线性表

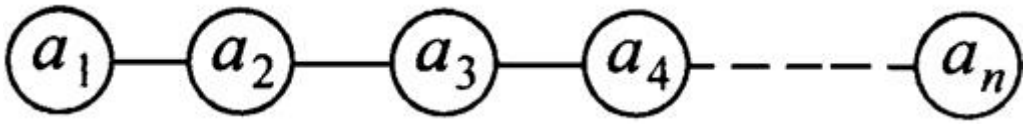
1、相关概念、特点和相关操作

定义

是一种最简单的线性结构。

- ☑ 线性表：简称表，是n（n≥0）个具有相同类型的数据元素的有限序列。
- ☑ 线性表的长度：线性表中数据元素的个数。
- ☑ 空表：长度等于零的线性表，记为：L=( )。
- ☑ 非空表记为：L =（a1, a2 , ..., ai-1, ai , ..., an)

图示



特征

n个数据元素的有限序列

线性表的基本特性：

1. 集合中必存在唯一的一个“第一元素”
2. 集合中必存在唯一的一个“最后元素”
3. 除最后元素在外，均有唯一的后继；
4. 除第一元素之外，均有唯一的前驱。

相关操作

创建、销毁、插入、删除、查找、遍历

抽象数据类型ADT

```
ADT List
{
    Data:
    Operation:
        InitList(&L)
        CreateList(&L)
        ListEmpty(L)
        ListLength(L)
        LocateElem(L,e)
        PriorElem(L,cur_e,&pre_e)
        NextElem(L,cur_e,&pre_e)
        ListInsert(&L,i,e)
        ListDelete(&L,i,&e)
        GetElem(L,i,&e)
        ListTraverse(L)
```

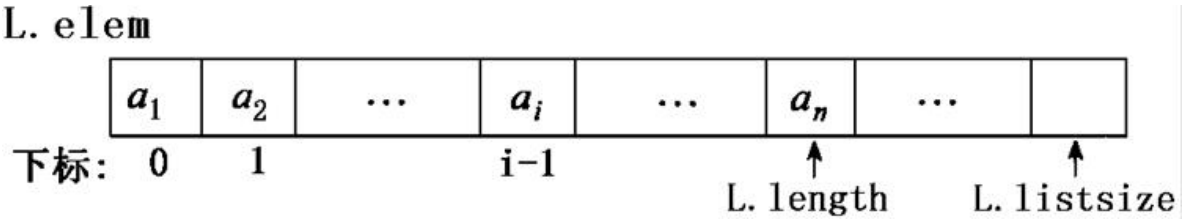
```
        DestroyList(&L)
    }//ADT List
```

2、顺序存储

定义

- 顺序存储结构：用一组地址连续的存储单元依次存储**数据结构**(在这里是线性表)里各个元素，这种存储结构称为线性表的顺序存储结构。
- 顺序表：按照顺序存储结构存储的线性表。

图示



基本操作的实现及算法复杂度分析

顺序表静态存储数据结构

```
[in SqList.h]
# define LIST_INIT_SIZE 100     //(默认)最大存储容量
typedef struct {
    ElemType  elem [LIST_INIT_SIZE];    //存储数据元素的一维数组
    int length;                         //线性表的当前长度
} SqList;
```

顺序表动态存储数据结构

```
[in SqList.h]
# define LISTINCREMENT  10          //(默认)增补空间量
typedef struct {
    ElemType  *elem;            // 存储数据元素的一维数组
    int length;                 // 线性表的当前长度
    int listsize;               // 当前分配的数组容量（以ElemType为单位）
    int incrementsize;          // 增补空间量（以ElemType为单位）
} SqList;
```

顺序表初始化操作

```
[in SqList.h]
void InitList_Sq( SqList &L, int maxsize=LIST_INIT_SIZE, int
incresize=LISTINCREMENT ) // 构造一个最大容量为maxsize的顺序表 L
{
     L.elem=(ElemType *)malloc(maxsize*sizeof(ElemType)); // 分配一个最大容量为
maxsize的数组空间
     if(!L.elem)  exit(1);            // 存储分配失败
      L.length=0;                     // 顺序表中当前所含元素个数为 0
```

```
      L.listsize=maxsize;             // 该顺序表可以容纳maxsize个数据元素
      L.incrementsize=incresize;      // 需要时可扩容incresize个元素空间
}// InitList_Sq

void InitList_Sq(int m)//简化写法，考试用
{
  elem = new T[m]；//动态申请一组连续的内存空间
  if(!elem) throw “内存分配失败”；
  length = 0；    // 长度为0
  listsize = m；  // 容量为已申请存储空间容量
}
```

顺序表求表长操作

```
[in SqList.h]
int ListLength_Sq(SqList L)
{
    return L.length;
}// ListLength_Sq
```

判断顺序表是否为空

```
[in SqList.h]
bool ListEmpty_Sq(SqList L)
{
    if(L.length==0)
    return true;
    else return false;
}
```

顺序表定位元素(返回与e值相等的元素下标)

```
[in SqList.h]
int LocateElem_Sq( SqList L, ElemType e)
{
    for(int i=0;i<L.length;i++)
        if(L.elem[i]==e)
            return i;   // 找到满足判定的数据元素为第 i 个元素
    return -1;                      // 该线性表中不存在满足判定的数据元素
 }//LocateElem_Sq
```

顺序表前插

在顺序表L的第i个元素之**前**插入新的元素e，若表中当前容量不足，则按预定义的增量扩容

```
[in SqList.h]
bool ListInsert_Sq(SqList &L, int i, ElemType e)
{
    int j;
```

```
        if(i<0||i>L.length)  return false;  // i值不合法
        if(L.length>=L.listsize)           // 当前存储空间已满，增补空间
        {
            L.elem=(ElemType *)realloc(L.elem,
(L.listsize+L.incrementsize)*sizeof(ElemType));
            if(!L.elem) exit(1);                    // 存储分配失败
            L.listsize+=L.incrementsize;            // 当前存储容量增加
        }
    for(j=L.length;j>i;j--)                // 被插入元素之后的元素后移
        L.elem[j]=L.elem[j-1];

    L.elem[i]=e;                           // 插入元素e
    L.length++;                            // 表长增1
    return true;
}// ListInsert_Sq
```

顺序表后插

在顺序表L的第i个元素之**后**插入新的元素e，若表中当前容量不足，则按预定义的增量扩容

```
[in SqList.h]
bool ListInsert_Sq(SqList &L, int i, ElemType e)
{
    int j;
        if(i<0||i>L.length)  return false;  // i值不合法
        if(L.length>=L.listsize)           // 当前存储空间已满，增补空间
        {
            L.elem=(ElemType *)realloc(L.elem,
(L.listsize+L.incrementsize)*sizeof(ElemType));
            if(!L.elem) exit(1);                    // 存储分配失败
            L.listsize+=L.incrementsize;            // 当前存储容量增加
        }
    int target=i+1;
    for(j=L.length;j>target;j--)               // 被插入元素之后的元素后移
        L.elem[j]=L.elem[j-1];

    L.elem[target]=e;                          // 插入元素e
    L.length++;                            // 表长增1
    return true;
}// ListInsert_Sq
```

顺序表尾插

在顺序表L的最后一个元素之**后**插入新的元素e，若表中当前容量不足，则按预定义的增量扩容

```
[in SqList.h]
bool ListInsert_Sq(SqList &L, int i, ElemType e)
{
    int j;
        if(i<0||i>L.length)  return false;  // i值不合法
        if(L.length>=L.listsize)           // 当前存储空间已满，增补空间
        {
            L.elem=(ElemType *)realloc(L.elem,
(L.listsize+L.incrementsize)*sizeof(ElemType));
```

```
            if(!L.elem) exit(1);                    // 存储分配失败
            L.listsize+=L.incrementsize;            // 当前存储容量增加
        }
    L.elem[L.length]=e;                             // 插入元素e
    L.length++;                                 // 表长增1
    return true;
}// ListInsert_Sq
```

顺序表删除元素操作

在顺序表L中删除第i个元素，并用e返回其值

```
[in SqList.h]
bool ListDelete_Sq(SqList &L,int i, ElemType &e)
{
      int  j;
if(i<0||i>L.length) return false;             // i值不合法
if(L.length<=0)  return false;                 // 表空无数据元素可删
     e=L.elem[i];                                   // 被删除元素的值赋给e
      for(j=i+1;j<=L.length-1;j++)                  // 被删除元素之后的元素前移
        L.elem [j-1]=L.elem [j];
      L.length--;                                   // 表长减1
      return true;
}// ListDelete_Sq
```

顺序表取元素操作

取出顺序表L中第i个元素，并用e返回其值。

```
[in SqList.h]
bool GetElem_Sq(SqList L,int i, ElemType &e)
{     // 取出顺序表L中第i个元素，并用e返回其值。
    if(i<0||i>L.length) return false;           // i值不合法
    if(L.length<=0)  return false;              // 表空无数据元素可取
    e=L.elem[i];                                // 被取元素的值赋给e
    return true;
}// GetElem_Sq
```

顺序表遍历输出各元素

```
[in SqList.h]
void ListTraverse_Sq(SqList L)
{
    int i;
    for(i=0;i<L.length;i++)
      cout<<setw(6)<<L.elem[i];
    cout<<endl;
}// ListTraverse_Sq
```

删除顺序表

```
[in SqList.h]
void DestroyList_Sq(SqList &L)
{       // 释放顺序表L所占存储空间
      free(L.elem);
      L.elem=NULL;
      L.listsize=0;
      L.length=0;
}// DestroyList_Sq
```

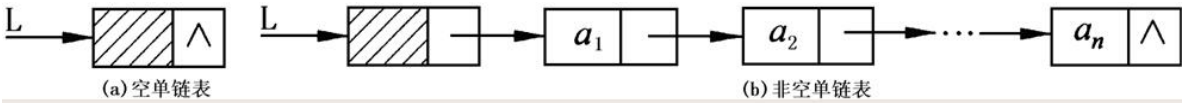
3、链式存储

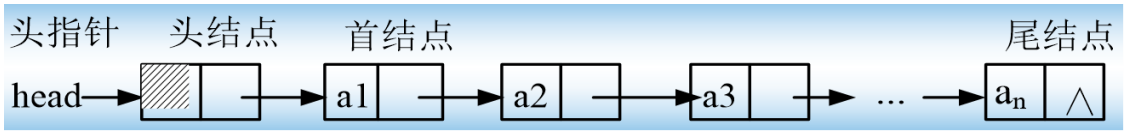
单链表

定义

- 链式存储结构：用一组任意的存储单元存储**数据结构**（这里是线性表)里的各元素(这组存储单元可以是地址连续的，也可以是不连续的)，并且每个存储元素有一个数据域，一个（或多个）指针域，数据域用来存储元素内容，指针域用来反映元素的逻辑关系(前后位置关系)，这种存储方式成为链式存储（也叫非随机存取的存储结构）。
- 链表：按照链式存储结构存储的线性表。
- 单链表：各元素只有一个指针域的链表（只含有一个指针域）

逻辑图示(带头结点的单链表)



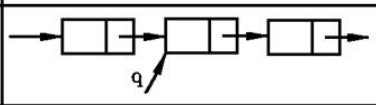
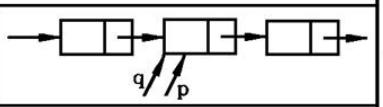
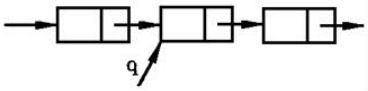
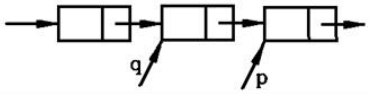
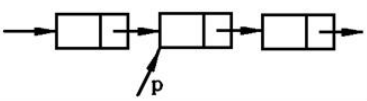
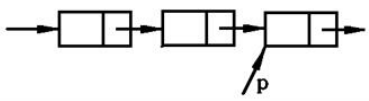
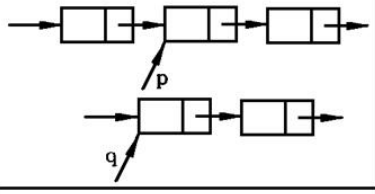
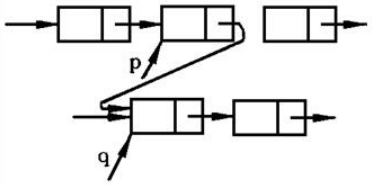
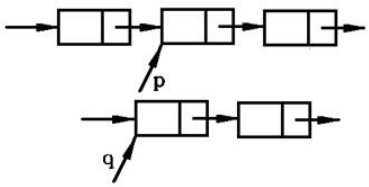
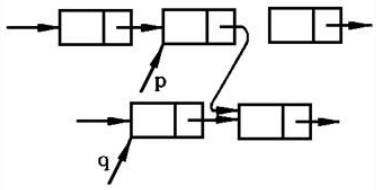
- 

数学表示

- 寻址 一般用p,q等字母表示操作性指针，例如
  如果 p = &a[i]，即 a[i] = p->data（或*(p).data*）
  那么
  a[i]的下一个元素的地址是： *p -> next（或*(p).next）
  a[i]的下一个元素的值是： p->next->data

注意事项

- 链表的元素一般称为“结点”。
- 头结点在链表中并不是必须的，仅仅是为了操作上的方便。
- 结点a[i]指其数据域为a[i]的结点，而p则是指向a[i]的指针，俗称"p结点"。
- **链表有带头结点的链表和不带头结点的链表之分，带头结点的链表第一个元素为头结点a0，L指向a0; 不带头结点的链表第一个元素为头结点a1，L指向a1。**
- 单链表的操作

操作内容	执行操作的语句	执行之前	执行之后
指针 指向结点	p=q;		
指针 指向后继	p=q->next;		
指针移动	p=p->next;		
链指针 改接	p->next=q;		
链指针 改接后继	p->next=q->next;		

单链表的数据结构

```
[in LinkList.h]
typedef struct Node {
    ElemType data;    //数据域
    struct  Node *next;   //指针域
}LNode,*LinkList;                  // LinkList为结构体指针类型
```

单链表的初始化

```
[in LinkList.h]
void InitList_L(LinkList &L)
    {
        L=(LNode *)malloc(sizeof(LNode));  // 申请存放一个结点数据所需要的内在空间
        if(!L)   exit(1);                  // 存储分配失败
        L->next=NULL;                     // 表头结点的指针域置空
    }// InitList_L
```

求单链表的长度

// L为带头结点的链表的头指针，函数返回L所指链表的长度

```
[in LinkList.h]
int ListLength_L( LinkList L )
    {
        LinkList  p;
        int k=0;
        p=L->next;                                  // p指向链表中的第一个结点
         while(p)
         { k++;  p=p->next; }                 // k计非空结点数
         return k;
    }// ListLength_L
```

单链表的定位操作

在L所指的单链表中查找第一个值和e 相等的结点，若存在，则返回其指针；

```
[in LinkList.h]
LNode *LocateElem_L( LinkList L,ElemType e)
    {
         LinkList  p;
         p=L->next;                                      // p指向链表中的第一个结点
         while (p&&p->data!=e ) p=p->next;
         return p;
    }// LocateElem_L
```

单链表的定位操作

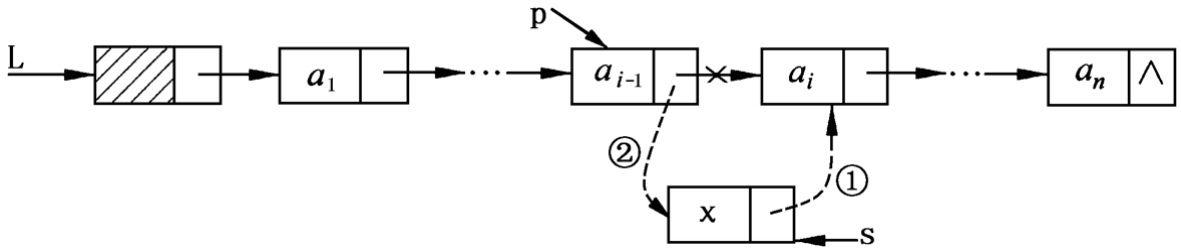
// 在L所指的单链表中查找第一个值和e相等的结点，若存在，则返回元素顺序

```
[in LinkList.h]
int LocateElem_L_2( LinkList L,ElemType e)
    {
         int i=-1;
         LinkList  p;
         p=L->next;                                    // p指向链表中的第一个结点
         while (p&&p->data!=e) {p=p->next;i++;}
         return i;
    }// LocateElem_L_2
```

（重点*）单链表插入元素操作

在带有头结点的单链表L中的第i个结点之前插入元素e

```
[in LinkList.h]
bool ListInsert_L( LinkList &L, int i, ElemType e)
    {
        LinkList p,s;
        int j;
        p=L;  j=0;
        while(p->next&&j<i-1)  { p=p->next; j++; }   // 寻找第i-1个结点,并让p指向此
结点
        if(j!=i-1)   return false;                    // i的位置不合理
        if((s=(LNode *)malloc(sizeof(LNode)))==NULL) exit(1);  // 存储分配失败
        s->data=e;
        s->next=p->next;  p->next=s;                   // 插入新结点
        return true;
    }// ListInsert_L
```



注意. while(p)和while(p->next)的区别：

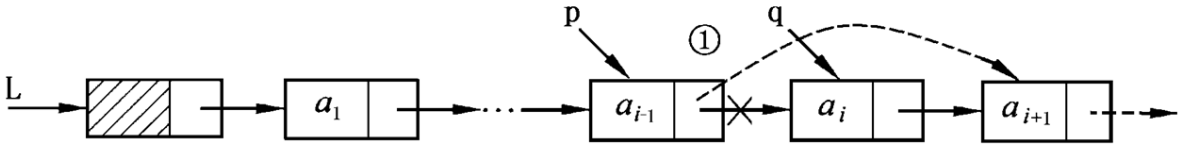
我们在操作链表的时候通常会用到

```
while(p)
{p=p->next;}
和
while(p->next)
{p=p->next;}
```

前者是让工作指针p访问完整个链表，循环结束时，p并没有指向最后一个元素an，而是超出了控制范围，后者是让工作指针p访问完除最后一个元素之外的所有元素，循环结束时，p指向最后一个元素an。

（重点*）单链表删除元素操作

```
[in LinkList.h]
bool ListDelete_L( LinkList &L, int i, ElemType &e)
    {    // 删除带有头结点的单链表L中的第i个结点，并让e返回其值
        LinkList p,q;
        int j;
        p=L;  j=0;
        while(p->next->next&&j<i-1){ p=p->next; j++; } //寻找第i-1个结点,并让p指向此
结点
        if(j!=i-1)   return false;                    // i的位置不合理
        q=p->next;                                    // q指向其后继
        p->next=q->next;                              // 删除q所指结点
        e=q->data;    free(q);
        return true;
    }// ListDelete_L
```



单链表取元素操作

取出单链表L中第i个元素，并用e返回其值

```
[in LinkList.h]
bool GetElem_L(LinkList L,int i, ElemType &e)
    {    // 取出单链表L中第i个元素，并用e返回其值
        LinkList p;
        int j;
        p=L;  j=0;
        while(p->next&&j<i){ p=p->next; j++; }  // 寻找第i个结点,并让p指向此结点
         if(j!=i)   return false;                        // i的位置不合理
         e=p->data;                                       // 被取元素的值赋给e
         return true;
    }// GetElem_L
```

创建单链表（尾插）

已知一维数组A[n]中存有线性表的数据元素，利用尾插法创建单链表L

```
[in LinkList.h]
void CreateList_L_Rear(LinkList &L,ElemType a[],int n )
    {   // 已知一维数组A[n]中存有线性表的数据元素，利用尾插法创建单链表L
        LinkList p,q;   int i;
        L=(LinkList)malloc(sizeof(LNode));     // 创建立头结点
        q=L;                                  // q始终指向尾结点，开始时尾结点也是头结点
        for(i=0;i<n;i++)
        {   p=(LinkList)malloc(sizeof(LNode));          // 创建新结点
            p->data=a[i];                               // 赋元素值
            q->next=p;                                  // 插入在尾结点之后
            q=p;                                         // q指向新的表尾
        }
        q->next=NULL;                                    // 表尾结点next域置空
     }// CreateList_L_Rear
```

创建单链表（头插）

已知一维数组A[n]中存有线性表的数据元素，利用头插法创建单链表L

```
[in LinkList.h]
void CreateList_L_Front(LinkList &L,ElemType a[],int n )
    {
        LinkList p;   int i;
        L=(LinkList)malloc(sizeof(LNode));                  //创建立头结点
        L->next=NULL;
        for(i=n-1;i>=0;i--)
        {    p=(LinkList)malloc(sizeof(LNode));               //创建新结点
             p->data=a[i];                                    // 赋元素值
             p->next=L->next;                                // 插入在头结点和第一个结点之
间
             L->next=p;
        }
    }// CreateList_L_Front
```

遍历输出单链表各元素数据

```
[in LinkList.h]
 void ListTraverse_L(LinkList L)
    {
         LinkList p=L->next;
         while(p)                  //遍历完整个单链表
      {  cout<< setw(6)<<p->data;
         p=p->next;
      }
        cout<<endl;
    }// ListTraverse_L
```

销毁单链表

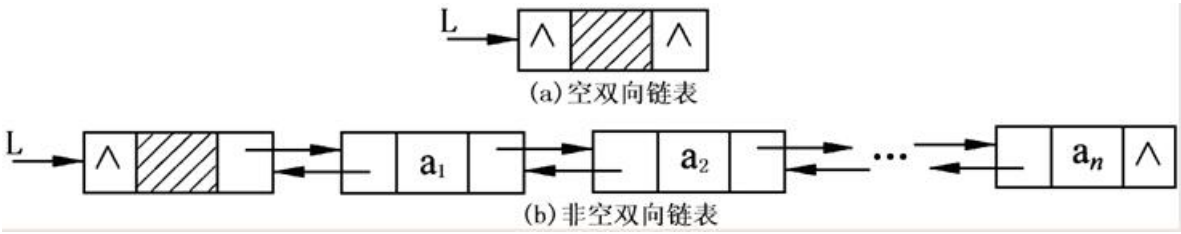
```
[in LinkList.h]
void DestroyList_L(LinkList &L )
    {
        LinkList p,p1;
         p=L;
        while(p)            //遍历完整个单链表
        {   p1=p;
            p=p->next;
            free(p1);
        }
             L=NULL;
    }// DestroyList_L
```

双向链表

定义

- 双（向）链表就是每个结点中含有两个指针域的链表，其中一个指针域存放其前趋结点的地址，另一个指针域存放其后继结点的地址。

逻辑图示



- **头结点**：双链表中第一个结点(上图中的a1之前的结点a0)
- **表头指针**：存放双链表中第一个结点的地址的指针。(指向a0的指针)
- **开始结点**：存放双链表的第一个元素的结点。(a1)
- **表尾结点**：双链表中最后一个结点，表尾结点的指针域指针为空。(an)

数学表示

- 寻址 一般用p,q等字母表示操作性指针，可用前驱指针域prior和后继指针域next访问表中任意节点，例如：
  p = &a[i]（即a[i] = p->data）
  那么
  a[i]的上一个元素地址是 p -> prior，上一个元素值是 p->prior->data
  a[i]的下一个元素地址是 p -> next，下一个元素值是 p->next->data
  一般的，有
  p->next-prior = p = p->prior->next

注意事项

- 节点p的存储地址既存放在其前驱结点的后继指针域中，也存在其后继节点的前驱指针域中。因此可随意在其上向前或向后移动，使得操作更加容易。
- 操作上比单链表更加便利，但存储开销增大。
- 双(向)链表是非循环的，有首尾节点，注意与双向循环链表区别开来

双链表的数据结构

```
[in DuLinkList.h]
typedef struct DuNode {
    ElemType data;
    struct  DuNode *prior; //前驱指针域
    struct  DuNode *next;  //后继指针域
}DuLNode,*DuLinkList;
```

双链表的初始化

```
[in DuLinkList.h]
void InitList_Dul(DuLinkList &L)
{
    L=(DuLNode *)malloc(sizeof(DuLNode));
    if(!L)   exit(1);
    L->next=NULL;      //后继指针置空
    L->prior=NULL;   //前驱指针置空
}// InitList_Dul
```

求双链表的长度

// DL为带头结点的链表的头指针，函数返回L所指链表的长度

```
[in DuLinkList.h]
int ListLength_Dul(DuLinkList L )
    {
        DuLinkList  p;
        int k=0;
        p=L->next;
         while(p)              //当p所指向的结点还有值
         { k++;  p=p->next; }
         return k;
    }// ListLength_Dul
```

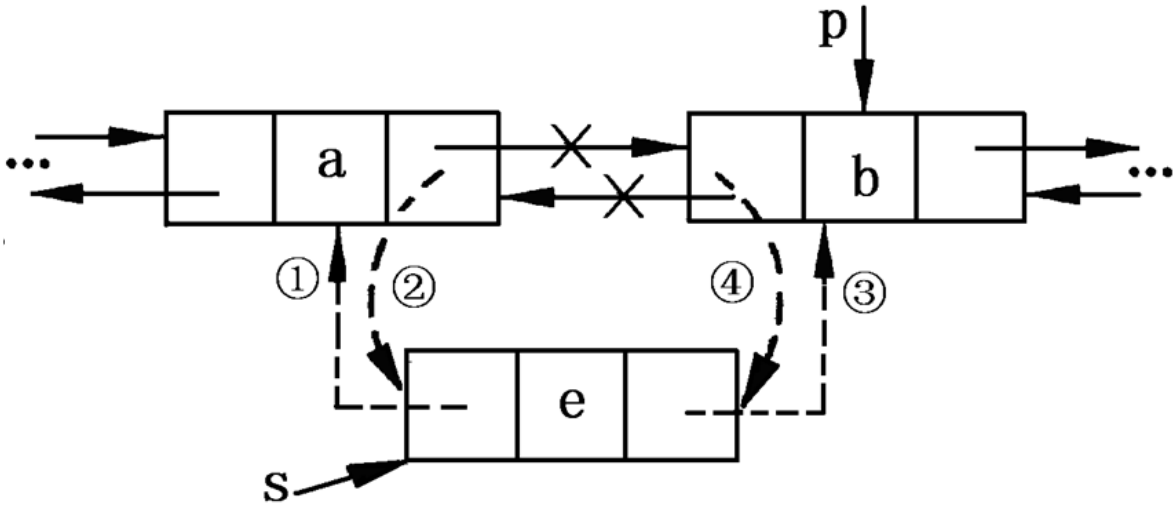
双链表的定位操作

在dL所指的双链表中查找第一个值和e相等的结点，若存在，则返回其指针；

```
[in DuLinkList.h]
int LocateElem_DL(DuLinkList L,ElemType e)
{
    DuLinkList  p;
    int k=0;
    p=L->next;
    while(p&&e!=p->data)
    {k++;p=p->next;}
    return p;
}
```

---

*（重点）*双链表插入元素操作



在带头结点的双向链表DL中第i个结点之前插入元素e

```
[in DuLinkList.h]
bool ListInsert_DuL(DuLinkList &L,int i,ElemType e)
{
   DuLinkList p,s,q;
   int j;
```

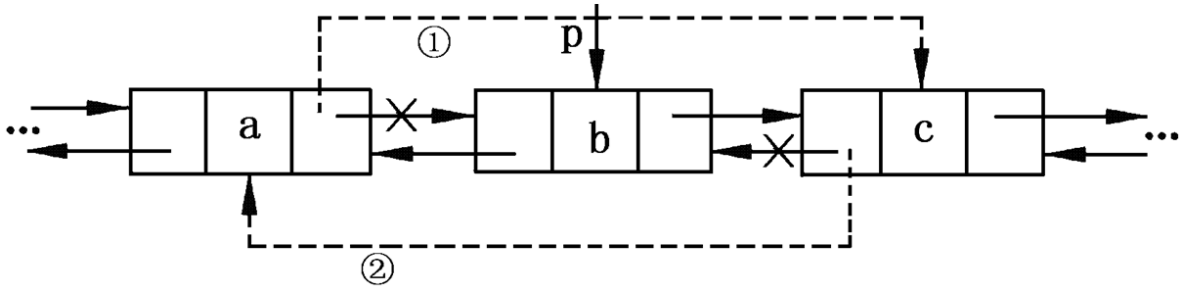
```
    q=L;  j=0;
    while(q->next&&j<i-1)  { q=q->next; j++; }    // 寻找第i-1个结点,并让q指向此结点
    if(j!=i-1)   return false;                    // i的位置不合理
    s=(DuLNode *)malloc(sizeof(DuLNode));
    if(!s)   exit(1);                             // 存储分配失败
    s->data=e;
    if(q->next)                                    // 插入的位置不是表尾
    {
         p=q->next;                              // p指向待插入的位置
         s->prior=p->prior;                      // 修改指针
         p->prior->next=s;
         s->next=p;
         p->prior=s;
    }
    else                                         // 插入的位置是表尾
    {
         q->next=s;
         s->prior=q;
         s->next=NULL;
    }
    return true;
}
```

- 注意，这里的语句4不能换到语句1之前，另外还要区分是不是插到末尾。

---

*（重点）*双链表删除元素操作

删除带有头结点的双向链表DL中的第i个结点，并让e返回其值




```
[in DuLinkList.h]
bool ListDelete_Du(DuLinkList &L, int i, ElemType &e)
{  int j = 0;
   DuLinkList p=L;
   while(p->next&&j<i){ p=p->next; j++; }    // 寻找第i个结点,并让p指向此结点
   if(j!=i)   return false;                  // i的位置不合理
   if(p->next)                               // 待删除的不是表尾结点
     p->next->prior=p->prior;                // 结点p的前驱作为结点p的后继的前驱
     p->prior->next=p->next;                 // 结点p的后继作为结点p的前驱的后继

     e=p->data;
     free(p);
     return true;
 }
```

双链表取元素操作

取出双链表DL中第i个元素，并用e返回其值

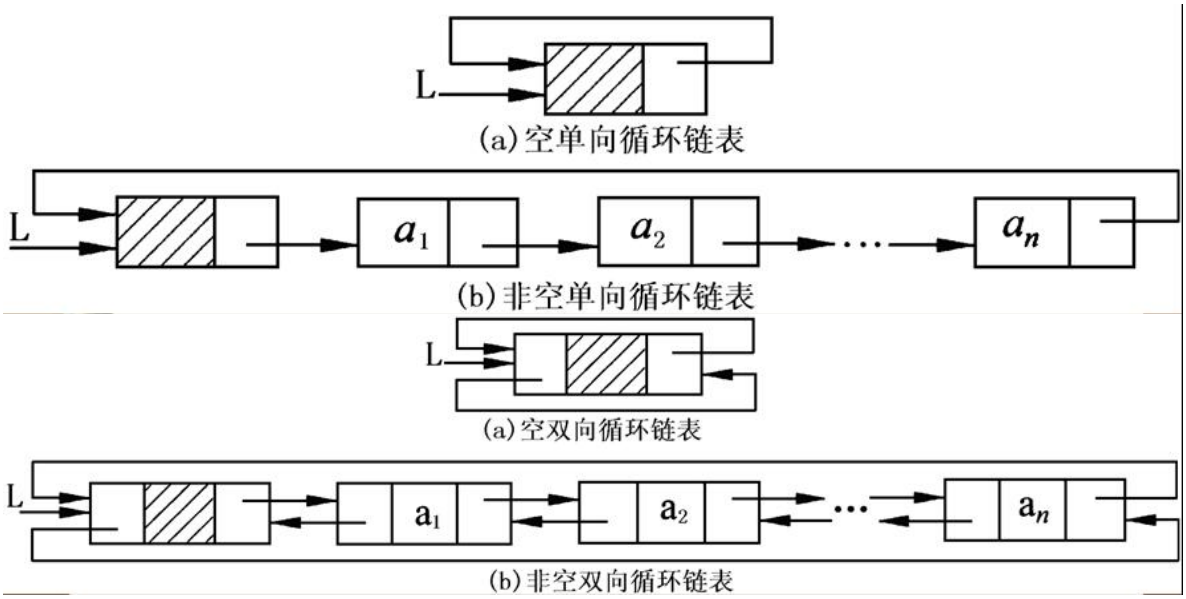
```
[in DuLinkList.h]
bool GetElem_DL(DuLinkList L,int i, ElemType &e)
{
        DuLinkList p;
        int j;
        p=L;  j=0;
        while(p->next&&j<i){ p=p->next; j++; }
// 寻找第i个结点,并让p指向此结点
         if(j!=i)   return false;                          // i的位置不合理
         e=p->data;                                         // 被取元素的值赋给e
         return true;
}
```

循环链表

定义

- 即约瑟夫环，是另一种形式的链式存储结构，它的基本思想是利用结点的空指针域，在链尾和链头之间增加链接，形成环状数据结构。
- 一般有两种形式的循环链表，即单向循环链表和双向循环链表。
- 单向循环链表中，表尾结点的指针域不为空，回指第一个结点，整个链表形成一个环。
- 在双向循环链表中，除了表尾结点的后继指针域回指第一个结点外，同时表头结点的前驱指针域回指表尾结点，这样在链表中构成了两个环。

逻辑图示



注意事项

- 在初始化操作中，分别将语句L->next = NULL;L- prior = NULL 改成: L->next = L; L->prior = L;
- 在其他操作中，循环控制条件不是判断p,p->next或p->next->next是否为空，二十判断他们是否等于头指针
- 工作指针p的初值应该与循环控制条件相对应，也就是说，若赋值语句为"p=L"，则循环控制表达式为"p->next!=L"，若赋值语句为"p=L->next;";则循环控制表达式为"p!=L"。

双向循环链表的数据结构

同双向链表

双向循环链表初始化

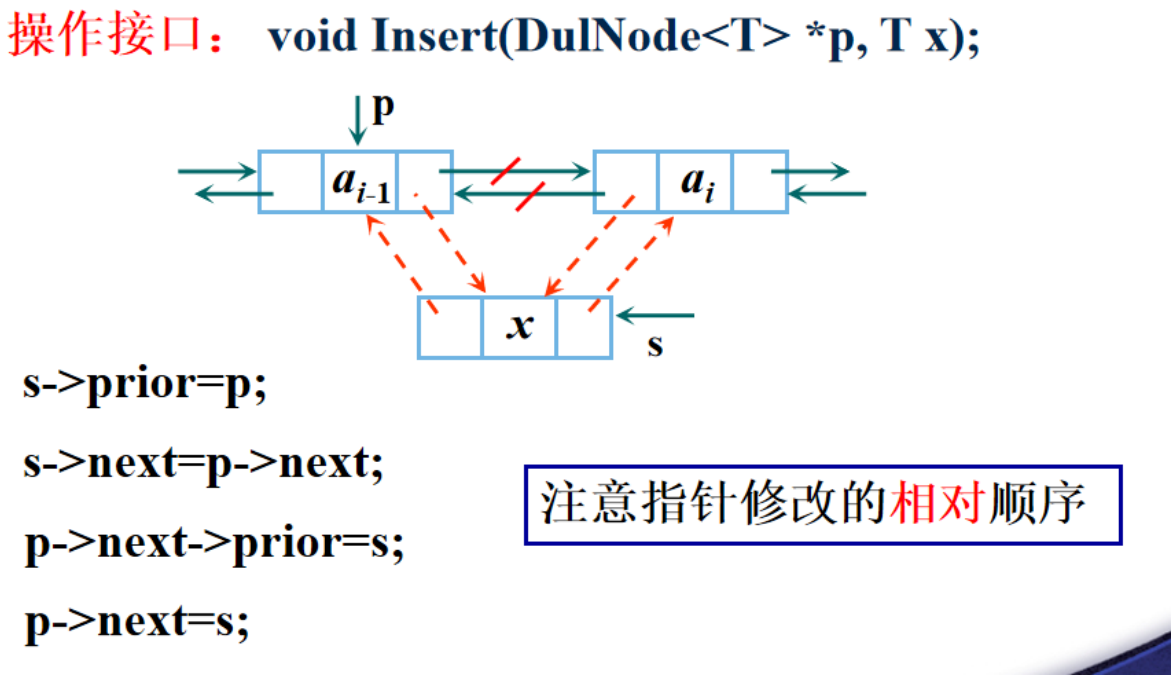
初始化双向循环链表DL

```
[in DuLinkList_C.h]
void InitList_DuL_C(DuLinkList &L)
{
  L=(DuLNode *)malloc(sizeof(DuLNode));     // 申请存放一个结点数据所需要的内在空间
  if(!L)   exit(1);                         // 存储分配失败
  L->next=L;                                // 表头结点作为表头结点的后继
  L->prior=L;                               // 表头结点作为表头结点的前驱
}
```

（重点*）双向循环链表的插入元素操作

在带头结点的双向链表DL中第i个结点之前插入元素e

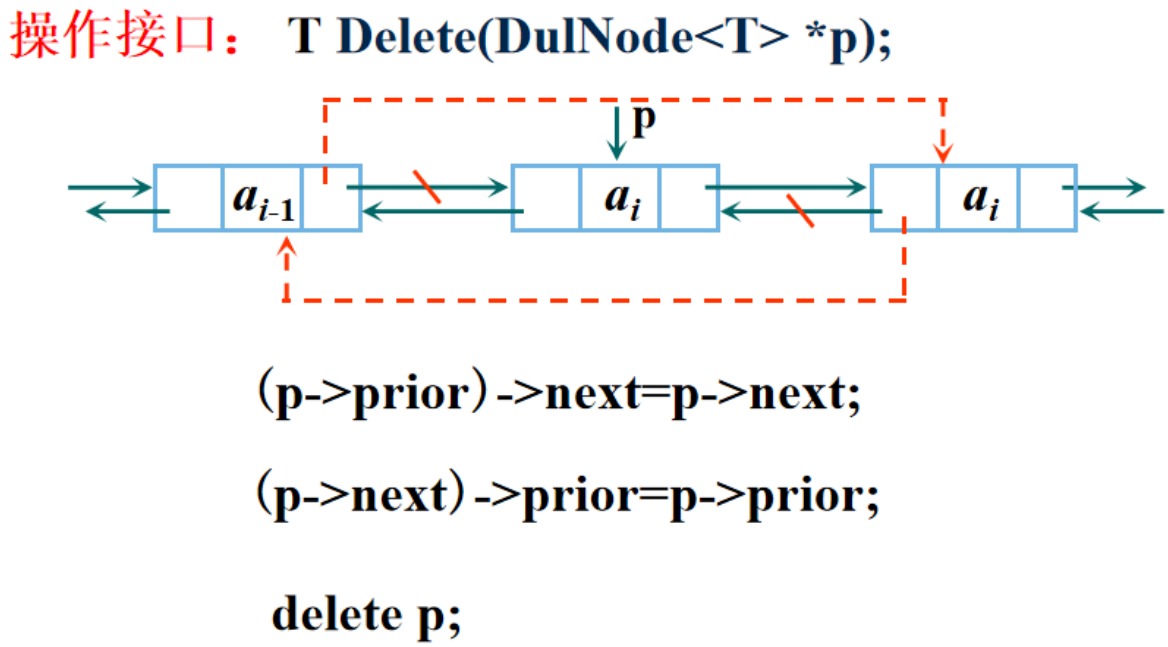
```
[in DuLinkList_C.h]
bool ListInsert_DuL_C(DuLinkList &L,int i,ElemType e)
{
   DuLinkList p,s;
   int j;
   p = L->next;j=1
   while(p!= L&&j<1)  { p = p->next; j++; }   // 寻找第i-1个结点,并让p指向此结点
   if(j!=i)   return false;                   // i的位置不合理
   s=(DuLNode *)malloc(sizeof(DuLNode));
   if(!s)   exit(1);                            // 存储分配失败
   s->data=e;
   s->prior=p->prior;                  // 修改指针
   p->prior->next=s;
   s->next=p;
   p->prior=s;
   return true;
}
```



（重点*）双向循环链表的删除元素操作

删除带有头结点的双向链表DL中的第i个结点，并让e返回其值

```
[in DuLinkList_C.h]
bool ListDelete_DuL_C(DuLinkList &L, int i, ElemType &e)
{   DuLinkList p;
    int j;
    p->L->next;j=1
    while(p->next!=L&&j<i){ p=p->next; j++; }   // 寻找第i个结点,并让p指向此结点
    if(j!=i)   return false;                    // i的位置不合理
    e = p->data;
    p->prior->next = p->next;
    p->next->prior = p->prior;
    free(p);
    return true;
 }
```



4、综合应用

能够从时间和空间复杂度的角度分析比较线性表两种存储结构的不同特点，了解各自应用场景，能够针对具体问题选择合适的结构。