BML MUNJAL UNIVERSITY™

Data Structures & Algorithms

Stack

Stack

A stack is a linear collection of similar items, where an item to be added to the stack must be placed on top of the stack and items that are removed from the stack must be removed from the top (i.e. elements can be added and removed from the stack only at the top).

A stack is a data container where data elements are inserted and removed according to the Last-In-First-Out (LIFO) principle.

Real life example
Stack of tray in cafeteria - When you go to a cafeteria to get lunch you take a tray from the stack of trays. You must always take the tray that is on the top of the stack. The cafeteria personnel put clean trays on the stack. The clean trays are always added to the top of the stack, so the tray that's on top is the last one that was added.

Stack

Operations
Two main operations are
push the item into the stack, and
pop the item out of the stack.
push adds an item to the top of the stack, **pop** removes the item from the top.

Stack

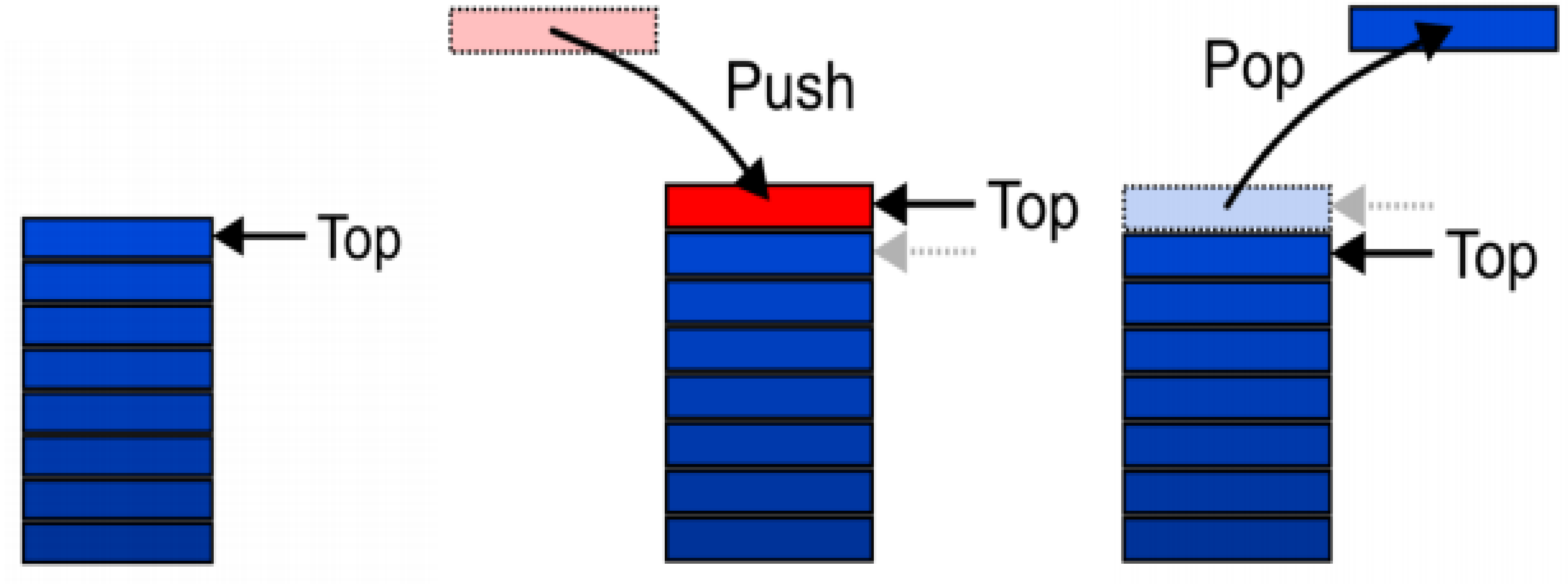


Figure The top, push, and pop operations on a stack.

Stack

Implementations

Array based implementation
Singly Linked List based implementation

Array Implementation of Stack

```
top=-1   // Initialization

push(S[], top, size, item)
{
        if (top == size-1) {
           print(“Stack overflow");
           return;
        }
        top++;
        S[top] = item;
 }
```

Time Complexity : O(1)

Array Implementation of Stack

```
pop(S[], top)
{
    if ( top == -1) {
         print(“Stack underflow");
         return;
    }
    top--;
    return S[top+1];
}
```

Time Complexity : O(1)

Linked list Implementation of Stack

Push
Insert in the beginning of a singly linked list

Pop
Deletion from the beginning of a singly linked list

```
typedef struct node
{
  int data;
  struct node *next;
} stackNode;
struct stackNode *top;
top=NULL;
```

Stack

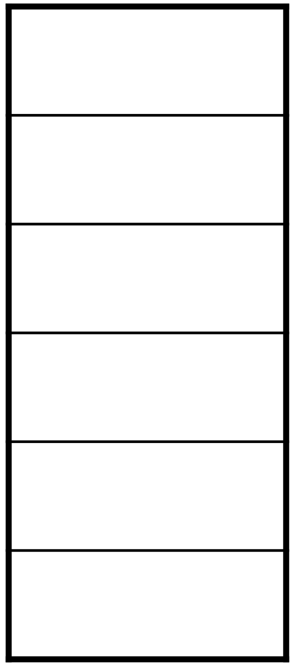
Applications

During function calls,
During recursive function calls,
Infix to postfix expression conversion,
Postfix expression evaluation,
Checking balanced parentheses in an arithmetic expression

Evaluating expressions in postfix notation

Evaluate the following postfix expression using a stack:

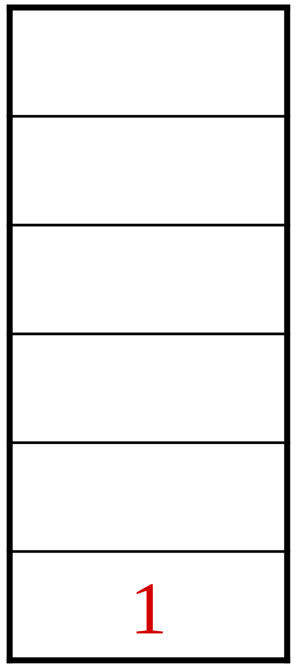
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Evaluating expressions in postfix notation

Push 1 onto the stack

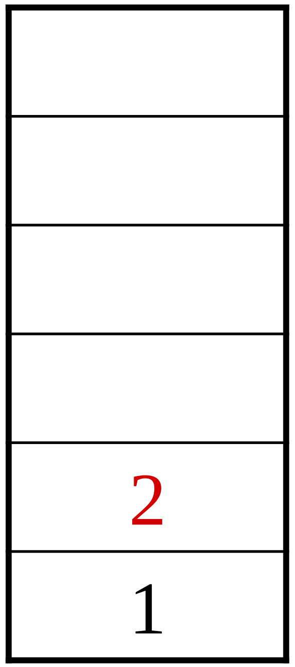
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Evaluating expressions in postfix notation

Push 1 onto the stack

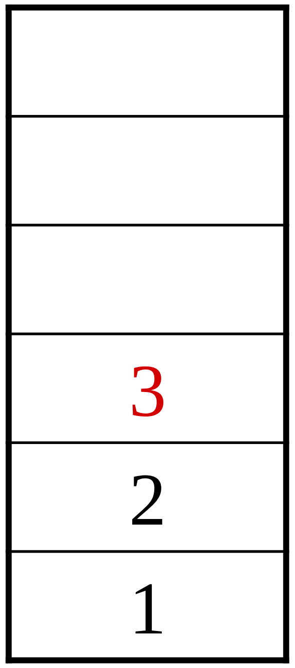
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Evaluating expressions in postfix notation

Push 3 onto the stack

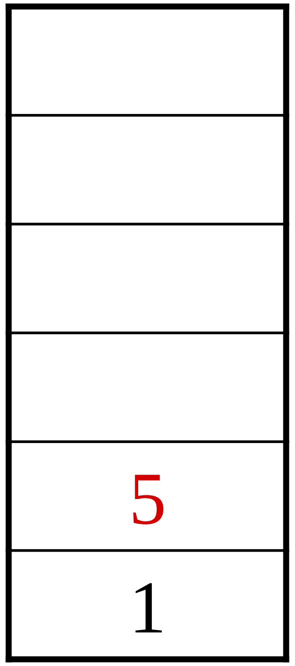
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Evaluating expressions in postfix notation

Pop 3 and 2 and push 2 + 3 = 5

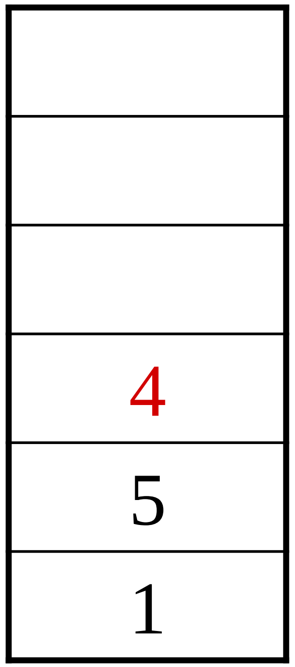
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Evaluating expressions in postfix notation

Push 4 onto the stack

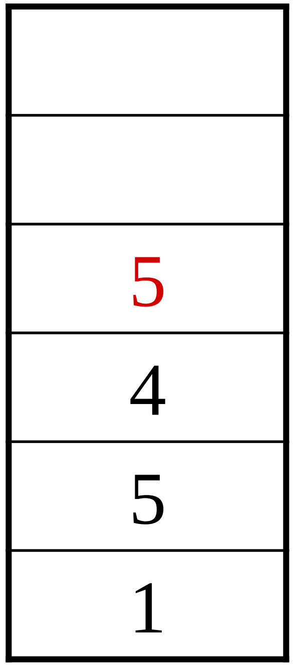
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Evaluating expressions in postfix notation

Push 5 onto the stack

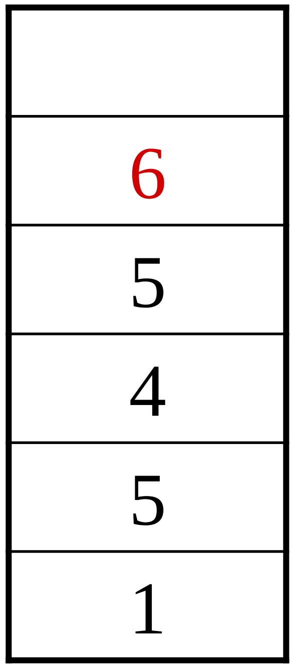
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Evaluating expressions in postfix notation

Push 6 onto the stack

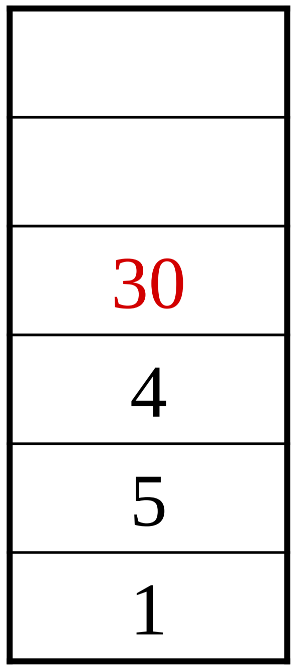
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Evaluating expressions in postfix notation

Pop 6 and 5 and push 5 × 6 = 30

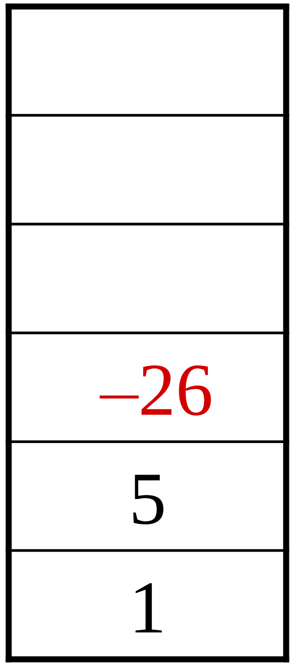
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Evaluating expressions in postfix notation

Pop 30 and 4 and push 4 – 30 = –26

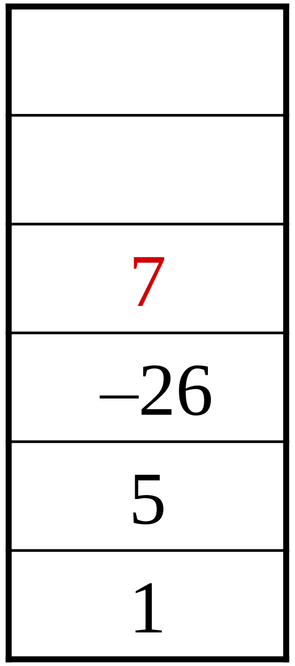
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Evaluating expressions in postfix notation

Push 7 onto the stack

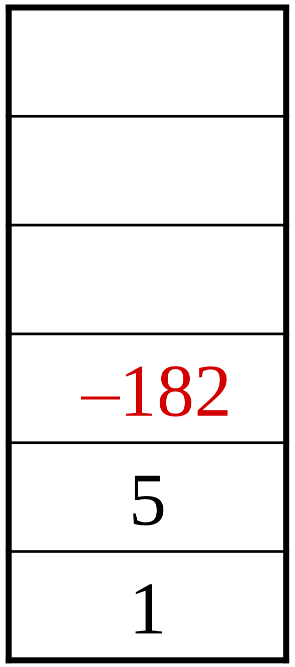
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Evaluating expressions in postfix notation

Pop 7 and –26 and push –26 × 7 = –182

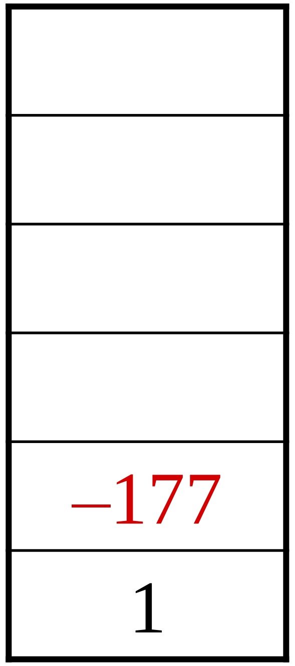
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Evaluating expressions in postfix notation

Pop –182 and 5 and push –182 + 5 = –177

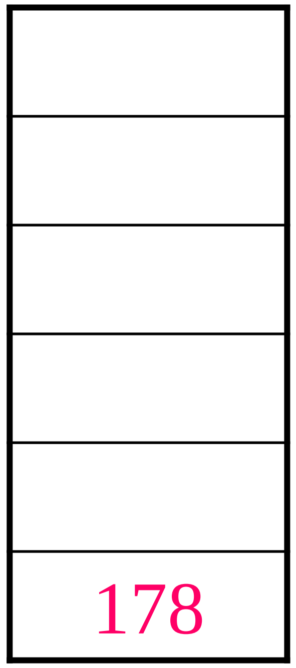
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Evaluating expressions in postfix notation

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

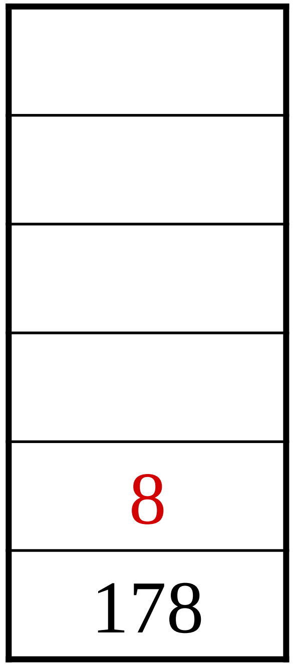
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Evaluating expressions in postfix notation

Push 8 onto the stack

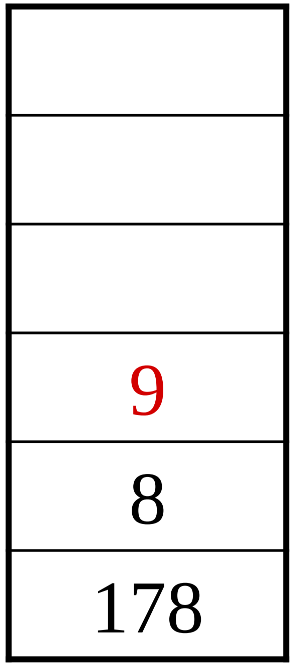
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Evaluating expressions in postfix notation

Push 1 onto the stack

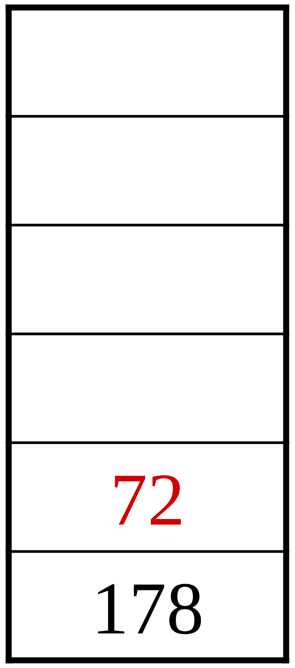
1 2 3 + 4 5 6 × − 7 × + − 8 9 × +



Evaluating expressions in postfix notation

Pop 9 and 8 and push 8 × 9 = 72

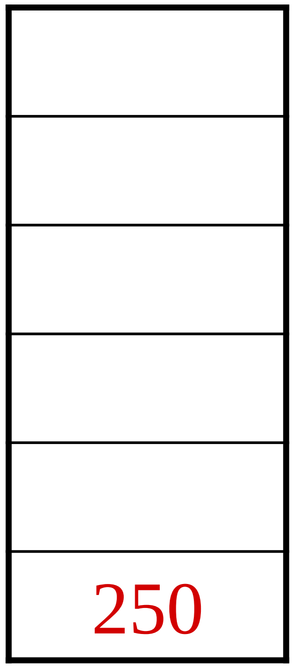
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Evaluating expressions in postfix notation

Pop 72 and 178 and push 178 + 72 = 250

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Stack

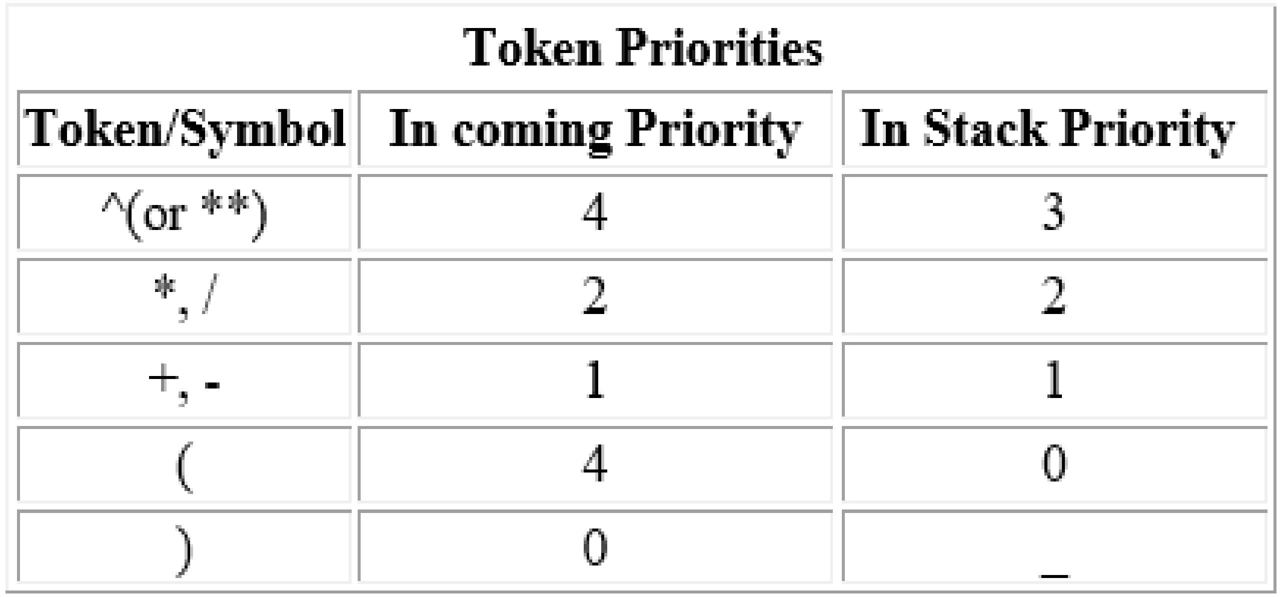
Postfix_evaluation

```
Step 1. Start with an empty stack
Step 2. Read through the postfix expression from left to right, scan one symbol at a time
          and if you encounter
     2a. an operand, push it onto the stack
     2b. an operator <op>, pop top two elements from the stack, say a and b,
          evaluate b <op>  a and push the result back onto the stack.
Step 3. Continue until you get to the end of the expression. At the end, the stack should
contain exactly one value: the solution.
```

Stack

Infix to postfix expression conversion

Conversion takes place as per the operator’s priority table



Token Priorities

Token/Symbol	In coming Priority	In Stack Priority
^(or **)	4	3
*, /	2	2
+, -	1	1
(	4	0
)	0	_

Operators are taken out of the stack as long as their in-stack priority is greater than or equal to the incoming priority of the new operator.

Stack

Start with an initially empty stack.
Read the given infix expression, one symbol/token at a time from left to right and perform the following operations :

1. If the read symbol is an operand, then add the operand to the postfix string(output).
2. If the read symbol is an operator and stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
3. If the incoming symbol is a left parenthesis, push it on the stack.
4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis, which is popped but not output. Discard the pair of parentheses.
5. If the incoming symbol has higher precedence than the symbol on the top of the stack, push it on the stack.
6. If the incoming symbol has lower or equal precedence than the stack top symbol then pop all the symbols inside the stack(all higher priority operators), and add them to the postfix string(output) and push the incoming symbol on the top of the stack.
7. If reading of all symbols are completed, and stack is not empty pop and print all operators on the stack and add these operator(s) to the postfix string (output).

Repeat STEP 1 to 7 until all symbols are processed from the input string.

Stack

Convert the following infix expressions into postfix expression: A * (B + C)

current symbol	**operator stack**	**postfix string**
A		A
*	*	A
(	* (	A
B	* (	A B
+	* (+	A B
C	* (+	A B C
)	*	A B C +
		A B C + *

For you to do

1. Convert the following infix expressions into postfix expressions
   - a. (a + b * (c + d) - e / f * g + h)
   - b. a + b * c + (d * e + f) * g

2. Evaluate the value of the following postfix expressions
   - a. 6 5 2 3 + 8 * + 3 + *
   - b. 20 2 * 9 + 14 7 / - 5 3 * +