

第三章 栈和队列

3.1 栈

- 3.1.1 栈的基本概念
- 3.1.2 栈的存储结构
- 3.1.3 栈的操作算法
- 3.1.4 栈的应用

3.2 队列

- 3.2.1 队列的基本概念
- 3.2.2 队列的存储结构
- 3.2.3 队列的应用
- 3.2.4 队列应用

- 栈和队列也是线性表，只不过是两种**操作受限**的线性表。
- 如果从数据类型角度看，他们是和线性表大不相同的两类重要的抽象数据类型。
- 栈和队列广泛应用于各种软件系统中。

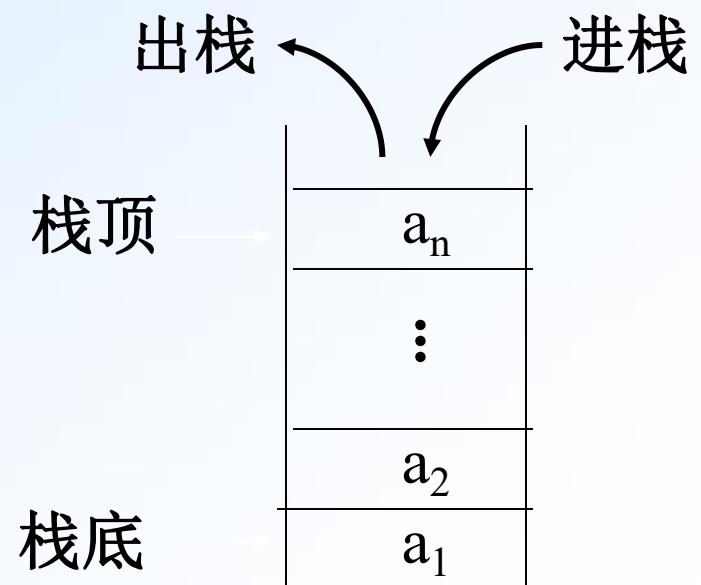
3.1 栈的基本概念

- **栈** -- 是限制仅在线性表的一端进行插入和删除运算的线性表。

- **栈顶 (top)** -- 允许插入和删除的一端。
- **栈底 (bottom)** -- 不允许插入和删除的一端。
- **空栈** -- 表中没有元素。

- 栈的特点：

后进先出 (*LIFO*)



【例3-1】 假定有4个元素A, B, C, D, 按所列次序进栈, 试写出所有可能的出栈序列。注意, 每一个元素进栈后都允许出栈, 如ACDB就是一种出栈序列。

解: 可能的出栈序列有ABCD, ABDC, ACBD, ACDB, ADCB, BACD, BADC, BCAD, BCDA, BDCA, CBAD, CBDA, CDBA, DCBA。

可能的出栈序列个数可用Catalan列计算, 即 $\frac{1}{n+1} C_{2n}^n$

栈的基本运算

1. **初始化** — 创建一个空栈;
2. **判栈空** — 判断栈是否为空栈;
3. **进栈** — 往栈中插入（或称推入）一个元素;
4. **退栈** — 从栈中删除（或称弹出）一个元素;
5. **取栈顶元素**

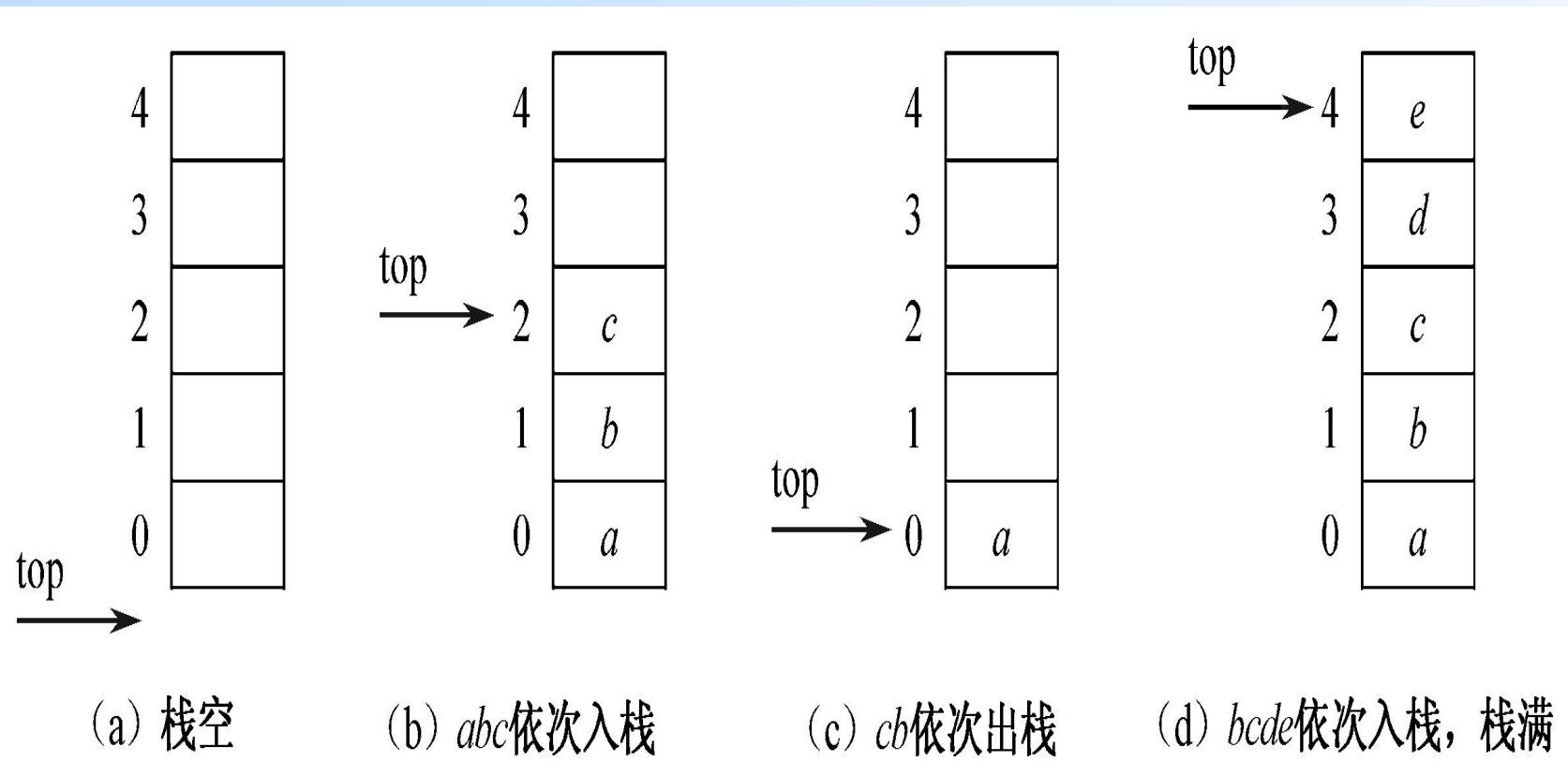
栈的存储结构

- 栈的顺序存储结构（顺序栈）
- 栈的链式存储结构（链栈）

顺序栈

- 栈的顺序存储结构简称为**顺序栈**。可用数组来实现顺序栈。
- 因为栈底位置是固定不变的，所以可以将栈底位置设置在数组的两端的任何一个端点；
- 栈顶位置是随着进栈和退栈操作而变化的，故需用一个整型变量top表示栈顶位置。

栈操作的示意图



顺序栈的类模板定义：

```
template <class T, int MaxSize >
class SeqStack{
    T data[MaxSize];      //存放栈元素的数组
    int top;               //栈顶指针
public:
    SeqStack( );          //构造函数
    void Push(T x);       //入栈
    T Pop();               //出栈
    T Top();               //取栈顶元素
    bool Empty();          //判断栈是否为空
};
```

顺序栈的基本操作的实现

1、构造函数

```
template <class T,int MaxSize>
SeqStack<T,MaxSize>::SeqStack( )
{
    top=-1;
}

}
```

2、入栈操作

```
template <class T,int MaxSize>
void SeqStack<T,MaxSize>::Push( )
{
    if (top== MaxSize-1)
        {cerr<<"上溢"; exit(1);}
    top++;
    data[top]=x;
}
```

3、退栈操作

```
template <class T,int MaxSize>
T SeqStack<T,MaxSize>::Pop( )
{
    if (top== -1)
        { cerr<<"下溢"; exit(1); }
    x = data[top];
    top--;
    return x;
}
```

4、取栈顶元素

```
template <class T,int MaxSize>
T SeqStack<T,MaxSize>::Top()
{
    if (top==-1)
        { cerr<<"下溢"; exit(1); }
    return data[top];
}
```

5、判栈空操作

```
template <class T,int MaxSize>
bool SeqStack<T,MaxSize>::Empty( )
{
    return top== -1;
}
```

6、栈遍历

```
template <class T,int MaxSize>
void SeqStack<T,MaxSize>:: StackTraverse()
{
    for( i=0; i<=top; i++)
        cout<<data[i];
}
```

● 链 栈

- 栈的链式存储结构称为**链栈**。

- 链栈的设置---

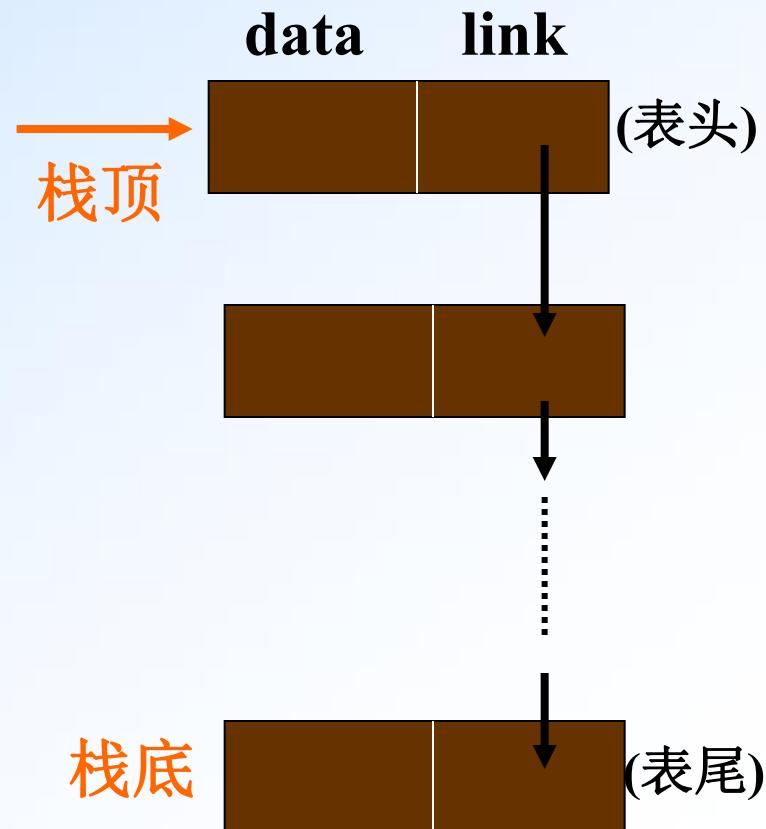
(1) 栈顶位置放在哪里合适?

栈顶指向表头位置

进栈/退栈仅在表头进行

(2) 是否需要附加头结点?

不需要!



链栈的类模板定义：

```
template <class T, int MaxSize >
```

```
class LinkStack{
```

```
    Node<T> *top;      //栈顶指针
```

```
public:
```

```
    LinkStack( );        //构造函数
```

```
    ~LinkStack( );       //析构函数
```

```
    void Push(T x);     //入栈
```

```
    T Pop();            //出栈
```

```
    T Top();             //取栈顶元素
```

```
    bool Empty();        //判断链栈是否为空栈
```

```
};
```

链栈的基本操作的实现

1、构造函数

```
template <class T,int MaxSize>
LinkStack<T, MaxSize>::LinkStack( )
{
    top=NULL;
}
```

2、入栈操作

```
template <class T,int MaxSize>
void LinkStack<T, MaxSize>::Push(T x )
{
    s=new Node<T>;
    s->data = x;
    s->next = top;
    top=s;
}
```

3、退栈操作

```
template <class T, int MaxSize>
T LinkStack<T, MaxSize>::Pop()
{
    if (top==NULL) {cerr<<"下溢"; exit(1);}
    x=top->data;
    p=top;
    top=top->next;
    delete p;
    return x;
}
```

4、取栈顶元素

```
template <class T,int MaxSize>
T LinkStack<T, MaxSize>::Top()
{
    if (top==NULL)
        {cerr<<"下溢"; exit(1);}
    return top->data;
}
```

5、判栈空操作

```
template <class T,int MaxSize>
bool LinkStack<T, MaxSize>::Empty()
{
    return top==NULL;
}
```

6、栈遍历

```
template <class T, int MaxSize>
void LinkStack<T, MaxSize>::StackTraverse( )
{
    for( p=top; p; p=p->next)
        cout<<p->data;
}
```

栈的应用举例

例1：括号匹配问题：

假设一个算术表达式中允许包含两种括号：圆括号与方括号，其嵌套的次序随意。请设计一个算法判断一个算术表达式中的括号是否匹配。

问题：

1. 括号匹配过程如何组织？
2. 在括号匹配过程中如何应用栈？

```
void check() {  
    SeqStack<int,100> S;  
    char ch[80], *p, e;  
    printf("请输入表达式\n");  
    gets(ch);  
    p = ch;  
    while( *p ) // 没到串尾  
        switch(*p) {  
            case '(':  
            case '[':  
                S.Push(*p++);  
                break;      // 左括号入栈，且p++
```

```
case ')':  
case ']': if( !S.Empty() ) {  
    e=S.Pop();           // 弹出栈顶元素  
    if(*p=='') && e!='(' || *p==']' && e!= '['){  
        printf("左右括号不配对\n");  
        exit(1);  
    }  
    else{  
        p++;  
        break; // 跳出switch语句  
    }  
} // Empty  
else { // 栈空  
    printf("缺乏左括号\n");  
    exit(1);  
}
```

```
    default: p++;      // 其它字符不处理，指针向后移  
}  
  
if(S.Empty()) // 字符串结束时栈空  
    printf("括号匹配\n");  
  
else  
    printf("缺乏右括号\n");  
}
```

- 例2：表达式求值问题：

输入包含+、-、*、/、圆括号和正整数组成的中缀算术表达式，以“@”作为表达式结束符。设计一个算法，对给定的中缀算术表达式进行求值。

2 + 4 * 6 - 8 / (5 - 3) + 7 @

- 假定采用“**算符优先法**”对中缀表达式进行求值。
 - 一个表达式是由操作数、运算符、界限符组成。
 - 任何两个相继出现的算符之间的优先级关系为>、=、<。
 - 算符之间的优先级关系完全决定了操作数的运算次序。

算符间的优先关系

θ_1

θ_2

	+	-	\times	/	()	@
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
\times	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(<	<	<	<	<	=	
)	>	>	>	>		>	>
@	<	<	<	<	<		=

说明:

- 左括号 ‘(’ 的优先级最高
- 当 $\theta_1 = \theta_2$ 时，令
 $\theta_1 > \theta_2$ (左结合性)
- 为了算法处理的方便，在表达式的最左边添加一个 ‘@’。
- 优先关系相等的情况只有
‘(’ = ‘)’ 和 ‘@’ = ‘@’

• 算术表达式求值示例： 8 / (5 - 3) @

- 基于算符优先法，如何组织和实现表达式求值过程？
- 如何利用栈的后进先出特性？

- **数据结构的设计**
- 为实现算符优先算法，需引入两个栈：
 - **OPTR**: 用于保存运算符
 - **OPND**, 用于保存操作数

表达式求值算法的设计：

- 1、首先将操作数栈OPND置为空栈，将表达式起始符“@”压入运算符栈OPTR中作为栈底元素。
- 2、依次读入表达式的每个字符，直至当前字符与运算符栈的栈顶元素均为“@”时，结束下列循环：
 - 若是操作数，则进操作数栈OPND；
 - 若是运算符，则进行以下判断：
 - 若当前运算符**高于**OPTR栈顶运算符，将当前运算符入栈OPTR，继续读入下一字符；
 - 若当前运算符**低于**OPTR栈顶运算符，栈顶运算符退栈，并弹出操作数栈的两个栈顶操作数进行运算，再将运算结果压入操作数栈OPND中。
 - 若当前运算符**等于**栈顶运算符，表明栈顶运算符为“（”，当前运算符为“）”，则栈顶运算符退栈。

算术表达式求值过程示例： **$8 / (5 - 3) @$**

```
int EvaluateExpression() {
    SeqStack<char,100> OPTR;
    SeqStack<int,100> OPND;
    OPTR.Push('@');
    c = getchar();
    while( c!='@' || OPTR.Top()!='@' ){
        if( InOPTR(c, OP) )      // c是运算符
            processing...
        else if(c>='0'&&c<='9') { // c是操作数
            OPND.Push(c); c=getchar();
        }
        else { // c是非法字符
            printf("ERROR\n"); exit(1);
        }
    }
    return OPND.Top();
}
```

```
x=OPTR.Top();
switch( Precede(x, c) ){
    case‘<’: // 栈顶元素优先权低
        OPTR.Push(c); c=getchar();
        break;
    case‘=’: //脱括号并接收下一字符
        x=OPTR.Pop(); c=getchar();
        break;
    case‘>’: // 退栈并将运算结果入栈
        theta=OPTR.Pop();
        b=OPND.Pop();
        a=OPND.Pop();
        OPND.Push(Operate(a,theta,b));
        break;
}
```

后缀表达式

- 后缀表达式（逆波兰式）的概念
 - 后缀是指把操作符放在两个操作数的后面
 - $3 \ 5 / \ 6 + \quad (3/5+6)$
 - $16 \ 9 \ 4 \ 3 + * - \quad (16-9*(4+3))$
 - $25 \ x \ a \ a \ b \ b + * \quad (25+x)*(a*(a+b)+b)$
- 后缀表达式的特点?
 - 计算过程完全按照运算符出现的先后次序进行。
 - 在后缀表达式中，不存在括号，也不存在优先级的差别。

后缀表达式求值的实现方法

- 对于给定的后缀表达式，如何对其进行求值？
 - 8 5 3 - /
 - 是否需要引入**操作数栈OPND**和**运算符栈OPTR**？
- 由于对后缀表达式的计算完全按照运算符出现的先后次序进行，而且每次运算只是对该运算符之前的两个操作数执行，因此，可以在一个**操作数栈**的帮助下执行计算过程。

例3：将中缀表达式转换为后缀表达式

$$2*(x+y)/(1-x) \rightarrow 2\ x\ y\ +\ *\ 1\ x\ -\ /$$

- 将中缀表达式变成等价的后缀表达式方法：
 - 表达式中的操作数次序不变
 - 运算符出现次序根据计算次序进行调整
 - 去掉圆括号。
- 如何实现中缀到后缀的转换？
 - 基本过程类似于中缀表达式的求值

从中缀表达式到后缀式的转换算法

- 设立一个运算符栈**OPTR**
- 依次读入表达式中的每个字符，对于不同类型的字符按不同情况进行处理：
 - ① 若读到的是操作数，则输出该操作数，并读入下一个字符。
 - ② 若读到的是左括号，则把它压入到**OPTR**栈中，并读入下一个字符。
 - ③ 若读到的是右括号，则将**OPTR**栈从栈顶直到左括号之前的运算符依次出栈并输出，然后将左括号也出栈，并读入下一个字符。
 - ④ 若读到的是运算符（c），则应与运算符栈的栈顶元素（**pre_op**）进行比较：
若**pre_op**<**c**，则将**c**入栈，并读入下一个字符；
若**pre_op**>=**c**，则将**pre_op**出栈并输出。
- 按照以上过程扫描到中缀表达式结束符@时，把栈中剩余的运算符依次出栈并输出，就得到了转换成的后缀表达式。

栈与递归

- 函数调用与栈的关系?
 - 在高级语言编制的程序中，调用函数和被调函数之间的链接和信息交换需通过栈来进行。
- 在一个函数中运行被调函数之前，系统需完成三件事：
 - ① 将所有的实参、返回地址等信息传递给被调函数；
 - ② 为被调函数的局部变量分配存储区；
 - ③ 将控制转移到被调函数的入口。
- 从被调函数返回调用函数时，系统也应完成三件事：
 - ① 保存被调函数的计算结果；
 - ② 释放被调函数的数据区；
 - ③ 依据被调函数保存的地址将控制转移到调用函数。

栈与递归

- 函数调用与栈的关系?
 - 当有多个函数构成嵌套调用时，按照“**后调用先返回**”的原则，函数之间的信息传递与控制转移必须通过“**栈**”来实现；
 - 即系统将整个程序运行时所需的数据空间安排在一个**栈**中，每当调用一个函数时，就为它在**栈顶**分配一个存储区。

栈与递归

- 递归的概念

$$\text{Fact}(n) = \begin{cases} 1 & n = 0 \\ n * \text{Fact}(n-1) & n > 0 \end{cases}$$

- 递归的执行过程在机器中如何实现?

- 对于函数的调用，系统是通过栈来实现的。
 - 在执行递归函数的过程中也需要一个“递归工作栈”。

- 递归工作栈的作用是：

- 1) 将递归调用时的实际参数和函数返回地址传递给下一层的递归函数。
- 2) 保存本层的参数和局部变量，以便从下一层返回时重新使用它们。

3.2 队列 (Queue)

一. 基本概念

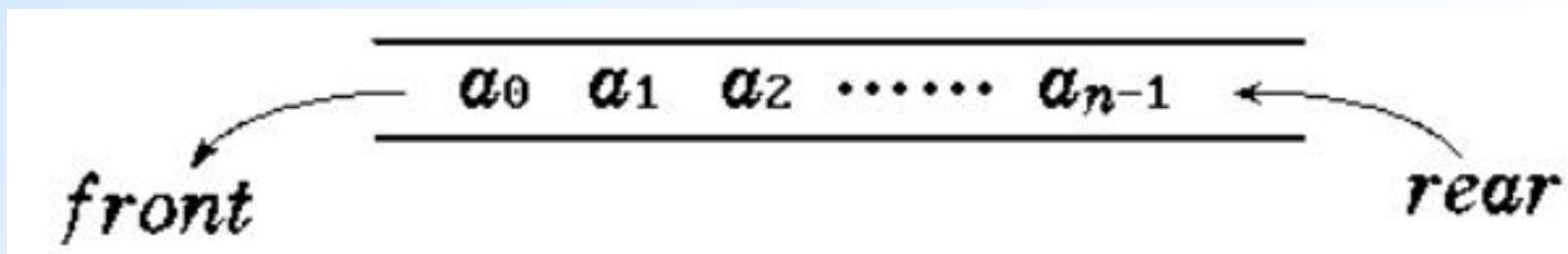
-队列是只允许在一端删除，在另一端插入的线性表。

-**队头(front)**: 指允许删除的一端。

-**队尾(rear)**: 允许插入的一端。

• 特性

-先进先出(FIFO, First In First Out)



☞队列的基本运算：

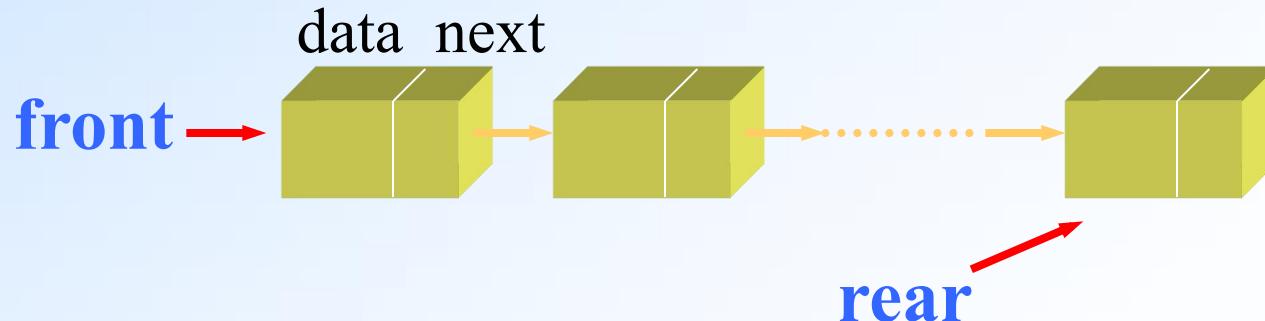
- 📖 初始化队列 (**InitQueue**)
- 📖 入队 (**EnQueue**)
- 📖 出队 (**DeQueue**)
- 📖 取队头元素 (**GetHead**)
- 📖 判队列是否为空 (**QueueEmpty**)
- 📖 置空队列 (**ClearQueue**)

队列的存储结构

- 队列的顺序存储结构（顺序队列）
- 队列的链式存储结构（链队列）

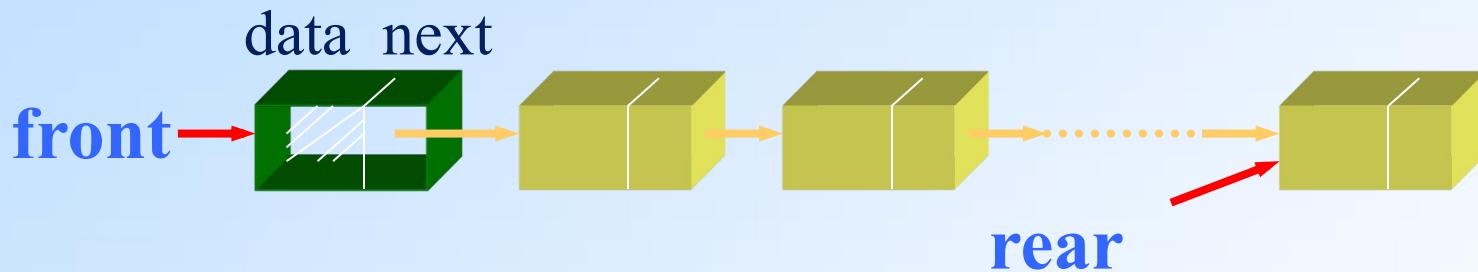
二. 链队列—队列的链式表示与实现

- 队列的链式存储结构简称为链队列，它是限制**仅在表头删除和表尾插入**的单链表。
- 显然仅有单链表的头指针不便于在表尾做插入操作，为此再增加一个尾指针，指向链表的最后一个结点。



● 链队列是否需要头结点？

- 为了操作的方便，在链队列中也添加一个头结点，并使队头指针指向头结点。



● 如何判断队满、队空？

- 链队列在进队时一般不考虑队满问题
- 队空条件为 $Q.front == Q.rear$

链队列的类定义

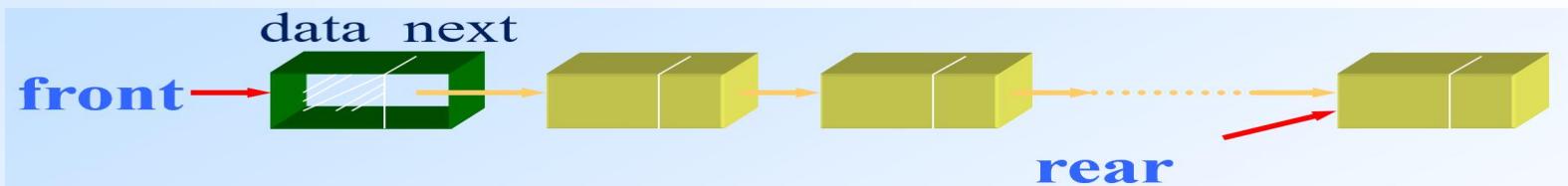
```
template <class T>
class LinkQueue{
    Node<T> *front, *rear;
public:
    LinkQueue();
    ~LinkQueue();
    void EnQueue(T x);
    T DeQueue();
    T GetQueue();
    int Length();
    bool Empty();
};
```

// 构造一个空队列Q

```
template <class T>
LinkQueue<T>::LinkQueue()
{
    s=new Node<T>;
    s->next=NULL;
    front=rear=s;
}
```

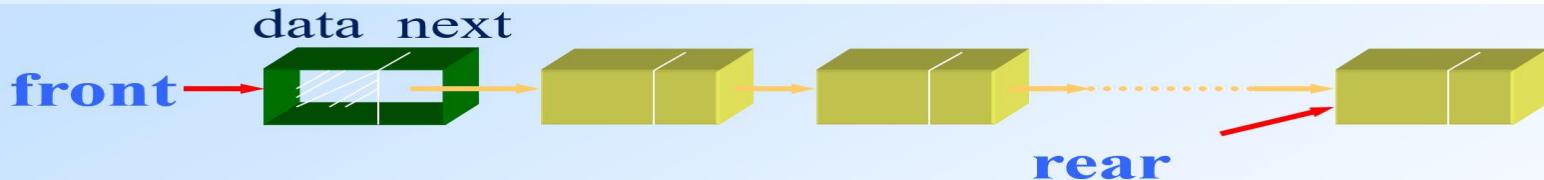
// 入队 — 插入元素e为Q的新的队尾元素

```
template <class T>
void LinkQueue<T>::EnQueue(T x)
{
    s=new Node<T>;
    s->data=x;
    s->next=NULL;
    rear->next=s;    //将结点s插入到队尾
    rear=s;           //将队尾指针指向结点s
}
```



//出队 — 若队列不空,删除Q的队头元素,用e返回其值,并返回OK,否则返回ERROR

```
template <class T>
T LinkQueue<T>::DeQueue()
{
    if (rear==front)  {cerr<<"下溢"; exit(1);}
    p=front->next;
    x=p->data;
    front->next=p->next;
    if (p->next==NULL) rear=front;
    delete p;
    return x;
}
```

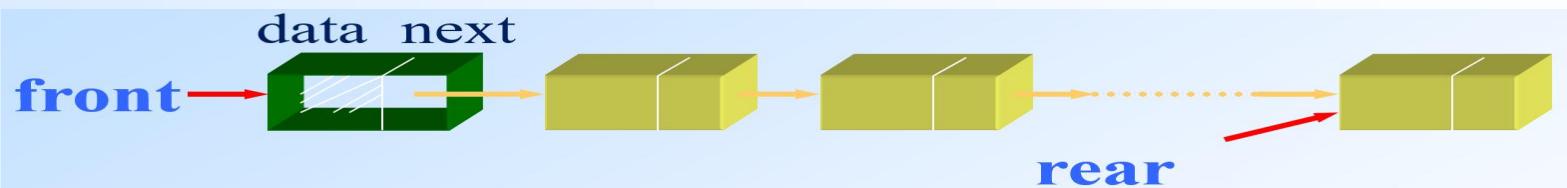


// 判队空

```
template <class T>
bool LinkQueue<T>::Empty()
{
    return front==rear;
}
```

// 析构函数

```
template <class T>
void LinkQueue<T>::~LinkQueue(T x)
{
    while( front )
    {
        rear=front->next;
        delete(front);
        front=rear;
    }
}
```



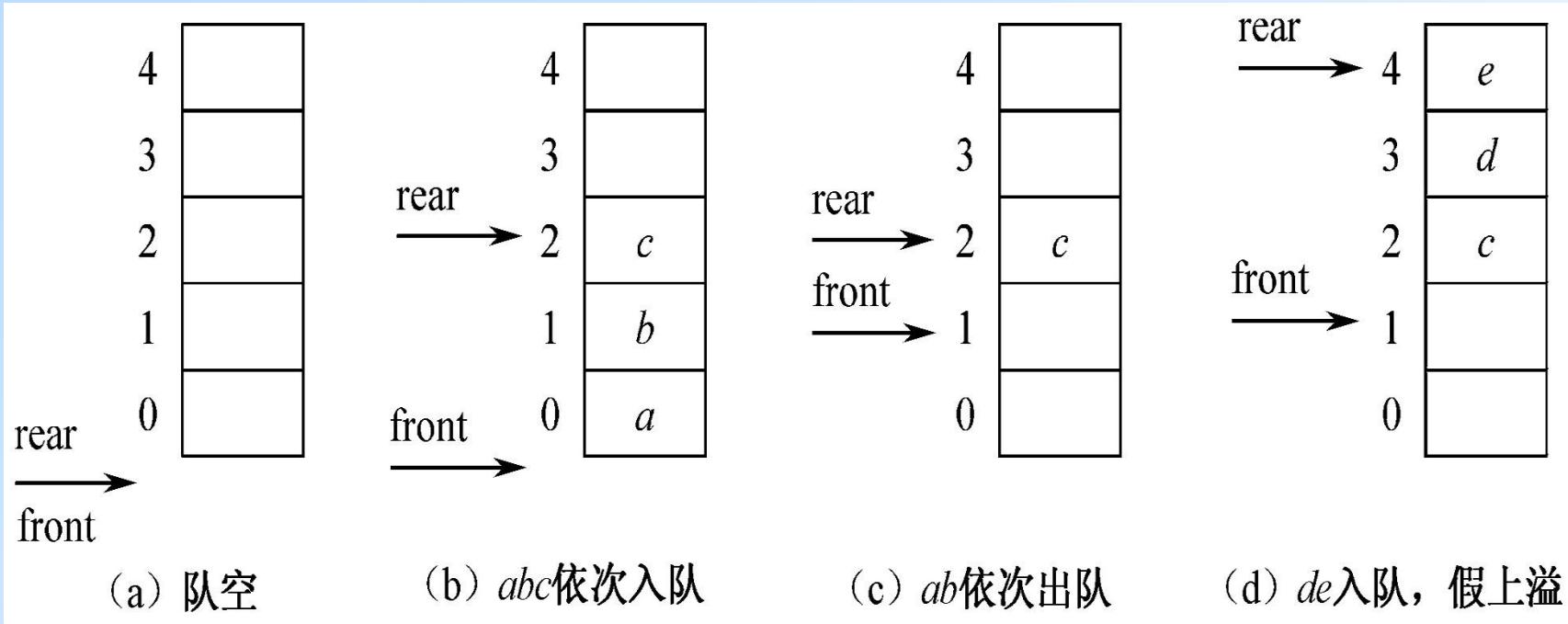
三. 循环队列——队列的顺序表示和实现

- 队列的顺序存储结构称为**顺序队列**，
顺序队列实际上是运算受限的顺序表。
- 顺序队列也是用一个数组空间来存放当前队列中的元素。由于队列的**队头**和**队尾**的位置是变化的，因而要设两个指针和分别指示队头和队尾元素在队列中的位置。

顺序队列的类定义：

```
template <class T, int MaxSize >
class SeqQueue
{
    T data[MaxSize];          //存放队列元素的数组
    int front, rear;          //队头和队尾指针
public:
    SeqQueue();               //构造函数，置空队
    void EnQueue(T x);        //将元素x入队
    T DeQueue();               //将队头元素出队
    T GetQueue();              //取队头元素（并不删除）
    bool Empty();               //判断队列是否为空
    int Length();                //求队列长度
};
```

队列操作的示意图

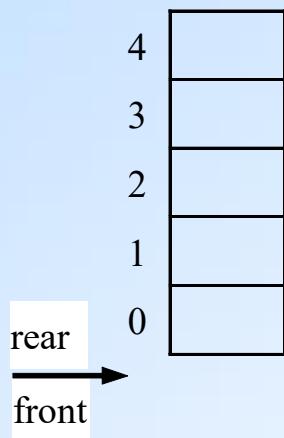


- 队头和队尾指针在队列初始化时均约定为： -1。
- 队尾指针指向**当前**队尾元素
队头指针指向当前队头元素的**上一个**位置

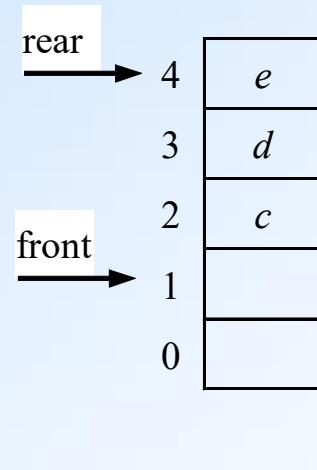
- 问题1：随着入队、出队操作的进行，队列中可能出现什么问题？
 - “假上溢” 现象

“假上溢” 现象的解决如何解决？

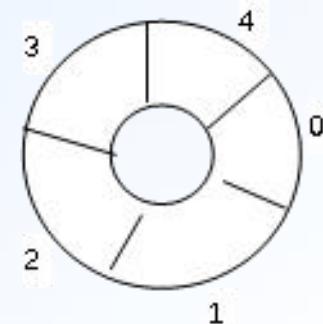
- 把队列看成是环状（循环）队列

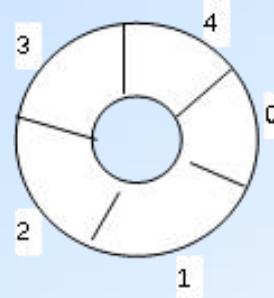
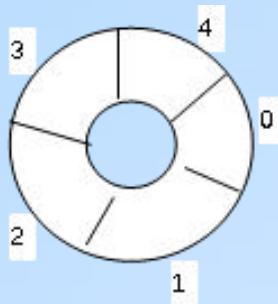


(a) 队空



(b) 假上溢





- 问题2：在循环队列中如何判别“**队满**”与“**队空**”状态？
 - 另外引入一个标志变量以区别“队满”与“队空”
 - 牺牲一个元素空间的方法。
 - 即入队前，测试尾指针在循环意义下加1后是否等于头指针，若相等，则认为是队满。

// 构造函数（初始化空队列）

```
template <class T,int MaxSize>
SeqQueue<T,MaxSize>:: SeqQueue ()
{
    front=rear=0;
}
```

// 入队运算

```
template <class T,int MaxSize>
void SeqQueue<T,MaxSize>::EnQueue(T x)
{
    if ((rear+1) % MaxSize == front)
    { cerr<<"上溢"; exit(1); }
    rear=(rear+1) % MaxSize;
    data[rear]=x;
}
```

// 出队运算

```
template <class T,int MaxSize>
T SeqQueue<T,MaxSize>::DeQueue( )
{
    if (rear==front)
    { cerr<<"下溢"; exit(1); }
    front=(front+1) % MaxSize;
    return data[front];
}
```

// 求队列长度运算

```
template <class T,int MaxSize>
int SeqQueue<T,MaxSize>::Length( )
{
    return (rear-front+ MaxSize) % MaxSize;
}
```

3.2.4 队列应用

- 例1：设有 n 个人排成一列，从前往后“0，1”连续报数。凡是报到“0”的人出列，凡是报到“1”的人立即站到队伍的最后。反复进行报数，直到所有人均出列为止。要求给出这 n 个人的出列顺序。

例如， $n=5$ 时，初始序列为1、2、3、4、5，出队序列为1、3、5、4、2。

问题分析

- 数据结构的设计？

- 个人排成的队伍很自然的用队列进行模拟。
 - 哪种队列存储结构合适？

- 链队列

- 算法设计？

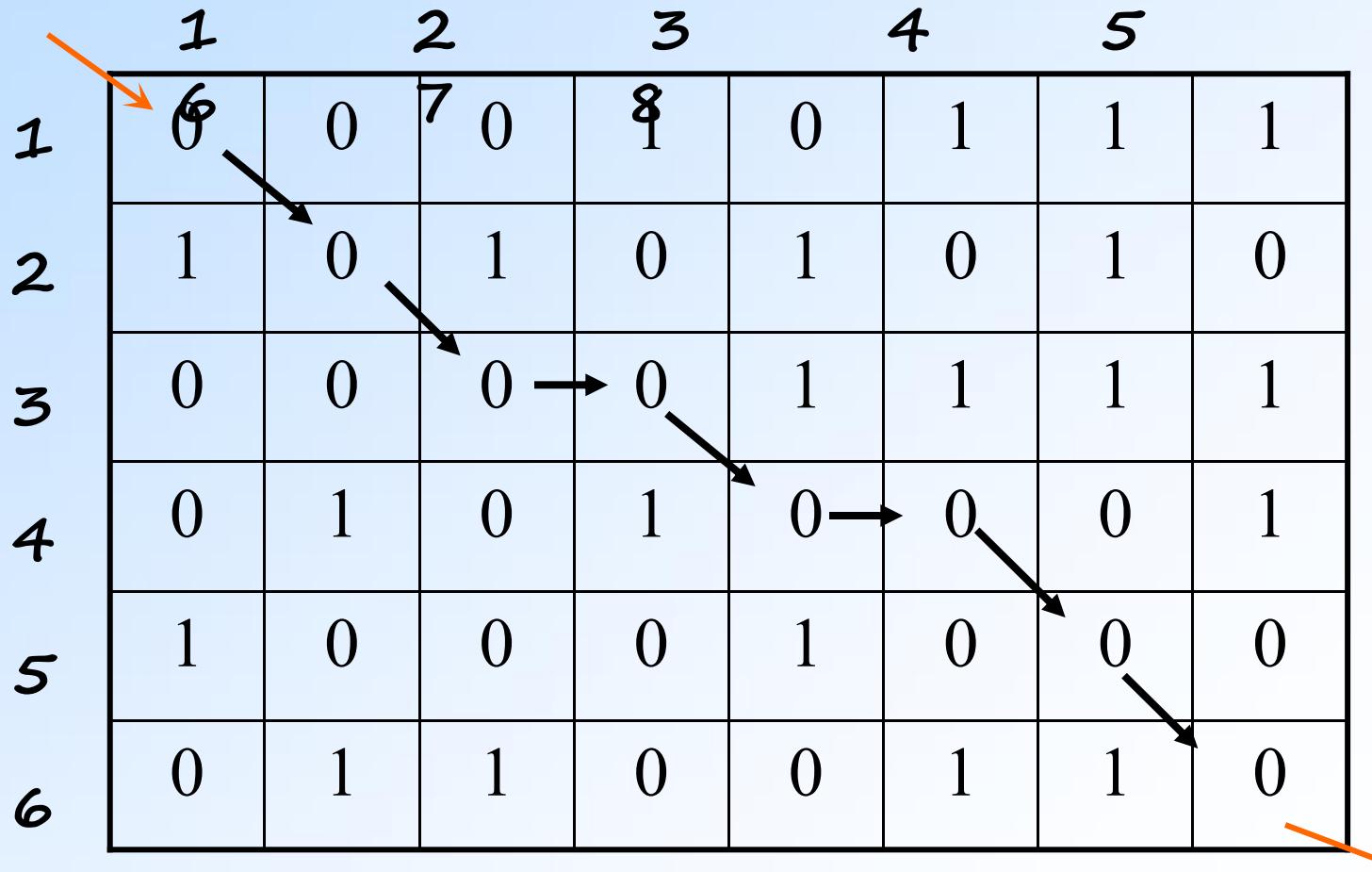
- 人员出列的过程就是一个反复进行出队和入队操作的问题。
 - 反复执行以下步骤，直至队列为空：
 - ① 将队头元素出队，并输出其编号。
 - ② 若队列不空，则再出队一个元素，并将该元素再次入队。

算法描述：

```
void Number(){
    LinkQueue<int> linkq;
    for(i=1;i<=n;i++) //初始化队列，让n个人入队
        linkq.Enqueue(i);
    while( !linkq.Empty() ){
        x=linkq.DeQueue(); //报到"0"的人出列
        cout<<x;
        if( !linkq.Empty() ) { //报到"1"的人到队伍最后
            y=linkq.DeQueue();
            linkq.Enqueue(y);
        }
    }
}
```

3.2.4 队列应用

- 例2：求迷宫的最短路径。



迷宫最短路径的搜索算法如何设计？

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

需要解决的问题1：如何从某一坐标点出发搜索其四周的邻点？

需要解决的问题2：如何组织搜索过程？

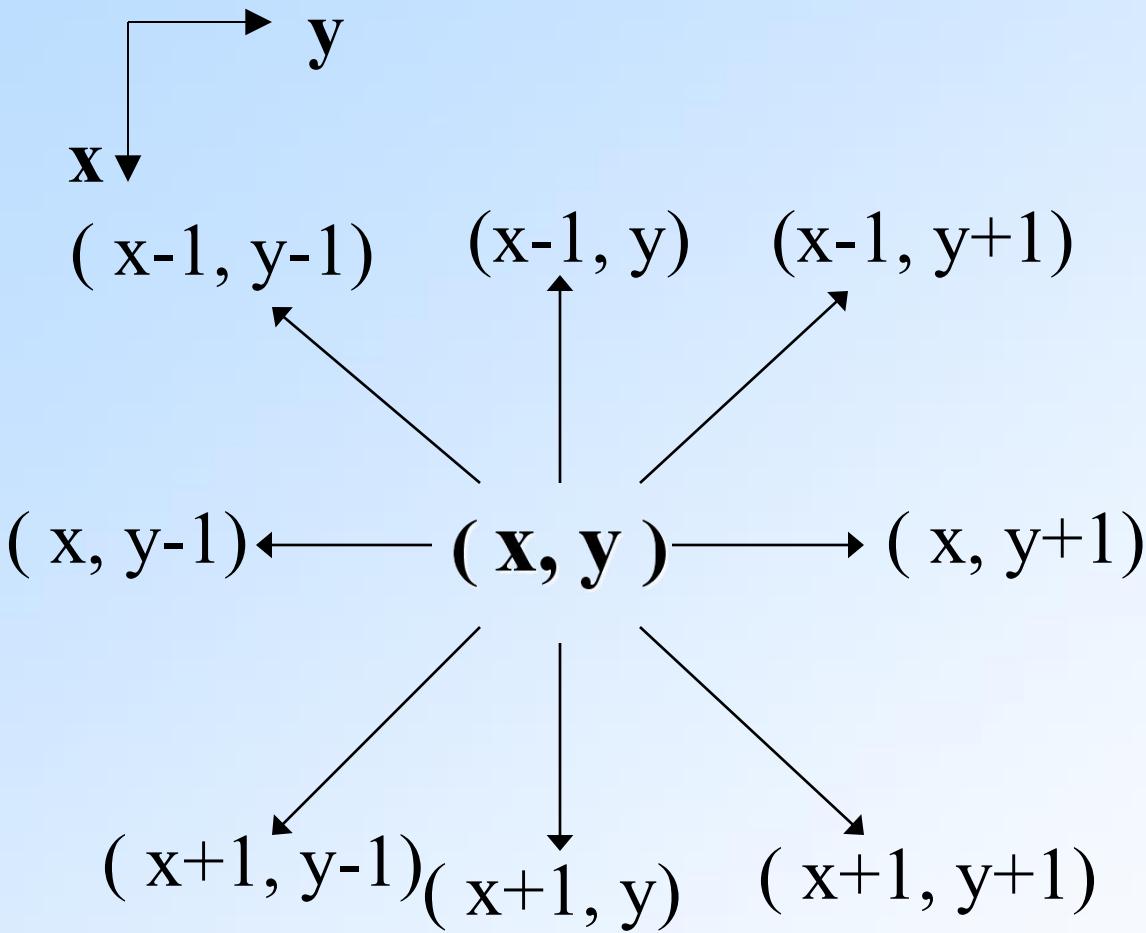
需要解决的问题3：如何防止重复到达某坐标点？

需要解决的问题4：如何输出搜索路径？

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

The diagram shows a 6x8 grid with coordinates ranging from (1,1) to (6,8). The grid contains binary values (0 or 1). A sequence of arrows indicates a search path starting at (1,1), moving right to (1,2), then down-right to (2,3), then right to (3,4), then down-right to (4,5), then right to (5,6), then down-right to (6,7), and finally right to (6,8). The path is highlighted by orange arrows.

需要解决的问题1：如何从某一坐标点出发搜索其四周的邻点？



	x	y
0	0	+1
1	+1	+1
2	+1	0
3	+1	-1
4	0	-1
5	-1	-1
6	-1	0
7	-1	+1

坐标增量数组move

需要解决的问题2：如何组织搜索过程？

	x	y	pre
1	1	1	0
2	2	2	1
3	3	3	2
4	3	1	2
5	3
6	3	1	3
...

需要解决的问题3：如何防止重复到达某坐标点？

需要解决的问题4：如何输出搜索路径？

	1	...	12	13	14	15	16	17	18	19	20	
x	1	...	5	2	5	6	5	6	6	5	6	...
y	1	...	6	6	3	1	7	5	4	8	8	...
pre	0	...	8	9	10	10	11	12	14	16	16	...



```
const int SIZE= 64  
const int M=10  
const int N=10  
int m=M-2, n=N-2;
```

```
struct SqType{  
    int x, y;  
    int pre;  
};  
SqType sq[SIZE];
```

```
struct moved{  
    int x, y;  
} move[8];  
  
int maze[M][N];
```

```
int ShortPath(int maze[][]N) {  
    int i, j, v, front, rear, x, y;  
    sq[1].x=1; sq[1].y=1; sq[1].pre=0;  
    front=1; rear=1;  
    maze[1][1]= -1;  
    while( front <= rear ){  
        x=sq[front].x;  y=sq[front].y;  
        for (v=0;v<8;v++) {  
            i = x+move[v].x;  
            j = y+move[v].y;  
            if ( maze[i][j]==0 ){  
                rear++;  
                sq[rear].x=i;  sq[rear].y=j;  
                sq[rear].pre=front;  
                maze[i][j]=-1;
```

```
if ( (i==m)&&(j==n ){  
    PrintPath(sq,rear);  
    Restore(maze);  
    return 1;  
}  
} //end if  
} //end for  
front++;  
} //end while  
return 0;  
}
```

```
void PrintPath(SqType sq[], int rear)
```

```
{
```

```
    int i;
```

```
    i = rear;
```

```
    do {
```

```
        printf(“\n(%d,%d)”,sq[i].x,sq[i].y);
```

```
        i=sq[i].pre;
```

```
    } while ( i != 0 );
```

```
}
```

本章小结

主要学习要点如下：

- (1) 理解栈和队列的特点及它们的差异。
- (2) 掌握顺序栈和链栈的定义及操作。
- (3) 掌握循环队列和链队列的定义及操作。
- (4) 掌握栈和队列的应用。