数据结构与算法

主讲教师：刘峤

第3章 线性表
(linear list)

第2章内容简介

- 什么是线性表
  - 线性表是最基本的数据结构，用于表示线性结构
  - 线性结构中的数据元素之间存在一种**顺序关系**
- 本章主要内容
  - 线性表的基本概念及其**顺序**和**链式**存储结构
  - 线性表的几种典型应用
- 本节学习要点
  - 顺序存储结构：动态内存分配，元素地址计算方法
  - 链式存储结构：指针成员，链式关系表达与计算

线性表的基本操作

① listInit(L) // 初始化操作

② listLength(L) // 求线性表长度

③ listGet(L,i) // 取元素

④ listLocate(L,x) // 定位函数

⑤ listPrior(L,e) // 求前驱函数

⑥ listNext(L,e) // 求后继函数

⑦ listInsert(L,i.e) // 前插操作

⑧ listDelete(L,i) // 删除操作

⑨ listEmpty(L) // 判空表函数

⑩ listClear(L) // 清空表函数

2.1 顺序表和链表

顺序表和链表：线性表的定义

- 定义：线性表是n个同类型数据元素的有限序列，记为：

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

  - 其中：L 为表名；i 为数据元素 a_i 在线性表中的位序
  - n 为线性表的表长 → n=0 时称为空表

☙ 特点：数据元素之间的关系是

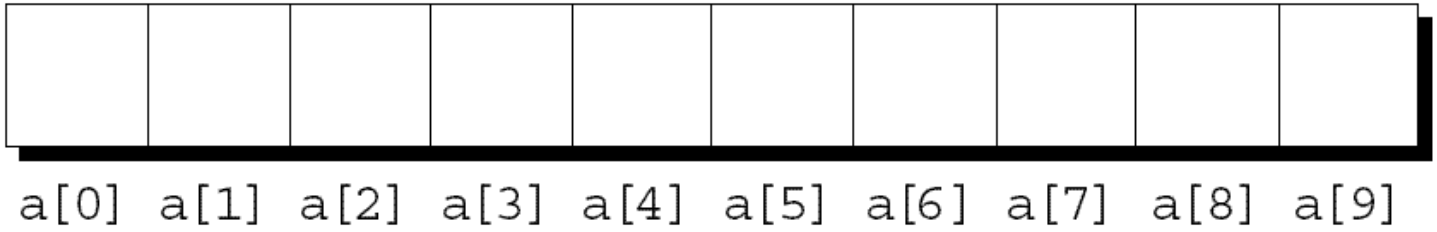
  - a_{i-1} 是 a_i 的直接前驱，a_{i+1} 是 a_i 的直接后继
  - 除第一元素 a_1 外，均有唯一的前驱
  - 除最后元素 a_n 外，均有唯一的后继
  - 思考：这是说的逻辑结构还是物理结构？ 逻辑结构!

☙ 线性表的其它特点

  - 表中元素 a_i 的数据类型相同，位序 i 从1开始

线性表的两类存储结构

1. 顺序存储结构（顺序表）

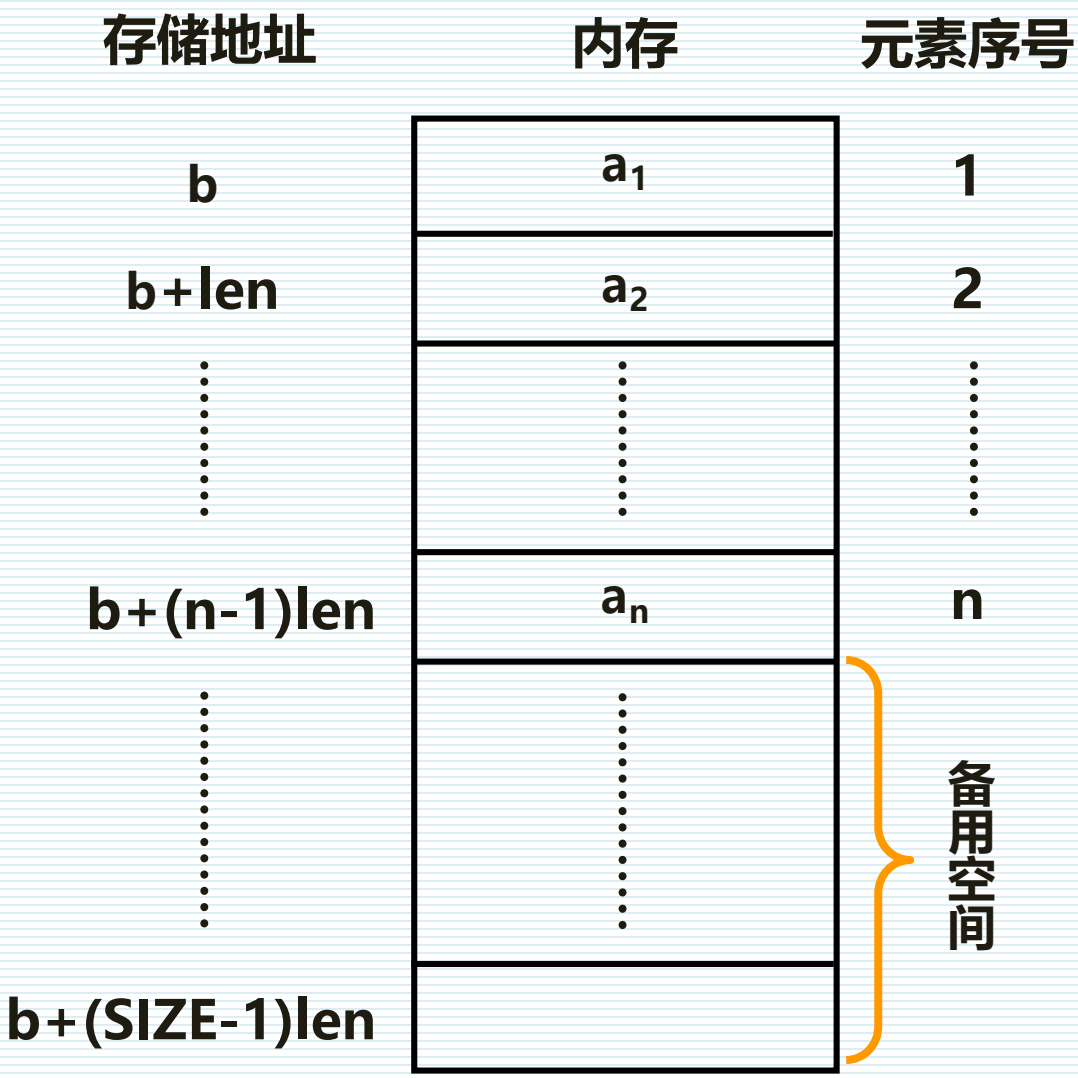


2. 链式存储结构（链表）

理解线性表

- 理解线性表的要点是
  - 表中元素具有逻辑上的顺序性
  - 表中元素个数有限，且每个元素均不可再分
  - 表中元素的数据类型都相同（占有相同数量的存储空间）
  - 一维数组是一种典型的线性表，然而二者并非等价关系
    - 数组是一种特殊的线性表
      - 这种线性表中的数据元素本身也可以是线性表
    - 线性表不仅可以采用数组实现，还可以采用链表实现
      - 逻辑上相邻的元素，物理上不一定相邻
      - 顺序存储结构 v.s. 链式存储结构

线性表的顺序存储结构



线性表的顺序存储结构

- **顺序表**（Sequential List）
  - 定义：用一组地址连续的存储单元存放一个线性表
  - 地址计算方法：$\mathbf{Loc(a_{i+1}) = Loc(a_i) + Len}$

    $\mathbf{Loc(a_i) = Loc(a_1) + (i-1)*Len}$
    - Len 表示一个元素占用的存储单元个数
    - $Loc(a_i)$ 表示线性表第 i 个元素的地址
    - $Loc(a_1)$ 表示起始地址（也称为基地址）
  - 特点：能够实现随机存取
    - 数据元素的关系：逻辑上相邻 = 物理地址相邻
  - 实现：可用一维数组实现

使用数组实现顺序表的静态存储

```
#define maxSize 100        // 最大允许长度
typedef int ElemType;      // 元素的数据类型

struct  List    // 声明顺序表类型
{
        ElemType data[maxSize];  // 存储数组
        int length;              // 表中当前元素个数
};

typedef struct List TList;
```

思考：顺序表的静态存储方式有何缺点？如何解决？

C语言知识回顾：动态内存分配

- 声明一个指针变量不会自动分配内存
  - 在对指针执行间接访问前必须对其初始化
    - 或者使它指向预先分配好的内存
    - 或者为指针所指向的目标**动态分配内存**
- 动态内存分配函数： **malloc()**
  - 函数原型：**void *malloc (unsigned int size)**;
    - 在操作系统管理的内存动态存储区开辟一块空间
    - 分配一段长度为**size个字节**的连续空间
    - 若分配成功则返回一个指向该区域起始地址的指针
    - 否则（例如内存空间不够）返回空指针（NULL）
    - 返回的**指针类型**需由程序员指定！

void的含义

- void关键字表达的意思是“无类型”
  - void只起限制函数和指针行为的作用
  - 不能声明或定义void类型的变量
    - 因为当对象类型不确定时，则它的大小也是未确定的
  - 例如：**void a; // illegal use of type 'void'**
- void * 表示“无类型指针”
  - void*型指针被用于如下两种情况
    - 对象的确切类型未知
    - 在特定环境下对象的类型会发生变化
  - 任何非const类型的指针都可以被赋值给void*型的指针

指针的强制类型转换

- 若指针p1和p2的类型相同，则可以直接相互赋值
- 若p1和p2指向不同的数据类型？
  - 则必须使用强制类型转换运算符
  - 把赋值运算符右边的指针类型转换为左边指针的类型
- 例如：**float *p1; int *p2; p1 = p2;**

  Error：'=' : cannot convert from 'int *' to 'float *'

  - 须改为：**p1 = (float *) p2;**

指针的强制类型转换

- 任何类型的指针都可以赋值给void * （无需类型转换）
  - 例如：**void *p1; int *p2; p1 = p2;**
- 反之则不正确：
  - 例如：**void *p1; int *p2; p2 = p1;**

Error: '=' : cannot convert from 'void *' to 'int *'

void关键字的使用

- 若函数参数可以是任意类型指针，则应声明为void *
  - 内存操作函数memcpy和memset的函数原型分别为：

```
void * memcpy (void *dest, const void *src, size_t len);

void * memset ( void * buffer, int c, size_t num );
```

  - 因此任何类型的指针都可以传入memcpy和memset中

void关键字的使用

- 若函数参数可以是任意类型指针，则应声明为void *
  - 因此任何类型的指针都可以传入memcpy和memset中
    - **int A[100]， B[100];**
    - **memset (A，0, 100*sizeof(int));**
    - 运行结果：将数组A清0
    - **memcpy (A，B, 100*sizeof(int));**
    - 将B的内容拷贝到A指向的空间

常用动态内存分配函数： free函数

- 函数原型：void **free**(**void** ***p**);
- 功能：释放指针p指向的存储空间，free函数无返回值
- 使用动态内存分配函数需：#include “**stdlib.h**”
- 良好的编程习惯
  - 最好在**同一个函数内**动态分配和释放存储空间
  - 最好在定义指针时将指针初始化为**NULL**
  - 最好在释放指针后也将指针赋值为**NULL**
  - 这样便于使用p==NULL语句判断指针是否有效

```
int A[100], B[100];
memset ( A, 0, 100*sizeof(int) );
```

如果把0改成1呢？会发生什么？

你可以试试，然后打印输出

行，我试一下 …… 不过，这是什么东西？？

```
16843009 16843009 16843009
Process returned 0 (0x0)
Press any key to continue.
```

额……你猜？

地址？

地址?

您呼叫的用户目前不在服务区，嘀一声后请留言

..... 啊，我知道了!

对于每个字节来讲，用1填充后变为：00000001

整数四个字节的值为：2^24+2^16+2^8+1 =

16843009!

咳咳，那什么，其实我知道，跟你说怕你印象不深

线性表操作示例1：合并顺序表

合并顺序表：问题需求

1. 给定两个顺序表LA和LB

2. 要求将LB中的元素合并到LA中

3. 并确保合并后LA中的元素不重复

例1：合并线性表（采用动态存储）

- 已知：集合A和B分别用两个线性表LA和LB表示
- 需求：求A∪B并用线性表LA表示
- 算法设计思路
  - 从LB中逐一取出元素，判断该元素是否在LA中
  - 若在则pass，若不在则将该元素插入到LA中
  - 物理结构设计：采用顺序表
- 思考：怎样实现上述算法?
  - 首先需要构造LA和LB
  - 然后需要从LB中逐一取出元素 $\mathbf{b_i}$
  - 接着判断 $\mathbf{b_i}$ 是否已在LA中
  - 如果$\mathbf{b_i}$不在A中，则将 $\mathbf{b_i}$ 插入LA中

例1：合并线性表（采用动态存储）

- 算法设计
  - 如何构造顺序表LA和LB?
    - 使用动态存储方式声明和定义（注意内存回收）
  - 怎样从LB中逐一取出元素 b_i ?
    - 采用循环对LB中元素进行遍历（需要知道元素个数）
  - 怎样判断 b_i 是否已在LA中?
    - 循环遍历LA + 元素比较
  - 如果b_i不在A中，怎样将 b_i 插入LA中?
    - 查找适当的插入位置 + 元素移动 + 元素拷贝
  - 提示：编写函数实现对数据的操作!

例1：合并线性表算法

```
int list_union (TList*  La, TList * Lb){
   ElemType e;  int idx, status, i, j, n = Lb->length;
   ……    // 初始化函数调用(略)
   for ( i = 0; i < n; ++i ){
        e = get_item(Lb, i);
        idx = list_locate(La, e);
        if( !idx ){   // 如果e不在La中，插入到第一个位置
                status = list_insert(La, 0, e);
                if(status) { printf("插入元素出错");  exit(0);}
        }
   }
   …… // 善后处理函数调用(略)
   return status;
}
```

怎样构造LA和LB

- 首先确定顺序表中存储的数据（元素）类型
  - 为简化示例我们使用如下的自定义类型

```
typedef  int  ElemType;

ElemType list[100];
```

怎样构造LA和LB

- **接下来确定线性表的数据类型**
  - 思考：如果ElemType为int型，我们会怎么做？
    - 如果采用静态存储： **int a[nsize];** // nsize 为常量
    - 如果采用动态存储？
    - 提示：数组名是什么？ **指针！ int *pa;**
  - 类似地可以将线性表定义为指针类型： **ElemType***

ElemType *plist;

那么问题来了：如何保存表长信息？

保存线性表长度信息

- 为了保存线性表的长度信息，修改线性表声明如下

```
typedef struct {
    ElemType * data;    // 数据指针
    int length;         // 当前表中元素个数
    int capacity;       // 记录线性表容量
} TList;

TList * plist;          // 线性表声明
```

怎样构造LA和LB

- 线性表初始化
  - 函数声明： **TList* list_init (int num);**
  - 参数：线性表的表长
  - 返回值： TList类型的指针，指向长度为num的空表
- 销毁线性表
  - 函数声明： **void list_destroy(TList* p);**
  - 参数：线性表指针
  - 操作结果：释放表p所占空间

线性表初始化

```
TList * list_init (int num) {
    TList *p = (TList *) malloc(sizeof(TList));
    if( !p ){ printf("list_init申请空间失败");  exit(0);  }
    p->length = 0;          // 空表：当前表中元素个数为零
    p->capacity = num;  // 该线性表的总容量
    int nsize = sizeof(ElemType) * num;
    p->data = (ElemType*) malloc(nsize);
     if( !p->data ){
            printf("list_init申请空间失败");
            free(p);   exit(0);
     }
     return p;
}
```

销毁线性表

```
void list_destroy(TList * p){
        if( p ) {
                if( p->data ){
                        free(p->data);
                }
                free(p);
        }
}
```

例1：合并线性表（采用动态存储）

- 算法设计
  - 如何构造顺序表LA和LB?
    - 使用动态存储方式声明和定义（注意内存回收）
  - 怎样从LB中逐一取出元素 bi ？
    - 采用循环对LB中元素进行遍历（需要知道元素个数）
  - 怎样判断 $\mathbf{b_i}$ 是否已在LA中?
    - 循环遍历LA + 元素比较
  - 如果$\mathbf{b_i}$不在A中，怎样将 $\mathbf{b_i}$ 插入LA中?
    - 查找适当的插入位置 + 元素移动 + 元素拷贝
  - 提示：编写函数实现对数据的操作!

怎样从LB中逐一取出元素?

- 函数原型：**ElemType get_item(TList* L, int idx)**
  - 参数：顺序表L和拟取出的元素在表中的下标idx
  - 若下标合法，返回相应元素值，否则给出错误提示

```
ElemType get_item(TList* L, int idx){
        if ((idx >= 0)&&(idx < L->length)){
                return L->data[idx];
        }
        else{
                printf("元素位置不合法"); exit(0);
        }
}
```

例1：合并线性表（采用动态存储）

- 算法设计
  - 如何构造顺序表LA和LB?
    - 使用动态存储方式声明和定义（注意内存回收）
  - 怎样从LB中逐一取出元素 bi?
    - 采用循环对LB中元素进行遍历（需要知道元素个数）
  - 怎样判断 bi 是否已在LA中?
    - 循环遍历LA + 元素比较
  - 如果$\mathbf{b_i}$不在A中，怎样将 $\mathbf{b_i}$ 插入LA中?
    - 查找适当的插入位置 + 元素移动 + 元素拷贝
  - 提示：编写函数实现对数据的操作!

怎样判断 bi 是否已在LA中

- 按值查找操作函数原型

int list_locate(TList *L, ElemType e)

- 初始条件：表L存在， e 是给定的一个数据元素
- 操作结果：在表L中查找首个值为 e 的数据元素
  - 若找到则返回其序号
  - 否则返回 -1（查找失败）

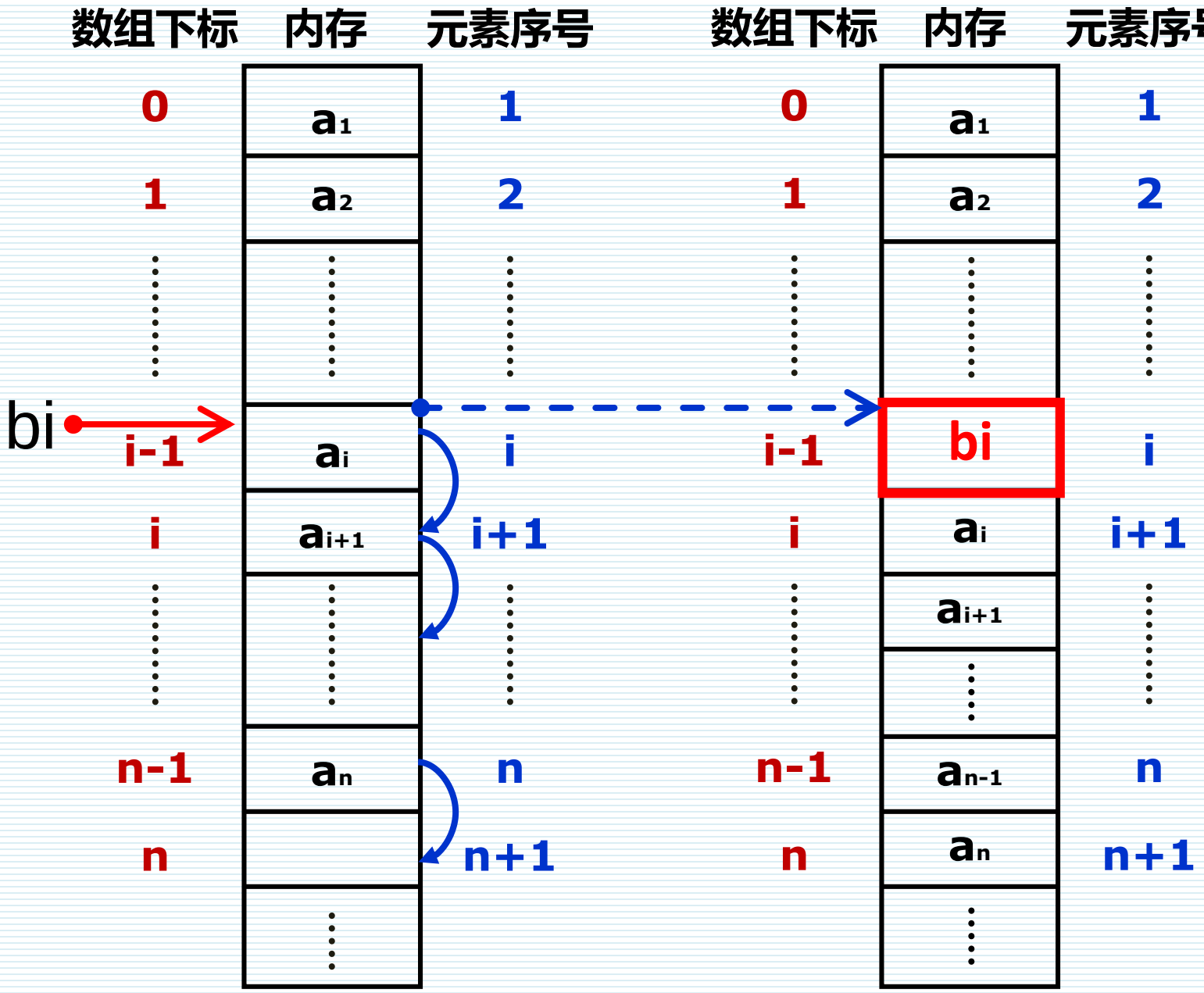
怎样判断 bi 是否已在LA中

```
int list_locate (TList* L, ElemType e) {
        int id = 0;
       while ((id < L->length)&&( e != L->data[id]))
                id++;
        if (id < L->length)  return i;
        else{
                printf("此元素在顺序表中不存在" );
                return 0;
        }
}
```

例1：合并线性表（采用动态存储）

- 算法设计
  - 如何构造顺序表LA和LB?
    - 使用动态存储方式声明和定义（注意内存回收）
  - 怎样从LB中逐一取出元素 b_i ？
    - 采用循环对LB中元素进行遍历（需要知道元素个数）
  - 怎样判断 b_i 是否已在LA中?
    - 循环遍历LA + 元素比较
  - 如果bi不在A中，怎样将 bi 插入LA中?
    - 查找适当的插入位置 + 元素移动 + 元素拷贝
  - 提示：编写函数实现对数据的操作!

如果bi不在LA中，怎样实现将bi 插入LA中



怎样插入元素

- 向线性表中插入元素的函数原型
  - **int list_insert(TList* L, int idx, ElemType e)**
  - 参数：S-目标线性表；idx-插入位置；e-待插入的元素
- 功能描述：令 n = L->length
  - 若0<= idx <= n-1，且表中还有未用的存储空间
    - 在线性表L下标为idx的位置上插入值为 e 的新元素
    - 原下标为 idx+1 … n -1的数据元素的序号变为 idx +2，… n
    - 修改：表长 = 原表长+1，返回0（success）
  - 若表中没有可用空间：返回 1 （overflow）
  - 若插入的位置不合理：返回-1 （range_error）

怎样插入元素

```
int list_insert(TList *L, int idx, ElemType e) {
    int i, n = L->length;
    if (n >= L->capacity){
        return  1;      // 顺序表表满，插入元素会导致上溢
    }
    else if ((idx< 0) || ( idx > n)){
        return  -1;     // 插入位置不合理
    }
    else  {
       for( i = n;  i > idx;  --i ) {    // 数据元素右移
                L->data[i] = L->data[i-1];
       }
        L->data[idx] = e;          // 数据元素拷贝
        L->length ++;              // 修改表长
    }
    return 0;
}
```

思考：怎样删除指定元素？

例1：合并线性表 — 小结

- 给定：集合A和B分别用两个线性表LA和LB表示
- 需求：求A∪B并用线性表LA表示
- 用基本操作（自定义函数）实现
  - 初始化LA和LB（分配内存并填充给定内容）
  - 依次从LB中取出第 i 个数据元素
    - list_get(Lb, i) ;
  - 判 bi 是否在LA中存在?
    - list_locate(La, bi);
  - 如果 bi不在A中，则将 bi 插入到LA中
    - list_insert(La, idx, bi) ;
  - 最后销毁线性表LB，程序返回

顺序表小结

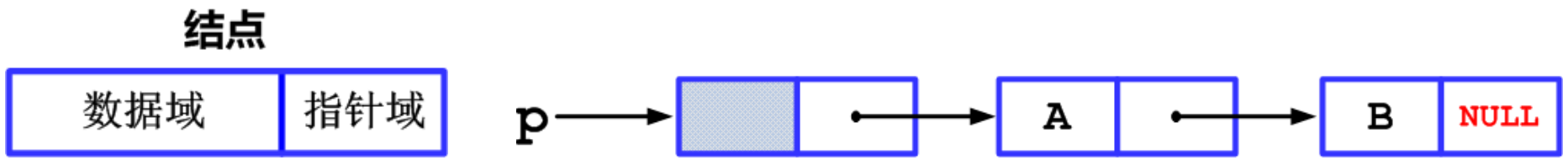
- **线性表的顺序存储结构的优缺点**
  - **优点**
    - **逻辑上相邻的元素，物理存储位置也相邻**
    - **可随机存取任一元素（通过地址计算）**
    - **存储空间使用紧凑（连续存储）**
  - **缺点**
    - **插入或删除操作需要移动大量的元素**
    - **需按最大容量预先分配空间（空间利用不充分）**
    - **线性表容量难以扩充**

线性表的链式存储结构

线性表的链式存储结构

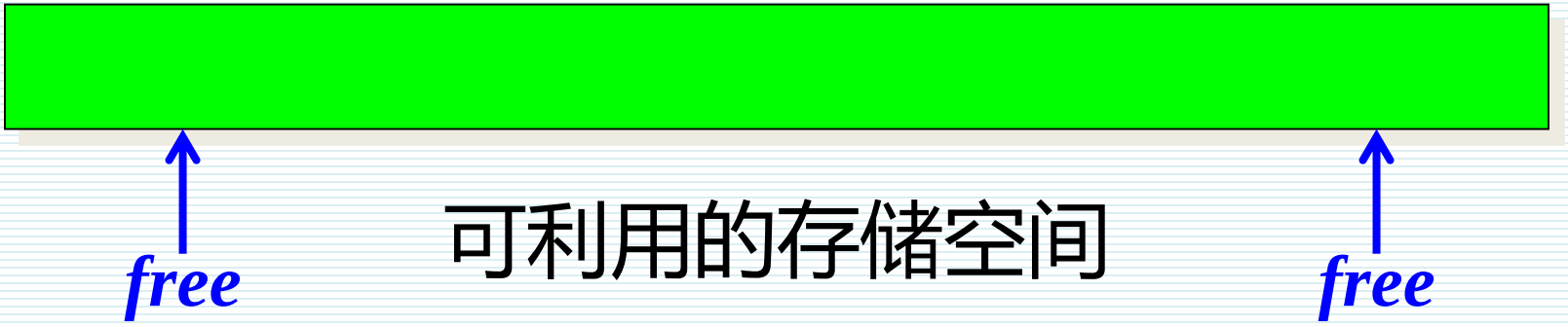
- 线性表的链链式存储是指用结点表示线性表的元素
  - 数据元素之间的偏序关系采用指针表达
- 按照逻辑结构，可以将线性链表分为如下四类
  - 单链表
  - 循环链表
  - 双向链表
  - 循环链表

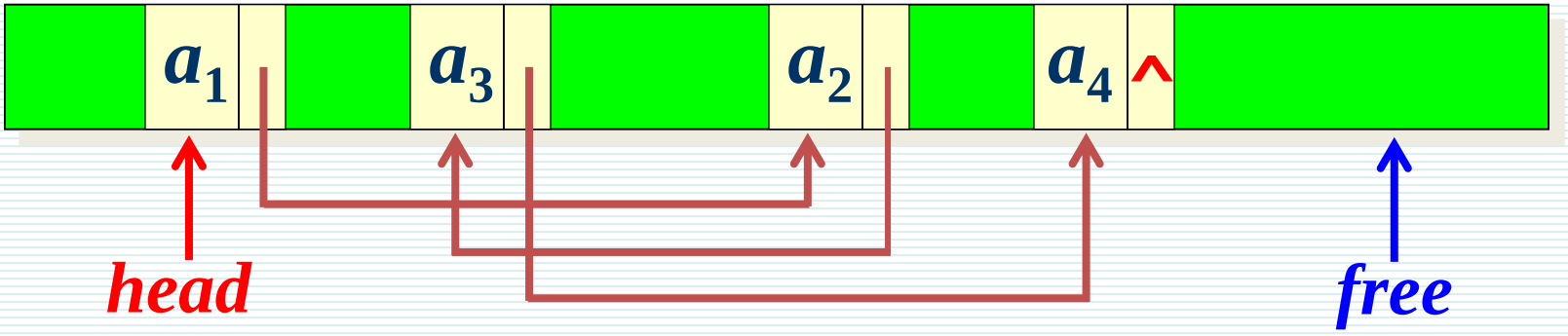
线性表的链式存储结构



- 特点：
  - 每个数据元素 a_i（称为结点）由两部分构成
    - 数据域：存储元素本身信息
    - 指针域：指示直接后继的存储位置（Next指针）
  - 用一组任意的存储单元存储线性表的数据元素
    - 用不连续的存储单元存放逻辑上相邻的元素
  - 线性表可按需进行扩充

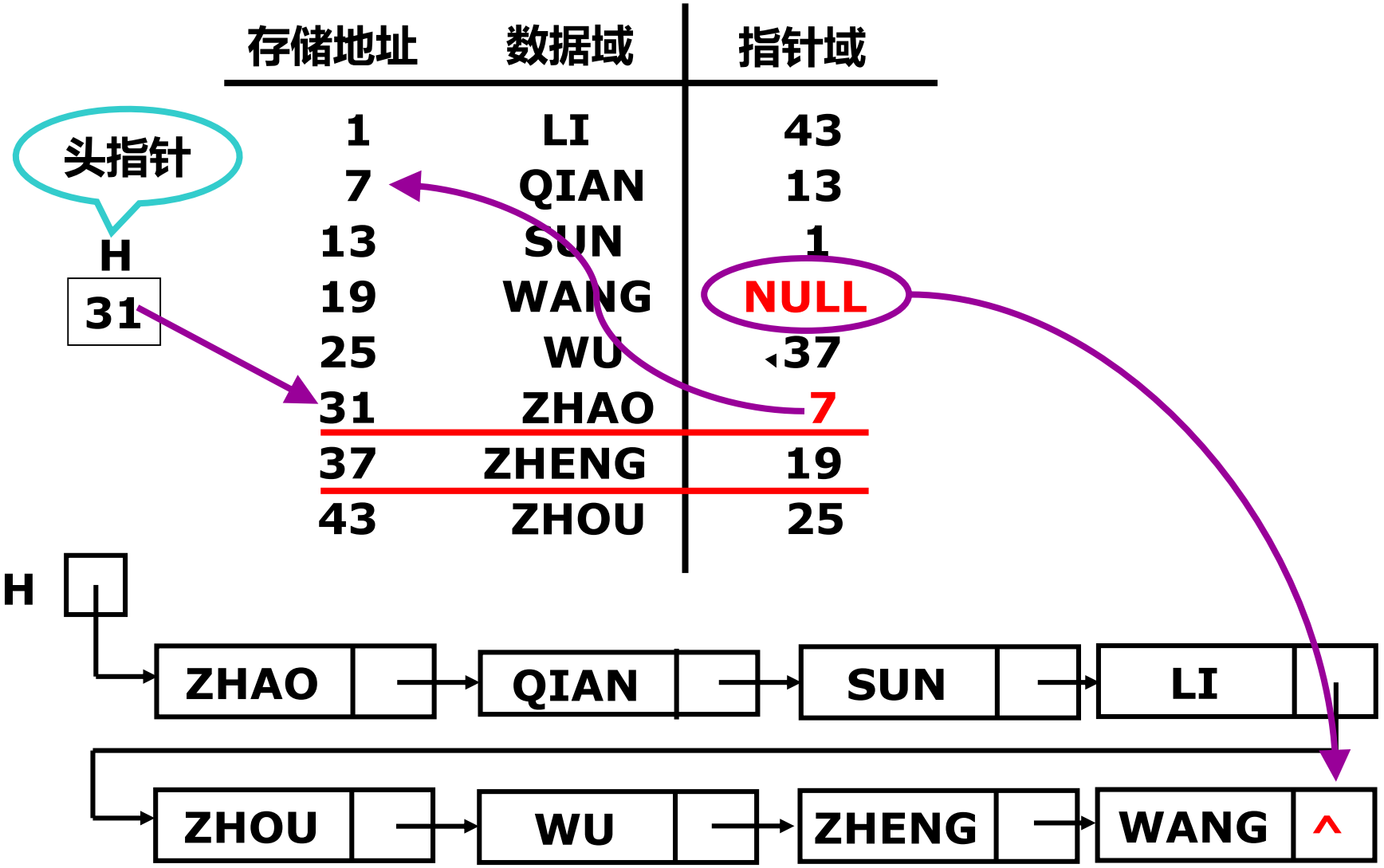
单链表的存储映像





经过一段运行后的单链表结构

例：线性表 (ZHAO, QIAN, SUN, LI, ZHOU, WU, ZHENG, WANG)



一个指针就是一个数！ **最后一个单元的Next指针指向NULL**

线性链表（也称为单链表）

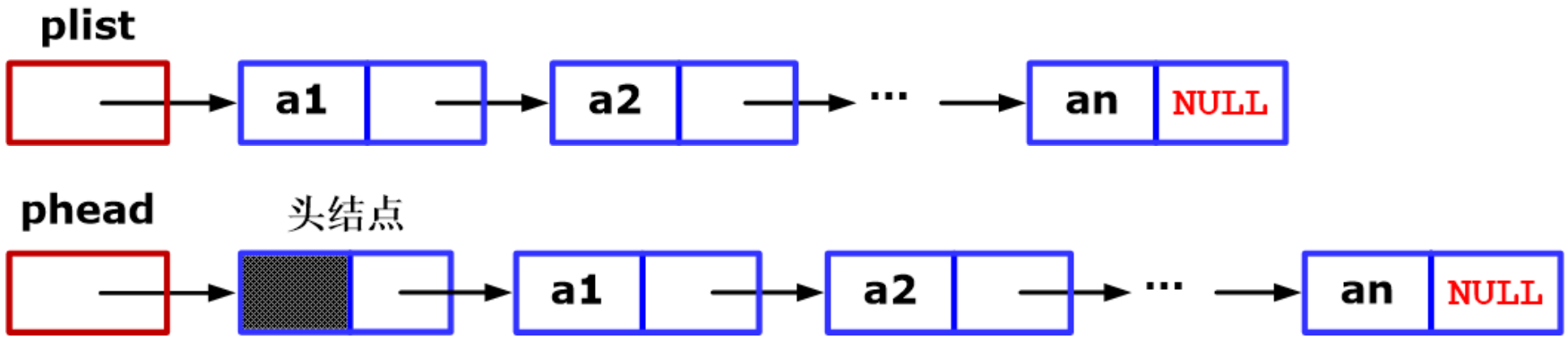
定义：结点中只含一个指针域的链表称为线性链表

```
typedef struct node {
    ElemType data;
    struct node  *next;
}TNode,*PList;
```


```
PList  phead;  // ⇔ TNode *phead;   链表头指针
```

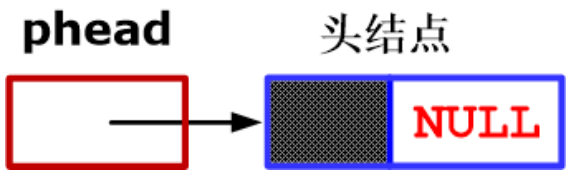
- (*p)　　表示p所指向的结点
- p->data　　⇔ (*p).data 表示p指向结点的数据域
- p->next　　⇔ (*p).next 表示p指向结点的指针域
- 生成新结点　　p =(Tnode*)malloc(sizeof(TNode));
- 销毁结点：　　free(p) // 回收内存

带头结点和不带头结点的链表



- 不带头结点的单向链表（如图 plist 所示）
  - 头指针存放链表第一个节点的地址
  - 对空链表，则头指针为NULL

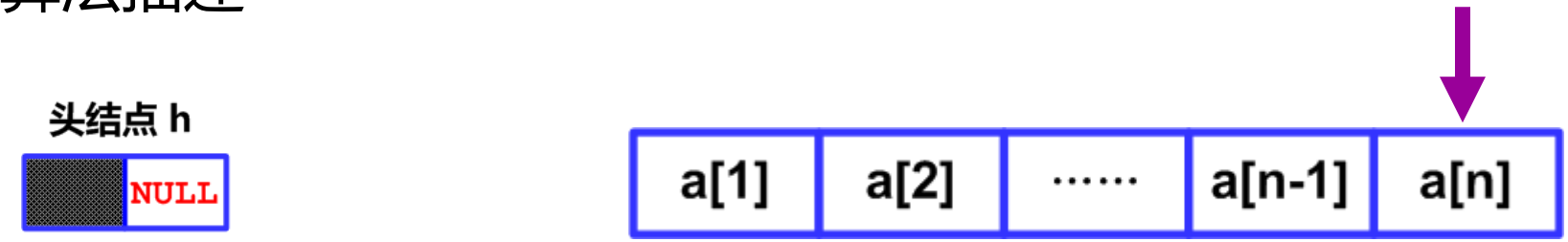
- 带头结点的单链表（如图**phead**所示）



  - 在单链表第一个结点前附设一个专门结点，称为头结点
  - 该头结点指针域存放第一个元素的指针/地址
  - 该头结点永远存在：空表的头结点的指针域为NULL

动态建立单链表

- 假设线性表n个元素已存放在数组a中
- 要求：建立一个单链表，以h为头指针
- 算法描述



内容回顾：单链表的数据结构定义

```
typedef struct node {
    ElemType data;
    struct node  *next;
}TNode,*PList;
```

```
PList  phead;  // ⇔ TNode *phead;   链表头指针
```

思考：现在有头结点了吗?

创建带头结点的单链表：头插法

```
PList  list_create(ElemType* pa, int n){
    int i;  PList phead, pnode;
    phead = (PList)malloc(sizeof(TNode)); // 判空略
    phead->next = NULL;  // 建立带头结点的空表
    for(i=n-1; i>=0; --i){
       pnode = (PList)malloc(sizeof(TNode));
       if(!pnode){
              exit(0);  // 判空，若内存申请失败则退出
       }
       pnode->data = pa[i];
       // 将新结点插入到表头位置（头结点之后）
       pnode->next =  phead->next;
       phead->next = pnode;
   }
   return phead ;
}
```

算法复杂度?

$$T(n)=O(n)$$

创建带头结点的单链表：尾插法

```
PList list_create(ElemType* pa, int n){
    int i;  PList phead, ptail, pnode;
    phead = (PList)malloc(sizeof()); // 判空略
    phead->next = NULL;  // 建立带头结点的空表
    ptail = phead;                  // 设置一个尾指针，方便插入
    for(i=0; i<n; ++i){
        pnode = (PList)malloc(sizeof(TNode));
        if(!pnode){
            printf("申请空间失败！"); exit(0);  // 判空
        }
        pnode->data = pa[i]; pnode->next = NULL;
        // 将新结点插入到表尾位置
        ptail->next = pnode;
        ptail = pnode;
    }
    return phead ;
}
```

算法复杂度?

$$T(n)=O(n)$$

带头结点的单链表的基本运算：求表长

```
int length(PList phead){
    phead = phead->next;    // 思考：为什么这样做?
    int len = 0;
    while(phead != NULL){
        len++;
        phead = phead->next;  // 移向下一个结点
    }
    return len;
}
```

算法复杂度?

$$T(n)=O(n)$$

课堂作业：请模仿写出求不带头结点的单链表长度的算法

带头结点的单链表的基本运算：取指定元素

```
PList get(PList phead, int idx)
{
    int i = 0;
    while( phead && (i < idx) ){
        phead = phead->next;
        i++;
    }
    return phead;
}
```

请课后模仿写出在不带头结点的单链表中取指定元素的算法

带头结点的单链表的基本运算：查找

- **查找单链表中是否存在结点x**
  - 若有则返回指向x结点的指针，否则返回NULL

while循环中比较和赋值语句的频度为
- 若x在链表中：语句频度为x的序号
- 若x不在链表中：语句频度为n

```
PList locate(PList phead, ElemType e){
    int i = 0;
    PList p = phead->next;
    while(p && (p->data != e)){
        i++;
        p = p->next;
    }
    return p;
}
```

算法复杂度？

$$T(n)=O(n)$$

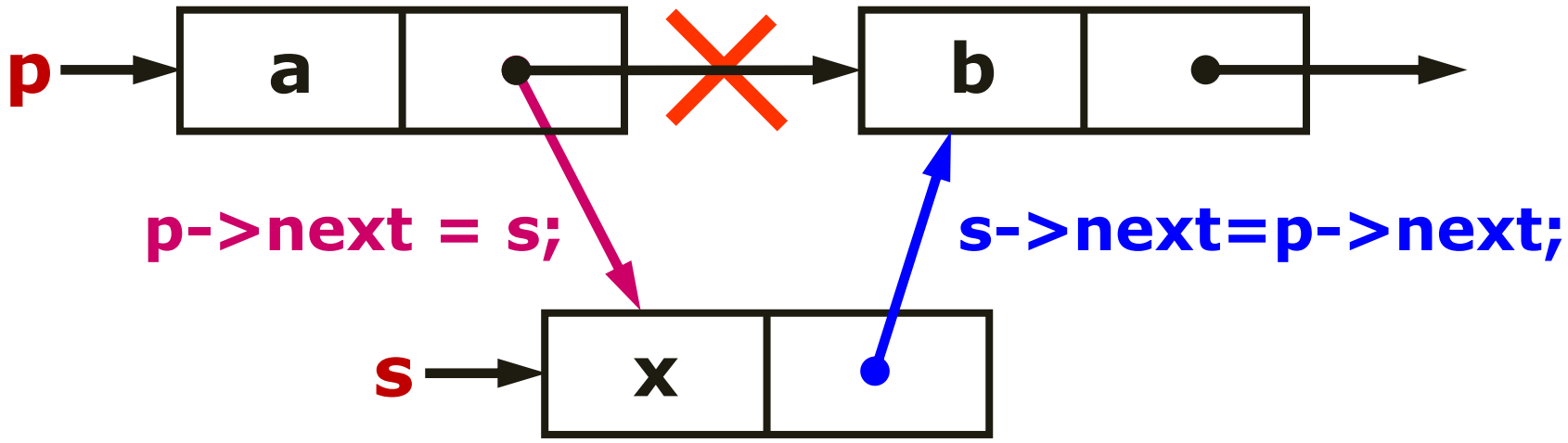
思考：如果不用p指针？

定位带头单链表的第idx个结点：返回指向该结点的指针

```
Plist get_position(PList phead, int idx) {
    PList p = NULL;  int i = 0;
    while(phead && i < idx ){
        i++;
        phead = phead ->next;
    }
    if (phead && i == idx ){
        p = phead;
    }
    return p;
}
```

单链表的基本运算：插入

- 插入：在线性表的某个给定结点**a**之后插入结点**x**
  - 算法描述（已知指针**p**指向结点 **a**，**s** 指向 **x**）



注意：对 p->next 进行赋值时，原来存放的地址被覆盖

思考：若a是链表的最后的一个结点?

带头结点的单链表：在第idx个结点前插入给定结点

```
int insert(PList phead, int idx, ElemType e){
    int status=0;  PList pre, pnode;
    // 寻找插入位置的前驱结点，以pre指向它
    pre = get_position(phead, idx-1);
    if (pre) {
        pnode = (PList) malloc(sizeof(TNode));
        if (pnode) {
                pnode->data = e;
                pnode->next = pre->next;
                pre->next = pnode;
        }else{
                status = -1;  // 为新结点分配空间失败
        }
    }
    return status;
}
```

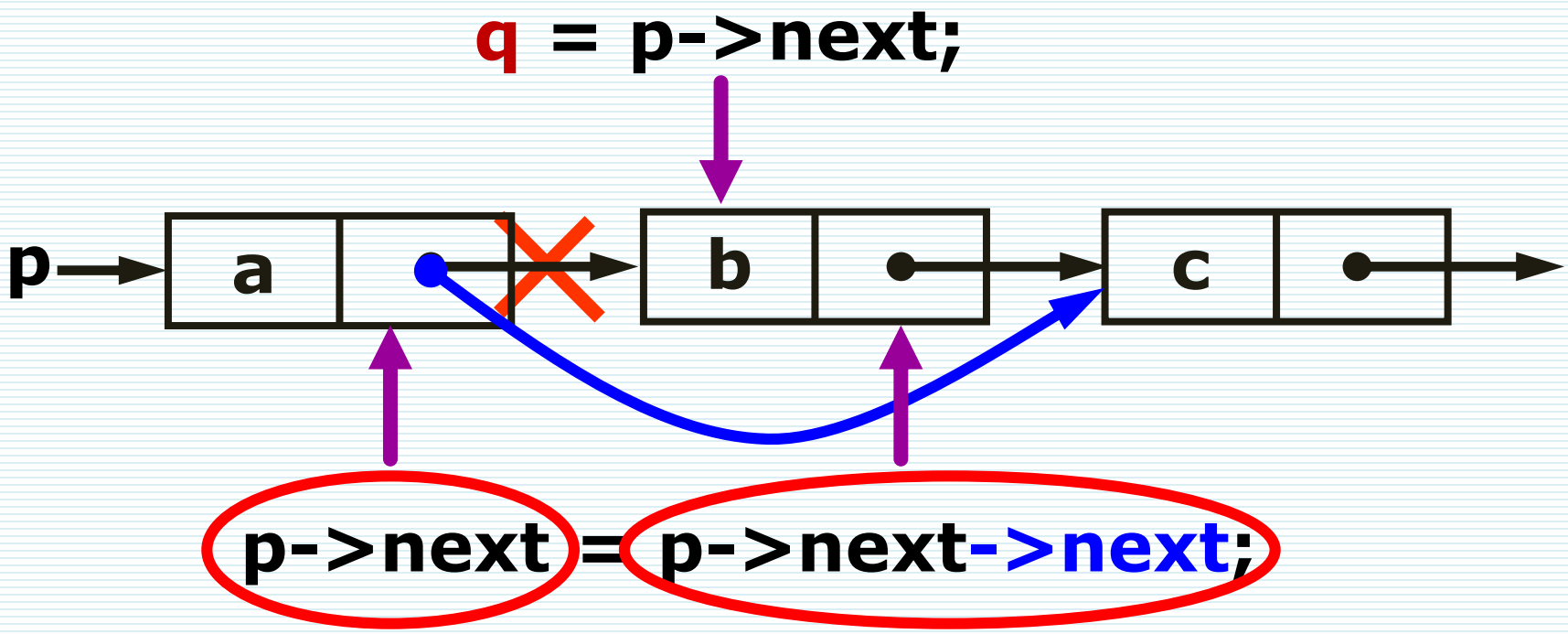
请思考：怎样删除指定元素?

不带头结点的单链表：在结点p之前插入给定结点

```
void insert(PList *pph, PList p, ElemType e){
   PList pre, phead = *pph;
   PList pnode = (PList)malloc(sizeof(TNode));
   if(pnode == NULL){ printf("申请空间失败！"); exit(0); }
   pnode ->data = e;
   if(p == phead){   // 在表头插入
        pnode->next = phead;   *pph = pnode;  return;
   }
   pre = phead;       // 在表的中间或末尾进行插入
   while(pre && (pre->next != p)) pre = pre->next;
   pnode ->next = pre->next;
   pre->next = pnode;
}
```

单链表的基本运算：删除

- 从单链表中删除结点 b（设p指向b的前驱结点a）



q->next = NULL; 或 free(q);

带头结点的单链表的基本运算：删除指定位置结点

```
void remove(PList phead, int idx) {
    PList p, q=NULL;
    p = get_position(phead, idx-1);  // 查找前驱
    if (p){
        q = p->next;
        p->next = q->next;
        if(q) {
            free(q);
        }
    }
}
```

不带头结点的单链表的基本运算：删除指定内容结点e

```
void remove(PList *pplist, E e) {
    PList p = *pplist, pre = p;
    while(p && (p->data != e)){ // 查找前驱
        pre = p; p = p->next;
    }
    if(p){  // 查找到指定结点e
        if(pre == *plist){  // 删除第一个结点
            *pplist = p->next; free(p);
        }
        else{
            pre->next = p->next; free(p);
        }
    }
}
```

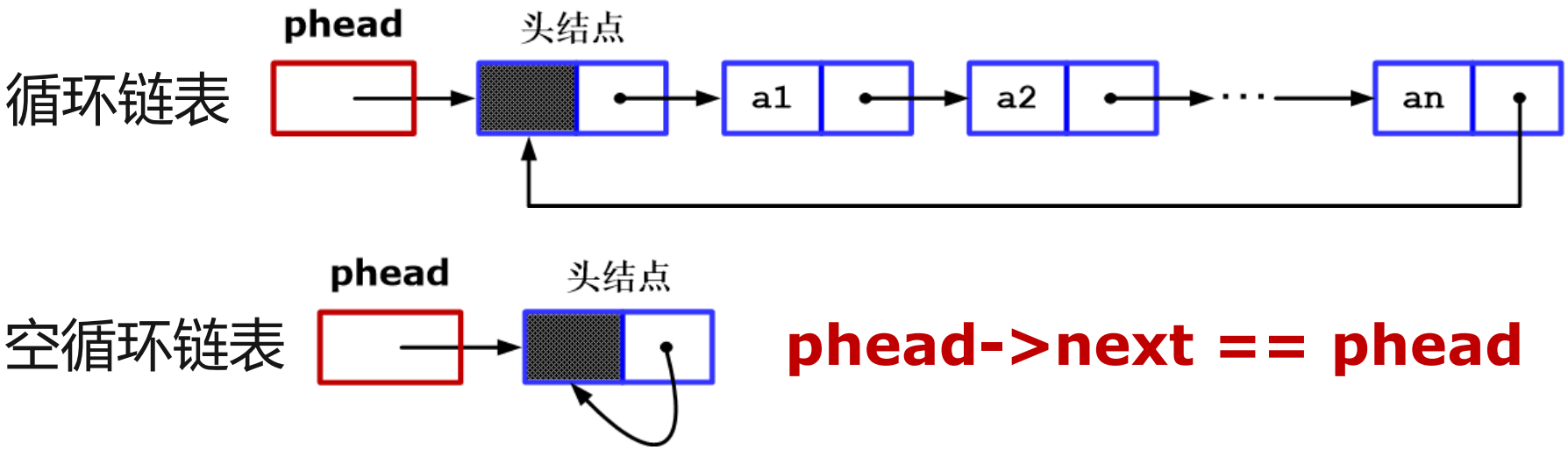
带头结点的单链表保序合并操作

```
void merge(PList La, PList Lb){
    PList pa, pb, ptail;
    pa = La->next, pb = Lb->next; ptail = La;
    while(pa && pb){
        if(pa->data <= pb->data){
            ptail->next = pa;  ptail = pa;  pa = pa->next;}
        else{
            ptail->next = pb;  ptail = pb;  pb = pb->next;}
    }
    if(pa) ptail->next = pa;  // LA中尚有剩余结点
    else ptail->next = pb;
    free(Lb);                 // 将Lb的表头释放
}
```

单链表小结

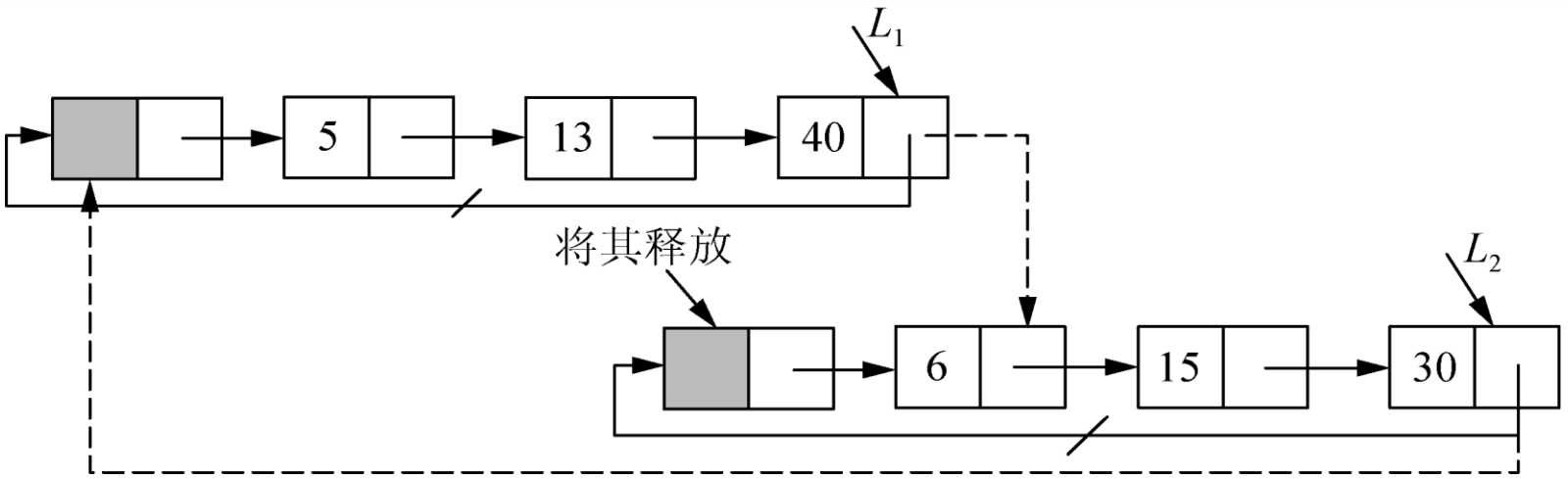
- 单链表的特点
  - 它是一种动态结构
    - 不需预先分配空间
    - 链表结点使用的内存空间的地址是非连续的
  - 内存开销方面
    - 为保存指针需占用额外存储空间
    - 可以按需分配内存
  - 基本操作的效率
    - 不能通过地址计算进行元素访问：即不能随机存取
    - 元素查找速度慢

循环链表(circularly linked list)



- 环状链表：表中最后一个结点的指针指向头结点
- 特点：从表中任一结点出发均可找到表中其他结点
- 操作与单链表基本一致　**但循环条件不同！**
  - 单链表尾结点：　p->next == NULL
  - 循环链表尾结点：　p->next == phead

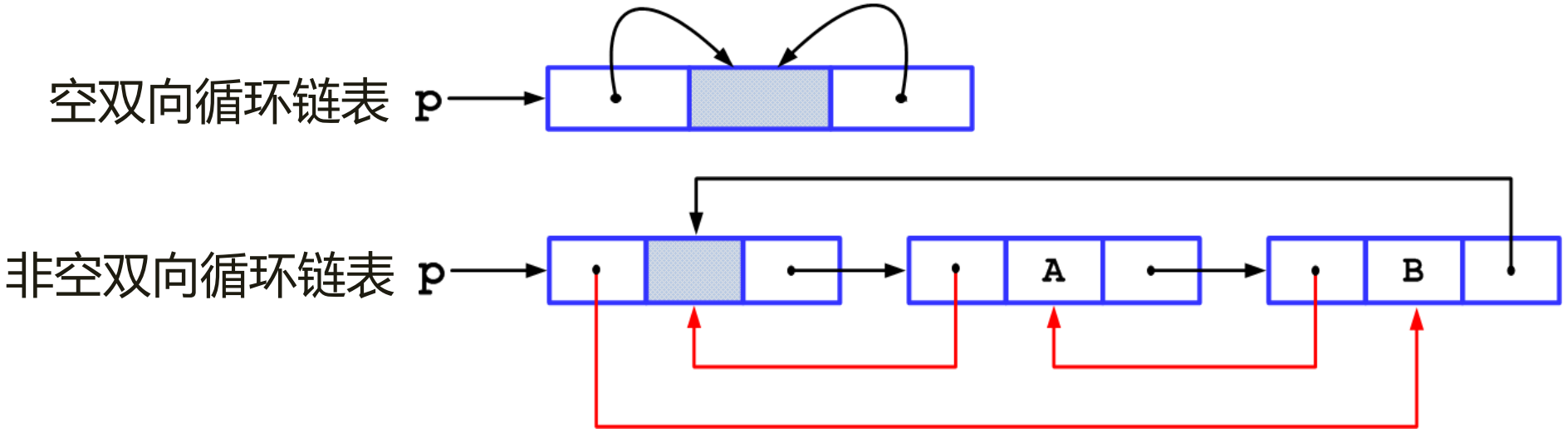
两个单向循环链表的合并



- 实际工作中常用尾指针表示单向循环链表

```
void join(PList L1, PList L2) {
    PList p = L1->next;          // 记录L1的头结点
    L1->next = L2->next->next;
    free(L2->next);              // 释放L2的头结点
    L2->next = p;
}
```

双向循环链表（double circular linked list）



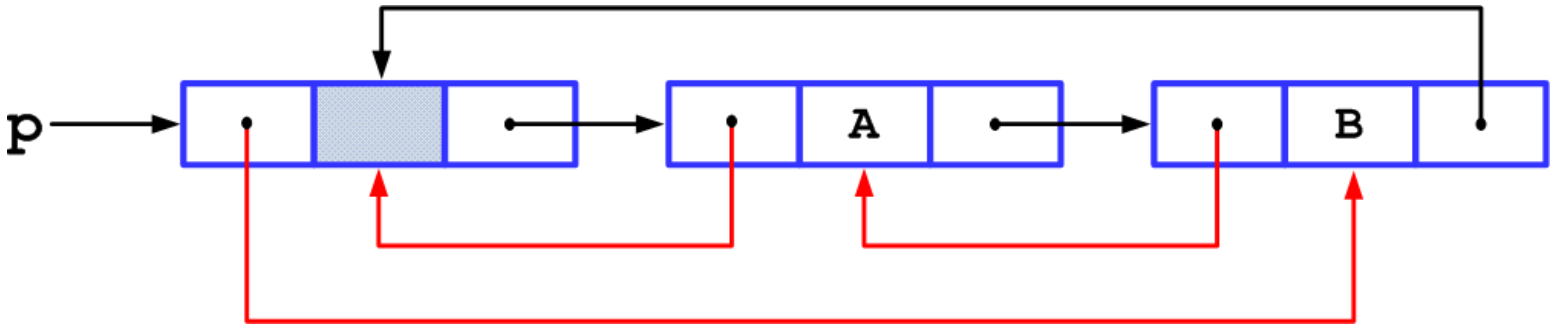
- 为克服单链表的单向性缺点，引入双向链表的概念

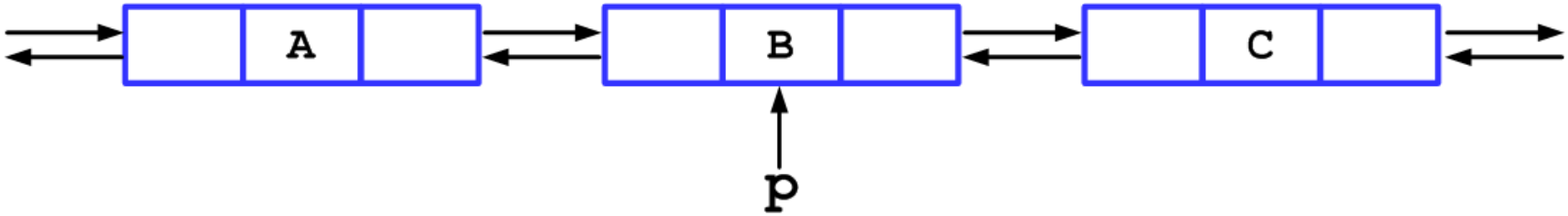
双向循环链表的结点构成

结点定义

```
typedef struct dnode {
        ElemType  data;
        struct dnode *pre,*next;
} Tnode, *PList;
```

双向循环链表（double circular linked list）

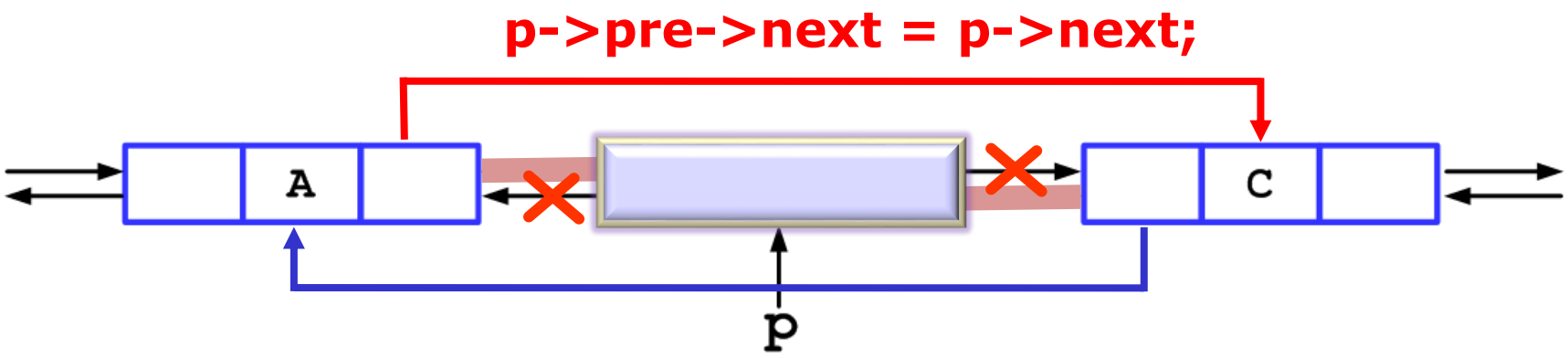




思考： **p->pre->next = ?**

p->pre->next = p = p->next->pre

双向循环链表的元素删除操作



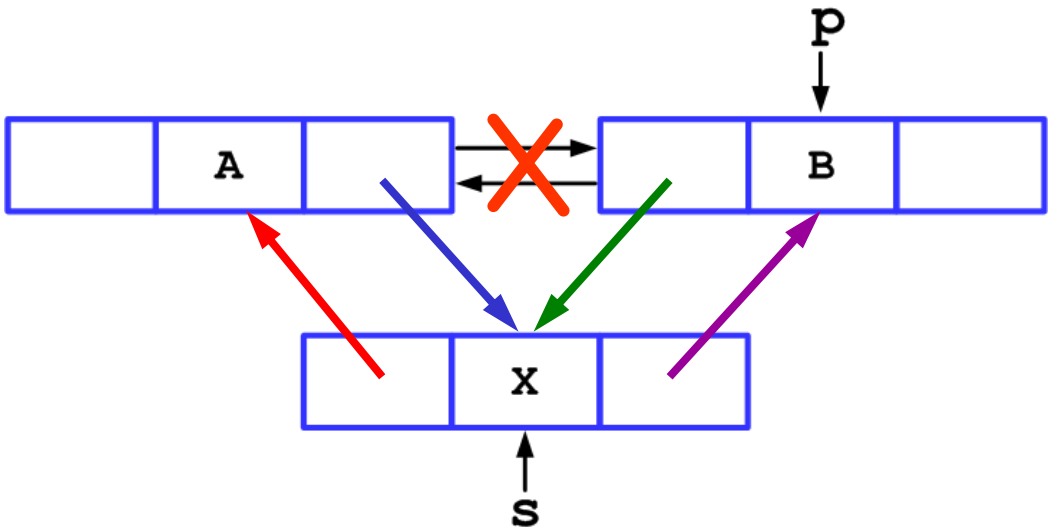

```
int dulist_delete(PList p, int i, ElemType *e) {
    if (!( p = get_position(p, i))) return 1;
    *e = p->data;
    p->pre->next = p->next;
    p->next->pre = p->pre;
    free( p );
    return 0;
}
```

算法复杂度?

$T(n)=O(1)$

请思考：如果删除后表为空?

双向循环链表的元素插入操作




```
int dulist_insert(PList p, int i, ElemType x) {
    if (! (p= get_position(p, i)))  return 1;
    if (! (s= (PList)malloc(sizeof(TNode))))
            return 1;   // 为新结点申请内存失败
    s->data= x;
    s->pre = p->pre;  p->pre->next = s;
    s->next = p;   p->pre = s;
    return 0;
}
```

算法复杂度?

$T(n)=O(1)$

请思考：如果在p后插入，如何操作?

例2：线性表的倒置

例2：线性表倒置

- 问题：把线性表 $(a_1, a_2, \ldots, a_n)$ 变为逆序 $(a_n, a_{n-1}, \ldots, a_1)$
  - 要求：分别采用顺序表和单链表进行实现
- 顺序表：对应数据元素进行交换
  - 如a_1与a_n交换、a_2与a_{n-1}交换，以此类推 ……
- 单链表：改变前驱后继关系
  - 倒置前：a_1是a_2的前驱、a_2是a_1的后继
  - 倒置后：a_1是a_2的后继、a_2是a_1的前驱

例2：线性表倒置：采用顺序表（动态分配内存）

```
void  list_reverse (TList *pa){
        int start=0, end = pa->length - 1;
        ElemType temp;
        while ( start < end ){   思考：能否用for循环?
              temp = pa->list[start];
              pa->list[start] = pa->list[end];
              pa->list[end] = temp;
              start++;  end--;
        }
}
```

例2：线性表倒置：采用单链表

```
void  list_reverse(PList ph){
    PList pa, pb = p->next;
    ph->next = NULL;
    while (pb){
        pa = pb;                    // pa用于”摘除”结点
        pb = pb->next;              // pb用于”跟踪”剩余链表
        pa ->next = ph->next;       // 向表ph插入结点
        ph->next = pa;
    }
}
```

思考：有无头结点?

这是什么意思?

线性表小结：顺序与链式存储结构比较

顺存储结构的特点	链式存储结构的特点
逻辑上相邻的元素 其物理位置也相邻	逻辑上相邻的元素 其物理位置不一定相邻
可随机存取表中任一元素	非随机存取存储结构 元素存取必须从头指针开始
是一种静态存储结构 必须按最大可能预分配存储空间 存储空间利用率低 表的容量难以扩充	是一种动态存储结构
插入删除时需移动大量元素 平均移动元素为n/2	插入删除运算非常方便 只需修改相应指针值

作业1

1. 编程练习：单链表的基本操作

   - 上机实现的**带头结点的**和**不带头结点的**单链表的所有相关操作，并编写main()函数进行验证。

2. 编程练习：用c语言编程实现带头单向循环链表

   - 实现创建、删除、插入、判断是否为空、清空等操作

2. 编程练习：用c语言编程实现带头双向循环链表

   - 实现创建、删除、插入、判断是否为空、清空等操作

本章作业

- 作业提交方式：发至助教邮箱 **373230727@qq.com**
  - 源代码以.c后缀保存为文本文件（第一题示例：1.c）
  - 将三道题的.c文件打包成一个rar文件（不要工程文件）
  - rar文件命名方式：学号-姓名-HW2.rar
  - **特别提示：文件提交格式错误视为未提交作业！**
  - **截止时间：2019年3月28日晚24:00（过时无效）**


UESTC
1956