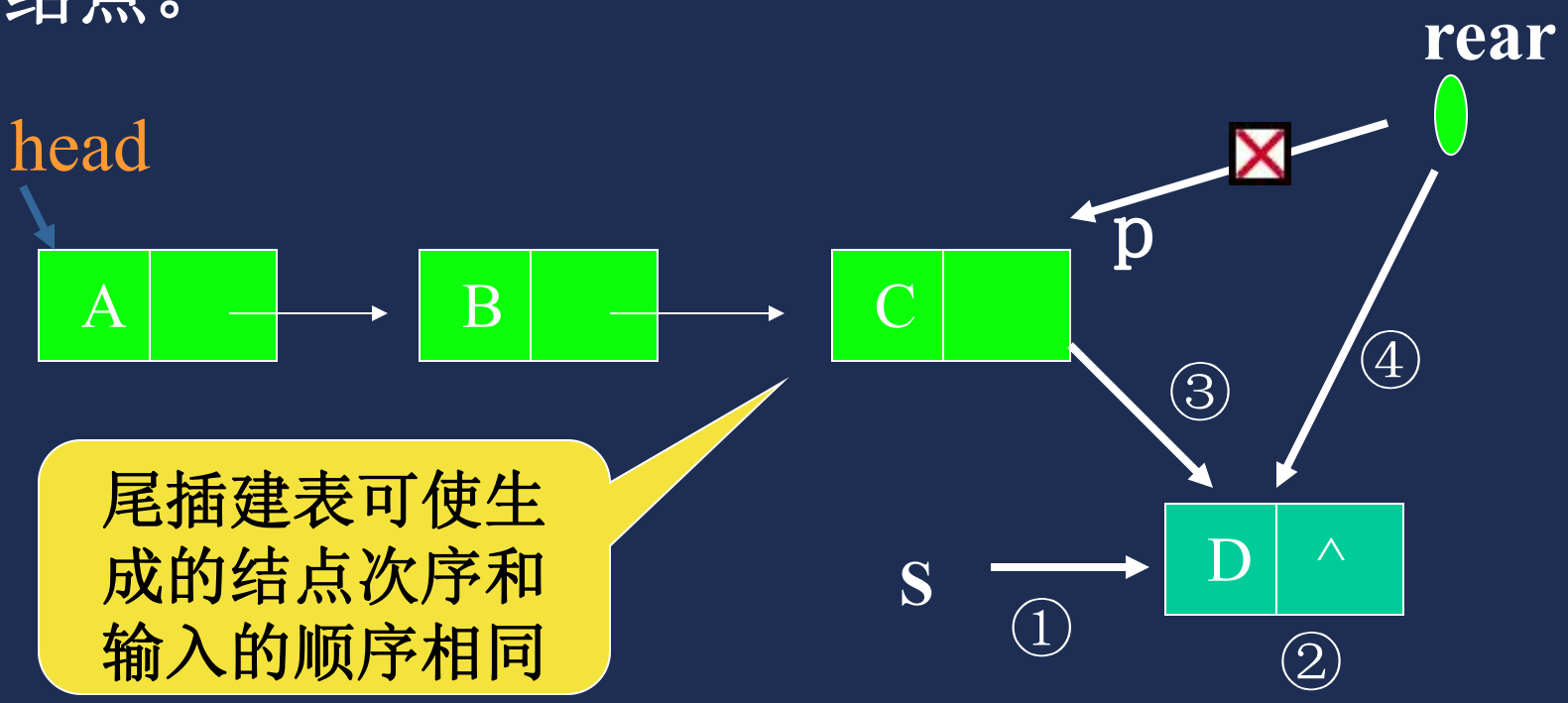
☞ 头插法建表算法:

```
template <class T>
LinkList<T>:: LinkList( T a[ ], int n ){
    head=new Node<T>;    //生成头结点
    head->next=NULL;
    for (i=0; i<n; i++)
    {
        s=new Node<T>;
        s->data=a[i];
        s->next=head->next;
        head->next=s;
    }
}
```

---- ② 尾插建表法：

算法思想：将新结点插入到当前链表的表尾上.

可增加一个尾指针rear，使其始终指向链表的尾结点。



☞ 尾插法建表算法:

```
template <class T>
LinkList<T>:: LinkList(T a[ ], int n){
    head=new Node<T>;    //生成头结点
    rear=head;
    for (i=0; i<n; i++)
    {
        s=new Node<T>;
        s->data=a[i];
        rear->next=s;
        rear=s;
    }
    rear->next=NULL;
}
```

如果单链表中没有设头结点，该算法对吗？

头结点带来以下两个优点？

a、由于开始结点的位置被存放在头结点的指针域中，所以在链表的第一个位置上的操作就和在表的其它位置上的操作一致，无需进行特殊处理；

b、无论链表是否为空，其头指针是指向头结点，因此空表和非空表的处理也就统一了。

2. 求单链表长度

```
template <class T>
int LinkList<T>::ListLength( )
{
    num=0;
    p = head->next;
    while( p )
    {
        p = p->next;
        num++;
    }
    return num;
}
```

3. 按位置查找操作

- 设单链表的长度为n，要查找表中第pos个结点。
- 算法基本步骤如下：
  - 从头结点开始顺链扫描，用指针p指向当前扫描到的结点，用*j*作统计已扫描结点数的计数器，当p扫描下一个结点时，*j*自动加1。
  - P的初值指向开始结点，*j*的初值为1。
  - 若p为空，则抛出查找位置非法异常；否则p指向需查找的元素，返回p所指向结点的数据。

3. 按位置查找操作

```
template <class T>
T  LinkList<T>::Get( int pos )
{
    p=head->next;
    j=1;
    while ( p && j<pos )
    {
          p=p->next;
          j++;
    }
    if ( !p || j>pos ) { cerr<<"查找位置非法";  exit(1); }
    else return p->data;
}
```

4. 按值查找操作

- 在链表中，查找是否有结点等于给定值key的结点，若有的话，则返回首次找到的值为key的结点的存储位置；否则返回NULL.

4. 按值查找操作

```
template <class T>
Node<T>* LinkList<T>::Locate( T key )
{
    p=head->next;
    while( p && p->data!=key )
        p=p->next;
    return p;
}
```

5. 单链表的插入操作

- 插入运算是将值为item的新结点插入到表的第i个结点的位置上，即插入到a_{i-1}与a_i之间。
- 算法步骤：
  - 首先找到a_{i-1}的存储位置p；
  - 然后生成一个数据域为item的新结点s；
  - 调整指针，将新结点s插入到表中。

☞单链表的插入算法：

```
template <class T>
void LinkList<T>::Insert(int i, T item)
{
      p=head; j=0;
      while (p && j<i-1)  //找到第i-1个结点的位置
      {
          p=p->next;
          j++;
       }
       if (!p)  {cerr<<"插入位置非法"; exit(1);}
       else {
           s=new Node<T>;   //生成元素值为item的新结点s
           s->data=item;
           s->next=p->next;   //用后插法将s插入到结点p的后面
           p->next=s;
       }
}
```

插入算法的平均时间复杂度为：$O(n)$

6. 单链表的删除操作

- 删除运算是将表的第i个结点删去。
- 算法步骤：
  - 首先找到a_i的直接前驱结点 a_{i-1}的存储位置p；
  - 令p–>next指向a_i 的直接后继结点；
  - 最后释放结点a_i的空间。

☞单链表的删除算法：

```
template <class T>
T LinkList<T>::Delete( int i )
{
     p=head; j=0;
     while (p && j<i-1)  //查找第i-1个结点
    {
        p=p->next;
        j++;
    }
    if (!p || !p->next)  { cerr<<"删除位置非法"; exit(1);}
    q=p->next; x=q->data;
     p->next=q->next;
     delete q;
     return x;
}
```

7. 单链表的析构函数

- 单链表类中由new运算符生成的结点空间无法自动释放，因此需要利用析构函数将单链表的存储空间加以释放。

☞单链表的析构函数：

```
template <class T>
LinkList<T>:: ~LinkList()
{
    p=head;
    while( p )
    {
        q=p;
        p=p->next;
        delete q;
     }
     head=NULL;
}
```

8. 单链表的其他操作举例

- 例1:已知递增有序的两个单链表L1和L2，要求将L2合并到L1中，且结果链表依然保持递增有序。

  为了实现合并操作，设置3个指针，指针p1和p2分别指向单链表L1和L2中等待比较的数据结点，指针p3始终指向结果有序单链表的表尾。

算法步骤如下：

① 初始化指针，使指针p1指向单链表L1的第一个数据结点，p2指向单链表L2的第一个数据结点，指针p3指向单链表L1的头结点。

② 当p1和p2都不为空时，循环执行以下操作：

如果p1->data小于p2->data，则将p1所指向的结点链入结果有序单链表的表尾，并将指针p1和p3后移。

否则，将p2所指向的结点链入结果有序单链表的表尾，并将指针p2和p3后移。

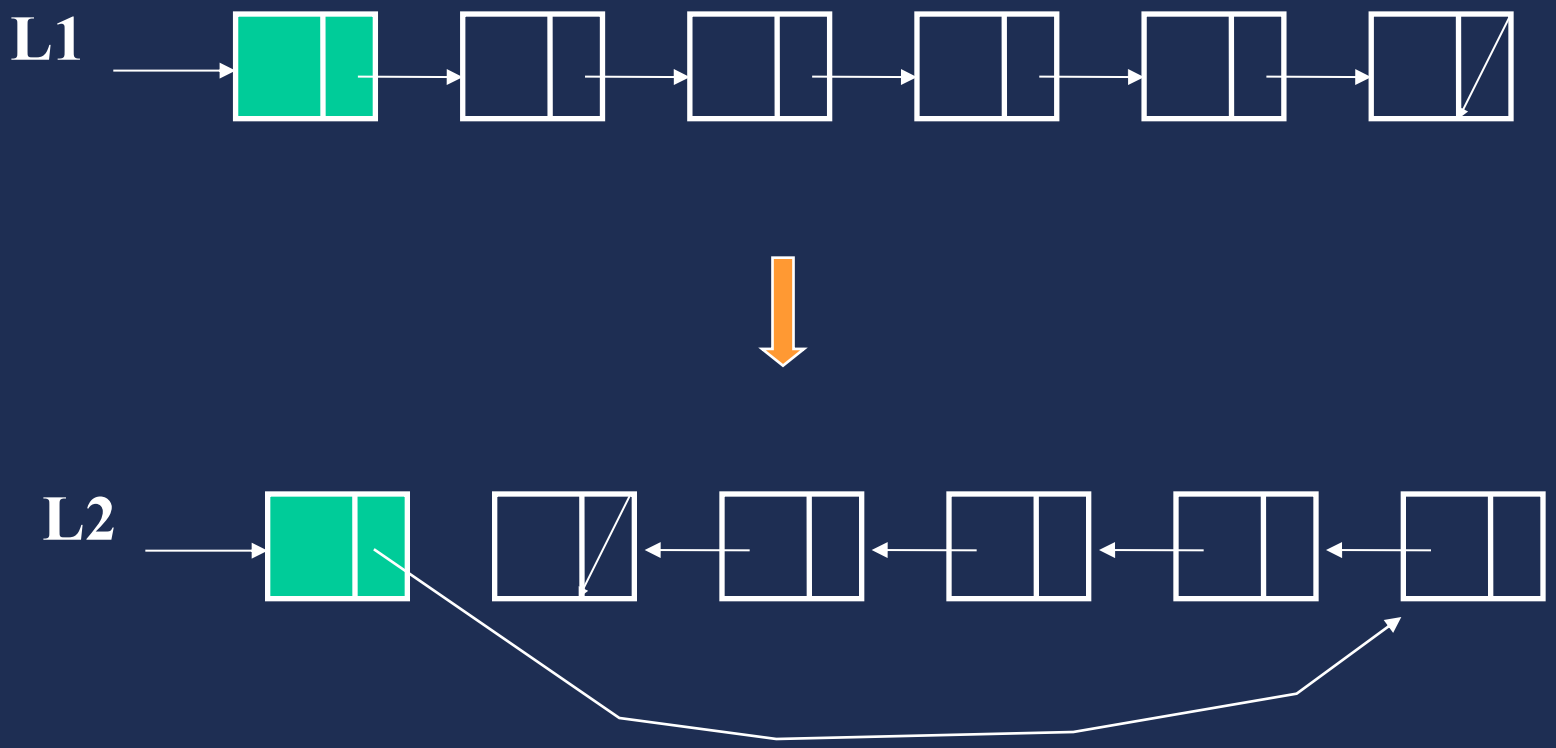
③ 如果p1不为空，则将p1所指向的剩余结点链入结果有序单链表的表尾；如果p2不为空，则将p2所指向的剩余结点链入结果有序单链表的表尾。

```
void MergeLinkList(LinkList &L1, LinkList &L2){
    p1=L1.head->next;  p2=L2.head->next;
    p3=L1.head;
    while( p1!=NULL && p2!=NULL )
     {      if( p1->data < p2->data ){
                p3->next=p1;   p1=p1->next;
                p3=p3->next;
            }
            else{
                p3->next=p2;   p2=p2->next;
                p3=p3->next;
            }
    }
    if( p1 )  p3->next=p1;
    else p3->next=p2;
    delete L2.head;
}
```

- 例2: 有一个单链表, 其结点值以非递减有序排列, 写一个算法删除该单链表中多余的值相同的结点.

```
void  DeleteRepetition ( LinkList& L ){
    p = L.head->next;
    if ( p ) {
      while( p->next ){
          if( p->data==p->next->data ){
              q=p->next;
              p->next=q->next;
              delete (q);
          }
          else p=p->next;
      }
    }
}
```

- **例3：** 写出将单链表L1逆转的算法，逆转后的单链表为L2，L2仍占用L1所占用的存储单元，要求算法使用的附加单元尽可能少。



```
void InvertList( LinkList& L ){
    p = L.head->next;
    if ( !p ) return;
    q = p->next;
    p->next = NULL;
    while ( q ){
        r = q->next;
        q->next = p;
        p = q;
        q = r;
    }
    L.head->next = p;
}
```

方法二：采用头插法建立链表的思想


```
void InvertList( LinkList& L ){
    q = L.head->next;
    L.head->next = NULL;
    while ( q ){
        p = q;
        q=q->next;
        p->next=L.head->next;
        L.head->next = p;
    }
}
```

单链表与顺序表的比较：

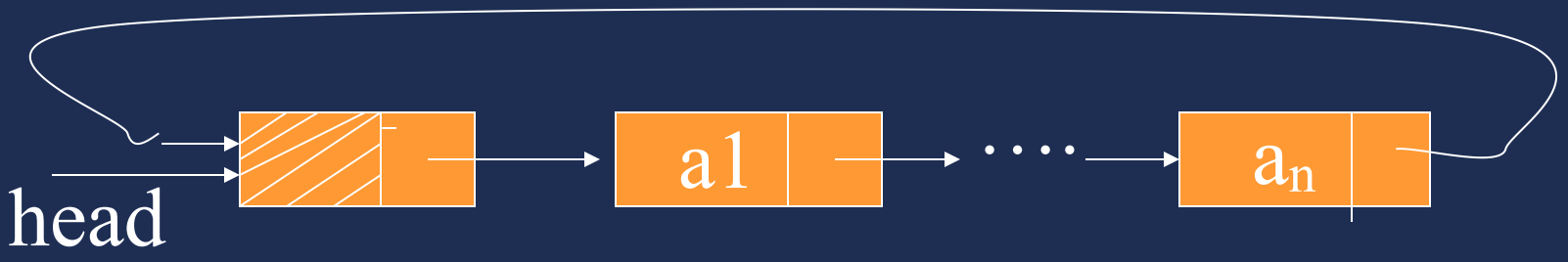
(1) 单链表的存储密度比顺序表低，它多占用了存储空间。但在许多情况下，链式的分配比顺序分配有效，顺序表必须分配足够大的连续的存储空间，而链表可以利用零星的存储单元。

(2) 对于顺序表，可随机访问任一个元素，而在单链表中，需要顺着链逐个进行查找.

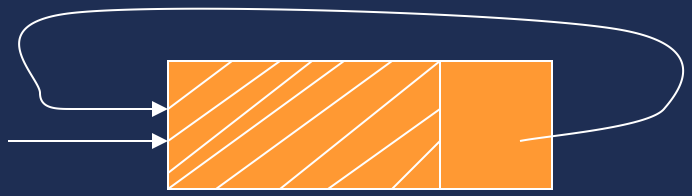
(3) 在单链表里进行插入、删除运算比在顺序表里容易得多。

2.3.2 循环链表

- 循环链表是一种头尾相接的链表。
- 其特点是无须增加存储量，仅对表的链接方式稍作改变，即可使得表处理更加方便灵活。
- 为了使空表和非空表的处理一致，循环链表中也可设置一个头结点。这样，空循环链表仅有一个自成循环的头结点表示。



(1) 非空表



(2)空表

在用头指针表示的单循环链表中，找开始结点a_1的时间是$O(1)$，然而要找到终端结点a_n，则需从头指针开始遍历整个链表，其时间是$O(n)$

- 在很多实际问题中，表的操作常常是在表的首、尾位置上进行，此时头指针表示的单循环链表就显得不够方便.

- 实际中多采用尾指针表示单循环链表。

- 由于循环链表中没有NULL指针，故涉及遍历操作时，其终止条件就不再像非循环链表那样判断p或p->next是否为空，而是判断它们是否等于某一指定指针，如头指针或尾指针等。

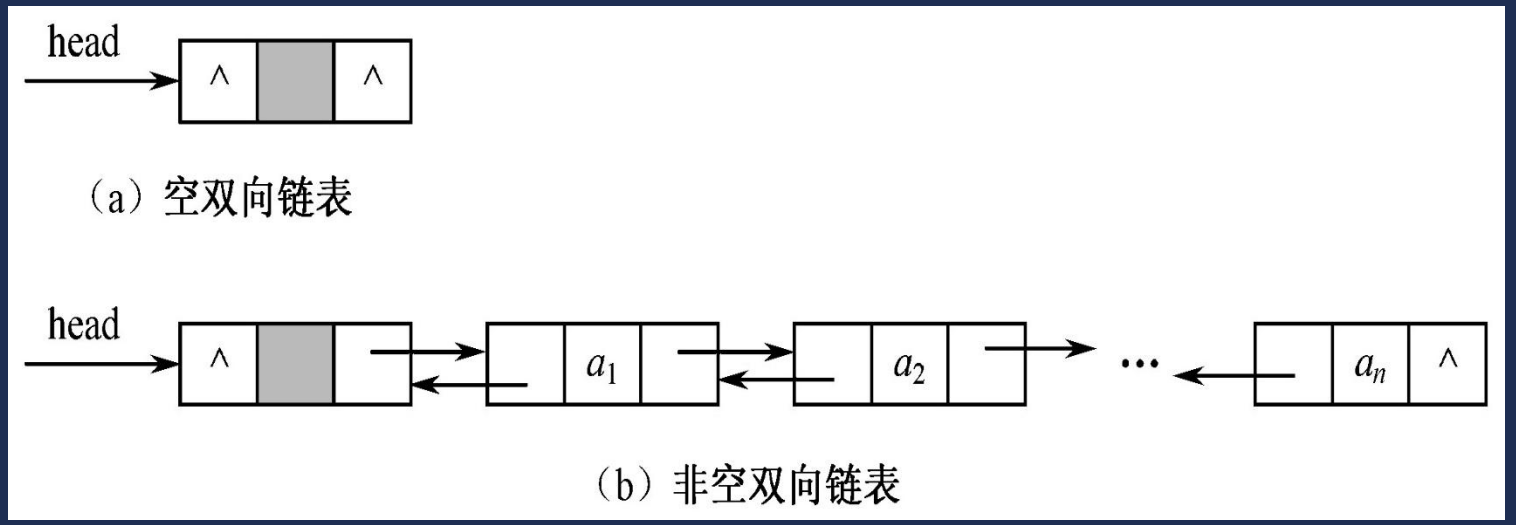
例: 将两个循环链表La和Lb链接成一个循环链表。

```
p = La->next;
La->next=(Lb->next)->next
delete(Lb->next);
Lb->next=p;
```

2.3.3 双链表

- 单链表的缺点：
  - 从某个结点出发，只能顺着指针向后单向查找其它结点。
  - 若要查找当前结点的前趋结点，则需从表头指针出发。

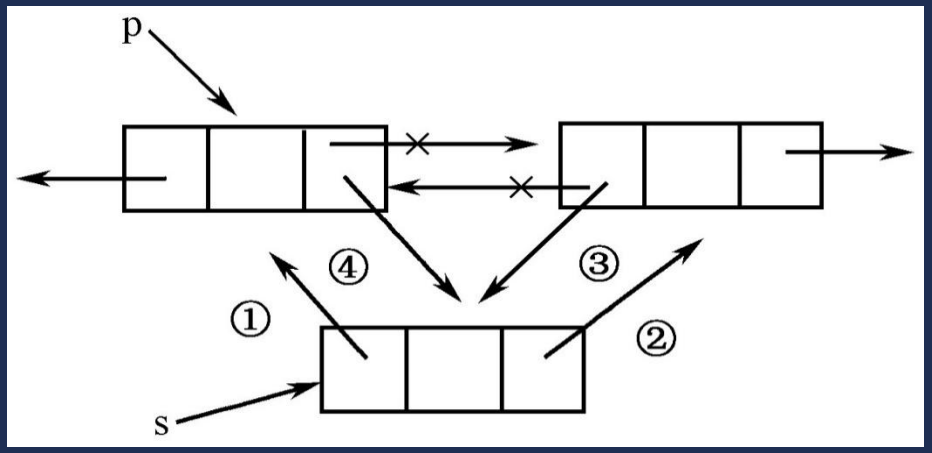
- 双链表
  - 在单链表的每个结点里再增加一个指向其直接前趋的指针域prior。这样形成的链表中有两个方向不同的链。



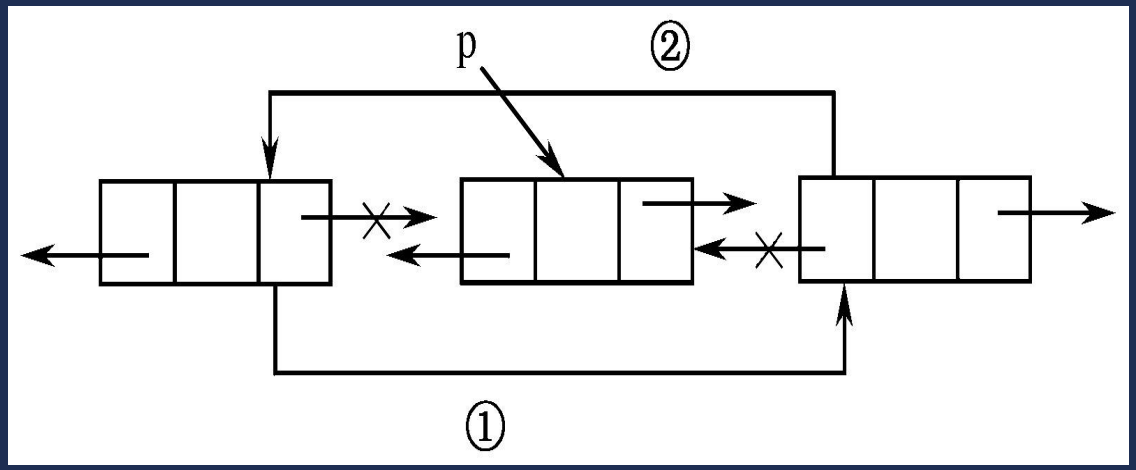
- 和单链表类似，双链表一般也是由头指针唯一确定的。
- 将头结点和尾结点链接起来也能构成循环链表，并称之为双向循环链表。

双链表的基本运算

- (后)插入操作（将结点 s 插入*p之后）



- **删除操作**（在双向链表中删除p结点）



上述两个步骤的顺序是否可以颠倒？

2.4 线性表的应用

- 应用一：求集合的并集
  - 假设有两个集合*A*和*B*，试编写算法求*A*＝*A*∪*B*

- 问题1：集合使用什么数据结构表示？
  - 线性表
- 问题2：采用什么存储结构？
  - 顺序结构
- 问题3：算法设计？

集合类的定义：

```
template <class T, int MaxSize>      //定义类模板
class SeqSet
{
  public:
    SeqSet( );                  //无参构造函数
    SeqSet(T a[], int n);          //有参构造函数
    void Union(SeqSet &setB);     //求并集
    void Print();                 //输出集合各元素
  private:
    T data[MaxSize];        //存放数据元素的数组
    int length;             //集合的长度
    int Locate(T item);              //查找
    void Insert(int i, T item);     //插入元素
};
```

- **Union算法的设计与实现：**

  将存在于顺序表***B***中而不存在顺序表***A***中的数据元素插入到线性表***A***中。

```
template <class T, int MaxSize>
void SeqSet<T, MaxSize>::Union( SeqSet &setB )
{
    for( int i=0; i<setB.length; i++)
        if( Locate(setB.data[i])==0 )
            Insert( length, setB.data[i] );
}
```

应用二： 一元多项式的加法

- 问题描述
  - 已知按升幂表示的两个一元多项式

    $A(x)=a_0+a_1x+a_2x^2+\ldots+a_nx^n$和

    $B(x)=b_0+b_1x+b_2x^2+\ldots+b_mx^m$，
  - 求 $A(x)=A(x)+B(x)$。

应用二： 一元多项式的加法

- 问题1：使用什么数据结构表示一元多项式？
- 问题2：采用什么存储结构合适？
- 问题3：算法设计？

应用二： 一元多项式的加法

- $P_n(x) = p_0+p_1x+p_2x^2+ \ldots +p_nx^n$
- 在计算机中可以用一个线性表P来表示：

$$P = (p_0, p_1, p_2, \cdots, p_n)$$

每一项的指数都隐含在系数p_i的序号里。

- 但如果指数很高并且变化很大时，这种表示方法就不合适了。这时需要把每一项的系数和指数都存储下来，也就是对于每一项都用两个数据项来存储。
即为如下的形式：

$$P = ((p_1,e_1), (p_2,e_2), \ldots, (p_m,e_m))$$

- 如何选择表示一元多项式的存储结构？
  - 顺序结构
  - 链表结构 √
    - 由于对多项式的运算涉及多项式的项的增减变化，所以链表结构合适

单链表结点类型定义：

```
struct PolyNode
{    float  coef;          //系数域
     int    exp;           //指数域
     PolyNode   *next;     //指针域
};
```

一元多项式类的定义：

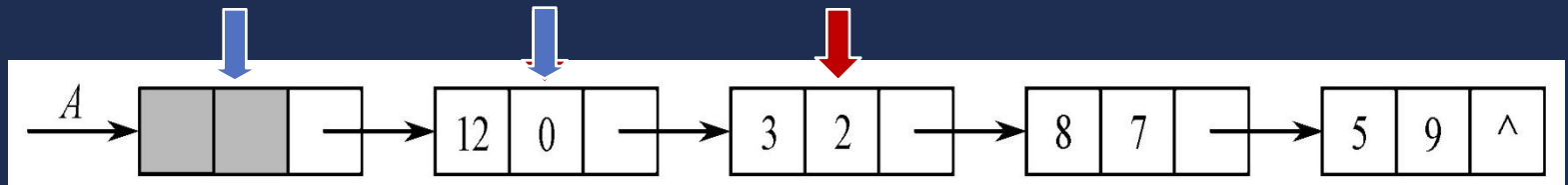
```
class Polynomial
{
   PolyNode *head;        //单链表的头指针
  public:
    Polynomial( );         //建立空的多项式
    Polynomial(PolyNode a[], int n); //建立有n项的多项式
    ~ Polynomial();        //析构函数
    friend void Add(Polynomial &A, Polynomial &B)
    friend void Minus(Polynomial &A, Polynomial &B)
    ……
};
```

实现？

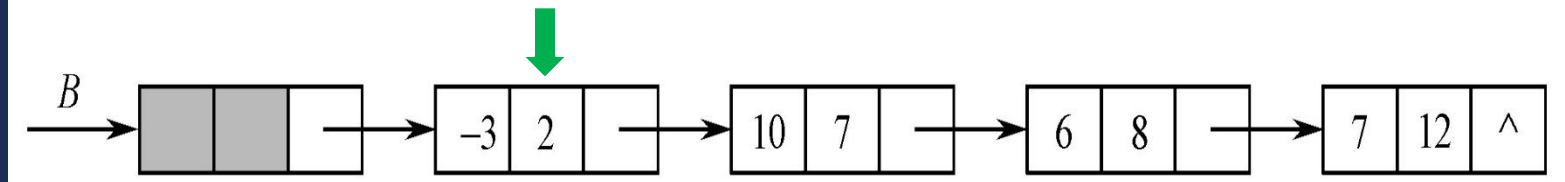
算法的设计？

两个多项式相加的运算规则：

- 对所有指数相同的项，将其对应系数相加，若和不为0，则构成“和多项式”中的一项。
- 将所有指数不相同的项直接链到和多项式中。



（a）一元多项式$A(x)$形成的单链表



（b）一元多项式$B(x)$形成的单链表

算法基本步骤：

初始化；
while (A链和B链都没处理完)
{ if (指针指向当前结点的指数项相同)
{ 系数相加，若和非0，在C链中添加新的结点；
A、B链的指针均前移；}
else
{ 以指数小的项的系数添入C链中的新结点；
指数小的相应链指针前移；}
}
if (A链未处理完)
{ 处理A链；}
if (B链未处理完)
{ 处理B链；}

```
void Add( Polynomial &A, Polynomial &B)
{
        pa=A.head->next; pc=A.head;
        pb=B.head->next;
        while( pa & pb ){
                if(pa->exp==pb->exp){    //指数相等
                    pa->coef=pa->coef+pb->coef;
                    if(pa->coef==0){              //系数相加等于0
                        q=pa;    //删除pa所指结点
                        pa=pa->next;
                        delete q;
                    }
                    else{      //系数相加不等于0
                        pc->next=pa;
                        pc=pa;
                        pa=pa->next;
                    }
```

```
                q=pb;  //删除pb所指结点
                pb=pb->next;
                delete q;
            }  //指数相等情况处理结束
            else if(pa->exp<pb->exp) {
                pc->next=pa;  pc=pa;
                pa=pa->next;
            }
            else{
                pc->next=pb;  pc=pb;
                pb=pb->next;
            }
        }
        if(pa) pc->next=pa;
        else pc->next=pb;
        delete B.head;
}
```