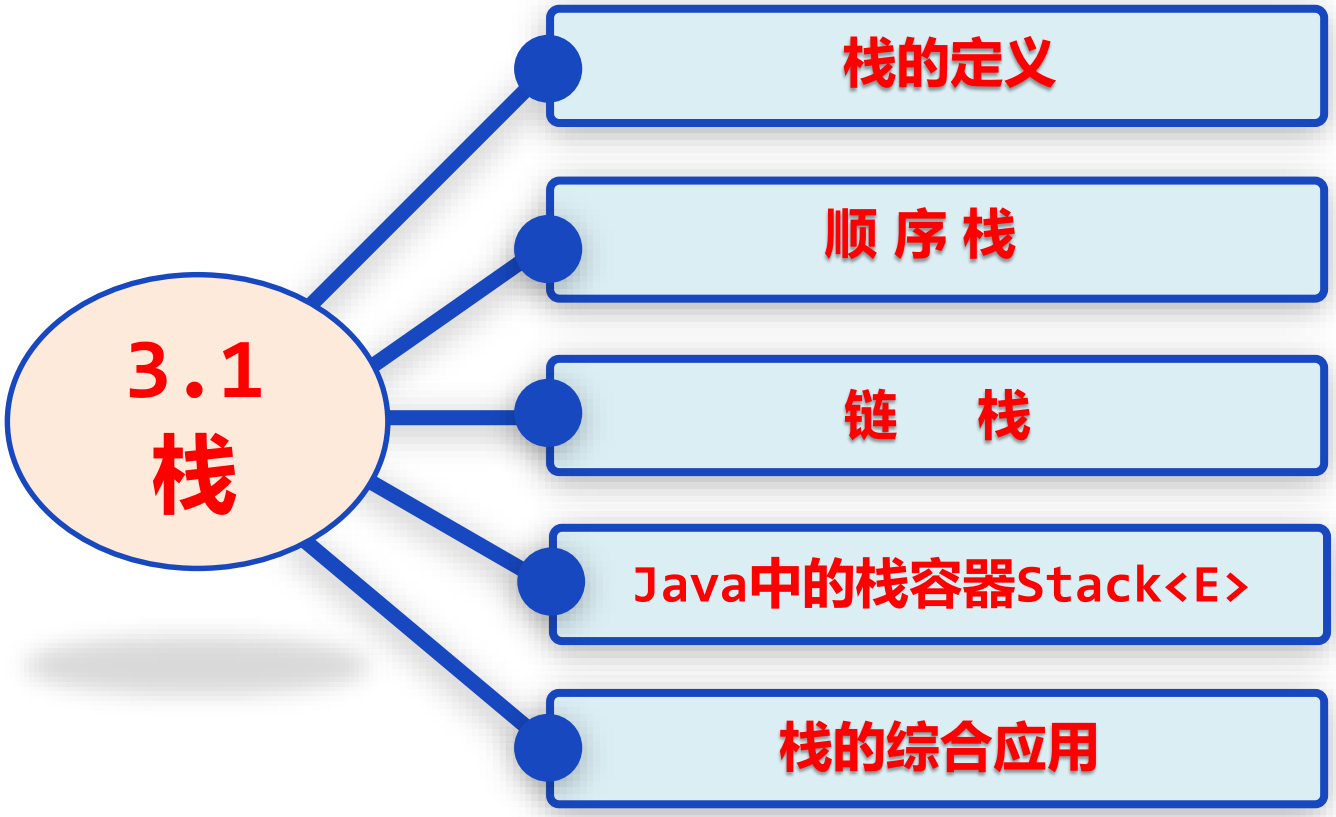
第3章 栈和队列

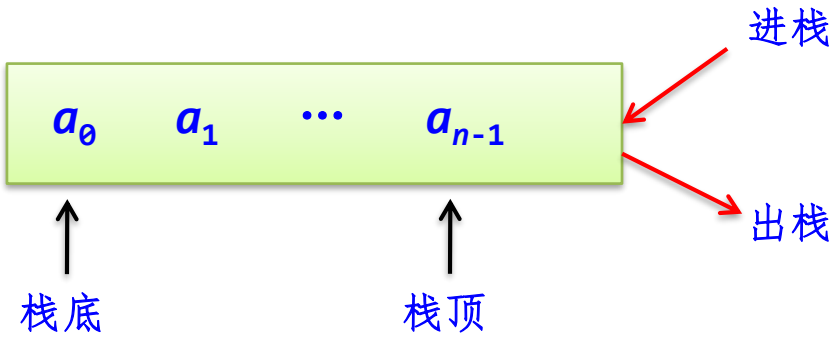
提纲

CONTENTS


3.1
栈
栈的定义
顺 序 栈
链　　栈
Java中的栈容器Stack<E>
栈的综合应用

3.1.1 栈的定义

- 栈（stack）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（top），表的另一端称为栈底（bottom）。
- 栈的插入操作通常称为进栈或入栈（push），栈的删除操作通常称为退栈或出栈（pop）。



后放置的木块先取出来

栈的主要特点：

- 后进先出，即后进栈的元素先出栈。
- 每次进栈的元素都作为新栈顶元素，每次出栈的元素只能是当前栈顶元素。
- 栈也称为后进先出表或者先进后出表。

ADT Stack

{

数据对象：

D={a_i | 0≤i≤n-1，n≥0，元素a_i为E类型}

数据关系：

R={r}

r={<a_i，a_{i+1}> | a_i，a_{i+1}∈D， i=0，…，n-2}

基本运算：

empty()：判断栈是否为空，若空栈返回真；否则返回假。

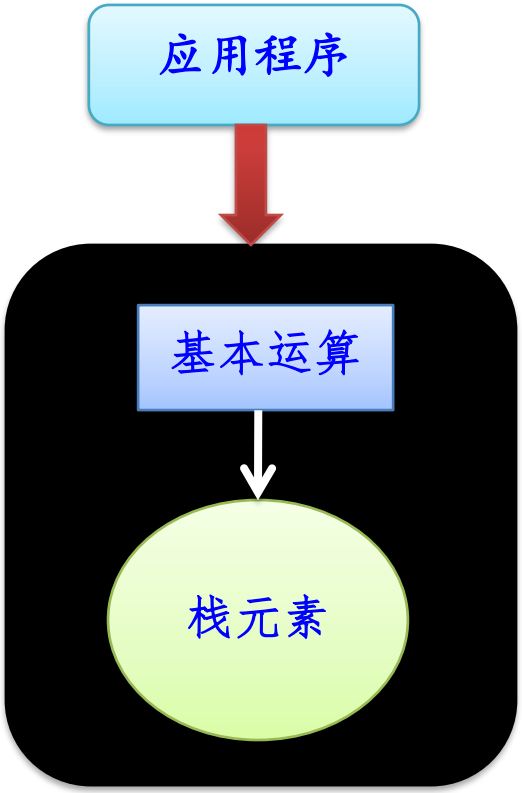
push(e)：进栈操作，将元素e插入到栈中作为栈顶元素。

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。

}

栈抽象数据类型 = 线性结构 + 栈的基本运算


应用程序
基本运算
栈元素

示例

一个栈的进栈序列是a、b、c、d、e，则栈的不可能的输出序列是（ ）。

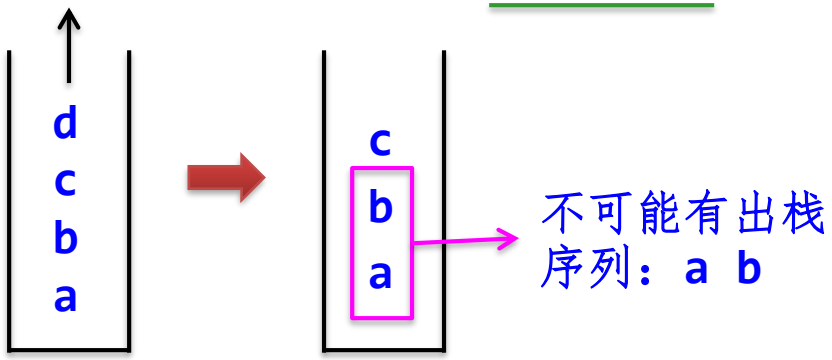
A.edcba　　B.decba

C.dceab　　D.abcde

方法1：用栈模拟进行判断

考虑 C.dceab

d



示例

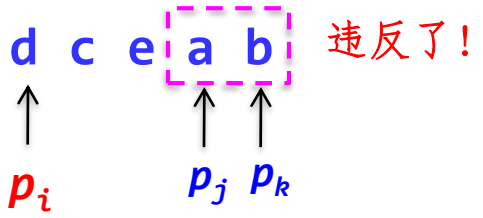
一个栈的进栈序列是a、b、c、d、e，则栈的不可能的输出序列是（　）。

A.edcba　　B.decba

C.dceab　　D.abcde

方法2：利用判断准则

判断准则：输入序列为1，2，…，n，（p_1，p_2，…，p_n）是1，2，…，n的一种排列，利用一个栈得到输出序列（p_1，p_2，…，p_n）的充分必要条件是不存在这样的i、j、k满足$i<j<k$的同时也满足$p_j<p_k<p_i$。



1～*n*共产生 $\frac{1}{n+1}C_{2n}^{n}$ （卡特兰数）种合法出栈序列。

例如，*n*=5

4 5 3 2 1　　合法

3 1 2 5 4　　不合法

↑

1 4 2 3 5　　不合法

↑

示例

已知一个栈的进栈序列是1，2，3，…，n，其输出序列是p_1，p_2，…，p_n，若$p_1=n$，则p_i的值为（　）。

A.i　　B.$n-i$　　C.$n-i+1$　　D.不确定

1，2，3，…，n ⟹ n，…

输出序列唯一

$p_1=n$

$p_2=n-1$

$p_3=n-2$

…

$p_n=1$

$p_i+i=n+1$即$p_i=n-i+1$

2013年全国硕士研究生入学统一考试题

示例

2.一个栈的入栈序列为1,2,3，…,n ，其出栈序列是$p_1,p_2,p_3,\cdots,p_n$。若$p_2=3$，则p_3可能取值的个数是（　）。

A.$n-3$　　B.$n-2$　　C.$n-1$　　D.无法确定

n，…,4

1进，1出，2进，3进，3出，2出，…

1 3 2

p_1 p_2 p_3

或者　1进，2进，2出，3进，3出，1出，…

2 3 1

p_1 p_2 p_3

或者　1进，2进，2出，3进，3出，4进，4出，…，1出

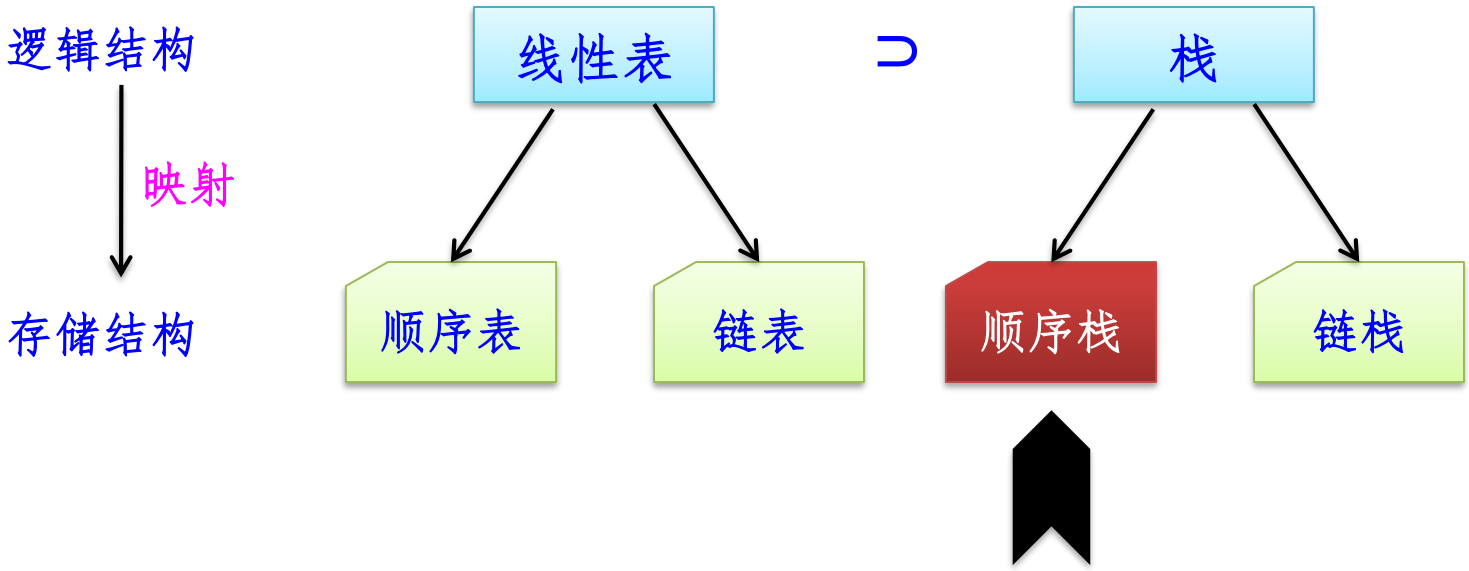
2 3 4

p_1 p_2 p_3

…　p_3除了3外都可能!

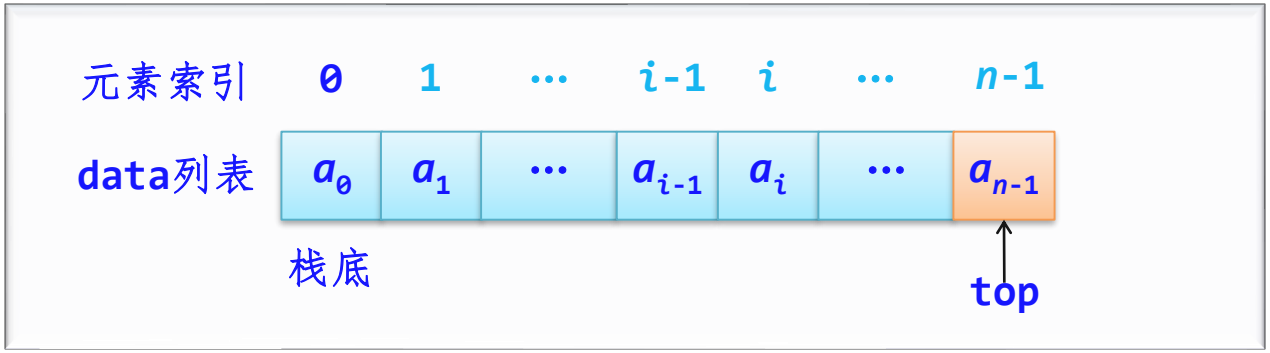
3.1.2 栈的顺序存储结构及其基本运算算法实现

栈的实现方式



顺序栈实现

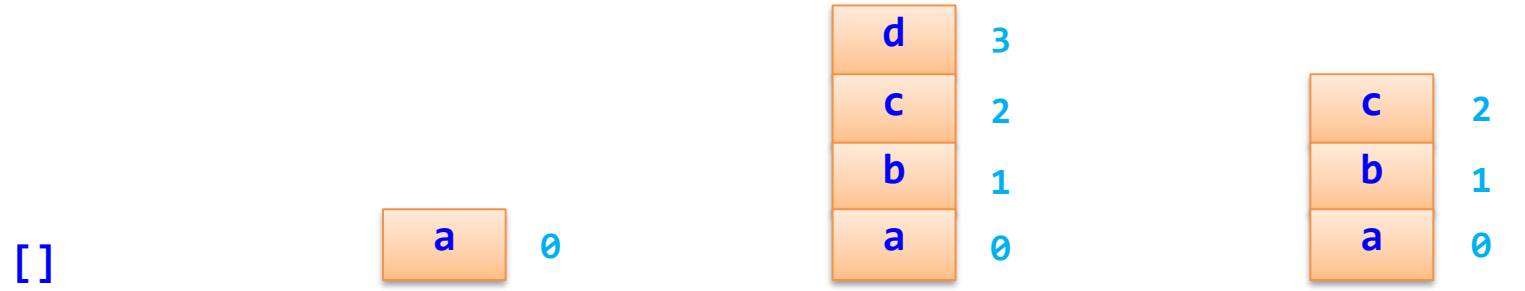
常规做法：



利用动态列表的做法：

- 利用Python列表具有动态扩展的功能。
- 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)恰好为栈中实际元素个数。

顺序栈



（a）空栈　　（b）元素a进栈　　（c）元素b、c、d进栈　　（d）元素d出栈

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。

顺序栈类SqStack

```
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

（1）判断栈是否为空empty()

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

（2）进栈push(e)

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

（3）出栈pop()

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

（4）取栈顶元素gettop()

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

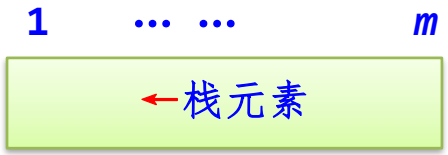
顺序栈的几个问题

问题1：若采用数组data[1..*m*]存放栈元素，回答以下问题：

（1）只能以data[1]端作为栈底吗？

（2）为什么不能以data数组的中间位置作为栈底？

答：（1）也可以将data[*m*]端作为栈底。



（2）栈中元素是从栈底向栈顶方向生长的，如果以data数组的中间位置作为栈底，那么栈顶方向的另外一端空间就不能使用，造成空间浪费，所以不能以data数组的中间位置作为栈底。

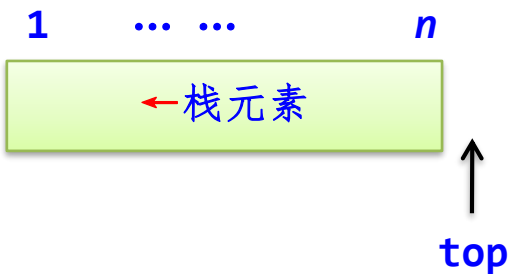
问题2（常规顺序栈）：

若一个栈用数组data[1..*n*]存储，初始栈顶指针top为*n*+1，则以下元素*x*进栈的正确操作是（　）。

A.top++; data[top]=*x*;　　B.data[top]=*x*; top++;

C.top--; data[top]=*x*;　　D.data[top]=*x*; top--;

答：初始栈顶指针top为*n*+1，说明data[*n*]端作为栈底，在进栈时top应递减，由于不存在data[*n*+1]的元素，所以在进栈时应先将top递减，再将*x*放在top处（top指向栈顶元素）。答案为C。

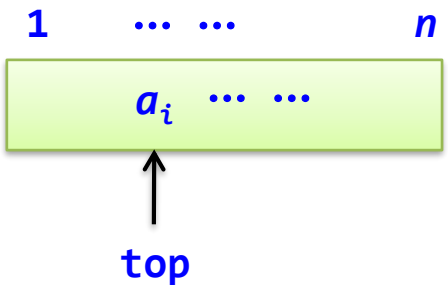


若一个栈用数组data[1..*n*]存储，初始栈顶指针top为*n*+1，则以下元素*x*出栈的正确操作是（　）。

A.*x*=data[top]; top++;　　B.top++; *x*=data[top];

C.*x*=data[top]; top--;　　D.top--; *x*=data[top];

答：进栈操作是：top--;data[top]=*x*（ top指向栈顶元素）；出栈操作与进栈操作相反，应该为*x*=data[top]; top++;。题答案为A。

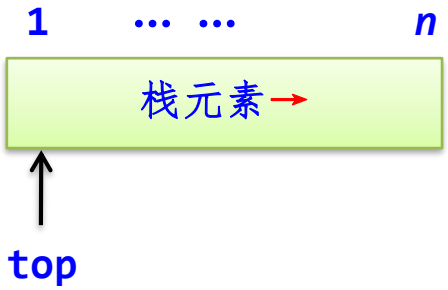


若一个栈用数组data[1..*n*]存储，初始栈顶指针top为1，则以下元素*x*进栈的正确操作是（　）。

A.top++; data[top]=*x*;　　B.data[top]=*x*; top++;

C.top--; data[top]=*x*;　　D.data[top]=*x*; top--;

答：初始栈顶指针top为1，说明data[1]端作为栈底，在进栈时top应递增，由于存在data[1]的元素，所以在进栈时应先将*x*放在top处，再top递增（top指向栈顶元素的前一个位置！）。答案为B。

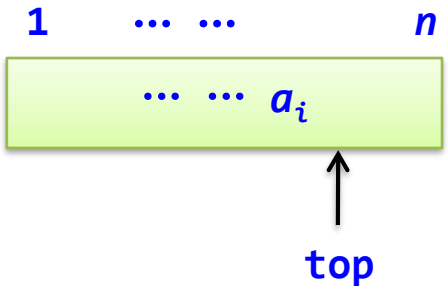


若一个栈用数组data[1..*n*]存储，初始栈顶指针top为1，则以下元素*x*出栈的正确操作是（　）。

A.*x*=data[top]; top++;　　B.top++; *x*=data[top];

C.*x*=data[top]; top--;　　D.top--; *x*=data[top];

答：进栈操作是：　B.data[top]=*x*; top++;（top指向栈顶元素的前一个位置）。出栈操作与进栈操作相反，应该为top--; *x*=data[top]。题答案为D。

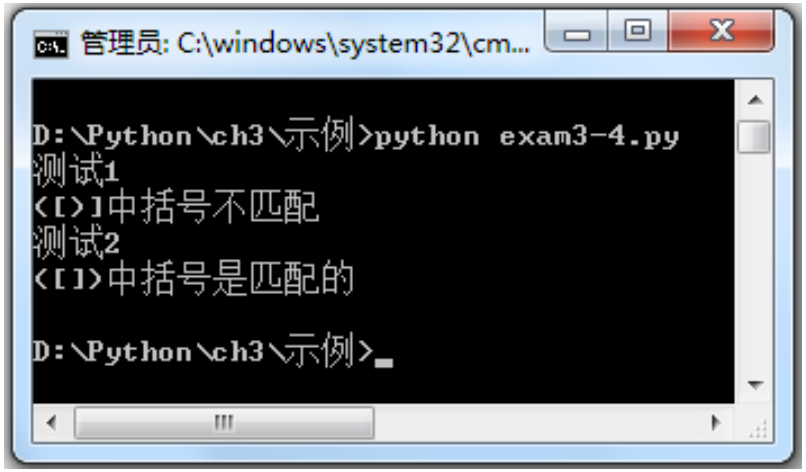


3.1.3 顺序栈的应用算法设计示例

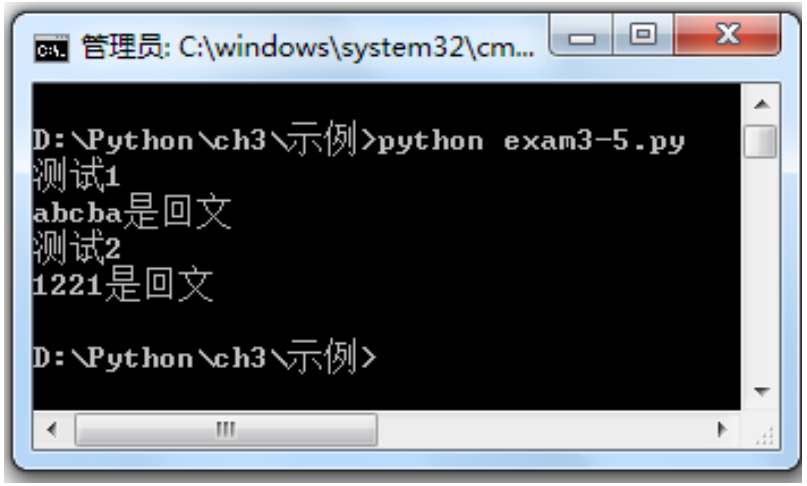
【例3.4】设计一个算法利用顺序栈检查用户输入的表达式中括号是否配对（假设表达式中可能含有圆括号、中括号和大括号）。并用相关数据进行测试。

```
from SqStack import SqStack          #引用顺序栈SqStack
def isMatch(str):                    #判断表达式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'('返回假
        st.pop()
      if e==']':
        if st.empty() or st.gettop()!='[':
          return False               #栈空或栈顶不是'['返回假
        st.pop()
      if e=='}':
        if st.empty() or st.gettop()!='{':
          return False;              #栈空或栈顶不是'{'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+"中括号是匹配的")
else:
  print(str+"中括号不匹配")
print("测试2")
str="([])"
if isMatch(str):
  print(str+"中括号是匹配的")
else:
  print(str+"中括号不匹配")
```



【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。



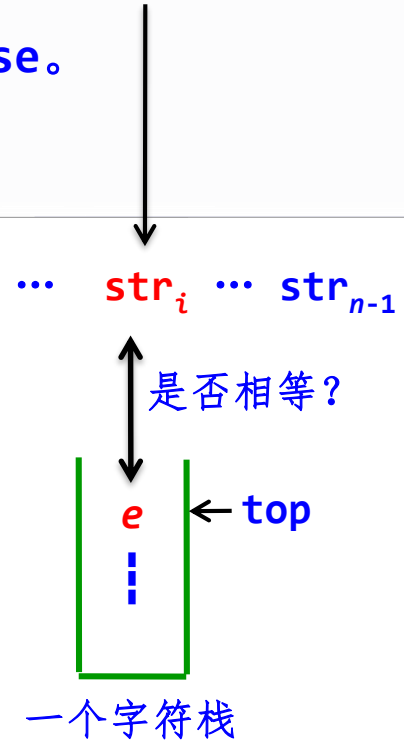
解：用str存放表达式，其中含n个字符。若str的前半部分的反向序列与str的后半部分相同，则是回文，否则不是回文。判断过程如下：

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。

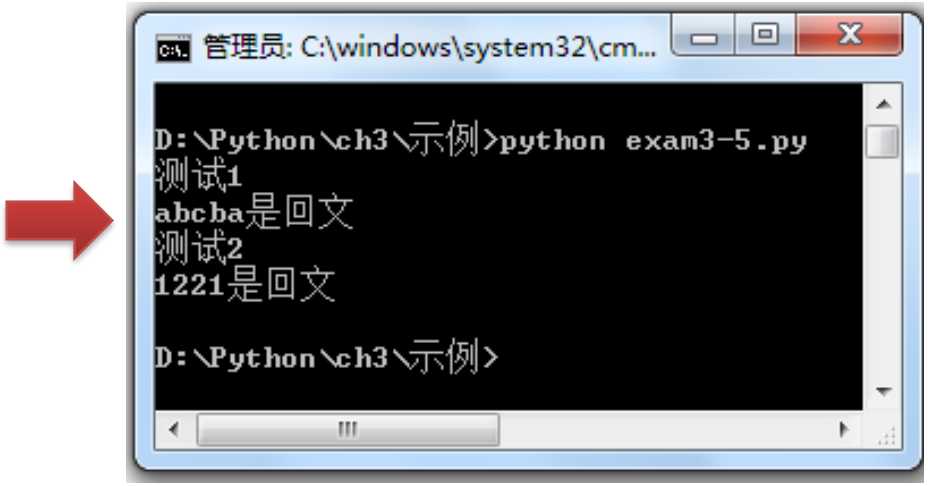
③ i继续遍历其他后半部分字符，每访问一个字符，则出栈一个字符，两者进行比较，如图所示，若不相等返回False。

④ 当str遍历完毕返回True。



```
from SqStack import 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")
```

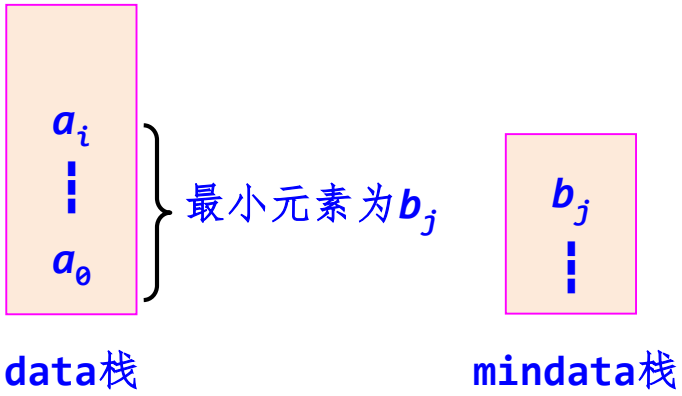


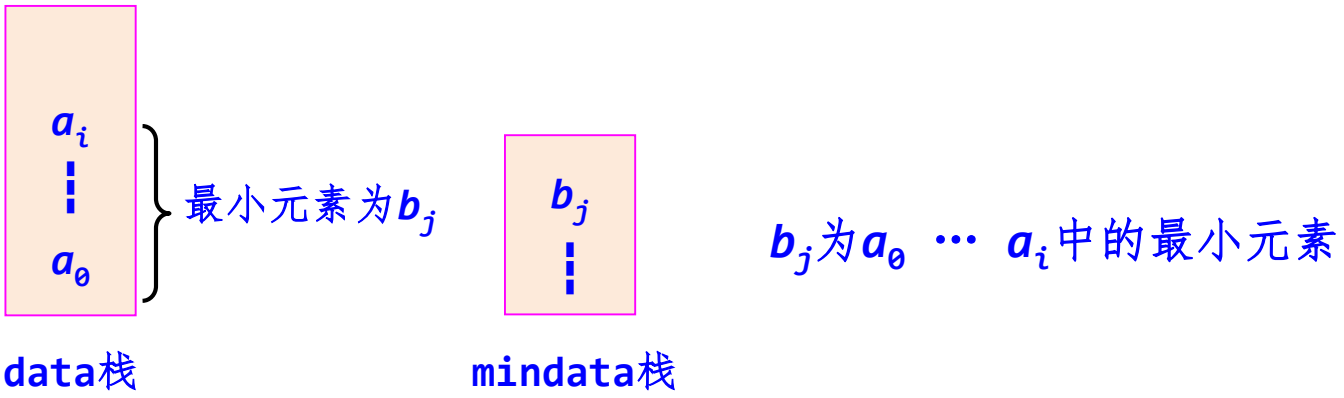
【例3.6】设计最小栈。定义栈的数据结构，添加一个Getmin()方法用于返回栈中的最小元素。要求方法Getmin()、push以及pop的时间复杂度都是O(1)。例如：

```
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
```

解：设计满足题目要求的顺序栈类为STACK：

- 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，…，a_i（$i \geqslant 1$）进栈到data栈后，min栈的栈顶元素b_j为a_0，a_1，…，a_i中的最小元素（含后进栈的重复最小元素），如下图所示。





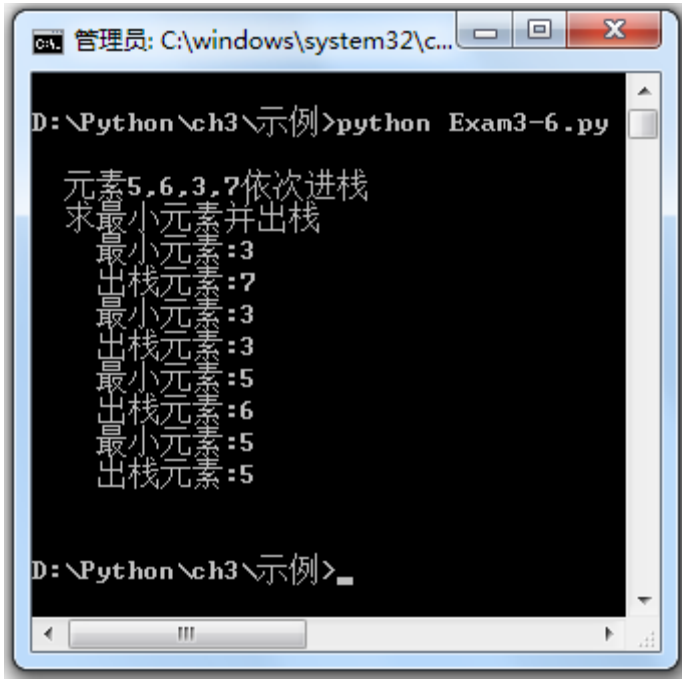
STACK类的主要运算算法设计如下：

- 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。
- 进栈方法push(x)的操作是，当data栈空或者进栈元素x小于等于当前栈中最小元素（即x≤Getmin()）时，则将x进mindata栈。最后将x进data栈。
- 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。
- 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。

```
class STACK:                                    #含Getmin()的栈类
  def __init__(self):                           #构造方法
    self.data=[]                                #存放主栈中元素，初始为空
    self.__mindata=[]                           #存放min栈中元素，初始为空
  #min栈基本运算算法

  def __minempty(self):                         #判断min栈是否空
    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栈栈顶元素
    assert not self.__minempty()                #检测min栈为空的异常
    return self.__mindata[-1];
```

```
#主栈基本运算算法
def empty(self):                    #判断主栈是否空
  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()
```



共享栈问题

【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。

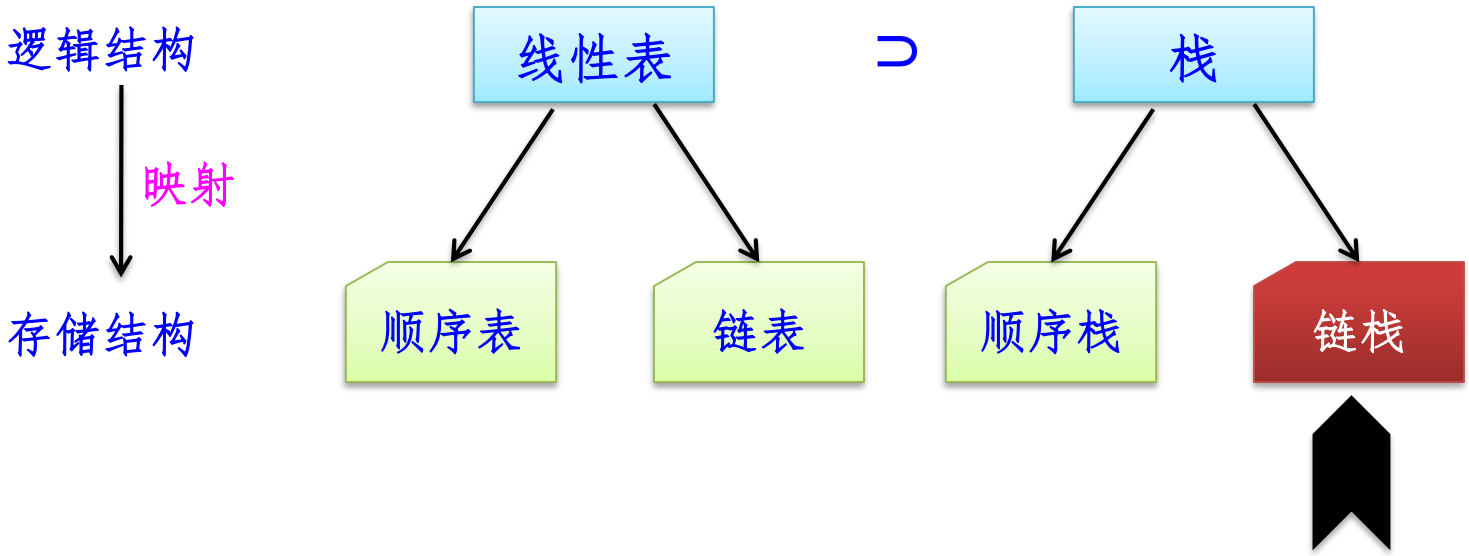
解：为了尽量利用空间，减少溢出的可能，可以让两个的栈顶相向即进栈元素迎面增长的存储方式，为此设置两个栈的栈顶指针分别为top1和top2（均指向对应栈的栈顶元素）。

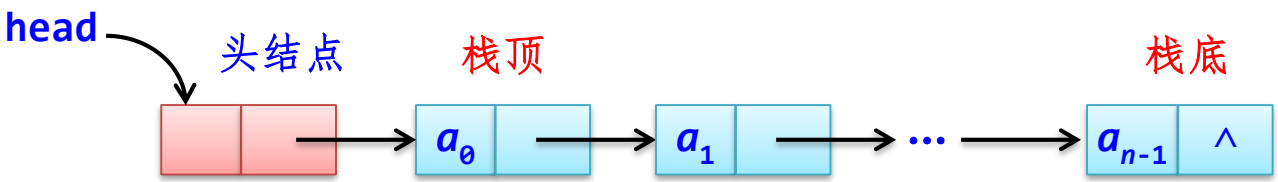
- 栈S1空的条件是top1=-1。
- 栈S1满的条件是top1=top2-1;
- 元素e进栈S1（栈不满时）的操作是：top1++;s[top1]=e。
- 元素出栈S1（栈不空时）的操作是：e=s[top1];top1--。

- 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--;s[top2]=e。
- 元素出栈S2（栈不空时）的操作是：e=s[top2];top2++。

3.1.4 栈的链式存储结构及其基本运算算法实现

栈的实现方式





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：

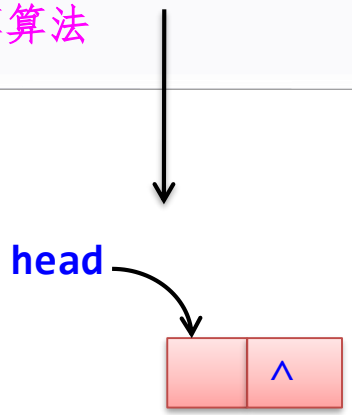
- 栈空的条件：head.next==None。
- 由于只有在内存溢出才会出现栈满，通常不考虑这种情况。
- 元素e进栈操作：将包含该元素的结点s插入作为首结点。
- 出栈操作：返回首结点值并且删除该结点。

和单链表一样，链栈中每个结点的类型LinkNode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

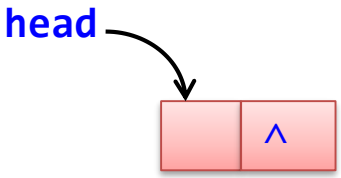
```
class LinkStack:                    #链栈类
  def __init__(self):               #构造方法
    self.head=LinkNode()            #头结点head
    self.head.next=None
  #栈的基本运算算法
```



链栈的基本运算算法

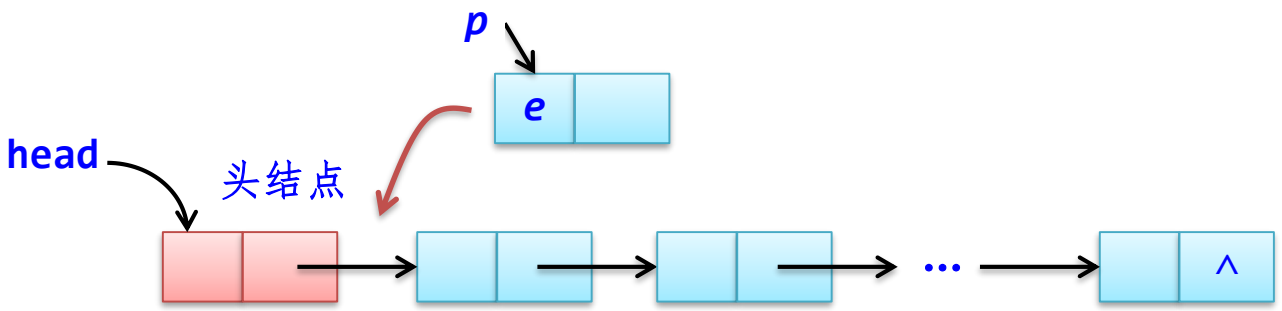
（1）判断栈是否为空empty()

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



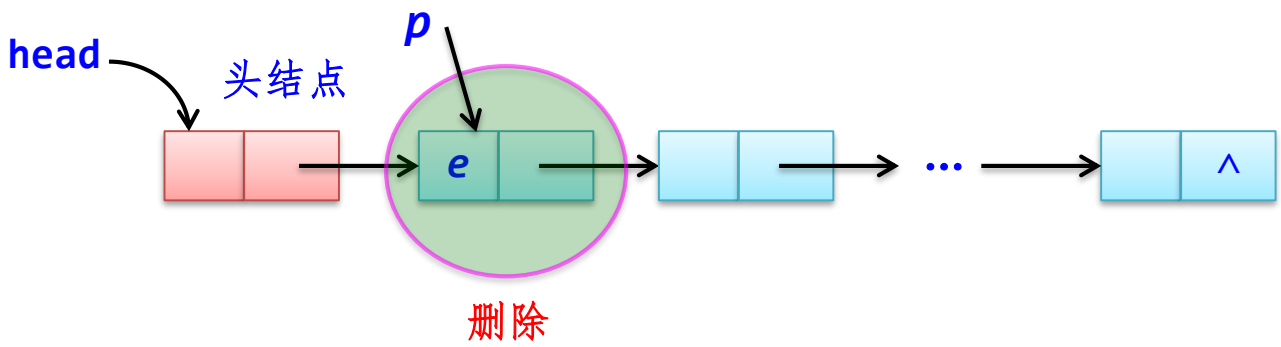
（2）进栈push(e)

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



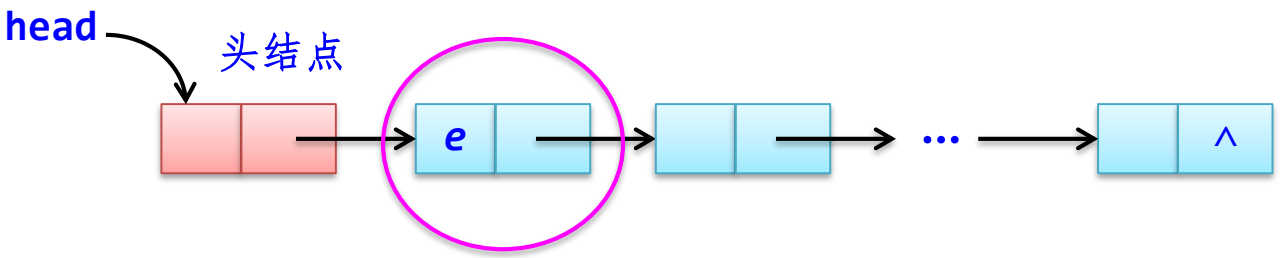
（3）出栈pop()

```
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()

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



链栈的几个问题

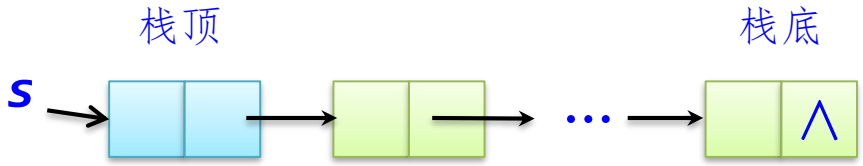
问题1：在以下几种存储结构中，哪个最适合用作链栈？

（1）带头结点的单链表

（2）不带头结点的循环单链表

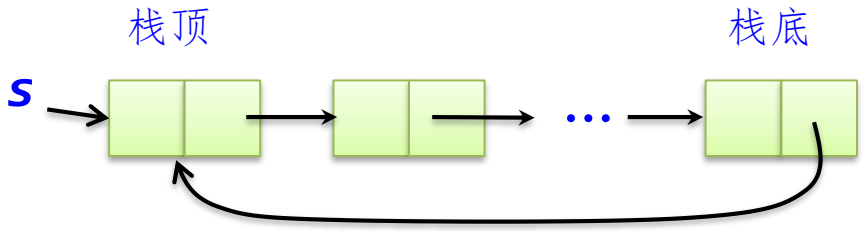
（3）带头结点的双链表

（1）



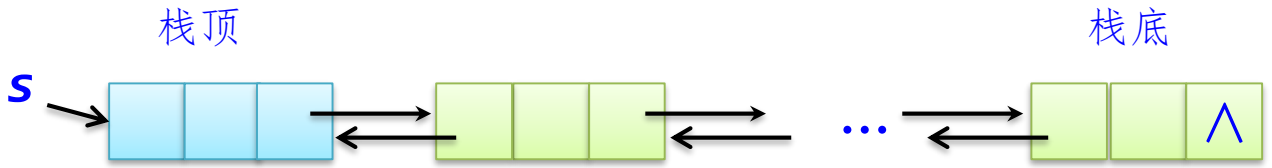
push和pop均为O(1)

（2）



push和pop均为O(*n*)

（3）



push和pop均为O(1)

（1）最好！

问题2：在一个算法中需要建立多个栈时可以选用以下三种方案之一，试问这三种方案之间相比各有什么优缺点？

（1）分别用多个顺序存储空间建立多个独立的顺序栈。

（2）多个栈共享一个顺序存储空间。

（3）分别建立多个独立的链栈。

答：（1）优点是每个栈仅用一个顺序存储空间时，操作简单。缺点是分配空间小了，容易产生溢出，分配空间大了，容易造成浪费，各栈不能共享空间。

（2）优点是多个栈仅用一个顺序存储空间，充分利用了存储空间，只有在整个存储空间都用完时才会产生溢出。缺点是当栈个数大于等于3时其中一个栈满时需要向左、右查询有无空闲单元的过程复杂且十分耗时。

（3）优点是多个链栈一般不考虑栈的溢出，采用动态空间分配具有良好的适应性。缺点是栈中元素要以指针相链接，比顺序存储多占用了存储空间。

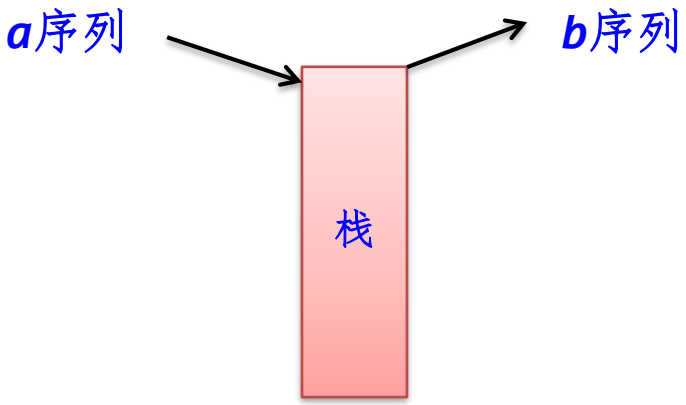
3.1.5 链栈的应用算法设计示例

【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所有元素逆置。例如链栈st中元素从栈底到栈顶为（1，2，3，4），逆置后为（4，3，2，1）。

解：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中所有元素并保存在一个数组*a*中，再将数组*a*中所有元素依次进栈。

```
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的所有元素进栈
    st.push(a[j])
  return st
```

【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：

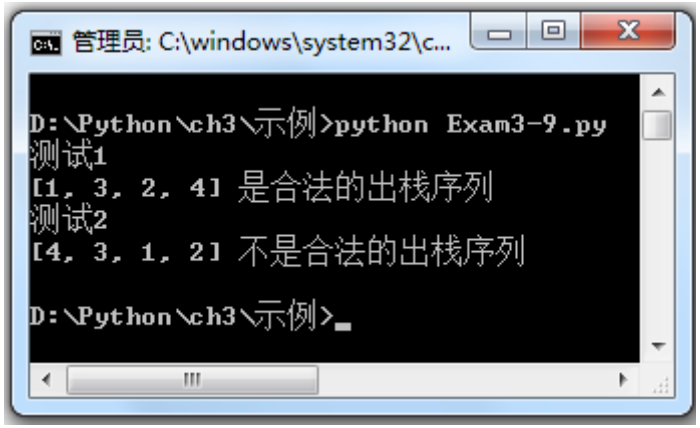
① 将a[i]进栈，i++。

② 栈不空并且栈顶元素与b[j]相同时循环：出栈元素e，j++。

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。

```
from Linktack import LinkStack    #引用链栈LinkStack
def isSerial(a,b,n):              #判断b是否为a的出栈序列算法
  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")
```



3.1.6 栈的综合应用

求解问题中需要临时保存一些数据元素：

- 先保存的后处理：栈
- 先保存的先处理：队列

简单表达式求值

exp：仅包含“+”、“-”、“*”、“/”、正整数和小括号的合法数学表达式–**中缀表达式**。

后缀表达式：就是运算符在操作数的后面，已经考虑了运算符的优先级，不包含括号，只含操作数和运算符。

1+2*3

123*+

手工转换产生后缀表达式

5+2*(1+6)-8/2

加括号

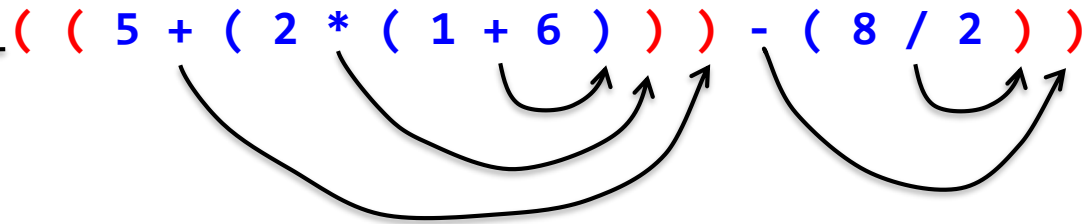
5+(2*(1+6))-8/2

5+(2*(1+6))-(8/2)

(5+(2*(1+6)))-(8/2)

((5+(2*(1+6)))-(8/2))

除括号



5 2 1 6 + * + 8 2 / -

exp求值过程

- 将表达式exp转换成后缀表达式postexp。
- 对该后缀表达式求值。

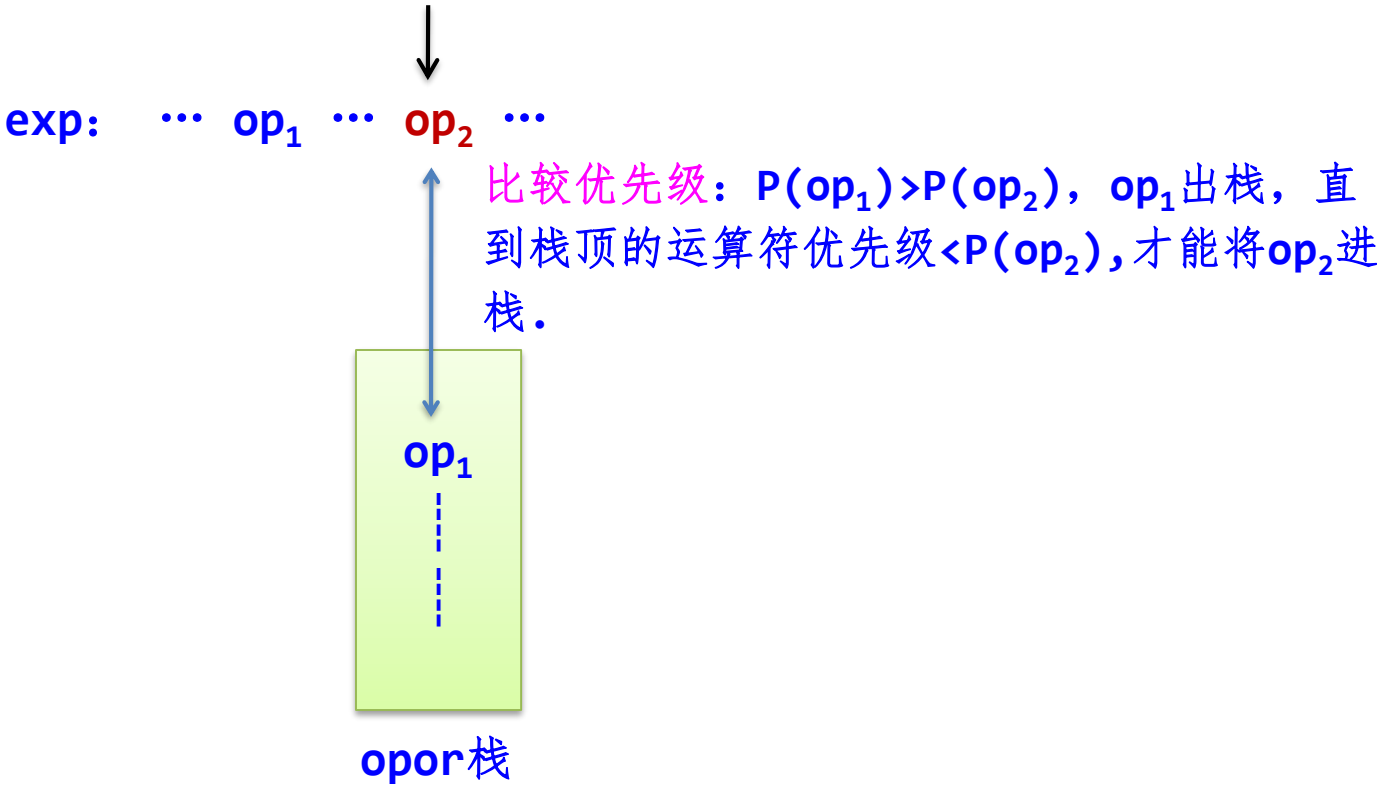
Python列表表示

设计求表达式值的类ExpressClass

```
class ExpressClass:                          #求表达式值类
  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
        …
```

exp ⇨ postexp

使用运算符栈opor：先出栈的运算符先做运算！



转换过程

while (若exp未读完)

{ 从exp读取字符ch;

ch为数字：将后续的所有数字均依次存放到postexp中;

ch为左括号'('：将'('进栈到opor;

ch为右括号')'：将opor栈中'('以前的运算符依次出栈并存放到
postexp中,再将'('退栈;

若ch的优先级高于栈顶运算符优先级，则将ch进栈；否则出栈并存放到
postexp中,再将ch进oper栈;

}

字符串exp扫描完毕,则退栈opor的所有运算符并存放到postexp中

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opor栈		(
56	将56存入postexp中	[56]	(
-	由于opor中’(‘以后没有字符，则直接将'-'进opor栈	[56]	(-
20	将20存入postexp中	[56,20]	(-
)	将栈opor中'(‘以后的运算符依次出栈并存入postexp，然后将'('出栈	[56,20,'-']	
/	将'/'进opor栈	[56,20,'-']	/
(	将'('进opor栈	[56,20,'-']	/(
4	将4存入postexp	[56,20,'-',4]	/(
+	由于opor中'(‘以后没有运算符，则直接将'+'进opor栈	[56,20,'-',4]	/(+
2	将2存入postexp	[56,20,'-',4,2]	/(+
)	将opor栈中'(‘以后的运算符依次出栈并存入postexp，然后将'('出栈	[56,20,'-',4,2,'+']	/
	exp扫描完毕，将opor栈中所有运算符出栈并存入postexp，得到最后的后缀表达式	[56,20,'-',4,2,'+','/']	

```
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“(”之后的运算符退栈
        self.postexp.append(e)          #退栈运算符添加到postexp
      opor.pop()                        #再将(退栈
```

```
        elif ch=="+" or ch=="-":      #判定为加或减号
          while not opor.empty() and opor.gettop()!="(":
            e=opor.pop()              #将栈中不低于ch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"*"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))           #连续数字符转换为整数运算数
    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()                   #将栈中所有运算符退栈并添加到postexp
  self.postexp.append(e)
```

后缀表达式postexp求值

使用运算数栈opand

```
while (若postexp未读完)
{   从postexp中读取一个元素ch;
    ch为数值：将该数值进栈opand;
    ch为'+'：从opand栈出栈两个数值a和b,计算c=b+a;将c进栈opand;
    ch为'-'：从opand栈出栈两个数值a和b,计算c=b-a;将c进栈opand;
    ch为'*'：从opand栈出栈两个数值a和b,计算c=b*a;将c进栈opand;
    ch为'/'：从opand栈出栈两个数值a和b,若a不零,计算c=b/a;将c
        进栈opand;
}
opand栈中唯一的数值即为表达式值
```

后缀表达式[56,20,'-',4,2,'+','/']的求值过程

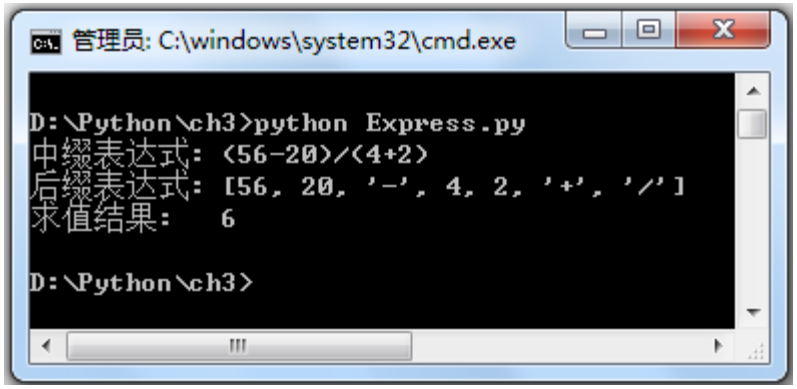
ch序列	说明	st栈
56	遇到56，将56进栈	56
20	遇到20，将20进栈	56,20
'-'	遇到'-'，出栈两次，将56-20=36进栈	36
4	遇到4，将4进栈	36,4
2	遇到2，将2进栈	36,4,2
'+'	遇到'+'，出栈两次，将4+2=6进栈	36,6
'/'	遇到'/'，出栈两次，将36/6=6进栈	6
	postexp扫描完毕，算法结束，栈顶数值6即为所求	

```
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
```

主程序

```
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()))
```



示例

2018年全国硕士研究生入学统一考试题

1、若栈S_1中保存整数，栈S_2中保存运算符，函数F()依次执行下述各步操作：

（1）从S_1中依次弹出两个操作数*a*和*b*；

（2）从S_2中弹出一个运算符op；

（3）执行相应的运算*b* op *a*；

（4）将运算结果压入S_1中。

假定S_1中的操作数依次是5，8，3，2（2在栈顶），S_2中的运算符依次是*，-，+（+在栈顶）。调用3次F()后，S1栈顶保存的值是（　）。

A.-15　　B.15　　C.-20　　D.20

按求后缀表达式值的过程操作！

示例

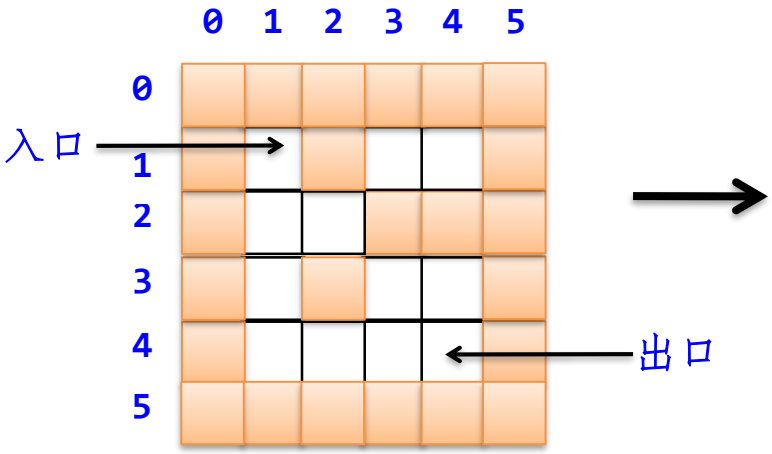
2014年全国硕士研究生入学统一考试题

2．假设栈初始为空，将中缀表达式*a/b+(c*d-e*f)/g*转换为等价的后缀表达式的过程中， 当扫描到*f*时，栈中的元素依次是（ ）。

A．+ (* -　　B．+ (- *　　C．/ + (* - *　　D．/ + - *

按中缀转换为后缀表达式的过程操作！

求迷宫问题

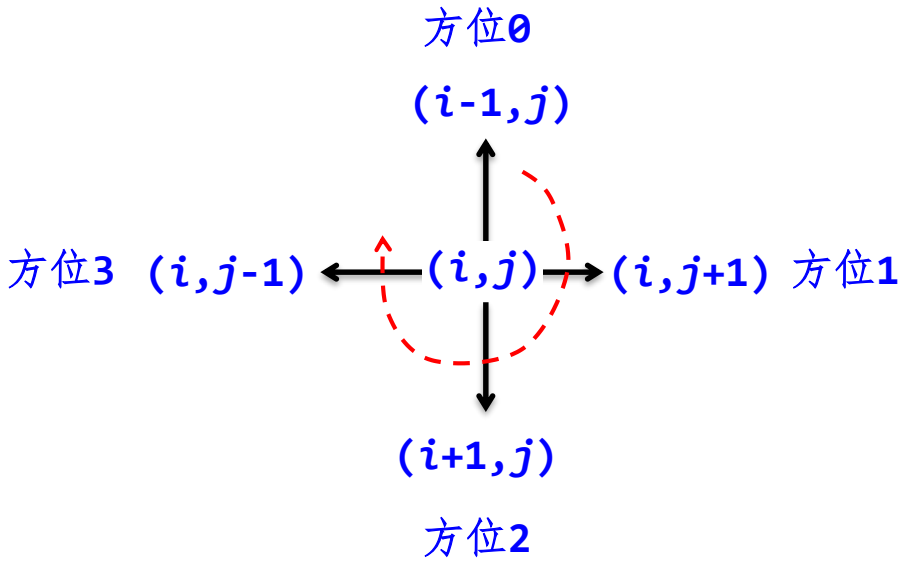


一个迷宫图

```
int [][] 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}
};
```

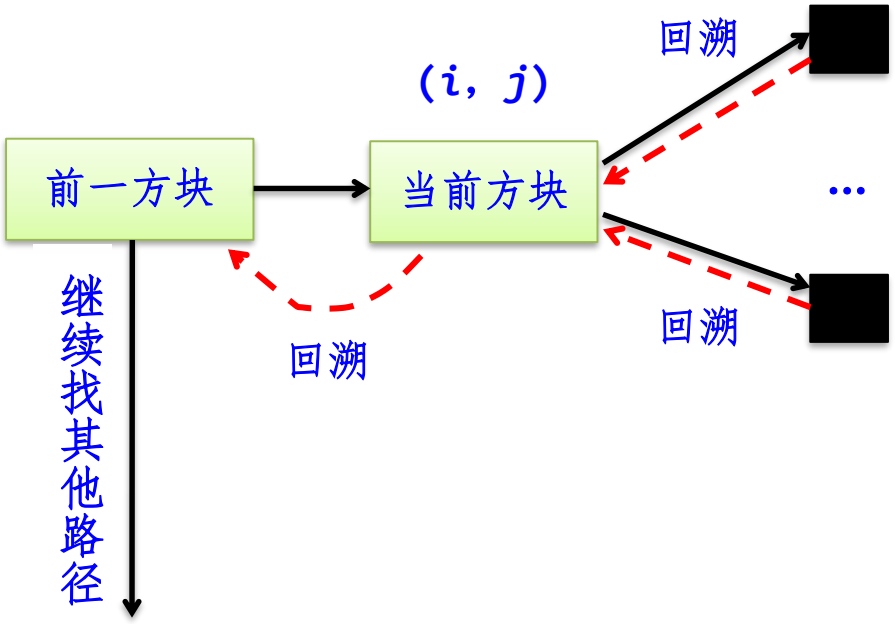
求从入口到出口的一条简单路径

试探顺序



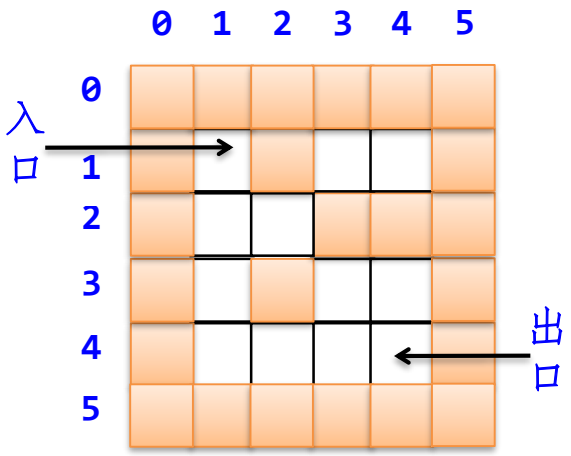

```
dx=[-1,0,1,0]          #x方向的偏移量
dy=[0,1,0,-1]          #y方向的偏移量
```

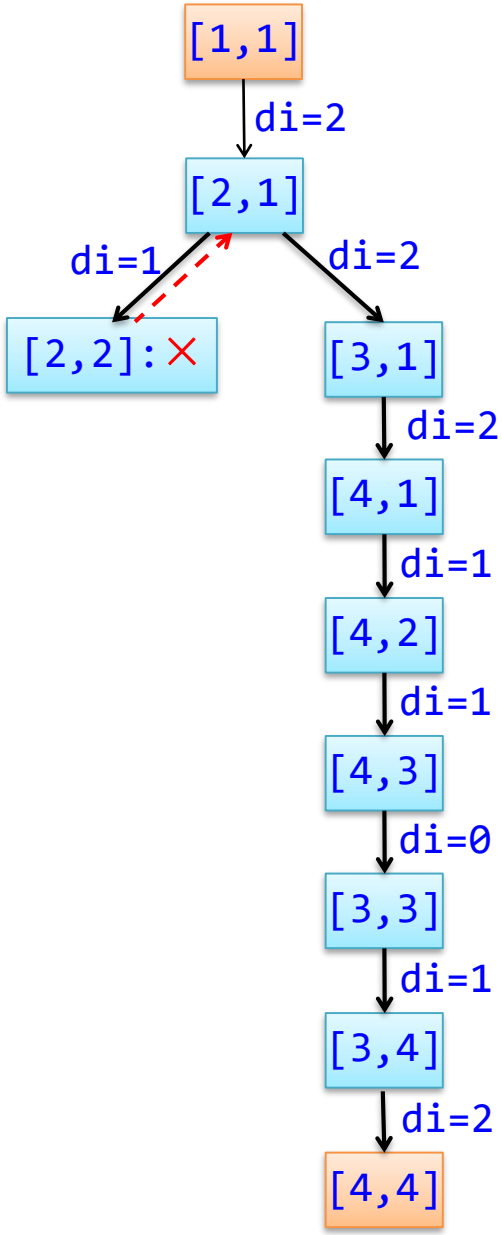
迷宫问题的搜索过程

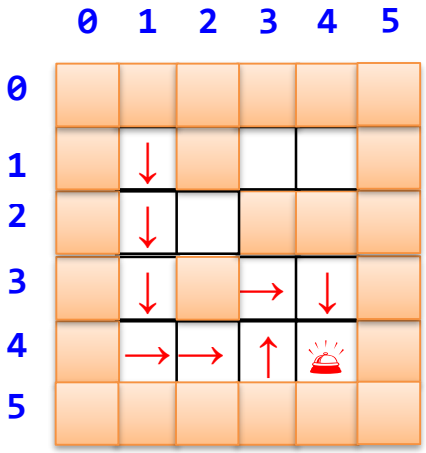


- 用栈记录走过的路径
- 路径：由方块和方块之间的走向（方位）构成


```
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
```


0 1 2 3 4 5
0
1
2
3
4
5
入口
出口


[1,1]
di=2
[2,1]
di=1
di=2
[2,2]:×
[3,1]
di=2
[4,1]
di=1
[4,2]
di=1
[4,3]
di=0
[3,3]
di=1
[3,4]
di=2
[4,4]

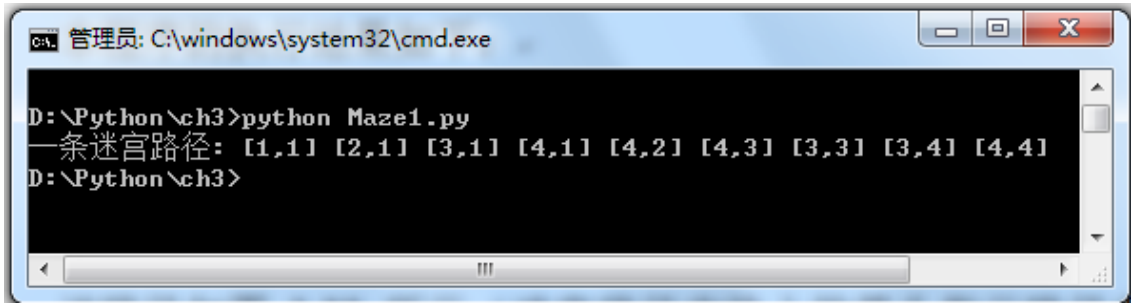

0 1 2 3 4 5
0
1
2
3
4
5

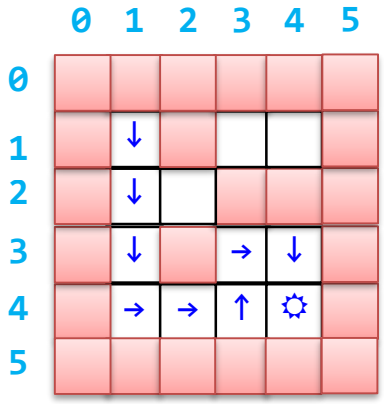
```
def mgpath(xi,yi,xe,ye):                #求一条从(xi,yi)到(xe,ye)的迷宫路径
  global mg                             #迷宫数组为全局变量
  st=SqStack()                          #定义一个顺序栈
  dx=[-1,0,1,0]                         #x方向的偏移量
  dy=[0,1,0,-1]                         #y方向的偏移量
  e=Box(xi,yi,-1)                       #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
    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
```

设计主程序

```
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")
```





- 为什么找到的路径不一定是最短路径？
- 如何求所有的迷宫路径？