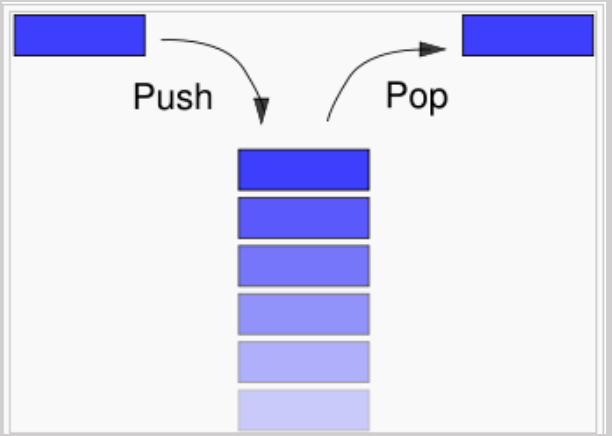
Stack

A **stack** is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

Stack is a FILO (***first-in last-out***) data structure.

The element which is placed (inserted) first, is accessed last. In stack terminology

- insertion operation is called **PUSH** operation;
- removal operation is called **POP** operation.



Time Complexity

To access or edit any element stored in a stack, the time taken is **O(N)** as to reach any specific element, all the elements before it has to be removed.

The searching operation also takes a total time of **O(N)**, as reaching any specific element isn't possible without popping the elements stored before it.

Operations like insertion or deletion in a stack take constant time i.e. **O(1)**.

Applications

- Stack is used for evaluating expression with operands and operations
- Matching tags in HTML and XML
- Undo function in any text editor
- Stacks are used for parenthesis matching
- Stacks are useful for function calls, storing the activation records and deleting them after returning from the function. It is very useful in processing the function calls.
- Stacks help in reversing any set of data or strings.

Real-life Applications

- CD/DVD stand
- Stack of books in a book shop
- Undo and Redo mechanisms
- The history of a web browser
- Call logs, E-mails, and Google photos in any gallery are also stored in form of a stack.
- YouTube downloads and Notifications are also shown in LIFO format(the latest appears first).

Advantages

Stack helps in managing data that follows the LIFO technique.

When a function is called, the local variables and other function parameters are stored in the stack and automatically destroyed once returned from the function. Hence, efficient function management.

Stacks are more secure and reliable as they do not get corrupted easily.

Stack cleans up the objects automatically.

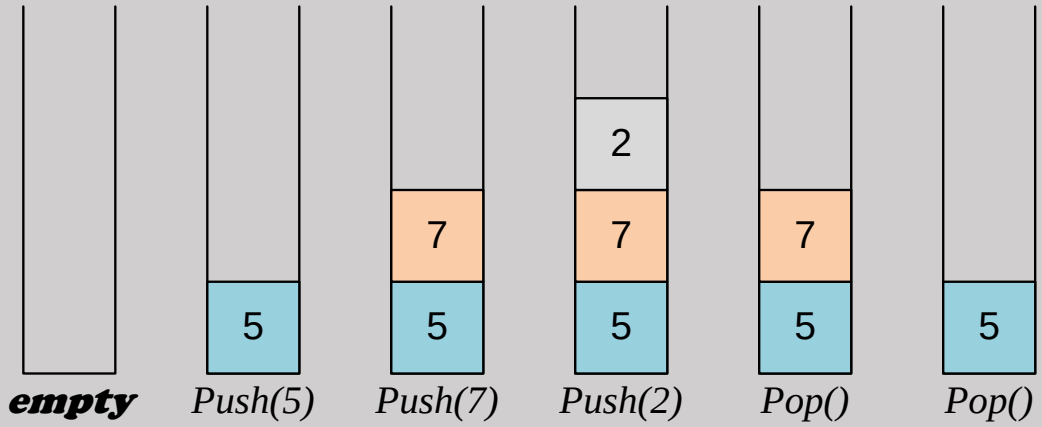
Disadvantages

- Stack memory is of limited size.
- The total size of the stack must be defined before
- Too many objects can lead to stack overflow
- Random Accessing is not possible

Stack

Stack is a container shall support the following operations:

- ***empty*** – test whether container is empty;
- ***size*** – return size;
- ***top*** – access top element
- ***push*** – insert element to the back;
- ***pop*** – remove front element;



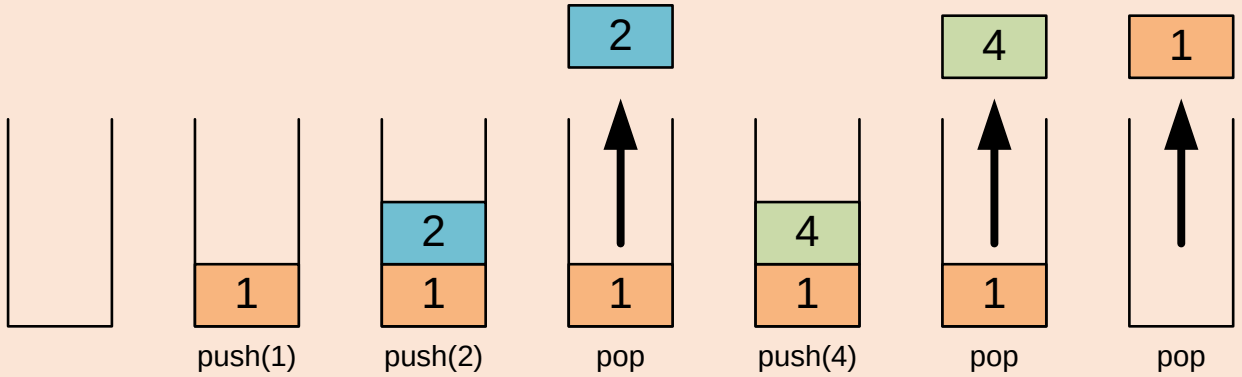

```
stack<int> s; // Create an empty stack s.
// Push to the stack the squares of numbers
for (int i = 1; i <= 10; i++) s.push(i*i);

// Print the top element and the size of the stack
printf("Top element is %d\n", t.top());
printf("Stack size is %d\n", t.size());
```

E-OLYMP 5087. Implement a stack

Simulate **push** and **pop** operations.

Input. The first line contains the number of operations n ($1 \le n \le 10^5$). In the next n lines the first number is the operation number, the second number (only for the "first" operation) is a number to add, it is positive integer, not greater than 10^5.

Output. Print all removed numbers one by one, each on a separate line.



Sample input

```
6
1 1
1 2
2
1 4
2
2
```

Sample output

```
2
4
1
```

E-OLYMP 6122. Simple stack

Design and implement the data structure “stack”. Write the program to simulate the stack operations, implement the next commands:

- **push** *n* - Add to the stack the number *n* (value *n* is given after the command). Print **ok**.
- **pop** - Remove the last element from the stack. Print the value of this element.
- **back** - Print the value of the last element, not removing it from the stack.
- **size** - Print the number of elements in the stack.
- **clear** - Clear the stack and print **ok**.
- **exit** - Print bye and terminate.

E-OLYMP 5327. Bracket sequence

The bracket sequence is a correct arithmetic expression, from which all numbers and signs are removed. For example,

1 + (((2 + 3) + 5) + (3 + 4)) → ((( )) ( ))

Print “YES” if the bracket sequence is correct and “NO” otherwise.

Input. Given a sequence of opening and closing brackets of length not more than 4000000.

Output. Print “YES” if the bracket sequence is correct and “NO” otherwise.

Sample input 1

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

Sample output 1

```
YES
```

Sample input 2

```
(()
```

Sample output 2

```
NO
```

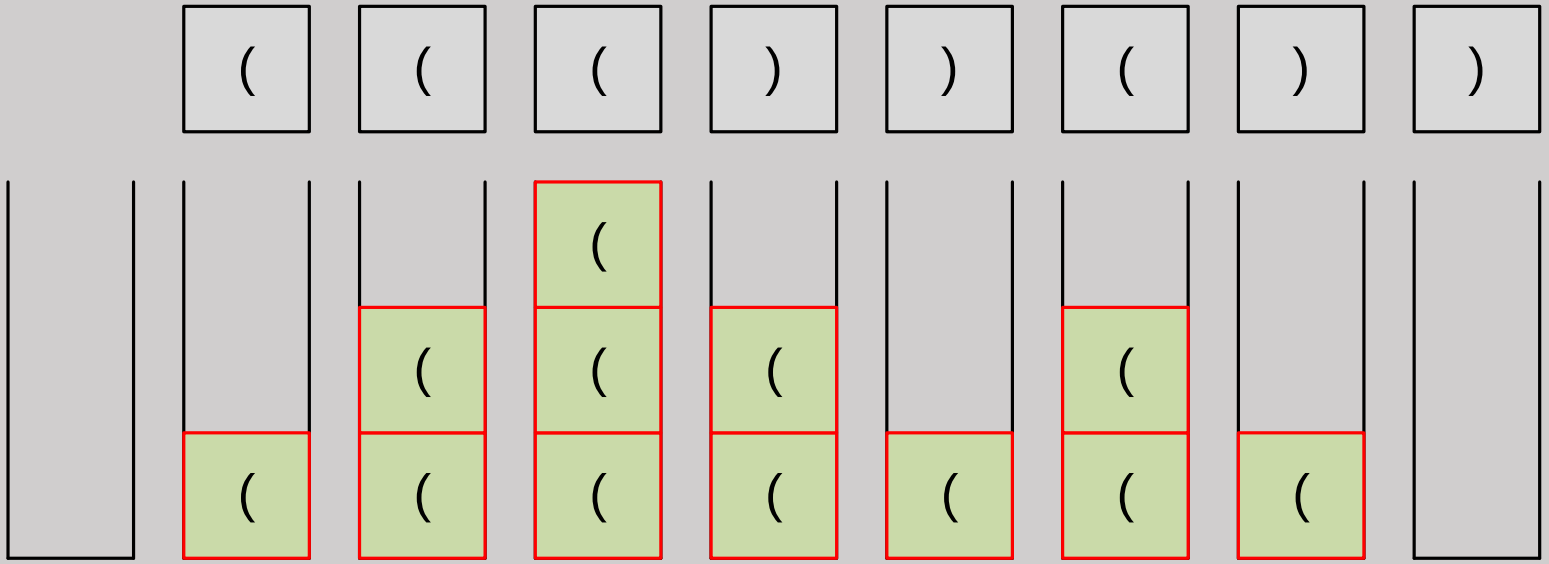
How to solve a problem?

E-OLYMP 5327. Bracket sequence

Declare a stack where we'll save the opening brackets only. When the next symbol arrives, we do the following operation with the stack:

- if the symbol '(' is encountered, then **push** it onto the stack;
- if the symbol ')' is encountered, then **pop** the top element from the stack. If the stack is empty, then the sequence is not parenthetical (at some stage, the number of closing parentheses is greater than the number of opening ones);

At the end of processing the input line, the stack must be empty.



Is it possible not to use memory for stack?

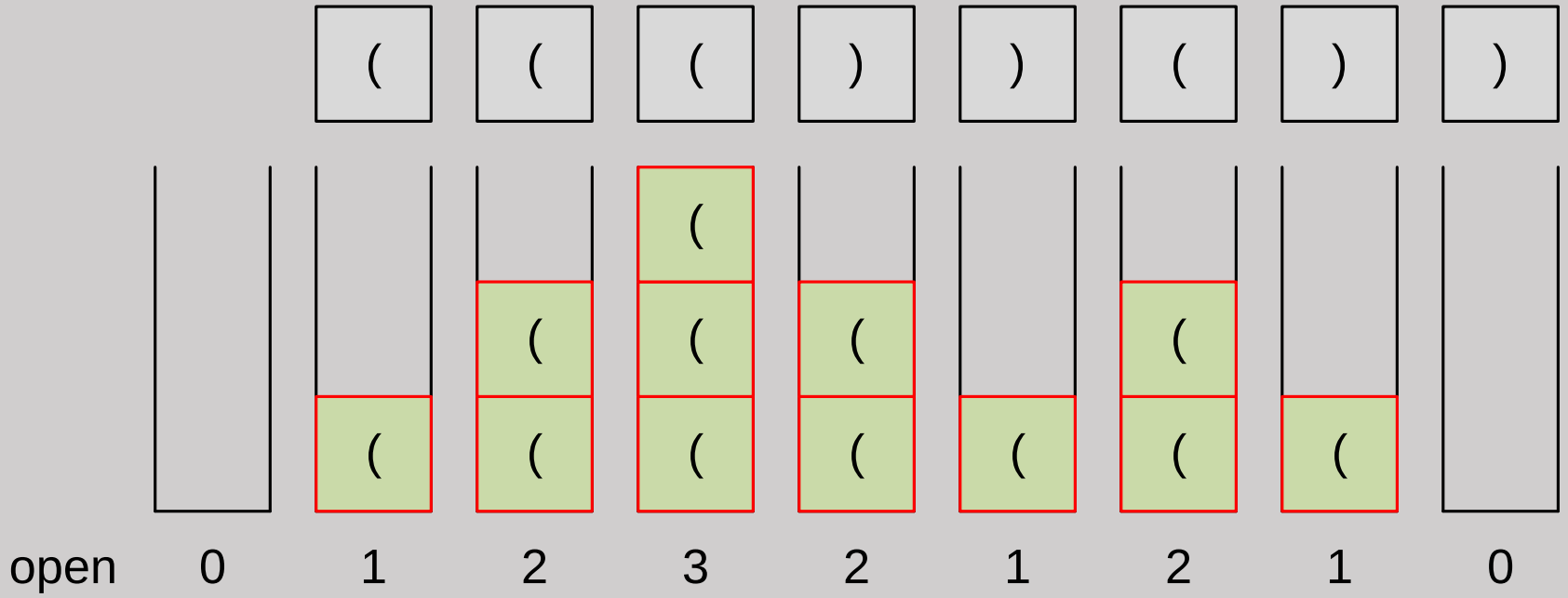
What if the input string is big enough?

E-OLYMP 5327. Bracket sequence

You can simulate the stack using one variable.

Let the variable *open* stores the number of open parentheses in the stack.

- When the ‘(‘ symbol is pushed onto the stack, the *open*++ operation is performed.
- When an element is removed from the top of the stack, operation *open*-- is performed.



2479. Parentheses balance

You are given a string consisting of parentheses ( ) and [ ]. A string of this type is said to be correct:

- if it is the empty string
- if A and B are correct, AB is correct,
- if A is correct, (A) and [A] is correct.

Write a program that takes a sequence of strings of this type and check their correctness.

Input. The first line contains the number of test cases *n*. Each of the next *n* lines contains the string of parentheses ( ) and [ ].

Output. For each test case print in a separate line “Yes” if the expression is correct or “No” otherwise.

Sample input	**Sample output**
3 ([]) (([()])) ([()[]()])()	Yes No Yes

2479. Parentheses balance

You are given a string consisting of parentheses ( ) and [ ]. A string of this type is said to be correct:

- if it is the empty string
- if A and B are correct, AB is correct,
- if A is correct, (A) and [A] is correct.

Write a program that takes a sequence of strings of this type and check their correctness.

Be careful! Input line can be empty like

Sample input

```
3
([])

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

Sample output

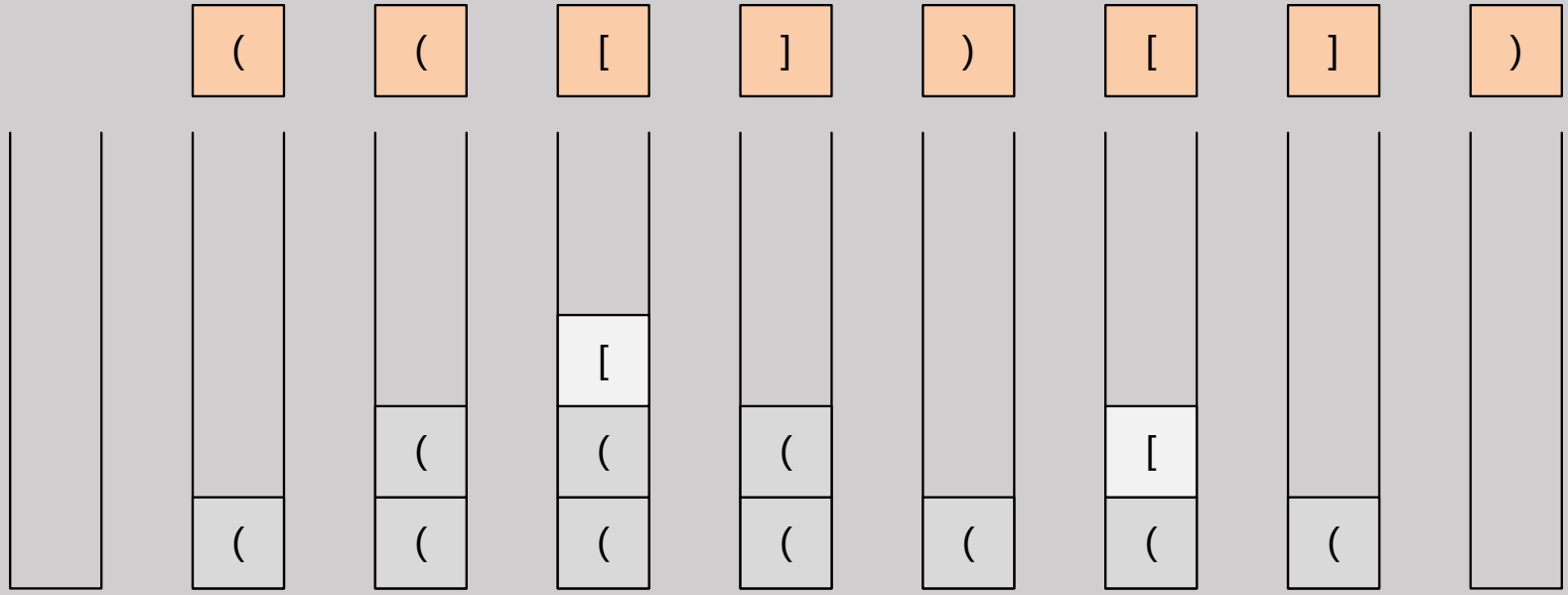
```
Yes
Yes
Yes
```

2479. Parentheses balance

Process sequentially the symbols of the input string and:

- if the current character is an opening parenthesis (round or square), push it into the stack.
- if the current character is a closing bracket, then the corresponding opening parenthesis must be at the top of the stack. If this is not the case, or if the stack is empty, then the expression is not correct.

At the end of processing the correct line, stack should be empty.



E-OLYMP 5060. Reverse Polish notation

Reverse Polish notation (RPN) is a mathematical notation in which every operator follows all of its operands. It is also known as postfix notation and does not need any parentheses as long as each operator has a fixed number of operands. For example:

- the expression 2 + 4 in RPN is represented like 2 4 +
- the expression 2 * 4 + 8 in RPN is represented like 2 4 * 8 +
- the expression 2 * (4 + 8) in RPN is represented like 2 4 8 + *

Evaluate the value of an arithmetic expression in Reverse Polish Notation. Valid operators are +, -, *, /. Operator / is an integer division (14 / 3 = 4). Each operand may be an integer or another expression.

Input. One line contains expression written in Reverse Polish notation. The length of expression is no more than 100 symbols.

Output. Print the value of expression given in Reverse Polish notation.

Sample input 1

```
2 4 * 8 +
```

Sample output 1

```
16
```

Sample input 2

```
2 4 8 + *
```

Sample output 2

```
24
```

E-OLYMP 5060. Reverse Polish notation

Example 1. The expression “2 4 * 8 +” is equivalent to “2 * 4 + 8”.

Example 2. The expression “2 4 8 + *” is equivalent to “2 * (4 + 8)”.

Question 1. Represent “(2 + 4) * 8” in reverse polish notation.

Question 2. Represent “(2 + 4) * (6 + 1)” in reverse polish notation.

Question 3. Represent “(5 + 3) / 4” in reverse polish notation.

E-OLYMP 5060. Reverse Polish notation

Example 1. The expression “2 4 * 8 +” is equivalent to “2 * 4 + 8”.

Example 2. The expression “2 4 8 + *” is equivalent to “2 * (4 + 8)”.

Question 1. Represent “(2 + 4) * 8” in reverse polish notation. 2 4 + 8 *

Question 2. Represent “(2 + 4) * (6 + 1)” in reverse polish notation. 2 4 + 6 1 + *

Question 3. Represent “(5 + 3) / 4” in reverse polish notation. 5 3 + 4 /

Question 4. Find the value of 3 4 + 1 2 + *

Question 5. Find the value of 1 2 3 4 + * -

E-OLYMP 5060. Reverse Polish notation

Example 1. The expression “2 4 * 8 +” is equivalent to “2 * 4 + 8”.

Example 2. The expression “2 4 8 + *” is equivalent to “2 * (4 + 8)”.

Question 1. Represent “(2 + 4) * 8” in reverse polish notation. 2 4 + 8 *

Question 2. Represent “(2 + 4) * (6 + 1)” in reverse polish notation. 2 4 + 6 1 + *

Question 3. Represent “(5 + 3) / 4” in reverse polish notation. 5 3 + 4 /

Question 4. Find the value of 3 4 + 1 2 + * (3 + 4) * (1 + 2) = 21

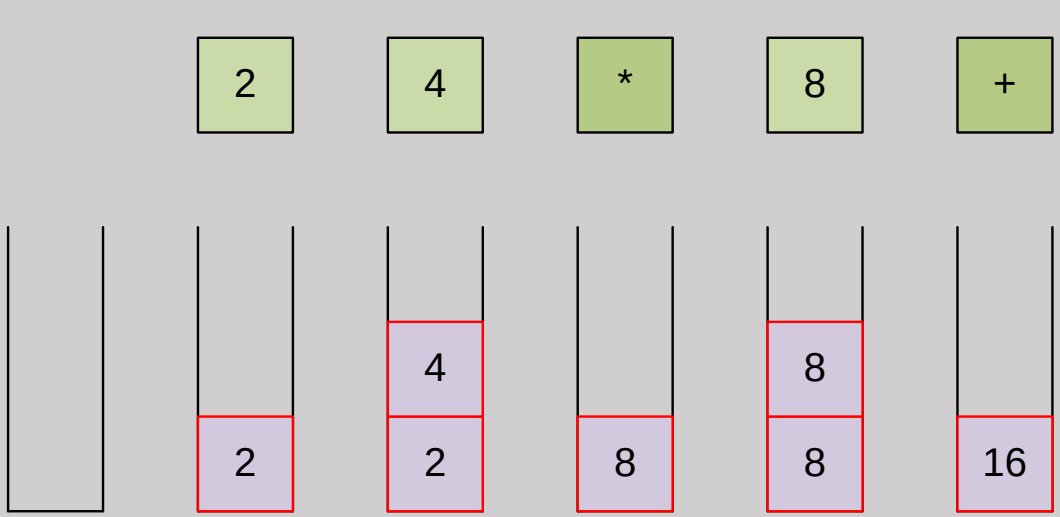
Question 5. Find the value of 1 2 3 4 + * - (1 - (2 * (3 + 4))) = -13

E-OLYMP 5060. Reverse Polish notation

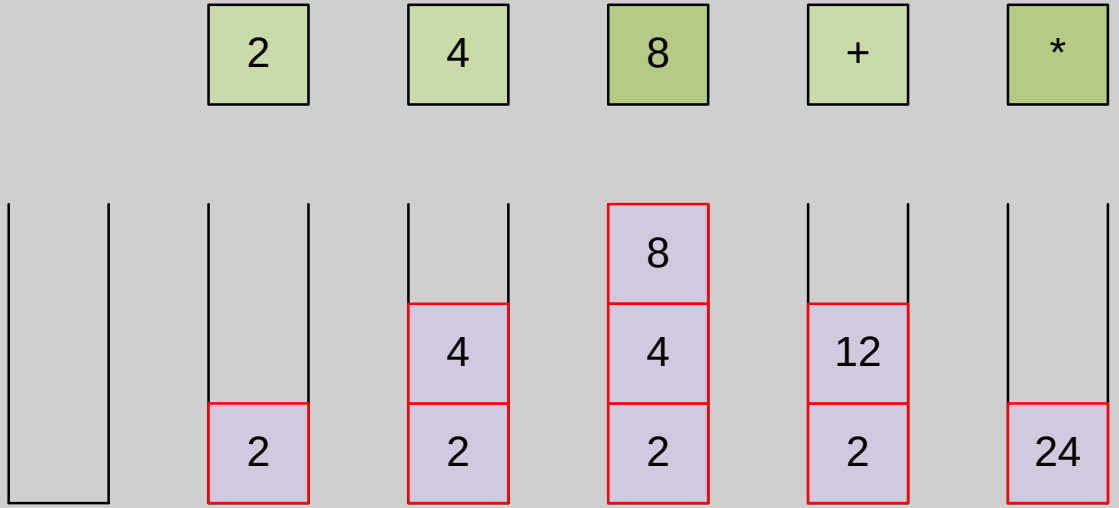
Let’s partition the input expression into terms, which are the number or one of the four operators. The terms will be processed as follows:

- if term is a *number*, push it into stack;
- if term is an *operator*, extract two numbers from the stack, perform the operation and push the result into stack.

When the expression is processed, the stack contains one number that is the result of calculations.



2 * 4 + 8



2 * (4 + 8)

E-OLYMP 940. Majority element

Given an array of size n, find the majority element. The majority element is the element that appears more than $[n / 2]$ times.

Input. First line contains number n ($1 \leq n \leq 100$). Second line contains n positive integers.
Output. If the array contains majority element, then print it. Otherwise print -1.

Sample input 1

```
7
3 3 5 4 2 3 3
```

Sample output 1

```
3
```

Sample input 2

```
4
2 3 2 3
```

Sample output 2

```
-1
```

Consider the sequence of numbers:

4, 5, 4, 6, 7, 5, 1, 5, 7

Add the minimum number of integers so that it will have a majority element

E-OLYMP 940. Majority element

Given an array of size n, find the majority element. The majority element is the element that appears more than $[n / 2]$ times.

Input. First line contains number n ($1 \leq n \leq 100$). Second line contains n positive integers.
Output. If the array contains majority element, then print it. Otherwise print -1.

Sample input 1

```
7
3 3 5 4 2 3 3
```

Sample output 1

```
3
```

Sample input 2

```
4
2 3 2 3
```

Sample output 2

```
-1
```

Consider the sequence of numbers:

4, 5, 4, 6, 7, 5, 1, 5, 7, 5, 5, 5, 5

Add the minimum number of integers so that it will have a majority element

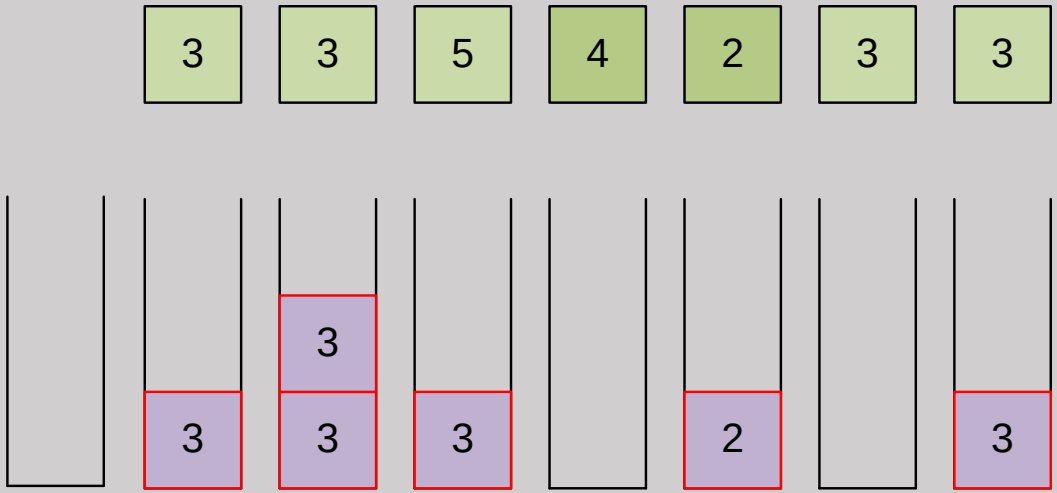
E-OLYMP 940. Majority element

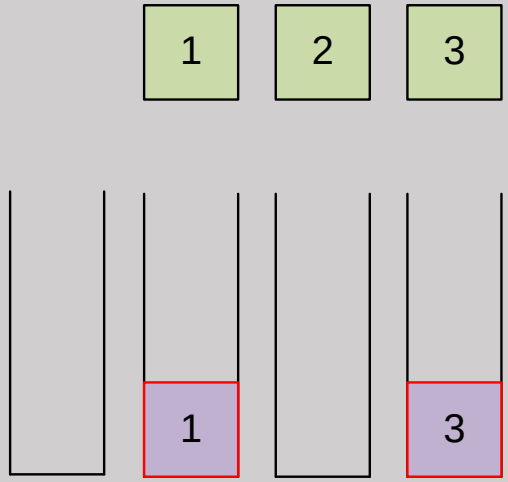
Let *x* be the majority element. Let's start to process the input data. Each number equals to *x* we shall ***push*** into the stack. If any other number arrives, we shall ***pop*** one number out of stack. At the end of data processing the top of the stack will contain the majority element.

Initially stack is empty. When processing the next element *a*:

- If stack is empty, then **push**(*a*);
- If the top of the stack contains number *a*, then **push**(*a*);
- If the top of the stack contains number other than *a*, then **pop**();

Majority element
is on the top of the stack
But what to do in this case?

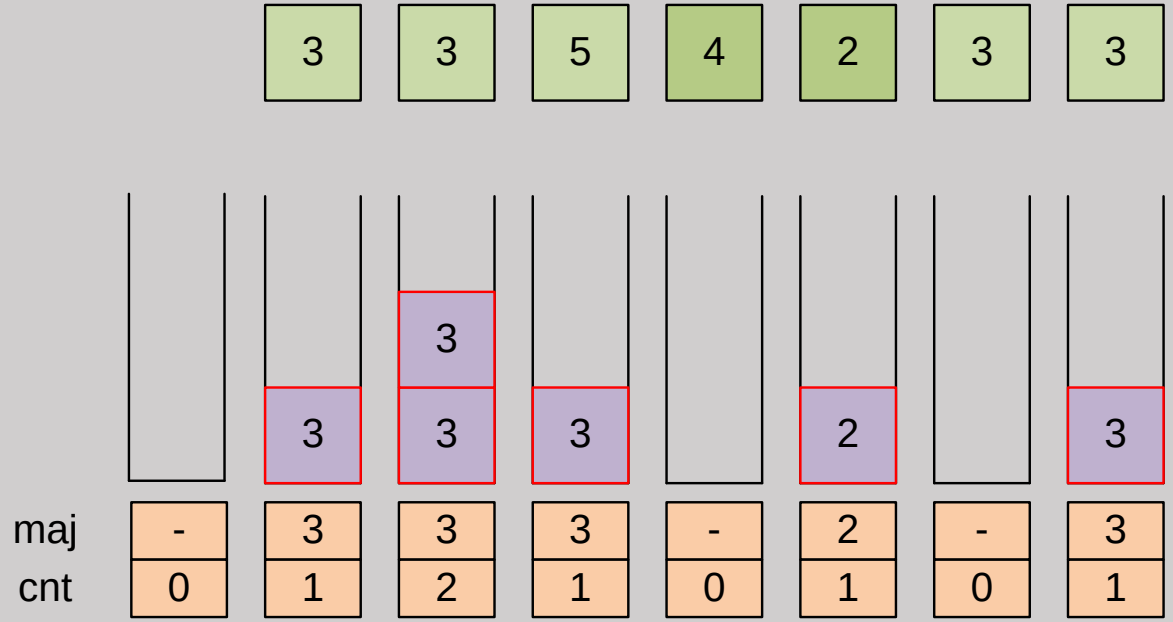




E-OLYMP 940. Majority element

At any time stack contains only one element (possibly multiple times), so let's simulate the stack with two variables:

- *maj* – number in the stack;
- *cnt* – number of times the number *maj* appears in the stack;



If at the end of processing all the numbers, the top of the stack contains some number *x* (if stack is empty, then there is no majority element), then we must check whether it is a majority element. To do this, it is necessary to calculate how many times the number *x* occurs in the original array. If *x* occurs more than [n/2] times, then the answer is positive.

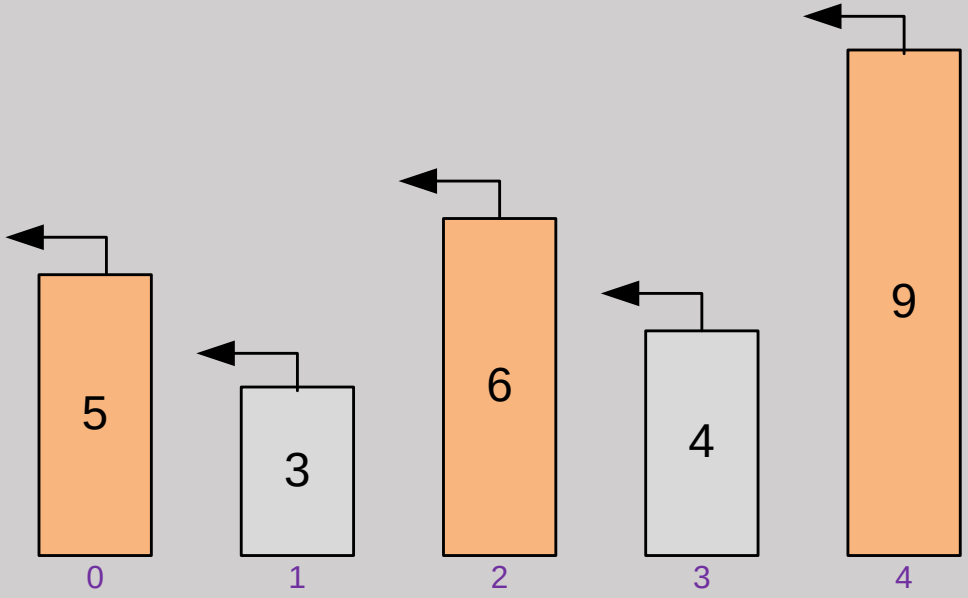
E-OLYMP 10762. Soldiers row

There is a row of n soldiers, identified by indices between 0 and $n - 1$.

Soldier i can see only the soldiers with indices between 0 and $i - 1$.

Soldier has **clear visibility** if he is at least as tall as all of those in front of him. If he doesn't have clear visibility, it means that at least one of the others in front of him is taller.

For each soldier find out if he has clear visibility, and if not, find the closest previous soldier that is taller than him.



Which soldiers have clear visibility?

For each soldier find the closest previous soldier that is taller than him.

E-OLYMP 10762. Soldiers row

There is a row of n soldiers, identified by indices between 0 and $n - 1$.

Soldier i can see only the soldiers with indices between 0 and $i - 1$.

Soldier has **clear visibility** if he is at least as tall as all of those in front of him. If he doesn't have clear visibility, it means that at least one of the others in front of him is taller.

For each soldier find out if he has clear visibility, and if not, find the closest previous soldier that is taller than him.

Input. First line contains number of soldiers n ($1 \le n \le 10^5$). Second line contains the heights of n soldiers.

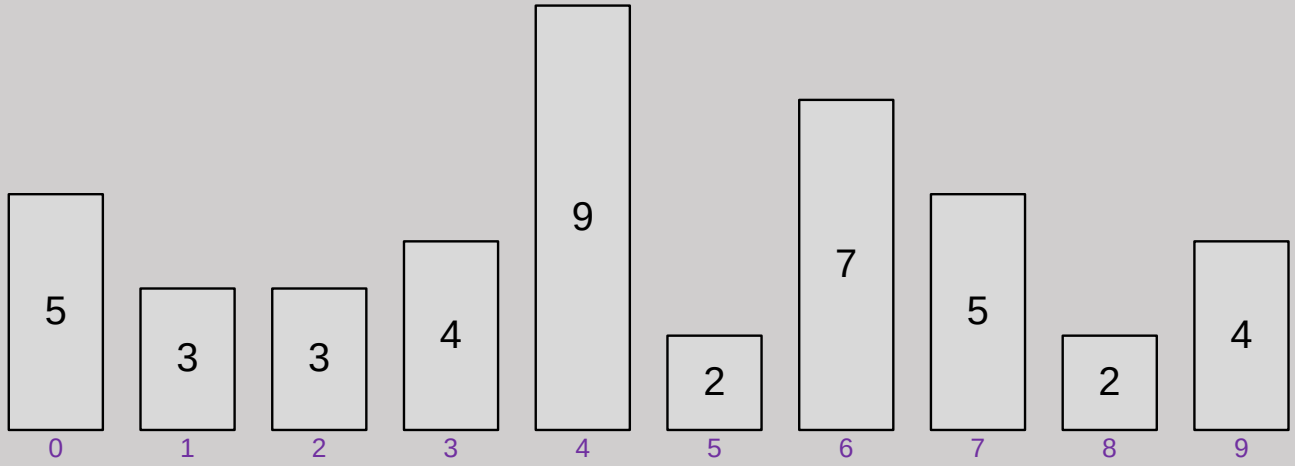
Output. Print n numbers. i-th number must contain the number of the closest previous soldier who is taller than the i-th soldier. If i-th soldier has a clear visibility, print -1.

Sample input

```
10
5 3 3 4 9 2 7 5 2 4
```

Sample output

```
-1 0 0 0 -1 4 4 6 7 7
```

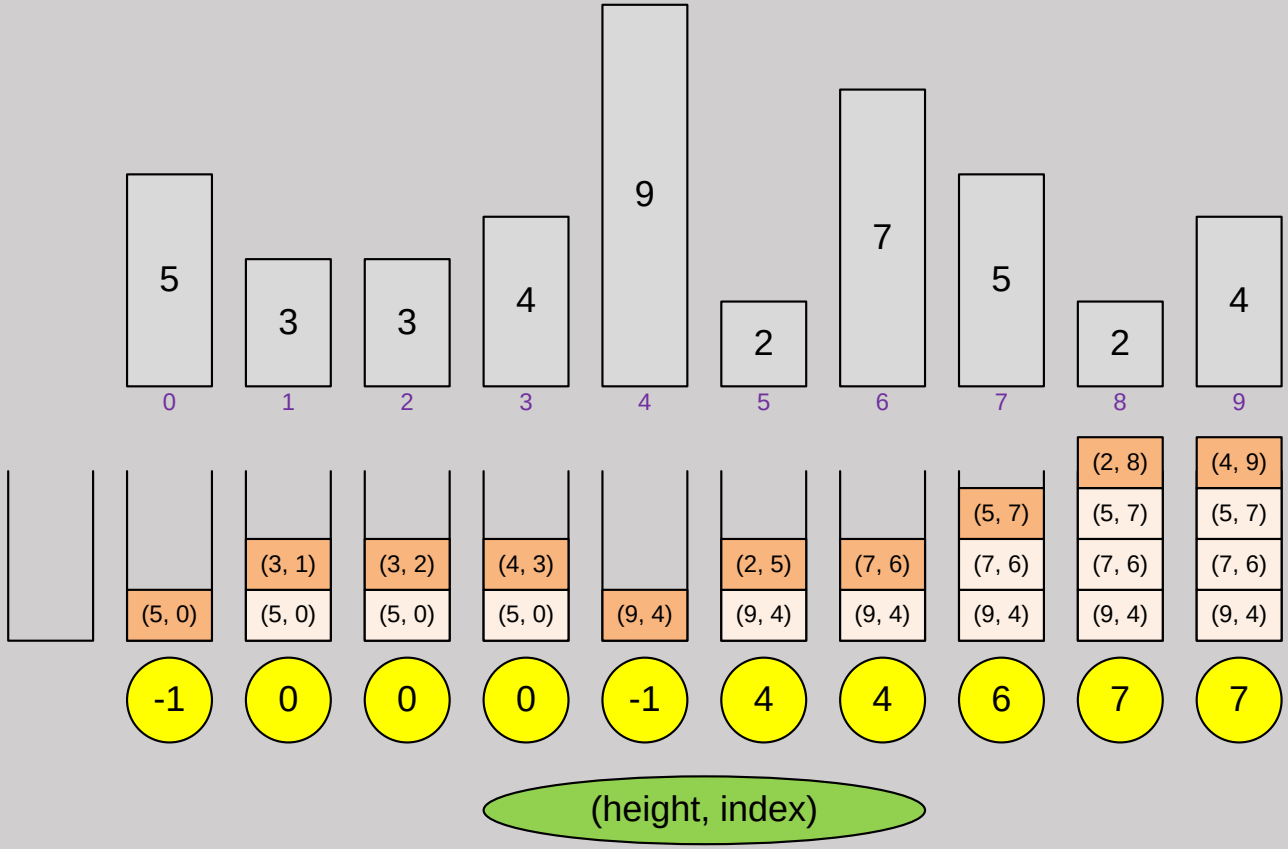


E-OLYMP 10762. Soldiers row

Let’s declare a stack of pairs that will store information about the soldier: the height and the index. We’ll process the soldiers sequentially from left to right. When processing the *i*-th soldier:

- delete from the stack soldiers with heights not exceeding the height of the *i*-th soldier;
- the soldier at the top of the stack will be the closest previous soldier that is higher than the *i*-th one;
- push to the stack the information about the current soldier;

At any moment of time, the stack stores soldiers in descending order of their heights. When the next soldier arrives, all soldiers, no higher than him, are removed from the stack. Then a new soldier takes place at the top of the stack.



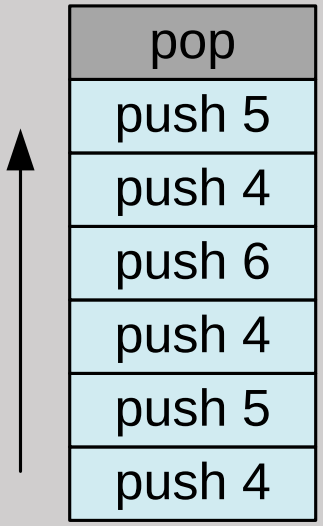
E-OLYMP 10379. Maximum frequency stack

Design a stack-like data structure to push elements to the stack and pop the most frequent element from the stack. The possible commands are:

- **push** *n* – pushes an integer *n* onto the top of the stack;
- **pop** – removes and prints the most frequent element in the stack. If there is a tie for the most frequent element, the element closest to the stack's top is removed and printed.

Input. Each line contains a single command.

Output. For each ***pop*** command print on a separate line the corresponding result.



Which element will be removed?

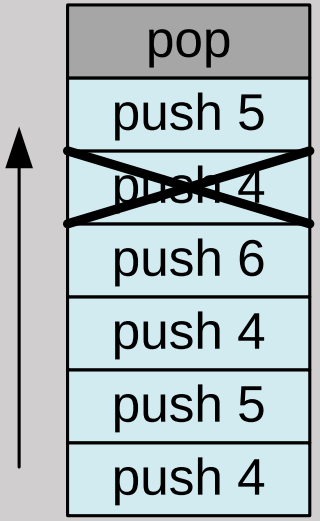
E-OLYMP 10379. Maximum frequency stack

Design a stack-like data structure to push elements to the stack and pop the most frequent element from the stack. The possible commands are:

- **push** *n* – pushes an integer *n* onto the top of the stack;
- **pop** – removes and prints the most frequent element in the stack. If there is a tie for the most frequent element, the element closest to the stack's top is removed and printed.

Input. Each line contains a single command.

Output. For each ***pop*** command print on a separate line the corresponding result.



Which element will be removed?

4

Let we have many pop commands.

In which order elements will be removed?

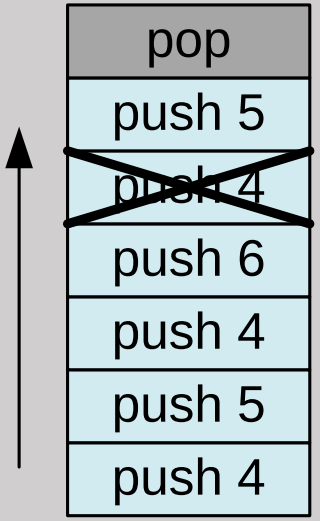
E-OLYMP 10379. Maximum frequency stack

Design a stack-like data structure to push elements to the stack and pop the most frequent element from the stack. The possible commands are:

- **push** *n* – pushes an integer *n* onto the top of the stack;
- **pop** – removes and prints the most frequent element in the stack. If there is a tie for the most frequent element, the element closest to the stack's top is removed and printed.

Input. Each line contains a single command.

Output. For each ***pop*** command print on a separate line the corresponding result.



Which element will be removed?

4

Let we have many pop commands.

In which order elements will be removed?

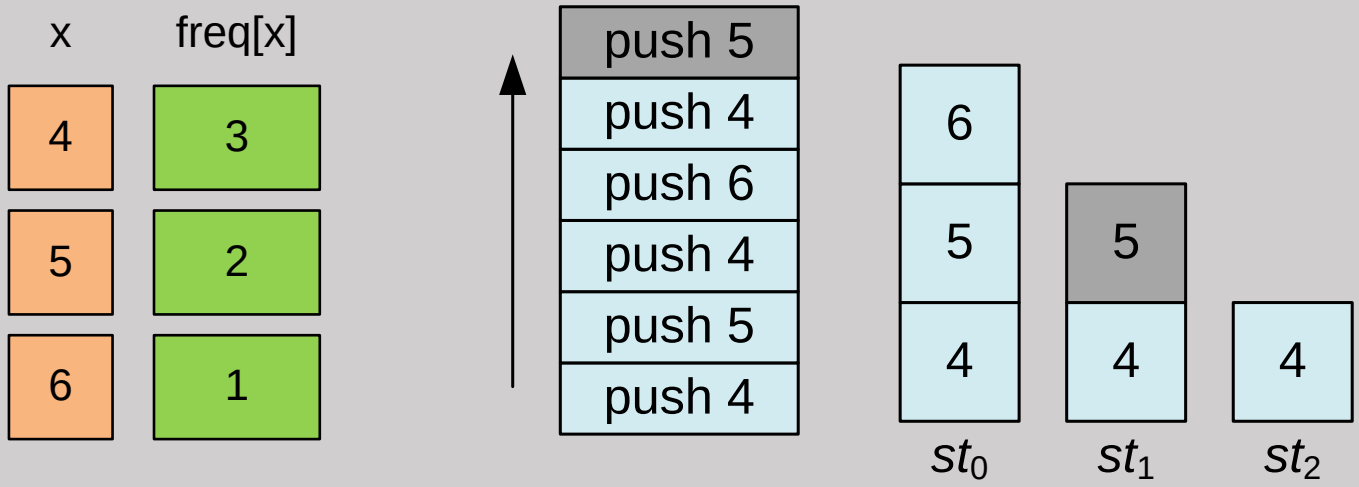
5 4 6 5 4

E-OLYMP 10379. Maximum frequency stack

For each number *x* we'll store the number of times freq[*x*] that it occurs in the stack. Let's choose map as the structure for freq.

Declare an array of stacks `vector<stack<int>> st`. Here st[*i*] stores elements that occur on the stack *i* + 1 times (the numbering of cells in the st array starts from zero). The order in which the elements are in st[*i*] matches the order in which they are pushed onto the stack.

Let the number *x* be pushed to the stack for the *k*-th time. If the element st[*k* – 1] does not exist, then add (push_back) the element to the st array. Then push *x* to the top of the stack st[*k* – 1].



Next operations are:

push 4

push 6

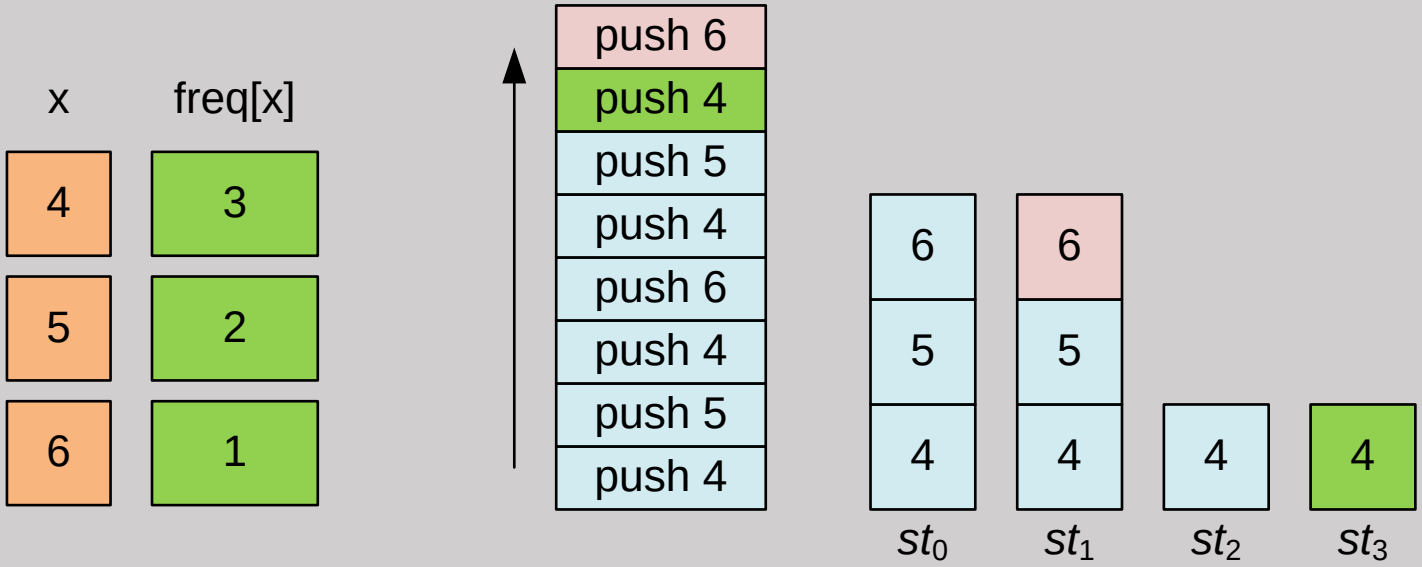
Where the elements will be added?

E-OLYMP 10379. Maximum frequency stack

For each number *x* we'll store the number of times freq[*x*] that it occurs in the stack. Let's choose map as the structure for freq.

Declare an array of stacks `vector<stack<int>> st`. Here st[*i*] stores elements that occur on the stack *i* + 1 times (the numbering of cells in the st array starts from zero). The order in which the elements are in st[*i*] matches the order in which they are pushed onto the stack.

Let the number *x* be pushed to the stack for the *k*-th time. If the element st[*k* – 1] does not exist, then add (push_back) the element to the st array. Then push *x* to the top of the stack st[*k* – 1].



Next operations are:

push 4

push 6

Where the elements will be added?

In which order elements will be removed?

E-OLYMP 4259. Minimum in the stack

Implement a data structure with the next operations:

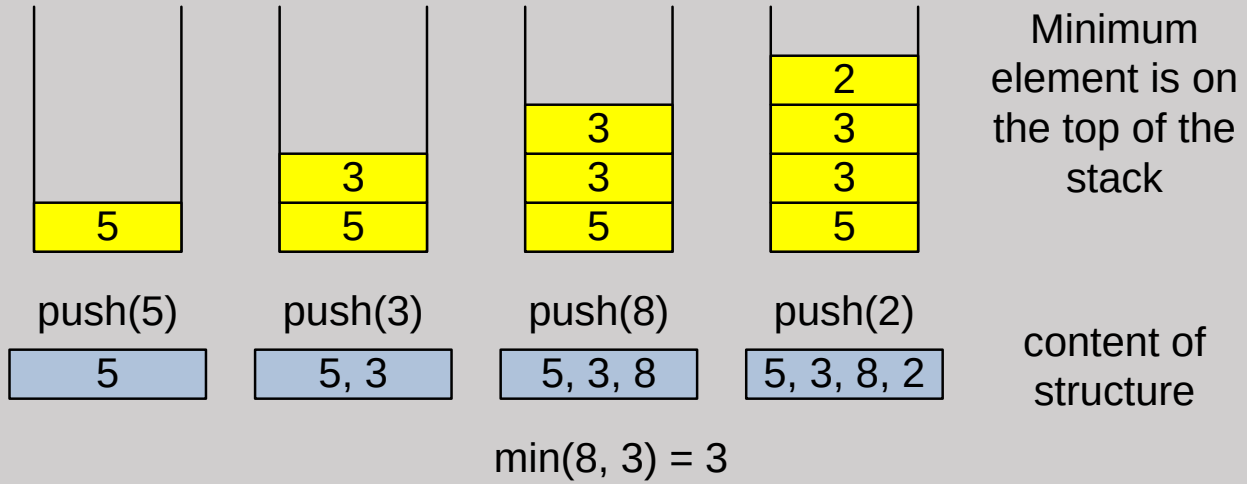
1. **Push** *x* to the end of the structure.
2. **Pop** the last element from the structure.
3. Print the minimum element in the structure.

What is the minimum element in the stack?

E-OLYMP 4259. Minimum in the stack

Implement a data structure with the next operations:

1. **Push** *x* to the end of the structure.
2. **Pop** the last element from the structure.
3. Print the minimum element in the structure.



Obviously, the required data structure is the **stack**.

The **pop**() method will be modeled as usual by removing the top element of the stack.

The **push**(*x*) method will be rewritten as follows:

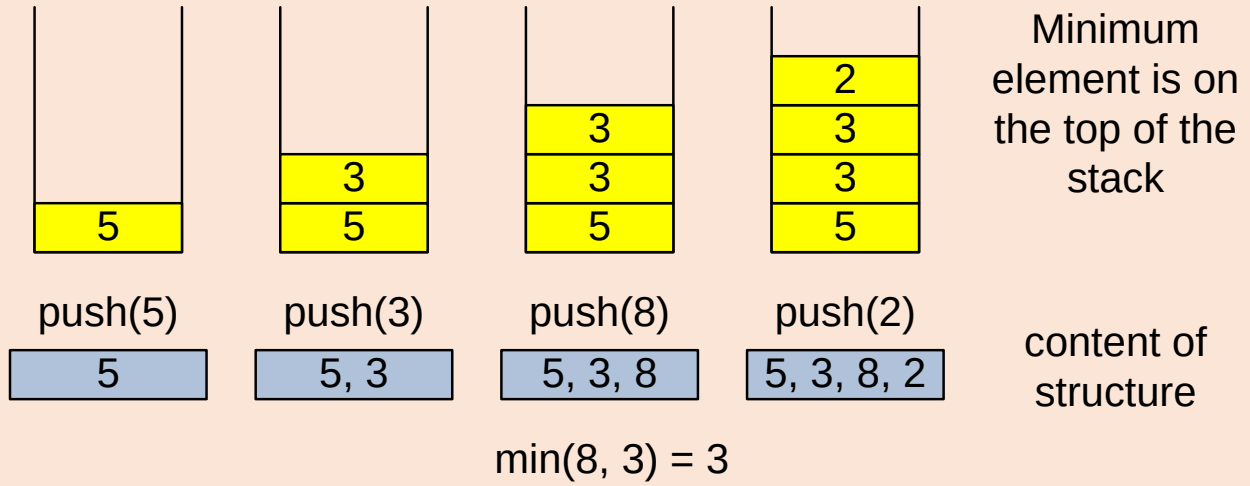
- If stack is empty, ***push*** *x* to the stack;
- Otherwise, ***push*** to the stack the minimum between *x* and the current value of its top.

Thus, the minimum element of the stack will always be at the top. At the same time, we lose the values pushed onto the stack, although in reality they are not needed for further requests.

E-OLYMP 9035. Class MinStack

Implement a data structure with the next operations:

1. **Push** *x* to the end of the structure.
2. **Pop** the last element from the structure.
3. Print the minimum element in the structure.
4. Print the top element in the structure.



We have now both operations:

- **get the minimum element from the stack**
- **get the top element from the stack**

How to implement it?

E-OLYMP 9035. Class MinStack

Implement a data structure with the next operations:

1. **Push** *x* to the end of the structure.
2. **Pop** the last element from the structure.
3. Print the minimum element in the structure.
4. Print the top element in the structure.

Use two stacks:

- **Minimum stack**
- **Standard stack**

We have now both operations:

- **get the minimum element from the stack**
- **get the top element from the stack**

How to implement it?