



LATEX *main*

Sun for morning, moon for night, and you forever.

作者：Lyshmily.Y & 木易

组织：Lyshmily.Y

时间：September 8, 2024

版本：V.1.0

邮箱：yjlpku.outlook.com & 845307723@qq.com



在没有结束前，总要做很多没有意义的事，这样才可以在未来某一天，用这些无
意义的事去堵住那些讨厌的缺口



目录

	A
第1章 绪论	1
1.1 数据结构.....	1
1.2 算法和算法分析.....	2
第2章 线性表	3
2.1 线性表的定义和基本操作.....	3
2.2 线性表的顺序表示.....	4
2.2.1 顺序表	4
2.2.1.1 顺序表定义和函数声明	4
2.2.1.2 顺序表初始化	5
2.2.1.3 顺序表增加	5
2.2.1.4 顺序表插入	6
2.2.1.5 顺序表删除	6
2.2.1.6 顺序表查找	6
2.2.1.7 顺序表辅助函数	7
2.3 线性表的链式表示.....	8
2.3.1 单链表	8
2.3.1.1 单链表定义和函数声明	8
2.3.1.2 单链表初始化	8
2.3.1.3 单链表查找	9
2.3.1.4 单链表插入	9
2.3.1.5 单链表删除	11
2.3.1.6 单链表辅助函数	11
2.3.1.7 头插法 & 尾插法	12

2.3.1.8 单链表反转	13
2.3.2 双链表	13
2.3.2.1 双链表定义和函数声明	14
2.3.2.2 双链表初始化	14
2.3.2.3 双链表查找	14
2.3.2.4 双链表插入	15
2.3.2.5 双链表删除	17
2.3.2.6 双链表辅助函数	17
2.3.2.7 头插法 & 尾插法	18
2.3.2.8 双链表反转	19
2.3.3 循环链表	19
2.3.4 静态链表	20
2.4 Summary	21
2.4.1 顺序表与链表比较	21
2.4.2 单链表、双链表、循环链表比较	22
第3章 栈和队列	23
3.1 栈	23
3.1.1 栈定义和函数声明	24
3.1.1.1 顺序栈、链栈、共享栈定义	24
3.1.1.2 函数声明	24
3.1.2 顺序栈	24
3.1.2.1 顺序栈初始化	25
3.1.2.2 顺序栈出入栈	25
3.1.2.3 顺序栈辅助函数	25
3.1.3 链栈	26
3.1.3.1 链栈初始化	27
3.1.3.2 链栈出入栈	27
3.1.3.3 链栈辅助函数	27
3.1.4 共享栈	29
3.1.4.1 共享栈初始化	29
3.1.4.2 共享栈出入栈	29
3.1.4.3 共享栈辅助函数	30
3.2 队列	32
3.2.1 队列定义和函数声明	33
3.2.1.1 循环队列、链表队列定义	33
3.2.1.2 函数声明	33



3.2.2 循环队列	33
3.2.2.1 循环队列初始化	34
3.2.2.2 循环队列出入队	34
3.2.2.3 循环队列辅助函数	34
3.2.3 链队列	36
3.2.3.1 链队列初始化	36
3.2.3.2 链队列出入队	36
3.2.3.3 链队列辅助函数	37
3.2.4 双端队列	38
3.3 栈和队列的应用	38
3.3.1 括号匹配	38
3.3.2 前缀、中缀、后缀表达式求值	38
3.3.3 表达式转换	39
3.3.4 队列应用	39
3.4 数组	39
3.4.1 特殊矩阵的压缩	39





第1章 绪论

◆ 1.1 数据结构

定义 1.1.1 (基本术语)

1. **数据 (Data):** 客观事物的符号, 是所有能输入到计算机中并被计算机程序处理的符号的集合.
2. **数据元素 (Data Element):** 数据的基本单位, 也被称为元素、记录, 数据元素用于完整地描述一个对象, 比如一名学生记录、树中棋盘的一个格局和图中的一个顶点.
3. **数据项 (Data Item):** 是组成数据元素的、有独立含义的、不可分割的最小单位. 一个数据元素可以由一个或多个数据项组成. 比如一名学生记录中包含学号、姓名、性别、年龄等数据项.
4. **数据对象 (Data Object):** 性质相同的数据元素的集合, 是数据的一个子集. 比如学生集合、图中的顶点集合.
5. **数据结构 (Data Structure):** 是相互之间存在一种或者多种特定关系的数据元素的集合



定义 1.1.2 (数据结构三要素)

1. 逻辑结构: 集合、线性结构(一对一)、树形结构(一对多)、图形结构(多对多)
2. 物理结构: 顺序存储结构(逻辑和物理都相邻)、链式存储结构(指针表示)、散列存储
3. 数据运算: 初始化、插入、删除、查找、更改、遍历



定义 1.1.3 (数据类型和抽象数据类型)

1. 数据类型 (**Data Type**): 一个值的集合和定义在此集合上的一组操作的总称
2. 抽象数据类型 (**Abstract Data Type, ADT**): 是指一个数学模型以及定义在此数学模型上的一组操作, 具体包括数据对象、数据对象上关系的集合以及对数据对象的基本操作的集合

1.2 算法和算法分析**定义 1.2.1 (算法的定义及其特性)**

一、算法: 是解决特定问题求解步骤的描述, 是指令的有限序列

二、算法的特性:

1. 有穷性: 算法在执行有限步之后终止
2. 确定性: 算法的每一步骤都有确切的含义, 不会出现二义性
3. 可行性: 算法的每一步都是可行的, 可以通过执行有限次完成
4. 输入: 一个算法有零个或多个输入
5. 输出: 一个算法有一个或多个输出

三、评价算法优劣的标准:

1. 正确性: 算法的输出结果符合要求
2. 可读性: 算法是容易阅读和理解的
3. 健壮性: 算法对不合理数据有较好的处理能力
4. 高效性: 包括时间和空间两个方面, 时间高效指的是算法设计合理, 执行效率高, 空间高效指的是算法执行过程中所需的存储空间最小





第 2 章 线性表

2.1 线性表的定义和基本操作

定义 2.1.1 (线性表)

$$L = (a_1, a_2, a_3, \dots, a_i, a_{i+1}, \dots, a_n)$$

1. 有相同数据类型的 n 个数据元素的有限序列
2. 存在唯一的一个被称作“第一个”的数据元素
3. 存在唯一的一个被称作“最后一个”的数据元素
4. 除第一个外，每个数据元素有且仅有一个直接前驱
5. 除最后一个外，每个数据元素有且仅有一个直接后继



定义 2.1.2 (线性表基本操作)

1. **InitList(&L)** 初始化线性表，构造一个空的线性表 L
2. **DestroyList(&L)** 销毁线性表，销毁线性表 L
3. **ListInsert(&L, i, e)** 插入操作，在线性表 L 的第 i 个位置插入元素 e
4. **ListDelete(&L, i, &e)** 删除操作，删除线性表 L 的第 i 个位置的元素，并用 e 返回其值
5. **GetElem(L, i, &e)** 取值操作，返回线性表 L 的第 i 个位置的元素
6. **LocateElem(L, e)** 定位操作，在线性表 L 中查找与给定值 e 相等的元素
7. **Length(L)** 求表长，返回线性表 L 的元素个数
8. **Empty(L)** 判空操作，若 L 为空表，则返回 TRUE，否则返回 FALSE
9. **PrintList(L)** 输出操作，按顺序输出线性表 L 的所有元素



2.2 线性表的顺序表示

2.2.1 顺序表

定义 2.2.1 (顺序表)

1. 用顺序存储的方式实现线性表
2. 顺序存储: 把逻辑上相邻的元素存储在物理上相邻的存储单元中, 元素之间的关系由存储单元之间的邻接关系来体现



```
1 #include <assert.h>
2 #include <stdbool.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <time.h>
7 #include <math.h>
8
9 typedef int Status;
10
11 #define OK 1
12 #define ERROR 0
13 #define OVERFLOW -1
```

2.2.1.1 顺序表定义和函数声明

```
1 // 定义顺序表
2 typedef struct{
3     int length;
4     int MaxSize;
5     int* data;
6 }SqList;
7
8 // 函数声明
9
10 Status InitList(SqList * L); // 初始化
11 Status IncreaseSize(SqList * L, int len); // 增加表长
12 Status ListInsert(SqList * L, int i, int e); // 插入
13 Status ListDelete(SqList * L, int i, int* e); // 删除
14 int LocateElem(SqList L, int e); // 按值查找
15 Status GetElem(SqList L, int i, int* e); // 按位查找
16 int Length(SqList L); // 表长
17 Status Empty(SqList L); // 判空
18 void PrintList(SqList L); // 打印
19 void Printelem(int e); // 打印单个数据元素
```



2.2.1.2 顺序表初始化

```
● ● ●  
1 // 初始化  
2 Status InitList(SqList * L)  
3 {  
4     L->data = malloc(INITSIZE * sizeof(int));  
5     if (!L->data)  
6     {  
7         printf("内存不足，分配失败!\n");  
8         return OVERFLOW;  
9     }  
10    L->length = 0;  
11    return OK;  
12 }
```

2.2.1.3 顺序表增加

```
● ● ●  
1 // 增加表长  
2 Status IncreaseSize(SqList * L, int len)  
3 {  
4     int* p = L->data;  
5     L->data = (int*)malloc((L->MaxSize + len) * sizeof(int));  
6     if (!L->data)  
7     {  
8         printf("内存不足，分配失败!\n");  
9         return OVERFLOW;  
10    }  
11    for (int i = 0; i < L->length; i++)  
12    {  
13        L->data[i] = p[i];  
14    }  
15    L->MaxSize += len;  
16    free(p);  
17    return OK;  
18 }
```



2.2.1.4 顺序表插入

```

1 // 插入
2 Status ListInsert(SqlList * L, int i, int e)
3 {
4     if (i < 1 || i > L->length + 1)
5     {
6         printf("插入位置不合法!\n");
7         return ERROR;
8     }
9     if (L->length >= L->MaxSize) IncreaseSize(L, INITSIZE);
10    for (int j = L->length - 1; j >= i-1; j--)
11    {
12        L->data[j + 1] = L->data[j];
13    }
14    L->data[i - 1] = e;
15    L->length++;
16    return OK;
17 }
```

2.2.1.5 顺序表删除

```

1 // 删除
2 Status ListDelete(SqlList * L, int i, int* e)
3 {
4     if (i < 1 || i > L->length)
5     {
6         printf("删除位置不合法!\n");
7         return ERROR;
8     }
9     *e = L->data[i - 1];
10    for (int j = i; j < L->length; j++)
11    {
12        L->data[j - 1] = L->data[j];
13    }
14    L->length--;
15    return OK;
16 }
```

2.2.1.6 顺序表查找

(1). 按位置查找

```

1 // 按位查找
2 Status GetElem(SqlList L, int i, int* e)
3 {
4     if (i < 1 || i > L.length)
5     {
6         printf("查找位置不合法!\n");
7         return ERROR;
8     }
9     *e = L.data[i - 1];
10    return OK;
11 }
```



(2). 按值查找

```
● ● ●  
1 // 按值查找  
2 int LocateElem(SqList L, int e)  
3 {  
4     for (int i = 0; i < L.length; i++)  
5     {  
6         if (L.data[i] == e)  
7             return i + 1;  
8     }  
9     printf("查找元素不存在!\n");  
10    return ERROR;  
11 }
```

2.2.1.7 顺序表辅助函数

(1). *Length*

```
● ● ●  
1 // 表长  
2 int Length(SqList L)  
3 {  
4     return L.length;  
5 }
```

(2). *Empty*

```
● ● ●  
1 // 判空  
2 Status Empty(SqList L)  
3 {  
4     if (L.length == 0)  
5         return OK;  
6     else  
7         return ERROR;  
8 }
```

(3). *PrintList*

```
● ● ●  
1 // 打印  
2 void PrintList(SqList L)  
3 {  
4     for (int i = 0; i < L.length; i++)  
5     {  
6         Printelem(L.data[i]);  
7     }  
8     printf("\n");  
9 }  
10  
11 void Printelem(int e)  
12 {  
13     printf("%d ", e);  
14 }
```



2.3 线性表的链式表示

2.3.1 单链表

定义 2.3.1 (单链表)

1. 单链表第一个元素在删除和增加需要特殊处理
2. 头插法和尾插法的区别, 头插法实现反置
3. 在增删、初始化操作传入引用, 在判空、打印以及求表长传入值



2.3.1.1 单链表定义和函数声明

```

1 // 定义链表节点
2 typedef struct LNode{
3     int data;
4     struct LNode *next;
5 }LNode;
6 typedef LNode* LinkList;
7 // 函数声明
8 Status InitList(LinkList *L); // 初始化
9 Status Empty(LinkList L); // 判空
10 LNode * LocateElem(LinkList L, int e); // 按值查找
11 LNode * GetElem(LinkList *L, int i); // 按位查找
12 Status InsertNextNode(LNode *p, int e); // 指定元素后插入
13 Status InsertPreNode(LNode *p, int e); // 指定元素前插入
14 Status ListInsert(LinkList *L, int i, int e); // 按位置插入
15 Status ListDelete(LinkList *L, int i, int* e); // 删除
16 Status DeleteNode(LNode *p); // 删除指定节点
17 int Length(LinkList L); // 表长
18 void PrintList(LinkList L); // 打印链表
19 void PrintNode(LNode p); // 打印节点
20 LinkList TailInsert(LinkList *L,int n); // 尾插法
21 LinkList HeadInsert(LinkList *L,int n); // 头插法
22 void ListReverse(LinkList *L); // 翻转链表

```

2.3.1.2 单链表初始化

```

1 // 初始化
2 Status InitList(LinkList *L)
3 {
4     *L = (LNode *)malloc(sizeof(LNode));
5     if (*L == NULL)
6     {
7         printf("内存不足, 分配失败!\n");
8         return OVERFLOW;
9     }
10    (*L) ->next = NULL;
11    return OK;
12 }

```



2.3.1.3 单链表查找

(1). 按位置查找

```
● ● ●
1 // 按位查找
2 LNode * GetElem(LinkList *L, int i)
3 {
4     if (i < 1)
5     {
6         printf("查找位置不合法!\n");
7         return NULL;
8     }
9     LNode *p = *L;
10    int j = 0;
11    while (p != NULL && j < i)
12    {
13        p = p->next;
14        j++;
15    }
16    return p;
17 }
```

(2). 按值查找

```
● ● ●
1 // 按值查找
2 LNode * LocateElem(LinkList L, int e)
3 {
4     LNode *p = L->next;
5     while(p != NULL && p->data != e)
6     {
7         p = p->next;
8     }
9     return p;
10 }
```

2.3.1.4 单链表插入

(1). 指定元素后插入

```
● ● ●
1 //指定元素后插入
2 Status InsertNextNode(LNode *p, int e)
3 {
4     LNode *s = (LNode *)malloc(sizeof(LNode));
5     if (s == NULL)
6     {
7         printf("内存不足，分配失败!\n");
8         return OVERFLOW;
9     }
10    if (p == NULL)
11        return ERROR;
12    s->data = e;
13    s->next = p->next;
14    p->next = s;
15    return OK;
16 }
```



(2). 指定元素前插入

```
1 //指定元素前插入
2 Status InsertPreNode(LNode *p, int e)
3 {
4     LNode *s = (LNode *)malloc(sizeof(LNode));
5     if (s == NULL)
6     {
7         printf("内存不足，分配失败!\n");
8         return OVERFLOW;
9     }
10    if (p == NULL)
11        return ERROR;
12    s->next = p->next;
13    p->next = s;
14    s->data = p->data;
15    p->data = e;
16    return OK;
17 }
```

(3). 按位置插入

```
1 // 按位置插入
2 Status ListInsert(LinkList *L, int i, int e)
3 {
4     if (i < 1)
5     {
6         printf("插入位置不合法!\n");
7         return ERROR;
8     }
9     LNode *p = (LNode *)malloc(sizeof(LNode));
10    if (p == NULL)
11    {
12        printf("内存不足，分配失败!\n");
13        return OVERFLOW;
14    }
15    if (i == 1)
16    {
17        p->data = e;
18        p->next = (*L)->next;
19        (*L)->next = p;
20        return OK;
21    }
22    p = GetElem(L, i-1);
23    if (p == NULL)
24        return ERROR;
25    return InsertNextNode(p, e);
26 }
27 }
```



2.3.1.5 单链表删除

```

● ● ●
1 // 删除
2 Status ListDelete(LinkList *L, int i, int* e)
3 {
4     if (i < 1)
5     {
6         printf("删除位置不合法!\n");
7         return ERROR;
8     }
9     if (i == 1)
10    {
11         if ((*L)->next == NULL)
12             return ERROR;
13         *e = (*L)->next->data;
14         (*L)->next = (*L)->next->next;
15         return OK;
16     }
17     LNode *p = GetElem(L,i-1);
18     if (p == NULL || p->next == NULL)
19         return ERROR;
20     LNode *q = p->next;
21     *e = q->data;
22     p->next = q->next;
23     free(q);
24     return OK;
25 }

```

2.3.1.6 单链表辅助函数

(1). *Length*

```

● ● ●
1 // 表长
2 int Length(LinkList L)
3 {
4     int len = 0;
5     LNode *p = L;
6     while(p->next != NULL)
7     {
8         p = p->next;
9         len++;
10    }
11    return len;
12 }

```

(2). *Empty*

```

● ● ●
1 // 判空
2 Status Empty(LinkList L)
3 {
4     if (L->next == NULL)
5         return OK;
6     return ERROR;
7 }

```



(3). PrintList

```
1 // 打印
2 void PrintList(LinkList L)
3 {
4     LNode *p = L->next;
5     while(p != NULL)
6     {
7         PrintNode(*p);
8         p = p->next;
9     }
10    printf("\n");
11 }
12
13 // 打印单个数据元素
14 void PrintNode(LNode p)
15 {
16     printf("%d ", p.data);
17 }
```

2.3.1.7 头插法 & 尾插法

(1). 头插法

```
1 // 头插法实现单链表
2 LinkList HeadInsert(LinkList *L,int n)
3 {
4     int e;
5     for (int i = 0; i < n; i++)
6     {
7         scanf("%d ", &e);
8         LNode *p = (LNode*)malloc(sizeof(LNode));
9         if (p == NULL)
10         {
11             printf("内存不足，分配失败!\n");
12             return NULL;
13         }
14         p->data = e;
15         p->next = (*L)->next;
16         (*L)->next = p;
17     }
18     return *L;
19 }
```

(2). 尾插法



```

1 // 尾插法实现单链表
2 LinkList TailInsert(LinkList *L,int n)
3 {
4     int e;
5     LNode *r = *L;
6     for (int i = 0; i < n; i++)
7     {
8         scanf("%d ", &e);
9         LNode* p = (LNode*)malloc(sizeof(LNode));
10        if (p == NULL)
11        {
12            printf("内存不足，分配失败!\n");
13            return NULL;
14        }
15        p->data = e;
16        p->next = r->next;
17        r->next = p;
18        r = p;
19    }
20    return *L;
21 }
22 }
```

2.3.1.8 单链表反转

```

1 // 反转单链表
2 void ListReverse(LinkList *L)
3 {
4     LNode *p,*q;
5     p = (*L)->next;
6     (*L)->next = NULL;
7     while(p != NULL)
8     {
9         q = p->next;
10        p->next = (*L)->next;
11        (*L)->next = p;
12        p = q;
13    }
14 }
```

2.3.2 双链表

定义 2.3.2(双链表)

1. 双链表有两个指针域, 分别指向直接前驱和直接后继
2. 头插法和尾插法的区别, 头插法实现反置
3. 在插入和删除时, 第一个和最后一个节点有特殊情况



2.3.2.1 双链表定义和函数声明

```
● ● ●
1 // 定义链表节点
2 typedef struct DNode{
3     int data;
4     struct DNode *prior;
5     struct DNode *next;
6 }DNode;
7 typedef DNode* DLinkedList;
8 // 函数声明
9 Status InitList(DLinkedList *L); // 初始化
10 Status Empty(DLinkedList L); // 判空
11 DNode * LocateElem(DLinkedList L, int e); // 按值查找
12 DNode * GetElem(DLinkedList *L, int i); // 按位查找
13 Status InsertNextNode(DNode *p, int e); // 指定元素后插入
14 Status InsertPreNode(DNode *p, int e); // 指定元素前插入
15 Status ListInsert(DLinkedList *L, int i, int e); // 按位置插入
16 Status ListDelete(DLinkedList *L, int i, int* e); // 删除
17 Status DeleteNode(DNode *p); // 删除指定节点
18 int Length(DLinkedList L); // 表长
19 void PrintList(DLinkedList L); // 打印链表
20 void PrintNode(DNode p); // 打印节点
21 DLinkedList TailInsert(DLinkedList *L, int n); // 尾插法
22 DLinkedList HeadInsert(DLinkedList *L, int n); // 头插法
23 void ListReverse(DLinkedList *L); // 翻转链表
```

2.3.2.2 双链表初始化

```
● ● ●
1 // 初始化
2 Status InitList(DLinkedList *L)
3 {
4     (*L) = (DNode *)malloc(sizeof(DNode));
5     if ((*L) == NULL)
6     {
7         printf("内存不足，分配失败!\n");
8         return OVERFLOW;
9     }
10    (*L)->prior = NULL;
11    (*L)->next = NULL;
12    return OK;
13 }
```

2.3.2.3 双链表查找

(1). 按位置查找

```

1 // 按位查找
2 DNode * GetElem(DLinkList *L, int i)
3 {
4     DNode * p = (*L);
5     int j = 0;
6     while (p->next != NULL && j < i)
7     {
8         p = p->next;
9         j++;
10    }
11    return p;
12 }

```

(2). 按值查找

```

1 // 按值查找
2 DNode * LocateElem(DLinkList L, int e)
3 {
4     DNode *p = L->next;
5     while (p != NULL)
6     {
7         if (p->data == e)
8             return p;
9         p = p->next;
10    }
11    return p;
12 }

```

2.3.2.4 双链表插入

(1). 指定元素后插入

```

1 // 指定元素后插入
2 Status InsertNextNode(DNode *p, int e)
3 {
4     DNode *q = (DNode*)malloc(sizeof(DNode));
5     if (q == NULL)
6     {
7         printf("内存不足，分配失败!\n");
8         return OVERFLOW;
9     }
10    if (p == NULL)
11        return ERROR;
12    q->data = e;
13    p->next->prior = q;
14    q->next = p->next;
15    q->prior = p;
16    p->next = q;
17    return OK;
18 }

```

(2). 指定元素前插入



```
1 //指定元素前插入
2 Status InsertPreNode(DNode *p, int e)
3 {
4     DNode *q = (DNode*)malloc(sizeof(DNode));
5     if (q == NULL)
6     {
7         printf("内存不足，分配失败!\n");
8         return OVERFLOW;
9     }
10    if (p == NULL)
11        return ERROR;
12    q->data = e;
13    p->prior->next = q;
14    q->prior = p->prior;
15    q->next = p;
16    p->prior = q;
17    return OK;
18 }
```

(3). 指定位置插入

```
1 // 按位置插入
2 Status ListInsert(DLinkList *L, int i, int e)
3 {
4     if (i < 1)
5     {
6         printf("插入位置不合法!\n");
7         return ERROR;
8     }
9     DNode *p = (DNode*)malloc(sizeof(DNode));
10    if (p == NULL)
11    {
12        printf("内存不足，分配失败!\n");
13        return OVERFLOW;
14    }
15    if (i == 1)
16    {
17        p->data = e;
18        p->next = (*L)->next;
19        (*L)->next->prior = p;
20        (*L)->next = p;
21        p->prior = (*L);
22        return OK;
23    }
24    p = GetElem(L,i-1);
25    return InsertNextNode(p,e);
26 }
27 }
```

2.3.2.5 双链表删除

```

● ● ●
1 // 删除
2 Status ListDelete(DLinkList *L, int i, int* e)
3 {
4     if (i < 1)
5     {
6         printf("删除位置不合法!\n");
7         return ERROR;
8     }
9     DNode *p = (*L)->next;
10    if (i == 1)
11    {
12        if (p == NULL)
13            return ERROR;
14        (*L)->next = p->next;
15        *e = p->data;
16        if (p->next != NULL)
17            p->next->prior = (*L);
18        free(p);
19        return OK;
20    }
21    p = GetElem(L, i);
22    if (p == NULL)
23        return ERROR;
24    *e = p->data;
25    return DeleteNode(p);
26 }

```

2.3.2.6 双链表辅助函数

(1). *Length*

```

● ● ●
1 // 表长
2 int Length(DLinkList L)
3 {
4     int len = 0;
5     while (L->next != NULL)
6     {
7         len++;
8         L = L->next;
9     }
10    return len;
11 }

```

(2). *Empty*

```

● ● ●
1 // 判空
2 Status Empty(DLinkList L)
3 {
4     if (L->next == NULL)
5         return OK;
6     return ERROR;
7 }

```



(3). PrintList

```
1 // 打印链表
2 void PrintList(DLinkList L)
3 {
4     DNode *p = L->next;
5     while(p != NULL)
6     {
7         PrintNode(*p);
8         p = p->next;
9     }
10    printf("\n");
11 }
12
13 // 打印节点
14 void PrintNode(DNode p)
15 {
16     printf("%d ", p.data);
17 }
```

2.3.2.7 头插法 & 尾插法

(1). 头插法

```
1 // 头插法
2 DLinkList HeadInsert(DLinkList *L, int n)
3 {
4     for (int i = 0; i < n; i++)
5     {
6         int e;
7         scanf("%d ", &e);
8         DNode *p = (DNode*)malloc(sizeof(DNode));
9         if (p == NULL)
10         {
11             printf("内存不足，分配失败!\n");
12             return NULL;
13         }
14         p->data = e;
15         p->next = (*L)->next;
16         (*L)->next = p;
17         p->prior = (*L);
18         if ((*L)->next != NULL)
19             (*L)->next->prior = p;
20     }
21     return *L;
22 }
```

(2). 尾插法



```

1 // 尾插法
2 DLinkList TailInsert(DLinkList *L, int n)
3 {
4     DNode *r = *L;
5     for (int i = 0; i < n; i++)
6     {
7         int e;
8         scanf("%d ", &e);
9         DNode *p = (DNode*)malloc(sizeof(DNode));
10        if (p == NULL)
11        {
12            printf("内存不足，分配失败!\n");
13            return NULL;
14        }
15        p->data = e;
16        p->next = r->next;
17        r->next = p;
18        p->prior = r;
19        r = p;
20    }
21    return *L;
22 }
```

2.3.2.8 双链表反转

```

1 //翻转链表
2 void ListReverse(DLinkList *L)
3 {
4     DNode *p,*q;
5     p = (*L)->next;
6     (*L)->next = NULL;
7     while(p != NULL)
8     {
9         q = p->next;
10        p->next = (*L)->next;
11        p->prior = (*L);
12        if ((*L)->next != NULL)
13            (*L)->next->prior = p;
14        (*L)->next = p;
15        p = q;
16    }
17 }
```

2.3.3 循环链表

定义 2.3.3 (循环链表)

1. 分为循环单链表和循环双链表，与单链表和双链表的区别在于尾节点指向头节点
2. 判空操作与单链表、双链表存在差异
3. 插入、删除、以及判断表头、表尾节点有一定变化



2.3.4 静态链表

定义 2.3.4 (静态链表)

1. 使用数组的链表，数组第一个元素当做头节点，每个位置除了保存数据元素外，还保存下一个元素的位置
2. 初始化时将所有位置的下一个元素位置清空
3. 插入、删除操作需要找到上一个节点的位置，然后将下一个元素位置清空



2.4 Summary

2.4.1 顺序表与链表比较

表 2.1: 顺序表与链表比较

存储结构		顺序表
比较项目		
空间	存储空间	① 预先分配 ② 会导致空间闲置或溢出
	存储密度	① 存储密度 = 1 ② 不需要额外空间来表示节点的逻辑关系
时间	存取元素	① 随机存取 ② 按位置访问元素时间复杂度 $O(1)$
	插入、删除	平均移动一半的元素, 时间复杂度 $O(n)$
适用情况		① 表长变化不大, 确定变化范围 ② 很少进行插入或删除, 经常按照元素位置序号访问数据

存储结构		链表
比较项目		
空间	存储空间	① 动态分配 ② 不会出现空间闲置或溢出
	存储密度	① 存储密度 < 1 ② 需要借助指针来表示节点的逻辑关系
时间	存取元素	① 顺序存取 ② 按位置访问元素时间复杂度 $O(n)$
	插入、删除	不需要移动元素, 确定插入或删除位置后, 时间复杂度 $O(n)$
适用情况		① 表长变化很大 ② 频繁进行插入或删除操作



2.4.2 单链表、双链表、循环链表比较

表 2.2: 单链表、双链表、循环链表比较

操作名称 链表名称	查找 表头节点	查找 表尾节点	查找节点 $*p$ 前驱节点
带头节点 单链表 L	L -> next 时间复杂度 $O(1)$	从 L -> next 依次向后遍历 时间复杂度 $O(n)$	无法找到前驱节点
带头节点头指针 L 循环单链表	L -> next 时间复杂度 $O(1)$	从 L -> next 依次向后遍历 时间复杂度 $O(n)$	p -> next 可以找到前驱节点 时间复杂度 $O(n)$
带头节点尾指针 R 循环单链表	R -> next 时间复杂度 $O(1)$	R 时间复杂度 $O(1)$	p -> next 可以找到前驱节点 时间复杂度 $O(n)$
带头节点 双向循环链表 L	L -> next 时间复杂度 $O(1)$	L -> prior 时间复杂度 $O(1)$	p -> prior 可以找到前驱节点 时间复杂度 $O(1)$





第3章 栈和队列

3.1 栈

定义 3.1.1 (栈)

1. 只允许在一端进行插入和删除操作的线性表
2. 栈有栈顶、栈底两个重要元素, 栈顶元素是最后一个插入的元素, 栈底元素是最先插入的元素
3. 栈的插入操作叫做进栈, 栈的删除操作叫做出栈
4. 栈的特点是后进先出, 简称 **LIFO, Last In First Out**
5. n 个不同元素进栈, 出栈元素不同的排列个数: $\text{Catalan}(n) = \frac{1}{n+1} \binom{n}{2n}$

定义 3.1.2 (栈基本操作)

1. **InitStack(&S)** 初始化栈 S
2. **DestroyStack(&S)** 销毁栈 S
3. **GetTop(S, &e)** 获取栈顶元素, 将栈 S 的栈顶元素赋值给 e
4. **Push(&S, e)** 压栈
5. **Pop(&S, &e)** 出栈
6. **Length(S)** 求栈中元素个数
7. **Empty(S)** 判空
8. **PrintStack(S)** 输出操作, 输出栈 S 的所有元素



3.1.1 栈定义和函数声明

3.1.1.1 顺序栈、链栈、共享栈定义

```
● ● ●
1 # define MAXSIZE 10
2 # define SHARESIZE 20
3 // 顺序栈定义
4 typedef struct SqStack{
5     int data[MAXSIZE];
6     int top;
7 }SqStack;
8 // 链栈定义
9 typedef struct LNode{
10    int data;
11    struct LNode* next;
12 }LNode;
13 typedef LNode* Stack;
14 // 共享栈定义
15 typedef struct ShareStack{
16     int data[SHARESIZE];
17     int Ttop;
18     int Dtop;
19 }ShareStack;
```

3.1.1.2 函数声明

```
● ● ●
1 // 函数声明
2 Status InitStack(Stack *S); // 初始化栈
3 Status Empty(Stack S); // 判空
4 Status Push(Stack *S, int e); // 压栈
5 Status Pop(Stack *S, int *e); // 出栈
6 Status GetTop(Stack S, int * e); // 获取栈顶元素
7 int Length(Stack S); // 栈中元素个数
8 void PrintStack(Stack S); // 打印栈中元素
```

3.1.2 顺序栈

定义 3.1.3 (顺序栈)

顺序栈是用顺序表实现的栈，使用 **top** 指针指定栈顶元素在顺序表中的位置，栈的容量大小固定，需要判空和判满。



3.1.2.1 顺序栈初始化

```
● ● ●
1 // 初始化
2 Status InitSqStack(SqStack *S)
3 {
4     if (S == NULL)
5         return ERROR;
6     S->top = -1;
7     return OK;
8 }
```

3.1.2.2 顺序栈出入栈

```
● ● ●
1 // 压栈
2 Status SqPush(SqStack *S, int e)
3 {
4
5     if (S->top == MAXSIZE - 1)
6     {
7         printf("顺序栈已满, 压栈失败!\n");
8         return ERROR;
9     }
10    S->top++;
11    S->data[S->top] = e;
12    return OK;
13 }
14
15 // 出栈
16 Status SqPop(SqStack *S, int *e)
17 {
18     if (S->top == -1)
19     {
20         printf("顺序栈已空, 出栈失败!\n");
21         return ERROR;
22     }
23     *e = S->data[S->top];
24     S->top--;
25     return OK;
26 }
```

3.1.2.3 顺序栈辅助函数

(1). 判断栈是否为空

```
● ● ●
1 // 判空
2 Status SqEmpty(SqStack S)
3 {
4     if (S.top == -1)
5         return OK;
6     return ERROR;
7 }
```



(2). 获取栈顶元素

```
1 // 获取栈顶元素
2 Status SqGetTop(SqStack S, int *e)
3 {
4     if (S.top == -1)
5     {
6         printf("顺序栈已空, 获取栈顶元素失败!\n");
7         return ERROR;
8     }
9     *e = S.data[S.top];
10    return OK;
11 }
```

(3). 获取栈中元素个数

```
1 // 栈中元素个数
2 int SqLength(SqStack S)
3 {
4     return S.top + 1;
5 }
```

(4). 打印栈中元素

```
1 // 打印栈中元素
2 void PrintSqStack(SqStack S)
3 {
4     for (int i = S.top; i>=0; i--)
5     {
6         printf("%d ", S.data[i]);
7     }
8     printf("\n");
9 }
```

3.1.3 链栈

定义 3.1.4 (链栈)

链栈是使用链表实现的栈, 不需要判满, 只需要判空, 出栈和入栈操作都在链表的头部进行, 只需要表头指针, 一般使用不带头节点的单链表实现.



3.1.3.1 链栈初始化

```
● ● ●  
1 // 初始化栈  
2 Status InitStack(Stack *S)  
3 {  
4     (*S) = NULL;  
5     return OK;  
6 }
```

3.1.3.2 链栈出入栈

```
● ● ●  
1 // 压栈  
2 Status Push(Stack *S, int e)  
3 {  
4     LNode *p = (LNode*)malloc(sizeof(LNode));  
5     if (p == NULL)  
6     {  
7         printf("内存不足，分配失败!\n");  
8         return OVERFLOW;  
9     }  
10    p->data = e;  
11    if ((*S) == NULL)  
12    {  
13        p->next = NULL;  
14        (*S) = p;  
15        return OK;  
16    }  
17    p->next = (*S);  
18    (*S) = p;  
19    return OK;  
20 }  
21  
22 // 出栈  
23 Status Pop(Stack *S, int *e)  
24 {  
25     if ((*S) == NULL)  
26     {  
27         printf("链栈已空，出栈失败!\n");  
28         return ERROR;  
29     }  
30     *e = (*S)->data;  
31     (*S) = (*S)->next;  
32     return OK;  
33 }
```

3.1.3.3 链栈辅助函数

(1). 判断栈是否为空



```
● ● ●  
1 // 判空  
2 Status Empty(Stack S)  
3 {  
4     if (S == NULL)  
5         return OK;  
6     return ERROR;  
7 }
```

(2). 获取栈顶元素

```
● ● ●  
1 // 获取栈顶元素  
2 Status GetTop(Stack S, int * e)  
3 {  
4     if (S == NULL)  
5     {  
6         printf("链栈已空, 获取栈顶元素失败!\n");  
7         return ERROR;  
8     }  
9     *e = S->data;  
10    return OK;  
11 }
```

(3). 获取栈中元素个数

```
● ● ●  
1 // 栈中元素个数  
2 int Length(Stack S)  
3 {  
4     int len = 0;  
5     while(S != NULL)  
6     {  
7         len++;  
8         S = S->next;  
9     }  
10    return len;  
11 }
```

(4). 打印栈中元素

```
● ● ●  
1 // 打印栈中元素  
2 void PrintStack(Stack S)  
3 {  
4     while(S != NULL)  
5     {  
6         printf("%d ", S->data);  
7         S = S->next;  
8     }  
9     printf("\n");  
10 }
```



3.1.4 共享栈

定义 3.1.5 (共享栈)

共享栈是两个栈共享一个顺序表，两个栈的栈底分别在顺序表的两端，两个栈的栈顶指针分别向中间移动，当两个栈的栈顶指针相遇时，表示栈满；共享栈主要是为了充分利用空间，提高空间利用率。



3.1.4.1 共享栈初始化

```
● ● ●
1 // 初始化
2 Status InitShareStack(ShareStack *S)
3 {
4     if (S == NULL)
5         return ERROR;
6     S->Ttop = -1;
7     S->Dtop = SHARESIZE;
8     return OK;
9 }
```

3.1.4.2 共享栈出入栈

(1). 入栈

```
● ● ●
1 // T 压栈
2 Status ShareTPush(ShareStack *S, int e)
3 {
4
5     if (S->Ttop + 1 == S->Dtop)
6     {
7         printf("共享栈已满, T 压栈失败!\n");
8         return ERROR;
9     }
10    S->Ttop++;
11    S->data[S->Ttop] = e;
12    return OK;
13 }
14
15 // D 压栈
16 Status ShareDPush(ShareStack *S, int e)
17 {
18
19     if (S->Ttop + 1 == S->Dtop)
20     {
21         printf("共享栈已满, D 压栈失败!\n");
22         return ERROR;
23     }
24     S->Dtop--;
25     S->data[S->Dtop] = e;
26     return OK;
27 }
```



(2). 出栈

```
1 // T 出栈
2 Status ShareTPop(ShareStack *S, int *e)
3 {
4     if (S->Ttop == -1)
5     {
6         printf("T 栈已空, 出栈失败!\n");
7         return ERROR;
8     }
9     *e = S->data[S->Ttop];
10    S->Ttop--;
11    return OK;
12 }
13
14 // D 出栈
15 Status ShareDPop(ShareStack *S, int *e)
16 {
17     if (S->Dtop == SHARESIZE)
18     {
19         printf("D 栈已空, 出栈失败!\n");
20         return ERROR;
21     }
22     *e = S->data[S->Dtop];
23     S->Dtop++;
24     return OK;
25 }
```

3.1.4.3 共享栈辅助函数

(1). 判断栈是否为空

```
1 // 判空
2 Status ShareEmpty(ShareStack S)
3 {
4     if (S.Ttop == -1 && S.Dtop == SHARESIZE)
5         return OK;
6     return ERROR;
7 }
```

(2). 获取栈顶元素



```
● ● ●
1 // 获取 T 栈顶元素
2 Status ShareTGetTop(ShareStack S, int *e)
3 {
4     if (S.Ttop == -1)
5     {
6         printf("T 栈已空, 获取栈顶元素失败!\n");
7         return ERROR;
8     }
9     *e = S.data[S.Ttop];
10    return OK;
11 }
12
13 // 获取 D 栈顶元素
14 Status ShareDGetTop(ShareStack S, int *e)
15 {
16     if (S.Dtop == SHARESIZE)
17     {
18         printf("D 栈已空, 获取栈顶元素失败!\n");
19         return ERROR;
20     }
21     *e = S.data[S.Dtop];
22     return OK;
23 }
```

(3). 获取栈中元素个数

```
● ● ●
1 // T 栈中元素个数
2 int LengthT(ShareStack S)
3 {
4     return S.Ttop + 1;
5 }
6
7 // D 栈中元素个数
8 int LengthD(ShareStack S)
9 {
10    return SHARESIZE - S.Dtop;
11 }
```

(4). 打印栈中元素



```

1 // 打印 T 栈中元素
2 void PrintTStack(ShareStack S)
3 {
4     for (int i = S.Ttop; i>=0; i--)
5     {
6         printf("%d ",S.data[i]);
7     }
8     printf("\n");
9 }
10
11 // 打印 D 栈中元素
12 void PrintDStack(ShareStack S)
13 {
14     for (int i = S.Dtop; i<=SHARESIZE-1; i++)
15     {
16         printf("%d ",S.data[i]);
17     }
18     printf("\n");
19 }

```

3.2 队列

定义 3.2.1 (队列)

- 只允许在一端进行插入操作, 在另一端进行删除操作的线性表
- 队列有队头、队尾两个重要元素, 队头元素是最先插入的元素, 队尾元素是最后插入的元素
- 队列的插入操作叫做入队, 队列的删除操作叫做出队
- 队列的特点是先进先出, 简称 **FIFO, First In First Out**



定义 3.2.2 (特殊队列)

- 循环队列: 顺序表实现的队列, 解决顺序表的假溢出问题
- 双端队列: 两端都可以进行插入和删除操作的队列
- 操作受限的双端队列: 两端都可以进行插入和删除操作, 但是有限制, 比如只有一端可以插入或者删除



定义 3.2.3 (队列基本操作)

- InitQueue(&Q)** 初始化队列 Q
- DestroyQueue(&Q)** 销毁队列 Q
- GetHead(Q, &e)** 获取队首元素, 将队列 Q 队首元素赋值给 e
- EnQueue(&S, e)** 入队
- DeQueue(&S, &e)** 出队
- Length(Q)** 求队列中元素个数
- Empty(Q)** 判空
- PrintQueue(S)** 输出操作, 输出队列 Q 的所有元素



3.2.1 队列定义和函数声明

3.2.1.1 循环队列、链表队列定义

```

1 // 循环队列定义
2 typedef struct SqQueue{
3     int data[MAXSIZE];
4     int front,rear;
5 }SqQueue;
6
7 // 链表队列定义
8 typedef struct LNode{
9     int data;
10    struct LNode *next;
11 }LNode;
12 typedef struct Queue{
13     int size;
14     LNode *front, *rear;
15 }Queue;

```

3.2.1.2 函数声明

```

1 // 函数声明
2 Status InitQueue(Queue *Q); // 初始化队列
3 Status EmptyQueue(Queue *Q); // 判空
4 Status EnQueue(Queue *Q, int e); // 入队
5 Status DeQueue(Queue *Q, int* e); // 出队
6 Status GetHead(Queue Q,int* e); // 获取队头元素
7 int LengthQueue(Queue Q); // 队列长队
8 void PrintQueue(Queue Q); // 打印队列

```

3.2.2 循环队列

定义 3.2.4 (循环队列)

循环队列是使用顺序表实现的队列,为防止出现假溢出,将顺序表的首尾相连,当队尾指针指向顺序表的最后一个位置时,再插入元素时,将队尾指针指向顺序表的第一个位置.

实现方法:

- 队列最后一个位置不存储元素,队列中元素个数最多为 $MAXSIZE - 1$,队列满的条件是 $(Q.front + 1) \% MAXSIZE == Q.rear$,队列空的条件是 $Q.front == Q.rear$
- 设置一个标志位 **flag**,如果 $flag = 0$,表示上一次是入队操作,如果 $flag = 1$,表示上一次是出队操作,当 $Q.front == Q.rear$ 时,根据 $flag$ 的值判断是队列满还是队列空,如果 $flag = 0$,队列满,如果 $flag = 1$,队列空



3.2.2.1 循环队列初始化

```
● ○ ● ●
1 // 初始化队列
2
3 Status InitSqQueue(SqQueue *Q)
4 {
5     if (Q == NULL)
6         return ERROR;
7     Q->front = 0;
8     Q->rear = 0;
9     return OK;
10 }
```

3.2.2.2 循环队列出入队

(1). 入队

```
● ○ ● ●
1 //入队
2
3 Status EnSqQueue(SqQueue *Q, int e)
4 {
5     if ((Q->rear+1)%MAXSIZE == Q->front)
6     {
7         printf("循环队列已满，入队失败!\n");
8         return ERROR;
9     }
10    Q->data[Q->rear] = e;
11    Q->rear = (Q->rear+1)%MAXSIZE;
12    return OK;
13 }
```

(2). 出队

```
● ○ ● ●
1 //出队
2
3 Status DeSqQueue(SqQueue *Q, int* e)
4 {
5     if (EmptySqQueue(Q)){
6         printf("循环队列已空，出队失败!\n");
7         return ERROR;
8     }
9     *e = Q->data[Q->front];
10    Q->front = (Q->front+1)%MAXSIZE;
11    return OK;
12 }
```

3.2.2.3 循环队列辅助函数

(1). 获取队首元素



```
1 //获取队头元素
2
3 Status SqGetHead(SqQueue *Q, int* e)
4 {
5     if (EmptySqQueue(Q))
6     {
7         printf("循环队列已空, 获取队头元素失败!\n");
8         return ERROR;
9     }
10    *e = Q->data[Q->font];
11    return OK;
12 }
```

(2). 队列长度

```
1 //队列长队
2
3 int LengthSqQueue(SqQueue Q)
4 {
5     return (Q.rear-Q.font+MAXSIZE)%MAXSIZE;
6 }
```

(3). 判断队列是否为空

```
1 // 判空
2
3 Status EmptySqQueue(SqQueue *Q)
4 {
5     if (Q->font == Q->rear)
6         return OK;
7     return ERROR;
8 }
```

(4). 打印队列

```
1 // 打印队列
2 void PrintSqQueue(SqQueue Q)
3 {
4     if (Q.font == Q.rear)
5         return;
6     for (int i = Q.font; i!=Q.rear; i=(i+1)%MAXSIZE)
7     {
8         printf("%d ",Q.data[i]);
9     }
10    printf("\n");
11 }
```



3.2.3 链队列

定义 3.2.5 (链队列)

链队列是使用链表实现的队列，只需要存储队头节点和队尾节点的指针。

实现方法：

1. 队列入队，将新节点作为队尾节点的后继节点，队尾节点变为新节点
2. 队列出队，将队头节点作为出队节点，队头节点变为其后继节点，释放队头节点的空间



3.2.3.1 链队列初始化

```
1 // 初始化队列
2
3 Status InitQueue(Queue *Q)
4 {
5     (*Q).font = NULL;
6     (*Q).rear = NULL;
7     (*Q).size = 0;
8     return OK;
9 }
```

3.2.3.2 链队列出入队

(1). 入队

```
1 //入队
2
3 Status EnQueue(Queue *Q, int e)
4 {
5     LNode* p = (LNode*)malloc(sizeof(LNode));
6     if (p == NULL)
7     {
8         printf("内存不足，分配失败!\n");
9         return OVERFLOW;
10    }
11    p->data = e;
12    p->next = NULL;
13    if ((*Q).font == NULL)
14    {
15        (*Q).font = p;
16        (*Q).rear = p;
17        (*Q).size++;
18        return OK;
19    }
20    (*Q).rear->next = p;
21    (*Q).rear = p;
22    Q->size++;
23    return OK;
24 }
```



(2). 出队

```
● ● ●  
1 // 出队  
2 Status DeQueue(Queue *Q, int* e)  
3 {  
4     if ((*Q).font == NULL)  
5     {  
6         printf("链表队列已空, 出队失败!\n");  
7         return ERROR;  
8     }  
9     *e = (*Q).font->data;  
10    LNode *p = (*Q).font;  
11    if (p->next == NULL)  
12    {  
13        (*Q).font = NULL;  
14        (*Q).rear = NULL;  
15        free(p);  
16    }  
17    (*Q).font = p->next;  
18    (*Q).size--;  
19    return OK;  
20 }
```

3.2.3.3 链队列辅助函数

(1). 获取队首元素

```
● ● ●  
1 // 获取队头元素  
2  
3 Status GetHead(Queue Q, int* e)  
4 {  
5     if (Q.font == NULL)  
6     {  
7         printf("链表队列已空, 获取队头元素失败!\n");  
8         return ERROR;  
9     }  
10    *e = Q.font->data;  
11    return OK;  
12 }
```

(2). 队列长度

```
● ● ●  
1 // 队列长队  
2  
3 int LengthQueue(Queue Q)  
4 {  
5     return Q.size;  
6 }
```

(3). 判断队列是否为空



```

1 // 判空
2
3 Status EmptyQueue(Queue *Q)
4 {
5     if ((*Q).font == NULL)
6         return OK;
7     return ERROR;
8 }

```

(4). 打印队列

```

1 // 打印队列
2
3 void PrintQueue(Queue Q)
4 {
5     LNode *p = Q.font;
6     while(p != NULL)
7     {
8         printf("%d ", p->data);
9         p = p->next;
10    }
11    printf("\n");
12 }

```

3.2.4 双端队列

3.3 栈和队列的应用

3.3.1 括号匹配

定理 3.3.1 (算法步骤)

1. 遇到左括号压栈, 遇到右括号出栈
2. 如果出栈为空, 或者出栈元素和当前右括号不匹配, 返回错误
3. 如果遍历完所有括号, 栈为空, 返回正确



3.3.2 前缀、中缀、后缀表达式求值

定理 3.3.2 (表达式求值)

一、前缀表达式: 运算符在前, 操作数在后

1. 从右向左扫描表达式, 遇到操作数入栈, 遇到运算符, 从栈中弹出两个操作数, 进行运算
2. 第一个弹出的操作数是右操作数, 第二个弹出的操作数是左操作数, 运算结果入栈
3. 最后栈中的元素就是表达式的值



二、中缀表达式：运算符在中间，操作数在两边，直接计算

三、后缀表达式：运算符在后，操作数在前

1. 从左向右扫描表达式，遇到操作数入栈，遇到运算符，从栈中弹出两个操作数，进行运算
2. 第一个弹出的操作数是左操作数，第二个弹出的操作数是右操作数，运算结果入栈
3. 最后栈中的元素就是表达式的值



3.3.3 表达式转换

定理 3.3.3 (表达式转换：手算)

一、中缀表达式 → 前缀表达式：按照运算顺序，从左向右扫描，把操作符放在左边，操作数放在右边，将得到的前缀表达式作为操作数继续进行操作，直到得到最终的前缀表达式

二、中缀表达式 → 后缀表达式：按照运算顺序，从左向右扫描，把操作符放在右边，操作数放在左边，将得到的后缀表达式作为操作数继续进行操作，直到得到最终的后缀表达式

三、前缀表达式 → 后缀表达式和后缀表达式 → 前缀表达式：先将前缀表达式转换为中缀表达式，再将中缀表达式转换为后缀表达式；同理可以将后缀表达式转换为前缀表达式



3.3.4 队列应用

定义 3.3.1 (队列应用)

1. 树的层次遍历，按照根节点入队，出队，再将左右孩子入队，出队，直到队列为空
2. 图的广度优先搜索，按照顶点入队，出队，再将与该顶点相邻的顶点入队，出队，直到队列为空
3. 模拟操作系统的进程调度，先到的进程先执行，后到的进程后执行 (FCFS)
4. 图的深度有限搜索，按照顶点入栈，出栈，再将与该顶点相邻的顶点入栈，出栈，直到栈为空



3.4 数组

定义 3.4.1 (数组)



3.4.1 特殊矩阵的压缩

