第一章回顾

- 什么是数据、数据结构?
- 本书主要研究哪几种数据结构?
- 什么是数据结构、逻辑结构、物理结构?
- 什么是数据对象、数据元素、数据项，及其之间的关系?
- 什么是数据类型、抽象数据类型?
- 抽象数据类型的定义、表示、实现。
- 什么是算法(基本结构)、算法有哪几个重要特性、算法设计的要求是什么、算法的时间复杂度如何度量?

- 线性表的概念
- 线性表的顺序表示和实现
- 线性表的链式表示和实现
- 各种存储类型之比较
- 线性表的应用

- 线性表的概念
- 线性表的顺序表示和实现
- 线性表的链式表示和实现
- 各种存储类型之比较
- 线性表的应用

线性表的概念

- **文字定义：一个线性表是n个数据元素的有限序列。**

一个数据元素可以由若干个数据项组成，这时，也可以称数据元素为记录。含有大量记录的线性表又称“文件”。

数据元素为字符

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

例2：学生成绩表

（90, 97, 60, 75,…,84）

数据元素为整数

例3：学生健康情况登记表：

文件(file)

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
……	……	……	……	……

数据元素（结点、记录）由5个数据项（字段、域）组成。

线性表中的数据元素可以是各种各样的，但同一线性表中的元素必定具有相同特性（属于同一数据对象）。

线性表中的数据元素之间存在着序偶关系 $<a_{i-1}, a_i>$ 。

线性表的概念(续)

- 最常用且最简单的一种数据结构。

- 特点：4个“惟一”。

线性结构的特点：

数据元素的非空有限集

- 存在唯一的一个被称作“第一个”的数据元素；
- 存在唯一的一个被称作“最后一个”的数据元素；
- 除第一个之外的数据元素均只有一个前驱；
- 除最后一个之外的数据元素均只有一个后继。

例：

法学系	8523101
国贸系	8522105
工商系	8523150
计算机系	8521088
会计系	8525789
统计系	8528136
…	…
外语系	8523026

“第一个”数据元素

直接前驱

直接后继

“最后一个”数据元素

线性表的ADT定义

ADT List {

数据对象：D＝{ ai | ai ∈ElemSet, i=1,2,...,n, n≥0 }

数据关系：R1＝{ <ai-1 ,ai >|ai-1 ,ai∈D, i=2,...,n }

基本操作：

结构初始化操作

结构销毁操作

引用型操作

加工型操作

} ADT List

{ 结构初始化 }

InitList (&L)

操作结果：构造一个空的线性表 L 。

{ 销毁结构 }

DestroyList(&L)

初始条件：线性表 L 已存在。

操作结果：销毁线性表 L 。

{ 引用型操作 }

操作的结果不改变线性表中的数据元素，也不改变数据元素之间的关系。

ListEmpty(L)

初始条件：线性表 L 已存在。

操作结果：若 L 为空表，则返回 TRUE，否则返回 FALSE。

ListLength(L)

初始条件：线性表 L 已存在。

操作结果：返回 L 中元素个数。

PriorElem(L, cur_e, &pre_e)

初始条件：线性表 L 已存在。

操作结果：若 cur_e 是 L 中的数据元素，则用 pre_e 返回它的前驱，否则操作失败，pre_e 无定义。

NextElem(L, cur_e, &next_e)

初始条件：线性表 L 已存在。

操作结果：若 cur_e 是 L 中的数据元素，则用 next_e 返回它的后继，否则操作失败，next_e 无定义。

GetElem(L, i, &e)

初始条件：线性表 L 已存在，1≤i≤LengthList(L)。

操作结果：用 e 返回 L 中第 i 个元素的值。

LocateElem(L, e, compare( ))

初始条件：线性表 L 已存在，compare( ) 是判定函数。

操作结果：返回 L 中第 1 个与 e 满足关系 compare( ) 的元素的位序。若这种元素不存在，则返回 0。

ListTraverse(L, visit( ))

初始条件：线性表 L 已存在，visit( ) 为访问函数。

操作结果：依次对 L 的每个元素调用函数 visit( )。

一旦 visit( ) 失败，则操作失败。

{ 加工型操作 }　　操作的结果或修改表中的数据元素，或修改元素之间的关系

ClearList(&L)

初始条件：线性表 L 已存在。

操作结果：将 L 重置为空表。

思考：ClearList(L) 操作与DestroyList(L) 操作 的区别

PutElem(&L, i, e)

初始条件：线性表 L 已存在，1≤i≤LengthList(L)。

操作结果：L 中第 i 个元素赋值为 e 的值。

ListInsert(&L, i, e)

初始条件：线性表 L 已存在，1≤i≤LengthList(L)+1。

操作结果：在 L 的第 i 个元素之前插入新的元素 e，

L 的长度增 1。

ListDelete(&L, i, &e)

初始条件：线性表 L 已存在且非空，1≤i≤LengthList(L)。

操作结果：删除 L 的第 i 个元素，并用 e 返回其值，

L 的长度减 1。

} ADT List

基本操作的应用举例

- 引用型操作、加工型操作的特征（相对谁来说的）。
- 例如，主程序中有如下代码：

```
List  myList;

//其他代码（省略）

ListEmpty(myList);     //引用型操作

ListInsert(myList,3,12);   //加工型操作
```

（在第三个位置之前插入元素12）

ListEmpty(L)

ListInsert(&L, i, e)

线性表ADT基本操作的简单应用

例2-1：已知集合 A 和 B，求这两个集合的并集，

使 A = A∪B，且 B 不再单独存在。

要在计算机中求解，首先要确定“如何表示集合”。

用线性表表示集合

以线性表 LA 和 LB 分别表示集合 A 和 B，两个线性表的数据元素分别为集合 A 和 B 中的成员。

由此，上述集合求并的问题便可演绎为：

扩大线性表 LA，将存在于线性表 LB 中而不存在于线性表 LA 中的数据元素插入到线性表 LA 中去。

1. 从 Lb 中取出一个数据元素；　GetElem (Lb, *i*, &*e*)
ListDelete (&Lb, *i*, &*e*)

2. 依次在 La 中进行查询；　LocateElem (La, *e*, equal())

3. 若不存在，则插入之。　ListInsert (&La, *n* + 1, *e*)

重复上述三步直至 Lb 中的数据元素取完为止。

其中的每一步能否利用线性表类型中定义的基本操作来完成呢？

算法 2.1

```
void union(List &La, List Lb)
{  La_len = ListLength(La);        Lb_len =ListLength(Lb);
   for (i = 1; i <= Lb_len; i ++)
   {   GetElem (Lb, i, &e);        // 取 Lb 中第 i 个数据元素赋给 e
       if(!LocateElem (La, e, equal()))
          ListInsert (&La, ++La_len, e);
           // La 中不存在和 e 相同的数据元素，则插入之
   }
 DestroyList (Lb);    // 销毁线性表 Lb
} // union
```

假设执行时间与表长无关

执行时间与表长成正比

时间复杂度：$O(\text{ListLength (La)} \times \text{ListLength (Lb)})$

例2.2 合并两个有序表

LA = (3,5,8,11)

LB = (2,6,8,9,11,15,20)

LC = (2,3,5,6,8,8,9,11,11,15,20)

思路:

1．分别从 La 和 Lb 中取得当前元素 a_i 和 b_j；

2．若 $a_i \le b_j$，则将 a_i 插入到 Lc 中，否则将 b_j 插入到 Lc 中。

算法 2.2

```
void MergeList(List La, List Lb, List &Lc) {
  InitList(&Lc);                                   假设执行时间与表长无关
  i = j = 1; k = 0;
  La_len = ListLength(La);    Lb_len = ListLength(Lb);
  while ((i ≤ La_len) && ( j ≤ Lb_len)) {   // La 和 Lb 均未取完
    GetElem(La, i, a_i ); GetElem(Lb, j, b_j );
    if (a_i ≤ b_j ) {ListInsert(Lc, ++k, a_i ); ++i; }
    else { ListInsert(Lc, ++k, b_j ); ++j; }
  }
  while (i≤La_len) {GetElem(La, i++, a_i);  ListInsert(Lc, ++k, a_i);}
  while (j≤Lb_len) {GetElem(Lb, j++, b_j);  ListInsert(Lc, ++k, b_j);}
}
```

时间复杂度： O(ListLength(La) + ListLength(Lb))

在实际的程序设计中**要使用**线性表的基本操作，

必须**先实现**线性表类型。

确定存储结构

实现基本操作

- 线性表的概念
- 线性表的顺序表示和实现
- 线性表的链式表示和实现
- 各种存储类型之比较
- 线性表的应用

线性表的顺序存储结构

- 在计算机中用一组地址连续的存储单元依次存储线性表的各个数据元素，称作线性表的顺序存储结构或顺序映象。用这种方法存储的线性表称作顺序表。

线性表的顺序存储结构示意图

线性表的顺序存储结构（续）

假设线性表的每个元素需占 l 个存储单元，则第 $i+1$ 个元素的存储位置和第 i 个元素的存储位置之间满足关系：

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

由此，所有数据元素的存储位置均可通过基地址得到：

$$LOC(a_i) = LOC(a_1) + (i-1) \times l$$

特点：以物理位置相邻表示逻辑关系；任一元素均可随机存取。

结论：已知位置、获取该位置上的元素非常方便，与该线性表的长度无关。

考虑到线性表因插入元素而使存储空间不足的问题，应允许数组容量进行动态扩充。（静态顺序存储->动态顺序存储）

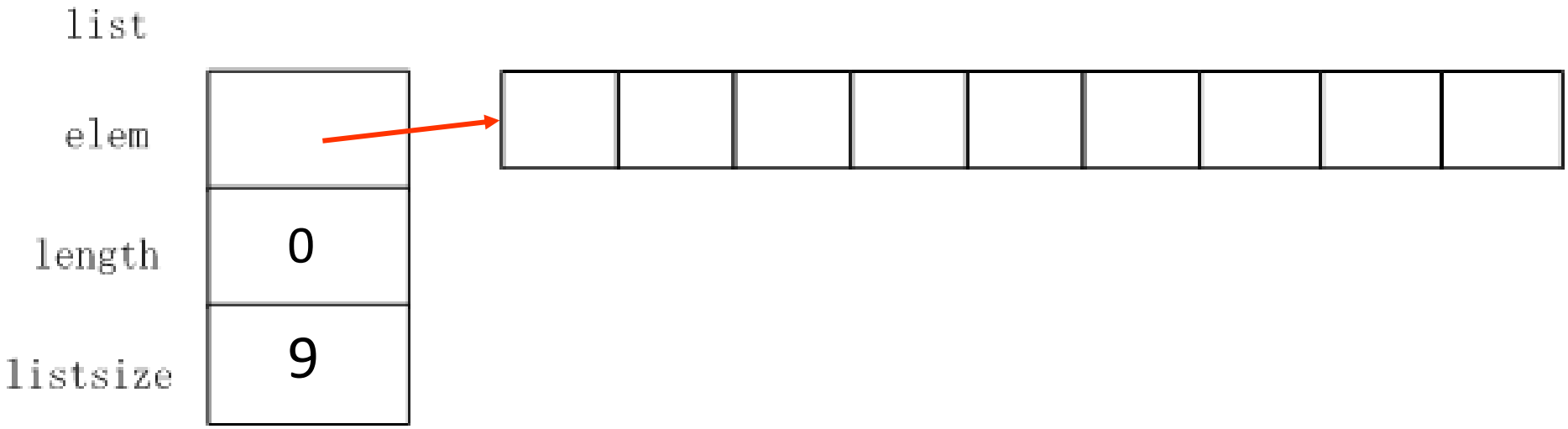
```
#define LIST_INIT_SIZE 100  //线性表存储空间的初始分配量
#define LISTINCREMENT 10 //线性表存储空间的分配增量
typedef struct {
    ElemType  *elem;  //数组指针，指示线性表的基地址
   int length;  //当前长度
    int listsize;  //当前分配的存储容量(以sizeof(ElemType)为单位)
} SqList;
```

注意

C 语言中的数组下标从 “0” 开始，因此若 L 是 Sqlist 类型的顺序表，则表中第 *i* 个元素是 L.elem[*i* -1]。

线性表的操作举例
--初始化操作

- 构造一个空的线性表（顺序表）



初始化---类c语法描述

```
bool InitList(SqList   &l)
{
        l.elem=(ElemType *)
          malloc(LIST_INIT_SIZE*sizeof(ElemType));
        if(!l.elem)
                exit  (OVERFLOW);
        l.length=0;
        l.listsize=LIST_INIT_SIZE;
        return  OK;
}
```

初始化---用c语言描述

```
bool InitList(SqList *l)
{
        l->elem=(ElemType *)
          malloc(LIST_INIT_SIZE*sizeof(ElemType));
        if(!l->elem)
                return false;
        l->length=0;
        l->listsize=LIST_INIT_SIZE;
        return true;
}
```

- **插入操作：** 在第i个位置之前插入新元素

线性表的插入运算是指在表的第 i $(1 \le i \le n+1)$ 个位置上，插入一个新结点 b，使长度为 n 的线性表 $(a_1, \ldots, a_{i-1}, a_i, \ldots, a_n)$ 变成长度为 $n+1$ 的线性表 $(a_1, \ldots, a_{i-1}, b, a_i, \ldots, a_n)$

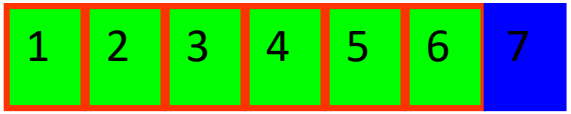
算法思想：

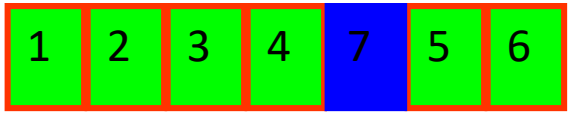
1）检查 i 值是否超出所允许的范围 $(1 \le i \le n+1)$，若超出，则进行“超出范围”错误处理；

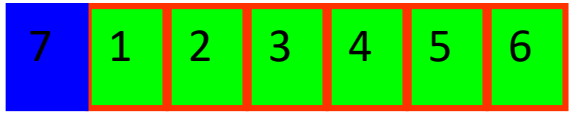
2）将线性表的第 i 个元素和它后面所有元素均后移一个位置；

3）将新元素写入到空出的第 i 个位置上；

4）使线性表的长度增 1。







算法 2.4

```
Status ListInsert_Sq(Sqlist &L,int i,ElemType e){
    if(i<1||i>L.length+1) return ERROR;
    if(L.length>=L.listsize){   //当前存储已满,增加分配
    newbase=(ElemType*)realloc(L.elem,
          (L.listsize+ LISTINCREMENT)*sizeof(ElemType));
    If(!newbase) exit(OVERFLOW);
    L.elem=newbase;
    L.listsize+= LISTINCREMENT;
    }
    q=&(L.elem[i-1]);
    for(p =&(L.elem[L.length-1]) ;p>=q ; - -p) *(p+1)=*p;
    //插入位置之后元素后移
      *q=e;
     ++L.length;
     return  OK;
    }
```

插入算法的时间复杂度分析

- **问题规模**是表的长度，设它的值为 n。
- 算法的时间主要花费在向后移动元素的 for 循环语句上。该语句的循环次数为 $(n-i+1)$。由此可看出，所需移动结点的次数不仅依赖于表的长度 n，而且还与插入位置 i 有关。
- 当插入位置在表尾 $(i=n+1)$ 时，不需要移动任何元素；这是最好情况，其时间复杂度 $O(1)$。
- 当插入位置在表头 $(i = 1)$ 时，所有元素都要向后移动，循环语句执行 n 次，这是最坏情况，其时间复杂度 $O(n)$。

- **算法的平均时间复杂度**：设 p_i 为在第 i 个元素之前插入一个元素的概率，则在长度为 n 的线性表中插入一个元素时所需移动元素次数的期望值为

$$E_{is} = \sum_{i=1}^{n+1} p_i (n - i + 1)$$

假设在表中任何位置 $(1 \le i \le n+1)$ 上插入结点的机会是均等的，**则**

$$p_i = \frac{1}{n+1} \qquad E_{is} = \frac{1}{n+1} \sum_{i=1}^{n+1} (n - i + 1) = \frac{n}{2}$$

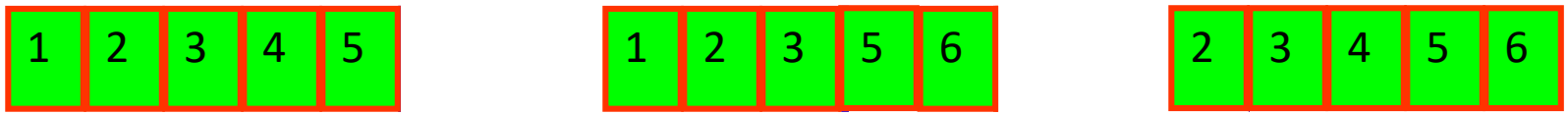
由此可见，**在顺序表上做插入运算，平均要移动表上一半元素**。当表长 n 较大时，算法的效率相当低。算法的平均时间复杂度为 $O(n)$。

- **删除操作**

线性表的删除运算是指将线性表的第 i $(1 \le i \le n)$ 个结点删除，使长度为 n 的线性表 $(a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$ 变成长度为 n -1 的线性表 $(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n)$

算法思想：

1) 检查 i 值是否超出所允许的范围 $(1 \le i \le n)$，若超出，则进行“超出范围”错误处理；
2) 将线性表的第 i 个元素后面的所有元素均前移一个位置；
3) 使线性表的长度减 1。



算法 2.5

```
Status ListDelete_Sq(SqList &L, int i, ElemType &e) {
  if ((i <1) || (i > L.length)) return ERROR;  // 删除位置不合法
    p = &(L.elem[i -1]);  // p为被删除元素的位置
    e = *p;  // 被删除元素的值赋给 e
    q = L.elem+L.length-1;  // 表尾元素的位置
    for (++p; p <= q; ++p) *(p-1) = *p;
        // 被删除元素之后的元素左移
    --L.length;  //表长减 1
    return OK;
}//ListInsert_sq
```

删除算法的复杂度分析

- 问题规模是表的长度，设它的值为 n。
- 算法的时间主要花费在向前移动元素的 for 循环语句上。该语句的循环次数为 $(n - i)$。由此可看出，所需移动结点的次数不仅依赖于表的长度 n，而且还与删除位置 i 有关。
- 当删除位置在表尾 $(i = n)$ 时，不需要移动任何元素；这是最好情况，其时间复杂度 $O(1)$。
- 当删除位置在表头 $(i = 1)$ 时，有 n-1 个元素要向前移动，循环语句执行 n-1 次，这是最坏情况其时间复杂度 $O(n)$。

- **算法的平均时间复杂度**：设 q_i 为删除第 i 个元素的概率，则在长度为 n 的线性表中删除一个元素时所需移动元素次数的期望值为

$$E_{dl} = \sum_{i=1}^{n} q_i (n-i)$$

假设在表中任何位置($1 \le i \le n$)删除结点的机会是均等的，则：

$$q_i = \frac{1}{n} \qquad E_{dl} = \frac{1}{n}\sum_{i=1}^{n}(n-i) = \frac{n-1}{2}$$

由此可见，在顺序表上做删除运算，平均约要移动表上一半元素。当表长 n 较大时，算法的效率相当低。算法的平均时间复杂度为 $O(n)$。

课堂练习

1、一个线性表第一个元素的存储地址是 100，每个元素的长度为 2， 则第 5 个元素的地址是 (　　)。

（A）110 （B）108（C）100 （D）120

B

2、向一个有 127 个元素的顺序表中插入一个新元素并保持原来顺序不变，平均要移动（ ）个元素。

（A）64（B）63 （C）63.5　（D）7

C

3、顺序存储结构是通过 _____ 表示元素之间的关系的

(A) 逻辑上相邻　(B) 物理上地址相邻　(C) 指针　(D) 下标

B

4、对于顺序存储的线性表，访问结点和删除结点的时间复杂度分别为（

(A) O (1) 、O (n)　　(B) O (1) 、O (1)

(C) O (n) 、O (1)　　(D) O (n) 、O (n)

A

- 线性表的概念
- 线性表的顺序表示和实现
- 线性表的链式表示和实现
- 各种存储类型之比较
- 线性表的应用

线性表的链式存储—链表

用一组物理位置任意的存储单元来存放线性表的数据元素。 这组存储单元既可以是连续的，也可以是不连续的，甚至是零散分布在内存中的任意位置上的。因此，链表中元素的逻辑次序和 物理次序不一定相同。

例：线性表：（赵，钱，孙，李，周，吴，郑，王）

顺序表

存储地址	存储状态
0031	赵
0033	钱
0035	孙
0037	李
0039	周
0041	吴
0043	郑
0045	王

头指针 H

0031

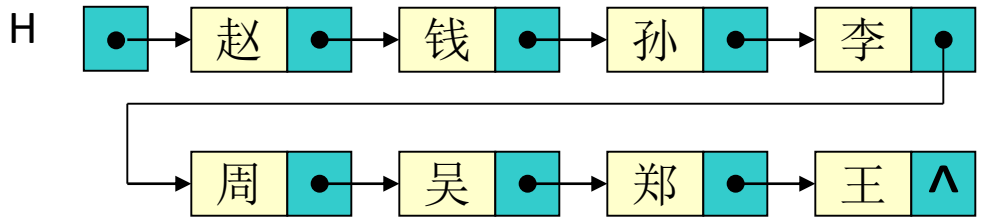
链表

存储地址	数据域	指针域
0001	李	0043
0007	钱	0013
0013	孙	0001
0019	王	NULL
0025	吴	0037
0031	赵	0007
0037	郑	0019
0043	周	0025

结点

链表 单链表

链指针



单链表是由头指针唯一确定，因此单链表可以用头指针的名字来命名。

单链表的表示

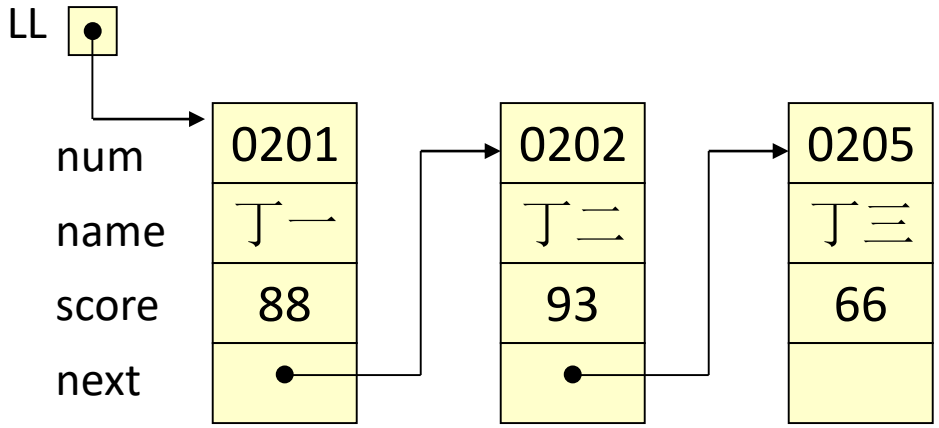
单链表在 C 语言中可用“结构指针”来描述：

```
typedef  struct  Lnode{
    //声明结点的类型和指向结点的指针类型
    ElemType      data;   //数据元素的类型
    struct   Lnode  *next;  //指示结点地址的指针
}Lnode, *LinkList;
```

结构体类　型

指向 LNode 结构体类型的指针

例子：



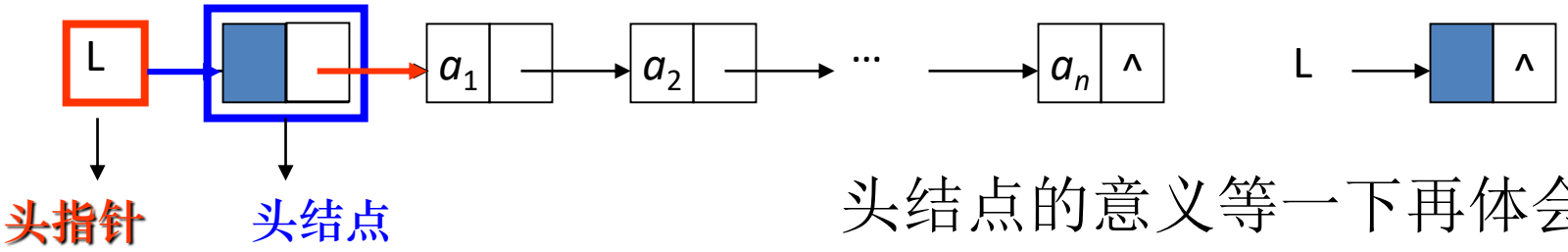

```
struct Student
{ char num[8];   //数据域
  char name[8];  //数据域
  int score;     //数据域
  struct Student *next;  // next 既是 struct Student
                         // 类型中的一个成员，又指
                         // 向 struct Student 类型的数据。
}Stu_1, Stu_2, Stu_3, *LL;
```

头结点：在单链表的第一个结点之前人为地附设的一个结点。

头结点
- 数据域
  - 不存放任何数据
  - 存放附加信息（链表的结点个数等）。
- 指针域　存放第一个结点的地址

（若线性表为空表，则“空”，用 ^ 表示。）

头指针存放头结点的地址。



头结点的意义等一下再体会

以后没特别说明，都是带头结点的单链表

单链表的基本操作

1、查找运算

- **按序号查找（GetElem(L, *i*, &*e*)在链表中的实现）**

在单链表中，即使知道被访问结点的序号 *i*，也不能象顺序表中那样直接按序号 *i* 访问结点，而只能从头指针出发，顺链域 next 逐个结点往下搜索，直到搜索到第 *i* 个结点为止。因此，**单链表是非随机存取的存储结构**。

设单链表的长度为 n，要查找表中第 i 个结点，仅当 $1 \le i \le n$ 时，i 的值是合法的。其算法如下：

算法 2.8

```
Status GetElem_L(LinkList L, int i, ElemType &e) {
  p = L –> next; j = 1; // 初始化，p 指向第一个结点，
                        // j 为计数器
  while ( p && j < i ) { p = p –> next; ++j; }
  if ( !p || j > i ) return ERROR;  // 第 i 个元素不存在
    e = p –> data; // 取第 i 个元素
    return OK;
} // GetElem_L
```

算法的时间复杂度为：$O(n)$

- **按值查找（LocateElem(L, e) 在链表中的实现）**

按值查找是在单链表中查找结点值等于给定值 key 的结点，若有的话，则返回首次找到的其值为 key 的结点的存储位置；否则返回 NULL。其算法如下：

```
LinkList  GetElem_L1(LinkList L1, ElemType key)
  {
    p = L1 –> next;
    while ( p && p –> data!=key)
       p = p –> next;
    return p;
  } // GetElem_L1
```

该算法的执行时间与 key 有关，时间复杂度为：$O(n)$

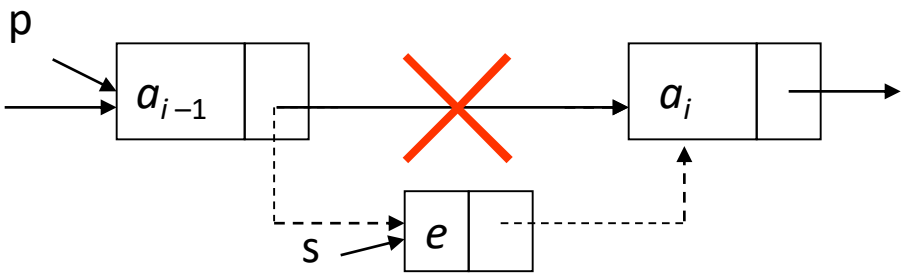
2、插入运算（ListInsert(&L, *i*, *e*)在链表中的实现）

步骤：　1、首先找到 a_{i-1} 的存储位置 p。

2、生成一个数据域为 *e* 的新结点。

3、插入新结点：①、新结点的指针域指向结点 a_i。

②、结点 a_{i-1} 的指针域指向新结点。



s –> next = p –> next;　　p –> next = s;

算法 2.9

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

如果L不是带头结点的链表情况如何呢？

时间复杂度：$O(n)$。

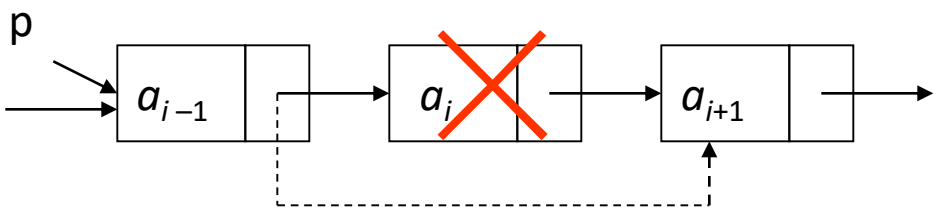
3、删除运算（ListDelete(&L, *i*, &*e*)在链表中的实现）

步骤：

1、首先找到 a_{i-1} 的存储位置 p。

2、令 p –> next 指向 a_{i+1}。

3、释放结点 a_i 的空间。



p –> next = p –> next –> next;

作用是对链表进行操作时，可以对空表、非空表的情况以及对首元结点进行统一处理，编程更方便。

```
Status ListDelete_L(LinkList &L, int i, ElemType &e) {
    p = L;   j = 0;
    while ( p –>next && j < i –1) { p = p –> next; ++j; }
    if (!(p –> next) || j > i –1) return ERROR; // 删除位置不合理
    q = p –> next;    p –> next = q –> next; // 删除并释放结点
    e = q –> data;    free(q);
    return OK;
} // ListDelete_L
```

如果L不是带头结点的链表情况如何呢？

时间复杂度为：$O(n)$

在链表上实现插入和删除运算，无须移动结点，仅需修改指针。

因为每个新生成的结点的插入位置在表尾，则算法中必须维持一个始终指向已建立的链表表尾的指针。

算法 2.11

```
void CreateList_L(LinkList &L, int n) {
// 逆位序输入 n 个元素的值，建立带表头结点的单链表 L。
  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
    scanf(&p –> data);   // 输入元素值
    p –> next = L –> next; L –> next = p;  // 插入到表头 }
} // CreateList_L
```

初始化

算法的时间复杂度为：$O(n)$

课堂练习

1、线性表采用链式存储结构时，其地址 ( )。

D

(A) 必须是连续的　　(B) 部分元素的地址必须是连续的

(C) 一定是不连续的　　(D) 连续与否均可以

2、在一个单链表中，在 p 之后插入 s 所指结点，则执行 ( )。

B

（A）s->next=p;p->next=s;　（B）s->next=p->next;p->next=s;

（C）s->next=p->next;p=s;　（D）p->next=s;s->next=p;

3、在一个单链表中，若删除 p 所指结点的后续结点，则执行 ( )

A

（A）p->next=p->next->next;

（B）p=p->next; p->next=p->next->next;

（C）p->next=p->next;

（D）p =p->next->next;

静态链表表示

```
#define MAXSIZE 1000        / /链表的最大长度

typedef  struct{
    ElemType data;
    int cur;
}component,  SLinkList[MAXSIZE];
```

静态链表

0		1
1	a	2
2	b	3
3	c	4
4	d	5
5	f	6
6	g	7
7	h	8
8	i	0
9		
10		

（a）初始化

0		1
1	a	2
2	b	3
3	c	4
4	d	9
5	f	6
6	g	7
7	h	8
8	i	0
9	e	5
10		

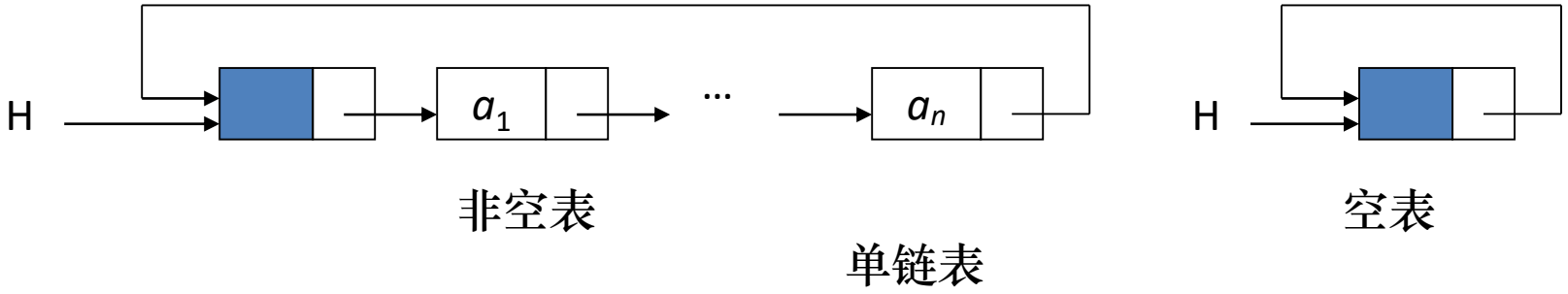
（b）插入 e 后

0		1
1	a	2
2	b	3
3	c	4
4	d	9
5	f	6
6	g	8
7	h	8
8	i	0
9	e	5
10		

（c）删除 h 后

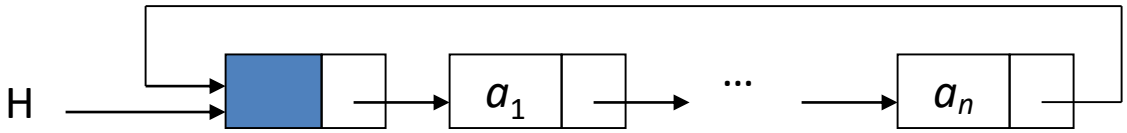
循环链表

循环链表：是一种头尾相接的链表（即：表中最后一个结点的指针域指向头结点，整个链表形成一个环）。

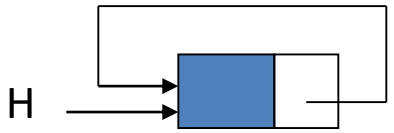


优点：从表中任一结点出发均可找到表中其他结点。

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头指针。



非空表



空表

头指针表示单循环链表

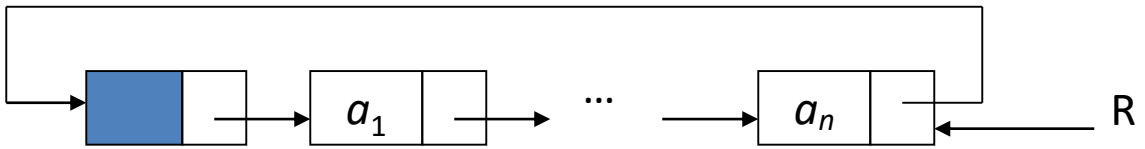
- 找 a_1 的时间复杂度：$O(1)$
- 找 a_n 的时间复杂度：$O(n)$ **不方便**

注意：表的操作常常是在表的首尾位置上进行。

尾指针表示单循环链表

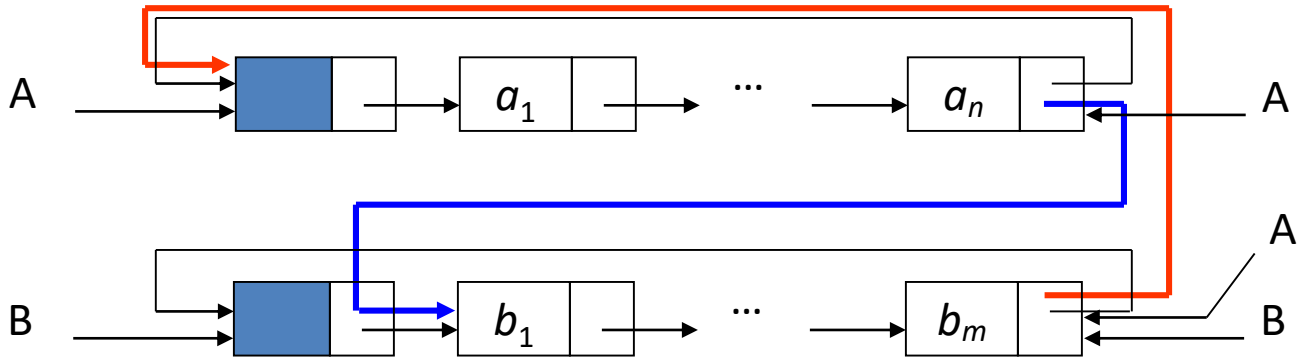
- a_1 的存储位置是：R–>next–>next
- a_n 的存储位置是：R

时间复杂度：$O(1)$



例子：将两个线性表合并成一个线性表。

仅需将一个表的表尾和另一个表的表头相接。



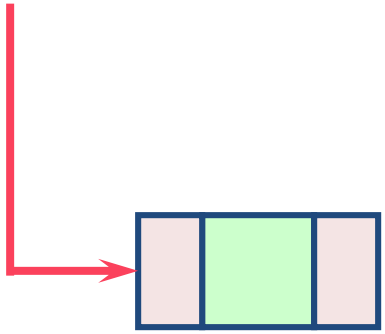
C = A –>next　　A –>next = B –>next –>next

B –>next = C　　A=B

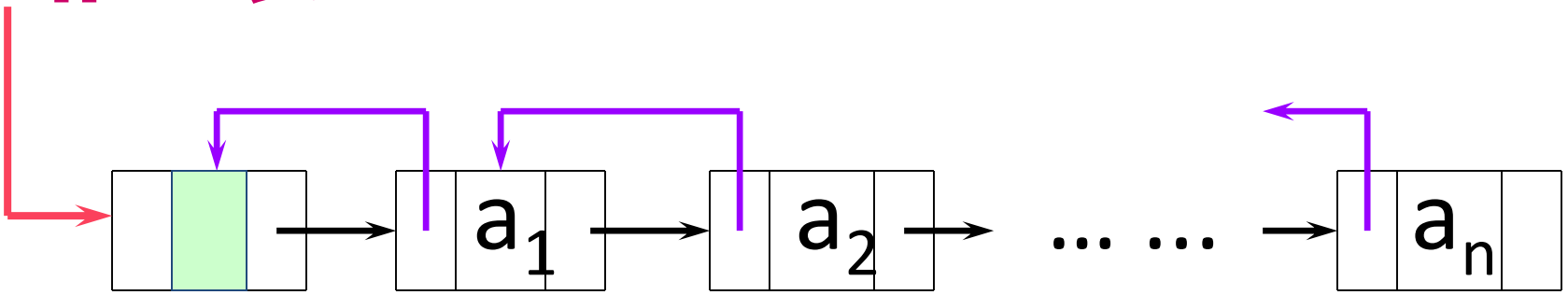
当线性表以上图的循环链表作存储结构时，此操作仅需改变两个指针即可。**时间复杂度是** $O(1)$。

双向链表

空表

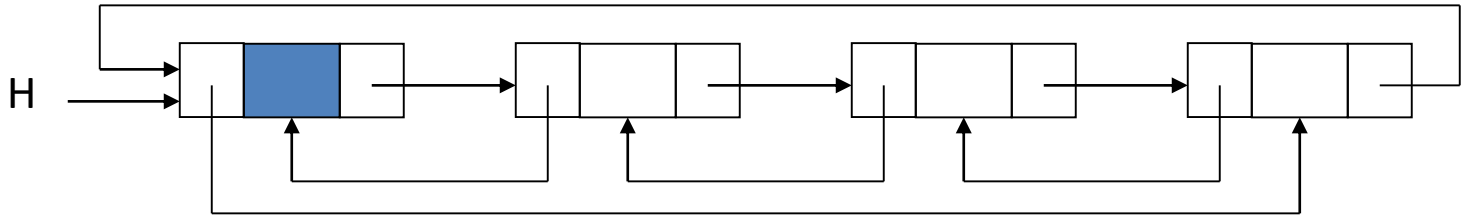


非空表

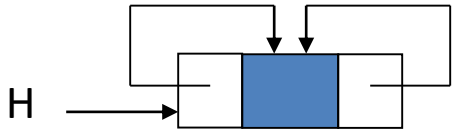


双向循环链表

和单链的循环表类似，双向链表也可以有循环表，让头结点的前驱指针指向链表的最后一个结点，让最后一个结点的后继指针指向头结点。



非空表



空表

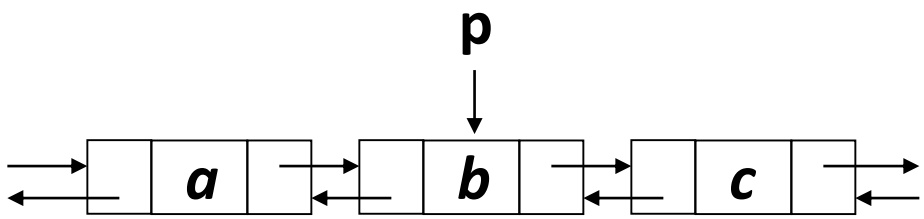
双向链表的结构可定义如下：

```
typedef struct DuLNode{
   Elemtype              data;
   struct DuLNode    *prior, *next;
} DuLNode, *DuLinkList;
```

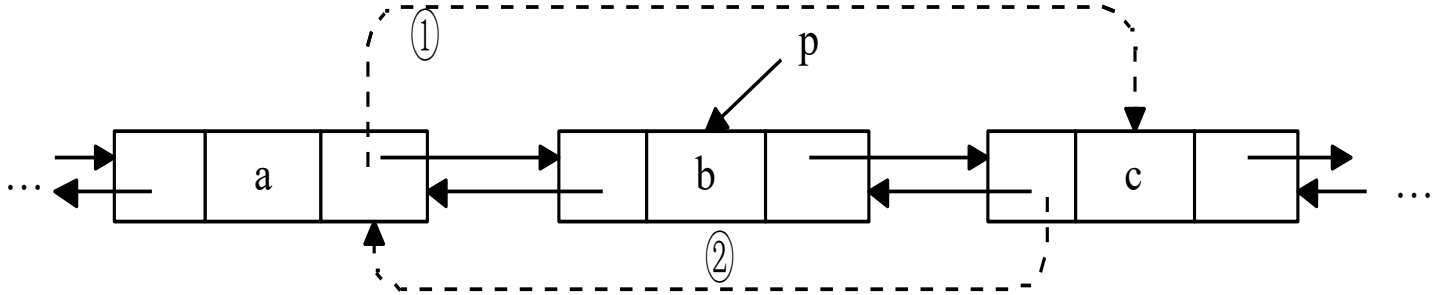
结点结构

双向链表结构的对称性（设指针 p 指向某一结点）：

p –> prior –> next = p = p –> next –> prior



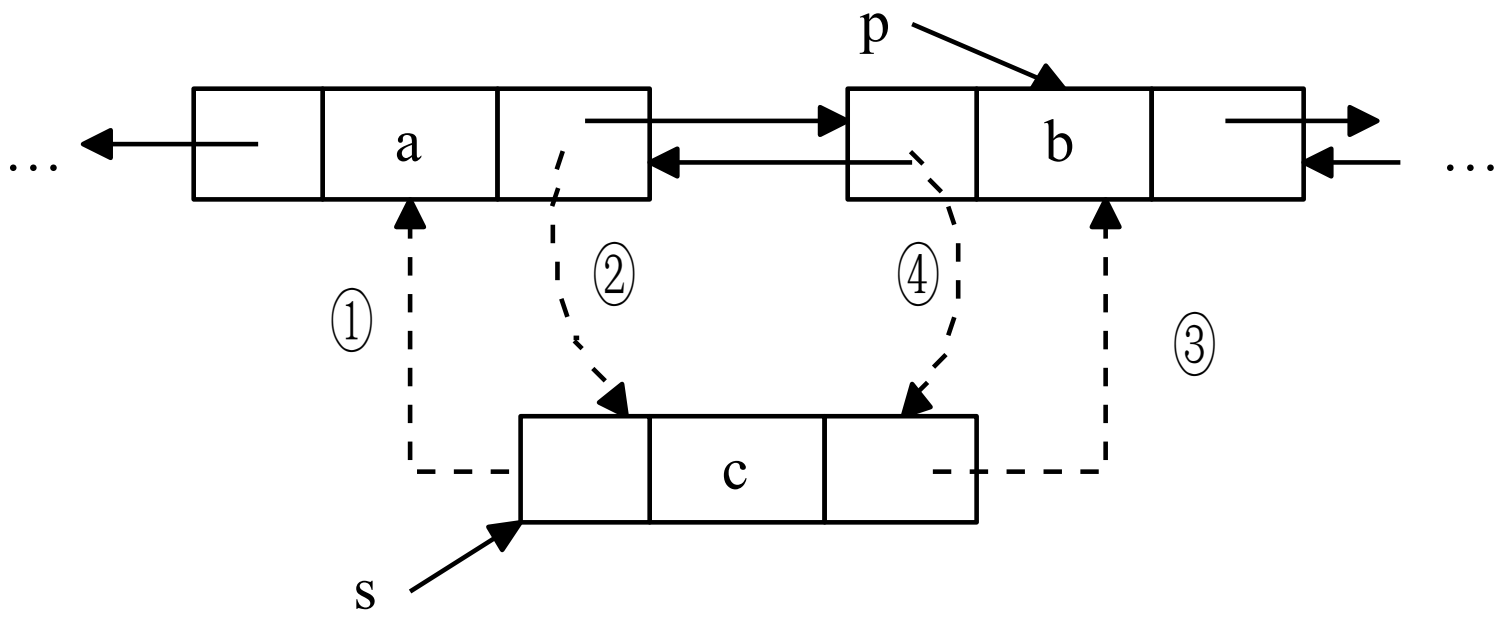
双链表的删除结点过程



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

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

双链表的插入结点过程



s->prior = p-> prior;

p-> prior->next= s;

s->next= p;

p->prior = s;

- 线性表的概念
- 线性表的顺序表示和实现
- 线性表的链式表示和实现
- 各种存储类型之比较
- 线性表的应用

四种存储方式的比较

- {顺序、链式}，{静态、动态}

1、顺序存储的固有特点：

逻辑顺序与物理顺序一致，本质上是用数组存储线性表的各个元素（即随机存取）；存储密度大，存储空间利用率高。

2、链式存储的固有特点：

元素之间的关系采用这些元素所在的节点的”指针”信息表示(插、删不需要移动节点)。

3、静态存储的固有特点：

在程序运行的过程中不用考虑追加内存的分配问题。

4、动态存储的固有特点：

可动态分配内存；有效的利用内存资源，使程序具有可扩展性。

问：动态顺序表和动态链式表各有哪些优缺点？

答：动态顺序存储：

优点：存储密度大，存储空间利用率高，可随机存取。

节点空间可动态申请追加。

缺点：插入或删除元素时不方便。

动态链式存储：

优点：插入或删除元素时很方便，使用灵活。

结点空间可以动态申请和释放；

缺点：存储密度小，存储空间利用率低，非随机存取。

事实上，链表插入、删除运算的快捷是以空间代价来换取时间。

问：顺序表、链表各自的使用场合？

答：顺序表适宜于做查找这样的静态操作；

链表宜于做插入、删除这样的动态操作。

若线性表的长度变化不大，且其主要操作是查找，则采用顺序表；

若线性表的长度变化较大，且其主要操作是插入、删除操作，则采用链表。

- 线性表的概念
- 线性表的顺序表示和实现
- 线性表的链式表示和实现
- 各种存储类型之比较
- 线性表的应用

一元多项式的表示及相加

符号多项式的表示及其操作是线性表处理的典型用例。

一个一元多项式 $P_n(x)$ 可以表示为： $P_n(x)=p_0+p_1x+p_2x^2+\ldots+p_nx^n$ (最多有 $n+1$ 项)**它由 $n+1$ 个系数唯一确定。**

因此可用一个线性表 P 来表示：$P = (p_0, p_1, p_2, \ldots, p_n)$

每一项的指数 i 隐含在其系数 p_i 的序号里。

假设 $Q_m(x)$ 是一元 m 次多项式，同样可用线性表 Q 表示：

$$Q = (q_0, q_1, q_2, \ldots, q_m)$$

若 $m < n$，则两个多项式相加的结果 $R_n(x)= P_n(x)+ Q_m(x)$ 可用线性表 R 来表示：

$$R = (p_0+ q_0, p_1+ q_1, p_2+q_2, \ldots, p_m+ q_m, p_{m+1}, \ldots, p_n)$$

只存储系数的方案对存在大量零系数的多项式并不适用。

例如： $S(x) = 1 + 3x^{10000} + 2x^{20000}$

要用一个长度为 20001 的线性表来表示，表中仅有 3 个非零系数，会浪费大量存储空间。

若只存储非零系数项，则必须同时存储相应的指数。

一般一元 n 次多项式 (只表示非零系数项) 可写成：

$$P_n(x) = p_1 x^{e_1} + p_2 x^{e_2} + \Lambda + p_m x^{e_m}$$

其中 $p_i \neq 0$ (i =1, 2, …, m)，$n = e_m > e_{m-1} > \ldots > e_1 \geq 0$

用一个长度为 m 且每个数据元素有两个数据项（系数项和指数项）的线性表 $((p_1, e_1), (p_2, e_2), \ldots, (p_m, e_m))$ 便可唯一确定多项式 $P_n(x)$。 对于 $S(x)$ 类的多项式将大大节省空间。

一元多项式存储结构
- 顺序存储结构
- 链式存储结构

究竟采用拿一种？

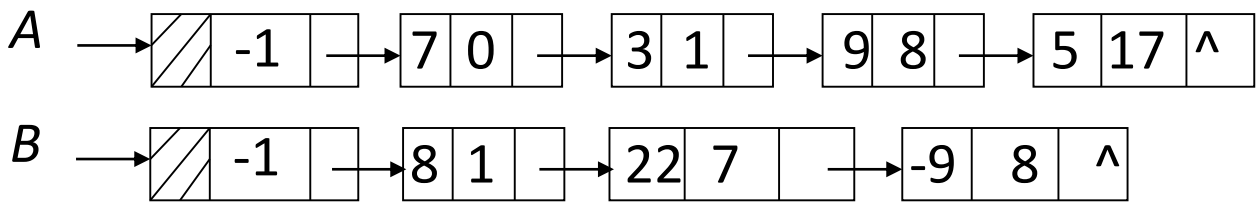
根据运算确定
- 不改变系数和指数的运算，采用顺序存储结构；
- 改变系数和指数的运算，采用链式存储结构。

例：假设多项式 $A_{17}(x)=7+3x+9x^8+5x^{17}$

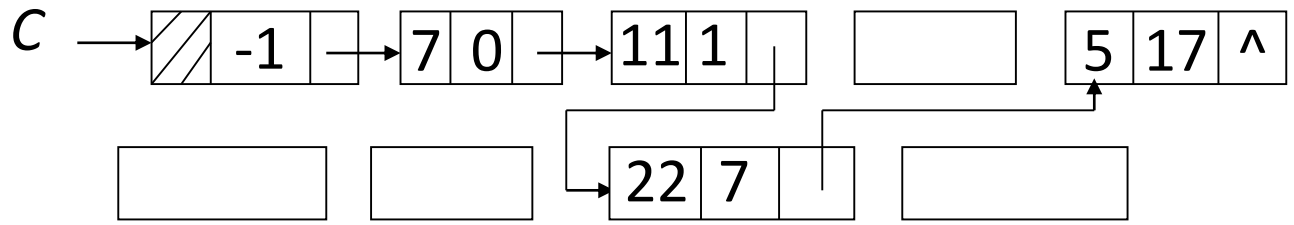
与 $B_8(x)=8x+22x^7-9x^8$

已经用单链表表示，其头指针分别为 A 与 B，

如下图所示。



将两个多项式相加为 $C_{17}(x)=7+11x+22x^7+5x^{17}$



一元多项式抽象数据类型的动态链式表示

```
typedef struct{
        float  coef; //系数
        int exp;       //指数
}term, ElemType;

typedef  LinkList  Polynomal;


Polynomal  pl;
```

操作举例：构造多项式

```
void  CreatePolyn(Polynomal &p,int m){
      //构建一个有序链表p，其中元素为结构体，有m个元素
       InitList(p);  h=p;
        e.coef=0.0;
        e.expn=-1;
        SetCurElem(h,e);//设置头节点的数据元素
        for(i=1;i<m+1;i++){
          向结构体变量e中输入系数和指数；
        if(!LocateElem(p,e,q,(*cmp)())){
              if(MakeNode(s,e))//生成节点s;s是指针变量
                    InsFirst(q,s);//将s节点插在q节点之前
          }
    }
}
```

小结

- 线性表的概念
- 线性表ADT
- 线性表的顺序表示和实现
- 线性表的链式表示和实现
- 各种存储类型之比较
- 线性表的应用

Thank You !