# 物件導向程式設計(Object Oriented Programming)：實作堆疊模組與前置式運算

Last In First Out

空串列

附加字串

附加字串

```
s = []
item = "*"
s.append(item)
item = "123"
s.append(item)
item = "2"
s.append(item)
print(s.count==0)
print(s)
item = s.pop()
print(item)
print(s)
```

彈出字串

```
False
['*', '123', '2']
2
['*', '123']
```

最後附加的字串，先彈出字串

# 以串列實作
# 資料結構-堆疊

item = "*"

s.push(item)

將item推入堆疊

S

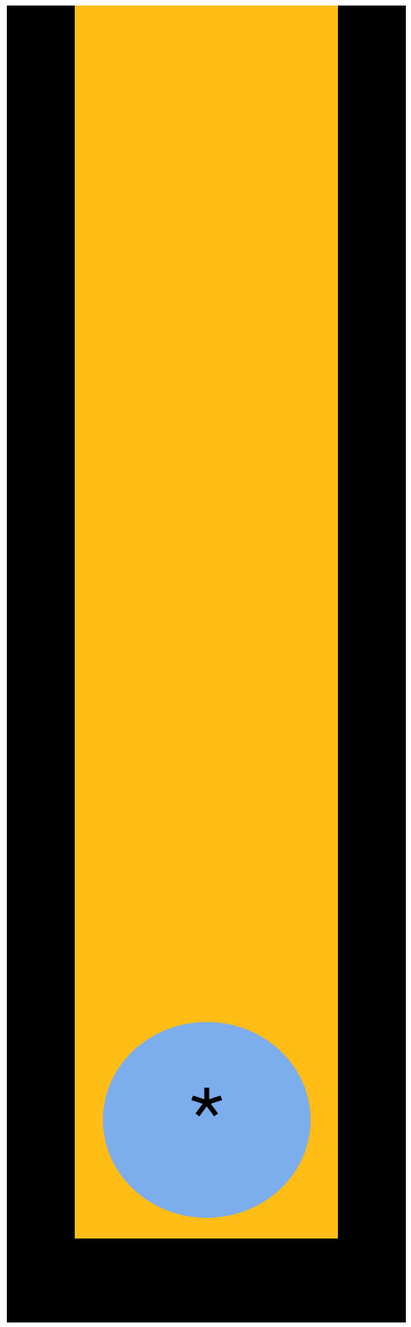
堆疊s，代表物件．
s.push()代表方法

*

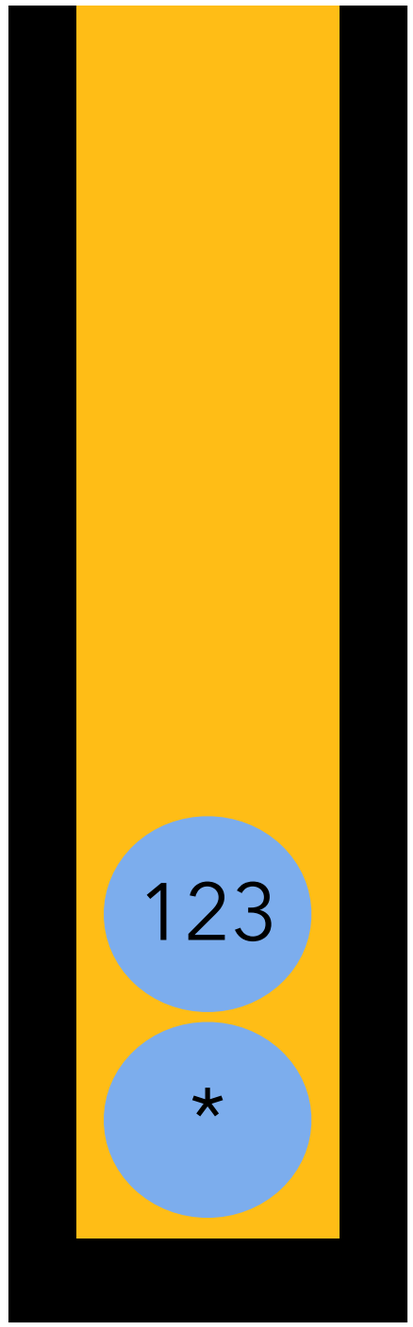
S

item = "123"
s.push(item)

將item推入堆疊




S

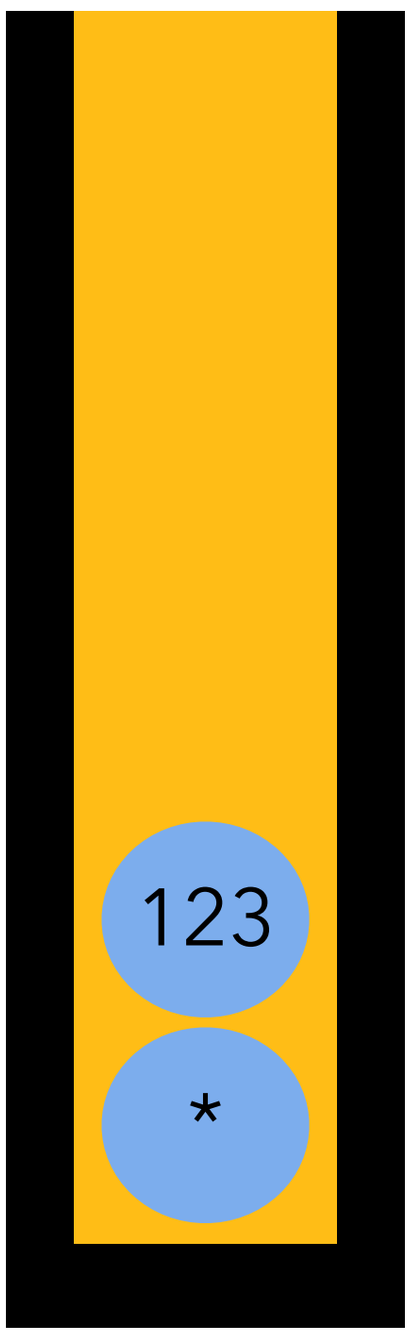



S

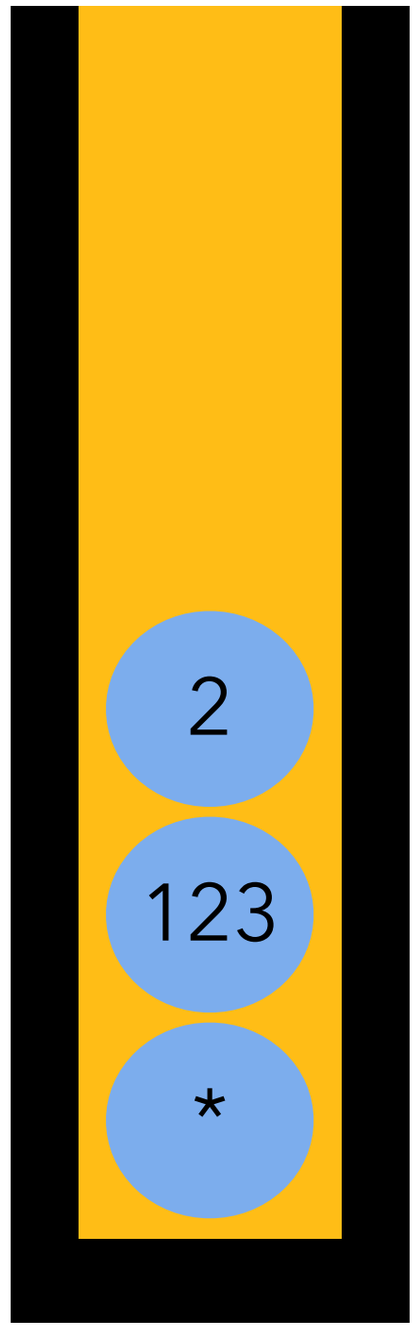
item = "2"

s.push(item)

將item推入堆疊

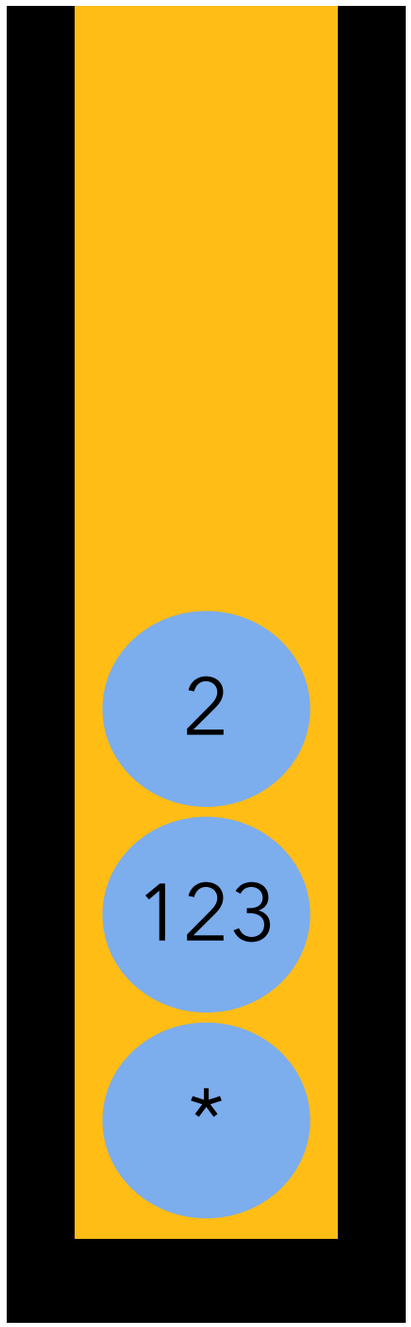



S




S

item = s.pop()




s

將item從堆疊彈出

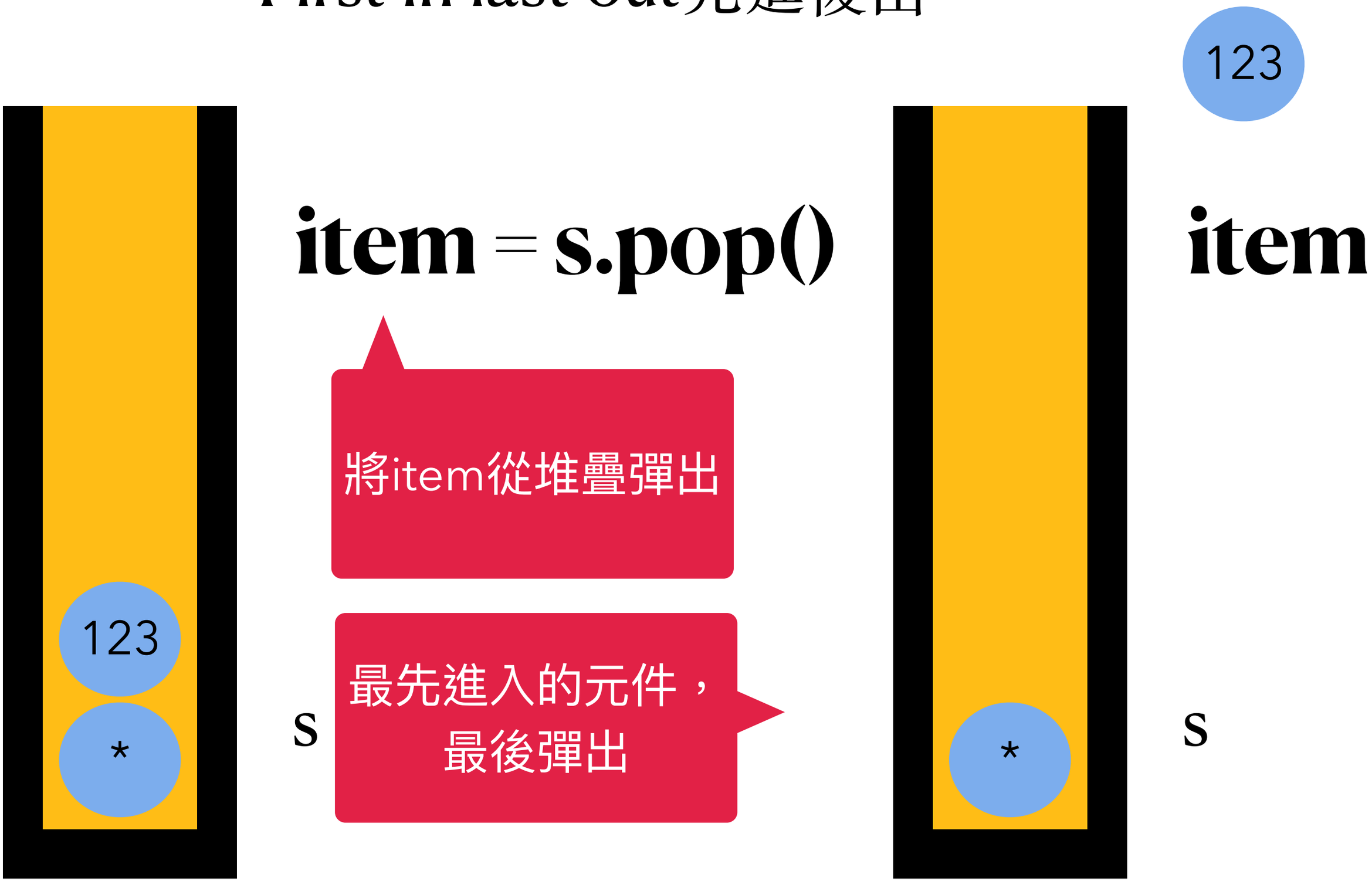
s.pop()代表方法


s

2

item

# First in last out先進後出




item = s.pop()
將item從堆疊彈出
S
item
S

# print(s.is_empty())

是否空堆疊？

s.empty()代表方法

S

stack2模組

# stack2.py

定義類別
Stack

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

class是保留字
宣告類別 Stack

__init__
是保留字
代表初始
化方法

當宣告一變數為類別Stack時，將該變數的items欄位，設定為空串列

# stack2.py

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

在類別設計階段，還沒有宣告為該類別的變數，Python以self代表未來將宣告的變數

def是保留字定義方法

self是保留字代表將連結的變數名稱

將變數(self)的items欄位初始化為空串列

# 類別的特性(property)與方法

# stack2.py

物件特性：欄位items，代表串列，用來儲存資料

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

輸入參數為目前正在定義的類別變數

定義方法
is_empty

輸出，return回傳布林值，代表是否為空串列

# stack2.py

```
class Stack:
    def __init__(self):
        self.items = []
    def is_empty(self):
        return self.items == []
    def push(self, data):
        self.items.append(data)
```

定義方法push

將輸入參數
data附加在串列中

```
class Stack:
    def __init__(self):
        self.items = []
    def is_empty(self):
        return self.items == []
    def push(self, data):
        self.items.append(data)
    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            print("stack is empty")
```

定義方法pop

檢查堆疊是否為空串列

回傳彈出的串列元件

否則，印出訊息

# stack2.py

```python
class Stack:

    def __init__(self):
        self.items = []

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

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

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            print("stack is empty")
```

```
Python Console
>>> from stack2 import Stack
>>> s = Stack()
>>> s.push("*")
>>> s.pop()
'*'
>>> s.is_empty()
True
```

# stack2.py

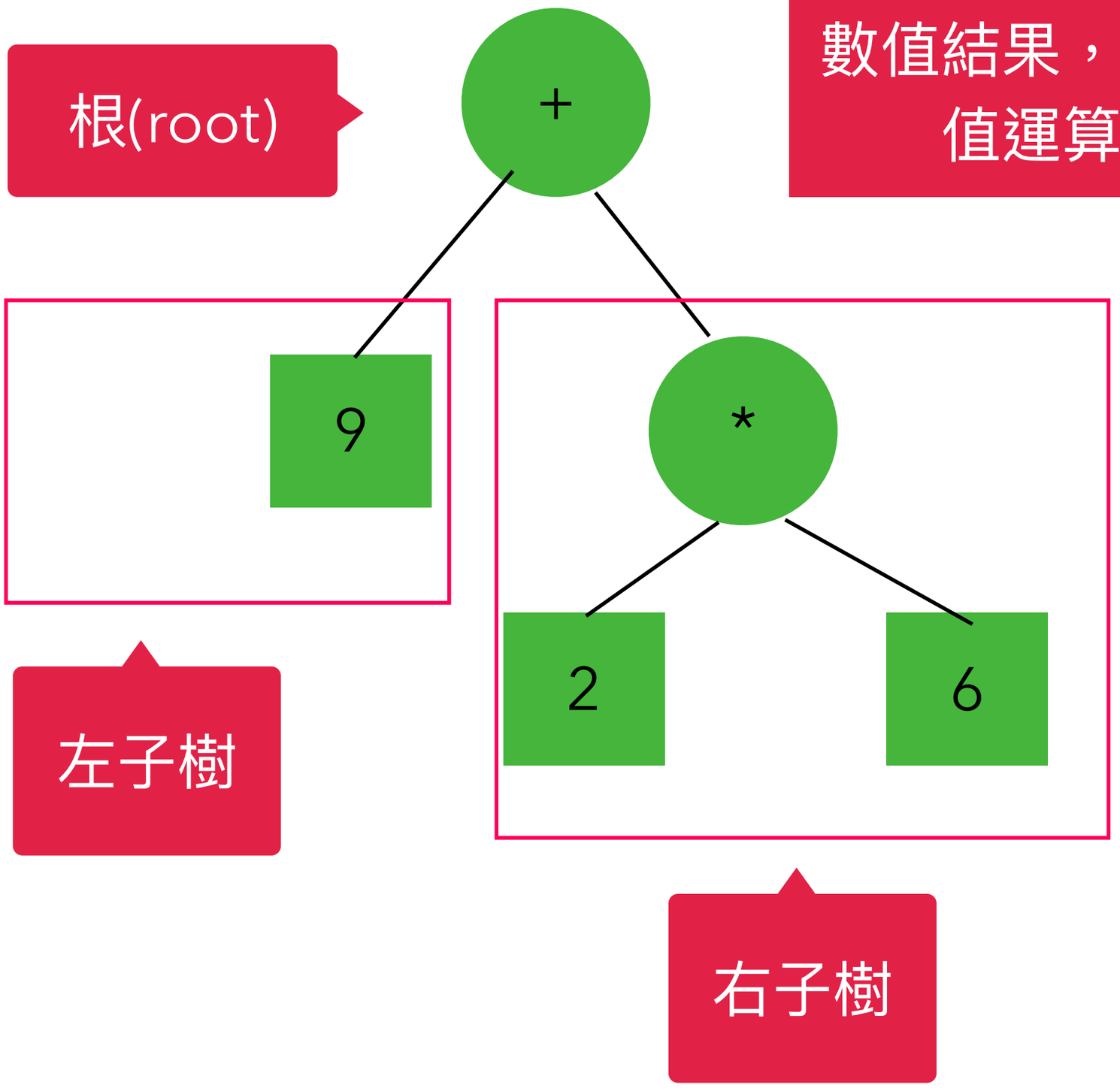
```python
class Stack:

    def __init__(self):
        self.items = []

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

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

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            print("stack is empty")
```

```
>>> from stack2 import Stack
>>> a = Stack()
>>> b = Stack()
>>> a.push("*")
>>> b.push("+")
>>> a.pop()
'*'
>>> b.pop()
'+'
```

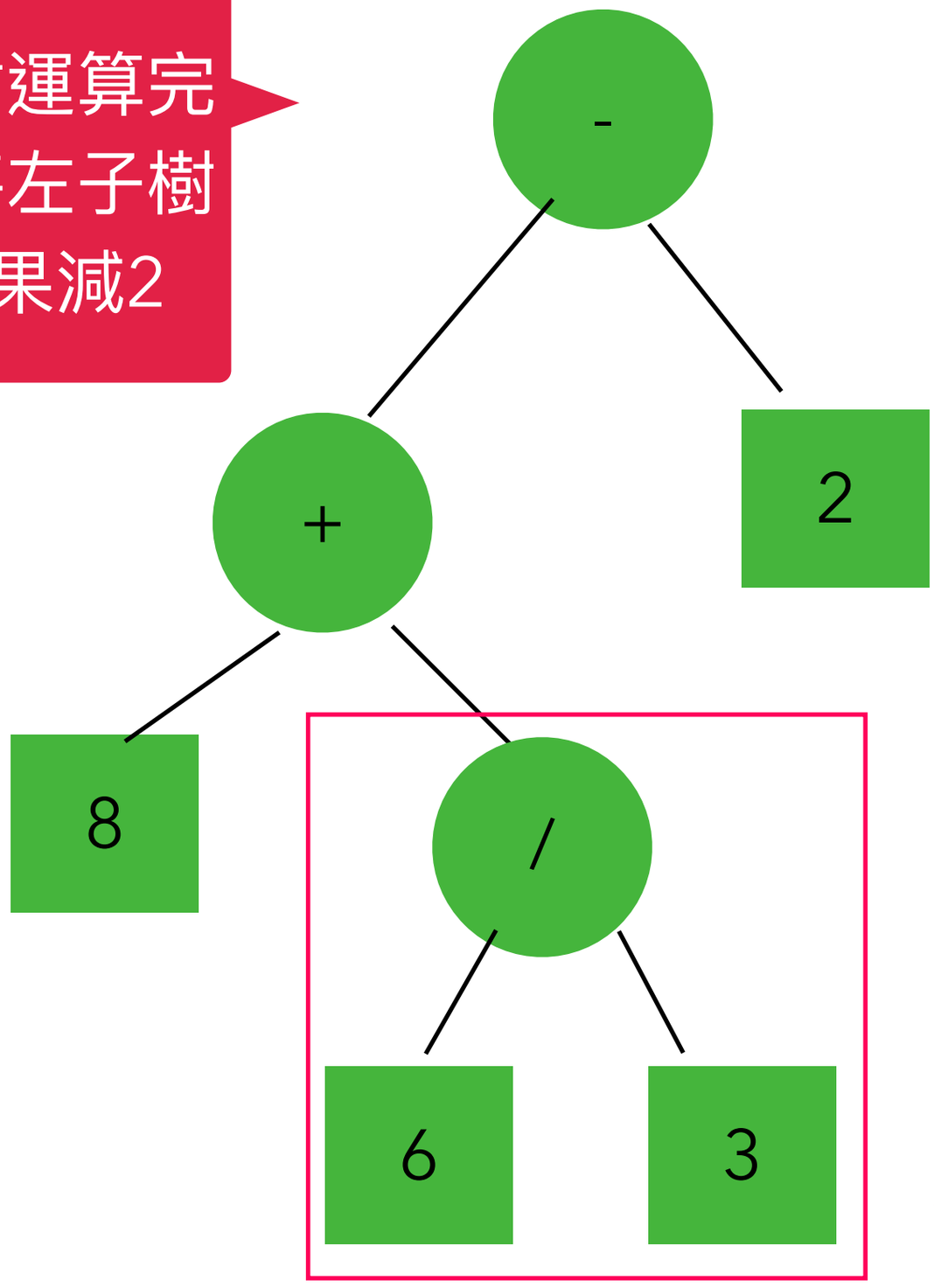
```
>>> from stack2 import Stack
>>> a = Stack()
>>> b = Stack()
>>> a.push("*")
>>> b.push("+")
>>> a.pop()
'*'
>>> b.pop()
'+'
>>> a.pop()
stack is empty
>>> b.pop()
stack is empty
```

# 堆疊與運算式的二元樹表示

以二元樹(binary tree)表示運算式，將左子樹運算的數值結果，與右子樹的數值運算結果相加

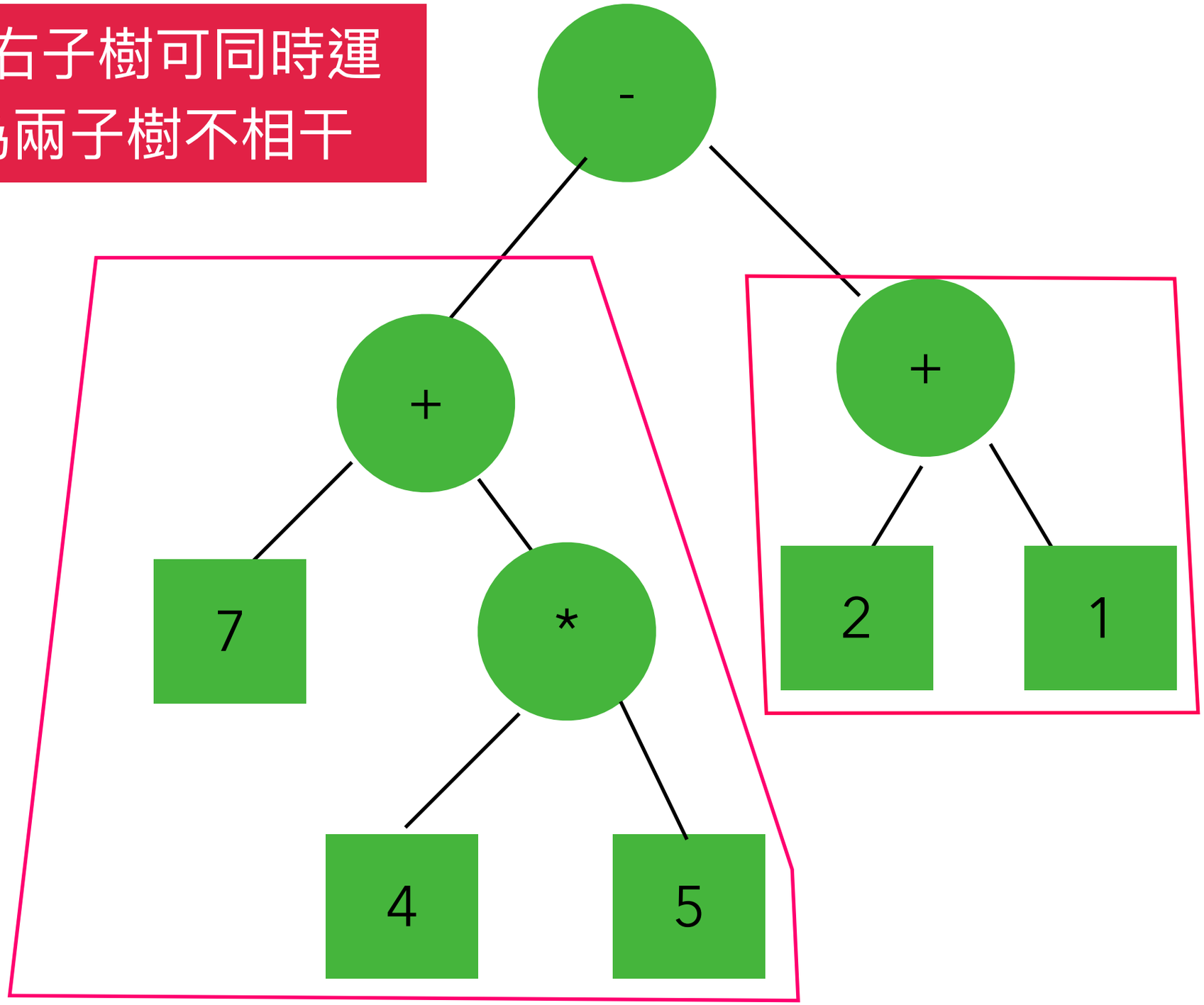


當左子樹運算完成後，將左子樹運算結果減2

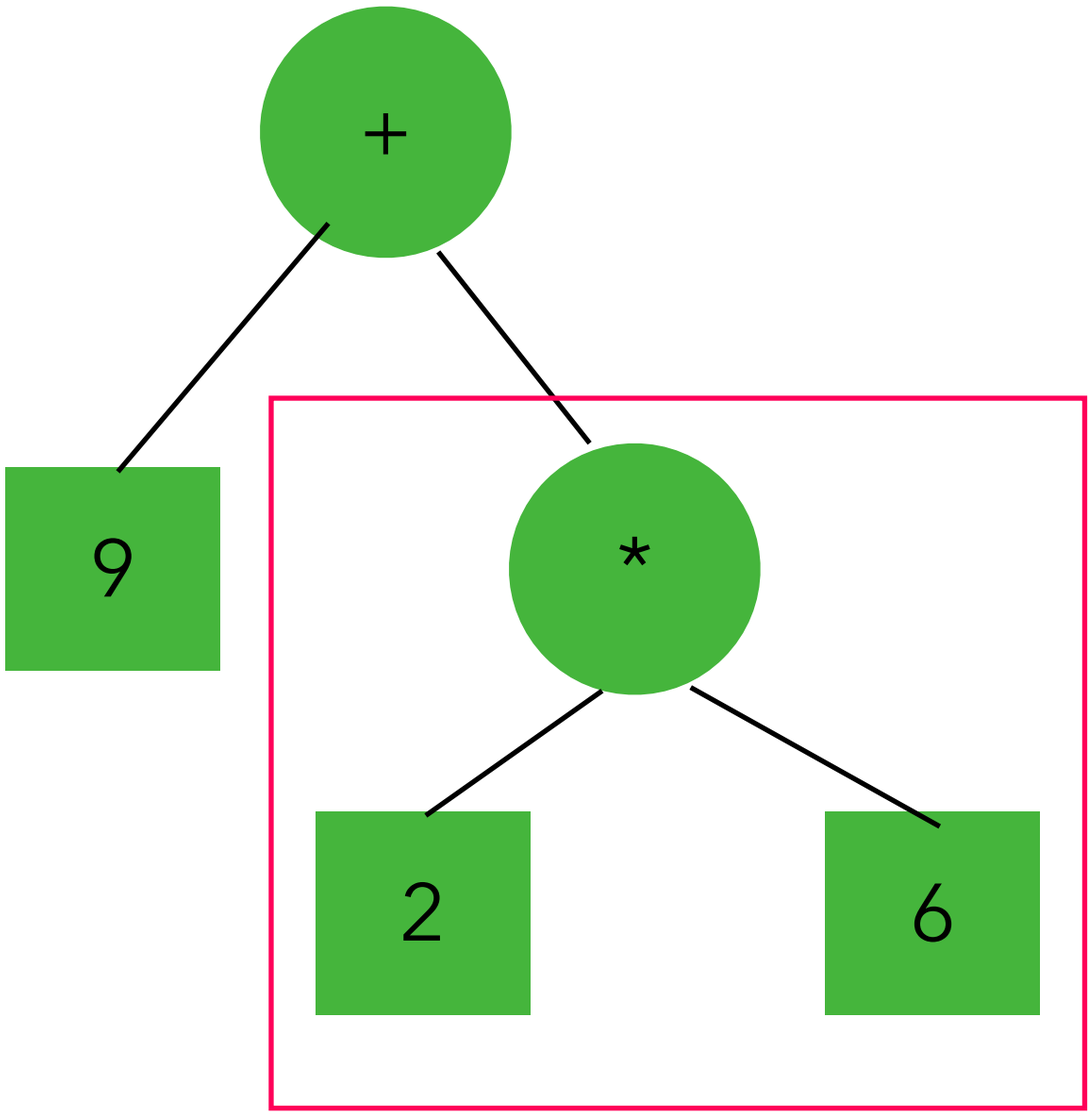



先進行底部的子樹運算

左子樹與右子樹可同時運算，因為兩子樹不相干

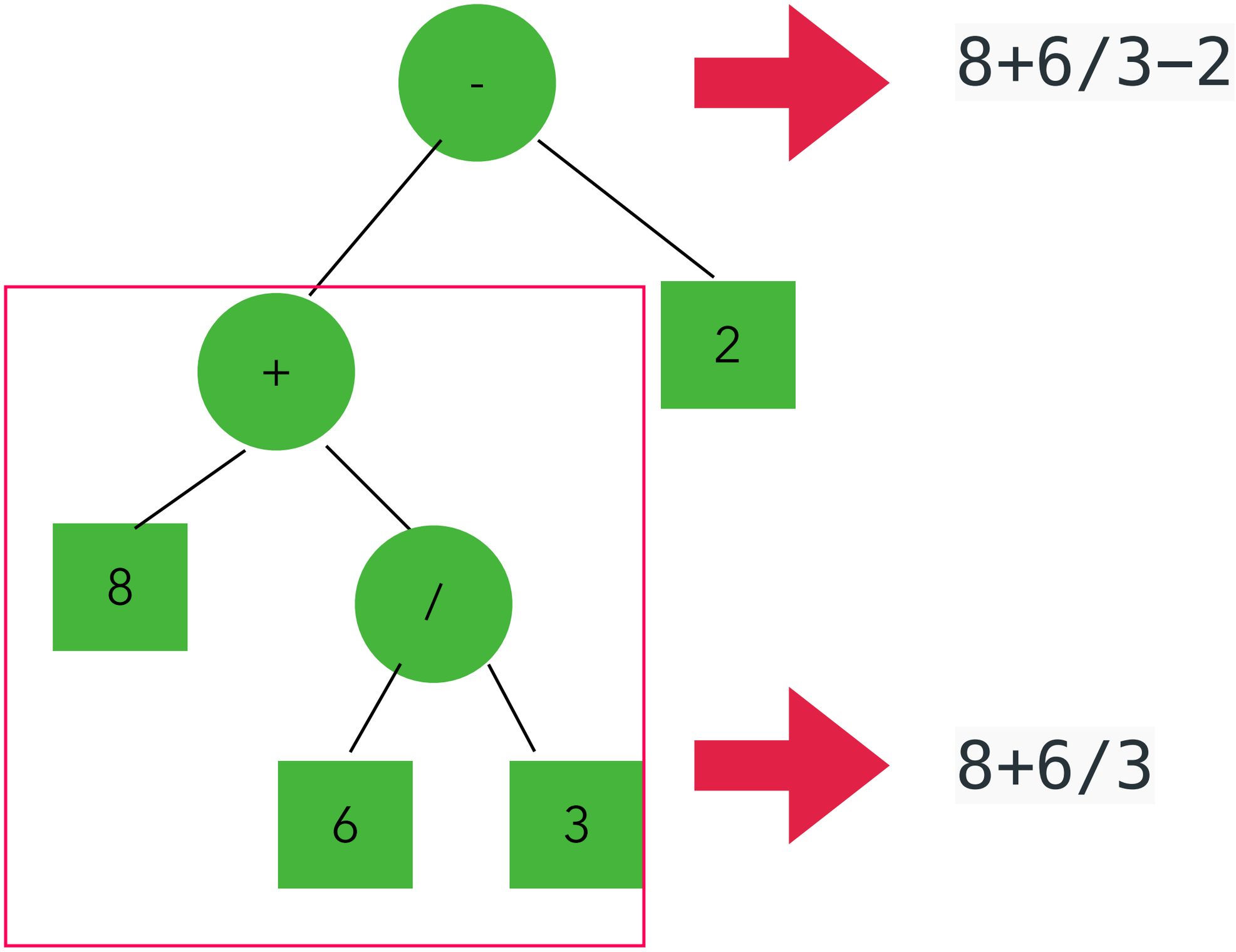


將二元樹表示為中置式：將運算子寫在兩個運算元的中間



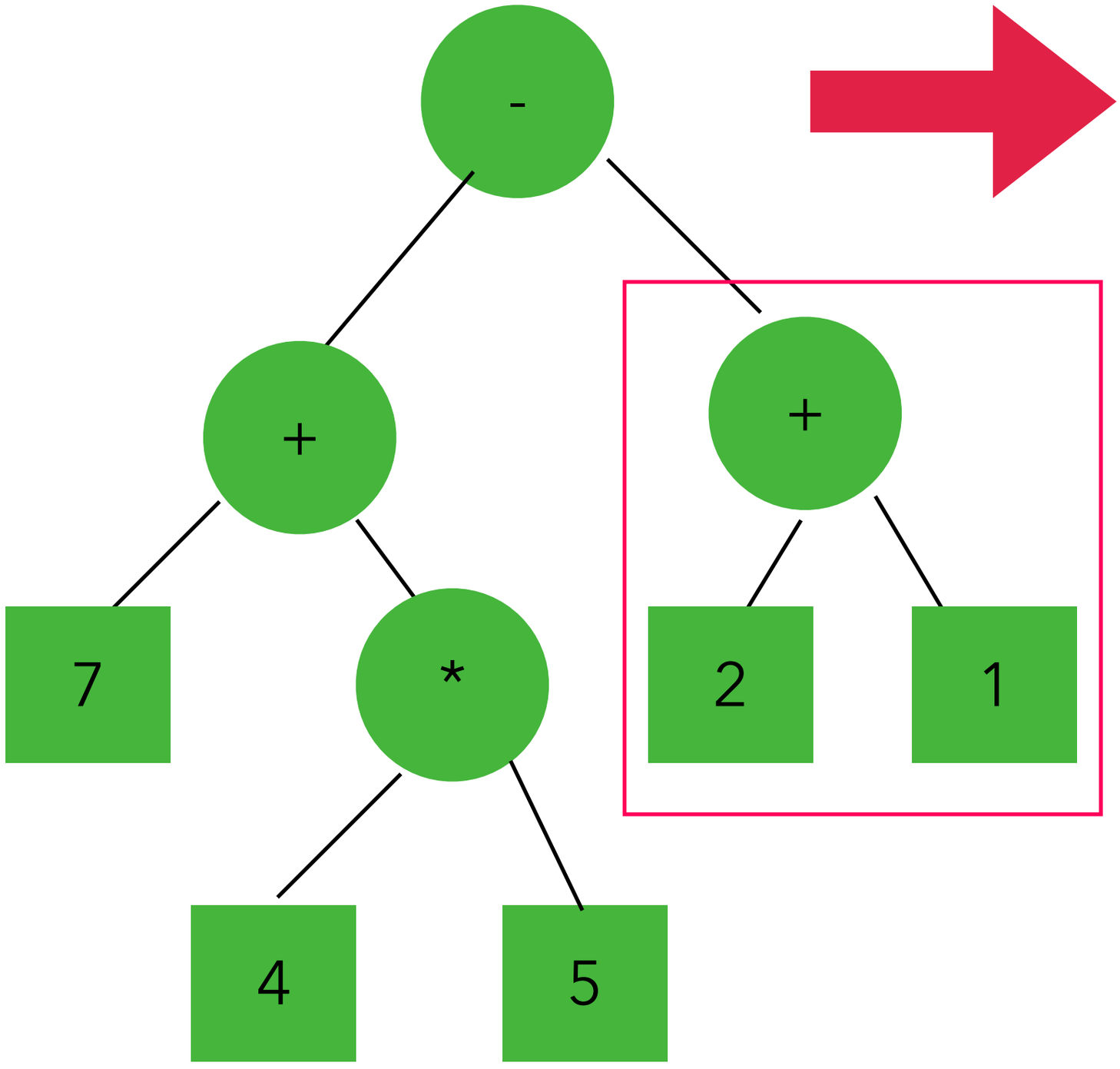

9+2*6

2*6



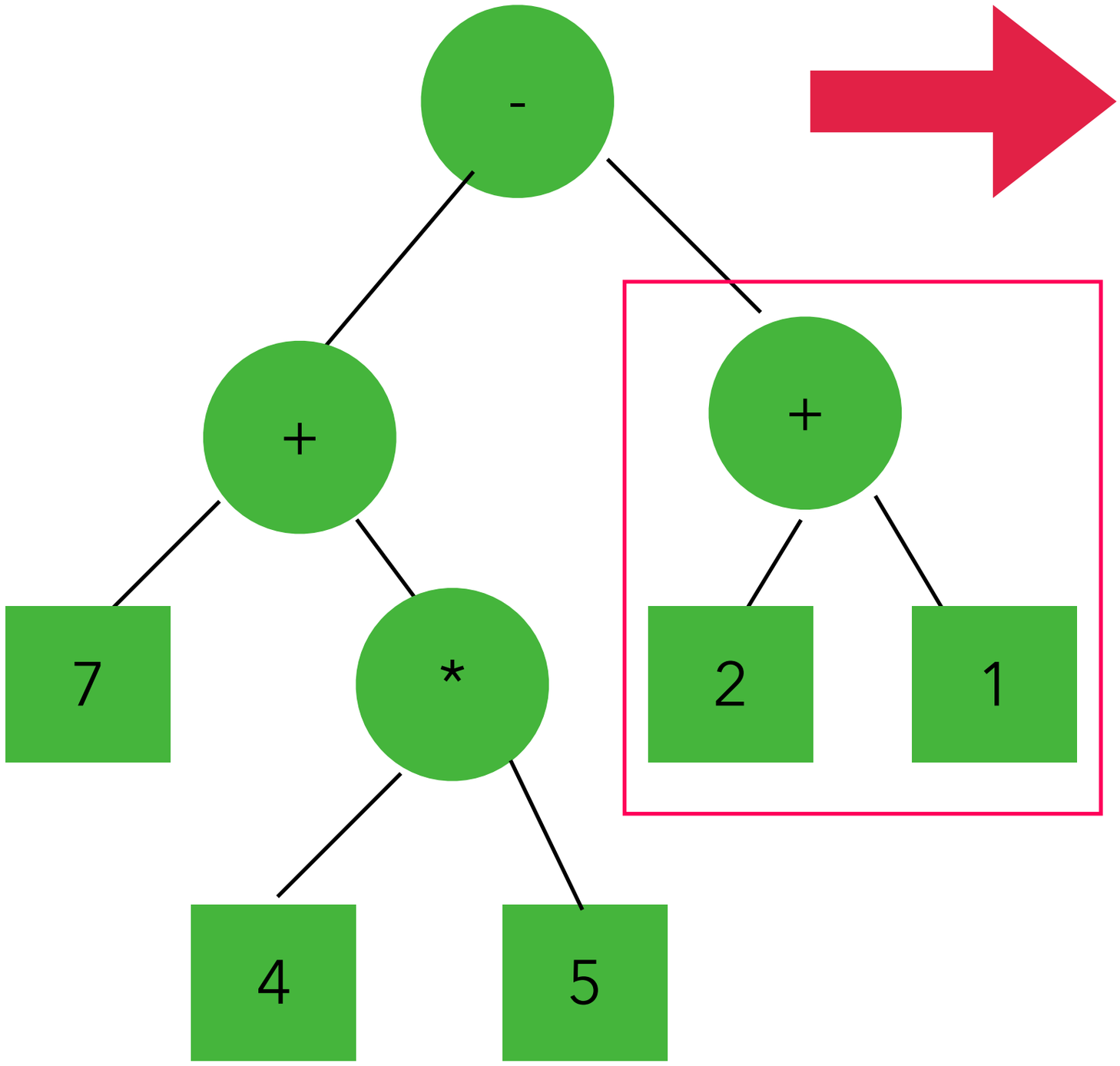

8+6/3−2

8+6/3




7+4*5−2+1

中置式的運算結果與二元樹的運算結果不符




7+4*5−(2+1)

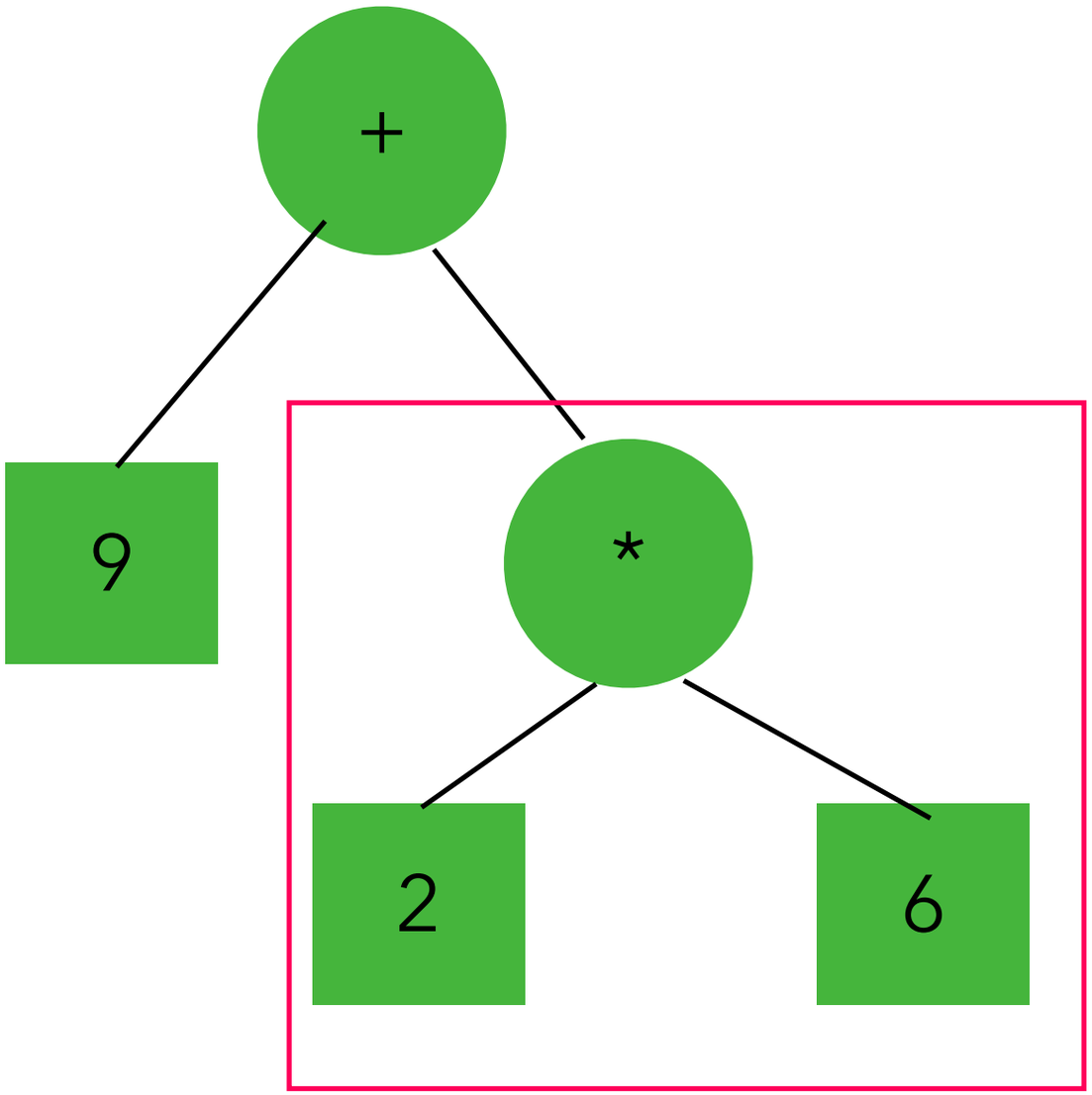
中置式需要使用括號，才能使得運算結果與二元樹表示式相符合

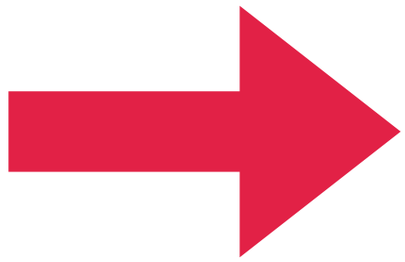
將二元樹表示轉為前置式：

**A.**先寫運算子

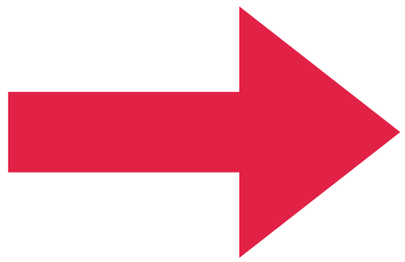
**B.**再寫左邊的運算元

**C.**最後再寫右邊的運算元

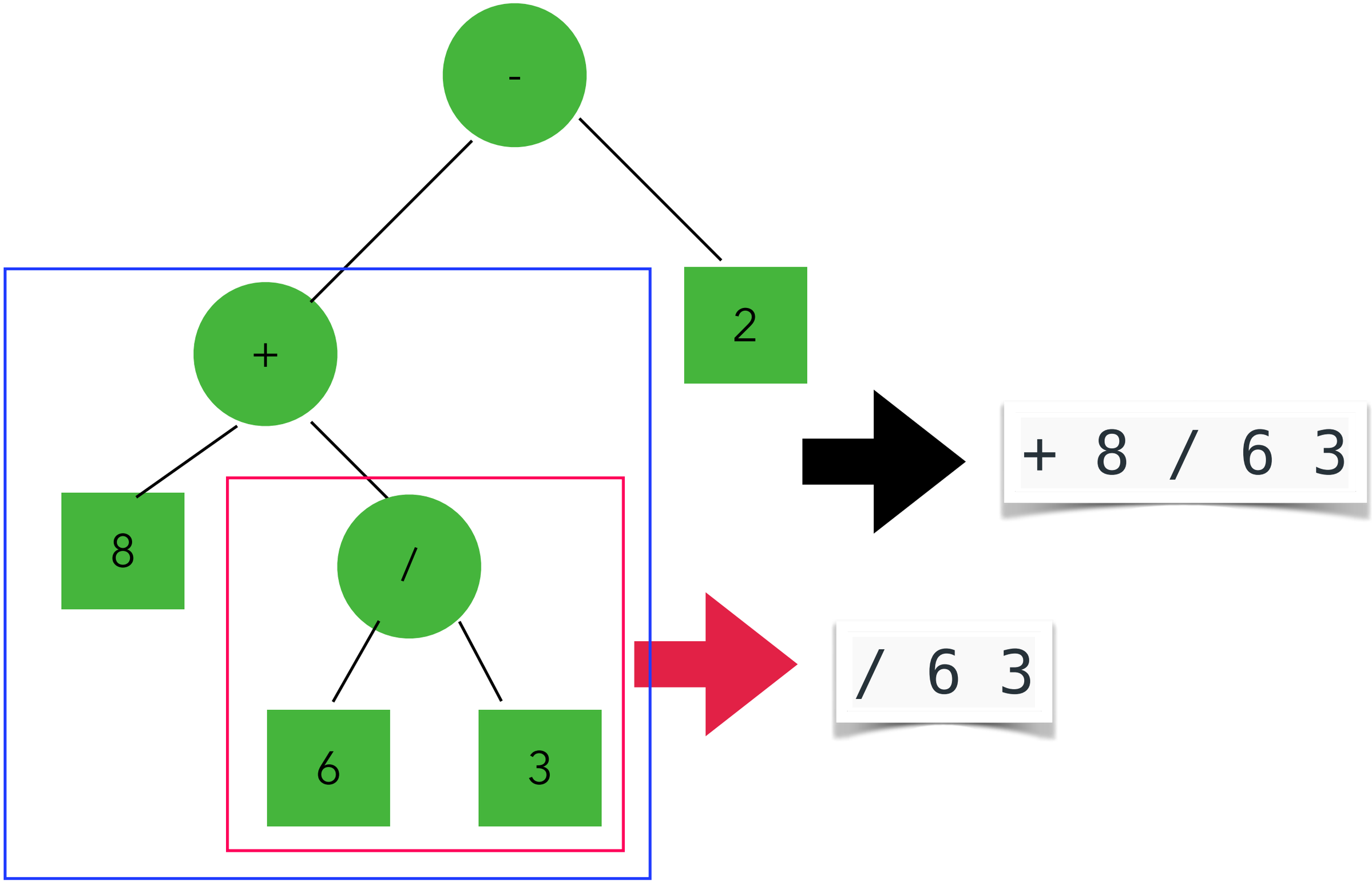

+
9
*
2
6

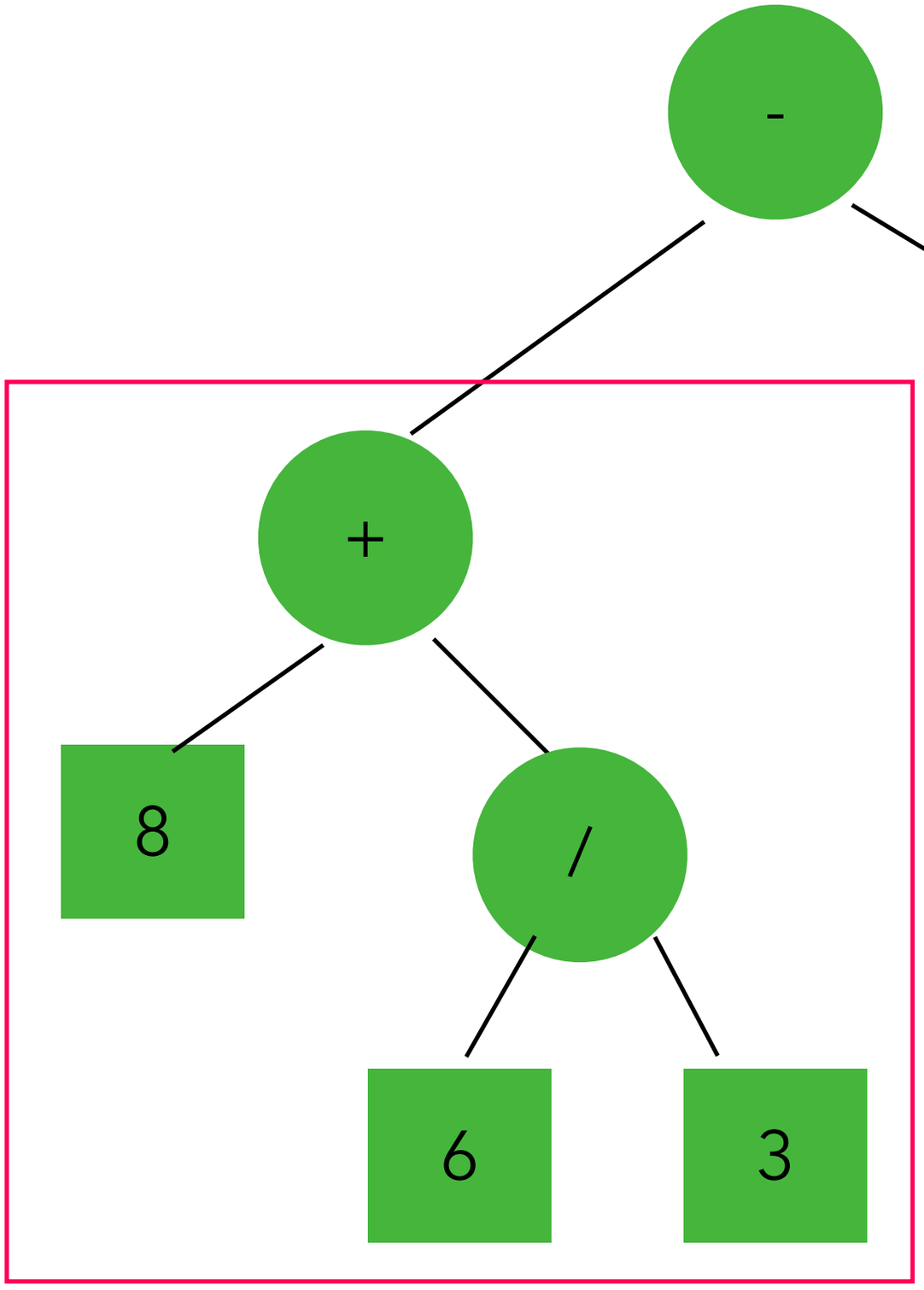



+ 9 * 2 6

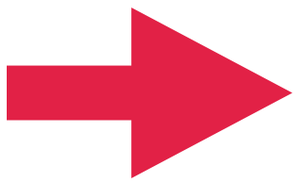



* 2 6

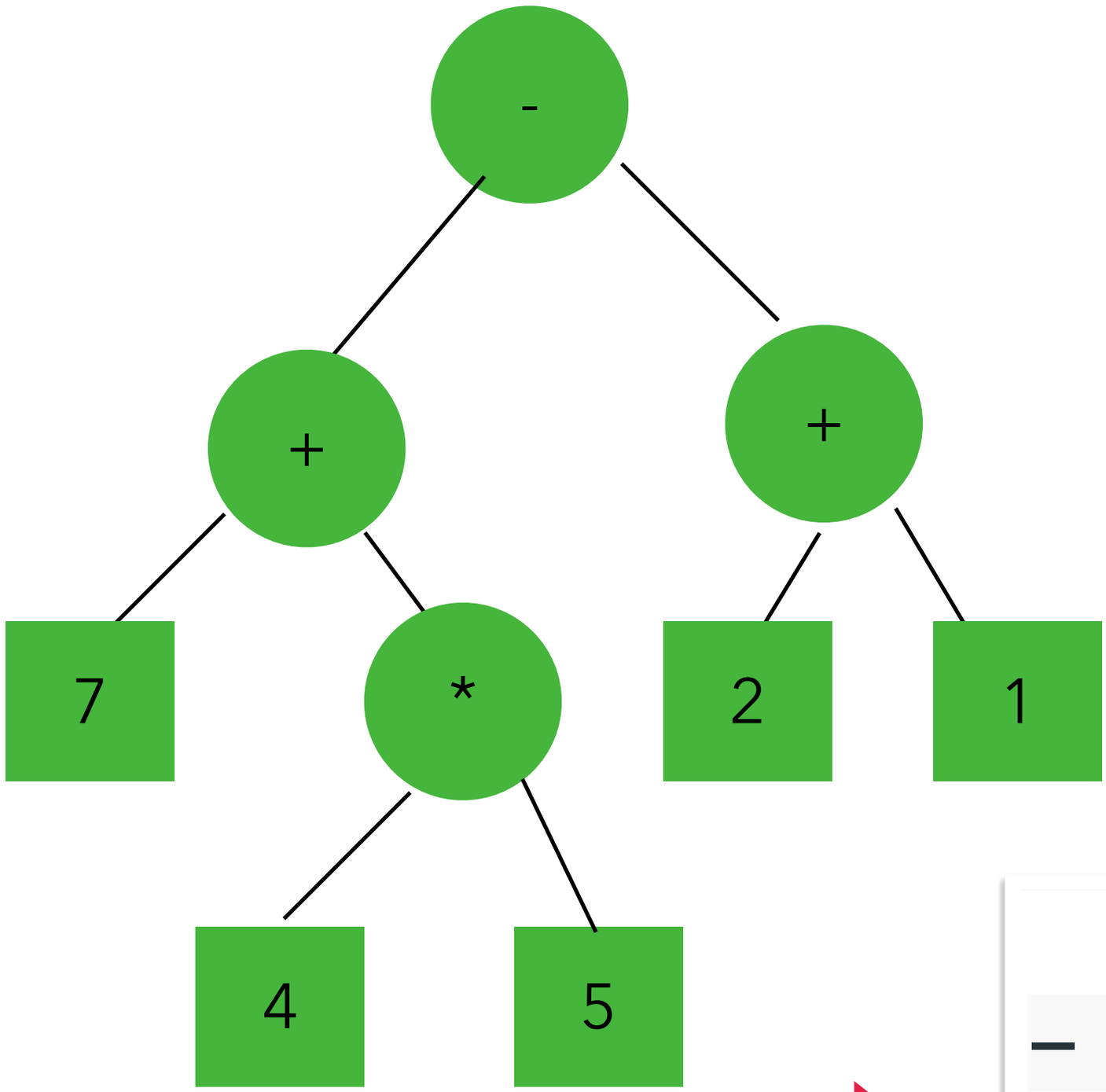

-
+
2
8
/
6
3
+ 8 / 6 3
/ 6 3


-
+
2
8
/
6
3


- + 8 / 6 3 2




+ 8 / 6 3




```
- + 7 * 4 5 + 2 1
```

# 如何撰寫Python程式計算前置式？

# 問題一

# 輸入前置式

```
ss = input("a prefix expression: ")
items = ss.split()
print(items)
```

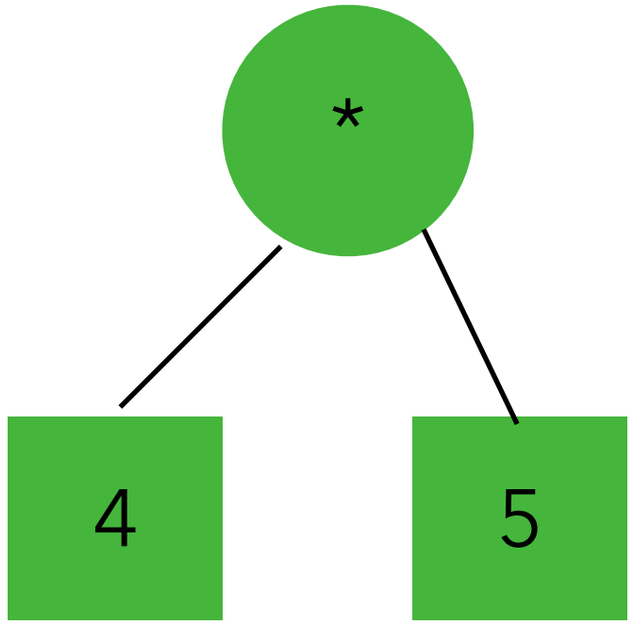
```
a prefix expression: - + 7 * 4 5 + 2 1
['-', '+', '7', '*', '4', '5', '+', '2', '1']
```

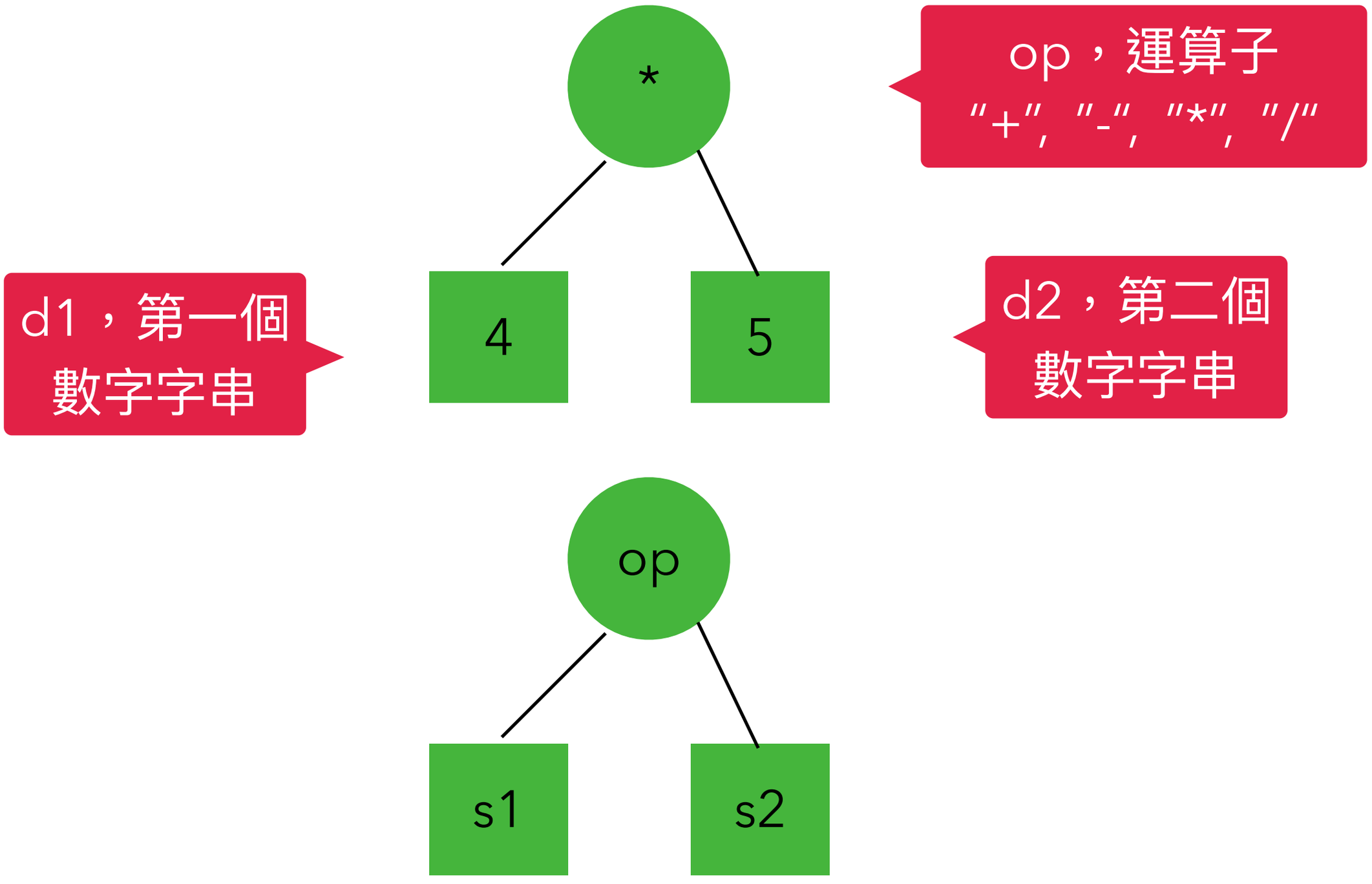
item為串列，元素為字串

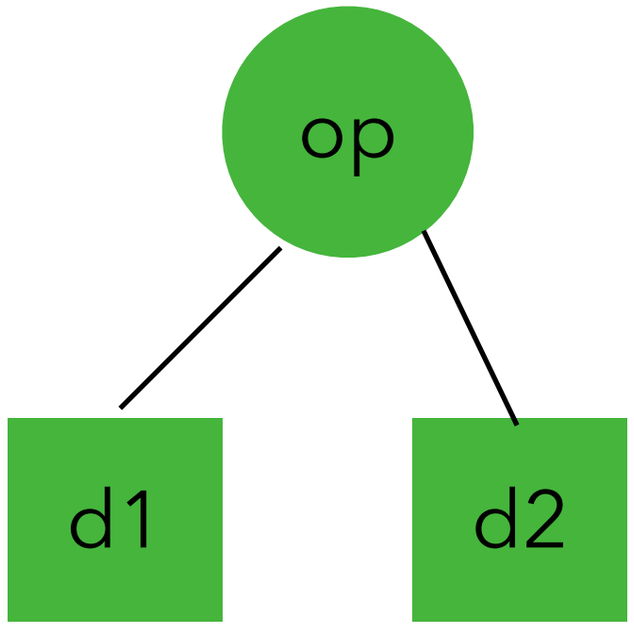
運算子，"+", "-", "*", "/"

# 問題二

## 計算單一的二元運算




*
4
5
op
s1
s2
op，運算子
"+", "-", "*", "/"
d1，第一個
數字字串
d2，第二個
數字字串




**eval(d1+op+d2)**

合併**d1+op+d2**得到運算式字串，使用**eval(d1+op+d2)**得到算式答案

使用
eval(s1+op+s2)得
到算式答案

```
op = "*"
s1 = "4"
s2="5"
eval(s1+op+s2)
```

```
op = "/"
s1 = "4"
s2 = "5"
str(eval(s1+op+s2))
'0.8'
```

將輸出轉為
字串

# 問題三

# 使用堆疊計算前置式

計算步驟：

1. 輸入中置式字串ss，以ss.split()，將字串分割成字元，儲存在items中
2. 宣告堆疊s
3. for i in range(len(items)-1,-1,-1):

如果items[i]是數字，推入堆疊s

否則

A. 設定該元素為op
B. 從堆疊s中彈出元素，設定為d1，再從堆疊中彈出元素，設定為d2
C. 使用問題二中的方法算出答案ans，再將ans推入堆疊s

```
ss = input("a prefix expression: ")
items = ss.split()
print(items)
for i in range(len(items)-1,-1,-1):
    print(items[i])
```

```
a prefix expression: - + 7 * 4 5 + 2 1
['-', '+', '7', '*', '4', '5', '+', '2', '1']
1
2
+
5
4
*
7
+
-
```

最後一個元素先處理

第一個元素最後處理

串列items中包含9個元素，範圍從9到-1，每次遞減-1，不包含最後的-1，但包含0

```
ss = input("a prefix expression: ")
items = ss.split()
print(items)
for i in range(len(items)-1,-1,-1):
    print(items[i])
```

宣告一個堆疊，名稱為s

如果items[i]是數字，推入堆疊s

否則

A. 設定該元素為op
B. 從堆疊s中彈出元素，設定為d1，再從堆疊中彈出元素，設定為d2
C. 使用問題二中的方法算出答案ans，再將ans推入堆疊s

# 模擬運算

+ 9 * 2 6

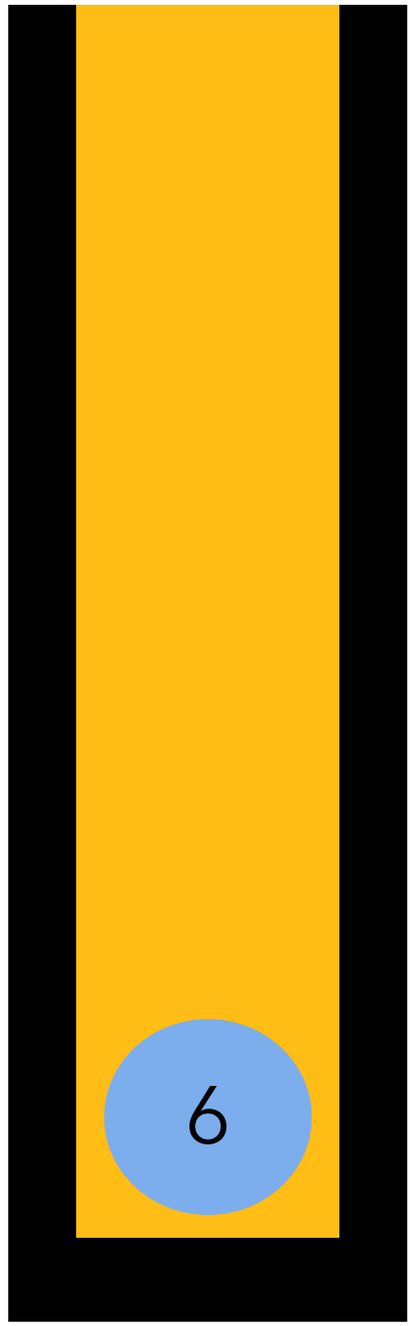
d1 op d2 ans

6

# 如果items[i]是數字，推入堆疊s

否則

A. 設定該元素為op
B. 從堆疊s中彈出元素，設定為d1，再從堆疊中彈出元素，設定為d2
C. 使用問題二中的方法算出答案ans，再將ans推入堆疊s




S

+ 9 * 2 6

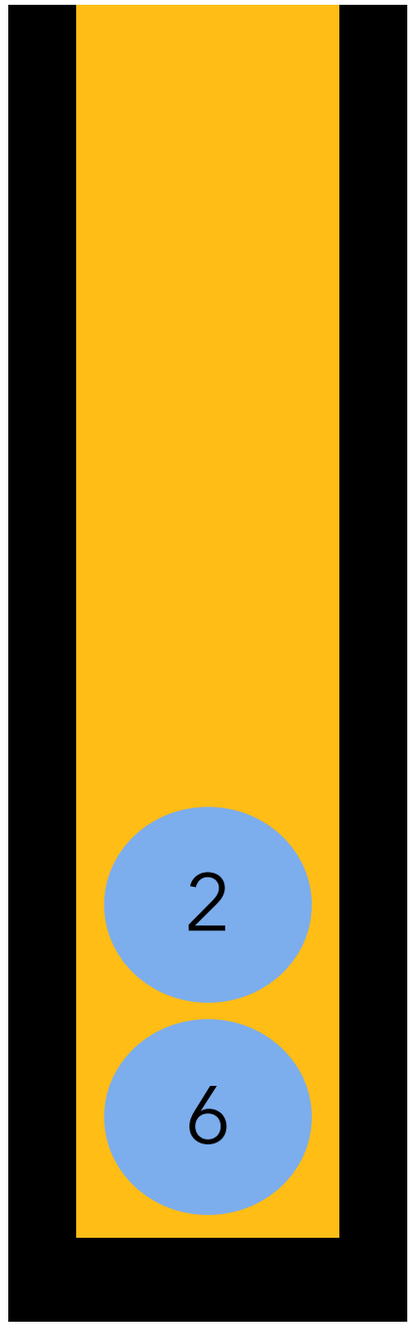
2

d1 | op | d2 | ans

**如果items[i]是數字，推入堆疊s**

**否則**

A. 設定該元素為op
B. 從堆疊s中彈出元素，設定為d1，再從堆疊中彈出元素，設定為d2
C. 使用問題二中的方法算出答案ans，再將ans推入堆疊s



+ 9 * 2 6

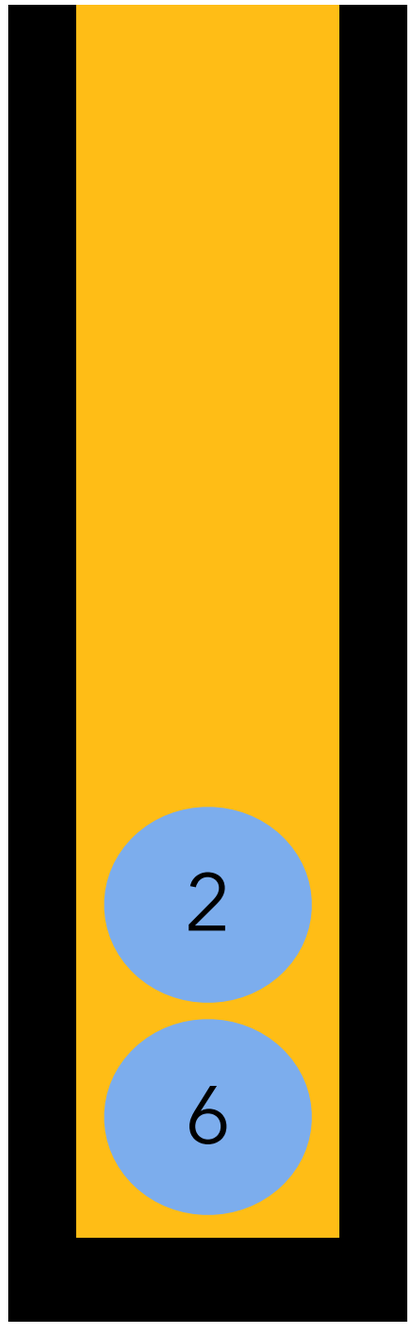
*

d1 * d2 ans

# 如果items[i]是數字，推入堆疊s

否則

A. 設定該元素為op
B. 從堆疊s中彈出元素，設定為d1，再從堆疊中彈出元素，設定為d2
C. 使用問題二中的方法算出答案ans，再將ans推入堆疊s




S

+ 9 * 2 6

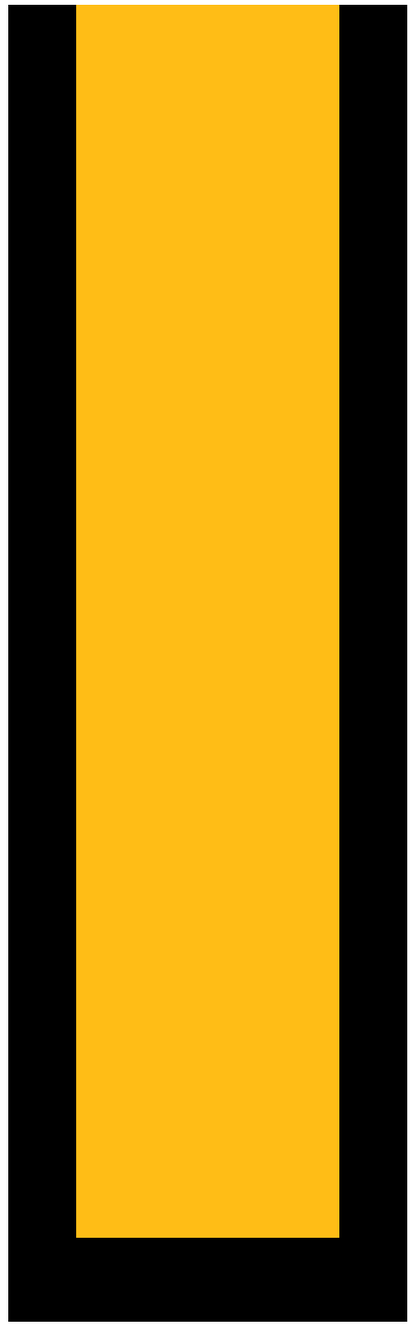
*

2 * 6 12

# 如果items[i]是數字，推入堆疊s

否則

A. 設定該元素為op
B. 從堆疊s中彈出元素，設定為d1，再從堆疊中彈出元素，設定為d2
C. 使用問題二中的方法算出答案ans，再將ans推入堆疊s



S

+ 9 * 2 6

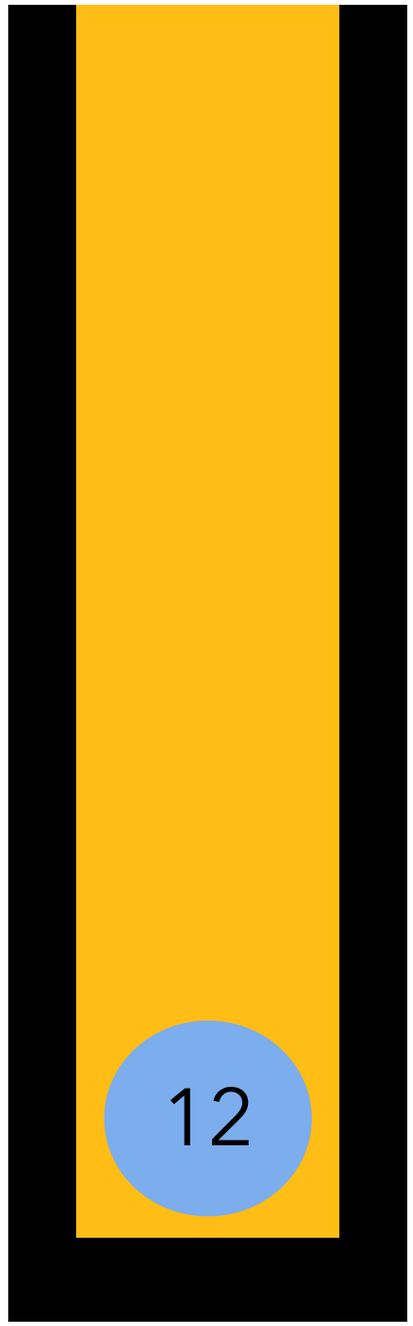
*

2 * 6 12

# 如果items[i]是數字，推入堆疊s

# 否則

A. 設定該元素為op
B. 從堆疊s中彈出元素，設定為d1，再從堆疊中彈出元素，設定為d2
C. 使用問題二中的方法算出答案ans，再將ans推入堆疊s




S

+ 9 * 2 6

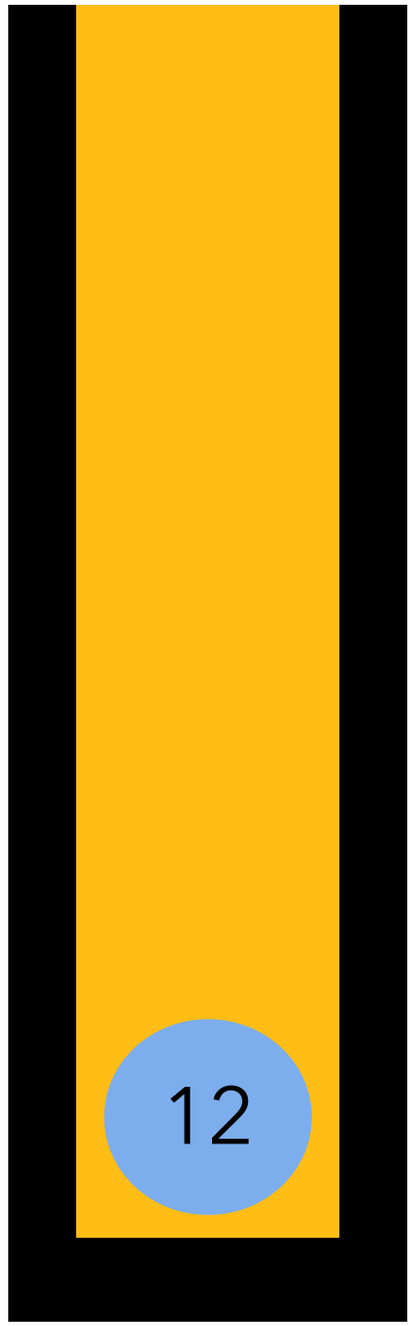
9

d1 op d2 ans

# 如果items[i]是數字，推入堆疊s

否則

A. 設定該元素為op
B. 從堆疊s中彈出元素，設定為d1，再從堆疊中彈出元素，設定為d2
C. 使用問題二中的方法算出答案ans，再將ans推入堆疊s




S

+ 9 * 2 6

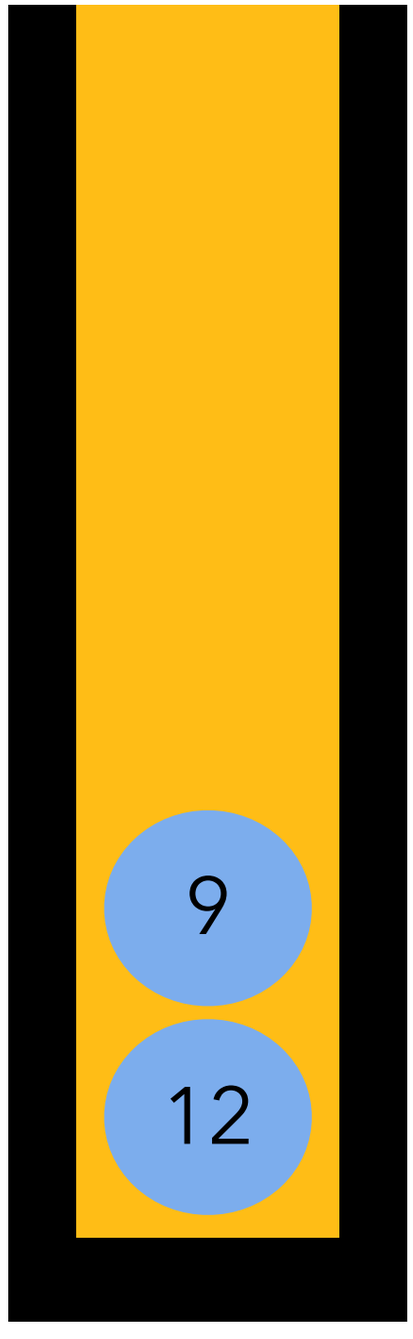
9

d1 | op | d2 | ans

**如果items[i]是數字，推入堆疊s**

**否則**

A. 設定該元素為op
B. 從堆疊s中彈出元素，設定為d1，再從堆疊中彈出元素，設定為d2
C. 使用問題二中的方法算出答案ans，再將ans推入堆疊s



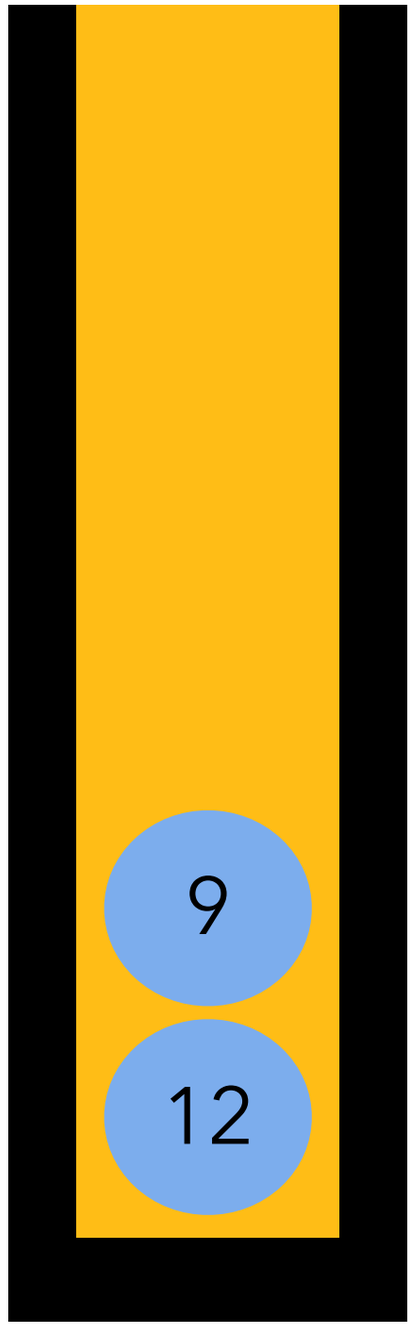
+ 9 * 2 6

+

d1 op d2 ans

# 如果items[i]是數字，推入堆疊s
# 否則

A. 設定該元素為op
B. 從堆疊s中彈出元素，設定為d1，再從堆疊中彈出元素，設定為d2
C. 使用問題二中的方法算出答案ans，再將ans推入堆疊s




S

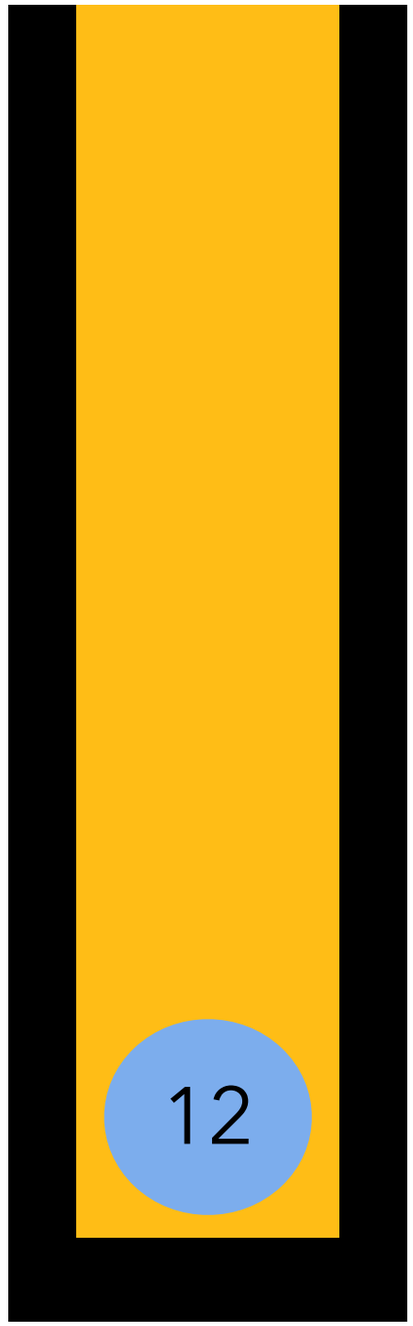
+ 9 * 2 6

9 + 12 Ans

+

# 如果items[i]是數字，推入堆疊s
# 否則

A. 設定該元素為op
B. 從堆疊s中彈出元素，設定為d1，再從堆疊中彈出元素，設定為d2
C. 使用問題二中的方法算出答案ans，再將ans推入堆疊s




S

+ 9 * 2 6

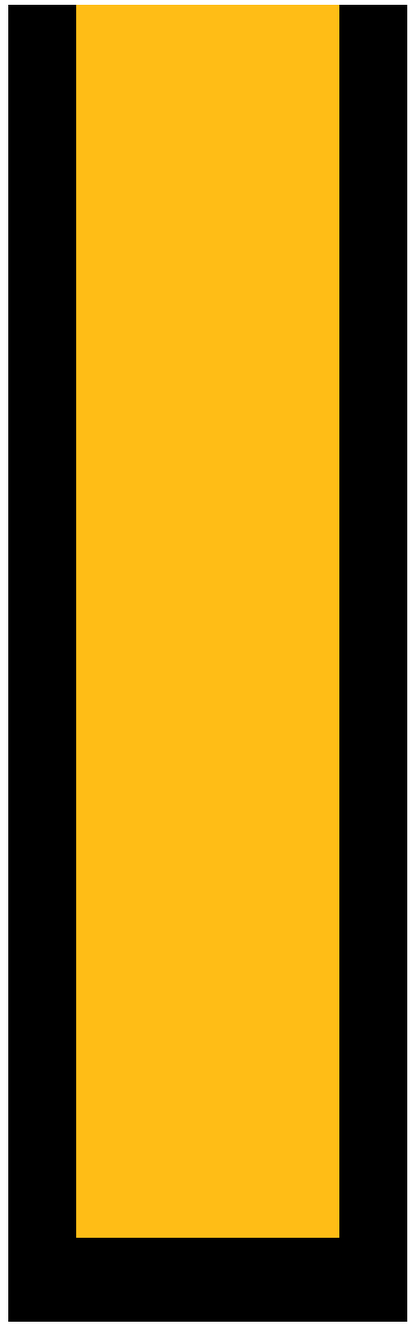
+

12 + 9 21

# 如果items[i]是數字，推入堆疊s

否則

A. 設定該元素為op
B. 從堆疊s中彈出元素，設定為d1，再從堆疊中彈出元素，設定為d2
C. 使用問題二中的方法算出答案ans，再將ans推入堆疊s



S

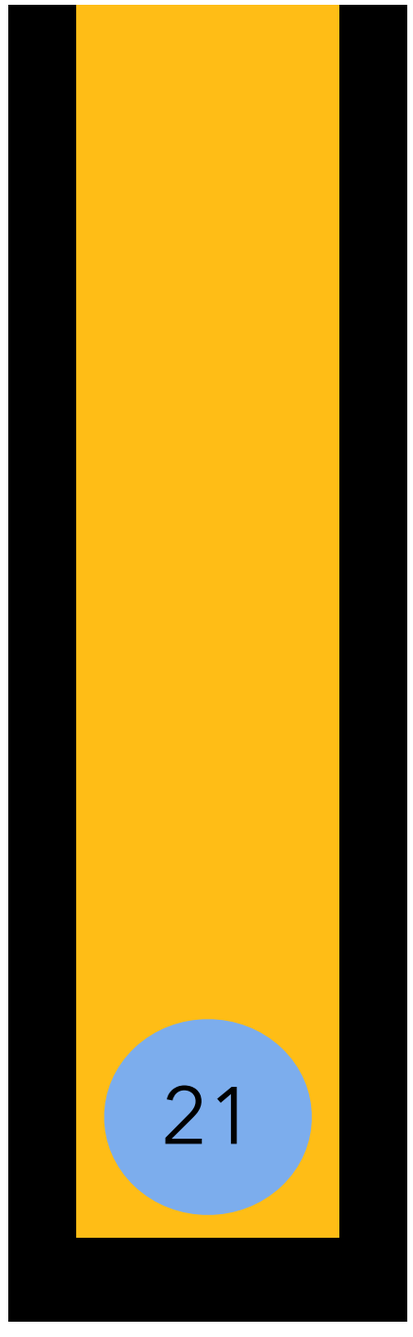
+ 9 * 2 6

+

12 + 9 21

# 如果items[i]是數字，推入堆疊s
# 否則

A. 設定該元素為op
B. 從堆疊s中彈出元素，設定為d1，再從堆疊中彈出元素，設定為d2
C. 使用問題二中的方法算出答案ans，再將ans推入堆疊s




S

# 互動操作堆疊

```
from stack2 import Stack
s = Stack()
while True:
    print('push <value>')
    print('pop')
    print('quit')
    do = input('What to do?').split()
    op = do[0].strip().lower()
if op == 'push':
    s.push(int(do[1]))
elif op == 'pop':
    if s.is_empty():
        print('Stack is empty')
    else:
        print('Popped value:',s.pop())
elif op == 'quit':
    break
```

push或pop?

將整數推入堆疊

將整數彈出堆疊