linear structure
线性结构

What is linear structure

什么是线性结构Linear Structure

linear structure is a set of ordered data items, where each data item has a unique precursor and successor

线性结构是一种有序数据项的集合 ，其中每个数据项都有唯一的前驱和后继

Except the first has no precursor, the last has no successor

除了第一个没有前驱，最后一个没有后继

When a new data item is added to the data set,it will only add before or after the original data item

新的数据项加入到数据集中时，只会加入到原有某个数据项之前或之后

Data set with this property, is called a linear structure

具有这种性质的数据集，就称为线性结构

What is linear structure

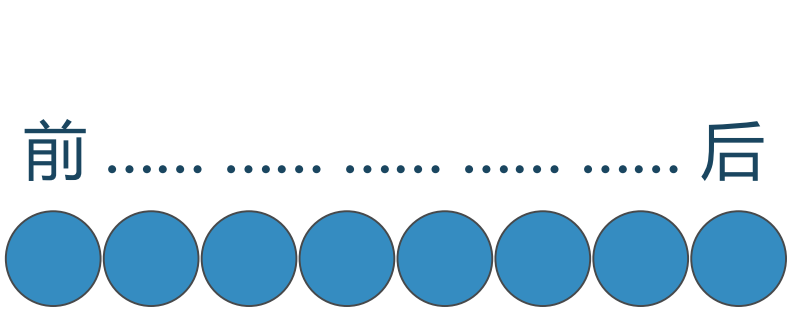
什么是线性结构Linear Structure

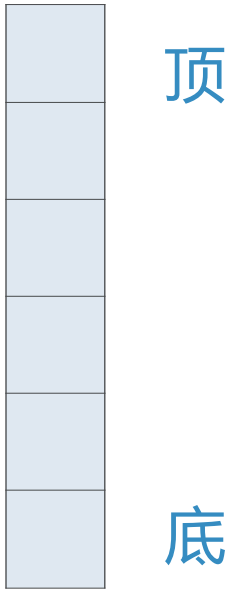
Linear structure always has two ends, in different cases, the names of the two ends are also different

线性结构总有两端 ，在不同的情况下，两端的称呼也不同

Sometimes called "left" "right" end, "front" "back" end, or "top" and "base" End

有时候称为“左” “右”端、 “前” “后”端、 “顶” “底”端





What is linear structure

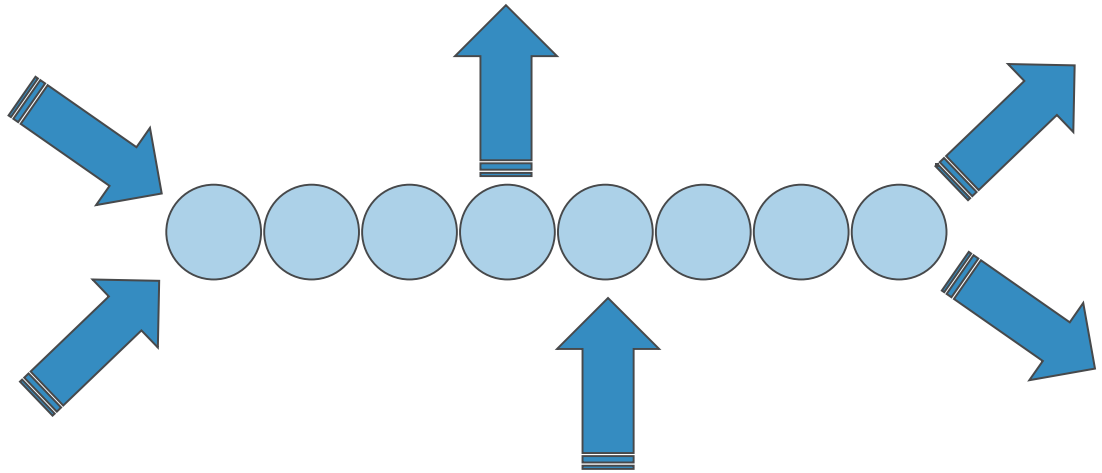
什么是线性结构Linear Structure

The name at both ends is not the key, the key difference between different linear structures is in the way the data items increase or decrease

两端的称呼并不是关键 ，不同线性结构的关键区别在于数据项增减的方式

Some structures only allow the data items to be added from one end, while others allow them to be removed from both ends

有的结构只允许数据项从一端添加，而有的结构则允许数据项从两端移除



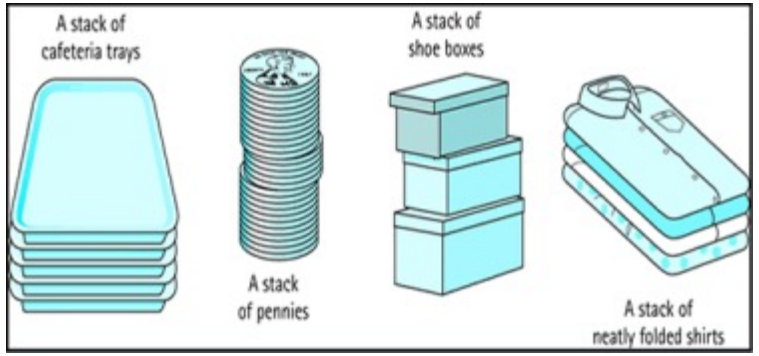
What is linear structure
什么是线性结构Linear Structure

We started with the 4 simplest but powerful structures to investigate the data structure: Stack, Queue, two-end Deque, and list

我们从4个最简单但功能强大的结构入手，开始研究数据结构，栈Stack ， 队列Queue ，双端队列Deque 和列表List

What these datasets have in common is that there are only sequential relationships between the data items, and all of them are linear structures

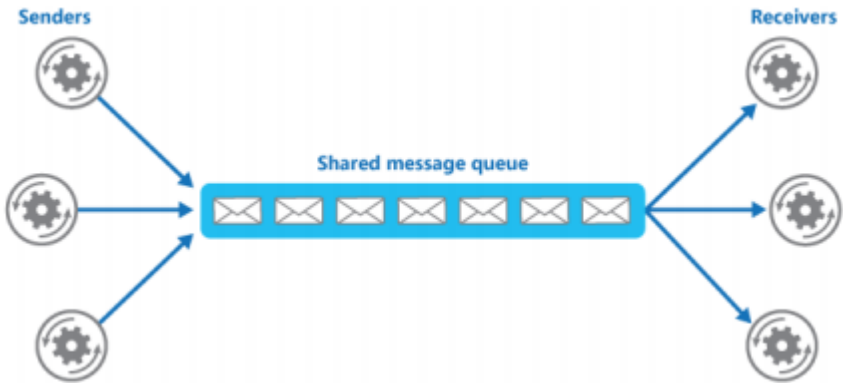
这些数据集的共同点在于，数据项之间只存在先后的次序关系，都是线性结构

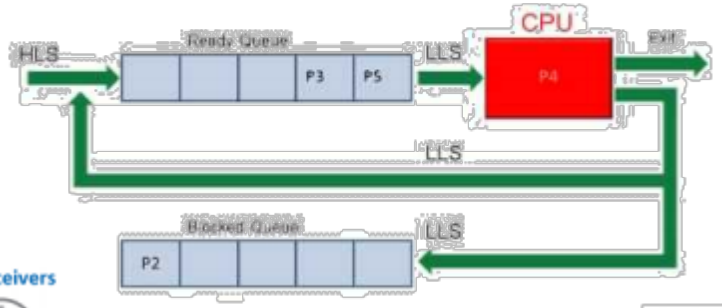


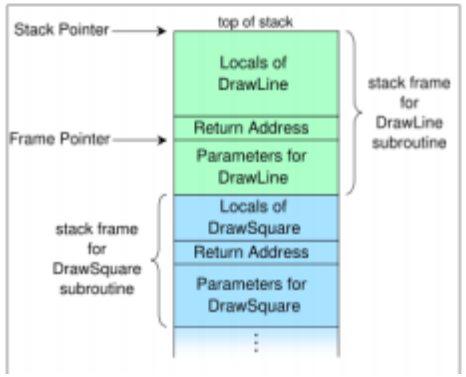
What is linear structure
什么是线性结构Linear Structure

These linear structures are the most widely used data structures,
这些线性结构是应用最广泛的数据结构，

They appear in various algorithms, used to solve a large number of important problems
它们出现在各种算法中，用来解决大量重要问题







Stack abstract data type and Python implementation

栈抽象数据类型及Python实现

Stack : What is a stack?

栈Stack：什么是栈？

A sequential set of data items, in which both the addition and removal of data items occur only at the same end

一种有次序的数据项集合，在栈中，数据项的加入和移除都仅发生在同一端

This end is called the stack "top", and the other end is called the stack "Bottom"

这一端叫栈“顶top”，另一端叫栈“底Bottom”

There are many applications of stacks in your daily life

生活中有很多栈的应用

Plates, trays, stacks of books, etc

盘子、托盘、书堆等等

Stack: What is a stack?
栈Stack：什么是栈？

The closer a data item to the bottom of the stack, the longer they remain in the stack

距离栈底越近的数据项，留在栈中的时间就越长

The latest data items added to the stack will be removed first

而最新加入栈的数据项会被最先移除

This order is often called "LIFO": Last in First out

这种次序通常称为 “后进先出LIFO” ： Last in First out

This is an order based on the preservation time of the data items: the shorter the time,the closer to the top of the stack, and the longer the time, the closer to the bottom of the stack

这是一种基于数据项保存时间的次序，时间越短的离栈顶越近，而时间越长的离栈底越近

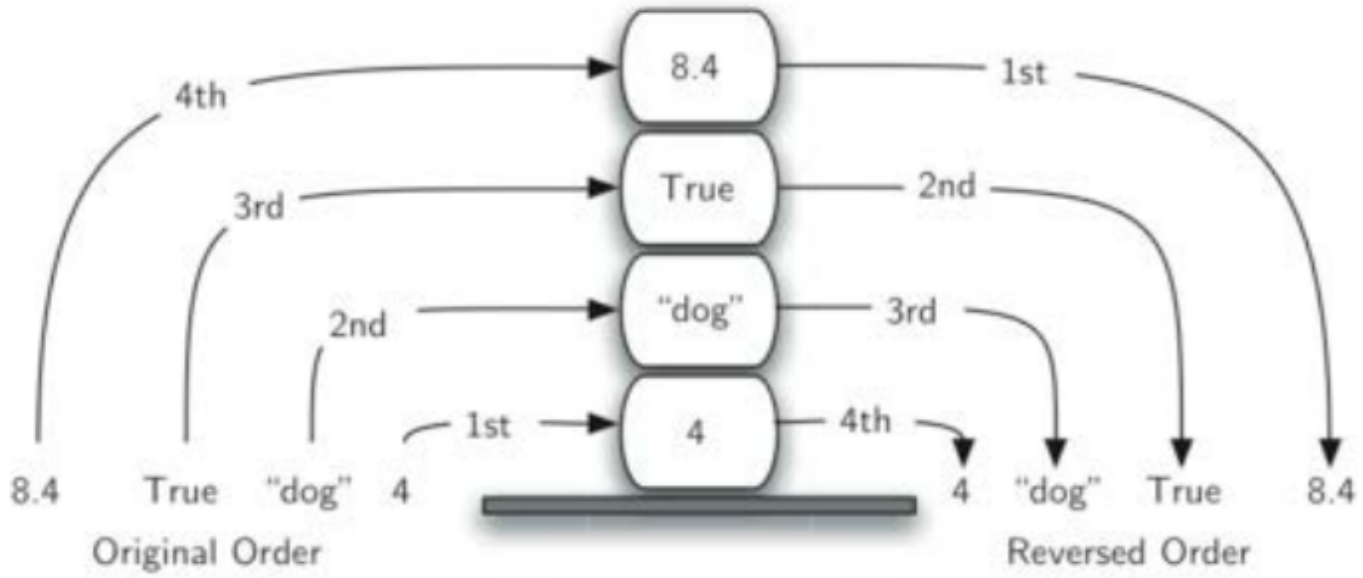
Characteristics of the stack: reverse the order
栈的特性：反转次序

We observe a stack formed by a mixture of python native data objects

我们观察一个由混合的python原生数据对象形成的栈

In and out of the stack, are in the opposite order

进栈和出栈的次序正好相反



Characteristics of the stack: reverse the order
栈的特性：反转次序

This kind of access order reverse feature, we have also encountered in some computer operations

这种访问次序反转的特性 ，我们在其他许多计算机操作上页碰到过

The "back back" button of browser, the first back is the recently visited web page

浏览器的“后退back”按钮，最先back的是最近访问的网页

Word "Undo" button, the first to revoke is the most recent operation

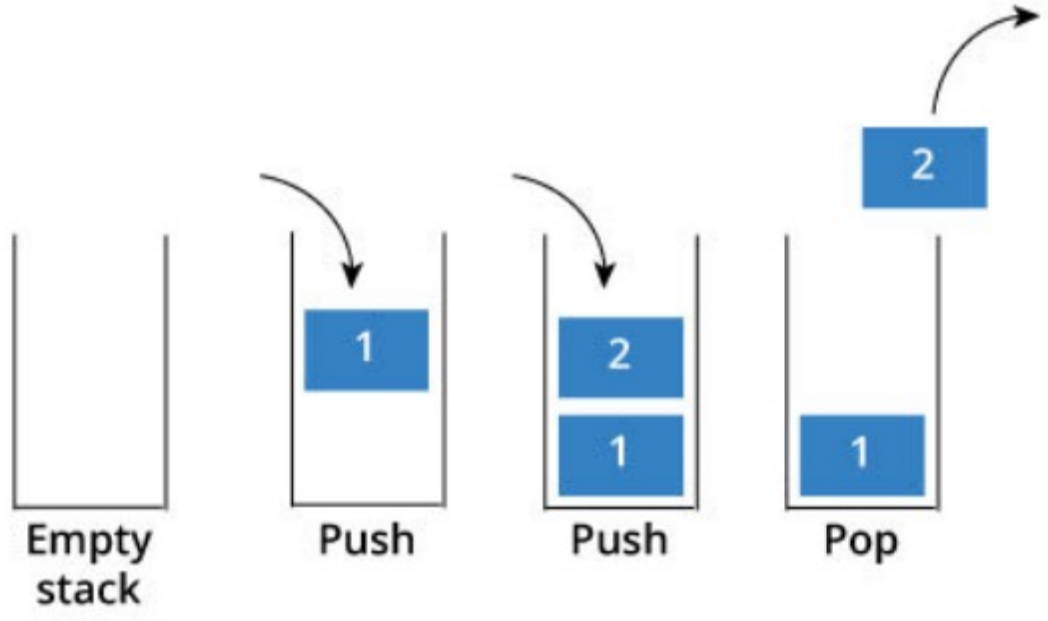
Word的“Undo”按钮，最先撤销的是最近操作

The Abstract Datatype (ADT), Stack

抽象数据类型Stack

The abstract data type "stack" is a sequential data set, and each data item is only added from the top of the stack and removed from the data set, with the characteristics of backward, first-out LIFO

抽象数据类型“栈”是一个有次序的数据集， 每个数据项仅从“栈顶”一端加入到数据集中、从数据集中移除 ，栈具有后进先出LIFO的特性



The Abstract Datatype, Stack
抽象数据类型Stack

The abstract data type "stack" is defined as the following action
抽象数据类型“栈”定义为如下的操作

Stack(): Create an empty stack with no data items

Stack()：创建一个空栈，不包含任何数据项

push(item): Add item to the top of the stack without return value

push(item)：将item加入栈顶，无返回值

pop(): removes the top of the stack data item,and returns, and the stack is modified

pop()：将栈顶数据项移除，并返回，栈被修改

peek(): Pop into the stack top data item, returns the stack top data item but is not removed, and the stack is not modified

peek()：“窥视”栈顶数据项，返回栈顶的数据项但不移除，栈不被修改

isEmpty(): Whether the return stack is an empty stack

isEmpty()：返回栈是否为空栈

size (): How many data items are there in the return stack

size()：返回栈中有多少个数据项

Abstract data type Stack: Operation Sample

抽象数据类型Stack：操作样例

Stack Operation	Stack Contents	Return Value
s= Stack()	[]	Stack object
s.isEmpty()	[]	True
s.push(4)	[4]	
s.push('dog')	[4,'dog']	
s.peek()	[4,'dog']	'dog'
s.push(True)	[4,'dog',True]	
s.size()	[4,'dog',True]	3
s.isEmpty()	[4,'dog',True]	False
s.push(8.4)	[4,'dog',True,8.4]	
s.pop()	[4,'dog',True]	8.4
s.pop()	[4,'dog']	True
s.size()	[4,'dog']	2

Implementing ADT Stack in Python
用Python实现ADT Stack

After clearly defining the abstract data type Stack, we see how to implement it with Python

在清楚地定义了抽象数据类型Stack之后 ，我们看看如何用Python来实现它

Python object-oriented mechanism that can be used to implement user-defined types

Python的面向对象机制 ，可以用来实现用户自定义类型

- **Implement ADT Stack as a Class of Python**

  将ADT Stack实现为Python的一个Class

- **Implement the operation of ADT Stack as the method of Class**

  将ADT Stack的操作实现为Class的方法

- **Since Stack is a dataset, we can use Python's native dataset to implement, and we chose the most common dataset used by List**

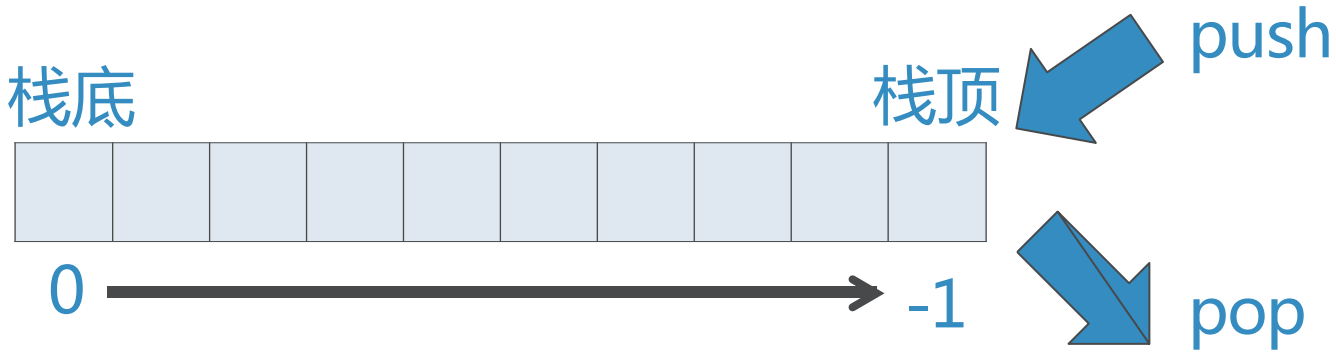
  由于Stack是一个数据集，所以可以采用Python 的原生数据集来实现，我们选用最常用的数据集 List来实现

Implementing ADT Stack in Python
用Python实现ADT Stack

One detail: both ends of the Stack correspond to the list settings
一个细节： Stack的两端对应list设置

You can set either end of the List (index =0 or-1) to the top of the stack
可以将List的任意一端(index=0或者-1)设置为栈顶

We choose the end of the List (index = -1) as the top of the stack, so that the operation of the stack can be achieved by the append and pop of the list, very simple!
我们选用List的末端(index=-1)作为栈顶，这样栈的操作就可以通过对list的append和pop 来实现，很简单！



Implementing ADT Stack in Python

用Python实现ADT Stack

```
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items)-1]

    def size(self):
        return len(self.items)
```

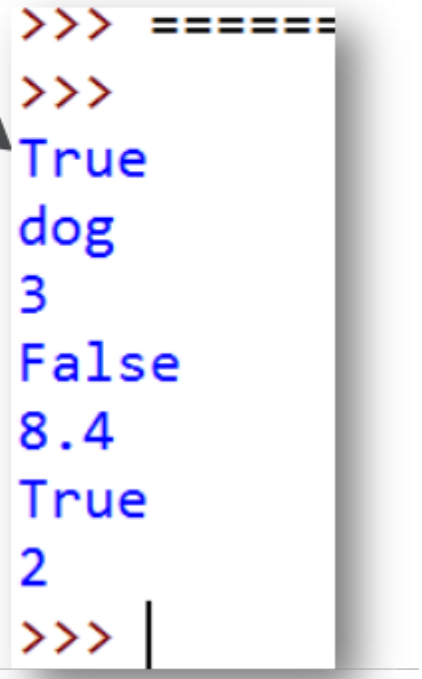
The Stack test code

Stack测试代码

```
from pythonds.basic.stack import Stack

s=Stack()

print(s.isEmpty())
s.push(4)
s.push('dog')
print(s.peek())
s.push(True)
print(s.size())
print(s.isEmpty())
s.push(8.4)
print(s.pop())
print(s.pop())
print(s.size())
```

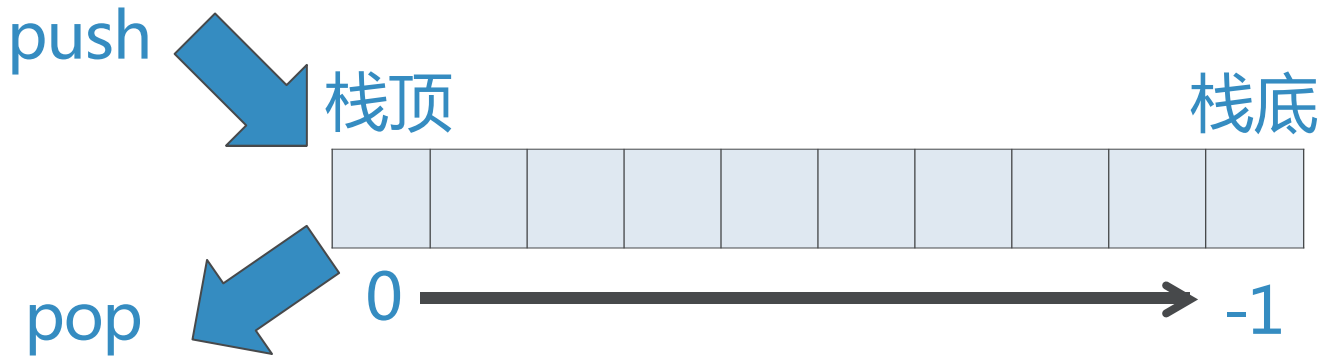


Another implementation of the ADT Stack

ADT Stack的另一个实现

If we use the other end of the List (the initial end index =0) as the top of the Stack, we can also achieve the Stack

如果我们把List的另一端（首端index=0） 作为Stack的栈顶 ，同样也可以实现Stack



Another implementation of the ADT Stack
ADT Stack的另一个实现

The stability of interface allows for different implementations of the ADT
接口的稳定性支持了不同ADT的实现方案

The performance is different, the top version of the top end (left), the push / pop complexity is O(n), while the implementation of the top end of the stack (right),
the push / pop complexity is O(1)
性能有所不同，栈顶首端的版本(左)，其push/pop的复杂度为O(n)，而栈顶尾端的实现(右)，其push/pop的复杂度为O(1)

```
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.insert(0,item)

    def pop(self):
        return self.items.pop(0)

    def peek(self):
        return self.items[0]

    def size(self):
        return len(self.items)
```

```
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items)-1]

    def size(self):
        return len(self.items)
```

Application of the stack: simple bracket matching

栈的应用：简单括号匹配

Application of the stack: simple bracket matching
栈的应用：简单括号匹配

we have all written expressions like this:

我们都写过这样的表达式：

```
(5+6)*(7+8)/(4+3)
```

Here the parentheses are used to specify the computational priority of the expression terms

这里的括号是用来指定表达式项的计算优先级

Some functional languages, such as Lisp, use a lot of parentheses when the function is defined

有些函数式语言，如Lisp，在函数定义的时候会用到大量的括号

For example: (defun square (n)
(*n n))

比如：

```
(defunsquare(n)
(*nn))
```

This sentence defines a function that computes the square values

这个语句定义了一个计算平方值的函数

Application of the stack: simple bracket matching
栈的应用：简单括号匹配

of course, the use of parentheses must follow the "balance" rule
当然，括号的使用必须遵循“平衡”规则

Firstly, each open bracket should correspond to exactly a closed bracket;
首先，每个开括号要恰好对应一个闭括号；

Secondly, each pair of open and closed brackets should be correctly nested
其次，每对开闭括号要正确的嵌套

Correct parentheses: (()()()()), ((()())), (()((())()))
正确的括号：(()()()()),((()())),(()((())()))

Wrong brackets: ((((()), ())), (()()(()
错误的括号：((((()),())),(()()(()

The identification of whether the brackets match correctly is the basic algorithm for many language compilers
对括号是否正确匹配的识别，是很多语言编译器的基础算法

Application of the stack: simple bracket matching
栈的应用：简单括号匹配

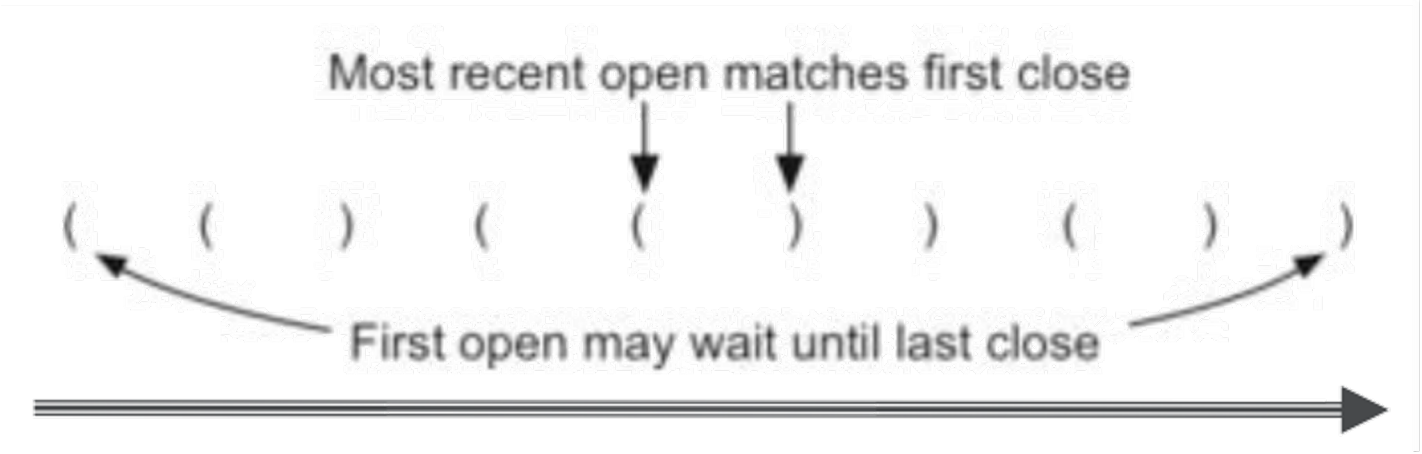
Now look at how to construct a bracket matching recognition algorithm
下面看看如何构造括号匹配识别算法

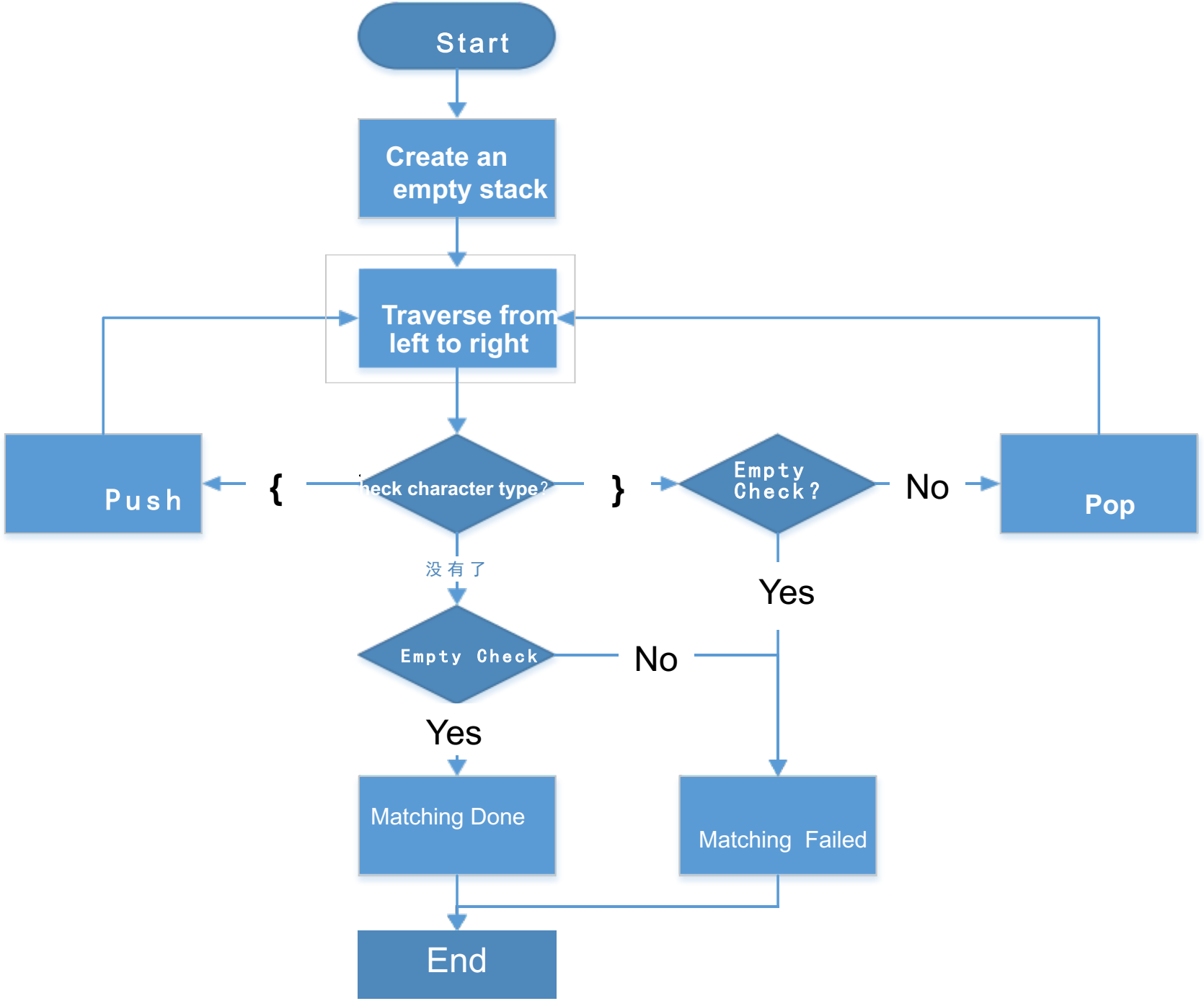
Scan the bracket string from left to right, the latest opened left bracket, should matches the right bracket first encountered.

从左到右扫描括号串，最新打开的左括号，应该匹配最先遇到的右括号

Thus, the first left bracket (first opened) should match the last right bracket (last encountered). This kind of order reversal identification, just accords with the characteristics of the stack!

这样，第一个左括号(最早打开)，就应该匹配最后一个右括号(最后遇到)这种次序反转的识别，正好符合栈的特性！




Start
Create an empty stack
Traverse from left to right
Push
{
heck character type?
}
Empty Check?
No
Pop
Yes
没有了
Empty Check
No
Yes
Matching Done
Matching Failed
End

```
from pythonds.basic.stack import Stack

def parChecker(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol == "(":
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                s.pop()

        index = index + 1

    if balanced and s.isEmpty():
        return True
    else:
        return False

print(parChecker('((()))'))
print(parChecker('(()'))
```

Match of more brackets
更多种括号的匹配

In practical applications, we will encounter more parentheses
在实际的应用里，我们会碰到更多种括号

square brackets, "[]", as used for the list in python
如python中列表所用的方括号“[]”
"{}", the curly brackets used in the dictionary
字典所用的花括号“{}”
parentheses "()" used for tuples and expressions
元组和表达式所用的圆括号“()”

These different parentheses may be mixed together, so it is important to note the respective open and closing matches
这些不同的括号有可能混合在一起使用，因此就要注意各自的开闭匹配情况

Match of more brackets
更多种括号的匹配

The following of these are matches:
下面这些是匹配的

```
{{([][])}()}
```

```
[[{{(())}}]]
```

```
[][][]({})
```

The following are mismatches:

下面这些是不匹配的

```
([)](())])
```

```
[{()]
```

General bracket matching algorithm: code
通用括号匹配算法：代码

Where to is needed

需要修改的地方

Encounter a variety of left brackets still into the stack

碰到各种左括号仍然入栈

When you encounter various right brackets, you need to judge whether the left bracket on the top of the stack belongs to the right bracket the same kind

碰到各种右括号的时候需要判断栈顶的左括号是否跟右括号属于同一种类

```
def parChecker(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol == "(":
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                s.pop()

        index = index + 1

    if balanced and s.isEmpty():
        return True
    else:
        return False
```

```
from pythonds.basic.stack import Stack

def parChecker(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol in "([{":
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                top = s.pop()
                if not matches(top,symbol):
                    balanced = False
        index = index + 1
    if balanced and s.isEmpty():
        return True
    else:
        return False

def matches(open,close):
    opens = "([{"
    closers = ")]}"
    return opens.index(open) == closers.index(close)


print(parChecker('{{([][])}()}'))
print(parChecker('[{()]'))
```

General parenthesis matching algorithm

通用括号匹配算法

The HTML / XML documents also have open and closed tags similar to the parentheses, and the verification and operation of such hierarchical structured documents can also be achieved through the stack

HTML/XML文档也有类似于括号的开闭标记，这种层次结构化文档的校验、操作也可以通过栈来实现

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Hello World</title>
6  </head>
7  <body>
8  <h1>Hello!</h1>
9  </body>
10 </html>
```

Application of the stack: decimal conversion to binary

栈的应用：十进制转换为二进制

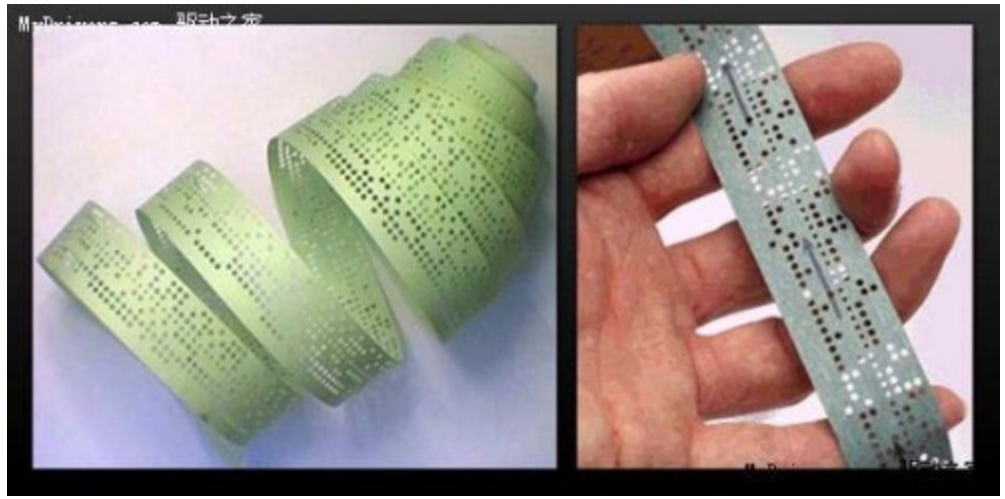
Decimal converted to binary
十进制转换为二进制

Binary is the most basic concept in the computer principle. As the logic gate circuit that constitutes the most basic component of a computer, its input and output are only two states: 0 and 1

二进制是计算机原理中最基本的概念，作为组成计算机最基本部件的逻辑门电路，其输入和输出均仅为两种状态：0和1

But the decimal system is the most basic numerical concept in the traditional human culture, and without the transition between the decimal systems, people interacting with computers will be quite difficult

但十进制是人类传统文化中最基本的数值概念，如果没有进制之间的转换，人们跟计算机的交互会相当的困难



Decimal converted to binary
十进制转换为二进制

"Inession" is how many characters are used to represent integers

所谓的“进制”，就是用多少个字符来表示整数

Decimal is 0~9 these ten digital characters, binary is 0,1 two characters

十进制是0~9这十个数字字符，二进制是0、1两个字符

We often need to convert integers between binary and decimal

我们经常需要将整数在二进制和十进制之间转换

Such as: $(233)_{10}$The corresponding binary number of is $(11101001)_2$

Here is this:

如：$(233)_{10}$的对应二进制数为$(11101001)_2$，具体是这样：

$(233)_{10}=2\times10^2+3\times10^1+3\times10^0$

$(11101001)_2=1\times2^7+1\times2^6+1\times2^5+0\times2^4+1\times2^3+0\times2^2+0\times2^1+1\times2^0$

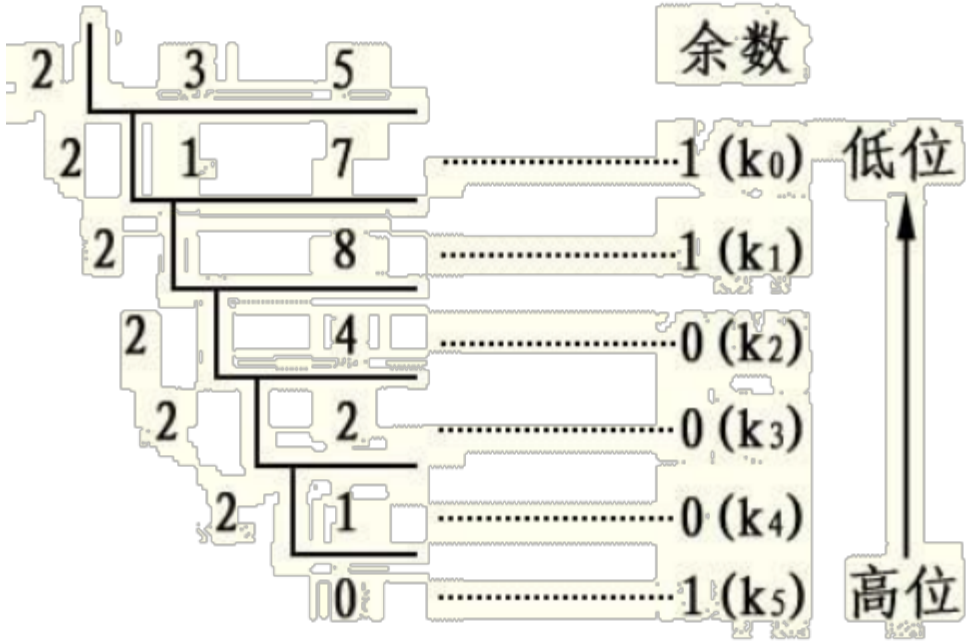
Decimal converted to binary
十进制转换为二进制

The decimal is converted to binary using the algorithm "divided by 2 for remainder"

十进制转换为二进制，采用的是“除以2求余数”的算法

Divide the integer by 2, the remainder obtained each time is the binary from low to high

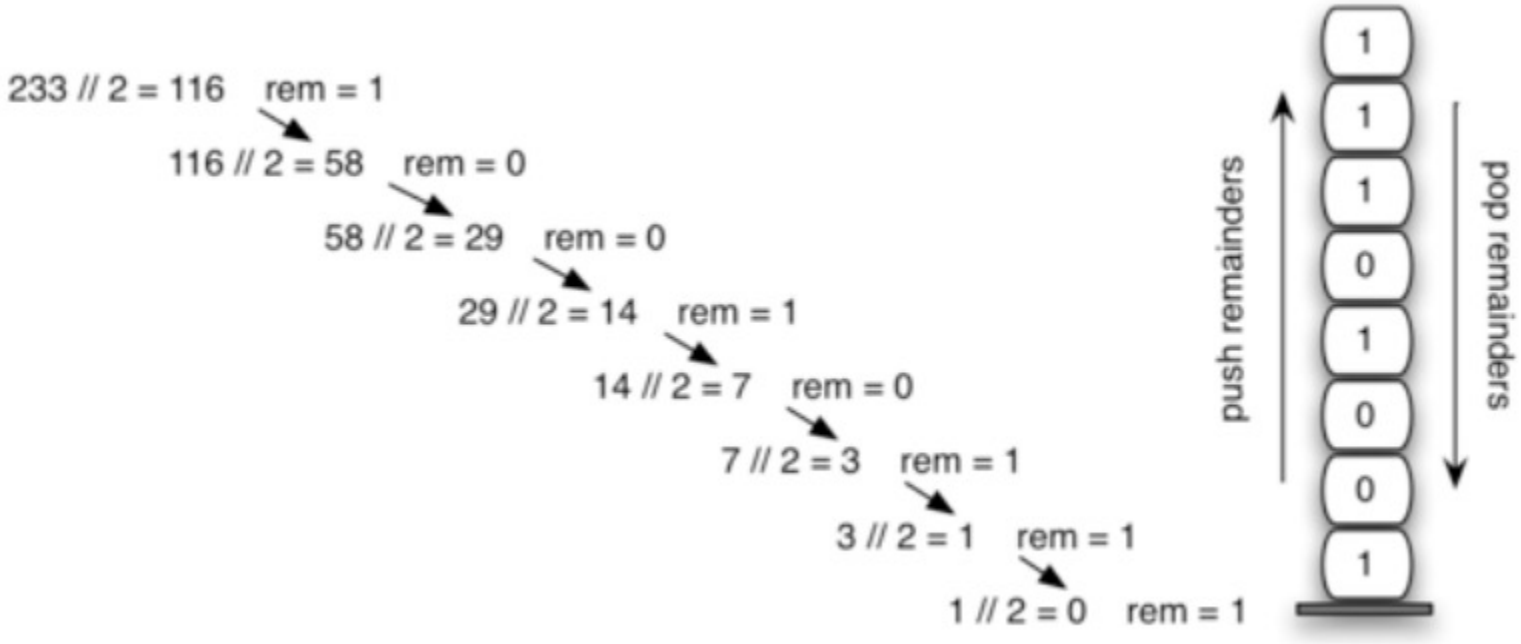
将整数不断除以2，每次得到的余数就是由低到高的二进制位



Decimal converted to binary
十进制转换为二进制

The process of "dividing by 2" results in the remainder being obtained in the order from low to high, while the output is obtained from high to low, so that a stack is needed to reverse the order

“除以2”的过程，得到的余数是从低到高的次序，而输出则是从高到低，所以需要一个栈来反转次序



Decimal converted to binary: code
十进制转换为二进制：代码

```
from pythonds.basic.stack import Stack

def divideBy2(decNumber):
    remstack = Stack()

    while decNumber > 0:
        rem = decNumber % 2
        remstack.push(rem)
        decNumber = decNumber // 2

    binString = ""
    while not remstack.isEmpty():
        binString = binString + str(remstack.pop())

    return binString

print(divideBy2(42))
```

For the remainder
求余数

The integer in addition to
整数除

Extend to more precant transformations
扩展到更多进制转换

The algorithm of converting decimal to binary can be easily extended to convert to arbitrary N-adic

十进制转换为二进制的算法，很容易可以扩展为转换到任意N进制

Just change the "divide by 2 for remainder" algorithm to the "divide by N for remainder" algorithm

只需要将“除以2求余数”算法改为“除以N求余数”算法就可以

计算机中另外两种常用的进制是八进制和十六进制

$(233)_{10}$等于$(351)_8$和$(E9)_{16}$

$(351)_8=3\times8^2+5\times8^1+1\times8^0$　　$(E9)_{16}=14\times16^1+9\times16^0$

Extend to more precant transformations
扩展到更多进制转换

The main question is how to represent octal and hexadecimal
主要的问题是如何表示八进制及十六进制

Binary has two different numbers 0,1

二进制有两个不同数字0、1

Decimal has ten different numbers 0,1,2,3,4,5,6, and7、8、9

十进制有十个不同数字0、1、2、3、4、5、6、7、8、9

Octal is available in eight different numbers 0,1,2,3,4,5,6, and 7

八进制可用八个不同数字0、1、2、3、4、5、6、7

The sixteen different numbers of the hexadecimal are 0,1,2,3,4,5,6,7,8,9, A, B, C, D, E, and F

十六进制的十六个不同数字则是0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F

Decimal conversion to any decimal under sixteen: code

十进制转换为十六以下任意进制：代码

```
from pythonds.basic.stack import Stack

def baseConverter(decNumber,base):
    digits = "0123456789ABCDEF"

    remstack = Stack()

    while decNumber > 0:
        rem = decNumber % base
        remstack.push(rem)
        decNumber = decNumber // base

    newString = ""
    while not remstack.isEmpty():
        newString = newString + digits[remstack.pop()]

    return newString

print(baseConverter(25,2))
print(baseConverter(25,16))
```

Expression transformation
表达式转换

infix expression
中缀表达式

We usually see an expression like this: B * C, and it's easy to know that this is B multiplied by C

我们通常看到的表达式象这样：B*C，很容易知道这是B乘以C

This representation of an (operator) in the middle of an (operand) is called the infix notation

这种操作符(operator)介于操作数(operand)中间的表示法，称为“中缀”表示法

But sometimes infix notation causes confusion, such as "A + B * C"

但有时候中缀表示法会引起混淆，如“A+B*C”

Is A + B and then multiplied by C

是A+B然后再乘以C

Or do you go to B * C and then add A?

还是B*C然后再去加A？

Priority in the infix expression
中缀表达式中的优先级

The concept of an operator "priority" has been introduced to eliminate the confusion

人们引入了操作符“优先级”的概念来消除混淆

The operator that specifies a high priority is calculated first

规定高优先级的操作符先计算

Operators of the same priority are calculated from left to right

相同优先级的操作符从左到右依次计算

In this way, A+B*C has no doubt that it is the product of A plus B multiplies C

这样A+B*C就没有疑义是A加上B与C的乘积

Parentheses were also introduced to indicate the mandatory priority, with the highest priority; And in nested parentheses, the inner layer has higher precedence

同时引入了括号来表示强制优先级，括号的优先级最高，而且在嵌套的括号中，内层的优先级更高

This way (A + B) * C is the sum of A plus B, then multiplied by C

这样(A+B)*C就是A与B的和再乘以C

The infix expression in full parenthesis
全括号中缀表达式

While people are used to this representation, computer processing had better to specify all the order of calculations explicitly without handling complex priority rules

虽然人们已经习惯了这种表示法，但计算机处理最好是能明确规定所有的计算顺序，这样无需处理复杂的优先规则

Introduce a full bracket expression: add brackets on both sides of the expression terms

引入**全括号表达式**：在所有的表达式项两边都加上括号

A + B * C + D, which shall be expressed as ((A + (B * C)) + D)

A+B*C+D，应表示为((A+(B*C))+D)

Can you move the position of the operator in the expression slightly?

可否将表达式中操作符的位置稍**移动**一下？

Prefix and postfix expressions
前缀和后缀表达式

For example, the infix expression, A + B
例如中缀表达式A+B

Move the operator to the front to become “+AB”
将操作符移到前面，变为“+AB”

Or move the operator to the end and become “AB+”
或者将操作符移到最后，变为“AB+”

We get two other representations of the expression: "prefix" and "postfix" notation

我们就得到了表达式的另外两种表示法：“前缀”和“后缀”表示法

Defined by the position of the operator relative to the operand
以操作符相对于操作数的位置来定义

Prefix and postfix expressions
前缀和后缀表达式

This way, A + B * C will become the prefix "+ A * BC" and the suffix "ABC *+"

这样A+B*C将变为前缀的“+A*BC”，后缀的“ABC*+”

To help understand, the subexpressions are underlined

为了帮助理解，子表达式加了下划线

中缀表达式	前缀表达式	后缀表达式
A+B	+AB	AB+
A+B*C	+A*BC	ABC*+

Prefix, infix, and ostsuffix expressions
前缀、中缀和后缀表达式

Look at the infix expression "(A + B) * C", according to the conversion rules, the prefix expression is "* + ABC", and the postfix expression is "AB+C *"

再来看中缀表达式“(A+B)*C”，按照转换的规则，前缀表达式是“*+ABC”，而后缀表达AB+C*”

The amazing thing happened: the necessary parentheses in the infix expression disappear in the prefix and the suffix expressions

神奇的事情发生了，在中缀表达式里必须的括号，在前缀和后缀表达式中消失了

In prefix and postfix expressions, the order of operators completely determines the order of calculation without any confusion

在前缀和后缀表达式中，操作符的次序完全决定了运算的次序，不再有混淆

So in many cases, computer representations of expressions should avoid complex infix expressions

所以在很多情况下，表达式的计算机表示都避免用复杂的中缀形式

Prefix, infix, and postfix expressions
前缀、中缀和后缀表达式

See more examples below
下面看更多的例子

infix expression 中缀表达式	prefix expression 前缀表达式	The postfix expression 后缀表达式
A+B*C+D	++A*BCD	ABC*+D+
(A+B)*(C+D)	*+AB+CD	AB+CD+*
A*B+C*D	+*AB*CD	AB*CD*+
A+B+C+D	+++ABCD	AB+C+D+

infix expressions converted to prefix and postfix forms
中缀表达式转换为前缀和后缀形式

So far we have only manually converted a few infix expressions to the forms of prefix and postfix
目前为止我们仅手工转换了几个中缀表达式到前缀和后缀的形式

There must be an algorithm to convert any complex expressions
一定得有个算法来转换任意复杂的表达式

To decompose the complexity of the algorithm, we start with the infix expression in "full brackets" form
为了分解算法的复杂度，我们从“全括号”中缀表达式入手

Let's look at A + B * C, if written in full brackets:
我们看A+B*C，如果写成全括号形式：
(A + (B * C)), the computational order is explicitly expressed
(A+(B*C))，显式表达了计算次序

We note that each pair of brackets, contains a complete set of operators and operands
我们注意到每一对括号，都包含了一组完整的操作符和操作数

infix expressions converted to prefix and postfix forms
中缀表达式转换为前缀和后缀形式

Look at the right bracket of the subexpression (B*C), if you move the operator * to the position of the right bracket, replace it, and then delete the left bracket to get the BC*, which just transforms the infix expression to the postfix form

看子表达式(B*C)的右括号，如果把操作符*移到右括号的位置，替代它，再删去左括号，得到BC*，这个正好把子表达式转换为后缀形式

Then move more operators to the corresponding right bracket

然后再把更多的操作符移动到相应的右括号处

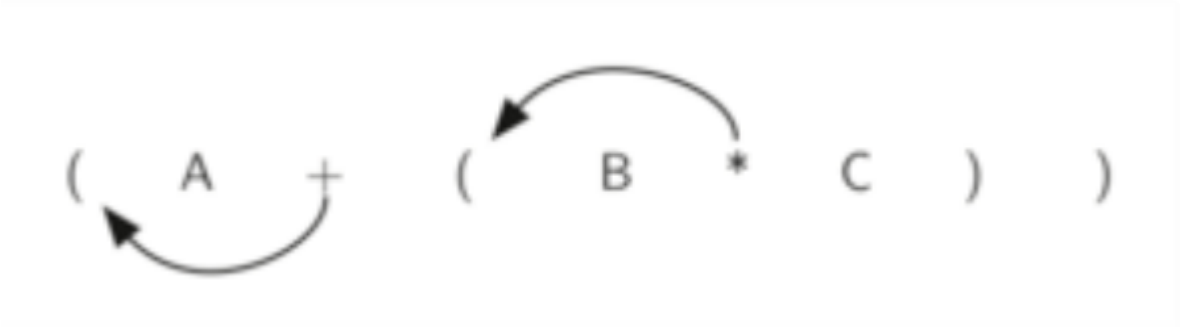
Instead, delete the left bracket, and the entire expression completes the transition to the suffix expression

替代之，再删去左括号，那么整个表达式就完成了到后缀表达式的转换

infix expressions converted to prefix and postfix forms
中缀表达式转换为前缀和后缀形式

Similarly, if we move the operator to the replace the left bracket, and then delete all the right brackets, we also get the prefix expression

同样的，如果我们把操作符移动到左括号的位置替代之，然后删掉所有的右括号，也就得到了前缀表达式



infix expressions converted to prefix and postfix forms
中缀表达式转换为前缀和后缀形式

So, no matter how complex the expression is, it needs to convert to a prefix or a suffix, with only two steps

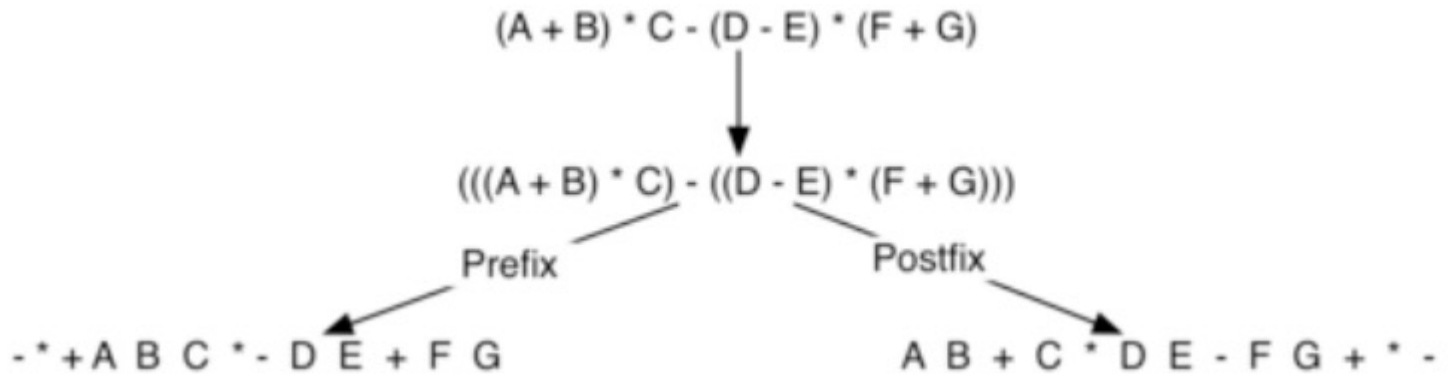
所以说，无论表达式多复杂，需要转换成前缀或者后缀，只需要两个步骤

①Converts the infix expression to full brackets

将中缀表达式转换为全括号形式

②Move all the operators to the left bracket (prefix) or the right bracket (postfix) where the subexpression is located, replace them, and then delete all the parentheses

将所有的操作符移动到子表达式所在的左括号(前缀)或者右括号(后缀)处，替代之，再删除所有的括号



General algorithm to transfer infix to postfix
通用的中缀转后缀算法

We will discuss the general infix-transfer-postfix algorithm
我们来讨论下通用的中缀转后缀算法
First, let's look at the infix expression A+B*C, whose corresponding postfix expression is ABC*+
首先我们来看中缀表达式A+B*C，其对应的后缀表达式是ABC*+

The order of the operand ABC does not change
操作数ABC的顺序没有改变

The order of appearance of the operator, is reversed in the postfix expression
操作符的出现顺序，在后缀表达式中反转了

Due to the higher priority of * than +, the order of operators in the postfix expression coincides with the order of operations.
由于*的优先级比+高，所以后缀表达式中操作符的出现顺序与运算次序一致。

General algorithm to transfer infix to postfix
通用的中缀转后缀算法

In the conversion of a infix expression to a postfix form, the operator outputs later than the operand

在中缀表达式转换为后缀形式的处理过程中，操作符比操作数要晚输出

So before scanning to the corresponding second operand, you need to save the operator first

所以在扫描到对应的第二个操作数之前，需要把操作符先保存起来

These temporary operators, due to the priority rules, may also have to reverse the order of the output.

而这些暂存的操作符，由于优先级的规则，还有可能要反转次序输出。

In A+B*C, Although'+' appears first,the priority is after this* ，so it has to wait for * to be processed before it can be processed again

在A+B*C中，+虽然先出现，但优先级比后面这个*要低，所以它要等*处理完后，才能再处理。

This reversal feature allows us to consider using stacks to save temporarily untreated operators

这种反转特性，使得我们考虑用栈来保存暂时未处理的操作符

General algorithm to transfer infix to postfix
通用的中缀转后缀算法

Look again at the (A + B) * C, the corresponding suffix form is AB +C *

再看看(A+B)*C，对应的后缀形式是AB+ C*

The output of + here is earlier than *, mainly because parentheses raise the priority of + above *

这里+的输出比*要早，主要是因为括号使得+的优先级提升，高于括号之外的*

Reviewing the "full bracket" expression in the previous section, the operator in the postfix expression should appear in the right bracket position corresponding to the left bracket

回顾上节的“全括号”表达式，后缀表达式中操作符应该出现在左括号对应的右括号位置

So when encounter left brackets, mark down, and the priority of later operator is raised.In this way, the operator can be output as soon as it is scanned into the corresponding right bracket

所以遇到左括号，要标记下，其后出现的操作符优先级提升了，一旦扫描到对应的右括号，就可以马上输出这个操作符

General algorithm to transfer infix to postfix
通用的中缀转后缀算法

In summary, in the process of character-by-character scan from left to right

总结下，在从左到右扫描逐个字符扫描中

In the process of fixing expressions, a stack is used to temporarily hold the unprocessed operators

缀表达式的过程中，采用一个**栈**来暂存未处理的操作符

In this way, the operator at the top of the stack is recently temporarily stored, and when a new operator is encountered, you need to compare with the operator at the top of the stack to the next priority, and then be processed.

这样，**栈顶**的操作符就是**最近**暂存进去的，当遇到一个新的操作符，就需要跟栈顶的操作符比较下优先级，再行处理。

General algorithm to transfer infix to suffix: process
通用的中缀转后缀算法：流程

In the following algorithm description, the fixed expression is composed of a series of words (token) separated by spaces

后面的算法描述中，约定中缀表达式是由空格隔开的一系列单词(token)构成，

Operator words include the */+-()

操作符单词包括*/+-()

The operand words are single-letter identifiers like A, B, and etc.

而操作数单词则是单字母标识符A、B、C等。

First, create an empty stack opstack to save the operator temporarily and an empty postfixList to save the suffix expression, converting the infix expression into a word (token) list

首先，创建空栈opstack用于暂存操作符，空postfixList用于保存后缀表达式，将中缀表达式转换为单词(token)列表

```
A+B*C=split=>['A','+','B','*','C']
```

General algorithm to transfer infix to postfix: process
通用的中缀转后缀算法：流程

Scan the infix expression word list from left to right
从左到右扫描中缀表达式单词列表

If the word is an operand, it is added directly to the end of postfix expression list
如果单词是操作数，则直接添加到后缀表达式列表的末尾

If the word is the left bracket "(", press into the top of the opstack stack
如果单词是左括号“(”，则压入opstack栈顶

If the word is the right bracket ")", pop up the top operator of opstack repeatedly and add to the end of the output list. If left bracket is the operator "*/+-", we pushed to the top of the opstack stack
如果单词是右括号“)”，则反复弹出opstack栈顶操作符，加入到输出列表末尾，左括号如果单词是操作符“*/+-”，则压入opstack栈顶

- But before pushing, compare its priority with the top of the stack operator
  但在压入之前，要比较其与栈顶操作符的优先级
- If the top of the stack is higher than or equal to it, the top-of-stack operator is repeatedly popped and added to the end of the output list
  如果栈顶的高于或等于它，就要反复弹出栈顶操作符，加入到输出列表末尾
- until the top of the stack has a lower operator priority than it
  直到栈顶的操作符优先级低于它

General algorithm to transfer infix to postfix: process
通用的中缀转后缀算法：流程

After scanning the infix expression word list, pop all remaining operators in the opstack stack in turn and add them to the end of the output list

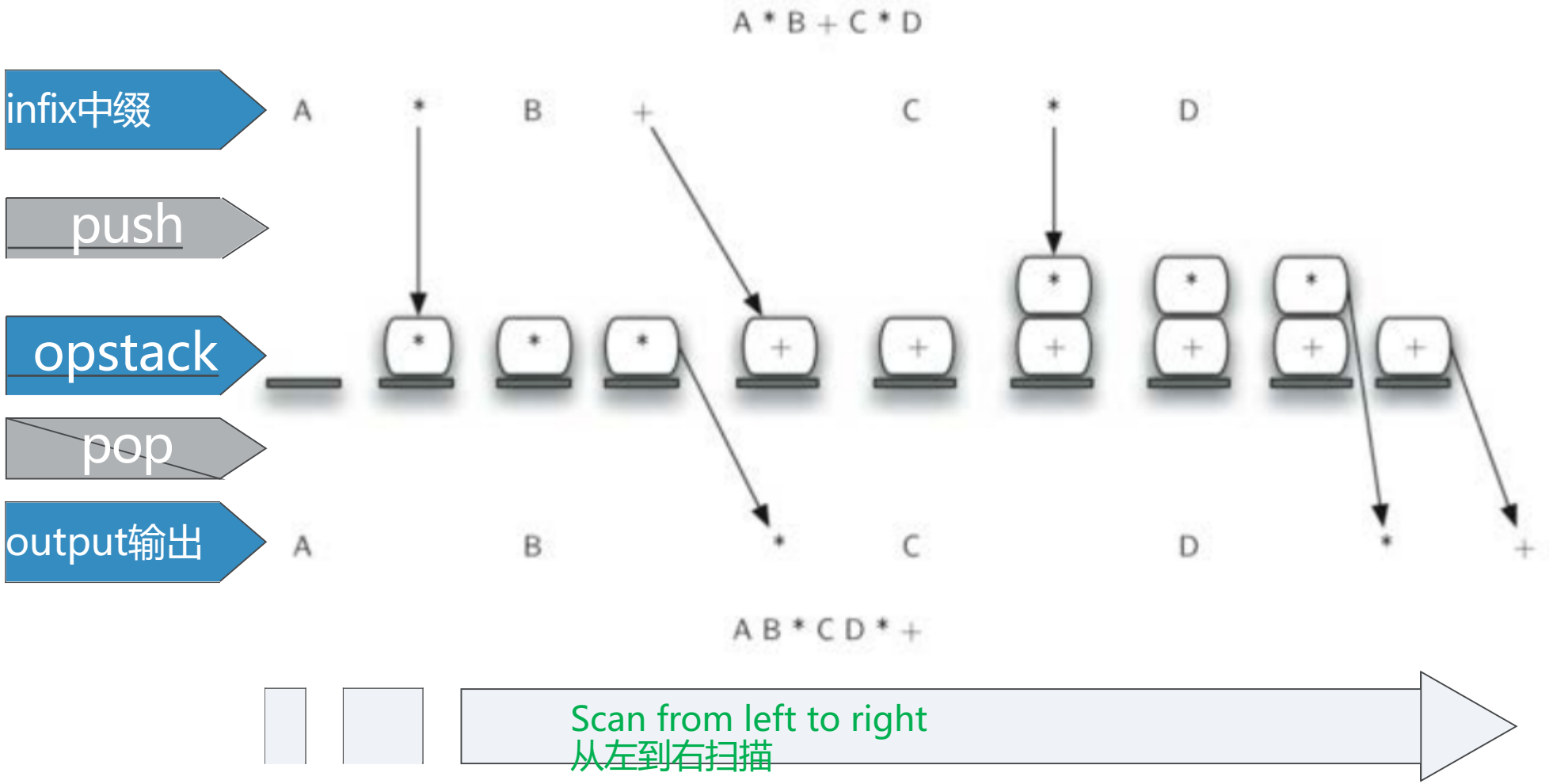
中缀表达式单词列表扫描结束后，把opstack栈中的所有剩余操作符依次弹出，添加到输出列表末尾

The output list is merged by join into a suffix expression string, and after that the algorithm ends.

把输出列表再用join方法合并成后缀表达式字符串，算法结束。

General algorithm to transfer infix to suffix: Example

通用的中缀转后缀算法：实例



code
代码

```
from pythonds.basic.stack import Stack

def infixToPostfix(infixexpr):
    prec = {}
    prec["*"] = 3
    prec["/"] = 3
    prec["+"] = 2
    prec["-"] = 2
    prec["("] = 1
    opStack = Stack()
    postfixList = []
    tokenList = infixexpr.split()

    for token in tokenList:
        if token in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" or token in "0123456789":
            postfixList.append(token)
        elif token == '(':
            opStack.push(token)
        elif token == ')':
            topToken = opStack.pop()
            while topToken != '(':
                postfixList.append(topToken)
                topToken = opStack.pop()
        else:
            while (not opStack.isEmpty()) and \
               (prec[opStack.peek()] >= prec[token]):
                  postfixList.append(opStack.pop())
            opStack.push(token)

    while not opStack.isEmpty():
        postfixList.append(opStack.pop())
    return " ".join(postfixList)
```

Record the operator priority
记录操作符优先级

Parses the expressions to the word list
解析表达式到单词列表

操作数

(

)

操作符

合成后缀表达式字符串

Postfix expression evaluation
后缀表达式求值

Postfix expression evaluation
后缀表达式求值

As the end of the stack structure, let's discuss the "postfix expression evaluation" problem, which is different from the infix-to-suffix conversion problem. In the process of scanning the suffix expression from left to right, since the operator is behind the operand, you need to temporarily store the operands, and when you encounter an operator, perform the actual calculation on the two temporarily stored operands.

作为栈结构的结束，我们来讨论“**后缀表达式求值**”问题，跟中缀转换为后缀问题不同，在对后缀表达式从左到右扫描的过程中，由于操作符在操作数的**后面**，所以要**暂存操作数**，在碰到操作符的时候，再将暂存的两个操作数进行实际的计算

Still a feature of the stack: the operator only acts on the two closest operands

仍然是栈的特性：操作符只作用于离它**最近**的两个操作数

Postfix expression evaluation
后缀表达式求值

Such as "456 * +": We first scan two operands, 4 and 5, but we still don't know what to do with these two operands. We need to continue scanning the following symbols to know to continue the scanning, and when we encounter operand 6, we still can't know how to calculate. Under this condition, continue to temporarily store on the stack, until "*", now we know that the two operands 5 and 6 on the top of the stack are multiplied

如“456*+”，我们先扫描到4、5两个操作数，但还不知道对这两个操作数能做什么计算，需要继续扫描后面的符号才能知道继续扫描，又碰到操作数6，还是不能知道如何计算，继续暂存入栈，直到“*”，现在知道是栈顶两个操作数5、6做乘法

Postfix expression evaluation
后缀表达式求值

We pop up the two operands and compute the result 30
我们弹出两个操作数，计算得到结果30

need to pay attention to:
需要注意：

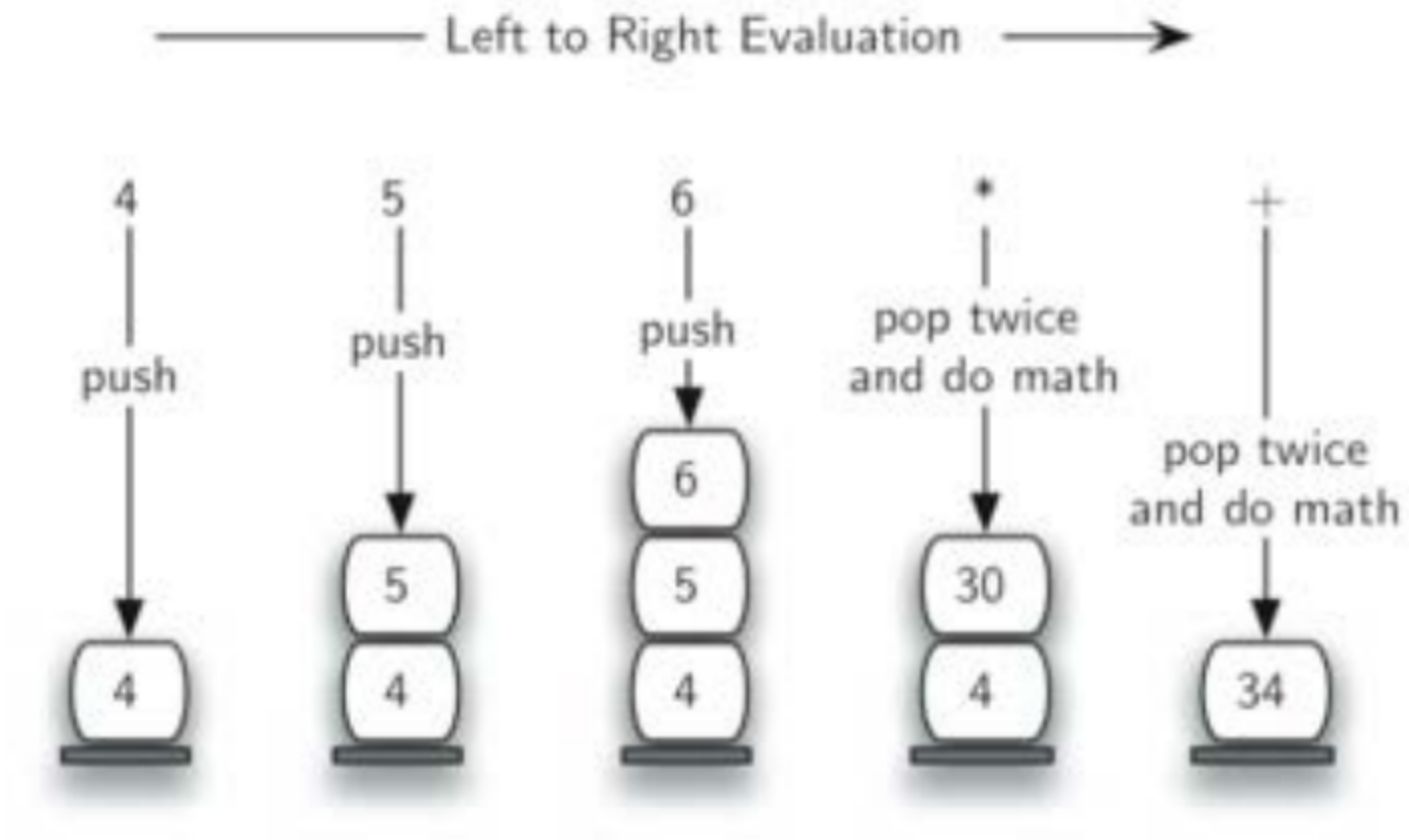
First pop-up is the right operand
先弹出的是右操作数

Later pop-up is the left operand, this is very important for -/!
后弹出的是左操作数，这个对于-/很重要！

In order to continue the subsequent calculation, we need to press this intermediate result 30 into the top of the stack and continue scanning the symbols which are behind it. When all the operators are processed, only one operand is left in the stack, which is the value of the expression
为了继续后续的计算，需要把这个中间结果30压入栈顶，继续扫描后面的符号，当所有操作符都处理完毕，栈中只留下1个操作数，就是表达式的值

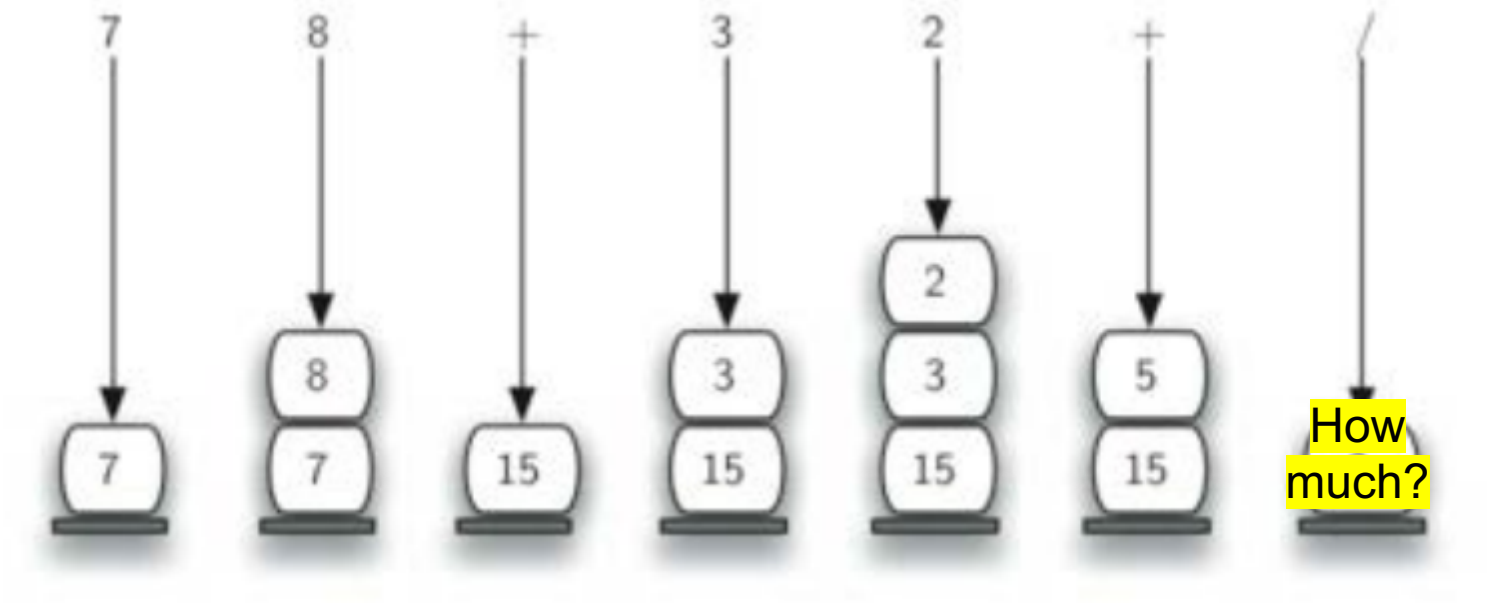
Postfix expression evaluation: example

后缀表达式求值：实例



Postfix expression evaluation: example

后缀表达式求值：实例



Postfix expression evaluation: Process
后缀表达式求值：流程

Create an empty stack operandStack to store the operand temporarily
创建空栈operandStack用于暂存操作数

Parse the suffix expression to a list of words (token), then scan the word list from left to right
将后缀表达式用split方法解析为单词(token)的列表，从左到右扫描单词列表

- If the word is an operand, convert the word to an integer(int), pressing into the top of the operandStack stack
  如果单词是一个操作数，将单词转换为整数int，压入operandStack栈顶
- If the word is an operator (* / + -),then start evaluating, pop two operands from the top of the stack.Pop the right operand firstly, then the left operand, and push the value back to the top of the stack
  如果单词是一个操作符(*/+-)，就开始求值，从栈顶弹出2个操作数，先弹出的是右操作数，后弹出的是左操作数，计算后将值重新压入栈顶

After ending the word list scanning, the value of the expression pops up the value at the top of the stack, returns.
单词列表扫描结束后，表达式的值就在栈顶弹出栈顶的值，返回。

code
代码

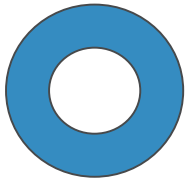
operands
操作数

operator
操作符

```
def postfixEval(postfixExpr):
    operandStack = Stack()
    tokenList = postfixExpr.split()

    for token in tokenList:
        if token in "0123456789":
            operandStack.push(int(token))
        else:
            operand2 = operandStack.pop()
            operand1 = operandStack.pop()
            result = doMath(token,operand1,operand2)
            operandStack.push(result)
    return operandStack.pop()

def doMath(op, op1, op2):
    if op == "*":
        return op1 * op2
    elif op == "/":
        return op1 / op2
    elif op == "+":
        return op1 + op2
    else:
        return op1 - op2
```



Discuss

1）In the array representation of a linear table of **n** elements, the operation with time complexity O(1) is (　).

在n个元素的线性表的数组表示中，时间复杂度为O(1)的操作是（　）。

Ⅰ.Accessing the i(1≤i≤n) th node and finding the predecessor of the i(2≤i≤n) th node

访问第i（1≤i≤n）个结点和求第i（2≤i≤n）个结点的直接前驱

Ⅱ.Insert a new node after the last node

在最后一个结点后插入一个新的结点

Ⅲ.Delete the first node

删除第1个结点

Ⅳ.Insert a node after the i(1≤i≤n) th node

在第i（1≤i≤n）个结点后插入一个结点

A. Ⅰ　　B.Ⅱ,Ⅲ　　C. Ⅰ,Ⅱ　　D. Ⅰ,Ⅱ,Ⅲ

Ⅰ正确；Ⅱ中，在最后位置插入新结点不需要移动元素，时间复杂度为O(1)；Ⅲ中，被删结点后的结点需依次前移，时间复杂度为O(n)；Ⅳ中，需要后移n-i个结点，时间复杂度为O(n)。

2）It is assumed that an array a[n] is used to store a stack in order, top is used to represent the top pointer of the stack, top==-1 is used to represent the stack is em to be not full. When the element x is put on the stack, the operation to be performed is(

假定利用数组a[n]顺序存储一个栈，用top表示栈顶 ,并已知栈未满，当元素x进栈时所执行的操作为（　）。

A.a[--top]=x　　B.a[top--]=x　　C.a[++top]=x　　D.a[top++]=x

初始时top为-1，则第一个元素入栈后，top为0，即指向栈顶元素，故入栈时应先将指针top加1，再将元素入栈，只有选项C符合题意。

Discuss

3）Use *S* to represent the stacking operation and *X* to represent the stacking operation. If the stacking order of elements is 1234, in order to obtain 1342 stacking order, the corresponding operation sequence of *S* and *X* is (　).

用S表示进栈操作，用X表示出栈操作，若元素的进栈顺序是1234，为了得到1342出栈顺序，相应的S和X的操作序列为（　）。

A. SXSXSSXX　　B. SSSXXSXX　　C. SXSSXXSX　　D. SXSSXSXX

采用排除法，选项A、B、C得到的出栈序列分别为1243、3241、1324。由1234得到1342的进出栈序列为：1进、1出、2进、3进、3出、4进、4出、2出，故选D。

4）The suffix expression for `a*(b+c)-d` is (　).

表达式a*(b+c)-d的后缀表达式是（　）。

A. `abcd*+-`　　B. `abc+*d-`　　C. `abc*+d-`　　D. `-+*abcd`

后缀表达式中，每个计算符号均直接位于其两个操作数的后面，按照这样的方式逐步根据计算的优先级将每个计算式进行变换，即可得到后缀表达式。

另解：将两个直接操作数用括号括起来，再将操作符提到括号后，最后去掉括号。如下：((a*(b+c))-d)，提出操作符并去掉括号后，可得后缀表达式为abc+*d-。

Discuss

5）If an integer is stored in stack S1 and an operator is stored in stack S2, function f( ) performs the following operations in sequence:

若栈s1中保存整数，栈s2中保存运算符，函数F( )依次执行下述各步操作:

1）从S1中依次弹出两个操作数a和b。 Two operands A and B are popped out from S_1.

2）从S2中弹出一个运算符op。 Pop up an operator OP from S_2.

3）执行相应的运算b op a。 The corresponding operation B OP A is performed.

4）将运算结果压入S1中。 Press the result of the operation into S_1.

假定S1中的操作数依次是5，8，3，2（2在栈顶），S2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（　）。 Assume that the operands in S_1 are 5, 8, 3, 2 (2 at the top of the stack), and the operators in S_2 are *, -, +(+at the top of the stack). After three F calls, the value saved at the top of the S_1 stack is ( ).

A.-15　　B.15　　C.-20　　D.20

S1第一次弹出a，b即2，3；S2弹出“+”；操作为b+a，即3+2，压入S1。现在S1为5，8，5。
S1第二次弹出a，b即5，8；S2弹出“-”；操作为b-a，即8-5，压入S1。现在S1为5，3。
S1第三次弹出a，b即3，5；S2弹出“*”；操作为b*a，即5*3，压入S1。现在S1为15。所以三次后S1栈顶为15

Discuss

1）The sequential storage of linear tables is also known as sequential tables. It stores data elements in a linear table sequentially using a set of memory cells with consecutive addresses, so that logically adjacent elements are physically adjacent as well. The first element is stored at the beginning of the linear table, followed by the i+1 element.

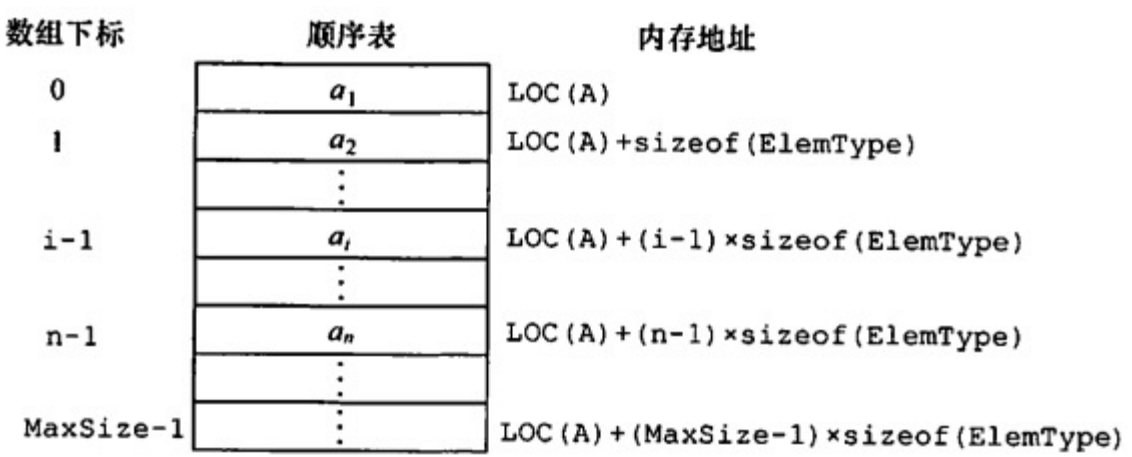
线性表的顺序存储又称顺序表。它是用一组地址连续的存储单元依次存储线性表中的数据元素，从而使得逻辑上相邻的两个元素在物理位置上也相邻。第1个元素存储在线性表的起始位置,第i个元素的存储位置后面紧接着存储的是第i+1个元素。

Analyze the time complexity of inserting, deleting, and finding nodes in the order table based on the definition of the order table.

请根据顺序表的定义分析顺序表插入、删除和按值查找节点的时间复杂度。

图：线性表的顺序存储

Sequential storage of linear tables



Discuss

1）插入操作 *insert operation*

最好情况：在表尾插入（即i=n+1），元素后移语句将不执行，时间复杂度为O(1)。

最坏情况：在表头插入（即i=1），元素后移语句将执行n次，时间复杂度为O(n)。

平均情况：假设p_i（p_i=1/(n+1)）是在第i个位置上插入一个结点的概率，则在长度为n的线性表中插入一个结点时，所需移动结点的平均次数为

$$\sum_{i=1}^{n+1} p_i\,(n-i+1) = \sum_{i=1}^{n+1} \frac{1}{n+1}(n-i+1) = \frac{1}{n+1}\sum_{i=1}^{n+1}(n-i+1) = \frac{1}{n+1}\frac{n(n+1)}{2} = \frac{n}{2}$$

因此，线性表插入算法的平均时间复杂度为O(n)。

Best case: *insert at the end of the table (i=n+1), the element backward statement will not execute, and the time complexity will be O(1).*

Worst case: *Inserting i n the header (i=1), the element backward statement executes n times with O(n) time complexity.*

Average case: *Assuming that PI (pi=1/(n+1)) is the probability of inserting a node at the first position, the average number of times a node needs to be moved when inserting a node i n a linear table with length n is as follows.*

Therefore, the average time complexity of the linear table insertion algorithm is O(n).

Discuss

2）删除操作 *delete operation*

最好情况：删除表尾元素（即i=n），无须移动元素，时间复杂度为O(1)。

最坏情况：删除表头元素（即i=1），需移动除表头元素外的所有元素，时间复杂度为O(n)。

平均情况：假设p_i（p_i=1/n）是删除第i个位置上结点的概率，则在长度为n的线性表中删除一个结点时，所需移动结点的平均次数为

$$\sum_{i=1}^{n} p_i\,(n-i) = \sum_{i=1}^{n} \frac{1}{n}(n-i) = \frac{1}{n}\sum_{i=1}^{n}(n-i) = \frac{1}{n}\frac{n(n-1)}{2} = \frac{n-1}{2}$$

因此，线性表删除算法的平均时间复杂度为O(n)。

Best case: *Delete the tail element (i=n), without moving the element, with a time complexity of O (1).*

Worst case: *Deleting a header element (i=1) requires moving all but the header element with an O(n) time complexity.*

Average case: *Assuming that PI (pi=1/n) is the probability of deleting a node at the ith position, the average number of times a node needs to be moved when deleting a node i n a linear table with length n is as follows.*

Therefore, the average time complexity of the linear table deletion algorithm is O(n).

Discuss

3）按值查找操作 *find by value operation*

最好情况：查找的元素就在表头，仅需比较一次，时间复杂度为O(1)。

最坏情况：查找的元素在表尾（或不存在）时，需要比较n次，时间复杂度为O(n)。

平均情况：假设p_i（p_i=1/n）是查找的元素在第i（`1<=i<=L.length`）个位置上的概率，则在长度为n的线性表中查找值为e的元素所需比较的平均次数为

$$\sum_{i=1}^{n} p_i \times i = \sum_{i=1}^{n} \frac{1}{n} \times i = \frac{1}{n} \frac{n(n+1)}{2} = \frac{n+1}{2}$$

因此，线性表按值查找算法的平均时间复杂度为O(n)。

Best case: *The element you are looking for is in the header and only needs to be compared once, with an O (1) time complexity.*

Worst case: *When looking for an element at the end of a table (or nonexistent), it needs to be compared n times with an O (n) time complexity.*

Average case: *Assuming that PI (pi=1/n) is the probability of finding the element at position I (1<=i<=L.length), the average number of comparisons needed to find an element with value e i n a linear table of length n is as follows*

Therefore, the average time complexity of the value-by-value lookup algorithm for linear tables is O(n).

Discuss

2) Known operators include +, -, *, /, left and right parentheses. When the infix expression `a+b-a*((c+d)/e-f)+g` is converted into the equivalent suffix expression `ab+acd+e/f-*-g+`, the stack is used to store the operators whose operation order cannot be determined temporarily. Try to analyze the changes of the operators stored in the stack during the conversion process when the stack is empty at the beginning.

已知操作符包括+、-、*、/、左括号和右括号。将中缀表达式a+b-a*((c+d)/e-f)+g转换为等价的后缀表达式ab+acd+e/f-*-g+时，用栈来存放暂时还不能确定运算次序的操作符。试分析初始栈为空时，转换过程中同时保存在栈中的操作符的变化。

将中缀表达式`a+b-a*((c+d)/e-f)+g`转换为相应的后缀表达式，需要根据操作符<op>的优先级来进行栈的变化，我们用icp来表示当前扫描到的运算符ch的优先级,该运算符进栈后的优先级为isp,则运算符的优先级如下表所示[isp是栈内优先(in stack priority)数，icp是栈外优先(in coming priority)数]：

操作符	#	(	*,/	+,-	)
isp	0	1	5	3	6
icp	0	6	4	2	1

To convert infix expression `a+b-a*((c+d)/e-f)+g` into a corresponding suffix expression, we need to change the stack according to the priority of the operator < op >. We use ***icp*** to represent the priority of the currently scanned operator ***ch***, and the priority of the operator after it is put on the stack is ***isp***. Then the priority of the operator is as shown in the following table [***isp*** is the number of in stack priority, and ***icp*** is the number of out of stack priority].

Discuss

make some noise

We add the symbol '#' after the expression to indicate the end of the expression. The specific conversion process is as follows:

我们在表达式后面加上符号‘#’，表示表达式结束。具体转换过程如下:

步骤	扫描项	项类型	动　　作	栈内内容	输出
0			‘#’进栈，读下一符号	#	
1	a	操作数	直接输出	#	a
2	+	操作符	isp(‘#’)<icp(‘+’),进栈	#+	
3	b	操作数	直接输出	#+	b
4	-	操作符	isp(‘+’)<icp(‘-’),退栈并输出	#	+
5			isp(‘#’)<icp(‘-’),进栈	#-	
6	a	操作数	直接输出	#-	a
7	*	操作符	isp(‘-’)<icp(‘*’),进栈	#-*	
8	(	操作符	isp(‘*’)<icp(‘(’),进栈	#-*(	
9	(	操作符	isp(‘(’)<icp(‘(’),进栈	#-*((	
10	c	操作数	直接输出	#-*((	c

Discuss

步骤	扫描项	项类型	动　　作	栈内内容	输出
11	+	操作符	isp('(')<icp('+'),进栈	#-*((+	
12	d	操作数	直接输出	#-*((+	d
13	)	操作符	isp('+')>icp(')'),退栈并输出	#-*((	+
14			isp('(')==icp(')'),直接退栈	#-*(	
15	/	操作符	isp('(')<icp('/'),进栈	#-*(/	
16	e	操作数	直接输出	#-*(/	e
17	-	操作符	isp('/')>icp('-'),退栈并输出	#-*(	/
18			isp('(')<icp('-'),进栈	#-*(-	
19	f	操作数	直接输出	#-*(-	f
20	)	操作符	isp('-')>icp(')'),退栈并输出	#-*(	-
21			isp('(')==icp(')'),直接退栈	#-*	
22	+	操作符	isp('*')>icp('+'),退栈并输出	#-	*

Discuss

步骤	扫描项	项类型	动　　作	栈内内容	输出
23			isp(‘-’)>icp(‘+’),退栈并输出	#	-
24			isp(‘#’)<icp(‘+’),进栈	#+	
25	g	操作数	直接输出	#+	g
26	#	操作符	isp(‘+’)>icp(‘#’),退栈并输出	#	+
27			isp(‘#’)==icp(‘#’),退栈并输出		

即相应的后缀表达式为ab+acd+e/ f-*-g+。

Discuss

3）Use a stack to realize the non recursive calculation of the following recursive functions:

利用一个栈实现以下递归函数的非递归计算：

$$P_n(x)=\begin{cases}1, & n=0\\ 2x, & n=1\\ 2xP_{n-1}(x)-2(n-1)P_{n-2}(x), & n>1\end{cases}$$

算法思想：设置一个栈用于保存n和对应的$P_n(x)$值，栈中相邻元素的$P_n(x)$有题中关系。然后边出栈边计算$P_n(x)$，栈空后该值就计算出来了。算法的实现如下：

Algorithm idea: a stack is set to store n and the corresponding PN (x) value, and the PN (x) of adjacent elements in the stack has a relationship with the title. Then, PN (x) is calculated while the stack is out. After the stack is empty, the value is calculated. The algorithm is implemented as follows:

Discuss

```
double p(int n,double x){
    struct stack{
        int no;                 //保存n
        double val;             //保存Pn(x)值
    }st[MaxSize];
    int top=-1,i;               //top为栈st的下标值变量
    double fv1=1,fv2=2*x;       //n=0，n=1时的初值
    for(i=n;i>=2;i--){
        top++;
        st[top].no=i;
    }                           //入栈
    while(top>=0){
        st[top].val=2*x*fv2-2*(st[top].no-1)*fv1;
        fv1=fv2;
        fv2=st[top].val;
        top--;                  //出栈
    }
    if(n==0){
        return fv1;
    }
    return fv2;
}
```