ICT-2101
Data Structure

Lecture 06

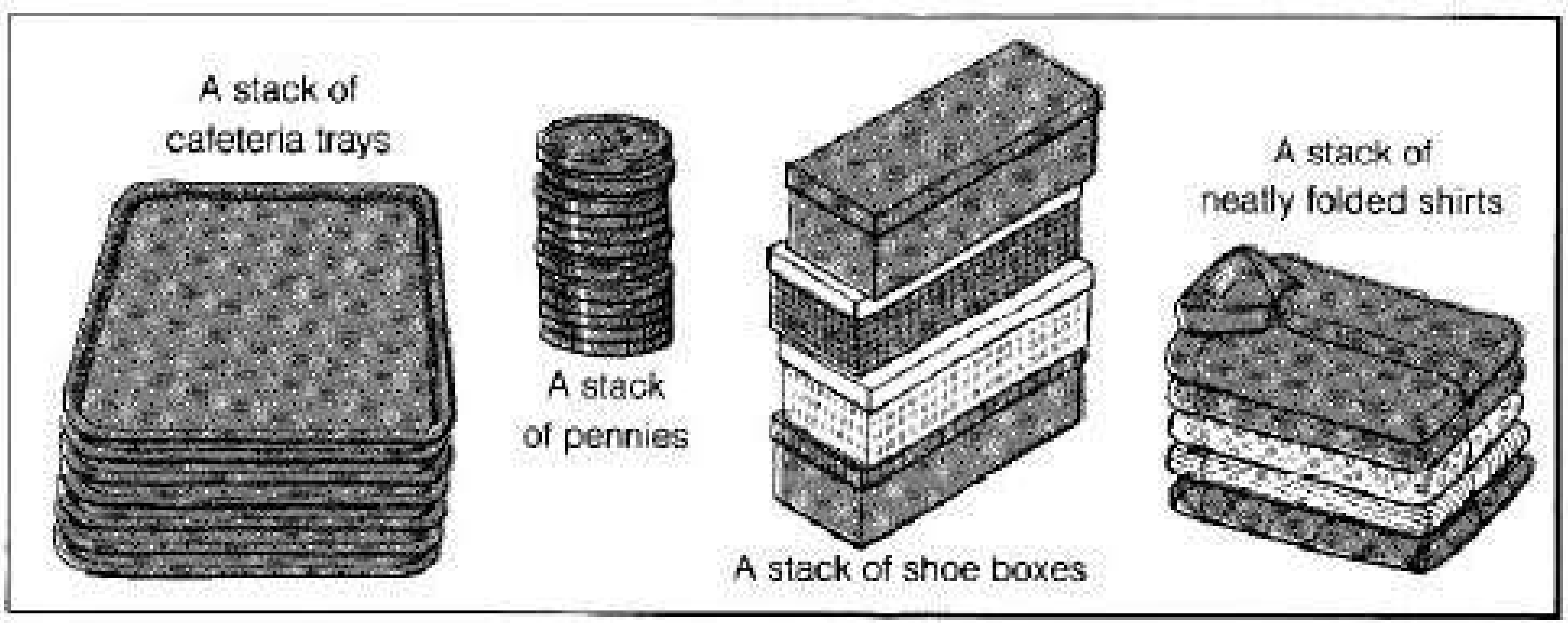
Stack

What is STACKS ?

- It is an ordered group of homogeneous items of elements. Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack). The last element to be added is the first to be removed (LIFO: Last In, First Out).

- A stack is a list of elements in which an element may be inserted or deleted only at one end, called ***TOP of the stack. The elements are removed in reverse order of that*** in which they were inserted into the stack.

What is STACKS ?

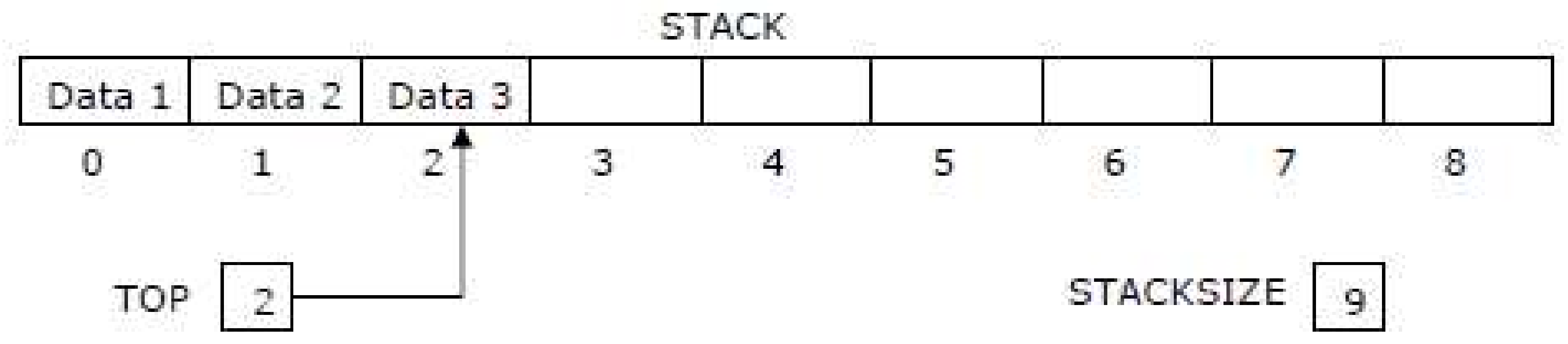


Basic operations

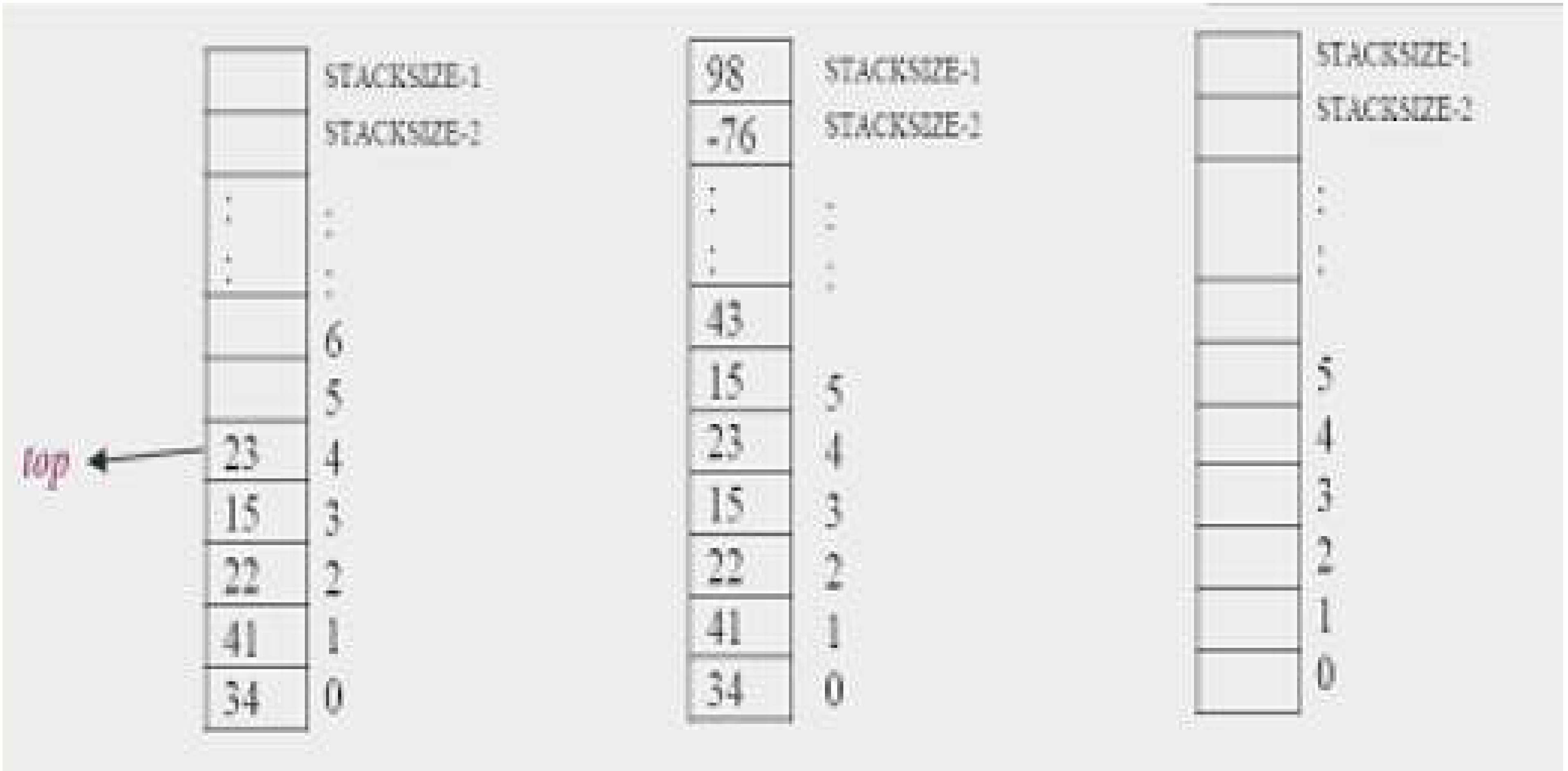
- These are two basic operations associated with stack:
  - ***Push()* is the term used to insert/add an element into a stack.**
  - ***Pop()* is the term used to delete/remove an element from a stack.**
- There are two ways to represent *Stack in memory. One is using array and other is* using linked list.

Array representation of stacks

- Usually the stacks are represented in the computer by a linear array.
- In the following algorithms/procedures of pushing and popping an item from the stacks, we have considered, a linear array STACK, a variable TOP which contain the location of the top element of the stack; and a variable STACKSIZE which gives the maximum number of elements that can be hold by the stack.



Array representation of stacks



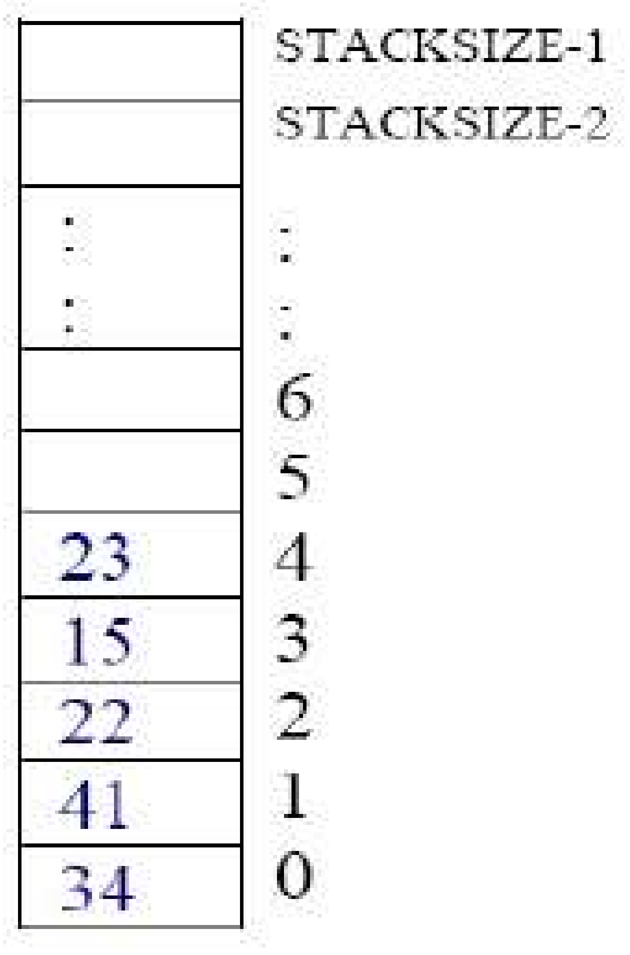
A Stack with 5 elements top=4

A full stack top=stacksize-1

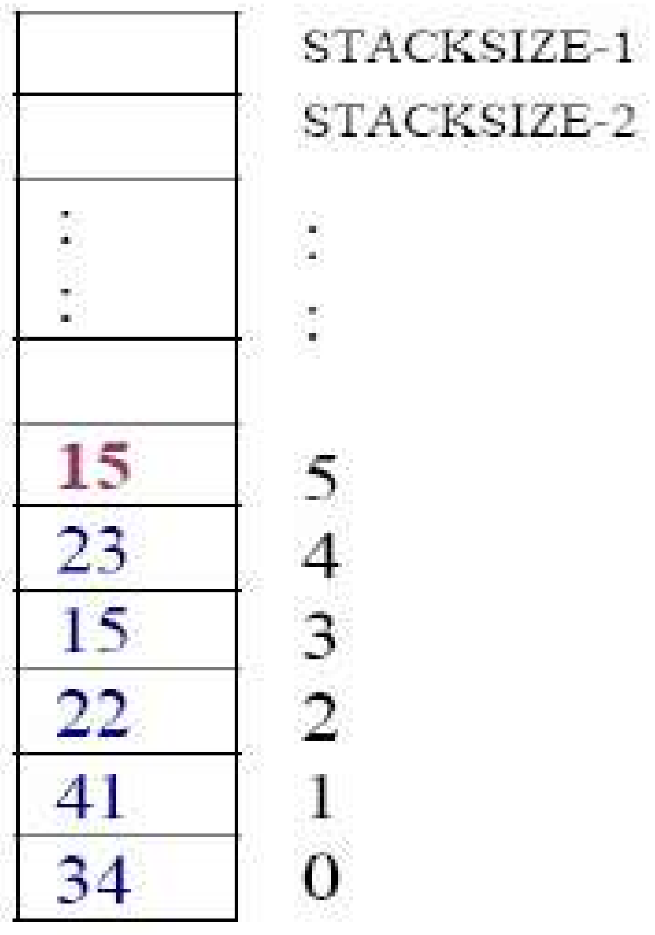
An empty Stack top=-1

Push Operation

- Push an item onto the top of the stack (insert an item)



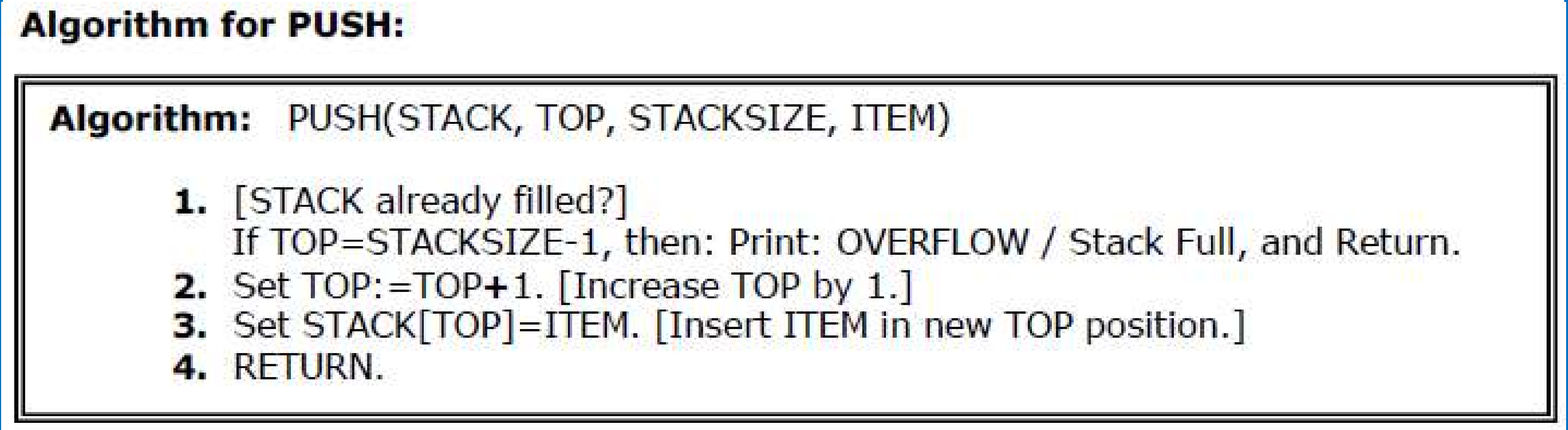
Before PUSH
(top=4, count=5)



After PUSH
(top=5, count= 6)

Push Operation

- Push an item onto the top of the stack (insert an item)

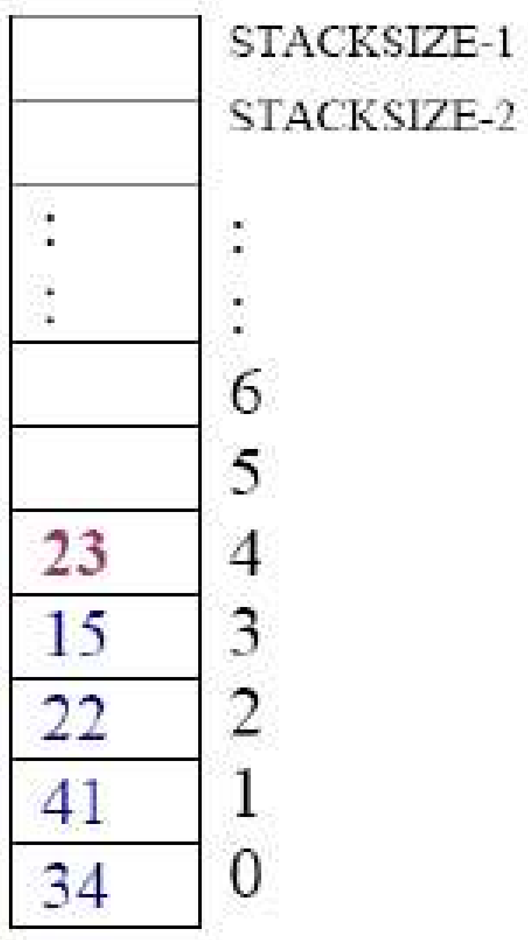


Algorithm for PUSH:

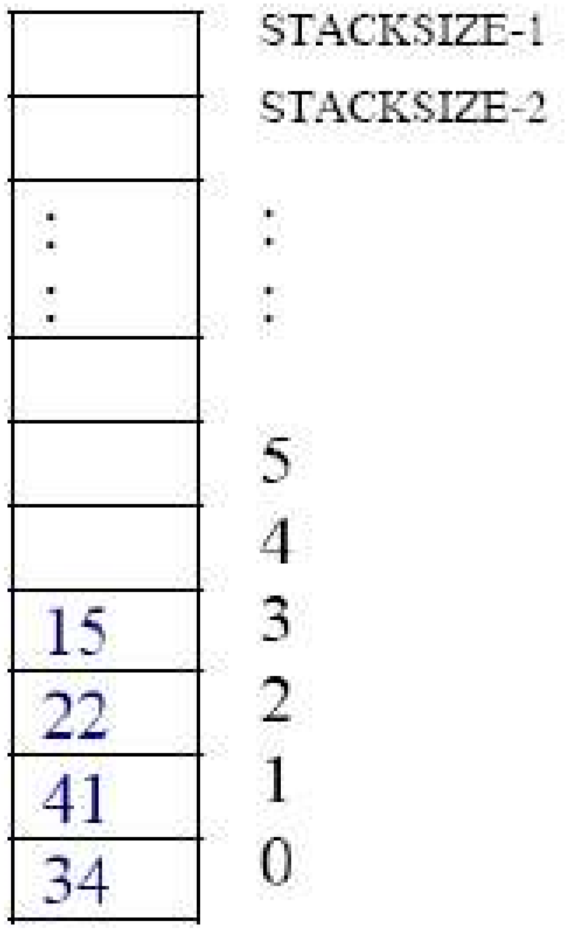
```
Algorithm:   PUSH(STACK, TOP, STACKSIZE, ITEM)

    1. [STACK already filled?]
       If TOP=STACKSIZE-1, then: Print: OVERFLOW / Stack Full, and Return.
    2. Set TOP:=TOP+1. [Increase TOP by 1.]
    3. Set STACK[TOP]=ITEM. [Insert ITEM in new TOP position.]
    4. RETURN.
```

Pop Operation

- Pop an item off the top of the stack (delete an item)



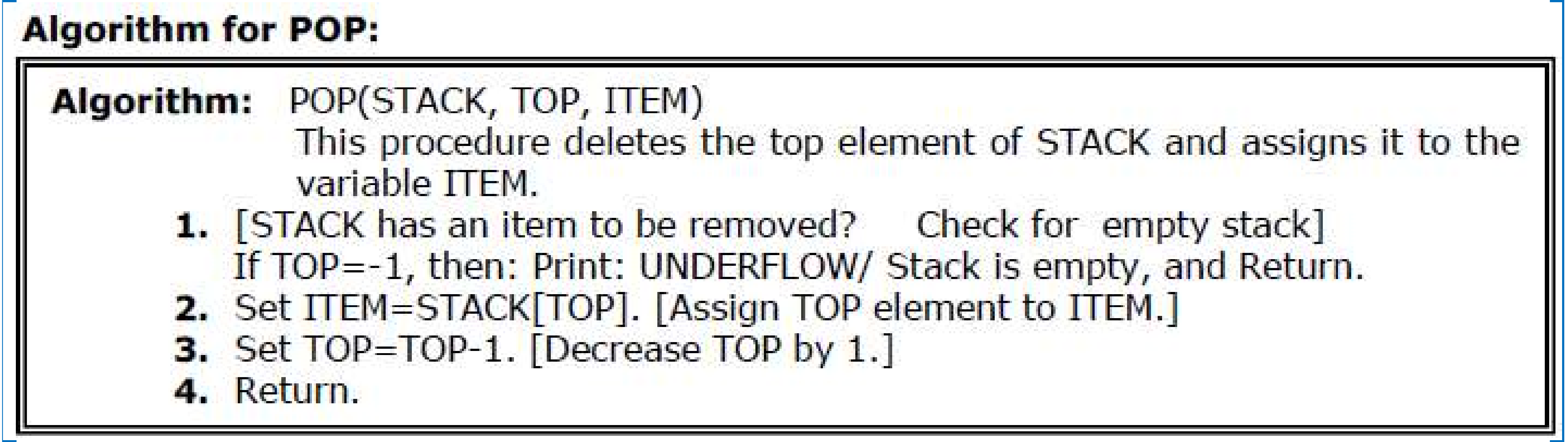
Before POP
(top=4, count=5)



After POP
(top=3 count=4)

Pop Operation

- Pop an item off the top of the stack (delete an item)



Algorithm for POP:

```
Algorithm:  POP(STACK, TOP, ITEM)
            This procedure deletes the top element of STACK and assigns it to the
            variable ITEM.
   1. [STACK has an item to be removed?   Check for  empty stack]
      If TOP=-1, then: Print: UNDERFLOW/ Stack is empty, and Return.
   2. Set ITEM=STACK[TOP]. [Assign TOP element to ITEM.]
   3. Set TOP=TOP-1. [Decrease TOP by 1.]
   4. Return.
```

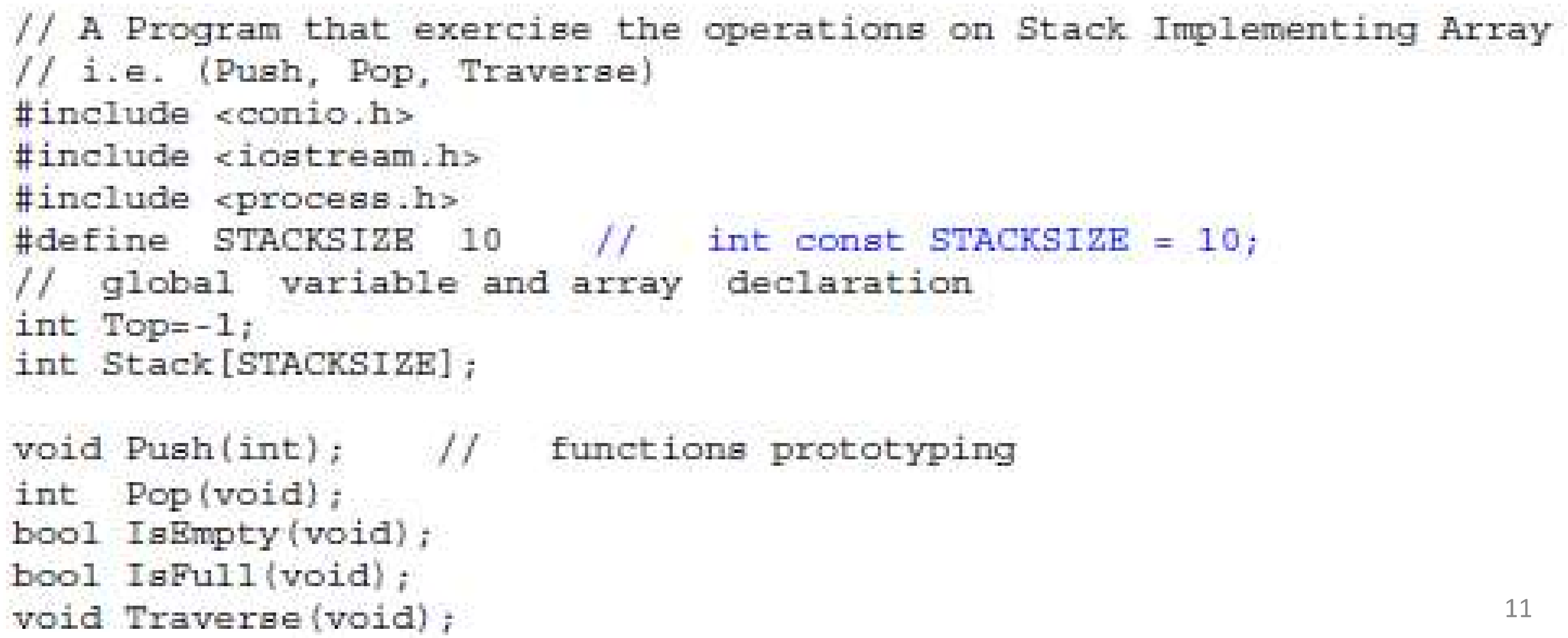
Here are the minimal operations we'd need for an abstract stack (and their typical names):

o **Push**: Places an element/value on *top of the stack.*

o **Pop**: Removes value/element from *top of the stack.*

o **IsEmpty:** Reports whether the stack is Empty or not.

o **IsFull:** Reports whether the stack is Full or not.

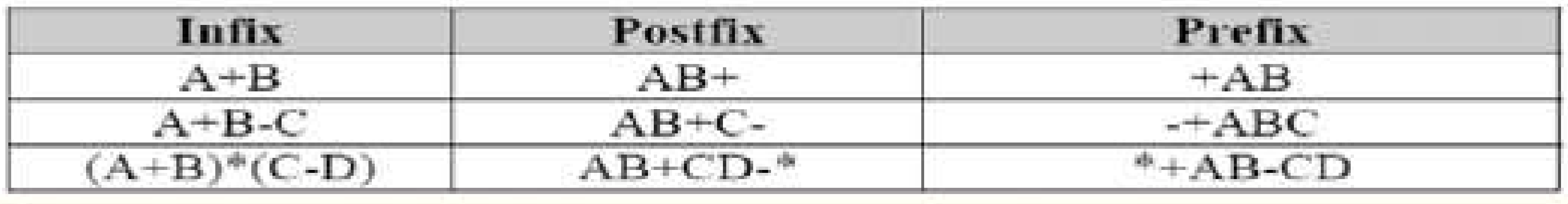


```
// A Program that exercise the operations on Stack Implementing Array
// i.e. (Push, Pop, Traverse)
#include <conio.h>
#include <iostream.h>
#include <process.h>
#define  STACKSIZE  10    //   int const STACKSIZE = 10;
//  global  variable and array  declaration
int Top=-1;
int Stack[STACKSIZE];

void Push(int);    //   functions prototyping
int  Pop(void);
bool IsEmpty(void);
bool IsFull(void);
void Traverse(void);
```

```
int main( )
{ int item,  choice;
   while( 1 )
     {
        cout<< "\n\n\n\n\n";
        cout<< "     *******  STACK  OPERATIONS   ********* \n\n";
        cout<< "  1-  Push item \n  2-  Pop Item \n";
        cout<< "  3-  Traverse / Display  Stack Items \n  4-  Exit.";
        cout<< " \n\n\t  Your choice ---> ";
        cin>> choice;
        switch(choice)
        { case 1: if(IsFull())cout<< "\n Stack Full/Overflow\n";
                  else
                   { cout<< "\n Enter a number: ";    cin>>item;
                    Push(item); }
                  break;
          case 2: if(IsEmpty())cout<< "\n Stack is empty) \n";
                         else
                 {item=Pop();
                        cout<< "\n deleted from Stack = "<<item<<endl;}
                   break;
            case 3: if(IsEmpty())cout<< "\n Stack is empty) \n";
                 else
                  { cout<< "\n List of Item pushed on Stack:\n";
                      Traverse();
                          }
                   break;
            case 4:  exit(0);
            default:
                    cout<< "\n\n\t Invalid Choice: \n";
          } // end of  switch block

         }  // end of while loop

    }  // end of of main() function


void Push(int item)
{
  Stack[++Top]  = item;
}

int Pop( )
{
   return Stack[Top--];
}

bool IsEmpty( )
{ if(Top == -1 ) return true else return false; }

bool IsFull( )
{ if(Top == STACKSIZE-1 ) return true else return false; }

void Traverse( )
 { int TopTemp = Top;
    do{ cout<< Stack[TopTemp--]<<endl;} while(TopTemp>= 0);
 }
```

Application of the Stack (Arithmetic Expressions)

INFIX , POST FIX AND PRE FIX NOTATIONS :



Infix	Postfix	Prefix
A+B	AB+	+AB
A+B-C	AB+C-	-+ABC
(A+B)*(C-D)	AB+CD-*	*+AB-CD

Example: 3+4 (infix), 3 4 + (postfix), + 3 4 (prefix)

- Stacks are used by compilers to help in the process of converting infix to postfix arithmetic expressions and also evaluating arithmetic expressions.
- Arithmetic expressions consisting variables, constants, arithmetic operators and parentheses.

Application of the Stack (Arithmetic Expressions)

- To evaluate a complex infix expression, a compiler would first convert the expression to postfix notation, and then evaluate the postfix version of the expression.
- We use the following three levels of precedence for the five binary operations.

Precedence	**Binary Operations**
Highest	Exponentiations (^)
Next Highest	Multiplication (*), Division (/) and Mod (%)
Lowest	Addition (+) and Subtraction (-)

Application of the Stack (Arithmetic Expressions)

- Example:

(66 + 22) * 5 – 567 / 42

to postfix

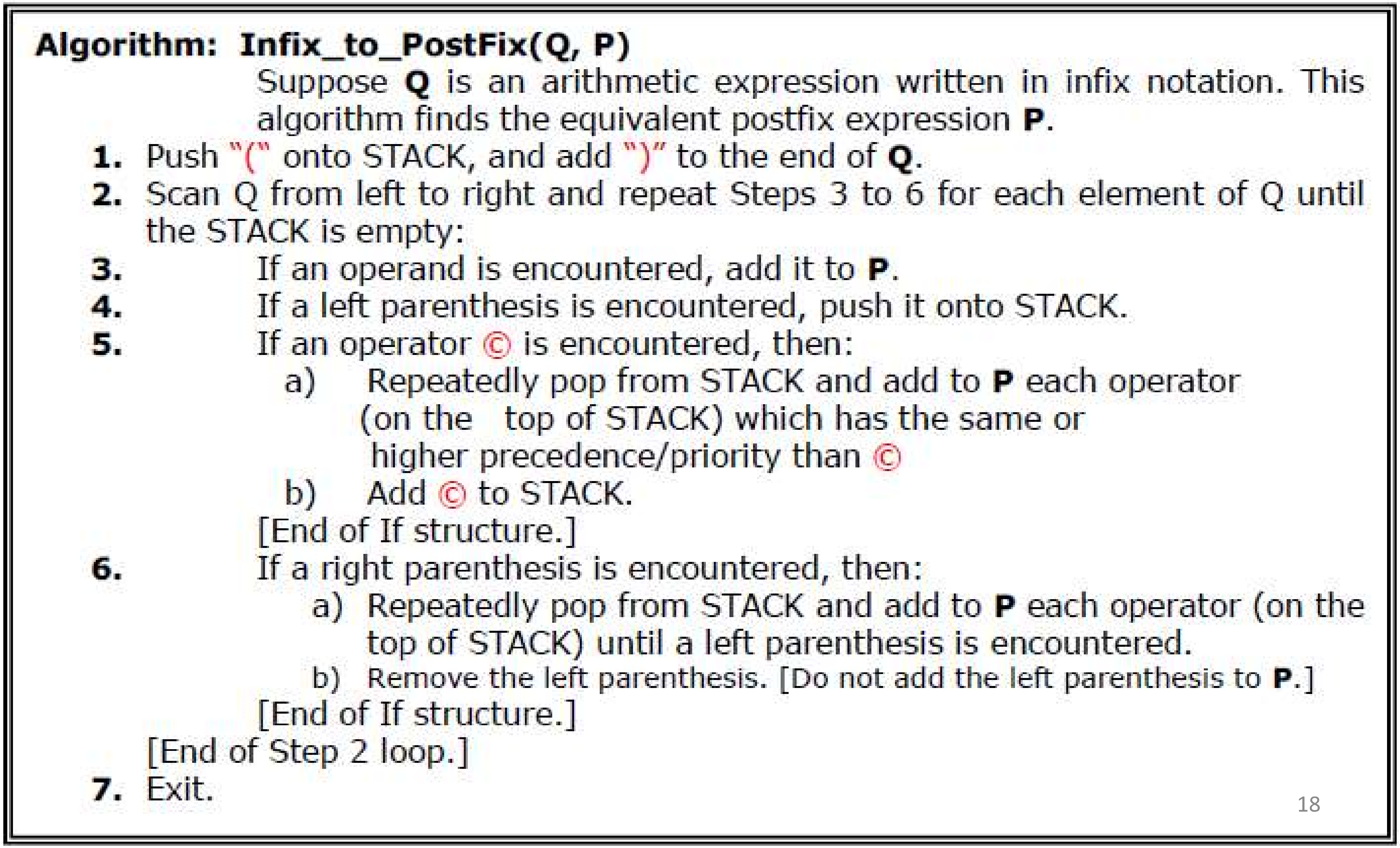
66 22 + 5 * 567 42 / –

Transforming Infix Expression into Postfix Expression

- The following algorithm transform the infix expression **Q into its equivalent** postfix expression **P. It uses a stack to temporary hold the operators and left** parenthesis.

- The postfix expression will be constructed from left to right using operands from **Q** and operators popped from STACK.

Transforming Infix Expression into Postfix Expression



```
Algorithm:  Infix_to_PostFix(Q, P)
            Suppose Q is an arithmetic expression written in infix notation. This
            algorithm finds the equivalent postfix expression P.
  1. Push "(" onto STACK, and add ")" to the end of Q.
  2. Scan Q from left to right and repeat Steps 3 to 6 for each element of Q until
     the STACK is empty:
  3.        If an operand is encountered, add it to P.
  4.        If a left parenthesis is encountered, push it onto STACK.
  5.        If an operator © is encountered, then:
              a)  Repeatedly pop from STACK and add to P each operator
                  (on the  top of STACK) which has the same or
                  higher precedence/priority than ©
              b)  Add © to STACK.
            [End of If structure.]
  6.        If a right parenthesis is encountered, then:
               a) Repeatedly pop from STACK and add to P each operator (on the
                  top of STACK) until a left parenthesis is encountered.
               b) Remove the left parenthesis. [Do not add the left parenthesis to P.]
            [End of If structure.]
     [End of Step 2 loop.]
  7. Exit.
```

- Convert **Q: A+(B * C – (D / E ^ F) * G) * H** into postfix form showing stack status .
- Now add **“)”** at the end of expression

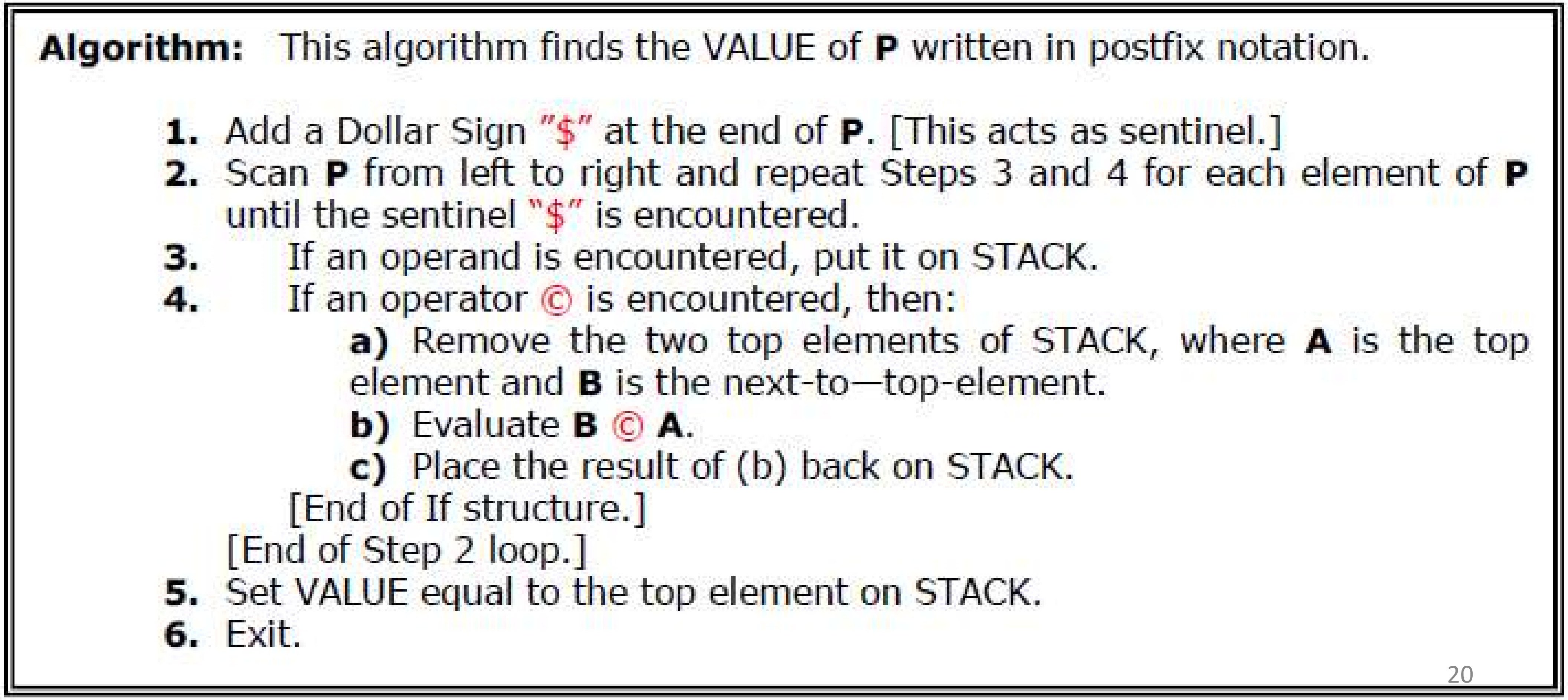
A+(B * C – (D / E ^ F) * G) * H) and also Push a **“(“** on Stack.

Example

Symbol Scanned	Stack	Expression Y
	(	
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
-	(+(-	ABC*
(	(+(-(	ABC*
D	(+(-(	ABC*D
/	(+(-(/	ABC*D
E	(+(-(/	ABC*DE
^	(+(-(/^	ABC*DE
F	(+(-(/^	ABC*DEF
)	(+(-	ABC*DEF^/
*	(+(-*	ABC*DEF^/
G	(+(-*	ABC*DEF^/G
)	(+	ABC*DEF^/G*-
*	(+*	ABC*DEF^/G*-
H	(+*	ABC*DEF^/G*-H
)	empty	**ABC*DEF^/G*-H*+**

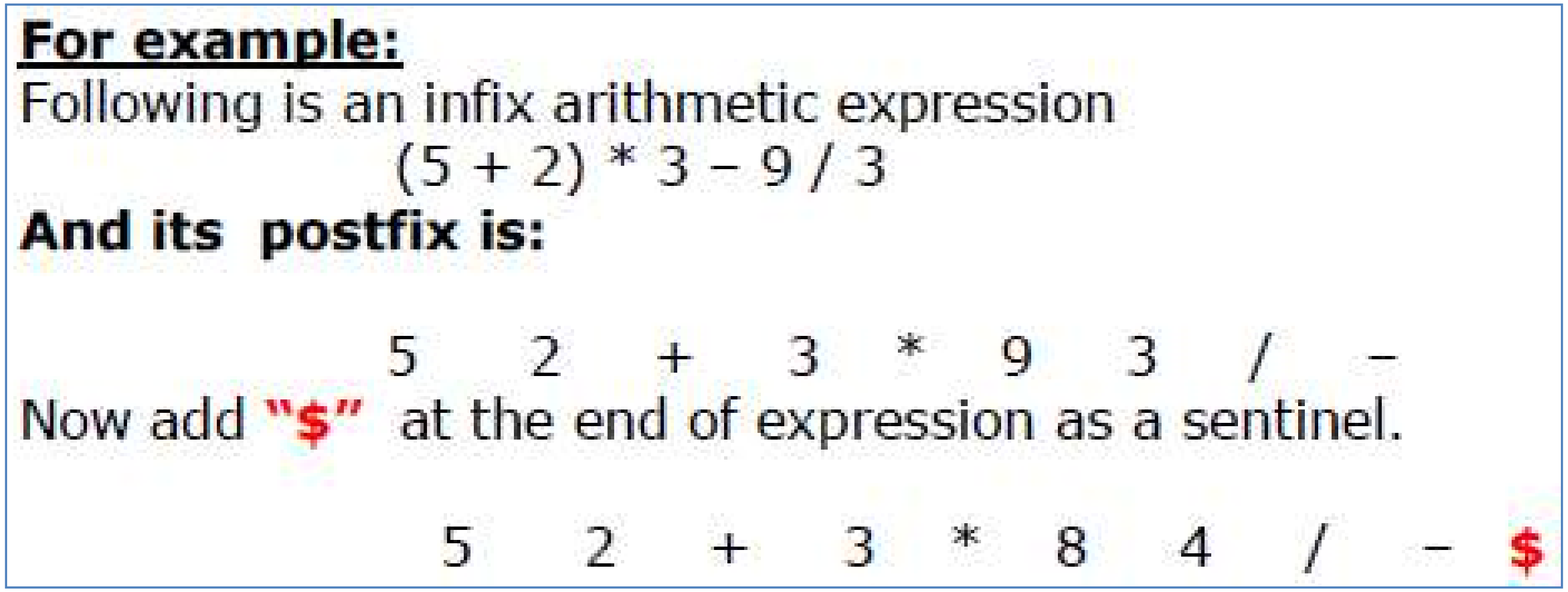
Evaluation of Postfix Expression

- If P is an arithmetic expression written **in postfix notation**. **This algorithm** uses STACK to hold operands, and evaluate **P.**



```
Algorithm:   This algorithm finds the VALUE of P written in postfix notation.

1. Add a Dollar Sign "$" at the end of P. [This acts as sentinel.]
2. Scan P from left to right and repeat Steps 3 and 4 for each element of P
   until the sentinel "$" is encountered.
3.      If an operand is encountered, put it on STACK.
4.      If an operator © is encountered, then:
            a) Remove the two top elements of STACK, where A is the top
            element and B is the next-to—top-element.
            b) Evaluate B © A.
            c) Place the result of (b) back on STACK.
        [End of If structure.]
   [End of Step 2 loop.]
5. Set VALUE equal to the top element on STACK.
6. Exit.
```

Evaluation of Postfix Expression



<u>For example:</u>
Following is an infix arithmetic expression

(5 + 2) * 3 – 9 / 3

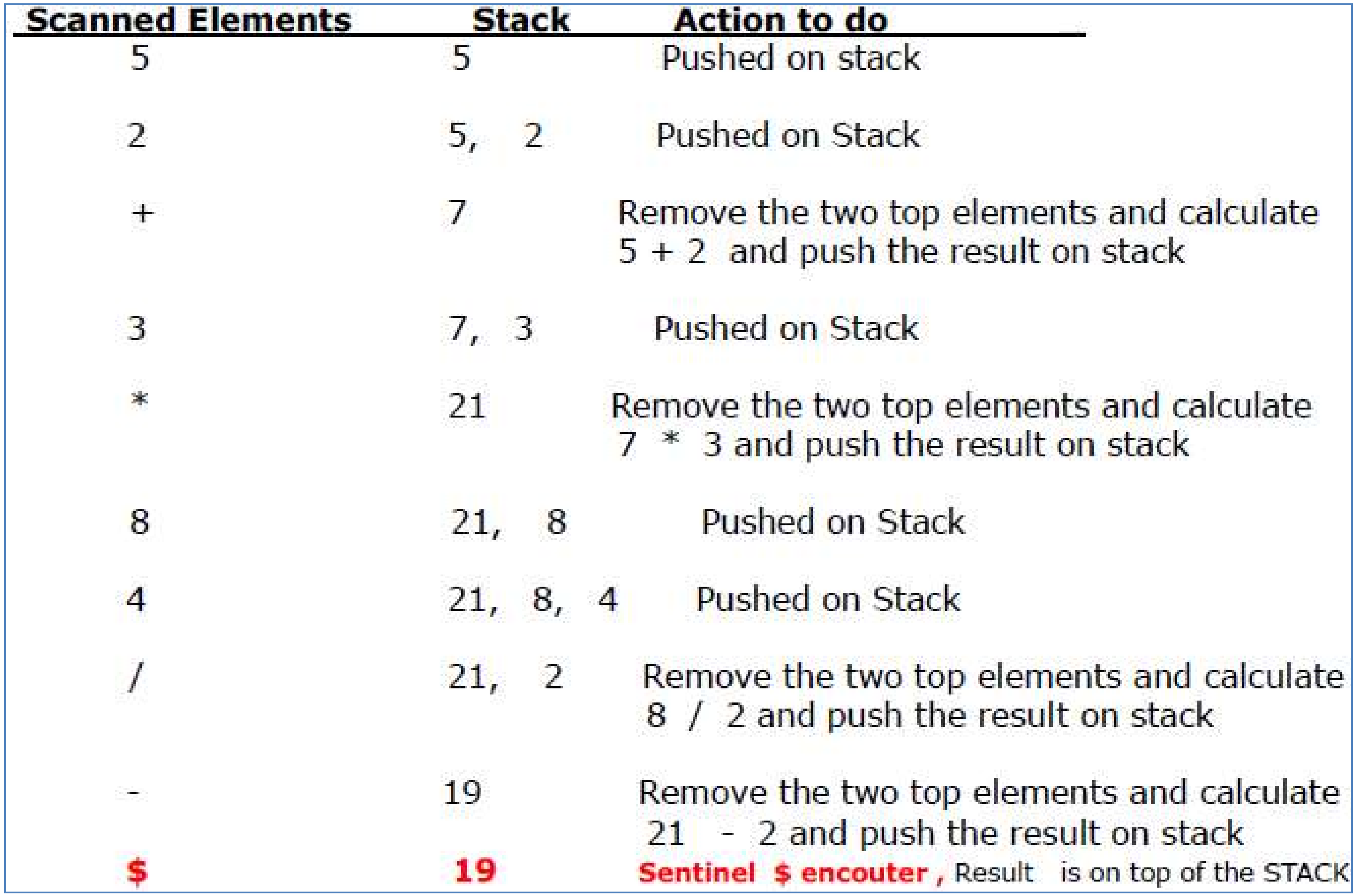
And its postfix is:

5 2 + 3 * 9 3 / –

Now add **"$"** at the end of expression as a sentinel.

5 2 + 3 * 8 4 / – **$**

Evaluation of Postfix Expression



Scanned Elements	Stack	Action to do
5	5	Pushed on stack
2	5, 2	Pushed on Stack
+	7	Remove the two top elements and calculate 5 + 2 and push the result on stack
3	7, 3	Pushed on Stack
*	21	Remove the two top elements and calculate 7 * 3 and push the result on stack
8	21, 8	Pushed on Stack
4	21, 8, 4	Pushed on Stack
/	21, 2	Remove the two top elements and calculate 8 / 2 and push the result on stack
-	19	Remove the two top elements and calculate 21 - 2 and push the result on stack
$	**19**	**Sentinel $ encouter ,** Result is on top of the STACK

Thank you