第3章 栈和队列

栈和队列是两种常用的数据结构，它们的数据元素的逻辑关系也是线性关系，但在运算上不同于线性表。

本章主要学习要点如下。

（1）栈、队列和线性表的异同，栈和队列抽象数据类型的描述方法。

（2）顺序栈的基本运算算法的设计方法。

（3）链栈的基本运算算法的设计方法。

（4）顺序队的基本运算算法的设计方法。

（5）链队的基本运算算法的设计方法。

（6）Python 中的双端队列（deque）和优先队列（heapq）及其应用。

（7）综合运用栈和队列解决一些复杂的实际问题。

3.1 栈

本节先介绍栈的定义，然后讨论栈的存储结构和基本运算算法设计，最后通过两个综合实例说明栈的应用。

扫一扫 视频讲解

3.1.1 栈的定义

先看一个示例，假设有一个田鼠洞，口径只能容纳一只田鼠，有若干只田鼠依次进洞，如图 3.1 所示，当到达洞底时，这些田鼠只能一只一只地按与原来进洞时相反的次序回退出洞，如图 3.2 所示。在这个例子中，田鼠洞就是一个栈，由于其口径只能容纳一只田鼠，所以不论洞中有多少只田鼠，它们只能是一只一只地排列，从而构成一种线性关系。再看看田鼠洞的主要操作，显然有进洞和出洞，进洞只能从洞口进，出洞也只能从洞口出。

抽象起来，栈是一种只能在同一端进行插入或删除操作的线性表。表中允许进行插入、删除操作的一端称为**栈顶**。栈顶的当前位置是动态的，可以用一个称为栈顶指针的位置指示器来指示。表的另一端称为**栈底**。当栈中没有数据元素时称为**空栈**。栈的插入操作通常称为**进栈**或**入栈**，栈的删除操作通常称为**退栈**或**出栈**。

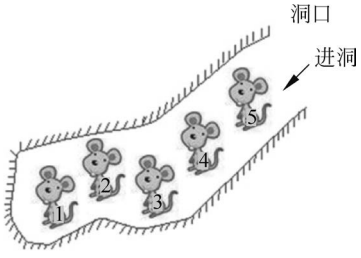


图 3.1 田鼠进洞的情况

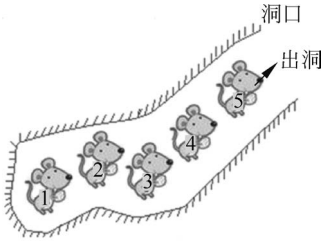


图 3.2 田鼠出洞的情况

说明：对于线性表，可以在中间和两端的任何地方插入和删除元素，而栈只能在同一端插入和删除元素。

栈的主要特点是“后进先出”或者“先进后出”，即后进栈的元素先出栈。每次进栈的元素都放在原当前栈顶元素之前成为新的栈顶元素，每次出栈的元素都是当前栈顶元素，栈顶元素出栈后次栈顶元素变成新的栈顶元素。栈也称为**后进先出表**。

抽象数据类型栈的定义如下：

ADT Stack {
　数据对象：
　　$D=\{a_i \mid 0\leqslant i\leqslant n-1, n\geqslant 0\}$
　数据关系：
　　$R=\{r\}$
　　$r=\{<a_i, a_{i+1}> \mid a_i, a_{i+1}\in D, i=0,\cdots,n-2\}$
　基本运算：
　　empty()：判断栈是否为空，若为空栈返回真，否则返回假。
　　push(e)：进栈操作，将元素 e 插入栈中作为栈顶元素。
　　pop()：出栈操作，返回栈顶元素。

```
        gettop(): 取栈顶操作,返回当前的栈顶元素。
}
```

【例 3.1】 若元素进栈的顺序为 1234，能否得到 3142 的出栈序列？

解 为了使 3 作为第一个出栈元素，应该让 1、2、3 依次进栈，再出栈 3，接着要么 2 出栈，要么 4 进栈后出栈，第 2 次出栈的元素不可能是 1，所以得不到 3142 的出栈序列。

【例 3.2】 用 S 表示进栈操作，X 表示出栈操作，若元素进栈的顺序为 1234，为了得到 1342 的出栈顺序，给出相应的 S 和 X 操作串。

解 为了得到 1342 的出栈顺序，其操作过程是 1 进栈，1 出栈，2 进栈，3 进栈，3 出栈，4 进栈，4 出栈，2 出栈，因此相应的 S 和 X 操作串为 SXSSXSXX。

扫一扫 视频讲解

【例 3.3】 设 n 个元素的进栈序列是 1，2，3，…，n，通过一个栈得到的出栈序列是 p_1，p_2，p_3，…，p_n，若 $p_1=n$，则 p_i（$2\leqslant i\leqslant n$）的值是什么？

解 当 $p_1=n$ 时，说明进栈序列的最后一个进栈的元素最先出栈，此时出栈序列只有一种，即 n，$n-1$，…，2，1，或 $p_1=n$，$p_2=n-1$，…，$p_{n-1}=2$，$p_n=1$，也就是说 $p_i+i=n+1$，推出 $p_i=n-i+1$。

扫一扫 视频讲解

3.1.2 栈的顺序存储结构及其基本运算算法的实现

由于栈中元素的逻辑关系与线性表的相同，因此可以借鉴线性表的两种存储结构来存储栈。

在采用顺序存储结构存储时，用列表 data 来存放栈中元素，称为**顺序栈**。顺序栈存储结构如图 3.3 所示，由于 Python 列表提供了一端动态扩展的功能，为此将 data[0]端作为栈底，另外一端 data[−1]作为栈顶，其中的元素个数 len(data)恰好为栈中实际的元素个数。

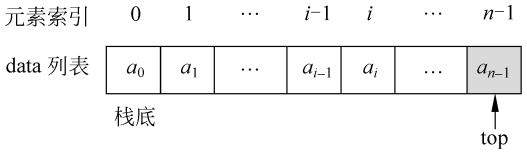


图 3.3 顺序栈的示意图

图 3.4 所示为一个栈的动态示意图，图 3.4(a)表示一个空栈，图 3.4(b)表示元素 a 进栈以后的状态，图 3.4(c)表示元素 b、c、d 进栈以后的状态，图 3.4(d)表示出栈元素 d 以后的状态。

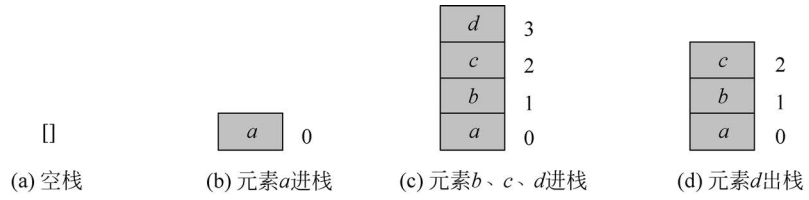


图 3.4 栈操作示意图

从中看到，顺序栈的四要素如下。

① 栈空条件：len(data)==0 或者 not data。

② 栈满条件：由于 data 列表可以动态扩展，所以不必考虑栈满。

③ 元素 e 进栈操作：将 e 添加到栈顶处。

④ 出栈操作：删除栈顶元素并返回该元素。

说明：顺序栈中的元素始终是向一端生长的，如果采用具有固定容量 capacity 的列表存放栈元素，需要增加一个指向栈顶元素的栈顶指针 top，这样栈中实际的元素个数恰好为 top+1，栈满条件改为 top==capacity-1。在顺序栈中既可以将 data[0]端作为栈底，也可以将 data[-1]端作为栈底，但不能将 data 列表的中间位置作为栈底。

顺序栈类 SqStack 设计如下(用 SqStack.py 文件存放)：

```
class SqStack:
    def __init__(self):                    #构造方法
        self.data=[]                       #存放栈中元素,初始为空
    #栈的基本运算算法
```

顺序栈的基本运算算法如下。

1) 判断栈是否为空：empty()

若 len(data)为 0 表示空栈。对应的算法如下：

```
def empty(self):                           #判断栈是否为空
    if len(self.data)==0:
        return True
    return False
```

2) 进栈：push(e)

元素进栈只能从栈顶进，不能从栈底或中间位置进栈，如图 3.5 所示。

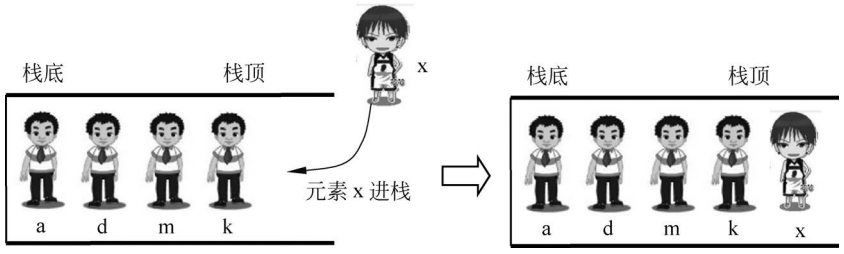


图 3.5　元素进栈示意图

元素 e 进栈可以直接利用 data 列表的 append()方法添加元素 e(列表的 append()方法的时间复杂度为 $O(1)$)。对应的算法如下：

```
def push(self,e):                          #元素 e 进栈
    self.data.append(e)
```

3) 出栈：pop()

元素出栈只能从栈顶出，不能从栈底或中间位置出栈，如图 3.6 所示。

在出栈中，当栈空时抛出异常，否则直接利用 data 列表的 pop()方法出栈栈顶元素(列表的 pop()方法的时间复杂度为 $O(1)$)。对应的算法如下：

```
def pop(self):                             #元素出栈
    assert not self.empty()                #检测栈为空
    return self.data.pop()
```

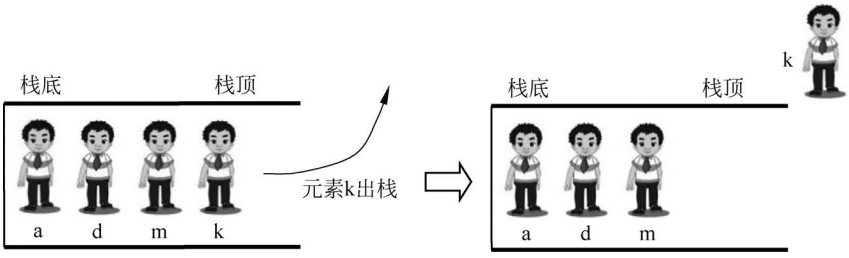


图 3.6　元素出栈示意图

4）取栈顶元素：gettop()

在栈不为空的条件下，返回栈顶元素 data[len(data)－1]（或者 data[－1]），不移动栈顶指针。对应的算法如下：

```
def gettop(self):                          # 取栈顶元素
    assert not self.empty()                # 检测栈为空
    return self.data[-1]
```

扫一扫

视频讲解

从以上看出，栈的各种基本运算算法的时间复杂度均为 $O(1)$。

3.1.3　顺序栈的应用算法设计示例

【例 3.4】　设计一个算法，利用顺序栈判断用户输入的表达式中的括号是否配对（假设表达式中可能含有圆括号、方括号和花括号），并用相关数据进行测试。

解　因为各种括号的匹配过程遵循最近匹配原则，任何一个右括号与前面最靠近的未匹配的同类左括号进行匹配，所以采用一个栈来实现匹配过程。

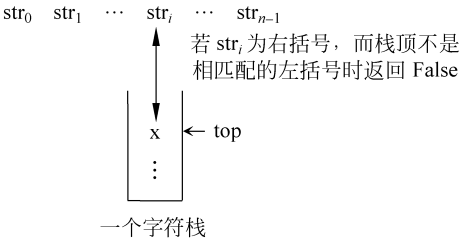


图 3.7　用一个栈判断 str 中的括号是否匹配

用 str 字符串存放含有各种括号的表达式，建立一个字符顺序栈 st，用 i 遍历 str，当遇到各种类型的左括号时进栈，当遇到右括号时，若栈空或者栈顶元素不是匹配的左括号时返回 False（中途就可以确定括号不匹配），如图 3.7 所示，否则退栈一次继续判断。当 str 遍历完毕，栈 st 为空返回 True，否则返回 False。

对应的完整程序如下：

```
from SqStack import SqStack                # 引用顺序栈 SqStack
def isMatch(str):                          # 判断表达式的各种括号是否匹配的算法
    st=SqStack()                           # 建立一个顺序栈
    i=0
    while i<len(str):
        e=str[i]
        if e=='(' or e=='[' or e=='{':
            st.push(e)                     # 将左括号进栈
        else:
            if e==')':
                if st.empty() or st.gettop()!='(':
```

```
                return False                    #栈空或栈顶不是'('时返回假
            st.pop()
        if e==']':
            if st.empty() or st.gettop()!='[':
                return False                    #栈空或栈顶不是'['时返回假
            st.pop()
        if e=='}':
            if st.empty() or st.gettop()!='{':
                return False                    #栈空或栈顶不是'{'时返回假
            st.pop()
    i+=1                                        #继续遍历 str
return st.empty()

#主程序
print("测试 1")
str="([)]"
if isMatch(str):
    print(str+"方括号是匹配的")
else:
    print(str+"方括号不匹配")
print("测试 2")
str="([])"
if isMatch(str):
    print(str+"方括号是匹配的")
else:
    print(str+"方括号不匹配")
```

上述程序的执行结果如下：

```
测试 1
([)]方括号不匹配
测试 2
([])方括号是匹配的
```

扫一扫

视频讲解

【例 3.5】 设计一个算法，利用顺序栈判断用户输入的字符串表达式是否为回文，并用相关数据进行测试。

解 用 str 存放表达式，其中含 n 个字符，建立一个顺序栈 st，可以将 str 中的 n 个字符 str_0，str_1，…，str_{n-1} 依次进栈再连续出栈，得到反向序列 str_{n-1}，…，str_1，str_0，若 str 与该反向序列相同，则是回文，否则不是回文。其可以改为更高效的方法，若 str 的前半部分的反向序列与 str 的后半部分相同，则是回文，否则不是回文。判断过程如下：

① 用 i 从头开始遍历 str，将前半部分字符依次进栈。

② 若 n 为奇数，i 增 1 跳过中间的字符。

③ i 继续遍历其他后半部分字符，每访问一个字符，则出栈一个字符，两者进行比较，如图 3.8 所示，若不相等返回 False。

④ 当 str 遍历完毕返回 True。

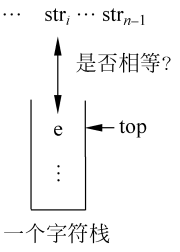


图 3.8 用一个栈判断 str 是否为回文

对应的完整程序如下：

```
from SqStack import SqStack                 #引用顺序栈 SqStack
def isPalindrome(str):                      #判断是否为回文的算法
```

```
    st=SqStack()                                    #建立一个顺序栈
    n=len(str)
    i=0
    while i<n//2:                                   #将 str 的前半字符进栈
        st.push(str[i])
        i+=1                                        #继续遍历 str
    if n%2==1:                                      #n 为奇数时
        i+=1                                        #跳过中间的字符
    while i<n:                                      #遍历 str 的后半字符
        if st.pop()!=str[i]:
            return False                            #若 str[i]不等于出栈字符返回 False
        i+=1
    return True                                     #是回文返回 True

#主程序
print("测试 1")
str="abcba"
if isPalindrome(str):
    print(str+"是回文")
else:
    print(str+"不是回文")
print("测试 2")
str="1221"
if isPalindrome(str):
    print(str+"是回文")
else:
    print(str+"不是回文")
```

上述程序的执行结果如下：

```
测试 1
abcba 是回文
测试 2
1221 是回文
```

扫一扫

视频讲解

【例 3.6】 设计最小栈。定义栈的数据结构，添加一个 Getmin()方法用于返回栈中的最小元素，要求方法 Getmin()、push()以及 pop()的时间复杂度都是 $O(1)$。例如：

```
push(5)      #栈元素：(5)          最小元素：5
push(6)      #栈元素：(6,5)        最小元素：5
push(3)      #栈元素：(3,6,5)      最小元素：3
push(7)      #栈元素：(7,3,6,5)    最小元素：3
pop()        #栈元素：(3,6,5)      最小元素：3
pop()        #栈元素：(6,5)        最小元素：5
```

其中栈元素按照栈顶到栈底的顺序列出。

解 由于可能有连续的进栈和出栈操作，并且栈中元素可能重复，所以仅保存栈中的一个最小元素不能满足题目的要求，为此设计满足题目要求的顺序栈类 STACK，它含 data 和 mindata 两个列表，data 列表表示 data 栈（主栈），mindata 列表表示 mindata 栈，后者作为存放当前最小元素的辅助栈。

当元素 $a_0,a_1,\cdots,a_i(i\geqslant 1)$进到 data 栈后，min 栈的栈顶元素 b_j 为 $a_0,a_1,\cdots,a_i$ 中的最小元素（含后进栈的重复最小元素），如图 3.9 所示。

例如，前面的栈操作中 data 和 mindata 栈的变化如图 3.10 所示。STACK 类的主要运算算法设计如下：

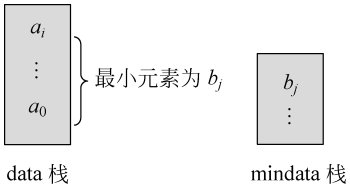


图 3.9 data 栈和 mindata 栈

① Getmin()方法用于返回栈中的最小元素，其操作是取 mindata 栈的栈顶元素。

② 进栈方法 push(x)的操作是，当 data 栈空或者进栈元素 x 小于或等于当前栈中最小元素(即 $x \leqslant$Getmin())时，将 x 进 mindata 栈。最后将 x 进 data 栈。

③ 出栈方法 pop()的操作是，当 data 栈不空时，从 data 栈出栈元素 x，若 mindata 栈的栈顶元素等于 x，则同时从 mindata 栈出栈 x。最后返回 x。

④ 取栈顶方法 gettop()的操作是，当 data 栈不空时，返回 data 栈的栈顶元素。

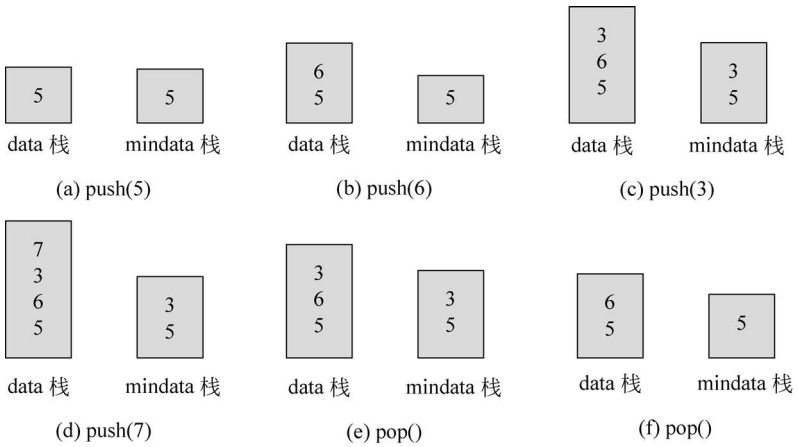


图 3.10 栈操作中 data 栈和 mindata 栈的变化情况

显然上述 4 个运算算法的时间复杂度均为 $O(1)$。对应的程序如下：

```
class STACK:                              #含 Getmin()的栈类
    def __init__(self):                   #构造方法
        self.data=[]                      #存放主栈中的元素,初始为空
        self.__mindata=[]                 #存放 min 栈中的元素,初始为空

    #min 栈的基本运算算法
    def __minempty(self):                 #判断 min 栈是否为空
        return len(self.__mindata)==0
    def __minpush(self,e):                #元素进 min 栈
        self.__mindata.append(e)
    def __minpop(self):                   #元素出 min 栈
        assert not self.__minempty()      #检测 min 栈为空的异常
        return self.__mindata.pop()
    def __mingettop(self):                #取 min 栈的栈顶元素
        assert not self.__minempty()      #检测 min 栈为空的异常
        return self.__mindata[-1]

    #主栈的基本运算算法
    def empty(self):                      #判断主栈是否为空
        return len(self.data)==0
    def push(self,x):                     #元素进主栈
```

```
            if self.empty() or x<=self.Getmin():
                self.__mindata.append(x)          #栈空或者 x≤min 栈顶元素时进 min 栈
            self.data.append(x)                   #将 x 进主栈
        def pop(self):                            #元素出主栈
            assert not self.empty()               #检测主栈为空的异常
            x=self.data.pop()                     #从主栈出栈 x
            if x==self.__mingettop():             #若栈顶元素为最小元素
                self.__minpop()                   #min 栈出栈一次
            return x
        def gettop(self):                         #取主栈的栈顶元素
            assert not self.empty()               #检测主栈为空的异常
            return self.data[-1]
        def Getmin(self):                         #获取栈中的最小元素
            assert not self.empty()               #检测主栈为空的异常
            return self.__mindata[-1]             #返回 min 栈的栈顶元素,即主栈中的最小元素

    #主程序
    st=STACK()
    print("\n 元素 5,6,3,7 依次进栈")
    st.push(5)
    st.push(6)
    st.push(3)
    st.push(7)
    print("求最小元素并出栈")
    while not st.empty():
        print("   最小元素:%d" %(st.Getmin()))
        print("   出栈元素:%d" %(st.pop()))
    print()
```

上述程序的执行结果如下：

```
元素 5,6,3,7 依次进栈
求最小元素并出栈
        最小元素:3
        出栈元素:7
        最小元素:3
        出栈元素:3
        最小元素:5
        出栈元素:6
        最小元素:5
        出栈元素:5
```

扫一扫

视频讲解

【例 3.7】 设有两个栈 S1 和 S2，它们都采用顺序栈存储，并且共享一个固定容量的存储区 $s[0..M-1]$，为了尽量利用空间，减少溢出的可能，请设计这两个栈的存储方式。

解 为了尽量利用空间，减少溢出的可能，可以采用让两个栈的栈顶相向（即进栈元素迎面增长）的存储方式，为此设置两个栈的栈顶指针分别为 top1 和 top2（均指向对应栈的栈顶元素），如图 3.11 所示。

栈 S1 空的条件是 top1＝－1；栈 S1 满的条件是 top1＝top2－1；元素 e 进栈 S1（栈不满时）的操作是 top1 增 1；s[top1]＝e；元素 e 出栈 S1（栈不空时）的操作是 e＝s[top1]；top1 减 1。

栈 S2 空的条件是 top2＝M；栈 S2 满的条件是 top2＝top1＋1；元素 e 进栈 S2（栈不满时）的操作是 top2 减 1；s[top2]＝e；元素 e 出栈 S2（栈不空时）的操作是 e＝s[top2]；top2 增 1。

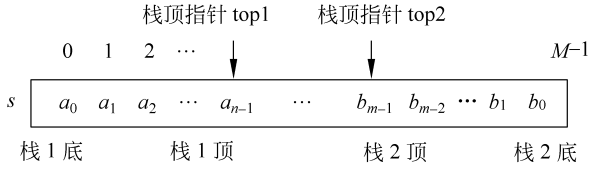


图 3.11 两个顺序栈的存储结构

说明：本例的共享栈主要适合将固定容量的空间用作两个栈，不适合 3 个或者更多栈共享，因为超过两个栈共享时栈的运算性能较低。

扫一扫 视频讲解

3.1.4 栈的链式存储结构及其基本运算算法的实现

采用链式存储的栈称为链栈，这里采用单链表实现。链栈的优点是不需要考虑栈满上溢出的情况。这里用带头结点的单链表 head 表示链栈，如图 3.12 所示，首结点是栈顶结点，尾结点是栈底结点，栈中元素自栈顶到栈底依次是 a_0，a_1，…，a_{n-1}。

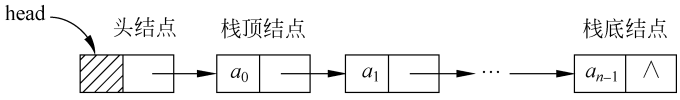


图 3.12 链栈的存储结构

从该链栈的存储结构看到，初始时只含有一个头结点 head 并置 head.next 为 None，这样链栈的四要素如下。

① 栈空条件：head.next==None。

② 栈满条件：由于只有在内存溢出才会出现栈满，通常不考虑这种情况。

③ 元素 e 进栈操作：将包含该元素的结点 s 插入作为首结点。

④ 出栈操作：返回首结点值并且删除该结点。

和普通单链表一样，链栈中每个结点的类型 LinkNode 定义如下：

```
class LinkNode:                          # 单链表结点类
    def __init__(self,data=None):        # 构造方法
        self.data=data                   # data 属性
        self.next=None                   # next 属性
```

链栈类 LinkStack 的设计如下(用 LinkStack.py 文件存放)：

```
class LinkStack:                         # 链栈类
    def __init__(self):                  # 构造方法
        self.head=LinkNode()             # 头结点 head
        self.head.next=None
    # 栈的基本运算算法
```

在链栈中实现栈的基本运算的算法如下。

1) 判断栈是否为空：empty()

若 head.next 为 None 表示空栈，即单链表中没有任何数据结点。对应的算法如下：

```
def empty(self):                         # 判断栈是否为空
    if self.head.next==None:
```

```
            return True
        return False
```

2）进栈：push(*e*)

新建包含数据元素 *e* 的结点 *p*，将 *p* 结点插入头结点之后。对应的算法如下：

```
def push(self,e):                          #元素 e 进栈
    p=LinkNode(e)
    p.next=self.head.next
    self.head.next=p
```

3）出栈：pop()

在链栈空时抛出异常，否则让 *p* 指向首结点，删除结点 *p* 并返回该结点值。对应的算法如下：

```
def pop(self):                             #元素出栈
    assert self.head.next!=None            #检测空栈的异常
    p=self.head.next
    self.head.next=p.next
    return p.data
```

4）取栈顶元素：gettop()

在链栈空时抛出异常，否则返回首结点值。对应的算法如下：

```
def gettop(self):                          #取栈顶元素
    assertself.head.next!=None             #检测空栈的异常
    return self.head.next.data
```

3.1.5 链栈的应用算法设计示例

扫一扫 视频讲解

【例 3.8】 设计一个算法，利用栈的基本运算将一个整数链栈中的所有元素逆置。例如链栈 st 中的元素从栈底到栈顶为(1,2,3,4)，逆置后为(4,3,2,1)。

解 因为这里要求利用栈的基本运算来设计算法，所以不能直接采用单链表逆置方法。先出栈 st 中的所有元素并保存在列表 *a* 中，再将列表 *a* 中的所有元素依次进栈。对应的算法如下：

```
from LinkStack import LinkStack            #引用链栈 LinkStack
def Reverse(st):                           #逆置栈 st
    a=[]
    while not st.empty():                  #将出栈的元素放到列表 a 中
        a.append(st.pop())
    for j in range(len(a)):                #将列表 a 中的所有元素进栈
        st.push(a[j])
    return st
```

扫一扫 视频讲解

【例 3.9】 有一个含 1～*n* 的 *n* 个整数的序列 *a*，通过一个栈可以产生多种出栈序列，设计一个算法采用链栈判断序列 *b*(为 1～*n* 的某个排列)是否为一个合适的出栈序列，并用相关数据进行测试。

解 建立一个整型链栈 st，用 *i*、*j* 分别遍历 *a*、*b* 序列(初始值均为 0)，在 *a* 序列没有遍

历完时循环。

① 将 $a[i]$进栈，i 增 1。

② 栈不空并且栈顶元素与 $b[j]$相同时循环：出栈元素 e，j 增 1。

在上述过程结束后，如果栈空返回 True，表示 b 序列是 a 序列的出栈序列；否则返回 False，表示 b 序列不是 a 序列的出栈序列。

例如，$a=[1,2,3,4,5]$，$b=[3,2,1,5,4]$，$i=0$，$j=0$，判断过程如下：

① 栈空，$a[0]$进栈（i 增 1 为 $i=1$）；$b[0]\neq$栈顶元素，$a[1]$进栈（i 增 1 为 $i=2$）；$b[0]\neq$栈顶元素，$a[2]$进栈（i 增 1 为 $i=3$），如图 3.13(a)所示。

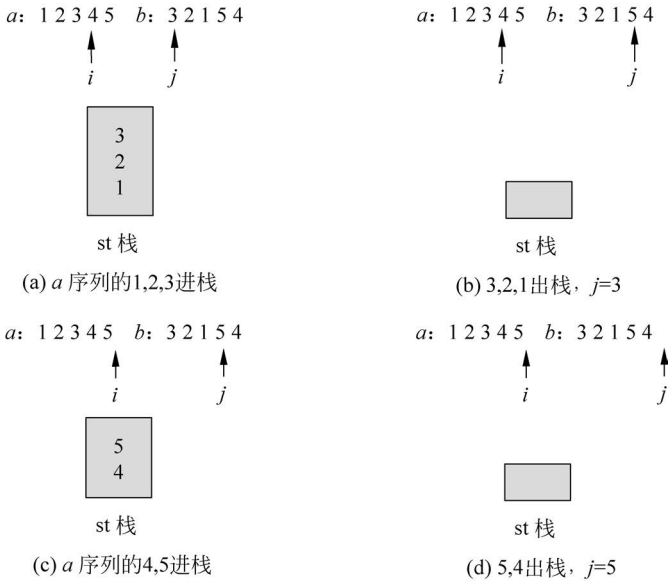


图 3.13 判断 $b=[3,2,1,5,4]$是否为出栈序列的过程

② $b[0]=$栈顶元素，出栈一次，j 增 1（$j=1$）；$b[1]=$栈顶元素，出栈一次，j 增 1（$j=2$）；$b[2]=$栈顶元素，出栈一次，j 增 1（$j=3$），如图 3.13(b)所示。

③ 栈空，$a[3]$进栈（i 增 1 为 $i=4$）；$b[3]\neq$栈顶元素，$a[4]$进栈（i 增 1 为 $i=5$，a 序列遍历完毕），如图 3.13(c)所示。

④ $b[3]=$栈顶元素，出栈一次，j 增 1（$j=4$）；$b[4]=$栈顶元素，出栈一次，j 增 1（$j=5$），如图 3.13(d)所示。

此时 a 序列遍历完毕，栈空返回 True，表示 b 序列是 a 序列的出栈序列。

又例如，$a=[1,2,3]$，$b=[3,1,2]$，$i=0$，$j=0$，判断过程如下：

① 栈空，$a[0]$进栈（i 增 1 为 $i=1$）；$b[0]\neq$栈顶元素，$a[1]$进栈（i 增 1 为 $i=2$）；$b[0]\neq$栈顶元素，$a[2]$进栈（i 增 1 为 $i=3$，a 序列遍历完毕），如图 3.14(a)所示。

② $b[0]=$栈顶元素，出栈一次，j 增 1（$j=1$）；$b[1]\neq$栈顶元素，如图 3.14(b)所示。

此时 a 序列遍历完毕，栈不空返回 False，表示 b 序列不是 a 序列的出栈序列。

说明：对于给定的 n，在上述两个示例中 $a=[1,2,\cdots,n]$，b 是 $1\sim n$ 的某个排列，判断 b 序列是否为 a 序列的出栈序列。上述算法适合 a 和 b 序列均为 $1\sim n$ 任意排列的情况。

对应的完整程序如下：

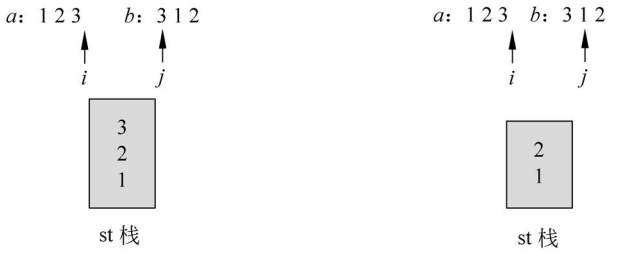


图 3.14 判断 $b=[3,1,2]$是否为出栈序列的过程

```
from LinkStack import LinkStack                    #引用链栈 LinkStack
def isSerial(a,b,n):                               #判断 b 是否为 a 的出栈序列的算法
    st=LinkStack()                                 #建立一个链栈
    i,j=0,0
    while i<n:                                     #遍历 a 序列
        st.push(a[i])
        i+=1                                       #i 后移
        while not st.empty() and st.gettop()==b[j]:
            st.pop()                               #出栈
            j+=1                                   #j 后移
    return st.empty()                              #栈空返回 True,否则返回 False

#主程序
n=4
a=[1,2,3,4]
print("测试 1")
b=[1,3,2,4]
if isSerial(a,b,n):
    print(b,"是合法的出栈序列")
else:
    print(b,"不是合法的出栈序列")
print("测试 2")
c=[4,3,1,2]
if isSerial(a,c,n):
    print(c,"是合法的出栈序列")
else:
    print(c,"不是合法的出栈序列")
```

上述程序的执行结果如下：

```
测试 1
[1,3,2,4]是合法的出栈序列
测试 2
[4,3,1,2]不是合法的出栈序列
```

说明：本题也可以采用顺序栈，其过程与上述算法类似。

3.1.6 栈的综合应用

本节通过利用栈求简单算术表达式的值和求解迷宫问题两个示例来说明栈的应用。

1. 用栈求简单表达式的值

❑ 问题描述

这里限定的简单算术表达式(简称为表达式)求值问题是,用户输入一个仅包含'+'、'-'、'*'、'/'、正整数和圆括号的合法算术表达式,计算该表达式的运算结果。

❑ 数据组织

扫一扫

视频讲解

表达式用字符串 exp 表示。在设计相关算法中用到两个栈,即一个运算符栈 opor 和一个运算数栈 opand,它们均为顺序栈 SqStack 的栈对象,其定义如下:

```
opor=SqStack()                    #运算符栈
opand=SqStack()                   #运算数栈
```

❑ 设计运算算法

运算符位于两个运算数中间的表达式称为**中缀表达式**,例如 exp="1+2*(4+12)"就是一个中缀表达式,中缀表达式是最常用的一种表达式形式。计算中缀表达式一般遵循"从左到右,先乘除,后加减,有括号时先括号内,后括号外"的规则,因此中缀表达式不仅要依赖运算符的优先级,而且还要处理括号。

所谓**后缀表达式**,就是运算符放在运算数的后面。后缀表达式有这样的特点,已经考虑了运算符的优先级,不包含括号,只含运算数和运算符。这里后缀表达式用列表 postexp 存放,前面 exp 对应的 postexp 为[1,2,4,12,'+','*','+']。

后缀表达式的求值十分简单,其过程是从左到右遍历后缀表达式,若遇到一个运算数,就将它进运算数栈;若遇到一个运算符 op,就从运算数栈中连续出栈两个运算数,假设为 a 和 b,计算 b op a 之值,并将计算结果进运算数栈;对整个后缀表达式遍历结束后,栈顶元素就是计算结果。

假设给定的简单表达式 exp 是正确的,其求值过程分为两步,即先将中缀表达式 exp 转换成后缀表达式 postexp,然后对后缀表达式求值。设计求表达式值的类 ExpressClass 如下:

```
class ExpressClass:                       #求表达式值的类
    def __init__(self,str):              #构造方法
        self.exp=str                      #存放中缀表达式
        self.postexp=[]                   #存放后缀表达式
    def getpostexp(self):                 #返回 postexp
        return self.postexp
    def Trans(self):                      #将 exp 转换为 postexp
        …
    def getValue(self):                   #计算后缀表达式 postexp 的值
        …
```

1) 中缀表达式转换成后缀表达式

在将正确的中缀表达式 exp 转换成后缀表达式 postexp 时仅用到运算符栈 opor,其转换过程是遍历 exp,遇到数字符,将连续的数字符转换为一个整数后添加到 postexp;遇到'(',将其进栈;遇到')',退栈运算符并添加到 postexp,直到退栈的是'('为止(该左括号不添加到 postexp 中);遇到运算符 op_2,将其与栈顶运算符 op_1 的优先级进行比较,只有当 op_2 的优先级高于 op_1 的优先级时才直接将 op_2 进栈,否则将栈中'('(如果有)之前的优先级等于或大于 op_2 的运算符均退栈并添加到 postexp,如图 3.15 所示,再将 op_2 进栈。

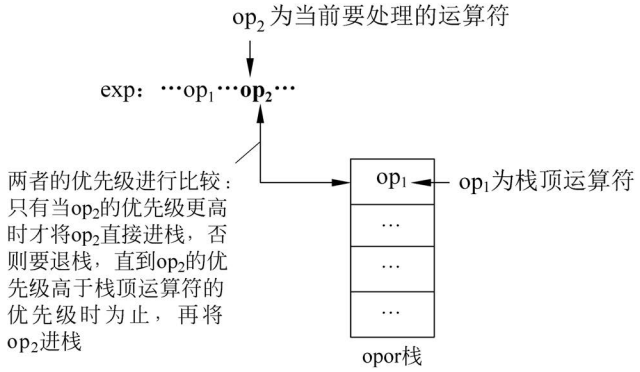


图 3.15　当前运算符的操作

上述过程的说明如下，在遍历 exp 的任何运算符 op 时，除非遍历结束，都不能确定是否立即执行 op，所以将其暂时保存在 opor 栈中。假设 exp 中只有 op_1 和 op_2 运算符，op_2 的处理过程如下：

① 当 op_2 和 op_1 的优先级相同时，由于 op_1 先进栈，说明 exp 中 op_1 在 op_2 的前面，按中缀表达式的运算规则，先做 op_1，即出栈 op_1 并添加到 postexp 中（按后缀表达式的求值过程，先添加的先执行），再将 op_2 进栈。

② 当 op_2 的优先级低于 op_1 的优先级时，显然先做 op_1，也就是出栈 op_1 并添加到 postexp 中，再将 op_2 进栈。

③ 当 op_2 的优先级高于 op_1 的优先级时，按中缀表达式的运算规则，op_2 应该在 op_1 之前做，此时直接将 op_2 进栈，以后 op_2 一定先于 op_1 出栈，从而满足该运算规则。

④ 当 op_2 为'('时，表示开始处理一个表达式（该表达式形如“(…)”，此时遇到开头的'('或者开始处理一个子表达式，所以无论栈中有什么运算符，都直接将'('进栈。

⑤ 当 op_2 为')'时，表示一个表达式或者子表达式处理结束，由于假设表达式中的括号是匹配的，所以栈中一定存在'('，设从栈顶到栈底方向的第一个'('的位置为 p，如图 3.16 所示，将从栈顶到 p 位置前的所有运算符出栈并添加到 postexp，再出栈'('，该'('不需要添加到 postexp。

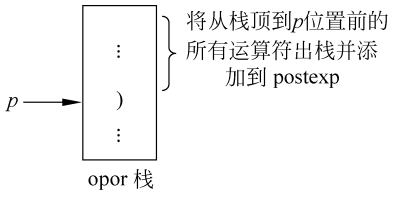


图 3.16　遇到')'的处理方式

由于中缀表达式和后缀表达式中所有运算数的相对次序相同，所以遇到每个运算数都直接添加到 postexp。

针对这里的简单算术表达式，只有'*'和'/'运算符的优先级高于'+'和'-'运算符的优先级，所以上述过程简化如下：

```
while（若 exp 未读完）{
    从 exp 读取字符 ch
    ch 为数字符：将后续的所有数字符转换为一个整数后{添加到 postexp 中
    ch 为左括号'('：将'('进栈
    ch 为右括号')'：将栈中与之匹配的'('后进栈的运算符依次出栈并添加到 postexp 中，再将'('退栈
    ch 为'+'或'-'：将 opor 栈中'('后进栈的（如果有'('）所有运算符出栈并添加到 postexp，再将 ch 进栈
```

ch为'*'或'/':将opor栈中'('后进栈的(如果有'(')所有'*'或'/'运算符出栈并添加到postexp,再将ch进栈

}

若字符串exp扫描完毕,则退栈所有运算符并添加到postexp

例如,对于exp="(56－20)/(4＋2)",其转换成后缀表达式的过程如表3.1所示。

表3.1 表达式"(56－20)/(4＋2)"转换成后缀表达式的过程

ch	操　作	postexp	opor栈
(	将'('进栈		(
56	将56存入postexp中	[56]	(
－	由于栈顶为'(',直接将'－'进栈	[56]	(－
20	将20添加到postexp	[56,20]	(－
)	将栈中'('后进栈的运算符出栈并添加到postexp,再将'('出栈	[56,20,'－']	
/	将'/'进栈	[56,20,'－']	/
(	将'('进栈	[56,20,'－']	/(
4	将4添加到postexp	[56,20,'－',4]	/(
＋	由于栈顶为'(',直接将'＋'进栈	[56,20,'－',4]	/(＋
2	将2添加到postexp	[56,20,'－',4,2]	/(＋
)	将栈中'('后进栈的运算符出栈并添加到postexp,再将'('出栈	[56,20,'－',4,2,'＋']	/
	exp扫描完毕,将栈中所有的运算符依次出栈并添加到postexp,得到最后的后缀表达式	[56,20,'－',4,2,'＋','/']	

根据上述原理得到的中缀表达式转后缀表达式的算法如下:

```
def Trans(self):                                    #将exp转换为postexp
    opor=SqStack()                                  #定义运算符栈
    i=0                                             #i作为exp的索引
    while i<len(self.exp):                          #遍历exp
        ch=self.exp[i]                              #提取str[i]字符ch
        if ch=="(":                                 #判定为左括号,将左括号进栈
            opor.push(ch)
        elif ch==")":                               #判定为右括号
            while not opor.empty() and opor.gettop()!="(":
                e=opor.pop()                        #将栈中最近的"("之前的运算符退栈
                self.postexp.append(e)              #退栈运算符添加到postexp
            opor.pop()                              #再将(退栈
        elif ch=="+" or ch=="-":                    #判定为加号或减号
            while not opor.empty() and opor.gettop()!="(":
                e=opor.pop()                        #将栈中不低于ch优先级的所有运算符退栈
                self.postexp.append(e)              #退栈运算符添加到postexp
            opor.push(ch)                           #再将"+"或"-"进栈
        elif ch=="*" or ch=="/":                    #判定为*号或/号
            while not opor.empty():
                e=opor.gettop()
                if e!="(" and (e=="*" or e=="/"):
                    e=opor.pop()                    #将栈中不低于ch优先级的所有运算符退栈
                    self.postexp.append(e)          #退栈运算符添加到postexp
                else: break
```

```
            opor.push(ch)                          #再将"*"或"/"进栈
        else:                                      #处理数字字符
            d=""
            while ch>="0" and ch<="9":             #判定为数字
                d+=ch                              #提取所有连续的数字字符
                i+=1
                if i<len(self.exp):
                    ch=self.exp[i]
                else:
                    break
            i-=1                                   #退一个字符
            self.postexp.append(int(d))            #将连续的数字字符转换为整数并添加到 postexp
        i+=1                                       #继续处理其他字符
    while not opor.empty():                        #此时 exp 扫描完毕,栈不空时循环
        e=opor.pop()                               #将栈中所有的运算符退栈并添加到 postexp
        self.postexp.append(e)
```

2）后缀表达式求值

在后缀表达式求值中仅用到运算数栈 opand，后缀表达式求值的过程如下：

```
while（若 postexp 未读完）{
    从 postexp 读取一个元素 opv
    opv 为'+': 出栈两个数值 a 和 b,计算 c=b+a,再将 c 进栈
    opv 为'-': 出栈两个数值 a 和 b,计算 c=b-a,再将 c 进栈
    opv 为'*': 出栈两个数值 a 和 b,计算 c=b*a,再将 c 进栈
    opv 为'/': 出栈两个数值 a 和 b,若 a 不为零,计算 c=b/a,再将 c 进栈
    opv 为数值: 将该数值进栈
}
opand 栈中唯一的数值即为表达式的值
```

例如，postexp=[56,20,'-',4,2,'+','/']的求值过程如表 3.2 所示。

表 3.2　后缀表达式“56#20#-4#2#+/”的求值过程

ch 序列	说　明	opand 栈
56	遇到 56，将 56 进栈	56
20	遇到 20，将 20 进栈	56,20
'-'	遇到'-'，出栈两次，将 56-20=36 进栈	36
4	遇到 4，将 4 进栈	36,4
2	遇到 2，将 2 进栈	36,4,2
'+'	遇到'+'，出栈两次，将 4+2=6 进栈	36,6
'/'	遇到'/'，出栈两次，将 36/6=6 进栈	6
	postexp 遍历完毕，算法结束，栈顶数值 6 即为所求	

根据上述计算原理得到计算后缀表达式的值的算法如下：

```
def getValue(self):                                #计算后缀表达式 postexp 的值
    opand=SqStack()                                #定义运算数栈
    i=0
    while i<len(self.postexp):                     #遍历 postexp
        opv=self.postexp[i]                        #从后缀表达式中取一个元素 opv
        if opv=="+":                               #判定为+号
            a=opand.pop()                          #退栈取运算数 a
```

```
            b=opand.pop()                    #退栈取运算数 b
            c=b+a                            #计算 c
            opand.push(c)                    #将计算结果进栈
        elif opv=="-":                       #判定为-号
            a=opand.pop()                    #退栈取运算数 a
            b=opand.pop()                    #退栈取运算数 b
            c=b-a                            #计算 c
            opand.push(c)                    #将计算结果进栈
        elif opv=="*":                       #判定为*号
            a=opand.pop()                    #退栈取运算数 a
            b=opand.pop()                    #退栈取运算数 b
            c=b*a                            #计算 c
            opand.push(c)                    #将计算结果进栈
        elif opv=="/":                       #判定为/号
            a=opand.pop()                    #退栈取运算数 a
            b=opand.pop()                    #退栈取运算数 b
            assert a!=0                      #检测 a 为 0 的情况
            c=b/a                            #计算 c
            opand.push(c)                    #将计算结果进栈
        else:                                #处理运算数
            opand.push(opv)                  #将运算数 opv 进栈
        i+=1                                 #继续处理 postexp 的其他元素
    return opand.gettop()                    #栈顶元素即为求值结果
```

❑ 设计主程序

设计以下主程序求简单算术表达式“(56-20)/(4+2)”的值：

```
str="(56-20)/(4+2)"
print("中缀表达式: "+str)
obj=ExpressClass(str)
obj.Trans()
print("后缀表达式:",obj.getpostexp())
print("求值结果:    %g" %(obj.getValue()))
```

❑ 程序执行结果

本程序的执行结果如下：

```
中缀表达式: (56-20)/(4+2)
后缀表达式: [56,20,'-',4,2,'+','/']
求值结果:    6
```

上述先转换为后缀表达式再对后缀表达式求值这两步可以合并起来，同样需要设置运算符栈 opor 和运算数栈 opand，合并后的过程是遍历表达式 exp：

① 遇到数字符，将后续的所有数字符合起来转换为数值，进栈到 opand。

② 遇到左括号，进栈到 opor。

③ 遇到右括号，将 opor 栈中与之匹配的'('后进栈的运算符依次出栈并做相应的计算。

④ 遇到运算符 opv，只有优先级高于 opor 栈顶的运算符才直接将 opv 进栈 opor，否则出栈 op 并执行 op 计算，直到栈顶是运算符'('(如果有)时为止，此时'('出栈，最后将 opv 进栈 opor。

若简单表达式遍历完毕，出栈 opor 的所有运算符并执行 op 计算。

其中执行 op 计算的过程是，出栈 opand 两次得到运算数 a 和 b，执行 $c=b$ op a，然后 c

进栈 opand。最后 opand 栈的栈顶运算数就是简单表达式的值。

2. 用栈求解迷宫问题

❑ 问题描述

扫一扫

视频讲解

给定一个 $M\times N$ 的迷宫图，求一条从指定入口到出口的路径。假设迷宫图如图 3.17 所示（其中 $M=6, N=6$，含外围加上的一圈不可走的方块，这样做的目的是避免在查找时出界），迷宫由方块构成，空白方块表示可以走的通道，带阴影方块表示不可走的障碍物。要求所求路径必须是简单路径，即在求得的路径上不能重复出现同一个空白方块，而且从每个方块出发每一步只能走向上、下、左、右 4 个相邻的空白方块。

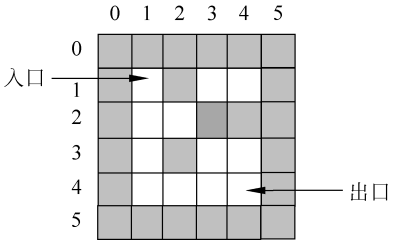


图 3.17 迷宫示意图

❑ 迷宫的数据组织

为了表示迷宫，设置一个数组 mg，其中的每个元素表示一个方块的状态，为 0 时表示对应方块是通道，为 1 时表示对应方块不可走。为了设计算法方便，在迷宫的外围加了一圈围墙。图 3.17 所示的迷宫对应的迷宫数组 mg（由于在迷宫外围加了一圈围墙，所以数组 mg 的外围元素均为 1）如下：

```
mg=[[1,1,1,1,1,1],[1,0,1,0,0,1],[1,0,0,1,1,1],[1,0,1,0,0,1],[1,0,0,0,0,1],
    [1,1,1,1,1,1]]
```

❑ 设计运算算法

求迷宫问题就是在一个指定的迷宫中求出一条从入口到出口的路径。在求解时通常用的是“穷举求解”的方法，即从入口出发，沿着某个方位向前试探，若能走通，则继续往前走；否则进入死胡同，沿原路退回，换一个方位再继续试探，直到所有可能的通路都试探完为止。

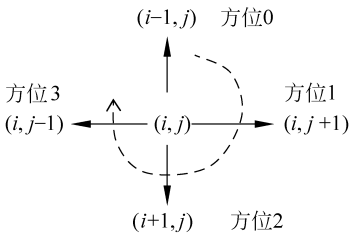


图 3.18 方位图

对于迷宫中的每个方块，有上、下、左、右 4 个相邻方块，如图 3.18 所示，第 i 行第 j 列的方块的位置记为 (i,j)，规定上方方块为方位 0，并按顺时针方向递增编号。对应的方位偏移量如下：

```
dx=[-1,0,1,0]                    #x方向的偏移量
dy=[0,1,0,-1]                    #y方向的偏移量
```

为了保证在任何位置上都能沿原路退回（称为回溯），需要用一个后进先出的栈来保存从入口到当前方块的路径，也就是说每个可走的方块都要进栈，栈中保存的每个方块除了位置信息外，还有走向信息，即从该方块走到相邻方块的方位 di。st 栈用 SqStack 对象表示。每个方块的 Box 类定义如下：

```
class Box:                       #方块类
    def __init__(self,i1,j1,di1):#构造方法
        self.i=i1                #方块的行号
```

```
self.j=j1                                  #方块的列号
self.di=di1                                #di是可走相邻方位的方位号
```

说明：栈是一种具有记忆功能的数据结构，在应用中重点是确定栈元素保存哪些信息。这里看一个日常生活的例子，如图 3.19 所示，假设小明住在 A 地，想到 C 地去看望好朋友，但他不熟悉路线。他从 A 地出发，走到了 B 地，有两条道路，于是他习惯性地走了上方的道路，结果遇到了一条小河，他过不去，只好回到 B 地。如果他不记下前面走过的路线，他会继续在这条路线上陷入死循环，永远见不到好朋友。小明是个聪明的孩子，他会记下前面走过的路线，于是在 B 地走另外一条(下方的)道路，结果很快找到了 C 地，高兴地见到了好朋友。在这个例子中，小明要记下走过的每个地点以及所走方向，小明的记忆功能可以用栈来实现。也就是说，在求解迷宫问题中用栈保存每个走过的方块以及所走方位。

求解从入口(xi,yi)到出口(xe,ye)迷宫路径的过程是先将入口进栈(其初始方位设置为－1)，在栈不空时循环：

① 取栈顶方块 b(不退栈)。

② 若 b 方块是出口，则输出栈中的所有方块即为一条迷宫路径，返回 True。

③ 否则从 b 方块的新方位 di=b.di+1 开始试探相邻方块是否可走。

④ 若找到 b 方块的 di 方位的相邻方块(i,j)可走，则走到相邻方块(i,j)，操作是修改栈顶 b 方块的 di 属性为该 di 值，并将(i,j)方块(对应 $b1$)进栈(其初始方位设置为－1)。

⑤ 若 b 方块找不到相邻可走方块，说明当前路径不可能走通(进入死胡同)，b 方块不会是迷宫路径上的方块，则原路回退(即回溯)，操作是将 b 方块出栈，从次栈顶方块(试探路径上 b 方块的前一个方块)做相同的试探，如图 3.20 所示(图中虚线表示回退)。如果一直回退到出口，而出口也没有未试探过的相邻可走方块，说明不存在迷宫路径，返回 False。

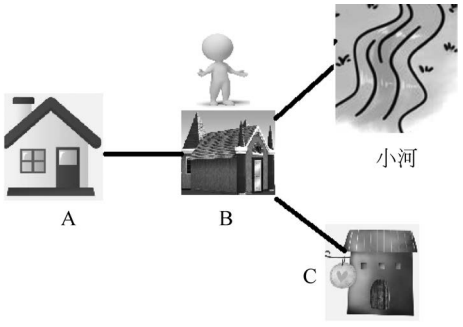


图 3.19　小明找好朋友的过程

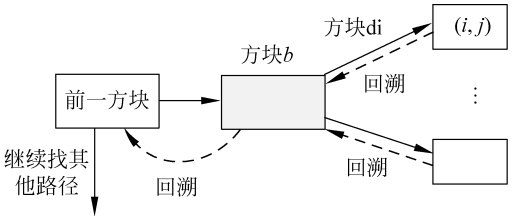


图 3.20　求迷宫路径的回溯过程

为了保证试探的可走相邻方块不是已走路径上的方块，如(i,j)已进栈，在试探$(i+1,j)$的下一个可走方块时又试探到(i,j)，这样可能引起死循环，为此在一个方块进栈后将对应的 mg 数组元素值改为－1(变为不可走的相邻方块)，当退栈时(表示该栈顶方块没有可走相邻方块)将其恢复为 0。图 3.17 中求从入口(1,1)到出口(4,4)的迷宫路径的搜索过程如图 3.21 所示，图中带“×”的方块是死胡同方块，走到这样的方块后需要回溯，找到出口后，栈中方块对应一条迷宫路径。

说明：这里的迷宫数组 mg 除了表示一个迷宫外，还通过将元素值设置为－1 以记忆路径，当找到出口后，恰好该迷宫路径上所有方块的 mg 元素值均为－1，这样可以在出口处继

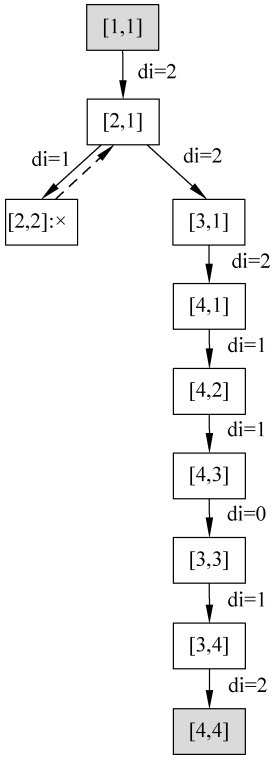


图 3.21　用栈求(1,1)到(4,4)迷宫路径的搜索过程

续回退查找其他迷宫路径。如果在一个方块出栈时不将其 mg 元素值恢复为 0，尽管可以找到一条迷宫路径(当存在迷宫路径时)，但会将所有试探方块的 mg 元素值均置为－1，这样不能找到其他可能存在的迷宫路径。

求解迷宫问题的 mgpath()函数如下：

```
def mgpath(xi,yi,xe,ye):                    #求一条从(xi,yi)到(xe,ye)的
                                            #迷宫路径
    global mg                               #迷宫数组为全局变量
    st=SqStack()                            #定义一个顺序栈
    dx=[-1,0,1,0]                           #x方向的偏移量
    dy=[0,1,0,-1]                           #y方向的偏移量
    e=Box(xi,yi,-1)                         #建立入口方块对象
    st.push(e)                              #入口方块进栈
    mg[xi][yi]=-1                           #为避免来回找相邻方块,将进栈的方块置为-1
    while not st.empty():                   #栈不空时循环
        b=st.gettop()                       #取栈顶方块,称为当前方块
        if b.i==xe and b.j==ye:             #找到了出口,输出栈中所有方块构成一条路径
            for k in range(len(st.data)):
                print("["+str(st.data[k].i)+','+str(st.data[k].j)+"]",end=' ')
            return True                     #找到一条路径后返回 True
        find=False                          #否则继续找路径
        di=b.di
        while di<3 and find==False:         #找 b 的一个相邻可走方块
            di+=1                           #找下一个方位的相邻方块
            i,j=b.i+dx[di],b.j+dy[di]       #找 b 的 di 方位的相邻方块(i,j)
            if mg[i][j]==0:                 #(i,j)方块可走
```

```
                find=True
        if find:                                    #找到了一个相邻可走方块(i,j)
            b.di=di                                 #修改栈顶方块的 di 为新值
            b1=Box(i,j,-1)                          #建立相邻可走方块(i,j)的对象 b1
            st.push(b1)                             #b1 进栈
            mg[i][j]=-1                             #为避免来回找相邻方块,将进栈的方块置为-1
        else:                                       #没有路径可走,则退栈
            mg[b.i][b.j]=0                          #恢复当前方块的迷宫值
            st.pop()                                #将栈顶方块退栈
    return False                                    #没有找到迷宫路径,返回 False
```

当成功找到出口后,st 栈中从栈底到栈顶恰好是一条从入口到出口的迷宫路径,输出该迷宫路径并返回 True,否则说明找不到迷宫路径,返回 False。

❑ **设计主程序**

设计以下主程序求如图 3.17 所示的迷宫图中从(1,1)到(4,4)的一条迷宫路径:

```
xi,yi=1,1
xe,ye=4,4
print("一条迷宫路径:",end=' ')
if not mgpath(xi,yi,xe,ye):                         #(1,1)->(4,4)
    print("不存在迷宫路径")
```

❑ **程序执行结果**

本程序的执行结果如下:

```
一条迷宫路径:[1,1] [2,1] [3,1] [4,1] [4,2] [4,3] [3,3] [3,4] [4,4]
```

该路径如图 3.22 所示(迷宫路径方块上的箭头指示路径中下一个方块的方位),显然这个解不是最优解(即不是最短路径)。在后面使用队列求解时可以找出最短路径。

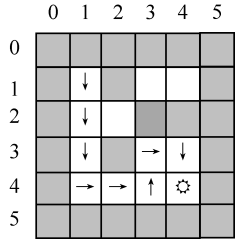


图 3.22 用栈找到的一条迷宫路径

3.2 队 列

本节先介绍队列的定义,然后讨论队列的存储结构和基本运算算法设计,最后通过迷宫问题的求解说明队列的应用。

扫一扫

视频讲解

3.2.1 队列的定义

同样先看一个示例,假设有一座独木桥,桥右侧有一群小兔子要过桥到桥左侧去,桥宽

只能容纳一只兔子，那么这群小兔子怎么过桥呢？结论是只能一个接一个地过桥，如图 3.23 所示。在这个例子中，独木桥就是一个队列，由于其宽度只能容纳一只兔子，所以不论有多少只兔子，它们只能是一只一只地排列过桥，从而构成一种线性关系。再看看独木桥的主要操作，显然有上桥和下桥，上桥表示从桥右侧走到桥上，下桥表示离开桥。

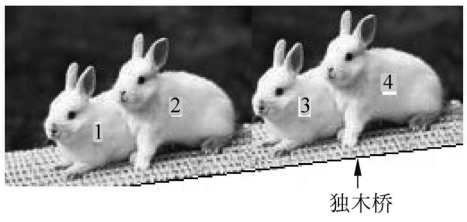


图 3.23　一群小兔子过独木桥

归纳起来，队列（简称为队）是一种操作受限的线性表，其限制为仅允许在表的一端进行插入，而在表的另一端进行删除。把进行插入的一端称作**队尾**（rear），把进行删除的一端称作**队头**或**队首**（front）。向队列中插入新元素称为**进队**或**入队**，新元素进队后就成为新的队尾元素；从队列中删除元素称为**出队**或**离队**，元素出队后，其直接后继元素就成为队首元素。

由于队列的插入和删除操作分别是在表的一端进行的，每个元素必然按照进入的次序出队，所以又把队列称为**先进先出表**。

抽象数据类型队列的定义如下：

```
ADT Queue {
    数据对象：
        D={a_i | 0≤i≤n-1,n≥0}
    数据关系：
        R={r}
        r={<a_i,a_i+1> | a_i,a_i+1∈D,i=0,1,…,n-2}
    基本运算：
        empty()：判断队列是否为空，若队列为空，返回真，否则返回假。
        push(e)：进队，将元素 e 进队作为队尾元素。
        pop()：出队，从队头出队一个元素。
        gethead()：取队头，返回队头元素的值而不出队。
}
```

【例 3.10】　若元素的进队顺序为 1234，能否得到 3142 的出队序列？

解　进队顺序为 1234，则出队顺序只能是 1234（先进先出），所以不能得到 3142 的出队序列。

3.2.2　队列的顺序存储结构及其基本运算算法的实现

由于队列中元素的逻辑关系与线性表的相同，所以可以借鉴线性表的两种存储结构来存储队列。

当队列采用顺序存储结构存储时，用列表 data 来存放队列中的元素，另外设置两个指针，队头指针为 front（实际上是队头元素的前一个位置），队尾指针为 rear（正好是队尾元素的位置）。

为了简单起见，这里使用固定容量的列表 data（容量用常量 MaxSize 表示），如图 3.24 所示，队列中从队头到队尾为 $a_0,a_1,\cdots,a_{n-1}$。采用顺序存储结构的队列称为**顺序队**。

顺序队分为非循环队列和循环队列两种方式，下面先讨论非循环队列，并通过说明该类型队列的缺点引出循环队列。

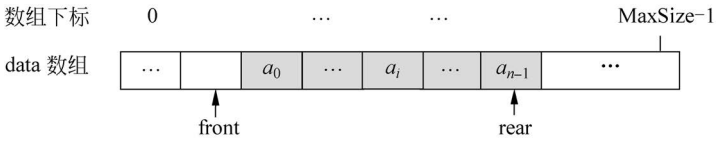


图 3.24 顺序队示意图

扫一扫

视频讲解

1. 非循环队列

图 3.25 是一个非循环队列的动态变化示意图(MaxSize=5)。图 3.25(a)表示一个空队；图 3.25(b)表示进队 5 个元素后的状态；图 3.25(c)表示出队一次后的状态；图 3.25(d)表示再出队 4 次后的状态。

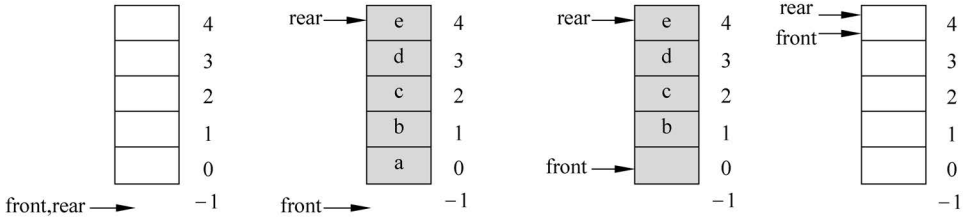


图 3.25 队列操作的示意图

从中看到，初始时置 front 和 rear 均为－1(front==rear)，非循环队列的四要素如下。

① 队空条件：front==rear，图 3.25(a)和图 3.25(d)满足该条件。

② 队满(队上溢出)条件：rear==MaxSize－1(因为每个元素进队都让 rear 增 1，当 rear 到达最大下标时不能再增加)，图 3.25(d)满足该条件。

③ 元素 e 进队操作：先将队尾指针 rear 增 1，然后将元素 e 放在该位置(进队的元素总是在尾部插入的)。

④ 出队操作：先将队头指针 front 增 1，然后取出该位置的元素(出队的元素总是从头部出来的)。

非循环队列类 SqQueue 的定义如下(用 SqQueue.py 文件存放)：

```
MaxSize=100                                  #假设容量为 100
class SqQueue:                               #非循环队列类
    def __init__(self):                      #构造方法
        self.data=[None]*MaxSize             #存放队列中的元素
        self.front=-1                        #队头指针
        self.rear=-1                         #队尾指针
    #队列的基本运算算法
```

在非循环队列中实现队列的基本运算算法如下。

1) 判断队列是否为空：empty()

若满足 front==rear 条件，返回 True，否则返回 False。对应的算法如下：

```
def empty(self):                             #判断队列是否为空
    return self.front==self.rear
```

2) 进队运算：push(e)

元素 e 进队只能从队尾插入，不能从队头或中间位置进队，仅改变队尾指针，如图 3.26

所示。进队操作是在队不满的条件下先将队尾指针 rear 增 1，然后将元素 *e* 放到该位置处，否则抛出异常。对应的算法如下：

```
def push(self,e):                          # 元素 e 进队
    assert not self.rear==MaxSize-1        # 检测队满
    self.rear+=1
    self.data[self.rear]=e
```

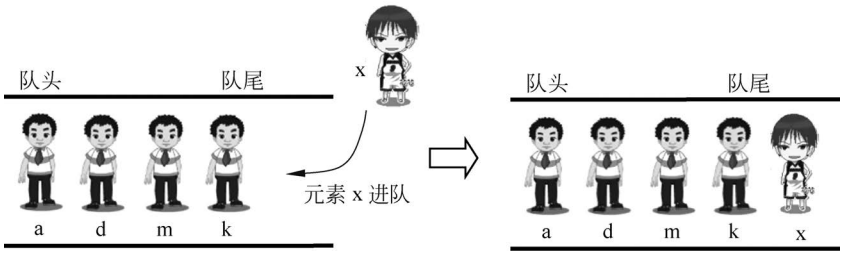


图 3.26　元素进队的示意图

3）出队：pop()

元素出队只能从队头删除，不能从队尾或中间位置出队，仅改变队头指针，如图 3.27 所示。

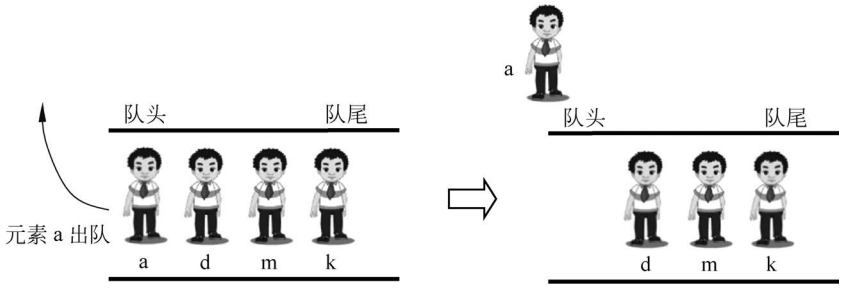


图 3.27　元素出队的示意图

出队操作是在队列不为空的条件下将队头指针 front 增 1，并返回该位置的元素值，否则抛出异常。对应的算法如下：

```
def pop(self):                             # 出队元素
    assert not self.empty()                # 检测队空
    self.front+=1
    return self.data[self.front]
```

4）取队头元素：gethead()

与出队类似，但不需要移动队头指针 front。对应的算法如下：

```
def gethead(self):                         # 取队头元素
    assert not self.empty()                # 检测队空
    return self.data[self.front+1]
```

上述算法的时间复杂度均为 $O(1)$。

扫一扫
视频讲解

2. 循环队列

在前面的非循环队列中，元素进队时队尾指针 rear 增 1，元素出队时队头指针 front 增

1，当进队 MaxSize 个元素后，满足设置的队满条件，即 rear==MaxSize-1 成立，此时即使出队若干元素，队满条件仍成立(实际上队列中有空位置)，这种队列中有空位置但仍然满足队满条件的上溢出称为**假溢出**。也就是说，非循环队列存在假溢出现象。为了克服非循环队列的假溢出，充分使用数组中的存储空间，可以把 data 数组的前端和后端连接起来，形成一个循环数组，即把存储队列元素的表从逻辑上看成一个环，称为**循环队列**(也称为**环形队列**)。

循环队列首尾相连，当队尾指针 rear=MaxSize-1 时，再前进一个位置就应该到达 0 位置，这可以利用数学上的求余运算(%)实现。

① 队首指针循环进 1：front=(front+1) % MaxSize。

② 队尾指针循环进 1：rear=(rear+1) % MaxSize。

循环队列的队头指针和队尾指针均初始化为 0，即 front=rear=0。在进队元素和出队元素时，队头和队尾指针都循环前进一个位置。

那么循环队列的队满和队空的判断条件是什么呢？若设置队空条件是 rear==front，如果进队元素的速度快于出队元素的速度，队尾指针很快就赶上了队头指针，此时可以看出循环队列队满时也满足 rear==front，所以这种设置无法区分队空和队满。

实际上循环队列的结构与非循环队列相同，也需要通过 front 和 rear 标识队列状态，一般是采用它们的相对值(即|front-rear|)实现的，当 data 数组的容量为 MaxSize 时，则队列的状态有 MaxSize+1 种，分别是队空、队中有 1 个元素、队中有 2 个元素、……、队中有 MaxSize 个元素(队满)。而 front 和 rear 的取值范围均为 0～MaxSize-1，这样|front-rear|只有 MaxSize 个值，显然 MaxSize+1 种状态不能直接用|front-rear|区分，因为必定有两种状态不能区分。为此让队列中最多只有 MaxSize-1 个元素，这样队列恰好只有 MaxSize 种状态，就可以通过 front 和 rear 的相对值区分所有状态了。

在规定队列中最多只有 MaxSize-1 个元素时，设置队空条件仍然是 rear==front。当队列中有 MaxSize-1 个元素时一定满足(rear+1)%MaxSize==front。这样循环队列在初始时置 front=rear=0，其四要素如下。

① 队空条件：rear==front。

② 队满条件：(rear+1)%MaxSize==front(相当于试探进队一次，若 rear 达到 front，则认为队满了)。

③ 元素 e 进队操作：rear=(rear+1)%MaxSize，将元素 e 放置在该位置。

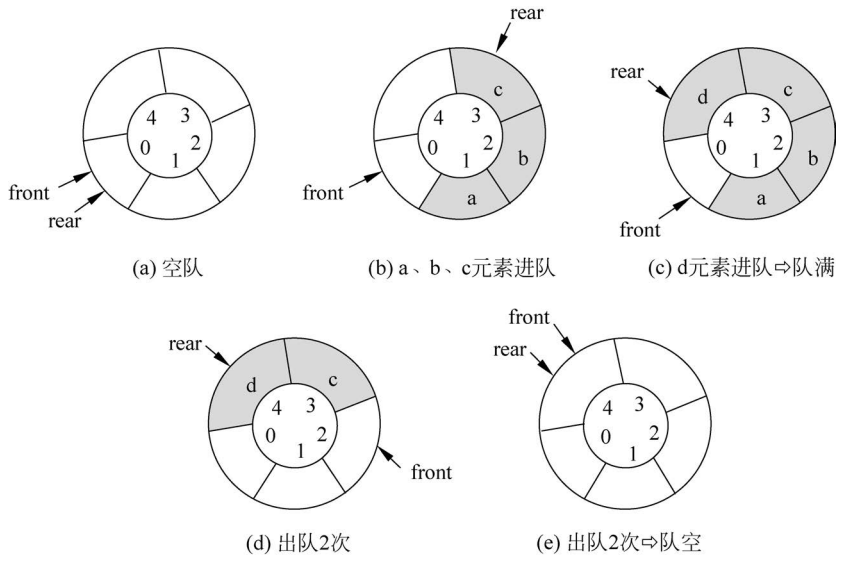
④ 元素出队操作：front=(front+1)%MaxSize，取出该位置的元素。

图 3.28 说明了循环队列的几种状态，这里假设 MaxSize 等于 5。图 3.28(a)为空队，此时 front=rear=0；图 3.28(b)表示队列中有 3 个元素，当进队元素 d 后，队中有 4 个元素，此时满足队满的条件。

循环队列类 CSqQueue 的定义如下(用 CSqQueue.py 文件存放)：

```
MaxSize=100                              # 全局变量,假设容量为 100
class CSqQueue:                          # 循环队列类
    def __init__(self):                  # 构造方法
        self.data=[None] * MaxSize       # 存放队列中的元素
        self.front=0                     # 队头指针
```

```
        self.rear=0                                    # 队尾指针
    # 队列的基本运算算法
```



(a) 空队　(b) a、b、c元素进队　(c) d元素进队⇨队满

(d) 出队2次　(e) 出队2次⇨队空

图 3.28　循环队列进队和出队操作示意图

在这样的循环队列中，实现队列的基本运算算法如下。

1）判断队列是否为空：empty()

若满足 front==rear 条件，返回 True，否则返回 False。对应的算法如下：

```
def empty(self):                                       # 判断队列是否为空
    return self.front==self.rear
```

2）进队：push(e)

在队列不满的条件下，先将队尾指针 rear 循环增 1，然后将元素 e 放到该位置处，否则抛出异常。对应的算法如下：

```
def push(self,e):                                      # 元素 e 进队
    assert (self.rear+1)%MaxSize!=self.front           # 检测队满
    self.rear=(self.rear+1)%MaxSize
    self.data[self.rear]=e
```

3）出队：pop()

在队列不为空的条件下将队头指针 front 循环增 1，并返回该位置的元素值，否则抛出异常。对应的算法如下：

```
def pop(self):                                         # 出队元素
    assert not self.empty()                            # 检测队空
    self.front=(self.front+1)%MaxSize
    return self.data[self.front]
```

4）取队头元素：gethead()

与出队类似，但不需要移动队头指针 front。对应的算法如下：

```
def gethead(self):                          # 取队头元素
    assert not self.empty()                 # 检测队空
    head=(self.front+1)%MaxSize             # 求队头元素的位置
    return self.data[head]
```

上述 4 个算法的时间复杂度均为 $O(1)$。

3.2.3 循环队列的应用算法设计示例

扫一扫 视频讲解

【例 3.11】 在 CSqQueue 循环队列类中增加一个求元素个数的算法 size()。对于一个整数循环队列 qu，利用队列基本运算和 size()算法设计进队和出队第 k($k\geqslant 1$，队头元素的序号为 1)个元素的算法。

解 在前面的循环队列中，队头指针 front 指向队中队头元素的前一个位置，队尾指针 rear 指向队中的队尾元素，可以求出队中元素个数=(rear−front+MaxSize)%MaxSize。为此在 CSqQueue 循环队列类中增加 size()算法如下：

```
def size(self):                             # 返回队中元素个数
    return ((self.rear-self.front+MaxSize)%MaxSize)
```

在队列中并没有直接进队和出队第 k($k\geqslant 1$)个元素的基本运算，进队第 k 个元素 e 的算法思路是出队前 $k-1$ 个元素，边出边进，再将元素 e 进队，最后将剩下的元素边出边进。该算法如下：

```
def pushk(qu,k,e):                          # 进队第 k 个元素 e
    n=qu.size()
    if k<1 or k>n+1:
        return False                        # 参数 k 错误返回 False
    if k<=n:
        for i in range(1,n+1):              # 循环处理队中的所有元素
            if i==k:
                qu.push(e)                  # 将 e 元素进队到第 k 个位置
            x=qu.pop()                      # 出队元素 x
            qu.push(x)                      # 进队元素 x
    else: qu.push(e)                        # k=n+1 时直接进队 e
    return True
```

出队第 k($k\geqslant 1$)个元素 e 的算法思路是出队前 $k-1$ 个元素，边出边进，再出队第 k 个元素 e，e 不进队，最后将剩下的元素边出边进。该算法如下：

```
def popk(qu,k):                             # 出队第 k 个元素
    n=qu.size()
    assert k>=1 and k<=n                    # 检测参数 k 错误
    for i in range(1,n+1):                  # 循环处理队中的所有元素
        x=qu.pop()                          # 出队元素 x
        if i!=k: qu.push(x)                 # 将非第 k 个元素进队
        else: e=x                           # 取第 k 个出队的元素
    return e
```

扫一扫 视频讲解

【例 3.12】 对于循环队列来说，如果知道队头指针和队中元素个数，则可以计算出队尾指针。也就是说，可以用队中元素个数代替队尾指针。设计出这种循环队列的判队空、进队、出队和取队头元素的算法。

解 本例的循环队列包含 data 列表、队头指针 front 和队中元素个数 count，可以由 front 和 count 求出队尾位置，公式如下。

```
rear1=(self.front+self.count)%MaxSize
```

初始时 front 和 count 均置为 0。队空条件为 count==0；队满条件为 count==MaxSize；元素 e 进队操作是先根据上述公式求出队尾指针 rear1，将 rear1 循环增 1，然后将元素 e 放置在 rear1 处；出队操作是先将队头指针循环增 1，然后取出该位置的元素。设计本例的循环队列类 CSqQueue1 如下：

```
MaxSize=100                                  #全局变量,假设容量为 100
class CSqQueue1:                             #本例的循环队列类
    def __init__(self):                      #构造方法
        self.data=[None]*MaxSize             #存放队列中的元素
        self.front=0                         #队头指针
        self.count=0                         #队中元素个数
    def empty(self):                         #判断队列是否为空
        return self.count==0
    def push(self,e):                        #元素 e 进队
        rear1=(self.front+self.count)%MaxSize
        assert self.count!=MaxSize           #检测队满
        rear1=(rear1+1) % MaxSize
        self.data[rear1]=e
        self.count+=1                        #元素个数增 1
    def pop(self):                           #出队元素
        assert not self.empty()              #检测队空
        self.count-=1                        #元素个数减 1
        self.front=(self.front+1)%MaxSize    #队头指针循环增 1
        return self.data[self.front]
    def gethead(self):                       #取队头元素
        assert not self.empty()              #检测队空
        head=(self.front+1)%MaxSize          #求队头元素的位置
        return self.data[head]
```

说明：本例设计的循环队列中最多可保存 MaxSize 个元素。

从上述循环队列的设计看出，如果将 data 数组的容量改为可以扩展的，在队满时新建更大容量的数组 newdata 后，不能像顺序表、顺序栈那样简单地将 data 中的元素复制到 newdata 中，需要按队列操作，将 data 中的所有元素出队后进队到 newdata 中，这里不再详述。

扫一扫

视频讲解

3.2.4 队列的链式存储结构及其基本运算算法的实现

队列的链式存储结构也是通过由结点构成的单链表实现的，此时只允许在单链表的表首进行删除操作（出队）和在单链表的表尾进行插入操作（进队），这里的单链表是不带头结点的，需要使用两个指针（即队首指针 front 和队尾指针 rear）来标识，front 指向队首结点，rear 指向队尾结点。用于存储队列的单链表简称为**链队**。

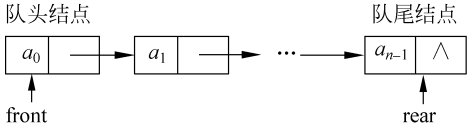


图 3.29 链队的存储结构示意图

链队的存储结构如图 3.29 所示，链队中存放元素的结点类 LinkNode 定义如下：

```
class LinkNode:                          # 链队结点类
    def __init__(self,data=None):        # 构造方法
        self.data=data                   # data 属性
        self.next=None                   # next 属性
```

设计链队类 LinkQueue 如下(用 LinkQueue.py 文件存放):

```
class LinkQueue:                         # 链队类
    def __init__(self):                  # 构造方法
        self.front=None                  # 队头指针
        self.rear=None                   # 队尾指针
    # 队列的基本运算算法
```

图 3.30 说明了一个链队的动态变化过程。图 3.30(a)是链队的初始状态,图 3.30(b)是进队 3 个元素后的状态,图 3.30(c)是出队两个元素后的状态。

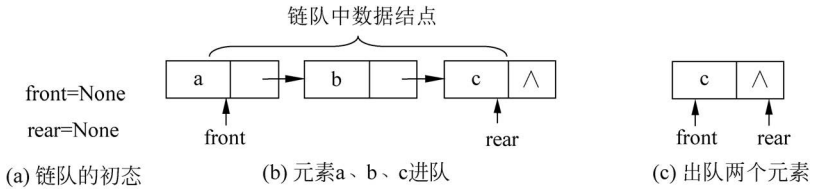


图 3.30 一个链队的动态变化过程

从图 3.30 中看到,初始时置 front=rear=None。链队的四要素如下。

① 队空条件:front=rear==None,不妨仅以 front==None 作为队空条件。

② 队满条件:由于只有在内存溢出时才出现队满,通常不考虑这样的情况。

③ 元素 e 进队操作:在单链表的尾部插入一个存放 e 的 s 结点,并让队尾指针指向它。

④ 出队操作:取出队首结点的 data 值并将其从链队中删除。

对应队列的基本运算算法如下。

1) 判断队列是否为空:empty()

链队的 front 为空表示队列为空,返回 True,否则返回 False。对应的算法如下:

```
def empty(self):                         # 判断队列是否为空
    return self.front==None
```

2) 进队:push(e)

创建存放元素 e 的结点 s。若原队列为空,则将 front 和 rear 均指向 s 结点,否则将 s 结点链接到单链表的末尾,并让 rear 指向它。对应的算法如下:

```
def push(self,e):                        # 元素 e 进队
    s=LinkNode(e)                        # 新建结点 s
    if self.empty():                     # 原链队为空
        self.front=self.rear=s
    else:                                # 原链队不为空
        self.rear.next=s                 # 将 s 结点链接到 rear 结点的后面
        self.rear=s
```

3) 出队:pop()

若原队为空,抛出异常;若队中只有一个结点(此时 front 和 rear 都指向该结点),取首

结点的 data 值赋给 e，并删除它，即置 front＝rear＝None；否则说明链队中有多个结点，取首结点的 data 值赋给 e，并删除它。最后返回 e。对应的算法如下：

```
def pop(self):                          #出队操作
    assert not self.empty()             #检测队空
    if self.front==self.rear:           #原链队只有一个结点
        e=self.front.data               #取首结点值
        self.front=self.rear=None       #置为空队
    else:                               #原链队有多个结点
        e=self.front.data               #取首结点值
        self.front=self.front.next      #front 指向下一个结点
    return e
```

4）取队头元素：gethead()

与出队类似，但不需要删除首结点。对应的算法如下：

```
def gethead(self):                      #取队头元素
    assert not self.empty()             #检测队空
    e=self.front.data                   #取首结点值
    return e
```

上述 4 个算法的时间复杂度均为 $O(1)$。

扫一扫

视频讲解

3.2.5 链队的应用算法设计示例

【例 3.13】 采用链队求解第 2 章例 2.16 的约瑟夫问题。

解 先定义一个链队 qu，对于(n，m)约瑟夫问题，依次将 1～n 进队。循环 n 次出列 n 个小孩：每次循环先出队 $m-1$ 次，将所有出队的元素立即进队（将他们从队头出队后插入队尾），再出队第 m 个元素并且输出（出列第 m 个小孩）。

对应的程序如下：

```
from LinkQueue import LinkQueue
def Jsequence(n,m):                     #求约瑟夫序列
    qu=LinkQueue()                      #定义一个链队
    for i in range(1,n+1):              #进队编号为 1～n 的 n 个小孩
        qu.push(i)
    for i in range(1,n+1):              #共出列 n 个小孩
        j=1
        while j<=m-1:                   #出队 m-1 个小孩，并将他们进队
            qu.push(qu.pop())
            j+=1
        x=qu.pop()                      #出队第 m 个小孩
        print(x,end=' ')
    print()

#主程序
print()
print("测试 1：n=6,m=3")
print("出列顺序：",end=' ')
Jsequence(6,3)
print("测试 2：n=8,m=4")
print("出列顺序：",end=' ')
Jsequence(8,4)
```

上述程序的执行结果如下：

```
测试 1：n=6,m=3
出列顺序：3 6 4 2 5 1
测试 2：n=8,m=4
出列顺序：4 8 5 2 1 3 7 6
```

说明：与第 2 章例 2.16 相比，这里相当于用带首尾结点指针的链队替代了循环单链表。

扫一扫

视频讲解

3.2.6 Python 中的双端队列

双端队列是在队列的基础上扩展而来的，其示意图如图 3.31 所示。双端队列与队列一样，元素的逻辑关系也是线性关系，但队列只能在一端进队，在另外一端出队，而双端队列可以在两端进行进队和出队操作，具有队列和栈的特性，因此使用更加灵活。

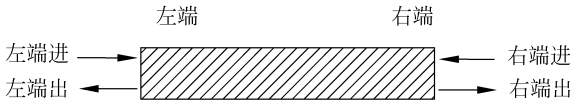


图 3.31 双端队列示意图

Python 提供了一个集合模块 collections，里面封装了多个集合类，其中包括 deque，即双端队列(double-ended queue)。

1. 创建双端队列

创建一个双端队列的基本方法如下。

1）创建一个空双端队列

使用的语法格式如下：

```
qu=deque()
```

此时 qu 为空，它是一个可以动态扩展的双端队列。

2）创建一个固定长度的双端队列

使用的语法格式如下：

```
qu=deque(maxlen=N)
```

此时 qu 为空，但固定长度为 N，当有新的元素加入而双端队列已满时会自动移除最老的那个元素。

3）由一个列表元素创建一个双端队列

使用的语法格式如下：

```
qu=deque(L)
```

此时 qu 包含列表 L 中的元素。

2. 双端队列的函数

deque 没有提供判空方法，可以使用内置函数 len()求其中的元素个数，通过 len(qu)==0 或者 notqu 判断双端队列是否为空，其时间复杂度为 $O(1)$。

3. 双端队列的方法

deque 提供的主要方法如下。

① deque.clear()：清除双端队列中的所有元素。

② deque.append(x)：在双端队列的右端添加元素 x，时间复杂度为 $O(1)$。

③ deque.appendleft(x)：在双端队列的左端添加元素 x，时间复杂度为 $O(1)$。

④ deque.pop()：在双端队列的右端出队一个元素，时间复杂度为 $O(1)$。

⑤ deque.popleft()：在双端队列的左端出队一个元素，时间复杂度为 $O(1)$。

⑥ deque.remove(x)：在双端队列中删除首个和 x 匹配的元素（从左端开始匹配的），如果没有找到抛出异常，其时间复杂度为 $O(n)$。

⑦ deque.count(x)：计算双端队列中元素为 x 的个数，时间复杂度为 $O(n)$。

⑧ deque.extend(L)：在双端队列的右端添加列表 L 的元素。例如，qu 为空，L=[1,2,3]，执行后 qu 从左向右为[1,2,3]。

⑨ deque.extendleft(L)：在双端队列的左端添加列表 L 的元素。例如，qu 为空，L=[1,2,3]，执行后 qu 从左向右为[3,2,1]。

⑩ deque.reverse()：把双端队列里的所有元素中的逆置。

⑪ deque.rotate(n)：双端队列的移位操作，如果 n 是正数，则队列中的所有元素向右移动 n 个位置；如果是负数，则队列中的所有元素向左移动 n 个位置。

4. 用双端队列实现栈

栈只在一端进行进栈和出栈操作，若定义 st=deque()，也就是用 deque 实现栈，其两种方式如下：

① 以左端作为栈底（左端保持不动），右端作为栈顶（右端动态变化，st[-1]为栈顶元素），栈操作在右端进行，则用 append()作为进栈方法，pop()作为出栈方法；

② 以右端作为栈底（右端保持不动），左端作为栈顶（左端动态变化，st[0]为栈顶元素），栈操作在左端进行，则用 appendleft()作为进栈方法，popleft()作为出栈方法。

例如，以下程序为采用方法①将 deque 作为栈的使用方法。

```
from collections import deque                    #引用 deque
st=deque()
st.append(1)
st.append(2)
st.append(3)
while len(st)>0:
    print(st.pop(),end=' ')                      #输出：3 2 1
print()
```

5. 用双端队列实现普通队列

普通队列只在一端进行进队操作，在另外一端进行出队操作，若定义 qu=deque()，也就是用 deque 实现队列，其两种方式如下：

① 以左端作为队头（出队端），右端作为队尾（进队端），则用 popleft()作为出队方法，append()作为进队方法。在队列非空时 qu[0]为队头元素，qu[-1]为队尾元素；

② 以右端作为队头（出队端），左端作为队尾（进队端），则用 pop()作为出队方法，

appendleft()作为进队方法。在队列非空时 qu[-1]为队头元素,qu[0]为队尾元素。

例如,以下程序为采用方法①将 deque 作为普通队列的使用方法。

```
from collections import deque
qu=deque()
qu.append(1)
qu.append(2)
qu.append(3)
while len(qu)>0:
    print(qu.popleft(),end=' ')                    #输出: 1 2 3
print()
```

扫一扫

视频讲解

3.2.7 队列的综合应用

本节通过用队列求解迷宫问题来讨论队列的应用。

❑ 问题描述

参见 3.1.6 节。

❑ 迷宫的数据组织

参见 3.1.6 节。

❑ 设计运算算法

用队列求迷宫路径的思路是从入口开始试探,当试探到一个方块 b 时,若为出口,输出对应的迷宫路径并返回,否则找其所有相邻可走方块,假设找到的顺序为 $b_1,b_2,\cdots,b_k$,称 b 方块为这些方块的前趋方块,称这些方块为 b 方块的后继方块,然后按 $b_1,b_2,\cdots,b_k$ 的顺序试探每一个方块。为此用一个队列存放这些方块,这里采用 deque 实现队列,定义如下:

```
qu=deque()                                         #定义一个队列
```

当找到出口后如何求出迷宫路径呢?由于每个试探的方块可能有多个后继方块,但一定只有唯一的前趋方块(除了入口方块外),为此设置队列中的元素类型如下:

```
class Box:                                         #方块类
    def __init__(self,i1,j1):                      #构造方法
        self.i=i1                                  #方块的行号
        self.j=j1                                  #方块的列号
        self.pre=None                              #前趋方块
```

假设当前试探的方块位置是(i,j),对应的队列元素(Box 对象)为 b,如图 3.32 所示,一次性找它所有的相邻可走方块,假设有 4 个相邻可走方块(实际上最多 3 个),则这 4 个相邻可走的方块均进队,同时置每个 b_i 的 pre 属性(即前趋方块)为 b,即 $b_i.\text{pre}=b$。所以找到出口后,从出口通过 pre 属性回退到入口即找到一条迷宫路径。

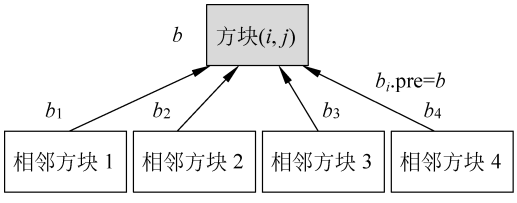


图 3.32 当前方块 b 和相邻方块

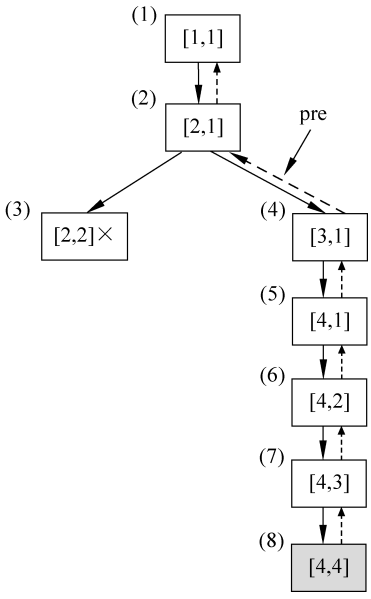


图 3.33　用队列求从(1,1)到(4,4)迷宫路径的搜索过程

查找一条从(xi,yi)到(xe,ye)的迷宫路径的过程是首先建立入口方块(xi,yi)的 Box 对象 b，将 b 进队，在队列 qu 不为空时循环：出队一次，称该出队的方块 b 为当前方块，做如下处理。

① 如果 b 是出口，则从 b 出发沿着 pre 属性回退到出口，找到一条迷宫逆路径 path，反向输出该路径后返回 True。

② 否则，按顺时针方向一次性查找方块 b 的 4 个方位中的相邻可走方块，每个相邻可走方块均建立一个 Box 对象 $b1$，置 b1.pre=b，将 $b1$ 进 qu 队列。与用栈求解一样，一个方块进队后，将其迷宫值置为−1，以避免回过来重复搜索。

如果队空都没有找到出口，表示不存在迷宫路径，返回 False。

在图 3.17 所示的迷宫图中求从入口(1,1)到出口(4,4)迷宫路径的搜索过程如图 3.33 所示，图中带“×”的方块表示没有相邻可走方块，每个方块旁的数字表示搜索顺序，找到出口后，通过虚线(即 pre)找到一条迷宫逆路径。

用队列求解迷宫问题的 mgpath()算法如下：

```
def mgpath(xi,yi,xe,ye):                    #求(xi,yi)到(xe,ye)的一条迷宫路径
    global mg                               #迷宫数组为全局变量
    dx=[-1,0,1,0]                           #x方向的偏移量
    dy=[0,1,0,-1]                           #y方向的偏移量
    qu=deque()                              #定义一个队列
    b=Box(xi,yi)                            #建立入口结点 b
    qu.appendleft(b)                        #结点 b 进队
    mg[xi][yi]=-1                           #进队方块置为-1
    while len(qu)!=0:                       #队不空时循环
        b=qu.pop()                          #出队一个方块 b
        if b.i==xe and b.j==ye:             #找到了出口,输出路径
            p=b                             #从 b 出发回推导出迷宫路径并输出
            path=[]                         #path 存放逆路径
            while p!=None:                  #找到入口为止
                path.append("["+str(p.i)+","+str(p.j)+"]")
                p=p.pre
            for i in range(len(path)-1,-1,-1):      #反向输出 path 得到正向路径
                print(path[i],end=' ')
            return True                     #找到一条路径时返回 True
        for di in range(4):                 #循环扫描每个相邻方位的方块
            i,j=b.i+dx[di],b.j+dy[di]       #找 b 的 di 方位的相邻方块(i,j)
            if mg[i][j]==0:                 #找相邻可走方块
                b1=Box(i,j)                 #建立后继方块结点 b1
                b1.pre=b                    #设置其前趋方块为 b
                qu.appendleft(b1)           #b1 进队
                mg[i][j]=-1                 #进队方块置为-1
    return False                            #未找到任何路径时返回 False
```

❑ 设计主程序

设计以下主程序用于求图 3.17 所示的迷宫图中从(1,1)到(4,4)的一条迷宫路径：

```
xi,yi=1,1
xe,ye=4,4
print("一条迷宫路径:",end=' ')
if not mgpath(xi,yi,xe,ye):                    #(1,1)->(4,4)
    print("不存在迷宫路径")
print()
```

❑ 程序执行结果

本程序的执行结果如下：

```
一条迷宫路径:[1,1] [2,1] [3,1] [4,1] [4,2] [4,3] [4,4]
```

该路径如图 3.34 所示，迷宫路径上方块的箭头表示其前趋方块的方位。显然这个解是最优解，也就是最短路径。至于为什么用栈求出的迷宫路径不一定是最短路径，而用队列求出的迷宫路径一定是最短路径，该问题将在第 7 章中说明。

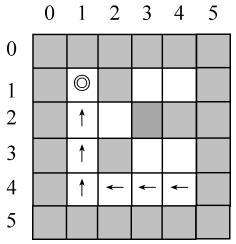


图 3.34 用队列求出的一条迷宫路径

扫一扫

视频讲解

3.2.8 优先队列

所谓优先队列，就是指定队列中元素的优先级，按优先级越大越优先出队，而普通队列中按进队的先后顺序出队，可以看成进队越早越优先。实际上优先队列就是第 9 章中讨论的堆，根按照大小分为大根堆和小根堆，大根堆的元素越大越优先出队(即元素越大优先级越大)，小根堆的元素越小越优先出队(即元素越小优先级越大)。

在 Python 中提供了 heapq 模块，其中包含的堆的基本操作方法用于创建堆，默认情况下创建小根堆。其主要方法如下。

① heapq.heapify(heap)：把列表 heap 调整为堆。

② heapq.heappush(heap,item)：向堆 heap 中插入元素 item(进队 item 元素)，该方法会维护堆的性质。

③ heapq.heappop(heap)：从堆 heap 中删除最小元素并且返回该元素值。

④ heapq.heapreplace(heap,item)：从堆 heap 中删除最小元素并且返回该元素值，同时将 item 插入并且维护堆的性质。它优于调用函数 heappop(heap)和 heappush(heap,item)。

⑤ heapq.heappushpop(heap, item)：把元素 item 插入堆 heap 中，然后从 heap 中删除最小元素并且返回该元素值。它优于调用函数 heappush(heap,item)和 heappop(heap)。

⑥ heapq.nlargest(*n*,iterable[,key])：返回迭代数据集合 iterable 中第 *n* 大的元素，可以指定比较的 key。它比通常计算多个列表中第 *n* 大的元素的方法更方便、快捷。

⑦ heapq.nsmallest(*n*,iterable[, key])：返回迭代数据集合 iterable 中第 *n* 小的元素，可以指定比较的 key。它比通常计算多个列表中第 *n* 小的元素的方法更方便、快捷。

⑧ heapq.merge(*iterables)：把多个堆合并，并返回一个迭代器。

使用 heapq 创建优先队列有两种方式，一种是使用一个空列表，然后使用 heapq.heappush()添加元素；另一种是使用 heap.heapify(heap)方法将 heap 列表转换成堆结构。

例如，定义一个 heap 列表，将其调整为小根堆，调用一系列 heapq 方法如下：

```
import heapq
heap=[6,5,4,1,8]                          #定义一个列表 heap
heapq.heapify(heap)                       #将 heap 列表调整为堆
print(heap)                               #输出：[1,5,4,6,8]
heapq.heappush(heap,3)                    #进队 3
print(heap)                               #输出：[1,5,3,6,8,4]
print(heapq.heappop(heap))                #输出：1
print(heap)                               #输出：[3,5,4,6,8]
print(heapq.heapreplace(heap,2))          #输出：3(出队最小元素，再插入 2)
print(heap)                               #输出：[2,5,4,6,8]
print(heapq.heappushpop(heap,1))          #输出：1(插入 1，再出队最小元素)
print(heap)                               #输出：[2,5,4,6,8]
```

由于 heapq 不支持大根堆，那么如何创建大根堆呢？对于数值类型，一个最大数的相反数就是最小数，可以通过对数值取反，仍然创建小根堆的方式来获取最大数。

例如，一个列表 list 的元素形如“[年龄，姓名]”，假设所有的年龄都不相同，需要求最大的年龄及对应的姓名。

将所有年龄取相反数，建立相应的小根堆，出队最小元素 s，其 $-s[0]$ 即为最大的年龄，$s[1]$ 为对应的姓名。对应的程序及其输出结果如下：

```
import heapq
list=[[20,"Mary"],[24,"John"],[21,"Smith"]]          #定义一个列表
heap=[]
for i in range(len(list)):
    heap.append([-list[i][0],list[i][1]])
print(heap)                          #输出：[[-20, 'Mary'], [-24, 'John'], [-21, 'Smith']]
heapq.heapify(heap)                  #将 heap 列表调整为堆
print(heap)                          #输出：[[-24, 'John'], [-20, 'Mary'], [-21, 'Smith']]
s=heapq.heappop(heap)                #出队最大者
print("年龄最大为%d 岁，是%s" %(-s[0],s[1]))   #输出：年龄最大为 24 岁，是 John
```

扫一扫

自测题

3.3 练 习 题

1. 简述线性表、栈和队列的异同。

2. 有 5 个元素，其进栈次序为 $abcde$，在各种可能的出栈次序中，以元素 c、d 最先出栈（即 c 第一个且 d 第二个出栈）的次序有哪几个？

3. 假设以 I 和 O 分别表示进栈和出栈操作，则初态和终态为栈空的进栈和出栈的操作序列可以表示为仅由 I 和 O 组成的序列，称可以实现的栈操作序列为合法序列（例如 IIOO 为合法序列，IOOI 为非法序列）。试给出区分给定序列为合法序列或非法序列的一般准则。

4. 什么叫“假溢出”？如何解决假溢出？

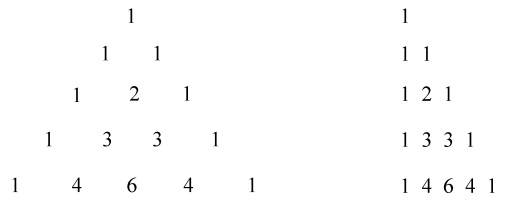
5. 假设循环队列的元素存储空间为 data[0..$m-1$]，队头指针 f 指向队头元素，队尾指针 r 指向队尾元素的下一个位置（例如 data[0..5]，队头元素为 data[2]，则 front=2，队尾元素为 data[3]，则 rear=4），则在少用一个元素空间的前提下表示队空和队满的条件各是什么？

6. 在算法设计中，有时需要保存一系列临时数据元素，如果先保存的后处理，应该采用什么数据结构存放这些元素？如果先保存的先处理，应该采用什么数据结构存放这些元素？

7. 给定一个字符串 str，设计一个算法采用顺序栈判断 str 是否为形如“序列 1@序列 2”的合法字符串，其中序列 2 是序列 1 的逆序，在 str 中恰好只有一个@字符。

8. 设计一个算法利用一个栈将一个循环队列中的所有元素倒过来，队头变队尾，队尾变队头。

9. 对于给定的正整数 $n(n>2)$，利用一个队列输出 n 阶杨辉三角形，5 阶杨辉三角形如图 3.35(a)所示，其输出结果如图 3.35(b)所示。



(a) n=5的杨辉三角形　　(b) 输出结果

图 3.35　5 阶杨辉三角形及其输出结果

10. 有一个整数数组 a，设计一个算法将所有偶数位元素移动到所有奇数位元素的前面，要求它们的相对次序不改变。例如，$a=\{1,2,3,4,5,6,7,8\}$，移动后 $a=\{2,4,6,8,1,3,5,7\}$。

11. 设计一个循环队列，用 data[0..MaxSize−1]存放队列元素，用 front 和 rear 分别作为队头和队尾指针，另外用一个标志 tag 标识队列可能空(False)或可能满(True)，这样加上 front==rear 可以作为队空或队满的条件，要求设计队列的相关基本运算算法。

12. 设计一个算法以 Python 中 deque 作为栈求一条迷宫路径。

13. 给定一个含 n 个整数的数组 a，设计一个算法利用优先队列求其中第 $k(1\leqslant k\leqslant n)$ 小元素(不是第 k 个不同的元素)。例如，$a=(1,3,2,2)$，$k=1$ 时答案为 1，$k=2$ 时答案为 2，$k=3$ 时答案为 2，$k=4$ 时答案为 3。

3.4 上机实验题

3.4.1 基础实验题

1. 设计整数顺序栈的基本运算程序，并用相关数据进行测试。
2. 设计整数链栈的基本运算程序，并用相关数据进行测试。
3. 设计整数循环队列的基本运算程序，并用相关数据进行测试。
4. 设计整数链队的基本运算程序，并用相关数据进行测试。

3.4.2 应用实验题

1. 一个 b 序列的长度为 n，其元素恰好是 1～n 的某个排列，编写一个实验程序判断 b 序列是否是以 1,2,…,n 为进栈序列的出栈序列，如果不是，输出相应的提示信息；如果是，

输出由该进栈序列通过一个栈得到 b 序列的过程。

2. 改进用栈求解迷宫问题的算法，累计如图 3.17 所示的迷宫的路径条数，并输出所有迷宫路径。

3. 括号匹配问题：在某个字符串（长度不超过 100）中有左括号、右括号和大/小写字母，规定（与常见的算术表达式一样）任何一个左括号都从内到外与它右边距离最近的右括号匹配。编写一个实验程序，找到无法匹配的左括号和右括号，输出原来的字符串，并在下一行标出不能匹配的括号，不能匹配的左括号用“$”标注，不能匹配的右括号用“?”标注。例如，输出样例如下：

```
( (ABCD(x)
$$
)(rttyy())sss)(
?             ?$
```

4. 修改《教程》3.2 节中的循环队列算法，使其容量可以动态扩展，当进队时，若元素当前容量满时按两倍扩大容量；当出队时，若当前容量大于初始容量并且元素的个数只有当前容量的 1/4，缩小当前容量为一半。通过测试数据说明队列容量变化的情况。

5. 采用不带头结点只有一个尾结点指针 rear 的循环单链表存储队列，设计出这种链队的进队、出队、判队空和求队中元素个数的算法。

6. 对于如图 3.36 所示的迷宫图，编写一个实验程序，先采用队列求一条最短迷宫路径长度 minlen（路径中经过的方块个数），再采用栈求所有长度为 minlen 的最短迷宫路径。在搜索所有路径时进行这样的优化操作：当前路径尚未到达出口但长度超过 minlen，便结束该路径的搜索。

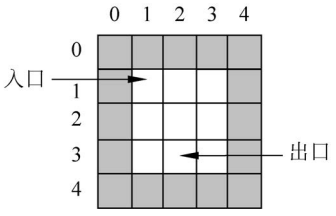


图 3.36 一个迷宫的示意图

3.5 LeetCode 在线编程题

扫一扫 视频讲解

1. LeetCode20——有效的括号

问题描述：给定一个只包括 '('、')'、'{'、'}'、'['、']' 的字符串，判断字符串是否有效。有效字符串需满足左括号必须用相同类型的右括号闭合，左括号必须以正确的顺序闭合。注意空字符串可被认为是有效字符串。例如，输入字符串"()"，输出为 True；输入字符串"([)]"，输出为 False。要求设计满足题目条件的如下方法：

```
defisValid(self,s: str) -> bool:
```

扫一扫 视频讲解

2. LeetCode150——逆波兰表达式求值

问题描述：根据逆波兰表示法求表达式的值，有效的运算符包括 +、-、*、/。每个运算对象可以是整数，也可以是另一个逆波兰表达式。假设给定的逆波兰表达式总是有效的，即表达式总会得出有效数值且不存在除数为 0 的情况。其中整数除法只保留整数部分。例如，输入["2","1","+","3","*"]，输出结果为 9；输入["4","13","5","/","+"]，输出结

果为 6。要求设计满足题目条件的如下方法：

```
def evalRPN(self,tokens: List[str]) -> int:
```

3. LeetCode71——简化路径

扫一扫 视频讲解

问题描述：以 UNIX 风格给出一个文件的绝对路径，并且简化它，也就是说将其转换为规范路径。在 UNIX 风格的文件系统中，一个点(.)表示当前目录本身，两个点(..)表示将目录切换到上一级(指向父目录)，两者都可以是复杂相对路径的组成部分。注意，返回的规范路径必须始终以斜杠(/)开头，并且两个目录名之间只有一个斜杠/，最后一个目录名(如果存在)不能以/结尾。此外，规范路径必须是表示绝对路径的最短字符串。例如，输入字符串"/home/"，输出结果为"/home"，输入字符串"/a//b////c/d//./../..."，输出结果为"/a/b/c"。要求设计满足题目条件的如下方法：

```
def simplifyPath(self,path: str) -> str:
```

4. LeetCode51——n 皇后

扫一扫 视频讲解

问题描述：n 皇后问题研究的是如何将 n 个皇后放置在 $n\times n$ 的棋盘上，并且使皇后彼此之间不能相互攻击。给定一个整数 n，返回所有不同的 n 皇后问题的解决方案。每一种解法包含一个明确的 n 皇后问题的棋子放置方案，在该方案中'Q'和'.'分别代表了皇后和空位。例如输入 4，输出(共两个解法)结果如下：

```
[
 [".Q..",                                    #解法 1
  "...Q",
  "Q...",
  "..Q."],
 ["..Q.",                                    #解法 2
  "Q...",
  "...Q",
  ".Q.."]
]
```

要求设计满足题目条件的如下方法：

```
def solveNQueens(self,n: int) -> List[List[str]]:
```

5. LeetCode622——设计循环队列

扫一扫 视频讲解

问题描述：循环队列是一种线性数据结构，其操作表现基于 FIFO(先进先出)原则，并且队尾被连接在队首之后以形成一个循环，它也被称为“环形缓冲器”。循环队列的一个好处是用户可以利用这个队列之前用过的空间。在一个普通队列里，一旦一个队列满了就不能插入下一个元素，即使在队列的前面仍有空间，但是在使用循环队列时可以使用这些空间去存储新的值。设计应该支持如下操作。

① MyCircularQueue(k)：构造器，设置队列长度为 k。

② Front()：从队首获取元素。如果队列为空，则返回－1。

③ Rear()：获取队尾元素。如果队列为空，则返回－1。

④ enQueue(value)：向循环队列插入一个元素。如果成功插入，则返回 True。

⑤ deQueue()：从循环队列中删除一个元素。如果成功删除，则返回 True。

⑥ isEmpty()：检查循环队列是否为空。

⑦ isFull()：检查循环队列是否已满。

例如：

```
MyCircularQueue circularQueue=new MycircularQueue(3)        #设置长度为 3
circularQueue.enQueue(1)                                    #返回 True
circularQueue.enQueue(2)                                    #返回 True
circularQueue.enQueue(3)                                    #返回 True
circularQueue.enQueue(4)                                    #返回 False,队列已满
circularQueue.Rear()                                        #返回 3
circularQueue.isFull()                                      #返回 True
circularQueue.deQueue()                                     #返回 True
circularQueue.enQueue(4)                                    #返回 True
circularQueue.Rear()                                        #返回 4
```

提示：所有的值都在 0～1000 的范围内，操作数将在 1～1000 的范围内，不要使用内置的队列库。

扫一扫 视频讲解

6. LeetCode119——杨辉三角 Ⅱ

问题描述：给定一个非负索引 k，其中 $k \leqslant 33$，返回杨辉三角的第 k 行。在杨辉三角中，每个数是它左上方和右上方的数的和。例如输入整数 3，输出为[1,3,3,1]。要求设计满足题目条件的如下方法：

```
def getRow(self, rowIndex: int) -> List[int]:
```

扫一扫 视频讲解

7. LeetCode347——前 k 个高频元素

问题描述：给定一个非空的整数数组，返回其中出现频率前 k 个高的元素。例如，输入 nums=[1,1,1,2,2,3]，$k=2$，输出结果为[1,2]；输入 nums=[1]，$k=1$，输出结果为[1]。可以假设给定的 k 总是合理的，且 $1 \leqslant k \leqslant$ 数组中不相同的元素的个数。另外算法的时间复杂度必须优于 $O(n\log_2 n)$，n 是数组的大小。要求设计满足题目条件的如下方法：

```
def topKFrequent(self, nums: List[int], k: int) -> List[int]:
```

扫一扫 视频讲解

8. LeetCode23——合并 k 个排序链表

问题描述：合并 k 个排序链表，返回合并后的排序链表。分析和描述算法的复杂度。例如，输入如下：

```
[
  1-> 4-> 5,
  1-> 3-> 4,
  2-> 6
]
```

输出的链表为 1-> 1-> 2-> 3-> 4-> 4-> 5-> 6。要求设计满足题目条件的如下方法：

```
def mergeKLists(self, lists: List[ListNode]) -> ListNode:
```