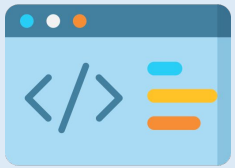
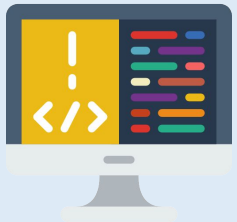
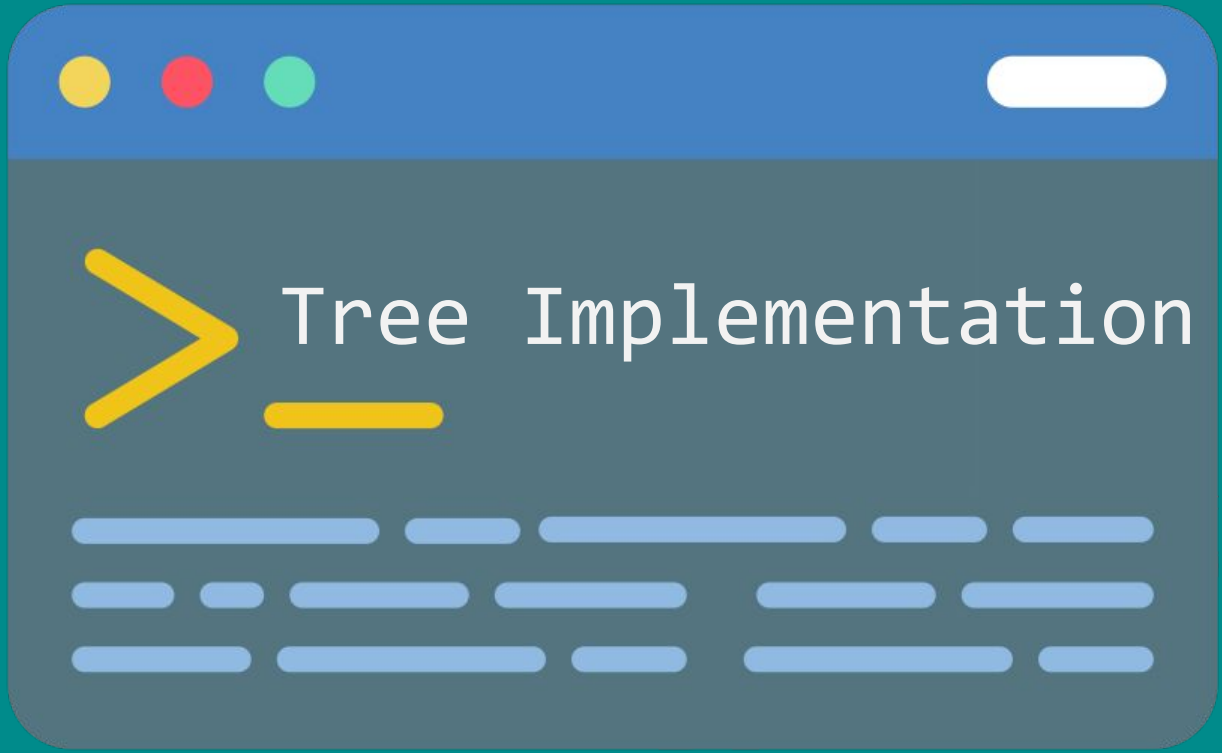


Tree Traversal



TA : LJH


Tree Implementation

Tree Implementation

```
vector<vector<int>> adj_list(n);
for(int i = 0; i < n - 1; ++i) {
    //input an edge (u, v)
    adj_list[u].push_back(v);
    adj_list[v].push_back(u);
}
```

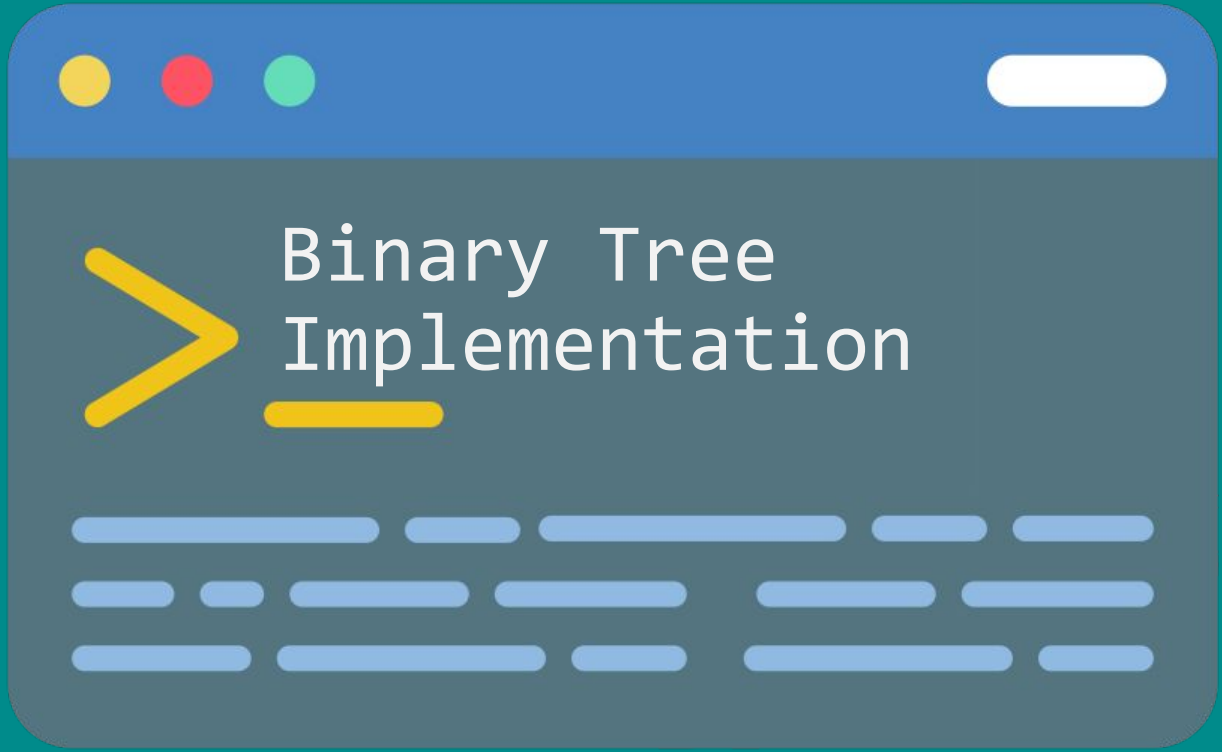
在前面的課程有教過如何儲存一張圖，而我們也可以把一棵樹當成一張無向圖來儲存。

Tree Implementation

```
vector<vector<int>> adj_list(n);
for(int i = 0; i < n - 1; ++i) {
    //input an edge (u, v)
    adj_list[u].push_back(v);
    adj_list[v].push_back(u);
}
```

在前面的課程有教過如何儲存一張圖，而我們也可以把一棵樹當成一張無相圖來儲存。
如果這棵樹的節點數為 n ，那他就會有 n - 1 條邊。

Binary Tree

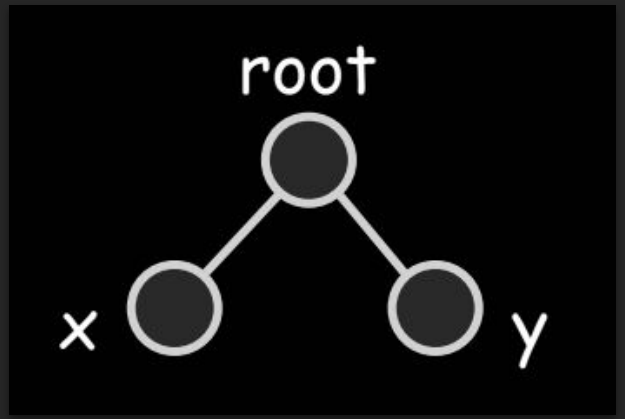

Binary Tree
Implementation

Binary Tree Implementation

```
struct Node {
    int value;
    Node *left;
    Node *right;
    Node(int x, Node *l = NULL, Node *r = NULL) : {}
};
```

對於一些比較特別的樹(ex ： 二元樹)我們也可以用其他方式去儲存，像是自己寫一個節點的 struct，用兩個指標指向那個節點的左子樹跟右子樹的根

Binary Tree Implementation



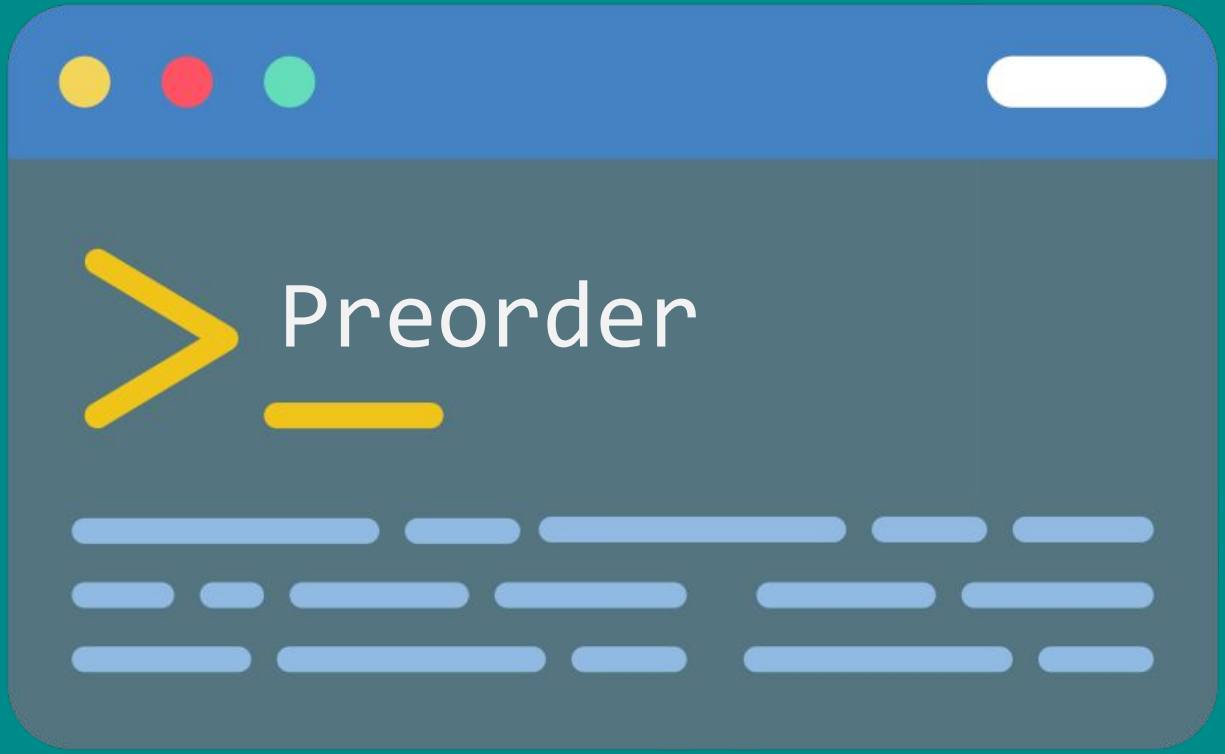

```
Node *lc = new Node(2);
Node *rc = new Node(3);
Node *root = new Node(1, lc, rc);
```

對於一些比較特別的樹(ex ： 二元樹)我們也可以用其他方式去儲存，像是自己寫一個節點的 struct，用兩個指標指向那個節點的左子樹跟右子樹的根

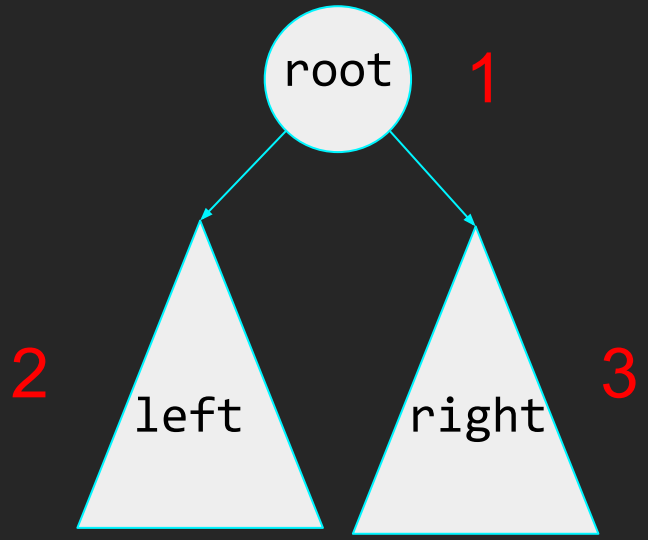

二元樹的遍歷

- 遍歷：前序、中序、後序
- 差異：什麼時候輸出節點的值

Binary Tree Traversal

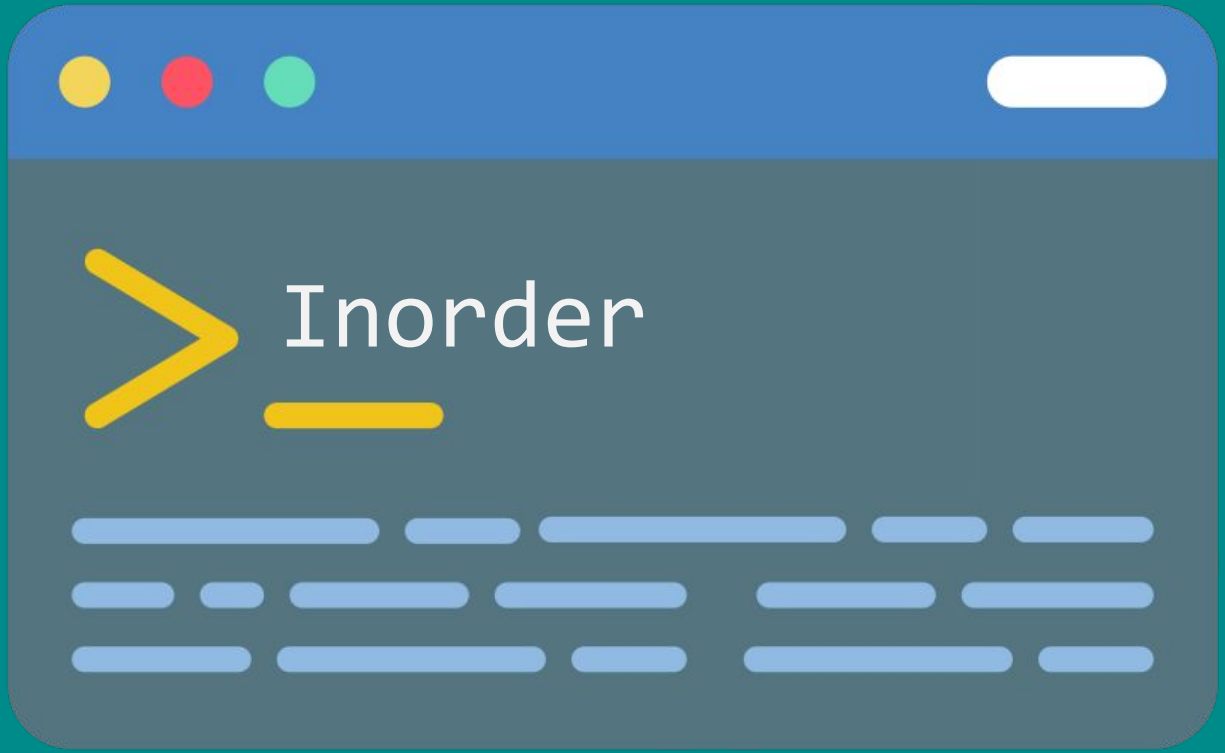

Preorder

Preorder

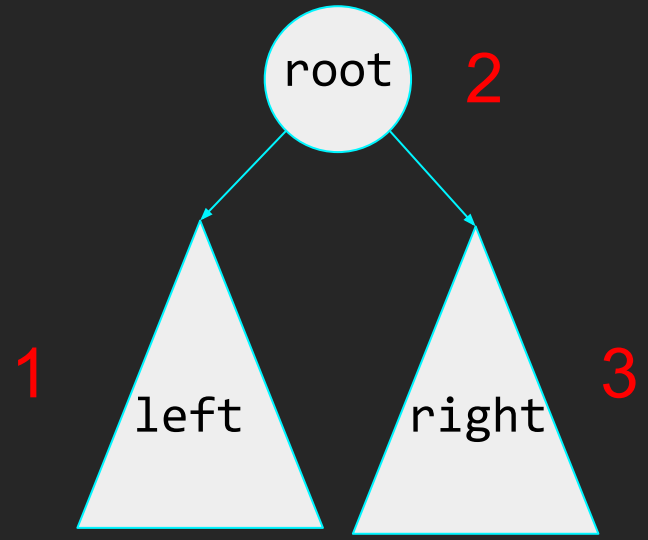



```
void preorder(node *t) {
    if(!t) return;
    cout << t->val << ' ';
    preorder(t->l);
    preorder(t->r);
}
```

先輸出根節點的值，再前序遍歷左子樹，最後遍歷右子樹。

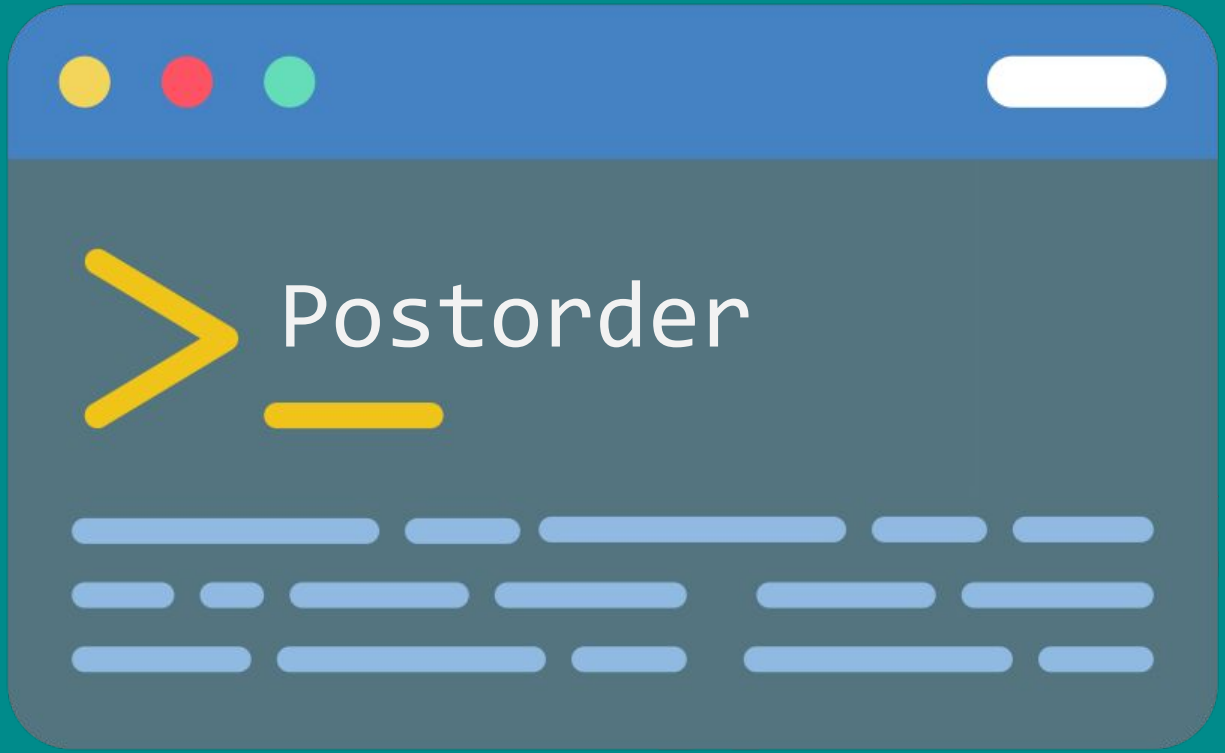

Inorder

Inorder

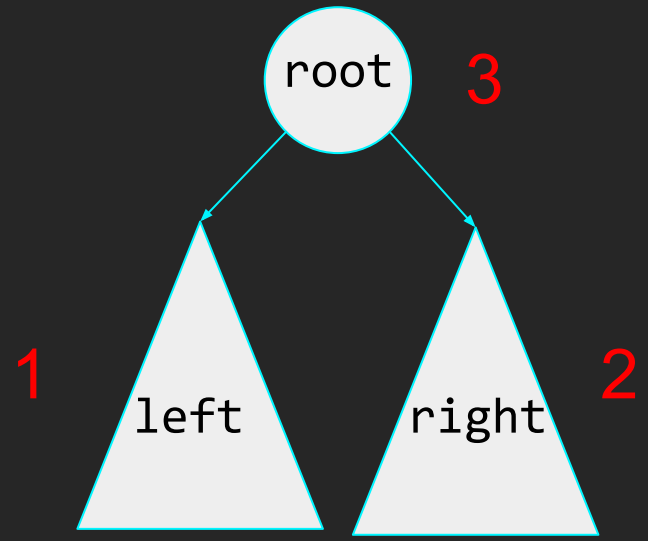



```
void inorder(node *t) {
    if(!t) return;
    inorder(t->l);
    cout << t->val << ' ';
    inorder(t->r);
}
```

先遍歷左子樹，然後輸出根節點的值，最後遍歷右子樹。

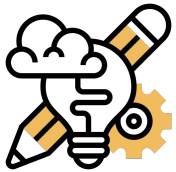

Postorder

Postorder

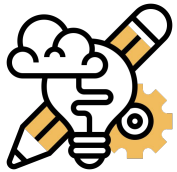



```
void postorder(node *t) {
    if(!t) return;
    postorder(t->l);
    postorder(t->r);
    cout << t->val << ' ';
}
```

先遍歷左子樹，再遍歷右子樹，最後輸出節點的值。



例題 - Last in Preorder



#QUESTION

給你一個有根的二元樹，請你輸出這棵樹前序遍歷最後輸出的值為何？

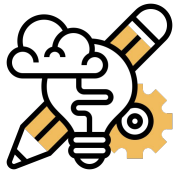
#INPUT FORMAT

第一行有一個正整數 N (1 ≤ N ≤ 10^6)，接下來有 N 行，每行都有兩個整數 l_i , r_i，代表節點 i 的 left child 和 right child，如果沒有 left/right child 的話則為 -1。

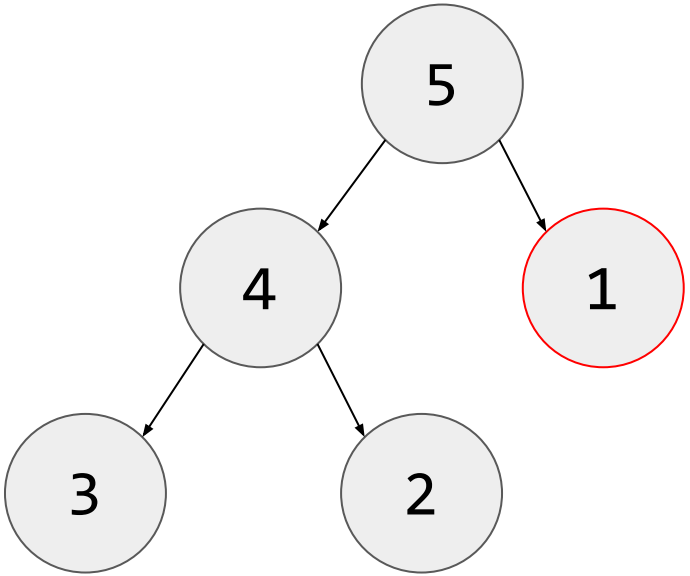
#OUTPUT FORMAT

輸出一個正整數，

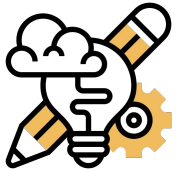
例題 - Last in Preorder



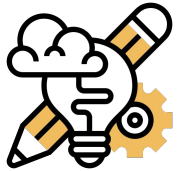
#SAMPLE TESTCASE



OUTPUT : 1



例題 - Last in Preorder



#SOLUTION

直接按照輸入建立一棵 Binary Tree，然後前序遍歷即可。

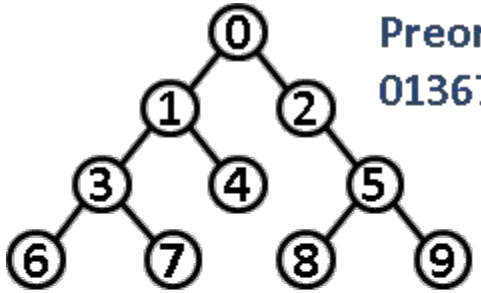
時間複雜度為 O(N)。

https://gist.github.com/LJH-coding/c077c62c07521933275a149a8a976705


Binary Tree
Reconstruction

我們可以只透過一種遍歷方式去構造出原本的二元樹嗎？

不行！只有一種序無法重建二元樹，會有多種結果



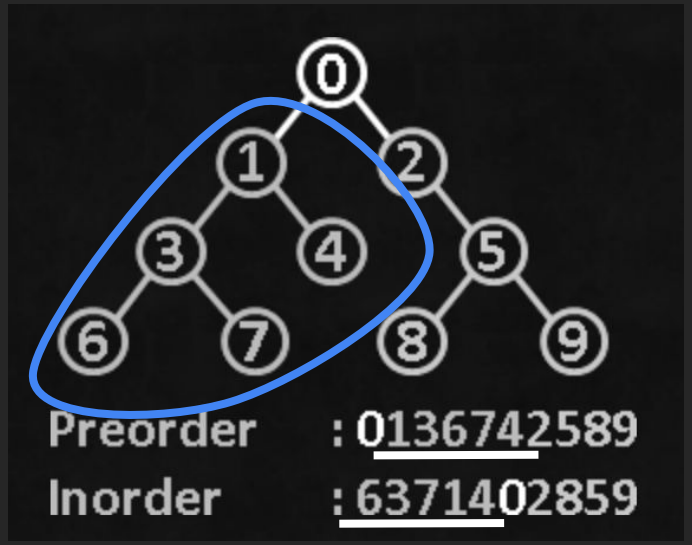
Preorder：
0136742589

但如果我們有前序和中序的遍歷就
可以構造出一棵唯一的樹！

Binary Tree Reconstruction

前序遍歷：根節點 -> 左子樹 -> 右子樹

中序遍歷：左子樹 -> 根節點 -> 右子樹

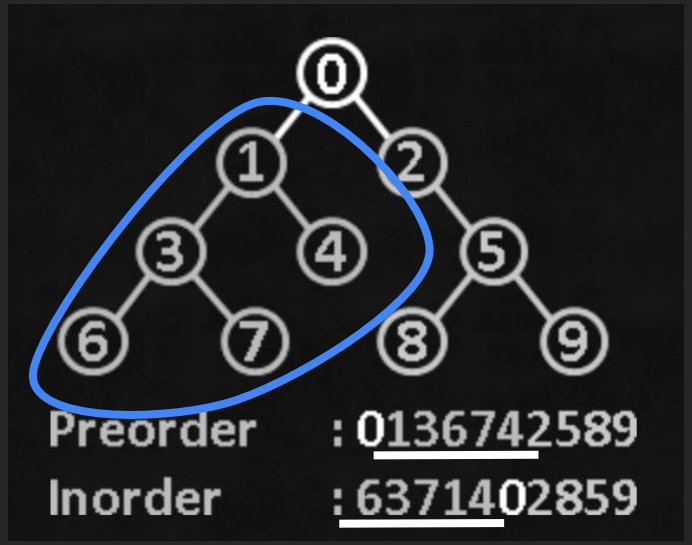


我們可以根據前序遍歷先輸出根節點的特性把中序遍歷的序列拆成兩邊，
左邊就是左子樹的中序遍歷；
右邊就是右子樹的中序遍歷。

Binary Tree Reconstruction

前序遍歷：根節點 -> 左子樹 -> 右子樹

中序遍歷：左子樹 -> 根節點 -> 右子樹



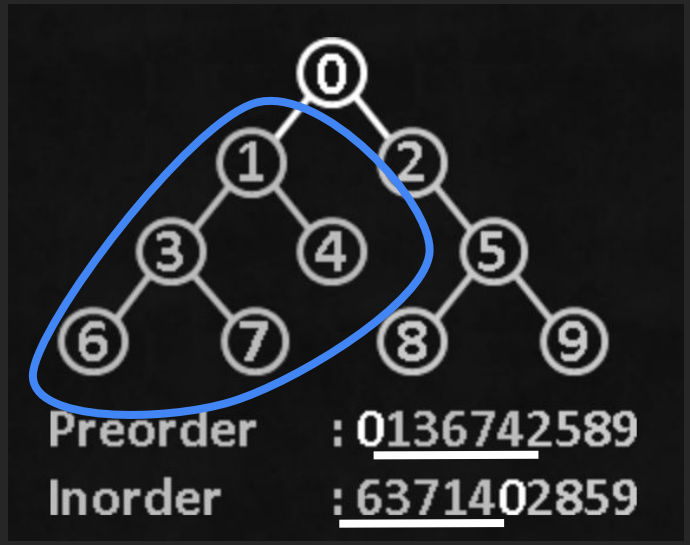
我們可以根據前序遍歷先輸出根節點的特性把中序遍歷的序列拆成兩邊，
左邊就是左子樹的中序遍歷；
右邊就是右子樹的中序遍歷。

這樣就可以把根節點連到左子樹跟右子樹，然後遞迴下去繼續建構子樹即可。

Binary Tree Reconstruction

前序遍歷 : 根節點 -> 左子樹 -> 右子樹

中序遍歷 : 左子樹 -> 根節點 -> 右子樹



遍歷前序 O(N)

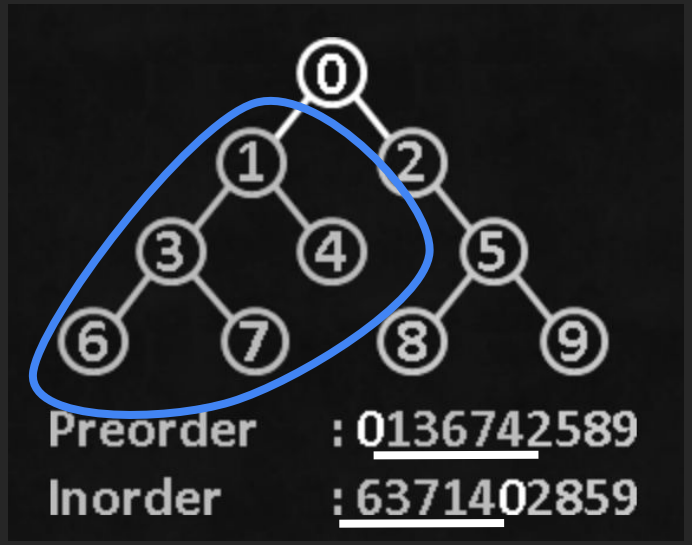
找到根節點在中序的位置 O(N)

= 構建整棵樹 O(N^2)

Binary Tree Reconstruction

前序遍歷：根節點 -> 左子樹 -> 右子樹

中序遍歷：左子樹 -> 根節點 -> 右子樹



遍歷前序 O(N)

找到根節點在中序的位置 O(1)
（可以用陣列 or hash map 儲存每個節點在中序的位置）

= 構建整棵樹 O(N)

build(l, r) 代表現在要根據中序遍歷 [l, r] 構造出原本的樹
假設根節點的位置在 pos ，則:

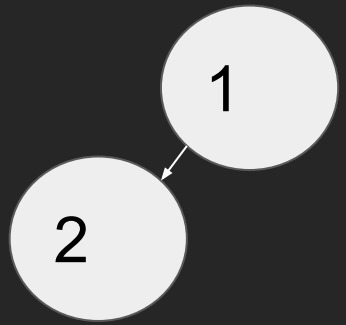
- 如果 l > r :回傳空樹
- 否則:左子樹為 build(l, pos - 1) ，右子樹為 build(pos + 1, r)

ex : preorder: 1 2 4 7 8 5 3 6 9 10
     inorder: 7 4 8 2 5 1 3 9 6 10

build(l, r) 代表現在要根據中序遍歷 [l, r] 構造出原本的樹
假設根節點的位置在 pos ，則:

- 如果 l > r :回傳空樹
- 否則:左子樹為 build(l, pos - 1) ，右子樹為 build(pos + 1, r)

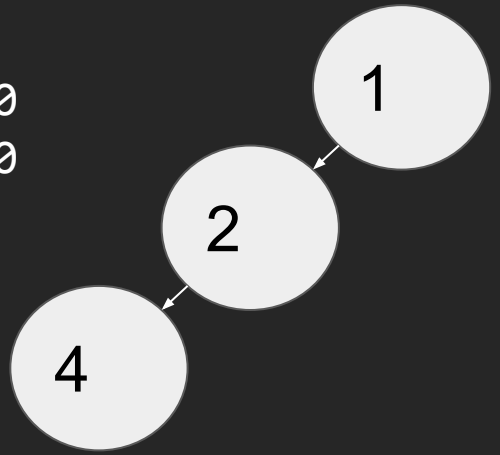
ex : preorder: 1 2 4 7 8 5 3 6 9 10
     inorder: 7 4 8 2 5 1 3 9 6 10



build(l, r) 代表現在要根據中序遍歷 [l, r] 構造出原本的樹
假設根節點的位置在 pos ，則:

- 如果 l > r :回傳空樹
- 否則:左子樹為 build(l, pos - 1) ，右子樹為 build(pos + 1, r)

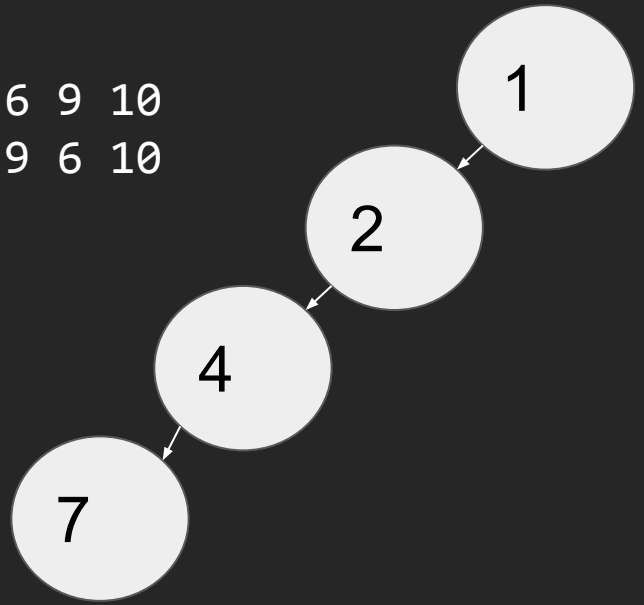
ex : preorder: 1 2 4 7 8 5 3 6 9 10
     inorder: 7 4 8 2 5 1 3 9 6 10



build(l, r) 代表現在要根據中序遍歷 [l, r] 構造出原本的樹
假設根節點的位置在 pos ，則:

- 如果 l > r :回傳空樹
- 否則:左子樹為 build(l, pos - 1) ，右子樹為 build(pos + 1, r)

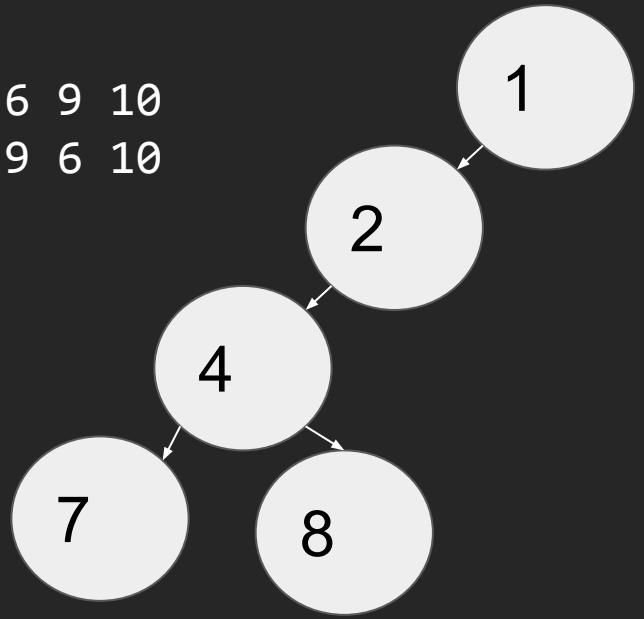
ex : preorder: 1 2 4 7 8 5 3 6 9 10
     inorder: 7 4 8 2 5 1 3 9 6 10



build(l, r) 代表現在要根據中序遍歷 [l, r] 構造出原本的樹
假設根節點的位置在 pos ，則:

- 如果 l > r :回傳空樹
- 否則:左子樹為 build(l, pos - 1) ，右子樹為 build(pos + 1, r)

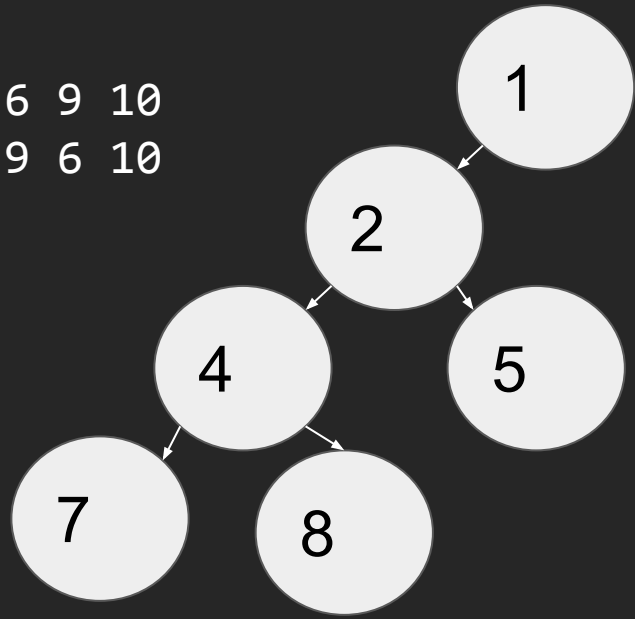
ex : preorder: 1 2 4 7 8 5 3 6 9 10
     inorder: 7 4 8 2 5 1 3 9 6 10



build(l, r) 代表現在要根據中序遍歷 [l, r] 構造出原本的樹
假設根節點的位置在 pos ，則:

- 如果 l > r :回傳空樹
- 否則:左子樹為 build(l, pos - 1) ，右子樹為 build(pos + 1, r)

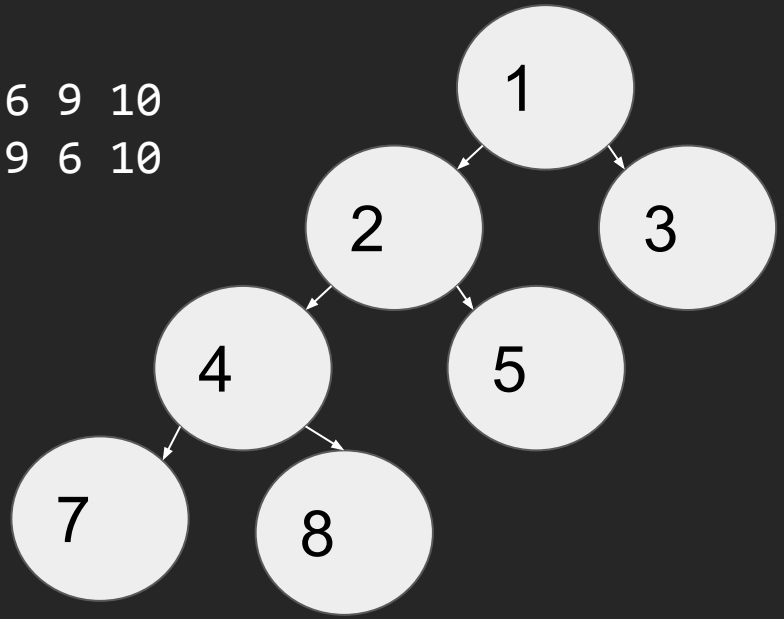
ex : preorder: 1 2 4 7 8 5 3 6 9 10
     inorder: 7 4 8 2 5 1 3 9 6 10



build(l, r) 代表現在要根據中序遍歷 [l, r] 構造出原本的樹
假設根節點的位置在 pos ，則:

- 如果 l > r :回傳空樹
- 否則:左子樹為 build(l, pos - 1) ，右子樹為 build(pos + 1, r)

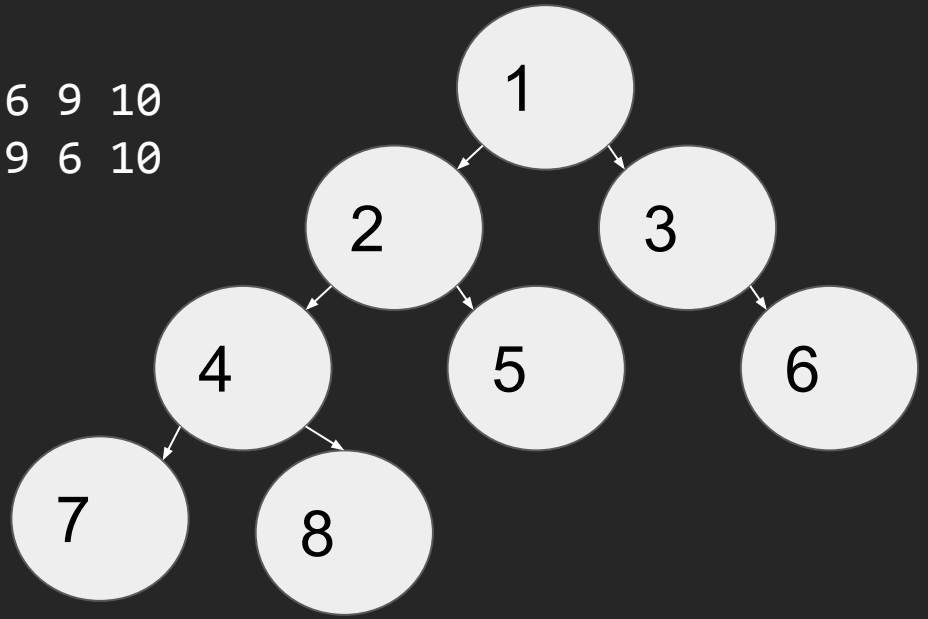
ex : preorder: 1 2 4 7 8 5 3 6 9 10
     inorder: 7 4 8 2 5 1 3 9 6 10



build(l, r) 代表現在要根據中序遍歷 [l, r] 構造出原本的樹
假設根節點的位置在 pos ，則:

- 如果 l > r :回傳空樹
- 否則:左子樹為 build(l, pos - 1) ，右子樹為 build(pos + 1, r)

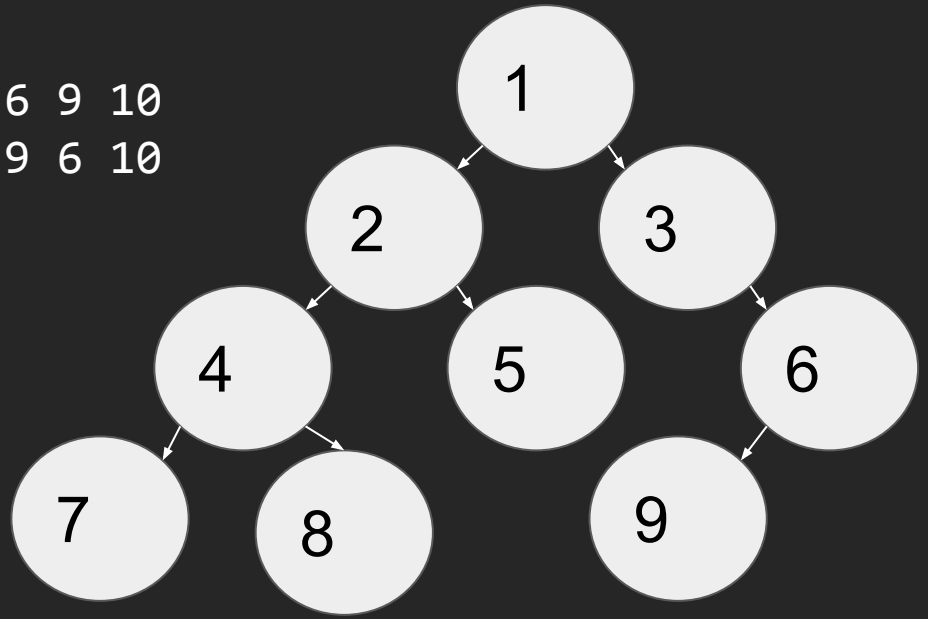
```
ex : preorder: 1 2 4 7 8 5 3 6 9 10
     inorder:  7 4 8 2 5 1 3 9 6 10
```



build(l, r) 代表現在要根據中序遍歷 [l, r] 構造出原本的樹
假設根節點的位置在 pos ，則:

- 如果 l > r :回傳空樹
- 否則:左子樹為 build(l, pos - 1) ，右子樹為 build(pos + 1, r)

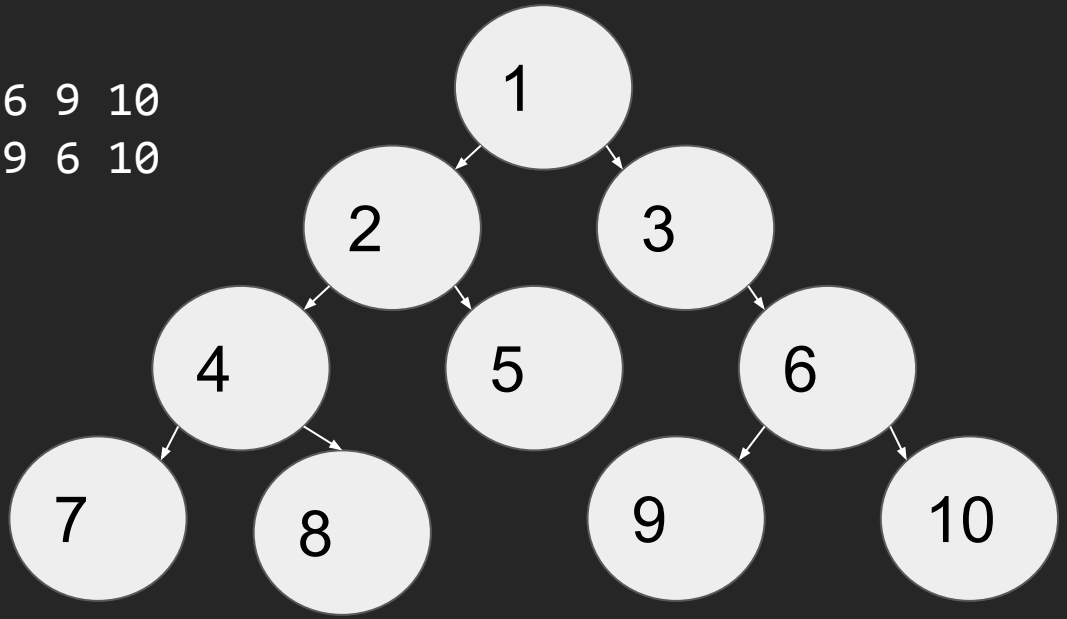
```
ex : preorder: 1 2 4 7 8 5 3 6 9 10
     inorder:  7 4 8 2 5 1 3 9 6 10
```



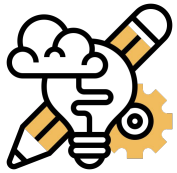
build(l, r) 代表現在要根據中序遍歷 [l, r] 構造出原本的樹
假設根節點的位置在 pos ，則:

- 如果 l > r :回傳空樹
- 否則:左子樹為 build(l, pos - 1) ，右子樹為 build(pos + 1, r)

```
ex : preorder: 1 2 4 7 8 5 3 6 9 10
     inorder:  7 4 8 2 5 1 3 9 6 10
```



例題 - Tree Reconstruct



#QUESTION

給你 N 個節點二元樹的前序和中序遍歷，請你求出他的後序遍歷。

#INPUT FORMAT

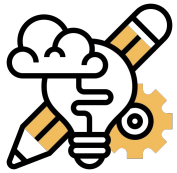
第一行有一正整數 N (1 ≤ N ≤ 200000)，代表二元樹的節點數

接下來有兩行，每一行有 N 個數字，兩行分別代表這顆二元樹的前序遍歷與中序遍歷

#OUTPUT FORMAT

請輸出一行，代表這顆二元樹的後序遍歷

例題 - Tree Reconstruct



#SOLUTION

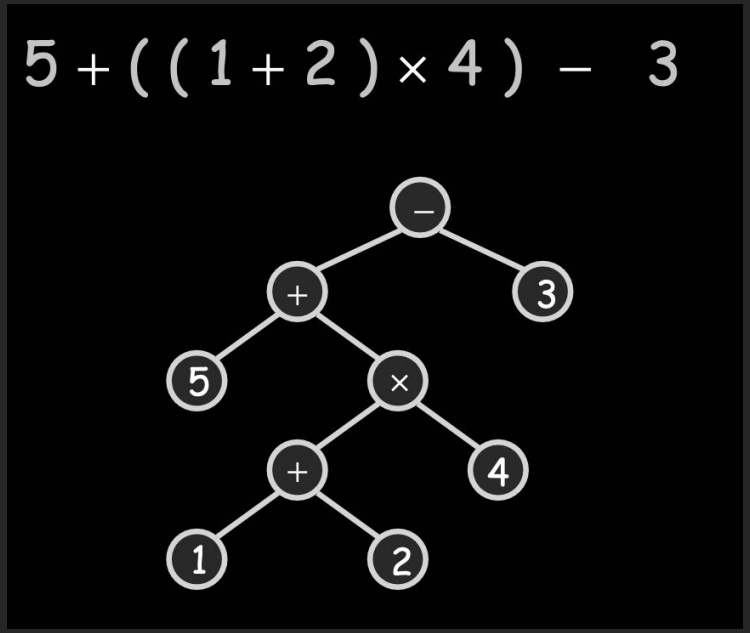
根據剛剛所學到的構造方式，把二元樹構造出來再後序遍歷即可。

時間複雜度為 O(N)

https://gist.github.com/LJH-coding/3954f6833b319ca8b50ef3da184a51de

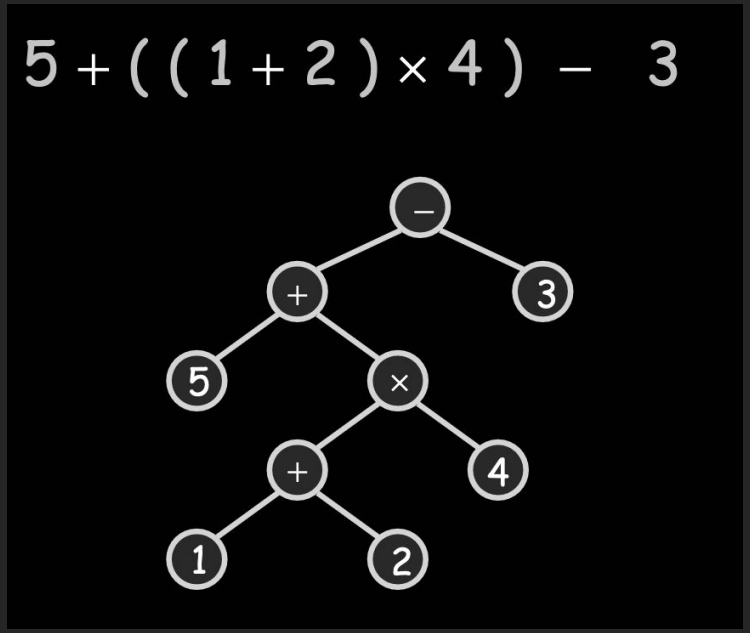
Expression Tree

Expression Tree

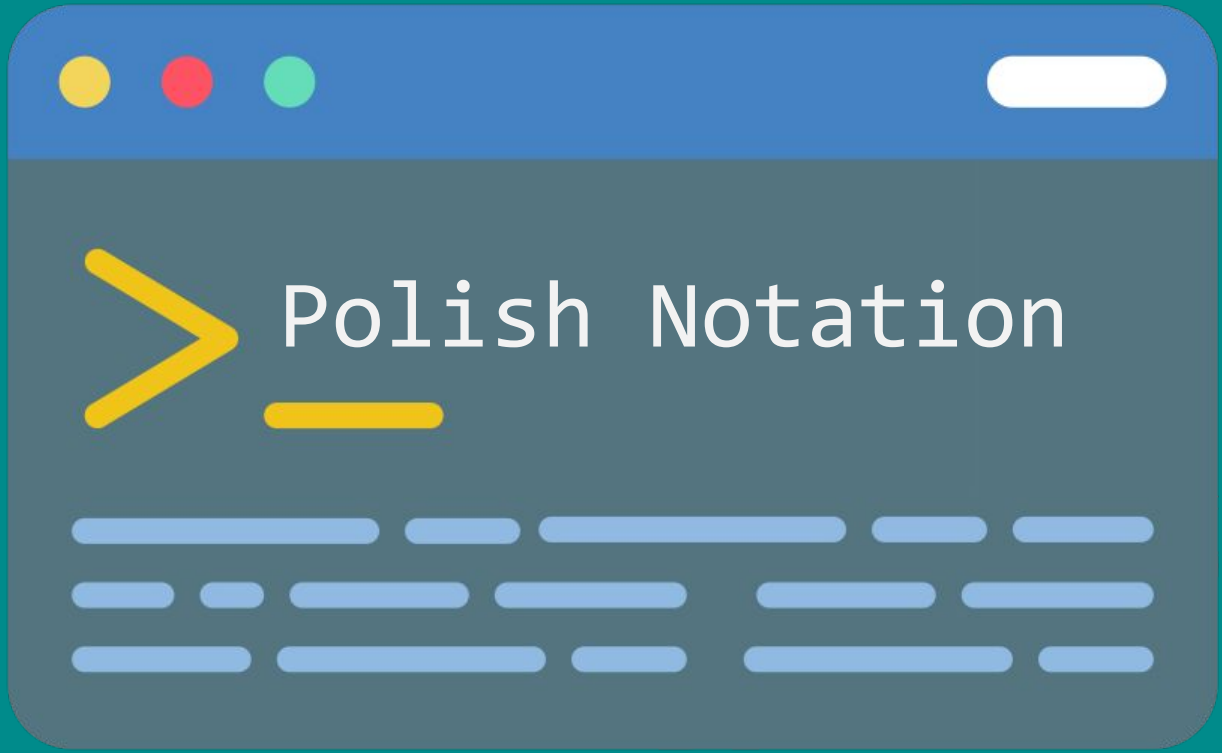


對於每個數學運算式，都有他對應的表達樹(Expression Tree)，而這棵表達樹的中序遍歷就是原本的數學運算式。

Expression Tree



透過表達式去建構一棵表達樹的實用度並不是很高，所以接下來會著重在表達樹遍歷的應用以及他們之間的相互轉換


Polish Notation

波蘭表示法(Polish Notation)

- 表達樹的前序遍歷(Prefix Notation)
- 運算子在數字前面
- 不需要括號

Expression : 5 + ((1 + 2) * 4) - 3

Polish Notation : - + 5 * + 1 2 4 3

Polish Notation (Prefix Notation)

給定一個 Prefix Notation，要怎麼算出表達式的結果呢？

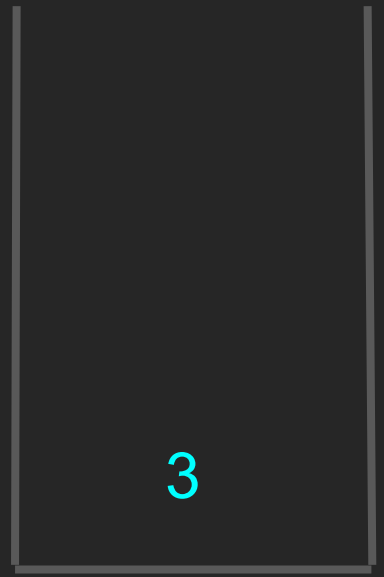
我們可以用 stack 來計算！

從右開始往左遍歷 prefix notation，如果遇到數字就把他丟進 stack 裡，如果遇到運算子就把 stack 最上面的兩個數字取出來進行運算，然後把運算結果丟回 stack 裡面。
最後 stack 剩下的數字就是表達式的運算結果

ex : prefix notation - + 5 * + 1 2 4 3

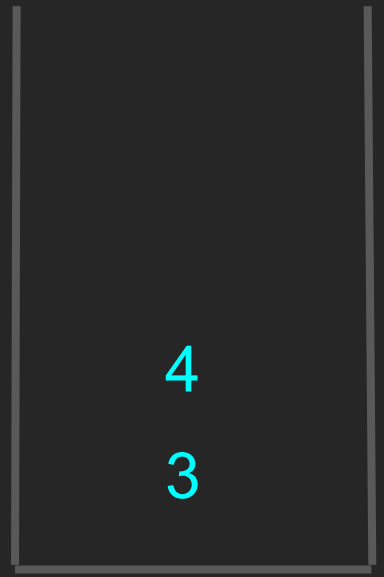
從右開始往左遍歷 prefix notation，如果遇到數字就把他丟進 stack 裡，如果遇到運算子就把 stack 最上面的兩個數字取出來進行運算，然後把運算結果丟回 stack 裡面。
最後 stack 剩下的數字就是表達式的運算結果

ex : prefix notation - + 5 * + 1 2 4 3



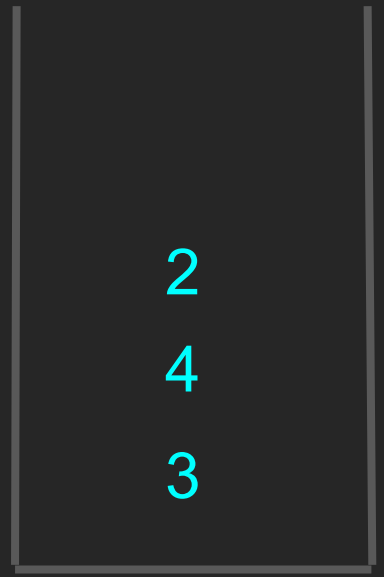
從右開始往左遍歷 prefix notation，如果遇到數字就把他丟進 stack 裡，如果遇到運算子就把 stack 最上面的兩個數字取出來進行運算，然後把運算結果丟回 stack 裡面。
最後 stack 剩下的數字就是表達式的運算結果

ex : prefix notation - + 5 * + 1 2 4 3



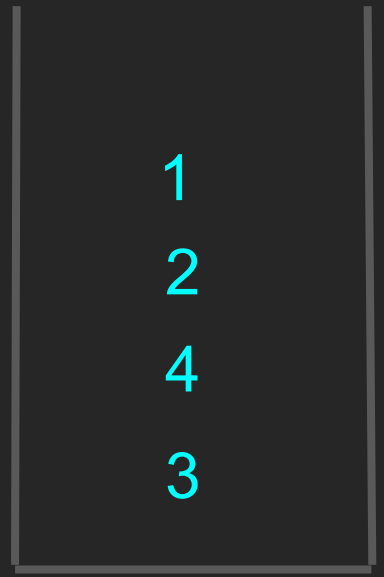
從右開始往左遍歷 prefix notation，如果遇到數字就把他丟進 stack 裡，如果遇到運算子就把 stack 最上面的兩個數字取出來進行運算，然後把運算結果丟回 stack 裡面。
最後 stack 剩下的數字就是表達式的運算結果

ex : prefix notation - + 5 * + 1 2 4 3



從右開始往左遍歷 prefix notation，如果遇到數字就把他丟進 stack 裡，如果遇到運算子就把 stack 最上面的兩個數字取出來進行運算，然後把運算結果丟回 stack 裡面。
最後 stack 剩下的數字就是表達式的運算結果

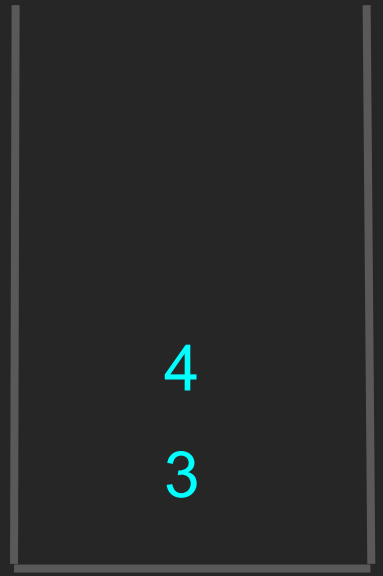
ex : prefix notation - + 5 * + 1 2 4 3



從右開始往左遍歷 prefix notation，如果遇到數字就把他丟進 stack 裡，如果遇到運算子就把 stack 最上面的兩個數字取出來進行運算，然後把運算結果丟回 stack 裡面。
最後 stack 剩下的數字就是表達式的運算結果

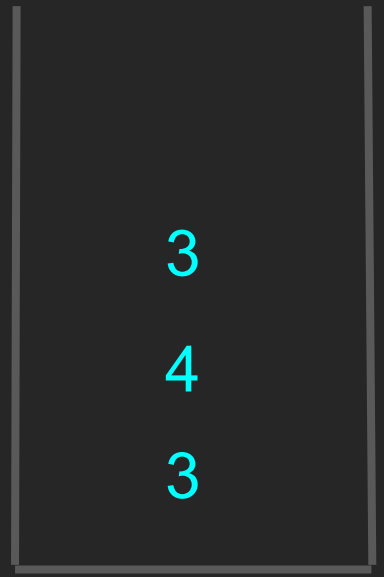
ex : prefix notation - + 5 * + 1 2 4 3

1 + 2 = 3



從右開始往左遍歷 prefix notation，如果遇到數字就把他丟進 stack 裡，如果遇到運算子就把 stack 最上面的兩個數字取出來進行運算，然後把運算結果丟回 stack 裡面。
最後 stack 剩下的數字就是表達式的運算結果

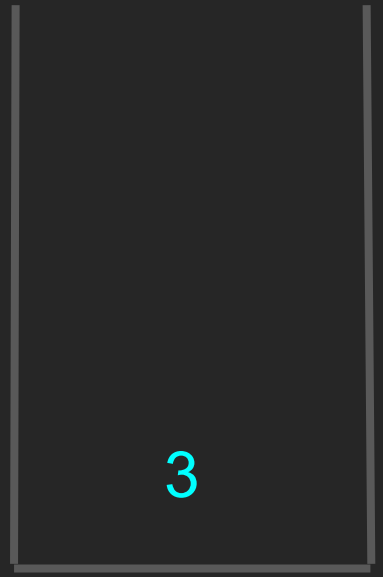
ex : prefix notation - + 5 * + 1 2 4 3



從右開始往左遍歷 prefix notation，如果遇到數字就把他丟進 stack 裡，如果遇到運算子就把 stack 最上面的兩個數字取出來進行運算，然後把運算結果丟回 stack 裡面。
最後 stack 剩下的數字就是表達式的運算結果

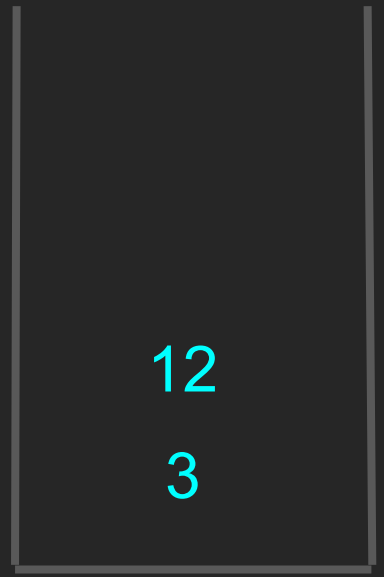
ex : prefix notation - + 5 * + 1 2 4 3

3 * 4 = 12



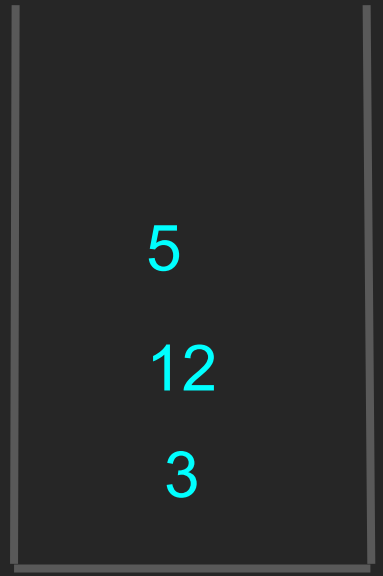
從右開始往左遍歷 prefix notation，如果遇到數字就把他丟進 stack 裡，如果遇到運算子就把 stack 最上面的兩個數字取出來進行運算，然後把運算結果丟回 stack 裡面。
最後 stack 剩下的數字就是表達式的運算結果

ex : prefix notation - + 5 * + 1 2 4 3



從右開始往左遍歷 prefix notation，如果遇到數字就把他丟進 stack 裡，如果遇到運算子就把 stack 最上面的兩個數字取出來進行運算，然後把運算結果丟回 stack 裡面。
最後 stack 剩下的數字就是表達式的運算結果

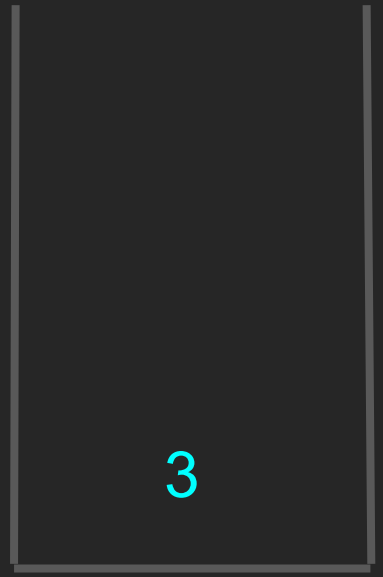
ex : prefix notation - + 5 * + 1 2 4 3



從右開始往左遍歷 prefix notation，如果遇到數字就把他丟進 stack 裡，如果遇到運算子就把 stack 最上面的兩個數字取出來進行運算，然後把運算結果丟回 stack 裡面。
最後 stack 剩下的數字就是表達式的運算結果

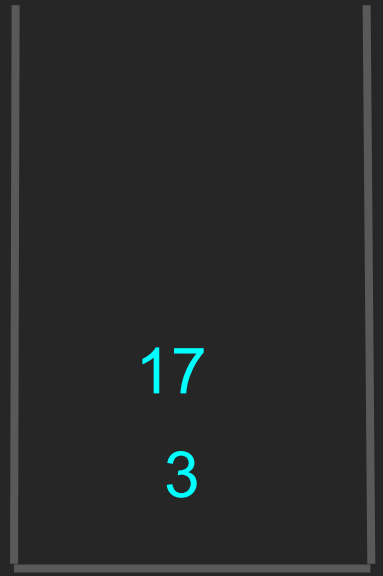
ex : prefix notation - + 5 * + 1 2 4 3

5 + 12 = 17



從右開始往左遍歷 prefix notation，如果遇到數字就把他丟進 stack 裡，如果遇到運算子就把 stack 最上面的兩個數字取出來進行運算，然後把運算結果丟回 stack 裡面。
最後 stack 剩下的數字就是表達式的運算結果

ex : prefix notation - + 5 * + 1 2 4 3



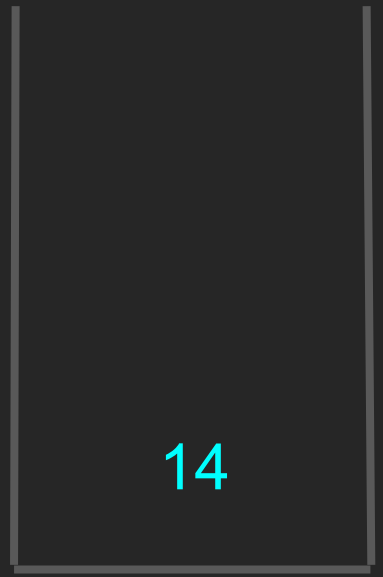
從右開始往左遍歷 prefix notation，如果遇到數字就把他丟進 stack 裡，如果遇到運算子就把 stack 最上面的兩個數字取出來進行運算，然後把運算結果丟回 stack 裡面。
最後 stack 剩下的數字就是表達式的運算結果

ex : prefix notation - + 5 * + 1 2 4 3

17 - 3 = 14

從右開始往左遍歷 prefix notation，如果遇到數字就把他丟進 stack 裡，如果遇到運算子就把 stack 最上面的兩個數字取出來進行運算，然後把運算結果丟回 stack 裡面。
最後 stack 剩下的數字就是表達式的運算結果

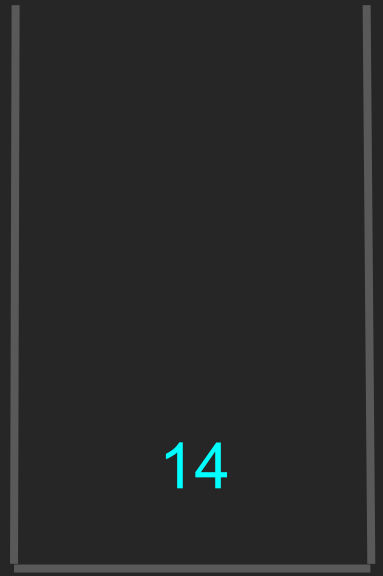
ex : prefix notation - + 5 * + 1 2 4 3



從右開始往左遍歷 prefix notation，如果遇到數字就把他丟進 stack 裡，如果遇到運算子就把 stack 最上面的兩個數字取出來進行運算，然後把運算結果丟回 stack 裡面。
最後 stack 剩下的數字就是表達式的運算結果

ex : prefix notation - + 5 * + 1 2 4 3

運算結果為 14！



逆波蘭表示法(Reverse Polish Notation)

- 表達樹的後序遍歷(Postfix Notation)
- 運算子在數字後面
- 不需要括號

Expression : 5 + ((1 + 2) * 4) - 3

Reverse Polish Notation : 5 1 2 + 4 * + 3 -

Reverse Polish Notation (Postfix Notation)

給定一個 Suffix Notation，要怎麼算出表達式的結果呢？

跟計算 prefix notation 的方式相同,
只是順序變成從左往右。


Shunting Yard
Algorithm

Shunting Yard Algorithm

是一種用於把表達樹的中序表達式轉換成後序表達式的經典算法

而數學表達式，只需要透過這個演算法，轉換成逆波蘭表示法(後序表示法)，即可輕鬆地計算出解答。

- 從左到右遍歷中序表達式

Shunting Yard Algorithm

- 從左到右遍歷中序表達式
  - 如果讀到數字就直接輸出

Shunting Yard Algorithm

- 從左到右遍歷中序表達式
  - 如果讀到數字就直接輸出
  - 如果讀到運算符號，則不斷地檢查存放運算子的 stack 頂端元素的優先度是否 >= 該運算符號，如果是的話就輸出。
    檢查完成之後把這個運算符號放入 stack 中

Shunting Yard Algorithm

- 從左到右遍歷中序表達式
  - 如果讀到數字就直接輸出
  - 如果讀到運算符號，則不斷地檢查存放運算子的 stack 頂端元素的優先度是否 >= 該運算符號，如果是的話就輸出。
    檢查完成之後把這個運算符號放入 stack 中
  - 如果讀到 (就直接放入 stack 中

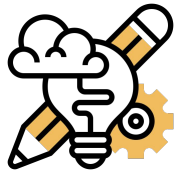
Shunting Yard Algorithm

- 從左到右遍歷中序表達式
  - 如果讀到數字就直接輸出
  - 如果讀到運算符號，則不斷地檢查存放運算子的 stack 頂端元素的優先度是否 >= 該運算符號，如果是的話就輸出。
    檢查完成之後把這個運算符號放入 stack 中
  - 如果讀到 (就直接放入 stack 中
  - 如果讀到) 就不斷地把 stack 內的運算子輸出，直到 stack 的頂端為 (為止

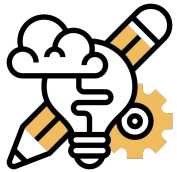
Shunting Yard Algorithm

- 從左到右遍歷中序表達式
  - 如果讀到數字就直接輸出
  - 如果讀到運算符號，則不斷地檢查存放運算子的 stack 頂端元素的優先度是否 >= 該運算符號，如果是的話就輸出。
    檢查完成之後把這個運算符號放入 stack 中
  - 如果讀到 (就直接放入 stack 中
  - 如果讀到) 就不斷地把 stack 內的運算子輸出，直到 stack 的頂端為 (為止
- 依序輸出 stack 內剩餘的運算子

Shunting Yard Algorithm



例題 - Calculator



#QUESTION

給你一個算數表達式，這個表達式內最多會有 5 個變數，變數名稱為 A ~ E，接下來有 T 筆輸入，每次輸入 5 個變數分別對應的值，對於每筆輸入請你輸出他算術表達式計算出來的解為何。

#INPUT FORMAT

第一行有一字串 expr，為題目的算數表達式。(1 <= |expr| <= 2 * 10^5)

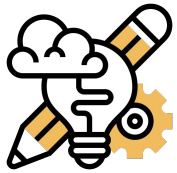
第二行有一正整數 T，

接下來 T 行每行都有五個整數 A, B, C, D, E，代表表達式中變數對應的值。

#OUTPUT FORMAT

輸出一共 T 行，每一行輸出該表達式計算出來的值。

例題 - Calculator



#SOLUTION

直接把輸入的表達式透過 Shunting Yard Algorithm 轉換成後序表達式，然後用 stack 計算出解答即可。

https://gist.github.com/LJH-coding/9b6970b69ef079a17dd397a81dce3681