第二章 线性表

总结：廖兴宇　15116404939　liaoxingyu@nwpu.edu.cn

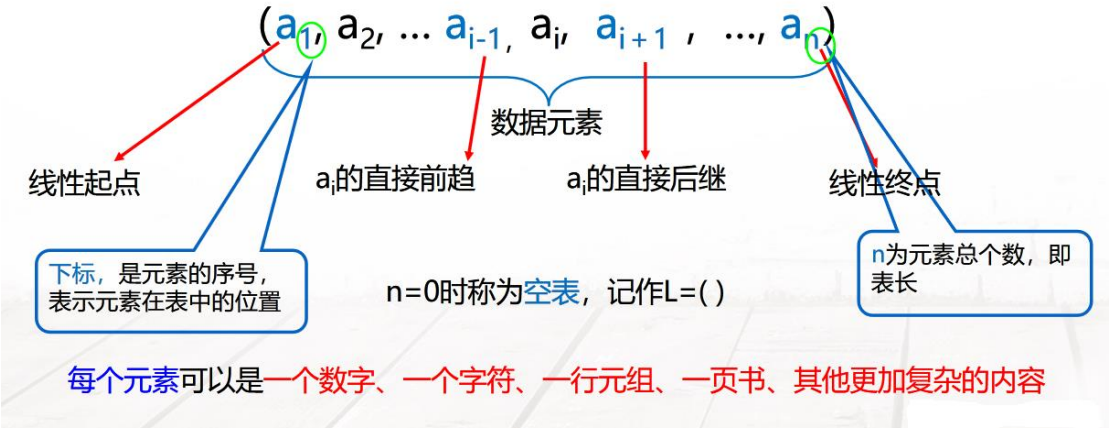
1　线性结构的概念

数据结构中线性结构指的是数据元素之间存在着“一对一”的线性关系的数据结构。线性结构是一个有序数据元素的集合。

线性结构特点：

- 线性结构有唯一的首元素（第一个元素）
- 线性结构有唯一的尾元素（最后一个元素）
- 除首元素外，所有的元素都有唯一的“前驱”
- 除尾元素外，所有的元素都有唯一的“后继”
- 数据元素之间存在“一对一”的关系，即除了第一个和最后一个数据元素之外，其它数据元素都是首尾相接的。

如数组（a1,a2,a3,…,an），a1 为第一个元素，an 为最后一个元素，此集合即为一个线性结构的集合。



常用的线性结构有： 线性表，栈，队列，双队列，循环队列，一维数组，串。其中，栈和队列只是属于逻辑上的概念（仅仅是一种思想，一种理念，实际中不存在），线性表则是在内存中数据的一种组织、存储的方式。

2　线性表的概念

线性表（linear list）是线性结构中的一种，一个线性表是 n 个具有相同特性的数据元素的有限序列。 数据元素是一个抽象的符号，其具体含义在不同的情况下一般不同。 在稍复杂的线性表中，一个数据元素可由多个数据项（item）组成，此种情况下常把数据元素称为记录（record），含有大量记录的线性表又称文件（file）。

例1　分析26 个英文字母组成的英文表

（A，B，C，D，……，Z）

数据元素都是字母;　元素间关系是线性

例2　分析学生情况登记表

数据项

学号	姓名	性别	年龄	班级
2001011810205	于春梅	女	18	2001级电信016班
2001011810260	何仕鹏	男	18	2001级电信017班
2001011810284	王　爽	女	18	2001级通信011班
2001011810360	王亚武	男	18	2001级通信012班
:	:	:	:	:

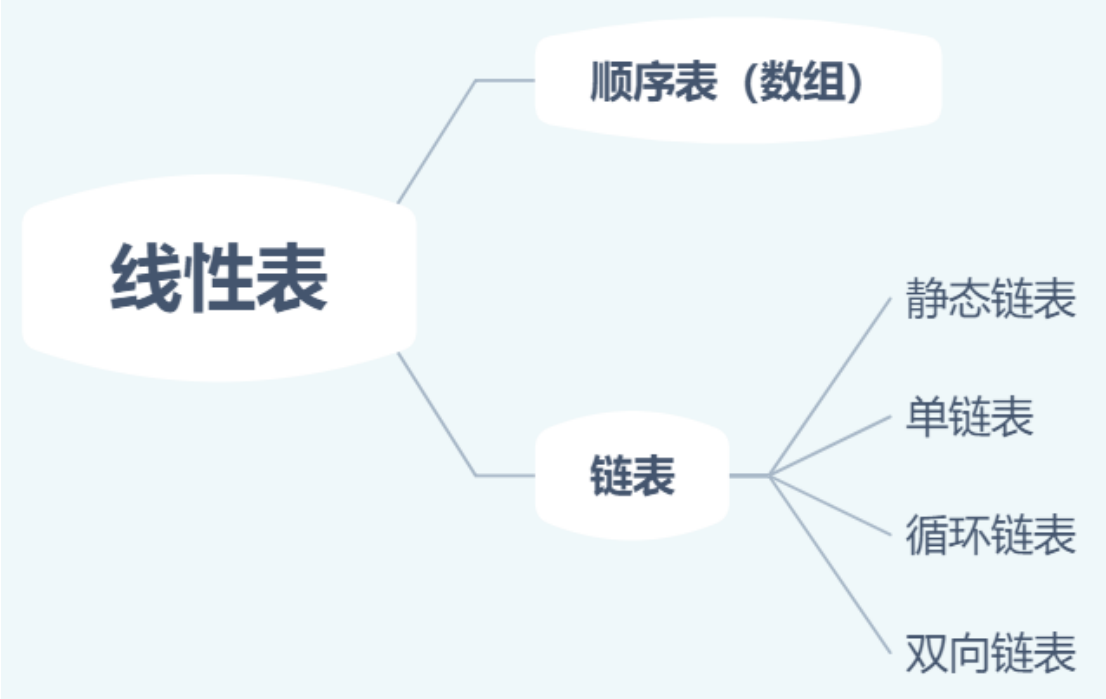
记录

数据元素都是记录；元素间关系是线性，含有大量记录的线性表又称文件。

文件

同一线性表中的元素必定具有相同特性

线性表包括顺序和链式两种表示和实现方式。 其中，链式又分为静态链表、单链表、循环链表、双向链表这几种表示和存储形式。



3　线性表的抽象数据类型 ADT

线性表是一个相当灵活的数据结构，它的长度可根据需要增长或缩短，即对线性表的数据元素不仅可以进行访问，还可以进行插入和删除等操作。抽象数据类型线性表 ADT List 的定义如下：

ADT List {

数据对象：D={a_i | a_i ∈ ElemSet，i=1,2,3,⋯,n n⩾0}

数据关系：R={ < a_{i-1}, a_i > |a_{i-1},a_i ∈D, i=2,3,⋯,n}

基本操作:

IniList (&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 已经存在。

操作结果：销毁线性表 L。

ClearList (&L)

初始条件：线性表 L 已经存在。

操作结果：将 L 重置为空表。

ListEmpty (L)

初始条件：线性表 L 已经存在。

操作结果：若 L 为空表，则返回 TURE，否则返回 FALSE。

ListLength (L)

初始条件：线性表 L 已经存在。

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 已经存在。

操作结果：返回 L 中第一个与 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，则返回结果为 0。

PriorElem (L, cur_e, &pre_e)

初始条件：线性表 L 已经存在。

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

NextElem (L, cur_e, &next_e)

初始条件：线性表 L 已经存在。

操作结果：若 cur_e 是 L 的数据元素，且不是最后一个，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 存在且非空，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。

ListTravarse(L,visit())

初始条件：线性表 L 已经存在。

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

实例 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中的成员），现要求一个新的集合 $A=A\cup B$。

```
void union (List &La, List Lb) {
    La_len = ListLength(La);   // 求线性表 La 的长度
    Lb_len = ListLength(Lb);   // 求线性表 Lb 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 相同的数据元素，则插入之。
           ListInsert (La, ++La_len, e);
    }
}
```

具体操作步骤为：
1、从线性表 LB 中取出一个数据元素；
2、依值在线性表 LA 中进行查询；
3、若不存在，则将它插入到 LA 中。

GetElem, ListInsert 这两个操作的执行时间与表长无关，但是 LocateElem 的执行时间与表长成正比。时间复杂度为：*O(ListLength(LA)×ListLength(LB))*

实例 2：已知线性表 LA 和 LB 中的元素按值非递减有序排列，现在要求将 LA 和 LB 归并成一个新的线性表 LC，且 LC 中的数据元素仍然按值非递减有序排列。

```
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排列
   //归并 La 和 Lb 得到新的线性表 Lc，Lc 的元素也按值非递减排列
     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;
           }
     }
     while (i<=La_len) {
           GetElem (La, i++, ai);
           ListInsert (Lc, ++k, ai);
     }
     while (j<=Lb_len){
           GetElem (Lb, j++, bj);
           ListInsert (Lc, ++k, bj);
     }
}
```

具体操作步骤为：
1、分别从 LA 和 LB 中取得当前元素 ai 和 bj；

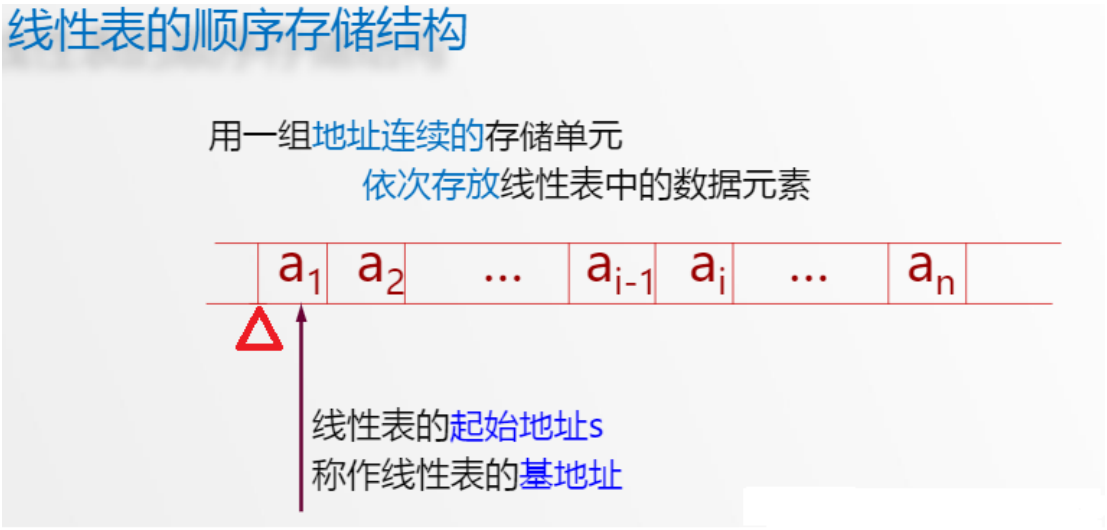
2、若 ai≤bj，则将 ai 插入到 LC 中，否则将 bj 插入到 LC 中。直到将 LA 表中的元素全部插入到 LC 表中。

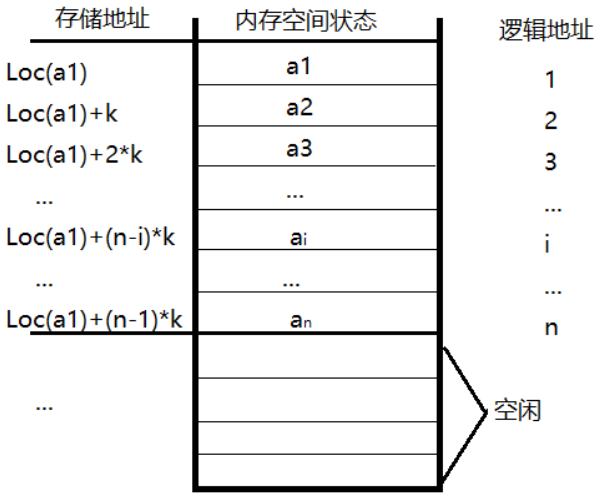
3、若 LA 表的元素已经全部插入到 LC 表中，而 LB 表还有元素没有插入，则直接将 LB 表剩余的元素插入到 LC 中。

虽然包含 3 个 while 循环，但是对于某一组具体的输入（LA 和 LB），后两个循环（while）语句只执行一个循环体。时间复杂度为：*O(ListLength(LA)+ListLength(LB))*

4 线性表的顺序表示与实现

线性表的顺序存储是指在内存中用地址连续的一块存储空间来依次存放线性表的数据元素，用这种存储形式存储的线性表称为顺序表。顺序表的主要特点是：以元素在计算机内“物理地址相邻”来表示数据元素之间的逻辑关系。即用物理位置相邻来表示逻辑关系，任一元素均可随机存取。





假设线性表的每个元素需要占用 k 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 个元素的存储位置 $LOC(a_i)$ 之间满足以下的关系：

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

一般来说，线性表的第 i 个数据元素 a_i 的存储位置可以表示为：

$$LOC(a_i) = LOC(a_1) + (i-1) \times k$$

4.1 顺序表的类型定义

```
#define  LIST_INIT_SIZE      100     // 最大长度
#define  LISTINCREMENT     10       //线性表存储空间的分配增量
typedef  struct
{
    ElemType  *elem;     // 指向数据元素的基地址（数组指针）
    int  length;          // 线性表的当前长度
    int  listsize;        //当前分配的存储容量（以 sizeof (ElemType)为单位）
 }SqList;
```

其中 elem 为数组指针，其指示线性表的基址。length 指示线性表的当前长度， listsize 指示顺序表当前分配的存储空间大小，一旦因插入元素而空间不足时，可以进行再分配，即为顺序表增加一个大小为存储 LISTINCREMENT 个数据元素的空间。

4.2 顺序表的初始化、清空与销毁操作

1） 顺序表初始化（创建空顺序表）

顺序表的初始化操作就是为顺序表分配一个预定义大小（LIST_INIT_SIZE）的数组空间，并将线性表的当前长度设置为 0。

```
Status InitList_Sq (SqList &L)  //构造一个空的顺序表 L
{
    L.elem = (ElemType *) malloc (LIST_INIT_SIZE * sizeof (ElemType));
    //使用 malloc 函数为顺序表申请分配一片连续的存储空间。
    if (!L.elem)
        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;
}
```

顺序表的存储空间是静态分配的，在程序运行之前必须明确规定其存储规模，若线性表的长度变化较大，则存储规模难以预先确定，估计过大容易造成空间浪费，估计太小又可能导致空间溢出（空间不足）。时间复杂度为 *O(1)*, 空间复杂度为 *O(n)*。

2） 清空顺序表

清空顺序表就是将顺序表的长度设置为 0。

```
Status ClearList_Sq (SqList &L)  //清空顺序表 L
{
    L.length = 0;      //将顺序表的长度置为 0。
    return OK;
}
```

3）销毁顺序表

注销顺序表就是将顺序表所占存储空间释放，表长度设置为 0。

```
Status DestroyList_Sq (SqList &L)  //销毁顺序表 L
{
    Free (L.elem);      // 释放存储空间。
    L.elem = NULL;
    L.length = 0;       // 表长置 0。
    L.listsize = 0;     // 存储容量置 0。
    return OK;
}
```

时间复杂度为 *O(1)*, 空间复杂度为 *O(1)*。

4.3 顺序表随机存取元素

1）获取顺序表 L 中第 i 个元素的内容

查找顺序表 L 中第 i 个元素，并用 e 来返回其值。

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

时间复杂度为 *O(1)*, 空间复杂度为 *O(1)*。

2）获取顺序表 L 中第一个值与 e 相等元素的位序

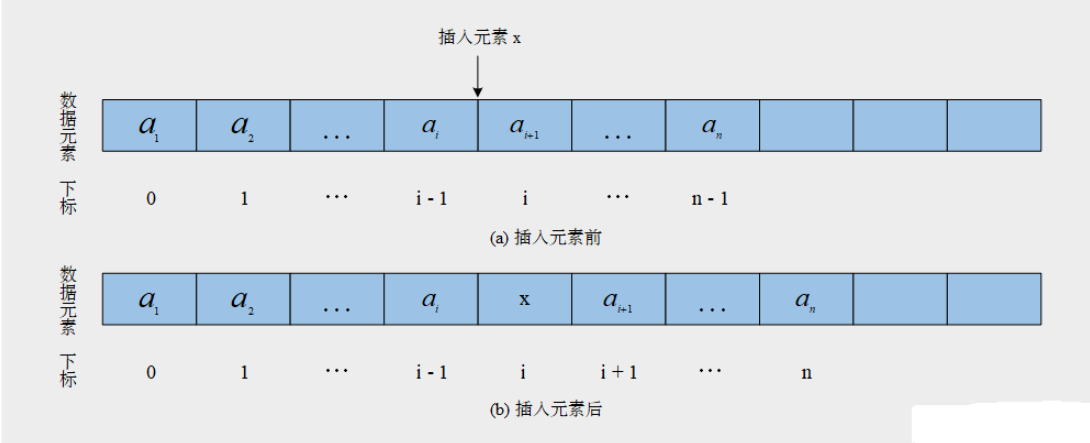
查找顺序表 L 中第 1 个与 e 相等元素的位置。在顺序表 L 中查找与 e 相等的元素，若 L.elem[i]==e, 则找到该元素并返回其在表 L 中的位序 i+1；若找不到，则返回 0。

```
int LocateELem_Sq (SqList L, ElemType e)  // 在线性表 L 中查找值为 e 数据元素的位序。
{
    for (i=0; i< L.length; i++)
       if (L.elem[i]==e)
          return i+1;
          //第 i-1 的单元存储着第 i 个元数（比如：第 0 个单元存储着第 1 个元素）。
    return 0;  //若找不到，则返回 0。
}
```

基本操作是进行两个元素之间的比较，若 L 中存在和 e 相同的元素，则比较次数 i （ 1<= i <= L.length），若不存在 L 中则要比较 L.length 次。因此 LocateELem_Sq 的平均时间复杂度为 *O（L.length）*即为 *O(n)*, 空间复杂度为 *O(1)*。

4.4 顺序表插入元素

顺序表的插入操作是指在顺序表的第 i-1 个数据元素和第 i 个数据元素之间插入一个新的数据元素，就是要使长度为 n 的线性表 (a_1, …, a_{i-1} , a_i, … a_n) 变成长度为 n+1 的线性表 (a_1, …, a_{i-1} , **b**, a_i, … a_n), 数据元素 a_{i-1} 与 a_i 之间的逻辑关系发生了变化。




```
Status ListInsert_Sq (SqList &L, int i, ElemType e){
    //在顺序线性表 L 中第 i 个位置之前插入新的元素 e。
    //i 的合法值为 1<= i <=ListLength_Sq(L) + 1，ListLength_Sq(L)返回 L 中数据元素个数。
    if (i < 1 || i > L.length+1)
        return ERROR;  //如果 i 的值不合法，返回错误。
    if (L.length >= L.listsize){
        ElemType *newbase;
        newbase = (ElemType *) realloc (L.elem, (L.listsize + LISTINCREMENT) * sizeof(ElemType));
        if (!newbase)  //如果空间分配失败，返回溢出。
            return OVERFLOW;
        L.elem = newbase;  //把分配的新地址给列表。
        L.listsize += LISTINCREMENT;  //增加存储容量。
    }
    ElemType *q, *p;
    q = &(L.elem[i-1]);  //q 为插入的位置
    for (p = &(L.elem[L.length - 1]); p >= q; --p)
        *(p + 1) = *p;  //插入位置之后的元素右移
    *q = e;  //插入 e
    ++L.length;  //列表长度加 1
    return OK;
}
```

完成顺序表的插入操作需要经过如下的步骤：

1） 线性表中指定位置及其之后的元素，依次向后移动一个位置，空出索引 i 的位置；

2） 数据元素 x 插入到第 i 个存储位置；

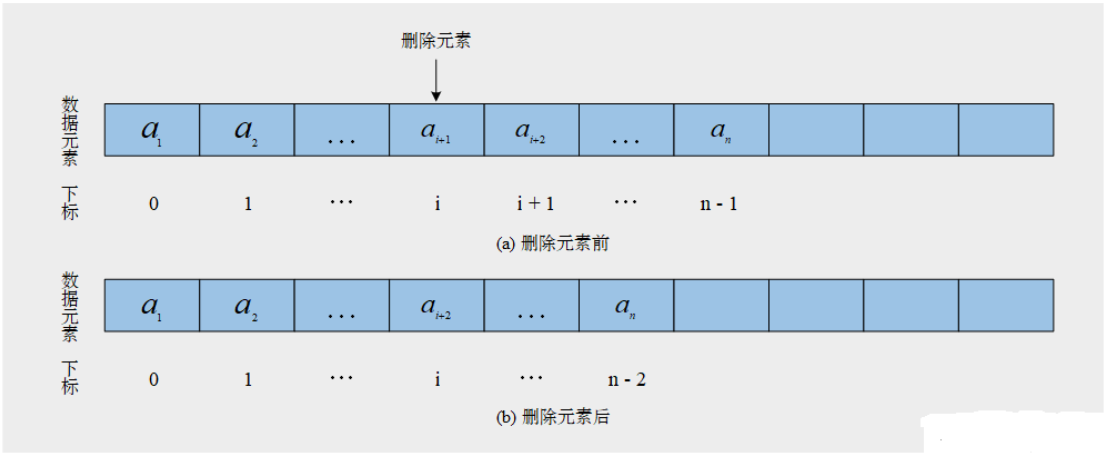
3） 插入结束后使顺序表的长度增加 1。

需要注意的是，如果顺序表空间已满（L.length >= L.listsize），则首先需要分配新的空

间。如果提供的索引大于列表的大小，则将新元素 x 附加到列表的末尾。插入操作主要的时间消耗在于移动插入位置之后的元素，由于新元素插入位置 i 的范围是 1<= i <= L.length，最佳情况是只需将新元素 x 附加到列表的末尾，此时的时间复杂度为 *O(1)*，最坏的情况是插入位置是第一个位置，此时需要移动 L.length 个元素，时间复杂度为 O（L.length）即为 *O(n)*。顺序表插入操作的空间复杂度为 *O(1)*。

4.5 顺序表删除元素

顺序表的删除操作是指将顺序表的第 i 个数据元素删除，就是要使长度为 n 的线性表 (a_1, …, a_{i-1} , a_i, … a_n) 变成长度为 n-1 的线性表 (a_1, …, a_{i-1} , a_{i+1}, … a_n), 数据元素 a_{i-1}、a_i 与 a_{i+1} 之间的逻辑关系发生了变化。




```
Status ListDelete_Sq (SqList &L, int i, ElemType &e) {
        //在顺序线性表中删除第 i 个元素，并用 e 返回其值
        //i 的合法值为 1<= i <= L.length_Sq(L)
        if (i < 1 || i > L.length)
             return ERROR;
         ElemType *p = &(L.elem[i-1]);  //p 指向删除元素的位置
         e = *p;   //被删除元素的值赋给 e
         ElemType *q = L.elem + L.length - 1;   //q 指向列表尾元素的位置。
         for(++p; p <= q; ++p)
              *(p-1) = *p;  //被删除元素之后的元素位置依次左移。
         --L.length;  //表长减 1
         return OK;
}
```

完成顺序表的删除操作需要经过如下的步骤：

1） 线性表中指定位置及其之后的元素，依次向前移动一个位置，用 i+1 位置的元素（a_{i+1}）覆盖 i 位置的元素（a_i），从而保证逻辑上相邻的元素物理位置也相邻；

2） 删除结束后使顺序表的长度减 1。

需要注意的是：最好情况是删除元素在表的最末尾（i=L.length），此时不需要移动其他元素（循环 0 次），最好的时间复杂度为 O(1)。最坏情况是删除表头元素（i=1），需要将后续的 L.length-1 个元素全都向前移动（循环 L.length-1 次），因此最坏情况下的时间复杂度为 O（L.length-1）即为 O(n)。

当在顺序表中某个位置“插入”或“删除”一个数据元素时，其时间主要耗费在移动元素上（换句话说，移动元素的操作为预估算法时间复杂度的基本操作），而移动元素的个数取决于“插入”和“删除”元素的位置。

假设 p_i 是在第 i 个元素之前插入一个元素的概率，则在长度为 n 的顺序表中插入一个元素时所需移动元素次数的期望值（平均次数）为：

$$E_{is} = p_1 \times n + p_2 \times (n-1) + p_3 \times (n-2) + \cdots + p_n \times 1 + p_{n+1} \times 0$$

// $p_1 \times n$：在第 1 个位置插入元素需要向后移动 n 个元素。
// $p_{n+1} \times 0$：在第 n+1 个位置插入元素时无需向后移动元素。因为提供的索引 n+1 大于列表的大小 n，则需要先分配新的空间，并将新元素 x 附加到列表的末尾，需要移动元素的数目为 0。

$$E_{is} = \sum_{i=1}^{n+1} p_i \times (n-i+1)$$

假设 q_i 是删除第 i 个元素的概率，则长度为 n 的顺序表中删除一个元素时所需移动元素次数的期望值（平均次数）为：

$$E_{dl} = q_1 \times (n-1) + q_2 \times (n-2) + q_3 \times (n-3) + \cdots + q_{n-1} \times 1 + q_n \times 0$$

// $q_1 \times (n-1)$：在第 1 个位置删除元素需要向前移动 n-1 个元素。
// $q_n \times 0$：在第 n 个位置删除元素时无需向后移动元素。

$$E_{dl} = \sum_{i=1}^{n} p_i \times (n-i)$$

我们可以假设在顺序表的任何位置上“插入”和“删除”元素都是等概率的，即为：

$$p_i = \frac{1}{n+1} \quad , \quad q_i = \frac{1}{n}$$

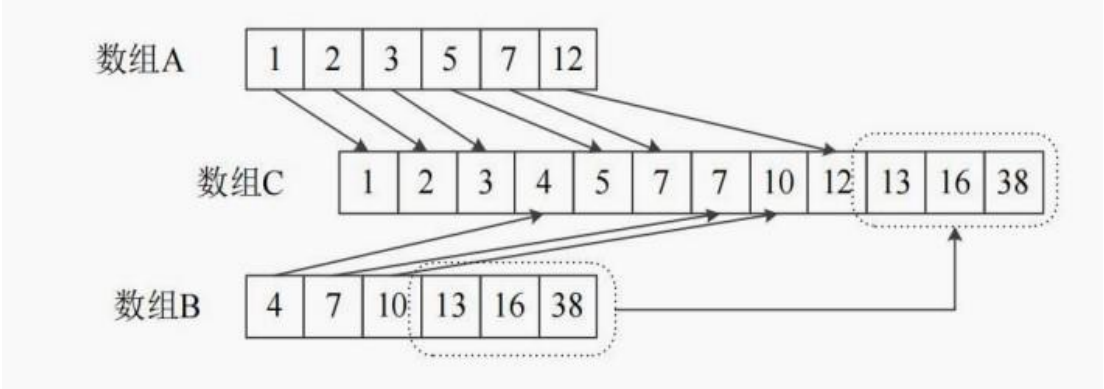
$$E_{is} = \frac{1}{n+1} \sum_{i=1}^{n+1} (n-i+1) = \frac{n}{2}$$

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

在顺序表中“插入”和“删除”一个数据元素，平均约移动表中一半的元素。若表长为 n，则算法 ListInsert_Sq 和 ListDelete_Sq 的时间复杂度为 $O(n)$。

4.6 顺序表合并元素

已知线性表 La 和 Lb 中的元素按值非递减有序排列，现在要求将 La 和 Lb 归并成一个新的线性表 Lc，且 Lc 中的数据元素仍然按值非递减有序排列。




```
void MergeList_Sq (SqList La, SqList Lb, SqList &Lc)
{
    //已知顺序表 La 和 Lb 的元素按值非递减排列
    //归并 La 和 Lb 得到新的顺序表 Lc，Lc 的元素也按值非递减排列
    pa = La.elem;
    pb = Lb.elem;
    Lc.listsize = Lc.length = La.length + Lb.length;     //Lc 的表长为 La+Lb
    pc = Lc.elem = (ElemType*)malloc(Lc.listsize * sizeof(ElemType));    //给 pc 分配空间
    if (!Lc.elem)
        exit(OVERFLOW);   //存储分配失败
    pa_last = La.elem + La.length - 1;   //La 最后一个元素的位置
    pb_last = Lb.elem + Lb.length - 1;   //Lb 最后一个元素的位置
    while (pa <= pa_last && pb <= pb_last)    //当 pa 和 pb 没有超出相应的表时
    {
        //归并
        //*pa 指针存储值
        //&pa 指针地址
        if (*pa <= *pb)          //当 pa<=pb 时，pa 赋给 pc 并且两个指针+1
            *pc++ = *pa++;
        else                     //当 pa > pb 时，pb 赋给 pc 并且两个指针+1
            *pc++ = *pb++;
    }
    //当表的长度不一样时，插入剩余的部分
    while (pa <= pa_last)
        *pc++ = *pa++;          //插入 La 的剩余元素
    while (pb <= pb_last)
        *pc++ = *pb++;          //插入 Lb 的剩余元素
}
```

具体操作步骤为：

1) 分别从 La 和 Lb 中取得当前元素 ai 和 bj；

2) 若 ai≤bj，则将 ai 插入到 Lc 中，否则将 bj 插入到 Lc 中。直到将 La 表中的元素全部插入到 LC 表中；

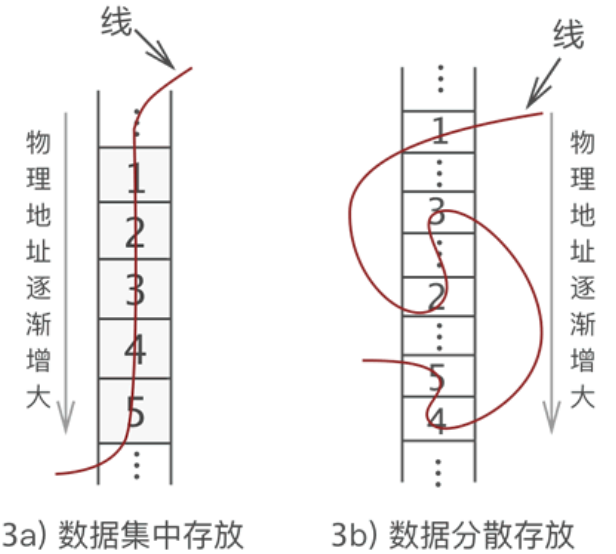
3) 若 La 表的元素已经全部插入到 Lc 表中，而 Lb 表还有元素没有插入，则直接将 Lb 表剩余的元素插入到 Lc 中。

虽然包含 3 个 while 循环，但是对于某一组具体的输入（La 和 Lb），后两个循环（while）语句只执行一个循环体。MergeList_Sq 的时间复杂度为：O (ListLength (La) ＋ListLength (Lb))， 即为 *O (n)*。

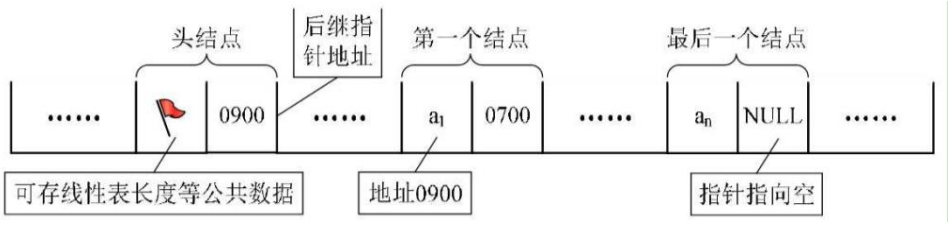
5 线性表的链式表示与实现

5.1 链表线性表的概念

线性表的链式存储结构的特点是用一组任意的存储单元存储线性表的数据元素（这组存储单元可以是连续的，也可以是不连续的）。因此，为了表示每个数据元素 a_i 与其直接后继数据元素 a_{i+1} 之间的逻辑关系，对数据元素 a_i 来说，除了存储其本身的信息之外，还需要存储一个指示其直接后继的信息（即直接后继的存储位置）。



3a) 数据集中存放　　3b) 数据分散存放



5.2 链式线性表的类别

根据链表每个结点包含的指针域数目和链接关系，其可以分为：单向链表，单向循环链表，双向链表，双向循环链表，十字链表。

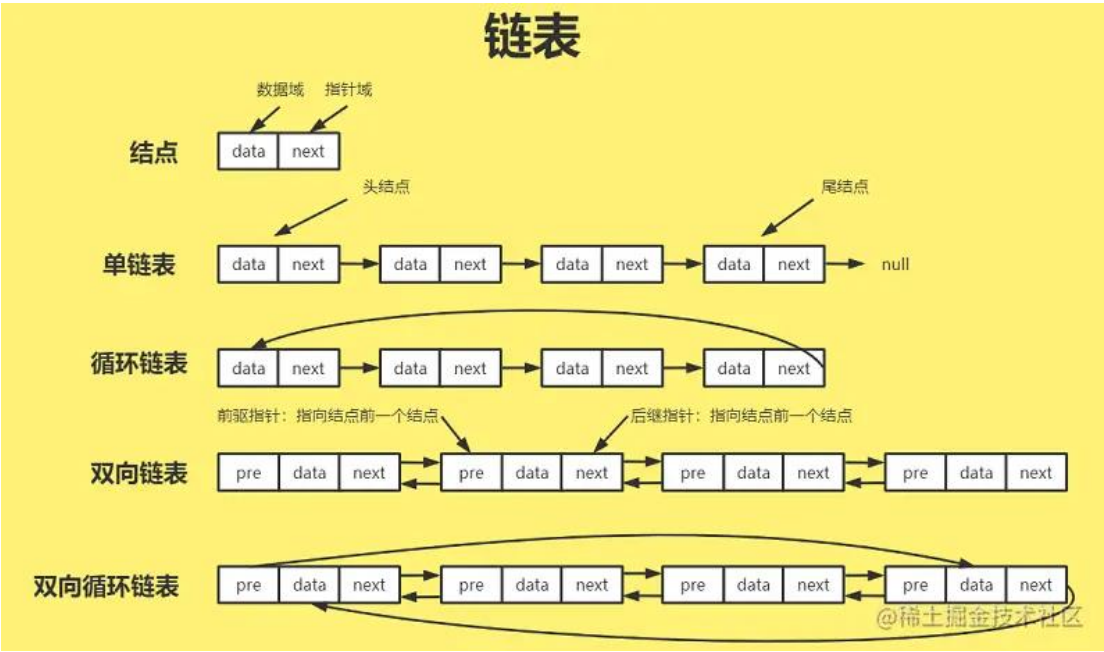
单向链表只有一个指向当前结点的后继结点的指针。其优点是插入和删除元素简单，遍历时不会出现死循环。缺点是只能从头到尾遍历，只能找到后继，无法找到前驱，就是只能前进不能后退；

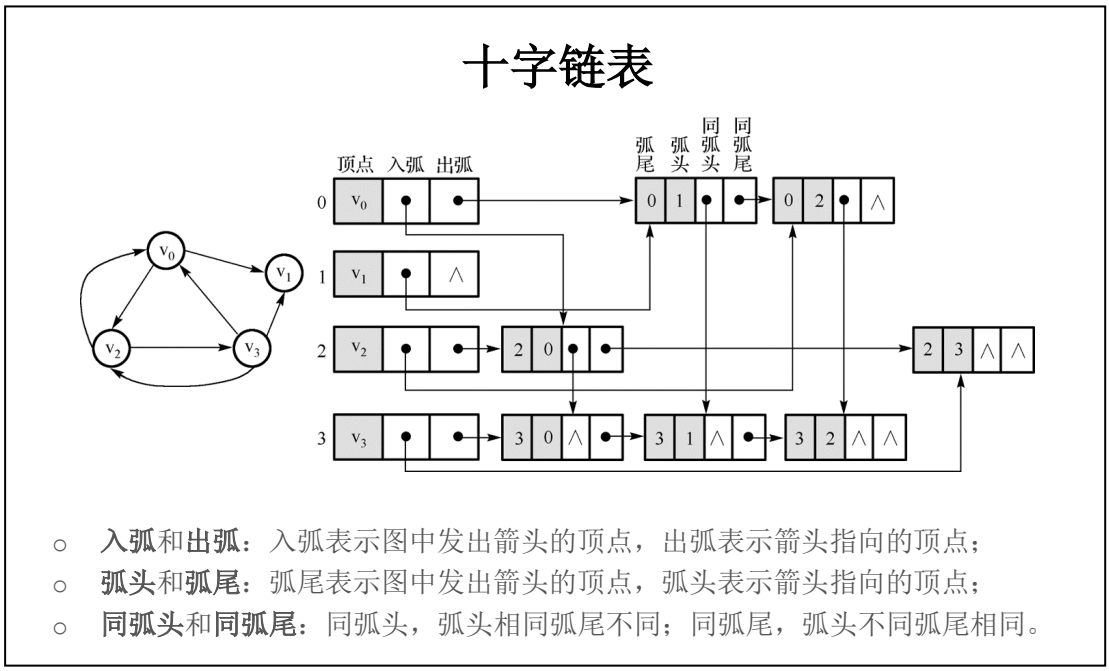
单向循环链表是一种“循环性”的单向链表，其相对于单向链表最大的特点是“尾结点指向头结点”。1）在插入和删除元素的时间复杂度为 $O(1)$，比数组快很多；2）不需要考虑到队列的结尾，因为循环链表的末尾与开头是相连的，能够高效地解决在队列的任意一端插入和删除元素；

双向链表是在单向链表的基础上增加了 1 个指向前驱结点的指针。其优点是可快速找到当前结点的前驱和后继结点，可进可退。其缺点是 插入和删除结点相对单向链表操作要更复杂，而且需要多分配 1 个指针存储空间；

双向循环链表是在双向链表的头结点之前增加了一个指向表末尾结点的指针，将尾结点的后驱指针指向表头结点。支持从任意一个结点开始双向遍历整个链表，支持循环访问元素，双向循环链表插入和删除元素的效率比双向链表更高（适合于高频率插入和删除元素的场景）；

十字链表是有向图的另一种链式存储结构。1）它是将有向图的邻接表和逆邻接表结合起来得到的一种链表，既容易找到 v_i 为尾的弧，也容易找到以 v_i 为头的弧，因而容易求得顶点的出度和入度；2）十字链表除了结构复杂一点外，利用十字链表来创建图，算法的时间复杂度和邻接表是相同的。

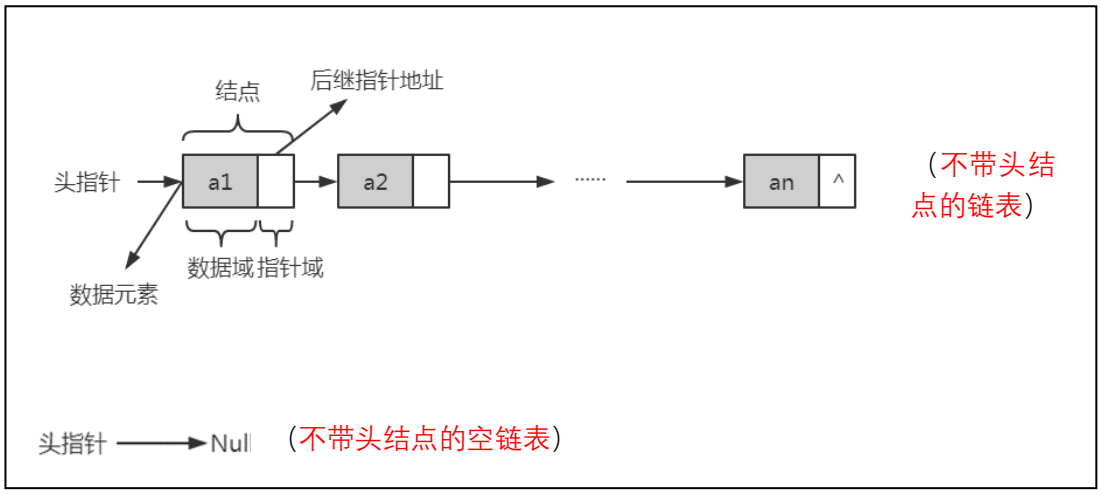




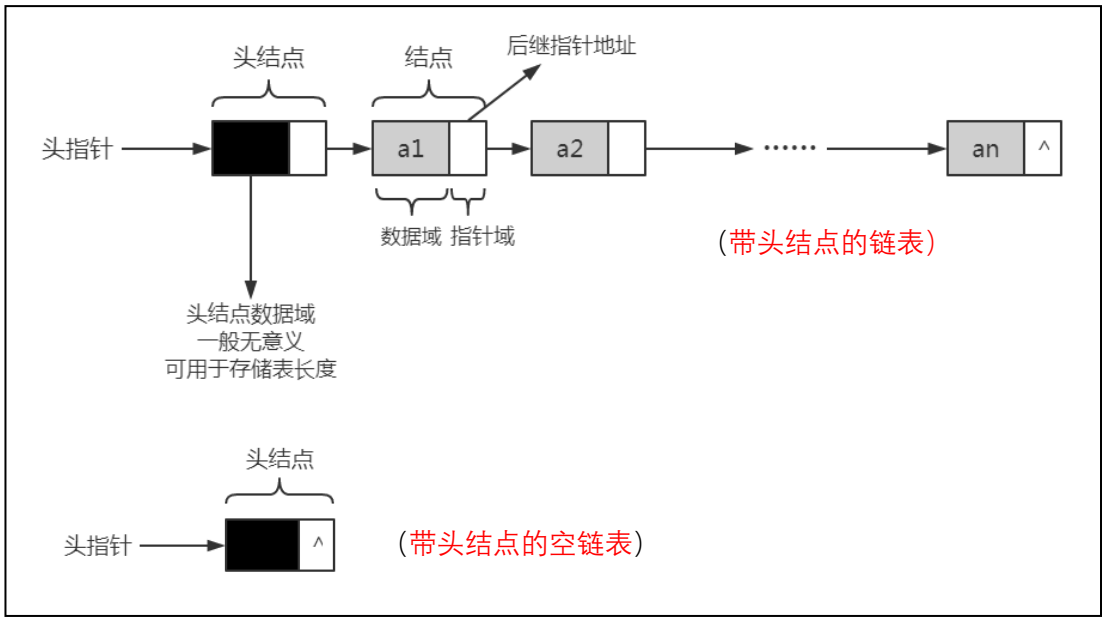
5.3 单链表的概念

链表中每个结点中只包含一个指针域，故称之为单链表。单链表的两种结构：1）不带头结点的单链表；2）带头结点的单链表。头结点的设立是为了操作的统一与方便，其放在第一个结点之前，其数据域一般无意义（当然有些情况下也可存放链表的长度、用做监视哨等）。设置头结点后，**对于在第一个结点前插入结点和删除第一个结点的操作，就与对其它结点的操作能统一了**（没设置头结点之前，这两样操作需要特殊处理。设置以后，这两样操作就与其他结点一样能统一处理了）。 **链表的头指针必不可少，整个链表的存取必须从头指针开始进行**。

1）不带头结点的单链表



2）带头结点的单链表

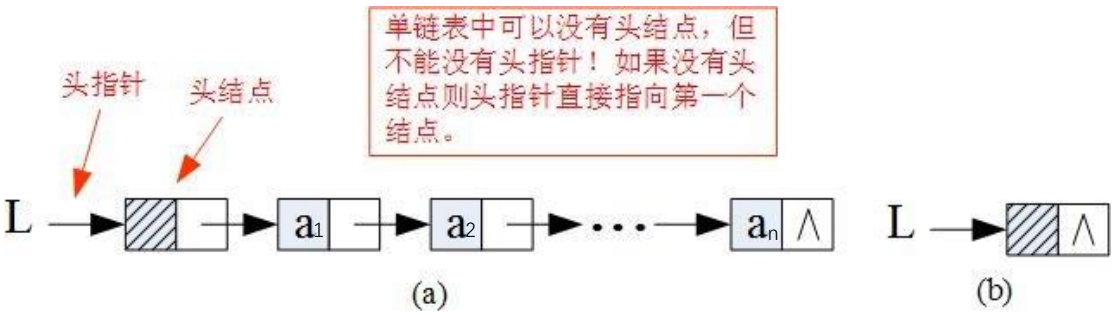


5.4 线性表的单链表存储结构

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

5.5 单链表查询数据元素

假设单链表 L 存在，查询第 *i* 个元素的值，并用 *e* 来返回该值。要在单链表 L 中取得第 *i* 个数据元素必须从头指针出发寻找。因此，**单链表是非随机存取的存储结构**。

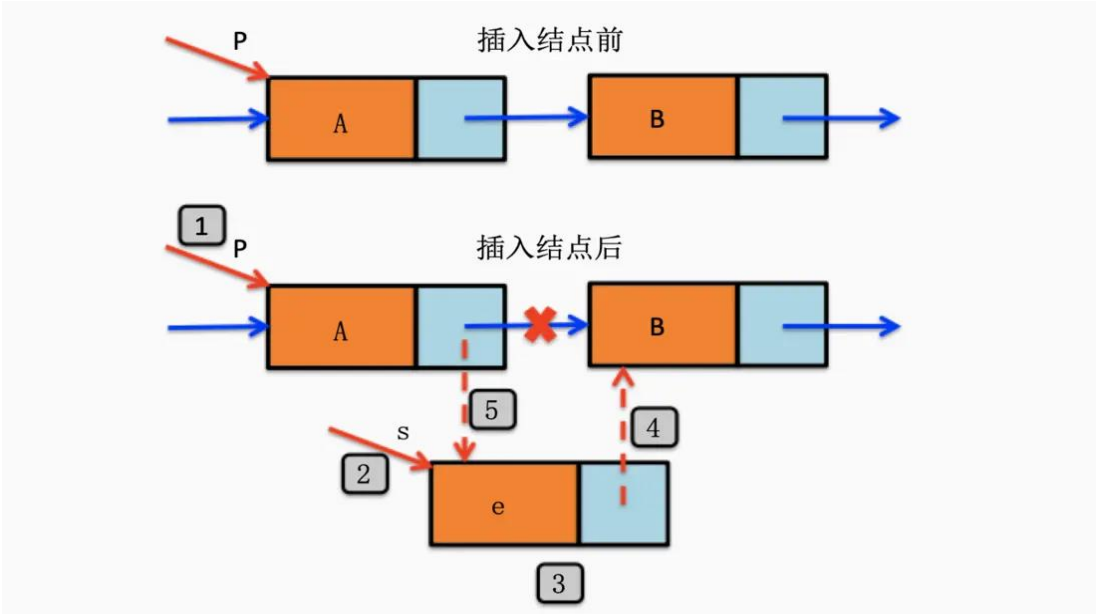


```
Status GetElem_L (LinkList L, int i, ElemType &e)
{
      //L 为带头结点的单链表的头指针
      //当第 i 个元素存在时，其值赋给 e 并返回 OK，否则返回 Error
      p=L->next; j=1;    //初始化，p 指向第一个结点，j 为计数器
      while(p&&j<i) {    //顺指针向后查询，直到 p 指向第 i 个元素或者 p 为空
          p = p -> next;
          ++j;
      }
      if(!p||j>i)  //第 i 个元素不存在
          return ERROR;
      e = p->data;    //取第 i 个元素
      return OK;
}
```

GetElem_L 算法的基本操作是比较 *j* 和 *i* 并后移指针 *p*，while 循环体中的语句频度与被查元素在表中位置有关。若 1 <= *i* <= *n*，则频度为 *i*（能找到的情况下，即 *i* 在表长的范围内，比较 *i* 次）；否则频度为 *n*（未找到的情况下，即 *i* 不在表长的范围内，需要比较 *n* 次），因此该算法的时间复杂度为 *O(n)*。

5.6 单链表插入数据元素

假设单链表 L 存在，在 L 中第 *i* 个位置（1 <= *i* <= *n*+1）之前插入值为 *e* 的新结点。单链表插入操作分为以下三步：1）**查找**：在单链表 L 中查找到第 *i*-1 个结点并用指针 *p* 指向该结点；2）**申请**：申请新结点 s，将其数据域的值置为 *e*；3）**插入**：通过修改指针域将新结点 s 挂入单链表 L。

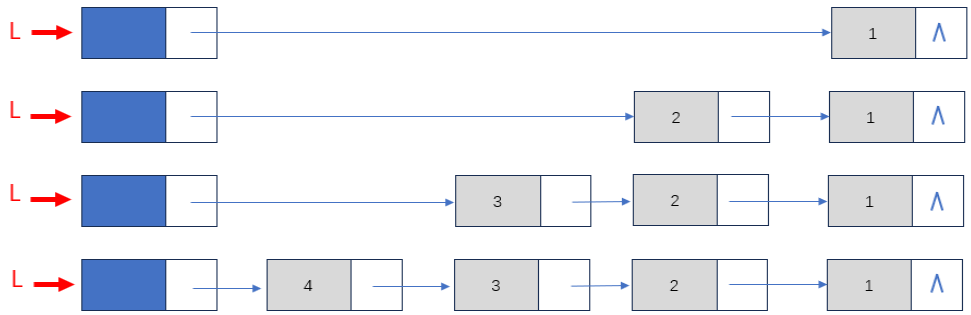


```
Status ListInsert_L ( LinkList &L, int i, ElemType e )
{
    //在带头结点的单链表 L 中第 i 个位置之前插入元素 e
    p = L; j = 0;
    while ( p && j < i -1) {    //寻找第 i-1 个结点
        p = p->next;
        ++j;
    }
    if (!p || j > i -1)    //i 小于 1 或者大于表长+1
            return ERROR;
    s = ( LinkList ) malloc ( sizeof ( LNode ) );    //动态申请一个新结点
    s->data = e; s->next = p->next; p->next = s;  //新结点采用头插法插入 L 中
    return OK;
}
```

ListInsert_L 算法的基本操作是比较 *j* 和 *i* 并后移指针 *p*，while 循环体中的语句频度与被查元素在表中位置有关。若 1 <= *i* <= *n*，则频度为 *i*-1（若 *i* <= *n*，即在表中间插入元素，则需要比较 *i*-1 次）；否则频度为 *n*（若 *i* > *n*, 即在表尾插入元素，则需要比较 *n* 次），因此该算法的时间复杂度为 *O(n)*。

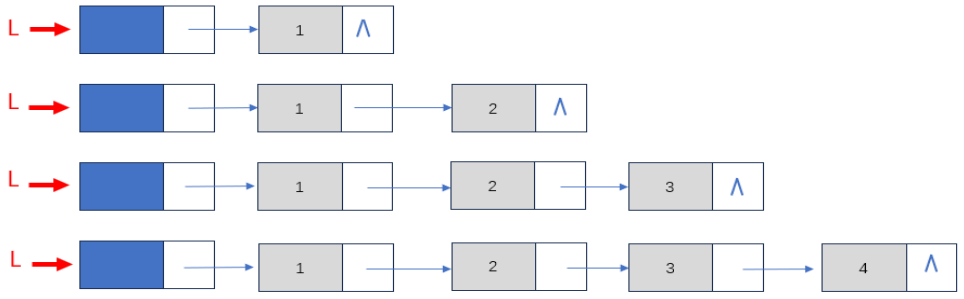
头插法：头插法的特点是输入数据的次序与生成链表结点的次序相反，即逆序插入。例如：输入数据的次序为：1, 2, 3, 4, 则生成的链表结点的次序是：4→3→2→1

输入序列：1, 2, 3, 4，采用头插法构造的链表结果如下：



尾插法：尾插法的特点是输入数据的次序与生成链表结点的次序相通，即顺序插入。例如：输入数据的次序为：1, 2, 3, 4, 则生成的链表结点的次序是：1→2→3→4

输入序列：1, 2, 3, 4，采用尾插法构造的链表结果如下：



头插法的特定：

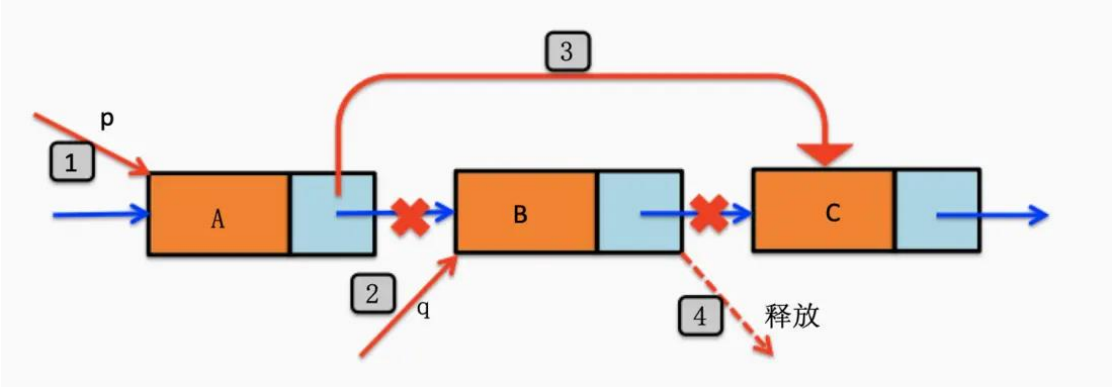
- **插入速度快（不需要遍历旧链表）**
- **首结点每次插入都会变化，首结点永远是最新的元素**
- **遍历时是按照插入相反的顺序进行**
- **由于首结点不断在变化，所以需要额外维护首结点的引用，否则找不到首结点**

尾插法的特定：

- **插入速度慢（需要遍历旧链表到最后一个元素）**
- **首结点永远固定不变**
- **遍历时是按照插入相同的顺序进行**

5.7 单链表删除数据元素

假设单链表 L 存在，删除第 *i* 个元素（**1 <= *i* <= *n***），并用 *e* 来返回该元素的值。结点删除过程分为以下两个步骤：1）**查询**：通过计数方式找到第 *i*-1 个结点并由指针 *p* 指向；2）**删除**：删除第 *i* 个结点并释放结点空间。

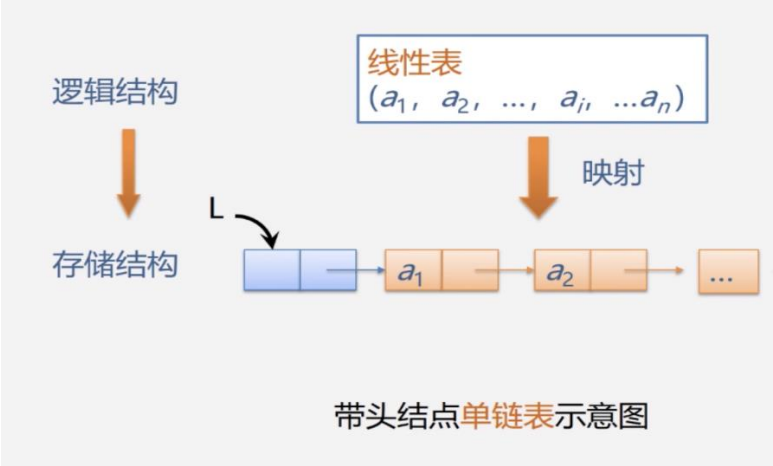



```
Status ListDelete_L (LinkList &L, int i, ElemType &e)
{
    //在带头结点的单链表 L 中，删除第 i 个元素，并由 e 返回其值
    p = L; j = 0;
    while (p->next && j<i-1) {  //查找第 i 个结点，并令 p 指向该结点的前驱
            p = p->next; ++j;
    }
    if (!(p->next) || j>i-1)   return  ERROR;   //删除位置不合理
    q = p->next; p->next = q->next;     //删除结点并释放空间
    e = q->data; free(q);
    return OK;
}
```

对于 ListDelete_L 算法，要删除第 *i* 个结点 (1 <= *i* <= *n*)，则必须先找到第 *i*-1 个结点（即要修改指针的结点），其时间复杂度为 O(*n*)。

5.8 动态生成单链表

建立线性表的链式存储结构的过程就是一个动态生成链表的过程。即从“空表”到初始状态起，依次建立各元素结点，并逐个插入链表。



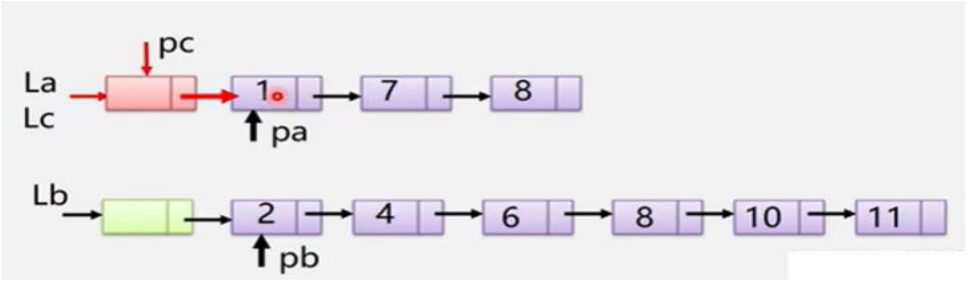
带头结点单链表示意图

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

CreateList_L 是一个从表尾到表头逆向建立单链表的算法，其时间复杂度为 O(*n*)。

5.9 单链表归并算法

将两个有序链表 La 和 Lb 合并成一个有序链表 Lc。



```
void MergeList_L (LinkList &La, LinkList &Lb, LinkList &Lc)
{
      //已知单链表 La 和 Lb 的元素按值非递减排序
      //归并 La 和 Lb 得到新的单链表 Lc, Lc 的元素也按值非递减排列
      pa=La->next;
      pb=Lb->next;
      Lc=pc= La;     //用 La 的头结点作为 Lc 的头结点
      while(pa&&pb)
      {
            if(pa->data <= pb->data) {
                  pc->next=pa; pc=pc->next; pa=pa->next;
             }
             else {
                  pc->next=pb; pc=pc->next; pb=pb->next;
             }
       }
       pc->next=pa? pa: pb;    //插入剩余段
       free(Lb);     //释放 Lb 的头结点
}
```

MergeList_L 算法的思想：需设置 3 个指针 pa, pb 和 pc, 其中 pa 和 pb 分别指向链表 La 和链表 Lb 表中当前待比较插入的结点，而 pc 指向 Lc 表中当前最后一个结点，若 pa→data <= pb→data，则 pa 所指向结点链接到 pc 所指结点之后，否则将 pb 所指结点链接到 pc 所指结点之后。MergeList_L 算法的时间复杂度为 O（La.length+Lb.length）即为 O(*n*)，只需修改指针指向，空间复杂度为 O(1)。

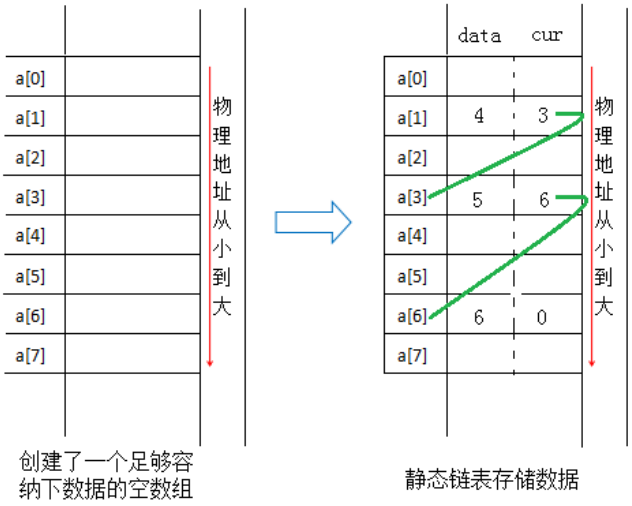
5.10 静态链表的概念

静态链表：线性存储结构的一种，兼顾顺序表和链表的优点，是顺序表和链表的升级；静态链表的数据全部存储在数组中(顺序表)，但存储的位置是随机的，数据之间的一对一关系是通过一个整型变量(称为“游标”，类似指针的功能)来维系。

静态链表中的节点 {
   数据域：用于存储数据元素的值
   游标：即数组下标，表示直接后继元素所在数组中的位置
}

```
typedef struct
{
  int data; //静态链表节点中的数据
  int cur;    //静态链表节点中的游标
}component;
```

例：使用静态链表存储数据元素 4、5、6，过程如下：



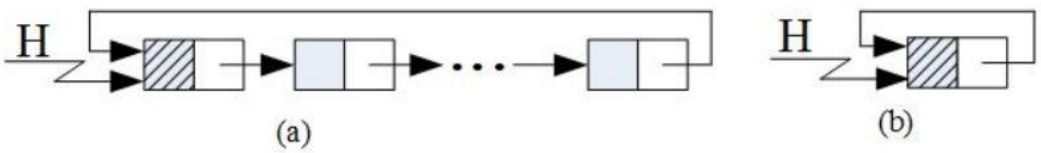
通常静态链表会将第一个数据元素放到数组下标为 1(即 a[1])的位置中。如上图中从 a[1]存储的数据元素 4 开始，通过存储的游标变量 3，可以在 a[3]中找到元素 4 的直接后继元素 5；通过元素 a[3]存储的游标变量 6，可在在 a[6]中找到元素 5 的直接后继元素 6；这样一直到某元素的游标变量为 0 截止(a[0]默认不存储数据元素)

实例 1：在静态链表 S 中实现的定位函数 LocateElem。

```
int LocateElem_SL (SLinkList S, ElemType e) {
      // 在静态单链表 L 中查询第 1 个值为 e 的元素
      // 若找到，则返回它在 L 中的位序，否则返回 0
      i = S[0].cur;
      while (i && S[i].data != e) {
            i= S[i].cur;
      }
      return i
}
```

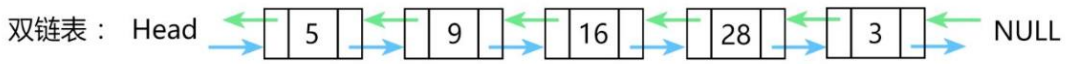
5.11 循环链表的概念

循环链表是另一种形式的链式存储结构。它的特点是表中最后一个结点的指针域指向头结点，整个链表形成一个环。循环链表的操作和线性表基本一致，差别仅在于算法中的循环条件不是 p 或者 p→next 是否为 NULL，而是它们是否等于头指针。



5.12 双向链表的概念

双向链表是指表中每个结点都有两个指针域，其中一个指向直接后继，另一个指向直接前驱。在单链表中，NextElem 的执行时间为 O(1)，PriorElem 的执行时间为 O(*n*)，而双向链表克服了单链表单向性的缺点，能够将 PriorElem 的执行时间降为为 O(1)。



5.13 双向循环链表的概念及操作包括插入和删除操作

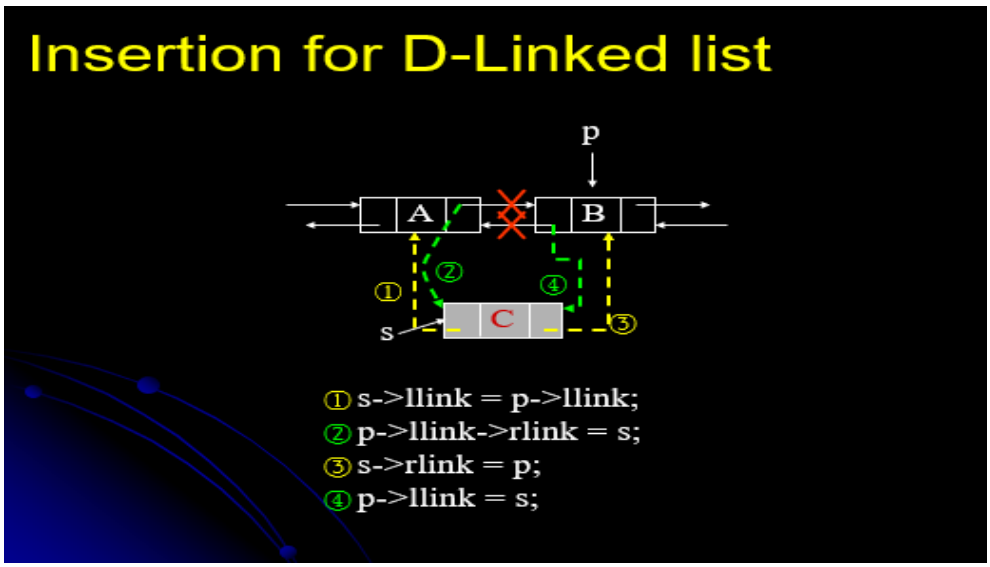
双向循环链表是在双向链表的头结点之前增加了一个指向表末尾结点的指针，将尾结点的后驱指针指向表头结点。支持从任意一个结点开始双向遍历整个链表，支持循环访问元素，双向循环链表插入和删除元素的效率比双向链表更高（适合于高频率插入和删除元素的场景）；

实例 1：双向循环链表的插入操作

结点定义：

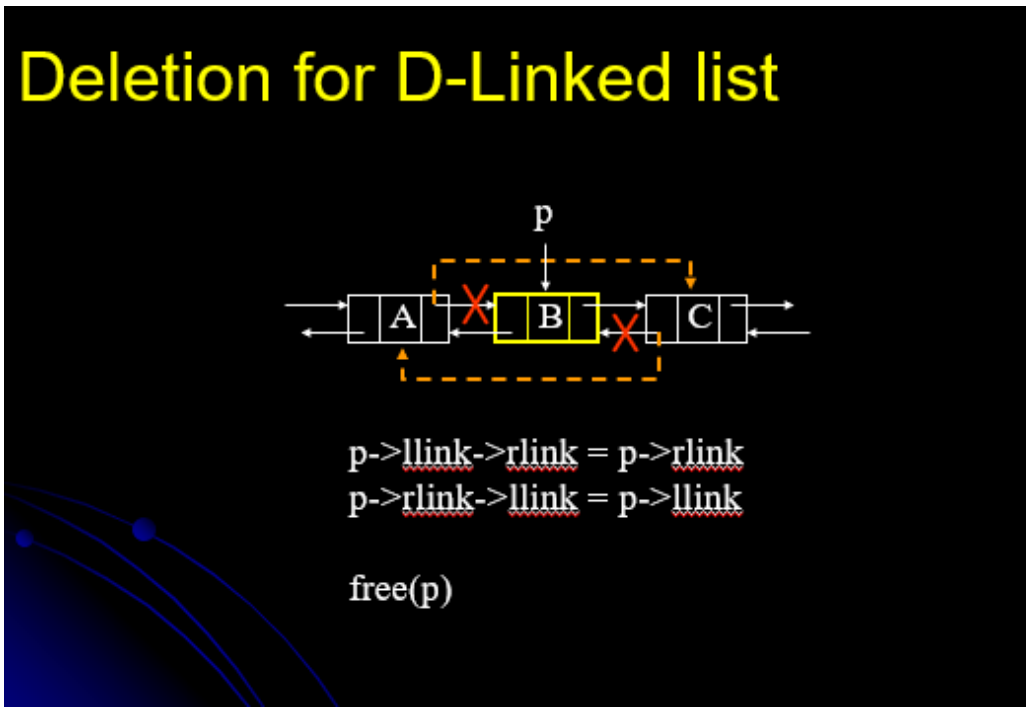
```
typedef  struct DoubleNode    /* Node definition */
{
        DataType   info;
        struct DoubleNode  *llink, *rlink;
}DoubleNode, *PDoubleNode;

typedef struct DoubleList        /* 双向链表类型 */
{
      PDoubleNode  head;      /* 指向第一个结点 */
      PDoubleNode  tail;      /* 指向最后一个结点 */
};
PDoubleList  pdlist  /* pdlist 是指向双链表类型的指针变量 */
```



```
void  insert_dbllink(PDoubleList pdlist, int i , DataType x)
/* 在带有头结点的双链表 pllist 中求第 i 个位置前插入元素 x */
{       PDoubleNode  p;
        if (!(p=GetData_dbllink(pdlist, i)))     //判断 i 的合法性
                printf("Out of range!\n");
        else
        {
                PDoubleNode  s;
                s = (P DoubleNode)malloc( sizeof( DoubleNode) );
                if ( s == NULL )
                        printf( "Out of space!!!\n" );
                else {
                        s->info = x;
                        s->llink = p->llink;          p->llink->rlink = s;
                        s->rlink = p;                 p->llink = s;
                }
        }
}
```

实例 2：双向循环链表的删除操作



```
void  delete_dbllink(PDoubleList pdlist, int i)
/* 删除带头结点的双链表 pllist 中的第 i 个元素*/
{
      PDoubleNode  p;
      if (!(p=GetData_dbllink(pdlist,i)))
            printf("Out of range!\n");
      else
      {
            p->llink->rlink = p->rlink;
            p->rlink->llink = p->llink;
            free(p);
      }
}
```

6 顺序表和链表的应用举例

实例 1：一元多项式加法

$$LA = 1 - 10x^{6} + 2x^{8} + 7x^{14}$$

$$LB = -x^{4} + 10x^{6} - 3x^{10} + 8x^{14} + 4x^{18}$$

$$LC = LA + LB = 1 - x^{4} + 2x^{6} - 3x^{10} + 15x^{14} + 4x^{18}$$

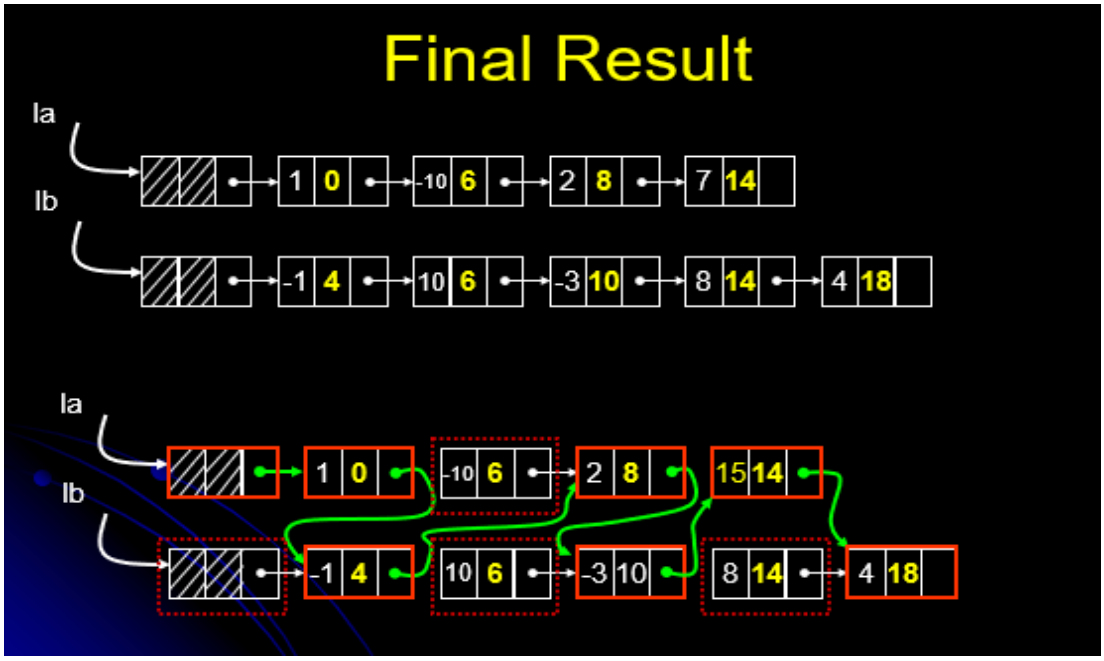
根据一元多项式相加的运算规则：对于两个一元多项式中所有指数相同的项，对应系数相加，若其和不为零，则构成"和多项式"中的一项；对于两个一元多项式中所有指数不相同的项，则分别复抄到"和多项式"中去。

按照基本操作的定义，"和多项式"链表中的结点无需另生成，而应该从两个多项式的链表中摘取。假设指针qa和qb分别指向多项式A和多项式B中当前进行比较的某个结点，则比较两个结点中的指数项，有以下三种情况：

1、指针qa所指结点的指数值<指针qb所指结点的指数值，则应摘取qa指针所指结点插入到"和多项式"链表中去。

2、指针qa所指结点的指数值>指针qb所指结点的指数值，则应摘取qb指针所指结点插入到"和多项式"链表中去。

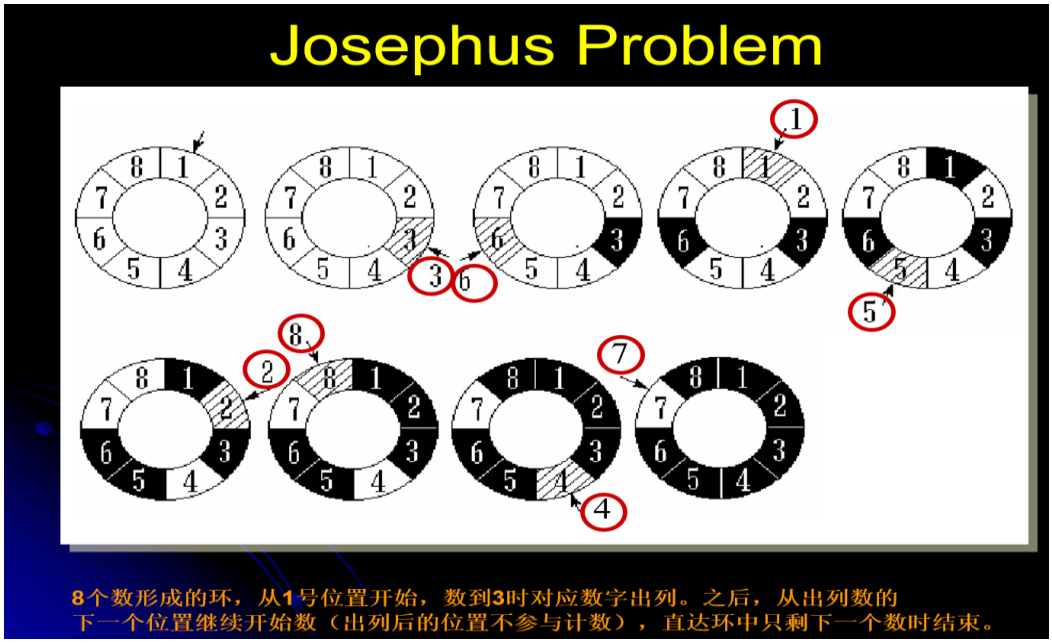
3、指针qa所指结点的指数值=指针qb所指结点的指数值，则将两个结点中的系数相加，若和数不为0，则修改qa所指结点的系数值，同时释放qb所指结点；反之，从多项式A的链表中删除响应结点，并释放指针qa和qb所指结点。




```
PLinkList AddPolyn (PLinkList la, PLinkList lb)
{
    Link  pc;    /* 用来指向新链表的尾结点的 */
    Link  hb;    /* 指向第二个链表的头结点（为了最后删除它） */
    Link  pa;    /* 指向第一个链表的当前结点 */
    Link  pb;    /* 指向第二个链表的当前结点*/
    Link  temp; /* 删除结点时做临时变量用 */
    ElemType  a,b;  /* 分别存放两个链表当前结点的数据 */
    float  sum;       /* 存放两个链表中当前结点的系数和 */
    pc = la->head;
    pa = pc->next;
    hb = lb->head;
    pb = hb->next;
    while( pa && pb)
    {
        a = pa->data;  b = pb->data;
        switch (cmp(a,b))  /*a.expn and b.expn */
        {
            case -1:       /* 第一个链表中当前结点的指数值小 */
                pc->next=pa;     /* Link the node to the end of ha */
                pc=pa;        /* move the tail pointer to pa */
                pa=pa->next;     /* move to the next node of pa */
                break;
            case 0:        /* 指数值相等 */
                sum = a.coef + b.coef;
                if ( sum != 0.0 ) {
                    pa->data.coef=sum;
```

```
                pc->next=pa;    /* Link pa to the result polyn */
                pc=pa;      /* Let ha still point to the tail */
                pa=pa->next;
            }
            else {   /* 释放 pa 所指向的结点的空间 */
                temp=pa;   /* pa is to be deleted, let temp point to it */
                pa=pa->next;    /* Let qa point to the next node */
                free(temp); /* Free the space */
            }
           /* 释放 pb 所指向的结点的空间 */
            temp = pb;
            pb = pb->next;   /* let pb point to the next node */
            free(temp);
            break;
        case 1: /* 第一个链表中当前结点的指数值大 */
            pc->next = pb;
            pc = qb;
          pb = pb->next;
          break;
      }  /* End of Switch */
   }  /* End of while(!pa && !pb) */
   pc->next = pa ? pa : pb;   /* Link the rest nodes of polynomial 1 or 2 */
   free(hb);     /* Free the head node of the 2nd polynomial  */
   return (la);
}  /* End of AddPolyn() */
```

实例 2：Josephus 环问题



约瑟夫环(约瑟夫问题)是一个数学的应用问题:已知 n 个人(以编号 1，2，3...n 分别表示)围坐在一张圆桌周围。从编号为 k 的人开始报数，数到 m 的那个人出列;他的下一个人又从 1 开始报数，数到 m 的那个人又出列;依此规律重复下去，直到圆桌周围的人全部出列。

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

typedef struct Node {
    int data;
    struct Node *next;
} Node;

Node* createList(int n) {
    Node *head, *prev, *cur;
    head = (Node*) malloc(sizeof(Node));
    head->data = 1;
    prev = head;
    for (int i = 2; i <= n; i++) {
        cur = (Node*) malloc(sizeof(Node));
        cur->data = i;
        prev->next = cur;
        prev = cur;
    }
    prev->next = head; // 将链表尾部连接到链表头部，形成循环链表
    return head;
}

void josephus(int m, int n) {
    Node *p, *q;
    p = createList(n);
    q = p;
    while (q->next != p) {
        q = q->next;
    }
    while (p != q) {
        for (int i = 1; i < m; i++) {
            p = p->next;
            q = q->next;
        }
        printf("%d ", p->data);
        q->next = p->next;
        free(p);
        p = q->next;
```

```
    }
    printf("%d\n", p->data);
    free(p);
}

int main() {
    int m = 3, n = 10;
    josephus(m, n);
    return 0;
}
```