第二章 线性表

学习目标

- 掌握线性表的逻辑结构，线性表的顺序存储结构和链式存储结构的描述方法及其特点；
- 熟练掌握线性表在顺序存储结构和链式存储结构下查找、插入、删除等基本操作的实现；
- 能够从时间和空间复杂度的角度综合比较两种存储结构的不同特点。

2.1 线性表的定义

线性表(Linear List)：是由$n(n\geq 0)$个数据元素(结点)

a_1，a_2，…a_n组成的有限序列。该序列中的所有结点具有相同的数据类型。其中下角标i表示该元素在线性表中的位置或序号，数据元素的个数n称为线性表的长度。

当$n=0$时，称为空表。

当$n>0$时，将非空的线性表记作：$(a_1, a_2, \ldots a_n)$

a_1称为线性表的第一个(首)结点，a_n称为线性表的最后一个(尾)结点。

线性结构是最常用、最简单的一种数据结构。在这种结构中：

① 存在一个唯一的被称为“第一个”的数据元素；

② 存在一个唯一的被称为“最后一个”的数据元素；

③ 除第一个元素外，每个元素均有唯一一个直接前驱；

④ 除最后一个元素外，每个元素均有唯一一个直接后继。

线性表是一种典型的线性结构

Student name list

姓 名	学 号	性 别	年龄	成绩
王小林	790631	男	18	88
陈 红	790632	女	20	95
刘建平	790633	男	21	76
张立立	790634	男	17	90
……	……	……	……	……

逻辑特征

$L=(a_1,\ldots,a_{i-1},a_i,a_{i+1},\ldots,a_n)$

- ✓ 有限性：个数是有穷的。
- ✓ 相同性：元素类型相同。
- ✓ 相继性：

a_1为表中第一个元素，无前驱元素，a_n为表中最后一个元素，无后继元素；

对于$1<i<n$，a_{i-1}为a_i的直接前驱，a_{i+1}为a_i的直接后继。

线性表的基本操作

设L是线性表，e为数据元素，i为位置变量。

1）求线性表长度
2）获得第i个位置上的元素
3）按值e进行查找
4）在第i个位置插入e
5）删除第i个位置上的元素

线性表的抽象数据类型定义

ADT List {
Data
$D=\{a_i, a_i \in \text{ElementType}, i=1,2,\ldots,n, n\geq 0\}$
$R=\{<a_{i-1},a_i>|a_{i-1},a_i \in D, i=2,\ldots,n\}$
基本操作
InitList(&L)
后件：构造一个空的线性表L。

求表的长度
ListLength(L)
输出：表的长度；
操作结果：计算表的长度。

取表L中第i个数据元素赋值给e
GetElem(L,i,e)
输入：位置参数i；
输出：e是ElementType类型；
初始条件：$1\leq i\leq$ ListLength(L)
操作结果：e被赋予表中第i个位置的元素值。

在L中第i个位置插入新的数据元素e
ListInsert(L,i,e)
输入：位置参数i, e是ElementType类型；
初始条件：$1\leq i\leq$ ListLength(L)+1
操作结果：在L中第i个位置插入新的数据元素e，表长加1。

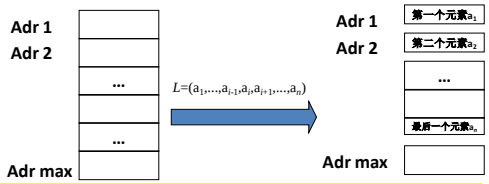
删除表中第i个数据元素
ListDelete (L,i)
输入：位置参数i；
初始条件：$1\leq i\leq$ ListLength(L)
操作结果：删除L中第i个位置的数据元素，表长减1。

按值查找
SearchElem(e,i)
输入：e是ElementType的类型；
输出：位置参数i；
初始条件：$1\leq$ ListLength(L)
操作结果：若e存在表中，返回第一个e的位置为i，否则返回0。

}ADT List

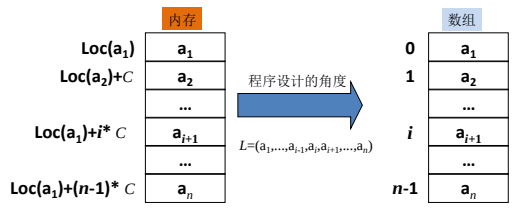
2.2 线性表的连续设计存储方式

存储方式：将线性表中的元素**依次**存放在连续的存储空间中。
用这种存储方式存储的线性表又称**顺序表。**



$LOC(a_i) = LOC(a_1) + (i-1) * C \quad 1<=i<=n$

高级语言环境下，顺序表的实现：数组。



$LOC(a_i) = LOC(a_1) + (i-1) * C \quad 1<=i<=n$

存储特点

- 元素之间逻辑上的关系，用物理上的相邻关系来表示

（用物理上的连续性刻画逻辑关系）。

逻辑上相邻的元素，在物理位置上也相邻。

- 是一种随机访问存取的结构，也就是说可以按元素的位置之间的关系进行访问，其位置可以由公式直观的计算出来。

$LOC(a_i) = LOC(a_1) + (i-1) * C \quad 1 <= i <= n$

Implementation：利用数组结构实现

```
#define  OK   1
#define  ERROR   -1
#define  MAX_SIZE  100
typedef  int  Status ;
typedef  int  ElemType ;
typedef  struct  sqlist
{   ElemType   Elem_array[MAX_SIZE] ;
    int    length ;
} SqList ;
```


顺序表的基本操作

顺序存储结构中，很容易实现线性表的一些操作：初始化、赋值、查找、修改、插入、删除、求长度等。

1）顺序表初始化

```
Status Init_SqList( SqList *L )
{ L->elem_array=( ElemType * )
    malloc(MAX_SIZE*sizeof( ElemType ) ) ;
  if ( !L -> elem_array ) return  ERROR ;
  else {   L->length= 0 ;    return OK ; }
}
```

建立了一个空表。

2）在第 *i* 个位置上插入元素 *e*

在线性表 $L=(a_1,\ldots,a_{i-1},a_i,a_{i+1},\ldots,a_n)$ 中的第 $i(1 \le i \le n+1)$ 个位置上插入一个新结点 e，使其成为线性表：

$L=(a_1,\ldots,a_{i-1},e,a_i,a_{i+1},\ldots,a_n)$

实现步骤

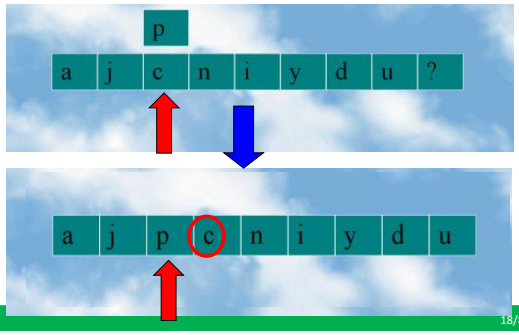
（1）将线性表 L 中的第 i 个至第 n 个结点后移一个位置。

（2）将结点 e 插入到结点 a_{i-1} 之后。

（3）线性表长度加1。

Case:

i=3，*e*=‘P’



顺序表实现说明：在第 *i* 个位置上插入元素*e*

InsertElem(L,*i*,*e*)

输入：*L* 是顺序表，*i* 是插入的位置，*e* 插入的数据元素；

初始条件：0≤*i*≤L->length

操作结果：将数据元素*e* 插入到顺序表的第 *i* 个位置上。


```
int InsertElem(sqlist *L, ElemType x,int i)
{ int j;
 if  (L->length>=MAX_SIZE)
   {   printf("线性表溢出!\n"); return  ERROR ; }
 else
  if((i<0)||(i>(L->length)
   { printf("error\n"); return -1;}  \\非法位置
  else
   {   for (j=L->length-1;j>=i-1;j--)
         L->Elem_array[j+1]=L->Elem_array[j];

       L->Elem_array[i-1]=e;   /*  在i-1位置插入结点  */
       L->length++ ;
       return  OK ;
 }
```


时间复杂度分析

在线性表*L*中的第*i*个元素之前插入新结点，其时间主要耗费在表中结点的移动操作上，因此，可用结点的移动来估计算法的时间复杂度。

设在线性表*L*中的第*i*个元素之前插入结点的概率为P_i，不失一般性，设各个位置插入是等概率，则$P_i=1/(n+1)$，而插入时移动结点的次数为$n-i+1$。

总的平均移动次数：$E_{insert}=\sum p_i*(n-i+1)$ $(1\leq i\leq n)$

$\therefore E_{insert}=n/2$ 。

即在顺序表上做插入运算，平均要移动表上一半结点。当表长*n*较大时，算法的效率相当低。

算法的时间复杂度为$O(n)$。

3）删除第 *i* 个位置上的元素

删除线性表$L=(a_1,\ldots,a_{i-1},a_i,a_{i+1},\ldots,a_n)$中的第$i(1\leq i\leq n)$个位置的元素$a_i$，使其成为线性表：

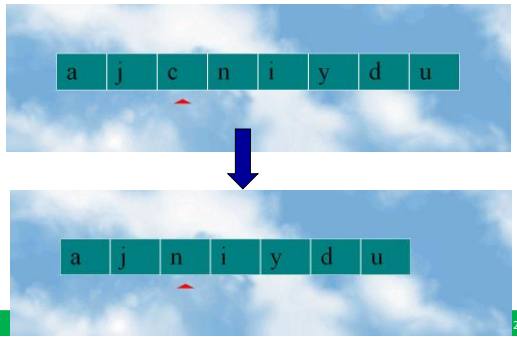
$L=(a_1,\ldots,a_{i-1},a_{i+1},\ldots,a_n)$

实现步骤

（1）将线性表*L*中的第*i*+1个至第*n*个结点依次向前移动一个位置。

（2）线性表长度减1。

Case: *i*=3



删除第 *i* 个位置上的元素

DeleteElem(*L*, *i*)

输入：位置 *i*；

初始条件：表不为空，且

0≤*i*≤ L->length-1

处理结果：删除位置*i*上的元素。

时间复杂度分析

删除线性表L中的第i个元素，其时间主要耗费在表中结点的移动操作上，因此，可用结点的移动来估计算法的时间复杂度。

设在线性表L中删除第i个元素的概率为P_i，不失一般性，设删除各个位置是等概率，则$P_i=1/n$，而删除时移动结点的次数为$n-i$。

则总的平均移动次数：$E_{delete}=\sum p_i*(n-i)$ $(1\leqslant i\leqslant n)$

$\therefore E_{delete}=(n-1)/2$ 。

即在顺序表上做删除运算，平均要移动表上一半结点。当表长n较大时，算法的效率相当低。

算法的时间复杂度为$O(n)$。

在插入和删除操作中

- 特别要关注表长及位置。
- 插入：
  - 最坏：i=1，移动次数为n
  - 最好: i=表长+1，移动次数为0
  - 平均：等概率情况下，平均移动次数$n/2$
- 删除：
  - 最坏：i=1，移动次数为n-1
  - 最好: i=表长，移动次数为0
  - 平均：等概率情况下，平均移动次数(n-1)/2

4）在顺序表中查找元素e

实现步骤

（1）依次比较元素e与顺序表L中的每个元素。

（2）查找成功，返回e的位置。

（3）否则返回查找不成功。

按值查找

LocateElem1(L, e,i)

输入：元素 e;

输出：元素 e的位置;

初始条件：L 不为空;

处理结果:返回元素 e的位置i或不成功-1


```
int LocateElem1(SqList *L, ElemType e)
{ i=0;
  if (L->length==0) return ERROR;
  while ( i<= L->length-1 && e != L-> Elem_array[i]) ++i;
  if (i<= L->length) return i;
      else return ERROR;
}
```

时间复杂度为$O(n)$。

5）在顺序表中查找位置i上的元素

LocateElem2(L, i, e)

输入：位置 i;

输出：位置i上的元素e;

初始条件：L非空且

$0\leqslant i\leqslant$ L->length-1;

处理结果:返回位置i上的元素e.

```
DataType LocateElem2(SqList *L, int i, ElemType e)
{
 if (L->length==0) return ERROR;
Else
 If ((i<0)||(i> L->length-1))
  { printf("error\n");
   return ERROR;}  \\非法位置
 return L-> Elem_array[i];
}
```

时间复杂度为$O(1)$。

顺序表性能分析

1）优点
- 顺序表的结构简单
- 顺序表的存储效率高，是紧凑结构,无须为表示节点间的逻辑关系而增加额外的存储空间
- 顺序表是一个随机存储结构（直接存取结构）

2）缺点
- 在顺序表中进行插入和删除操作时，需要移动数据元素，算法效率较低。
- 对长度变化较大的线性表，或者要预先分配较大空间或者要经常扩充线性表，给操作带来不方便。

2.3 线性表的链式存储结构

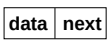
一组任意的存储单元存储线性表中的数据元素及其逻辑关系。存储单元可以是连续的，也可以是不连续的，甚至是零散分布在内存中的任意位置上的。为了正确表示结点间的逻辑关系，在存储每个结点值的同时，还必须存储指示其直接后继（或直接前驱）结点的地址(或位置)，称为指针(pointer)或链(link)，这两部分组成了链表中的结点结构。

2.3.1 单链表存储

链表是通过每个结点的指针域将线性表的n个结点按其逻辑次序链接在一起的。

- 只含有一个指针的链表结点称为单链表，
- 指针指向后继元素的地址（或前驱）。

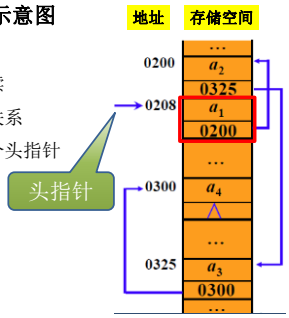
具有一个指针的结点



data：数据域，存放结点的值。
next：指针域，存放结点的直接后继的地址。

(a_1, a_2, a_3, a_4)的存储示意图

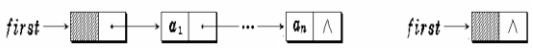
- 存储空间的地址不连续
- 含有指针来存储逻辑关系
- 单链表中必须含有一个头指针



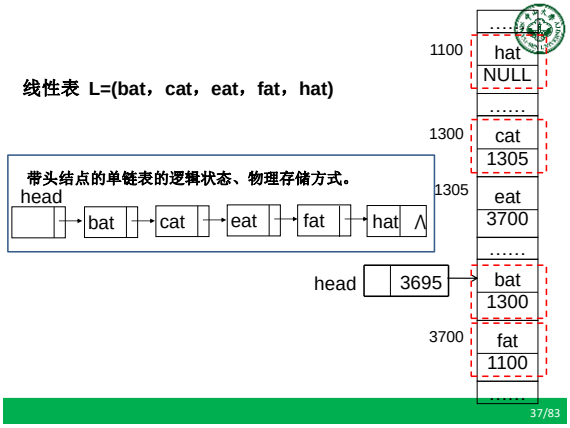
为操作方便，总是在链表的第一个结点之前附设一个头结点(头指针)head指向第一个结点。头结点的数据域可以不存储任何信息(或链表长度等信息)。

由于开始结点的位置被存放在头结点的指针域中，所以在链表的第一个位置上的操作就和在表的其他位置上操作一致，无须进行特殊处理；

无论链表是否为空，其头指针都是指向头结点的非空指针（空表中头结点的指针域空），因此可以统一处理空表和非空表。



线性表 L=(bat，cat，eat，fat，hat)



结点的类型说明

```
typedef struct Lnode
  { ElemType data;    /*数据域，保存结点的值 */
    struct Lnode *next;    /*指针域*/
  }LNode;    /*结点的类型 */
```


结点的实现：

结点是通过动态分配和释放来的实现，即需要时分配，不需要时释放。

动态分配： p=(LNode*)malloc(sizeof(LNode));

函数malloc分配了一个类型为LNode的结点变量的空间，并将其首地址放入指针变量p中。

动态释放 free(p) ;

系统回收由指针变量p所指向的内存区。P必须是最近一次调用malloc函数时的返回值。

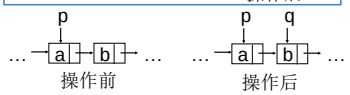
结点的操作：

```
LNode  *p;
p=(LNode*)malloc(sizeof(LNode));
p->data=20;  p->next=NULL ;
```

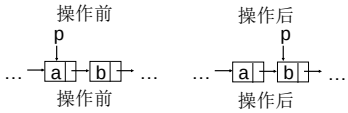

常见的指针操作

① q=p；

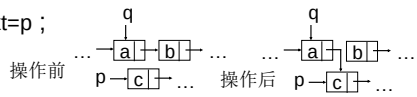
② q=p->next；



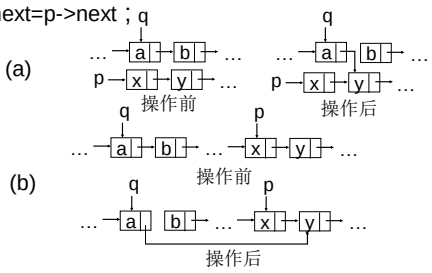
③ p=p->next；



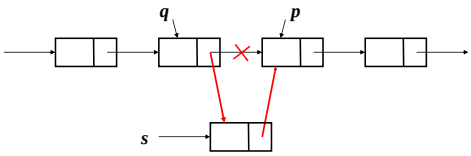
④ q->next=p；



⑤ q->next=p->next；



（在*p*结点所在的位置插入：在*p*结点前插入）

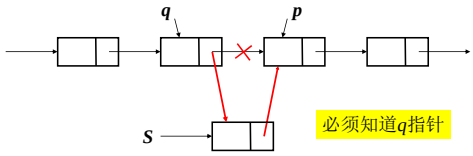


建立新结点*s*，向新结点中添入内容

查找插入结点的位置*p*,以及*p*结点的前驱结点*q*

将新结点链入链中

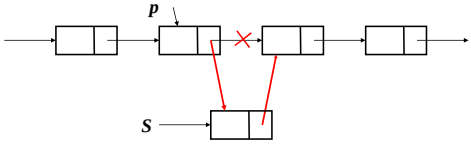
（在*p*结点所在的位置插入：在*p*结点前插入）




```
S->data=a;
S->next=p;
q->next=S;
```

```
S->data=a;
q->next=S;
S->next=p;
```


（在*p*结点之后插入）




```
S->data=a;
p->next=S;
S->next=p->next;
```

```
S->data=a;
S->next=p->next;
p->next=S;
```

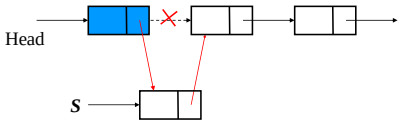

2.3.2 单链表的基本操作

- 建立单链表
- 查找
- 插入
- 删除
- 合并

1）头插入法建立单链表

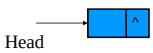
算法:

1. **生成一个头结点 Head**
2. **生成一个待插入的结点 S**
3. **将S插入在Head后面**
4. **重复2,3步，直到结束**



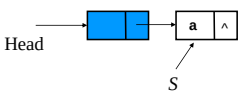
Case: L=(a,b,c,d,e)

Step 1



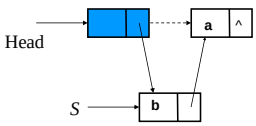

```
Head=(LNode*)malloc(size
of(LNode));
Head->next=NULL;
```

Step 2: insert a

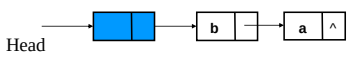


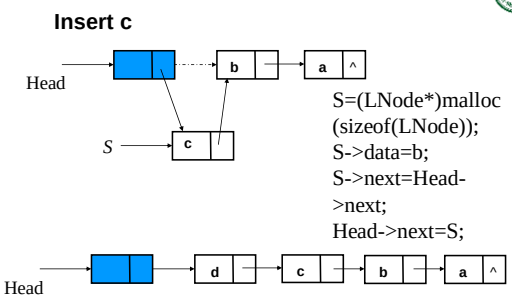

```
S=(LNode*)malloc
(sizeof(LNode));
S->data=a;
S->next=Head->next;
Head->next=S;
```


Insert b




```
S=(LNode*)malloc
(sizeof(LNode));
S->data=b;
S->next=Head->next
Head->next=S;
```





建立单链表的算法

1. To create Head
2. To create *S*
3. Insert *S*

```
Void CreateList(Head)
{
  Head=(LNode*)malloc
(sizeof(LNode));
  Head->next=NULL;
  scanf("%c",ch);
  while (ch!='#') do
  {
    S=(LNode*)malloc
    (sizeof(LNode));
    S->data=ch;
    S->next=Head->next;
    Head->next=S;
    scanf("%c",ch);
  }
}
```



```
Void CreateList2(Head) 尾插入法建表
{Head=(LNode*)malloc
(sizeof(LNode));
  Head->next=NULL;
  Last=Head;
  scanf("%c",ch);
  while (ch!='#') do
   {  S=(LNode*)malloc
      (sizeof(LNode));
      S->data=ch;
      S->next=NULL;
      Last->next=S;
      (              )=S;
      scanf("%c",ch);
   }
}
```


无论是哪种插入方法，如果要插入建立的单线性链表的结点是*n*个，算法的时间复杂度均为*O*(*n*)。

2）按序号查找第*i*个元素

链表不是随机存取结构。因此，单链表不能象在顺序表中那样直接按序号*i*访问结点，只能从链表的头结点出发，沿指针next逐个结点进行搜索，直到搜索到第*i*个结点为止。

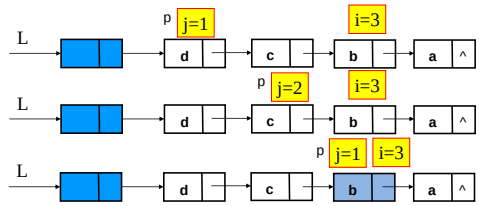
输入：位置参数*i*；
输出：*e*是ElementType类型；
前件：1≤*i*≤LenList(*L*)；　　//表长度
后件：*e*被赋予表中第*i*个位置的元素值。


```
Status GetElem_L(LinkList L, int i,DataType &e)
{   LinkList p;
    if ((i<1)||(i> LenList(L)) return ERROR;
    p=L->next;  int j=1;  从第一个结点开始查找
    while (j<i) { p=p->next; ++j; }
    e=p->data;
    return OK;
}
```

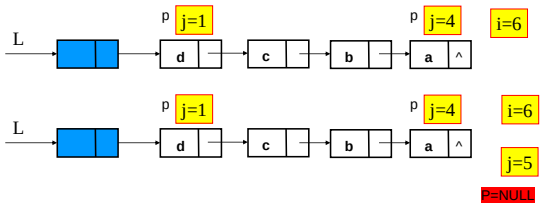
改变初始条件：

输入：位置参数i；
输出：e是ElementType类型；
前件：$1 \le i$；长度问题通过判断是否为空来解决
后件：e被赋予表中第i个位置的元素值。

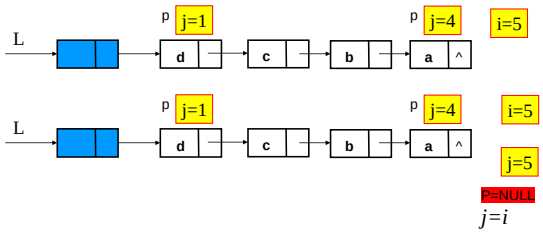
- ✓ 声明一个指针p指向链表第一个结点；计数j从1开始；
- ✓ 当$j<i$且p不为空时，让p指针向后移动进行表的遍历，j累加1；
- ✓ 当$j<i$且p为空，说明第i个元素不存在；
- ✓ 当$j=i$,p不为空，则查找成功。当$j=i$,p为空，则查找失败。



- ✓ 声明一个指针p指向链表第一个结点；计数j从1开始；
- ✓ 当$j<i$且p不为空时，让p指针向后移动进行表的遍历，j累加1；
- ✓ 当$j<i$且p为空，说明第i个元素不存在；
- ✓ 当$j=i$,p不为空，则查找成功。当$j=i$,p为空，则查找失败。



- ✓ 声明一个指针p指向链表第一个结点；计数j从1开始；
- ✓ 当$j<i$且p不为空时，让p指针向后移动进行表的遍历，j累加1；
- ✓ 当$j<i$且p为空，说明第i个元素不存在；
- ✓ 当$j=i$,p不为空，则查找成功。当$j=i$,p为空，则查找失败。




```
Status GetElem_L(LinkList L, int i,DataType &e)
{   LinkList p;
    if (i<1) return ERROR;
    p=L->next;  int j=1;     从第一个结点开始查找
    while (j<i&&p!=NULL) { p=p->next; ++j; }
    if (p!=NULL) {e=p->data; return OK;}
     else return ERROR;
}
```

移动指针p的频度：
***i*<1时：0次; *i*∈[1,*n*]：*i*-1次；*i*>*n*：*n*次。**
∴ 时间复杂度: *O*(*n*)。

3）查找是否有元素*e*

输入：
输出：
前件：
后件：

```
int LinkLocate_L (LinkList L, int e)
{  int i;   LinkList  p;
   p=L->next;  i=1;
   while (p!=NULL && p->data != e)
         {p= p->next; i++;}
   if (p==NULL)  { printf ("Not found! \n");
              return(0);
            }
   else { printf ("i=%d\n",i);return (i); }
}
```

算法的时间复杂度为$O(n)$。

4）在单链表的第*i*个位置插入元素

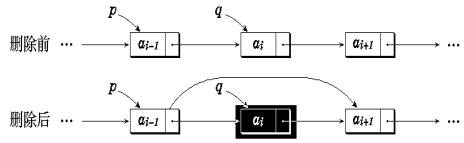
算法基本思想：修改第***i-1***个位置元素的指针

```
Status GetElem_L(LinkList L, int i,DataType &e)
{   LinkList p;
    if (i<1) return ERROR;
    p=L->next;  int j=1;     从第一个结点开始查找
    while (j<i&&p!=NULL) { p=p->next; ++j; }
    if (p!=NULL) {e=p->data; return OK;}
     else return ERROR;
}
```

P指向哪个结点？


```
Status InserElem (LinkList L, int i,DataType &e)
{   LinkList p;
    if (i<1) return ERROR;
    p=L->next;  int j=1; q=L;  从第一个结点开始查找
    while (j<i&&p!=NULL) { q=p; p=p->next; ++j; }
    if (p==NULL) return ERROR;
     else {
       s = (LinkList) malloc( sizeof (LNode));
       s->data = e; s->next = q->next;
       q->next = s;
}
}
```


5）删除单链表第*i*个元素



实际是要修改第*i*-1个元素的指针。


```
Status DetElem_L(LinkList L, int i)
{   LinkList p;
    if (i<1) return ERROR;
    p=L->next;  int j=1; q=L;  从第一个结点开始查找
    while (j<i&&p!=NULL) { q=p; p=p->next; ++j; }
    if (p==NULL) return ERROR;
     else {
       q->next = p->next ;
}
}
```


自查：熟练单链表上的操作

建立单链表
单链表上按位置查找
单链表上按值查找
删除单链表中的元素
两个递增有序的单链表合并

2.3.3 双向链表

双向链表(Double Linked List) :指的是构成链表的每个结点中设立两个指针域：一个指向其直接前趋的指针域prior，一个指向其直接后继的指针域next。这样形成的链表中有两个方向不同的链，故称为双向链表。

和单链表类似，双向链表一般增加头指针也能使双链表上的某些运算变得方便。

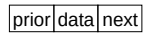
将头结点和尾结点链接起来也能构成循环链表，并称之为双向循环链表。

双向链表是为了克服单链表的单向性的缺陷而引入的。

双向链表的结点及其类型定义

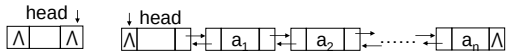
双向链表的结点的类型定义如下。

```
typedef struct Dulnode
   {   ElemType  data ;
       struct Dulnode  *prior , *next ;
   }DulNode ;
```



双向链表结点形式

带头结点的双向链表形式



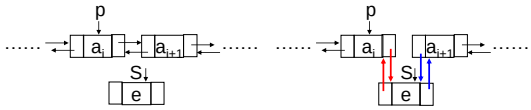
空双向链表　　非空双向链表

双向链表结构具有对称性，设p指向双向链表中的某一结点，则其对称性可用下式描述：

(p->prior)->next=p=(p->next)->prior ;

结点p的存储位置存放在其直接前趋结点p->prior的直接后继指针域中，同时也存放在其直接后继结点p->next的直接前趋指针域中。

(1) 双向链表的插入 将值为e的结点插入双向链表中。插入前后链表的变化如图所示。



插入时仅仅指出直接前驱结点p，操作时必须注意先后次序是：“先右后左”。部分语句组如下：

S->data=e;

S->next=p->next; p->next->prior=S; (蓝色)

p->next=S; S->prior=p; (红色)

操作次序非常重要

(2) 双向链表的结点删除

设要删除的结点为p，删除时可以不引入新的辅助指针变量，可以直接先断链，再释放结点。部分语句组如下：

```
p->prior->next=p->next;
p->next->prior=p->prior;
free(p);
```

注意：

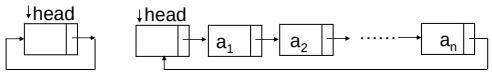
与单链表的插入和删除操作不同的是，在双向链表中插入和删除必须同时修改两个方向上的指针域的指向。

2.3.4 循环链表

循环链表(Circular Linked List)：是一种头尾相接的链表。其特点是最后一个结点的指针域指向链表的头结点，整个链表的指针域链接成一个环。

从循环链表的任意一个结点出发都可以找到链表中的其它结点，使得表处理更加方便灵活。

带头结点的单循环链表的示意图如下。



空表　　非空表

循环链表的操作

对于单循环链表，除链表的合并外，其它的操作和单线性链表基本上一致，仅仅需要在单线性链表操作算法基础上作以下简单修改：

⑴ 判断是否是空链表：head->next==head ；

⑵ 判断是否是表尾结点：p->next==head ；

双向循环链表的操作

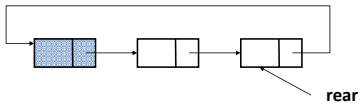
⑴ 判断是否是空链表：head->next==head

head->prior==head ；

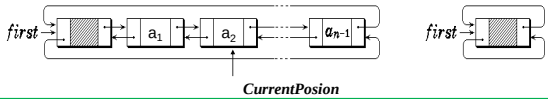
⑵ 判断是否是表尾结点：p->next==head ；

2.3.5 其他形式的循环链表

只有尾结点的循环单链表



具有当前位置指针的双向循环链表



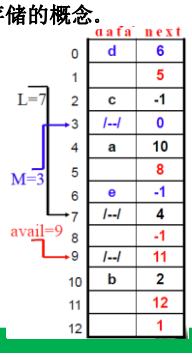
连续设计和链接设计的对比

比较参数	连续设计	链接设计
表的容量	固定，不易扩充	灵活，易扩充
存取操作	随机访问存取	顺序访问存取
时间	插入删除费时间	访问元素费时间
空间	估算长度，浪费空间	实际长度

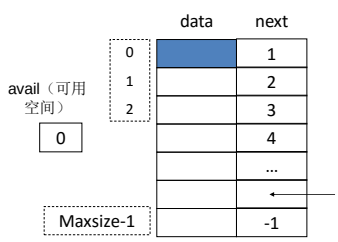
2.4 链接存储设计的数组实现

利用静态数组实现链接存储的概念。

举例：
线性表 L=(a,b,c)
线性表 M=(d,e)
空闲表 avail=9



结点说明




```
#define maxsize 1024    \\存储池最大容量
typedef int datatype;
typedef int cursor;
typedef struct          \\结点类型
{ datatype data;
  cursor next;
} node;

node nodepool[maxsize];    \\存储池
cursor avail;
```

Initial array nodepool

```
INITIALIZE()
{ int i;
  for (i=0;i<maxsize-1;i++)
    nodepool[i].next＝i+1;
  nodepool[maxsize-1].next＝-1;
  avail ＝0;
}
```


结点的分配算法(带头结点）

```
cursor GETNODE()
{ cursor p;
  if (nodepool[avail].next ==-1) p＝-1;
  else
    { p＝ nodepool[avail].next;
      q= nodepool[p].next
      nodepool[avail].next ＝q;
    }
  return p;
}
```


结点的回收算法

```
FREENODE(cursor p)
{
  nodepool[p].next＝ nodepool[avail].next;
 nodepool[avail].next ＝p;
}
```


静态链表查找算法

```
cursor FINDSTLIST(cursor L,int i)
{ cursor p; int j;
  p＝L; j＝0;
  while ((nodepool[p].next!=-1)&&(j<i))
   { p＝nodepool[p].next;
     j++;
   }
  if (i==j) return(p);
  else return -1;
}
```