§2. 线性表

2.1. 线性表的类型定义

2.1.1. 线性结构的特点

★ 线性结构：元素间存在一对一的关系

- 存在唯一一个称为"第一"的数据元素
- 存在唯一一个称为"最后"的数据元素
- 除最后一个外，每个元素仅有一个后继
- 除第一个外，每个元素仅有一个前驱

2.1.2. 线性表的含义

具有相同特征的数据元素的有限序列

★ 数据元素可以是任意形式，较复杂的一般表示为一个记录，由若干数据项组成，则整个线性表又可称为文件

2.1.3. 线性表的长度

序列中所含的元素的个数称为线性表的长度，用n(n≥0)表示

★ 当n=0时，表示一个空表，即表中不含任何元素

§2. 线性表

2.1. 线性表的类型定义

2.1.4. 线性表的表示形式

设序列中第i个元素为a_i（$1 \leqslant i \leqslant n$），则线性表一般表示为：

$(a_1, a_2, \ldots, a_i, \cdots, a_n)$

★ a_1为第1个元素，称为表头元素，a_n为最后一个元素，称为表尾元素

★ 一个线性表可以用一个标识符来命名，例：

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

★ 线性表中的元素在位置上是有序的，即第i个元素a_i处在第i-1个元素a_{i-1}后面和第i+1个元素a_{i+1}的前面，这种位置上的有序性就是一种线性关系

★ 称i为数据元素a_i在线性表中的位序

2.1.5. 序偶

$\langle a_{i-1}, a_i \rangle$称为一个序偶，表示线性表中数据元素的相邻关系

★ a_{i-1}称为序偶的第一元素，a_i称为第二元素

a_{i-1}称为a_i的直接前驱，a_i称为a_{i-1}的直接后继

★ 当$i=1, 2, \ldots, n-1$时， a_i有且仅有一个直接后继

当$i=2, 3, \ldots, n$ 时， a_i有且仅有一个直接前驱

2.1.6. 抽象数据类型的线性表的定义(P. 19 - 20)

★ 在写算法时，假设这些基本操作均已实现，可以直接使用

§2. 线性表

2.2. 线性表的顺序表示和实现

2.2.1. 顺序表示的特点

用一组地址连续的存储单元依次存储线性表的数据元素，借助元素在存储器中的相对位置来表示元素间的逻辑关系

★ 假设线性表的每个元素需占用L个存储单元，并以所占的第一个单元的存储地址作为数据元素的起始存储位置，则线性表中第i+1个数据元素的存储位置$Loc(a_{i+1})$和第i个数据元素的存储位置$Loc(a_i)$之间满足下列关系：

$$Loc(a_{i+1}) = Loc(a_i) + L$$

★ 线性表的第i个元素a_i的存储位置和a_1的关系为：

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

★ a_1(表头元素)通常称作线性表的起始位置或基地址

★ 每个元素的存储位置和起始位置相差一个和数据元素在线性表中的位序成正比的常数（即L）

★ 只要确定了线性表的起始位置，即可随机存取表中任一元素

★ C/C++语言中数组具备顺序存储的特点，但数组大小必须固定，因此不直接使用数组，而是用动态申请空间的方法模拟数组，方便线性表的扩大

★ 形式化定义中线性表从1..n，C/C++中数组从0..n-1

假设数据元素为int型，则：

（1）采用数组形式：

```
#define MAX_NUM 100
int a[MAX_NUM];
```

可用a[i]形式访问，当线性表中元素满100后，无法再增加，如果初始值设置很大，则会造成巨大的浪费

（2）采用C动态申请空间模拟数组形式：

```
#define MAX_NUM 100
int *a;
a = (int *)malloc(MAX_NUM * sizeof(int));
```

也可用a[i]形式访问，当线性表中元素满100后，可以再增加，方法：

```
a = (int *)realloc(a, (MAX_NUM+10)*sizeof(int));
```

附：realloc函数的原型定义及使用

```
void *realloc(void *ptr, unsigned int newsize);
```

★ 表示为指针ptr重新申请newsize大小的空间

★ ptr必须是malloc/calloc/realloc返回的指针

★ 新老空间可重合，也可能不重合，若不重合，原空间原有内容会被复制到新空间，再释放原空间

★ 更多内容可自行查阅（Linux下 man realloc 查询）

```
#include <iostream>
#include <stdlib.h> //VS2017可省略
using namespace std;
int main()
{
    void *p;
    p = malloc(1024 * 1024);
    //省略申请是否成功的判断
    cout << p << endl;

    p = realloc(p, 1024 * 1024 + 1024); //换为+4096再试
    //省略申请是否成功的判断
    cout << p << endl;

    free(p);
    return 0;
}
```

用 VS2017
VC++6.0
CodeBlocks
Linux 分别编译
观察两次的p地址是否相同

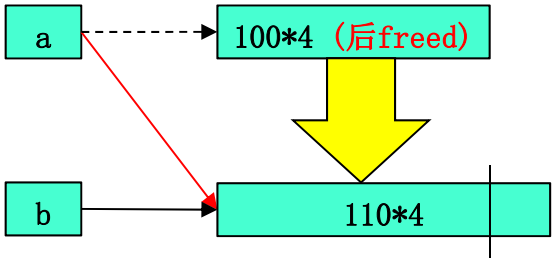
（3）采用C++的动态申请空间模拟数组形式：

```
#define MAX_NUM 100
int *a;
a = new int[MAX_NUM];
```

也可用a[i]形式访问，当线性表中元素满100后，可以再增加，方法：

```
int *b = new int[MAX_NUM+10]; //申请新
for(i=0; i<MAX_NUM; i++)  //原=>新
    b[i] = a[i];
delete a;  //释放原空间
a = b;     //原指针指向新空间
```

思考：如果新空间小于原空间，应如何？



§2. 线性表

2.2. 线性表的顺序表示和实现

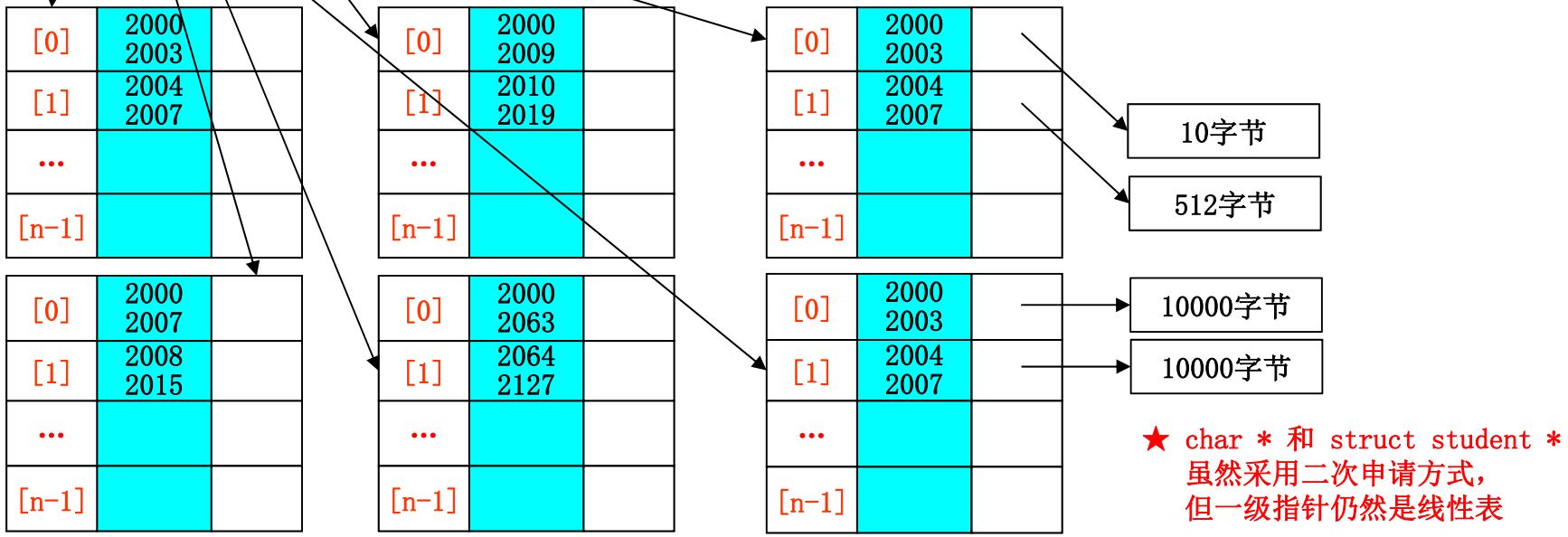
2.2.2. 线性表顺序表示的基本操作的实现

2.2.2.1. C语言版

★ 线性表的数据类型

线性表允许存放任何类型的数据，不失一般性，讨论以下四种类型(六种形式)：

int	：整形数据	
double	：浮点型数据	
char []	：定长或不定长字符串	（图示中假设定长10字节）
char *	：定长或不定长字符串（二次申请）	（图示中假设不定长）
struct student	：复杂结构体（几十 ~ 几千字节）	（图示中假设定长64字节）
struct student *	：复杂结构体（二次申请）	（图示中假设定长10000字节）



★ char * 和 struct student * 虽然采用二次申请方式，但一级指针仍然是线性表

§2. 线性表

2.2. 线性表的顺序表示和实现

2.2.2. 线性表顺序表示的基本操作的实现

2.2.2.1. C语言版

★ 程序的组成

◆ linear_list_sq.h	：头文件
◆ linear_list_sq.c	：具体实现
◆ linear_list_sq_main.c	：使用（测试）示例

说明：
- 从思维上把这个程序理解为两个人/小组完成，其中头文件(.h)和(_sq.c)看做一个人/小组的工作(基本功能的实现)，而将测试程序(_main.c)看做另一个人/小组在使用(用他人提供的基本操作函数来实现自己的应用目标)，两方层次不同(底层向上层提供支持)，适合团队合作和分工
- 假设为一个大型程序中的一个子集
- 程序实现后要进行详尽的测试，通过测试并稳定后，尽量不要修改(设计时尽量考虑得完全一些)
- 测试程序可以修改/调整，只要符合使用要求即可

§2. 线性表

2.2. 线性表的顺序表示和实现

2.2.2. 线性表顺序表示的基本操作的实现

2.2.2.1. C语言版

★ 程序的组成

◆ linear_list_sq.h	：头文件
◆ linear_list_sq.c	：具体实现
◆ linear_list_sq_main.c	：使用（测试）示例

★ 算法与程序的区别

- 算法采用抽象数据接口，抽象数据操作
  => 程序必须有明确的数据定义以及数据操作方法
- 算法的返回值，错误处理都可以抽象
  => 程序必须明确且类型匹配
- 算法可以不定义主程序及配套函数
  => 程序必须补充完整
- 算法可以不定义临时变量
  => 程序必须补充完整

★ 与书上算法的区别

- C语言无引用，需要用指针代替
- 临时变量算法中无定义，程序要补齐
- 某些形式化定义和实际表示之间有区别

§2.线性表

2.2.线性表的顺序表示和实现

2.2.2.线性表顺序表示的基本操作的实现

2.2.2.1.C语言版

★ 程序的组成

★ 算法与程序的区别

★ 与书上算法的区别

- C语言无引用，需要用指针代替
- 临时变量算法中无定义，程序要补齐
- 某些形式化定义和实际表示之间有区别

★ linear_list_sq.h 中各定义项的解析

```
/* linear_list_sq.h 的组成 */
#define TRUE          1
#define FALSE         0
#define OK            1
#define ERROR         0
#define INFEASIBLE   -1
#define OVERFLOW     -2

typedef int Status;
```

P.10 预定义常量和类型

(1) 预定义常量和类型：

```
// 函数结果状态代码
#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;
```

```
/* linear_list_sq.h 的组成 */
#define LIST_INIT_SIZE      100 //初始大小为100(可按需修改)
#define LISTINCREMENT       10  //空间分配增量(可按需修改)



typedef struct {
    int *elem;              //存放动态申请空间的首地址
    int length;             //记录当前长度
    int listsize;           //当前分配的元素的个数
} sqlist;
```

```
/* 相当于两步
   1、先定义结构体类型
   2、用typedef声明为新类型 */
struct _sqlist_ {
    int *elem;
    int length;
    int listsize;
};

typedef struct _sqlist_ sqlist;
```

/* linear_list_sq.h 的组成 */

```
#define LIST_INIT_SIZE      100
#define LISTINCREMENT       10
typedef struct {
    int *elem;
    int length;
    int listsize;
} sqlist;
```

ADT List {
数据对象：D = { a_i | a_i ∈ ElemSet, i = 1,2,…,n, n≥0 }
数据关系：Rl = { <a_{i-1},a_i> | a_{i-1},a_i ∈ D, i = 2,…,n }
基本操作：

InitList(&L)
操作结果：构造一个空的线性表 L。

DestroyList(&L)
初始条件：线性表 L 已存在。
操作结果：销毁线性表 L。

ClearList(&L)
初始条件：线性表 L 已存在。
操作结果：将 L 重置为空表。

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 中数据元素个数。

GetElem(L, i, &e)
初始条件：线性表 L 已存在，1≤i≤ListLength(L)。
操作结果：用 e 返回 L 中第 i 个数据元素的值。

LocateElem(L, e, compare())
初始条件：线性表 L 已存在，compare()是数据元素判定函数。
操作结果：返回 L 中第 1 个与 e 满足关系 compare()的数据元素的位序。若这样的数据元素不存在，则返回值为 0。

Status Status InitList(sqlist *L);

算法转程序时：
★ 引用都表示为指针(C无引用)
★ 每个形参都要有类型定义

```
Status    InitList(sqlist *L);
Status    DestroyList(sqlist *L);
Status    ClearList(sqlist *L);
Status    ListEmpty(sqlist L);
int       ListLength(sqlist L);
Status    GetElem(sqlist L, int i, int *e);
int       LocateElem(sqlist L, int e, Status (*compare)(int e1, int e2));
Status    PriorElem(sqlist L, int cur_e, int *pre_e);
Status    NextElem(sqlist L, int cur_e, int *next_e);
Status    ListInsert(sqlist *L, int i, int e);
Status    ListDelete(sqlist *L, int i, int *e);
Status    ListTraverse(sqlist L, Status (*visit)(int e));
```

★ P.19-20 抽象数据类型定义转换为实际的C语言定义的函数原型说明

- 引用都表示为指针
- 每个形参都要有类型定义，其中compare和visit区别较大

问：当类型是double时，
需要做什么改变？

```
/* linear_list_sq.h 的组成 */
#define LIST_INIT_SIZE      100 //初始大小为100(可按需修改)
#define LISTINCREMENT       10  //空间分配增量(可按需修改)

typedef struct {
    double *elem;           //存放动态申请空间(当数组用)的首地址
    int length;             //记录当前长度
    int listsize;           //当前分配的元素的个数
} sqlist;

Status   InitList(sqlist *L);
Status   DestroyList(sqlist *L);
Status   ClearList(sqlist *L);
Status   ListEmpty(sqlist L);
int      ListLength(sqlist L);
Status   GetElem(sqlist L, int i, double *e);
int      LocateElem(sqlist L, double e, Status (*compare)(double e1, double e2));
Status   PriorElem(sqlist L, double cur_e, double *pre_e);
Status   NextElem(sqlist L, double cur_e, double *next_e);
Status   ListInsert(sqlist *L, int i, double e);
Status   ListDelete(sqlist *L, int i, double *e);
Status   ListTraverse(sqlist L, Status (*visit)(double e));
```

为什么不用换？

问：当类型是char[]时，
需要做什么改变？

```
/* linear_list_sq.h 的组成 */
#define LIST_INIT_SIZE      100 //初始大小为100(可按需修改)
#define LISTINCREMENT       10  //空间分配增量(可按需修改)

typedef struct {
    char (*elem)[10];       //存放动态申请空间(当数组用)的首地址
    int length;             //记录当前长度
    int listsize;           //当前分配的元素的个数
} sqlist;

Status   InitList(sqlist *L);
Status   DestroyList(sqlist *L);
Status   ClearList(sqlist *L);
Status   ListEmpty(sqlist L);
int      ListLength(sqlist L);
Status   GetElem(sqlist L, int i, char *e);
int      LocateElem(sqlist L, char *e, Status (*compare)(char *e1, char *e2));
Status   PriorElem(sqlist L, char *cur_e, char *pre_e);
Status   NextElem(sqlist L, char *cur_e, char *next_e);
Status   ListInsert(sqlist *L, int i, char *e);
Status   ListDelete(sqlist *L, int i, char *e);
Status   ListTraverse(sqlist L, Status (*visit)(char *e));
```

```
double cur_e, double *pre_e);
main:
double d1=5.2, d2;
PriorElem(L,  d, &d2);

main:
char s1[10], s2[10];
PriorElem(L, s1,  s2);

问题1：调用方式不一致，是否正确？
```

问：当类型是char[]时，
需要做什么改变？

```
/* linear_list_sq.h 的组成 */
#define LIST_INIT_SIZE      100 //初始大小为100(可按需修改)
#define LISTINCREMENT       10  //空间分配增量(可按需修改)

typedef struct {
    char (*elem)[10];       //存放动态申请空间(当数组用)的首地址
    int length;             //记录当前长度
    int listsize;           //当前分配的元素的个数
} sqlist;

Status   InitList(sqlist *L);
Status   DestroyList(sqlist *L);
Status   ClearList(sqlist *L);
Status   ListEmpty(sqlist L);
int      ListLength(sqlist L);
Status   GetElem(sqlist L, int i, char (*e)[10]);
int      LocateElem(sqlist L, char *e, Status (*compare)(char *e1, char *e2));
Status   PriorElem(sqlist L, char *cur_e, char (*pre_e)[10]);
Status   NextElem(sqlist L, char *cur_e, char (*next_e)[10]);
Status   ListInsert(sqlist *L, int i, char *e);
Status   ListDelete(sqlist *L, int i, char (*e)[10]);
Status   ListTraverse(sqlist L, Status (*visit)(char *e));
```

```
double cur_e, double *pre_e);
main:
double d1=5.2, d2;
PriorElem(L,  d, &d2);

main:
char s1[10], s2[10];
PriorElem(L, s1, &s2);

问题2：若要求保持一致，如何做？
```

问：当类型是char *时，
需要做什么改变？

```
/* linear_list_sq.h 的组成 */
#define LIST_INIT_SIZE      100 //初始大小为100(可按需修改)
#define LISTINCREMENT       10  //空间分配增量(可按需修改)

typedef struct {
    char **elem;            //存放动态申请空间(当数组用)的首地址
    int length;             //记录当前长度
    int listsize;           //当前分配的元素的个数
} sqlist;

Status   InitList(sqlist *L);
Status   DestroyList(sqlist *L);
Status   ClearList(sqlist *L);
Status   ListEmpty(sqlist L);
int      ListLength(sqlist L);
Status   GetElem(sqlist L, int i, char **e);
int      LocateElem(sqlist L, char *e, Status (*compare)(char *e1, char *e2));
Status   PriorElem(sqlist L, char *cur_e, char **pre_e);
Status   NextElem(sqlist L, char *cur_e, char **next_e);
Status   ListInsert(sqlist *L, int i, char *e);
Status   ListDelete(sqlist *L, int i, char **e);
Status   ListTraverse(sqlist L, Status (*visit)(char *e));
```

问：当类型是struct student时，需要做什么改变？

注：纯C编译器，struct student e 不能写成 student e

```
/* linear_list_sq.h 的组成 */
#define LIST_INIT_SIZE      100 //初始大小为100(可按需修改)
#define LISTINCREMENT       10  //空间分配增量(可按需修改)
struct student {
    ...
};
typedef struct {
    struct student *elem; //存放动态申请空间(当数组用)的首地址
    int length;           //记录当前长度
    int listsize;         //当前分配的元素的个数
} sqlist;
Status   InitList(sqlist *L);
Status   DestroyList(sqlist *L);
Status   ClearList(sqlist *L);
Status   ListEmpty(sqlist L);
int      ListLength(sqlist L);
Status   GetElem(sqlist L, int i, struct student *e);
int      LocateElem(sqlist L, struct student e, Status (*compare)( struct student e1, struct student e2));
Status   PriorElem(sqlist L, struct student cur_e, struct student *pre_e);
Status   NextElem(sqlist L, struct student cur_e, struct student *next_e);
Status   ListInsert(sqlist *L, int i, struct student e);
Status   ListDelete(sqlist *L, int i, struct student *e);
Status   ListTraverse(sqlist L, Status (*visit)(struct student e));
```

问：当类型是struct student *时，需要做什么改变？

注：纯C编译器，struct student e 不能写成 student e

```
/* linear_list_sq.h 的组成 */
#define LIST_INIT_SIZE      100 //初始大小为100(可按需修改)
#define LISTINCREMENT       10  //空间分配增量(可按需修改)
struct student {
    ...
};
typedef struct {
    struct student **elem; //存放动态申请空间(当数组用)的首地址
    int length;            //记录当前长度
    int listsize;          //当前分配的元素的个数
} sqlist;
Status   InitList(sqlist *L);
Status   DestroyList(sqlist *L);
Status   ClearList(sqlist *L);
Status   ListEmpty(sqlist L);
int      ListLength(sqlist L);
Status   GetElem(sqlist L, int i, struct student **e);
int      LocateElem(sqlist L, struct student *e, Status (*compare)( struct student *e1, struct student *e2));
Status   PriorElem(sqlist L, struct student *cur_e, struct student **pre_e);
Status   NextElem(sqlist L, struct student *cur_e, struct student **next_e);
Status   ListInsert(sqlist *L, int i, struct student *e);
Status   ListDelete(sqlist *L, int i, struct student **e);
Status   ListTraverse(sqlist L, Status (*visit)(struct student *e));
```

```
/* linear_list_sq.h 的组成 */
#define LIST_INIT_SIZE      100 //初始大小为100(可按需修改)
#define LISTINCREMENT       10  //空间分配增量(可按需修改)


typedef struct {
    int *elem;     //存放动态申请空间(当数组用)的首地址
    int length;    //记录当前长度
    int listsize; //当前分配的元素的个数
} sqlist;
```

问：当类型是
int
double
char[]
char *
struct student
struct student *
时，能否使改动尽可能少？

答：在数据结构中一般不讨论具体类型，引入一个通用类型（Elemtype）来表示元素的类型即可

```
// - - - - - 线性表的动态分配顺序存储结构 - - - - -
#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;          // 存储空间基址
    int        length;          // 当前长度
    int        listsize;        // 当前分配的存储容量(以 sizeof(ElemType)为单位)
}SqList;
```

P. 22

```
/* linear_list_sq.h 的组成 */
#define LIST_INIT_SIZE      100 //初始大小为100(可按需修改)
#define LISTINCREMENT       10  //空间分配增量(可按需修改)

typedef int ElemType;       //算法到程序的补充

typedef struct {
    ElemType *elem;         //存放动态申请空间的首地址
    int length;             //记录当前长度
    int listsize; //当前分配的元素的个数
} sqlist;

Status   InitList(sqlist *L);
Status   DestroyList(sqlist *L);
Status   ClearList(sqlist *L);
Status   ListEmpty(sqlist L);
int      ListLength(sqlist L);
Status   GetElem(sqlist L, int i, ElemType *e);
int      LocateElem(sqlist L, ElemType e, Status (*compare)(ElemType e1, ElemType e2));
Status   PriorElem(sqlist L, ElemType cur_e, ElemType *pre_e);
Status   NextElem(sqlist L, ElemType cur_e, ElemType *next_e);
Status   ListInsert(sqlist *L, int i, ElemType e);
Status   ListDelete(sqlist *L, int i, ElemType *e);
Status   ListTraverse(sqlist L, Status (*visit)(ElemType e));
```

问：当类型不同时，能否使改动尽可能少？

答：在数据结构中一般不讨论具体类型，引入一个通用类型（Elemtype）来表示元素的类型即可

问：算法转为程序时，如何对应实际类型？

答：用typedef声明新类型的方法来实现实际类型和通用类型间的映射

/* linear_list_sq.h 的组成 */

```
struct student {
    int   num;      //设学号为主关键字
    char  name[10];
    char  sex;
    float score;
    char  addr[30];
}   //算上填充，共52字节
```

问：当类型是
int
double
char[]
char *
struct student
struct student *
时，需要做什么改变？

答：实际使用时，6选1即可(只能打开其中一项，否则错)，函数声明部分不同类型完全一致

```
//typedef int ElemType;
typedef double ElemType;
//typedef char ElemType[10];
//typedef char* ElemType;
//typedef struct student ElemType;
//typedef struct student* ElemType;
```

```
typedef struct {
    ElemType *elem;
    int length;
    int listsize;
} sqlist;
```

不同类型一致，无任何变化

```
Status      InitList(sqlist *L);
Status      DestroyList(sqlist *L);
Status      ClearList(sqlist *L);
Status      ListEmpty(sqlist L);
int         ListLength(sqlist L);
Status      GetElem(sqlist L, int i, ElemType *e);
int         LocateElem(sqlist L, ElemType e,
                       Status (*compare)(ElemType e1, ElemType e2));
Status      PriorElem(sqlist L, ElemType cur_e, ElemType *pre_e);
Status      NextElem(sqlist L, ElemType cur_e, ElemType *next_e);
Status      ListInsert(sqlist *L, int i, ElemType e);
Status      ListDelete(sqlist *L, int i, ElemType *e);
Status      ListTraverse(sqlist L, Status (*visit)(ElemType e));
```

§2.线性表

2.2.线性表的顺序表示和实现

2.2.2.线性表顺序表示的基本操作的实现

2.2.2.1.C语言版

★ 程序的组成

★ 算法与程序的区别

★ 与书上算法的区别

- C语言无引用，需要用指针代替
- 临时变量算法中无定义，程序要补齐
- 某些形式化定义和实际表示之间有区别

★ linear_list_sq.h 中各定义项的解析

★ linear_list_sq.c 中各函数的具体实现

ElemType => int

```
/* linear_list_sq.c 的实现 */
#include <stdio.h>
#include <stdlib.h>                    //malloc/realloc函数
#include <unistd.h>                    //exit函数
#include "linear_list_sq.h"            //形式定义

/* 初始化线性表 */
Status InitList(sqlist *L)
{
    L->elem = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(ElemType));
    if (L->elem == NULL)
        exit(OVERFLOW);
    L->length = 0;
    L->listsize = LIST_INIT_SIZE;
    return OK;
}
```

★ main函数中
声明为 sqlist L;
调用为 InitList(&L);

★ 形参为指针，因为函数中要改变并返回
★ 书上为引用，因此 L.elem形式应变为 L->elem形式

ElemType => int

```
/* linear_list_sq.c 的实现 */
#include <stdio.h>
#include <stdlib.h>                    //malloc/realloc函数
#include <unistd.h>                    //exit函数
#include "linear_list_sq.h"            //形式定义

/* 初始化线性表 */
Status InitList(sqlist L)
{
    L.elem = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(ElemType));
    if (L.elem == NULL)
        exit(OVERFLOW);
    L.length = 0;
    L.listsize = LIST_INIT_SIZE;
    return OK;
}
```

★ main函数中
声明为 sqlist L;
调用为 InitList(L);

★ 形参为sqlist结构体变量，函数实现中
L-> 均改为 L. 是否正确？为什么？

```
/* linear_list_sq.c 的实现 */
/* 初始化线性表 */
Status InitList(sqlist L)
{
    L.elem = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(ElemType));
    if (L.elem == NULL)
        exit(OVERFLOW);
    L.length = 0;
    L.listsize = LIST_INIT_SIZE;
    return OK;
}
```

错误分析

★ main函数中
声明为 sqlist L;
调用为 InitList(L);

Step1：实参传形参

实参 L(12字节)	2000 2011	值未定 ???
形参 L(12字节)	2100 2111	值未定 ???

Step2：形参申请空间

实参 L(12字节)	2000 2011	值未定 ???
形参 L(12字节)	2100 2111	3000

错误：函数返回后，实参L
得不到申请的空间首址

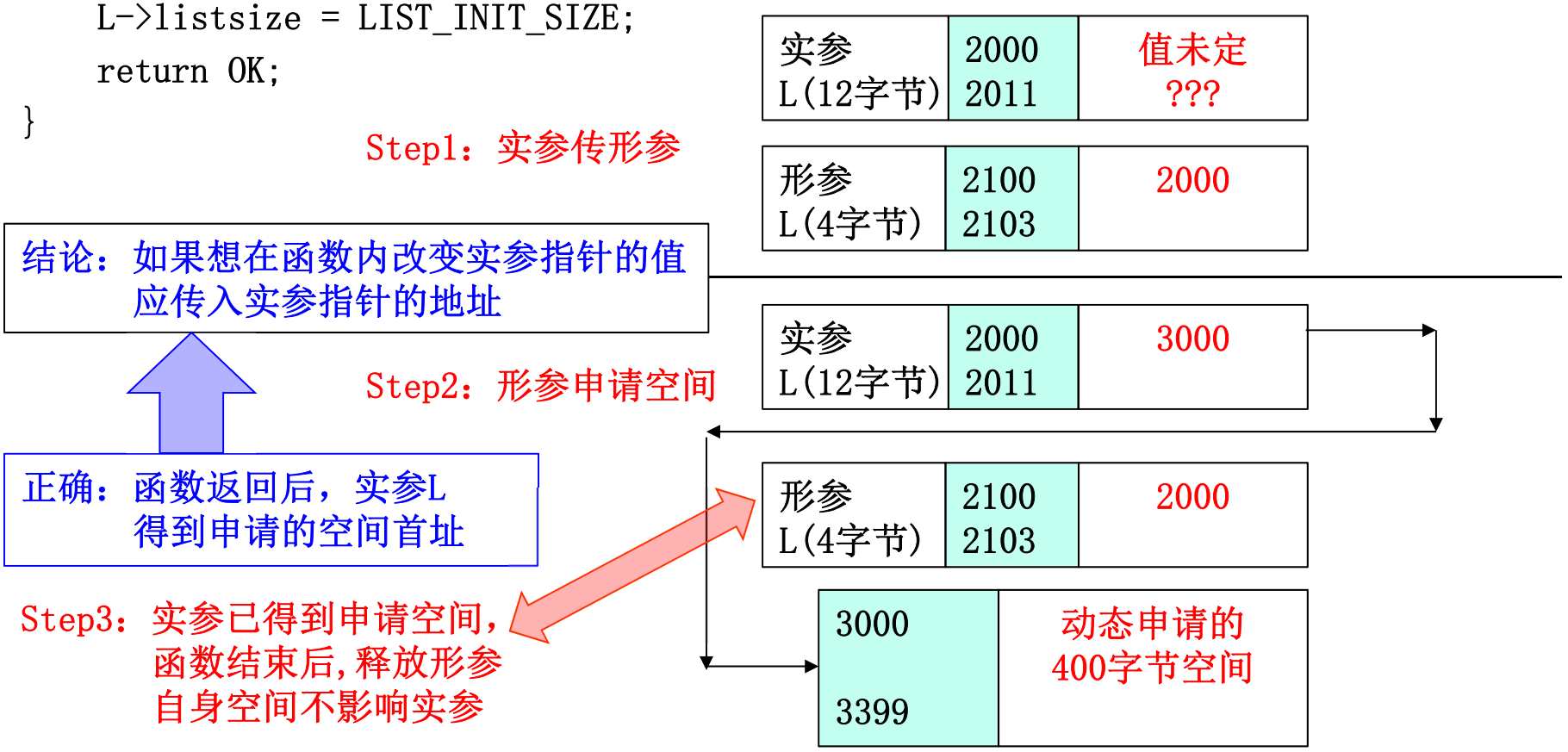
3000 3399	动态申请的 400字节空间

Step3：函数结束后，
释放形参自身空间，
实参并未得到申请空间

```
/* linear_list_sq.c 的实现 */
/* 初始化线性表 */
Status InitList(sqlist *L)
{
    L->elem = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(ElemType));
    if (L->elem == NULL)
        exit(OVERFLOW);
    L->length = 0;
    L->listsize = LIST_INIT_SIZE;
    return OK;
}
```

正确分析

★ main函数中
声明为 sqlist L;
调用为 InitList(&L);



```
/* linear_list_sq.c 的实现 */
#include <stdio.h>
#include <stdlib.h>                    //malloc/realloc函数
#include <unistd.h>                    //exit函数
#include "linear_list_sq.h"            //形式定义

/* 初始化线性表 */
Status InitList(sqlist *L)
{
    L->elem = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(ElemType));
    if (L->elem == NULL)
        exit(OVERFLOW);
    L->length = 0;
    L->listsize = LIST_INIT_SIZE;
    return OK;
}
```

★ main函数中
声明为 sqlist L;
调用为 InitList(&L);

★ 形参为指针，因为函数中要改变并返回
★ 书上为引用，因此 L.elem形式应变为 L->elem形式

问：当类型是
double
char[]
char *
struct student
struct student *
时，需要做什么改变？

答：不需要任何变化!!!

ElemType => int

```
/* linear_list_sq.c 的实现 */

/* 删除线性表 */
Status DestroyList(sqlist *L)
{
    /* 未执行 InitList，直接执行本函数，
       则可能出错，因为指针初值未定 */
    if (L->elem)
        free(L->elem);
    L->length   = 0; //可以不要
    L->listsize = 0; //可以不要

    return OK;
}
```

问：当类型是
double
char[]
char *
struct student
struct student *
时，需要做什么改变？

/* linear_list_sq.c 的实现 */

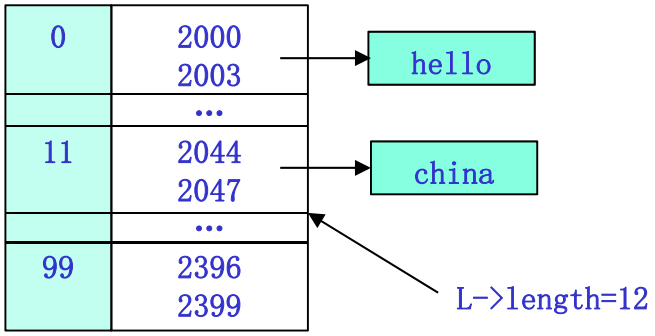
```
/* 删除线性表 */
Status DestroyList(sqlist *L)
{



    /* 未执行 InitList，直接执行本函数，
       则可能出错，因为指针初值未定 */
    if (L->elem)
            free(L->elem);
    L->length   = 0; //可以不要
    L->listsize = 0; //可以不要

    return OK;
}
```

```
/* 删除线性表 */
Status DestroyList(sqlist *L)
{
    int i;
    /* 首先释放二次申请空间 */
    for(i=0; i<L->length; i++)
        free(L->elem[i]);

    /* 未执行 InitList，直接执行本函数，
       则可能出错，因为指针初值未定 */
    if (L->elem)
            free(L->elem);
    L->length   = 0; //可以不要
    L->listsize = 0; //可以不要

    return OK;
}
```

问：当类型是
double
char[]
char *
struct student
struct student *
时，需要做什么改变？

答：当类型是
double
char[]
struct student时，
不需要任何变化!!!

答：当类型是
char *
struct student *时，
要首先释放二次申请空间



/* linear_list_sq.c 的实现 */

```
/* 删除线性表 */
Status DestroyList(sqlist *L)
{




    /* 未执行 InitList，直接执行本函数，
       则可能出错，因为指针初值未定 */
    if (L->elem)
            free(L->elem);
    L->length   = 0; //可以不要
    L->listsize = 0; //可以不要

    return OK;
}
```

```
/* 删除线性表 */
Status DestroyList(sqlist *L)
{
    int i;
    /* 首先释放二次申请空间 */
    for(i=0; i<L->length; i++)
        free(L->elem[i]);

    /* 未执行 InitList，直接执行本函数，
       则可能出错，因为指针初值未定 */
    if (L->elem)
            free(L->elem);
    L->length   = 0; //可以不要
    L->listsize = 0; //可以不要

    return OK;
}
```

类型为 char * 和 struct student *时，此处打开，其它类型时注释掉

问：当类型是
double
char[]
char *
struct student
struct student *
时，需要做什么改变？

问：如何处理具体类型不同时的代码差异？
答：按需注释/非注释

续问：有没有更好的方法？
续答：编译预处理 - 条件编译

请先去查看"条件编译"部分的课件

答：当类型是
double
char[]
struct student时，
不需要任何变化!!!

答：当类型是
char *
struct student *时，
要首先释放二次申请空间

§2.线性表

2.2.线性表的顺序表示和实现

2.2.2.线性表顺序表示的基本操作的实现

2.2.2.1.C语言版

★ linear_list_sq.h 中各定义项的解析

★ linear_list_sq.c 中各函数的具体实现

（前面全部废弃，用条件编译的方法完整实现六合一程序）

六合一

```
/* linear_list_sq.h 的组成 */
//#define ELEMTYPE_IS_INT           //不定义也行
//#define ELEMTYPE_IS_DOUBLE
//#define ELEMTYPE_IS_CHAR_ARRAY
//#define ELEMTYPE_IS_CHAR_P
//#define ELEMTYPE_IS_STRUCT_STUDENT
//#define ELEMTYPE_IS_STRUCT_STUDENT_P
```

定义6个宏定义，
目前全部disable，
使用时按需enable即可
每次只能enable一个!!!

```
/* P.10 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 LOVERFLOW        -2  //因为<math.h>中已有 OVERFLOW ，因此换一下

typedef int Status;
```

六合一

```
/* linear_list_sq.h 的组成 */
#define LIST_INIT_SIZE   100   //初始大小定义为100（可按需修改）
#define LISTINCREMENT    10    //若空间不够，每次增长10（可按需修改）

#ifdef ELEMTYPE_IS_DOUBLE
    typedef double ElemType;
#elif defined (ELEMTYPE_IS_CHAR_ARRAY)
    typedef char ElemType[10];
#elif defined (ELEMTYPE_IS_CHAR_P)
    typedef char* ElemType;
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    typedef struct student {
            int   num;
            char  name[10];
            char  sex;
            float score;
            char  addr[30];
    } ElemType;
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    typedef struct student {
            int   num;
            char  name[10];
            char  sex;
            float score;
            char  addr[30];
    } ET, *ElemType;  //此处为什么多一个ET类型的声明？后面讲
#else       //缺省当做int处理
    typedef int ElemType;
#endif
```

根据不同的宏定义决定ElemType的实际类型

六合一

```
/* linear_list_sq.h 的组成 */
#define LIST_INIT_SIZE   100   //初始大小定义为100（可按需修改）
#define LISTINCREMENT    10    //若空间不够，每次增长10（可按需修改）

#ifdef ELEMTYPE_IS_DOUBLE
    ......
#endif

typedef struct {
    ElemType *elem;      //存放动态申请空间的首地址
    int length;          //记录当前长度
    int listsize;        //当前分配的元素的个数
} sqlist;

Status      InitList(sqlist *L);
Status      DestroyList(sqlist *L);
Status      ClearList(sqlist *L);
Status      ListEmpty(sqlist L);
int         ListLength(sqlist L);
Status      GetElem(sqlist L, int i, ElemType *e);
int         LocateElem(sqlist L, ElemType e, Status (*compare)(ElemType e1, ElemType e2));
Status      PriorElem(sqlist L, ElemType cur_e, ElemType *pre_e);
Status      NextElem(sqlist L, ElemType cur_e, ElemType *next_e);
Status      ListInsert(sqlist *L, int i, ElemType e);
Status      ListDelete(sqlist *L, int i, ElemType *e);
Status      ListTraverse(sqlist L, Status (*visit)(ElemType e));
```

不同类型一致

不同类型一致

★ 引用都表示为指针

★ 每个形参都要有类型定义，其中compare和visit区别较大

六合一

```
/* linear_list_sq.c 的实现 */
#include <stdio.h>
#include <stdlib.h>          //malloc/realloc函数
#include <unistd.h>          //exit函数
#include <math.h>            //fabs函数
#include <string.h>          //strcpy/strcmp等函数
#include "linear_list_sq.h"  //形式定义

/* 初始化线性表 */
Status InitList(sqlist *L)
{
    L->elem = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(ElemType));
    if (L->elem == NULL)
        exit(OVERFLOW);
    L->length = 0;
    L->listsize = LIST_INIT_SIZE;
    return OK;
}
```

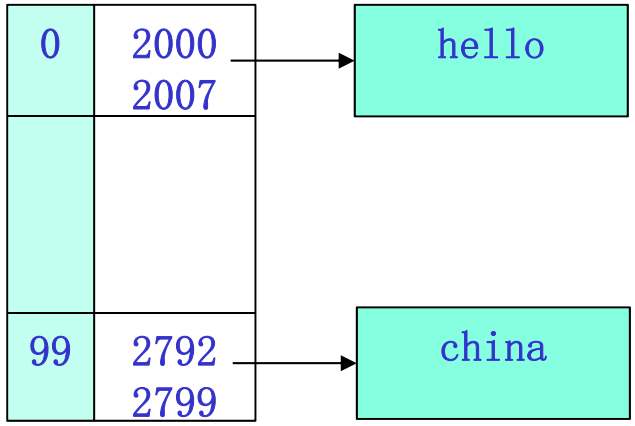
把各种数据类型
需要的库函数
一起包含进来

所有数据类型的
处理方法都相同
无变化

六合一

```
/* linear_list_sq.c 的实现 */

/* 删除线性表 */
Status DestroyList(sqlist *L)
{
   /* 两种指针类型需要释放二级空间 */
#if defined (ELEMTYPE_IS_CHAR_P) || defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    int i;

    /* 首先释放二级空间 */
    for(i=0; i<L->length; i++)
        free(L->elem[i]);
#endif

    /* 若未执行 InitList，直接执行本函数，则可能出错 */
    if (L->elem)
        free(L->elem);
    L->length   = 0; //可不要
    L->listsize = 0; //可不要
    return OK;
}
```

两种数据类型的
特殊处理方法
其它四种无



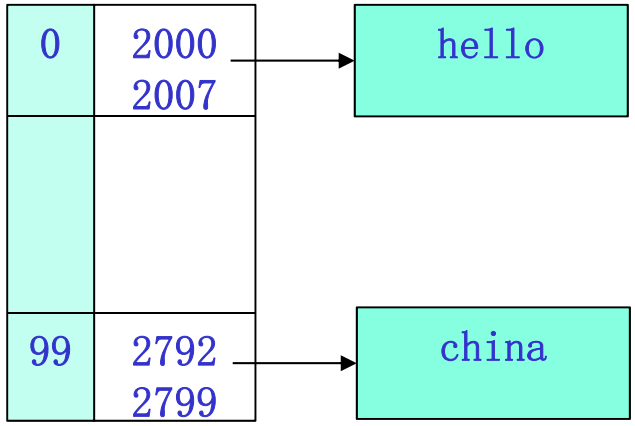
六种数据类型的
公共部分处理部分

/* linear_list_sq.c 的实现 */

六合一

```
/* 清除线性表（已初始化，不释放空间，只清除内容） */
Status ClearList(sqlist *L)
{
   /* 两种指针类型需要释放二级空间 */
#if defined (ELEMTYPE_IS_CHAR_P) || defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    int i;

    /* 首先释放二级空间 */
    for(i=0; i<L->length; i++)
        free(L->elem[i]);
#endif

    L->length = 0;
    return OK;
}
```

两种数据类型的
特殊处理方法
其它四种无



六种数据类型的
公共部分处理部分

/* linear_list_sq.c 的实现 */

六合一

```
/* 判断是否为空表 */
Status ListEmpty(sqlist L)
{
    if (L.length == 0)
        return TRUE;
    else
        return FALSE;
}
```

所有数据类型的处理方法都相同无变化

/* linear_list_sq.c 的实现 */

六合一

```
/* 求表的长度 */
int ListLength(sqlist L)
{
    return L.length;
}
```

所有数据类型的
处理方法都相同
无变化

六合一

```
/* linear_list_sq.c 的实现 */

/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e)
{
    /* i值合理范围[1..length]  */
    if (i<1 || i>L.length)
        return ERROR;

    e <= L.elem[i-1];

    return OK;
}
```

具体讨论不同数据类型的不同处理方法

```
/* linear_list_sq.c 的实现 */
```

六合一： ElemType是int/double

```
/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e)
{
    /* i值合理范围[1..length]  */
    if (i<1 || i>L.length)
        return ERROR;

    *e = L.elem[i-1];       //下标从0开始，第i个实际在elem[i-1]中
    return OK;
}
```

/* linear_list_sq.c 的实现 */

六合一： ElemType是char[]

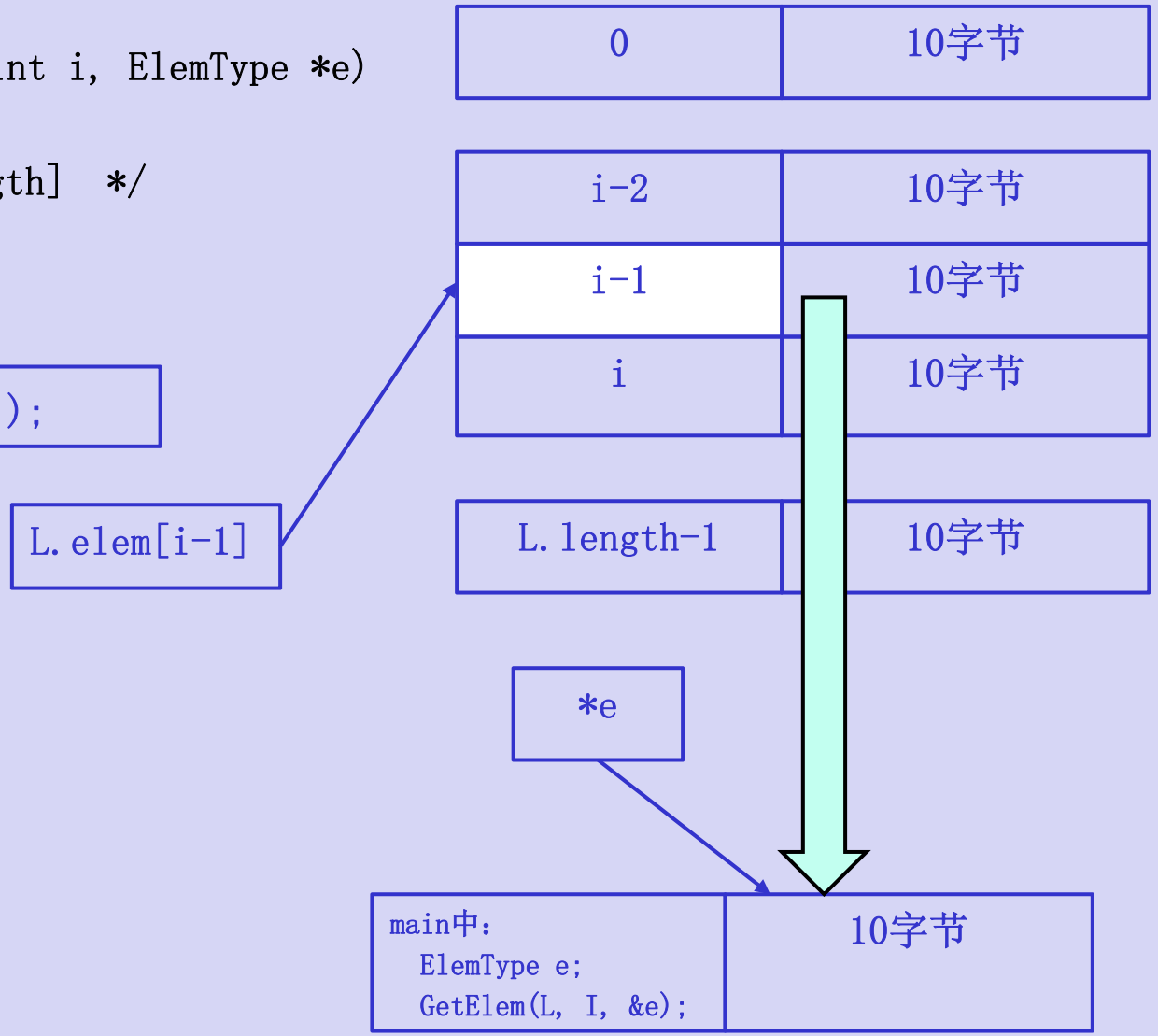
```
/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e)
{
    /* i值合理范围[1..length]  */
    if (i<1 || i>L.length)
        return ERROR;

    strcpy(*e, L.elem[i-1]);
    return OK;
}
```

因为：typedef char Elemtype[10];
所以：Elemtype e => char e[10];
Elemtype *e => char (*e)[10];
e是指向有10个字符组成的一维字符数组的指针
*e指向一维字符数组的首字符的指针

```
main中：
  ElemType e;
  GetElem(L, i, &e);
```

```
/* linear_list_sq.c 的实现 */

/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e)
{
    /* i值合理范围[1..length]  */
    if (i<1 || i>L.length)
        return ERROR;

    strcpy(*e, L.elem[i-1]);
    return OK;
}
```

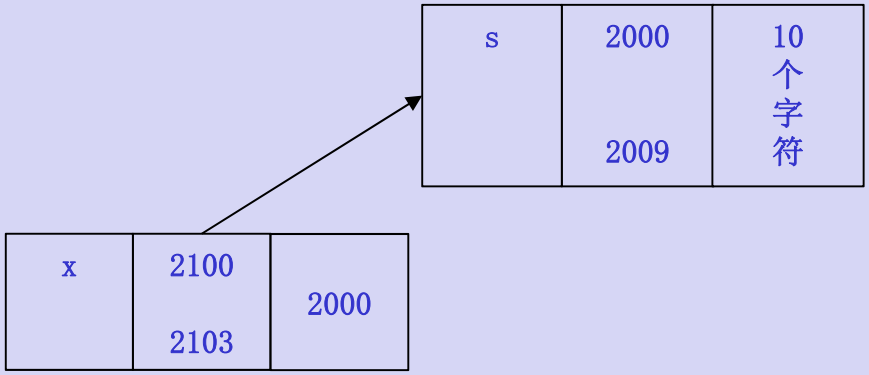
六合一： ElemType是char[]



```c
#include <stdio.h>
#include <string.h>

void f(char x[10])
{
    printf("%d %d\n", sizeof(x), sizeof(*x));
    strcpy(x, "hello");
}
int main()
{
    char s[10];
    f(s);
    printf("%s\n", s);

    return 0;
}
```

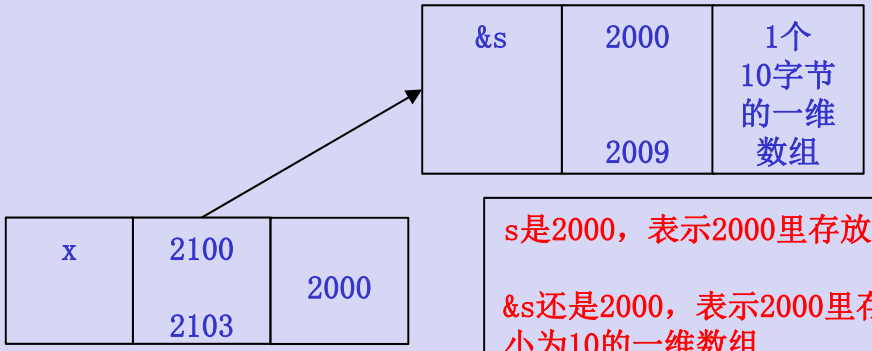
char x[10]
char *x
char x[任意数字]

4 1




```c
#include <stdio.h>
#include <string.h>

void f(char (*x)[10])
{
    printf("%d %d\n", sizeof(x), sizeof(*x));
    strcpy(*x, "hello");
}
int main()
{
    char s[10];
    f(&s);
    printf("%s\n", s);

    return 0;
}
```

4 10



s是2000，表示2000里存放一个字符

&s还是2000，表示2000里存放一个大小为10的一维数组

/* linear_list_sq.c 的实现 */

六合一： ElemType是char[]

```
/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e)
{
    /* i值合理范围[1..length]  */
    if (i<1 || i>L.length)
        return ERROR;

    strcpy(*e, L.elem[i-1]);
    return OK;
}
```

因为：typedef char Elemtype[10];
所以：Elemtype e => char e[10];
Elemtype *e => char (*e)[10];
e是指向有10个字符组成的一维字符数组的指针
*e指向一维字符数组的首字符的指针

main中：
ElemType e;
GetElem(L, i, &e);

```
/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType e)
{
    /* i值合理范围[1..length]  */
    if (i<1 || i>L.length)
        return ERROR;

    strcpy(e, L.elem[i-1]);
    return OK;
}
```

main中：
ElemType e;
GetElem(L, i, e);

如果形参写成ElemType e，也是可以的

但为了保持不同类型下GetElem的声明统一性，
仍使用ElemType *e

```
/* linear_list_sq.c 的实现 */

/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e)
{
    /* i值合理范围[1..length]  */
    if (i<1 || i>L.length)
        return ERROR;

    strcpy(*e, L.elem[i-1]);
    return OK;
}
```

六合一： ElemType是char *

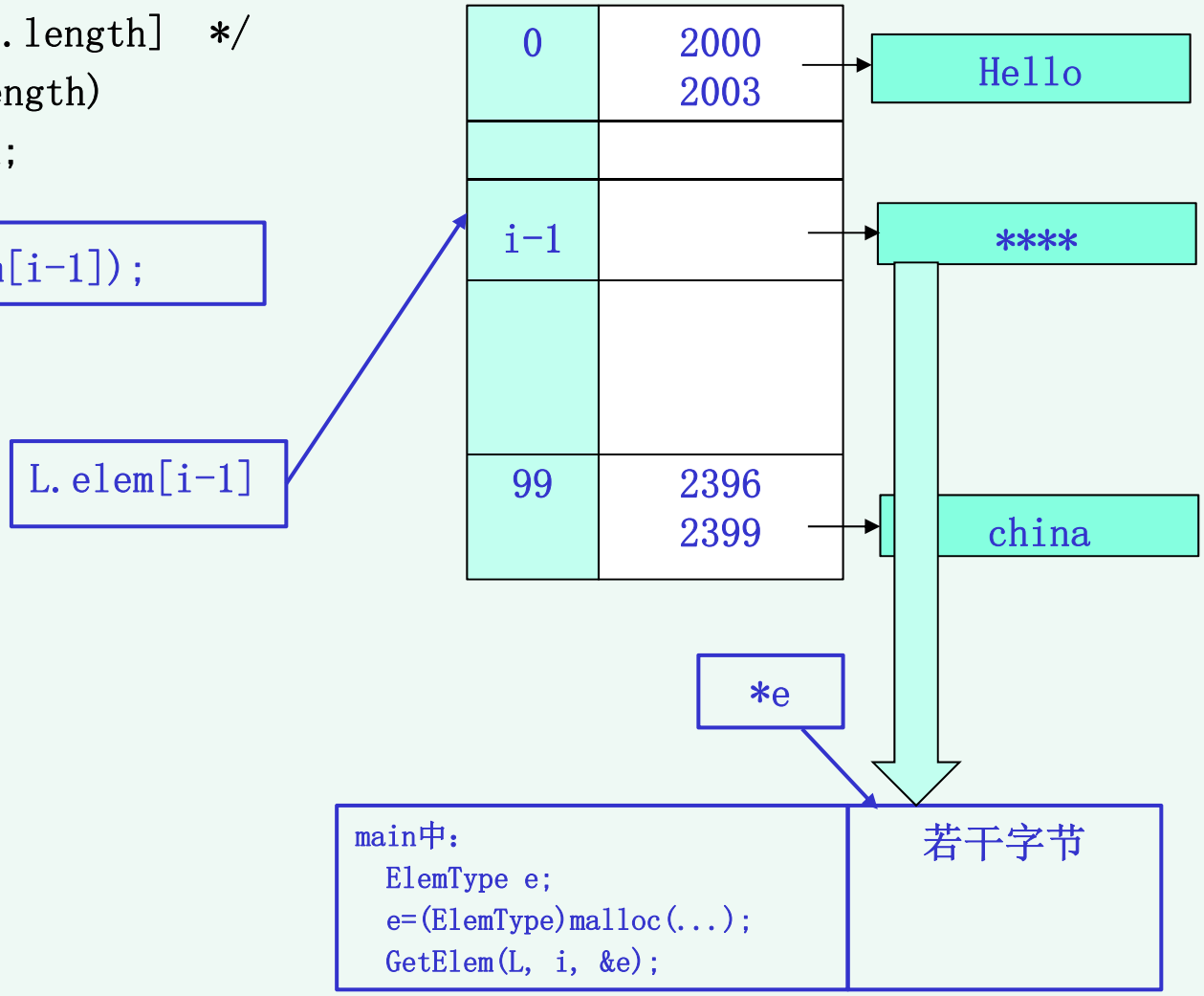
因为：typedef char* Elemtype;
所以：Elemtype e => char *e;
Elemtype *e => char **e;
e是基类型为char *的指针
*e是基类型为char的指针

```
main中：
  ElemType e;
  e=(ElemType)malloc(...);
  GetElem(L, i, &e);
```

/* linear_list_sq.c 的实现 */

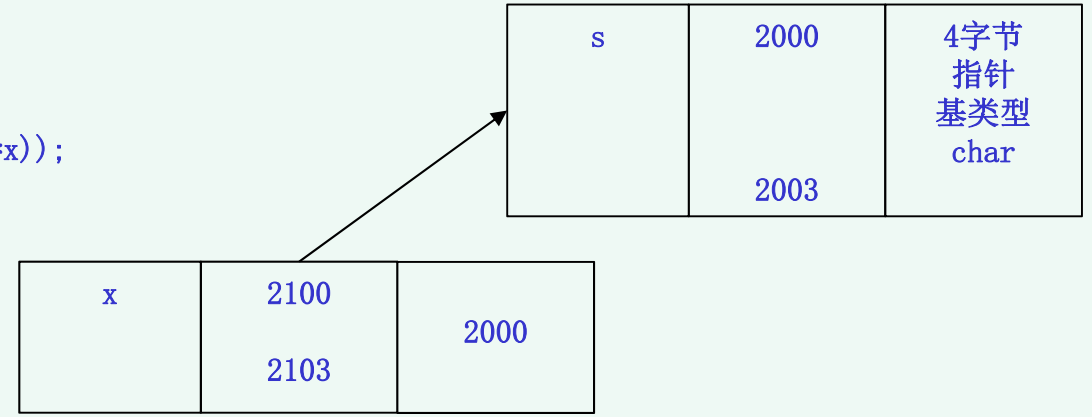
六合一： ElemType是char *

```
/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e)
{
    /* i值合理范围[1..length]  */
    if (i<1 || i>L.length)
        return ERROR;

    strcpy(*e, L.elem[i-1]);
    return OK;
}
```



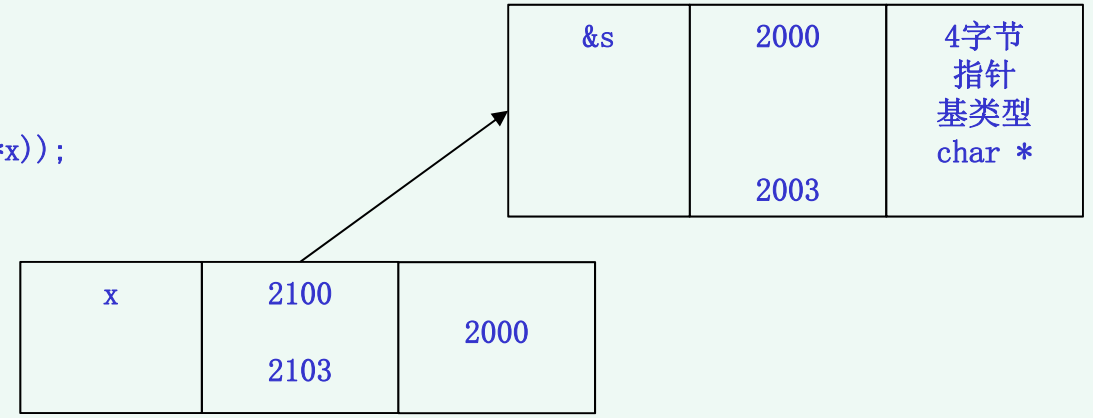
```c
#include <stdio.h>
#include <string.h>

void f(char *x)
{
    printf("%d %d\n", sizeof(x), sizeof(*x));
    return;
}

int main()
{
    char *s;
    f(s);

    return 0;
}
```

4 1




```c
#include <stdio.h>
#include <string.h>

void f(char **x)
{
    printf("%d %d\n", sizeof(x), sizeof(*x));
    return;
}

int main()
{
    char *s;
    f(&s);

    return 0;
}
```

4 4



s是2000，表示2000里存放一个指针

&s还是2000，表示2000里存放一个指针的指针

/* linear_list_sq.c 的实现 */

六合一： ElemType是char *

```
/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e)
{
    /* i值合理范围[1..length]  */
    if (i<1 || i>L.length)
        return ERROR;

    strcpy(*e, L.elem[i-1]);
    return OK;
}
```

因为：typedef char* Elemtype;
所以：Elemtype e => char *e;
Elemtype *e => char **e;
e是基类型为char *的指针
*e是基类型为char的指针

main中：

```
ElemType e;
e=(ElemType)malloc(...);
GetElem(L, i, &e);
```

```
/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType e)
{
    /* i值合理范围[1..length]  */
    if (i<1 || i>L.length)
        return ERROR;

    strcpy(e, L.elem[i-1]);
    return OK;
}
```

main中：

```
ElemType e;
e=(ElemType)malloc(...);
GetElem(L, i, e);
```

如果形参写成ElemType e，也是可以的

但为了保持不同类型下GetElem的声明统一性，仍使用ElemType *e

```
/* linear_list_sq.c 的实现 */

/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e)
{
    /* i值合理范围[1..length]  */
    if (i<1 || i>L.length)
        return ERROR;

    memcpy(e, &(L.elem[i-1]), sizeof(ElemType));
    return OK;
}
```

六合一： ElemType是struct student
假设占用52字节

main中：
```
ElemType e
GetElem(L, i, &e);
```

memcpy函数的使用：
```
void *memcpy(void *dest, const void *src, int n);
```
将从源地址开始的n个字节复制到目标地址中
★ 整体内存拷贝，不论中间是否有尾零

内存理解同char型数组
但无法保证尾零，因此
不能用strcpy

```
/* linear_list_sq.c 的实现 */

/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e)
{
    /* i值合理范围[1..length]  */
    if (i<1 || i>L.length)
        return ERROR;

    memcpy(*e, L.elem[i-1], sizeof(ET));
    return OK;
}

    memcpy(e, &(L.elem[i-1]), sizeof(ElemType));
```

六合一： ElemType是struct student *

内存理解同char型指针
但无法保证尾零，因此
不能用strcpy

```
main中：
  ElemType e;
  e=(ElemType)malloc(sizeof(ET));
  GetElem(L, i, &e);
```

```
main中：
  ElemType e
  GetElem(L, i, &e);
```

```
typedef struct student {
    int   num;
    char  name[10];
    char  sex;
    float score;
    char  addr[30];
} ET, *ElemType; //此处为什么多个ET类型的声明？后面讲
```

六合一

```
/* linear_list_sq.c 的实现 */
/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e)
{
    if (i<1 || i>L.length)
        return ERROR;

#if defined (ELEMTYPE_IS_CHAR_ARRAY) || defined (ELEMTYPE_IS_CHAR_P)
    strcpy(*e, L.elem[i-1]);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(e, &(L.elem[i-1]), sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    memcpy(*e, L.elem[i-1], sizeof(ET));
#else    //int和double直接赋值
    *e = L.elem[i-1];
#endif

    return OK;
}
```

不同数据类型的
不同处理方法

```
/* linear_list_sq.c 的实现 */

/* 查找符合指定条件的元素（返回值等于e的元素在线性表中的位序） */
int LocateElem(sqlist L, ElemType e)
{
    ElemType *p = L.elem;
    int       i = 1;
    while(i<=L.length && (*p 和 e不相等) ) { 宏定义方式实现
        i++;
        p++;
        }

    return (i<=L.length) ? i : 0;   //找到返回i，否则返回0
}
```

```
*p != e
fabs(*p, e)<1e-6
strcmp(*p, e)!=0
strcmp(*p, e)!=0
p->num != e.num
(*p)->num != e->num
```

/* linear_list_sq.c 的实现 */

书上程序的思路：
将不同比较方法放在main中，写成形式相同，内容不同的比较函数，传进函数指针

六合一

/* 查找符合指定条件的元素 */

```
int LocateElem(sqlist L, ElemType e, Status (*compare)(ElemType e1, ElemType e2))
{
    ElemType *p = L.elem;
    int       i = 1;

    while(i<=L.length && (*compare)(*p++, e)==FALSE)
        i++;

    return (i<=L.length) ? i : 0;   //找到返回i，否则返回0
}
```

所有数据类型的处理方法都相同无变化

思考：DestroyList
ClearList
GetElem
如何改，使函数中不出现宏定义而定义出现在main中？

注：只是训练代码能力而已，
实际上，放函数中更合理!!!

说明：到目前为止对不同数据类型的条件编译有两种方法
1、GetElem形式，宏定义放在函数中，main中无
2、LocateElem形式，宏定义放在main中，函数中无

为什么这么做？请透彻理解!!!
1、操作与用户有关的，应该放main
（例：判断是否相等，不同数据类型的条件可能不同，
甚至相同数据类型的判断条件也可能不同，
以struct student为例，有时候，仅学号相等
有时候，要学号姓名相等）
2、操作与用户无关的，应该放函数
（例：进行内存复制，和字符串长度/结构体的成员多少无关，都是统一标准和方法）

```
/* main中用于比较两个值是否相等的具体函数，与LocateElem中的函数
   指针定义相同，调用时传入 */

Status MyCompare(ElemType e1, ElemType e2)
{
#ifdef ELEMTYPE_IS_DOUBLE
    if (fabs(e1-e2)<1e-6)
#elif defined (ELEMTYPE_IS_CHAR_ARRAY) || defined (ELEMTYPE_IS_CHAR_P)
    if (strcmp(e1, e2)==0)
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    if (e1.num==e2.num)
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    if (e1->num==e2->num)
#else           //缺省当做int处理
    if (e1==e2)
#endif
        return TRUE;
    else
        return FALSE;
}
```

main中：
LocateElem(L, e, MyCompare);

六合一

```
/* linear_list_sq.c 的实现 */
/* 查找符合指定条件的元素的前驱元素 */
Status PriorElem(sqlist L, ElemType cur_e, ElemType *pre_e)
{   ElemType *p = L.elem;
    int       i = 1;

#ifdef ELEMTYPE_IS_DOUBLE
    while(i<=L.length && fabs(*p-cur_e)>=1e-6) {
#elif defined (ELEMTYPE_IS_CHAR_ARRAY) || defined (ELEMTYPE_IS_CHAR_P)
    while(i<=L.length && strcmp(*p, cur_e)) {
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    while(i<=L.length && p->num!=cur_e.num) {
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    while(i<=L.length && (*p)->num!=cur_e->num) {
#else         //缺省当做int处理
    while(i<=L.length && *p!=cur_e) {
#endif
        i++;
        p++;
        }

    if (i==1 || i>L.length) //找到第1个元素或未找到
        return ERROR;
#if defined (ELEMTYPE_IS_CHAR_ARRAY) || defined(ELEMTYPE_IS_CHAR_P)
    strcpy(*pre_e, *--p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(pre_e, --p, sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    memcpy(*pre_e, *--p, sizeof(ET));
#else         //int和double直接赋值
    *pre_e = *--p;
#endif
    return OK;
}
```

不同数据类型的
不同处理方法

思考：如果要求 PriorElem 函数中不出现
宏定义，要加几个函数指针？
注：同样是代码能力训练，实际应用中：
比较(上)相等的条件编译放main更合理
赋值(下)的条件编译放函数中更合理

不同数据类型的
不同处理方法

六合一

```
/* linear_list_sq.c 的实现 */
/* 查找符合指定条件的元素的后继元素 */
Status NextElem(sqlist L, ElemType cur_e, ElemType *next_e)
{   ElemType *p = L.elem;
    int       i = 1;

#ifdef ELEMTYPE_IS_DOUBLE
    while(i<L.length && fabs(*p-cur_e)>=1e-6) {
#elif defined (ELEMTYPE_IS_CHAR_ARRAY) || defined (ELEMTYPE_IS_CHAR_P)
    while(i<L.length && strcmp(*p, cur_e)) {
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    while(i<L.length && p->num!=cur_e.num) {
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    while(i<L.length && (*p)->num!=cur_e->num) {
#else         //缺省当做int处理
    while(i<L.length && *p!=cur_e) {
#endif
        i++;
        p++;
        }

    if (i>=L.length)        //未找到（最后一个元素未比较）
        return ERROR;
#if defined (ELEMTYPE_IS_CHAR_ARRAY) || defined (ELEMTYPE_IS_CHAR_P)
    strcpy(*next_e, *++p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(next_e, ++p, sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    memcpy(*next_e, *++p, sizeof(ET));
#else         //int和double直接赋值
    *next_e = *++p;
#endif
    return OK;
}
```

不同数据类型的
不同处理方法

思考：如果要求 NextElem 函数中不出现
宏定义，要加几个函数指针？
注：同样是代码能力训练，实际应用中：
比较(上)相等的条件编译放main更合理
赋值(下)的条件编译放函数中更合理

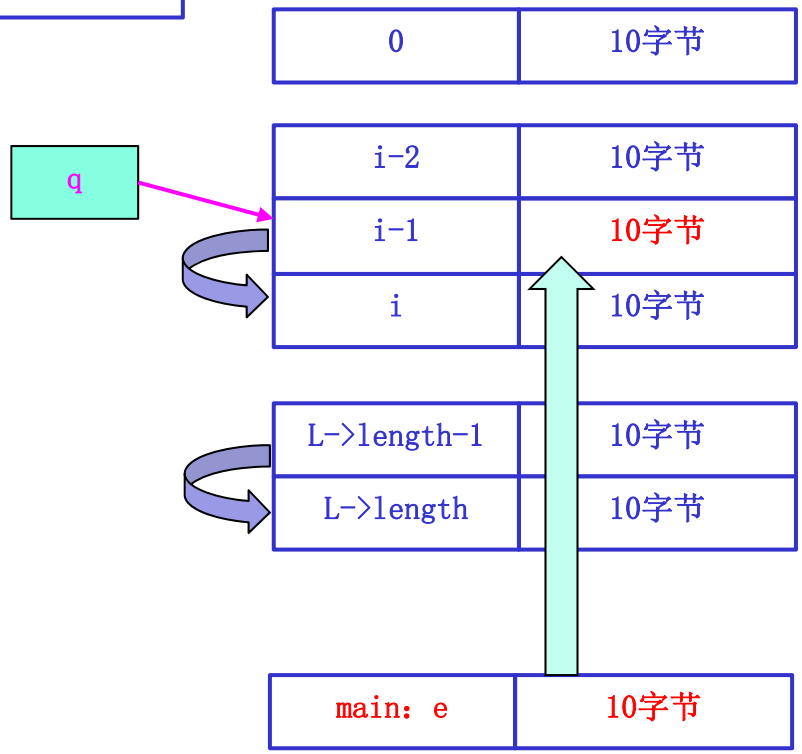
不同数据类型的
不同处理方法

```
/* linear_list_sq.c 的实现 */
/* 在指定位置前插入一个新元素 */
Status ListInsert(sqlist *L, int i, ElemType e)
{
    ElemType *p, *q; //如果是算法，一般可以省略，程序不能

    if (i<1 || i>L->length+1)    //合理范围是 1..length+1
        return ERROR;

    /* 空间已满则扩大空间 */
    if (L->length >= L->listsize) {
        ElemType *newbase;
        newbase = (ElemType *)realloc(L->elem,
                  (L->listsize+LISTINCREMENT)*sizeof(ElemType));
        if (!newbase)
            return OVERFLOW;

        L->elem = newbase;
        L->listsize += LISTINCREMENT;
        }

    q = &(L->elem[i-1]);  //第i个元素，即新的插入位置

    /* 从最后一个开始到第i个元素依次后移一格 */
    for (p=&(L->elem[L->length-1]); p>=q; --p)
#if defined (ELEMTYPE_IS_CHAR_ARRAY)
        strcpy(*(p+1), *p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
        memcpy(p+1, p, sizeof(ElemType));
#else  //int、double、char指针、struct student指针都是直接赋值
        *(p+1) = *p;
#endif
    /* 插入新元素，长度+1 */
#if defined (ELEMTYPE_IS_CHAR_ARRAY)
    strcpy(*q, e);
#elif defined (ELEMTYPE_IS_CHAR_P)
    /* 原L->elem[i-1]的指针已放入[i]中，
           重新申请空间，插入新元素，长度+1 */
    L->elem[i-1] = (ElemType)malloc
                ((strlen(e)+1) * sizeof(char));
    if (L->elem[i-1]==NULL)
        return LOVERFLOW;

    strcpy(*q, e);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(q, &e, sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    L->elem[i-1] = (ElemType)malloc(sizeof(ET));
    if (L->elem[i-1]==NULL)
        return LOVERFLOW;

    memcpy(*q, e, sizeof(ET));
#else    //int和double直接赋值
    *q = e;
#endif

    L->length ++;
    return OK;
}
```

六合一

不同数据类型的
不同处理方法

char []
struct student



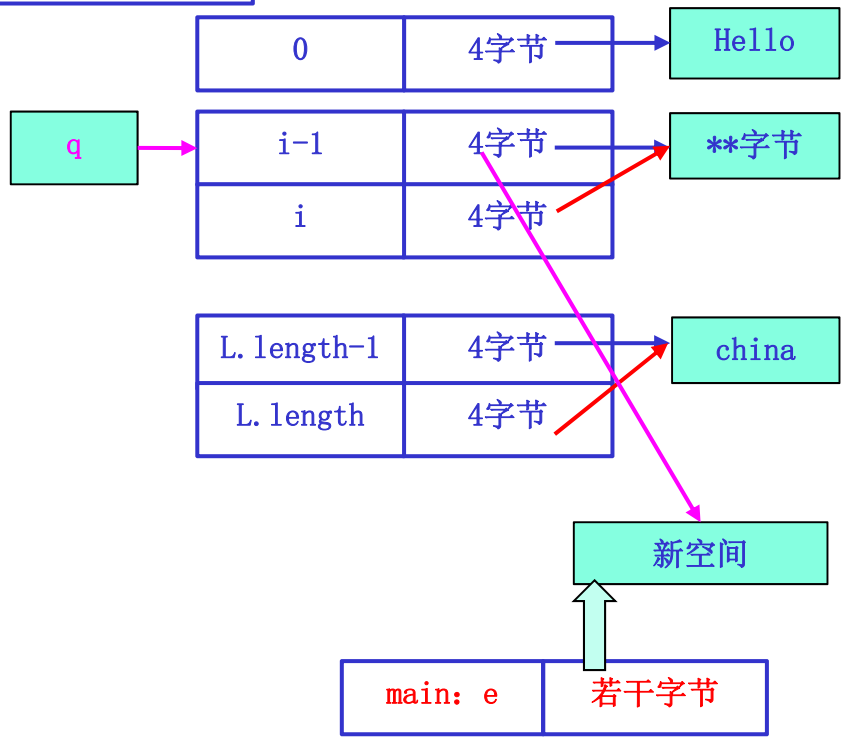

```
    q = &(L->elem[i-1]);  //第i个元素，即新的插入位置

    /* 从最后一个开始到第i个元素依次后移一格 */
    for (p=&(L->elem[L->length-1]); p>=q; --p)
#if defined (ELEMTYPE_IS_CHAR_ARRAY)
        strcpy(*(p+1), *p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
        memcpy(p+1, p, sizeof(ElemType));
#else  //int、double、char指针、struct student指针都是直接赋值
        *(p+1) = *p;
#endif
```

六合一

```
    /* 插入新元素，长度+1 */
#if defined (ELEMTYPE_IS_CHAR_ARRAY)
    strcpy(*q, e);
#elif defined (ELEMTYPE_IS_CHAR_P)
    /* 原L->elem[i-1]的指针已放入[i]中，
           重新申请空间，插入新元素，长度+1 */
    L->elem[i-1] = (ElemType)malloc
                ((strlen(e)+1) * sizeof(char));
    if (L->elem[i-1]==NULL)
        return LOVERFLOW;

    strcpy(*q, e);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(q, &e, sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    L->elem[i-1] = (ElemType)malloc(sizeof(ET));
    if (L->elem[i-1]==NULL)
        return LOVERFLOW;

    memcpy(*q, e, sizeof(ET));
#else    //int和double直接赋值
    *q = e;
#endif

    L->length ++;
    return OK;
}
```

不同数据类型的
不同处理方法

char *
struct student *




```
    q = &(L->elem[i-1]);  //第i个元素，即新的插入位置

    /* 从最后一个开始到第i个元素依次后移一格 */
    for (p=&(L->elem[L->length-1]); p>=q; --p)
#if defined (ELEMTYPE_IS_CHAR_ARRAY)
        strcpy(*(p+1), *p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
        memcpy(p+1, p, sizeof(ElemType));
#else  //int、double、char指针、struct student指针都是直接赋值
      *(p+1) = *p;
#endif
```

六合一

```
    /* 插入新元素，长度+1 */
#if defined (ELEMTYPE_IS_CHAR_ARRAY)
    strcpy(*q, e);
#elif defined (ELEMTYPE_IS_CHAR_P)
    /* 原L->elem[i-1]的指针已放入[i]中，
          重新申请空间，插入新元素，长度+1 */
    L->elem[i-1] = (ElemType)malloc
                ((strlen(e)+1) * sizeof(char));
    if (L->elem[i-1]==NULL)
        return LOVERFLOW;

    strcpy(*q, e);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(q, &e, sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    L->elem[i-1] = (ElemType)malloc(sizeof(ET));
    if (L->elem[i-1]==NULL)
        return LOVERFLOW;

    memcpy(*q, e, sizeof(ET));
#else    //int和double直接赋值
    *q = e;
#endif

    L->length ++;
    return OK;
}
```

不同数据类型的
不同处理方法

```
/* linear_list_sq.c 的实现 */
/* 删除指定位置的元素，并将被删除元素的值放入e中返回 */
Status ListDelete(sqlist *L, int i, ElemType *e)
{
    ElemType *p, *q; //如果是算法，一般可以省略，程序不能

    if (i<1 || i>L->length)  //合理范围是 1..length
        return ERROR;

    p = &(L->elem[i-1]);     //指向第i个元素

    //取第i个元素的值放入e中
#if defined (ELEMTYPE_IS_CHAR_ARRAY) ||
                                defined (ELEMTYPE_IS_CHAR_P)
    strcpy(*e, *p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(e, p, sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    memcpy(*e, *p, sizeof(ET));
#else    //int和double直接赋值
    *e = *p;
#endif

    q = &(L->elem[L->length-1]);    //指向最后一个元素

    //两种情况需要释放空间，其它4种不需要
#if defined (ELEMTYPE_IS_CHAR_P) ||
                          defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    free(*p);    //释放空间
#endif
```

六合一

```
    /* 从第i+1到最后，依次前移一格 */
    for (++p; p<=q; ++p) {
#if defined (ELEMTYPE_IS_CHAR_ARRAY)
        strcpy(*(p-1), *p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
        memcpy((p-1), p, sizeof(ElemType));
#else       //int、double、char指针、struct
student指针都是直接赋值
        *(p-1) = *p;
#endif
        }

    L->length --;       //长度-1
    return OK;
}
```

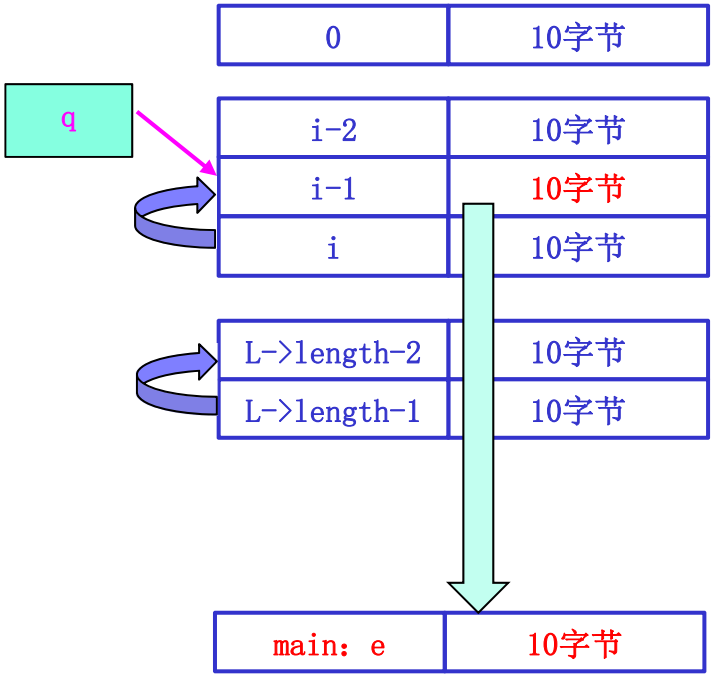
不同数据类型的
不同处理方法

```
/* linear_list_sq.c 的实现 */
/* 删除指定位置的元素，并将被删除元素的值放入e中返回 */
Status ListDelete(sqlist *L, int i, ElemType *e)
{
    ElemType *p, *q; //如果是算法，一般可以省略，程序不能

    if (i<1 || i>L->length)  //合理范围是 1..length
        return ERROR;

    p = &(L->elem[i-1]);     //指向第i个元素

    //取第i个元素的值放入e中
#if defined (ELEMTYPE_IS_CHAR_ARRAY) ||
                              defined (ELEMTYPE_IS_CHAR_P)
    strcpy(*e, *p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(e, p, sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    memcpy(*e, *p, sizeof(ET));
#else    //int和double直接赋值
    *e = *p;
#endif

    q = &(L->elem[L->length-1]);    //指向最后一个元素

    //两种情况需要释放空间，其它4种不需要
#if defined (ELEMTYPE_IS_CHAR_P) ||
                          defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    free(*p);    //释放空间
#endif
```

char []
struct student

六合一

```
    /* 从第i+1到最后，依次前移一格 */
    for (++p; p<=q; ++p) {
#if defined (ELEMTYPE_IS_CHAR_ARRAY)
        strcpy(*(p-1), *p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
        memcpy((p-1), p, sizeof(ElemType));
#else        //int、double、char指针、struct
student指针都是直接赋值
        *(p-1) = *p;
#endif
        }

    L->length --;        //长度-1
    return OK;
}
```

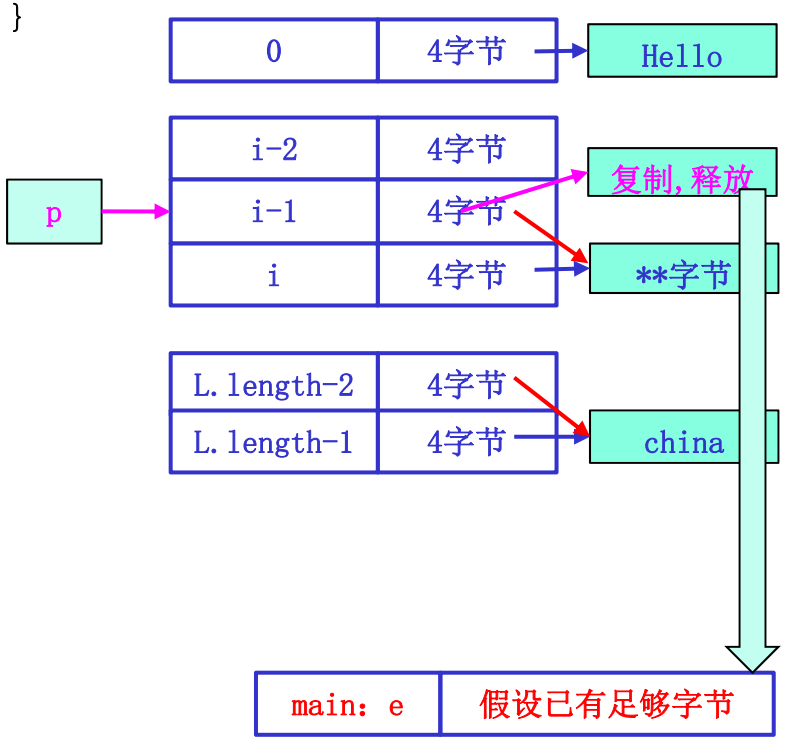


```
/* linear_list_sq.c 的实现 */
/* 删除指定位置的元素，并将被删除元素的值放入e中返回 */
Status ListDelete(sqlist *L, int i, ElemType *e)
{
    ElemType *p, *q; //如果是算法，一般可以省略，程序不能

    if (i<1 || i>L->length)  //合理范围是 1..length
        return ERROR;

    p = &(L->elem[i-1]);      //指向第i个元素

    //取第i个元素的值放入e中
#if defined (ELEMTYPE_IS_CHAR_ARRAY) ||
                              defined (ELEMTYPE_IS_CHAR_P)
    strcpy(*e, *p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    memcpy(e, p, sizeof(ElemType));
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    memcpy(*e, *p, sizeof(ET));
#else    //int和double直接赋值
    *e = *p;
#endif

    q = &(L->elem[L->length-1]);    //指向最后一个元素

    //两种情况需要释放空间，其它4种不需要
#if defined (ELEMTYPE_IS_CHAR_P) ||
                          defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    free(*p);    //释放空间
#endif
```

char *
struct student *

```
    /* 从第i+1到最后，依次前移一格 */
    for (++p; p<=q; ++p) {
#if defined (ELEMTYPE_IS_CHAR_ARRAY)
        strcpy(*(p-1), *p);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
        memcpy((p-1), p, sizeof(ElemType));
#else       //int、double、char指针、struct
student指针都是直接赋值
        *(p-1) = *p;
#endif
        }

    L->length --;        //长度-1
    return OK;
}
```

六合一



/* linear_list_sq.c 的实现 */

六合一

```
/* 遍历线性表 */
Status ListTraverse(sqlist L, Status (*visit)(ElemType e))
{   extern int line_count; //main中定义的换行计数器(与算法无关)
    ElemType *p = L.elem;
    int       i = 1;

    line_count = 0;         //计数器恢复初始值（与算法无关）
    while(i<=L.length && (*visit)(*p++)==TRUE)
        i++;
    if (i<=L.length)
            return ERROR;

    printf("\n");  //最后打印一个换行，只是为了好看，与算法无关
    return OK;
}
```

所有数据类型的处理方法都相同无变化

本函数牵涉到不同数据类型的输出格式，条件编译放在main中更合理!!!

```
/* main中用于比较访问线性表某个元素的值的具体函数，与 ListTraverse 中的函数
   指针定义相同，调用时传入 */

Status MyVisit(ElemType e)
{
#ifdef ELEMTYPE_IS_DOUBLE
    printf("%5.1f->", e);
#elif defined (ELEMTYPE_IS_CHAR_ARRAY) || defined (ELEMTYPE_IS_CHAR_P)
    printf("%s->", e);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT)
    printf("%d-%s-%c-%f-%s->", e.num, e.name, e.sex, e.score, e.addr);
#elif defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    printf("%d-%s-%c-%f-%s->", e->num, e->name, e->sex, e->score, e->addr);
#else          //缺省当做int处理
    printf("%3d->", e);
#endif

    /* 每输出10个，打印一个换行 */
    if ((++line_count)%10 == 0)
            printf("\n");
    return OK;
}
```

main中：
ListTraverse(L, MyVisit);

§2. 线性表

=> 另类思考：如何使 linear_list_sq.c 中不包含条件编译宏定义，而是将宏定义放在main中？

注：仅仅从代码角度理解，实用中不应该!!!

不同处理之处：二级指针整体释放、单个二级指针的申请/释放、比较、赋值、移动、遍历

```
/* 以删除线性表为例，具体参考源程序：08-linear_list_sq_C_6_in_1_another */
Status DestroyList(sqlist *L, Status (*sub_free)(sqlist *L))
{
    /* 按需释放空间 */
    (*sub_free)(L);

    /* 若未执行 InitList，直接执行本函数，则可能出错 */
    if (L->elem)
        free(L->elem);
    L->length   = 0;
    L->listsize = 0;

    return OK;
}
```

main中调用方法：

```
DestroyList(&L, MySubFree);
```

```
/* main中的函数，用于释放二级空间 */
Status MySubFree(sqlist *L)
{
#if defined (ELEMTYPE_IS_CHAR_P) ||
          defined (ELEMTYPE_IS_STRUCT_STUDENT_P)
    int i;
    /* 首先释放二级空间 */
    for(i=0; i<L->length; i++)
        free(L->elem[i]);
#endif
    return OK;
}
```

§2. 线性表

2.2. 线性表的顺序表示和实现

2.2.2. 线性表顺序表示的基本操作的实现

2.2.2.1. C语言版

2.2.2.2. C++语言版（略）

2.2.3. 线性表顺序表示的时间复杂度

```
/* linear_list_sq.c 的实现 */
#include <stdio.h>
#include <stdlib.h>                          //malloc/realloc函数
#include <unistd.h>                          //exit函数
#include "linear_list_sq.h"                  //形式定义

/* 初始化线性表 */                                                    O(1)
Status InitList(sqlist *L)
{
    L->elem = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(ElemType));
    if (L->elem == NULL)
        exit(OVERFLOW);
    L->length = 0;
    L->listsize = LIST_INIT_SIZE;
    return OK;
}
```

```
/* linear_list_sq.c 的实现 */

/* 删除线性表 */
Status DestroyList(sqlist *L)
{
    /* 按需释放二级指针 */

    /* 释放一级指针 */
    if (L->elem)
        free(L->elem);
    L->length   = 0; //可不要
    L->listsize = 0; //可不要

    return OK;
}
```

含二级指针释放的O(n)，否则O(1)

一般教材认为O(1)

```
/* linear_list_sq.c 的实现 */

/* 清除线性表（已初始化，不释放空间，只清除内容） */
Status ClearList(sqlist *L)
{
    L->length = 0;
    return OK;
}
```

O(1)

```
/* linear_list_sq.c 的实现 */

/* 判断是否为空表 */
Status ListEmpty(sqlist L)
{
    if (L.length == 0)
        return TRUE;
    else
        return FALSE;
}
```

O(1)

```
/* linear_list_sq.c 的实现 */

/* 求表的长度 */
int ListLength(sqlist L)
{
    return L.length;
}
```

O(1)

```
/* linear_list_sq.c 的实现 */

/* 取表中元素 */
Status GetElem(sqlist L, int i, ElemType *e)
{
    /* i值合理范围[1..length]  */
    if (i<1 || i>L.length)
        return ERROR;

    各种不同形式的赋值;
    return OK;
}
```

O(1)

```
/* linear_list_sq.c 的实现 */

/* 查找符合指定条件的元素 */
int LocateElem(sqlist L, ElemType e, Status (*compare)(ElemType e1, ElemType e2))
{
    ElemType *p = L.elem;
    int       i = 1;

    while(i<=L.length && (*compare)(*p++, e)==FALSE)
        i++;

    return (i<=L.length) ? i : 0;
}
```

O(n)

本函数中，(*compare)的调用为预估算法时间复杂度的基本操作

假设Pi是第i个元素被查找的概率，则在长度为n的线性表中查找一个元素所需比较次数的期望值为：

第1个元素：1次
第2个元素：2次
.....
第n个元素：n次

假设等概率，$pi=\frac{1}{n}$

$$\sum_{i=1}^{n} pi*i=\frac{1}{n}\sum_{i=1}^{n} i=\frac{1}{n}*\frac{n(n+1)}{2}=\frac{n+1}{2}$$

```
/* linear_list_sq.c 的实现 */

/* 查找符合指定条件的元素的前驱元素 */
Status PriorElem(sqlist L, ElemType cur_e, ElemType *pre_e)
{
    ElemType *p = L.elem;
    int       i = 1;
    /* 循环比较整个线性表 */
    while(i<=L.length && 各种形式判断不等 ) {
        i++;
        p++;
        }
    if (i==1 || i>L.length) //找到第1个元素或未找到
        return ERROR;

    各种不同形式的赋值;
    return OK;
}
```

O(n)

```
/* linear_list_sq.c 的实现 */

/* 查找符合指定条件的元素的后继元素 */
Status NextElem(sqlist L, ElemType cur_e, ElemType *next_e)
{
    ElemType *p = L.elem;
    int       i = 1;
    /* 循环比较整个线性表(不含尾元素) */
    while(i<L.length && 各种形式判断不等 ) {
        i++;
        p++;
        }
    if (i>=L.length)        //未找到（最后一个元素未比较）
        return ERROR;

    各种不同形式的赋值;
    return OK;
}
```

O(n)

```
/* linear_list_sq.c 的实现 */
/* 在指定位置前插入一个新元素 */
Status ListInsert(sqlist *L, int i, ElemType e)
{   ElemType *p, *q; //如果是算法，可以省略，程序不能
    if (i<1 || i>L->length+1)   //合理范围是 1..length+1
            return ERROR;
    /* 空间已满则扩大空间 */
    if (L->length >= L->listsize) {
            ElemType *newbase;
            newbase = (ElemType *)realloc(L->elem,
                    (L->listsize+LISTINCREMENT)*sizeof(ElemType));
            if (!newbase)
                return OVERFLOW;
            L->elem = newbase;
            L->listsize += LISTINCREMENT;
            //L->length暂时不变
            }

    q = &(L->elem[i-1]);  //第i个元素，即新的插入位置

    /* 从最后一个(length放在[length-1]中)开始到第i个元素依次后移 */
    for (p=&(L->elem[L->length-1]); p>=q; --p)
        各种不同形式的移动;

    /* 插入新元素，长度+1 */
    各种不同形式的赋值(含申请空间);
    L->length ++;
    return OK;
}
```

O(n)

P.25 分析

```
/* linear_list_sq.c 的实现 */
/* 删除指定位置的元素，并将被删除元素的值放入e中返回 */
Status ListDelete(sqlist *L, int i, ElemType *e)
{   ElemType *p, *q; //如果是算法，一般可以省略，程序不能
    if (i<1 || i>L->length) //合理范围是 [1..length]
        return ERROR;

    p = &(L->elem[i-1]);            //指向第i个元素
    各种不同形式的赋值(含释放空间);
    q = &(L->elem[L->length-1]);    //指向最后一个元素
                                    //也可以 q = L->elem+L->length-1
    /* 从第i+1到最后，依次前移一格 */
    for (++p; p<=q; ++p)
        各种不同形式的移动;
    L->length --; //长度-1
    return OK;
}
```

O(n)

P.25 分析

```
/* linear_list_sq.c 的实现 */

/* 遍历线性表 */
Status ListTraverse(sqlist L, Status (*visit)(ElemType e))
{
    extern int line_count; //main中定义的换行计数器(与算法无关)
    ElemType *p = L.elem;
    int       i = 1;

    line_count = 0;         //计数器恢复初始值（与算法无关）
    while(i<=L.length && (*visit)(*p++)==TRUE)
        i++;                          (*visit)为基本操作
    if (i<=L.length)
         return ERROR;

    printf("\n");  //最后打印一个换行，只是为了好看，与算法无关
    return OK;
}
```

指向函数的指针，主函数中调用时传入一个具体的函数名即可

O(n)

§2. 线性表

2.2. 线性表的顺序表示和实现

2.2.4. 线性表的复杂操作(顺序表示)

2.2.4.1. 集合的并操作(P.20 例2-1)

假设利用两个线性表LA和LB分别表示两个集合A和B(即线性表中的数据即为集合中的成员)，现要求一个新集合 A=AUB

★ LA扩大，LB不变

★ LB中的每个元素都在LA中进行查找，找到则忽略，找不到则插入LA中

★ 考虑插入效率，每次插入在最后(不移动)

```
//C形式实现
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, equal))
            ListInsert(La, ++La_len, e);
        }
}
```

1、因为La会改变，所以形参为&La
2、equal为判断是否相等的函数
3、++La_len表示插入在最后
4、写算法时，假设基本操作均已实现

```
//C++形式实现
void union(List &La, List Lb)
{   La_len = La.ListLength();
    Lb_len = Lb.ListLength();
    for(i=1; i<=Lb_len; i++) {
        Lb.GetElem(i, e);
        if (!La.LocateElem(e, equal))
            La.ListInsert(++La_len, e);
        }
}
```

假设La长度为m，Lb长度为n
ListLength() -> O(1)
GetElem() -> O(1)*n
ListInsert() -> O(1)*n //每次插最后
LocateElem() -> O(m)*n
时间复杂度为O(m*n)

若要求C=AUB，请自行思考实现方法

§2.线性表

2.2.线性表的顺序表示和实现

2.2.4.线性表的复杂操作(顺序表示)

2.2.4.1.集合的并操作(P.20 例2-1)

2.2.4.2.线性表的有序归并(P.20 例2-2)

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

★ 非递减有序(不是递增)，说明不同数据元素的值可能相同

★ LC初始为空，用两个指针(不是C/C++指针概念)分别指向LA、LB的首元素，比较两者大小，小的插入LC中，指针后移，剩余部分最后插入

2.2.4.2.线性表的有序归并(P.21 算法2.2)

★ 抽象算法

```
void MergeList(List La, list Lb, List &Lc)
{   InitList(Lc);  // 初始化LC
    i = j = 1;     // 用i,j下标来代表初始指针
    k=0;           // LC的初始长度
    La_len = ListLength(La);
    Lb_len = ListLength(Lb);
    while((i<=La_len) && (j<=Lb_len)) {
        GetELem(La, i, ai);
        GetElem(Lb, j, bj);
        if (ai<bj) {
            ListInsert(Lc, ++k; ai); //ai插入LC的最后(有序)
            i++;                     //LA的指针后移
            }
        else {
            ListInsert(lc, ++k, bj);//bj插入LC的最后(有序)
            j++;                    //LB的指针后移
            }
     } //end of while()
    while(i<=La_len) {          //若LA还有剩余元素，插入LC中(有序)
        GetElem(La, i++, ai);
        ListInsert(Lc, ++k, ai);
        }
    while(j<=Lb_len) {          //若LB还有剩余元素，插入LC中(有序)
        GetElem(Lb, j++, bj);
        ListInsert(Lc, ++k, bj);
        }
}
```

若i在若干次循环中保持不变
则GetElem(La,i,ai)被无意义重复(j同)
如何改？

两个while，只有一个被执行
不需要外面再嵌套if!!!

假设La的长度为m，Lb的长度为n
(1) 初始化 -> O(1)
(2) 复制La剩余+复制Lb剩余
-> O(x) [x为剩余元素]
(3) while归并 -> O(m+n-x)
时间复杂度为O(m+n)

2.2.4.2.线性表的有序归并(P.26 算法2.7)

★ 顺序表示的算法

```
void MergeList_sq(SqList La, Sqlist Lb, SqList &Lc)
{   //初始化
    pa = La.elem; //指向LA的首元素
    pb = Lb.elem; //指向LB的首元素
    Lc.listsize = Lc.length = La.length+Lb.length;
    pc = Lc.elem = (ElemType *)malloc(Lc.listsize*sizeof(ElemType));
    if (!Lc.elem)
        exit(OVERFLOW);
    pa_last = La.elem+La.length-1; //指向LA的尾元素
    pb_last = La.elem+Lb.length-1; //指向LB的尾元素
    while(pa<=pa_last && pb<=pb_last) {  //归并
        if (*pa <= *pb)
            *pc++ = *pa++;
        else
            *pc++ = *pb++;
        } //end of while
    while(pa<=pa_last)//若LA还有剩余元素，插入LC中(有序)
        *pc++ = *pa++;
    while(pb<=pb_last)//若LB还有剩余元素，插入LC中(有序)
        *pc++ = *pb++;
}
```

1、时间复杂度仍为O(m+n)不变
2、Lc一次申请全部空间，避免超过初始大小时反复扩大
3、跳过GetElem和ListInsert函数直接执行，效率高
4、*p++ 比 *(p+i) 效率高
5、避免GetElem的无意义重复

§2. 线性表

2.3. 线性表的链式表示和实现

2.3.1. 链式表示的特点

★ 顺序存储结构的缺点

- 插入/删除时要移动元素
- 分配的空间比实际所需空间大，且当线性表长度达到当前分配的存储容量时，需要增加存储容量的操作会导致内存的大量复制/移动
- 内存空间要求连续，无法充分利用

★ 链式存储结构的特点

不利用元素在存储器中的物理位置来表示其逻辑顺序，而是在每个元素中包含其直接后继元素的位置信息(最后一个为NULL)，因此可用任意单元存储，不必考虑是否连续以及物理上的先后顺序

★ 线性链表结点的组成

数据元素a_i的存储映象称为结点，由两部分组成：

{ 数据域：元素本身的信息
{ 指针域：存储直接后继元素的位置，称为指针/链

- 一个线性表的所有结点链结成一个链表(单链表)
- 指示链表中第一个结点的存储映象的存储位置，称为头指针
- 链表使用时，只关心逻辑位置，不关心物理位置

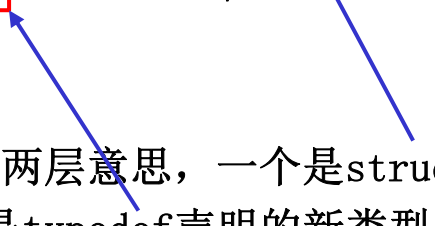
§2. 线性表

2.3. 线性表的链式表示和实现

2.3.1. 链式表示的特点

★ 单链表存储结构的描述(P.28)

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



- LNode有两层意思，一个是struct的名字，
  另一个是typedef声明的新类型名
- struct LNode不能省略，因为结构体中要使用
- LNode *p 相当于 LinkList p，都表示指针

★ 头指针、头结点与首元结点

首元结点：表示链表中第一个元素的结点

头结点：附设的结点，data域无意义，next域指向首元结点

头指针：指向链表中第一个结点(首元结点/头结点)的指针

§2. 线性表

2.3. 线性表的链式表示和实现

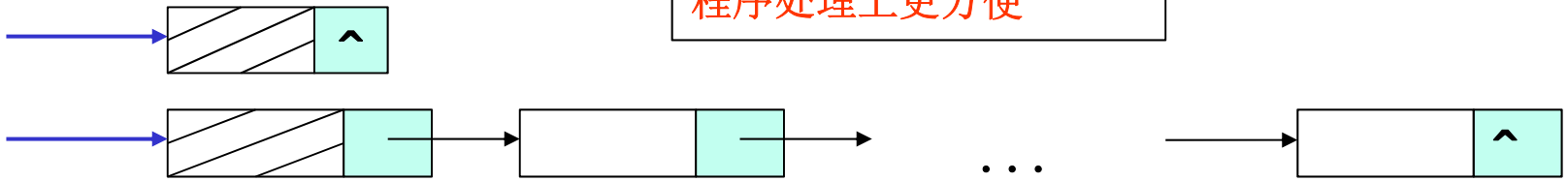
2.3.1. 链式表示的特点

★ 带头结点和不带头结点的单链表

带头结点的单链表：头指针指向头结点，头结点的指针域为NULL表示链表为空

- 初始化后头指针始终不变

虽然浪费一个O(1)空间，程序处理上更方便



不带头结点的单链表：头指针指向首元结点，头指针为NULL表示链表为空

- 头指针可能不断变化

§2.线性表

2.3.线性表的链式表示和实现

2.3.2.线性表链式表示的基本操作的实现

2.3.2.1.C语言版

★ 程序的组成

- linear_list_L.h　　：头文件
- linear_list_L.c　　：具体实现
- linear_list_L_main.c　：使用（测试）示例

★ ElemType => int （带头结点）

```
/* linear_list_L.h 的组成 */
#define TRUE          1
#define FALSE         0
#define OK            1
#define ERROR         0
#define INFEASIBLE   -1
#define OVERFLOW     -2

typedef int Status;

typedef int ElemType;       //可根据需要修改元素的类型

typedef struct LNode {
    ElemType       data;    //存放数据
    struct LNode *next;     //存放直接后继的指针
} LNode, *LinkList;
```

P.10 预定义常量和类型

P.28 形式定义

```
/* linear_list_L.h 的组成 */
Status   InitList(LinkList *L);
Status   DestroyList(LinkList *L);
Status   ClearList(LinkList *L);
Status   ListEmpty(LinkList L);
int      ListLength(LinkList L);
Status   GetElem(LinkList L, int i, ElemType *e);
int      LocateElem(LinkList L, ElemType e,
                    Status (*compare)(ElemType e1, ElemType e2));
Status   PriorElem(LinkList L, ElemType cur_e, ElemType *pre_e);
Status   NextElem(LinkList L, ElemType cur_e, ElemType *next_e);
Status   ListInsert(LinkList *L, int i, ElemType e);
Status   ListDelete(LinkList *L, int i, ElemType *e);
Status   ListTraverse(LinkList L, Status (*visit)(ElemType e));
```

将顺序实现中 sqlist => LinkList，其它未变，具体说明、注意等与顺序实现相同

```
/* linear_list_L.c 的实现 */
#include <stdio.h>
#include <stdlib.h>                    //malloc/realloc函数
#include <unistd.h>                    //exit函数
#include "linear_list_L.h"             //形式定义

/* 初始化线性表 */
Status InitList(LinkList *L)
{
    /* 申请头结点空间，赋值给头指针 */
    *L = (LNode *)malloc(sizeof(LNode));
    if (*L==NULL)
        exit(OVERFLOW);
    (*L)->next = NULL;
    return OK;
}
```

★ main函数中
声明为 LinkList L
调用为 InitList(&L);

★ 实参 LinkList L ，L是指针，
因此 &L 是指针的指针

★ 形参 LinkList *L，L是指针的指针

★ 换成 L 会错，具体原因同顺序表

```
/* linear_list_L.c 的实现 */

/* 删除线性表 */
Status DestroyList(LinkList *L)
{
    LinkList q, p = *L;

    /* 从头结点开始依次释放（含头结点） */
    while(p) {    //若链表为空，则循环不执行
        q=p->next; //抓住链表的下一个结点
        free(p);
        p=q;
        }

    *L=NULL;      //头指针置NULL，可不要
    return OK;
}
```

不能L，理由同InitList

```
/* linear_list_L.c 的实现 */

/* 清除线性表（保留头结点） */
Status ClearList(LinkList *L)
{
    LinkList q, p = (*L)->next;

    /* 从首元结点开始依次释放 */
    while(p) {
        q = p->next;    //抓住链表的下一个结点
        free(p);
        p = q;
        }

    (*L)->next = NULL; //头结点的next域置NULL
    return OK;
}
```

可以L，为什么？
此处：为了与顺序表保持一致而用*L

```
/* linear_list_L.c 的实现 */

/* 判断是否为空表 */
Status ListEmpty(LinkList L)
{
    /* 判断头结点的next域即可 */
    if (L->next==NULL)
         return TRUE;
    else
         return FALSE;
}
```

```
/* linear_list_L.c 的实现 */

/* 求表的长度 */
int ListLength(LinkList L)
{
    LinkList p = L->next; //指向首元结点
    int len=0;

    /* 循环整个链表，进行计数 */
    while(p) {
         p = p->next;
         len++;
         }

    return len;
}
```

```
/* linear_list_L.c 的实现 */

/* 求表的长度 */
int ListLength(LinkList L)
{
    LinkList p = L->next; //指向首元结点
    int len=0;

    /* 循环整个链表，进行计数 */
    while(p) {
         p = p->next;
         len++;
         }

    return len;
}
```

```
LinkList p=L;



while((p=p->next)!=NULL)
    len++;
```

```
/* linear_list_L.c 的实现 */

/* 取表中元素 */
Status GetElem(LinkList L, int i, ElemType *e)
{
    LinkList p=L->next; //指向首元结点
    int pos = 1;        //初始位置为1

    /* 链表不为NULL 且 未到第i个元素 */
    while(p!=NULL && pos<i) {
        p=p->next;
        pos++;
        }

    if (!p || pos>i)
        return ERROR;

    *e = p->data;
    return OK;
}
```

循环结束条件为两者之一不满足：
若p==NULL，则表示i值不合理
若pos>=i，则表示已找到第i个位置
或位置不合法!!!

问：能否只写 if (!p) ？
答：若i不合法(例如-1)则仅if(!p)无法判断

```
/* linear_list_L.c 的实现 */

/* 查找符合指定条件的元素 */
int LocateElem(LinkList L, ElemType e, Status (*compare)(ElemType e1, ElemType e2))
{
    LinkList p = L->next;  //首元结点
    int pos = 1;           //初始位置

    /* 循环整个链表  */
    while(p && (*compare)(e, p->data)==FALSE) {
        p=p->next;
        pos++;
        }

    return p ? pos:0;
}
```

循环结束条件为两者之一不满足：
若p==NULL：未找到
compare==TRUE：找到，此时p!=NULL

```
/* main函数中的MyCompare函数，与顺序实现完全一致，
   调用时：LocateElem(L, e, MyCompare)即可 */
Status MyCompare(ElemType e1, ElemType e2)
{
    if (e1==e2)
        return TRUE;
    else
        return FALSE;
}
```

```
/* linear_list_L.c 的实现 */

/* 查找符合指定条件的元素的前驱元素 */
Status PriorElem(LinkList L, ElemType cur_e, ElemType *pre_e)
{
    LinkList p = L->next;  //指向首元结点
    if (p==NULL)  //空表直接返回
        return ERROR;

    /* 从第2个结点开始循环整个链表(如果比较相等，保证有前驱) */
    while(p->next && p->next->data != cur_e)
        p = p->next;

    if (p->next==NULL) //未找到
        return ERROR;

    *pre_e = p->data;
    return OK;
}
```

循环结束条件为两者之一不满足：
若p->next==NULL　　：未找到
　p->next->data==cur_e：找到

```
/* linear_list_L.c 的实现 */

/* 查找符合指定条件的元素的前驱元素 */
Status PriorElem(LinkList L, ElemType cur_e, ElemType *pre_e)
{
    LinkList p = L; //指向头结点

    /* 循环整个链表并比较值是否相等 */
    while(p->next && p->next->data != cur_e)
        p = p->next;

    if (p->next==NULL || p==L) //未找到或首元结点或空表
        return ERROR;

    *pre_e = p->data;
    return OK;
}
```

循环结束条件为两者之一不满足：
若p->next==NULL：未找到/空表
p->next->data==cur_e：找到
但必须排除首元结点就相等

还可以设置两个指针，一个指向结点，一个指向结点的前驱，同步移动，具体方法自行思考

```
/* linear_list_L.c 的实现 */

/* 查找符合指定条件的元素的后继元素 */
Status NextElem(LinkList L, ElemType cur_e, ElemType *next_e)
{
    LinkList p = L->next;  //首元结点

    if (p==NULL)  //空表直接返回
        return ERROR;
    /* 有后继结点且当前结点值不等时继续 */
    while(p->next && p->data!=cur_e)
        p = p->next;

    if (p->next==NULL)
        return ERROR;
    *next_e = p->next->data;
    return OK;
}
```

循环结束条件为两者之一不满足：
若p->data==cur_e：找到
p->next==NULL：未找到

```
/* linear_list_L.c 的实现 */
/* 在指定位置前插入一个新元素 */
Status ListInsert(LinkList *L, int i, ElemType e)
{   LinkList s, p = *L;  //p指向头结点
    int      pos = 0;
    /* 寻找第i-1个结点 */
    while(p && pos<i-1) {
         p=p->next;
         pos++;
         }
    if (p==NULL || pos>i-1)  //i值非法则返回，同GetElem
         return ERROR;
    //执行到此表示找到指定位置，p指向第i-1个结点
    s = (LinkList)malloc(sizeof(LNode)); //新申请1个结点
    if (s==NULL)
         return OVERFLOW;
    s->data = e; //新结点数据域赋值
    s->next = p->next; //新结点的next是第i个
    p->next = s;       //第i-1个的next是新结点
    return OK;
}
```

可以L，为了与顺序表保持一致而用*L

★ i的合理范围 1..length+1

这两句顺序不能反

程序要保证下列情况都正确：
★ 空表
★ 有元素，首元前插入
★ 有元素，中间任意位置插入
★ 有元素，最后一个之后插入

```
/* linear_list_L.c 的实现 */
/* 删除指定位置的元素，并将被删除元素的值放入e中返回 */
Status ListDelete(LinkList *L, int i, ElemType *e)
{
    LinkList q, p = *L; //p指向头结点
    int      pos = 0;
    /* 寻找第i个结点（p->next是第i个结点） */
    while(p->next && pos<i-1) {
        p=p->next;
        pos++;
        }
    if (p->next==NULL || pos>i-1) //i值非法则返回，同GetElem
        return ERROR;
    //执行到此表示找到了第i个结点，此时p指向第i-1个结点
    q = p->next;          //q指向第i个结点
    p->next = q->next; //第i-1个结点的next域指向第i+1个

    *e = q->data;         //取第i个结点的值
    free(q);              //释放第i个结点

    return OK;
}
```

可以L，为了与顺序表保持一致而用*L

程序要保证下列情况都正确：

★ 空表
★ 仅有一个元素
★ 有元素，删任意位置
★ 有元素，删最后一个

```
/* linear_list_L.c 的实现 */

/* 遍历线性表 */
Status ListTraverse(LinkList L, Status (*visit)(ElemType e))
{
    extern int line_count; //main中定义的换行计数器(与算法无关)
    LinkList p = L->next;  //指向首元

    line_count = 0;                    //计数器恢复初始值（与算法无关）
    while(p && (*visit)(p->data)==TRUE)
        p=p->next;

    if (p)
        return ERROR;

    printf("\n");//最后打印一个换行，只是为了好看，与算法无关
    return OK;
}
```

```
/* main函数中的MyVisit函数，与顺序实现完全一致，
   调用时：ListTraverseElem(L, MyVisit)即可 */
Status MyVisit(ElemType e)
{
    printf("%3d->", e); //此句输出
    return OK;
}
```

```
/* linear_list_sq_main.c 中 MyCompare和MyVisit的实现 */
Status MyCompare(ElemType e1, ElemType e2)
{
    if (e1==e2)
        return TRUE;
    else
        return FALSE;
}

Status MyVisit(ElemType e)
{
    printf("%3d->", e); //此句输出

    /* 每输出MAX_NUM_PER_LINE个，打印一个换行 */
    if ((++line_count)%20 == 0)
        printf("\n");

    return OK;
}
```

这两个函数与顺序实现完全一致

§2.线性表

2.3.线性表的链式表示和实现

2.3.2.线性表链式表示的基本操作的实现

2.3.2.1.C语言版

★ 程序的组成

- linear_list_L.h　　：头文件
- linear_list_L.c　　：具体实现
- linear_list_L_main.c　：使用（测试）示例

★ ElemType => int （带头结点）

★ ElemType => int （不带头结点）

```
/* linear_list_L.h 的组成 */
#define TRUE          1
#define FALSE         0
#define OK            1
#define ERROR         0
#define INFEASIBLE   -1
#define OVERFLOW     -2

typedef int Status;

typedef int ElemType;       //可根据需要修改元素的类型

typedef struct LNode {
    ElemType       data;    //存放数据
    struct LNode *next;     //存放直接后继的指针
} LNode, *LinkList;
```

P.10 预定义常量和类型

P.28 形式定义

无变化

/* linear_list_L.h 的组成 */

```
Status    InitList(LinkList *L);
Status    DestroyList(LinkList *L);
Status    ClearList(LinkList *L);
Status    ListEmpty(LinkList L);
int       ListLength(LinkList L);
Status    GetElem(LinkList L, int i, ElemType *e);
int       LocateElem(LinkList L, ElemType e,
                                 Status (*compare)(ElemType e1, ElemType e2));
Status    PriorElem(LinkList L, ElemType cur_e, ElemType *pre_e);
Status    NextElem(LinkList L, ElemType cur_e, ElemType *next_e);
Status    ListInsert(LinkList *L, int i, ElemType e);
Status    ListDelete(LinkList *L, int i, ElemType *e);
Status    ListTraverse(LinkList L, Status (*visit)(ElemType e));
```

无变化

将顺序实现中 sqlist => LinkList，其它未变，具体说明、注意等与顺序实现相同

```
/* linear_list_L.c 的实现 */
#include <stdio.h>
#include <stdlib.h>                          //malloc/realloc函数
#include <unistd.h>                          //exit函数
#include "linear_list_L.h"                   //形式定义

/* 初始化线性表 */
Status InitList(LinkList *L)
{
    *L = NULL;  //头指针直接赋NULL
    return OK;
}
```

不能L，理由同带头结点

```
/* linear_list_L.c 的实现 */

/* 删除线性表 */
Status DestroyList(LinkList *L)
{
    LinkList q, p = *L;

    /* 从首元结点开始依次释放 */
    while(p) {    //若链表为空，则循环不执行
        q=p->next; //抓住链表的下一个结点
        free(p);
        p=q;
        }

    *L=NULL;      //头指针置NULL，可不要
    return OK;
}
```

无变化

不能L，理由同带头结点

```
/* linear_list_L.c 的实现 */

/* 清除线性表（保留头结点） */
Status ClearList(LinkList *L)
{
    LinkList q, p = *L; //指向首元

    /* 整个链表依次释放 */
    while(p) {
        q=p->next; //抓住链表的下一个结点
        free(p);
        p=q;
        }

    *L=NULL;        //头指针置NULL，必须要
    return OK;
}
```

不能L，理由同带头结点

与DestroyList完全相同

```
/* linear_list_L.c 的实现 */

/* 判断是否为空表 */
Status ListEmpty(LinkList L)
{
    /* 判断头指针即可 */
    if (L==NULL)
        return TRUE;
    else
        return FALSE;
}
```

```
/* linear_list_L.c 的实现 */

/* 求表的长度 */
int ListLength(LinkList L)
{
    LinkList p = L; //指向首元结点
    int len=0;

    /* 循环整个链表，进行计数 */
    while(p) {
        p = p->next;
        len++;
        }

    return len;
}
```

```
/* linear_list_L.c 的实现 */

/* 取表中元素 */
Status GetElem(LinkList L, int i, ElemType *e)
{
    LinkList p=L;          //指向首元结点
    int pos = 1;           //初始位置为1
    /* 链表不为NULL 且 未到第i个元素 */
    while(p!=NULL && pos<i) {
        p=p->next;
        pos++;
        }

    if (!p || pos>i)
        return ERROR;

    *e = p->data;
    return OK;
}
```

```
/* linear_list_L.c 的实现 */

/* 查找符合指定条件的元素 */
int LocateElem(LinkList L, ElemType e, Status (*compare)(ElemType e1, ElemType e2))
{
    LinkList p = L;          //首元结点
    int pos = 1;             //初始位置
    /* 循环整个链表  */
    while(p && (*compare)(e, p->data)==FALSE) {
        p=p->next;
        pos++;
        }

    return p ? pos:0;
}
```

```
/* main函数中的 MyCompare 函数
  （略） */
```

```
/* linear_list_L.c 的实现 */

/* 查找符合指定条件的元素的前驱元素 */
Status PriorElem(LinkList L, ElemType cur_e, ElemType *pre_e)
{
    LinkList p = L;          //指向首元结点
    if (p==NULL)  //空表直接返回
         return ERROR;

    /* 从第2个结点开始循环整个链表(如果比较相等，保证有前驱) */
    while(p->next && p->next->data != cur_e)
        p = p->next;

    if (p->next==NULL) //未找到
         return ERROR;

    *pre_e = p->data;
    return OK;
}
```

循环结束条件为两者之一不满足：
若p->next==NULL　　：未找到
　p->next->data==cur_e：找到

```
/* linear_list_L.c 的实现 */

/* 查找符合指定条件的元素的后继元素 */
Status NextElem(LinkList L, ElemType cur_e, ElemType *next_e)
{
    LinkList p = L;  //首元结点

    if (p==NULL)  //空表直接返回
        return ERROR;
    /* 有后继结点且当前结点值不等时继续 */
    while(p->next && p->data!=cur_e)
        p = p->next;

    if (p->next==NULL)
        return ERROR;
    *next_e = p->next->data;
    return OK;
}
```

循环结束条件为两者之一不满足：
若p->data==cur_e：找到
p->next==NULL：未找到

头指针的处理差别较大

```
/* linear_list_L.c 的实现 */
/* 在指定位置前插入一个新元素 */
Status ListInsert(LinkList *L, int i, ElemType e)
{   LinkList s, p = *L;     //p指向首元结点(可能为NULL)
    int      pos  = 1;      //因为p已指向首元，故起始位置是1
    /* 如果新结点成为首元，则需要改变L的值，其它位置插入则L不变 */
    if (i != 1) {
        /* 寻找第i-1个结点 */
        while(p && pos<i-1) {
            p=p->next;
            pos++;
            }
        if (p==NULL || pos>i-1)   //i值非法则返回
            return ERROR;
        }
    //执行到此，要么是i=1 的情况，要么是i>1但找到插入位置的情况
     s = (LinkList)malloc(sizeof(LNode));
    if (s==NULL)
        return OVERFLOW;
    s->data = e; //新结点数据域赋值
    if (i != 1) {    //插入位置非首元，此时p指向第i-1个结点
        s->next = p->next;  //新结点的next是p->next
        p->next = s;        //第i-1个的next是新结点
        }
    else {                  //插入位置是首元
        s->next = p;        //此时p就是L（包括L=NULL的情况）
        *L = s;             //头指针指向新结点
        }
    return OK;
}
```

不能L，与带头结点不同

头指针的处理差别较大

不能L，与带头结点不同

```
/* linear_list_L.c 的实现 */
/* 删除指定位置的元素，并将被删除元素的值放入e中返回 */
Status ListDelete(LinkList *L, int i, ElemType *e)
{
    LinkList q=NULL, p = *L;  //p指向首元结点(可能为NULL)
    int      pos  = 1;        //因为p已指向首元，故起始位置是1

    if (p==NULL)        //空表直接返回
        return ERROR;
    /* 如果删除的不是首元，则查找第i个结点 */
    if (i != 1) {
        /* 寻找第i个结点（p->next是第i个结点） */
        while(p->next && pos<i-1) {
            p=p->next;
            pos++;
            }
        if (p->next==NULL || pos>i-1)       //i值非法则返回
            return ERROR;

        q = p->next;        //q指向第i个结点
        p->next = q->next; //第i-1个结点的next域指向第i+1个
        }
    else { //要删除的是首元
        q = p;
        *L = q->next;    //如果只有一个结点，则q->next为NULL
        }
    *e = q->data;        //取第i个结点的值
    free(q);             //释放第i个结点
    return OK;
}
```

```
/* linear_list_L.c 的实现 */

/* 遍历线性表 */
Status ListTraverse(LinkList L, Status (*visit)(ElemType e))
{
    extern int line_count; //main中定义的换行计数器(与算法无关)
    LinkList p = L;    //指向首元

    line_count = 0;                    //计数器恢复初始值（与算法无关）
    while(p && (*visit)(p->data)==TRUE)
        p=p->next;

    if (p)
        return ERROR;

    printf("\n");//最后打印一个换行，只是为了好看，与算法无关
    return OK;
}
```

```
/* main函数中的 MyVisit 函数
   （略） */
```

§2.线性表

2.3.线性表的链式表示和实现

2.3.2.线性表链式表示的基本操作的实现

2.3.2.1.C语言版

2.3.2.2.C++语言版（暂时略）

§2.线性表

2.3.线性表的链式表示和实现

2.3.2.线性表链式表示的基本操作的实现

2.3.2.1.C语言版

2.3.2.2.C++语言版（暂时略）

2.3.3.线性表链式表示的时间复杂度

★ 以 ElemType => int 为例（其它均相同）

```
/* linear_list_L.c 的实现 */
#include <stdio.h>
#include <stdlib.h>                    //malloc/realloc函数
#include <unistd.h>                    //exit函数
#include "linear_list_L.h"             //形式定义

/* 初始化线性表 */
Status InitList(LinkList *L)
{
    /* 申请头结点空间，赋值给头指针 */
    *L = (LNode *)malloc(sizeof(LNode));
    if (*L==NULL)
        exit(OVERFLOW);
    (*L)->next = NULL;
    return OK;
}
```

O(1)

顺序表：O(1)

```
/* linear_list_L.c 的实现 */

/* 删除线性表 */
Status DestroyList(LinkList *L)
{
    LinkList q, p = *L;

    /* 从头结点开始依次释放（含头结点） */
    while(p) {    //若链表为空，则循环不执行
        q=p->next; //抓住链表的下一个结点
        free(p);
        p=q;
        }

    *L=NULL;       //头指针置NULL
    return OK;
}
```

O(n)

顺序表：O(1)

```
/* linear_list_L.c 的实现 */

/* 清除线性表（保留头结点） */
Status ClearList(LinkList *L)
{
    LinkList q, p = (*L)->next;

    /* 从首元结点开始依次释放 */
    while(p) {
        q = p->next;    //抓住链表的下一个结点
        free(p);
        p = q;
        }

    (*L)->next = NULL; //头结点的next域置NULL
    return OK;
}
```

O(n)

顺序表：O(1)

```
/* linear_list_L.c 的实现 */

/* 判断是否为空表 */
Status ListEmpty(LinkList L)
{
    /* 判断头结点的next域即可 */
    if (L->next==NULL)
        return TRUE;
    else
        return FALSE;
}
```

O(1)

顺序表：O(1)

```
/* linear_list_L.c 的实现 */

/* 求表的长度 */
int ListLength(LinkList L)
{
    LinkList p = L->next; //指向首元结点
    int len=0;

    /* 循环整个链表，进行计数 */
    while(p) {
        p = p->next;
        len++;
        }

    return len;
}
```

O(n)

顺序表：O(1)

```
/* linear_list_L.c 的实现 */

/* 取表中元素 */
Status GetElem(LinkList L, int i, ElemType *e)
{
    LinkList p=L->next; //指向首元结点
    int pos = 1;        //初始位置为1

    /* 链表不为NULL 且 未到第i个元素 */
    while(p!=NULL && pos<i) {
        p=p->next;
        pos++;
        }

    if (!p || pos>i)
        return ERROR;

    *e = p->data;
    return OK;
}
```

O(n)

顺序表：O(1)

```
/* linear_list_L.c 的实现 */

/* 查找符合指定条件的元素 */
int LocateElem(LinkList L, ElemType e, Status (*compare)(ElemType e1, ElemType e2))
{
    LinkList p = L->next;  //首元结点
    int pos = 1;           //初始位置

    /* 循环整个链表  */
    while(p && (*compare)(e, p->data)==FALSE) {
        p=p->next;
        pos++;
        }

    return p ? pos:0;
}
```

O(n)

顺序表：O(n)

```
/* linear_list_L.c 的实现 */

/* 查找符合指定条件的元素的前驱元素 */
Status PriorElem(LinkList L, ElemType cur_e, ElemType *pre_e)
{
    LinkList p = L->next;  //指向首元结点
    if (p==NULL)  //空表直接返回
        return ERROR;

    /* 从第2个结点开始循环整个链表(如果比较相等，保证有前驱) */
    while(p->next && p->next->data != cur_e)
        p = p->next;

    if (p->next==NULL) //未找到
        return ERROR;

    *pre_e = p->data;
    return OK;
}
```

O(n)

顺序表：O(n)

```
/* linear_list_L.c 的实现 */

/* 查找符合指定条件的元素的后继元素 */
Status NextElem(LinkList L, ElemType cur_e, ElemType *next_e)
{
    LinkList p = L->next;  //首元结点

    if (p==NULL)  //空表直接返回
        return ERROR;
    /* 有后继结点且当前结点值不等时继续 */
    while(p->next && p->data!=cur_e)
        p = p->next;

    if (p->next==NULL)
        return ERROR;
    *next_e = p->next->data;
    return OK;
}
```

O(n)

顺序表：O(n)

```
/* linear_list_L.c 的实现 */
/* 在指定位置前插入一个新元素 */
Status ListInsert(LinkList *L, int i, ElemType e)
{
    LinkList s, p = *L;  //p指向头结点
    int      pos = 0;
    /* 寻找第i-1个结点 */
    while(p && pos<i-1) {
        p=p->next;
        pos++;
        }
    if (p==NULL || pos>i-1)  //i值非法则返回，同GetElem
        return ERROR;
    //执行到此表示找到指定位置，p指向第i-1个结点
    s = (LinkList)malloc(sizeof(LNode)); //新申请1个结点
    if (s==NULL)
        return OVERFLOW;
    s->data = e; //新结点数据域赋值
    s->next = p->next; //新结点的next是第i个
    p->next = s;       //第i-1个的next是新结点
    return OK;
}
```

O(n)

顺序表：O(n)

两者虽然大O相同，但链表是查找n，顺序是移动n，基本操作的效率不同

这两句顺序不能反

```
/* linear_list_L.c 的实现 */
/* 删除指定位置的元素，并将被删除元素的值放入e中返回 */
Status ListDelete(LinkList *L, int i, ElemType *e)
{
    LinkList q, p = *L; //p指向头结点
    int      pos = 0;
    /* 寻找第i个结点 */
    while(p->next && pos<i-1) {
        p=p->next;
        pos++;
        }

    if (p->next==NULL || pos>i-1) //i值非法则返回，同GetElem
        return ERROR;
    //执行到此表示找到了第i个结点，此时p指向第i-1个结点
    q = p->next;          //q指向第i个结点
    p->next = q->next; //第i-1个结点的next域指向第i+1个

    *e = q->data;         //取第i个结点的值
    free(q);              //释放第i个结点
    return OK;
}
```

两者虽然大O相同，但链表是查找n，顺序是移动n，基本操作的效率不同

O(n)

顺序表：O(n)

```
/* linear_list_L.c 的实现 */

/* 遍历线性表 */
Status ListTraverse(LinkList L, Status (*visit)(ElemType e))
{   extern int line_count; //main中定义的换行计数器(与算法无关)
    LinkList p = L->next;  //指向首元

    line_count = 0;                    //计数器恢复初始值（与算法无关）
    while(p && (*visit)(p->data)==TRUE)
        p=p->next;

    if (p)
        return ERROR;

    printf("\n");//最后打印一个换行，只是为了好看，与算法无关
    return OK;
}
```

O(n)

顺序表：O(n)

§2. 线性表

2.3. 线性表的链式表示和实现

2.3.4. 线性表的复杂操作(链式表示)

2.3.4.1. 集合的并操作(P.20 例2-1 的链式实现)

```
void union_L(LinkList &La, LinkList Lb)
{
    LinkList p, q, r;
    for(q=Lb->next; q; q=q->next) {
        for(p=La->next; p; p=p->next)
            if (p->data == q->data)
                break;
        if (!p) { //表示La中无此元素(q->data)
            r = (LinkList)malloc(sizeof(LNode));
            if (!r)
                exit(OVERFLOW);
            r->data  = q->data;    //将q的数据域复制到r中
            r->next  = La->next;   //链式结构，插入首元效率最高
            La->next = r;
            }
        }
}
```

思考：目前是将Lb中要归并的元素复制后插入La中，因此算法完成后Lb不变

若要求将Lb中要归并的元素直接插入到La中，其余元素释放，即算法完成后Lb被销毁，应该如何实现？

§2. 线性表

2.3. 线性表的链式表示和实现

2.3.4. 线性表的复杂操作(链式表示)

2.3.4.2. 线性表的有序归并(P.20 例2-2 的链式实现)

★ 假设要求La，Lb归并到Lc中，完成后La/Lb不再存在

```
void MergeList_L(LinkList &La, LinkList &Lb, LinkList &Lc)
{
    LinkList pa = La->next, pb = Lb->next, pc;
    Lc = pc = La;       //借用La的头结点做Lc的头结点(pa已指向La->next)
    while (pa && pb) {
        if (pa->data <= pb->data) {
            pc->next = pa;  //将pa所指结点直接接在Lc的最后
            pc = pa;        //pc指向Lc的最后
            pa = pa->next;  //pa移向下一个结点
            }
        else {
            pc->next = pb;  //将pb所指结点直接接在Lc的最后
            pc = pb;         //pc指向Lc的最后
            pb = pb->next;   //pb移向下一个结点
            }
        }
    pc->next = pa ? pa : pb;  //将pa/pb中剩余段直接接在Lc的最后
    free(Lb);
}
```

时间复杂度与顺序表相同 O(m+n)
但空间复杂度小

思考：若要求La/Lb的元素复制后加入Lc，
即结束后La/Lb仍然存在，应该如何实现？

§2. 线性表

2.3. 线性表的链式表示和实现

2.3.4. 线性表的复杂操作(链式表示)

2.3.4.3. 头插法建立带头结点的单链表(P.30算法2.11)

```
void CreateList_L(LinkList &L, int n)
{    //建立头结点
     L = new LNode;
     L->next = NULL;
     //循环从键盘读入数据并插入n个结点
     for (i=n; i>0; i--) {
         p = new LNode;
         cin >> p->data;  //若scanf需加&
         p->next = L->next;  //插入在头部
         L->next = p;
         }
}
```

§2. 线性表

2.3. 线性表的链式表示和实现

2.3.4. 线性表的复杂操作(链式表示)

2.3.4.4. 尾插法建立带头结点的单链表

思考：哪里仍有改进余地？

```
void CreateList_L_tail(LinkList &L, int n)
{    //建立头结点
     tp = L = new LNode;
     L->next = NULL;
     //循环从键盘读入数据并插入n个结点
     for (i=1; i<=n; i++) {
         p = new LNode;
         cin >> p->data;  //若scanf需加&
         p->next = NULL;  //因为插在最后，next置NULL
         tp->next = p;    //插入在尾部
         tp = p;          //指向新的尾结点
         }
}
```

尾插法要注意，头指针不能丢

§2. 线性表

2.3. 线性表的链式表示和实现

2.3.4. 线性表的复杂操作(链式表示)

2.3.4.4. 尾插法建立带头结点的单链表

```
void CreateList_L_tail(LinkList &L, int n)
{    //建立头结点
     tp = L = new LNode;
     L->next = NULL;
     //循环从键盘读入数据并插入n个结点
     for (i=1; i<=n; i++) {
         p = new LNode;
         cin >> p->data;   //若scanf需加&
         p->next = NULL;   //因为插在最后，next置NULL
         tp->next = p;     //插入在尾部
         tp = p;           //指向新的尾结点
         }
     tp->next = NULL;
}
```

尾插法要注意，头指针不能丢

改进,循环内不做无意义的赋值,效率高

§2. 线性表

2.3. 线性表的链式表示和实现

2.3.5. 静态链表的使用（适用于无动态内存申请的语言，此处仅简单示意，具体操作略）

★ 基本概念

将数据元素存储在数组中，但不以数组的下标来表示元素之间的逻辑关系，而是采用一个游标代替指针来表示结点之间的逻辑关系

下标	数据域	游标
0		1
1	张三	4
2		-1
3	王五	6
4	李四	3
5		-1
6	赵六	0
7		-1

当前状态：
张三->李四->王五->赵六

假设[0]不用，表示头结点
则 [0].cur表示首元的下标

cur = 0：表示尾元素
cur = -1：空闲空间

下标	数据域	游标
0		1
1	张三	7
2		-1
3	王五	6
4	李四	3
5		-1
6	赵六	0
7	张三丰	4

当前状态：
张三->张三丰->李四->王五->赵六

插入结点后游标值的修改

若找不到游标为-1的数组下标
则表示链表已满

下标	数据域	游标
0		1
1	张三	4
2		-1
3	王五	-1
4	李四	6
5		-1
6	赵六	0
7		-1

当前状态：
张三->李四->赵六

删除结点后游标值的修改

若[0].cur==0，表示链表空

出于效率的考虑，数据域可以
不清除(下次插入时会覆盖)

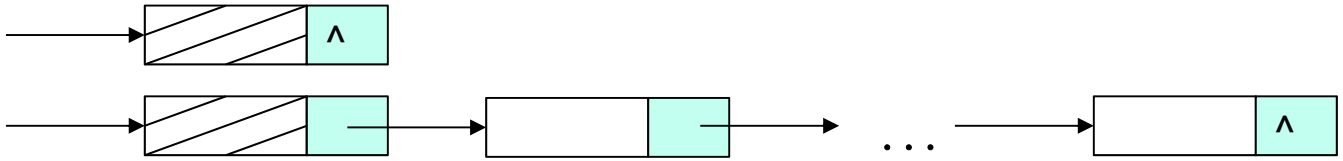
§2. 线性表

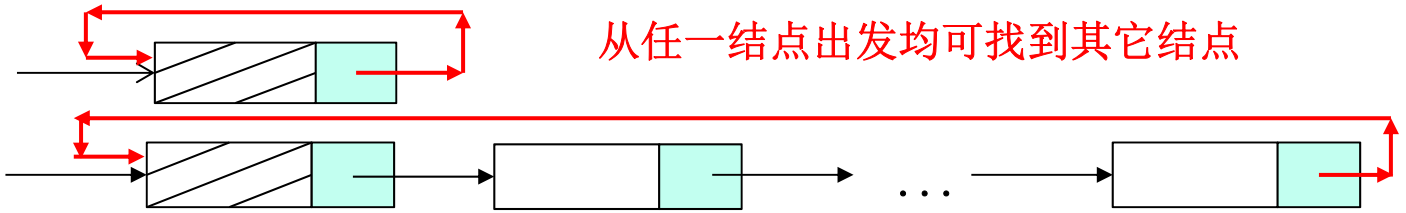
2.3. 线性表的链式表示和实现

2.3.6. 循环链表的使用

★ 特点

最后一个结点的指针域指向 { 头结点(带头结点)
首元结点(不带头结点)





★ 使用

基本方法同单链表，仅判断到达尾部的条件不同

```
if (p)                  if (p!=L)
while(p->next)          while(p->next!=L)
```

★ 具体实现

```
ElemType => int (带头结点)
```

```
/* linear_list_L.h 的组成 */
#define TRUE          1
#define FALSE         0
#define OK            1
#define ERROR         0
#define INFEASIBLE   -1
#define OVERFLOW     -2

typedef int Status;

typedef int ElemType;       //可根据需要修改元素的类型

typedef struct LNode {
    ElemType       data;    //存放数据
    struct LNode *next;     //存放直接后继的指针
} LNode, *LinkList;
```

P.10 预定义常量和类型

P.28 形式定义

无变化

/* linear_list_L.h 的组成 */

```
Status    InitList(LinkList *L);
Status    DestroyList(LinkList *L);
Status    ClearList(LinkList *L);
Status    ListEmpty(LinkList L);
int       ListLength(LinkList L);
Status    GetElem(LinkList L, int i, ElemType *e);
int       LocateElem(LinkList L, ElemType e,
                                 Status (*compare)(ElemType e1, ElemType e2));
Status    PriorElem(LinkList L, ElemType cur_e, ElemType *pre_e);
Status    NextElem(LinkList L, ElemType cur_e, ElemType *next_e);
Status    ListInsert(LinkList *L, int i, ElemType e);
Status    ListDelete(LinkList *L, int i, ElemType *e);
Status    ListTraverse(LinkList L, Status (*visit)(ElemType e));
```

无变化

将顺序实现中 sqlist => LinkList，其它未变，具体说明、注意等与顺序实现相同

```
/* linear_list_L.c 的实现 */
#include <stdio.h>
#include <stdlib.h>                          //malloc/realloc函数
#include <unistd.h>                          //exit函数
#include "linear_list_L.h"          //形式定义

/* 初始化线性表 */
Status InitList(LinkList *L)
{
    /* 申请头结点空间，赋值给头指针 */
    *L = (LNode *)malloc(sizeof(LNode));
    if (*L==NULL)
        exit(OVERFLOW);
    (*L)->next = (*L);      //头结点的next指向自己
    return OK;
}
```

```
/* linear_list_L.c 的实现 */

/* 删除线性表 */
Status DestroyList(LinkList *L)
{
    LinkList q, p = *L;

    /* 整个链表(含头结点)依次释放
       不能用while循环 */
    do {
        q=p->next; //抓住链表的下一个结点，若空表则q直接为NULL
        free(p);
        p=q;
    } while(p!=(*L));    //若链表为空，则循环不执行

    *L=NULL;     //头指针置NULL，可不要
    return OK;
}
```

```
/* 整个链表(含头结点)依次释放 */
while(p) {
    q=p->next;
    free(p);
    p=q;
    }
```

单链表

```
/* linear_list_L.c 的实现 */

/* 清除线性表（保留头结点） */
Status ClearList(LinkList *L)
{
    LinkList q, p = (*L)->next;

    /* 从首元结点开始依次释放 */
    while(p!=(*L)) {
        q = p->next;    //抓住链表的下一个结点
        free(p);
        p = q;
        }

    (*L)->next = (*L); //头结点的next域置L
    return OK;
}
```

DestroyList也可以用本方法先把其它结点全部释放，再单独处理头结点

```
/* linear_list_L.c 的实现 */

/* 判断是否为空表 */
Status ListEmpty(LinkList L)
{
    /* 判断头结点的next域即可 */
    if (L->next==L)
        return TRUE;
    else
        return FALSE;
}
```

```
/* linear_list_L.c 的实现 */

/* 求表的长度 */
int ListLength(LinkList L)
{
    LinkList p = L->next; //指向首元结点
    int len=0;

    /* 循环整个链表，进行计数 */
    while(p!=L) {
        p = p->next;
        len++;
        }

    return len;
}
```

```
/* linear_list_L.c 的实现 */

/* 取表中元素 */
Status GetElem(LinkList L, int i, ElemType *e)
{
    LinkList p=L->next; //指向首元结点
    int pos = 1;        //初始位置为1
    /* 链表不为NULL 且 未到第i个元素 */
    while(p!=L && pos<i) {
        p=p->next;
        pos++;
        }

    if (p==L || pos>i)
        return ERROR;

    *e = p->data;
    return OK;
}
```

```
/* linear_list_L.c 的实现 */

/* 查找符合指定条件的元素 */
int LocateElem(LinkList L, ElemType e, Status (*compare)(ElemType e1, ElemType e2))
{
    LinkList p = L->next;  //首元结点(如果是空表则值就是L)
    int pos = 1;           //初始位置

    /* 循环整个链表  */
    while(p!=L && (*compare)(e, p->data)==FALSE) {
        p=p->next;
        pos++;
        }

    return (p!=L) ? pos:0;
}
```

```
/* linear_list_L.c 的实现 */

/* 查找符合指定条件的元素的前驱元素 */
Status PriorElem(LinkList L, ElemType cur_e, ElemType *pre_e)
{
    LinkList p = L->next;  //指向首元结点
    if (p==L)      //空表直接返回
        return ERROR;

    /* 从第2个结点开始循环整个链表(如果比较相等，保证有前驱) */
    while(p->next!=L && p->next->data != cur_e)
        p = p->next;

    if (p->next==L) //未找到
        return ERROR;

    *pre_e = p->data;
    return OK;
}
```

```
/* linear_list_L.c 的实现 */

/* 查找符合指定条件的元素的后继元素 */
Status NextElem(LinkList L, ElemType cur_e, ElemType *next_e)
{
    LinkList p = L->next;  //首元结点

    if (p==L)      //空表直接返回
        return ERROR;
    /* 有后继结点且当前结点值不等时继续 */
    while(p->next!=L && p->data!=cur_e)
        p = p->next;

    if (p->next==L)
        return ERROR;
    *next_e = p->next->data;
    return OK;
}
```

```
/* linear_list_L.c 的实现 */
/* 在指定位置前插入一个新元素 */
Status ListInsert(LinkList *L, int i, ElemType e)
{
    LinkList s, p = *L;  //p指向头结点
    int      pos = 0;
    if (i==1)  //插入首元前单独处理
        goto INSERT; //无条件跳转
    /* 寻找第i-1个结点 */
    do {
        p=p->next;
        pos++;
    } while(p!=(*L) && pos<i-1);

    if (p==(*L) || pos>i-1)  //i值非法则返回，同GetElem
        return ERROR;

    //执行到此表示找到指定位置，p指向第i-1个结点
INSERT:
    s = (LinkList)malloc(sizeof(LNode)); //新申请1个结点
    if (s==NULL)
        return OVERFLOW;

    s->data = e; //新结点数据域赋值
    s->next = p->next; //新结点的next是第i个
    p->next = s;       //第i-1个的next是新结点

    return OK;
}
```

```
/* 寻找第i-1个结点 */
while(p && pos<i-1) {
    p=p->next;
    pos++;
   }
```

单链表

```
/* linear_list_L.c 的实现 */
/* 删除指定位置的元素，并将被删除元素的值放入e中返回 */
Status ListDelete(LinkList *L, int i, ElemType *e)
{   LinkList q, p = *L; //p指向头结点
    int      pos = 0;

    /* 寻找第i个结点（p->next是第i个结点） */
    while(p->next!=(*L) && pos<i-1) {
          p=p->next;
          pos++;
          }

    if (p->next==(*L) || pos>i-1) //i值非法则返回，同GetElem
          return ERROR;
    //执行到此表示找到了第i个结点，此时p指向第i-1个结点
    q = p->next;         //q指向第i个结点
    p->next = q->next; //第i-1个结点的next域指向第i+1个

    *e = q->data;        //取第i个结点的值
    free(q);             //释放第i个结点
    return OK;
}
```

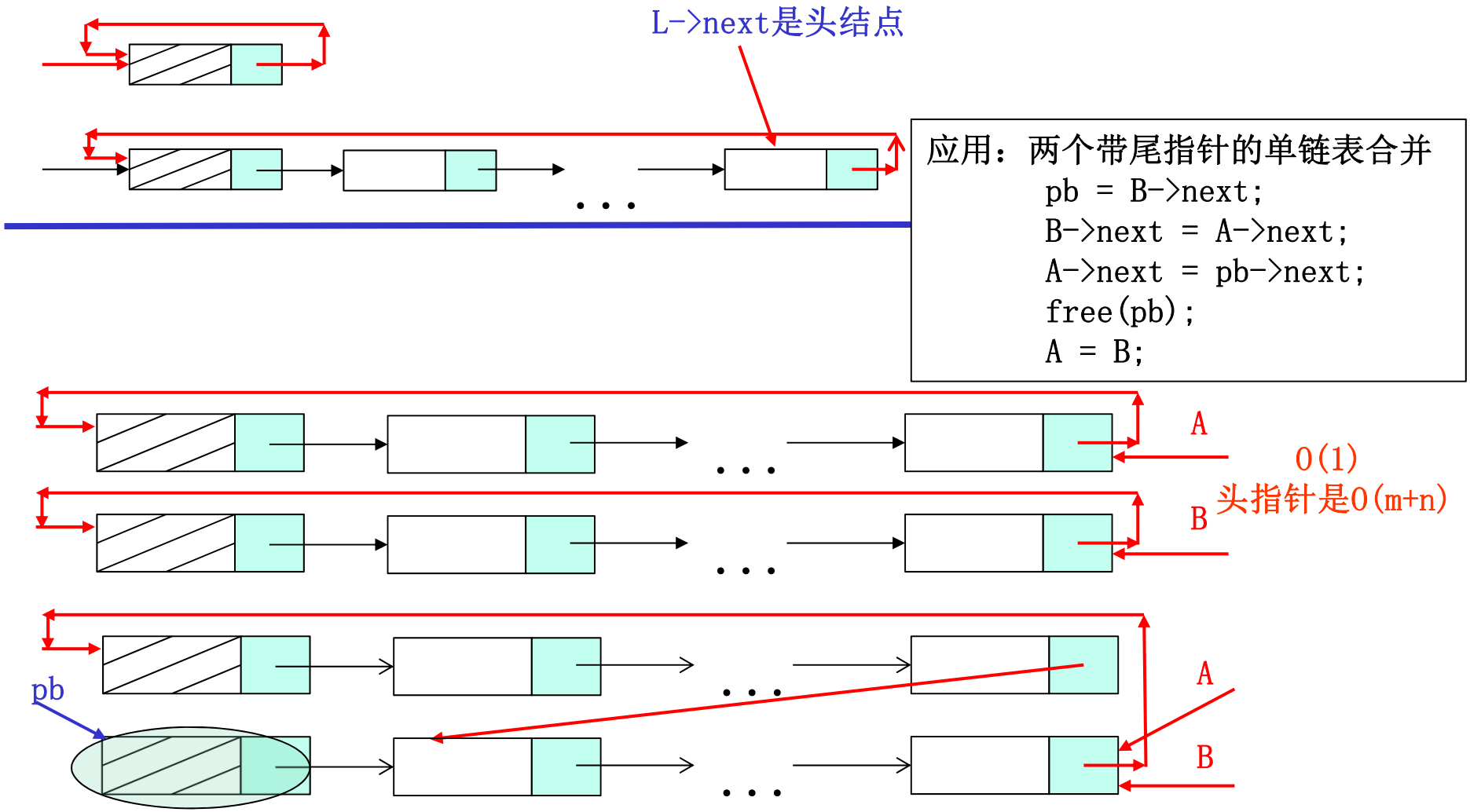
```
/* linear_list_L.c 的实现 */

/* 遍历线性表 */
Status ListTraverse(LinkList L, Status (*visit)(ElemType e))
{
    extern int line_count; //main中定义的换行计数器(与算法无关)
    LinkList p = L->next;  //指向首元

    line_count = 0;                    //计数器恢复初始值（与算法无关）
    while(p!=L && (*visit)(p->data)==TRUE)
        p=p->next;

    if (p!=L)
        return ERROR;

    printf("\n");//最后打印一个换行，只是为了好看，与算法无关
    return OK;
}
```

§2. 线性表

2.3. 线性表的链式表示和实现

2.3.6. 循环链表的使用

★ 特殊形式：带尾指针的循环链表



§2. 线性表

2.3. 线性表的链式表示和实现

2.3.7. 双向链表的使用

★ 单链表的缺陷

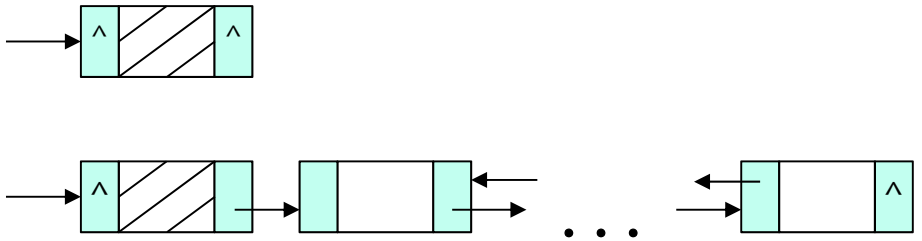
求后继易，求前驱难

例：假设头指针L，若已知q指向某结点，找q的前驱

```
p = L;
while(p && p->next!=q)      O(n)
    p=p->next;
```

★ 双向链表存储结构的描述(P.35-36)

```
typedef struct DuLNode {
    ElemType        data;
    struct DuLNode *prior;   //前驱指针
    struct DuLNode *next;    //后继指针
} DuLNode, *DuLinkList;
```



★ 具体实现

ElemType => int (带头结点)

```
/* linear_list_DL.h 的组成 */
#define TRUE          1
#define FALSE         0
#define OK            1
#define ERROR         0
#define INFEASIBLE   -1
#define OVERFLOW     -2

typedef int Status;

typedef int ElemType;       //可根据需要修改元素的类型
```

P.10 预定义常量和类型 无变化

```
typedef struct DuLNode {
    ElemType        data;  //存放数据
    struct DuLNode *prior; //存放直接前驱的指针
    struct DuLNode *next;  //存放直接后继的指针
} DuLNode, *DuLinkList;
```

P.35-36 形式定义

```
/* linear_list_DL.h 的组成 */
Status   InitList(DuLinkList *L);
Status   DestroyList(DuLinkList *L);
Status   ClearList(DuLinkList *L);
Status   ListEmpty(DuLinkList L);
int      ListLength(DuLinkList L);
Status   GetElem(DuLinkList L, int i, ElemType *e);
int      LocateElem(DuLinkList L, ElemType e,
                    Status (*compare)(ElemType e1, ElemType e2));
Status   PriorElem(DuLinkList L, ElemType cur_e, ElemType *pre_e);
Status   NextElem(DuLinkList L, ElemType cur_e, ElemType *next_e);
Status   ListInsert(DuLinkList *L, int i, ElemType e);
Status   ListDelete(DuLinkList *L, int i, ElemType *e);
Status   ListTraverse(DuLinkList L, Status (*visit)(ElemType e));
```

将单链表的 LinkList => DuLinkList，其它未变

```c
/* linear_list_DL.c 的实现 */
#include <stdio.h>
#include <stdlib.h>                    //malloc/realloc函数
#include <unistd.h>                    //exit函数
#include "linear_list_DL.h"            //形式定义

/* 初始化线性表 */
Status InitList(DuLinkList *L)
{
    /* 申请头结点空间，赋值给头指针 */
    *L = (DuLNode *)malloc(sizeof(DuLNode));
    if (*L==NULL)
        exit(OVERFLOW);
    (*L)->prior = NULL;
    (*L)->next = NULL;
    return OK;
}
```

后续函数中，这些差异不再标出

```
/* linear_list_DL.c 的实现 */

/* 删除线性表 */
Status DestroyList(DuLinkList *L)
{
    DuLinkList q, p = *L;

    /* 从头结点开始依次释放（含头结点） */
    while(p) {    //若链表为空，则循环不执行
        q=p->next; //抓住链表的下一个结点
        free(p);
        p=q;
        }

    *L=NULL;       //头指针置NULL
    return OK;
}
```

除换为DuLinkList外，
结构无变化

```
/* linear_list_DL.c 的实现 */

/* 清除线性表（保留头结点） */
Status ClearList(DuLinkList *L)
{
    DuLinkList q, p = (*L)->next;

    /* 从首元结点开始依次释放 */
    while(p) {
        q = p->next;    //抓住链表的下一个结点
        free(p);
        p = q;
        }
    (*L)->prior = NULL; //头结点的prior域置NULL
    (*L)->next = NULL; //头结点的next域置NULL
    return OK;
}
```

```
/* linear_list_DL.c 的实现 */

/* 判断是否为空表 */
Status ListEmpty(DuLinkList L)
{
    /* 判断头结点的next域即可 */
    if (L->next==NULL)
        return TRUE;
    else
        return FALSE;
}
```

除换为DuLinkList外，
结构无变化

```
/* linear_list_DL.c 的实现 */

/* 求表的长度 */
int ListLength(DuLinkList L)
{
    DuLinkList p = L->next; //指向首元结点
    int len=0;

    /* 循环整个链表，进行计数 */
    while(p) {
         p = p->next;
         len++;
         }

    return len;
}
```

除换为DuLinkList外，
结构无变化

除换为DuLinkList外，
结构无变化

```
/* linear_list_DL.c 的实现 */

/* 取表中元素 */
Status GetElem(DuLinkList L, int i, ElemType *e)
{
    DuLinkList p=L->next; //指向首元结点
    int pos = 1;          //初始位置为1
    /* 链表不为NULL 且 未到第i个元素 */
    while(p!=NULL && pos<i) {
        p=p->next;
        pos++;
        }

    if (!p || pos>i)
        return ERROR;

    *e = p->data;
    return OK;
}
```

除换为DuLinkList外，
结构无变化

```
/* linear_list_DL.c 的实现 */

/* 查找符合指定条件的元素 */
int LocateElem(DuLinkList L, ElemType e, Status(*compare)(ElemType e1, ElemType e2))
{
    DuLinkList p = L->next;  //首元结点
    int pos = 1;             //初始位置

    /* 循环整个链表  */
    while(p && (*compare)(e, p->data)==FALSE) {
         p=p->next;
         pos++;
         }

    return p ? pos:0;
}
```

除换为DuLinkList外，
结构无变化

```
/* linear_list_DL.c 的实现 */

/* 查找符合指定条件的元素的前驱元素 */
Status PriorElem(DuLinkList L, ElemType cur_e, ElemType *pre_e)
{
    DuLinkList p = L->next;  //指向首元结点
    if (p==NULL)  //空表直接返回
        return ERROR;

    /* 从第2个结点开始循环整个链表(如果比较相等，保证有前驱) */
    while(p->next && p->next->data != cur_e)
        p = p->next;

    if (p->next==NULL) //未找到
        return ERROR;

    *pre_e = p->data;
    return OK;
}
```

除换为DuLinkList外，
结构无变化

```
/* linear_list_DL.c 的实现 */

/* 查找符合指定条件的元素的后继元素 */
Status NextElem(DuLinkList L, ElemType cur_e, ElemType *next_e)
{
    DuLinkList p = L->next;  //首元结点

    if (p==NULL)  //空表直接返回
        return ERROR;

    /* 有后继结点且当前结点值不等时继续 */
    while(p->next && p->data!=cur_e)
        p = p->next;

    if (p->next==NULL)
        return ERROR;
    *next_e = p->next->data;

    return OK;
}
```

```
/* linear_list_DL.c 的实现 */
/* 在指定位置前插入一个新元素 */
Status ListInsert(DuLinkList *L, int i, ElemType e)
{
    DuLinkList s, p = *L;  //p指向头结点
    int      pos = 0;

    /* 寻找第i-1个结点 */
    while(p && pos<i-1) {
            p=p->next;
            pos++;
            }

    if (p==NULL || pos>i-1)  //i值非法则返回，同GetElem
            return ERROR;

    //执行到此表示找到指定位置，p指向第i-1个结点
    s = (DuLinkList)malloc(sizeof(DuLNode)); //新申请1个结点
    if (s==NULL)
            return OVERFLOW;
    s->data = e; //新结点数据域赋值
    s->next = p->next; //新结点的next是第i个
    if (p->next) //第i个结点可能不存在，要先判断
        p->next->prior = s; //第i个结点的prior是s
    s->prior = p;       //s的前驱是p
    p->next = s;        //p的后继是s

    return OK;
}
```

查找第i-1个结点的操作
除换为DuLinkList外，
结构无变化

```
/* linear_list_DL.c 的实现 */
/* 删除指定位置的元素，并将被删除元素的值放入e中返回 */
Status ListDelete(LinkList *L, int i, ElemType *e)
{
    LinkList q, p = *L; //p指向头结点
    int      pos = 0;

    /* 寻找第i个结点（p->next是第i个结点） */
    while(p->next && pos<i-1) {
            p=p->next;
            pos++;
            }

    if (p->next==NULL || pos>i-1) //i值非法则返回，同GetElem
            return ERROR;

    //执行到此表示找到了第i个结点，此时p指向第i-1个结点
    q = p->next;        //q指向第i个结点
    p->next = q->next; //第i-1个结点的next域指向第i+1个
    if (q->next) //第i+1个可能是NULL，先判断
        q->next->prior = p;
    *e = q->data;       //取第i个结点的值
    free(q);            //释放第i个结点

    return OK;
}
```

查找第i个结点的操作除换为DuLinkList外，结构无变化

```
/* linear_list_DL.c 的实现 */

/* 遍历线性表 */
Status ListTraverse(DuLinkList L, Status (*visit)(ElemType e))
{
    extern int line_count; //main中定义的换行计数器(与算法无关)
    DuLinkList p = L->next;  //指向首元

    line_count = 0;                    //计数器恢复初始值（与算法无关）
    while(p && (*visit)(p->data)==TRUE)
        p=p->next;




    if (p)
        return ERROR;

    printf("\n");//最后打印一个换行，只是为了好看，与算法无关
    return OK;
}
```

除换为DuLinkList外，
结构无变化

/* linear_list_DL.c 的实现 */

除换为DuLinkList外，
结构无变化

/* 遍历线性表 */

```
Status ListTraverse(DuLinkList L, Status (*visit)(ElemType e))
{
    extern int line_count; //main中定义的换行计数器(与算法无关)
    DuLinkList p = L->next;  //指向首元

    line_count = 0;                    //计数器恢复初始值（与算法无关）
```

逆序输出

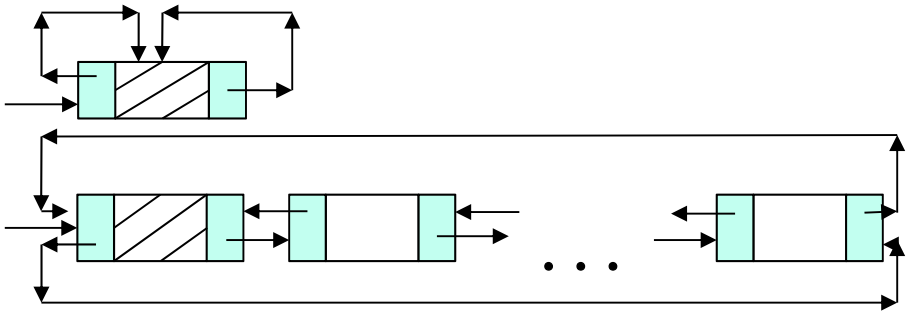
```
    while(p->next) //同单链表
        p=p->next;

    /* while执行后，p指向最后一个结点，再循环向前(忽略头结点) */
    while(p && p->prior && (*visit)(p->data)==TRUE)
        p=p->prior;

    if (p->prior) //正常情况下，p->prior==NULL 表示p指向了头结点，否则表示出错
         return ERROR;

    printf("\n");//最后打印一个换行，只是为了好看，与算法无关
    return OK;
}
```

§2. 线性表

2.3. 线性表的链式表示和实现

2.3.7. 双向链表的使用

★ 双向循环链表的使用



关于 P.36 算法2.18（适用于双向循环链表）

```
Status ListInsert_DuL(DuLinkList &L, int i, ElemType e)
{
    if (!(p=GetElemP_DuL(L, i)))
        return ERROR;

    if (!(s=(DuLinkList)malloc(sizeof(DuLNode))))
        return ERROR;

    s->data = e;
    s->prior = p->prior;
    p->prior->next = s;
    s->next = p;
    p->prior = s;
    return OK;
}
```

1≤i ≤length时，
返回指向第i个结点的指针
i=length+1时，返头指针

p指向在第i个结点，
在p前面插入新结点

问：为什么不适用于双向链表？

问：如何改进才能适用？

关于 P.37 算法2.19（适用于双向循环链表）

```
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;
}
```

1≤i ≤length时，
返回指向第i个结点的指针

问：为什么不适用于双向链表？

问：如何改进才能适用？

§2. 线性表

2.3. 线性表的链式表示和实现

2.3.8. 从实际应用出发定义的线性链表及基本操作

P. 37 - 38

§2. 线性表

0	1
1	3
2	2
3	3
4	6
5	0
6	0
7	0
8	15

0	1	0
1	3	1
2	2	2
3	3	3
4	6	4
5	15	8

2.4. 一元多项式的表示及相加

2.4.1. 一元多项式在计算机中的表示

★ 一元n次多项式的数学表示

$P_n(x)=p_0+p_1x+p_2x^2+...+p_nx^n$ （共n+1项）

其中第i项(0≤i≤n)的系数为P_i，指数为x^i

★ 一元n次多项式在计算机内的表示

线性表$P=(p_0, p_1, p_2, ... p_n)$，将指数隐含在序号中

例如：P=(1, 3, 2, 3, 6, 0, 0, 0, 15)

表示：$1+3x+2x^2+3x^3+6x^4+15x^8$

★ 一元n次多项式的存储(以 $1+3x+2x^2+3x^3+6x^4+15x^8$ 为例)

- 方法1：采用顺序结构的数组存储，数组的值为系数，数组下标为指数

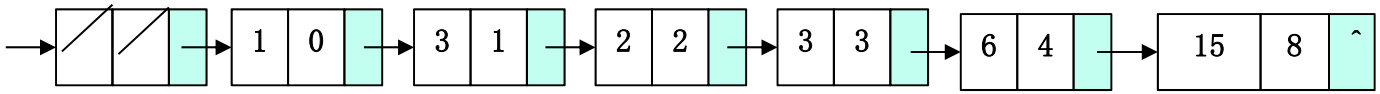
  缺点：当n很大，非零系数很少时浪费大量空间

  例如：$1+X^{10000}$

- 方法2：仍采用数组，每个元素两项，分别表示指数和系数

  缺点：数组有多少项不易确定

- 方法3：采用链表，每个结点两个数据项，分别表示指数和系数，一个next指针



★ 存储结构的定义(改P.42，仿链表定义)

```
typedef struct polynode {
    double    coef; //系数，可能是小数
    int       expn; //指数，一定是整数
    polynode *next;
} polynode, *polynomial;
```

§2. 线性表

2.4. 一元多项式的表示及相加

2.4.2. 一元多项式的相加

两个多项式P、Q分别有n和m项，表示为：

$P=(p_0, p_1, p_2, \ldots p_n)$

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

不失一般性，假设m<n，则两个多项式相加结果为:

$R=P+Q=(p_0+q_0, p_1+q_1, p_2+q_2, \ldots, p_m+q_m, p_m+1, \ldots p_n)$

★ 多项式相加的基本规则：

- 指数相同则系数相加，若系数相加之和为0，则该项删除
- 指数不同，则小的优先进入和多项式

  推论：如果链表无序，则每次从线性表中取一个元素，都要遍历另一个线性表才能查找指数相同项，因此保证线性表按指数递增排列可以提高效率

  => 两个有序线性表的归并，结果仍有序

§2. 线性表

★ 多项式相加的基本规则（三种情况）

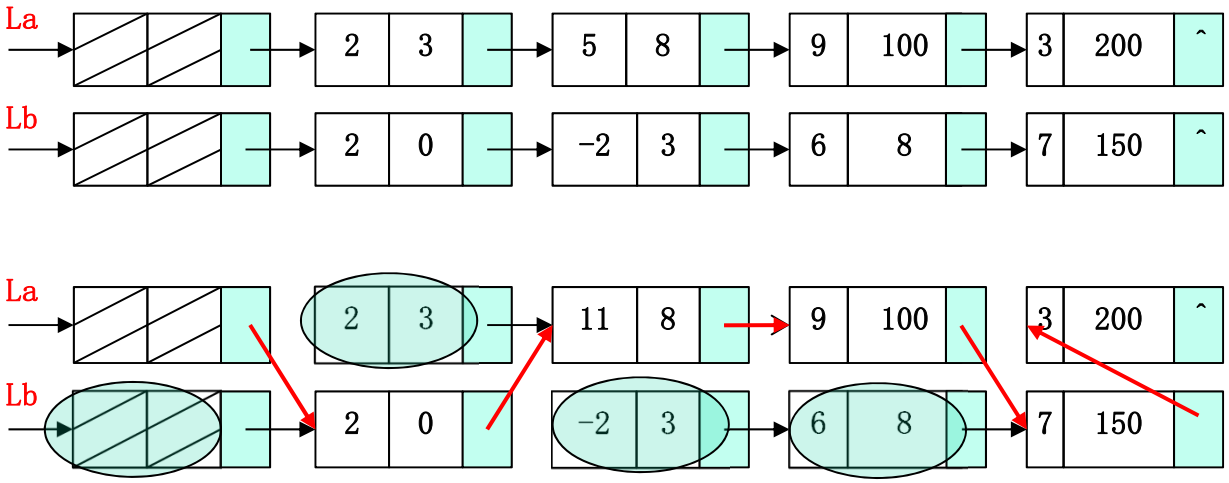
假设p、q分别指向表示多项式的有序链表La、Lb,通过将La、Lb归并(Lb插入La中)来完成多项式的相加

pre = La; （因为插入在pa前，因此要前驱指针）

p = La->next;

q = Lb->next;

分三种情况讨论：



① p->expn < q->expn （p指针后移）

```
pre = p;
p = p->next;
```

pre和p同步移动，保持相对位置不变

② p->expn == q->expn （合并后释放q，还可能释放p）

```
x = p->coef + q->coef;
if (fabs(x)>1e-6) {    //合并后系数不为0
    p->coef = x;
    pre = p;
    }
else {  //合并后系数为0
    pre->next = p->next;
    free(p);
    }
p = pre->next;
```

更新合并后的coef域
pre后移

pre不变，pre->next指向下一个

保持pre和p的
相对位置不变

//p后移

```
u = q;
q = q->next;
free(u);
```

合并后，q结点已无用
释放q结点

③ p->expn > q->expn （q插入在p前，pre后）

```
u = q->next;
q->next = p;     //q成为p的前驱
pre->next = q;   //插入pre的后面（p的前面）
pre = q;         //pre后移（p不变），仍保持相对位置不变
q = u;
```