

## 第二章 线性表

2022-03-05 23:05

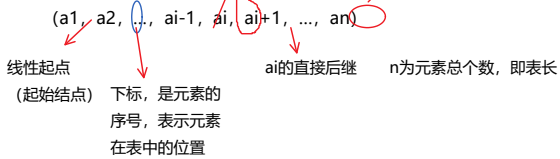
本章重点:

1. 线性表的定义和特点
2. 案例引入
3. 线性表的类型定义
4. 线性表的顺序表示和实现
5. 线性表的链式表示和实现
6. 顺序表和链表的比较
7. 线性表的应用
8. 案例分析与实现

### 一、线性表的定义和应用

#### 1. 定义

线性表是具有相同特性的数据元素的一个有限序列



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

- 其中数据元素的个数 $n$ 定义为表的长度。
- 当 $n=0$ 时称为空表。
- 将非空的线性表 ( $n > 0$ ) 记作:  $(a_1, a_2, \dots, a_n)$
- 这里的数据元素 $a_i$ 只是一个抽象的符号, 其具体含义在不同的情况下可以不同。

例子:

例1 分析26个英文字母组成的英文表  
(A, B, C, D, ..., Z)  
数据元素都是字母; 元素间关系是线性

例2 分析学生情况登记表

- 某单位历年拥有计算机的数量 (6, 17, 28, 50, 92, 188)
- 12星座 (白羊座、金牛座、双子座、巨蟹座、狮子座、处女座、天秤座、天蝎座、射手座、摩羯座、水瓶座、双鱼座)

同一线性表中的元素必定具有相同特性, 数据元素间的关系是线性关系

线性表的逻辑特征

- 在非空的线性表, 有且仅有一个开始结点 $a_1$ , 它没有直接前驱, 而有且仅有一个直接后继 $a_2$ ;
- 有且仅有一个终端结点 $a_n$ , 它没有直接后继, 而仅有一个直接前驱 $a_{n-1}$ ;
- 其余内部结点 $a_i (1 < i < n)$ 都有且仅有一个直接前驱 $a_{i-1}$ 和一个直接后继 $a_{i+1}$ 。
- 线性表是一种典型的数据结构

### 二、案例引入

【案例一】一元多项式的运算: 实现两个多项式加、减、乘运算

$P_n(x) = p_0x^0 + p_1x^1 + p_2x^2 + \dots + p_nx^n$

线性表  $P = (p_0, p_1, p_2, \dots, p_n)$   
(每一项的指数  $i$  隐含在其系数  $p_i$  的序号中)

例如:  $P(x) = 10 + 5x - 4x^2 + 3x^3 + 2x^4$

指数 (下标 <i>i</i> )	0	1	2	3	4
系数 $p[i]$	10	5	-4	3	2

用数组来表示

$R_n(x) = P_n(x) + Q_m(x)$  EASY

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

稀疏多项式  
 $S(x) = 1 + 3x^{10000} + 2x^{20000}$

将会造成存储空间很大的浪费, 怎么办?

【案例二】稀疏多项式

多项式非零项的数组表示

(a)  $A(x) = 7 + 3x + 9x^8 + 5x^{17}$  (b)  $B(x) = 8x + 22x^7 - 9x^8$

下标i	0	1	2	3
系数a[i]	7	3	9	5
指数	0	1	8	17

下标i	0	1	2
系数b[i]	8	22	-9
指数	1	7	8

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

线性表P = ((p1, e1), (p2, e2), ..., (pm, em))

线性表A = ((7, 0), (3, 1), (9, 8), (5, 17))

线性表B = ((8, 1), (22, 7), (-9, 8))

线性表A = ((7, 0), (3, 1), (9, 8), (5, 17))

线性表B = ((8, 1), (22, 7), (-9, 8))

● 创建一个新数组c

● 分别从头遍历比较a和b的每一项

✓ 指数相同，对应系数相加，若其和不为零，则在c中增加一个新项

✓ 指数不相同，则将指数较小的项复制到c中

● 一个多项式已遍历完时，将另一个剩余项依次复制到c中即可

数组c多大合适呢？

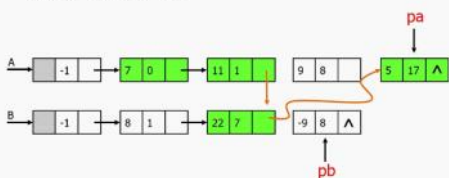
顺序存储结构存在的问题：

- 存储空间分配不灵活
- 运算的空间复杂度高

解决方法：→ 链式存储结构

$$A_{17}(x) = 7 + 3x + 9x^8 + 5x^{17}$$

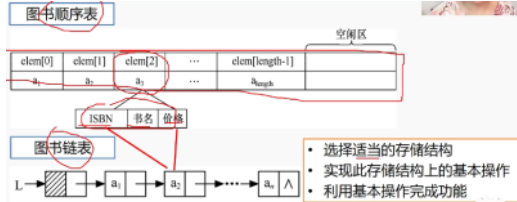
$$B_8(x) = 8x + 22x^7 - 9x^8$$



【案例三】图书、学生信息管理系统

需要的功能：查找、插入、删除、修改、排序、计数

- 图书表抽象为线性表
- 表中每本图书抽象线性表中数据元素



总结：

线性表中数据元素的类型可以为简单类型，也可以为复杂类型。

许多实际应用问题所涉及的基本操作有很大相似性，不应为每个具体应用单独编写程序

从具体应用中抽象出来共性的逻辑结构和基本操作（抽象数据类型），然后实现其存储结构和基本操作

### 三、线性表的类型定义

抽象数据类型线性表的定义如下：

ADT List

{

数据对象：D={ai | ai属于Elemset, (i=1,2,...,n,n>=0)}

数据关系：R={<ai-1, ai> | ai-1, ai属于D, (i=2,3,...,n)}

基本操作：

InitList(&L); DestroyList(&L);

ListInsert(&L,i,e); ListDelete(&L,i,&e);

....等

}ADT List

基本操作：

(一)

1) InitList(&L) (Initialization List)

- 操作结果：构造一个空的线性表L

2) DestroyList(&L)

- 初始条件：线性表L已经存在
- 操作结果：销毁线性表L

- 3) ClearList(&L)
  - 初始条件: 线性表L已经存在
  - 操作结果: 将线性表L重置为空表
- (二)
- 4) ListEmpty(L)
  - 初始条件: 线性表L已经存在
  - 操作结果: 若线性表L为空表, 则返回TURE; 否则返回FALSE。
- 5) ListLength(L)
  - 初始条件: 线性表L已经存在。
  - 操作结果: 返回线性表L中的数据元素个数
- (三)
- 6) GetElem(L,i,&e);
  - 初始条件: 线性表L已经存在,  $1 \leq i \leq \text{ListLength}(L)$
  - 操作结果: 用e返回线性表L中第i个数据元素的值。
- 7) LocateElem(L,e,compare())
  - 初始条件: 线性表L已经存在, compare()是数据元素判定函数
  - 操作结果: 返回L中第一个与e满足compare()是数据元素的位序。若这样的数据元素不存在则返回值为0。
- (四)
- 8) PriorElem(L, cur\_e,&pre\_e)

初始条件: 线性表L已经存在。

操作结果: 若cur\_e是L的数据元素, 且不是第一个, 则用pre\_e返回它的前驱否则操作失败; pre\_e无意义。
- 9) NextElem(L, cue\_e, &next\_e)

初始条件: 线性表L已经存在。

操作结果: 若cur\_e是L的数据元素, 且不是第最后个, 则用next\_e返回它的后继, 否则操作失败, next\_e无意义。
- (五)
- 10) ListInsert(&L,i,e)

初始条件: 线性表L已经存在,  $1 \leq i \leq \text{ListLength}(L)+1$

操作结果: 在L的第i个位置之前插入新的数据元素e, L的长度加一。

插入元素e之前(长度为n): (a1,a2,...,ai-1,ai,...,an)

插入元素e之后(长度为n+1): (a1,a2,...,ai-1,e,ai,...,an)
- (六)
- 11) ListDelete(&L,i,&e)

初始条件: 线性表L已经存在,  $1 \leq i \leq \text{ListLength}(L)$

操作结果: 删除L的第i个数据元素, 并用e返回其值, L的长度减一。

删除前 (长度为n)

(a1,a2,...,ai-1,ai,ai+1,...,an)

删除后 (长度为n-1)

(a1,a2,...,ai-1,ai+1,...,an)
- 12)ListTraverse(&L,visited())

初始条件: 线性表L已经存在

操作结果: 依次对线性表中的每个元素调用visited()

以上只是确定这些功能是什么, 至于如何做, 只有在确定了存储结构之后才考虑。

四、线性表的顺序表示和实现

在计算机中, 线性表有两种基本存储结构:

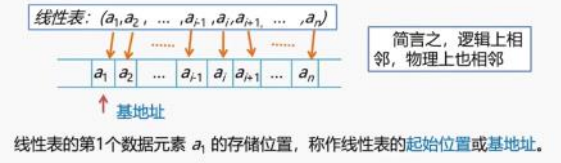
顺序存储结构和链式存储结构

本节先讨论顺序存储结构

1、线性表的顺序存储表示

线性表的顺序表示又称为顺序存储结构或顺序映像。

顺序存储定义: 把逻辑上相邻的数据元素存储在物理上相邻的存储单元中的存储结构。



2、顺序存储结构

例如: 线性表 (1, 2, 3, 4, 5, 6) 的存储结构:

1	2	3	4	5	6
---	---	---	---	---	---

是一个典型的线性表顺序存储结构。

若为下图:

1	2			3	4	5	6
---	---	--	--	---	---	---	---

依次存储, 地址连续——  
中间没有空出存储单元

地址不连续——

类C语言知识的补充:

一、顺序表类型的定义

顺序表类型定义

```
typedef struct {  
    ElemType data[];  
    int length;  
} SqList; //顺序表类型
```

```
typedef char ElemType;  
typedef int ElemType;  
  
typedef struct {  
    float p;  
    int e;  
}Polynomial;
```

若为下图：

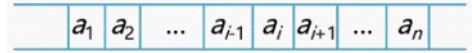
1	2			3	4	5	6
---	---	--	--	---	---	---	---

则不是一个线性表顺序存储结构

地址不连续——  
中间存在空的存储单元

线性表顺序存储结构占用一片连续的存储空间。知道了某个元素的存储位置就可以计算其他元素的存储位置。

### 3、线性表中元素存储位置的计算



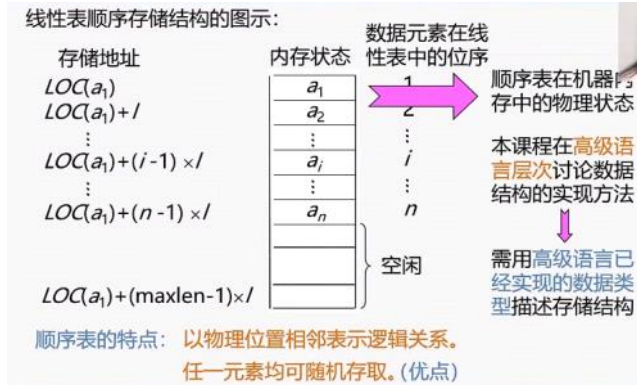
假设线性表的每个元素需占*l*个存储单元，则第*i*+1个数据元素的存储位置之间满足的关系为：

$$LOC(ai+1)=LOC(ai)+i$$

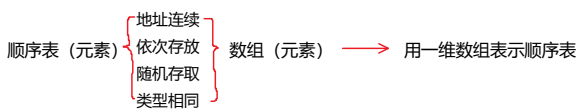
也可等于： $LOC(ai+1)=LOC(a1)+(i-1) \times l$

基地址

### 4、线性表顺序存储结构的图示：



### 5、顺序表的顺序存储表示



线性表长可变 (删除)

数组长度不可动态定义

一维数组的定义方式：

类型说明符 数组名 [ 常量表达式 ]

说明：常量表达式中可以包含常量和符号常量，不能包含变量。即C语言不允许对数组的大小作动态定义。

解决方法：用一变量表示顺序表的长度属性

```
#define LIST_INIT_SIZE 100 // 线性表存储空间的初始配置量
typedef struct {
    ElemType elem[LIST_INIT_SIZE];
    int length; // 当前长度
} SqList;
```

### 6、多项式是顺序存储结构类型定义

$P_n(x)=p_1*x^e1+p_2*x^e2+....+pm*x^em$

```
#define MAXSIZE 1000 // 多项式可能达到的最大长度
typedef struct {
    float p; // 多项式非零项的系数
    int e; // 指数
} Polynomial;

typedef struct {
    Polynomial *elem; // 存储空间基地址
    int length; // 多项式中当前项的个数
} SqList; // 多项式的顺序存储结构类型为SqList
```

### 7、图书表的顺序存储结构类型定义

```
#define MAXSIZE 10000 // 图书表可能达到的最大长度
typedef struct { // 图书信息定义
    char no[20]; // 图书ISBN
    char name[50]; // 图书名字
    float price; // 图书价格
} Book;

typedef struct {
    Book *elem; // 存储空间基地址
    int length; // 图书表中当前图书个数
} BookList;
```

```
ElemType data[];
int length;
} SqList; // 顺序表类型

typedef struct {
    float p;
    int e;
} Polynomial;

typedef struct {
    Polynomial *elem;
    int length;
} SqList;
```

Elem Type 表示对类型的重定义，方便对数据的更改

### 二、数组定义

数组静态分配

```
typedef struct {
    ElemType data[MaxSize];
    int length;
} SqList; // 顺序表类型
```

数组动态分配

```
typedef struct {
    ElemType *data;
    int length;
} SqList; // 顺序表类型
```

### 三、内存动态分配：

使用C语言的内存动态分配：

SqList L;

L.data=(ElemType\*)malloc(sizeof(Elem Type)\*MaxSize);

malloc(m)函数，开辟m字节长度的地址空间，并返回这段空间的首地址

sizeof (x) 运算，计算变量x的长度

Free(p)函数，释放指针p所指变量的存储空间，即彻底删除一个变量

引头文件：<stdlib.h>

使用c++的动态存储分配：

new 类型名T (初值列表)

功能：申请用于存放T类型对象的内存空间，并依初值列表赋以初值

结果值：成功：T类型的指针，指向新分配的内存；失败：0 (NULL)

```
int *p1= new int;
或 int *p1= new int(10);
```

delete 指针P

功能：释放指针P所指向的内存。P必须是new操作的返回值

```
delete p1;
```

### 四、c++中的参数传递

- 函数调用时传递给形参表的实参必须与形参三个一致：类型、个数、顺序
- 参数传递的两种方式：
  - 传值传递 (参数为整型、实型、字符型等)
  - 传地址：参数为指针变量、参数为引用类型、参数为数组名

#### 1、传值方式

把实参的值传递给函数局部工作区相应的副本中，函数使用这个副本执行必要的功能。

函数修改的副本的值，实参的值不变

```
#include <iostream.h>
void swap(float m,float n)
{float temp;
temp=m;
m=n;
n=temp;
}

void main()
{float a,b,*p1,*p2;
cin>>a>>b;
p1=&a; p2=&b;
swap(p1, p2);
cout<<a<<endl<<b<<endl;
}
```

#### 2、传地址方式—指针变量作参数

形参变化影响实参

```
#include <iostream.h>
void swap(float *m,float *n)
{float t;
t=*m;
*m=*n;
*n=t;
}

void main() {
    float a,b,*p1,*p2;
    cin>>a>>b;
    p1=&a; p2=&b;
    swap(p1, p2);
    cout<<a<<endl<<b<<endl;
}
```

形参变化不影响实参

```
#include <iostream.h>
void swap(float *m,float *n)
{float t;
t=*m;
*m=*n;
*n=t;
}

void main() {
    float a,b,*p1,*p2;
    cin>>a>>b;
    p1=&a; p2=&b;
    swap(p1, p2);
    cout<<a<<endl<<b<<endl;
}
```

#### 3、传地址方式—数组名做参数

传递的是数组的首地址



9787302111876	C语言程序设计	39
9787302104062	C语言程序设计	42
9787302104064	数据库原理	35
9787302104066	Java网络与案例	36
9787302104067	Java程序设计与应用案例	39
9787302104068	嵌入式操作系统及编程	33
9787302104069	软件测试	24
9787311231557	ELisp基础与应用	35

```

float price; //图书价格
}Book;

typedef struct{
    Book *elem; //存储空间的基地址
    int length; //图书表中当前图书个数
}SqList; //图书表的顺序存储结构类型为SqList

```

## 8、顺序表示意图



若为SqList \*L, 则变成—>

## 9、顺序表基本操作的实现

总结：

```

InitList(&L) //初始化操作, 建立一个空的线性表L
DestroyList(&L) //销毁已存在的线性表L
ClearList(&L) //将线性表清空
ListInsert(&L, i, e) //在线性表L中第i个位置元素, 用e返回
ListDelete(&L, i, e) //删除线性表L中第i个位置元素, 用e返回
IsEmpty(L) //若线性表为空, 返回true, 否则false
ListLength(L) //返回线性表L的元素个数
LocateElem(L, e) //L中查找与给定值e相等的元素, 若成功, 返回该元素在表中的序号
GetElem(L, i, &e) //将线性表L中的第i个位置元素返回给e

```

操作算法中用到的预定义常量和类型：

```

#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2
//Status 是函数的类型, 其值是函数结果状态代码
typedef int Status;
typedef char ElemType;

```

## 基本操作的实现：

### 1) 线性表L的初始化 (参数用引用)

```

Status InitList_Sq(SqList &L){ //构造一个空的顺序表L
    L.elem=new ElemType[MAXSIZE]; //为顺序表分配空间
    if(!L.elem) exit(OVERFLOW); //存储分配失败
    L.length=0; //空表长度为0
    return OK;
}

```

上面是使用了c++语法对其进行动态分配, 若想使用c语言的语法, 如下:  
L.elem = (ElemType\*) malloc (MAXSIZE\*sizeof(ElemType));

补充一: 销毁线性表L

```

void DestroyList(SqList &L){
    if (L.elem) delete L.elem; //释放存储空间
}

```

补充二: 清空线性表L

```

void ClearList(SqList &L){
    L.length=0; //将线性表的长度置为0
}

```

补充三: 求线性表L的长度

```

int GetLength(SqList L){
    return (L.length);
}

```

补充四: 判断线性表L是否为空

```

int IsEmpty(SqList L){
    if (L.length==0) return 1;
    else return 0;
}

```

## 3、传地址方式—数组名做参数

传递的是数组的首地址

对形参数组所做的任何改变都将反映到实参数组中

```

#include <iostream.h>
void sub(char b[]){
    b[] = "world";
}

void main (void ) {
    char a[10] = "hello";
    sub(a);
    cout << a << endl;
}

```

## 4、传地址方式—引用类型作参数

什么是引用—它用来给一个对象提供一个替代的名字。

```

#include <iostream.h>
void main(){
    int i=5;
    int &j=i;
    i=7;
    cout << "i=" << i << " j=" << j;
}

```

j是一个引用类型, 代表i的一个替代名  
i值改变时, j值也跟着改变, 所以会输出  
i=7 j=7

```

#include <iostream.h>
void swap(float &m, float &n){
    float temp;
    temp=m;
    m=n;
    n=temp;
}

void main(){
    float a,b;
    cin >> a >> b;
    swap(a,b);
    cout << a << endl << b << endl;
}

```

三点说明：

- 1) 传递引用给函数与传递指针的效果是一样的, 形参变化实参也发生变化。
- 2) 引用类型作形参, 在内存中并没有产生实参的副本, 它直接对实参操作; 而一般变量作参数, 形参与实参就占用不同的存储单元, 所以形参变量的值是实参变量的副本。因此, 当参数传递的数据量较大时, 用引用比用一般变量传递参数的时间和空间效率都好。
- 3) 指针参数虽然也能达到与使用引用的效果, 但在被调函数中需要重复使用“\*指针变量名”的形式进行运算, 这很容易产生错误且程序的阅读性较差; 另一方面, 在主调函数的调用处, 必须用变量的地址作为实参。

## 2) 顺序表的取值 (根据位置i获取相应位置数据元素的内容)

```
int GetElem(Sqlist L, int i, ElemType &e){
    if (i < 1 || i > L.length) return ERROR;
    //判断i值是否合理, 若不合理, 返回ERROR
    e = L.elem[i-1]; //第i-1的单元存储着第i个数据
    return OK;
}
```

按值查找: 在图书表中, 按照给定书号进行查找, 确定是否存在该图书  
如果存在输出第几个元素, 如果不存在, 输出0.

## 3) 顺序表的查找

>在线性表L中查找与指定值e相同的数据元素的位置

>从表的一端开始, 逐个进行记录的关键字和给定值的比较。找到, 返回该元素的位置序号, 未找到, 返回0。

for循环

```
int LocateElem(Sqlist L, ElemType e){
    //在线性表L中查找值为e的数据元素, 返回其序号 (是第几个元素)
    for (i=0; i < L.length; i++)
        if (L.elem[i] == e) return i+1; //查找成功, 返回序号
    return 0; //查找失败, 返回0
}
```

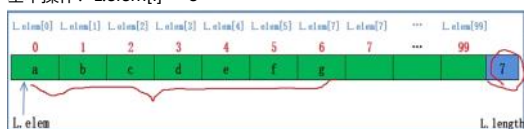
while循环

```
int LocateElem(Sqlist L, ElemType e){
    //在线性表L中查找值为e的数据元素, 返回其序号 (是第几个元素)
    i=0;
    while (i < L.length && L.elem[i] != e) i++;
    if (i < L.length) return i+1; //查找成功, 返回序号
    return 0; //查找失败, 返回0
}
```

算法分析:

>将记录的关键字同给定值进行比较

基本操作:  $L.elem[i] == e$



>平均查找长度ASL (Average Search Length) :

为确定记录在表中的位置, 需要与给定值进行比较的关键字的个数的期望值叫做查找算法的平均查找长度。

对含有  $n$  个记录的表, 查找成功时: 找到第  $i$  个记录需比较的次数。

$$ASL = \sum_{i=1}^n P_i \cdot i$$

第  $i$  个记录被查找的概率。

顺序查找的平均查找长度:

$$ASL = P_1 + 2P_2 + \dots + (n-1)P_{n-1} + nP_n = \frac{1}{n} (1 + 2 + \dots + n) = \frac{n(n+1)}{2}$$

假设每个记录的查找概率相等:  $P_i = 1/n$

则:  $ASL_{ss} = \sum_{i=1}^n \frac{1}{n} \cdot i = \frac{1}{n} \sum_{i=1}^n i = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}$

## 4) 线性表的插入



插入不同位置的算法演示:

- 插入位置在最后: 1 2 3 4 5 6 7
- 插入位置在中间: 1 2 3 4 7 5 6
- 插入位置在最前面: 7 1 2 3 4 5 6

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

算法思想:

- ①判断插入位置  $i$  是否合法。
- ②判断顺序表的存储空间是否已满, 若满返回ERROR。
- ③将第  $n$  至第  $i$  位的元素依次向后移动一个位置, 空出第  $i$  个位置。
- ④将要插入的新元素  $e$  放入第  $i$  个位置,
- ⑤表长加1, 插入成功返回OK。

算法实现:

```

Status ListInsert_Sq(SqList &L,int i,ElemType e){
① if(i<1 || i>L.length+1) return ERROR; //i值不合法
② if(L.length==MAXSIZE) return ERROR; //当前存储空间已满
③ for(j=L.length-1;j>=i-1;j--)
    L.elem[j+1]=L.elem[j]; //插入位置及之后的元素后移
④ L.elem[i]=e; //将新元素e放入第i个位置
⑤ L.length++; //表长增1
return OK;
}

```

算法分析:

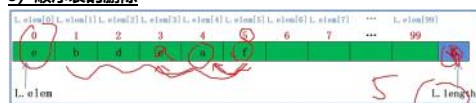
>算法时间主要耗费在移动元素的操作上

- >若插入在尾结点之后, 则根本无需移动 (特别快)
- >若插入在首结点之前, 则表中元素全部后移 (特别慢)
- >若要考虑在各种位置插入 (共n+1种可能) 的平均移动次数, 计算如下

$$E_{ins} = \frac{1}{n+1} \sum_{i=1}^{n+1} (n-i+1) = \frac{1}{n+1} (n + \dots + 1 + 0)$$

$$= \frac{1}{(n+1)} \frac{n(n+1)}{2} = \frac{n}{2}$$

## 5) 顺序表的删除



删除算法演示:

- 删除位置在最后: 1 2 3 4 5
- 删除位置在中间: 1 2 3 5 6
- 删除位置在最前面: 2 3 4 5 6

线性表的删除运算是指将表的第i (1<=i<=n) 个结点删除,

使长度为n的线性表 (a1, ..., ai-1, ai, ai+1, ..., an)

变成长度为n-1的线性表 (a1, ..., ai-1, ai+1, ..., an)

算法思想:

- ①判断删除位置i是否合法 (合法值为1<=i<=n)
- ②将欲删除的元素保留在e中。
- ③将第i+1至第n位的元素依次向前移动一个位置。
- ④表长减1, 删除成功返回OK。

算法实现:

```

Status ListDelete_Sq(SqList &L,int i){
① if((i<1)||(i>L.length)) return ERROR; //i值不合法
② for (j=i;j<=L.length-1;j++)
    L.elem[j-1]=L.elem[j]; //被删除元素之后的元素前移
③ L.length--; //表长减1
return OK;
}

```

算法分析:

>算法时间主要耗费在移动元素的操作上

- >若删除尾结点, 则根本无需移动 (特别快);
- >若删除首结点, 则表中n-1个元素全部前移 (特别慢);
- >若要考虑在各种位置删除(共n种可能)的平均移动次数, 该如何计算?

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

>顺序表删除算法的平均时间复杂度为O(n)

## 总结:

### 顺序表的操作算法分析

>时间复杂度

- >查找、插入、删除算法的平均时间复杂度为O(n)

>空间复杂度

- >显然, 顺序表操作算法的空间复杂度S(n)=O(1)  
(没有占用辅助空间)

### 顺序表优缺点

优点:

- >存储密度大 (结点本身所占存储量/结点结构所占存储量)
- >可以随机存取表中任一元素

缺点:

- >在插入、删除某一元素时, 需要移动大量元素
- >浪费存储空间

>属于静态存储形式，数据元素的个数不能自由扩充

## 五、线性表的链式表示和实现

### 1、链式表示基本概念

>链式存储结构

结点在存储器中的位置是任意的，即逻辑上相邻的数据元素在物理上不一定相邻

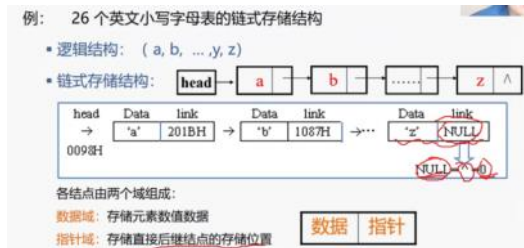
>线性表的链式表示又称为非顺序映像或链式映像。

- 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
- 链表中元素的逻辑次序和物理次序不一定相同

例一：



例二：



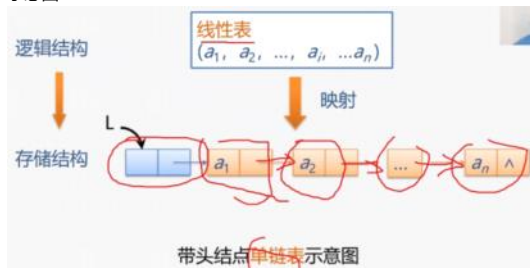
### 2、与链式存储有关的术语

1) **结点**：数据元素的存储映像。由数据域和指针域两个部分组成。



2) **链表**：n个结点由指针链组成一个链表。它是线性表的链式存储映像，称为线性表的链式存储结构

示意图：

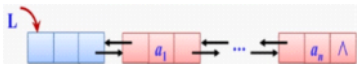


### 3、单链表、双链表、循环链表：

>结点只有一个指针域的链表，称为单链表或线性链表



>结点有两个指针域的链表，称为双链表

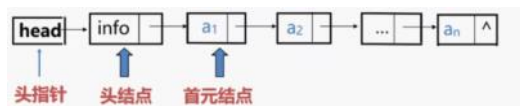


>首尾相接的链表称为循环链表



### 4、头指针、头结点及首元结点



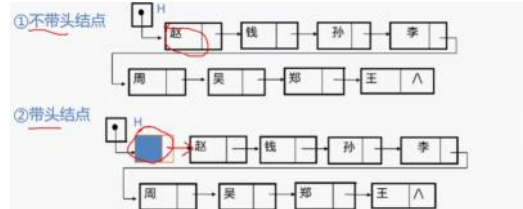


头指针：是指向链表中第一个结点的指针

首元结点：是指链表中存储第一个数据元素 $a_1$ 的结点

头结点：是在链表的首元结点之前附设的一个结点。

链表的存储结构示意图有以下两种形式：



## 补充：

【1】如何表示空表？

>无头结点时，头指针为空时表示空表

>有头结点时，头结点的指针域为空时表示空表

【2】在链表中设置头结点有什么好处

>便于首元结点的处理：首元结点的地址保存在头结点的指针域中，所以在链表的第一个位置上的操作和其他位置一致，必须进行特殊处理；

>便于空表和非空表的统一处理。无论链表是否为空，头指针都是指向头结点的非空指针，因此空表和非空表的处理也就统一了。

【3】头结点的数据域内装的是什么？

>头结点的数据域可以为空，也可存放线性表长度等附加信息，但此结点不能计入链表长度值。



## 5、链表（链式存储结构）的特点

>结点在存储器中的位置是任意的，即逻辑上相邻的数据元素在物理上不一定相邻。

>访问时只能通过头指针进入链表，并通过每个结点的指针域依次向后顺序扫描扫描其余结点，所以寻找第一个结点和最后一个结点所花费的时间不等。

顺序表——>随机存取

链表——>顺序存取

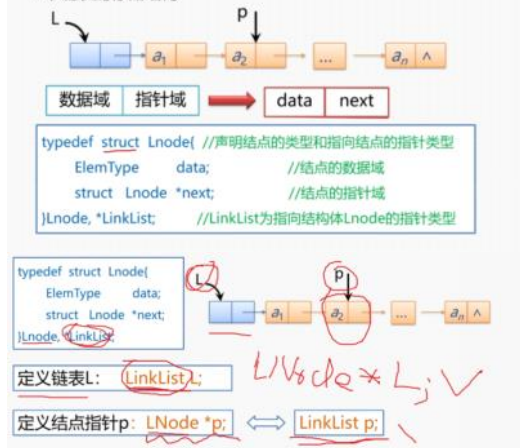
这种方法被称为顺序存取法

## 6、带头结点的单链表

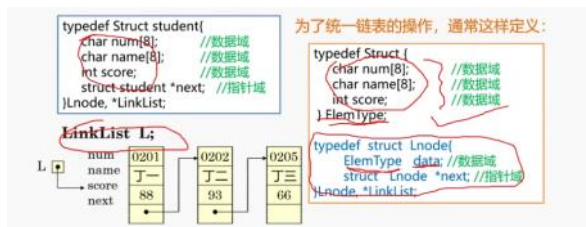


单链表是由表头唯一确定，因此单链表可以用头指针的名字来命名。若头指针名是 $L$ ，则把链表称为表 $L$ 。

▪ 单链表的存储结构



例题：存储学生学号、姓名、成绩的单链表结点类型定义如下：



## 7. 单链表基本操作的实现

【操作一】单链表的初始化（带头结点的单链表）

> 即构造一个如图的空表



【算法步骤】

- 1) 生成新结点作头结点，用头指针L指向头结点。
- 2) 将头结点的指针域置空。

【算法描述】

```
typedef struct LNode{  
    ElemType data;  
    struct LNode *next;  
}LNode, *LinkList;
```

```
Status InitList_L(LinkList &L){  
    L=new LNode; //或L= (LinkList) malloc ( sizeof (LNode));  
    L->next=NULL;  
    return OK;  
}
```

补充算法1——判断链表是否为空

空表：链表中无元素，称为空链表（头指针和头结点仍然存在）

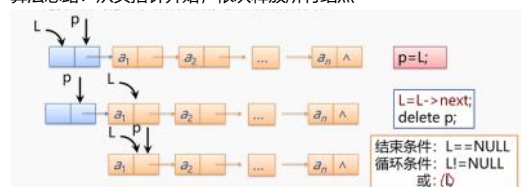
算法思路：判断头结点指针域是否为空

算法描述：

```
int ListEmpty(LinkList L){ //若L为空表，则返回1，否则返回0  
    if(L->next) //非空  
        return 0;  
    else  
        return 1;  
}
```

补充算法2——单链表的销毁：链表销毁后不存在

算法思路：从头指针开始，依次释放所有结点



算法描述：

```
Status DestroyList_L(LinkList &L){ //销毁单链表L  
    LNode *p; //或LinkList p;  
    while(L){  
        p=L;  
        L=L->next;  
        delete p;  
    }  
    return OK;  
}
```

补充算法3——清空链表：

链表仍然存在，但链表中无元素，成为空链表（头指针和头结点仍然存在）

算法思路：依次释放所有结点，并将头结点指针域设置为空。



算法描述：

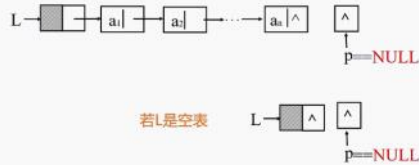
```

Status ClearList(LinkList & L){ // 将L重置为空表
    LNode *p,*q; //或LinkList p,q;
    p=L->next;
    while(p) { //没到表尾
        q=p->next;
        delete p;
        p=q;
    }
    L->next=NULL; //头结点指针域为空
    return OK;
}

```

#### 补充算法4——求单链表的表长

算法思路：从首元结点开始，依次计数所有结点



算法描述：

```

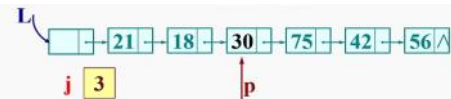
int ListLength_L(LinkList L){ //返回L中数据元素个数
    LinkList p;
    p=L->next; //p指向第一个结点
    i=0;
    while(p) { //遍历单链表,统计结点数
        i++;
        p=p->next;
    }
}

```

#### 【操作二】取值：取单链表中第i个元素的内容

算法思路：

假设分别取出表中第3个元素和第15个元素  $i=3$



从链表的头指针出发，顺着链域next逐个结点往下搜索，直至搜索到第i个结点为止。因此，链表不是随机存取结构

算法步骤：

1. 从第1个结点 ( $L \rightarrow next$ ) 顺链扫描，用指针p指向当前扫描到的结点，p初值  $p = L \rightarrow next$ 。
2. j 做计数器，累计当前扫描过的结点数，j 初值为1。
3. 当 p 指向扫描到的下一结点时，计数器 j 加1。
4. 当  $j == i$  时，p所指的结点就是要找的第 i 个结点。

算法描述：

```

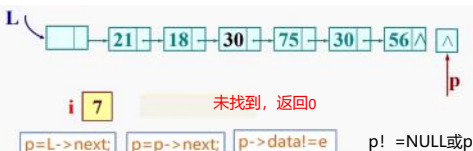
Status GetElem_L(LinkList L, int i, ElemType &e){ //获取线性表L中的
    某个数据元素的内容，通过变量e返回
    p=L->next; j=1; //初始化
    while(p && j < i){ //向后扫描，直到p指向第i个元素或p为空
        p=p->next; ++j;
    }
    if(!p || j > i) return ERROR; //第i个元素不存在
    e=p->data; //取第i个元素
    return OK;
} //GetElem_L

```

#### 【操作三】按值查找——根据指定数据获取该数据所在的位置（地址）

算法思路：

分别查找值为30和值为15的元素



算法描述

```

LNode *LocateElem_L(LinkList L, Elemtype e) {
    //在线性表L中查找值为e的数据元素
    //找到，则返回L中值为e的数据元素的地址，查找失败返回NULL
    p=L->next;
    while(p && p->data != e)
        p=p->next;
    return p;
}

```

查找、插入、删除算法时间效率分析：

#### 1.查找

```

LNode *LocateElem_L(LinkList L, Elemtype e) {
    //在线性表L中查找值为e的数据元素
}

```

```

p=p->next;
return p;
}

```

【操作四】按值查找——根据指定数据获取该数据位置序号

算法描述:

```

//在线性表L中查找值为e的数据元素的位置序号
int LocateElem_L (LinkedList L, Elemtype e) {
//返回L中值为e的数据元素的位置序号, 查找失败返回0
p=L->next; j=1;
while(p && p->data!=e)
{p=p->next; j++;}
if(p) return j;
else return 0;
}

```

查找、插入、删除算法时间效率分析:

1.查找

```

Lnode *LocateElem_L (LinkedList L, Elemtype e) {
//在线性表L中查找值为e的数据元素
//找到, 则返回L中值为e的数据元素的地址, 查找失败返回NULL
p=L->next;
while(p && p->data!=e)
p=p->next;
return p;
}

```

>因线性链表只能顺序存取, 即在查找时要从头指针找起, 查找的时间复杂度为 $O(n)$

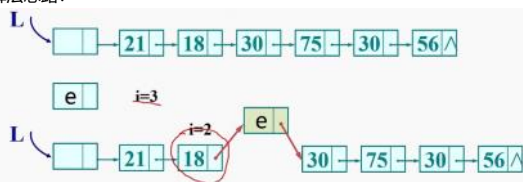
2. 插入和删除

>因线性链表不需要移动元素, 只要修改指针, 一般情况下时间复杂度为 $O(1)$

>但是, 如果要在单链表中进行前插或删除操作, 由于要从头查找前驱结点, 所耗时间复杂度为 $O(n)$ 。

【操作五】插入——在第i个结点前插入值为e的新结点

算法思路:



算法步骤:

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  $s$ 。
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

①  $s \rightarrow \text{next} = p \rightarrow \text{next};$     ②  $p \rightarrow \text{next} = s;$

步骤①与步骤②不能互换, 因为先执行步骤②会丢失掉 $a_i$ 的地址。

算法描述:

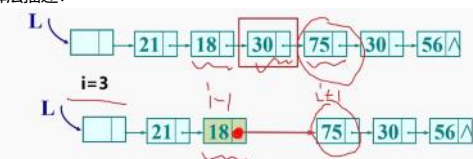
```

//在L中第i个元素之前插入数据元素e
Status ListInsert_L(LinkedList &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, p指向i-1结点
if(!p || j>i-1) return ERROR; //大于表长+1或者小于1, 插入位置非法
s=new LNode; s->data=e; //生成新结点s, 将结点s的数据域置为e
s->next=p->next; //将结点s插入L中
p->next=s;
return OK;
} //ListInsert_L

```

【操作六】删除——删除第i个结点

算法描述:



算法步骤:

1. 首先找到  $a_{i-1}$  的存储位置  $p$ , 保存要删除的 $a_i$ 的值。
2. 令  $p \rightarrow \text{next}$  指向  $a_{i+1}$ 。
3. 释放结点  $a_i$  的空间。

$p \rightarrow \text{next} = p \rightarrow \text{next} \rightarrow \text{next};$

算法描述:

```

//将线性表L中第i个数据元素删除
Status ListDelete_L(LinkedList &L, int i, ElemType &e) {
p=L; j=0;
while(p->next && j<i-1) { p=p->next; ++j; } //寻找第i个结点, 并令p指向其前驱
if(!p->next || j>i-1) return ERROR; //删除位置不合理
q=p->next; //临时保存被删结点的地址以备释放
p->next=q->next; //改变删除结点前驱结点的指针域
e=q->data; //保存删除结点的数据域
delete q; //释放删除结点的空间
return OK;
} //ListDelete_L

```

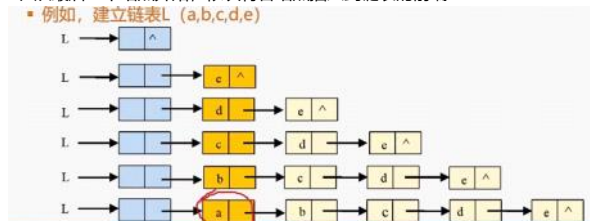
【操作七】建立单链表: 头插法——元素插入在链表头部, 也叫前插法

算法描述:

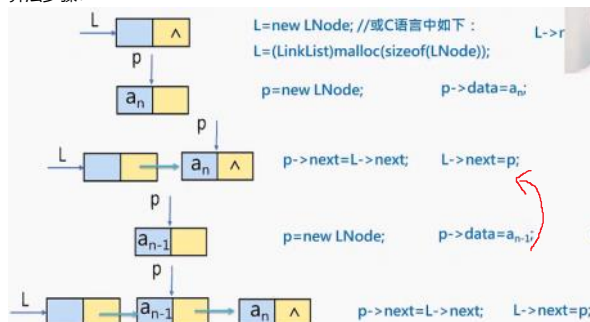
1. 从一个空表开始, 重复读入数据;
2. 生成新结点, 将读入数据存放到新结点的数据域中

3、从最后一个结点开始，依次将各结点插入到链表的前端

• 例如，建立链表L (a,b,c,d,e)



算法步骤:



算法描述:

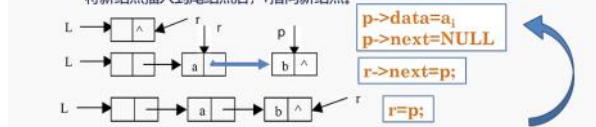
```
void CreateList_H(LinkList &L, int n){
    L = new LNode;
    L->next = NULL; //先建立一个带头结点的单链表
    for(i = n; i > 0; --i){
        p = new LNode; //生成新结点 p = (LNode*)malloc(sizeof(LNode));
        cin >> p->data; //输入元素值 scanf(&p->data);
        p->next = L->next; //插入到表头
        L->next = p;
    }
} //CreateList_H
```

算法的时间复杂度为  $O(n)$

【操作八】建立单链表：尾插法——元素插入在链表的尾部，也叫后插法

算法分析:

1. 从一个空表L开始，将新结点逐个插入到链表的尾部，尾指针r指向链表的尾结点。
2. 初始时，r同L均指向头结点。每读入一个数据元素则申请一个新结点，将新结点插入到尾结点后，r指向新结点。



算法描述:

//正位序输入n个元素的值，建立带头结点的单链表L

```
void CreateList_R(LinkList &L, int n){
    L = new LNode; L->next = NULL;
    r = L; //尾指针r指向头结点
    for(i = 0; i < n; ++i){
        p = new LNode; cin >> p->data; //生成新结点，输入元素值
        p->next = NULL;
        r->next = p; //插入到表尾
        r = p; //r指向新的尾结点
    }
} //CreateList_R
```

算法的时间复杂度是  $O(n)$

## 8. 循环链表

1) 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。

注意：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。

循环条件:

$p \neq \text{NULL} \rightarrow p \neq L$

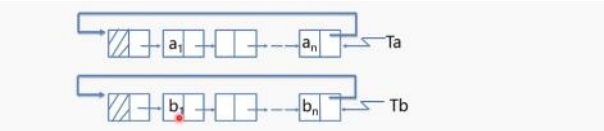
$p \rightarrow \text{next} \neq \text{NULL} \rightarrow p \rightarrow \text{next} \neq L$

单链表

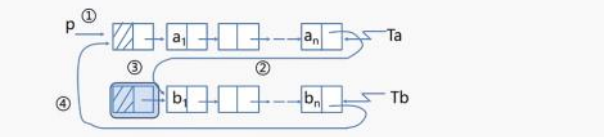
单循环链表



2) 带尾指针循环链表的合并 (将Tb合并在Ta之后)



操作分析:  
>p存表头结点 >Tb表头连接到Ta表尾 >释放Tb表头结点 >修改指针  
p=Ta->next; Ta->next=Tb->next->next; delete Tb->next; Tb->next=p;



算法描述:

```
LinkList Connect(LinkList Ta, LinkList Tb){
    //假设Ta、Tb都是非空的单循环链表
    p=Ta->next; //①p存表头结点
    Ta->next=Tb->next->next; //②Tb表头连接Ta表尾
    delete Tb->next; //③释放Tb表头结点
    Tb->next=p; //④修改指针
    return Tb;
}
```

时间复杂度是  $O(1)$ 。

9. 双向链表

为什么要讨论双向链表:

单链表的结点→有指示后继的指针域→找后继结点方便;  
即: 查找某结点的后继结点的执行时间为  $O(1)$ 。  
→无指示前驱的指针域→找前驱结点难: 从表头找单链表的这种缺点。  
即: 查找某结点的前驱结点的执行时间为  $O(n)$ 。  
可用双向链表来克服  
双向链表: 在单链表的每个结点里再增加一个指向其直接前驱的指针域 prior, 这样链表中就形成了有两个方向不同的链, 故称为双向链表。

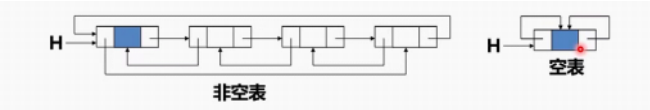
双向链表的结构可定义如下:

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

双向链表结点结构

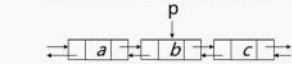
双向循环链表:

与单链的循环表类似, 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



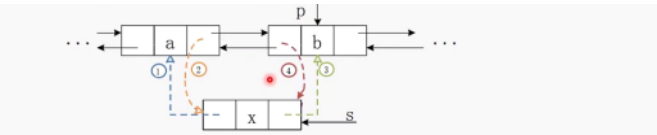
双向链表的对称性 (设指针p指向某一点):

p->prior->next=p=p->next->prior  
在双向链表中有些操作 (如: ListLength、GetElem等), 因仅涉及一个方向的指针, 故它们的算法与线性链表的相同。但在插入、删除时, 则需同时修改两个方向上的指针, 两者的操作的时间复杂度均为  $O(n)$ 。



【操作一】双向链表的插入

算法分析:



1.s->prior=p->prior;

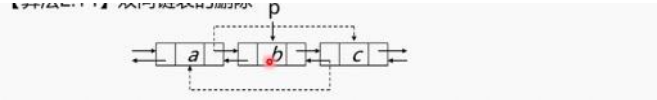
```
2.p->prior-next=s;
3.s->next=p;
4.p->prior=s;
```

算法描述:

```
void ListInsert_DuL(DuLinkList &L, Int i, ElemType(e)){
    //在带头结点的双向循环链表 L 中第 i 个位置之前插入元素 e
    if(!((p=GetElemP_DuL(L,i))) return ERROR;
    s=new DuLNode;    s->date = e;
    ① s->prior = p->prior; ② p->prior->next = s;
    ③ s->next = p;        ④ p->prior = s;
    return OK;
} // ListInsert_DuL
```

【操作二】双向链表的删除

算法分析:



```
1.p->prior->next=p->next;
2.p->next->prior=p->prior;
```

算法描述:

```
void ListDelete_DuL(DuLink &L, Int i, ElemType &e) {
    // 删除带头结点的双向循环链表 L 的第 i 个元素，并用 e 返回。
    if(!((p=GetElemP_DuL(L,i))) return ERROR;
    e = p->data;
    p->prior->next = p->next;
    p->next->prior = p->prior;
    free(p);
    return OK;
} // ListDelete_DuL
```

单链表、循环链表及双向链表的比较

时间效率:

	查找表头结点 (首元结点)	查找表尾结点	查找结点 * P 的前驱结点
带头结点的单链表 L	L->next 时间复杂度O(1)	从L->next依次向后遍历 时间复杂度O(n)	通过p->next无法找到其前驱
带头结点仅设头指针L的循环单链表	L->next 时间复杂度O(1)	从L->next依次向后遍历 时间复杂度O(n)	通过p->next可以找到其前驱 时间复杂度O(n)
带头结点仅设尾指针R的循环单链表	R->next 时间复杂度O(1)	R 时间复杂度O(1)	通过p->next可以找到其前驱 时间复杂度O(n)
带头结点的双向循环链表L	L->next 时间复杂度O(1)	L->prior 时间复杂度O(1)	p->prior 时间复杂度O(1)

链式存储结构的优点:

- > 结点空间可以动态申请和释放;
- > 数据元素的逻辑次序靠结点的指针来指示, 插入和删除时不需要移动数据元素


链式存储结构的缺点:

- > 存储密度小, 每个结点的指针域需额外占用存储空间。当每个结点的数据域所占字节不多时, 指针域所占存储空间的比重显得很大。
- > 链式存储结构是非随机存取结构。对任一结点的操作都要从头指针依指针链查找到该节点, 这增加了算法的复杂度。

存储密度:

存储密度是指结点数据本身所占的存储量和整个结点结构中所占的存储量之比,

存储密度 =  $\frac{\text{结点数据本身占用的空间}}{\text{结点占用的空间总量}}$

例如  存储密度=8/12=67%

一般地, 存储密度越大, 存储空间的利用率就越高。显然, 顺序表的存储密度为1 (100%), 而链表的存储密度小于1。

顺序表与链表的比较:

比较项目	顺序表	链表
空间	存储空间	存储空间
时间	存取元素	存取元素
插入、删除	插入、删除	插入、删除
适用情况	适用情况	适用情况

10、线性表的应用

1) 线性表的合并

> 问题描述:

假设利用两个线性表La和Lb分别表示两个集合A和B，现求一个新的集合A=AUB

La=(7, 5, 3, 11) Lb=(2, 6, 3)=>La=(7, 5, 3, 11, 2, 6)

>算法步骤:

依次取出Lb中的每个元素，执行以下操作:

- a.在La中查找该元素
- b.如果找不到，则将其插入La的最后

>算法描述:

```
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);
        if(!LocateElem(La, e)) ListInsert(&La, ++La_len, e);
    }
}
```

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

## 2) 有序表的合并

顺序表实现:

>问题描述:

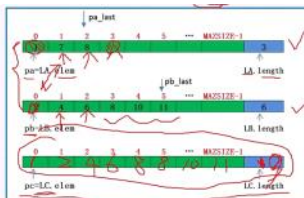
已知线性表La和Lb中的数据元素按值非递减有序排列，现要求将La和Lb归并为一个新的线性表Lc，且Lc中的数据元素仍按值非递减有序排列。

La=(1, 7, 8) Lb=(2, 4, 6, 8, 10, 11)=>Lc=(1, 2, 4, 6, 7, 8, 8, 10, 11)

>算法步骤

- a.创建一个空表Lc
- b.依次从La或Lb中“摘取”元素值较小的结点插入到Lc表的最后，直至其中一个表变为空表为止

c.继续将La或Lb其中一个表的剩余结点插入在Lc表的最后



>算法实现

```
void MergeList_Sq(SqList LA, SqList LB, SqList &LC){
    pa = LA.elem;
    pb = LB.elem;
    // 指针pa和pb的初值分别指向两个表的第一个元素
    LC.length = LA.length + LB.length;
    // 新表长度为符合并两表的长度之和
    LC.elem = new ElemType[LC.length];
    // 为合并后的新表分配一个数组空间
    pc = LC.elem;
    // 指针pc指向新表的第一个元素
    pa_last = LA.elem + LA.length - 1;
    // 指针pa_last指向LA表的最后一个元素
    pb_last = LB.elem + LB.length - 1;
    // 指针pb_last指向LB表的最后一个元素
    while(pa <= pa_last && pb <= pb_last){
        // 两个表都非空
        if(*pa <= *pb) *pc++ = *pa++;
        // 依次“摘取”两表中值较小的结点
        else *pc++ = *pb++;
    }
    while(pa <= pa_last) *pc++ = *pa++;
    // LA表已到达表尾，将LA中剩余元素加入LC
    while(pb <= pb_last) *pc++ = *pb++;
    // LB表已到达表尾，将LB中剩余元素加入LC
    // MergeList_Sq
}
```

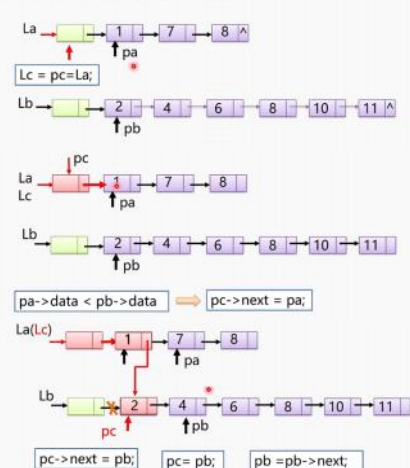
算法的时间复杂度为:  $O(\text{ListLength}(La) + \text{ListLength}(Lb))$

算法的空间复杂度为:  $O(\text{ListLength}(La) + \text{ListLength}(Lb))$

链表实现:

>算法描述

用La的头结点作为Lc的头结点

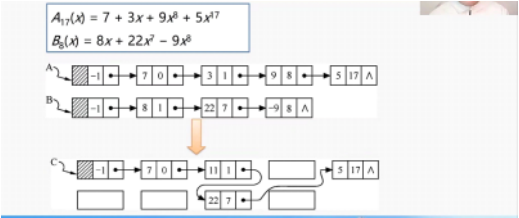
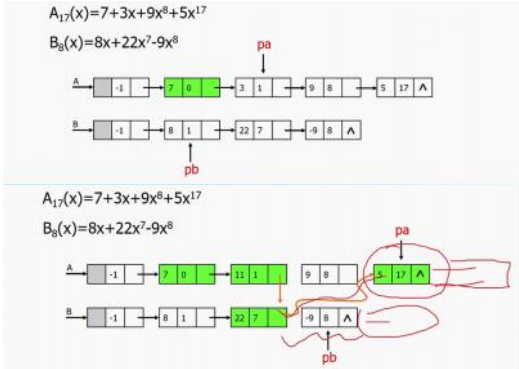




```
void CreatePolyn(Polynomial &P, int n){
    //输入m项的系数和指数，建立表示多项式的有序链表P
    P=new PNode;
    P->next=NULL;
    for(i=1;i<=n;i++){
        s=new PNode;
        cin>>s->coef>>s->expn;
        pre=P;
        q=P->next;
        while(q&&q->expn<s->expn){
            pre=q; q=q->next;
        }
        s->next=q;
        pre->next=s;
    }
}
```

多项式相加——

【算法步骤】



文字描述:

- ① 指针p1和p2初始化，分别指向Pa和Pb的头结点。
- ② p3指向和多项式的当前结点，初值为Pa的头结点。
- ③ 当指针p1和p2均未到达相应表尾时，则循环比较p1和p2所指结点对应的指数值 (p1->expn与p2->expn)，有下列3种情况：
  - 当p1->expn = p2->expn时，则将两个结点中的系数相加
    - 若和不为零，则修改p1所指结点的系数值，同时删除p2所指结点
    - 若和为零，则删除p1和p2所指结点；
  - 当p1->expn < p2->expn时，则应摘取p1所指结点插入到“和多项式”链表中；
  - 当p1->expn > p2->expn时，则应摘取p2所指结点插入到“和多项式”链表中；
- ④ 将非空多项式的剩余段插入到p3所指结点之后。
- ⑤ 释放Pb的头结点。

【案例二】图书管理系统

