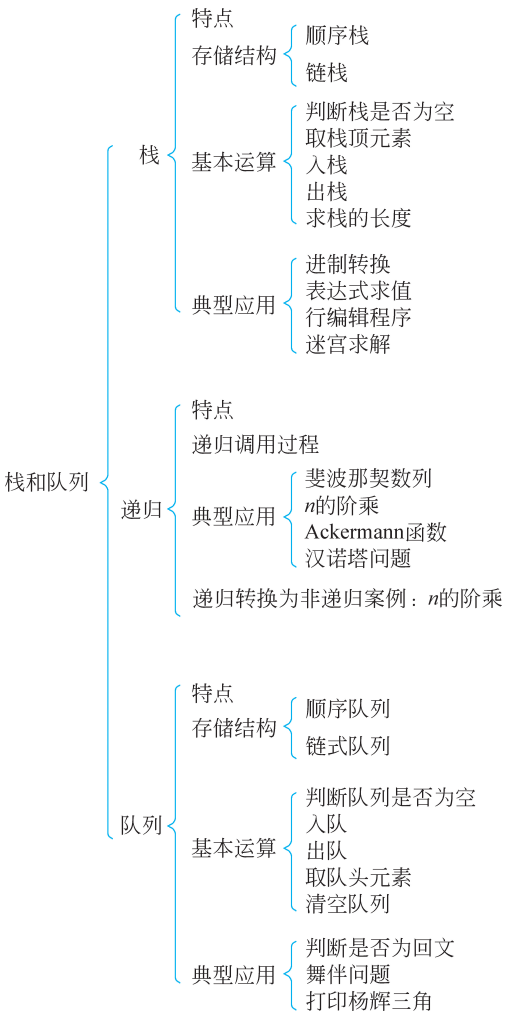
第3章 栈和队列

栈和队列是两种特殊的线性结构，从元素之间的逻辑关系上看，它们都属于线性结构，其特殊性在于插入和删除操作被限制在表的一端进行，因此栈和队列被称为操作受限的线性表。在实际生活中，栈和队列被广泛应用于软件开发过程。本章主要介绍栈和队列的基本概念、存储结构、基本运算及典型应用。

本章重难点：



3.1　栈

栈是一种只能在表的一端进行插入和删除操作的线性表。栈的应用非常广泛，例如，算术表达式求值、括号匹配、递归算法就是利用栈的设计思想求解问题的。本节主要介绍栈的定义、栈的存储结构及应用。

3.1.1　栈的基本概念

栈(stack)，也称为堆栈，它是限定仅在表尾进行插入和删除操作的线性表。允许插入、删除操作的一端称为**栈顶**(top)，另一端称为**栈底**(bottom)。栈顶是动态变化的，通常由一个称为栈顶指针(top)的变量指示。当表中没有元素时，称为**空栈**。

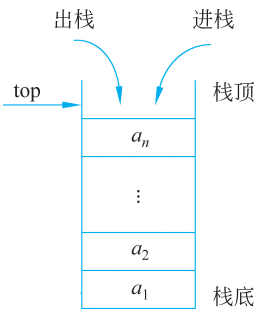


图 3-1　栈

栈的插入操作称为入栈或进栈，删除操作称为出栈或退栈。

将元素序列 $a_1,a_2,\cdots,a_n$ 依次进栈后，a_1 为栈底元素，a_n 为栈顶元素，如图 3-1 所示。最先入栈的元素位于栈底，最后入栈的元素成为栈顶元素，每次出栈的元素也是栈顶元素。因此，**栈是一种后进先出(Last In First Out，LIFO)的线性表。**

若将元素 a、b、c 和 d 依次入栈，最后将栈顶元素出栈，栈顶指针 top 的变化情况如图 3-2 所示。

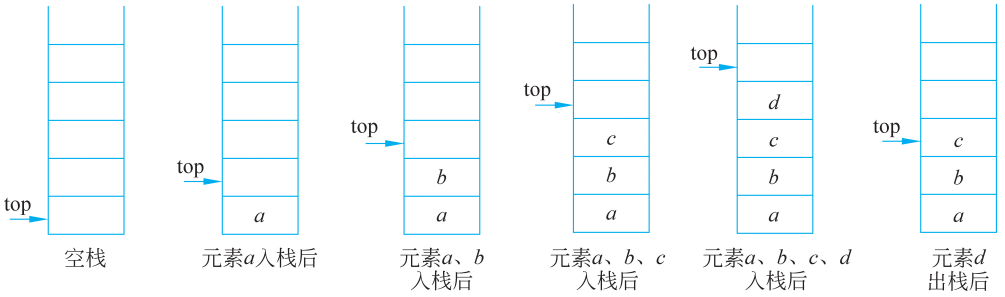


图 3-2　入栈和出栈时栈顶指针的变化情况

【例 3-1】　若 abc 为一个入栈序列，给出不可能的出栈序列。

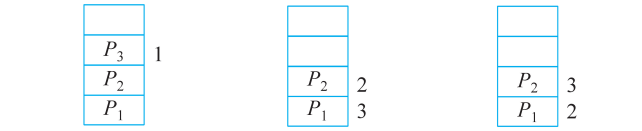
【分析】　根据栈的后进先出特性，出栈序列有 5 种可能：abc、acb、bac、bca 和 cba，而 cab 是不可能的出栈序列。这是因为 c 最先出栈，说明 c 位于栈顶，且 a 和 b 已经进栈并且还未出栈，根据入栈顺序和栈的性质，b 一定在 a 的上面，因此出栈序列一定是 cba，不可能是 cab。

【例 3-2】　一个栈的输入序列为 P_1、P_2、P_3，输出序列为 1、2、3，如果 $P_3=1$，则 P_1 的值(　　)。

A. 可能是 2　　B. 一定是 2　　C. 不可能是 2　　D. 不可能是 3

【分析】　因为 $P_3=1$ 且 1 是第一个出栈的元素，说明栈中还有 P_2、P_1 两个元素，其中 P_2 为新的栈顶元素，P_1 位于栈底，若 $P_2=2$、$P_1=1$，则出栈序列为 1、2、3；若 $P_2=3$、$P_1=$

2,则出栈序列为 1、3、2,因此 P_1 的值可能是 3,一定不能是 2,故选项 C 是正确的。P_1、P_2 和 P_3 在栈中的情况如图 3-3 所示。



已知入栈序列和出栈序列,且P_3=1　　若P_2=2、P_1=3,则输出序列为1、2、3　　若P_2=2、P_1=3,则输出序列为1、3、2

图 3-3　P_1、P_2 和 P_3 在栈中的情况

3.1.2　栈的抽象数据类型

栈的抽象数据类型描述如表 3-1 所示。

表 3-1　栈的抽象数据类型描述

数据对象	栈的数据对象集合为$\{a_1,a_2,\cdots,a_n\}$,每个元素都有相同的类型	
数据关系	栈中数据元素之间的关系具有线性表的特点:除了第一个元素 a_1 外,每一个元素有且只有一个直接前驱元素;除了最后一个元素 a_n 外,每一个元素有且只有一个直接后继元素	
基本操作	InitStack(&S)	初始化操作,建立一个空栈 S
	StackEmpty(S)	判断栈是否为空。若栈 S 为空,返回 True;否则返回 False
	GetTop(S,&e)	返回栈 S 的栈顶元素给 e
	PushStack(&S,e)	在栈 S 中插入元素 e,使其成为新的栈顶元素
	PopStack(&S,&e)	删除栈 S 的栈顶元素,并用 e 返回其值
	StackLength(S)	返回栈 S 的元素个数
	ClearStack(S)	清空栈 S

3.1.3　栈的顺序表示与实现

栈有两种存储结构:顺序存储和链式存储。本节主要介绍栈的顺序存储结构及基本操作的实现。

1. 顺序栈的类型定义

采用顺序存储结构的栈称为**顺序栈**。顺序栈利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素,可利用 Python 语言中的列表作为顺序栈的存储结构,同时附设一个栈顶指针 top,用于指向顺序栈的栈顶元素。当 top=0 时表示空栈。栈的顺序存储结构类型可在类中进行定义,具体由栈的初始化实现。

栈的顺序存储结构类型及基本运算通过自定义类实现,顺序栈的类名定义为 SeqStack,顺序栈的存储结构通过 SeqStack 的构造函数__init__(self)描述如下:

```
def __init__(self):
    self.top=0
    self.MAXSIZE=50
    self.stack = [None for x in range(0,self.MAXSIZE)]
```

其中,stack 用于存储栈中的数据元素的列表,top 为栈顶指针,MAXSIZE 为栈的最大容量。

当栈中元素个数为 MAXSIZE 时，称为栈满。如果继续入栈操作则会产生溢出，称为上溢。对空栈进行删除操作，就会产生下溢。

顺序栈的结构如图 3-4 所示。元素 a、b、c、d、e、f、g、h 依次入栈后，a 为栈底元素，h 为栈顶元素。在实际操作中，栈顶指针指向栈顶元素的下一个位置。

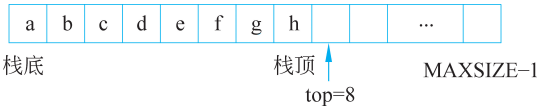


图 3-4　顺序栈的结构

入栈操作时，先判断栈是否已满，若未满，将元素压入栈中，即 S.stack[S.top]＝e，然后使栈顶指针加 1，即 S.top＋＝1。出栈操作时，先判断栈是否为空，若为空，使栈顶指针减 1，即 S.top－＝1，然后元素出栈，即 e＝S.stack[S.top]。判断顺序栈为空的条件为 self.top＝＝0，判断顺序栈已满的条件为 self.top＝self.MAXSIZE。

2. 顺序栈的基本操作

顺序栈的基本操作及类方法如表 3-2 所示。

表 3-2　顺序栈的基本操作及类方法

基本操作	类方法	基本操作	类方法
栈的初始化	__init__(self)	取栈顶元素	getTop(self)
判断栈是否为空	stackEmpty(self)	求栈的长度	stackLength(self)
入栈	pushStack(self,e)	清空栈	clearStack(self)
出栈	popStack(self)		

（1）初始化栈。

```
def init_(self):
        self.top=0
        self.MAXSIZE=50
        self.stack = [None for x in range(0,self.MAXSIZE)]
```

（2）判断栈是否为空。

```
def stackEmpty(self):
    if self.top==0:
        return True
    else:
        return False
```

（3）取栈顶元素。在取栈顶元素前，先判断栈是否为空。如果栈为空，则返回 None 表示取栈顶元素失败；否则，将栈顶元素返回。取栈顶元素的算法实现如下：

```
def getTop(self):
    if self.StackEmpty():
        print("栈为空,取栈顶元素失败!")
        return None
    else:
        return self.stack[self.top-1]
```

（4）将元素 e 入栈。在将元素 e 入栈前，需要先判断栈是否已满。若栈已满，返回 False

表示入栈操作失败;否则将元素 e 压入栈中,然后将栈顶指针 top 增 1,并返回 True 表示入栈操作成功。入栈操作的算法实现如下:

```
def pushStack(self,e):
    if self.top>=self.MAXSIZE:
        print("栈已满!")
        return False
    else:
        self.stack[self.top]=e
        self.top=self.top+1
        return True
```

(5) 将栈顶元素出栈。在将元素出栈前,需要先判断栈是否为空。若栈为空,则返回 None;若栈不为空,则先使栈顶指针减 1,然后将栈顶元素返回。出栈操作的算法实现如下:

```
def popStack(self):
    if self.StackEmpty():
        print("栈为空,不能进行出栈操作!")
        return None
    else:
        self.top=self.top-1
        x=self.stack[self.top]
        return x
```

(6) 求栈的长度。

```
def stackLength(self):
    return self.top
```

(7) 清空栈。

```
def clearStack(self):
    self.top=0
```

3. 顺序栈应用实例

【例 3-3】 任意给定一个数学表达式,如{5*(9-2)-[15-(8-3)/2]}+3*(6-4),设计一个算法判断表达式的括号是否匹配。

【分析】 为了检验括号是否匹配,可以设置一个栈。依次读入一个字符。

(1) 如果读入的是左括号,则直接进栈。

(2) 如果读入的是右括号,则进行以下判断:

- 它与当前栈顶的左括号是同类型的,说明这一对括号是匹配的,则将栈顶的左括号出栈;否则是不匹配的,继续读下一字符。
- 如果栈已经为空,说明缺少左括号,该表达式的括号不匹配。

(3) 如果字符序列已经读完,而栈中仍然有等待匹配的左括号,说明缺少右括号,该表达式的括号不匹配。

(4) 如果读入的是数字字符或运算符,则不进行处理,继续读下一字符。

当字符序列和栈同时变为空时,说明括号完全匹配。

算法实现如下:

```
import SeqStack as sq
```

```
def Match(e,ch):
  #判断左右两个括号是否为同类型,同类型则返回 True,否则返回 False
  if(e=='(' and ch==')'):
      return True
  elif (e=='[' and ch==']'):
      return True
  elif(e=='{' and ch=='}'):
      return True
  else:
      return False
if __name__== '__main__':
  S = sq.MySeqStack()
  ch = input('请输入算术表达式')
  i=0
  while i in range(len(ch)):
      if ch[i]=='(' or ch[i]== '[' or ch[i] == '{':  #如果是左括号,入栈
        S.pushStack(ch[i])
        i=i+1
      elif ch[i]==')' or ch[i]== ']' or ch[i]== '}':
        if S.stackEmpty():
            print("缺少左括号.\n")
            exit(-1)
        else:
            e=S.getTop()
            if Match(e,ch[i]):   #如果栈顶的左括号与读入的右括号匹配,则将栈顶的左括
                                 #号出栈
              e=S.popStack()
              i=i+1
            else:                #否则
              print("左右括号不匹配.\n")
              exit(-1)
      else:                      #如果是其他字符,则不处理,直接将 p 指向下一个字符
        i=i+1
  if(S.stackEmpty()):            #如果字符序列读入完毕,且栈已空,说明括号序列匹配
      print("括号匹配.\n")
  else:                          #如果字符序列读入完毕,且栈不空,说明缺少右括号
      print("缺少右括号.\n")
```

程序的运行结果如图 3-5 所示。

```
C:\Users\o.o\.conda\envs\tensorflow\python.exe D:/Python程序/数据结构/括号匹配.py
请输入算术表达式[9-8]+[8-(9+2)]
括号匹配.
```

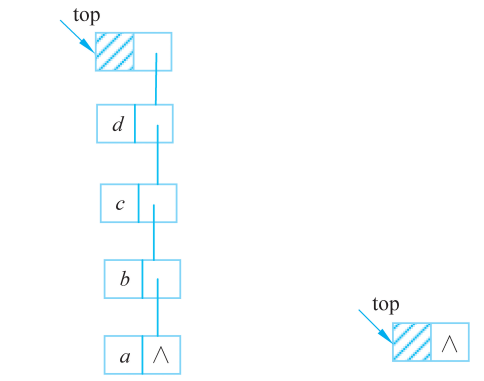
图 3-5　例 3-3 程序运行结果

3.1.4　栈的链式表示与实现

由于顺序栈采用顺序存储结构,需要事先静态分配存储空间,而存储规模往往又难以确定。如果栈空间分配过小,可能会造成溢出;如果栈空间分配过大,又造成存储空间浪费。因此,为了克服顺序栈的缺点,可采用链式存储结构表示栈。本节主要介绍栈的链式存储结构及基本操作。

1. 栈的存储结构

栈的链式存储结构用一组不一定连续的存储单元存放栈中的数据元素。一般来说,当栈中数据元素的数目变化较大或不确定时,采用链式存储结构是比较合适的。用链式存储结构表示的栈称为**链栈**或**链式栈**。



(a) 依次将a、b、c、d入栈后　　(b) 栈为空

图 3-6　带头结点的链栈

链栈通常用单链表表示。插入和删除操作都在栈顶指针的位置进行,这一端称为**栈顶**,通常由栈顶指针 **top** 指示。例如,元素 a、b、c、d 依次入栈的带头结点的链栈如图 3-6(a)所示。链栈为空时如图 3-6(b)所示。

栈顶指针 top 始终指向头结点,最先入栈的元素在链栈的栈底,最后入栈的元素成为栈顶元素。由于链栈的操作都是在链表的表头位置进行,因而链栈的基本操作(除了求链栈长度以外)的时间复杂度均为 $O(1)$。

链栈的结点类型描述如下:

```
class LinkStackNode:
    def __init__(self):
        self.data=None
        self.next=None
```

对于带头结点的链栈,初始化链栈时,有 top.next=None,判断栈空的条件为 top.next==None;对于不带头结点的链栈,初始化链栈时,有 top=None,判断栈空的条件为 top==None。

采用链式存储的栈不必事先估算栈的最大容量,只要系统有可用的存储空间,就能随时为结点申请存储空间,不用考虑栈满的情况。

2. 链栈的基本操作

链栈的基本操作通过 MyLinkStack 类实现,在该类中定义相关基本运算,结点的分配需要调用 LinkStackNode 类。链栈的基本操作及类方法如表 3-3 所示。

表 3-3　链栈的基本操作及类方法

基本操作	类方法
初始化链栈	__init__(self)
判断链栈是否为空	stackEmpty(self)
入栈	pushStack(self,e)
出栈	popStack(self)
取栈顶元素	getTop(self)
求链栈的长度	stackLength(self)
创建链栈	createStack(self)
清空链栈	clearStack(self)

（1）初始化链栈。初始化链栈由__init__(self)函数实现，需要调用链栈的结点类LinkStackNode初始化结点。初始时，头结点的指针域为空。初始化链栈的算法实现如下：

```
class MyLinkStack:
    def __init__(self):
        self.top=LinkStackNode()
```

（2）判断链栈是否为空。如果头结点指针域为空，说明链栈为空，返回 True；否则，返回 False。判断链栈是否为空的算法实现如下：

```
def stackEmpty(self):
    if self.top.next is None
        return True
    else:
        return False
```

（3）将元素 e 入栈。先生成一个结点，用 pnode 指向该结点，再将元素 e 的值赋给 pnode 结点的数据域，然后将新结点插入第一个结点之前。插入新结点的操作分为两个步骤：①pnode.next＝self.top.next；②self.top.next＝pnode。入栈操作如图 3-7 所示。

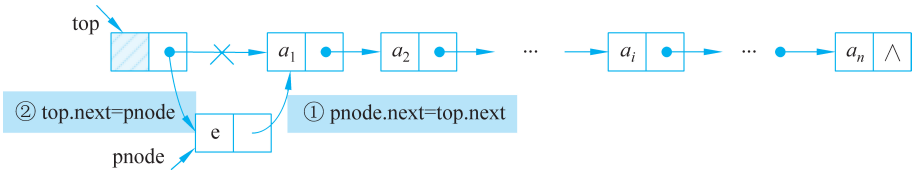


图 3-7　入栈操作

注意：在插入新结点时，需要注意修改指针的顺序不能颠倒。

将元素 e 入栈的算法实现如下：

```
def pushStack(self,e):
    pnode=LinkStackNode()
    pnode.data=e
    pnode.next=self.top.next
    self.top.next=pnode
```

（4）将栈顶元素出栈。先判断链栈是否为空。若链栈为空，返回 None，表示出栈操作失败；否则，将栈顶元素值赋给 x，释放该结点空间，最后将栈顶元素返回。出栈操作如图 3-8 所示。

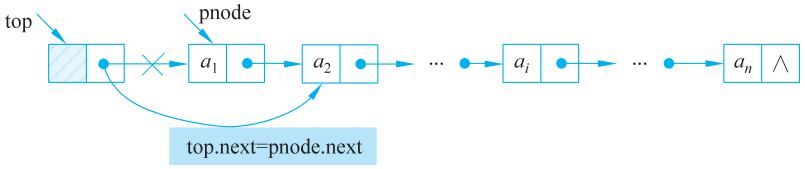


图 3-8　出栈操作

将栈顶元素出栈的算法实现如下：

```
def popStack(self):
    if self.stackEmpty():
```

```
            print("栈为空,不能进行出栈操作!")
            return None
        else:
            pnode=self.top.next
            self.top.next=pnode.next
            x=pnode.data
            del pnode
            return x
```

(5) 取栈顶元素。在取栈顶元素前要判断链栈是否为空。如果为空,则返回 None;否则,将栈顶元素返回。取栈顶元素的算法实现如下。

```
def getTop(self):
    if self.stackEmpty():
        print("栈为空,取栈顶元素失败!")
        return None
    else:
        return self.top.next.data
```

(6) 求链栈的长度。链栈的长度就是链栈的元素个数。从栈顶元素开始,通过 next 域找到下一个结点,并使用变量 len 计数,直到栈底为止,len 的值就是链栈的长度,将 len 返回即可,求链栈长度的时间复杂度为 $O(n)$。求链栈长度的算法实现如下:

```
def stackLength(self):
    p=self.top.next
    len=0
    while p is not None:
        p=p.next
        len=len+1
    return len
```

(7) 创建链栈。创建链栈主要利用链栈的插入操作实现,将用户输入的元素序列存入 eElem 中,然后依次取出每个元素,将其插入到链栈中,即将元素依次入栈。创建链栈的算法实现如下:

```
def createStack(self):
    print("请输入要入栈的整数:")
    eElem=list(map(int, input().split()))
    for e in eElem:
        pnode=LinkStackNode()
        pnode.data=e
        pnode.next=self.top.next
        self.top.next=pnode
```

测试代码如下:

```
if __name__=='__main__':
    S1 = MyLinkStack()
    S1.createStack()
    print(S1.getTop())
    print("长度",S1.stackLength())
    while S1.stackEmpty() is not True:
        e=S1.popStack()
```

```
        print("出栈元素:",e)
```

程序运行结果如下：

```
请输入要入栈的整数:
1 2 3 4
4
长度 4
出栈元素: 4
出栈元素: 3
出栈元素: 2
出栈元素: 1
```

(8) 清空链栈。在程序结束后要将申请的结点空间释放。从栈顶开始，依次通过 del 命令释放各结点空间，直到栈底为止。销毁链栈的算法实现如下：

```
def clearStack(self):
    while self.top is not None:
        p=self.top
        self.top=self.top.next
        del p
```

3. 链栈应用举例

【例 3-4】 利用链表模拟栈实现将十进制数 5678 转换为对应的八进制数。

【分析】 进制转换是计算机实现计算的基本问题。可以采用辗转相除法实现将十进制数转换为八进制数。将十进制数 5678 转换为八进制数的过程如图 3-9 所示。

	余数	
$5678 \div 8 = 709$	6	低位
$709 \div 8 = 88$	5	
$88 \div 8 = 11$	0	
$11 \div 8 = 1$	3	
$1 \div 8 = 0$	1	高位

$(5678)_{10} = (13056)_8$

图 3-9　十进制数 5678 转换为八进制数的过程

十进制数 5678 转换后的八进制数为 13056。观察图 3-9 的转换过程，每次不断利用被除数除以 8 得到商数后，记下余数，又将商数作为新的被除数继续除以 8，直到商数为 0 为止，把得到的余数按位序排列起来就是转换后的八进制数。十进制数 N 转换为八进制数的算法如下：

(1) 将 N 除以 8，记下其余数。

(2) 判断商是否为 0。如果为 0，结束程序；否则，将商送 N，转到(1)继续执行。

将得到的余数逆序排列就是转换后的八进制数，得到余数的顺序正好与八进制数的顺序相反，这正好可利用栈的后进先出特性，先把得到的余数序列放入栈中保存，最后依次出栈即得到八进制数。

在利用链表实现将十进制数转换为八进制数时，可以将每次得到的余数按照头插法插入链表，然后从链表的头指针开始依次输出结点的元素值，就得到了八进制数。这正好是元素的入栈与出栈操作，因此也可以利用栈的基本操作实现数的进制转换。

十进制数转换为八进制数的算法实现如下：

```
class LinkStackNode:
    def __init__(self):
        self.data=None
        self.next=None
class MyLinkStack:
```

```
        def __init__(self):
            self.top=LinkStackNode()
    def covert10to8(x):
        top=None
        while x != 0:
            p = LinkStackNode()
            p.data = x % 8
            p.next = top
            top = p
            x = x // 8
        num=[]
        while top is not None:
            p = top
            num.append(p.data)
            top = top.next
        return num
    if __name__=='__main__':
        x = int(input("请输入一个十进制整数"))
        y = covert10to8(x)
        print("转换后的八进制数是:",end="")
        for i in y:
            print(i,end='')
```

程序运行结果如图 3-10 所示。

```
C:\Users\o.o\.conda\envs\tensorflow\python.exe D:/Python程序/数据结构/进制转换.py
请输入一个十进制整数5678
转换后的八进制数是:13056
Process finished with exit code 0
```

图 3-10　例 3-4 程序运行结果


表达式求值 1

3.1.5　栈的典型应用

【例 3-5】　通过键盘输入一个表达式，如 6＋(7－1) * 3＋9/2，要求将其转换为后缀表达式，并计算该表达式的值。

【分析】　表达式求值是程序编译中的基本问题，它正是利用了栈的后进先出思想把人们便于理解的表达式翻译成计算机能够正确理解的表示序列。

一个算术表达式是由操作数和运算符组成的。为了简化问题求解，假设算术运算符仅由加、减、乘、除 4 种运算符和左、右圆括号组成。

例如，一个算术表达式为

$$6+(7-1)*3+9/2$$

算术表达式中的运算符总是出现在两个操作数之间，这种表达式被称为**中缀表达式**。计算机编译系统在计算一个算术表达式之前，要将中缀表达式转换为**后缀表达式**，然后对后缀表达式进行计算。后缀表达式就是算术运算符出现在操作数之后，并且不含括号。

计算机在求算术表达式的值时分为两个步骤：

(1) 将中缀表达式转换为后缀表达式。

(2) 计算后缀表达式的值。

1. 将中缀表达式转换为后缀表达式

要将一个算术表达式的中缀形式转化为后缀形式，首先需要了解四则运算规则。四则运算的规则如下：

(1) 先乘除，后加减。

(2) 同级别的运算从左到右进行。

(3) 先括号内，后括号外。

上面的算术表达式可转换为以下后缀表达式：

6 7 1－ 3 * ＋ 9 2 / ＋

不难看出，转换后的后缀表达式具有以下两个特点：

(1) 后缀表达式与中缀表达式的操作数出现顺序相同，只是运算符先后顺序改变了。

(2) 后缀表达式中不出现括号。

在利用后缀表达式进行算术运算时，编译系统不必考虑运算符的优先关系，仅需要从左到右依次扫描后缀表达式的各个字符，遇到运算符时，直接对运算符前面的两个操作数进行运算即可。

如何将中缀表达式转换为后缀表达式呢？可设置一个栈，用于存放运算符。本书约定 # 作为中缀表达式的结束标志，θ_1 为栈顶运算符，θ_2 为当前扫描的运算符。运算符的优先关系如表 3-4 所示。其中，＞表示 θ_1 的优先级高于 θ_2，＜表示 θ_1 的优先级低于 θ_2，＝表示二者优先级相同。

表 3-4　运算符的优先关系

θ_1	θ_2						
	＋	－	*	/	(	)	#
＋	＞	＞	＜	＜	＜	＞	＞
－	＞	＞	＜	＜	＜	＞	＞
*	＞	＞	＞	＞	＜	＞	＞
/	＞	＞	＞	＞	＜	＞	＞
(	＜	＜	＜	＜	＜	＝	
)	＞	＞	＞	＞		＞	＞
#	＜	＜	＜	＜	＜		＝

依次读入表达式中的每个字符，根据读取的当前字符进行以下处理：

(1) 初始化栈，并将 # 入栈。

(2) 若当前读入的字符是操作数，则将该操作数输出，并读入下一字符。

(3) 若当前字符是运算符，记作 θ_2，则将 θ_2 与栈顶的运算符 θ_1 比较。若 θ_1 优先级低于 θ_2，则将 θ_2 进栈；若 θ_1 优先级高于 θ_2，则将 θ_1 出栈并将其作为后缀表达式的一部分输出。然后继续比较新的栈顶运算符 θ_1 与当前运算符 θ_2 的优先级，若 θ_1 的优先级与 θ_2 相等，且 θ_1 为左括号，θ_2 为右括号，则将 θ_1 出栈，继续读入下一个字符。

(4) 如果 θ_2 的优先级与 θ_1 相等，且 θ_1 和 θ_2 都为 #，将 θ_1 出栈。

重复执行(2)～(4)，直到所有字符读取完毕。此时栈为空，完成了将中缀表达式转换为后缀表达式的过程，算法结束。

中缀表达式 6+(7−1)*3+9/2#转换为后缀表达式的具体过程如图 3-11 所示(为了转换方便,在要转换表达式的末尾加一个'#'作为结束标记)。

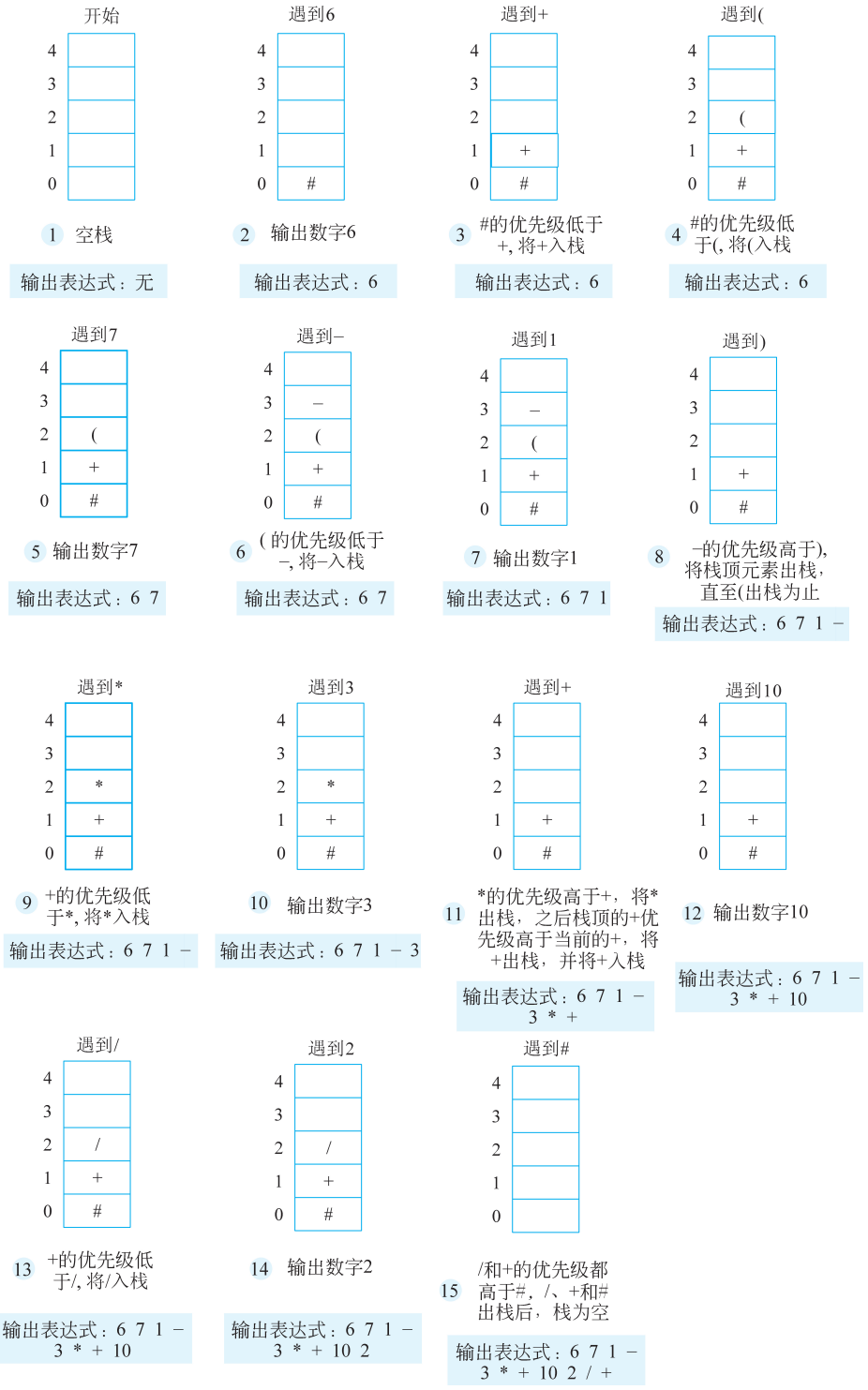


图 3-11　中缀表达式 6+(7−1)*3+9/2 转换为后缀表达式的过程

2. 求后缀表达式的值


表达式求值 2

将中缀表达式转换为后缀表达式后，就可以计算后缀表达式的值了。计算后缀表达式的值的规则如下：依次读入后缀表达式中的每个字符。如果是操作数，则将操作数入栈；如果是运算符，则将处于栈顶的两个操作数出栈，然后利用当前运算符进行运算，将运算结果入栈。重复执行上述过程，直到整个表达式处理完毕。

利用上述规则，后缀表达式的 6 7 1 － 3 ＊ ＋ 9 2 / ＋的运算过程如图 3-12 所示。

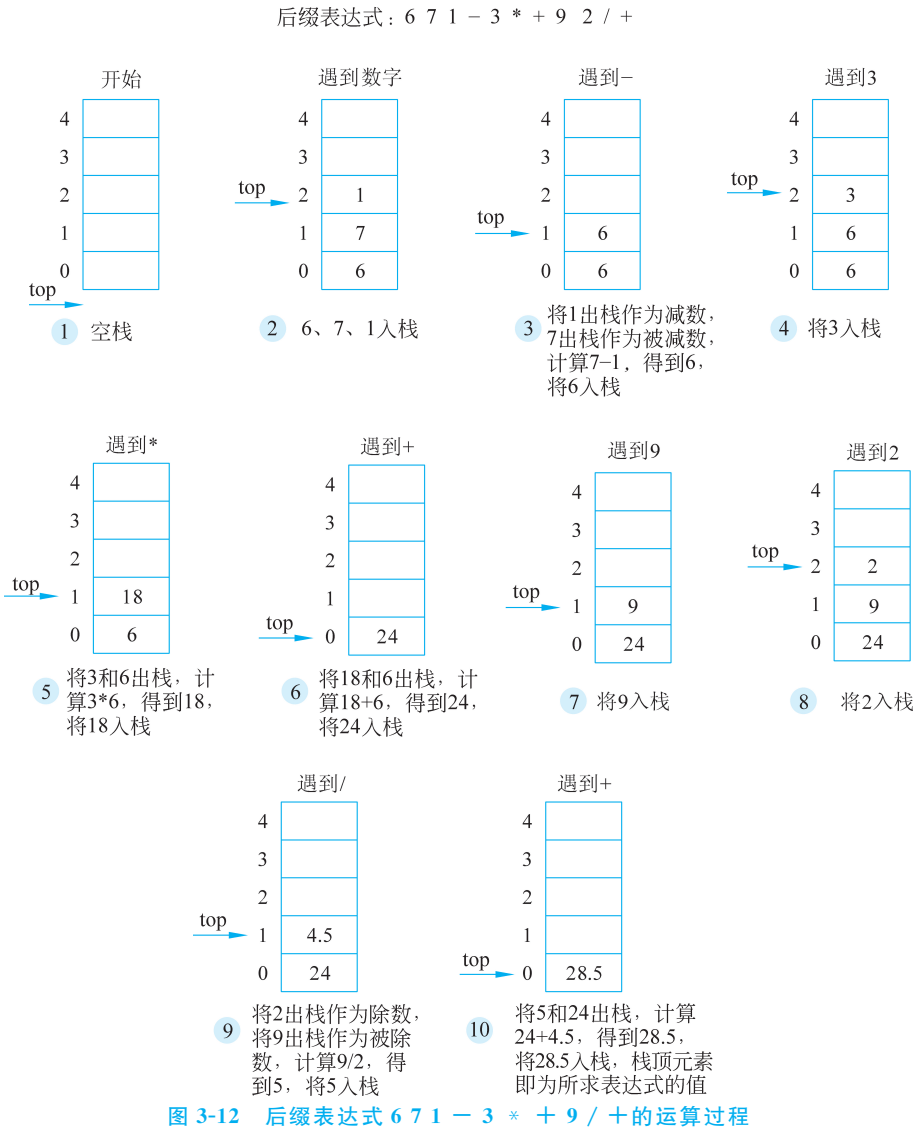


图 3-12　后缀表达式 6 7 1 － 3 ＊ ＋ 9 / ＋的运算过程

3. 算法实现

具体实现时，设置两个字符列表 str、exp 及一个栈 S，其中，str 用于存放中缀表达式的字符串，exp 用于存放转换后的后缀表达式的字符串，S 用于存放转换过程中遇到的运算符。具体如下：

(1) 将中缀表达式转换为后缀表达式的方法是：依次扫描列表 str 中的每个字符。如

果遇到的是数字,则将其直接存入列表 exp 中。如果遇到的是运算符,则将 S 的栈顶运算符与当前运算符比较。如果当前运算符的优先级高于栈顶运算符的优先级,则将当前运算符入栈;如果栈顶运算符的优先级高于当前运算符的优先级,则将栈顶运算符出栈,并保存到列表 exp 中。

(2) 求后缀表达式的值的方法是:依次扫描后缀表达式中的每个字符。如果是数字字符,将其转换为数字(数值型数据),并将其入栈;如果是运算符,则将栈顶的两个数字出栈,进行加、减、乘、除运算,并将结果入栈。当后缀表达式对应的字符串处理完毕后,将栈顶元素返回给被调用函数,即为所求表达式的值。

利用栈求算术表达式的值的算法实现如下:

```
def TranslateExpress(self,str,exp):
#中缀表达式转换为后缀表达式
    i=0
    j=0
    end=False
    ch=str[i]
    i=i+1
    while i <=len(str) and not end:
        if(ch=='('):                    #如果当前字符是左括号,则将其入栈
            self.PushStack(ch)
        elif(ch==')'):          #如果是右括号,将栈中的运算符出栈,并将其存入列表 exp 中
            while (self.GetTop()!=None and self.GetTop()!= '('):
                e=self.PopStack()
                exp[j] = e
                j =j+1
            e=self.PopStack()           #将左括号出栈
        elif(ch=='+' or ch=='-'):       #若遇到+和-,因为其优先级低于栈顶运算符的优
                                        #先级,所以先将栈顶字符出栈并送入列表 exp
                                        #中,然后将当前运算符入栈
            while not self.StackEmpty() and self.GetTop() != '(':
                e=self.PopStack()
                exp[j]=e
                j=j+1
            self.PushStack(ch)          #当前运算符入栈
        elif(ch=='*' or ch=='/' ):      #若遇到 * 和/, 先将同级运算符出栈并送入列表
                                        #exp 中,然后将当前运算符入栈
            while not self.StackEmpty() and self.GetTop()== '/' or self.GetTop()
== '*':
                e=self.PopStack()
                exp[j] = e
                j=j+1
            self.PushStack(ch)          #当前运算符入栈
        elif(ch==' '):                  #若遇到空格,忽略
            break
        else:                           #若遇到操作数,则将操作数直接送入列表 exp 中
            while (ch >= '0' and ch <= '9'):
                exp[j] = ch
                j=j+1
                if(i<len(str)):
                    ch = str[i]
```

```
                else:
                    end=True
                    break
                i=i+1
            i=i-1
        ch = str[i]                          #读入下一个字符,准备处理
        i=i+1
    while not self.StackEmpty():             #将栈中所有剩余的运算符出栈,送入列表 exp 中
        e=self.PopStack()
        exp[j] = e
        j=j+1
        def ComputeExpress(self,a):
            i=0
            while(i<len(a)):
                if(a[i]>='0' and a[i]<='9'):
                    self.PushStack(int(a[i]))      #处理之后将数字入栈
                else:
                    if(a[i]=='+'):
                        x1=self.PopStack()
                        x2=self.PopStack()
                        result=x1+x2
                        self.PushStack(result)
                    elif(a[i]=='-'):
                        x1=self.PopStack()
                        x2=self.PopStack()
                        result=x2-x1
                        self.PushStack(result)
                    elif(a[i]=='*'):
                        x1=self.PopStack()
                        x2=self.PopStack()
                        result=x1*x2
                        self.PushStack(result)
                    elif(a[i]=='/'):
                        x1=self.PopStack()
                        x2=self.PopStack()
                        result=x2/x1
                        self.PushStack(result)
                i=i+1
            if not self.StackEmpty():                #如果栈不空,将结果出栈,并返回
                result=self.PopStack()
            if self.StackEmpty():
                return result
            else:
                print("表达式错误")
                return result
    if __name__=='__main__':
        S = MySeqStack()
        str=input("请输入一个算术表达式:")
        exp=''
        str = [x for x in str[::1]]
        exp = [None for x in str[::1]]
        S.TranslateExpress(str,exp)
        exp2=[]
```

```
        exp2 = list(filter(None, exp))
        str="".join(str)
        print("表达式",str,"的值=",S.ComputeExpress(exp2))
```

程序运行结果如图 3-13 所示。

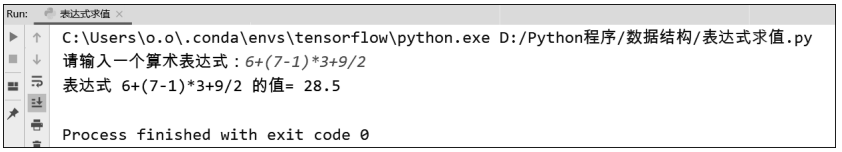


图 3-13　利用栈求算术表达式的值的程序运行结果

注意：(1) 在将中缀表达式转换为后缀表达式的过程中，遇到连续的数字字符，则需要将连续的数字字符作为一个数字处理，而不是作为两个数字或多个数字，这可以在函数 ComputeExpress 或 TranslateExpress 中进行处理。

(2) 在 ComputeExpress 函数中，遇到－运算符时，先出栈的为减数，后出栈的为被减数；对于/运算也一样。

【想一想】 能否在求解算术表达式的值时不输出转换的后缀表达式而直接进行求值？

【分析】 求算术表达式的值时，也可以同时进行将中缀表达式转换为后缀表达式和利用后缀表达式求值这两个过程，这需要定义两个栈：运算符栈 Opnd 和操作数栈 Optr，将原来操作数的输出变成将其送操作数栈操作，在运算符出栈时，需要将操作数栈中的元素输出并进行相应运算，然后将运算结果送入操作数栈。

算法实现如下：

```
def CalExpress(str):
#计算表达式的值
    Optr=OptStack()
    Opnd=OptStack()
    Optr.Push('#')
    n=len(str)
    i=0
    res=0
    a=[]
    print("运算符栈和操作数栈的变化情况如下:")
    while i<n or Optr.GetTop() is not None:
        if i<n and not IsOptr(str[i]):          #是操作数
            while i<n and not IsOptr(str[i]):
                a.append(str[i])
                i+=1
            if len(a)>=1:
                res=StrtoInt(a)
            a=[]
        if res!=0:
            Opnd.Push(res)                      #将当前运算结果送入运算符栈
            DispStackStatus(Optr,Opnd)
        res = 0
        if IsOptr(str[i]):                      #是运算符
```

```
            if Precede(Optr.GetTop(),str[i])=='<':
                Optr.Push(str[i])
                i+=1
                DispStackStatus(Optr,Opnd)
            elif Precede(Optr.GetTop(),str[i])=='>':
                theta=Optr.Pop()
                rvalue=Opnd.Pop()
                lvalue=Opnd.Pop()
                exp=GetValue(theta,lvalue,rvalue)
                Opnd.Push(exp)
                DispStackStatus(Optr,Opnd)
            elif Precede(Optr.GetTop(),str[i])=='=':
                theta=Optr.Pop()
                i+=1
                DispStackStatus(Optr,Opnd)
    return Opnd.GetTop()                          #返回表达式的值
def GetValue(ch,a,b):                             #求值
    if ch=='+':
        return a + b
    elif ch=='-':
        return a-b
    elif ch=='*':
        return a*b
    elif ch=='/':
        return a/b
if __name__=='__main__':                          #主函数
    str=input('请输入算术表达式串:')
    res = CalExpress(str)
    print('表达式',str,'的运算结果为:',res)
```

程序运行结果如图 3-14 所示。

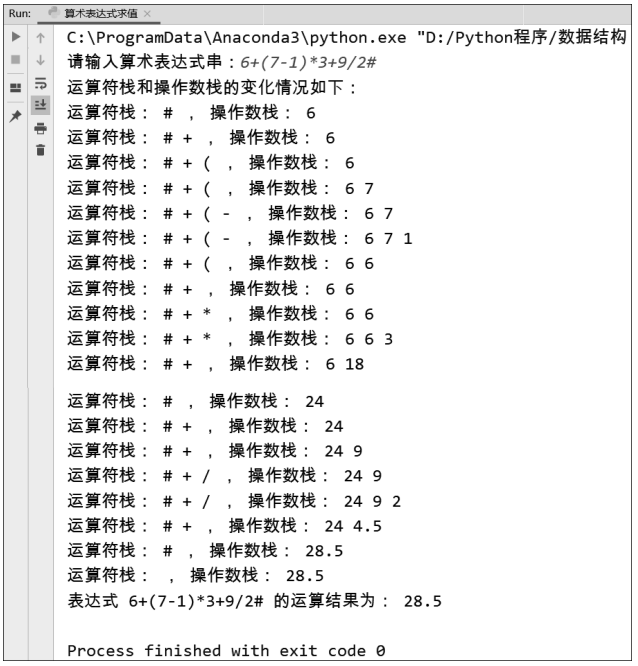


图 3-14　转换和求值同时进行的程序运行结果

【思考】 若遇到连续数字字符表示的多位数，如 123+16 * 20 中的 123、16 和 20，要将这些字符串转换为对应的整数。如果使用栈，该如何处理呢？

3.2 栈与递归

栈的后进先出的思想还体现在递归函数中。本节主要介绍栈与递归调用的关系、递归利用栈的实现过程以及递归的消除。

3.2.1 设计递归算法

递归是指在函数的定义中又出现了对自身的调用。如果一个函数在函数体中直接调用自身，称为直接递归函数；如果一个函数经过一系列中间调用间接地调用自身，称为间接递归函数。

1. 斐波那契数列

【例 3-6】 如果兔子在出生两个月后就有繁殖能力，以后一对兔子每个月能生出一对兔子，假设所有兔子都不死，那么一年以后共有多少对兔子呢？

不妨拿新出生的一对小兔子进行分析。第一、二个月小兔子没有繁殖能力，所以还是一对；两个月后，生下一对小兔子，共有 2 对兔子；三个月后，老兔子又生下一对，因为小兔子还没有繁殖能力，所以一共是 3 对兔子；以此类推，可以得出如表 3-5 所示的每个月兔子的对数。

表 3-5 每个月兔子的对数

经过的月数	1	2	3	4	5	6	7	8	9	10	11	12
兔子对数	1	1	2	3	5	8	13	21	34	55	89	144

从表 3-5 中不难看出，数字 1，1，2，3，5，8，…构成了一个数列，这个数列有一个十分明显的特征，即前面相邻两项之和构成后一项，可用数学函数表示如下。

$$\mathrm{Fib}(n)=\begin{cases}0, & n=0\\ 1, & n=1\\ \mathrm{Fib}(n-1)+\mathrm{Fib}(n-2) & n>1\end{cases}$$

求斐波那契数列的非递归算法实现如下：

```
def Fib(n):
    a,b=1,1
    i=0
    f=[]
    while i<n:
        f.append(a)
        a,b=b,a+b
        i=i+1
    return f
```

如果用递归方法实现，代码结构会更加清晰：

```
def Fib(n):                                  #使用递归方法计算斐波那契数列
    if n==0:                                 #若是第 0 项
        return 0                             #则返回 0
```

```
    elif n==1:                                  #若是第 1 项
        return 1                                #则返回 1
    else:                                       #其他情况
        return Fib(n-1)+Fib(n-2)                #第三项为前两项之和
    for i in range(1,11):
        print(Fib(i),end=' ')
```

当 $n=4$ 时，递归函数 Fib(n)的执行过程如图 3-15 所示。

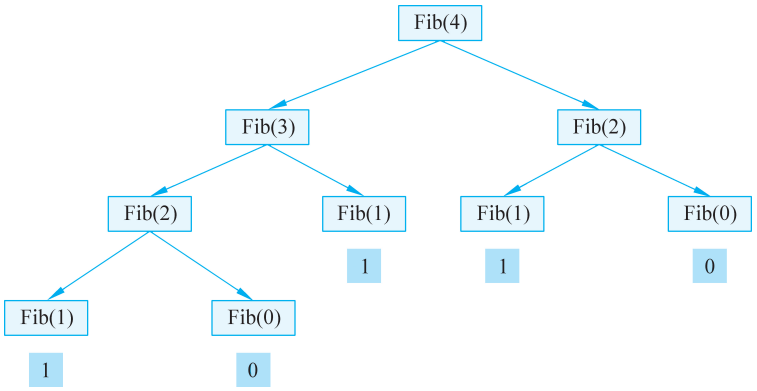


图 3-15　递归函数 Fib(n)的执行过程

2. 求 n 的阶乘

【例 3-7】　求 n 的阶乘的递归函数定义如下：

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

求 n 的阶乘的递归算法实现如下：

```
def Fact(n):                                    #求 n 的阶乘
    if n==1:
        return 1
    else:
        return n * Fact(n-1)
```

3. Ackermann 函数

【例 3-8】　Ackermann 函数定义如下：

$$\text{Ack}(m,n)=\begin{cases}n+1, & m=0\\ \text{Ack}(m-1,1), & m\neq 0,n=0\\ \text{Ack}(m-1,\text{Ack}(m,n-1)), & m\neq 0,n\neq 0\end{cases}$$

Ackermann 递归函数算法实现如下。

```
def Ack(m,n):                                   #Ackermann 递归算法实现
    if m==0:
        return n+1
    elif n==0:
        return Ack(m-1,1)
    else:
        return Ack(m-1,Ack(m,n-1))
```