第二章　线性表

2.1 线性表的概念

2.2 顺序表示

2.3 链接表示

2.4 应用举例-Josephus问题

另外介绍“动态顺序表”

程序里常需要保存一批某种类型的元素，这些元素的数目可能变化（可以加入或删除元素）

有时需要把这组元素看成一个序列，元素的顺序可能表示实际应用中的某种有意义的关系

这样一组元素可以抽象为元素的一个线性表。线性表是元素的集合，同时记录了元素的顺序关系

线性表是一种最基本的数据结构，在各种程序里应用广泛，还常作为实现更复杂的数据结构的基础

本章讨论线性表的概念，它的两种基本实现方式，它们的一些变形，以及若干应用实例

2.1 线性表的概念

- 元素: $\{e_0, \ldots, e_{N-1}\}$ 是一个元素集合
- 线性表：简称表，元素的有穷序列，可包含0个或多个元素，表示为 $(e_0, e_1, \ldots, e_{n-1})$，$n \geqslant 0$
- 元素位置称为下标，下标从0开始（一种选择）
- 表中元素个数称为表的长度，长度为0的表是空表
- 关系：$<e_0, e_1>, <e_1, e_2>, \ldots, <e_{n-2}, e_{n-1}>$。是一种顺序关系（线性关系）
- 例：整数的表，字符串的表，某种结构的表

线性表是一种线性结构

非空线性表中：

1. 存在唯一的称为“第一个”的元素；
2. 存在唯一的称为“最后一个”的元素；
3. 除第一个元素之外，表中的每个元素都有且只有一个前驱元素；
4. 除最后一个元素之外，表中的每个元素都有且只有一个后继元素。

可以把线性表作为一种数学对象，为它建立抽象模型

抽象模型

元素集合：$\mathbb{E}$　　$e \in \mathbb{E}$

线性表集合：$list\ \mathbb{E}$　　$\alpha, \beta \in list\ \mathbb{E}$

$\alpha = \langle e_0, e_1, \cdots, e_n \rangle$

基本操作

组合：$e \oplus \langle e_0, \cdots, e_n \rangle \stackrel{\text{def}}{=} \langle e, e_0, \cdots, e_n \rangle$

取头部：$hd\ \langle e_0, e_1, \cdots, e_n \rangle \stackrel{\text{def}}{=} e_0$

取尾部：$tl\ \langle e_0, e_1, \cdots, e_n \rangle \stackrel{\text{def}}{=} \langle e_1, \cdots, e_n \rangle$

判空表：$empty\ \alpha \stackrel{\text{def}}{=} \begin{cases} \text{true} & \text{if } \alpha = \langle\rangle \\ \text{false} & \text{otherwise} \end{cases}$

操作的基调（Signature，原型，类型特征）

组合：$\oplus : \mathbb{E} \times list\ \mathbb{E} \to list\ \mathbb{E}$

取头部：$hd : list\ \mathbb{E} \to \mathbb{E}$

取尾部：$tl : list\ \mathbb{E} \to list\ \mathbb{E}$

判空表：$empty : list\ \mathbb{E} \to \mathbb{B}$　　$\mathbb{B} = \{\text{true}, \text{false}\}$

定义操作，例如表的长度操作

基调：$\# : list\ \mathbb{E} \to \mathbb{N}$

定义：$\# \alpha \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } \alpha = \langle\rangle \\ 1 + \# tl\ \alpha & \text{if } \alpha \neq \langle\rangle \end{cases}$

其他操作可类似定义

代数定律

假设: $e \in \mathbb{E},\ \ \alpha, \beta \in list\ \mathbb{E},\ \ \beta \neq \langle\rangle$

定律:

$$hd(e \oplus \alpha) = e$$
$$tl(e \oplus \alpha) = \alpha$$
$$(hd\ \beta) \oplus (tl\ \beta) = \beta \quad \text{if } \beta \neq \langle\rangle$$
$$empty\ \langle\rangle = \textsf{true}$$
$$empty(e \oplus \alpha) = \textsf{false}$$
$$\#\langle\rangle = 0$$
$$\#(e \oplus \alpha) = 1 + \#\alpha$$

这样下去，可以定义更多的表操作以及操作之间的关系，做出一套表的理论

下面从更实际的角度考虑表的结构

类型与变量说明:

元素类型	DataType
元素变量说明	DataType x;
下标类型	Index
下标变量说明	Index p;
线性表类型	List
线性表变量说明	List list;

线性表的基本运算（可根据需要扩充）:

1. List createNullList (void)
   创建一个空线性表
2. int insert (List list, Index p, DataType x)
   在线性表list中下标为p的位置插入一个值为 x 的元素，返回成功与否的标志。操作把新元素插入序列中，保持原有元素的前后顺序不变。插入可能由于内部原因而失败，此时原表结构不变
3. int delete (List list, Index p)
   在线性表list中删除下标为p的元素，返回删除成功与否的标志。操作保证表中剩余元素的顺序不变。如果表里没有下标 p 则删除操作失败

4. Index locate (List list, DataType x)
   在线性表list中查找值为 x 的元素的位置。找到时返回元素在表中的位置，找不到返回一个特殊值
5. int isNull (List list)
   判别线性表list是否为空表。如果list是空表就返回真（在C里用非0），否则返回假

还可以有其他操作，例如

1. int deleteElem (List list, DataType x)
   在线性表 list 删除值为 x 的元素
2. void sortAscend (List list)
   将线性表list的元素按照值的上升顺序重新排列
3. ……

线性表与数组的比较:

线性表	数组
线性结构	线性结构
是一种更抽象的结构。少数语言内部提供了表类型，在多数语言里需要自己实现	是一种更具体的结构，通常作为语言最基本的数据机制，可直接使用
操作集合包括插入、删除等，还可根据需要考虑许多操作	操作只有通过下标访问，元素赋值或取元素值
表中元素个数可通过操作改变	元素个数不变，创建时确定

完全可能用数组作为实现线性表的一种具体表示结构

2.2 顺序表示

线性表的一种可能实现方式是顺序存放其中元素

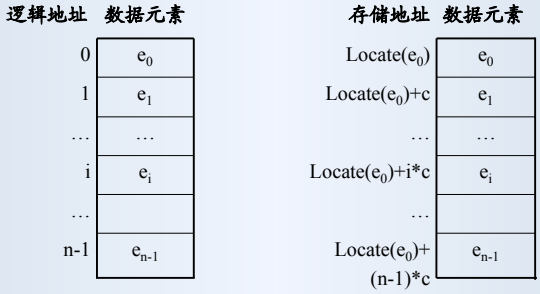
把元素顺序存放在一片连续存储区中，借助元素在内存的“物理位置”表示元素间的顺序逻辑关系（隐式表示元素间的关系），这样形成的结构称为顺序表

顺序表中任一元素的位置都可直接算出，因此可以在 O(1) 时间直接存取。元素 e_i 的地址计算公式:

$$Locate(e_i) = Locate(e_0) + c * i$$

其中 c = sizeof(DataType)，一个元素占用的存储量

线性表的顺序存储结构示意图

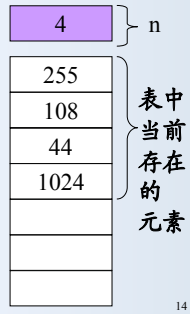


顺序表，可看作以数组作为基础实现的线性表

数组的元素个数固定，线性表的元素个数可以随操作改变，需要设法弥合两者之间的差异

方式：创建一个大数组，表元素存在数组前面一段

- 表元素应密集地连续存放，以便能用数组下标作为表元素索引（效率）
- 需要记录表中元素个数。必须维持元素个数记录和表中实际元素个数一致



数组大小确定了顺序表的最大容量（需根据实际确定），也使插入操作有可能失败

连续存放给删除操作的实现带来问题

顺序表的C实现：类型定义

定义1:

```
DataType  elements[NMAX];
int  n;       /* 元素个数n < NMAX  */
/*没反映 elements 和 n 的内在联系;应专门引进一个类型*/
```

定义2：（用结构将相关数据成分包装起来）

```
struct SeqList {
    int n;        /* 元素个数n < NMAX  */
    DataType  elements[NMAX];
};
```

最好是定义为类型：

```
typedef struct SeqList { ... ... } SeqList;
```


后一定义更清晰，更符合数据抽象原则（一个对象应是一个整体，一种对象应定义为类型），应该采用

为了使用方便，常定义指向顺序表类型的指针类型：

```
typedef  SeqList  *PSeqList;
```

如果函数 f 的原型是:

```
..... f (PSeqList  pl)
```

实际调用函数 f 应该采用的形式:

```
SeqList L1; /* 定义顺序表 L1 */
..... f(&L1);
```

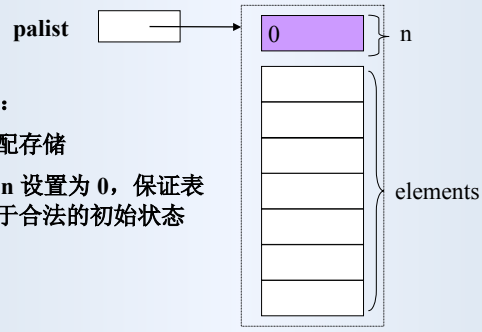
在 f 里可以完成对 L1 的任何操作（思考题：如果 f 的参数用 SeqList类型，情况有什么不同？）

顺序表基本运算的实现

- 算法2.1 创建空表
  PSeqList createNullList_seq(void)
- 算法2.2 插入
  int insert_seq(PSeqList palist, int p, DataType x)
- 算法2.3 删除
  int delete_seq(PSeqList palist, int p)
- 算法2.4 求表中某元素的下标
  int locate_seq(PSeqList palist, DataType x)
- 算法2.5 求 palist 所指表中下标为 p 的元素值
  DataType retrieve_seq(PSeqList palist, int p)
- 算法2.6 判断表是否为空
  int isNullList_seq(PSeqList palist)

算法2.1 创建空顺序表

PSeqList createNullList_seq(void)



需要：

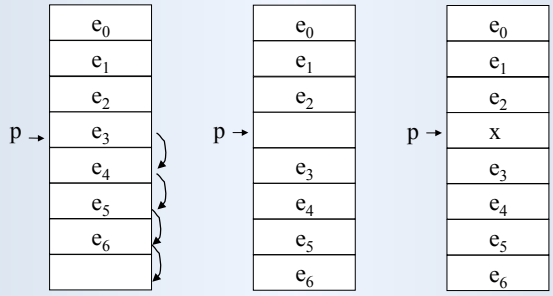
- 分配存储
- 把 n 设置为 0，保证表处于合法的初始状态

创建空顺序表

```
PSeqList createNullList_seq( void ) {
    PSeqList palist =
        (PSeqList)malloc(sizeof(SeqList));
    if (palist != NULL)
        palist->n = 0;                  /*分配成功，空表长度为0 */
    else
        printf("Out of space!\n"); /*分配失败*/
    return palist;
}
```


算法2.2 顺序表的插入

int insert_seq(PSeqList palist, int p, DataType x)

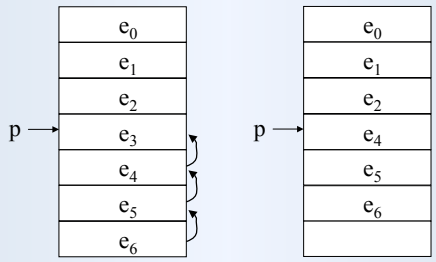


移动元素，腾出空位后，把新元素存入空位


```
/*在palist所指顺序表中下标为p的元素之前插入元素x*/
int insert_seq(PSeqList palist, int p, DataType x) {
    int q;
    if ( palist->n == NMAX ) {  /* 顺序表已满，插入失败 */
        printf("Seq-list overflow!\n");   return FALSE;
    }
    if (  p < 0  ||  p > palist->n  ) { /* 不存在下标为p的元素 */
        printf("Index of seq-list is out of range! \n");
        return FALSE;                /* 插入失败 */
    }
    for (q = palist->n - 1; q >= p; --q)  /* 元素逐一后移*/
        palist->elements[q+1] = palist->elements[q];
    palist->elements[p] = x;             /* 插入元素x */
    palist->n++;                         /* 元素个数加1 */
    return TRUE;
}
```


算法2.3 顺序表的删除

int delete_seq(PSeqList palist, int p)



元素逐个前移，覆盖掉被删除的元素

需要维护元素个数记录


```
/*在palist所指顺序表中删除下标为 p 的元素*/
int delete_seq( PSeqList palist, int p ) {
   int q;
   if (p < 0 || p > palist->n - 1 ) {      /* 不存在下标为p的元素 */
       printf("Index of seq-list is out of range!\n ");
       return FALSE;
   }

   for (q = p + 1; q < palist->n; ++ q)         /* 元素逐个前移 */
       palist->elements[q - 1] = palist->elements[q];
   palist->n--;                                 /* 元素个数减1 */
   return TRUE;
}
```


算法2.4 求表中某元素的下标

int locate_seq(PSeqList palist, DataType x)

```
int locate_seq(PSeqList palist, DataType x) {
   int q;
   for ( q = 0; q<palist->n; ++q)
       if (palist->elements[q] == x) return q;
   return  -1;
}
```

这里用 –1 表示元素没有找到

-1 不是合法位置值，因此适合用于表示特殊情况

算法2.5 求 palist 所指表中下标为 p 的元素值
DataType retrieve_seq(PSeqList palist, int p)

```
DataType retrieve_seq( PSeqList palist, int p ) {
  if ( p >= 0 && p < palist->n ) /* 存在下标为p的元素 */
    return palist->elements[p];

  printf("Index of seq-list is out of range.\n ");
  return SPECIAL;  /* 返回一个表中没有的特殊值，需定义 */
}
```

参数出错是一类难处理的麻烦问题

这里采用的方式是提供信息并返回一个特殊值

这是一种处理方式，也有缺点

使用表的程序片段（假定 DataType 是 int ）：

```
PSeqList pl1 = createNullList_seq( ); /* 创建空顺序表 */
insert_seq(pl1, 0, 9);
insert_seq(pl1, 1, 5);
insert_seq(pl1, 1, 1);
for (i = 3; i < 10; ++i) insert_seq(pl1, i, i*3);
… …
```

直接定义表变量：

```
SeqList L1;  /* 注意，这时表不在合法状态 */
set_empty_seq(&L1);  /* 初始化，把表的 n 成分置为 0 */
insert_seq(&L1, 0, 9);
insert_seq(&L1, 1, 5);
… …
```

直接写 “L1.n = 0;”初始化好不好？

算法分析

- 在 n 元素的顺序表里下标 i（第i+1个）的元素前插入元素需要移动n - i 个元素；删除下标为i（第i+1个）的元素需要移动n - i - 1个元素。
- 若在位置 i 插入和删除元素的概率分别是 p_i 和 p_i'
- 则插入时的平均移动数是：
$$m_i=\sum_{i=0}^{n}(n-i)\,p_i$$
- 删除的平均移动数是：
$$m_d=\sum_{i=0}^{n-1}(n-i-1)\,p_i'$$

这里考虑的是平均复杂性，问题较复杂，因为这些操作的时间开销依赖于参数（与插入删除位置有关）和分布情况

- 顺序表插入和删除操作的平均时间代价为*O*(n)（最坏也是 O(n) ，具体计算见书）
- 特殊情况：后端插入/删除的时间代价为O(1)，可考虑独立实现backinsert_seq(PSeqList,DataType) 和 backdelete_seq(PSeqList)
- 根据元素值的定位操作 locate_seq，需顺序与表中元素比较。定位各位置的概率平均分布时，操作平均需做 n/2 次比较，时间 O(n)

顺序实现（顺序表）的优点：O(1) 时间（随机直接, 按位置）存取元素，元素存储紧凑，不需要附加空间

缺点：需要连续的大片空间；插入删除常要大量移动元素；需事先确定空间大小，有时事先难以做出估计。存储区中可能有大量空闲单元

简单的阶段性总结：

- 定义了线性表的概念（还给出了一种抽象模型）
- 设计了线性表应的一组基本操作，明确了操作的含义，这样一组描述实际上定义了一个抽象数据类型。还提出了另一些可能操作。
- 提出了一种实现模型——顺序表，讨论了顺序实现的问题
- 在C语言里给出了顺序表的一个具体实现

注意：

- 表抽象数据类型的设计可以有许多变化（操作集合的选择）
- 除顺序方式外，还可能找到其他实现模型
- 在C语言里实现，完全可能采用其他方式和技术

2.3 链接表示

- 2.3.1 单链表
- 2.3.2 循环链表
- 2.3.3 双链表

可通过在与元素相关的地方存放对其他元素的引用，显式指明元素之间的关系。这样建起的结构称为链接结构，用这种方式实现的表称为链接表（链表）

引用的最常见实现方式是借助语言中的指针。链接结构的最常见实现方式是利用动态存储分配功能

2.3.1 单链表

最简单的链表是单链表：为每个元素关联一个链接，表示后继关系（这时没有表示前驱关系的信息）

- 结点：元素的存储映象，包括数据域和指针域

- 单链表：与表中n个元素对应的n个结点通过指针链接成一个链表。链表中每个结点只有一个指针域

- 头指针：指示单链表第一个结点的存储位置的指针，整个链表的存取必须从头指针开始

	存储地址	数据域	指针域
	100	D	131
	107	B	113
头指针	113	C	100
125	119	F	NULL
	125	A	107
	131	E	119

一个链表的存储示例

head

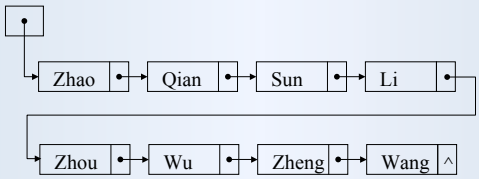


图2.6 一个线性链表的逻辑结构

实际中不需要关心结点的具体存储地址（与链接表结构无关），只需关心表的逻辑结构

链接表的操作也可以在逻辑结构上考虑

单链表的C实现

```
typedef struct Node Node, *PNode; /* 结点/指针类型*/
struct  Node {              /* 单链表结点结构 */
    DataType  info;
    PNode     link;
};
```

为提高可读性，可定义如下单链表类型：

```
typedef  struct Node  *LinkList;
```

定义链表头指针：

```
LinkList  llist;
```

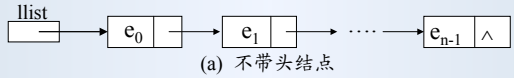

另一种定义方式（增加一层封装）：

```
typedef struct {    /* 单链表类型定义 */
    PNode  head;  /* 指向单链表中的第一个结点 */
} LinkList;
```

这种方式把链表的具体实现封装起来

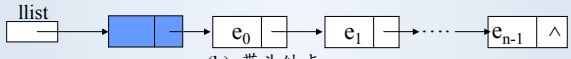
本书没采用这种方式

- 设llist是LinkList类型变量，指向单链表的首结点，如下图。表为空时，llist的值为NULL



(a) 不带头结点

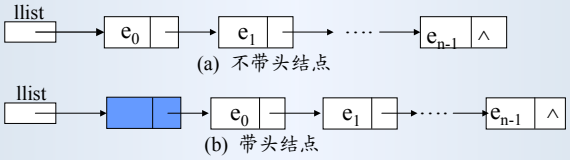
- 为方便运算实现，可在链表首结点前增加一个称为头结点的辅助结点（不属于表的内容）
- 头结点的info域可不存信息，或存放与整个表相关的信息（如元素个数），头结点的 link 域指向表的第一个实际结点，如下图



(b) 带头结点

无头结点时，在表头插入删除时需修改表头指针，在其他位置插入删除时修改其前驱结点的link，两种操作需要分别处理

有了头结点，操作的实现可以统一处理。因为表中每个“有效”结点都链接在另一结点的link域



(a) 不带头结点

(b) 带头结点

下面算法实现的是带头结点的单链表

单链表基本运算的实现

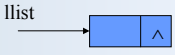
- 算法2.7 创建空单链表
  LinkList createNullList_link(void)
- 算法2.8 单链表的插入
  int insert_link(LinkList llist, int i, DataType x)
- 算法2.9 单链表的删除
  int delete_link(LinkList llist, DataType x)
- 算法2.10 在单链表中求某元素的存储位置
  PNode locate_link(LinkList llist, DataType x)
- 算法2.11 求单链表中下标为i的元素的存储位置
  PNode find_link(LinkList llist, int i)
- 算法2.12 判断单链表是否为空
  int isNullList_link(LinkList llist)

向下


```
/* 创建一个带头结点的空链表 */
LinkList createNullList_link( void ) {
    LinkList llist;
    /* 申请表头结点空间 */
    llist = (LinkList) malloc( sizeof( Node ) );
    if ( llist != NULL ) llist->link = NULL;
    return llist;
}
```

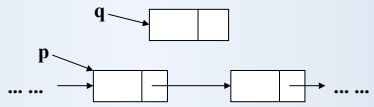
可以增加对错误情况的处理

操作正常完成后的状态:

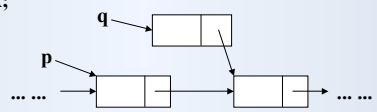


返回

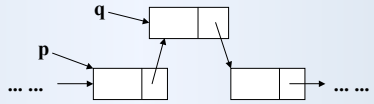
结点插入操作:



q->link = p->link;



p->link = q;



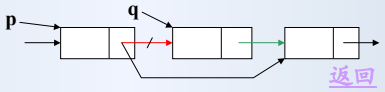
注意：指针更新次序不能错。


```
/*在带头结点单链表llist中下标i的(第i+1个)结点前插入元素x */
int insert_link(LinkList llist, int i, DataType x) {
    PNode p = llist, q;     int j;
    for (j = 0 ; p != NULL && j < i; j++)  p = p->link;
    if (j  !=  i) {          /* i<1或者大于表长 */
        printf("Index %d is out of range.\n", i); return FALSE;
    }
    q = (PNode)malloc( sizeof( Node ) );  /* 申请新结点 */
    if ( q == NULL ) {
        printf( "Out of space!\n" );    return FALSE;
    }
    /* 填充数据并插入链表中。*/
    q->info = x;    q->link = p->link;   p->link = q;
    return TRUE ;
}
```

返回


```
/* 在llist带有头结点的单链表中删除第一个值为x的结点 */
/* 这时要求 DataType 可以用 != 比较 */
int delete_link( LinkList llist, DataType x ) {
   PNode p = llist, q;
   /*找值为x的结点的前驱结点的存储位置 */
   while( p->link != NULL && p->link->info != x )  p = p->link;

   if( p->link == NULL ) {    /* 没找到值为x的结点 */
      printf("Datum does not exist!\n ");   return FALSE;
   }
   q = p->link;                    /* 找到值为x的结点 */
   p->link = p->link->link;   /* 删除该结点 */
   free( q );
   return TRUE;
}
```



返回

```
/* 在带头结点的单链表llist中找第一个值为x的结点 */
/* 找不到时返回空指针 */
PNode locate_link( LinkList llist, DataType x ) {
    PNode p;
    if (llist == NULL)  return NULL;

    for ( p = llist->link; p != NULL && p->info != x; )
        p = p->link;
    return p;
}
```

返回


```
/* 在带头结点单链表llist中找下标为i的(第i+1个)结点*/
/* 当表中无下标为i的(第i+1个)元素时，返回值为NULL */
PNode find_link( LinkList llist, int i ) {
    PNode p;
    int j;
    if (i < 0) {                                  /* 检查i的值 */
        printf("Index of llist is out of range.\n", i);
        return NULL;
    }

    for ( p = llist, j = 0; p != NULL && j <= i; j++)  p = p->link;

    if (p == NULL) printf("Index of llist is out of range.\n", i);

    return p;
}
```

返回

单链表与顺序表的比较

$$\text{存储密度} = \frac{\text{数据本身所占的存储量}}{\text{整个数据结构所占的存储量}}$$

(1) 单链表的link占用额外存储空间。但整个表的空间安排方式灵活；顺序表需要连续空间，要按估计的最大空间需求分配，所占存储里可能有大片闲置空间

(2) 顺序表支持对任一元素的O(1)按位置访问；单链表中只能顺链接逐个查找，按位置访问的时间为 O(n)

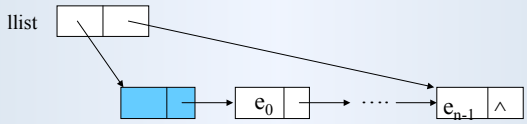
(3) 在单链表里的插入、删除运算不需要移动其它元素；而对顺序表可能需要移动许多元素

单链表适合于实现其中元素常常成批、顺序地处理的线性表

顺序表适合静态或大小可以估计的数据表示

单链表的前端插入和删除都是 O(1) 操作，一般位置的插入和删除是 O(n) 操作。

修改设计，增加表尾指针，可使后端插入变成 O(1) 操作:



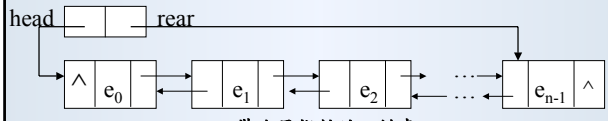
带表尾结点指针的链接表

采用这种表示，插入删除时需要考虑维护表尾指针，创建表的操作也需要修改

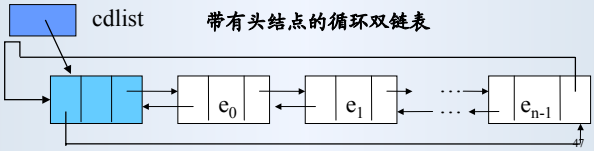
同样可以考虑有头结点和没有头结点的形式

2.3.3 双链表

其中的每个结点里设后继指针和前驱指针。既可以直接找到后继，也可以直接找到前驱



带头尾指针的双链表



带有头结点的循环双链表

双链表的实现

数据类型定义:

```
typedef struct DNode DNode,  *PNode; /* 类型 */

struct  DNode {              /* 双链表结点结构 */
   DataType  data;
   PNode     prev, next;
};

typedef  DNode  *DLinkList;  /* 双链表类型 */
```

双链表的一个特点：

有了指向一结点的指针，就可以把该结点从链表上取下（并保持链表的正常状态）

完成这一操作的语句序列（假定 p 指向被删结点）：

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

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


在一个结点的后面插入新结点

假定 q 指向新结点，p 指向表中结点（q 所指结点插入其后）

```
nxp = p->next;
q->next = nxp;
if (nxp != NULL) nxp->prev = q;
p->next = q;
q->prev = p;
```

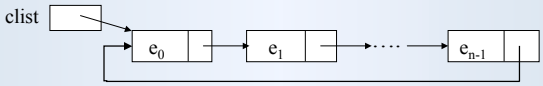
这里引进 nxp 只是为了使代码更清晰一些

在结点前插入新结点的操作与此类似

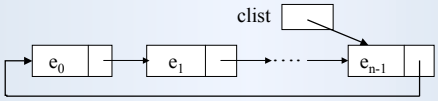
写链接表操作，需要特别注意指针赋值的顺序

2.3.2 循环链表

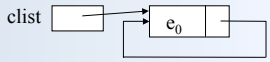
让单链表最后结点的指针指向头结点，就得到循环链表。优点：从表中任一结点出发都能访问所有结点



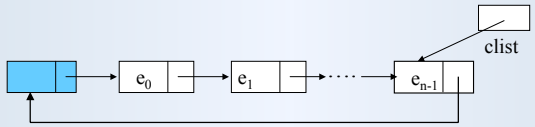
操作中经常要访问、修改首结点和末结点。让表头指针指向循环链表末结点，可以使对表头表尾的操作都方便



循环链接表插入第一个元素时需要特别处理：



也可以考虑带头结点的循环链表



实现循环链表

- 可以采用与单链表一样的数据类型定义
- 创建链表的操作需要修改。如有头结点的链表：

```
llist = (LinkList) malloc( sizeof( Node ) );
if ( llist != NULL ) llist->link = llist;
return llist;
```

- 无头结点的表插入第一个结点时，也需建立循环链接
- 判断空表的操作（有头结点的表）：

```
return llist->link = llist;
```

- 其他操作基本上可以照搬单链表的操作。根据所采用的设计，可能需要做些小调整

2.4 应用举例—Josephus问题

问题描述：

设有n个人围坐在一个圆桌周围，现从第s个人开始报数，数到第m的人出列，然后再从出列的下一人开始报数，数到第m的人出列，如此反复直到所有人全部出列。Josephus问题：对于任意给定的n，s和m，求出按出列次序得到的n个人员的序列。

以n=8，s=1，m=4为例，问题的求解过程如图2-15。图中s1指向开始报数位置，若初始的顺序为 n_1，n_2，n_3，n_4，n_5，n_6，n_7，n_8。则问题的解为n_4，n_8，n_5，n_2，n_1，n_3，n_7，n_6。

可用顺序表和循环单链表作为求解Josephus问题的程序的数据表示，用表元素表示人的编号（或名字）。

求解过程概要:

1）首先用线性表基本运算，如创建空线性表、插入元素等，构造出与具体Josephus问题实例对应的表;

2）从Josephus表中的第s个结点开始寻找、输出、删除表中第m个结点。然后从该结点后下一结点寻找、输出和删除表中第m个结点，重复此过程，直到删除完 Josephus表中所有元素

程序见书。虽然比较长，但意义很容易理解

顺序表实现中有如何形成循环计数的问题（取模）

顺序表的实现方式

前面讨论顺序表时采用的数据类型定义

```
#define NMAX 40
typedef struct SeqList {
    int n;        /* 元素个数n < NMAX  */
    DataType  elements[NMAX];
} SeqList;      /* SeqList 成为一个自定义类型名 */
```

这样做

优点：实现简单

缺点：不灵活，程序里通过 createNullList_seq 创建的每个 SeqList 的元素个数都一样

实际中常常需要几个表，但可能需要大小不同的表

可能希望下面这样的空表创建函数:

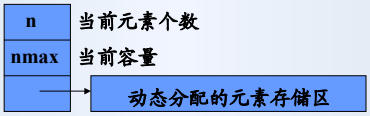
```
PSeqList createNullList_seq( int size );
```

用原类型就不行了，因为其中结构成员的数组大小是固定的

引进灵活性的方法是修改设计，把保存元素的数组和表的基本部分分开，动态分配保存表元素的数组

由于表中元素“数组”的大小（元素容量）在创建时才确定，不同的表可能不同，必须记录容量

修改的顺序表的基本布局图:




```
typedef struct {
   int n, nmax;
   DataType *elems;
} DSeqList, *PDSeqList;
```

nmax 记录表容量，elems 是指针，实际存储区动态分配

注意：为了灵活性，相关类型变复杂了

操作也会变得复杂一些:

- 创建操作需要重新写，比原来的复杂
- 还需要专门定义一个销毁表的函数
- 其他函数都不需要修改（还好）


```
PDSeqList createNullList_dseq( int size ) {
   PDSeqList lp = (PDSeqList)malloc(sizeof(DSeqList));
   if (lp != NULL) {
      lp->elems = (DataType*)malloc(sizeof(DataType)*size);
      if (lp->elems) {
         lp->n = 0;   lp->nmax = size; /* 设置表容量 */
         return lp;      /* 创建成功 */
      }
      else free(lp);      /* 存储分配失败，释放已分配存储*/
   }
   printf("Out of space!\n");  /*存储分配失败*/
   return NULL;
}
```

使用

```
PDSeqList list = createNullList_dseq(100);
```


销毁动态顺序表：

```
void destroy_dseq( PDSeqList lp) {
   free(lp->elems); /* 首先释放元素存储区 */
   free(lp);            /* 释放表结构 */
}
```

由于表元素的存储空间时独立分配的，必须专门删除。如果直接用（假定 dslp 指向一个动态顺序表）

```
free(dslp);
```

顺序表基本部分的释放了，元素存储区（另一存储块）就丢了

注意：现在不能简单地定义 DSeqList 变量并使用，因为没有元素存储区（可以专门定义函数来处理）

```
DSeqList list; /* 无元素存储区 */
```

动态顺序表

顺序表实现支持 O(1) 的元素访问，但需要静态（或创建时）确定容量。实际中容量有时很难确定

可否实现能在使用中动态改变大小的顺序表?

利用动态分配的技术，不难实现能动态变动大小的顺序表

现在的问题：在插入元素时可能遇到存储区满了的情况

解决办法：换一块更大的存储区

由于存储区是动态分配的，在运行中动态换一块并不困难（显示了指针和动态存储管理的威力）

元素存储区设定初始大小，插入时满则扩充。《从问题到程序》讨论了扩充策略和相应时间代价

类型定义与前面修改的顺序表类似:

```
enum { NBASE = 20 };
typedef int DataType; /* 根据需要确定 */
typedef struct {
   int n, nmax;
   DataType *elems;
} DSeqList, *PDSeqList;
```

elems 指向动态分配的元素存储区，nmax 是当时容量

表创建和前面动态分配存储区的顺序表类似，元素插入操作必须重写。表删除和其他操作可照搬

创建新的动态顺序表：

```
PDSeqList createNullList_dseq( void ) {
   PDSeqList lp = (PDSeqList)malloc(sizeof(DSeqList));
   if (lp != NULL) {
      lp->elems = (DataType*)malloc(sizeof(DataType)*NBASE);
      if (lp->elems) {
         lp->n = 0;      /*空表长度为0 */
         lp->nmax = NBASE;
         return lp;      /* 创建成功 */
      }
      else free(lp);     /* 存储分配失败，释放已分配存储*/
   }
   printf("Out of space!\n");  /*存储分配失败*/
   return NULL;
}
```


后端元素插入：

```
int backinsert_dseq(PDSeqList lp, DataType x) {
   if ( lp->n == lp->nmax ) { /* 存储区满，扩大一倍 */
      DataType *dp = (DataType*)realloc(lp->elems,
                                        lp->nmax*2*sizeof(DataType));
      if (dp == NULL) { /* 空间耗尽，已有元素还在原处 */
         printf("Seq-list overflow!\n");
         return FALSE;
      }
      lp->elems = dp;   lp->nmax *= 2;
   }

   lp->elems[n] = x;    lp->n++;       /* 插入，个数加1 */
   return TRUE;
}
```



```
int insert_dseq(PDSeqList lp, int p, DataType x) {
   int q;
   if ( p < 0 || p > lp->n ) { /* 不存在下标为p的元素 */
      printf("Index out of range! \n");    return FALSE;  }
   if ( lp->n == lp->nmax ) { /* 存储区满，扩大一倍 */
      DataType *dp = (DataType*)realloc(lp->elems,
                                        lp->nmax*2*sizeof(DataType));
      if (dp == NULL) { /* 空间耗尽，已有元素还在原处 */
         printf("Seq-list overflow!\n"); return FALSE;
      }
      lp->elems = dp;   lp->nmax *= 2;
   }
   for (q = lp->n - 1; q >= p; -- q)  /* 元素后移 */
      lp->elems[q+1] = lp->elems[q];
   lp->elems[p] = x;    lp->n++;       /* 插入，个数加1 */
   return TRUE;
}
```


复杂性分析:

按位置访问元素，O(1) 操作:

```
dslp->elems[i]
```

后端元素插入操作的情况比较复杂:

- 正常情况需要 O(1) 时间；当存储区满时可能需要把现有的元素复制到新存储区，因此是 O(n)
- 一次 O(n) 情况之后将有至少n 次正常情况（如果其间没有删除）。把该 O(n) 平均进去，总时间对操作的平均仍是常量（采用加倍的存储分配策略）
- 人们把这类情况称为分期付款式的 O(1) 复杂性

动态顺序表使用很广泛，被用在一些标准库的实现里

表结点空间的管理

- 静态数据结构：顺序表（用数组实现）
  在程序运行期间，数组大小不变化
- 动态数据结构：链表（通过许多小的结点实现）
  在程序运行期间，表大小（结点数）可自由变化

链表结点使用的空间通常通过系统提供的动态存储管理功能获得（C 语言库函数 malloc 和 free）

也可以自己管理，用一个数组实现某个类型的链表

- 建立一个数组（定义，或者动态分配）
- 数组元素是表结点结构，其 link 成分是整数
- link 成分里保存下一结点的数组下标作为（结点）链接

avail | 0 |

整型变量

数组元素通过整数下标“链接”起来。

整型变量 avail “指向”空闲单元形成的“链接表”，称为自由表（可用空间表）。遇到分配请求，就从自由表里“分配”结点

0		1
1		2
2		3
3		4
4		5
5		6
6		7
7		8
8		9
9		-1

用数组实现链表（初始状态）

list2 | 0 |

list1 | 4 |

avail | 8 |

0	d	6
1		3
2	c	-1
3		5
4	a	7
5		-1
6	e	-1
7	b	2
8		9
9		1

用数组实现链表（一个场景）

实现

结点结构： /* 根据需要定义类型，这里从略 */

```
struct NNode {
   DataType data;
   int link;
};
```

定义一个数组变量，其元素为 NNode 结构

```
enum { MEMSIZE = 1000 }; /* 根据需要确定 */
NNode array[MEMSIZE];
int avail = 0;
/* 把数组元素用 link 成分链接起来，-1 表示空链接 */
for (i = 0; i < MEMESIZE-1; ++i) array[i].link = i + 1;
array[MEMSIZE-1].link = -1;
```


用 -1 表示空链接，自然数表示正常链接（数组下标）

分配结点（link 是 int 类型的链接域），分配完成后 p 引用新分配的结点：

```
if (avail >= 0) {      p = avail; avail = array[avail].link;  }
```

释放结点（假定 p 是要释放结点的下标）：

```
array[p].link = avail;
avail = p;
```

基于上述语句可以定义自己的 mymalloc 和 myfree

表操作的实现与前面类似，但需注意链接的表示

小　　结

本章讨论了线性表的概念、存储表示和基本运算的实现。线性表是一种较简单的数据结构，是n个数据元素的有限序列。常用存储方式是顺序存储和链接存储。

在线性表的顺序存储实现中，元素之间的逻辑关系通过存储位置直接反映。其中只需存放数据元素自身信息，因此存储密度大、空间利用率高。另外，元素位置可以从下标出发通过简单算式算出，因此可以随机存取。这些是顺序存储结构的优点。

另一方面，顺序表中元素的插入和删除运算可能需要移动许多其它元素，长度变化较大的线性表必须按最大需要的空间分配存储，这些是线性表顺序存储结构的缺点。

在链接实现（链接表）里，元素间的关系通过显式链接表示，因此表中元素可以任意存放，灵活，可以通过修改链接的方式修改表的结构。但表中的链接占用存储空间，而且只能顺着链接查找所需元素，不能随机访问。