SNS COLLEGE OF TECHNOLOGY

Coimbatore-35
An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A+' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF AIML

19ITT101-PROGRAMMING IN C AND DATA STRUCTURES
I YEAR - II SEM

UNIT 4 – STACK AND QUEUE

TOPIC 7 – Postfix expression evaluation

Postfix Expression
A B +

Postfix Expression

- Infix expression is the form AOB
  - A and B are numbers or also infix expression
  - O is operator (+, -, *, /)
- Postfix expression is the form ABO
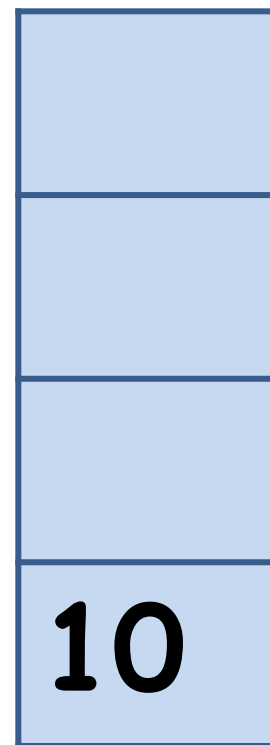 - A and B are numbers or also postfix expression
  - O is operator (+, -, *, /)

- The reason to convert infix to postfix expression is that we can compute the answer of postfix expression easier by using a stack.

Ex: 10 2 8 * + 3 -

•First, push(10) into the stack

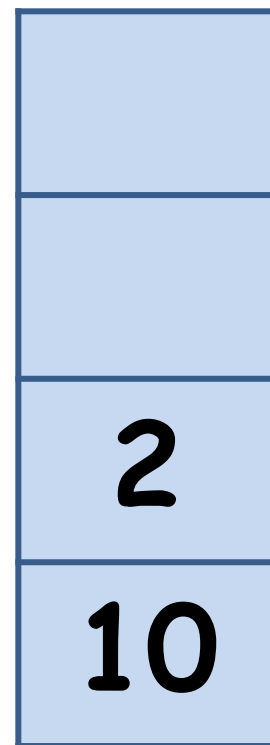| |
|---|
| |
| |
| |
| **10** |

Ex: 10 2 8 * + 3 -

•Then, push(2) into the stack

2
10

Ex: 10 2 8 * + 3 -

- Push(8) into the stack

8
2
10

Ex: 10 2 8 * + 3 -

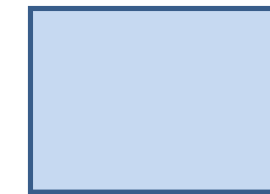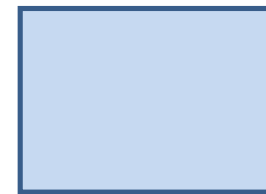•Now we see an operator *, that means we can get an n umber by calculation

8
2
10

Ex: 10 2 8 * + 3 -

• Now we see an operator *, that means we can get an new number by calculation
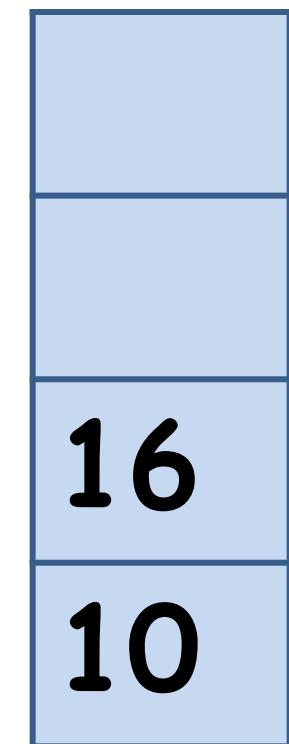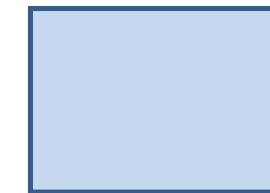
• Pop the first two numbers

2 * 8 = 16

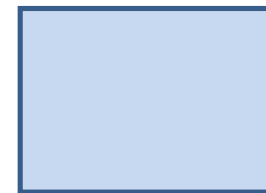| |
|---|
| |
| **8** |
| **2** |
| **10** |

Ex: 10 2 8 * + 3 -

•Now we see an operator *, that means we can get an new number by calculation

•Push the new number back

2   *   8 = 16
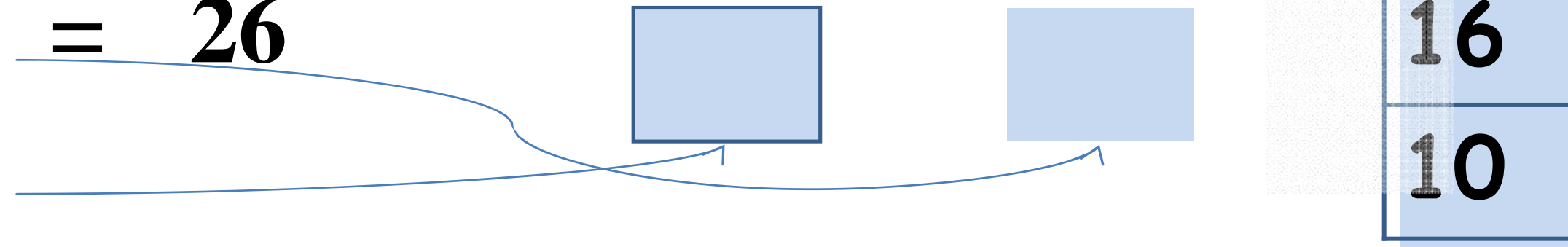
16
10

Ex: 10 2 8 * + 3 -

•Then we see the next operator + and perform the calculation

10 +$_+$ 16 = 26

16
10

Ex: 10 2 8 * + 3 -

•Then we see the next operator + and perform the calculation

•Push the new number back

| 10 + | 16 = 26 |

26
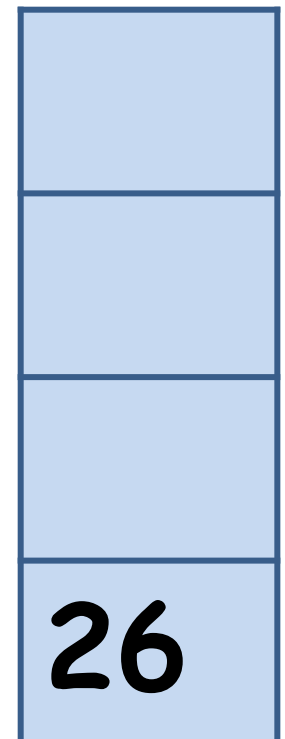
Ex: 10 2 8 * + 3 -

- We see the next number 3
- Push (3) into the stack

3
26
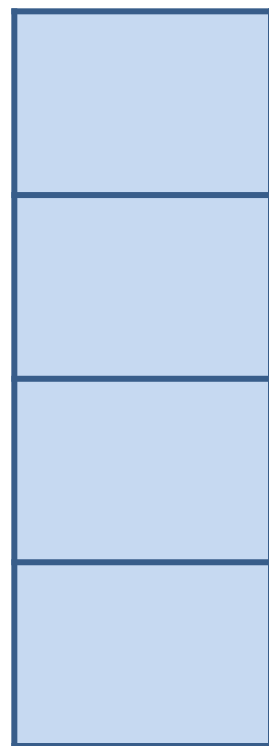
Ex: 10 2 8 * + 3 -

- The last operation

$$26 \quad - \quad 3 \quad = \quad 23$$

Ex: 10 2 8 * + 3 -

- The last operation

23
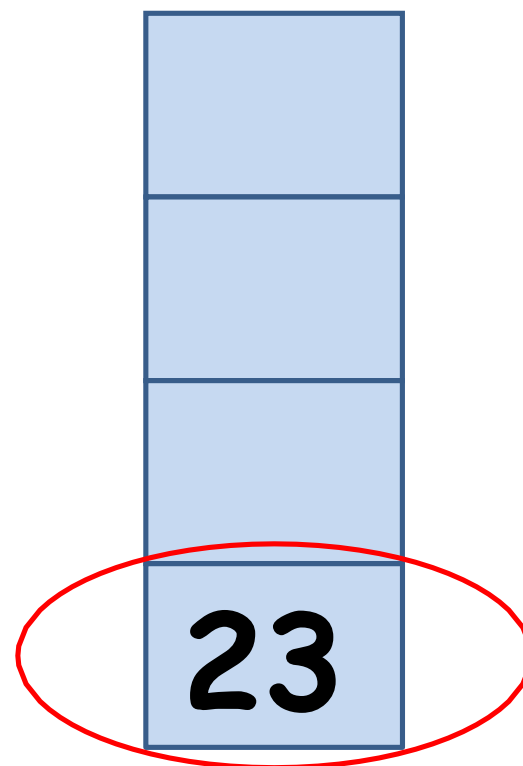
26 - 3 = 23

answer!

- Algorithm: maintain a stack and scan the postfix expression from left to right
  - If the element is a number, push it into the stack
  - If the element is a operator O, pop twice and get A and B respectively. Calculate BOA and push it back to the stack
  - When the expression is ended, the number in the stack is the final answer

- Now, we have to design an algorithm to transform infix expression to postfix

- Observation 1: The order of computation depends on the order of operators
  - The parentheses must be added according to the priority of operations.
  - The priority of operator * and / is higher then those of operation + and –
  - If there are more than one equal-priority operators, we assume that the left one's priority is higher than the right one's
    - This is called left-to-right parsing.

- Observation 1: The order of computation depends on the order of operators (cont.)
  - For example, to add parentheses for the expression 10 + 2 * 8 - 3,
  - we first add parenthesis to 2 * 8 since its priority is highest in the expression.
  - Then we add parenthesis to 10 + (2 * 8) since the priorities of + and – are equal, and + is on the left of -.
  - Finally, we add parenthesis to all the expression and get ((10 + (2 * 8)) - 3).

- Observation 1: The order of computation depends on the order of operators (cont.)

– The computation order of expression ((10 + (2 * 8)) - 3) is:

- 2 * 8 = 16 € ((10 + 16) - 3)
- 10 + 16 = 26 € (26 – 3)
- 26 – 3 = 23 € 23

- Simplify the problem, how if there are only +/- operators?

- Simplify the problem, how if there are only +/- operators?

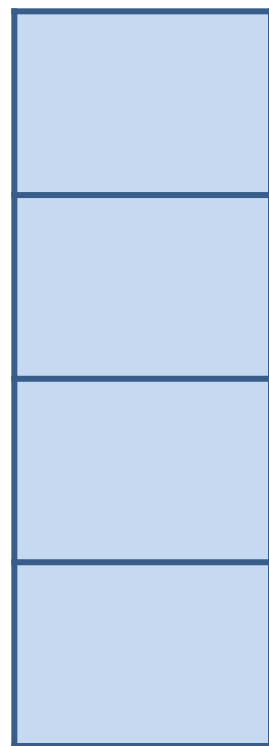- The leftmost operator will be done first

– Ex: 10 - 2 + 3 € 8 + 3 € 11

- Simplify the problem, how if there are only +/- operators?

- Algorithm: maintain a stack and scan the postfix expression from left to right
  - When we get a number, output it
  - When we get an operator O, pop the top element in the stack if the stack is not empty and then push(O) into the stack

- Simplify the problem, how if there are only +/- operators?
- Algorithm: maintain a stack and scan the postfix expression from left to right
  - When we get a number, output it
  - When we get an operator O, pop the top element in the stack if the stack is not empty and then push(O) into the stack
  - When the expression is ended, pop all the operators remain in the stack

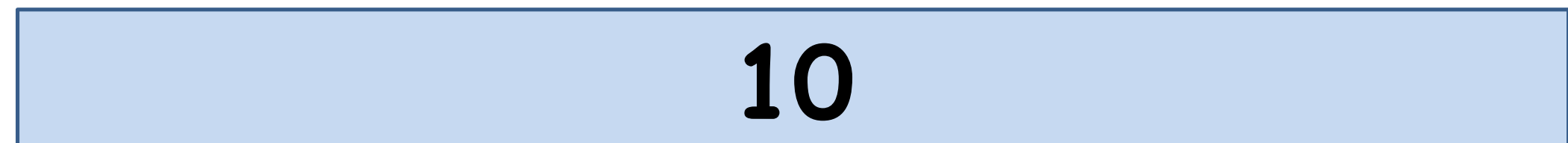Ex: 10 + 2 - 8 + 3

- We see the first number 10, output it

10

Ex: 10 + 2 - 8 + 3

•We see the first operator
+, push(+) into the stack because at this moment the stack is empty
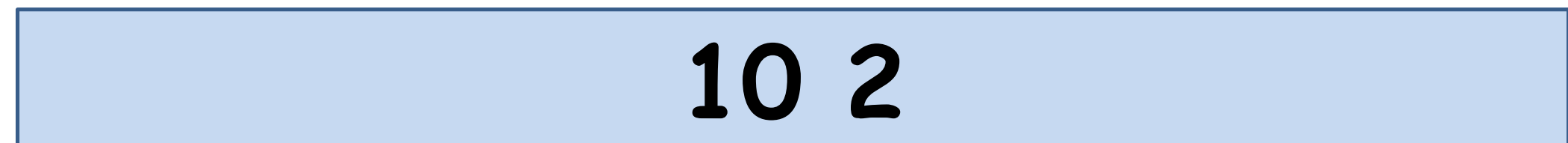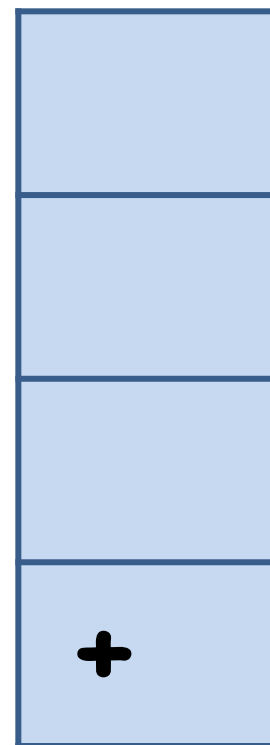
+

10

- Ex: 10 + 2 - 8 + 3
- We see the number 2, output it

+

10 2

Ex: 10 + 2 - 8 + 3

•We see the operator -, pop the operator + and push(-) into the stack

-

10 2 +

Transform Infix to Postfix

- Ex: 10 + 2 - 8 + 3

- We see the number 8, output it

-
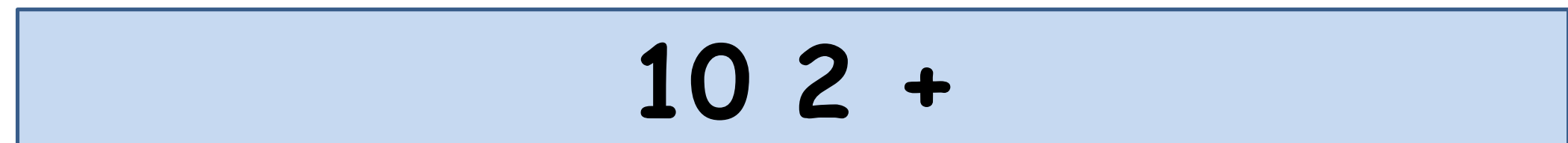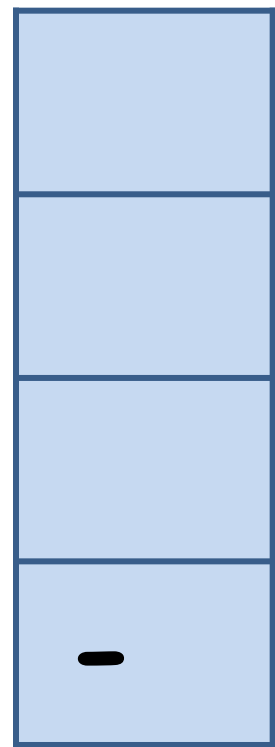
10 2 + 8

Ex: 10 + 2 - 8 + 3

•We see the operator +, pop the operator - and push(+) into the stack

+

10 2 + 8 –

Transform Infix to Postfix

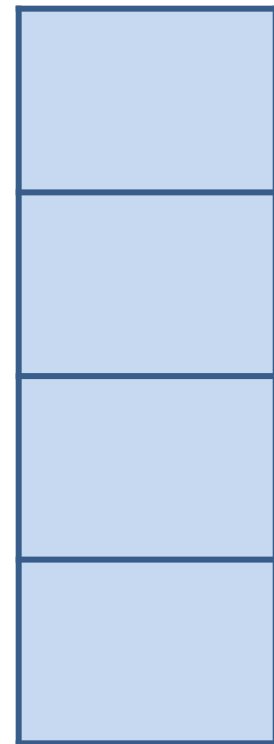- Ex: 10 + 2 - 8 + 3

- We see the number 3, output it

+
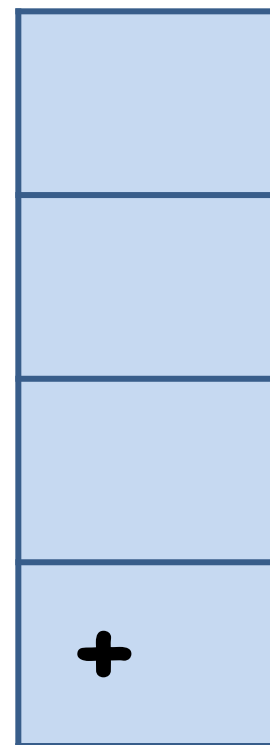
10 2 + 8 – 3

Ex: 10 + 2 - 8 + 3
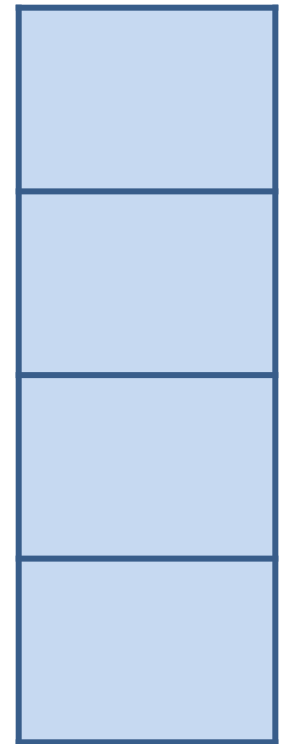
•We come to the end of the expression, then we pop all the operators in the stack

10 2 + 8 – 3 +

Ex: 10 + 2 - 8 + 3

•When we get an operator, we have to push it into the stack and pop it when we see the next operator.

•The reason is, we have to "wait" for the second operand of the operator

- How to solve the problem when there are operators +, -, *, / ?

- Observation 2: scan the infix expression from left to right, if we see higher- priority operator after lower-priority one, we know that the second operand of the lower-priority operator is an expression

  – Ex: a + b * c = a + (b * c) € a b c * +

  – That is, the expression b c * is the second operand of the operator "+"

- So, we modify the algorithm to adapt the situation

- Algorithm: maintain a stack and scan the postfix expression from left to right
  - When we get a number, output it
  - When we get an operator O, pop the top element in the stack until there is no operator having higher priority then O and then push(O) into the stack
  - When the expression is ended, pop all the operators remain in the stack

Ex: 10 + 2 * 8 - 3

- We see the first number 10, output it

| 10 |
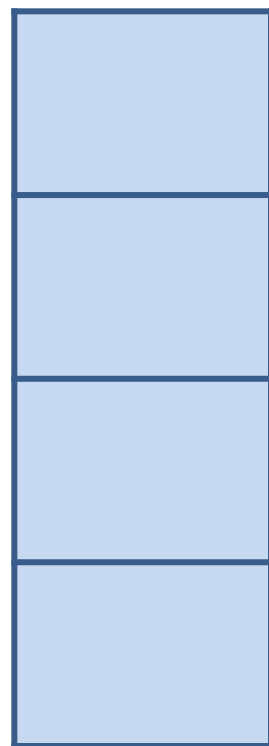
Ex: 10 + 2 * 8 - 3

- We see the first operator +, push it into the stack

+

10

- Ex: 10 + 2 * 8 - 3
- We see the number 2, output it

+

10 2
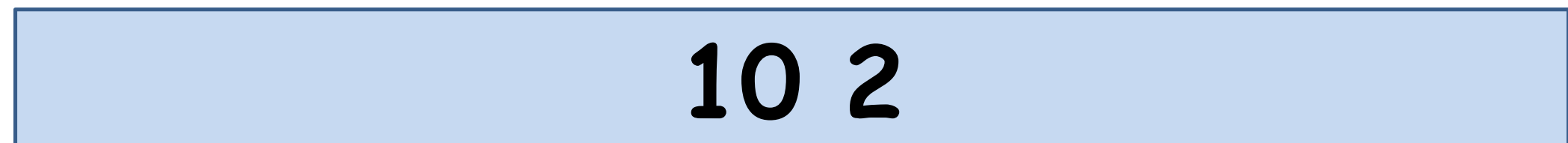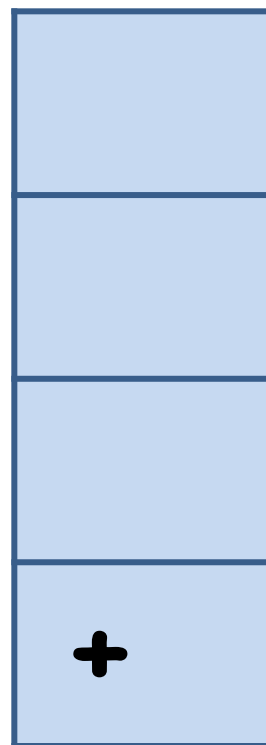
Ex: 10 + 2 * 8 - 3

•We see the operator *, since the top operator in the stack +, has lower priority then *, push(*)
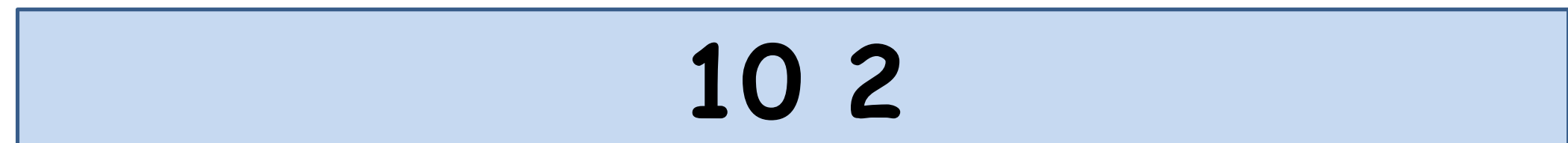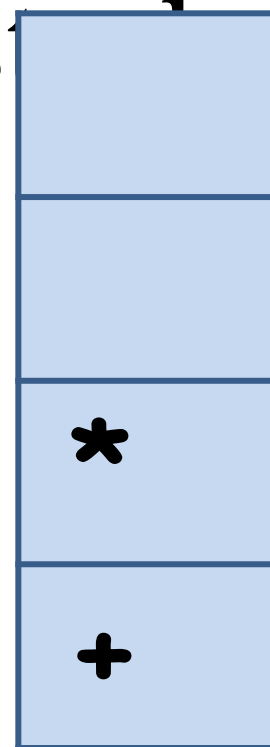
*
+

10 2

- Ex: 10 + 2 * 8 - 3
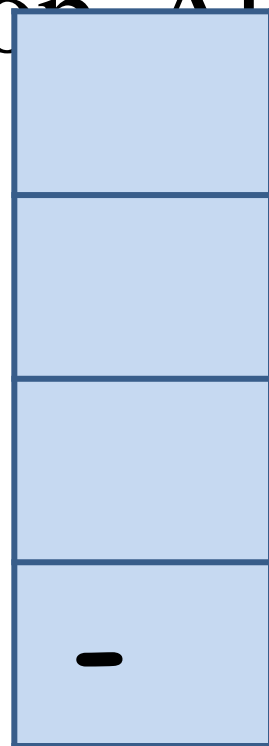- We see the number 8, output it

*
+
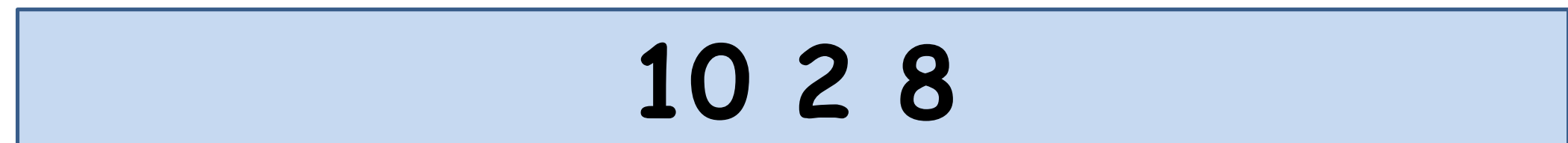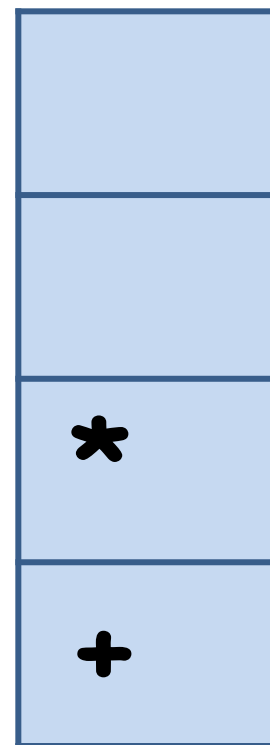
10 2 8

Ex: 10 + 2 * 8 - 3

•We see the operator -, because its priority is lower then *, we pop Also, because + is on the left of it, we pop +, too. Then push(-)

-

10 2 8 * +
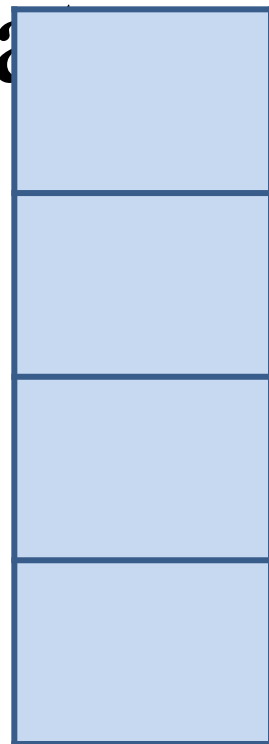
- Ex: 10 + 2 * 8 - 3
- We see the number 3, output it

-

10 2 8 * + 3

Ex: 10 + 2 * 8 - 3

•Because the expression is ended, we pop all the operators in the stack

```
10 2 8 * + 3 -
```