College of computer science & mathematics

Dep. Of Computer Science

DATA STRUCTURE
هياكل بيانات

Lecture 4 : Stack

Prepared & Presented by

Mohammed B. Omar

2023 -2024

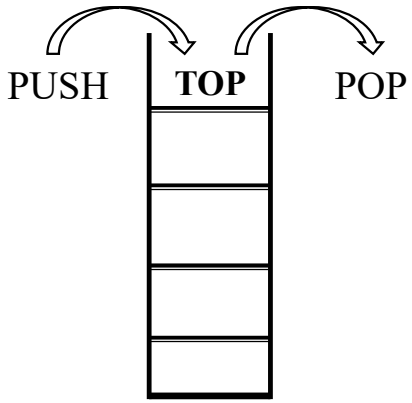
Stack:

What is Stack?

- Stack is a linear data structure in which the insertion and deletion operations are performed at only one end.

In a stack, adding and removing of elements are performed at a single position which is known as **"top"**. That means, a new element is added at top of the stack and an element is removed from the top of the stack. In stack, the insertion and deletion operations are performed based on **LIFO (Last In First Out) principle**.

- **A stack is a homogeneous collection of items of one type, arranged linearly with access at one end only.**



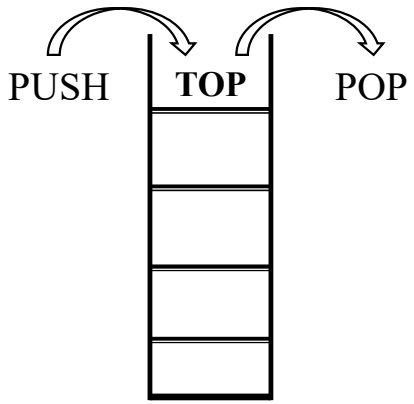
Stack:

Stacks associated with two basic operation i.e **PUSH and POP.**

- ✓ Push function means addition an element to the stack.
- ✓ Pop function means remove an element from the stack.

Stack:

In the figure, PUSH and POP operations are performed at a top position in the stack. That means, both the insertion and deletion operations are performed at one end (i.e., at Top)



- It is type of **linear data structure.**
- It follows **LIFO** (Last In First Out) property.
- It has only one pointer **TOP** that points the last or top most element of Stack.
- **Insertion and Deletion in stack can only be done from top only.**
- Initially, **the top is set to -1**. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.
- Insertion in stack is also known as a **PUSH operation**.
- Deletion from stack is also known as **POP operation** in stack.

There are some operations of Stack.

Stack	Create an empty stack
~Stack	Destroy an existing stack
isEmpty	Determine whether the stack is empty
isFull	Determine whether the stack is full
push	Add an item to the top of the stack
pop	Remove the item most recently added
peek	Retrieve the item most recently added

Stack:

Basic features of Stack

1. As all the deletion and insertion in a stack is done from top of the stack, the last added element will be the first to be removed from the stack.

2. This is the reason why the stack is also called Last In – First Out (LIFO) type of list.

3. It is interesting to notice that the most frequently accessible element in the stack are the most top elements, while the least accessible elements located at the bottom of the stack.

4. Stack is an ordered list of similar **data type**.

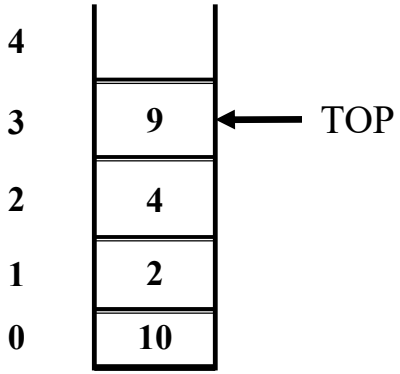
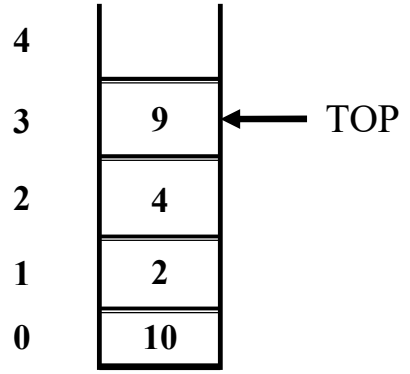


Figure represents a stack with the size of Five locations and includes Four elements.

Stack:

The stack can be represented using a single array with the required capacity (SIZE) and the appropriate type of data (DATATYPE) that will be stored in it (INT, FLOAT, ...etc.) with the use of an independent variable called (TOP) that is used as an indicator indicating the location of the highest element in the stack. Starting with the value of the indicator (TOP = -1) when the stack is free of elements, and the stack is defined programmatically in a language C++ :

```
const int size = 9 ;     {or any other value}
int stackelement;        {or any other type}
stackelement stack[size];
Int top;
```



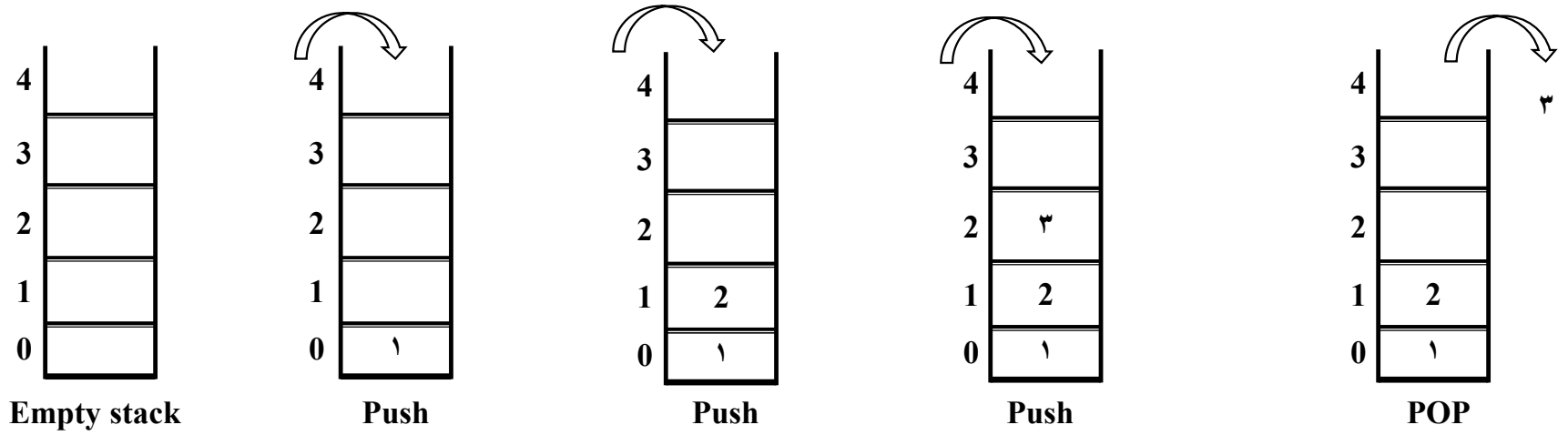
Stack:

There are two basic operations that can be performed on the stack, these are:

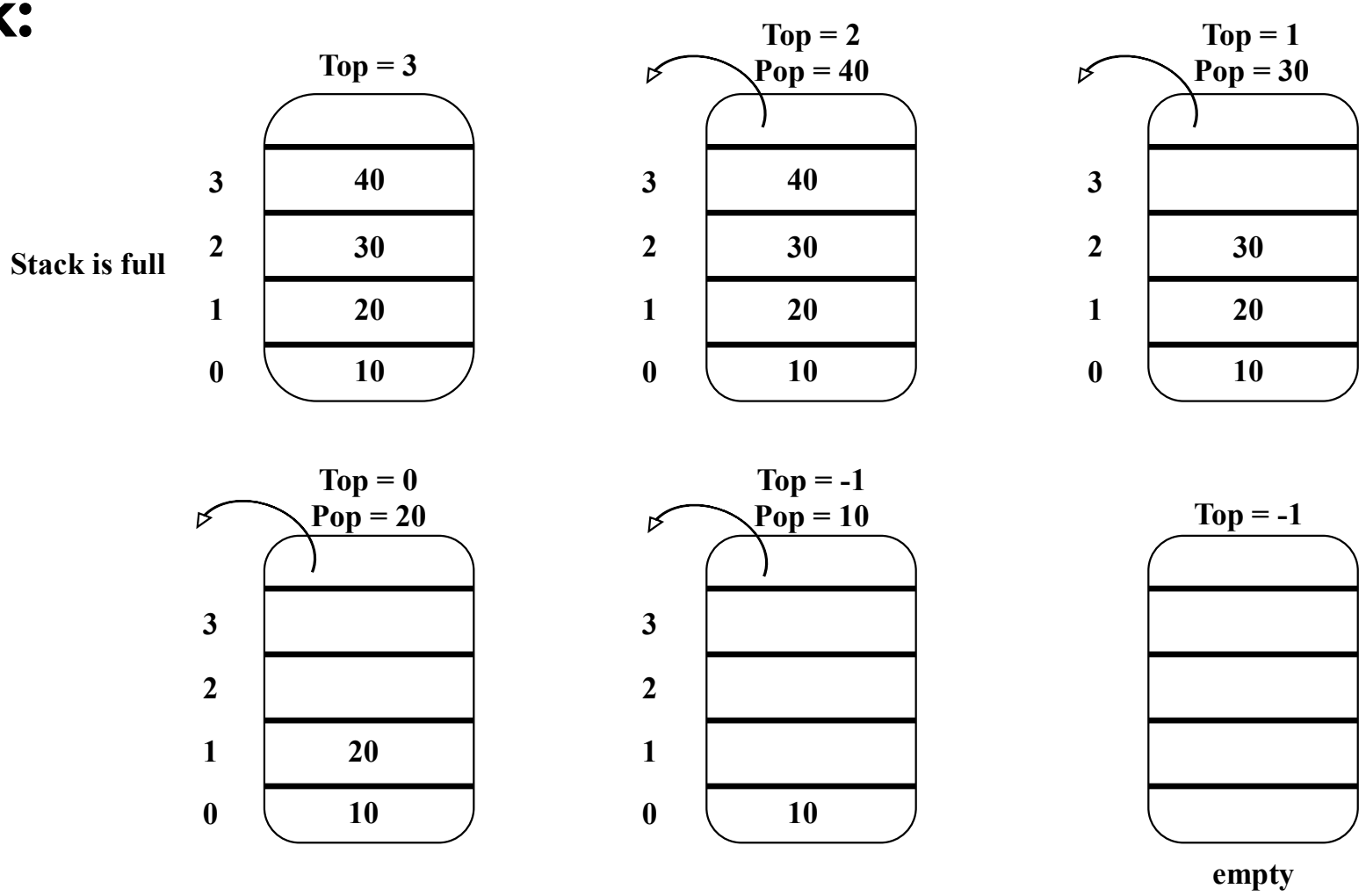
1**. PUSH:** is the process of adding a new element to the top of the stack. As a new element pushed to the stack, top will be incremented by one and denote to the added element. Adding a new element when the stack is full is called stack overflow.

2. **POP:** is the process of deleting an element from the top of the stack. After every pop operation, the top is decremented by one. If there are no elements in the stack and pop operation is performed, the result is called stack underflow.

Figure show an example of push and pop operations

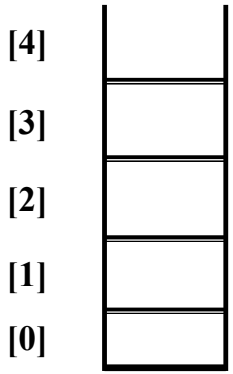


Stack:


Top = 3
3
40
2
30
1
20
0
10
Stack is full
Top = 2
Pop = 40
3
40
2
30
1
20
0
10
Top = 1
Pop = 30
3
2
30
1
20
0
10
Top = 0
Pop = 20
3
2
1
20
0
10
Top = -1
Pop = 10
3
2
1
0
10
Top = -1
empty

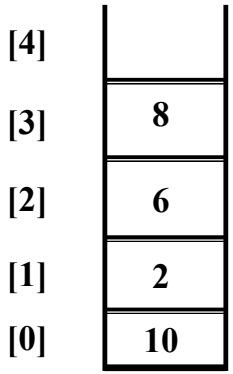
Stack:

Case One



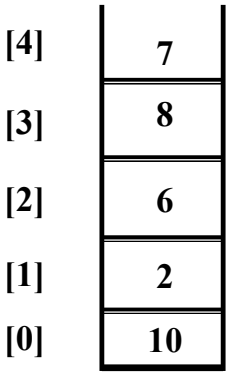
TOP = Null
Max = 5
Empty Stack

Case Two



TOP = [3]
Max = 5

Case Three



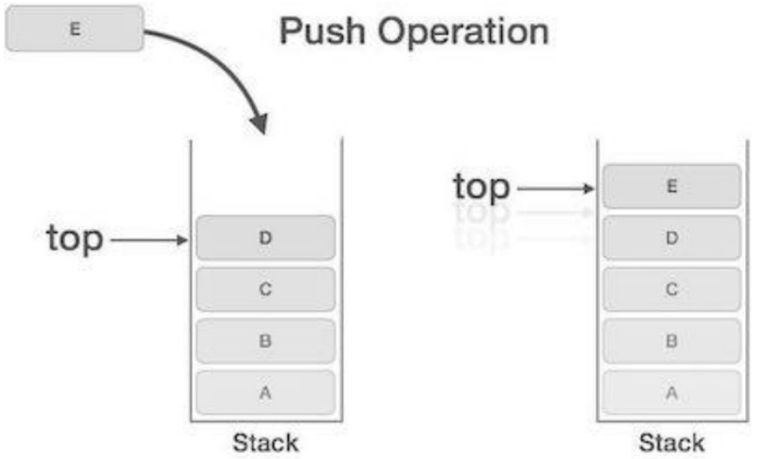
TOP = [4]
Max = 5
Full Stack
Top = Max - 1

Stack:

1. Insertion in Stack/Push Operation

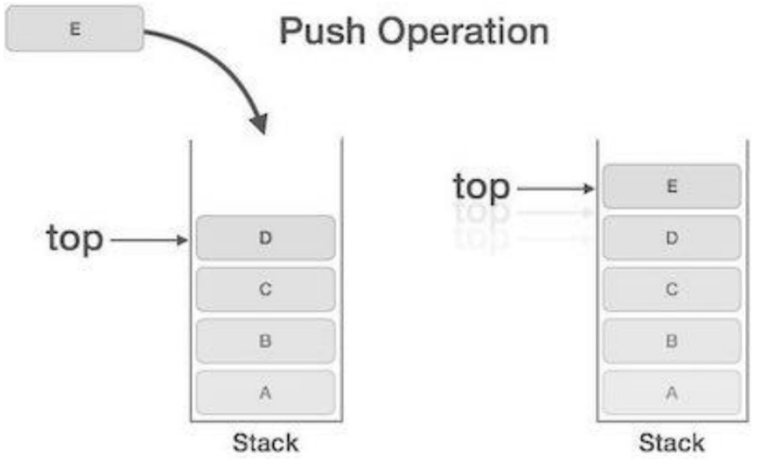
The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full. **If Top = Max -1 then the stack is full and no more insertion can be done**
- **Step 2** − If the stack is full, display "stack is FULL"and exit. **An overflow message is printed.**
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.



Stack:

Step 1: [Check for stack overflow]
if top >=MAXSTACK
cout<< "Stack overflow" and exit
Step 2: [Increment the pointer value by one]
top=top+1
Step 3: [Insert the item]
arr[top]=value
Step 4: Exit



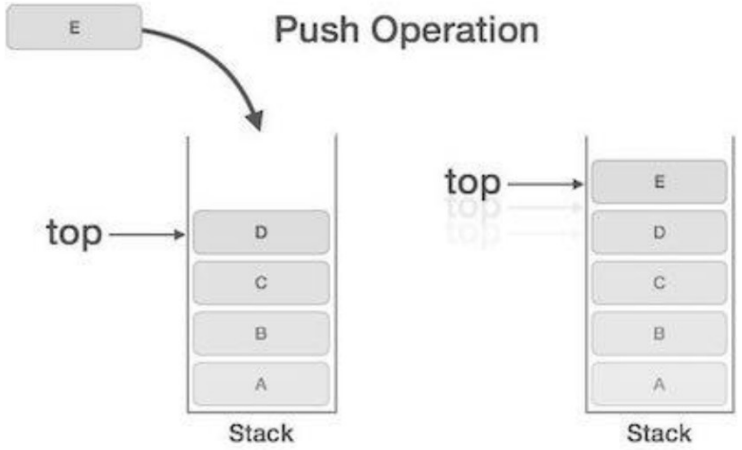
Stack:

Algorithm to Insert an Element in stack

STEP 1 : IF TOP = MAX -1
PRINT " OVERFLOW"
GO TO STEP 4
[END OF IF]

STEP 2 : SET TOP = TOP + 1;
STEP 3 : SET STACK[TOP] = DATA;

STEP 4 : END



TOP = 3
Max = 5
Top = Max – 1
5 - 1
= 4
Top = Top + 1
= 3 + 1
= 4

Stack[4]= E

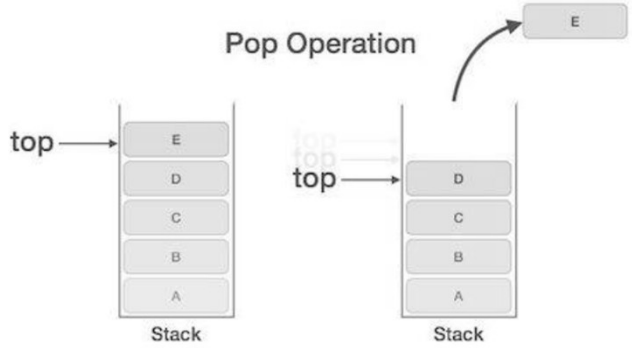
Stack:

2. Deletion/Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead top is decremented to a lower position in the stack to point to the next value.

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty. **Check if Top = Null mean that stack is empty and no more deletion can be done.**
- **Step 2** − If the stack is empty, produces an error and exit. **An Underflow message is printed.**
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Stack:

Step 1: [Check whether the stack is empty]

if top = -1

cout<< " Stack underflow " and exit

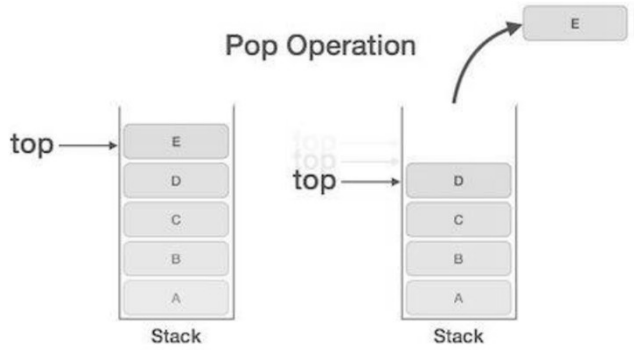
Step 2: [Remove the top most item]

value=arr[top]

top=top-1

Step 3: [Return the item of the stack]

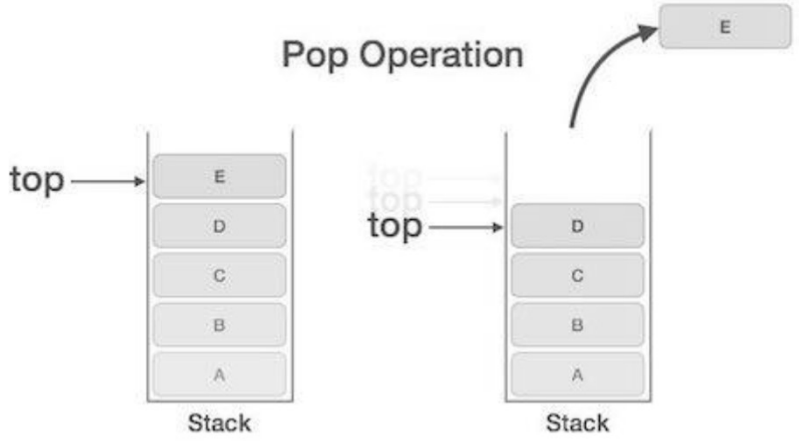
return(value)



Stack:

STEP 1 : IF TOP = NULL
PRINT "UNDERFLOW"
GO TO STEP 4
[END OF IF]

STEP 2 : SET VAL = STACK [TOP]
STEP 3 : SET TOP = TOP – 1
STEP 4 : END



TOP = 4
Val = stack [4]
Val = E
Top = top – 1
= 4 - 1
Top = 3

Stack:

display() - Displays the elements of a Stack

We can use the following steps to display the elements of a stack...

- **Step 1 -** Check whether **stack** is **EMPTY**. (**top == -1**)
- **Step 2 -** If it is **EMPTY**, then display **"Stack is EMPTY!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then define a variable '**i**' and initialize with top. Display **stack[i]** value and decrement **i** value by one (**i--**).
- **Step 3 -** Repeat above step until **i** value becomes '0'.

Stack:

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks:

- **peek() − get the top data element of the stack, without removing it.**
- **isFull() − check if stack is full.**
- **isEmpty() − check if stack is empty.**

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named top. The top pointer provides top value of the stack without actually removing it. First we should learn about procedures to support stack functions .

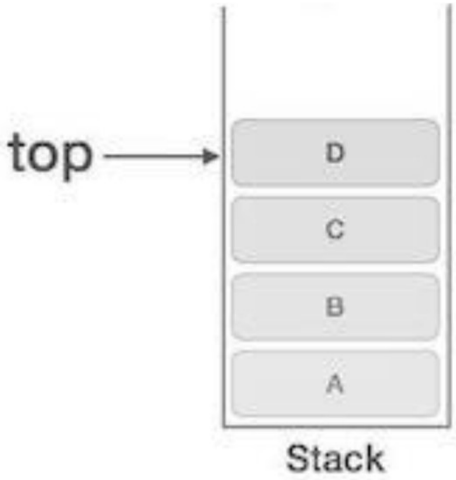
Stack:

Returns value of topmost element in stack (Peek)

- Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack.
- However , the peek operation first checks if the stack is empty, if TOP = Null, then an appropriate message is printed, else the value is returned.

STEP 1 : IF TOP = NULL
PRINT “STACK IS EMPTY”
GO TO STEP 3

STEP 2 : RETURN STACK[TOP]
STEP 3 : END



TOP = 3
Return stack [3]
= D

Stack:

Isfull()

STEP 1 : IF TOP = MAXSIZE
RETURN TRUE

[END OF IF]

STEP 2 : RETURN FALSE
STEP 3 : END

Isempty()

STEP 1 : IF TOP == -1
RETURN TRUE

[END OF IF]

STEP 2 : RETURN FALSE
STEP 3 : END

Where are Stacks used?

- Stack frames are used to store return addresses, parameters, and local variables in a function calling.
- Computer graphics (OpenGL). The sequence of transformations uses a last-specified, first-applied rule. Thus, a stack of transformations is maintained.
- Robotics: Instructions are stores in a stack. We can apply stack controllers such as repeat loops to these stacks.
- Architecture of computers uses stacks to do arithmetic's (eg. Intel FPU).

Stack:

Stack Terminology

1. **Size:** this term refers to the maximum size of the stack or the number of possibly added elements.

2. **TOP:** this term refers to the top of the stack. It is a stack pointer used to check overflow and under flow conditions. The initial value of TOP is -1 when the stack is empty.

3. **Stack underflow:** is the situation when the stack contains no elements. At this point the top of the stack points to the stack bottom.

4. **Stack overflow:** is the situation when the stack is full and no more elements can be added. At this point the top of the stack points to the highest location in the stack. See figure.

Position of Top	Status of Stack
-1	Stack is Empty
0	Only one element in Stack
N-1	Stack is Full
N	Overflow state of Stack

Stack:

Example 1:
What is the obtained string after performing the following sequence of push and pop:
PUSH(A), PUSH(B), POP, PUSH(C), POP, PUSH(D), PUSH(E), POP, POP, POP

Solution:

step	Operation	Stack	Output
1	PUSH(A)	A	
2	PUSH(B)	AB	
3	POP	A	B
4	PUSH(C)	AC	B
5	POP	A	BC
6	PUSH(D)	AD	BC
7	PUSH(E)	ADE	BC
8	POP	AD	BCE
9	POP	A	BCED
10	POP		BCEDA

So, the obtained string is (BCEDA)

Stack:

Example 2:

What is the obtained number after performing the following sequence of push and pop:
PUSH(1), POP, PUSH(2), PUSH(3), POP, PUSH(4), POP, PUSH(5), POP, POP.

Solution:

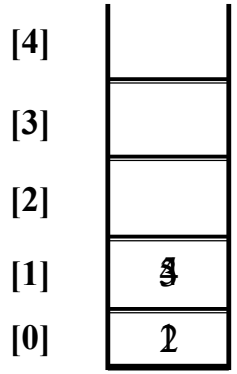
Step	Operation	Stack	Output
1-	PUSH(1)	1	
2-	Pop	…….	1
3-	PUSH(2)	2	1
4-	PUSH(3)	2 3	1
5-	POP	2	1 3
6-	PUSH(4)	2 4	1 3
7-	POP	2	1 3 4
8-	PUSH(5)	2 5	1 3 4
9-	POP	2	1 3 4 5
10-	POP	…….	1 3 4 5 2

So the obtained number is (13452)

Stack:

Example 2:
What is the obtained number after performing the following sequence of push and pop:
PUSH(1), POP, PUSH(2), PUSH(3), POP, PUSH(4), POP, PUSH(5), POP, POP.
Solution:

PUSH(1)
POP
PUSH(2)
PUSH(3)
POP
PUSH(4)
POP
PUSH(5)
POP
POP



Stack:

Example 3:
What is the required sequence of push and pop to obtain the string (CBDAE) from the initial input (ABCDE).
Solution:

step	Operation	Stack	Output
1	PUSH(A)	A	
2	PUSH(B)	AB	
3	PUSH(C)	ABC	
4	POP	AB	C
5	POP	A	CB
6	PUSH(D)	AD	CB
7	POP	A	CBD
8	POP		CBDA
9	PUSH(E)	E	CBDA
10	POP		CBDAE

Stack:

Example 3:

If the stack input set is in order 1,2,3,4,5, which of the following outputs is correct according to the stack operation method?

A- 2 4 5 3 1

B- 4 2 3 1 5

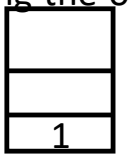
Solution:

Suppose that S means Stacking, which symbolizes the process of adding an element to the stack, and U means Unstacking, which symbolizes the process of deleting an element from the stack.

A- 2 4 5 3 1

- To output element 2, you must first enter elements 1 and 2, meaning that the sequence of performing the operations is:

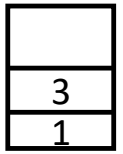
  SSU → 122



- To output item 4, after item 2 you must enter items 3,4, that is, the sequence of execution of operations in this case:

  SSU → 344

  SSUSSU



Stack:

Example 3:

If the stack input set is in order 1,2,3,4,5, which of the following outputs is correct according to the stack operation method?

A- 2 4 5 3 1

B- 4 2 3 1 5

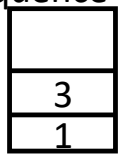
Solution:

A- 2 4 5 3 1

- To output item 5, after item 4 you must enter items 5, and then output items 5, that is, the sequence of execution of operations in this case:

  SU → 55

  SSUSSUSU



- According to the current state of the stack, elements 1 and 3 can be output sequentially, that is, the sequence of execution of operations is:

  UU → 31

  SSUSSUSUUU

So, such outputs can be obtained if the sequence of operations is in the final form while observe to the order of the inputs.

Stack:

Example 3:

If the stack input set is in order 1,2,3,4,5, which of the following outputs is correct according to the stack operation method?

A- 2 4 5 3 1

Step	Operation	Stack	Output
1-	PUSH(1)	1	
2-	PUSH(2)	12	
3-	POP	1	2
4-	PUSH(3)	13	2
5-	PUSH(4)	134	2
6-	POP(4)	13	24
7-	PUSH(5)	135	24
8-	POP(5)	13	245
9-	POP(3)	1	2453
10-	POP(1)		24531

Stack:

Example 3:

If the stack input set is in order 1,2,3,4,5, which of the following outputs is correct according to the stack operation method?

A- 2 4 5 3 1

B- 4 2 3 1 5

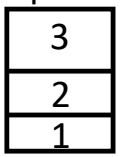
Solution:

Suppose that S means Stacking, which symbolizes the process of adding an element to the stack, and U means Unstacking, which symbolizes the process of deleting an element from the stack.

B- 4 2 3 1 5

- To output element 4, you must first enter elements 1,2,3 and 4, meaning that the sequence of performing the operations is:

  SSSSU → 1234



- To remove element 2 from the stack in its current state, element 3 must be removed before it, so this sequence of outputs 4, 2, 3, 1, 5 cannot be executed.

Stack:

Example 3:

If the stack input set is in order 1,2,3,4,5, which of the following outputs is correct according to the stack operation method?

B- 4 2 3 1 5

Solution:

Step	Operation	Stack	Output
1-	PUSH(1)	1	
2-	PUSH(2)	12	
3-	PUSH(3)	123	
4-	PUSH(4)	1234	
5-	POP(4)	123	4
6-	POP(3)	12	43
7-	POP(2)	1	432
8-	POP(1)		4321
9-	PUSH(5)	5	4321
10-	POP(5)		43215

Stack:

Example 3:

If the stack input set is in order 1,2,3,4,5, which of the following outputs is correct according to the stack operation method?

C- 4 5 1 2 3

D- 4 3 5 2 1

Solution:

Homework

Algebraic expression

- An algebraic expression is a legal combination of operands and the operators.
- Operand is the quantity (unit of data) on which a mathematical operation is performed.
- Operand may be a variable like x, y, z or a constant like 5, 4,0,9,1 .....
- Operator is a symbol which signifies a mathematical or logical operation between the operands.
- Example of familiar operators include +,-,*, /, ^
- Considering these definitions of operands and operators now we can write an example of expression as A*B+C.

Expression Parsing

The way to write arithmetic expression is known as a **notation**. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are:

1. Infix Notation
2. Prefix (Polish) Notation
3. Postfix (Reverse-Polish) Notation

1. Infix Notation:
We write expression in infix notation, e.g. a-b+c, where operators are used in-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

2. Prefix Notation
In this notation, operator is prefixed to operands, i.e. operator is written ahead of operands. For example, +ab. This is equivalent to its infix notation a+b. Prefix notation is also known as Polish Notation.

3. Postfix Notation
This notation style is known as Reversed Polish Notation. In this notation style, the operator is postfixed to the operands i.e., the operator is written after the operands. For example, ab+.

Expression Parsing

Arithmetic Expressions

1- Infix notation

An operator appears between its operands
Example : a + b
a + b * c + (d * e + f) * g

2- Prefix notation

An operator appears before its operands
Example : + a b
++a*bc*+*defg

3- Postfix notation

An operator appears after its operands
Example : a b +
abc*+de*f+g*+

Expression Parsing

Operator Precedence Associativity

1. **brackets { , [, (**
2. **Exponentiation ^ Highest Right Associative**
3. **Multiplication (*) & Division (/) Second Highest Left Associative**
4. **Addition (+) & Subtraction (−) Lowest Left Associative .**

Examples

Infix	Postfix	Prefix
A + B	AB +	+AB
(A + B) * (C + D)	AB+CD+*	*+AB+CD
A – B / (C * D ^ E)	ABCDE^*/-	-A/B*C^DE

Infix to Postfix (RPN) Algorithm

Scan the Infix from Left to Right, Output to RPN from Left to Right

1. Operator

✓ Pop and Output all higher priority operators from Stack

✓ Push the operator on Stack

2. "(": Push it on Stack
3. ")": Pop and Output all operators from Stack until "(" is Popped.
4. Operand: Output it
5. Finish: Pop and Output all Remainders.

Examples

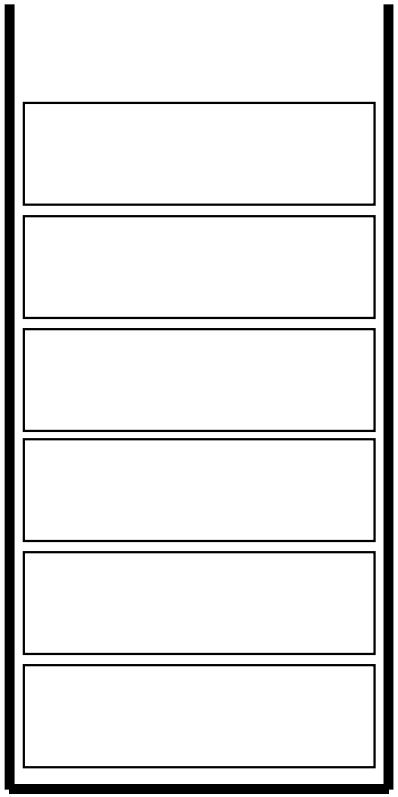
Infix to Postfix

•Suppose that we would like to rewrite

A+B*C in postfix

- ✓ A+B*C
- ✓ **A+(B*C)** Parentheses for emphasis
- ✓ **A+(BC*)** Convert the multiplication, Let D=BC*
- ✓ **A+D** Convert the addition
- ✓ **A(D)+** Replace the variable D with its imposed value
- ✓ **ABC*+ Postfix Form**

Examples

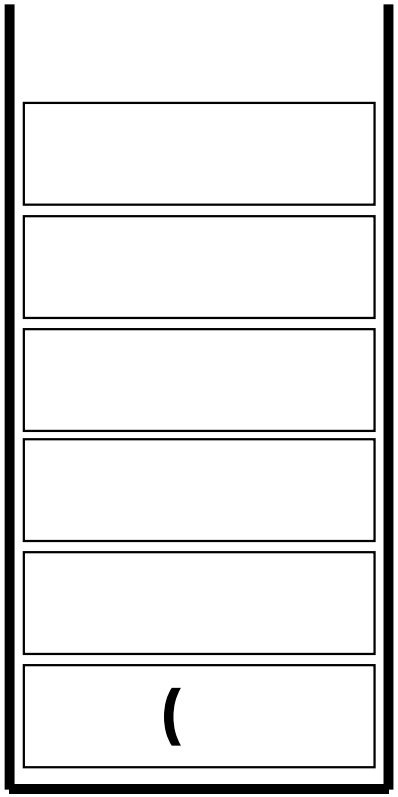


Infix expression

(a + b – c) * d – (e + f)

Postfix expression

Examples

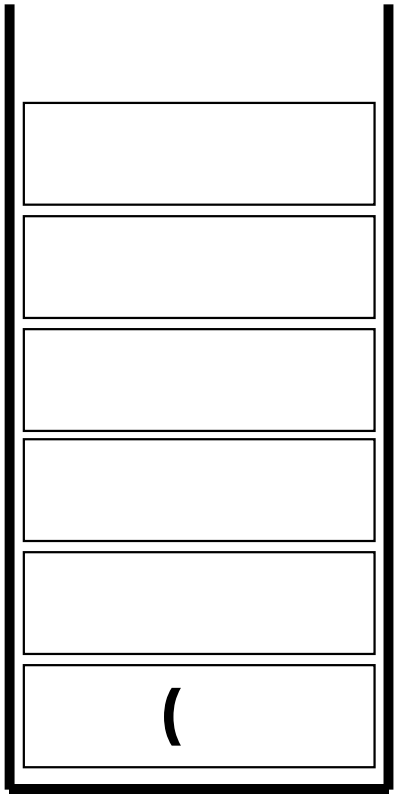


Infix expression

a + b – c) * d – (e + f)

Postfix expression

Examples



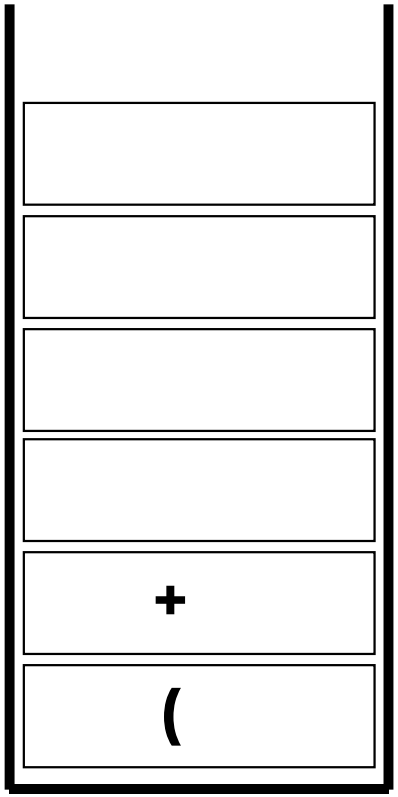
Infix expression

+ b – c) * d – (e + f)

Postfix expression

a

Examples



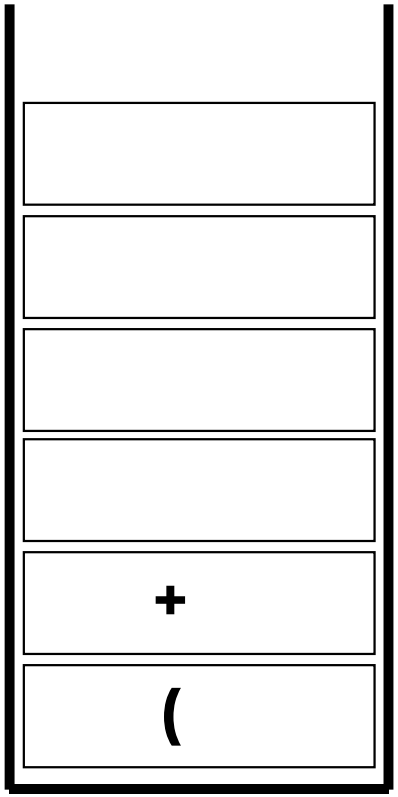
Infix expression

+ b – c) * d – (e + f)

Postfix expression

a

Examples



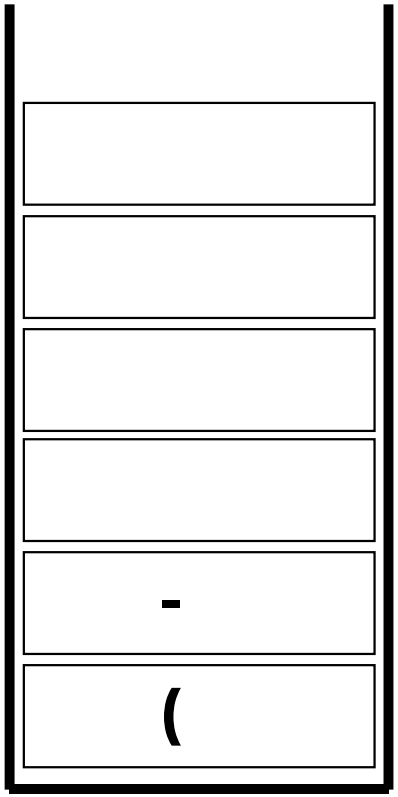
Infix expression

– c) * d – (e + f)

Postfix expression

a b

Examples



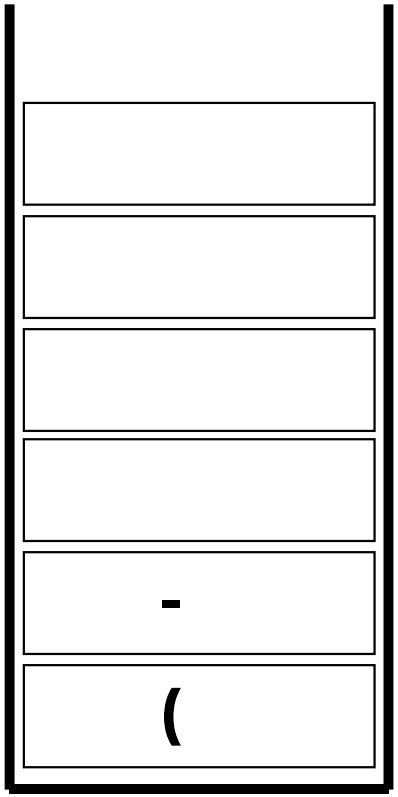
Infix expression

c) * d – (e + f)

Postfix expression

a b +

Examples



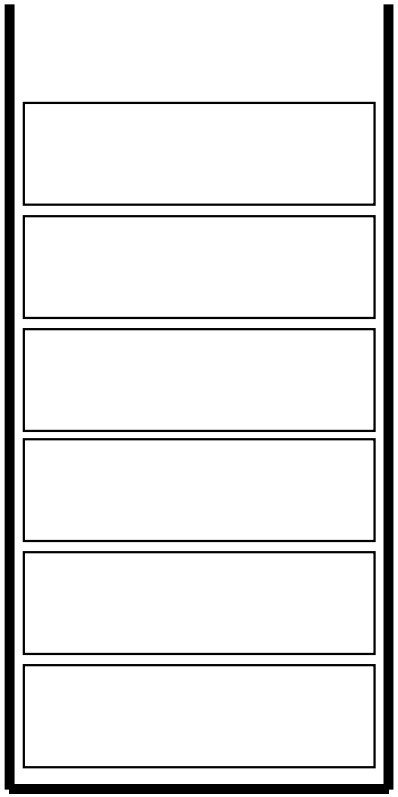
Infix expression

) * d – (e + f)

Postfix expression

a b + c

Examples



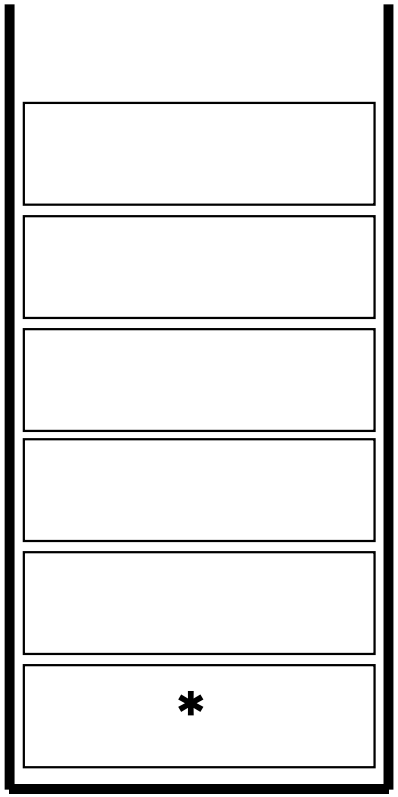
Infix expression

```
* d – ( e + f )
```

Postfix expression

```
a  b  +  c  -
```

Examples



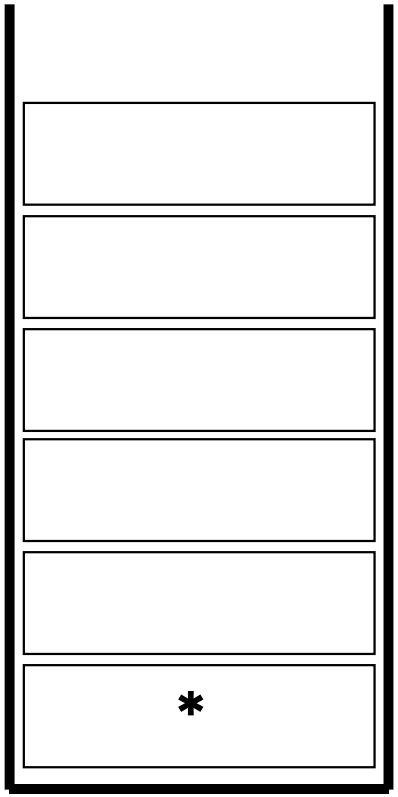
Infix expression

d – (e + f)

Postfix expression

a b + c -

Examples



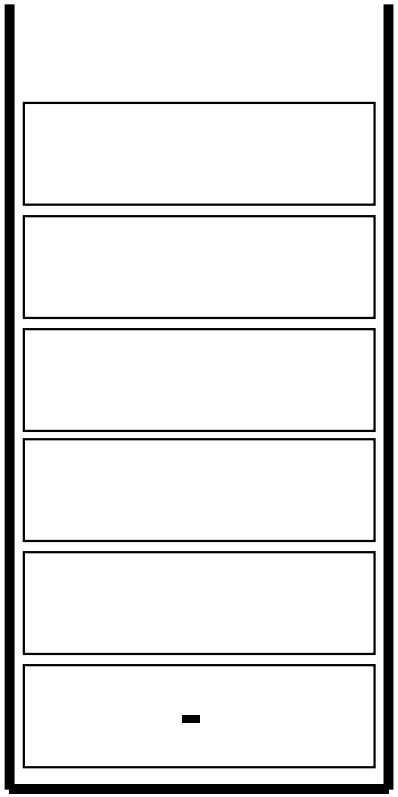
Infix expression

– (e + f)

Postfix expression

a b + c - d

Examples



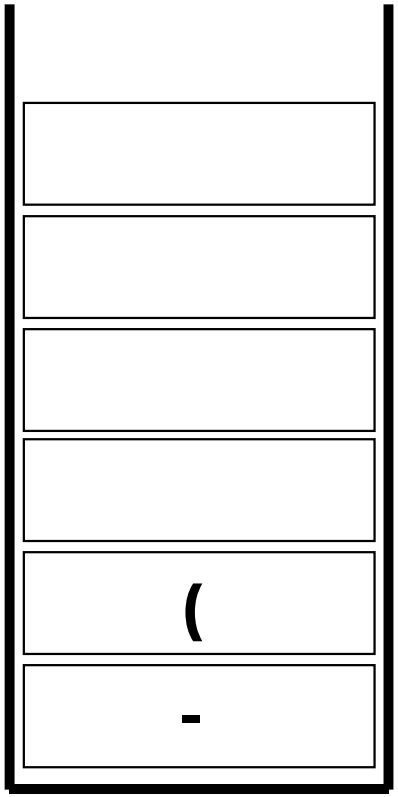
Infix expression

(e + f)

Postfix expression

a b + c - d *

Examples



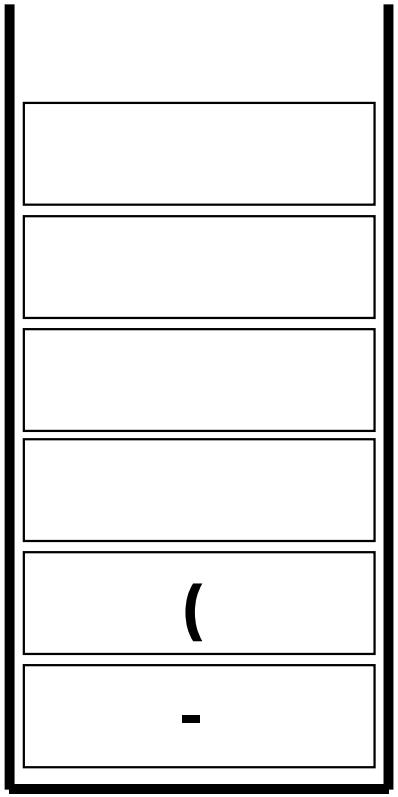
Infix expression

e + f)

Postfix expression

a b + c - d *

Examples



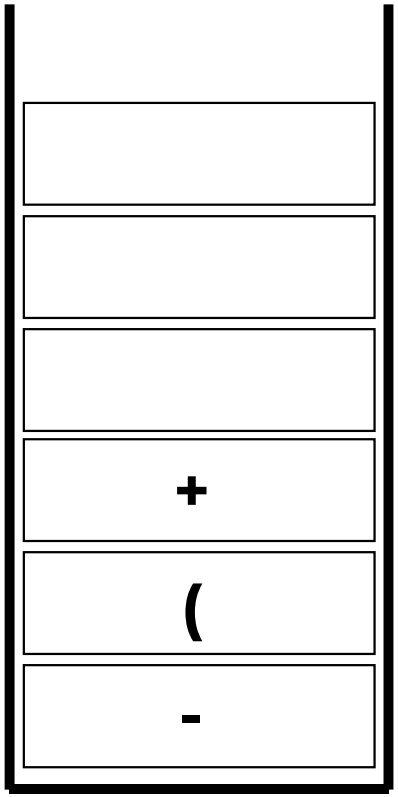
Infix expression

```
+ f )
```

Postfix expression

```
a b + c - d * e
```

Examples



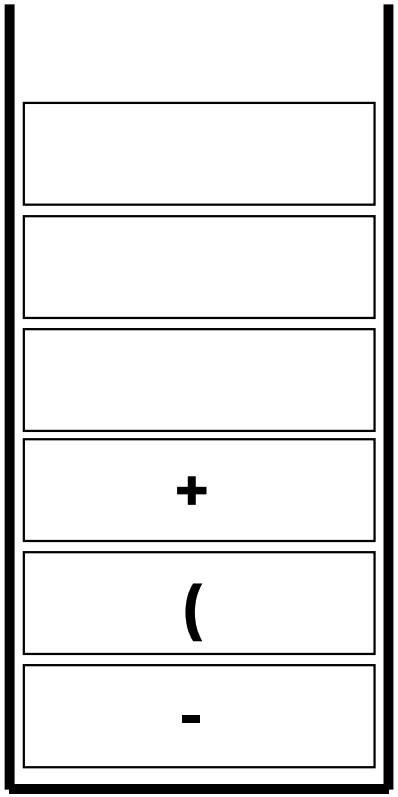
Infix expression

f)

Postfix expression

a b + c - d * e

Examples



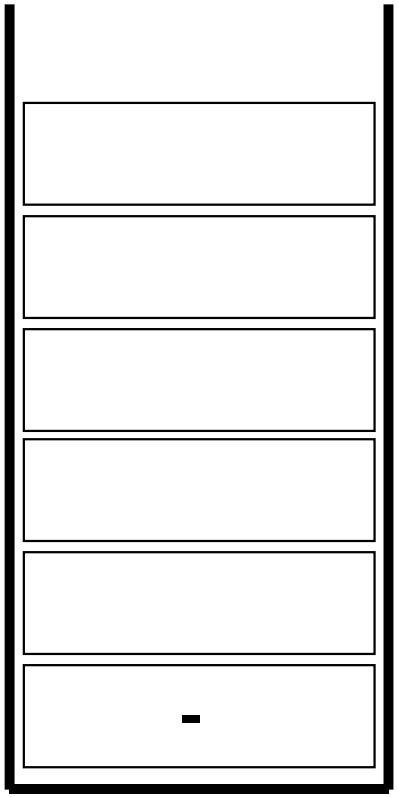
Infix expression

)

Postfix expression

a b + c - d * e f

Examples

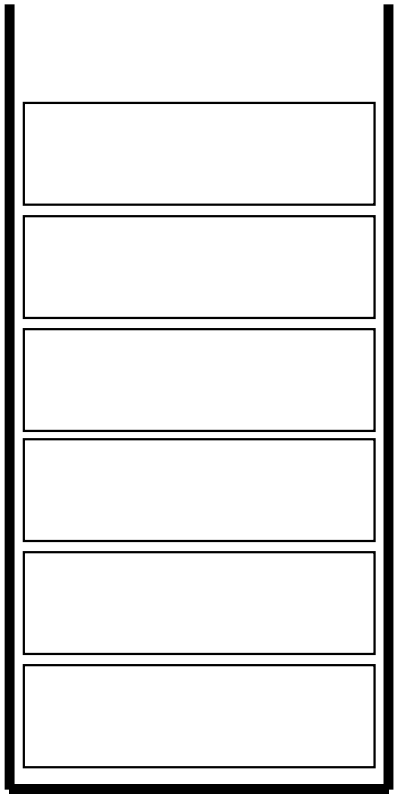


Infix expression

Postfix expression

a b + c - d * e f +

Examples



Infix expression

Postfix expression

a b + c - d * e f + -

Expression Parsing

Example:

Convert the following infix expression to the equivalent postfix notation.

1. A + B * C

Step	Input	Stack	Output
1-	A		A
2-	+	+	A
3-	B	+	AB
4-	*	+*	AB
5-	C	+*	ABC
6-			ABC*+

The postfix expression is: ABC*+

Expression Parsing

Example:

Convert the following infix expression to the equivalent postfix notation.

2. 3 + 4 * 5 / 6

Step	Input	Stack	Output
1-	3		3
2-	+	+	3
3-	4	+	34
4-	*	+*	34
5-	5	+*	345
6-	/	+/	345*
7-	6	+/	345*6
8-			345*6/+

The postfix expression is: 345*6/+

Expression Parsing

Example:

Convert the following infix expression to the equivalent postfix notation.

3. (30 + 23) * (43 -21) / (84 + 7)

Step	Input	Stack	Output
1-	(	(	
2-	30	(	30
3-	+	(+	30
4-	23	(+	30 23
5-	)		30 23 +
6-	*	*	30 23 +
7-	(	*(	30 23 +
8-	43	*(	30 23 + 43
9-	-	*(-	30 23 + 43
10-	21	*(-	30 23 + 43 21
11-	)	*	30 23 + 43 21 -
12-	/	/	30 23 + 43 21 - *

Expression Parsing

Example:

Convert the following infix expression to the equivalent postfix notation.

3. (30 + 23) * (43 -21) / (84 + 7)

Step	Input	Stack	Output
13-	(	/(	30 23 + 43 21 - *
14-	84	/(	30 23 + 43 21 - * 84
15-	+	/(+	30 23 + 43 21 - * 84
16-	7	/(+	30 23 + 43 21 - * 84 7
17-	)	/	30 23 + 43 21 - * 84 7 +
18-			30 23 + 43 21 - * 84 7+ /

The postfix expression is: 30 23 + 43 21 - * 84 7+ /

Expression Parsing

Example:

Convert the following infix expression to the equivalent postfix notation.

4. a – b * (c + d) / (e – f) ^ g * h

Step	Input	Stack1	Stack2
1-	A	A	
2-	-	A	-
3-	B	Ab	-
4-	*	Ab	-*
5-	(	Ab	-*(
6-	C	Abc	-*(
7-	+	Abc	-*(+
8-	D	Abcd	-*(+
9-	)	Abcd+	-*
10-	/	Abcd+*	-/
11-	(	Abcd+*	-/(
12-	E	Abcd+*e	-/(

Expression Parsing

Example:

Convert the following infix expression to the equivalent postfix notation.

4. a – b * (c + d) / (e – f) ^ g * h

Step	Input	Stack1	Stack2
13-	-	Abcd+*e	-/(-
14-	F	Abcd+*ef	-/(-
15-	)	Abcd+*ef-	-/
16-	^	Abcd+*ef-	-/^
17-	G	Abcd+*ef-g	-/^
18-	*	Abcd+*ef-g^/	-*
19-	H	Abcd+*ef-g^/h	-*
20-		Abcd+*ef-g^/h*-	……

Postfix to Infix Algorithm

Scan the RPN from **Left to Right**:

1. Operator:
    - Pop ExpR and ExpL from Stack
    - If prior(ExpR)<prior(Operator) then ExpR = "(" & ExpR & ")"
    - If prior(ExpL)<prior(Operator) then ExpL = "(" & ExpL & ")"
        - Push ExpL & Operator & ExpR on Stack
2. Operand: Push it on Stack
3. Finished: Pop and Output Infix

Postfix to Infix Example

Example of convert postfix = **AB+CD-/E+** to Infix = **(A+B)/(C-D)+E**

Step	Sym	Stack
1	A	A
2	B	A B
3	+	A+B
4	C	A+B C
5	D	A+B C D
6	-	A+B C-D
7	/	(A+B)/(C-D)
8	E	(A+B)/(C-D) E
9	+	(A+B)/(C-D)+E

CONVERSION OF INFIX TO PREFIX NOTATION

EXPRESSION :- A + B * C

scan

C	*	B	+	A

INFIX

Character(A) Scanned

Operator(*) scanned

Character (C) scanned

Push (*) into the stack

Pop(+) from the stack

Character (B) scanned

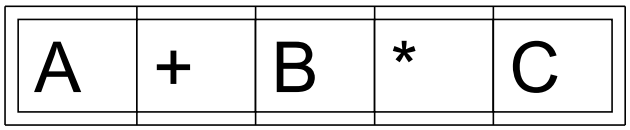
Priority of(*)is higher than (+),so(*) operator is poped from the stack

STACK

PREFIX

CONVERSION OF INFIX TO POSTFIX NOTATION

EXPRESSION :- A + B * C



INFIX

Character A scanned.

Operator (+) scanned

Character (B) scanned

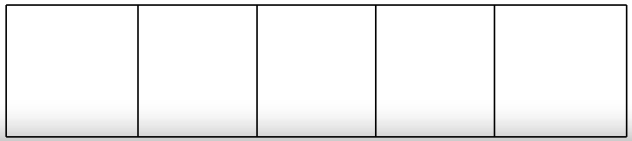
Priority of (*) is high, pop (*) from the stack

Operator (*) scanned

Character (c) Scanned

pop (+) from the stack

STACK



POSTFIX

CONVERSION OF PREFIX TO POSTFIX NOTATION

EXPRESSION :- + A * B C

scan

+	A	*	B	C

PREFIX

OPERATOR (+)SCANNED

CHARACTER(B) SCANNED

CHARACTER (C)SCANNED

CHARACTER (A) SCANNED

OPERATOR (*)SCANNED

Pop (+) from the stack

Priority of (*) is higher than (+) ,pop (*) from the stack

STACK

POSTFIX

CONVERSION OF POSTFIX TO INFIX NOTATION

EXPRESSION :- A B + C *

SCAN

A	B	+	C	*

POSTFIX

Operator(*) scanned, push to the stack

Pop (+) from the stack

Priority of (*) is higher than (+), so (+) is pop from the stack

Character (C) scanned

Operator(+) scanned

Character (B) scanned

Scanned character (B)

Push (+) to the stack

STACK

INFIX

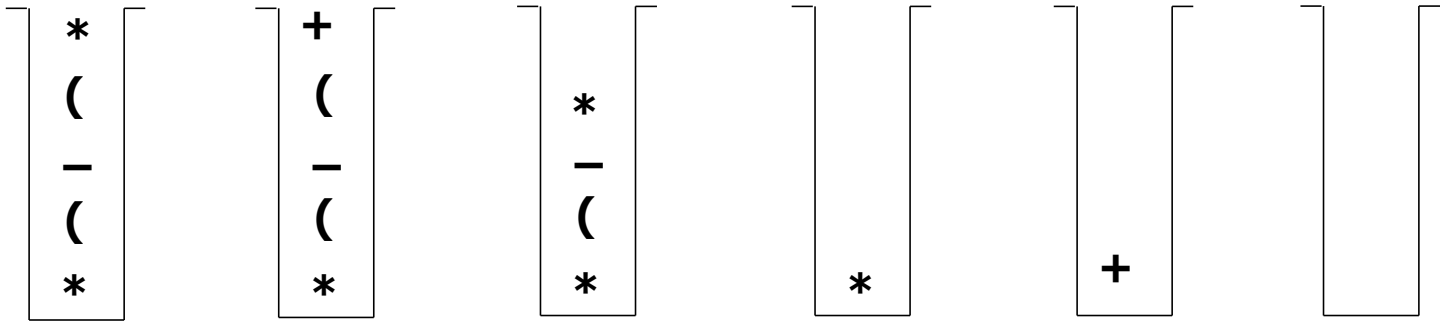
Converting Infix to Postfix with Stack

- Read expression from Left-to-Right and
  - if an operand is read copy it to the output,
  - if a left parenthesis is read push it into the stack,
  - when a right parenthesis is encountered, the operator at the top of the stack is popped off the stack and copied to the output ***until*** the symbol at the top of the stack is a left parenthesis. When that occurs, both parentheses are discarded,
  - if an operator is scanned and has a higher precedence than the operator at the top of the stack, the operator being scanned is pushed onto the stack,
  - while the precedence of the operator being scanned is lower than or equal to the precedence of the operator at the top of the stack, the operator at the top of the stack is popped and copied to the output,
  - when the end of the expression is reached on the input scan, the remaining operators in the stack are popped and copied to the output.

Example

Input: **4 * (2 – (6 * 3 + 4) * 2) + 1**

Output: **4 2 6 3 * 4 + 2 * – * 1 +**



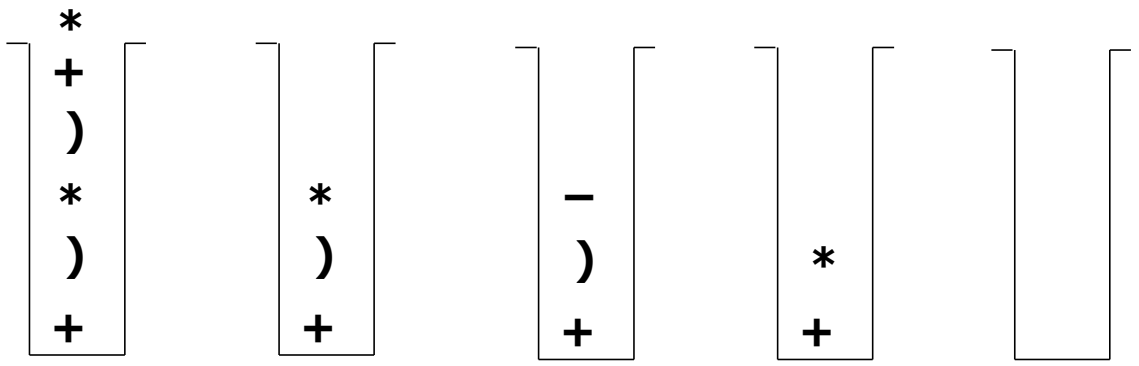
Converting Infix to Prefix with Stack

- Read expression from Right-to-Left and
  - if an operand is read copy it to the LEFT of the output,
  - if a right parenthesis is read push it into the stack,
  - when a left parenthesis is encountered, the operator at the top of the stack is popped off the stack and copied to the LEFT of the output ***until*** the symbol at the top of the stack is a right parenthesis. When that occurs, both parentheses are discarded,
  - if an operator is scanned and has a higher or equal precedence than the operator at the top of the stack, the operator being scanned is pushed onto the stack,
  - while the precedence of the operator being scanned is lower than to the precedence of the operator at the top of the stack, the operator at the top of the stack is popped and copied to the LEFT of the output,
  - when the end of the expression is reached on the input scan, the remaining operators in the stack are popped and copied to the LEFT of the output.

Example

Input: **4 * (2 – (6 * 3 + 4) * 2) + 1**

Output: **+ * 4 – 2 * + * 6 3 4 2 1**



Converting Infix to Prefix with Stack
2nd method

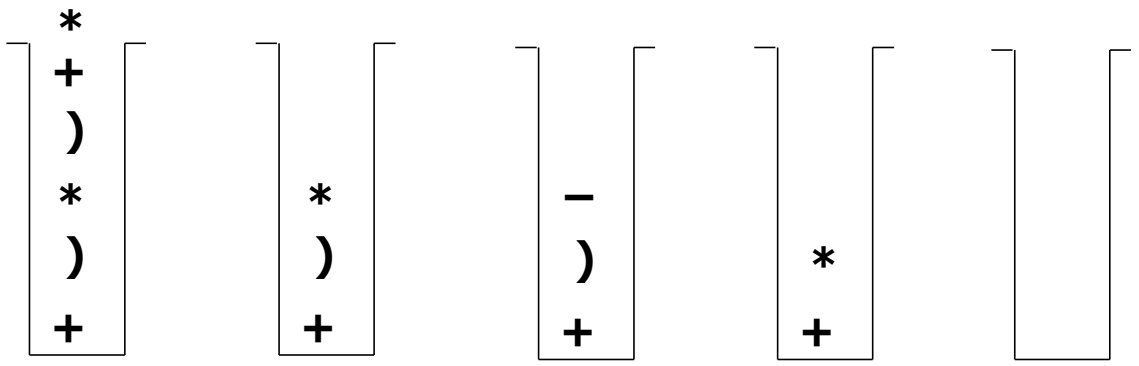
- Reverse the expression
- Read expression from Left-to-Right and
  - if an operand is read copy it to the output **(left-to-right)**,
  - if a right parenthesis is read push it into the stack,
  - when a left parenthesis is encountered, the operator at the top of the stack is popped off the stack and copied to the output ***until*** the symbol at the top of the stack is a right parenthesis. When that occurs, both parentheses are discarded,
  - if an operator is scanned and has a higher or equal precedence than the operator at the top of the stack, the operator being scanned is pushed onto the stack,
  - while the precedence of the operator being scanned is lower than to the precedence of the operator at the top of the stack, the operator at the top of the stack is popped and copied to the output,
  - when the end of the expression is reached on the input scan, the remaining operators in the stack are popped and copied to the output.
- Reverse the output

Example

Input: 4 * (2 – (6 * 3 + 4) * 2) + 1

Reverse: 1 +) 2 *) 4 + 3 * 6 (– 2 (* 4

Output: 1 2 4 3 6 * + * 2 – 4 * +



Reverse Output: + * 4 – 2 * + * 6 3 4 2 1

Exercises

- Using stack diagrams convert the following expressions into postfix and prefix forms of polish notation:

  a) $8 - 3 \times 4 + 2$

  b) $8 - 3 \times (4 + 2)$

  c) $(8 - 3) \times (4 + 2)$

  d) $(8 - 3) \times 4 + 2$

  e) $(-a + b) \times (c + a) - 5$

  f) $2 + ((-3 + 1) \times (4 - 2) + 3) \times 6 - (1 + 2 \times 3)$

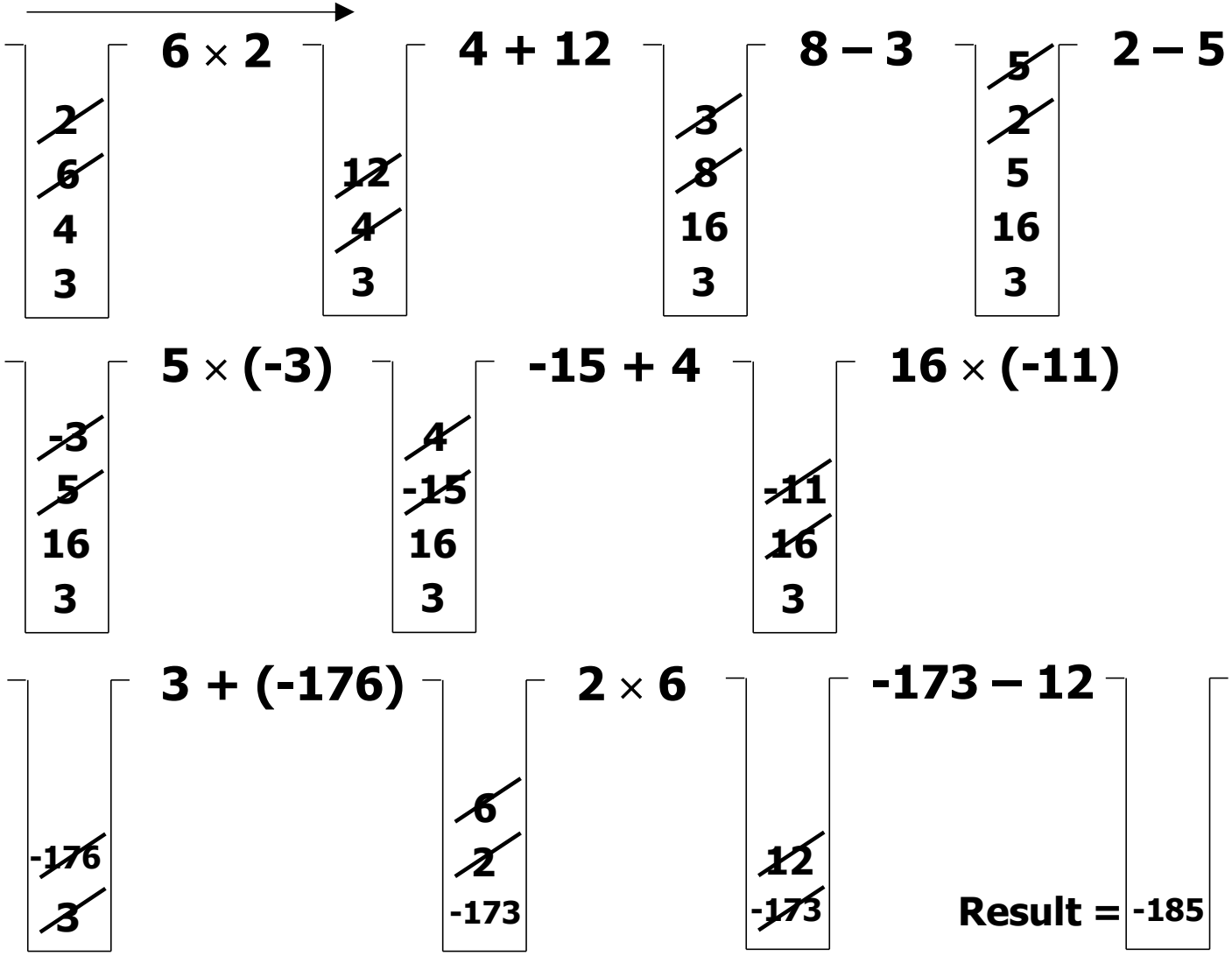
  g) $(5 > 4)$ and not $(3 = 2 - 1)$

Evaluation of Reverse Polish Expressions

- Most compilers use the polish form to translate expressions into machine language.

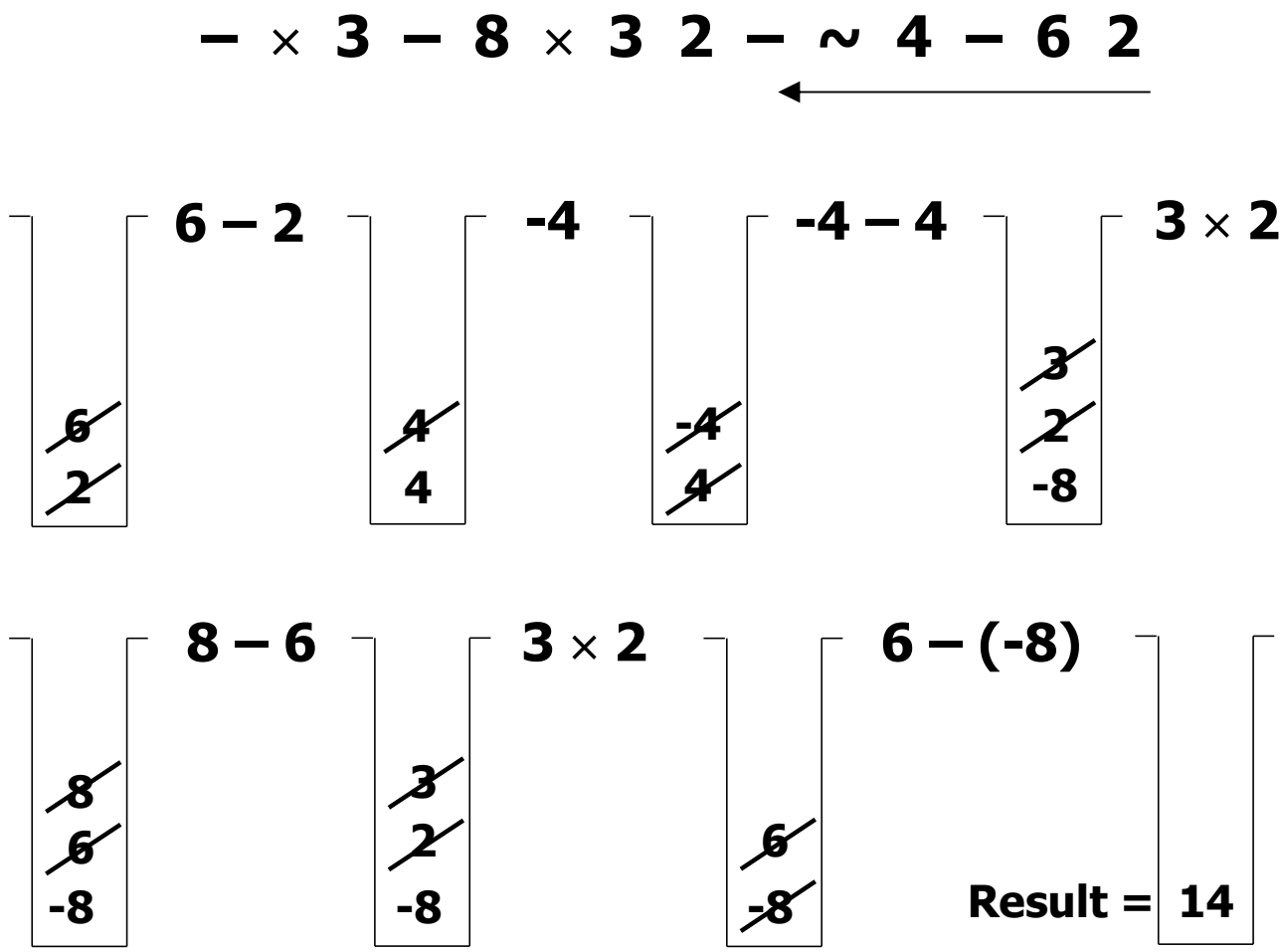
- Evaluation is done using a **stack** data-structure
  - Read expression from left to right and build the stack of numbers (operands).
  - When an operator is read two operands are **popped** out of the stack they are evaluated with the operator and the result is **pushed** into the stack.
  - At the end of the expression there must be only one operand into the stack (the solution) otherwise ERROR.

3 4 6 2 × + 8 3 − 2 5 − × 4 + × + 2 6 × −



Evaluation of Polish Expressions

- Evaluation is done using a **stack** data-structure
  - Read expression from right to left and build the stack of numbers (operands).
  - When an operator is read two operands are **popped** out of the stack they are evaluated with the operator and the result is **pushed** into the stack.
  - At the end of the expression there must be only one operand into the stack (the solution) otherwise ERROR.


− × 3 − 8 × 3 2 − ~ 4 − 6 2
6 − 2
6
2
-4
4
4
-4 − 4
-4
4
3 × 2
3
2
-8
8 − 6
8
6
-8
3 × 2
3
2
-8
6 − (-8)
6
-8
Result = 14

Thank You
&
Good luck