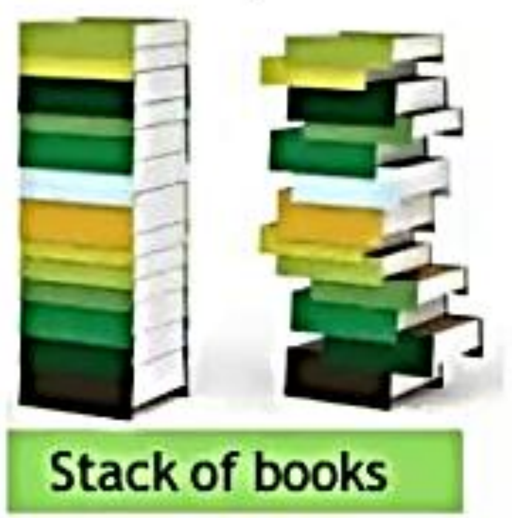
DATA STRUCTURES AND ALGORITHMS

CHAPTER 2: STACK



INTRODUCTION

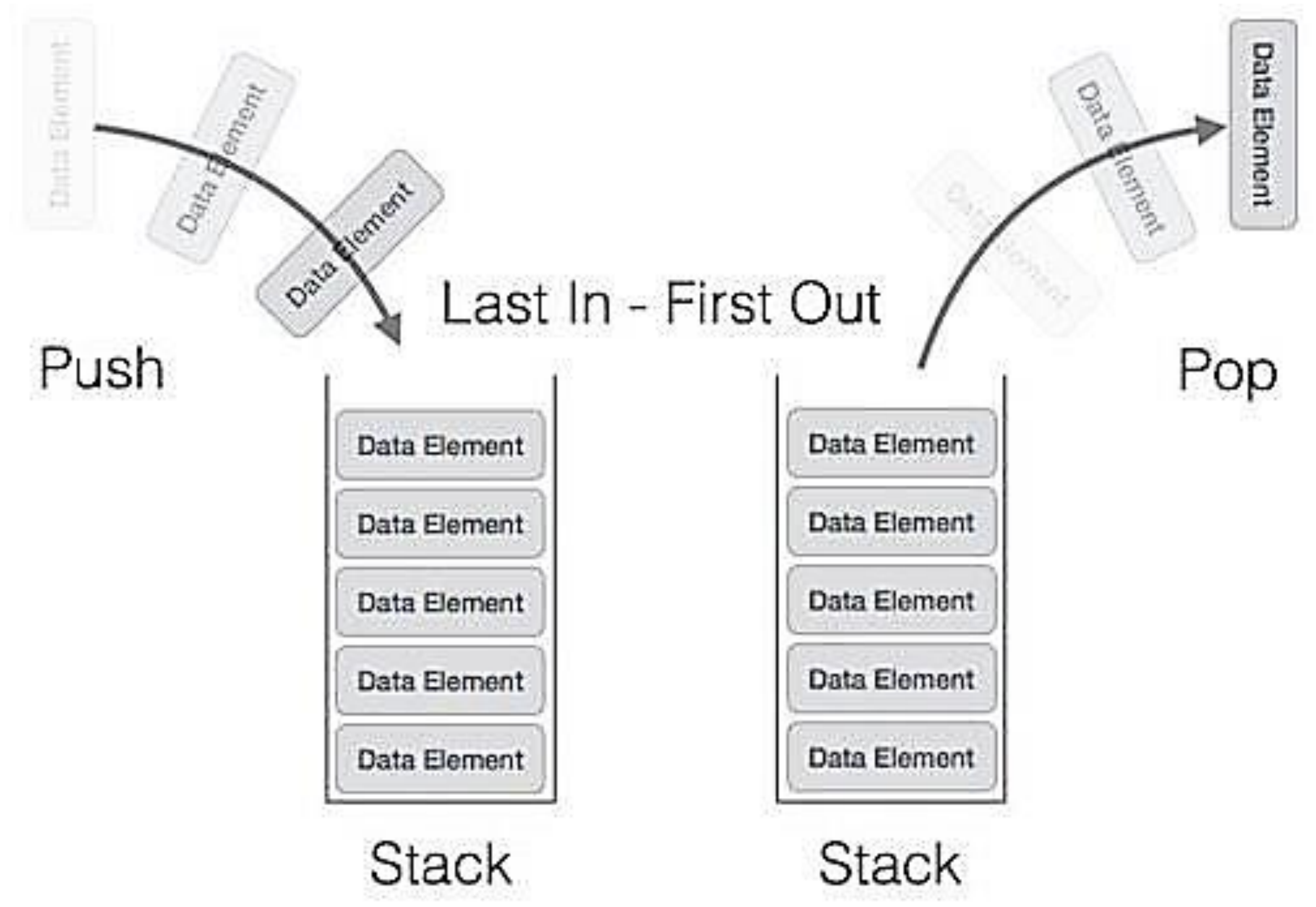
- Stack is a abstract data type.
- A stack is a simple data structure used for storing data.
- In a stack, the order in which the data arrives is important.
- A pile of plates in a cafeteria is a good example of a stack:

  The plates are added to the stack as they are cleaned and they are placed on the top. When a plate, is required it is taken from the top of the stack. The first plate placed on the stack is the last one to be used.
- Likewise, Stack allows all data operations at one end only.

- At any given time, we can only access the top element of a stack.
- This feature makes it LIFO data structure.
- LIFO stands for Last-in-first-out. (FILO stands for First-in-last-out)
  Here, the element which is placed (inserted or added) last, is accessed first.
- In stack terminology, insertion operation is called PUSH operation and removal operation is called POP operation.
- Stack can either be a fixed size one or it may have a sense of dynamic resizing.

STACK REPRESENTATION

-



BASIC OPERATIONS

- Stack is used for the following two primary operations
    - push( ) – Pushing (storing) an element on the stack.
    - pop( ) – Removing (accessing) an element from the stack.

- When data is PUSH onto stack, 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.

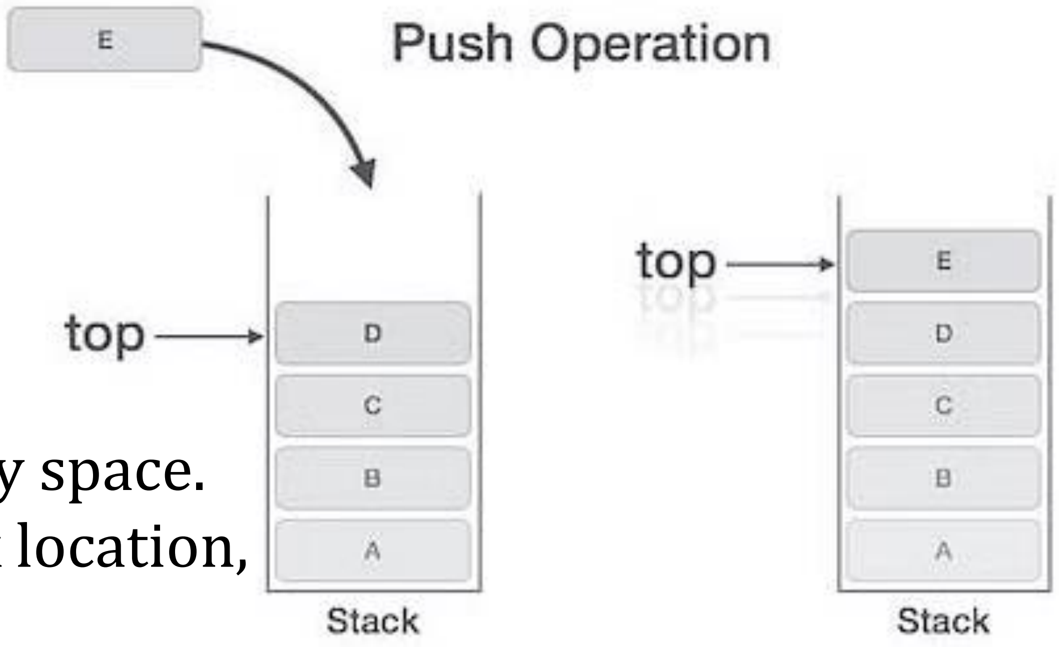
- At all times, we maintain a pointer to the last PUSH data on the stack. As this pointer always represents the top of the stack, hence named top. The top pointer provides top value of the stack without actually removing it.

APPLICATIONS

- Print reverse string
- Undo mechanism
- Recursion / function call itself
- Checking parenthesis balancing
- Infix, Postfix and Prefix Conversion

INSERT AN ELEMENT IN TO THE STACK

- Push operation is used to add new elements in to the stack.
- At the time of addition first check the stack is full or not.
- If the stack is full it generates an error message "stack overflow".
- Push operation involves a series of steps
  - Step 1 – Checks if the stack is full.
  - Step 2 – If the stack is full,
    produces an error and exit.
  - 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.



ALGORITHM

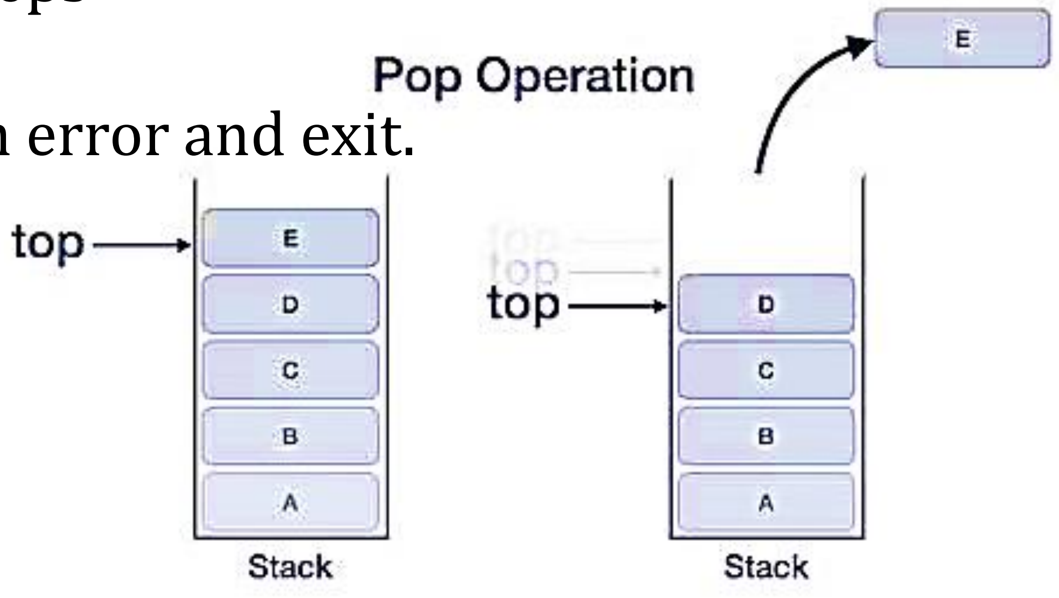
```
begin procedure push: stack, data
if stack is full
    return null
endif
top ← top + 1
stack[top] ← data
end procedure
```

IMPLEMENTATION

```
void push(int element) {
    if(isFull( )) {
        printf("Could not insert data,
            Stack is full.\n");
            }
    else {
        top = top + 1;
        stackarray[top] = element;
        //stackarray[++top] = element;
        }
}
```

REMOVE AN ELEMENT FROM THE STACK

- Pop operation is used to delete elements from the stack.
- At the time of deletion first check the stack is empty or not.
- If the stack is empty it generates an error message "stack underflow"
- A Pop operation may involve the following steps
    - Step 1 – Checks if the stack is empty.
    - Step 2 – If the stack is empty, produces an error and exit.
    - 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.



ALGORITHM

```
begin procedure pop: stack
if stack is empty
    return null
endif
data ← stack[top]
top ← top - 1
return data
end procedure
```

IMPLEMENTATION

```
int pop() {
    if(isempty()) {
        printf("Could not retrieve data,
            Stack is empty.\n");
    }
    else {
        data = stack[top];
        top = top - 1;
        return data;
        \\return stackarray[top--];
            }
}
```

PEEK() / TOP()

ALGORITHM

```
begin procedure peek
    return stack[top]
end procedure
```

IMPLEMENTATION

```
int peek( ) {
    return stack[top];
}
```

ISFULL()

ALGORITHM

```
begin procedure isfull
    if top equals to MAXSIZE - 1
        return true
    else
        return false
    endif
end procedure
```

IMPLEMENTATION

Method 1:

```
boolean isfull( ) {
     if(top == MAXSIZE-1)
         return true;
    else
         return false;
}
```

Method 2:

```
boolean isfull( ) {
         return top == maxsize – 1;
}
```

ISEMPTY()

ALGORITHM

```
begin procedure isempty
    if top less than 0
        return true
    else
        return false
    endif
end procedure
```

IMPLEMENTATION IN C

```
bool isempty( ) {
    \\ return top = -1;
    if(top == -1)
        return true;
    else
        return false;
```

Note: In implementation,
We initialize top at -1,
as the index in array starts from 0.
So we check if the top is below zero or -1
to determine if the stack is empty.

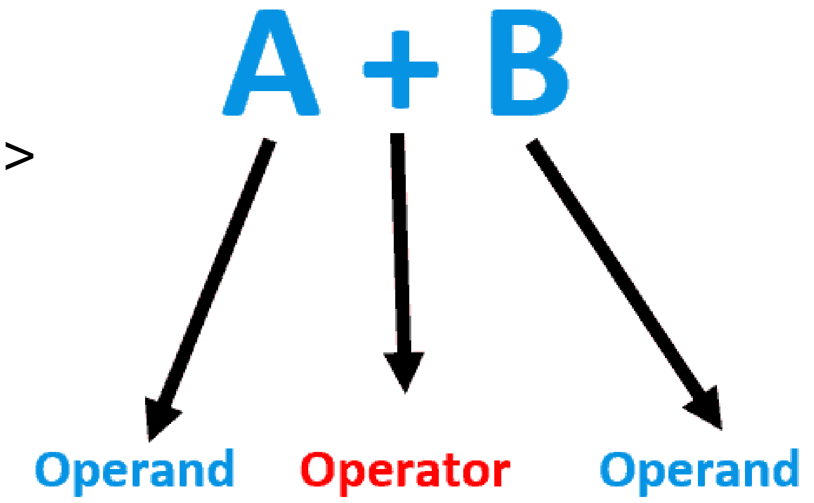
Complete Program in Java

CHAPTER 2: INFIX, PREFIX AND POSTFIX EXPRESSIONS

INFIX EXPRESSION

- Infix expressions are the most usual type of expression.
- **I**n the infix expression, we place the operator between the two operands it operates on.
- For example, the operator “+” appears between the operands A and B in the expression “A + B”.
- Syntax of infix notation is given below:

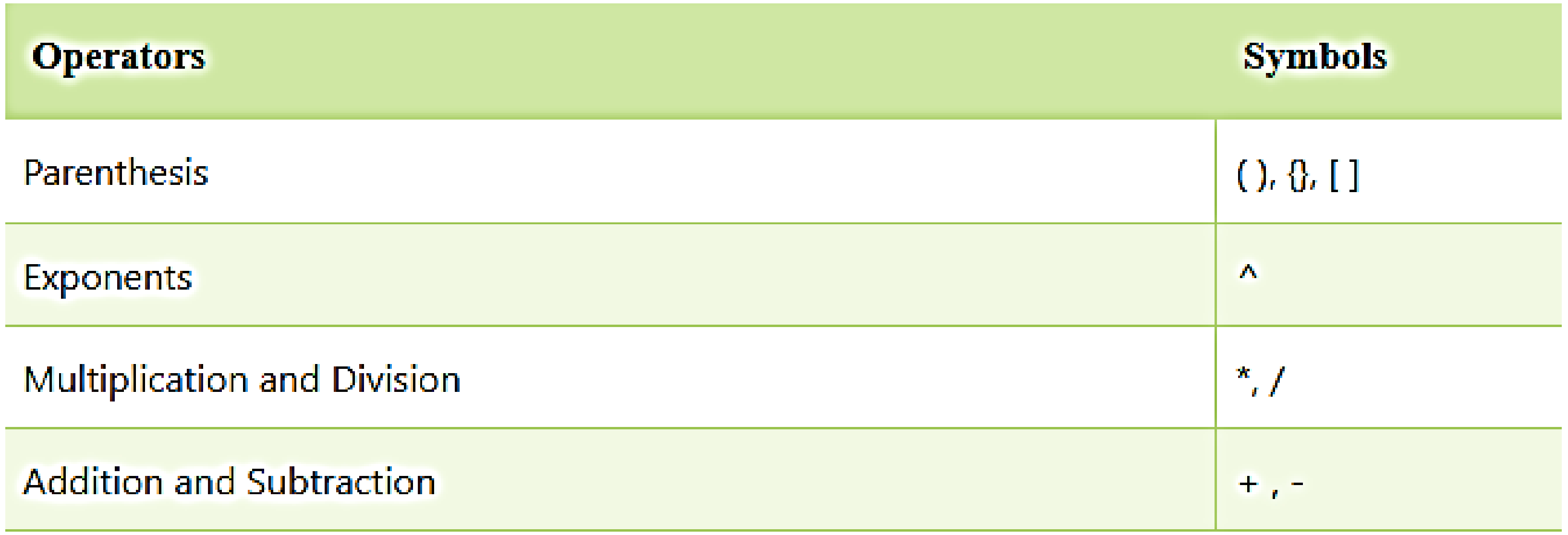
  <operand> <operator> <operand>



- Furthermore, infix expressions can also include parentheses to indicate the order of operations.

- Operator precedence rules specify the operator evaluation order in an expression. So, in an expression, operators with higher precedence are evaluated before operators with lower precedence.

- Some operator precedence rules follow:



Operators	Symbols
Parenthesis	( ), {}, [ ]
Exponents	^
Multiplication and Division	*, /
Addition and Subtraction	+ , -

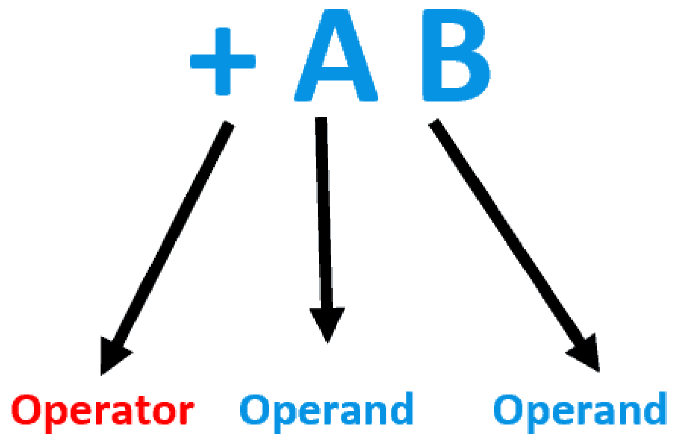
- **Note:** If an expression has multiple operators with the same precedence, the evaluation of those operators occurs from left to right.

PREFIX EXPRESSION

- Prefix expressions, also known as Polish notation, place the operator before the operands.

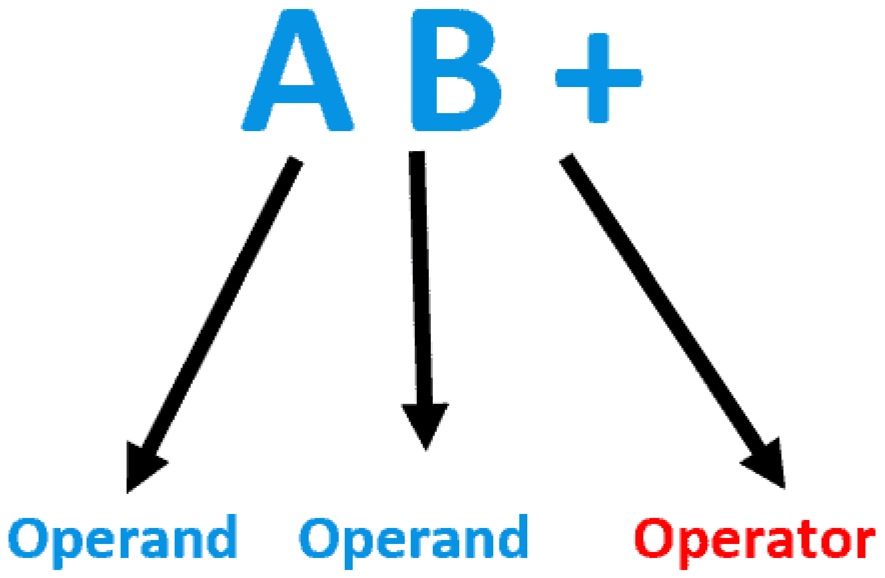
- **I**n the prefix expression, we place the operator before the two operands it operates on.

- For example, in the expression “+ A B”, we place the “+” operator before the operands A and B



POSTFIX EXPRESSION

- Postfix expressions, also known as reverse Polish notation, where we place the operator after the operands.
- For example, in the expression "A B +", the "+" we place the operator after the operands A and B. The figure next depicts the example:



EXAMPLE

Infix Expression	Prefix Expression	Postfix Expression
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +
(A + B) * C	* + A B C	A B + C *
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +

CHAPTER 2: CONVERSION OF INFIX EXPRESSIONS TO PREFIX AND POSTFIX

INFIX TO POSTFIX CONVERSION

- The major rules
  - ➢ The operations with highest precedence are converted first
  - ➢ That after a portion of an expression has been converted to postfix, it is to be treated as a single operand.

- There are three levels of operator precedence.

  ^

  * and /.

  + and –

- Furthermore, when operators of the same precedence are scanned,

  +, –, * and / are left associative, but ^ is right associative

- As we scan the infix expression from left to right, we will use a stack to keep the operators.

- Assume the infix expression is a string of tokens delimited by spaces.

- The operator tokens are *, /, +, and -, along with the left and right parentheses, (and).

- The operand tokens are the single-character identifiers A, B, C, and so on.

- Associative Rule:

  Left to Right : pop the stack and then push the incoming token

  Right to Left : push the incoming token on to the stack

Example:

1. Convert A + B * C to postfix.
   - ✓ A + B * C can be written as (A + (B * C))
   - ✓ Applying the rules of precedence,

     we first convert the portion of the expression that is evaluated first, namely the multiplication.
   - ✓ we obtain

     A+B*C

     A+(B*C) Parentheses for emphasis

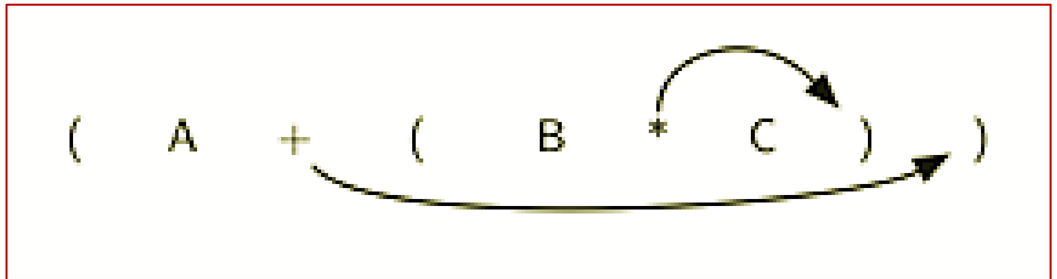
     A+(BC*) Convert the multiplication,

     Let D=BC*

     A+D Convert the addition

     A(D)+

     ABC*+ Postfix Form



2. Convert (A + B) * C to postfix.

   Convert the addition : A B + * C

   Convert the multiplication : A B + C *

INFIX TO POSTFIX CONVERSION USING STACK

- Stacks are used for converting an infix expression to a postfix expression.
- The stack that we use in the algorithm will change the order of operators from infix to Postfix.
- Postfix expressions do not contain parentheses.

- The following steps will produce a string of tokens in postfix order.
    1. Create an empty stack for keeping operators. Create an empty list for output.

    2. Scan the token list from left to right.

    3. If the token is an operand,
        - ➢ append it to the end of the output list.

    4. If the token is a left parenthesis,
        - ➢ push it on the stack.

    5. If the token is a right parenthesis,
        - ➢ pop the stack until the corresponding left parenthesis is removed.
        - ➢ Append each operator to the end of the output list.

6. If the token is an operator(*, /, +, or -,) which has the higher precedence than the top of the stack,
    - ➢ push it on the stack.

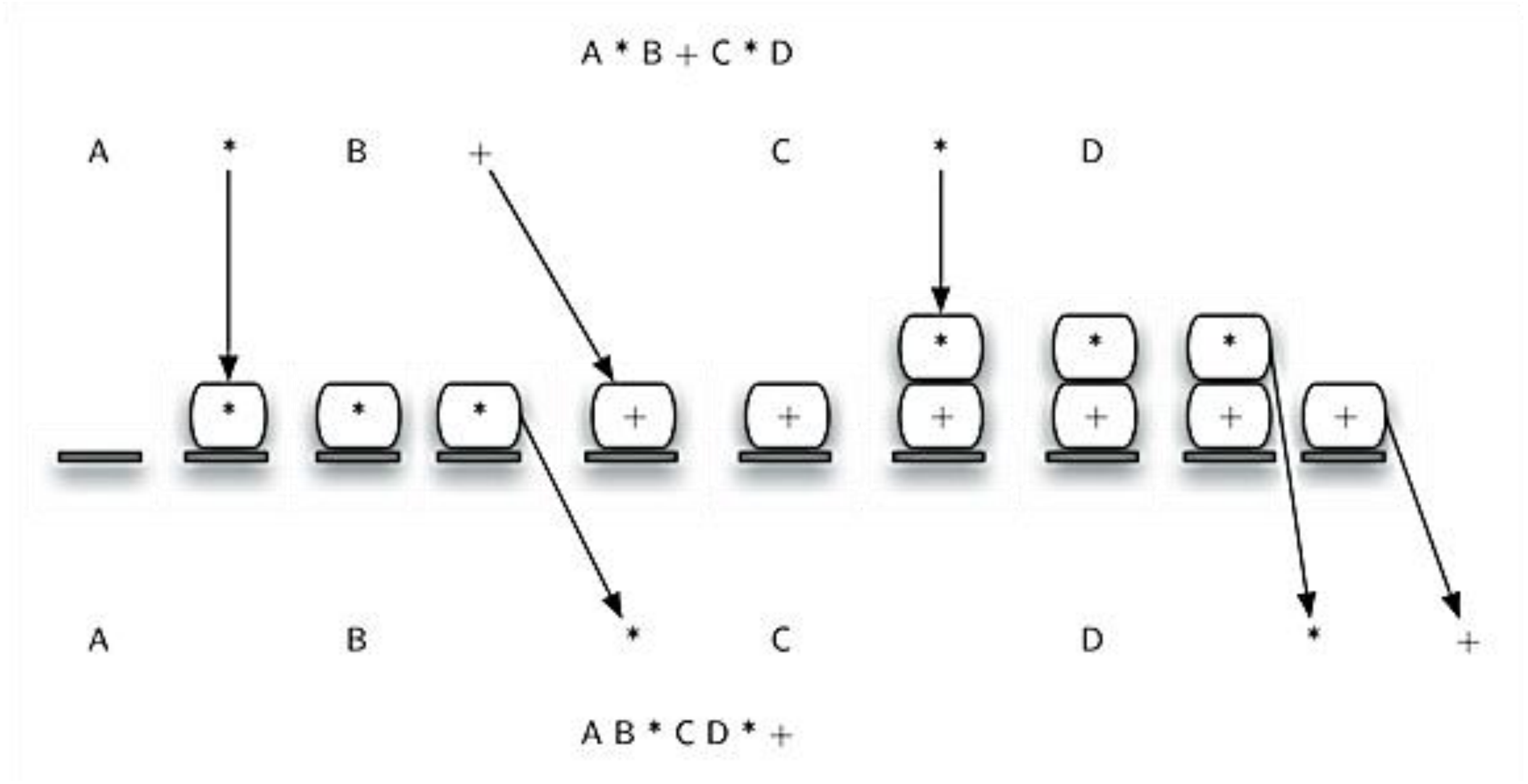
7. If the token is an operator which has the lower precedence than the top of the stack,
    - ➢ pop the stack and append each operator to the end of the output list .
    - ➢ Then test the token against the new top.

7. If the token is an operator which has the equal precedence with the top of the stack,
    - ➢ use the associative rule.

8. When the input expression has been completely processed,
    - ➢ check the stack. Any operators still on the stack can be removed and appended to the end of the output list.

EXAMPLE:

1. Convert A * B + C *D to Postfix expression



2. Convert A * B – (C + D) + E to Postfix expression

Input Character	Operations on Stack	Stack	Postfix Expression
A		Empty	A
*	Push	*	A
B		*	AB
-	Check and Push	-	AB*
(	Push	-(	AB*
C		-(	AB*C
+	Check and Push	-(+	AB*C
D			AB*CD
)	Pop and Append to Postfix till '('	-	AB*CD+
+	Check and Push	+	AB*CD+-
E		+	AB*CD+-E
End	Pop till Empty		**AB*CD+-E+**

EXERCISE

1) Convert the following infix expressions to postfix expressions.
   a) A + B – C
   b) A + B * C
   c) (A + B) / (C – D)
   d) ((A + B) * (C – D) + E) / (F + G)
   e) 2 * 3 / (2 – 1) + 5 * 3
   f) 8 – 2 + (3 * 4) / 2 ^ 2
   g) A + B * C / D – F + A ^ E

INFIX TO PREFIX CONVERSION

- Convert A + B * C to prefix
    - ➢ First scan:
        - ✓ In the above expression, multiplication operator has a higher precedence than the addition operator;
        - ✓ the prefix notation of B*C would be (*BC)
        - ✓ A + *BC
    - ➢ Second scan: the prefix would be:
        - ✓ +A *BC
- Associative Rule:

  Left to Right : push the incoming token on to the stack

  Right to Left : pop the stack and then push the incoming token

INFIX TO PREFIX CONVERSION USING STACK

- Rules for the conversion of infix to prefix expression:
  - ➢ First, reverse the infix expression given in the problem.
  - ➢ Scan the expression from left to right.
  - ➢ Whenever the operands arrive, append it to the output string.
  - ➢ If the operator arrives and the stack is found to be empty, then simply push the operator into the stack.
  - ➢ If the incoming operator has higher precedence than the TOP of the stack, push the incoming operator into the stack.
  - ➢ If the incoming operator has the same precedence with a TOP of the stack, use the associative rule

- If the incoming operator has lower precedence than the TOP of the stack,
  pop, and print the top of the stack.
  Test the incoming operator against the top of the stack again and pop the operator from the stack till it finds the operator of a lower precedence or same precedence.
  Then push the current operator onto the stack.

- If the incoming operator has the same precedence with the top of the stack and the incoming operator is ^,
  use the right associative rule

- When we reach the end of the expression,
  pop, and print all the operators from the top of the stack.

- If the operator is '(‘,
  then pop all the operators from the stack till it finds closing bracket in the stack.

- If the top of the stack is ')',
  push the operator on the stack.

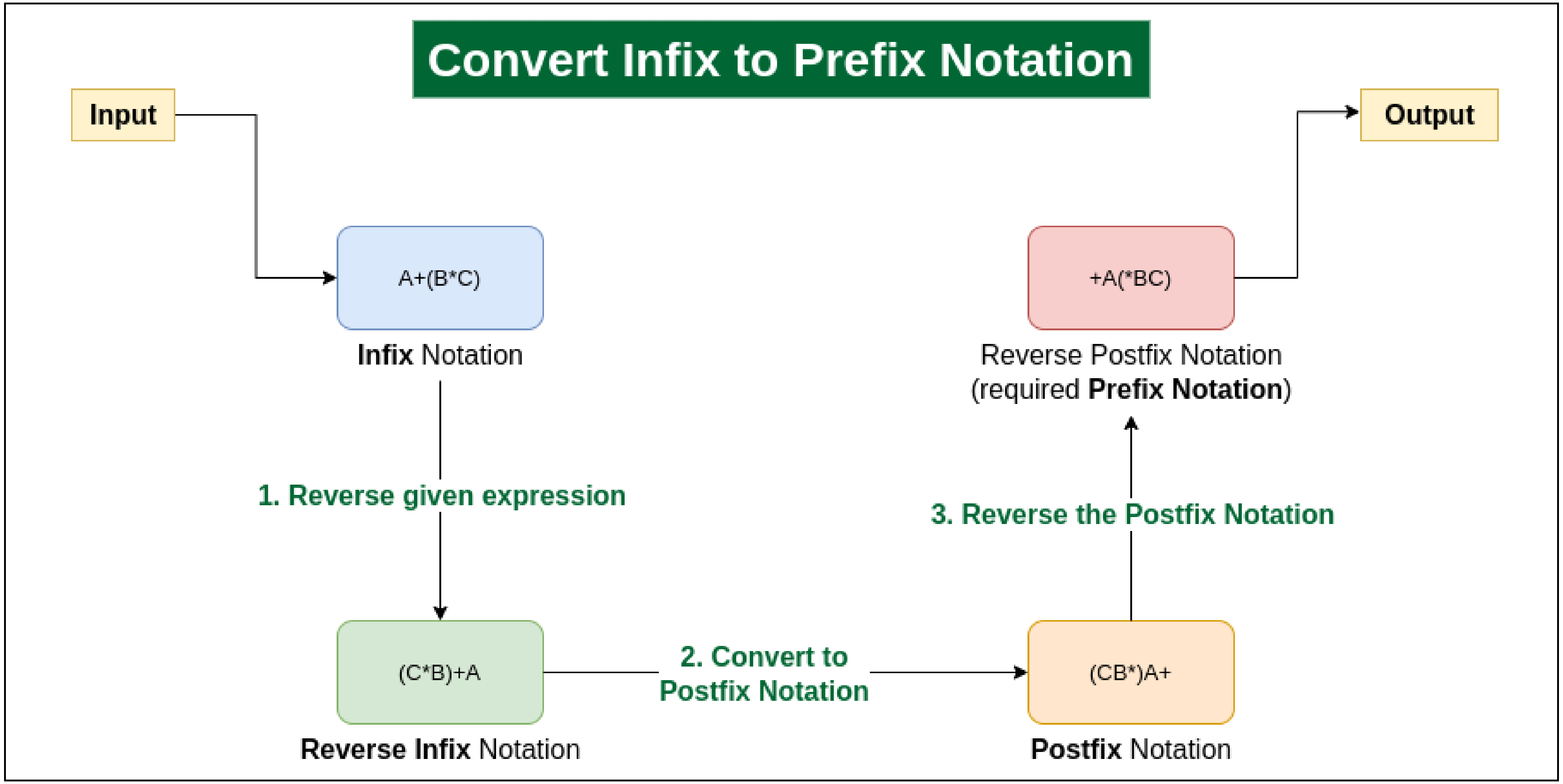
- At the end, reverse the output.

Method 2:

- Step 1: Reverse the infix expression.
  Note while reversing each '(' will become ')' and each ')' becomes '('.
- Step 2: Convert the reversed infix expression to "nearly" postfix expression.
  While converting to postfix expression, we will only pop the operators from stack that have greater precedence.
  - If the operator is '(',
    push the operator on the stack.

  - If the top of the stack is ')',
    pop all the operators from the stack till it finds closing bracket in the stack.

- Step 3: Reverse the postfix expression.


Convert Infix to Prefix Notation
Input
Output
A+(B*C)
Infix Notation
+A(*BC)
Reverse Postfix Notation (required Prefix Notation)
1. Reverse given expression
3. Reverse the Postfix Notation
(C*B)+A
2. Convert to Postfix Notation
(CB*)A+
Reverse Infix Notation
Postfix Notation

EXERCISE

1) Convert the following infix expressions to prefix expressions.
   a) A + B – C
   b) A + B * C
   c) (A + B) / (C – D)
   d) ((A + B) * (C – D) + E) / (F + G)
   e) 2 * 3 / (2 – 1) + 5 * 3
   f) 8 – 2 + (3 * 4) / 2 ^ 2
   g) A + B * C / D – F + A ^ E

PREFIX TO POSTFIX CONVERSION

- To convert a prefix to postfix expression, use a stack to hold the operands.

- Whenever an operator is found,
  we pop two operands from the stack and push a new operand.

- The final element at the top of the stack will be postfix expression.

PREFIX TO POSTFIX CONVERSION USING STACK

Algorithm for Prefix to Postfix:

- Read the Prefix expression in reverse order (from right to left)
- If the symbol is an operand, then
  push it onto the Stack
- If the symbol is an operator,
  then pop two operands from the Stack
  Create a string by concatenating the two operands and the operator after them.
  string = operand1 + operand2 + operator
  Then push the resultant string back to Stack
- Repeat the above steps until end of Prefix expression.

EXERCISE

1) Convert the following prefix expressions to postfix expressions.
   a) *-a/bc-/ade

POSTFIX TO PREFIX CONVERSION

- There are two ways of converting a postfix into a prefix expression:
    1. Conversion of Postfix to Prefix expression manually.
    2. Conversion of Postfix to Prefix expression using stack.

<u>Conversion of Postfix to Prefix expression manually</u>

- The following are the steps required to convert postfix into prefix expression:
    1. Scan the postfix expression from left to right.
    2. Select the first two operands from the expression followed by one operator.
    3. Convert it into the prefix format.
    4. Substitute the prefix sub expression by one temporary variable
    5. Repeat this process until the entire postfix expression is converted into prefix expression.

POSTFIX TO PREFIX CONVERSION USING STACK

Algorithm for Prefix to Postfix:

- Read the postfix expression from left to right

- If the symbol is an operand, then

  push it onto the Stack

- If the symbol is an operator,

  then pop two operands from the Stack

  Create a string by concatenating the two operands and the operator before them.

  string = operator + operand2 + operand1

  Then push the resultant string back to Stack

- Repeat the above steps until end of Postfix expression.

EXERCISE

1) Convert the following postfix expressions to prefix expressions.
   a) AB+CD-
   b) *+AB-CD
   c) abc/-ad/e-*

POSTFIX TO INFIX CONVERSION

- Scan the given postfix expression from left to right character by character.
- If the character is an operand, push it into the stack.
- But if the character is an operator, pop the top two values from stack.
- Concatenate this operator with these two values to get a new string.
  2nd top value + operator + 1st top value
- Now push this resulting string back into the stack.
- Repeat this process until the end of postfix expression.

Now the value in the stack is the infix expression.

EXERCISE

1) Convert the following postfix expressions to prefix expressions.
   a) AB+CD-
   b) *+AB-CD
   c) abc/-ad/e-*

PREFIX TO INFIX CONVERSION

- In prefix to infix conversion using stack involves rearranging the operators and operands of the prefix expression to a format where the operators appear between the operands.

- This can be achieved by recursively evaluating the expression and adding parentheses around the sub-expressions as needed to maintain the correct order of operations.

- The resulting infix expression will have the same meaning as the original prefix expression but in a more traditional format.

PREFIX TO INFIX CONVERSION USING STACK

Algorithm for Prefix to Infix:

- Read the Prefix expression in reverse order (from right to left)

- If the symbol is an operand, then push it onto the Stack

- If the symbol is an operator, then pop two operands from the Stack
  Create a string by concatenating the two operands and the operator between them.
  string = operand1 + operator + operand2
  Then push the resultant string back to Stack

- Repeat the above steps until the end of Prefix expression.

- At the end stack will have only resultant string

EXERCISE

1) Convert the following prefix expressions to postfix expressions.
   a) *+AB-CD
   b) *-A/BC-/AKL

EVALUATION OF PREFIX

```
Step 1 :   Put a pointer P at the end of the end

Step 2 :   If character at P is an operand push it to Stack

Step 3 :   If the character at P is an operator,
               pop two elements from the Stack.
               Operate on these elements according to the operator,
               and push the result back to the Stack

Step 4 :   Decrement P by 1 and go to Step 2 as long as there are characters left
           to be scanned in the expression

Step 5 :   The Result is stored at the top of the Stack, return it

Step 6 :   End
```

NOTE

In the prefix evaluation, while applying the operator,

we placed the first popped character at the first position,

then the operator,

and then the second popped element.

EXAMPLE:

Prefix Expression : -/*2*5+3652

Reversed Prefix Expression: 2563+5*2*/-

Token	Action	Stack
2	Push **2** to stack	[2]
5	Push **5** to stack	[2, 5]
6	Push **6** to stack	[2, 5, 6]
3	Push **3** to stack	[2, 5, 6, 3]
+	Pop **3** from stack	[2, 5, 6]
	Pop **6** from stack	[2, 5]
	Push **3+6 =9** to stack	[2, 5, 9]
5	Push **5** to stack	[2, 5, 9, 5]
*	Pop **5** from stack	[2, 5, 9]
	Pop **9** from stack	[2, 5]
	Push **5*9=45** to stack	[2, 5, 45]
2	Push **2** to stack	[2, 5, 45, 2]
*	Pop **2** from stack	[2, 5, 45]
	Pop **45** from stack	[2, 5]
	Push **2*45=90** to stack	[2, 5, 90]
/	Pop **5** from stack	[2, 5]
	Pop **90** from stack	[2]
	Push **90/5=18** to stack	[2, 18]
-	Pop **18** from stack	[2]
	Pop **2** from stack	[]
	Push **18-2=16** to stack	[16]

Result : 16

EVALUATION OF POSTFIX

Step 1 : Scan the expression from left to right and keep on storing the operands into a stack.

Step 2 : If ,character is an operand received,
place it on the stack

Step 3 : If character is an operator is received,
pop the two topmost elements
evaluate them
and push the result in the stack again.

Step 4 : Decrement P by 1 and go to Step 2 as long as there are characters left to be scanned in the expression

Step 5 : The Result is stored at the top of the Stack, return it

NOTE

In the prefix evaluation, while applying the operator,

we placed the first popped character at the first position,

then the operator,

and then the second popped element.

EXAMPLE:

Postfix Expression : 2536+5/2-**

Token	Action	Stack
2	Push **2** to stack	[2]
5	Push **5** to stack	[2, 5]
3	Push **3** to stack	[2, 5, 3]
6	Push **6** to stack	[2, 5, 3, 6]
+	Pop **6** from stack	[2, 5, 3]
	Pop **3** from stack	[2, 5]
	Push **3+6 =9** to stack	[2, 5, 9]
*	Pop **9** from stack	[2, 5]
	Pop **5** from stack	[2]
	Push **5*9=45** to stack	[2, 45]
*	Pop **45** from stack	[2]
	Pop **2** from stack	[]
	Push **2*45=90** to stack	[90]
5	Push **5** to stack	[90, 5]
/	Pop **5** from stack	[90]
	Pop **90** from stack	[]
	Push **90/5=18** to stack	[18]
2	Push **2** to stack	[18, 2]
-	Pop **2** from stack	[18]
	Pop **18** from stack	[]
	Push **18-2=16** to stack	[16]

Result : 16

EXERCISE

1) Evaluate the following prefix expressions and postfix expressions.
   a) + 5 * 4 6
   b) - * + 2 2 3 10
   c) + - * 2 2 / 16 8 5
   d) 2 3 4 * +
   e) 3 8 + 9 8 / -
   f) 3 4 * 2 5 * +
   g) 2 3 1 * + 9 –