第二节：线性结构

2.1 线性表及其实现

2.1.1 引子：多项式表示

例如：如何表示一个多项式？

方法 1：顺序存储结构直接表示

例如 $f(x)=4x^5-3x^2+1$，则可以表示为：

下标 i	0	1	2	3	4	5
a[i]	1	0	-3	0	0	4

但是如何表示多项式 $x+x^{2000}$ 呢？

方法 2：顺序存储结构表示非零项

每个非零项涉及两个信息：系数和指数。可以将一个多项式看成一个(a_i,j)二元组的集合。

用结构数组来表示：数组分量有系数 a_i 和指数 i 注册的结构，对应一个非零项。

下标 i	0	1
系数 ai	1	1
指数 i	1	2000

方法 3：链表结构存储非零项

链表中每个**结点**存储多项式中的一个**非零项**，包括**系数**和**指数**两个数据域以及一个**指针域**。

coef	expon	link

代码如下：

```
typedef struct PolyNode *Polynomial;
struct PolyNode{
    int coef;
    int expon;
    Polynomial links;
}
```

2.1.2 线性表及顺序存储

线性表是由同类型**数据元素**构成**有序序列**的线性结构。表中元素的个数被称为线性表的**长度**，线性表没有元素时被称为**空表**，表起始位置称为**表头**，结束位置为**表尾**。

线性表的抽象数据类型描述：

类型名称：线性表

数据对象集：线性表是 n（≥0）个元素构成的有序序列。

操作集：线性表 L∈List，整数 i 表示位置，元素 X∈ElementType。线性表主要操作有：

（1）List MakeEmpty()：初始化一个空线性表 L；

（2）ElementType FindKth(int K, List L)：根据位序 K，返回相应元素；

（3）int Find(ElementType X, List L)：在线性表 L 中查找 X 的第一次出现的位置；

（4）void Insert(ElementType X, int i , List L)：在位序 i 前插入一个新元素 X；

（5）void Delete(inti , List L)：删除指定位序 i 的元素；

（6）int Length(List L)：返回线性表 L 的长度 n。

线性表的顺序存储实现：利用数组的**连续存储空间顺序存放**线性表的各元素。

代码：

```
typedef struct LNode *List;
struct LNode{
    ElementType Data[MAXSIZE];
    int Last;
};
struct LNode L;
List PtrL;
```

访问下标为 i 的元素：L.Data[i]或 PtrL->Data[i]

线性表的长度：L.Last+1 或 PtrL->Last+1

主要操作的实现：

图 1

查找成功的平均比较次数为(n+1)/2，平均时间性能为 O(n)。

3.插入（第 i（1≤i≤n+1）个位置插入一个值为 X 的新元素）

先将第 i 个以后的元素全部往后挪一位（从最后一个元素开始倒着往后挪）

代码如下：

图 2

平均移动次数为 n/2，平均时间性能为 O(n)。

4.删除（删除表的第 i（1≤i≤n）个位置上的元素）

先把第 i 个元素删除，然后把第 i+1 到最后一个元素以此往前挪。

代码如图 3 所示。

图 3

平均移动次数为(n-1)/2，平均时间性能为 O(n)。

2.1.3 链式存储及操作

它**不要求逻辑上相邻的两个元素物理上也相邻**，而是通过“**链**”建立起数据元素之间的逻辑关系。

插入、删除**不需要**移动数据元素，只需要修改“链”。如图 4 所示，是一个链式存储的结构和代码。

图 4

主要操作的实现：

1.求表长

```
int Length(List PtrL)
{
    List p =PtrL; /*p 指向表的第一个结点*/
    int j = 0;
    while(p){
        p = p->Next;
        j++;/*当前 p 指向的是第 j 个结点*/
    }
```

```
    return j;
}
```

时间性能是 O(n)。

2.查找

（1）按照序号查找：FindKth

代码：

```
List FindKth(int K, List PtrL){
    Lst p = PtrL;
    int i =1;
    while(p!=NULL&&i<K){
        p = p->Next;i++;
    }
    if(i==K)
        return p;/*找到第 K 个，返回指针*/
    else return NULL;
}
```

（2）按值查找：Find

代码：

```
List Find(ElementType X, List PtrL){
    Lst p = PtrL;
    int i =1;
    while(p!=NULL && p->Data!=X)
        p=p->Next;
    return p;
}
```

时间性能是 O(n)。

3.插入（第 i-1（1≤i≤n+1）个结点后插入一个值为 X 的新结点）

（1）先构造一个新结点，用 s 指向；

（2）再找到链表的第 i-1 个结点，用 p 指向；

（3）然后修改指针，插入结点（p 之后插入新结点是 s）。

代码如图 5 所示。

图 5

4.删除（删除链表的第 i（1≤i≤n）个位置上的结点）

（1）先找到链表的第 i-1 个结点，用 p 指向；

（2）再用指针 s 指向要被删除的结点；

（3）然后修改指针，删除 s 所指向结点；

（4）最后释放 s 所指结点的空间（free(s)）。

实现代码见图 6。

图 6

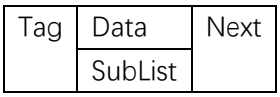
2.1.4 广义表与多重链表

1.广义表

图 7

所谓广义表（Generalized List）是**线性表的推广**。对于线性表来说，n 个元素都是基本的**单元素**；而对于广义表来说，这些元素不仅可以是单元素，也可以是另一个**广义表**。

类似如下：



代码如下：

```
typedef struct GNode *GList;
struct GNode{
    int Tag;  /*标志域：0 表示结点是单元素，1 表示结点是广义表*/
    union{      /*子表指针域 Sublist 与单元素数据域 Data 复用，即共用存储空间*/
      ElementType Data;
      GList SubList;
    }URegion;
    GList Next; /*指向后继结点*/
};
```

2.多重链表

多重链表：链表中的结点可能同时隶属于多个链。

多重链表中结点的指针域会有**多个**，如前面例子中包含了 Next 和 SubList 两个指针域。

但包含两个指针域的链表并不一定是多重链表，如**双向链表**不是多重链表。

多重链表有广泛的用途：基本上如**树、图**这样相对复杂的数据结构都可以用多重链表方式来实现存储。

例：见图 8

图 8

矩阵 A 的多重链表图如图 9 所示。

图 9

入口处的 Term 这个结点的值是指这个矩阵有 4 行 5 列，非零值有 7 个。

总的来说，可以做出以下总结。

图 10

2.2 堆栈

2.2.1 什么是堆栈

堆栈是一种线性结，也是一个具有一定操作约束的线性表。**后入先出（LIFO）**。

堆栈只在一端（栈顶，Top）做**插入（入栈，Push）、删除（出栈，Pop）**操作。其抽象数据类型描述如下图所示。

类型名称：堆栈（Stack）

数据对象集：一个有0个或多个元素的有穷线性表。

操作集：长度为MaxSize的堆栈S ∈ Stack，堆栈元素item ∈ ElementType

1、Stack CreateStack(int MaxSize)：生成空堆栈，其最大长度为MaxSize；
2、int IsFull(Stack S, int MaxSize)：判断堆栈S是否已满；
3、void Push(Stack S, ElementType item)：将元素item压入堆栈；
4、int IsEmpty (Stack S)：判断堆栈S是否为空；
5、ElementType Pop(Stack S)：删除并返回栈顶元素；

2.2.2 堆栈的顺序存储实现

栈的顺序存储结构通常由一个**一维数组**和**一个记录栈顶元素位置的变量**组成。

```
#define MaxSize <储存数据元素的最大个数>
typedef struct SNode *Stack;
struct SNode{
    ElementType Data[MaxSize];
    int Top;
}
```

常用操作：

（1）入栈

```
void Push(Stack PtrS, ElementType item)
{
    if(PtrS->Top==MaxSize-1){
        printf("堆栈满";
        return;
    }else{
        PtrS->Data[++(PtrS->Top)]=item;
        return;
    }
}
```

PtrS 是一个结构指针，准备入栈的元素是 ElementType 类型的。首先要判别堆栈满不满，由于下标从 0 开始，所以 Top 指向 MaxSize-1 时就代表满了。如果没满，则开始放数据。放数据的操作，首先将 Top 这个指针加 1，也就是移到上面去，然后把数据存进去，这行代码可以分解为：

```
(PtrS->Top)++;
PtrS->Data[PtrS->Top]=item;
```

（2）出栈

```
ElementType Pop(Stack PtrS)
{
    if(PtrS->Top==-1)//即代表堆栈是空的
    {
```

```
            printf("堆栈空");
            return ERROR;//ERROR 是 ElementType 的特殊值，标志错误
        }else
            return(PtrS->Data[(PtrS->Top)--];
    }
```

例：请用一个数组实现两个堆栈，要求最大的利用数组空间，使数组只要有空间入栈操作就可以成功。

分析：可以将数组这样划分成两个，然后相向而行。

->	<-

如何判断满了呢？不能直接把两个 Top 相加，因为两个堆栈的 Top 都是从数组最左端为起点的。一种比较聪明的方法是使这两个栈分别从数组的两头开始向中间生长；**当两个栈的栈顶指针相遇时，表示两个栈都满了**。

代码：

```
#define MaxSize <存储数据元素的最大个数>
struct DStack {
    ElementType Data[MaxSize];
    int Top1;       /* 堆栈 1 的栈顶指针*/
    int Top2;       /* 堆栈 2 的栈顶指针*/
} S;
S.Top1 = -1;
S.Top2 = MaxSize;
```

```
void Push( struct DStack *PtrS, ElementType item, int Tag )
{  /* Tag作为区分两个堆栈的标志，取值为1和2 */
   if ( PtrS->Top2 - PtrS->Top1 == 1) {   /*堆栈满*/
     printf("堆栈满");   return ;
   }
   if ( Tag == 1 ) /* 对第一个堆栈操作 */
      PtrS->Data[++(PtrS->Top1)] = item;
   else                /* 对第二个堆栈操作 */
      PtrS->Data[--(PtrS->Top2)] = item;
}
```

```
ElementType Pop( struct DStack *PtrS, int Tag )
{  /* Tag作为区分两个堆栈的标志，取值为1和2  */
    if ( Tag == 1 ) {   /* 对第一个堆栈操作  */
       if ( PtrS->Top1 == -1 ) { /*堆栈1空 */
          printf("堆栈1空");   return NULL;
       } else return PtrS->Data[(PtrS->Top1)--];
    } else {            /* 对第二个堆栈操作 */
       if ( PtrS->Top2 == MaxSize ) { /*堆栈2空 */
            printf("堆栈2空");  return NULL;
       } else  return PtrS->Data[(PtrS->Top2)++];
  }
}
```

2.2.3 堆栈的链式存储实现

栈的链式存储结构实际上就是一个单链表，叫做链栈。插入和删除操作只能在链栈的栈顶进行。**Top 在链表头上。**

```
typedef struct SNode *Stack;
struct SNode{
    ElementType Data;
    struct SNode *Next;
};
```

（1）堆栈初始化；（2）判断堆栈 S 是否为空

代码：

```
Stack CreateStack()
{
    /*构建一个堆栈的头结点，返回指针*/
    Stack S;
    S=(Stack)malloc(sizeof(struct SNode));
    S->Next =NULL;
    return S;
}

int IsEmpty(Stack S)
{
    /*判断堆栈 S 是否为空，若为空返回整数 1，否则返回 0*/
    return(S->Next==NULL;
}
/*将元素 item 压入堆栈 S*/
void Push(ElementType item, Stack S)
{
    struct SNode *TmpCell;
    TmpCell = (struct SNode *)malloc(sizeof(struct SNode));
    TmpCell->Element = item;
    TmpCell->Next=S->Next;
    S->Next=TmpCell;
}
```

```
ElementType Pop(Stack S)
{  /* 删除并返回堆栈S的栈顶元素 */
   struct SNode *FirstCell;
   ElementType TopElem;
   if( IsEmpty( S ) ) {
       printf("堆栈空");  return NULL;
   } else {
       FirstCell = S->Next;
       S->Next = FirstCell->Next;
       TopElem = FirstCell ->Element;
       free(FirstCell);
       return TopElem;
   }
}
```

2.2.4 堆栈应用：表达式求值

1.中缀表达式如何转换为后缀表达式?

从头到尾读取中缀表达式的每个对象，对不同对象按不同的情况处理。

（1）运算数：直接输出；

（2）左括号：压入堆栈；

（3）右括号：看到右括号，就立即把栈顶的运算符弹出并输出，直到遇到左括号（出栈，不输出）

（4）普通运算符：

①若优先级大于栈顶运算符时，则把它压栈；

②若优先级小于等于栈顶运算符时，则将栈顶运算符弹出并输出；再比较新的栈顶运算符，直到该运算符大于栈顶运算符优先级为止，然后将该运算符压栈。

（5）若各对象处理完毕，则把堆栈中存留的运算符一并输出。

（6）若栈里面有左括号，则后面遇到右括号之前的运算符都要比左括号高，直接压栈。当然注意的是，当左括号右边与右括号左边之间有多个运算符时，仍按照原来的优先级执行，即按照（4）中进行。

2.堆栈的其他应用：

（1）函数调用及递归实现；

（2）深度优先搜索；

（3）回溯算法等。

2.3 队列及实现

2.3.1 队列及顺序存储实现

队列是一种具有一定操作约束的线性表。它的插入和删除操作：只能在一端插入，而在另一端删除，也就是**先入先出（FIFO）**。

队列的抽象数据类型描述如下图所示。

类型名称：队列(Queue)

数据对象集：一个有0个或多个元素的有穷线性表。

操作集：长度为MaxSize的队列Q ∈ Queue，队列元素item ∈ ElementType

1、Queue CreatQueue(int MaxSize)：生成长度为MaxSize的空队列；

2、int IsFullQ(Queue Q, int MaxSize)：判断队列Q是否已满；

3、void AddQ(Queue Q, ElementType item)： 将数据元素item插入队列Q中；

4、int IsEmptyQ(Queue Q)： 判断队列Q是否为空；

5、ElementType DeleteQ(Queue Q)：将队头数据元素从队列中删除并返回。

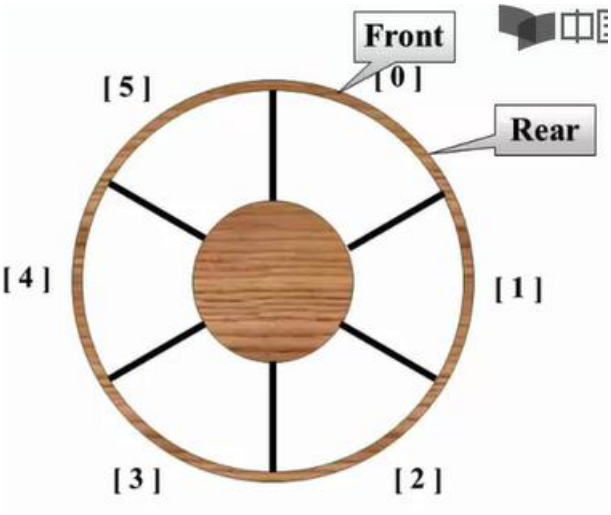
队列的顺序存储实现：

队列的顺序存储结构通常由一个一位数组和一个记录队列头元素位置的变量 front 以及一个记录队列尾元素位置的变量 rear 组成。

```
#define MaxSize <储存数据元素的最大个数>
struct QNode{
    ElementType Data[MaxSize];
    int rear;
    int front;
};
```

typedef struct QNode *Queue;

顺环队列



意思就是当后面的填完了，前面的去除了几个，再添加的话，就添加到前面去，这样的结果会造成 Front==Rear，这样的话，队列是空还是满呢？

为了解决这个问题，可以：使用额外标记：Size（加入一个元素是 size+1，删除一个元素的时候 size-1）或者 tag（加入一个元素 tag 为 1，删除的时候 tag 为 0，这个代表最后一个操作是添加还是删除）域；仅使用 n-1 数组空间。

队列顺序存储的常用操作：

（1）入队列

```
void AddQ(Queue PtrQ, ElementType item)
{
    if((PtrQ->rear+1)%MaxSize ==PtrQ->front){
        printf("队列满");
        return;
    }
    PtrQ->rear = (PtrQ->rear+1)%MaxSize;
    PtrQ->Data[PtrQ->rar]=item;
}
```

（2）出队列

```
ElementType DeleteQ(Queue PtrQ)
{
    if(PtrQ->front ==PtrQ->rear){
        printf("队列空");
        return ERROR;
    }else{
        PtrQ->front = (PtrQ->front+1)%MaxSize;
        return PtrQ->Data[PtrQ->front];
    }
}
```

2.3.2 队列的链式存储实现

队列的链式存储也可以使用一个单链表来实现。插入和删除操作分别在链表的两头进行。

front 在最头上，rear 在结尾。

```
struct Node{
    ElementType Data;
    struct Node *Next;
};
struct QNode{
    /*链队列结构*/
    struct Node *rear;/*指向队尾结点*/
    struct Node *front;/*指向队头结点*/
};
typedef struct QNode *Queue;
Queue PtrQ;
```

❖不带头结点的链式队列出队操作的一个示例：

```
ElementType DeleteQ ( Queue  PtrQ )
{   struct Node  *FrontCell;
    ElementType FrontElem;

    if  ( PtrQ->front == NULL) {
        printf("队列空");    return ERROR;
    }
    FrontCell = PtrQ->front;
    if ( PtrQ->front == PtrQ->rear)    /* 若队列只有一个元素 */
        PtrQ->front = PtrQ->rear = NULL;  /* 删除后队列置为空 */
    else
        PtrQ->front = PtrQ->front->Next;
    FrontElem = FrontCell->Data;
    free( FrontCell );                  /* 释放被删除结点空间  */
    return  FrontElem;
}
```

2.4 多项式的加法运算实现

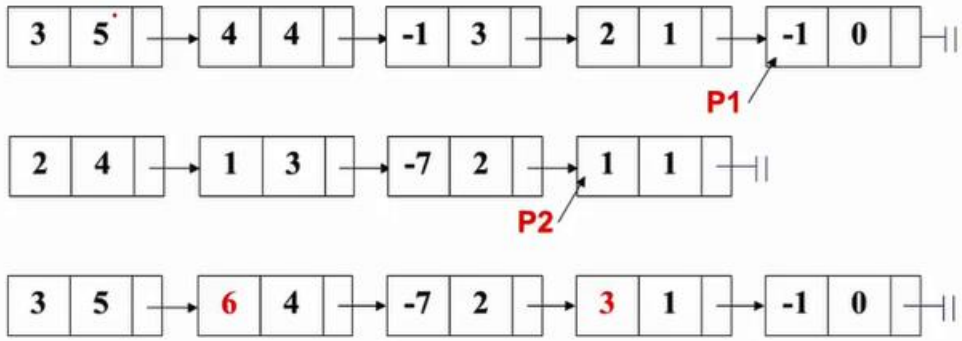
采用不带头结点的单向链表，按照指数递减的顺序排列各项。

多项式加法运算

算法思路：两个指针P1和P2分别指向这两个多项式第一个结点，不断循环：

- **P1->expon==P2->expon**: 系数相加，若结果不为0，则作为结果多项式对应项的系数。同时，P1和P2都分别指向下一项；
- **P1->expon>P2->expon**: 将P1的当前项存入结果多项式，并使P1指向下一项；
- **P1->expon<P2->expon**: 将P2的当前项存入结果多项式，并使P2指向下一项；

当某一多项式处理完时，将另一个多项式的所有结点依次复制到结果多项式中去。




```
Polynomial  PolyAdd (Polynomial P1, Polynomial P2)
{
        Polynomial front, rear, temp;
        int sum;
        rear = (Polynomial) malloc(sizeof(struct PolyNode));
        front = rear;       /* 由front 记录结果多项式链表头结点 */
        while ( P1 && P2 ) /* 当两个多项式都有非零项待处理时 */
                switch ( Compare(P1->expon, P2->expon) ) {
                case 1:
                        Attach( P1->coef, P1->expon, &rear);
                        P1 = P1->link;
                        break;
                case -1:
                        Attach(P2->coef, P2->expon, &rear);
                        P2 = P2->link;
                        break;
                case 0:
                        sum = P1->coef + P2->coef;
                        if ( sum ) Attach(sum, P1->expon, &rear);
                        P1 = P1->link;
                        P2 = P2->link;
                        break;
                }
        /* 将未处理完的另一个多项式的所有节点依次复制到结果多项式中去 */
        for ( ; P1; P1 = P1->link ) Attach(P1->coef, P1->expon, &rear);
        for ( ; P2; P2 = P2->link ) Attach(P2->coef, P2->expon, &rear);
        rear->link = NULL;
        temp = front;
        front = front->link; /*令front指向结果多项式第一个非零项 */
        free(temp);          /* 释放临时空表头结点 */
        return front;
```

```
void Attach( int c, int e, Polynomial *pRear )
{       Polynomial P;

        P = (Polynomial)malloc(sizeof(struct PolyNode));
        P->coef = c;   /* 对新结点赋值 */
        P->expon = e;
        P->link = NULL;
        (*pRear)->link = P;
        *pRear = P;        /* 修改pRear值 */
}
```