

数据结构与算法

线性表

线性表

- 2. 1 线性表的逻辑结构
- 2. 2 线性表的顺序存储结构
- 2. 3 线性表的链式存储结构
 - 单链表
 - 循环链表
 - 双向链表
- 2. 4 一元多项式的表示

线性表的逻辑结构

- 最常用而且最简单的一种数据结构。
- 定义：是由 n ($n \geq 0$) 个类型相同的数据元素 a_1, a_2, \dots, a_n 组成的有限序列，记作 $(a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$ 。
- 这里的数据元素 a_i ($1 \leq i \leq n$) 只是一个抽象的符号。
- a_i 是线性表中数据元素， n 是线性表长度，其中 a_i 称为 a_{i+1} 的直接前驱，简称为前驱， a_{i+1} 称为 a_i 的直接后继，简称为后继。

线性表的逻辑结构

线性表的特点

- 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，集合中的每个数据元素只有一个前驱。
- 除最后一个之外，只有一个后继。



线性表的逻辑结构

- 在较为复杂的线性表中，数据元素 (data elements) 可由若干数据项组成。
- 如学生成绩表中，每个学生及其各科成绩是一个数据元素，它由学号、姓名、各科成绩及平均成绩等数据项 (item) 组成，常被称为一个记录 (record)。
- 含有大量记录的线性表称为文件 (file)。
- 数据对象 (data object) 是性质相同的数据元素集合。

线性表的逻辑结构

- 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
.....

线性表的逻辑结构

线性表的4个特点

- 同一性：线性表由同类数据元素组成，每一个 a_i 必须属于同一数据对象。
- 有穷性：线性表由有限个数据元素组成，表长度就是表中数据元素的个数。
- 有序性：线性表中相邻数据元素之间存在着序偶关系 $\langle a_i, a_{i+1} \rangle$ 。
- 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。

线性表的抽象数据类型定义

ADT LinearList {

 数据元素: $D = \{a_i \mid a_i \in D_0, i=1, 2, \dots, n, n \geq 0\}$, D_0 为某一
 数据对象}

 关系: $S = \{\langle a_i, a_{i+1} \rangle \mid a_i, a_{i+1} \in D_0, i=1, 2, \dots, n-1\}$

 基本操作:

 (1) int Length()

 初始条件: 线性表已存在。

 操作结果: 返回线性表元素个数。

 (2) bool Empty()

 初始条件: 线性表已存在。

 操作结果: 如线性表为空, 则返回true, 否则返回false。

线性表的抽象数据类型定义

(3) void Clear()

初始条件: 线性表已存在。

操作结果: 清空线性表。 ?

(4) void Traverse(void (*visit)(const ElemtType &))

初始条件: 线性表已存在。

操作结果: 依次对线性表的每个元素调用函数(*visit)。

(5) StatusCode GetElem(int position, ElemtType &e)

初始条件: 线性表已存在, $1 \leqslant \text{position} \leqslant \text{Length}()$ 。

操作结果: 用e返回第position个元素的值。

线性表的抽象数据类型定义

(6) StatusCode SetElem(int position, const ElemType &e)

初始条件：线性表已存在， $1 \leqslant position \leqslant Length()$ 。

操作结果：将线性表的第position个位置的元素赋值为e。

(7) StatusCode Delete(int position, ElemType &e)

初始条件：线性表已存在， $1 \leqslant position \leqslant Length()$ 。

操作结果：删除线性表的第position个位置的元素，并用e返回其值，长度减1。

(8) StatusCode Insert(int position, const ElemType &e)

初始条件：线性表已存在， $1 \leqslant position \leqslant Length() + 1$ 。

操作结果：在线性表的第position个位置前插入元素e，长度加1。

} ADT LinearList

线性表的顺序存储结构

- 线性表的顺序表示：将线性表中的元素相继存放在一个连续的存储空间中。
- 以元素在计算机内“物理位置相邻”来表示线性表中数据元素之间的逻辑关系，叫做线性表的顺序存储结构或顺序映像。这种存储结构的线性表叫顺序表。
- 顺序表的特点：可利用一维数组描述存储结构，采用线性表的顺序存储方式。

线性表的顺序存储结构

- 顺序表的每一个数据元素的存储位置都和开始位置相差一个与数据元素在表中的位序成正比的常数。
- 可以随机访问顺序表中任一数据元素，且访问时间相等。
- 顺序存储结构是一种随机存取的存储结构。
- 数组具有随机存取的特性，通常用数组来描述数据结构中的顺序存储结构。

线性表的顺序存储结构

- 假设线性表中有n个元素，每个元素占k个单元，第一个元素的地址为 $\text{loc}(a_1)$ ，则可以通过如下公式计算出第i个元素的地址 $\text{loc}(a_i)$ ：

$$\text{loc}(a_i) = \text{loc}(a_1) + (i-1) \times k$$

其中 $\text{loc}(a_1)$ 称为基地址

线性表的顺序存储结构

存储地址	内存空间状态	逻辑地址
$\text{loc}(a_1)$	a_1	1
$\text{loc}(a_1) + k$	a_2	2
...
$\text{loc}(a_1) + (i-1)k$	a_i	i
...
$\text{loc}(a_1) + (n-1)k$	a_n	n
		空闲

线性表的顺序存储结构

- C语言中，由于其一维数组也是采用顺序存储表示，故可以用数组类型来描述顺序表。又因为除了用数组来存储线性表的元素之外，顺序表还应该用一个变量来表示线性表的长度属性，所以通常用结构类型来定义顺序表类型。
- C++的顺序表(SeqList)类的定义如下。

线性表的顺序存储结构

// 顺序表类模板

```
template <class ElemtType>
```

```
class SqList
```

```
{
```

```
protected:
```

// 顺序表实现的数据成员:

```
int count; // 元素个数
```

```
int maxSize; // 顺序表最大元素个数
```

```
ElemtType *elems; // 元素存储空间
```

// 辅助函数模板

```
bool Full() const; // 判断线性表是否已满
```

```
void Init(int size); // 初始化线性表
```

线性表的顺序存储结构

```
public:  
    // 抽象数据类型方法声明及重载编译系统默认方法声明:  
    SqList(int size = DEFAULT_SIZE);  
    virtual ~SqList();  
    int Length();  
    bool Empty();  
    void Clear();  
    void Traverse(void (*visit)(const ElemtType &));  
    StatusCode GetElem(int position, ElemtType &e);  
    StatusCode SetElem(int position, const ElemtType &e); // // 设置指定位置的元素值  
    StatusCode Delete(int position, ElemtType &e); // // 删除元素  
    StatusCode Insert(int position, const ElemtType &e); // // 插入元素  
};
```

// 构造函数模板
// 析构函数模板
// 求线性表长度
// 判断线性表是否为空
// 将线性表清空
// 遍历线性表
// 求指定位置的元素
// 设置指定位置的元素值
// 删除元素
// 插入元素

线性表的顺序存储结构

```
// 辅助函数模板的实现
template <class ElemtType>
bool SqList<ElemtType>::Full()
{
    // 操作结果：如线性表已满，则返回true，否则返回false
    return count == maxSize;
}
```

线性表的顺序存储结构

// 辅助函数模板的实现

```
template <class ElemtType>
void SqList<ElemtType>::Init(int size)
{
```

// 操作结果：初始化最大元素个数为size的空线性表

```
maxSize = size;           // 最大元素个数
if (elems != NULL) delete []elems; // 释放存储空间
elems = new ElemtType[maxSize]; // 分配存储空间
count = 0;                // 空线性表元素个数为0
}
```

线性表的顺序存储结构

- 顺序表上实现的基本操作：在顺序表存储结构中，很容易实现线性表的构造、第 i 个元素的访问。
 - 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.data[i-1]。
 - 这种存储结构，随机存取某个元素比较容易实现。

线性表的顺序存储结构

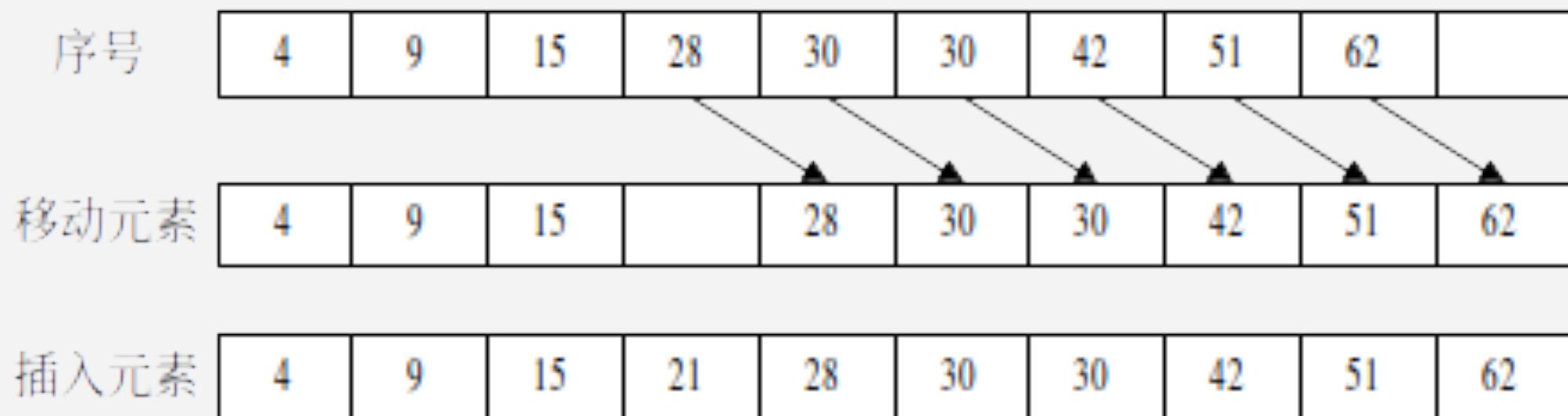
- 2种重要的操作，插入操作和删除操作相对比较复杂。
 - 在第*i*个元素之前插入一个元素时，需要将第*i*至第*n*（共*n-i+1*）个元素向后移动一个位置。
 - 删除第*i*个元素时，需要将第*i+1*至第*n*（共*n-i*）个元素向前移动一个位置。

线性表的顺序存储结构

- 顺序表的插入运算是指在表的第 i ($1 \leq i \leq n+1$) 个位置，插入一个新元素 e ，使长度为 n 的线性表 $(e_1, \dots, e_{i-1}, e_i, \dots, e_n)$ 变成长度为 $n+1$ 的线性表 $(e_1, \dots, e_{i-1}, e, e_i, \dots, e_n)$ 。
- 用顺序表作为线性表的存储结构时，由于结点的物理顺序必须和结点的逻辑顺序保持一致，因此必须将原表中位置 $n, n-1, \dots, i$ 上的结点，依次后移到位置 $n+1, n, \dots, i+1$ 上，空出第 i 个位置，然后在该位置上插入新结点 e 。
- 当 $i=n+1$ 时，是指在线性表的末尾插入结点，所以无需移动结点，直接将 e 插入表的末尾即可。

线性表的顺序存储结构

- 例如：已知线性表（4，9，15，28，30，30，42，51，62），需在第4个元素之前插入一个元素“21”，则需要将第9个位置到第4个位置的元素依次后移一个位置，然后将“21”插入到第4个位置，如图所示。



线性表的顺序存储结构

- 顺序表的删除运算是指将表的第*i* ($1 \leq i \leq n$) 个元素删去，使长度为*n*的线性表($e_1, \dots, e_{i-1}, e_i, e_{i+1}, \dots, e_n$)变成长度为*n-1*的线性表($e_1, \dots, e_{i-1}, e_{i+1}, \dots, e_n$)。
- 例如：顺序表(4, 9, 15, 21, 28, 30, 30, 42, 51, 62)删除第5个元素，则需将第6个元素到第10个元素依次向前移动一个位置。

序号	1	2	3	4	5	6	7	8	9	10
	4	9	15	21	28	30	30	42	51	62
删除 28 后	4	9	15	21	30	30	42	51	62	

线性表的顺序存储结构

```
template <class ElemType>
StatusCode SqList<ElemtType>::Insert(int position,
    const ElemtType &e)
//操作结果：在线性表的第position个位置前插入元素e,
//position的取值范围为1 ≤ position ≤ Length() + 1
//如线性表已满，则返回OVER_FLOW,
//如position合法，则返回SUCCESS, 否则返回ERROR
```

线性表的顺序存储结构

```
{  
    if (count == maxSize || position < 1 || position > Length() + 1)  
    { // 线性表已满，或position范围错  
        return ERROR;  
    }  
    else  
    { // 成功  
        ELEMType tmp;  
        for (int curPosition = len; curPosition >= position; curPosition--)  
        { // 插入位置之后的元素右移  
            GetElem(curPosition, tmp);  
            SetElem(curPosition + 1, tmp);  
        }  
        count++; // 插入后元素个数将自增1  
        SetElem(position, e); // 将e赋值到position位置处  
        return SUCCESS; // 插入成功  
    }  
}
```

线性表的顺序存储结构

```
template <class ElemtType>
StatusCode SqList<ElemtType>::Delete(int position, ElemtType &e)
// 操作结果：删除线性表的第position个位置的元素，并用e返回其值,
// position的取值范围为1≤position≤Length(),
// position合法时返回SUCCESS, 否则返回ERROR
{
    ...
    else
    {
        // position合法
        ElemtType tmp;
        GetElem(position, e);           // 用e返回被删除元素的值
        for (int curPosition = position + 1; curPosition <= Length(); curPosition++)
        {
            // 被删除元素之后的元素依次左移
            GetElem(curPosition, tmp); SetElem(curPosition - 1, tmp);
        }
        count--; // 删除后元素个数将自减1
        return SUCCESS;
    }
}
```

线性表的顺序存储结构

- 设计顺序表存储的两个集合求差值。(书上例子)
- 已知顺序表的元素类型为 int, 设计算法将其调整为左右两部分, 左边所有元素为奇数, 右边所有元素为偶数, 并要求算法的时间复杂度为 $O(n)$ 。 (书上例子)

线性表的顺序存储结构

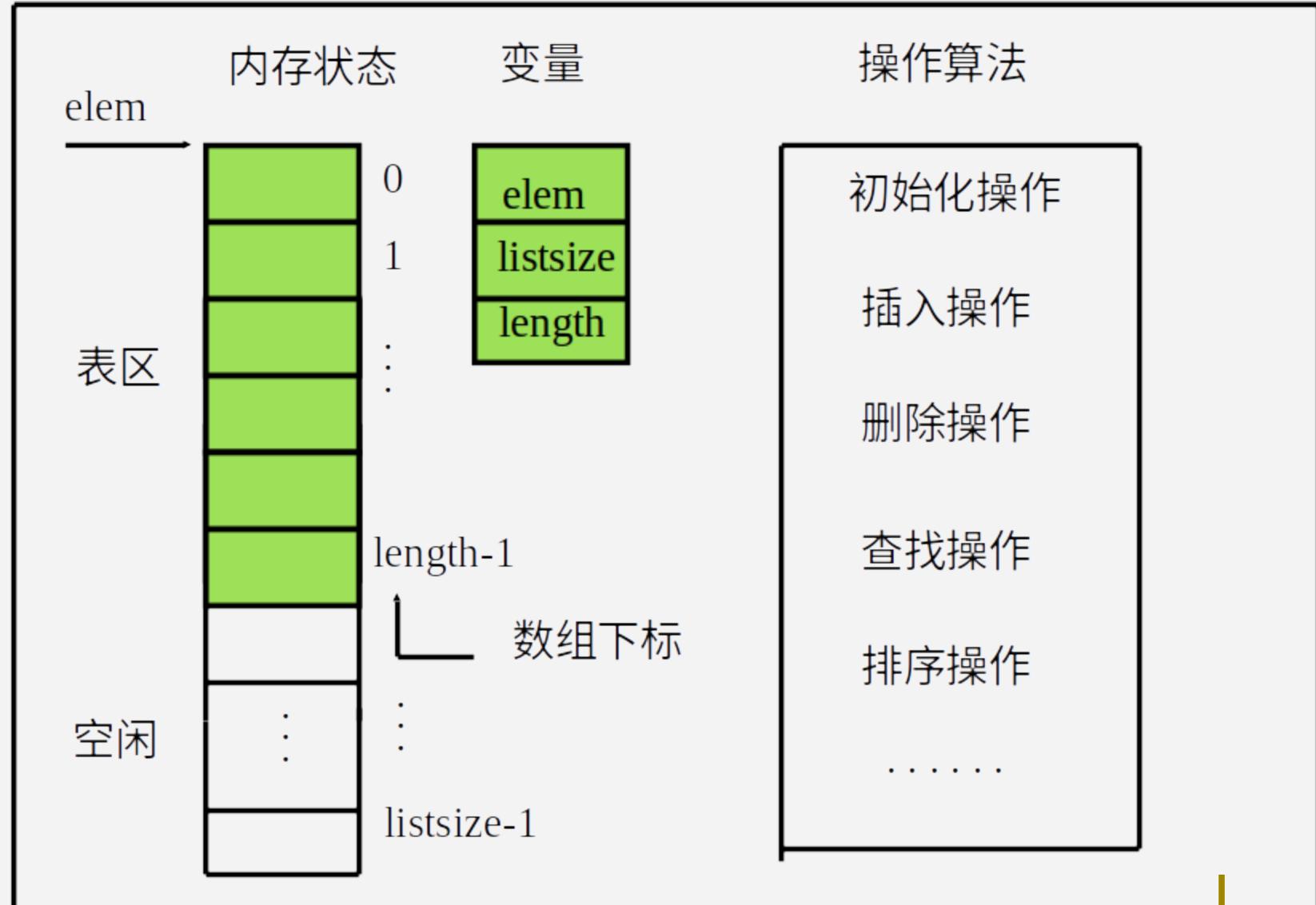
- 在顺序表中某个位置上插入或者删除一个元素时，时间的耗费主要是在元素的移动上。而移动元素的个数取决于插入或者删除的位置。
- $E_{is} = n/2$ $E_{dl} = (n-1)/2$
- 在顺序存储结构的线性表中插入或删除一个数据元素，平均约移动表中一半元素，按照前面的方法，可以知道其时间复杂度为 $O(n)$ 。

线性表的顺序存储结构

- 优点
 - 顺序表的结构简单
 - 顺序表的存储效率高，是紧凑结构
 - 顺序表是一个随机存储结构（直接存取结构）
- 缺点
 - 在顺序表中进行插入和删除操作时，需要移动数据元素，算法效率较低。
 - 对长度变化较大的线性表，或者要预先分配较大空间或者要经常扩充线性表，给操作带来不方便。
 - 原因：数组的静态特性造成

线性表的顺序存储结构

顺序表的整体概念：



线性表的链式存储结构

- 线性表的顺序存储结构的特点是逻辑关系上相邻的两个元素在物理位置上也相邻，可以随机存取任一元素；同时这个也是其弱点：插入和删除操作需要移动大量元素。
- 另外一种表示方法：链式存储结构。不要求逻辑关系上相邻的两个元素在物理位置上也相邻；插入和删除操作不需要移动元素。

线性表的链式存储结构

- 链式存储结构：用一组任意的存储单元存储线性表的数据元素（存储单元可以是连续的，也可以是不连续的）；
- 除了存储数据信息之外，还需要存储一个指示其直接后续的信息（直接后续的存储位置）；

线性表的链式存储结构

- 这两部分组成数据元素的存储映像，称为结点；
- N个结点链结成一个链表，称做链式存储结构。



线性表的链式存储结构

- 两个域：存储数据元素信息的域称为数据域；存储直接后续存储位置的域称为指针域。
- 指针域中存储的信息称为指针或链。
- 链表的存取，必须从头指针开始。头指针指示链表中第一个结点的存储位置。

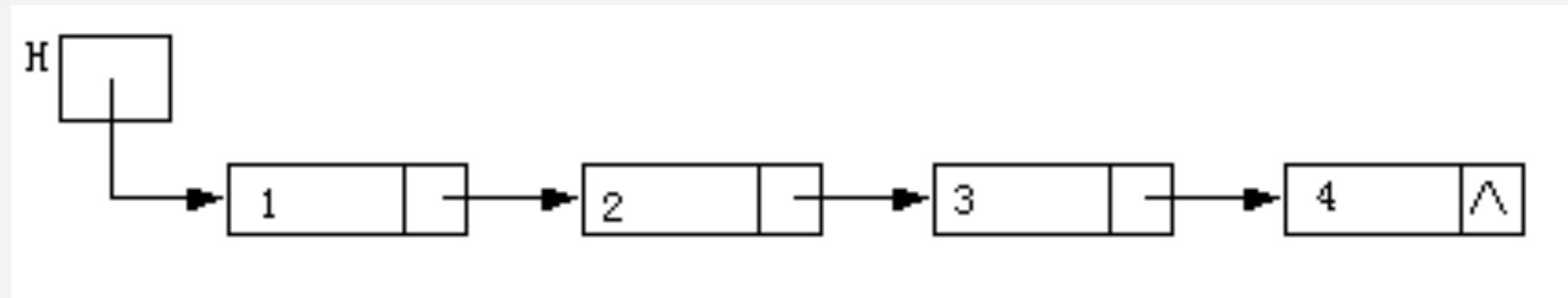
线性表的链式存储结构

线性表的链式存储结构有如下特点

- 数据元素之间的逻辑关系由节点中的指针域表示
- 每个元素的存储位置由其直接前驱的指针域指示
- 线性表的链式存储结构是非随机存取的存储结构
- 链式结构中的尾节点的直接后续为空（即此节点的指针域为空）（对于一般的单链表）

线性表的链式存储结构-单链表

- 用一组地址任意的存储单元存放线性表中的数据元素，以“结点的序列”表示的线性表称作单链表。
- $\text{data} + \text{next} = \text{结点}$ （表示数据元素）
- 每个数据单元中仅含一个指针域。

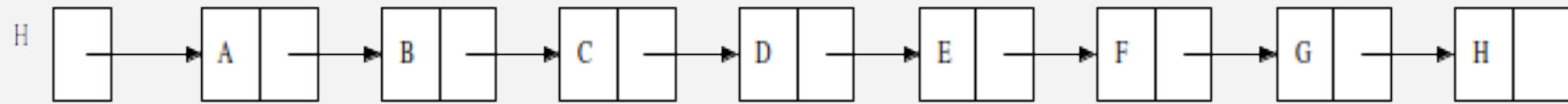


线性表的链式存储结构-单链表

例：图所示为线性表（A, B, C, D, E, F, G, H）的单链表存储结构，整个链表的存取需从头指针开始进行，依次顺着每个结点的指针域找到线性表的各个元素。

	存储地址	数据域	指针域
	1	D	43
	7	B	13
头指针 H	13	C	1
31	19	H	NULL
	25	F	37
	31	A	7
	37	G	19
	43	E	25

线性表的链式存储结构-单链表



单链表的逻辑状态



(a) 带头结点的空单链表



(b) 带头结点的单链表

带头结点单链表图示

线性表的链式存储结构-单链表

- 单链表是非随机存取的存储结构
- 如果 $p \rightarrow \text{data}$ 的值为 a_i ， 则 $p \rightarrow \text{next} \rightarrow \text{data}$ 的值为 a_{i+1} 。
- 在单链表中， 取得某个元素必须从头指针开始寻找。
- 需要掌握的单链表重要的运算是插入和删除操作。

线性表的链式存储结构-单链表类模板

```
template <class ElemType>
struct Node
{
    // 数据成员:
    ElemType data;           // 数据域
    Node<ELEMType> *next;   // 指针域

    // 构造函数模板:
    Node();                  // 无参数的构造函数模板
};
```

线性表的链式存储结构-单链表类模板

```
// 简单线性链表类模板
template <class ElemType>
class SimpleLinkList
{
protected:
    // 链表实现的数据成员:
    Node<ElemType> *head; // 头结点指针

    // 辅助函数模板:
    Node<ElemType> *GetElemPtr(int position);
    // 返回指向第position个结点的指针
    void Init();           // 初始化线性表
```

线性表的链式存储结构-单链表类模板

```
public:  
    // 抽象数据类型方法声明及重载编译系统默认方法声明:  
    SimpleLinkList();           // 无参数构造函数模板  
    virtual ~SimpleLinkList();  // 析构函数模板  
    int Length();               // 求线性表长度  
    bool Empty();                // 判断线性表是否为空  
    void Clear();                // 将线性表清空  
    void Traverse(void (*visit)(const ElemtType &)); // 遍历线性表  
    StatusCode GetElem(int position, ElemtType &e); // 求指定位置的元素  
    StatusCode SetElem(int position, const ElemtType &e);  
    // 设置指定位置的元素值
```

线性表的链式存储结构-单链表类模板

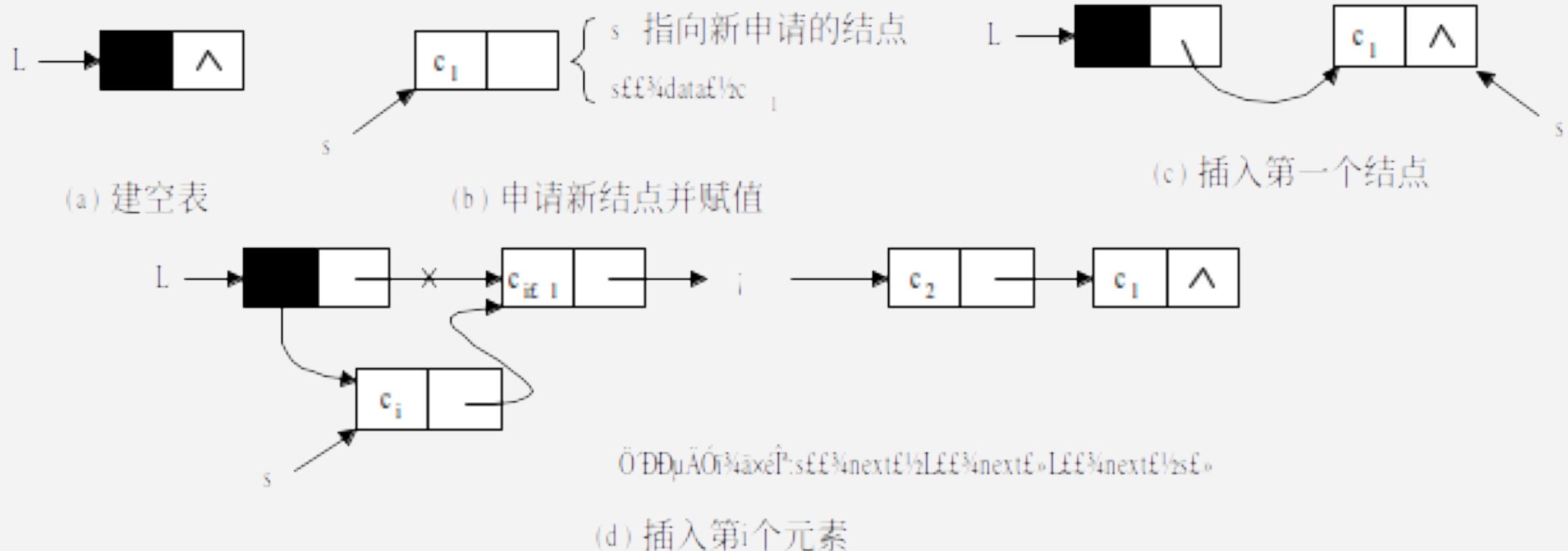
```
StatusCode Delete(int position, ElemType &e);           // 删除元素
StatusCode Insert(int position, const ElemType &e);        // 插入元素
SimpleLinkList(const SimpleLinkList<ElemType> &copy);
// 复制构造函数模板
SimpleLinkList<ElemType> &operator =
  (const SimpleLinkList<ElemType> &copy);
// 重载赋值运算符
};
```

线性表的链式存储结构-单链表

动态地建立单链表的常用方法有如下两种

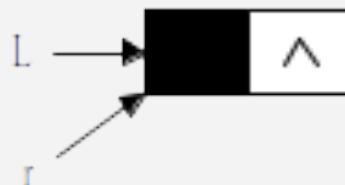
- 头插法建表：该方法从一个空表开始，重复读入数据，生成新结点，将读入数据存放到新结点的数据域中，然后将新结点插入到当前链表的表头上，直到读入结束标志为止。
- 尾插法建表：头插法建立链表虽然算法简单，但生成的链表中结点的次序和输入的顺序相反。若希望二者次序一致，可采用尾插法建表。该方法是将新结点插入到当前链表的表尾上，为此必须增加一个尾指针r，使其始终指向当前链表的尾结点。

线性表的链式存储结构-单链表

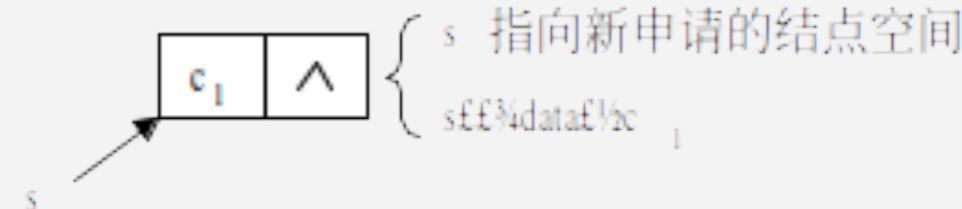


头插法建立单链表图示

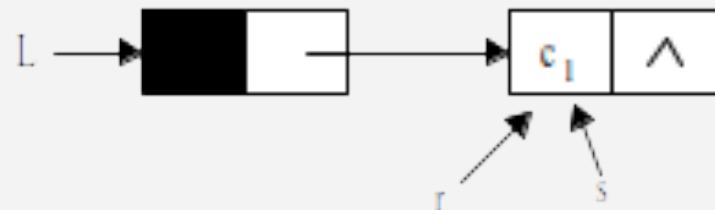
线性表的链式存储结构-单链表



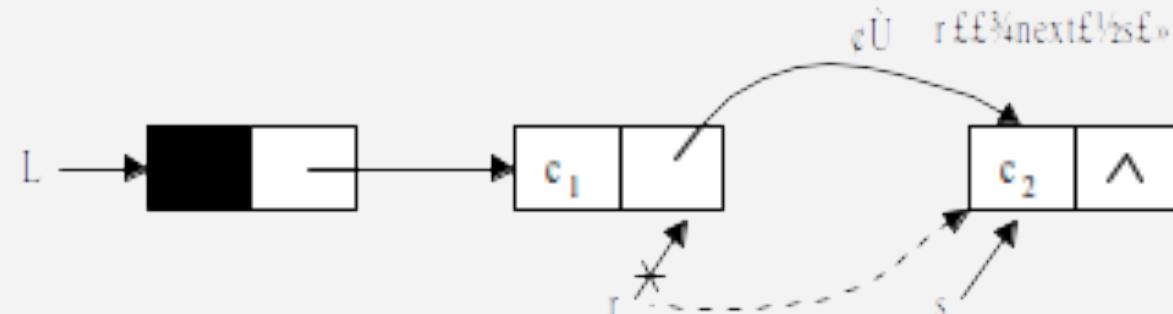
(a) 建空表



(b) 申请新结点并赋值



(c) 插入第一个结点



(d) 插入第二个结点

尾插法建表图示

线性表的链式存储结构-单链表

查找运算

按序号查找：在链表中，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，而只能从链表的头指针出发，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，链表不是随机存取结构。设单链表的长度为*n*，要查找表中第*i*个结点，仅当 $1 \leq i \leq n$ 时，*i*的值是合法的。

线性表的链式存储结构-单链表

查找运算

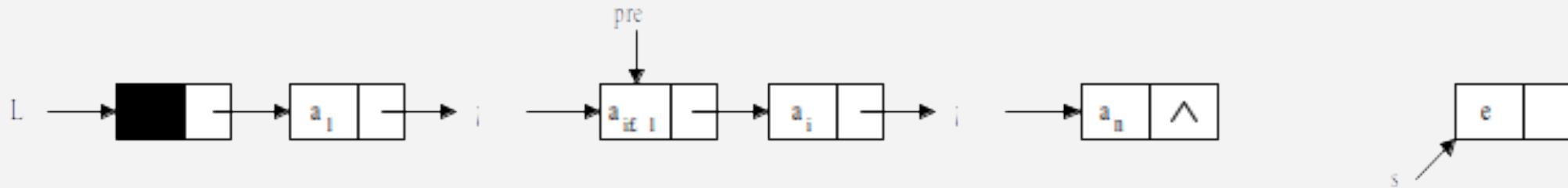
按值查找：按值查找是在链表中，查找是否有结点值等于给定值key的结点，若有的话，则返回首次找到的其值为key的结点的存储位置；否则返回NULL。查找过程从开始结点出发，顺着链表逐个将结点的值和给定值key作比较。

线性表的链式存储结构-单链表

插入运算

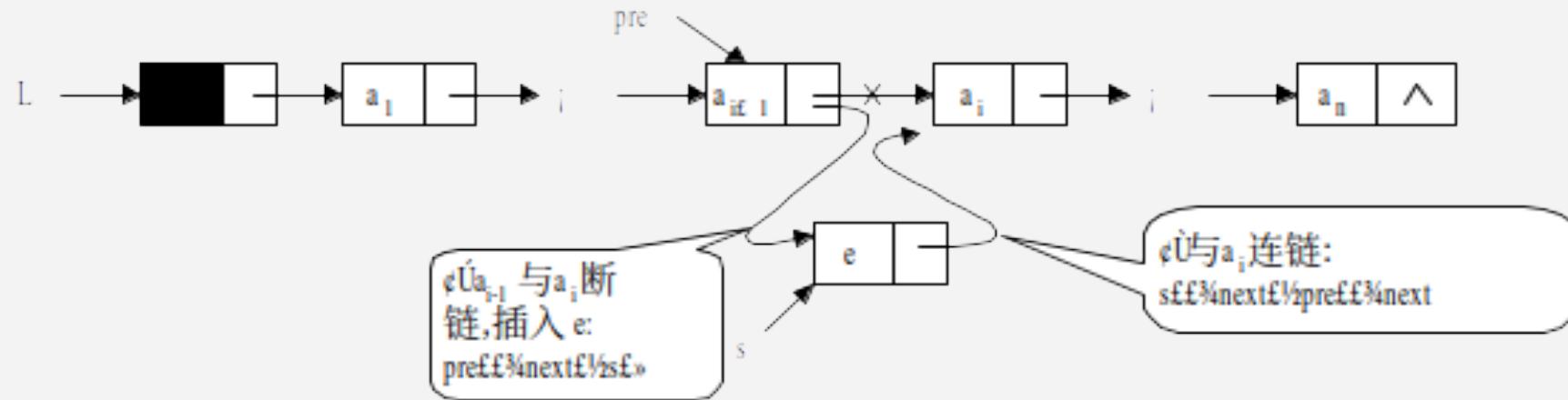
- 将值为x的新结点插入到表的第*i*个结点的位置上，即插入到 a_{i-1} 与 a_i 之间。
- 必须首先找到 a_{i-1} 的存储位置pos，然后生成一个数据域为x的新结点*p，并令结点*p的指针域指向新结点，新结点的指针域指向结点 a_i 。从而实现三个结点 a_{i-1} ，x和 a_i 之间的逻辑关系的变化。

线性表的链式存储结构-单链表



(a) 寻找第*i*个结点

(b) 申请新的结点



(c) 插入

在单链表第*i*个结点前插入一个结点的过程

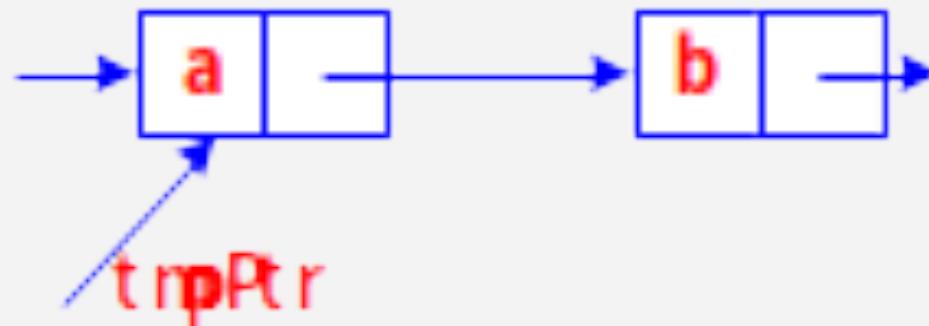
线性表的链式存储结构-单链表

```
 Status ListInsert_L(LNode* pHead, int i, ElemType e)
{
    LNode* p=pHead;
    int j=0;
    while(p && j<i)
    {
        p=p->next;  j++;
    }
    if(!p) return ERROR;
    LNode* s=(LNode*)malloc(sizeof(LNode));
    s->data=e;  s->next=p->next;
    p->next=s;
    return OK;
}
```

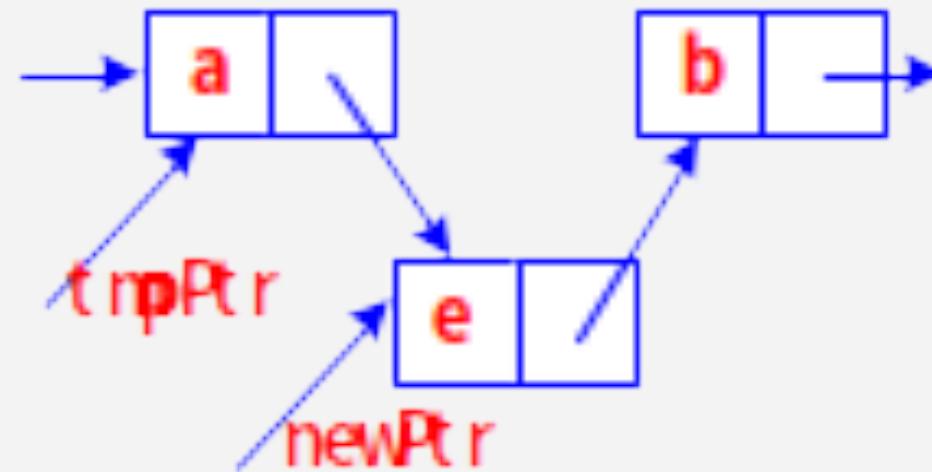
线性表的链式存储结构-单链表

(C++)

```
newPtr = new Node<ElemType>(e, tmpPtr->next);  
tmpPtr->next = newPtr;
```



(a) 插入前



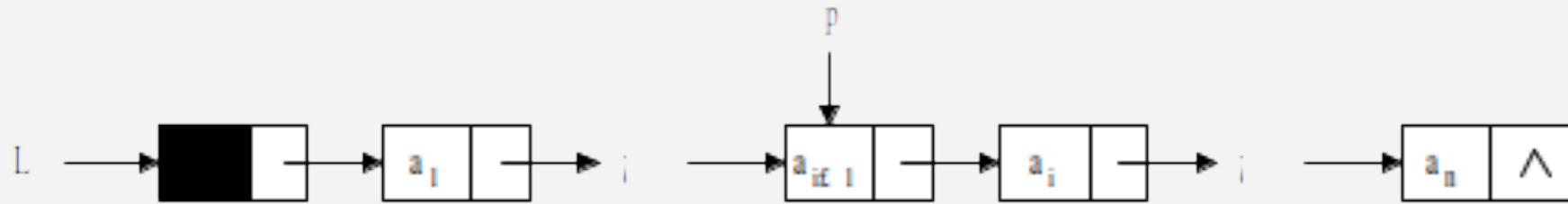
(b) 插入后

线性表的链式存储结构-单链表

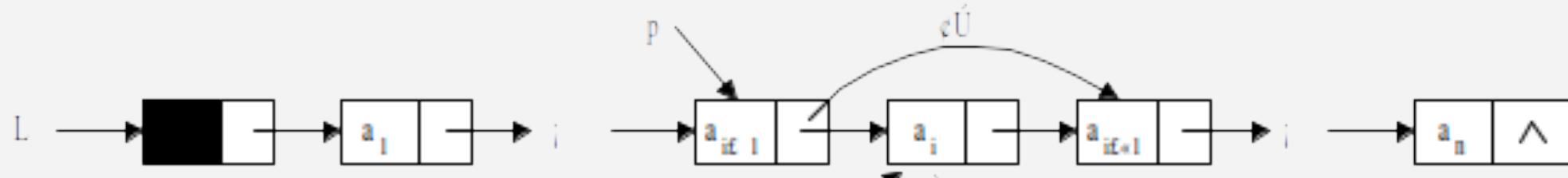
删除运算

- 将表的第*i*个结点删去。
- 因为在单链表中结点 a_i 的存储地址是在其直接前趋结点 a_{i-1} 的指针域next中，所以我们必须首先找到 a_{i-1} 的存储位置p。然后令 $p \rightarrow next$ 指向 a_i 的直接后继结点，即把 a_i 从链上摘下。最后释放结点 a_i 的空间，将其归还给“存储池”。

线性表的链式存储结构-单链表



(a) 寻找第*i*个结点
即指向它



(b) 删除并释放第*i*结点

单链表的删除过程

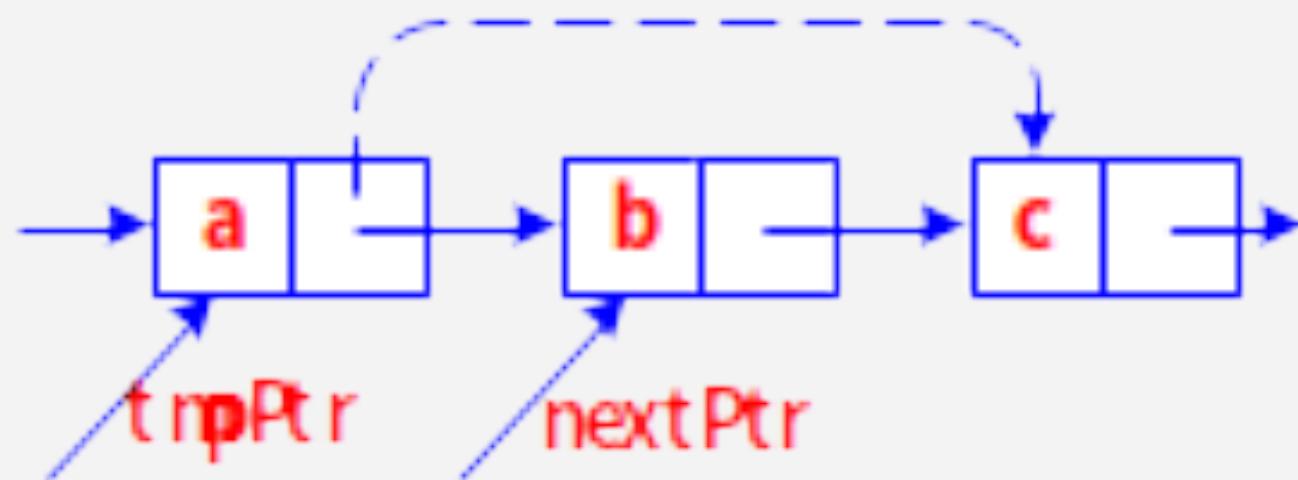
线性表的链式存储结构-单链表

```
 Status ListDelete_L(LNode* pHead, int i, ElemType &e)
{
    LNode* p=pHead;
    int j=0;
    while(p->next && j<i)
    {
        p=p->next;  j++;
    }
    if(!p->next) return ERROR;
    LNode* q=p->next;      p->next=q->next;
    e=q->data;          free(q);
    return OK;
}
```

线性表的链式存储结构-单链表

(C++)

```
nextPtr = tmpPtr->next;           // nextPtr为tmpPtr的后继  
tmpPtr->next = nextPtr->next;     // 删除结点  
delete nextPtr;                  // 释放被删结点
```



线性表的链式存储结构-单链表

理解头指针、头结点、首元结点的定义及差别

- 头结点是在插入、删除等操作时，为了算法统一而设立的（若无头结点，则在第一元素前插入结点或删除结点时，链表的头指针总在变化）。对链表的任何操作，均要从头结点开始。头指针往往被称作链表的名字，用于指向链表中的第一个结点，如果链表设有头结点，头指针则指向头结点；如果链表不设头结点，头指针则指向首元结点。
- 首元结点，是指链表中存储第一个数据元素的结点，它是头结点之后的第一个结点。
- 简而言之，头指针是用于指向链表起始的指针，头结点是一个为了方便操作而设置的特殊结点，首元结点是实际存储数据的结点。

线性表的链式存储结构-循环链表

- 循环链表是单链表的变形。
- 循环链表最后一个结点的next指针不为NULL，而是指向了表的前端。
- 为简化操作，在循环链表中往往加入表头结点。
- 循环链表的特点是：只要知道表中某一结点的地址，就可搜寻到所有其他结点的地址。
- 从表中任一结点出发均可找到其他结点。
- 其循环条件是移动的指针是否等于头指针。

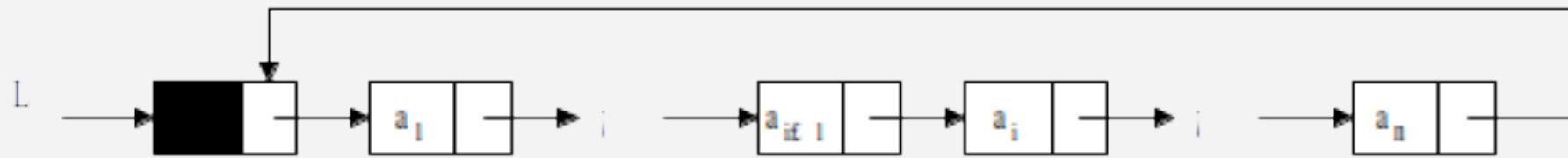
线性表的链式存储结构-循环链表

- 在单链表中，将终端结点的指针域NULL改为指向表头结点的或开始结点，就得到了单链形式的循环链表，并简单称为单循环链表。
- 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。

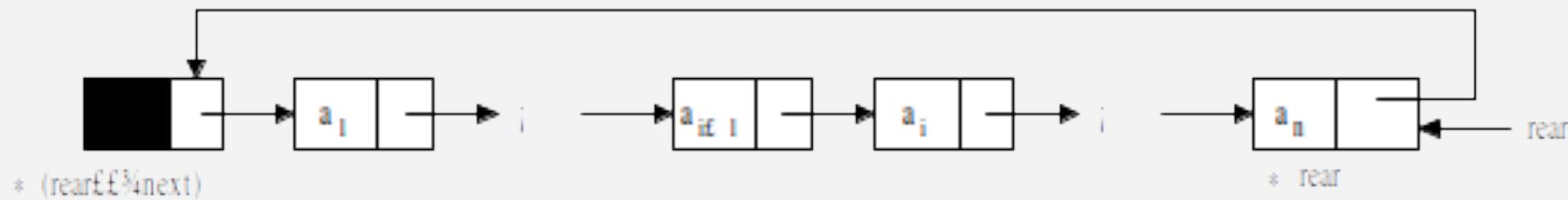
线性表的链式存储结构-循环链表



(a) 带头结点的空循环链表



(b) 带头结点的循环单链表的一般形式



(c) 采用尾指针的循环单链表的一般形式

在用头指针表示的单链表中，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 $O(n)$ 。

线性表的链式存储结构-循环链表

- 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。
- 如果改用尾指针rear来表示单循环链表，则查找开始结点 a_1 和终端结点 a_n 都很方便，它们的存储位置分别是

$(\text{rear} \rightarrow \text{next}) \rightarrow \text{next}$ 和 rear

显然，查找时间都是 $O(1)$ 。因此，实际中多采用尾指针表示单循环链表。

线性表的链式存储结构-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。

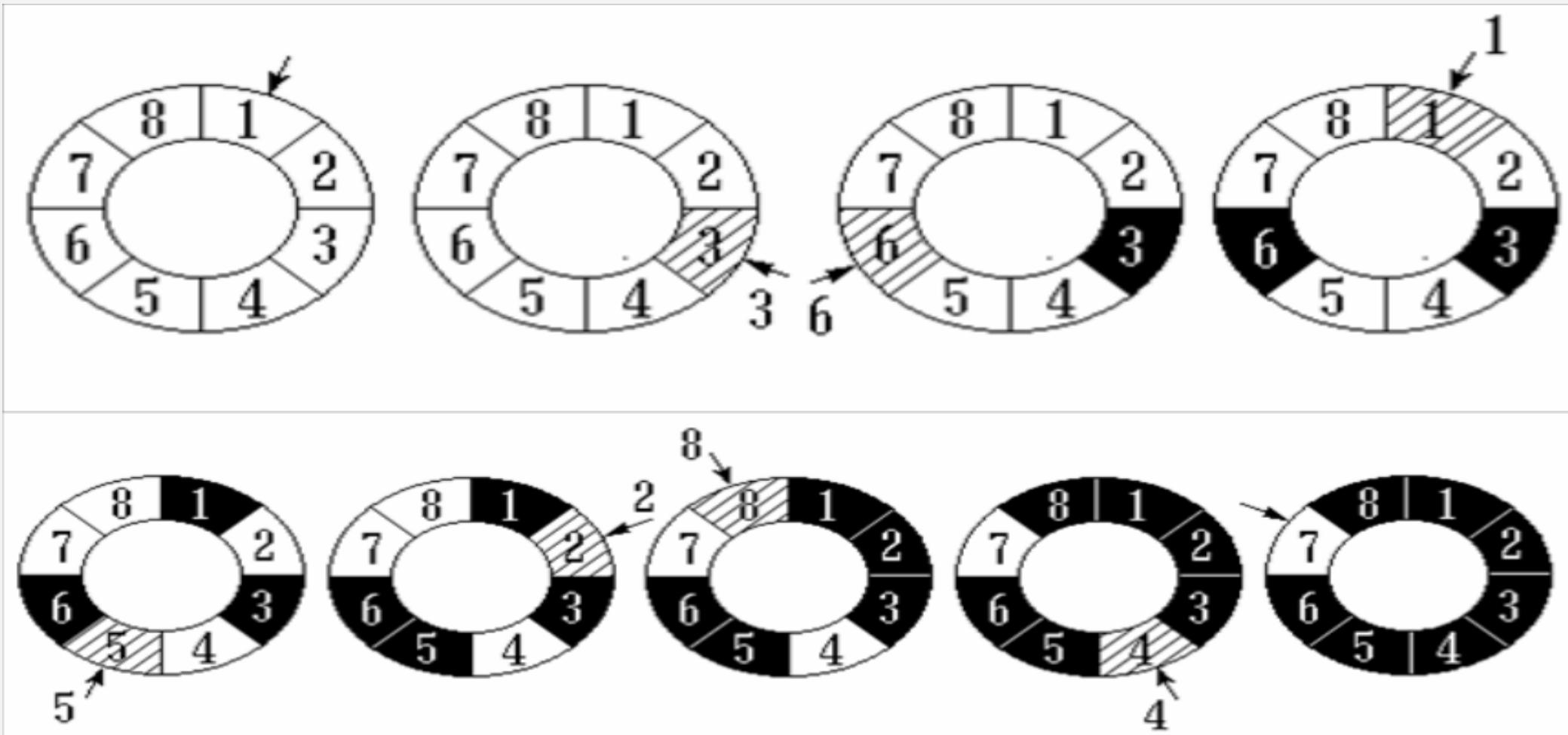
线性表的链式存储结构-循环链表

约瑟夫问题

n 个人围成一个圆圈，首先第1个人从1开始，一个人一个人顺时针报数，报到第 m 个人，令其出列。然后再从下一个人开始，从1顺时针报数，报到第 m 个人，再令其出列，…，如此下去，直到圆圈中只剩一个人为止。此人即为优胜者。

例如 $n = 8$ $m = 3$

线性表的链式存储结构-循环链表



线性表的链式存储结构-循环链表

```
void Josephus(int n, int m)
{
    SimpleCircLinkList<int> la;           // 定义空循环链表
    int pos = 0;                          // 报数到的人在链表中序号
    int out, winer;
    for (int k = 1; k <= n; k++) la.Insert(k, k);
    cout << "出列者:";

    for (int i = 1; i < n; i++) {          // 循环n-1次，让n-1个出列
        for (int j = 1; j <= m; j++) {
            pos++;
            if (pos > la.Length())
                pos = 1;
        }
        la.Delete(pos--, out);             // 报数到m的人出列
        cout << out << " ";
    }
    la.GetElem(1, winer);                 // 剩下的一个人为优胜者
    cout << endl << "优胜者:" << winer << endl;
}
```

线性表的链式存储结构-双向链表

- 循环单链表能够实现从任一结点出发沿着链能找到其前驱结点，但时间耗费是 $O(n)$ 。
- 如果希望从表中快速确定某一个结点的前驱，另一个解决方法就是在单链表的每个结点里再增加一个指向其前驱的指针域prior(back)。这样形成的链表中就有两条方向不同的链，我们可称之为双向链表。

线性表的链式存储结构-双向链表

- 双向链表是指在前驱和后继方向都能游历的线性链表。
- 结点中有两个指针域，其一指向直接后继，另一指向直接前驱。
- 双向链表的循环链表，存在两个环。
- 双向链表通常采用带表头结点的循环链表形式。

线性表的链式存储结构-双向链表

- 双向链表是指在前驱和后继方向都能游历的线性链表。
- 结点中有两个指针域，其一指向直接后续，另一指向直接前驱。
- 双向链表的循环链表，存在两个环。
- 双向链表通常采用带表头结点的循环链表形式。

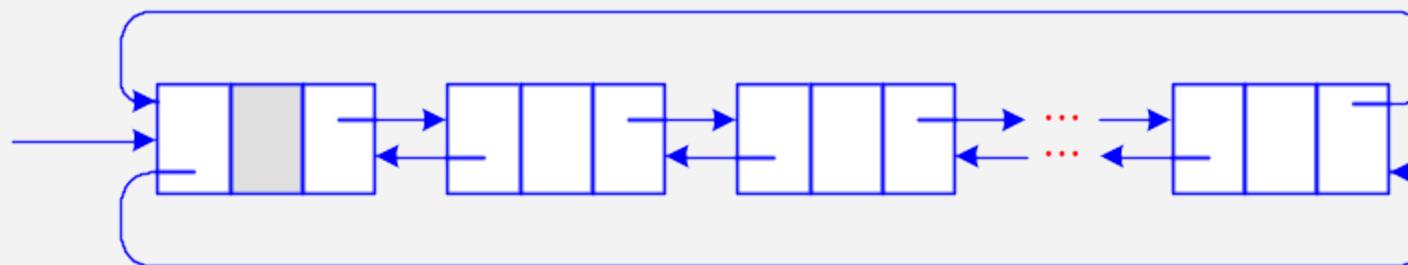
线性表的链式存储结构-双向链表



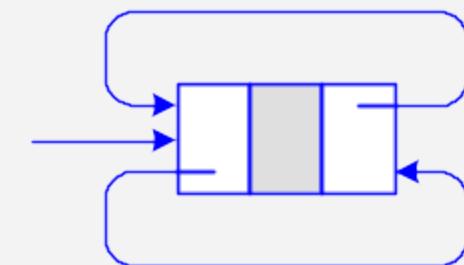
线性表的链式存储结构-双向链表

- 由于在双向链表中既有前向链又有后向链，寻找任一个结点的直接前驱结点与直接后继结点变得非常方便。设指针p指向双链表中某一结点，则有下式成立：

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



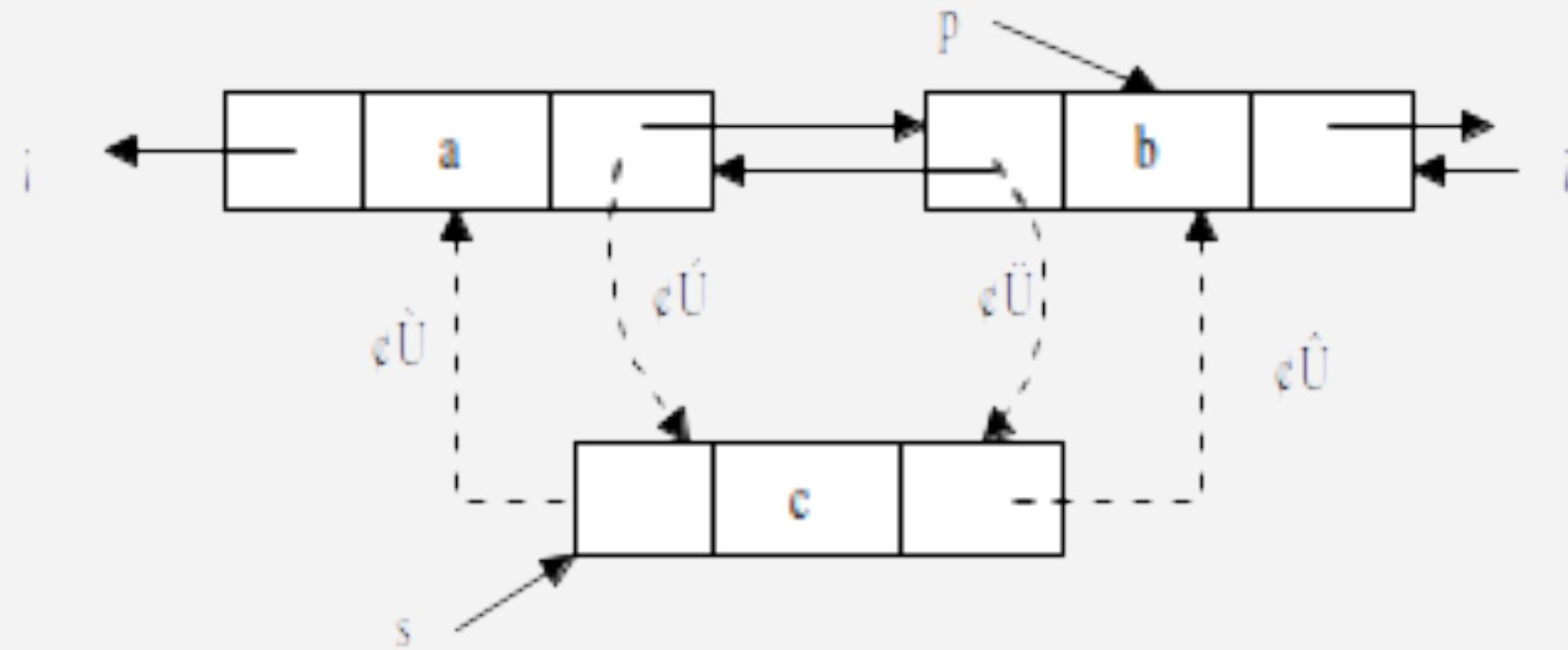
(a) 非空双向链表



(b) 空双向链表

线性表的链式存储结构-双向链表

插入操作

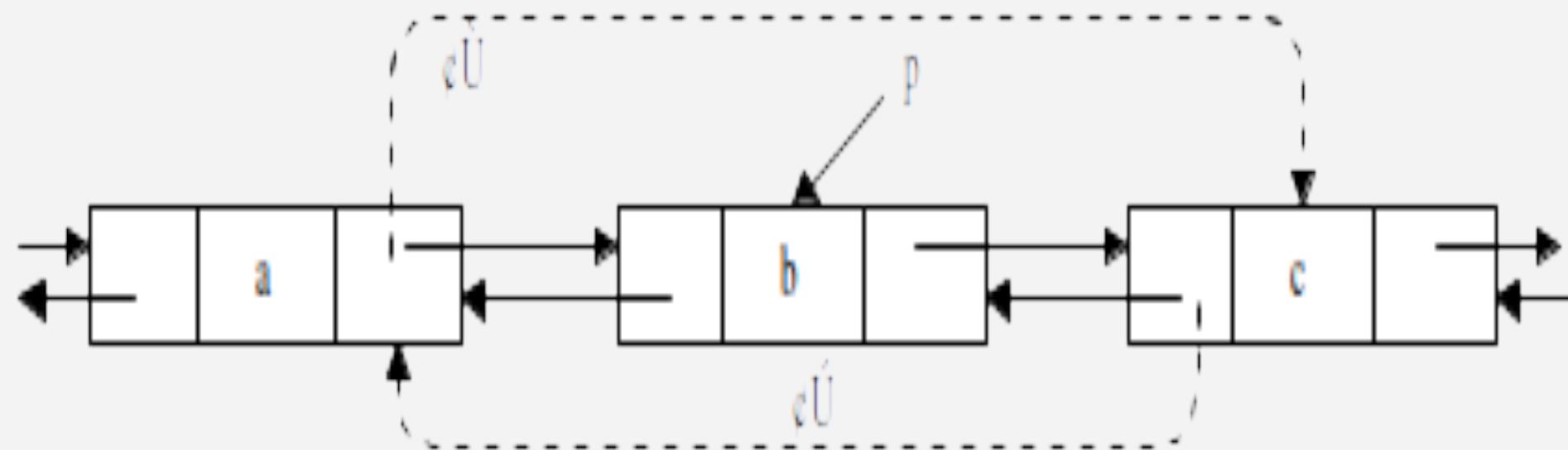


线性表的链式存储结构-双向链表

```
Status ListInsert_DuL(DuLinkList &L, int i, ElemType e)
{
    if(! (p=GetElemP_DuL(L, i))) return ERROR;
    if(! (s=(DuLinkList)malloc(sizeof(DuListNode))))
        return ERROR;
    s->data=e;
    s->prior=p->prior;  p->prior->next=s;
    s->next=p; p->prior=s;
    return OK;
}
```

线性表的链式存储结构-双向链表

删除操作



线性表的链式存储结构-双向链表

```
Status ListDelete_DuL(DuLinkList &L, int i, ElemType &e)
{
    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;
}
```

顺序表和链表的比较

基于空间的考虑

- 在链表中的每个结点，除了数据域外，还要额外设置指针域，从存储密度来讲，这是不经济的。所谓存储密度(Storage Density)，是指结点数据本身所占的存储量和整个结点结构所占的存储量之比。
- 存储密度=结点数据本身所占的存储量/结点结构所占的存储总量。
- 顺序表的存储密度为1，而链表的存储密度小于1。
- 例如单链表的结点的数据均为整数，指针所占空间和整型量相同，则单链表的存储密度为50%。因此若不考虑顺序表中的备用结点空间，则顺序表的存储空间利用率为100%，而单链表的存储空间利用率为50%。
- 由此可知，当线性表的长度变化不大，易于事先确定其大小时，为了节约存储空间，宜采用顺序表作为存储结构。

顺序表和链表的比较

基于时间的考虑

- 顺序表是由向量实现的，它是一种随机存取结构，对表中任一结点都可以在 $O(1)$ 时间内直接地存取，而链表中的结点，需从头指针起顺着链找才能取得。因此，若线性表的操作主要是进行查找，很少做插入和删除时，宜采用顺序表做存储结构。
- 在链表中的任何位置上进行插入和删除，都只需要修改指针。而在顺序表中进行插入和删除，平均要移动表中近一半的结点，尤其是当每个结点的信息量较大时，移动结点的时间开销就相当可观。
- 因此，对于频繁进行插入和删除的线性表，宜采用链表做存储结构。若表的插入和删除主要发生在表的首尾两端，则宜采用尾指针表示的单循环链表。

在链表结构中保存当前位置和元素个数

- 前面讲解了线性表的链式存储结构的实现处理简单。
- 但许多应用程序可能要几次引用同一个数据元素，对于这类应用程序，前面的链表实现效率低下。
- 最好在链表结构中保存当前位置和元素个数。

```
mutable int currentPosition;  
mutable Node<ElemType>* currentPtr;  
int nodeCount;
```

实例研究

[例] 一元多项式及其运算

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

主要运算: 多项式相加、相减、相乘等

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

实例研究-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： $f(x) = 3x^4 + 5x^2 + 1$

表示成：

i	0	1	2	3	4	...
a[i]	1	0	5	0	3	...

两个多项式相加： 两个数组对应分量相加

问题：如何表示多项式 $x+3x^{2000}$?

实例研究-顺序存储结构表示非零项

每个非零项*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，只存储非零项。

如： $f(x) = 3x^{18} - 12x^9 + 4x^2$

下标 <i>i</i>	0	1	2
系数 a_i	3	-12	4
指数 i	18	9	2

$$g(x) = 5x^{13} + 7x^9 - 3x + 1$$

下标 <i>i</i>	0	1	2	3
系数 a_i	5	7	-3	1
指数 i	13	9	1	0

实例研究-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

$$f: (3, 18) \ (-12, 9) \ (4, 2)$$

$$g: (5, 13) \ (7, 9) \ (-3, 1) \ (1, 0)$$

$$f+g: (3, 18) \ (5, 13) \ (-5, 9) \ (4, 2) \ (-3, 1) \ (1, 0)$$

实例研究-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

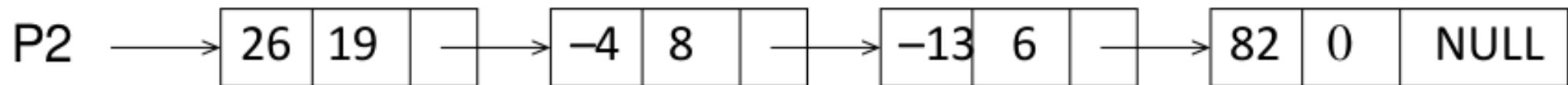
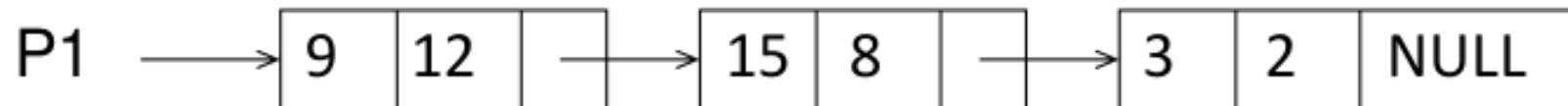
coef	expon	next
------	-------	------

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

实例研究-链表结构存储非零项

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

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



实例研究-链表结构存储非零项

```
status PolyAdd(Node* paHead, Node* pbHead, Node* pcHead)
{
    // c = a + b
    Node* pa = paHead->next, pb = pbHead->next, pc = pcHead;
    while(pa != NULL || pb != NULL) {
        if(pa != NULL && pb != NULL) {
            if(pa->exp > pb->exp) insert_R(pc, pa->conf, pa->exp), pa = pa->next;
            else if(pa->exp < pb->exp) insert_R(pc, pb->conf, pb->exp), pb = pb->next;
            else if(pa->exp == pb->exp) {
                if(pa->conf + pb->conf != 0) insert_R(pc, pa->conf+pb->conf, pa->exp);
                pa = pa->next, pb = pb->next;
            }
        } else if(pa != NULL) insert_R(pc, pa->conf, pa->exp), pa = pa->next;
        else if(pb != NULL) insert_R(pc, pb->conf, pb->exp), pb = pb->next;
    }
}
```

练习

- 1、对于一个头指针为head的带头结点的单链表，判定该表为空表的条件是（ ）
- A. $head==NULL$ B. $head\rightarrow next==NULL$
C. $head\rightarrow next==head$ D. $head!=NULL$
- 2、在一个长度为n的顺序表中第i个元素 ($1 \leq i \leq n$) 之前插入一个元素时，需向后移动_____个元素。
- 3、顺序存储方式只能用于存储线性结构。（ ）
- 4、线性表采用链表存储时，结点和结点内部的存储空间可以是不连续的。（ ）
- (B, $n-i+1$, \times , \times)

练习

- 5、取线性表的第*i*个元素的时间同*i*的大小有关。 ()
- 6、对于一个具有n个结点的单链表，在已知的结点*p后插入一个新结点的时间复杂度为_____，在给定值为x的结点后插入一个新结点的时间复杂度为_____。
- 7、顺序存储结构是通过_____表示元素之间的关系的；链式存储结构是通过_____表示元素之间的关系的。
(×, (0(1)、0(n)) , (物理上相邻、指针))

练习

```
void fun1(Node *L)
{
    p = L; q = NULL;
    while(p != NULL)
    {
        r = p -> next; p -> next = q; q = p; p = r;
    }
    L = q;
}
```

算法实现的功能是将无头结点的单链表L逆转

练习

```
void fun2(LinkList L1, LinkList L2)
{
    Node *p, *q, *r;
    q = L2 -> next;
    while(q)
    {
        p = L1;
        while(p -> next)
        {
            if(p -> next -> data == q -> data)
            {
                r = p -> next; p -> next = r -> next; free(r);
            }
            p = p -> next;
        }
        q = q -> next;
    }
    return;
}
```

算法实现的是集合差集运算，依次在链表L1中查找L2的元素，有则从L1中删除。