

**Stack (Push Down Stack)**

## Data Structures & Algorithms

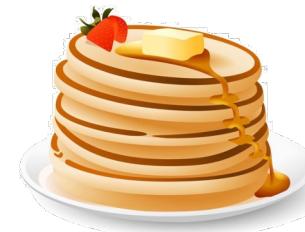
In this course, Data Structures & Algorithms :

1. **Data Structures** → abstract data types :

stack, queue, linked list, trees, heap, graph.

2. **Algorithms** : recursion, complexity (algorithm analysis) , hashing, searching, sorting.

## Stack



**Stacks** กองของ

## Stack

**Stack** : กองของ ของในกองมีลำดับ ordered collection of items  
มีปลายด้านบนเรียก **top** ของ **stack**  
เอาของเข้า (**push**) ออก (**pop**) ที่ **top** ของ **stack**

**LIFO**  
**Last in First out**

อันสุดท้าย ถูกเอาออกก่อน



## Queue

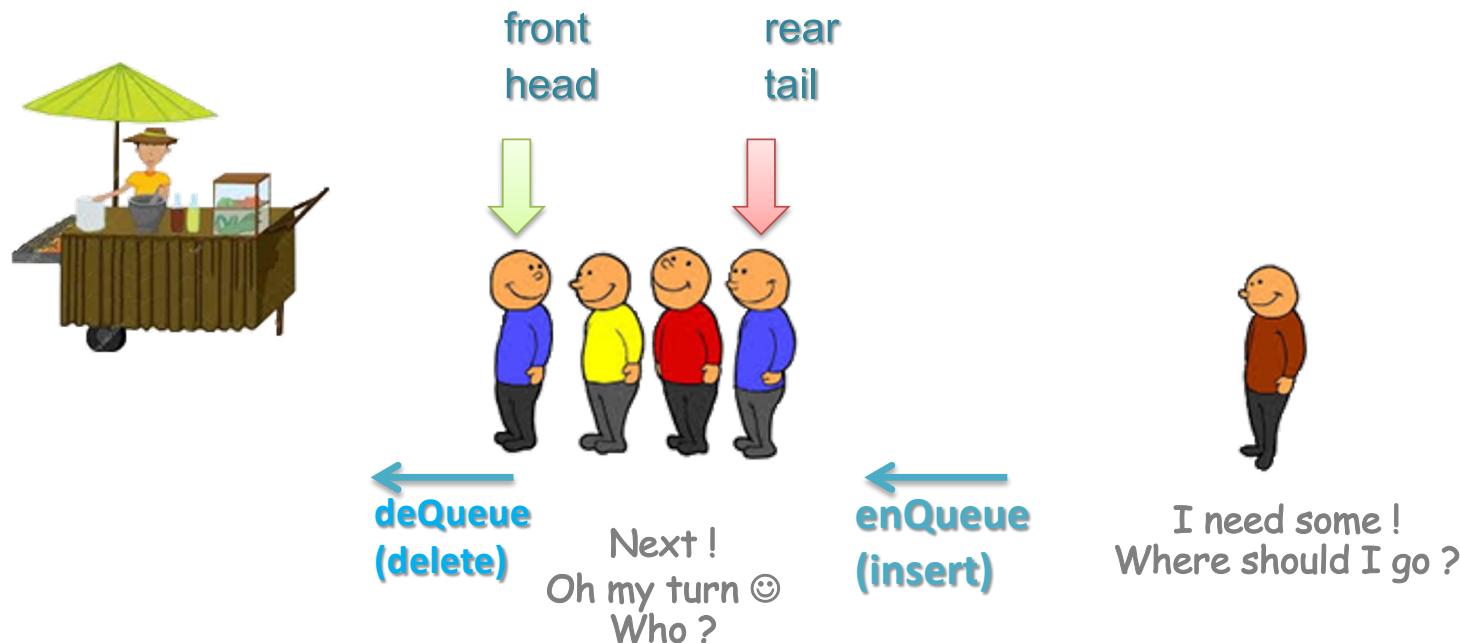


**Queue** ແກວຄອຍ  
?

## Queue แควค้อຍ

**Queue** : แควค้อຍ ของໃนແກມີລຳດັບ ordered collection of items  
ມີປາຍດ້ານທີ່ເຮັດວຽກ **rear/tail** ສໍາຮັບໄສຂອງເຂົ້າ (**enQueue**)  
ແລະອືກດ້ານທີ່ເຮັດວຽກ **front/head** ສໍາຮັບເອົາຂອງອອກ (**deQueue**)

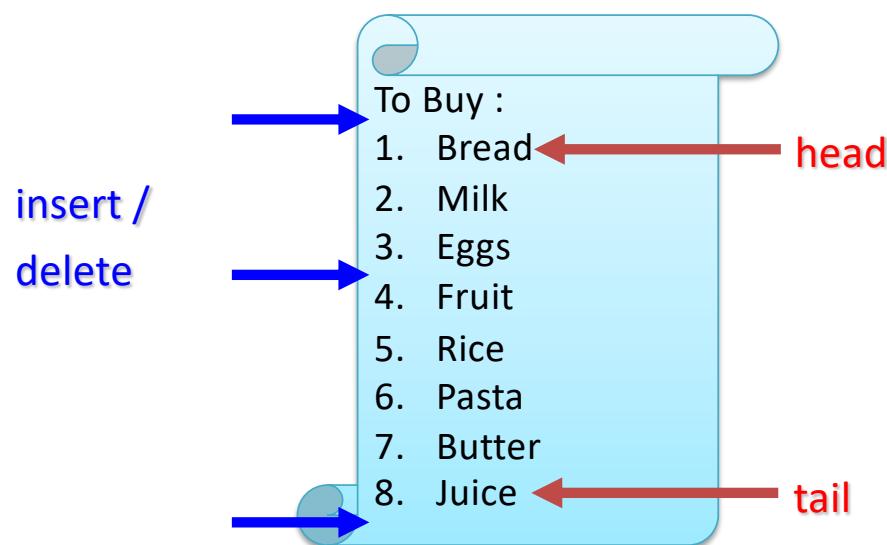
**FIFO List**  
**FirstInFirstOut**



## list

**List :** ordered collection of items ของใน list มีลำดับ  
ใส่ของเข้า (insert) เอาของออก (delete) ที่ได้ก็ได้

Superset of **Stack & Queue**



## Stack Applications

1. Parenthesis Matching
2. Evaluate Postfix Expression
3. Infix to Postfix Conversion (Reverse Polish Notation)
4. Function Call (clearly see in recursion)

## Parenthesis Matching

✗ ( a+b-c \* [d+e] / { f\* (g+h) } )

✗ ( a+b-c } \* [d+e] / { f\* (g+h) } )

✗ ( a+b-c ) \* [d+e] }

☺ ( 3 + 2 ) / { 4\*\*5 }

Match ?

Algorithm ?

## Parenthesis Matching

random  
↓  
$$[ (a+b)^* \{ (d+e)^{-3} \}]$$
  
↑                      ↑

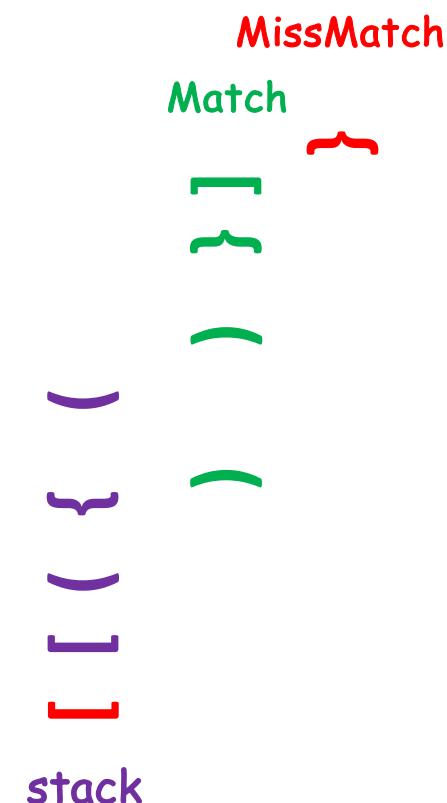
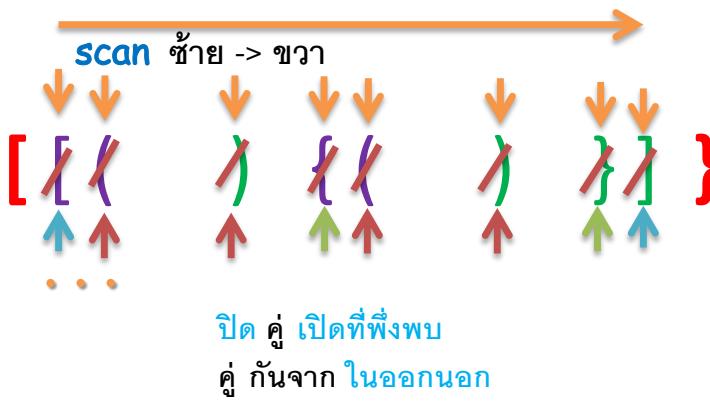
1. **Simulate** เลียนแบบ
  - Data Structure : Python List
  - Expensive
2. **Use Stack**

1. Repeat สุ่มคำแห่ง (until จะ || not match)
2. Scan ออกร่องข้าง
3.       if Left == open paren & Right == close paren
4.           if match :  
5.               Check out.  
6.               goto 2
7.       else จะ : Not match.
8.       else goto 1

## Parenthesis Matching

[ (a+b)\*{ (d+e)-3} ]

เพื่อให้เห็นชัด  
**Clear out**  
ส่วนที่ไม่เกี่ยวข้อง  
**scan**



## Stack Implementation

Logical ADT :

Implementation ?

1. Data : ของมีลำดับ มีปลายบน Python List

2. Methods :

1. init empty stack      init()       $S = []$

2. insert i ที่ top      push(i)       $S.append(i)$  ใส่ท้าย

3. เอาของที่ top ออก       $i = pop()$        $i = S.pop()$  จับท้าย

4. ดูของที่ top (ไม่เอาออก)       $i = peek()$

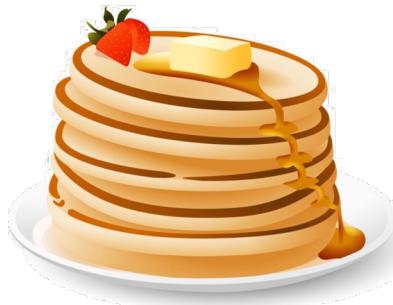
5. stack empty ?       $b = isEmpty()$

6. stack full ?       $b = isFull()$

7. หาจำนวนของใน stack       $i = size()$       Last in First out



## Stack Data Implementation



### 1. Data :

`__init__()` : constructor ให้ค่าตั้งต้น



2 underscores    2 underscores

## Data Implementation : \_\_init\_\_()

### 1. Data Implementation : Stack กองของซ้อนกัน ของมีลำดับ มีปลายด้านบน -> Python List

ทำใน constructor

`self` คือ object ที่เรียก method ในแต่ละครั้ง

เช่น `s = Stack()`

`self` หมายถึง `s` เมื่อเรียก `s = Stack(s)`

`self` จะถูก pass เป็น arg. ตัวแรก โดยอัตโนมัติ

docstring : ใน triple quote  
`print(Stack.__doc__)`  
→ docstring

constructor  
ถูกเรียกโดยอัตโนมัติเมื่อ  
instantiate instance ใหม่

items  
[ ]  
size  
0

```
class Stack:  
    """ class Stack  
        create empty stack  
    """  
    total = 0      # class data  
    def __init__(self):  
        self.items = []  
        self.size = 0  
        stack.total += 1
```

```
s = Stack()  
print(s.items)  []  
print(s.size)   0  
s2 = Stack()  
print(s.total) → 2
```

Class Data

สำหรับทุก stack

items , size:

Instance Attributes /data

สำหรับแต่ละ instance

เรียกชื่อ class :

สร้าง object ใหม่ (instantiate instance/obj)

ไปเรียก constructor พังก์ชัน \_\_init\_\_()

## Mutable Type Default Argument

Default argument :

ให้ = ค่านี้ เมื่อไม่มีการ pass ค่ามา

ค่า default จะถูกสร้างขึ้นครั้งเดียว

ณ function definition ใน scope ที่ define function

ต้องระวัง เมื่อเป็น mutable type

```
def f( L= [] ):
```

```
    print(L)
```

```
    L.append(1)
```

```
f()
```

```
f()
```

```
f([2])
```

```
f()
```

output

[ ]

[ 1 ]

[ 2 ]

[ 1 , 1 ]

default L → [ 1 , 1 , 1 ]

L → [ 2 , 1 ]

```
def f(L = None):
```

```
    if L is None:
```

```
        L = []
```

```
    else :
```

```
        pass
```

```
    L.append(1)
```

```
f()
```

```
f()
```

default L → None

[ 1 ]

## `__init__()` with Default Argument

```
class Stack:  
    """ class Stack  
        default : empty stack /  
        Stack([list])  
    """  
    def __init__(self, list = None):  
        if list == None:  
            self.items = []  
        else:  
            self.items = list
```

default argument  
ถ้าไม่มีการ pass arg. มา  
list = None  
ถ้า pass arg. มา  
list = ตัวที่ pass มา

```
s = Stack()  
s1 = Stack(['A', 'B', 'C'])
```

ไม่เหมือนกับ C++ & Java      ใน Python มี constructor ได้ตัวเดียว

## Stack Operation Implementation



1. Data :

`__init__()` : constructor ให้ค่าตั้งต้น

2. Methods (Operations) :

2. `push()` : ใส่ ด้านบน **top**

3. `pop()` : เอาออก ด้านบน **top**

4. `peek()` : ดู **top** ไม่เอาออก

5. `isEmpty()` : stack ว่าง ?

6. `size()` : มีของกี่อัน

## push()

```
class Stack:  
    def __init__(self, list = None):  
        if list == None:  
            self.items = []  
        else:  
            self.items = list  
        self.size = len(self.items)  
  
    def push(self, i):  
        self.items.append(i)  
        self.size += 1
```

Check Stack Overflow ?

-> No

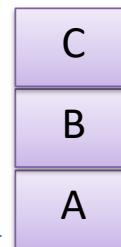
Python list automatically expanding size

list.append (i): insert i ที่ท้าย list

s.items

```
s = Stack()  
s.push('A')  
s.push('B')  
s.push('C')
```

[]  
['A']  
['A', 'B']  
['A', 'B', 'C'] →



## pop()

```
class Stack:  
    def __init__(self, list = None):  
        if list == None:  
            self.items = []  
        else:  
            self.items = list
```

```
def pop(self):    # remove & return อันบนสุด  
    return self.items.pop()
```

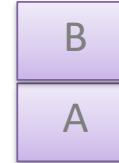
อย่าลืม return !!!

list.pop() : delete ตัวสุดท้ายของ list

list.pop(i) : delete ตัวที่ index i ของ list

```
print(s.items)  
print(s.pop())  
print(s.pop())  
s.pop()
```

[ 'A' , 'B' ]  
B  
A  
# error Stack Underflow



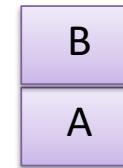
## peek()

```
class Stack:  
    def __init__(self, list = None):  
        if list == None:  
            self.items = []  
        else:  
            self.items = list  
  
    def peek(self):      # return อันบนสุด  
        return self.items[ -1 ]
```

-1 : last index

```
print(s.items)  
  
print(s.peek())  
  
print(s.items)
```

[ 'A' , 'B' ]  
  
B  
  
[ 'A' , 'B' ]

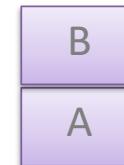


## isEmpty()

```
class Stack:  
    def __init__(self, list = None):  
        if list == None:  
            self.items = []  
        else:  
            self.items = list  
  
    def isEmpty(self):  
        return self.items == []    return len(self.items) == 0
```

```
print(s.items)  
print(s.isEmpty())
```

['A', 'B']  
false



## size()

```
class Stack:  
    def __init__(self, list = None):  
        if list == None:  
            self.items = []  
        else:  
            self.items = list
```

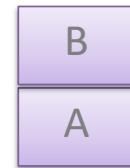
```
def size(self):  
    return len(self.items)
```

```
print(s.items)
```

[ 'A' , 'B' ]

```
print(s.size())
```

2



## Stack Implementation

```
class Stack:  
    """ class Stack  
        default : empty stack / Stack([...])  
    """  
  
    def __init__(self, list = None):  
        if list == None:  
            self.items = []  
        else:  
            self.items = list  
  
    def __str__(self):  
        s = 'stack of '+ str(self.size())+' items : '  
        for ele in self.items:  
            s += str(ele) + ' '  
        return s
```

```
def push(self, i):  
    self.items.append(i)  
  
def pop(self):  
    return self.items.pop()  
  
def peek(self):  
    return self.items[-1]  
  
def isEmpty(self):  
    return self.items == []  
  
def size(self):  
    return len(self.items)
```

s1 = Stack([1,2,3])

print(s1.items) [1, 2, 3]

print(s1) stack of 3 items : 1 2 3

\_\_str\_\_() ต้อง return string

## Writing Code

```
Def parenMatch(s) :  
    # code? គីទ?
```



## Design

**Code :** difficult

```
if openParen(c) :  
    s.push(c)  
elif closeParen(c) :  
    ch = s.pop()  
    match = isMatch(c, ch)
```



**Pseudocode :** easier

```
if (c is an open parenthesis)  
    push c to stack s  
else if (c is an close parenthesis)  
    pop ch from stack s  
    if (ch matches c)  
        match = true  
    else match = false
```



## Python : Parenthesis Matching

```
def parenMatching(str):
    s = Stack()
    i = 0                      # index : str[i]
    error = 0

    while i < len(str) and error == 0:
        c = str[i]
        if c in '{[(':
            s.push(c)
        else:
            if c in '}]':
                if s.size() > 0:
                    if not match(s.pop(),c):
                        error = 1      # open & close not match
                    else:           # empty stack
                        error = 2      # no open paren
        i += 1

        if s.size() > 0:          # stack not empty
            error = 3             # open paren(s) excesses
    return error,c,i,s
```

```
str = '[{a+b-c'
err,c,i,s = parenMatching(str)
if err == 1:
    print(str , 'unmatch open-close ')
elif err == 2:
    print(str , 'close paren excess')
elif err == 3:
    print(str , 'open paren(s) excess ', s.size(),'')
for ele in s.item:
    print(ele,sep=' ',end = "")
print()
else:
    print(str, 'MATCH')
```

```
def match(open, close):
    return (open == '(' and close == ')') or \
           (open == '{' and close == '}') or \
           (open == '[' and close == ']')
```

```
def match2(op,cl):
    opens = "({["
    closes = "})]"
    return opens.index(op) == closes.index(cl)
```

## Stack Application

1. **Parenthesis Matching**
2. **Evaluate Postfix Expression**
3. **Infix to Postfix Conversion (Reverse Polish Notation)**
4. **Function Call (clearly see in recursion)**

## Postfix Notation (Polish Notation)

Infix Form

a + b

Prefix Form

+ a b

Postfix Form

a b +

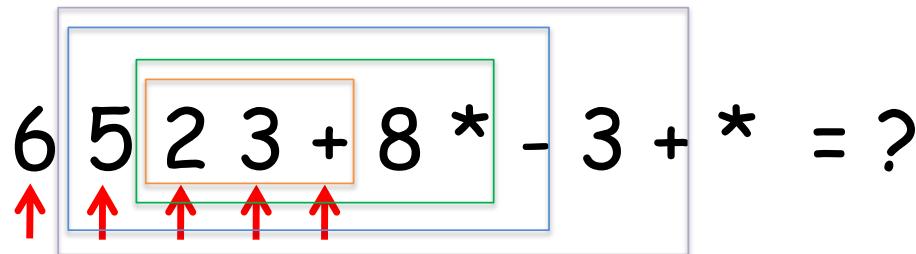
a + b \* c

+ a \* b c

a b c \* +

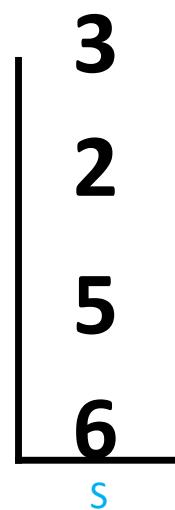
## Evaluate Postfix Notation

**Postfix Notation** มีชื่อรวมชาติเป็น **stack**: **operator** เป็นของ **operands 2** ตัวก่อนหน้ามัน



input: 6523

Push : 6, 5, 2, 3



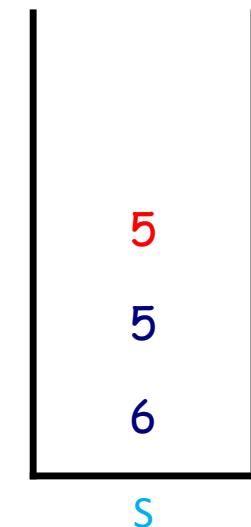
input: +

o<sub>2</sub> <- pop#1

o<sub>1</sub> <- pop#2

push

+  
5

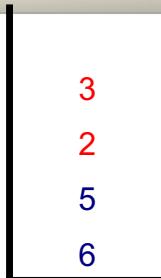


What 's next ?

6 5 2 3 + 8 \* - 3 + \* = ?

input: 6523

Push : 6 5 2 3

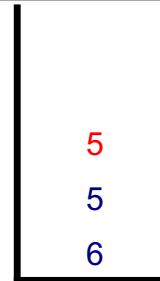


input: +

pop#1 → 3

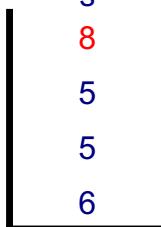
pop#2 → 2

push 2+3



input: 8

push 8

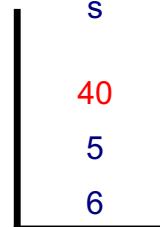


input: \*

pop#1 → 8

pop#2 → 5

push 5\*8

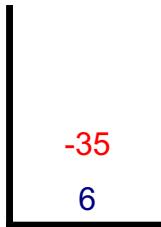


input: -

pop#1 → 40

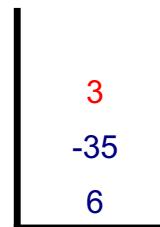
pop#2 → 5

push 5-40



input: 3

push 3

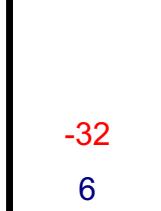


input: +

pop#1 → 3

pop#2 → -35

push -35+3

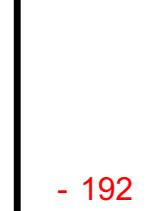


input: \*

pop#1 → -32

pop#2 → 6

push 6 \* -32



## Stack Application

1. **Parenthesis Matching**
2. **Evaluate Postfix Expression**
3. **Infix to Postfix Conversion (Reverse Polish Notation)**
4. **Function Call (clearly see in recursion)**

## Infix to Postfix Conversion

a\*b+c

==>

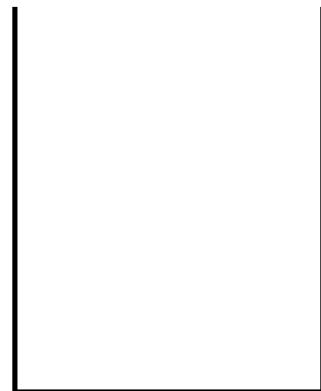
ab\*c+



Notice :

output: operands' order is the same as input's.

output



stack

Infix to Postfix Conversion  $a^*b+c$  ---- >

$ab^*c^+$

input:

$a^*b+c$

$a^*b+c$

$a^*b+c$

$a^*b+c$

$a^*b+c$

$a^*b+c$



stack:

\* 

\* 

+ 

+ 

output:

a

a

ab

ab\*

ab\*c

ab\*c+

Infix to Postfix Conversion  $a+b*c$  ---- >

$abc^*+$

input:	stack:	output:
$a+b*c$		$a$
$a+b*c$	+	$a$
$a+b*c$	+	$ab$
	*	
$a+b*c$	+	$ab$
	*	
$a+b*c$	+	$abc$
$a+b*c$		$abc^*+$

## Infix to Postfix Conversion

input:

$a+b*c-d$

$a+b*c-d$

$a+b*c-d$

$a+b*c-d$

$a+b*c-d$

$a+b*c-d$

$a+b*c-d$

$a+b*c-d$

stack:

$[ ]$

$[ + ]$

$[ + ]$

$[ * ]$

$[ + ]$

$[ * ]$

$[ + ]$

$[ - ]$

$[ - ]$

$[ ]$



$a+b*c-d \Rightarrow abc*+d-$

output:

$a$

$a$

$ab$

$ab$

$abc$

$abc*$

$+d$

$abc*+d$

$-$

## Infix to Postfix Conversion

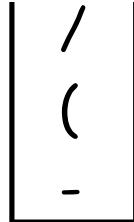
$$a+b*c-(d/e+f)*g \Rightarrow abc^*+de/f+g^*.-$$

$a+b*c-(d/e+f)*g$	$\boxed{+}$	$ab$
	$\boxed{*}$	
$a+b*c-(d/e+f)*g$	$\boxed{+}$	$abc$
$a+b*c-(d/e+f)*g$	$\boxed{-}$	$abc^*+$
	$\boxed{(}$	
$a+b*c-(d/e+f)*g$	$\boxed{-}$	$abc^*+d$
	$\boxed{/}$	
$a+b*c-(d/e+f)*g$	$\boxed{-}$	$abc^*+de$

## Infix to Postfix Conversion (cont.)

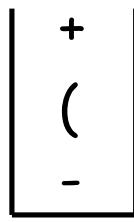
$a+b*c-(d/e+f)*g \Rightarrow$

$abc^*+de/f+g^*-$



$a+b*c-(d/e+f)*g$

$abc^*+de$



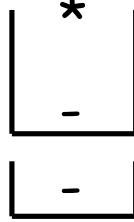
$a+b*c-(d/e+f)*g$

$abc^*+de/f$



$a+b*c-(d/e+f)*g$

$abc^*+de/f+$



$a+b*c-(d/e+f)^*g$

$abc^*+de/f+g$



$a+b*c-(d/e+f)^*g$

$abc^*+de/f+g^*-$



## Stack Application

1. Parenthesis Matching
2. Evaluate Postfix Expression
3. Infix to Postfix Conversion (Reverse Polish Notation)
4. Function Call (clearly see in recursion)