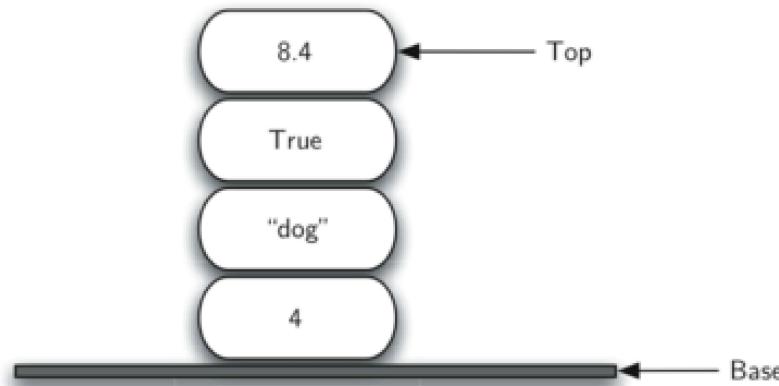


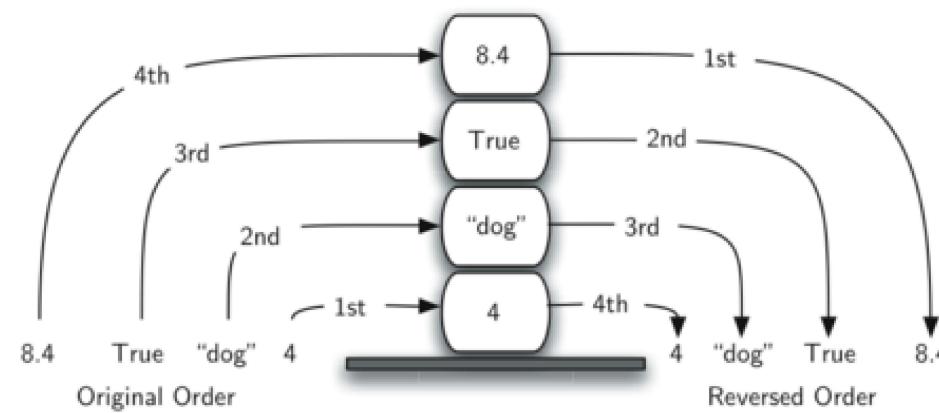
# 스택(Stack)

# 스택의 정의

- 항목의 추가 및 삭제: **탑**<sub>top</sub>이라 불리는 한 쪽 끝에서만 허용
- 탑: 가장 나중에 추가된 항목의 위치
- 베이스: 남아 있는 항목 중에서 가장 먼저 추가된 항목의 위치



## 후입선출(LIFO)



# Stack 추상 자료형

- `Stack(maxsize=0)`: 비어 있는 스택 생성. `maxsize`를 양의 정수로 지정하면 항목을 최대 그만큼 담을 수 있음. 아니면 제한 없음.
- `put(item)`: `maxsize`가 양의 정수이면서 항목을 추가하면 허용되는 항목의 수를 초과하는 경우 먼저 탑(top) 항목을 제거한 다음 새로운 항목을 탑에 추가. 반환값 없음.
- `get()`: 탑 항목 삭제. 삭제된 항목 반환.
- `peek()`: 탑 항목 반환. 하지만 삭제하진 않음.
- `empty()`: 스택이 비었는지 여부 판단. 부울값 반환.
- `full()`: 스택이 꽉 차 있는지 여부 판단. 부울값 반환.
- `qsize()`: 항목 개수 반환.

스택 연산	스택 항목	반환값
s = Stack(maxsize=4)	[]	
s.empty()	[]	True
s.put(4)	[4]	
s.put('dog')	[4, 'dog']	
s.peek()	[4, 'dog']	'dog'
s.put(True)	[4, 'dog', True]	
s.qsize()	[4, 'dog', True]	3
s.empty()	[4, 'dog', True]	False
s.put(8.4)	[4, 'dog', True, 8.4]	
s.full()	[4, 'dog', True, 8.4]	True
s.get()	[4, 'dog', True]	8.4
s.get()	[4, 'dog']	True
s.qsize()	[4, 'dog']	2

# 스택 자료구조 구현

```
class Stack:  
    def __init__(self, maxsize=0):  
        self.maxsize = maxsize  
        self._items = []  
  
    def __repr__(self):  
        return f"<{self._items}>"  
  
    def empty(self):  
        return not bool(self._items)  
  
    def full(self):  
        if self.maxsize <= 0:  
            return False  
        elif self.qsize() == self.maxsize:  
            return True  
        else:  
            return False  
  
    ... (이어짐)
```

```
class Stack:

    ... (중략)

    def put(self, item):
        if self.full() == True:
            self.get()

        self._items.append(item)

    def get(self):
        return self._items.pop()

    def peek(self):
        if not self.empty():
            return self._items[-1]
        else:
            print("스택이 비어 있어요!")

    def qsize(self):
        return len(self._items)
```

## 예제

```
>>> m = Stack()
>>> m.put("x")
>>> m.put("y")
>>> m.get()
>>> m.put("z")
>>> m.peek()
'z'
```

## 예제

```
>>> m = Stack()
>>> m.put("x")
>>> m.put("y")
>>> m.put("z")

>>> while not m.empty():
...     m.get()

>>> m.peek()
스택이 비어 있어요!
```

## 구현 방식에 따른 속도 차이

```
class Stack_left:  
    ... (중략)  
  
    def put(self, item):  
        if self.full() == True:  
            self.get()  
  
        self._items.insert(0, item)  
  
    def get(self):  
        return self._items.pop(0)
```

## 성능 비교

- 항목 추가에 걸리는 시간: Stack 클래스가 100배 정도 빠름
- 항목 삭제에 걸리는 시간: Stack 클래스가 1,000배 이상 빠름
- 리스트의 0번 인덱스를 탑으로 사용하면 항목을 추가하거나 삭제할 때마다 나머지 항목들의 위치를 이동시키기 위해 보다 많은 시간이 요구되기 때문

## queue 모듈의 LifoQueue 클래스

- queue 모듈의 LifoQueue 클래스가 스택 자료구조를 가리킴
- 앞서 정의한 Stack 클래스보다 10배 정도 느림

스택 실전 활용

## 괄호 짹맞추기 문제

- 짹이 맞지 않으면 오류 발생

```
>>> (5 + 6) * (7 + 8) / (4 + 3
      Input In [7]
      (5 + 6) * (7 + 8) / (4 + 3
                                         ^
SyntaxError: unexpected EOF while parsing
```

- 표현식을 작성할 때 여는 괄호 하나와 닫는 괄호 하나가 짹이 맞아야 함

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

- 괄호만 고려할 경우 다음 모양이 됨

```
( )()()
```

- 괄호 짹이 맞는 경우

((()())())

((((()))))

((()(((())())

- 괄호 짹이 맞지 않는 경우

((((((((

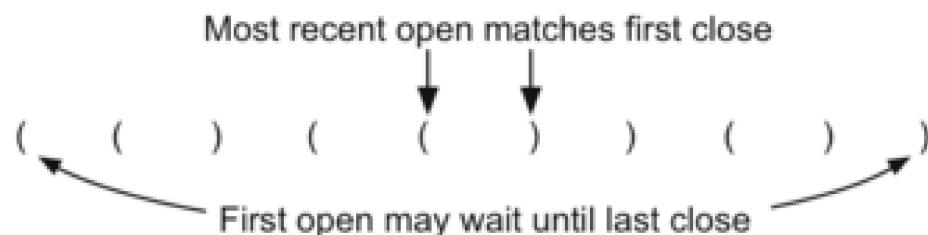
()))

((())())

- 괄호로 이루어진 문자열이 주어졌을 때 왼편부터 시작해서 여는 괄호와 닫는 괄호를 만날 때마다 아래 작업을 반복
  - 여는 괄호: 스택에 추가
  - 닫는 괄호: 스택의 탑 항목 삭제
- 위 작업을 반복하다 보면 아래 세 가지 경우가 발생
  - 문자열을 다 확인하기 전에 스택이 비워지는 경우: 닫는 괄호가 너무 많음
  - 끝까지 확인했을 때 스택이 비워지지 않은 경우: 여는 괄호가 너무 많음
  - 그렇지 않으면 모든 괄호의 짹이 맞음.

## 괄호 짹맞추기 알고리즘

```
def par_checker(symbol_string):
    s = Stack()
    for symbol in symbol_string:
        if symbol == "(":
            s.put(symbol)
        elif s.empty():
            return False
        else:
            s.get()
    return s.empty()
```



## 괄호 짹맞추기 문제(일반화)

- 소, 중, 대 세 종류의 괄호를 대상으로 짹맞추기 문제
  - ( , ): 튜플, 표현식 등
  - { , }: 사전, 집합 등
  - [ , ]: 리스트 등

- 괄호 짹이 맞는 경우

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

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

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

- 괄호 짹이 맞지 않는 경우

```
( [ ) )
```

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

```
[ { ( ) ]
```

```
def balance_checker(symbol_string):
    s = Stack()
    for symbol in symbol_string:
        if symbol in "([{":
            s.put(symbol)
        elif s.empty():
            return False
        elif not matches(s.get(), symbol):
            return False

    return s.empty()

def matches(sym_left, sym_right):
    all_lefts = "([{"
    all_rights = ")]}"
    return all_lefts.index(sym_left) == all_rights.index(sym_right)
```

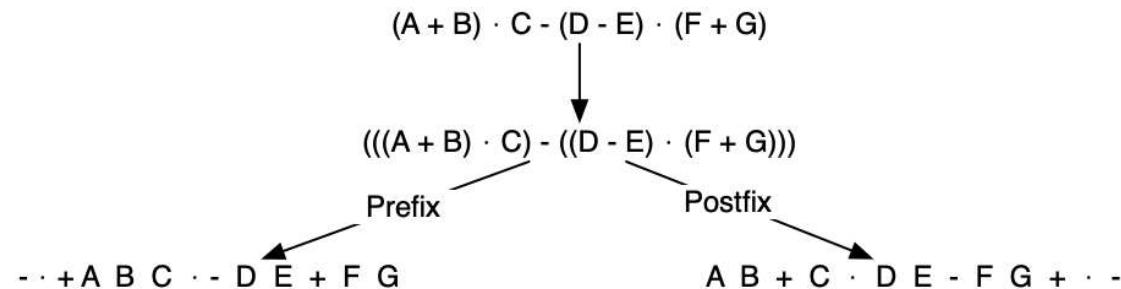
# 중위/전위/후위 표기법

중위 표기법	전위 표기법	후위 표기법
$x + y$	$+ x y$	$x y +$
$x + y * z$	$+ x * y z$	$x y z * +$
$(x + y) * z$	$* + x y z$	$x y + z *$

중위 표기법	전위 표기법	후위 표기법
$x + y * z + v$	$+ + x * y z v$	$x y z * + v +$
$(x + y) * (z + v)$	$* + x y + z v$	$x y + z v + *$
$x * y + z * v$	$+ * x y * z v$	$x y * z v * +$
$x + y + z + v$	$+ + + x y z v$	$x y + z + v +$

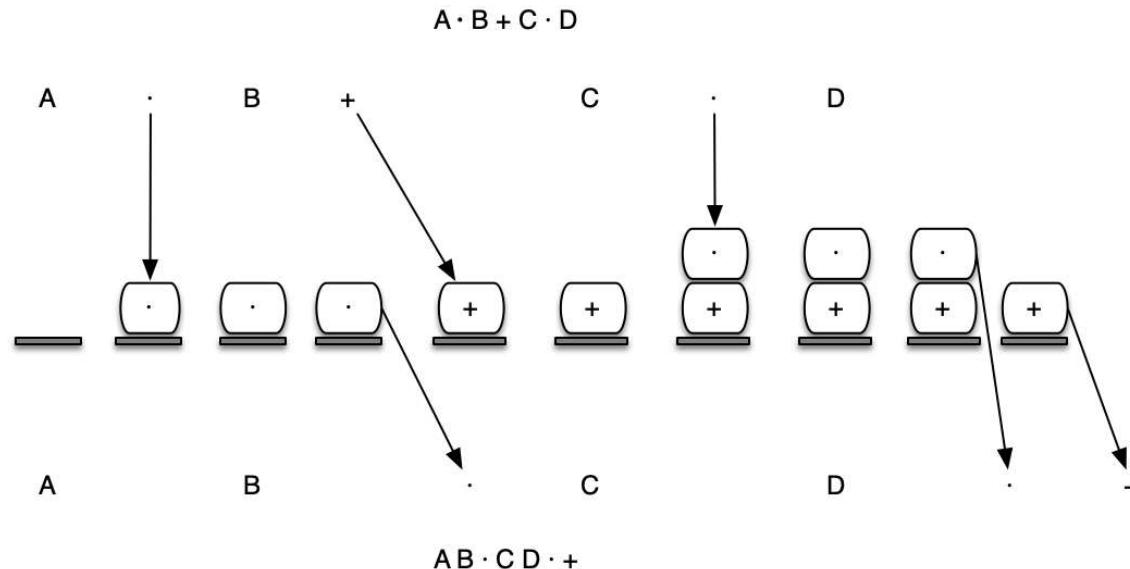
# 표기법 변환

- ( $A + B$ ) \*  $C - (D - E) * (F + G)$  를 전위/후위 표기법으로 표현하는 과정



## 표기법 변환 알고리즘

- $A * B + C * D$  를 후위 표기법으로 변환하는 과정



- 연산자와 여는 괄호를 쌓아 둘 스택 `op_stack`을 준비해 놓는다. 또한 후위 표현식에 사용될 기호를 차례대로 저장할 빈 리스트 `postfix_list`도 준비한다.

```
op_stack = Stack()  
postfix_list = []
```

- 문자열로 입력된 중위 표현식을 `split()` 메서드를 이용하여 리스트로 변환한다.

- 리스트의 항목(토큰, token)의 종류에 따라 아래 과정을 처리한다.

- 피연산자인 경우: `postfix_list`에 추가한다.
- 여는 괄호인 경우: `op_stack`에 추가한다.
- 닫는 괄호인 경우: `op_stack`에서 여는 괄호를 만날 때까지 탑을 빼서 `postfix_list`에 추가한다.
- 연산자(\*, /, +, -)인 경우: `op_stack`에 추가한다. 단, 먼저 `op_stack`으로부터 우선순위가 높거나 같은 연산자를 모두 탑에서 빼서 `postfix_list`에 추가해야 한다.

- 입력된 중위 표현식에 사용된 모든 기호를 처리했다면 `op_stack`에 남아있는 모든 연산자를 빼서 `postfix_list`에 추가한다.

## 연산자 우선순위

```
precedence = { '*' : 3, '/' : 3, '+' : 2, '-' : 2, '(' : 1}
```

```
def infix_to_postfix(infix_expr):
    op_stack = Stack()
    postfix_list = []
    token_list = infix_expr.split()

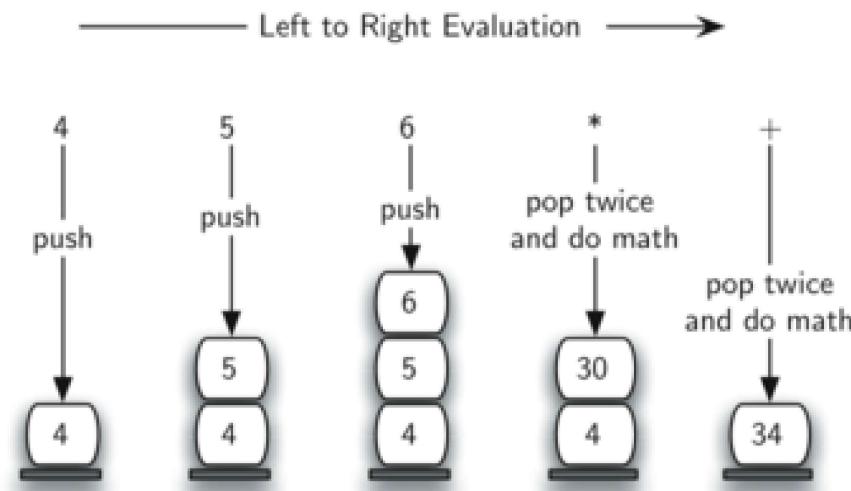
    for token in token_list:
        if token not in '()*+-':
            postfix_list.append(token)
        elif token == "(":
            op_stack.put(token)
        elif token == ")":
            top_token = op_stack.get()
            while top_token != "(":
                postfix_list.append(top_token)
                top_token = op_stack.get()
        else:
            while (not op_stack.empty())
                and (precedence[op_stack.peek()] >= precedence[token]):
                postfix_list.append(op_stack.get())
            op_stack.put(token)

    while not op_stack.empty():
        postfix_list.append(op_stack.get())

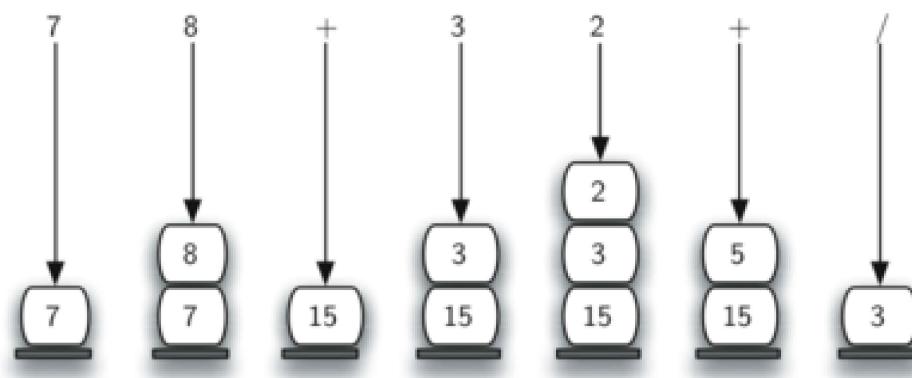
    return " ".join(postfix_list)
```

# 후위 표기법 표현식 계산

- 후위 표현식 4 5 6 \* + 을 계산하는 과정



- 후위 표현식 7 8 + 3 2 + / 을 계산하는 과정



1. `operand_stack` 라는 비어있는 스택을 준비한다.
2. 후위 표현식 문자열을 `split()` 메서드를 이용하여 리스트로 변환한다.
3. 리스트를 왼쪽에서부터 하나씩 차례대로 확인하면서 확인된 기호의 종류에 따라 아래 과정을 반복한다.
  - 피연산자인 경우: 수(number)로 변환한 다음 `operand_stack`에 추가한다.
  - `*`, `/`, `+`, `-` 중 하나인 경우: `operand_stack`을 두 번 `get()` 한다. 첫번째로 얻어진 값은 둘째 인자로, 두번째로 얻어진 값은 첫째 인자로 지정한다. 두 인자를 이용하여 연산을 실행한 다음 계산된 값을 다시 `operand_stack`에 추가한다.
4. 리스트의 모든 항목을 확인한 다음에 `operand_stack`에 납아있는 유일한 값을 주어진 표현식을 계산한 값으로 사용한다.

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

```
def postfix_eval(postfix_expr):
    operand_stack = Stack()
    token_list = postfix_expr.split()
    print(operand_stack)

    for token in token_list:
        if token not in '*/+-':
            operand_stack.put(int(token))
        else:
            operand2 = operand_stack.get()
            operand1 = operand_stack.get()
            result = do_math(token, operand1, operand2)
            operand_stack.put(result)
            print(operand_stack)

    return operand_stack.get()
```