## Parenthesis Checker

Given an expression string **exp**. Examine whether the pairs and the orders of “{“,”}”,”(“,”)”,”[“,”]” are correct in exp.
For example, the program should print 'balanced' for exp = “[()]{}{[()()]()}” and 'not balanced' for exp = “[(])”

**Input:**
The first line of input contains an integer T denoting the number of test cases. Each test case consists of a string of expression, in a separate line.

**Output:**
Print 'balanced' without quotes if the pair of parenthesis is balanced else print 'not balanced' in a separate line.

**Constraints:**
$1 \le T \le 100$
$1 \le |s| \le 10^5$

**Example:**
**Input:**
3
{([])}
()
([]

**Output:**
balanced
balanced
not balanced

---

## Next larger element

Given an array **A** of size **N** having distinct elements, the task is to find the next greater element for each element of the array in order of their appearance in the array. If no such element exists, output **-1**

**Input:**
The first line of input contains a single integer **T** denoting the number of test

cases.Then **T** test cases follow. Each test case consists of two lines. The first line contains an integer **N** denoting the size of the array. The Second line of each test case contains **N** space separated positive integers denoting the values/elements in the array **A**.

**Output:**
For each test case, print in a new line, the next greater element for each array element separated by space in order.

**Constraints:**
1 <= T <= 100

1 <= N <= $10^7$

1 <= $A_i$ <= $10^{18}$

**Example:**
**Input**
2
4
1 3 2 4
4
4 3 2 1
**Output**
3 4 4 -1
-1 -1 -1 -1

**Explanation**:
**Testcase1:** In the array, the next larger element to 1 is 3 , 3 is 4 , 2 is 4 and for 4 ? since it doesn't exist hence -1.

---

## Queue using two Stacks

Implement a Queue using 2 stacks **s1** and **s2** .
A Query **Q** is of 2 Types
**(i)** 1 x (a query of this type means  pushing **'x'** into the queue)
**(ii)** 2   (a query of this type means to pop element from queue and print the poped element)

**Example 1:**

```
Input:
Q = 5
```

```
Queries = 1 2 1 3 2 1 4 2
Output: 2 3
Explanation: In the first testcase
1 2 the queue will be {2}
1 3 the queue will be {2 3}
2   poped element will be 2 the queue
   will be {3}
1 4 the queue will be {3 4}
2   poped element will be 3.
```

**Example 2:**

```
Input:
Q = 4
Queries = 1 2 2 2 1 4
Output: 2 -1
Explanation: In the second testcase
1 2 the queue will be {2}
2   poped element will be 2 and
   then the queue will be empty
2   the queue is empty and hence -1
1 4 the queue will be {4}.
```

**Your Task:**
You are required to complete the two methods **push** which take one argument an integer **'x'** to be pushed into the queue and **pop** which returns a integer poped out from other queue(-1 if the queue is empty). The **printing** is done **automatically** by the **driver code**.

**Expected Time Complexity** : O(1) for both **push()** and O(N) for **pop().**
**Expected Auxilliary Space** : O(N).

**Constraints:**
1 <= Q <= 100
1 <= x <= 100

Note:The **Input/Ouput** format and **Example** given are used for system's internal purpose, and should be used by a user for **Expected Output** only. As it is a function problem, hence a user should not read any input from stdin/console. The task is to complete the function specified, and not to write the full code.

---

## Stack using two queues

Implement a Stack using two queues **q1** and **q2**.

**Example 1:**

```
Input:
push(2)
push(3)
pop()
push(4)
pop()
Output: 3 4
Explanation:
push(2) the stack will be {2}
push(3) the stack will be {2 3}
pop()   poped element will be 3 the
      stack will be {2}
push(4) the stack will be {2 4}
pop()   poped element will be 4
```

**Example 2:**

```
Input:
push(2)
pop()
pop()
push(3)
Output: 2 -1
```

### Your Task:

Since this is a function problem, you don't need to take inputs. You are required to complete the two methods **push()** which takes an integer **'x'** as input denoting the element to be pushed into the stack and **pop()** which returns the integer poped out from the stack(**-1** if the stack is empty).

**Expected Time Complexity:** O(1) for **push()** and O(N) for **pop()** (or vice-versa).
**Expected Auxiliary Space:** O(1) for both **push()** and **pop()**.

**Constraints:**
1 <= Number of queries <= 100
1 <= values of the stack <= 100

---

## Get minimum element from stack

You are given **N** elements and your task is to Implement a Stack in which you can get minimum element in O(1) time.

**Example 1:**

```
Input:
push(2)
push(3)
pop()
getMin()
push(1)
getMin()
Output: 3 2 1
Explanation: In the first test case for
query
push(2)  the stack will be {2}
push(3)  the stack will be {2 3}
pop()    poped element will be 3 the
       stack will be {2}
getMin() min element will be 2
push(1)  the stack will be {2 1}
getMin() min element will be 1
```

**Your Task:**
You are required to complete the three methods **push()** which take one argument an integer **'x'** to be pushed into the stack, **pop()** which returns a integer poped out from the stack and **getMin()** which returns the min element from the stack. (**-**1 will be returned if for **pop() and getMin()** the stack is empty.)

**Expected Time Complexity** : O(1) for all the 3 methods.
**Expected Auixilliary Space** : O(1) for all the 3 methods.

**Constraints:**
1 <= Number of queries <= 100
1 <= values of the stack <= 100