Syrian Private University

Algorithms & Data Structure I

Instructor: Dr. Mouhib Alnoukari

4 Stacks

Stacks

- A stack is an ordered collection of homogeneous data element where the insertion and deletion operations take place at one end only.
- A stack is a last in, first out (LIFO) data structure
  - Items are removed from a stack in the reverse order from the way they were inserted

Abstract Stack

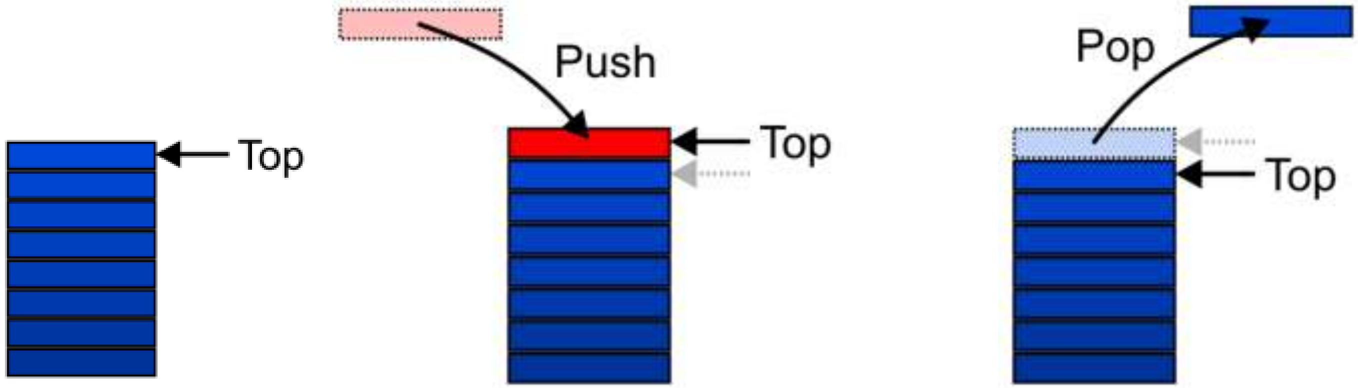
An Abstract Stack is an abstract data type which emphasizes specific operations:

- Uses a explicit linear ordering
- Insertions and removals are performed individually
- Inserted objects are *pushed onto* the stack
- The *top* of the stack is the most recently object pushed onto the stack
- When an object is *popped* from the stack, the current *top* is erased

Abstract Stack

Also called a *last-in–first-out* (LIFO) behaviour

– Graphically, we may view these operations as follows:



There are two exceptions associated with abstract stacks:

– It is an undefined operation to call either pop or top on an empty stack

Applications

Numerous applications:

- Parsing code:
  - Matching parenthesis
  - XML (e.g., XHTML)
- Tracking function calls
- Dealing with undo/redo operations
- Reverse-Polish calculators

The stack is a very simple data structure

- Given any problem, if it is possible to use a stack, this significantly simplifies the solution.

Stack: Applications

Problem solving:

– Solving one problem may lead to subsequent problems.
– These problems may result in further problems.
– As problems are solved, your focus shifts back to the problem which lead to the solved problem.

Notice that function calls behave similarly:

– A function is a collection of code which solves a problem.

Implementations

We will look at two implementations of stacks:

The optimal asymptotic run time of any algorithm is $\Theta(1)$:

- The run time of the algorithm is independent of the number of objects being stored in the container
- We will always attempt to achieve this lower bound

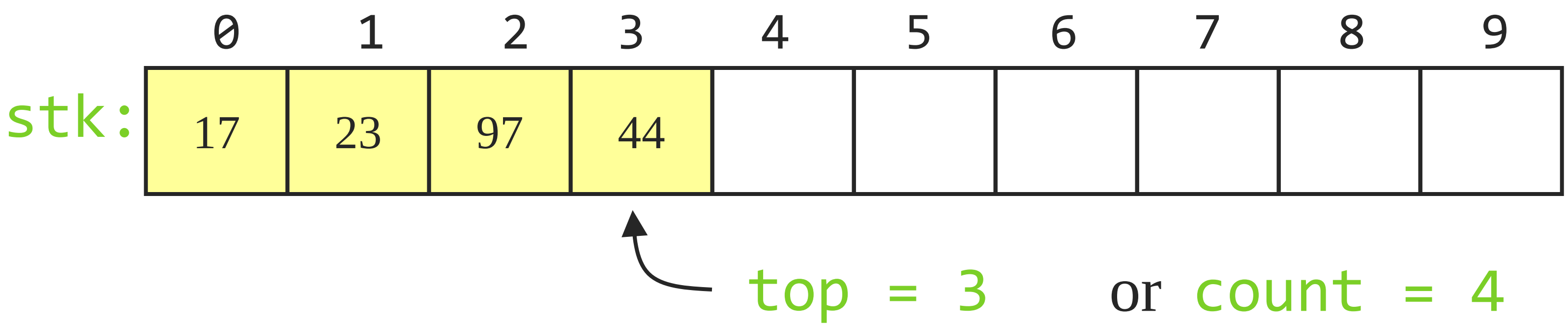
We will look at:

- One-ended arrays
- Singly linked lists

Array implementation of stacks

- First, we have to allocate a memory block of sufficient size to accommodate the full capacity of the Stack.
- To implement a stack, items are inserted and removed at the same end (called the top)
- Efficient array implementation requires that the top of the stack be towards the center of the array, not fixed at one end
- To use an array to implement a stack, you need both the array itself and an integer
  - The integer tells you either:
    - Which location is currently the top of the stack, or
    - How many elements are in the stack

Pushing and popping



- If the bottom of the stack is at location `0`, then an empty stack is represented by `top = -1` or `count = 0`
- To add (push) an element, either:
  - Increment `top` and store the element in `stk[top]`, or
  - Store the element in `stk[count]` and increment `count`
- To remove (pop) an element, either:
  - Get the element from `stk[top]` and decrement `top`, or
  - Decrement `count` and get the element in `stk[count]`

After popping

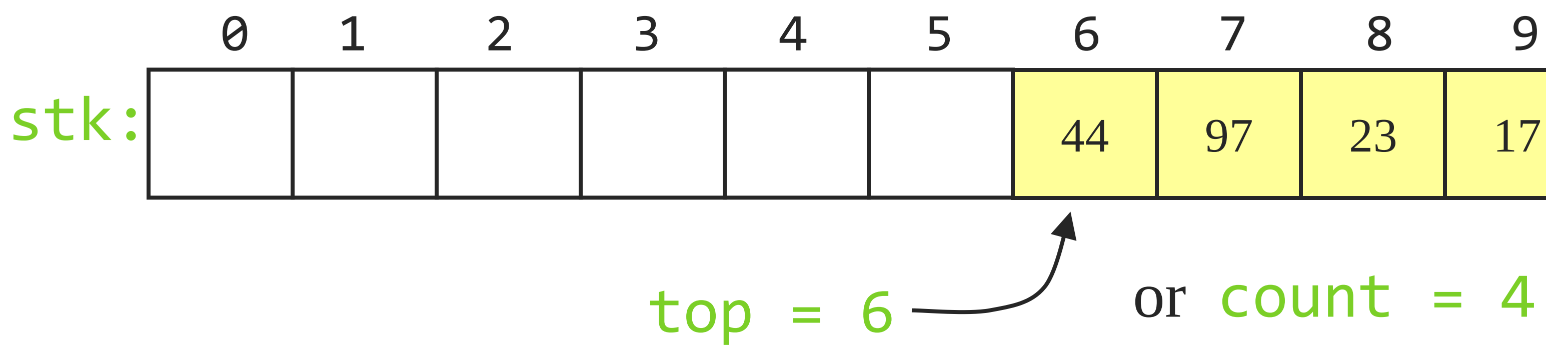
	0	1	2	3	4	5	6	7	8	9
`stk:`	17	23	97	44						

`top = 2` or `count = 3`

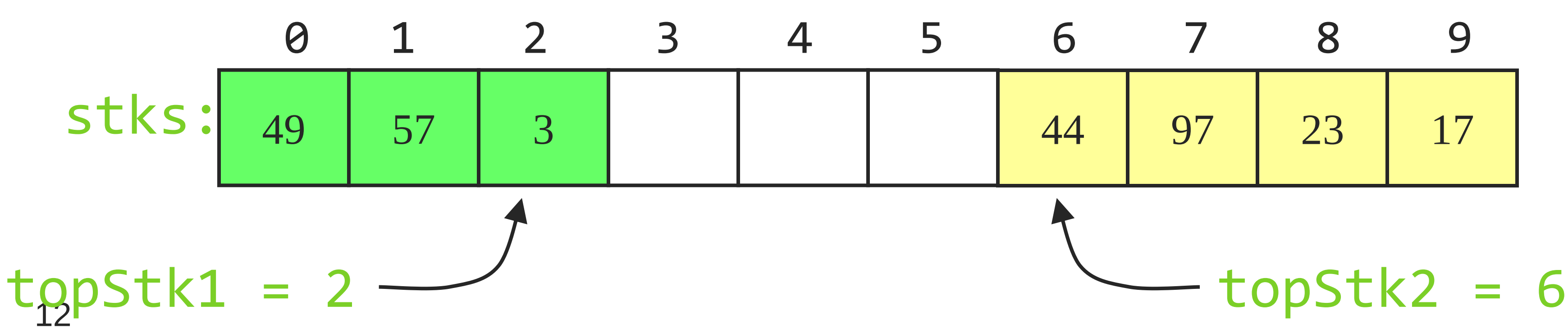
- When you pop an element, do you just leave the “deleted” element sitting in the array?
- The surprising answer is, *“it depends”*
  - If this is an array of primitives, *or* if you are programming in C or C++, *then* doing anything more is just a waste of time
  - If you are programming in Java, and the array contains objects, you should set the “deleted” array element to `null`
  - Why? To allow it to be garbage collected!

Sharing space

- Of course, the bottom of the stack could be at the *other* end.



- Sometimes this is done to allow two stacks to share the *same storage area.*



Stack Operations Implementation

```
STACK-EMPTY(S)
if S.top == 0
        return TRUE
else return FALSE
```

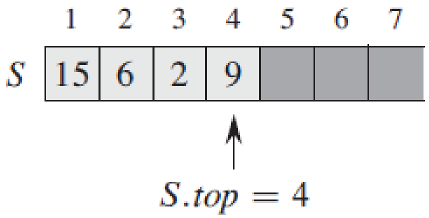
Stack Operations Implementation

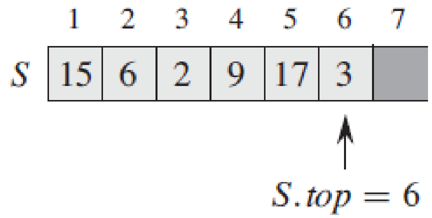
PUSH(S, x)

$S.top = S.top + 1$

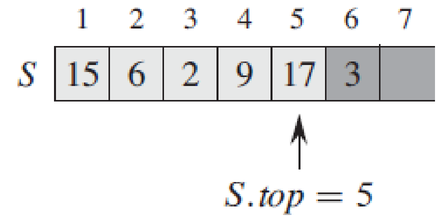
$S[S.top] = x$

Stack Operations Implementation





Push (S, 17)
Push (S, 3)



Pop (S)

Array Implementation

For one-ended arrays, all operations at the back are $\Theta(1)$

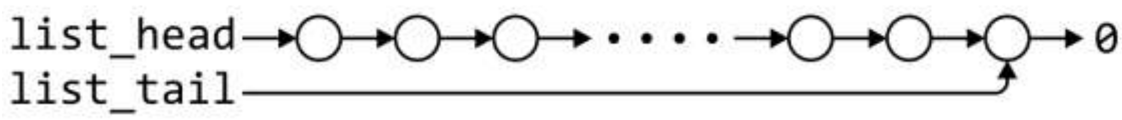
	Front/1st	Back/nth
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(n)$	$\Theta(1)$
Erase	$\Theta(n)$	$\Theta(1)$

Error checking

- There are two stack errors that can occur:
  - Underflow: trying to pop (or peek at) an empty stack.
  - Overflow: trying to push onto an already full stack.
- For underflow, you should throw an exception
  - If you don't catch it yourself, Java will throw an `ArrayIndexOutOfBounds` exception.
  - You could create your own, more informative exception.
- For overflow, you could do the same things
  - Or, you could check for the problem, and copy everything into a new, larger array.

Linked-List Implementation

Operations at the front of a singly linked list are all $\Theta(1)$

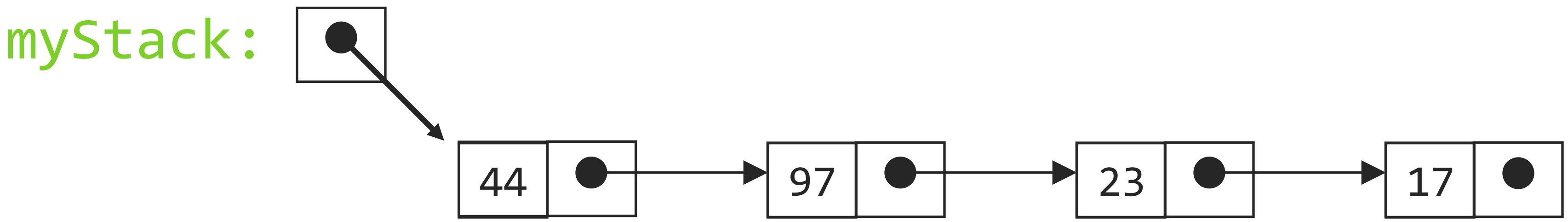


	Front/1st	Back/nth
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(n)$

The desired behavior of an Abstract Stack may be reproduced by performing all operations at the front.

Linked-list implementation of stacks

- Since all the action happens at the top of a stack, a singly-linked list (SLL) is a fine way to implement it.
- The header of the list points to the top of the stack.



- Pushing is inserting an element at the front of the list.
- Popping is removing an element from the front of the list.

Linked-list implementation details

- With a linked-list representation, overflow will not happen (unless you exhaust memory, which is another kind of problem)
- Underflow can happen, and should be handled the same way as for an array implementation
- When a node is popped from a list, and the node references an object, the reference (the pointer in the node) does *not* need to be set to `null`
  - Unlike an array implementation, it really *is* removed--you can no longer get to it from the linked list
  - Hence, garbage collection can occur as appropriate

Function Calls

- you write a function to solve a problem.
- the function may require sub-problems to be solved, hence, it may call another function.
- once a function is finished, it returns to the function which called it.

Function Calls

You will notice that when a function returns, execution and the return value is passed back to the last function which was called.

Today’s CPUs have hardware specifically designed to facilitate function calling.

Reverse-Polish Notation

Normally, mathematics is written using what we call *in-fix* notation:

$$(3 + 4) \times 5 - 6$$

The operator is placed between to operands

One weakness: parentheses are required

$$\begin{aligned} (3 + 4) \times 5 - 6 &= 29 \\ 3 + 4 \times 5 - 6 &= 17 \\ 3 + 4 \times (5 - 6) &= -1 \\ (3 + 4) \times (5 - 6) &= -7 \end{aligned}$$

Reverse-Polish Notation

Alternatively, we can place the operands first, followed by the operator:

$$(3 + 4) \times 5 - 6$$

$$3 \;\; 4 \;\; + \;\; 5 \;\; \times \;\; 6 \;\; -$$

Parsing reads left-to-right and performs any operation on the last two operands:

$$3 \;\; 4 \;\; + \;\; 5 \;\; \times \;\; 6 \;\; -$$

$$7 \quad 5 \;\; \times \;\; 6 \;\; -$$

$$35 \quad 6 \;\; -$$

$$29$$

Reverse-Polish Notation

Other examples:

$3\ 4\ 5\ \times\ +\ 6\ -$

$3\ \ 20\ \ +\ 6\ -$

$23\ \ 6\ -$

17

$3\ 4\ 5\ 6\ -\ \times\ +$

$3\ 4\ {-1}\ \times\ +$

$3\ \ {-4}\ \ +$

-1

Reverse-Polish Notation

Benefits:

– No ambiguity and no brackets are required.

– It is the same process used by a computer to perform computations:

  - operands must be loaded into registers before operations can be performed on them.

– Reverse-Polish can be processed using stacks.

Reverse-Polish Notation

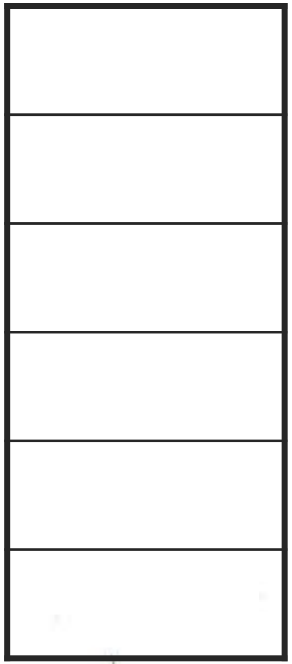
The easiest way to parse reverse-Polish notation is to use an operand stack:

- operands are processed by pushing them onto the stack.
- when processing an operator:
  - pop the last two items off the operand stack,
  - perform the operation, and
  - push the result back onto the stack

Reverse-Polish Notation

Evaluate the following reverse-Polish expression using a stack:

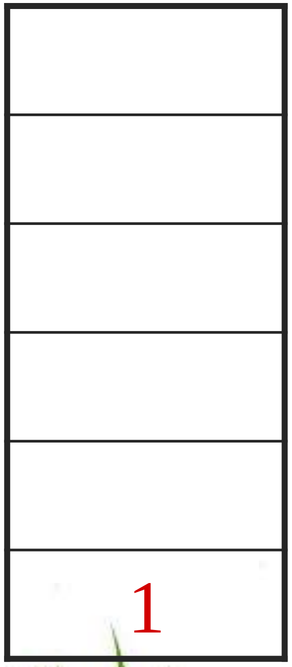
$$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$$



Reverse-Polish Notation

Push 1 onto the stack

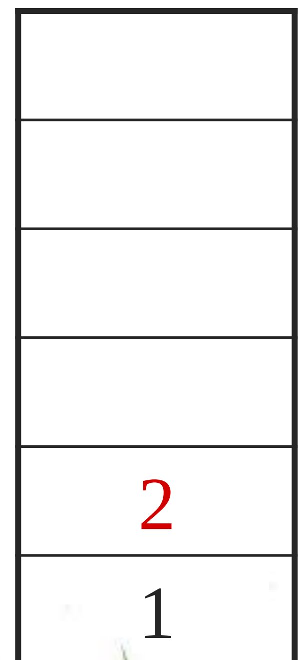
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

Push 1 onto the stack

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

Push 3 onto the stack

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +

3
2
1

Reverse-Polish Notation

Pop 3 and 2 and push 2 + 3 = 5

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +

5
1

Reverse-Polish Notation

Push 4 onto the stack

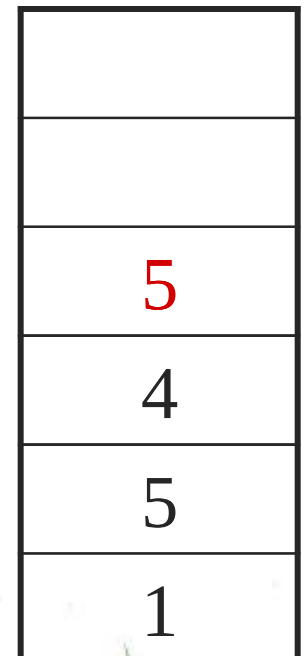
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +

4
5
1

Reverse-Polish Notation

Push 5 onto the stack

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

Push 6 onto the stack

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +

6
5
4
5
1

Reverse-Polish Notation

Pop 6 and 5 and push 5 × 6 = 30

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +

30
4
5
1

Reverse-Polish Notation

Pop 30 and 4 and push $4 - 30 = -26$

$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$

–26
5
1

Reverse-Polish Notation

Push 7 onto the stack

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +

7
–26
5
1

Reverse-Polish Notation

Pop 7 and –26 and push –26 × 7 = –182

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +

–182
5
1

Reverse-Polish Notation

Pop -182 and 5 and push $-182 + 5 = -177$

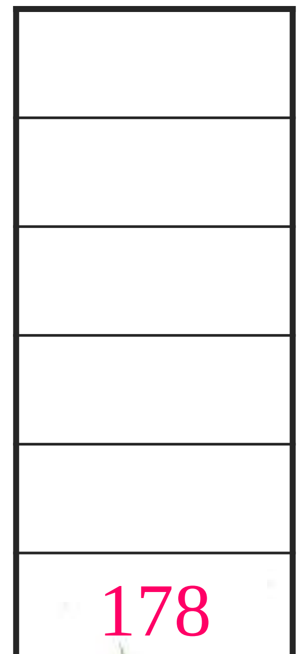
$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$

-177
1

Reverse-Polish Notation

Pop –177 and 1 and push 1 – (–177) = 178

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

Push 8 onto the stack

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +

8
178

Reverse-Polish Notation

Push 1 onto the stack

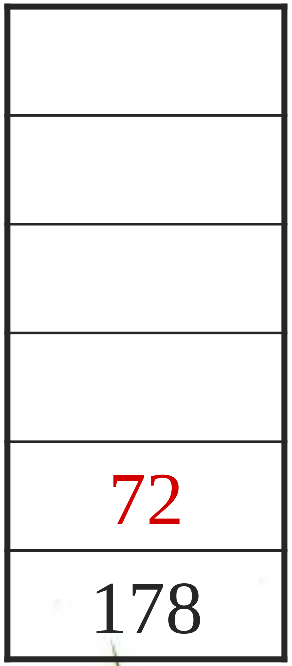
$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$

9
8
178

Reverse-Polish Notation

Pop 9 and 8 and push 8 × 9 = 72

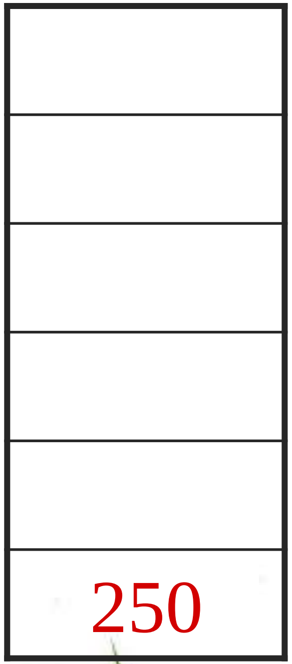
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

Pop 72 and 178 and push 178 + 72 = 250

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

Thus:

$$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$$

evaluates to the value on the top: 250

The equivalent in-fix notation is:

$$((1 - ((2 + 3) + ((4 - (5 \times 6)) \times 7))) + (8 \times 9))$$

We reduce the parentheses using order-of-operations:

$$1 - (2 + 3 + (4 - 5 \times 6) \times 7) + 8 \times 9$$

Reverse-Polish Notation

Incidentally,

$$1 - 2 + 3 + 4 - 5 \times 6 \times 7 + 8 \times 9 = -132$$

which has the reverse-Polish notation of

$$1\ 2\ -\ 3\ +\ 4\ +\ 5\ 6\ 7\ \times\ \times\ -\ 8\ 9\ \times\ +$$

For comparison, the calculated expression was:

$$1\ 2\ 3\ +\ 4\ 5\ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ 9\ \times\ +$$