第二章 线性表

第二章 线性表

2.1 线性表的概念和类型定义

2.2 线性表的顺序存储和实现

2.3 线性表的链式存储和实现

引例：多项式的表示

一元多项式及其运算

一元多项式： $f(x)=a_0+a_1x+\cdots+a_{n-1}x^{n-1}+a_nx^n$

主要运算： 多项式相加、相减、相乘等

【分析】如何表示多项式? 多项式的关键数据：

- 多项式项数**n**
- 各项系数a_i及指数**i**

方法1：顺序存储结构直接表示

数组各分量对应多项式各项：

a[i]：项x^i的系数a_i

例 如： $f(x)=4x^5-3x^2+1$

表示成：

下标i	0	1	2	3	4	5	……
a[i]	**1**	**0**	**–3**	**0**	**0**	**4**	**……**
	1		$-3x^2$			$4x^5$	

问题：两个多项式相加如何实现?

问题：如何表示多项式$x+3x^{2000}$？

方法2：顺序存储结构表示非零项

每个非零项$a_i x^i$涉及两个信息：系数a_i和指数i
可以将一个多项式看成是一个(a_i, i)二元组的集合。

用结构数组表示： 数组分量是由系数a_i、指数i组成的结构，对应一个非零项

例如： $P_1(x) = 9x^{12} + 15x^8 + 3x^2$ **和** $P_2(x) = 26x^{19} - 4x^8 - 13x^6 + 82$

下标i	0	1	2	……
系数a^i	9	15	3	–
指数i	12	8	2	–

(a) $P_1(x)$

下标i	0	1	2	3	……
系数a^i	26	–4	–13	82	–
指数i	19	8	6	0	–

(b) $P_2(x)$

按指数大小有序存储！

方法2：顺序存储结构表示非零项

相加过程：从头开始，比较两个多项式当前对应项的指数

P1: (9,12), (15,8), (3,2)

P2: (26,19), (-4,8), (-13,6), (82,0)

P3: (26,19) (9,12) (11,8) (-13,6) (3,2) (82,0)

$$P_3(x) = 26x^{19} + 9x^{12} + 11x^{8} - 13x^{6} + 3x^{2} + 82$$

方法3：链表结构存储非零项

链表中每个结点存储多项式中的一个非零项，包括系数和指数两个数据域以及一个指针域

coef	expon	link

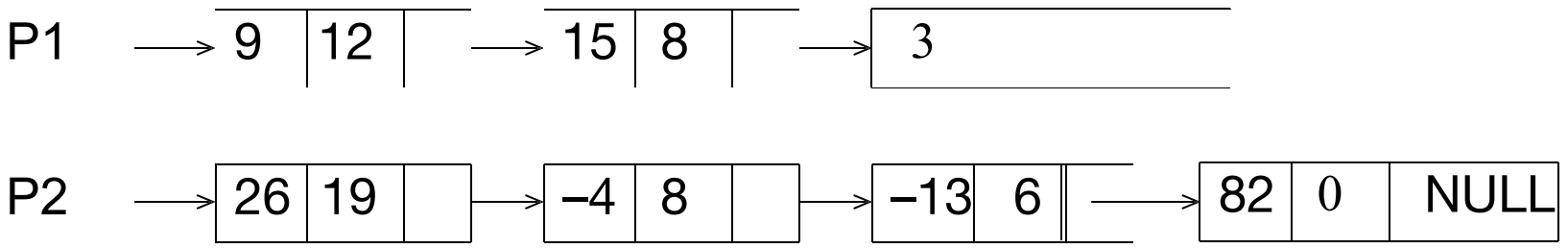
```
typedef struct PolyNode *Polynomial;
typedef struct PolyNode {
        int coef;
        int expon;
        Polynomial link;
}
```

例如：

$$P_1(x) = 9x^{12} + 15x^8 + 3x^2$$

$$P_2(x) = 26x^{19} - 4x^8 - 13x^6 + 82$$

链表存储形式为：



多项式表示问题的启示：

1. 同一个问题可以有不同的存储方法
2. 有一类共性问题：有序线性序列的组织和管理

2.1 线性表的概念和类型定义

- 线性表的概念
  - 定义 n（≥ 0）个数据元素的有限序列，记作 $(a_1, a_2, \cdots, a_n)$。

    a_i 是表中数据元素，n 是表长度。

    a_i 是 a_{i+1} 的直接前驱元素，

    a_i 是 a_{i-1} 的直接后继元素。
  - 遍历：逐项访问

    从前向后

2.1 线性表的概念和类型定义

- 线性表的特点
  - 存在一个唯一的称做“第一个”的数据元素；
  - 存在一个唯一的称做“最后一个”的数据元素；
  - 除第一个元素外，其他每一个元素有一个且仅有一个直接前驱；
  - 除最后一个元素外，其他每一个元素有一个且仅有一个直接后继。

抽象数据类型线性表的类型定义

ADT List {

数据对象 D={a_i | a_i∈ElemSet,i=1,2,..,n n>=0 }

数据关系 R={ 〈a_i，a_{i+1}〉 | a_i∈D, i=2,..,n }

基本操作

InitList(&L)

操作结果： 构造一个空的线性表L。

DestoryList（&L)

初始条件：线性表L已经存在。

操作结果：销毁线性表L。

…

}

线性表的基本操作：

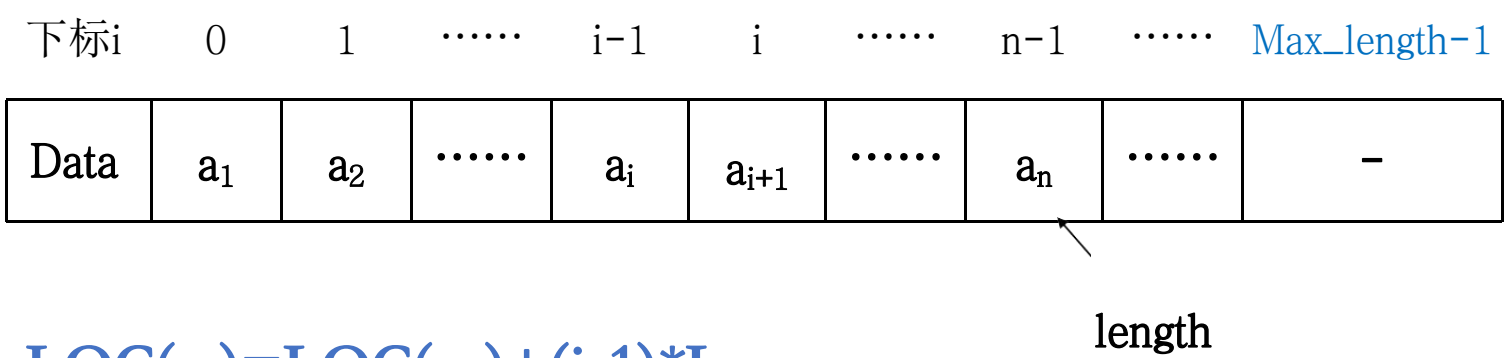
1. 初始化线性表L InitList(&L)
2. 销毁线性表L DestoryList(&L)
3. 清空线性表L ClearList(&L)
4. 求线性表L的长度 ListLength(L)
5. 判断线性表L是否为空 IsEmpty(L)
6. 获取线性表L中的某个数据元素内容 GetElem(L,i, & e)
7. 检索值为e的数据元素 LocateELem(L, & e)
8. 返回线性表L中e的直接前驱元素 PriorElem(L, & e)
9. 返回线性表L中e的直接后继元素 NextElem(L, & e)
10. 在线性表L中插入一个数据元素 ListInsert(&L,i, & e)
11. 删除线性表L中第i个数据元素 ListDelete(&L,i, & e)

2.2 线性表的顺序存储和实现

顺序表实现*方法一*使用静态分配

数据定义：

```
#define Max_length 10
typedef struct{
        ElemType data[Max_length];
        int length;
}SqList;
```



$LOC(a_i)=LOC(a_1)+(i-1)*L$

顺序存储结构的特点

（1）利用数据元素的存储位置表示线性表中相邻数据元素之间的前后关系，即线性表的逻辑结构与存储结构（物理结构）一致；

（2）在访问线性表时，可以利用上面的数学公式，快速地计算出任何一个数据元素的存储地址。可以粗略地认为，访问每个数据元素所花费的时间相等。这种存取元素的方法被称为*随机存取法*。

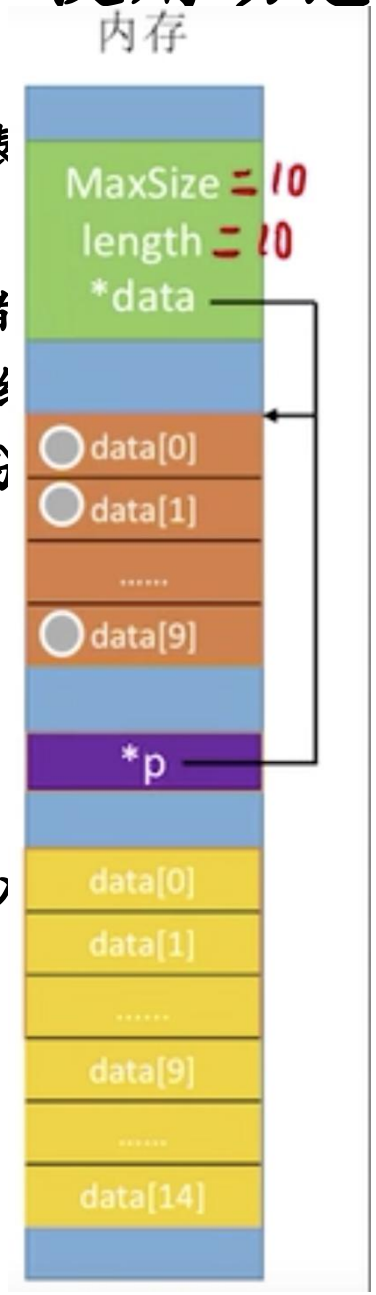
顺序表的基本操作——初始化

```
#include <stdio.h>
#define Max_length 10
typedef struct{
        ElemType dada[Max_length];
        int length;
}SqList;
 int main(){
     SqList L;
     InitList(L);
     //...后续操作 IncreaseSize(L,5);
     return  0;
 }
```

```
void InitList(SqList &L){
    for(int i=0;i<Max_length;i++)
        L.data[i]=0;
    L.length=0;
}
```

顺序表实现方法二使用动态分配

```
#include <stdlib.h>
#define InitSize 10   //顺序表
typedef struct {
    ElemType *data;   //存储
        int length;     //当前
        int listsize;   //当前分
 } SqList;
int main(){
    SqList L;
    InitList(L);
    //...往顺序表中随便插入
    IncreaseSize(L,5);
    return  0;
}
```




```
void InitList(SqList &L){
        //用malloc函数申请一片连续的存储空间
        L.data=(ElemType *)malloc(InitSize * sizeof(ElemType))
        L.length=0;
        L.listSize=InitSize;}

   //增加动态数组的长度
void IncreaseSize(SqList &L, int len){
    int *p=L.data;
    L.data=(ElemType *)malloc((L.listSize+len) * sizeof(ElemType))
    for(int i=0;i<L.length;i++){
        L.data[i]=p[i];
    }
    L.listSize=L.listSize+len;
    free(p);
}
```

顺序表的基本操作——插入

（第$i\ (1 \leqslant i \leqslant n+1)$个位置上插入一个值为X的新元素）

下标 i	0	1	……	i-1	i	……	n-1	……	MAXSIZE-1
Data	a_1	a_2	……	a_i	a_{i+1}	……	a_n	……	-

length

先移动，再插入

下标 i	0	1	……	i-1	i	i+1	……	n	……	SIZE -1
Data	a_1	a_2	……	X	a_i	a_{i+1}	……	a_n	……	-

length

顺序表的基本操作——插入

```
#define Max_length 10
typedef struct{
        ElemType dada[Max_length];
      int length;
}SqList;
int main(){
    SqList L;
    InitList(L);
    //...往顺序表中随便插入几个元素
    IncreaseSize(L,3,3);
    return  0;
}
```

```
void ListInsert(SqList &L, int i, int e){
    for(int j=L.length;j>=i;j--)
          L.data[j]=L.data[j-1];
    L.data[i-1]=e;
    L.length++;
}
```

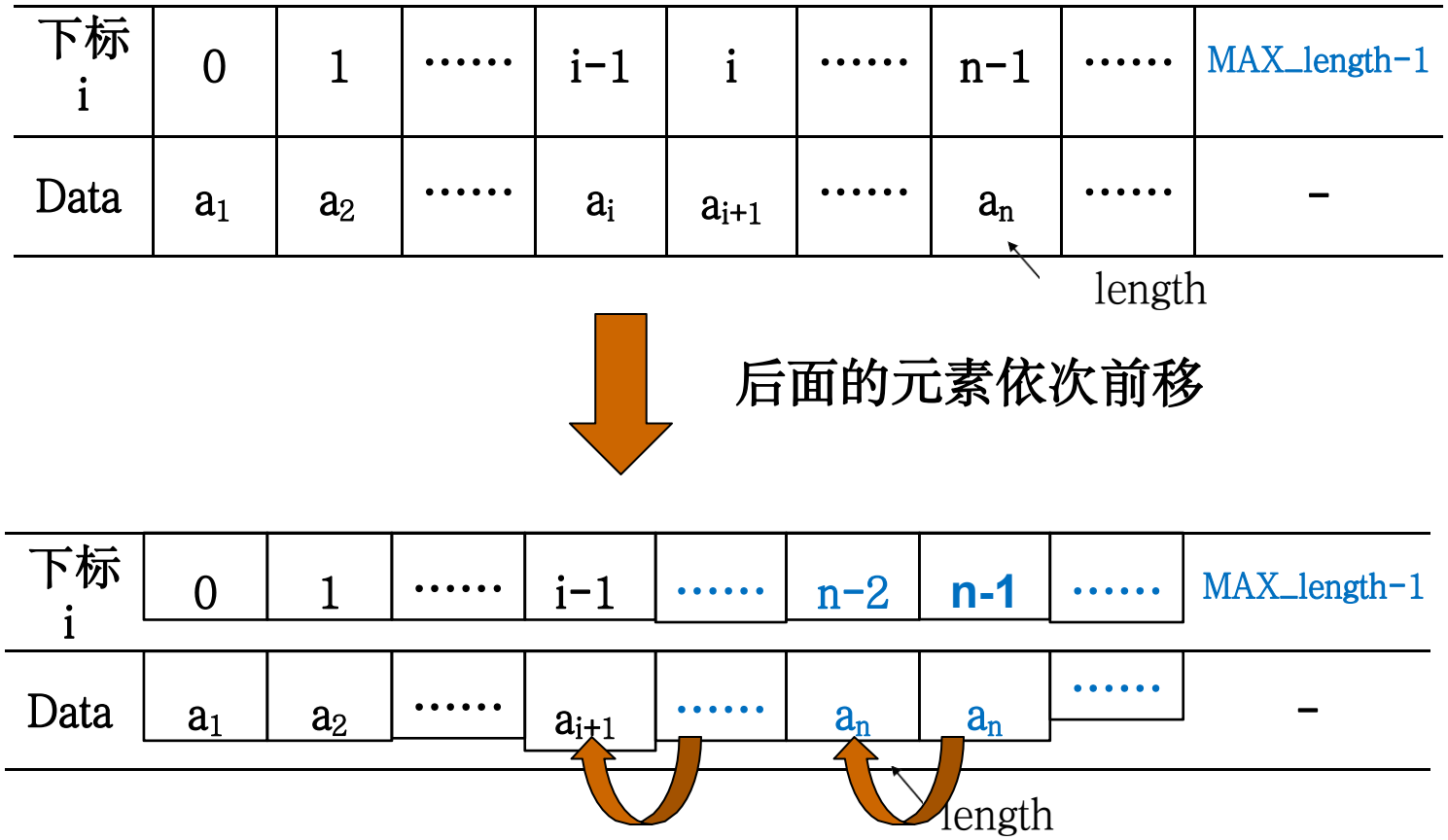
```
#define Max_length 10
typedef struct{
        ElemType dada[Max_length];
       int length;
}SqList;

void ListInsert(SqList &L, int i, int e){
    if(i<1||i>L.length+1)
        return false;
    if(L.length>=Max_length)
        return false;
    for(int j=L.length;j>=i;j--)
        L.data[j]=L.data[j-1];
    L.data[i-1]=e;
    L.length++;
    return  true;
}
```

结论：好的算法，应该具有“健壮性”。能处理异常，并给出反馈。

插入操作时间复杂度

```
#define Max_length 10
typedef struct{
        ElemType dada[Max_length];
       int length;
}SqList;

void ListInsert(SqList &L, int i, int e){
    if(i<1||i>L.length+1)
        return false;
    if(L.length>=Max_length)
        return false;
    for(int j=L.length;j>=i;j--)
        L.data[j]=L.data[j-1];
    L.data[i-1]=e;
    L.length++;
    return  true;
}
```

删除（删除表的第$i\ (1 \leqslant i \leqslant n)$个位置上的元素）



顺序表的基本操作——删除

```
#define Max_length 10
typedef struct{
        ElemType dada[Max_length];
       int length;
}SqList;
int main(){
    SqList L;
    InitList(L);
    //...往顺序表中随便插入几个元素
    int e=-1;
    if (ListDelete(L,3,e))
        printf("已删除第3个元素，值为=%d\n",e)
    else
        printf("位序i不合法\n")
```

```
void ListDelete(SqList &L, int i, int &e){
    if(i<1||i>L.length)
        return false;
    e=L.data[i-1];
    for(int j=i;j<L.length;j++)
        L.data[j-1]=L.data[j];
    L.length--;
return  true;
}
```

顺序表的基本操作——按位查找

```
#define Max_length 10
typedef struct{
        ElemType dada[Max_length];
      int length;
}SqList;

 ElemType GetElem(SqList L,int i){
     return  L.data[i-1];
 }
```

顺序表的基本操作——按值查找

```
#define Max_length 10
typedef struct{
        ElemType dada[Max_length];
      int length;
}SqList;
int LocateElem(SqList L,ElemType e){
    for(int i=0;i<L.length;i++)
        if(L.data[i]==e)
            return i+1;
    return 0;
}
```

作业1：

1、若线性表最常用的操作是存取第i个元素及其前驱和后继元素的值，为了提高效率，应采用

（）的存储方式。

A. 单链表　　B. 双向链表

C. 单循环链表

D. 顺序表

作业2：

2、若线性表最常用的操作是存取任意指定序号的元素和在最后进行插入删除操作，则利用

（）存储方式可以节省时间。

A. 顺序表　　B. 双链表

C. 带头节点的双循环链表

D. 单循环链表

作业3：

3、在n个元素的线性表的数组表示中，以下时间复杂度为O（1）的操作是（ ）。

①访问第i个结点（1≤i≤n）和求第i个节点的直接前驱（2≤i≤n）

②在最后一个结点后插入一个新的结点

③删除第1个节点

④在第i个结点后插入一个结点（1≤i≤n）

A. ①　　B. ② ③

C. ① ②　　D. ① ② ③

作业4:

综合应用题1:

设计一个高效的算法，将顺序表的所有元素逆置，要求算法的空间复杂度为O（1）。

作业5：

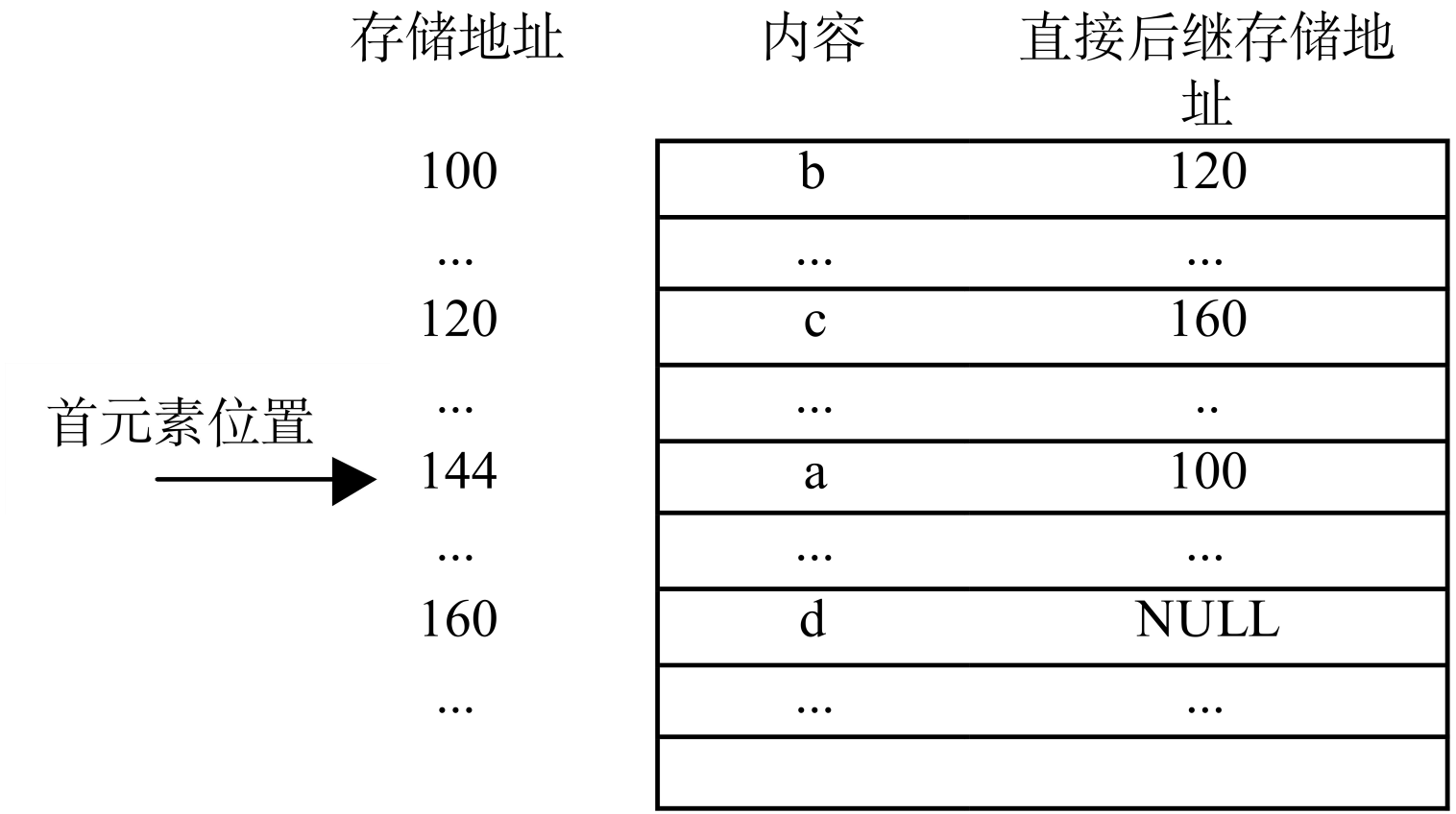
综合应用题2：

长度为n的顺序表L，编写一个时间复杂度为O（n）、空间复杂度为O（1）的算法，该算法删除线性表中所有值为x的数据元素。

2.3 线性表的链式存储和实现

- 线性表的顺序存储优点：
  - 它是一种简单、可随机存取。
- 线性表的顺序存储缺点：
  - 在做插入或删除元素的操作时，会产生大量的数据元素移动；
  - 对于长度变化较大的线性表，要一次性地分配足够的存储空间，但这些空间常常又得不到充分的利用；
  - 线性表的扩充比较麻烦。

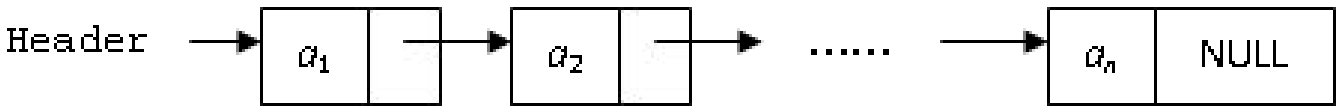
线性表的链式存储



线性表的链式存储实现

不要求逻辑上相邻的两个元素物理上也相邻；
通过“链”建立起数据元素之间的逻辑关系。

插入、删除不需要移动数据元素，只需要修改“链”。




```
struct LNode{
    ElementType Data;
    struct LNode *Next;
};
```

```
struct  LNode *p=(struct LNode *)malloc(sizeof(struct  LNode));
```

typedef <数据类型> <别名>

线性表的链式存储实现

```
typedef struct LNode{
    ElementType Data;
    struct LNode *Next;
} LNode, *LinkList;
```

要表示一个单链表时，只需要一个头指针L，指向单链表的第一个节点

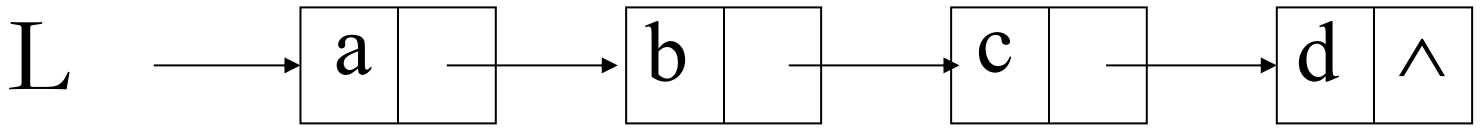
LNode *L; //声明一个指向单链表第一个节点的指针

LNode *p=(LNode *)malloc(sizeof(LNode));

LinkList L; //声明一个指向单链表第一个节点的指针

```
typedef struct LNode{
    ElementType Data;
    struct LNode *Next;
} LNode, *LinkList;

  //按位查找，返回第i个元素（带头节点）
  LNode * GetElem(LinkList L, int i){
        if(i<0)
          return NULL;
    LNode *p;
    p = L;
    int j = 0;
    while (p && j < i) {  // 寻找第i个结点
      p = p->next;
          ++j;
      return p;
    }
```

不带头节点的单链表



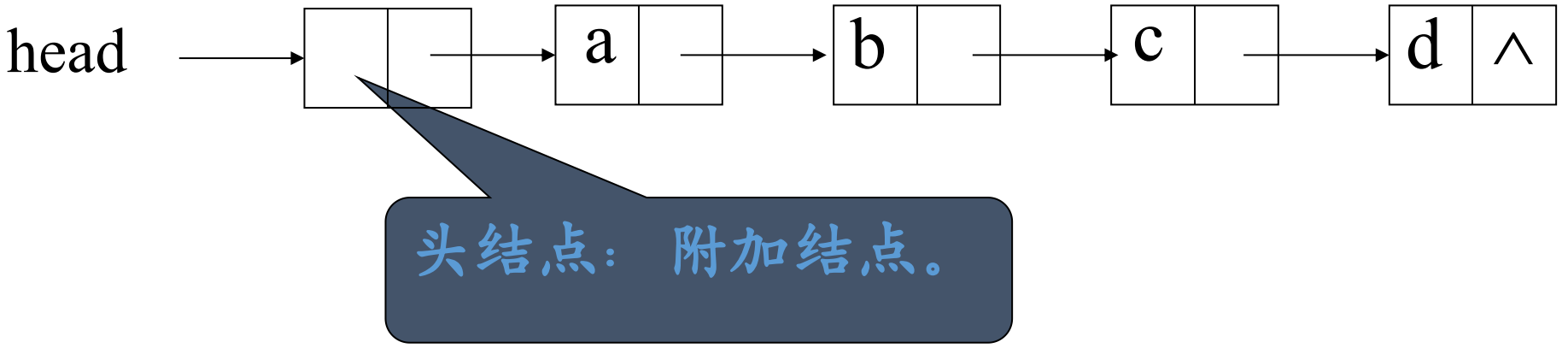
不带头节点的单链表

```
typedef struct LNode{
    ElementType Data;
    struct LNode *Next;
} LNode, *LinkList;
```

```
bool InitList(LinkList &L){
    L=NULL;
    return true;
}
void test(){
    LinkList L; //声明一个指向单链表的指针
    InitList(L);
    //.....
}
```

```
bool Empty(LinkList L){
    if (L==NULL)
        return true;
    else
        return false;
```

带头节点的单链表



带头节点的单链表

```
typedef struct LNode{
    ElementType Data;
    struct LNode *Next;
} LNode, *LinkList;

bool InitList(LinkList &L){
    L=(LNode *)malloc(sizeof(LNode));
    if (L==NULL)
        return false;
    L->next=NULL;
    return true;
}
void test(){
    LinkList L; //声明一个指向单链表的指针
    InitList(L);
    //.....}
```

```
bool Empty(LinkList L){
    if (L->next==NULL)
        return true;
    else
        return false;
```

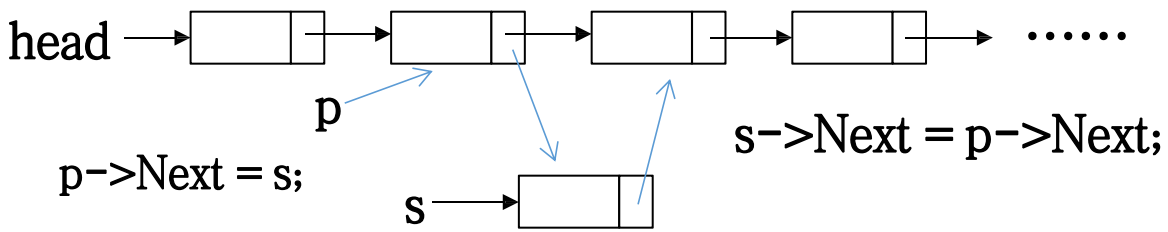
按位序插入（带头节点）

ListInsert（&L, i, e）

1　**先构造一个新结点，用s指向；**

2　**再找到链表的第*i-1*个结点，用p指向；**

3　**然后修改指针，插入结点（p之后插入新结点是 s）**

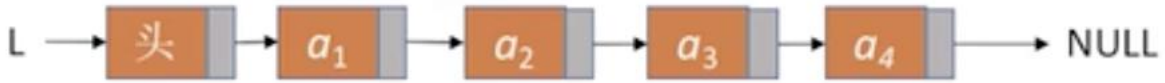


思考：修改指针的两个步骤如果交换一下，将会发生什么？

按位序插入（带头节点）

```
bool ListInsert(LinkList &L, int i, ElemType e) {
  LNode *p, *s;
  p = L;  int j = 0;
  while (p && j < i-1) {  // 寻找第i-1个结点
    p = p->next;
   ++j;
  }
  if (!p || j > i-1) return false;     // i小于1或者大于表长
  s = ( LNode *)malloc(sizeof(LNode));  // 生成新结点
  s->data = e;
  s->next = p->next;      // 插入L中
  p->next = s;
  return true;
}
```

```
typedef struct LNode{
      ElementType Data;
      struct LNode *Next;
} LNode, *LinkList;
```



时间复杂度：

按位序插入（不带头节点）

ListInsert（&L, i, e）

1、找到第i-1个结点，将新结点插入其后

2、不存在“第0个”结点，因此i=1时需要特殊处理

按位序插入（不带头节点）

```
bool ListInsert(LinkList &L, int i, ElemType e) {
  LNode *p, *s;
  if (i==1){
       s = ( LNode *)malloc(sizeof(LNode)); // 生成新结点
       s->data = e;
       s->next = L;      // 插入L中
       L = s;
       return true;
  }
   p = L;  int j = 1;
  while (p && j < i-1) {  // 寻找第i-1个结点
       p = p->next;
       ++j;
  }
  if (!p || j > i-1) return false;      // i小于1或者大于表长
  s = ( LNode *)malloc(sizeof(LNode)); // 生成新结点
  s->data = e;
  s->next = p->next;      // 插入L中
  p->next = s;
  return true;
}
```

```
typedef struct LNode{
      ElementType Data;
      struct LNode *Next;
} LNode, *LinkList;
```

按位序删除

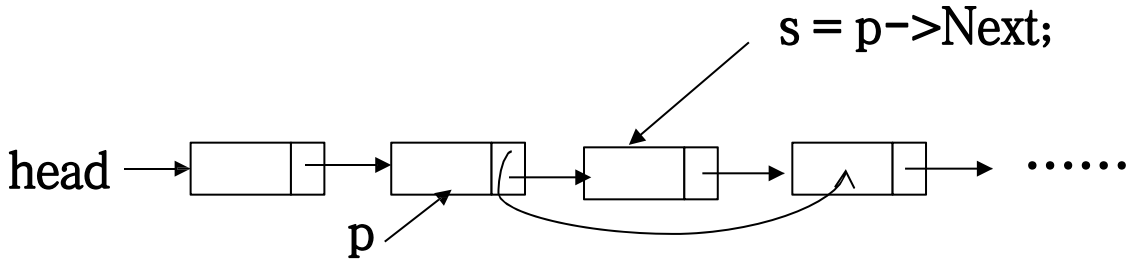
ListDelete(&L,i,&e)

1　　先找到链表的第*i-1*个结点，用p指向；

2　　再用指针s指向要被删除的结点（p的下一个结点）；

3　　然后修改指针，删除s所指结点;

4　　最后释放s所指结点的空间:free(s)。

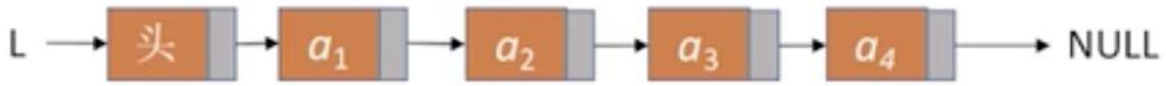


p->Next = s->next;

按位序删除（带头节点）

```
bool ListDelete(LinkList &L, int i, ElemType &e) {
  if(i<1)
      return false;
  LNode *p;
  p = L;  int j = 0;
  while (p && j < i-1) {  // 寻找第i-1个结点
       p = p->next;
      ++j;
  }
  if (!p || j > i-1) return false;      // i小于1或者大于表长
  if(p->next==NULL)
              return false;
  LNode  *q=p->next;
  e=q->data;
   p->next=q->next ;      // 插入L中
  free(q);
  return true;
}
```

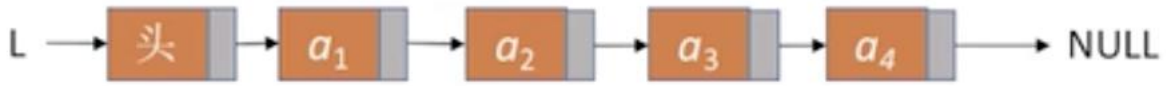
```
typedef struct LNode{
      ElementType Data;
      struct LNode *Next;
} LNode, *LinkList;
```



查找（带头节点）

//按位查找，返回第i个元素（带头节点）
LNode * GetElem(LinkList L, int i)
//按值查找，返回值为e的元素（带头节点）
LNode * LocateElem(LinkList L, ElemType e)

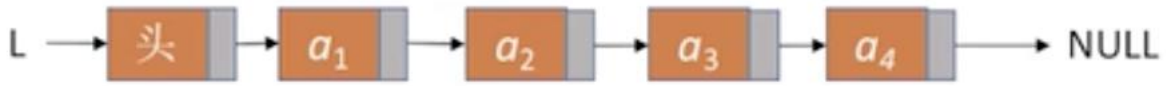
```
typedef struct LNode{
    ElementType Data;
    struct LNode *Next;
} LNode, *LinkList;
```



查找（带头节点）

```
//按位查找，返回第i个元素（带头节点）
LNode * GetElem(LinkList L, int i){
   if(i<0)
      return NULL;
   LNode *p;
   p = L;
   int j = 0;
  while (p && j < i) {  // 寻找第i个结点
       p = p->next;
      ++j;
   }
   return p;
  }
```

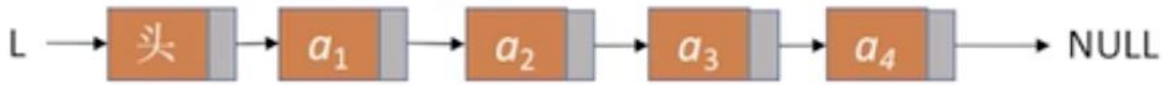
```
typedef struct LNode{
      ElementType Data;
      struct LNode *Next;
} LNode, *LinkList;
```



查找（带头节点）

```
//按值查找，返回值为e的元素（带头节点）
LNode * LocateElem(LinkList L, ElemType e) {
  LNode *p=L->next;
   while (p && p->data != e)
         p = p->next;
  return p;
  }
```

```
typedef struct LNode{
    ElementType Data;
    struct LNode *Next;
} LNode, *LinkList;
```



建立单链表

- 尾插法建立单链表
- 头插法建立单链表

尾插法建立单链表

```
LinkList  TailInsert(LinkList &L, int n) {
  // 逆位序输入（随机产生）n个元素的值，建立带表头结点的单链线性表L
    LNode *p;  int i;
    L = (LinkList)malloc(sizeof(LNode));
    LNode *r=L;            // 先建立一个带头结点的单链表
    for (i=n; i>0; --i) {
        p = (LNode *)malloc(sizeof(LNode));  // 生成新结点
        p->data = i;                  // 为数据域赋值
        r->next = p;    r = p;    // 插入到表头
    }
    r->next=NULL;
    return L;
}
```

头插法建立单链表

```
LinkList  TailInsert(LinkList &L, int n) {
  // 逆位序输入（随机产生）n个元素的值，建立带表头结点的单链线性表L
    LNode *p;  int i;
    L = (LinkList)malloc(sizeof(LNode));
    L->next = NULL;              // 先建立一个带头结点的单链表
    for (i=n; i>0; --i) {
        p = (LinkList)malloc(sizeof(LNode));  // 生成新结点
        p->data = i;                  // 为数据域赋值
        p->next = L->next;    L->next = p;    // 插入到表头
    }
     return L;
}
```

综合应用题

1、设计一个递归算法，删除不带头结点的单链表L中所有值为x的结点

综合应用题

2、在带头结点的单链表L中，删除所有值为x的节点，并释放其空间，假设值为x的结点不唯一，试编写算法以实现上述操作。

综合应用题

3、试L为带头节点的单链表，编写算法实现从尾到头反向输出每个结点的值。

综合应用题

4、试编写在带头结点的单链表L中删除一个最小值结点的高效算法（假设最小值结点是唯一的）。

综合应用题

5、已知一个带有表头结点的单链表，结点结构为

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域的值，并返回1；否则，只返回0。要求：

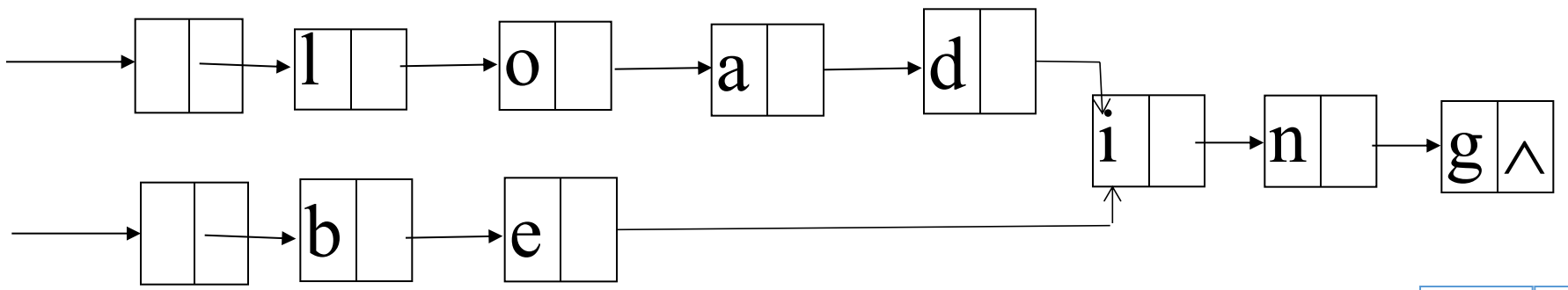
1）描述算法的基本 设计思想。

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，关键之处请给出简要注释。

综合应用题

6、假定采用带头结点的单链表保存单词，当两个单词有相同的后缀时，则可共享相同的后缀存储空间，例如，“loading”和“being”的存储映像如下图所示：



设str1和str2分别指向两个单词所在单链表的头结点，链表结点结构为

data	next

，请设计一个时间上尽可能高效的算法，找出由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：

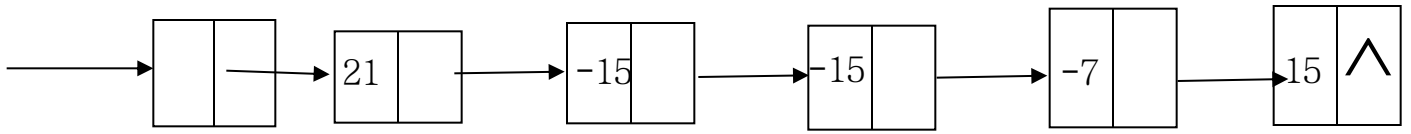
1）描述算法的基本设计思想。

2）根据设计思想，采用程序设计语言描述算法，关键之处请给出简要注释。

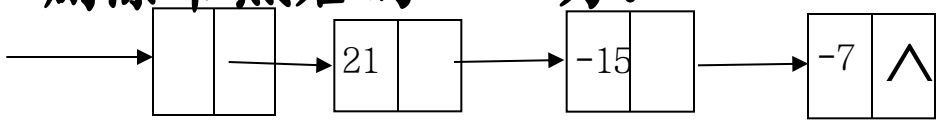
3）说明你所设计算法的时间复杂度

综合应用题

7、用单链表保存m个整数，结点的结构为：[data][link],且|data|≤n(n为正整数）。现要求设计一个时间复杂度尽可能高效的算法，对于链表中data的绝对值相等的结点，仅保留第一次出现的结点而删除其余绝对值相等的结点。例如，若给定的单链表head如下：



删除节点后的head为：

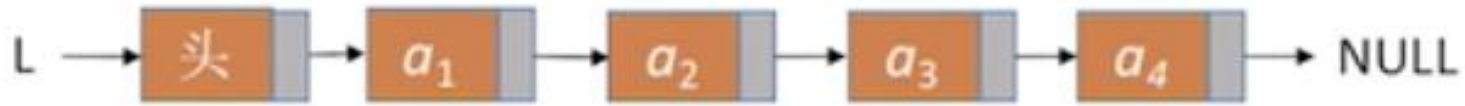


1）描述算法的基本 设计思想。

2）使用C或者C++语言，给出单链表结点的数据类型定义。

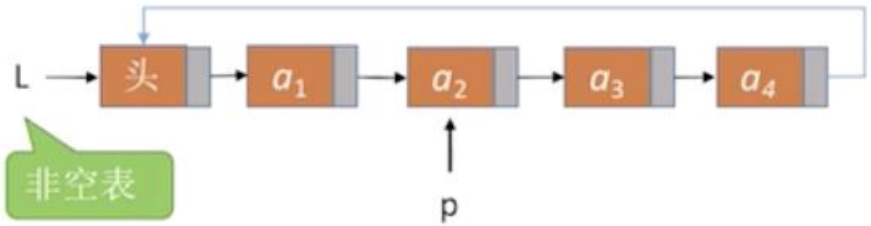
3）根据设计思想和实现步骤，采用C或者C++语言描述算法，关键之处请给出简要注释。4）说明你所设计算法的时间复杂度和空间复杂度。

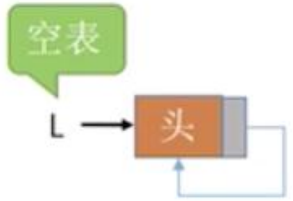
循环单链表



循环单链表

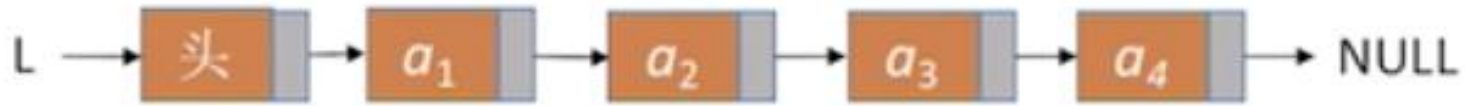
```
typedef struct LNode{
    ElementType Data;
    struct LNode *Next;
} LNode, *LinkList;

bool InitList(LinkList &L){
    L=(LNode *)malloc(sizeof(LNode));
    if (L==NULL)
        return false;
    L->next=L;
    return true;
}
```



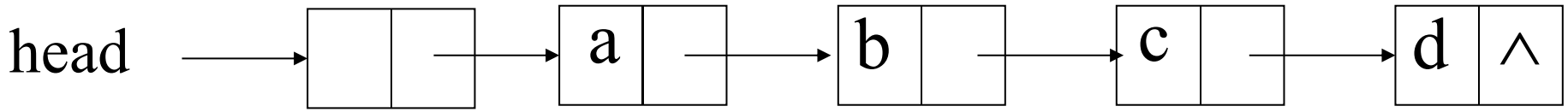


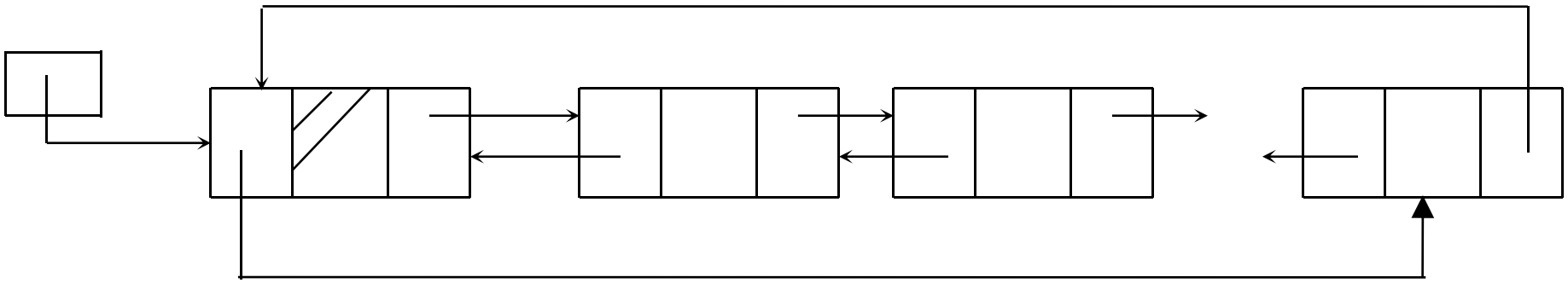

```
bool Empty(LinkList L){
    if (L->next==L)
        return true;
    else
        return false;
}

bool  isTail(LinkList L, LNode *p){
    if (p->next==L)
        return true;
    else
        return false;
}
```

循环单链表



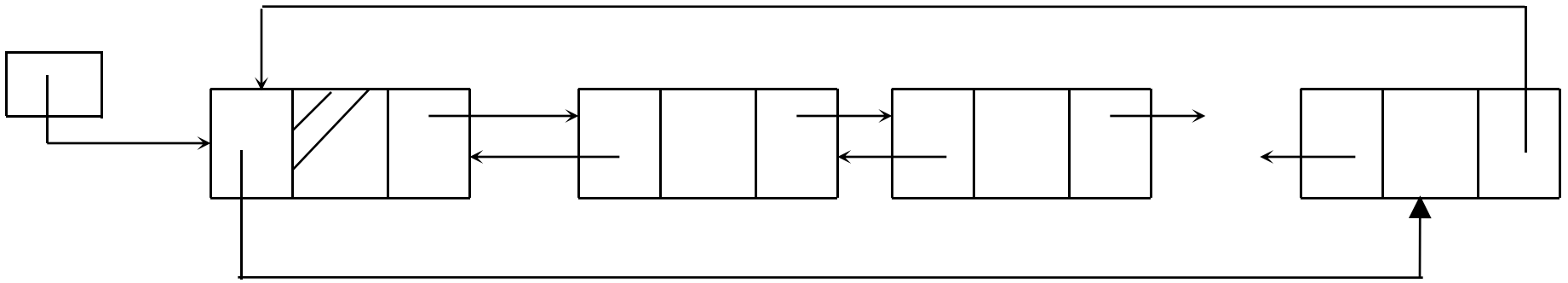
单链表 VS 双链表





表头结点的prior指向表尾结点；
表尾结点的next指向头节点

双链表



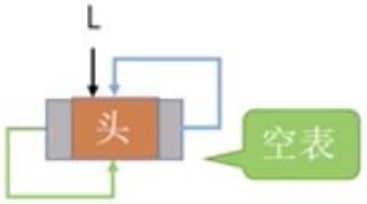
```
typedef  struct  DNode {
        ElemType data;
        struct  DNode *prior,*next;
 } DNode,* DLinkList;
```

循环双链表的初始化

```
typedef  struct  DNode {
        ElemType data;
        struct  DNode *prior,*next;
} DNode,* DLinkList;
```

```
bool InitDLinkList(DLinkList &L){
    L=(DNode *)malloc(sizeof(DNode));
    if (L==NULL)
            return false;
    L->prior=L;
    L->next=L;
    return true;
}
```

```
bool Empty(LinkList L){
    if (L->next==L)
        return true;
    else
        return false;
```



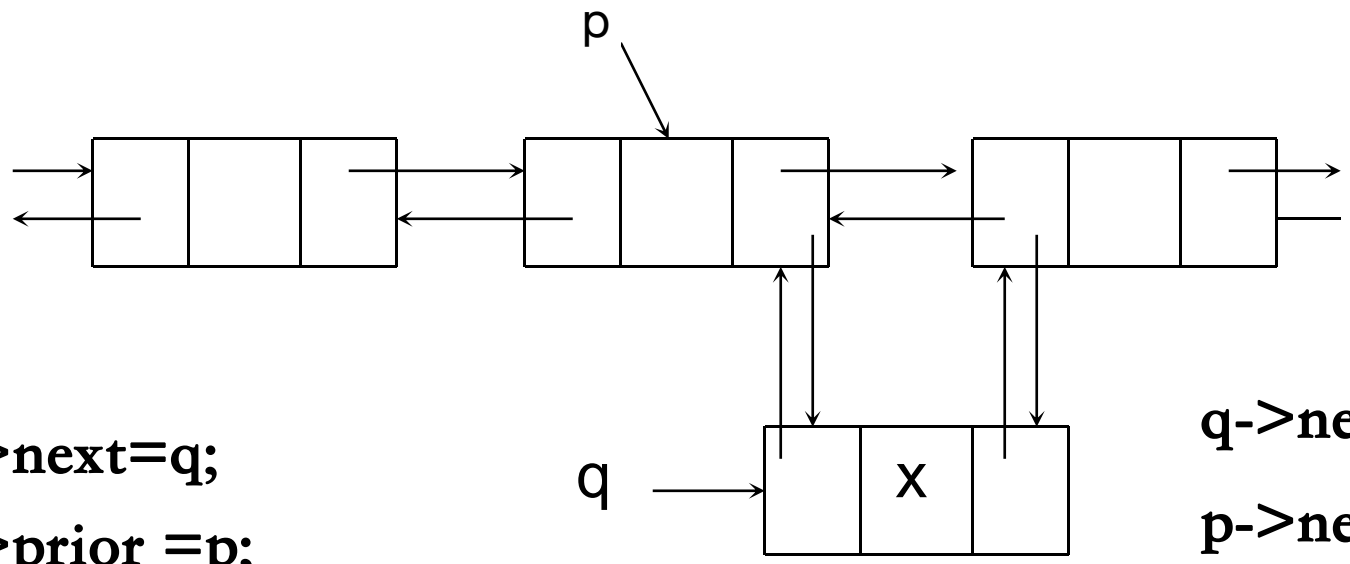
循环双链表的插入

```
typedef  struct  DNode {
        ElemType data;
        struct  DNode *prior,*next;
} DNode,* DLinkList;
```

```
bool InitDLinkList(DLinkList &L){
    L=(DNode *)malloc(sizeof(DNode));
    if (L==NULL)
            return false;
    L->prior=L;
    L->next=L;
    return true;
}
```

```
bool Empty(LinkList L){
    if (L->next==L)
        return true;
    else
        return false;
```

循环双链表的插入



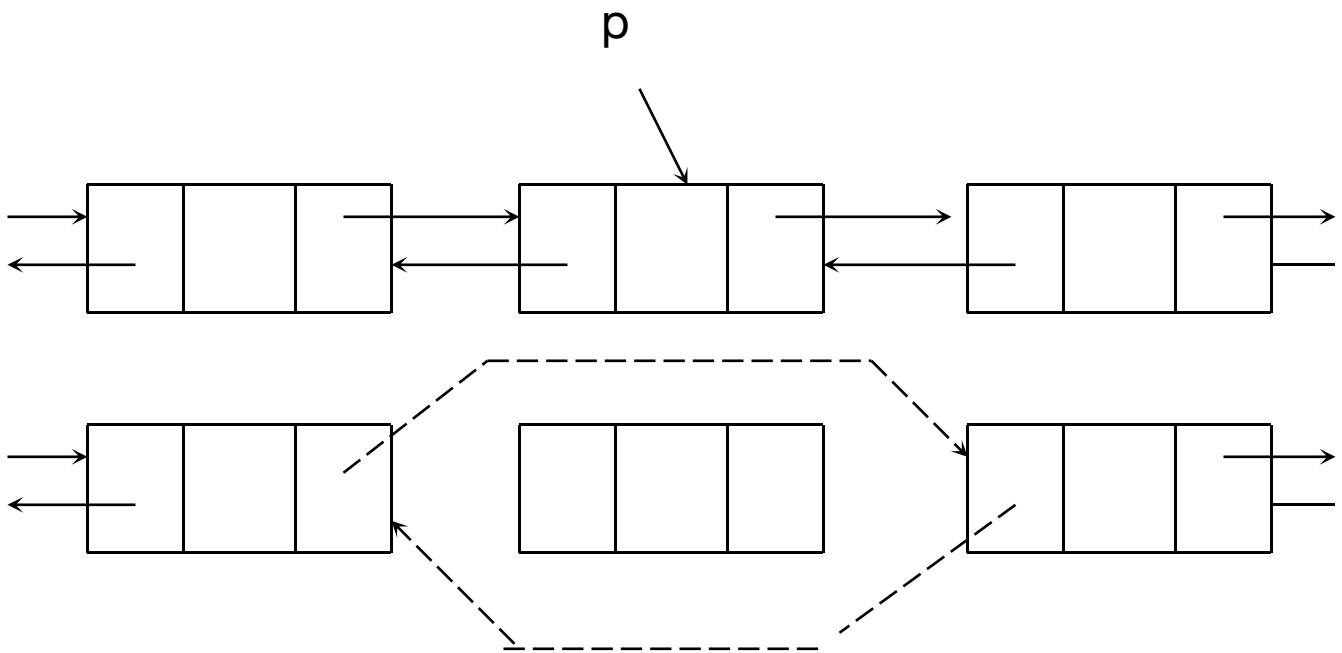
p->next=q;

q->prior =p;

q->next=p->next;

p->next-> prior =q;

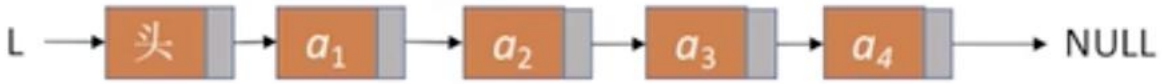
循环双链表的删除

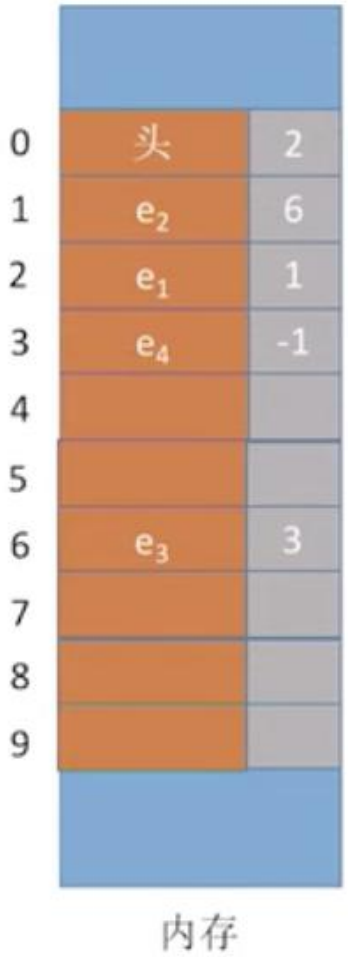


p-> prior ->next=p->next

p->next-> prior =p->prior

静态链表

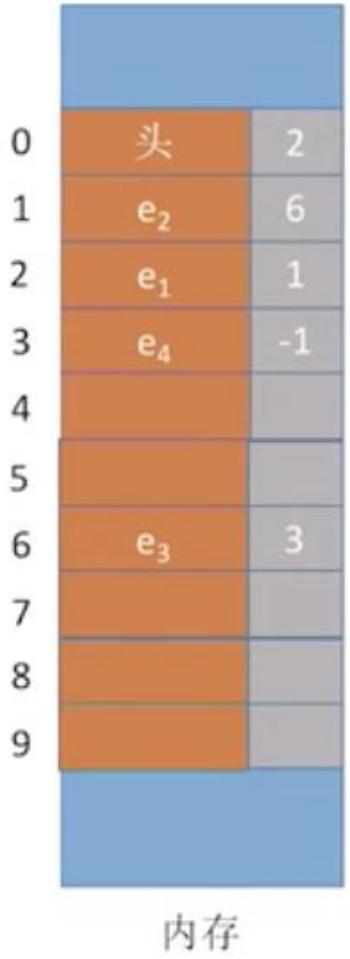




单链表：各个结点在内存散落分布。

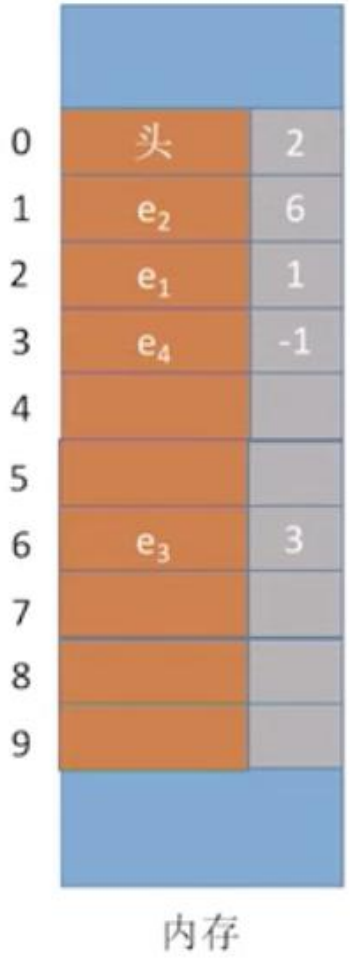
静态链表：分配一整片连续的内存空间，各个结点集中安置。

静态链表定义




```
#define  MaxSize  1000
struct Node{
      ElemType   data;
      int  next;
};


void testSLinkList(){
      struct Node  a[MaxSize];
      //.....
}
```

静态链表定义




```
#define  MaxSize  1000
typedef struct{
      ElemType   data;
      int  cur;
}SLinkList[MaxSize];
```

```
void testSLinkList(){
      SLinkList  a;
      //.....
}
```

```
#define  MaxSize  1000
struct Node{
      ElemType   data;
      int  next;
};
typedef  struct  Node  SLinkList[MaxSize];
```

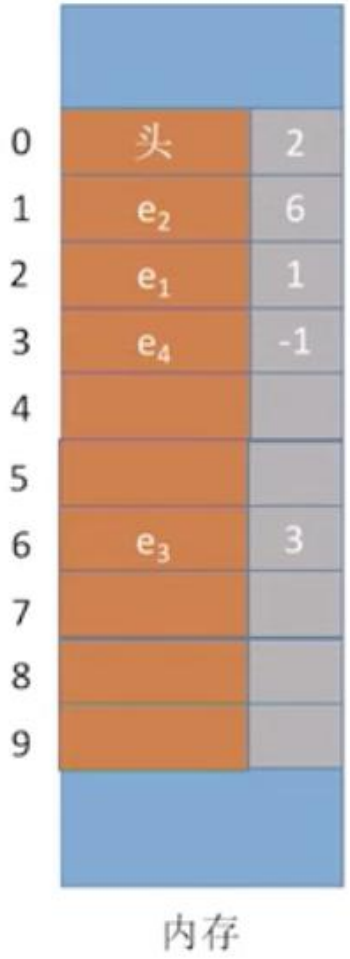
```
void testSLinkList(){
      struct Node  a[MaxSize];
      //.....
}
```

静态链表定义

```
#define  MaxSize  1000
typedef struct{
    ElemType   data;
    int  cur;
}SLinkList[MaxSize];
struct Node{
    ElemType   data;
    int  next;
};
```

```
void testSLinkList(){
    struct Node x;
    printf("sizeX=%d\n",  sizeof(x));

   struct Node a[MaxSize];
    printf("sizeX=%d\n",  sizeof(x));

    SLinkList  a;
    printf("sizeX=%d\n",  sizeof(b));
}
```

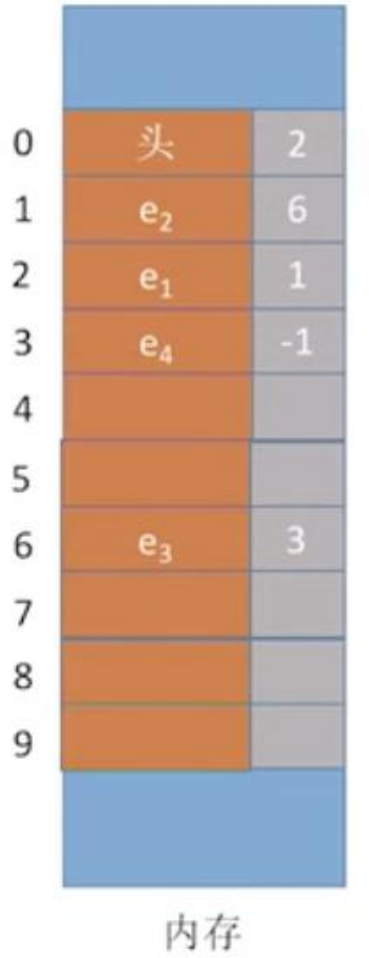
静态链表的基本操作




```
#define  MaxSize  1000
typedef struct{
      ElemType   data;
      int  cur;
}SLinkList[MaxSize];
```

```
//初始化静态链表
a[0]=-1;
```

静态链表的基本操作



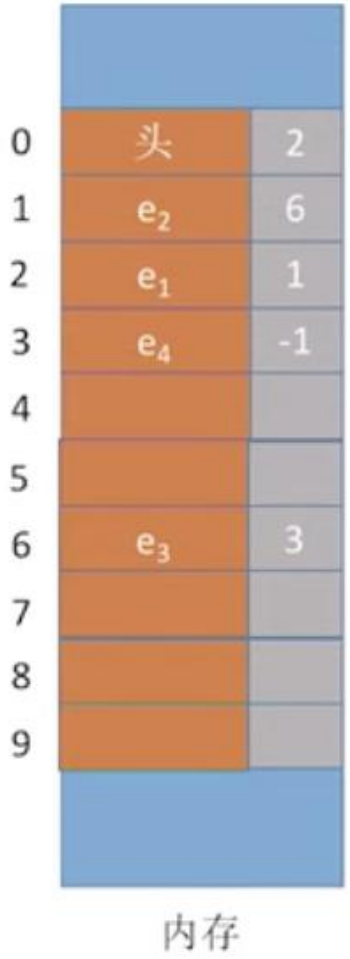

```
#define  MaxSize  1000
typedef struct{
      ElemType   data;
      int  cur;
}SLinkList[MaxSize];
```

查找：
从头节点出发挨个往后遍历结点

插入位序为i的结点：
（1）找到一个空的结点，存入数据元素
（2）从头节点出发找到位序为i-1的结点
（3）修改新结点的next
（4）修改i-1号结点的next

删除位序为i的结点：？？？

静态链表




```
#define  MaxSize  1000
typedef struct{
      ElemType   data;
      int  cur;
}SLinkList[MaxSize];
```

静态链表：用数组的方式实现的链表

优点：增、删操作不需要大量移动元素

缺点：不能随机存取，只能从头节点开始往后查找；容量固定不可变

适用场景：（1）不设置“指针”类型的高级程序设计语言中使用链表结构；（2）数据元素数量固定不变的场景（如操作系统的文件分配表FAT）。