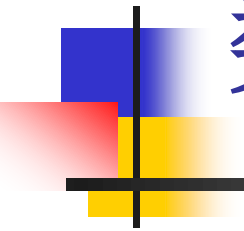
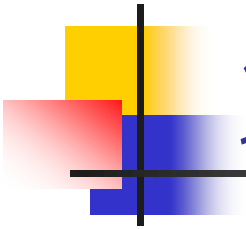


数据结构——线性表

第二章　线性表

重点：顺序表和链表上各种基本算法的实现及相关的时间性能分析

难点：线性表应用的算法设计



第二章 线性表

2.1 线性表的类型定义

- 定义

  $n(\geq 0)$个数据元素的有限序列,记作$(a_1, \ldots a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$其中，$a_i$是表中数据元素，$n$ 为表长 ($n=0$时称为空表)

- 逻辑特征

  $n>0$时

  - 有且仅有一个“第一个”数据元素
  - 有且仅有一个“最后一个”数据元素
  - 除第一个数据元素外，其它元素有且仅有一个直接前驱
  - 除最后一个数据元素外，其它元素有且仅有一个直接后继

2.1 线性表的类型定义-ADT List

- **ADT List定义**

ADT List{

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

数据关系：R={R1},R1={<ai-1,ai>|ai-1,ai∈D, i=2,3,…,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：将线性表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：返回线性表L中数据元素的个数

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( )是数据元素的判定函数

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。
若这样的元素不存在，则返回值为0

PriorElem(L, cur_e ,&pre_e)

初始条件：线性表L已存在

操作结果：若cur_e是L的数据元素，且不是第一个，则用pre_e返回它的前驱，
否则操作失败，pre_e无定义

NextElem(L, cur_e ,&next_e)

初始条件：线性表L已存在

操作结果：若cur_e是L的数据元素，且不是最后一个，则用next_e返回它的后继，
否则操作失败，next_e无定义

ListInsert(&L, i, e)

初始条件：线性表L已存在，1≤i≤ListLength(L)+1

操作结果：在L中第i个位置之前插入新的数据元素e，L的长度加1

ListDelete(&L, i, &e)

初始条件：线性表L已存在，1≤i≤ListLength(L)

操作结果：删除L的第i个数据元素，并用e返回其值，L的长度减1

ListTraverse(L, visit( ))

初始条件：线性表L已存在

操作结果：依次对L的每个数据元素调用函数visit( )。一旦visit( )失败，则操作失败

}ADT List

2.1 线性表的类型定义-ADT List

- **ADT List定义**
  - **P19**
    基本操作
    - 初始化、销毁
    - 查询：个体、整体
    - 动态变化：插入、删除
  - 重点掌握以下三个基本操作
    - **GetElem(L, i ,&e)**
    - **ListInsert(&L, i, e)**
    - **ListDelete(&L, i, &e)**

    注意: 接口的形式可以略有变化,如 **e = GetElem(L, i)**

2.1 线性表的类型定义-例2-1

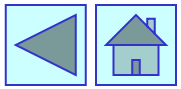
- 基于ADT List的算法设计
  - 例2-1 假设利用两个线性表La和Lb分别表示两个集合A和B（即：线性表中的数据元素即为集合中的成员），现要求一个新的集合A = A∪B。

    输入：线性表La、线性表Lb
    输出：变化了的La　　➔union(&La, Lb)
    处理方法：扩大线性表La，将存在于线性表Lb中而不存在于线性表La中的数据元素插入到线性表La中去。
    步骤：
    1．从线性表Lb中依次取得每个数据元素；
    2．依值在线性表La中进行查访；
    3．若不存在，则插入之。
    算法：算法2.1 P20

ListLength
GetElem

LocateElem

ListInsert



2.1 线性表的类型定义-例2-1

```
void union(List &La, List Lb) {
   // 将所有在线性表Lb中但不在La中的数据元素插入到La中
   La_len = ListLength(La);
   Lb_len = ListLength(Lb); // 求线性表的长度
   for (i = 1;  i <= Lb_len;  i++) {
      GetElem(Lb, i, e);        // 取Lb中第i个数据元素赋给e
      if(!LocateElem(La, e, equal))
         // La中不存在和e相同的元素，则插入之
         ListInsert(La, ++La_len, e);
   }
} // union
```

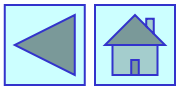
e是变参，调用时不加&符号

T(na，nb) =O(nb*na)，　na、nb表示La表和Lb表的长度

2.1 线性表的类型定义-例2-2

- 基于ADT List的算法设计
  - 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，要求将La和Lb归并成一个新的线性表Lc，且Lc中的数据元素仍按值非递减有序排列。

    输入：线性表La、线性表Lb(均按值非递减有序排列)
    输出：Lc(按值非递减有序排列)　　➔merge(La, Lb, &Lc)
    处理方法：∵La和Lb均按值非递减有序排列
    ∴设置两个位置指示器i和j，分别指向La和Lb中的某个元素，初始均指向第1个。Lc初始化为空表。
    比较i和j所指向的元素ai和bj，选取值小的插入到Lc的尾部，并使相应的位置指示器向后移。
    算法：算法2.2　P21



2.1 线性表的类型定义-例2-2

```
void MergeList(List La, List Lb, List &Lc) {
  // 已知线性表La和Lb中的元素按值非递减有序排列。
  // 归并La和Lb得到新的线性表Lc，Lc的元素按值非递减有序排列。
  InitList(Lc);
  i = j = 1;    k = 0;
  La_len = ListLength(La);
  Lb_len = ListLength(Lb);
  while ((i <= La_len) && (j <= Lb_len)) {
    // La和Lb均非空
    GetElem(La, i, ai);  GetElem(Lb, j, bj);
    if (ai <= bj) {
      ListInsert(Lc, ++k, ai);  ++i;
    }else {
      ListInsert(Lc, ++k, bj);  ++j; }
  }
```


2.1 线性表的类型定义-例2-2

```
    while (i <= La_len) {
        GetElem(La, i++, ai);
        ListInsert(Lc, ++k, ai);
    }
    while (j <= Lb_len) {
        GetElem(Lb, j++, bj);
        ListInsert(Lc, ++k, bj);
    }
} // MergeList
```

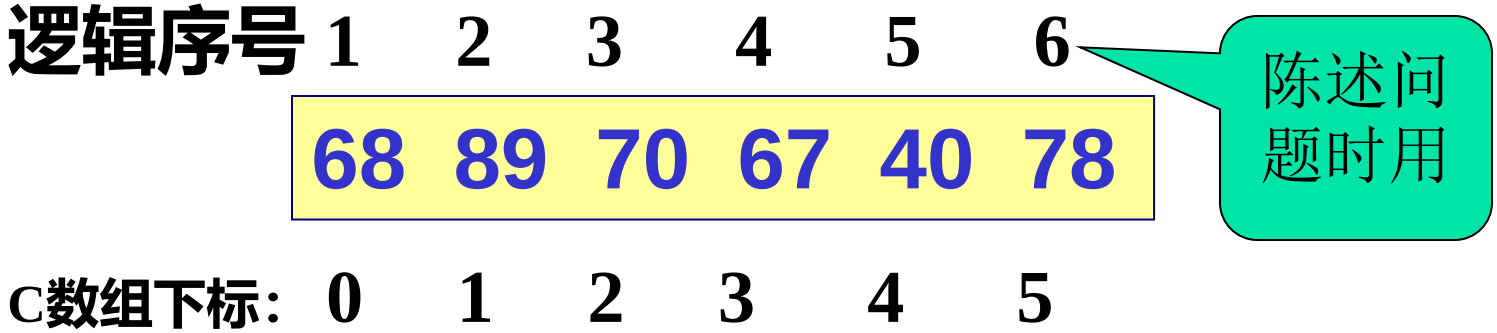
T(na, nb) =O(na+nb)

2.1 线性表的类型定义-结论

- 讨论
  - **不同的算法策略，时间复杂度通常也不同**
  - **输入线性表的特征会影响算法的策略（是否有序？）**
  - **输出线性表的特征也会影响算法的策略（是否要求有序？）**

2.2 线性表的顺序表示和实现

- 定义：**用一组地址连续的存储单元依次存储线性表中的数据元素**
- 特点：**逻辑上相邻，物理上也相邻。**



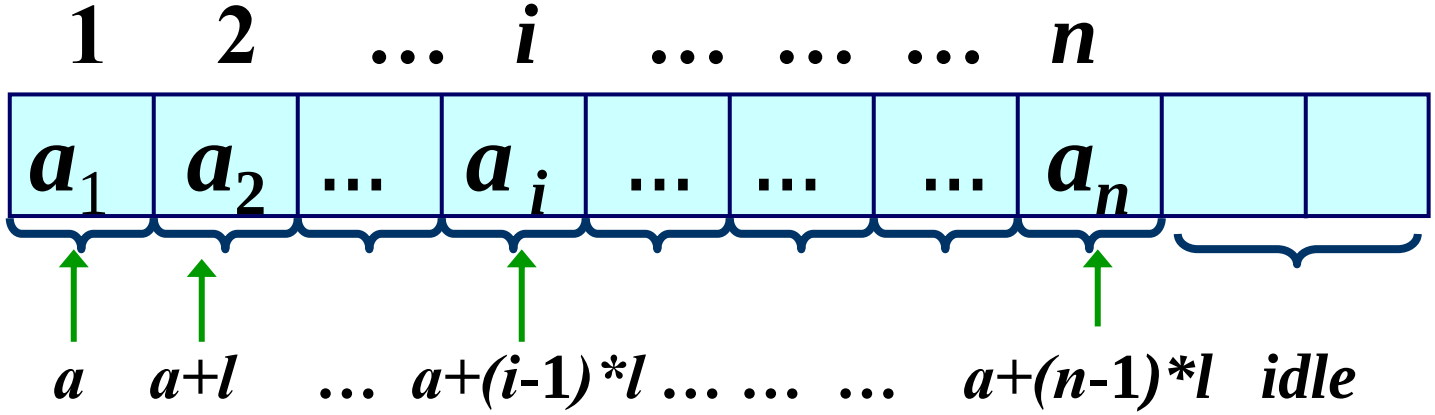
2.2 -顺序表的存储

假设每个元素占用*l*个存储单元，则元素的存储位置：

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

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

随机存取



2.2 –顺序表类型定义(方案一)

- 顺序表定义(方案一)

```
#define  LIST_SIZE  100
typedef  struct{
    ElemType      elem[LIST_SIZE];   /* 存储空间*/
    int           length;            /* 当前长度*/
}SqList_static;
```

评价:

1)LIST_SIZE过小，容易导致顺序表上溢;

2) LIST_SIZE过大，则会导致空间的利用率不高

2.2 –顺序表类型定义(方案二)

- 顺序表定义(方案二)

```
#define  LIST_INIT_SIZE  100  /* 存储空间的初始分配量*/
#define  LISTINCREMENT  10 /* 存储空间的分配增量   */
typedef  struct{
    ElemType    *elem;          /* 存储空间的基址          */
    int             length;         /* 当前长度                */
    int             listsize;       /* 当前分配的存储容量   */
}SqList;
```

该方案解决方案一中的“上溢”问题和“空间利用率不高”问题，但是有时间和空间代价的 。

1) 必须记载当前线性表的实际分配的空间大小;

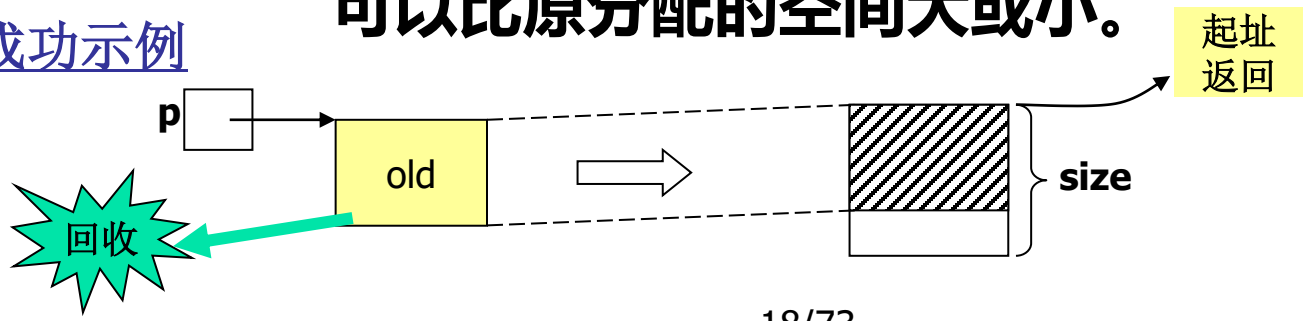
2) 当线性表不再使用时，应释放其所占的空间;

3) 要求实现该方案的语言具备内存空间的动态分配与释放能力。

2.2 –C中的动态分配与释放函数

- C中的动态分配与释放函数
  - void *malloc(unsigned int size)
    生成一大小为size的结点空间，将该空间的起始地址赋给p；
  - free(void *p)
    回收p所指向的结点空间;
  - void *realloc(void *p, unsigned int size)
    将p所指向的已分配内存区的大小改为size，size可以比原分配的空间大或小。

分配成功示例



2.2 –C中的动态分配与释放函数

- C中的动态分配与释放函数
  - void *malloc(unsigned int size)
    生成一大小为size的结点空间，将该空间的起始地址赋给p；
  - free(void *p)
    回收p所指向的结点空间;
  - void *realloc(void *p, unsigned int size)
    将p所指向的已分配内存区的大小改为size，size可以比原分配的空间大或小。

分配失败示例

2.2 –基本操作的实现(宏与Status)

- 基本操作的实现

  P10

```
/* 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
/* 函数结果状态类型，其值为上述的状态代码      */
typedef    int      Status;
```


2.2 –基本操作的实现(InitList_Sq)

- 基本操作的实现
  - **初始化** **Status InitList_Sq(SqList &L)**

```
Status  InitList_Sq( SqList &L) {
   // 构造一个空的线性表L
   L.elem = (ElemType *)
            malloc(LIST_INIT_SIZE*sizeof(ElemType));
   if ( L.elem == NULL )
      exit(OVERFLOW);                 // 存储分配失败
   L.length = 0;                      // 空表长度为0
   L.listsize = LIST_INIT_SIZE;       // 初始存储容量
   return OK;
} // InitList_Sq
```


2.2 –基本操作实现(LocateElem_Sq)

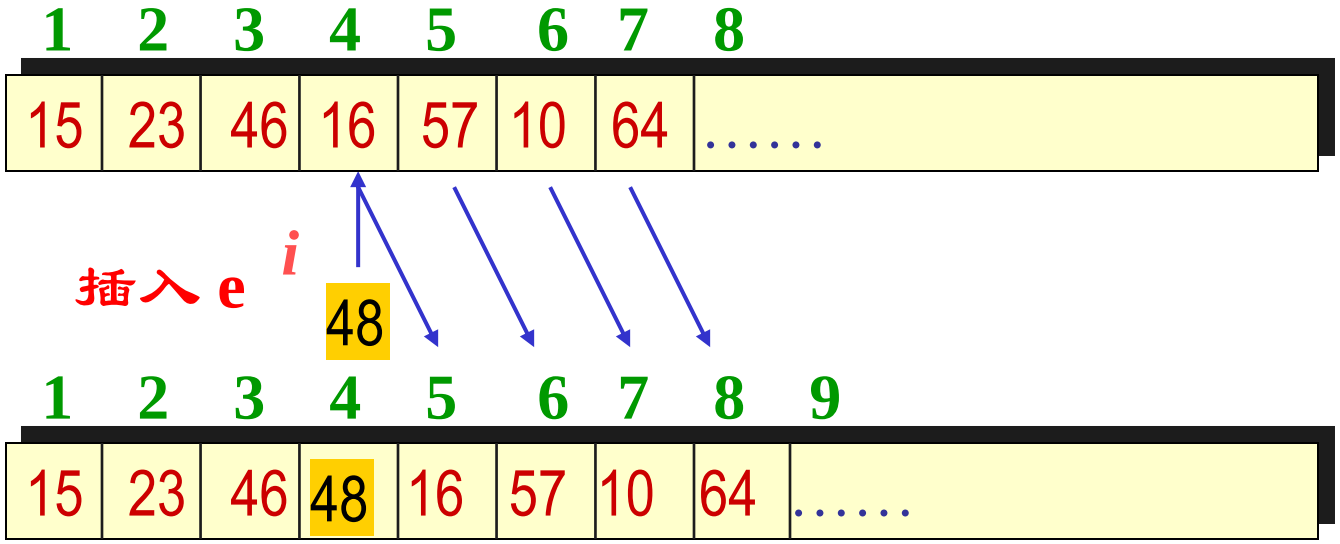
- 基本操作的实现
  - **求表长 L.length**
  - **取第i个元素 L.elem[i-1] (0<i<L.length+1)**
  - **按值查找**

```
int LocateElem_Sq(SqList L, ElemType e,
        Status (*compare)(ElemType, ElemType))
{
   for(i=0; i<L.length && !(*compare)(L.elem[i],e); i++);
   if (i<L.length) return i+1;
   else return 0;
}
```


2.2 –插入操作的实现

- 基本操作的实现
  - **插入操作**

    Status ListInsert_Sq(SqList &L, int i, ElemType e)



2.2 –插入操作的实现(分析设计)

- **插入操作——算法设计(算法2.4)**
  - **参数**：顺序表&L、插入位置i、插入元素e
  - **插入分析**：
    - 第i个位置放e，则原来第i~L.length个数据元素必须先后移，以腾出第i个位置;
    - 后移的顺序是从最后一个元素开始，逐个往后移

```
for ( j = L.length; j >= i; j--)  L.elem[j] = L.elem[j-1]; // 后移
L.elem[i -1] = e; //第i个元素存放在L.elem[i-1]中，起始下标为0
L.length = L.length+1;
```

  - **合法的位置**：i:1..L.length+1
  - **上溢及处理**：
    - 上溢发生的条件：L.length==L.listsize，
    - 处理：先申请一个有一定增量的空间：申请成功则原空间的元素复制到新空间，修改L.listsize，再进行插入工作；否则报错退出。

2.2 –插入操作的实现(算法)

```
Status  ListInsert_Sq( SqList &L, int i, ElemType e) {
   // 位置合法性的判断
   if ( i<1 || i>L.length +1 )     return ERROR;
   // 上溢时增加空间的分配
   if( L.length == L.listsize){
      newbase = (ElemType *) realloc(L.elem,
            (L.listsize+ LISTINCREMENT)*sizeof(ElemType));
      if ( newbase == NULL ) exit(OVERFLOW);
      L.elem = newbase;
      L.listsize += LISTINCREMENT;
   }
   // 插入元素
   for ( j = L.length; j >= i; j--)  L.elem[j] = L.elem[j-1];
   L.elem[i-1] = e;  L.length++;
   return OK;
}
```

为什么要引入newbase?

避免因空间分配失败时，将原空间地址丢失导致内存泄漏！

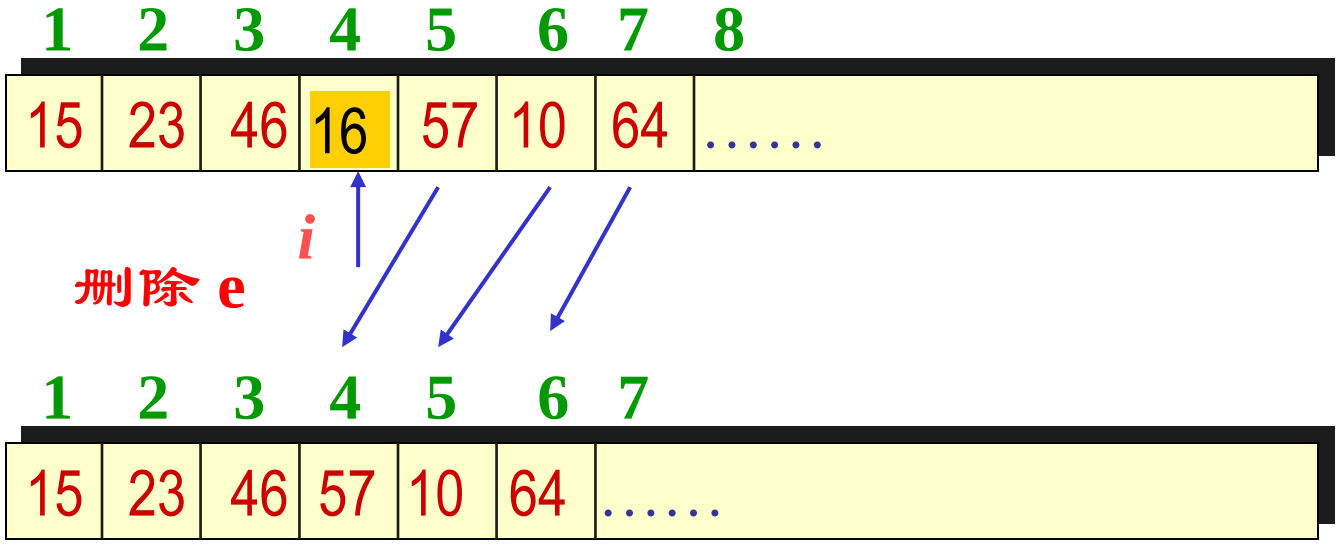
2.2 –插入操作的实现(算法分析)

- **时间复杂度**
  - 频次最高的操作：移动元素
  - 上溢时，空间的再分配与复制(realloc操作)分摊到每次插入操作中
  - 若线性表的长度为n，则：
    - *最好情况*：插入位置i为n+1，此时无须移动元素，T(n)=O(1);
    - *最坏情况*：插入位置i为1，此时须移动n个元素， T(n)=O(n);
    - *平均情况*：假设*pi*为在第i个元素之前插入一个元素的概率，则：
      插入时移动次数的期望值：$E_{is}=\sum_{i=1}^{n+1} p_i(n-i+1)$
      等概率时，即 $p_i=\frac{1}{n+1}$ ， $E_{is}=\frac{1}{n+1}\sum_{i=1}^{n+1}(n-i+1)=\frac{n}{2}$
      ∴ T(n) = O(n)
- **空间复杂度** S(n) = O(1)

2.2 –删除操作的实现

- 基本操作的实现
  - **删除操作**

Status ListDelete_Sq(SqList &L, int i, ElemType &e)



2.2 –删除操作的实现(分析设计)

- **删除操作——算法设计(算法2.5)**
  - **参数**：顺序表&L、删除位置i
  - **删除分析**：
    - 去掉第i 个元素，则原来第i+1~L.length个数据元素必须前移，以覆盖第i个位置;
    - 前移的顺序是从第i+1元素开始，逐个往前移

    for (j = i; j < L.length ; j++)　　L.elem[j-1] = L.elem[j];

    L.length = L.length-1;
  - **合法的位置：i:1..L.length**
  - **下溢及处理**：
    - 下溢发生的条件：**L.length≤0**
    - 处理：隐含在i的条件中

2.2 –删除操作的实现(算法)

- **删除操作——算法设计(算法2.5)**

```
Status  ListDelete_Sq( SqList &L, int i) {
    // 位置合法性的判断
    if ( i<1 || i>L.length )       return ERROR;
    // 删除
    for ( j = i; j < L.length ; j++)
      L.elem[j-1] = L.elem[j];
    L.length--;
    return OK;
}
```


2.2 –删除操作的实现(算法分析)

- **时间复杂度**
  - 频次最高的操作：移动元素
  - 若线性表的长度为n，则：
    - *最好情况*：删除位置i为n，此时无须移动元素，T(n)=O(1);
    - *最坏情况*：删除位置i为1，此时须前移n-1个元素，T(n)=O(n);
    - *平均情况*：假设*qi*为删除第i个元素的概率，则：
      删除时移动次数的期望值：$E_{de}=\sum_{i=1}^{n}q_i(n-i)$

      等概率时，即 $q_i=\frac{1}{n}$ ，$E_{de}=\frac{1}{n}\sum_{i=1}^{n}(n-i)=\frac{n-1}{2}$

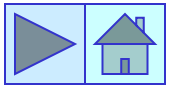
      ∴ T(n) = O(n)
- **空间复杂度** S(n) = O(1)

2.2 –算法设计(例2-1,2-2)

- 基于顺序表的算法设计

  例2-1和例2-2

  - 基本操作在顺序表中的实现
    - ListLength(La)　　La.length
    - GetElem(Lb, i, e)　　e = Lb.elem[i-1];
    - LocateElem(La, e, equal)
    - ListInsert(La, ++La_len, e)　　La.elem[La.length++] = e;
  - 基于顺序表实现例2-1和例2-2



算法2.6(P25-26)

```
int  LocateElem_sq(SqList L,ElemType e, Status
                   (*compare)(ElemType, ElemType)) {
        i=1;
        p=L.elem;
        while(i<=L.length && !(*compare)(*p++,e))
                ++i;
        if(i<=L.length)   return i;    // 查找成功
        else    return 0;     // 查找失败
  }
```

T(n)= O(L.length)

算法2.7(P26)

```
void MergeList_Sq(SqList La, SqList Lb, SqList &Lc) {
    pa=La.elem;     pb=Lb.elem;
    Lc.listsize=Lc.length=La.length+Lb.length;
    pc=Lc.elem=(ElemType*)malloc(Lc.listsize *sizeof(ElemType));
    if(!Lc.elem)   exit(OVERFLOW); // 存储分配失败
    pa_last=La.elem+La.length-1;  pb_last=Lb.elem+Lb.length-1;
    while(pa<=pa_last && pb<=pb_last) {
        if(*pa<=*pb)    *pc++=*pa++;
        else  *pc++=*pb++;
    }
    while(pa<=pa_last)   *pc++=*pa++;
    while(pb<=pb_last)   *pc++=*pb++;
}
```

T(n)= O(La.length+Lb.length)

讨论：顺序存储结构的不足

- **顺序映像的弱点**
  - **空间利用率**不高（预先按最大空间分配）
  - 表的**容量不可扩充**（针对顺序表的方案一）
  - 即使表的**容量可以扩充**（针对顺序表的方案二），由于其空间**再分配和复制的开销**，也不允许它频繁地使用
  - 插入或删除时需**移动**大量元素

如何解决？

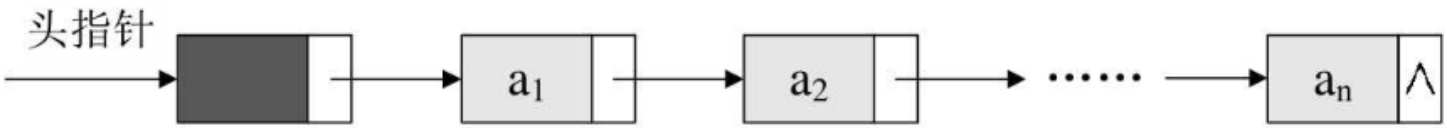
2.3 线性表的链式表示和实现

- 单链表
- 循环链表
- 静态链表
- 静态链表 vs. 动态链表
- 双向链表
- 其他表示

2.3.1 单链表–定义

- 单链表
  - **链表定义**
    - 特点

      逻辑上相邻的元素，物理上未必相邻；非随机存取



2.3.1 单链表–定义

- **链表定义**
  - **结点**–数据元素的存储映像，包括**数据域**和**指针域**（其信息称为**指针**或**链**）
  - **头指针**–链表存取的开始 ； **空(NULL)**-最后一个结点的指针

2.3.1 单链表–定义和基本操作

- **线性表的单链表存储结构：**

```
typedef struct LNode{
    ElemType  data;        //数据域
    struct LNode  *next;//后向指针域
}LNode, *LinkList;
```

- **基本操作的实现**
  - **取第i个元素** GetElem_L(LinkList L, int i, ElemType &e)
  - **插入操作** ListInsert_L(LinkList &L, int i, ElemType e)
  - **删除操作** ListDelete_L(LinkList &L, int i, ElemType &e)
  - **创建单链表** CreateList_L(LinkList &L)
    - 头插法　　尾插法

2.3.1 单链表–取第i个元素

- 取第i个元素**GetElem_L(LinkList L, int i, ElemType &e)**

参数：链表L、位置i、取得的第i个元素&e

算法：带头结点的单链表中取第i个元素

```
Status  GetElem_L( LinkList L, int i, ElemType &e) {
  // j表示当前p所指向的元素在线性表中的位序
  p = L; j = 1;
  while ( j < i && p != NULL ){
    p = p->next; j++;     // 后移指针并计数j
  }
  if ( p == NULL || j > i) return  ERROR;// 第i个元素不存在
  e = p->data;  // 取第i个元素
  return OK;
}
```

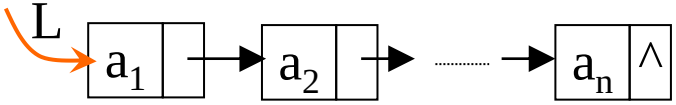
频次最高的操作

当1≤i≤表长n时，频度为i-1;
否则为n

T(n) =O(n)

2.3.1 单链表–头结点的引入

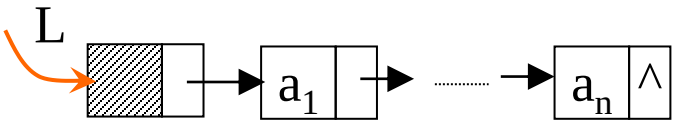
- **单链表中是否引入头结点?**
  - **无头结点**



空表时，**L为NULL**

第一个结点无前驱结点,只能通过某指针变量来指向，如**L**；其余结点均有前驱结点，故可通过其直接前驱结点的**next**域来指向，即……**->next**;

  - **有头结点**



空表时，**L**指向一结点(称为头结点)

第一个结点和其余结点均可统一表示为其直接前驱结点的**next**域所指向的结点，即……**->next**。

2.3.1 单链表–插入操作(分析设计)

- 插入操作**ListInsert_L(LinkList &L, int i, ElemType e)**

【设计思路】

相关结点：a_{i-1}和a_i

结点的表示：引入指针变量**LinkList p;**

∵链表结点的指针域是指向该结点的直接后继

∴当**p**指向a_{i-1}时，a_i可用***(p->next)**表示

耗时间！

关键步骤：

①使**p**指向a_{i-1} 结点

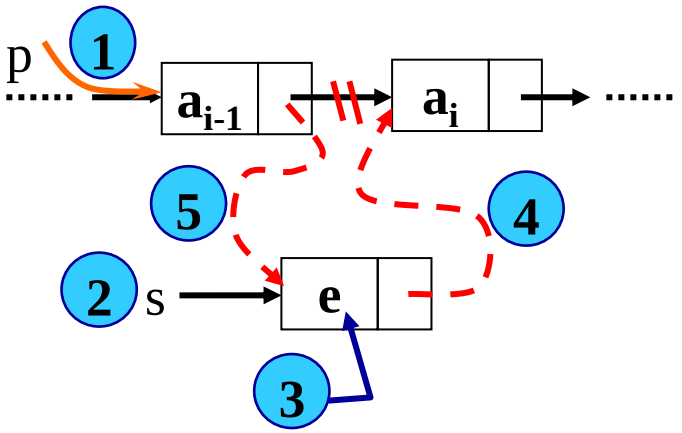
p≠NULL，则

② **s = (LinkList)malloc (sizeof(LNode));**

③ **s->data = e;**

④ **s->next = p->next;**

⑤ **p->next = s;**



分析操作间的依赖关系，确定操作的次序!

T(n) =O(n)

2.3.1 单链表–插入操作(分析设计)

- 插入操作**ListInsert_L(LinkList &L, int i, ElemType e)**

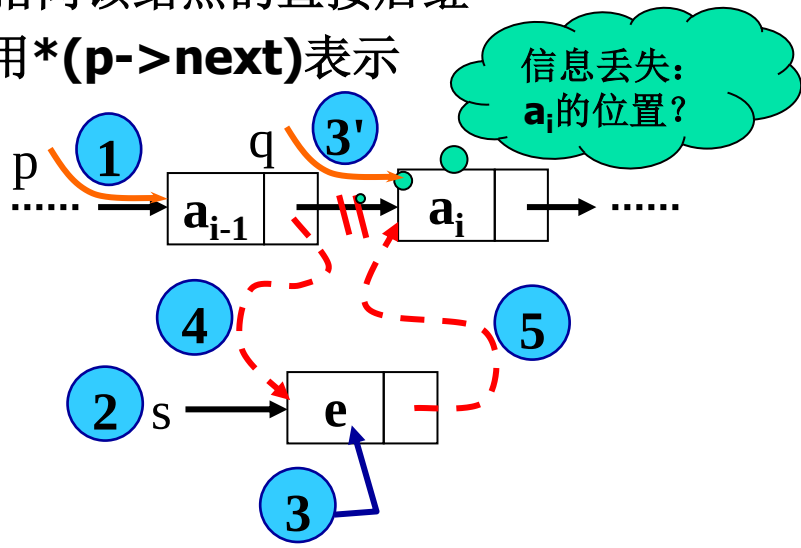
【设计思路】

相关结点：a_{i-1}和a_i

结点的表示：引入指针变量**LinkList p;**

∵链表结点的指针域是指向该结点的直接后继

∴当**p**指向a_{i-1}时，a_i可用***(p->next)**表示



关键步骤：

①使**p**指向a_{i-1} 结点

p≠NULL，则

② s = (LinkList)malloc (sizeof(LNode));

③ s->data = e;

③' q=p->next;

④ p->next = s;

⑤ s->next = q;

随意给出操作的次序!

2.3.1 单链表–插入操作(算法)

【有头结点的算法】

```
Status   ListInsert(LinkList &L, int i, ElemType e){
   // 有头结点，无须对i为1的插入位置作特殊处理
   p = L; j = 0;   // 对p,j初始化; *p为L的第j个结点
   while( p != NULL && j<i-1){
      p = p->next;          j++;     // 寻找第i-1个结点的位置
   }
   if( p == NULL || j>i-1)
      return ERROR;        // i小于1或大于表长
   s = (LinkList )malloc(sizeof(LNode)); // 生成新结点
   if ( s == NULL )
      exit(OVERFLOW);   // 空间分配不成功，报错返回
   s->data = e; s->next = p->next;  // 插入L中
   p->next = s;
   return OK;
}
```


2.3.1 单链表–插入操作(算法)

【无头结点的算法】

```
Status   ListInsert(LinkList &L, int i, ElemType e){
   //无头结点，须对i为1的插入位置作特殊处理
   if ( i==1){
      s = (LinkList )malloc(sizeof(LNode)); // 生成新结点
      if ( s == NULL )
         exit(OVERFLOW); // 空间分配不成功，报错返回
      s->data = e; s->next = L;  // 插入到链表L中
      L = s;                     // 修改链头指针L
   }else{
      p = L; j = 1;   // 对p,j初始化; *p为链表的第j个结点
      ……  //同有头结点算法的处理：①~⑤
   }
   return OK;
}
```

【思考】对比无头结点和有头结点在插入算法上的不同，分析其中的原因。

2.3.1 单链表–插入操作(小结)

- 链表操作的算法设计小结
  - 采用画图法
    - 先画初始状态
    - 画出结果状态
    - 标出各操作步骤的次序
      次序的确定方法：
      - 根据操作步骤之间的依赖关系
      - 随意确定一种次序➔信息的丢失
        在信息丢失前，增加操作步骤以暂存将要丢失的信息
  - 假设一些条件：有无头结点…

2.3.1 单链表–删除操作(分析设计)

- 删除操作ListDelete_L(LinkList &L, int i, ElemType &e)

【设计思路】

相关结点：a_{i-1}、a_i和a_{i+1}

结点的表示：引入指针变量**LinkList p;**

∵链表结点的指针域是指向该结点的直接后继

∴当**p**指向a_{i-1}时，a_i可用***(p->next)**表示，

a_{i+1}可用***(p->next->next)**表示

关键步骤：

①使**p**指向a_{i-1} 结点

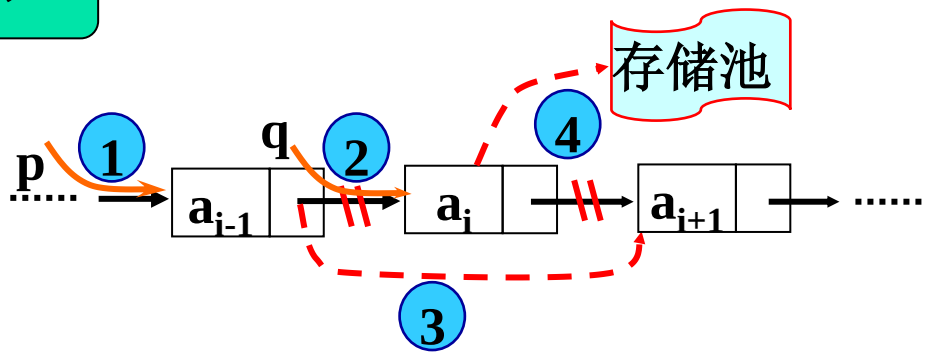
耗时间！

p->next≠NULL，则

② **q = p->next;**

③ **p->next = p->next->next;**

④ **free(q);**

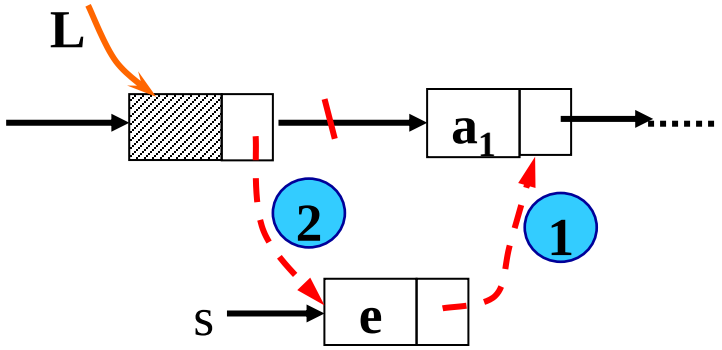


T(n) =O(n)

2.3.1 单链表–创建操作

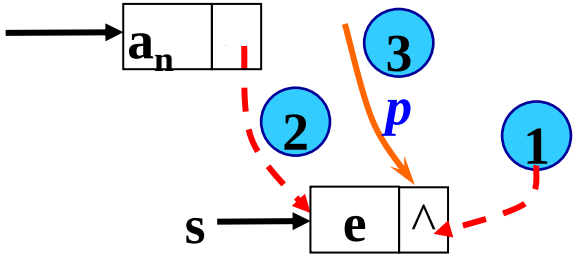
- 创建单链表 CreateList_L(LinkList &L)
  - 头插法

新结点插入到线性表中的第1个结点之前.



  - 尾插法

新结点插入到线性表中最后1个结点之后.



2.3.1 单链表–头插法创建

- 头插法(算法2.11, P30)

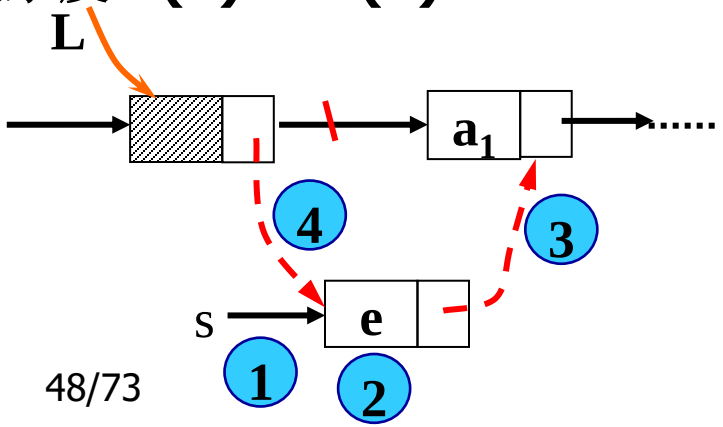
思想：每次将待插结点*s插入到第一个结点之前；当有头结点时，待插结点也可视为插入到第0个结点(头结点)之后。

插入步骤：以单链表中有头结点为例，单个结点的构造和插入步骤如下:① s = (LinkList)malloc(sizeof(LNode));

② scanf(&s->data); ③ s->next = L->next; ④ L->next = s;

算法分析：由上可看出每次插入一个结点所需的时间为O(1)

∴头插法创建单链表的时间复杂度 T(n) = O(n)



2.3.1 单链表–尾插法创建

- 尾插法

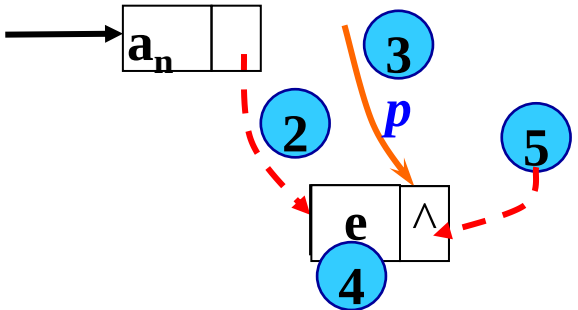
思想：待插结点插入到最后一个结点之后 。

插入步骤：① 获得最后一个结点的位置,使p指向该结点

② p->next = (LinkList)malloc(sizeof(LNode));

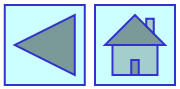
③ p = p->next; ④ scanf(&p->data); ⑤ p->next = NULL;

算法分析：要想获取最后一个结点的位置，必须从头指针开始顺着next链搜索链表的全部结点，该过程的时间复杂度是 O(n)。如果每次插入都按此方法获取最后一个结点的位置，则整个创建算法的时间复杂度为T(n) = O(n2)。



2.3.2 循环链表

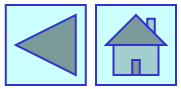
- 循环链表
  - 问题1
    如何从一个结点出发，访问到链表中的全部结点？
    ——循环链表
  - 问题2
    如何在O(1)时间内由链表指针访问到第一个结点和最后一个结点？
    ——头指针表示/尾指针表示
  - 与单链表在基本操作的实现上的差异
    如链表的创建
    循环结束条件的判断:p==NULL => p==L



2.3.3 静态链表

- 静态链表 (P31~34)
  - 问题：
    - 若语言不支持**指针类型**，能有链式存储吗？
      可以借用语言中的数组来表示链表——静态链表
      数组元素(数据元素的值、指示器)——结点
  - 类型定义

```
#define  MAXSIZE      1000
typedef struct{
  ElemType      data;
  int           cur;      //替代动态链表结点中的指针域
}component, SLinkList[MAXSIZE];
```

2.3.3 静态链表-与动态链表的差异

- 与动态链表使用上的差异
  - 链表的结点是数组中的一个分量
  - 数组分量中的**cur**代替动态链表中的指针域，用来指示直接后继或直接前驱结点在数组中的相对位置
  - 数组的第**0**个分量可以视为(备用链的)头结点
  - 以(**0**或)负数代表空指针**NULL**
  - 以整型游标**i**代替动态指针**p**，以**i = S[i].cur**代替**p = p->next**取下一个元素的位置

2.3.3 静态链表-动态分配与释放的模拟

- 动态分配与释放的模拟
  - **思想**：将未使用的分量用游标**cur**链成一个备用链；插入时，取备用链中的第一个结点作为插入的新结点；删除时将从链表中删除下来的结点链接到备用链上
  - **初始化链表(算法2.14, P33)**

    space[0].cur为备用链的头指针，**cur**为**0**表示空指针

```
void InitSpace_SL(SLinkList &space){
  for( i=0 ; i < MAXSIZE-1; i++)
    space[i].cur=i+1;
  space[MAXSIZE-1].cur = 0;
}
```

2.3.3 静态链表-动态分配与释放的模拟

- 动态分配与释放的模拟
  - **模拟动态分配(算法2.15, P33)**

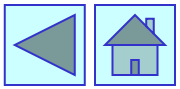
```
int Malloc_SL(SLinkList &space){
  i = space[0].cur;
  if ( space[0].cur!=0 ) space[0].cur = space[i].cur;
  return i;
}
```

  - **模拟动态释放(算法2.16, P33)**

```
void Free_SL(SLinkList &space, int k){
  space[k].cur = space[0].cur;  space[0].cur = k;
}
```

2.3.4 动态链表与静态链表

- 例2-1
  - 顺序表的实现：基本操作的简单替换
  - 动态链表
    - 无须特意求**La**、**Lb**的长度
      ——原目的是为了控制线性表的边界
    - 无须每次调用**GetElem_L()**
      ——它可以和外层的循环合二为一
    - 无须每次调用**ListInsert_L()**
      ——可以引入指针变量记录**La**的尾结点的位置
    - 可以利用**Lb**的原有结点空间
      ——前提：**Lb**本身不再被使用
    - 假设链表均有头结点，且是非循环的



2.3.4 动态链表与静态链表

- 动态链表：算法需要重新整合！

使pa指向La的最后一个结点

依次处理Lb中的每一个结点

在La中查找e

未查到，插入到La的尾部

查到，取Lb的下一个结点

```
void Union_L( LinkList &La, LinkList &Lb){
 for ( pa = La; pa->next != NULL; pa=pa->next);
 for ( pb = Lb; pb->next !=NULL;  ){
   e = pb->next->data;
   for(qa = La->next; qa != NULL && !equal(qa->data, e) ;
         qa=qa->next);
   if (qa == NULL) {
     pa->next = pb->next;
     pa = pa->next;
     pb->next = pb->next->next;
     pa->next = NULL;
   } else
     pb = pb->next;
 }//end of for
}
```

2.3.4 动态链表与静态链表

- 静态链表：和动态链表类似，也需要重新整合！

```
void Union_SL( SLinkList &space, int &sla, int &slb){
 for ( pa = sla; space[pa].cur != 0; pa=space[pa].cur);
 for ( pb = slb; space[pb].cur!=0;  ){
  e = space[space[pb].cur].data;
  for(qa = space[sla].cur;
        qa != 0 && !equal(space[qa].data, e) ;
        qa=space[qa].cur);
  if (qa == 0) {
    space[pa].cur = space[pb].cur;
    pa = space[pa].cur;
    space[pb].cur = space[space[pb].cur].cur;
    space[pa].cur = 0;
  } else
    pb = space[pb].cur ;
 }//end of for
}
```

使pa指向La的最后一个结点

依次处理Lb中的每一个结点

在La中查找e

未查到，插入到La的尾部

查到，取Lb的下一个结点

2.3.4 动态链表与静态链表

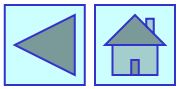
- 例2-2
  - 动态链表: (算法2.12, P31)
- 例2-3
  - 求差集，即(A-B)U(B-A)
  - 用静态链表: (算法2.17, P33~34)

2.3.5 双向链表

- 双向链表
  - 问题
    如何在O(1)时间内找到一个结点的直接前驱和后继？
    ——双向链表
  - 类型定义

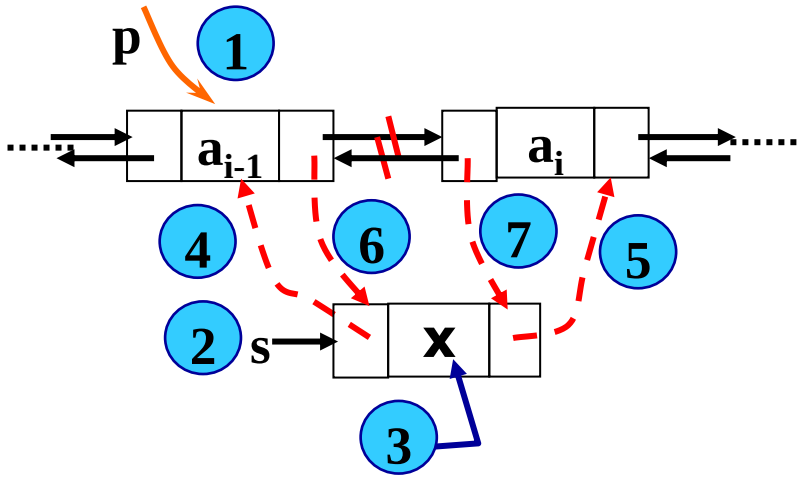
```
typedef struct DuLNode{
   ElemType                    data;
   struct DuLNode  *prior;
   struct DuLNode  *next;
}DuLNode, *DuLinkList;
```

  - 基本操作：插入、删除



2.3.5 双向链表

- 双向链表
  - 插入

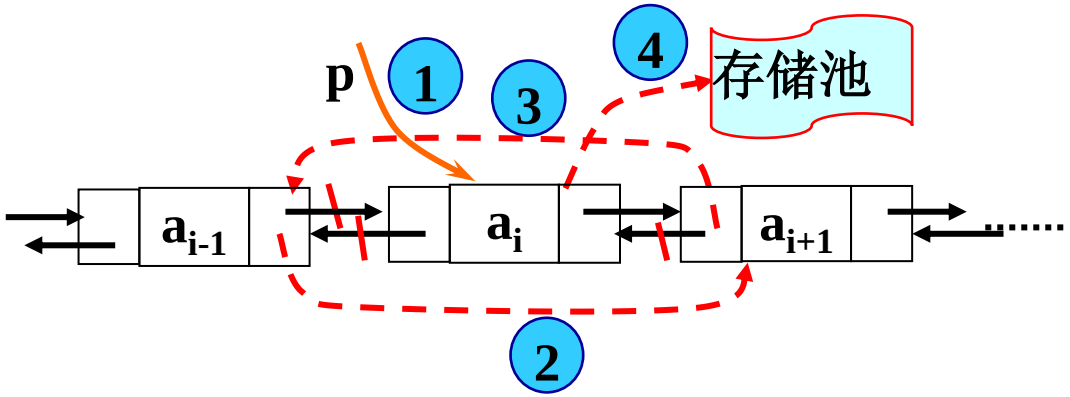


1. 若有头结点，则第i-1个结点一定存在，第i个结点可能不存在；故让p指向第i-1个结点；
2. 注意第7步：

```
if (s->next!=NULL) s->next->prior = s;
```

2.3.5 双向链表

- 双向链表
  - 删除



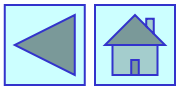
1. 若有头结点，则第i-1个结点和第i个结点一定存在，第i+1个结点可能不存在；故让p指向第i-1个结点或第i个结点；
2. 注意第3步：

```
if (p->next!=NULL) p->next->prior = p->prior;
```

2.3.6 其他表示

- 存储结构的具体设计
  结合实际问题操作的特征以及环境进行选取
  如单链表的另一种结构

```
typedef struct LNode{
        ElemType        data;
        struct LNode    *next;
 }LNode, *Link, *Position;
 typedef struct {
        Link            head, tail;
        int             len;
 }LinkList;
```

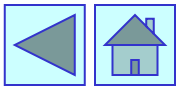



2.3.6 其他表示

- 有序表 ——一种特殊的线性表
  - 有序顺序表、有序链表
  - 插入操作的特殊性：对于有序表，在插入一个元素时，无须给出插入位置；其插入位置与插入元素的值(序关键字)相关。

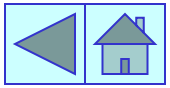
    ListInsert_L(LinkList & L, int i, ElemType e)

    ➔ **ListInsert_L(LinkList & L, ElemType e)**



2.4 基于链表的算法设计-问题描述

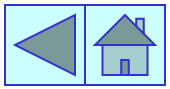
- 问题描述——合并、分解问题
  - 若干源表按一定条件合并成一个目标表
    - **示例：例2-1 A=A∪B; 例2-2 C=A∪B**
  - 将一个源表按一定条件分解成若干目标表
    - **示例：已知一线性链表中含有三类字符的数据元素，试将该链表分割为三个循环链表，其中每个循环链表中均只含一类字符。**
  - 若干源链表按一定条件重新组成若干目标链表
    - **示例：A=A∪B, B=A∩B**
  - 删除一个表中的部分元素
    - **示例：删除数据元素按值非递减有序排列的单链表中所有值大于mink且小于maxk的元素，同时释放被删结点空间。**
    - **示例：已知A,B,C三个递增有序表，要求对A表进行如下操作：删去那些既在B表中出现又在C表中出现的元素。**



2.4 基于链表的算法设计-问题分析

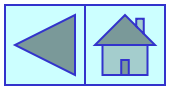
- 问题分析
  - 筛选处理的时间复杂度会因为源链表是否有序，而有不同的数量级
    - **问题1**：已知**A,B,C**三个表，要求对**A**表进行如下操作：删去那些既在**B**表中出现又在**C**表中出现的元素。
      O(LengthA*max(LengthB,LengthC))
    - **问题2**：已知**A,B,C**三个递增有序表，要求对**A**表进行如下操作：删去那些既在**B**表中出现又在**C**表中出现的元素。
      O(LengthA+LengthB+LengthC)

在下面的合并、分解的问题求解框架中，不考虑因源表是否有序而导致的筛选必须在遍历其它源表的全部结点才能确定的问题。



2.4 基于链表的算法设计-解题框架

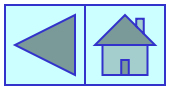
- 解题框架
  - **必要的初始化**
    - 各目标表的初始化
      - 对于仅对一源表进行分解删除的，该源表同时也是目标表，不需要初始化。
      - 目标表中是否含头结点，是否是循环链表，是否利用源表结点空间，会影响目标表的初始化
    - 引入源表指示器、目标表指示器
      - 源表中是否含头结点，会影响源表指示器的初始化
      - 目标表的创建方法(头插法/尾插法)会影响到目标表指示器的定义与设置



2.4基于链表的算法设计-解题框架

- 解题框架
  - **循环处理**

    当各源表指示器未到表尾时，根据筛选条件，循环地将满足条件的结点插入到目标表中(或根据题目要求释放掉)
    - 源表是否是循环链表，会影响源表表尾的判断
    - 对于选中的结点，其位置要暂存，便于插入到目标表中(或释放掉)；并且对应的源表指示器也要向后推进；
    - 选中结点插入到特定目标表的方法，跟目标表的创建方法有关
  - **处理尚未到达表尾的源表中的剩余表项**



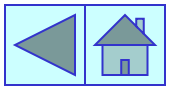
2.4 -例1：简单的单表分解释放

例1：简单的单表分解释放

题目要求: 删除数据元素按值非递减有序排列的单链表中所有值大于mink且小于maxk的元素，同时释放被删结点空间.

解题思路一:

依次扫描表中的各个结点，判断该结点的值是否大于**mink**且小于**maxk**，若符合则删除之。



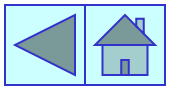
2.4 -例1：简单的单表分解释放

例1：简单的单表分解释放

题目要求: 删除数据元素按值非递减有序排列的单链表中所有值大于mink且小于maxk的元素，同时释放被删结点空间.

解题思路二:

- 对该链表从首结点开始，查找并记录如下四个结点的位置：
  - 第一个大于**mink**的结点的指针**p**及其前驱结点的指针**ppre**
  - 第一个大于等于**maxk**的结点指针**q**及其前驱结点的指针**qpre**；
- **ppre->next =q; qpre->next=NULL;**
- 删除从**p**到**qpre**指向的全部结点



2.4 -例1：简单的单表分解释放

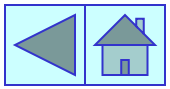
例1：简单的单表分解释放

思路二：

特殊情况：

- **q**为空，删除从**p**到表尾的全部结点，可以并入上述的一般情况；
- **p**为空，不存在这样的元素，不必删除

```
for( ppre= L, p=L->next; p!=NULL && p->data<=mink ;
      ppre=p, p=p->next);
if ( p != NULL ){
    for( qpre= ppre, q=p ; q!=NULL && q->data<maxk ;
         qpre=q, q=q->next);
    ppre->next=q;
    qpre->next=NULL;
    while ( p != NULL ){
      dp=p; p= p->next; free(dp);
    }
}
```

2.4 -例2：双向循环链表的自身变换

例2：双向循环链表的自身变换

题目要求: 将带头结点的双向循环链表L=(a_1,a_2,a_3,……,a_n)调整为

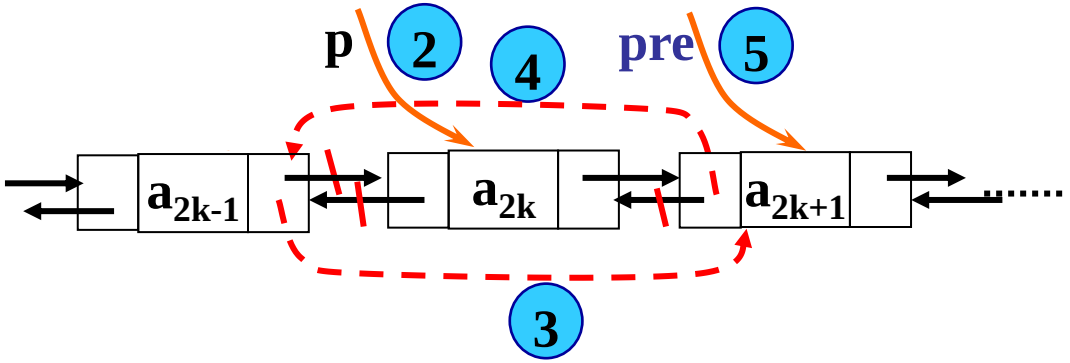
(a_1,a_3,……,a_n,……,a_4,a_2).

解题思路: 调整过程为一循环处理过程，每次循环执行如下两步：

1) 从表中删除一结点，结点空间不释放

若依次删除第**2**、**4**、……结点，则整个循环顺着后向链**next**域推进。

p: 指向待删除的结点，**pre:** 指向待删除结点的直接前驱



2.4 -例2：双向循环链表的自身变换

例2：双向循环链表的自身变换

题目要求:将带头结点的双向循环链表L=(a_1,a_2,a_3,……,a_n)调整为

(a_1,a_3,……,a_n,……,a_4,a_2).

解题思路: 调整过程为一循环处理过程，每次循环执行如下两步：

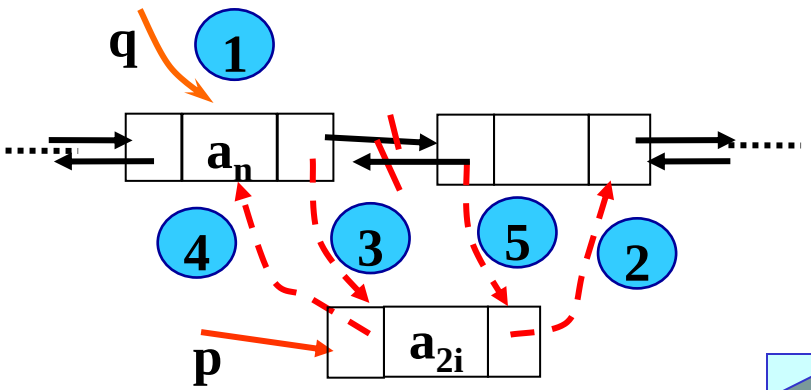
2) 将1)中去掉的结点*p插入到表中。

有两种插入方法：a)顺着后向链插入;b)顺着前向链插入。

a)顺着后向链next插入

每次插入到a_n结点的后面

q: 指向a_n结点



2.4 -例2：双向循环链表的自身变换

例2：双向循环链表的自身变换

题目要求:将带头结点的双向循环链表L=(a_1,a_2,a_3,……,a_n)调整为(a_1,a_3,……,a_n,……,a_4,a_2).

解题思路: 调整过程为一循环处理过程，每次循环执行如下两步：

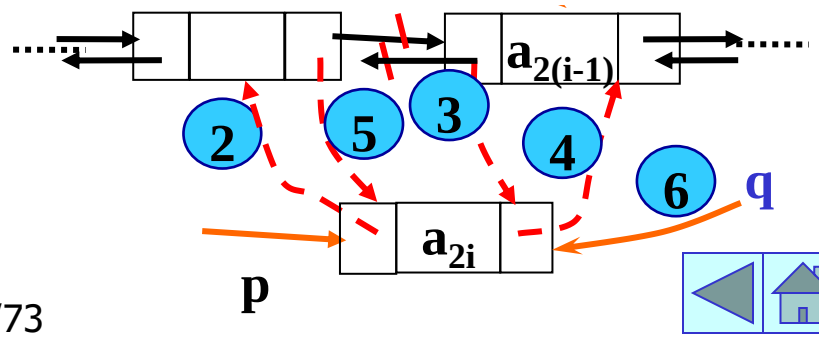
2) 将1)去掉的结点*p插入到表中。

有两种插入方法：a)顺着后向链插入;b)顺着前向链插入。

b)顺着前向链prior插入

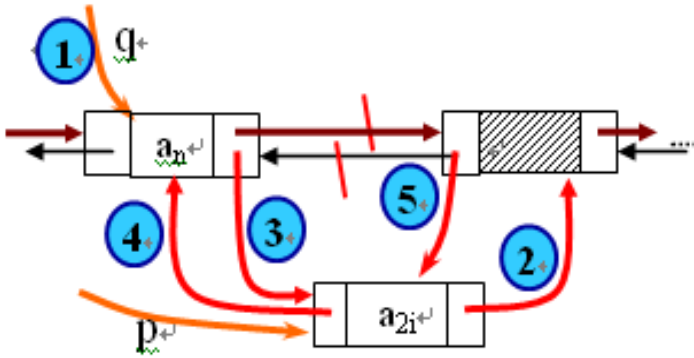
每次插入到上次新插入的结点之前, 初始插入到头结点之前;

q: 初始指向头结点,以后指向上次新插入的结点



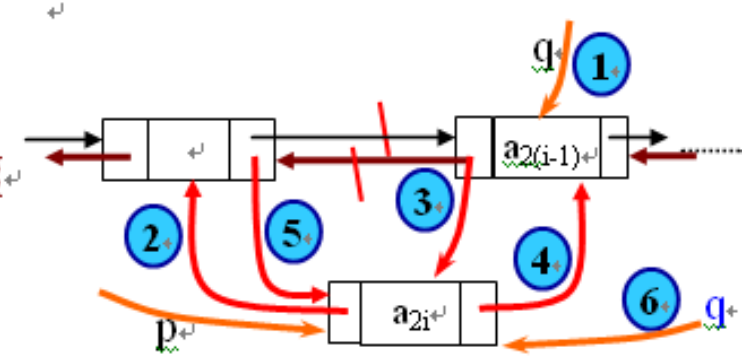
利用插入方法①

```
Status    AdjustCDuList( DuLinkList L)
{
    // 初始化待删除结点的前驱指针 pre 和表尾结点指针 q
    pre = L->next; q = L->prior;

    // 循环删除和插入，pre 未到达尾结点或尾结点的前驱
    while(pre!=q && pre->next!=q){
        // 将要删除的结点用 p 指示，改变链的关系
        p = pre->next; pre->next = p->next;
        p->next->prior = p->prior;
        pre = pre->next;                    // 修改 pre
        // 将 p 插入到 q 的后面
        p->next = q->next;      q->next = p;  // 后向链的调整（图中的②③）
        p->prior = q; p->next->prior = p;     // 前向链的调整（图中的④⑤）
    }
    return OK;
}
```



利用插入方法②

```
Status    AdjustCDuList( DuLinkList L)
{
    // 初始化待删除结点的前驱指针 pre 和待插入位置的后继指针 q
    pre = L->next; q = L;

    // 循环删除和插入，pre 未到达尾结点(q->prior)
    // 或尾结点的前驱
    while( pre != q->prior && pre->next != q->prior ){
        // 将要删除的结点用 p 指示，改变链的关系
        p = pre->next; pre->next = p->next;
        p->next->prior = p->prior;
        pre = pre->next;                    // 修改 pre
        // 将 p 插入到 q 的前面
        p->prior = q->prior ; q->prior = p;   // 前向链的调整（图中的②③）
        p->next = q;  p->prior->next = p;     // 后向链的调整（图中的④⑤）
        q = q->prior;                       // q 的调整(图中的⑥)
    }
    return OK;
}
```



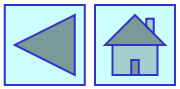
2.5 一元多项式的表示及相加

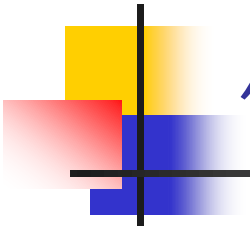
- 一元多项式的表示

  $P_n(x) = p_0 + p_1x + p_2x^2 + \cdots\cdots + p_nx^n$

  - 表示1　$P = (p_0, p_1, p_2, \cdots\cdots, p_n)$
    - 存储结构：顺序表，链表
    - 表长=多项式的次数+1，多项式存在许多系数为0的项
  - 表示2
    - 只存储非零系数项，(系数，指数)

      $P = ((p_1, e_1), (p_2, e_2), \cdots\cdots, (p_m, e_m))$
    - 存储结构：顺序表(求值)、链表(加、减、…)





作业

- **2.25　2.29　2.38**
- **3.1-3.7**（不写在作业本上，不用交）