Stack

DR. SANGA CHAKI

Contents

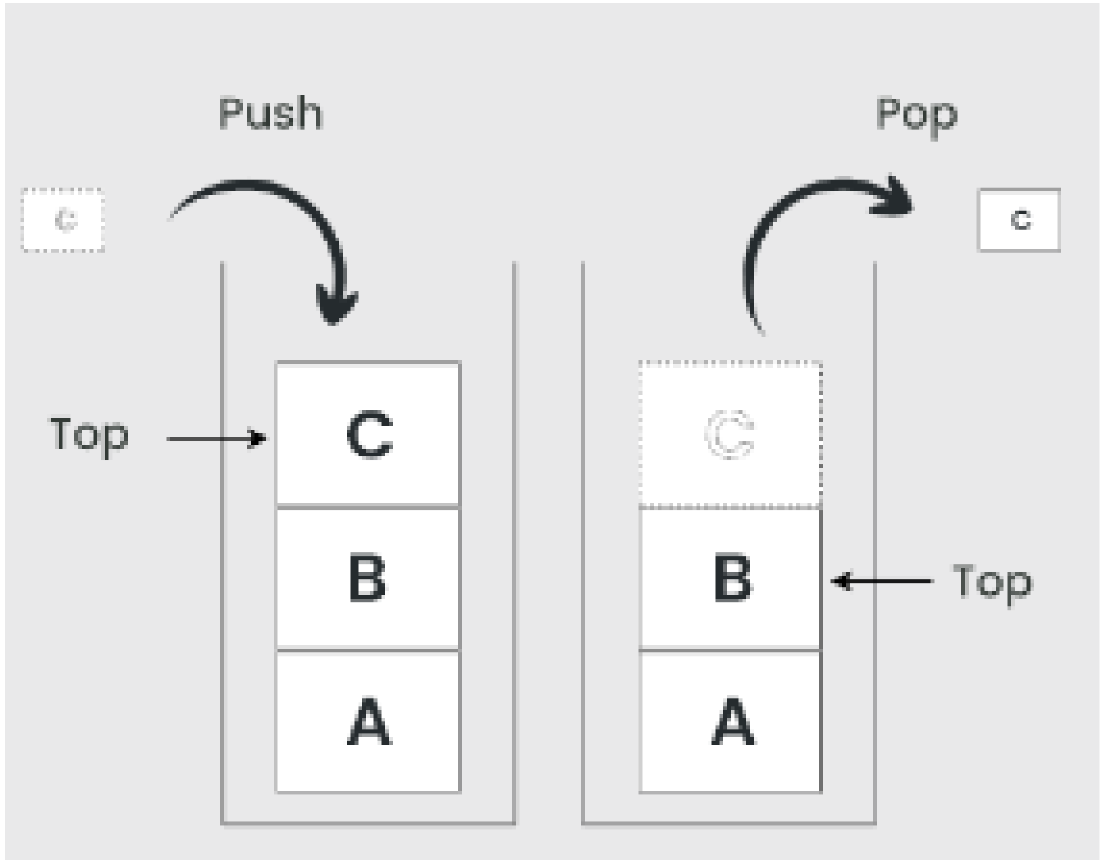
Introduction

What is a Stack?

1. Data structure for storing data
2. The order in which the data arrives is important
3. Its literally like a stack of plates – one stacked on top of the others
4. When a plate a required, it is taken from the top of the pile
5. When a plate needs to be kept in the stack, we do too from the top of the pile
6. The first plate placed on the stack is the last one to be used.

What is a Stack?

1. Definition:
    - A stack is an ordered list, in which insertion and deletion are done at one end which is called as the **top** of the stack.
    - The last element to be inserted is the first one to be deleted.
    - So, it is called a last in first out (LIFO) list.

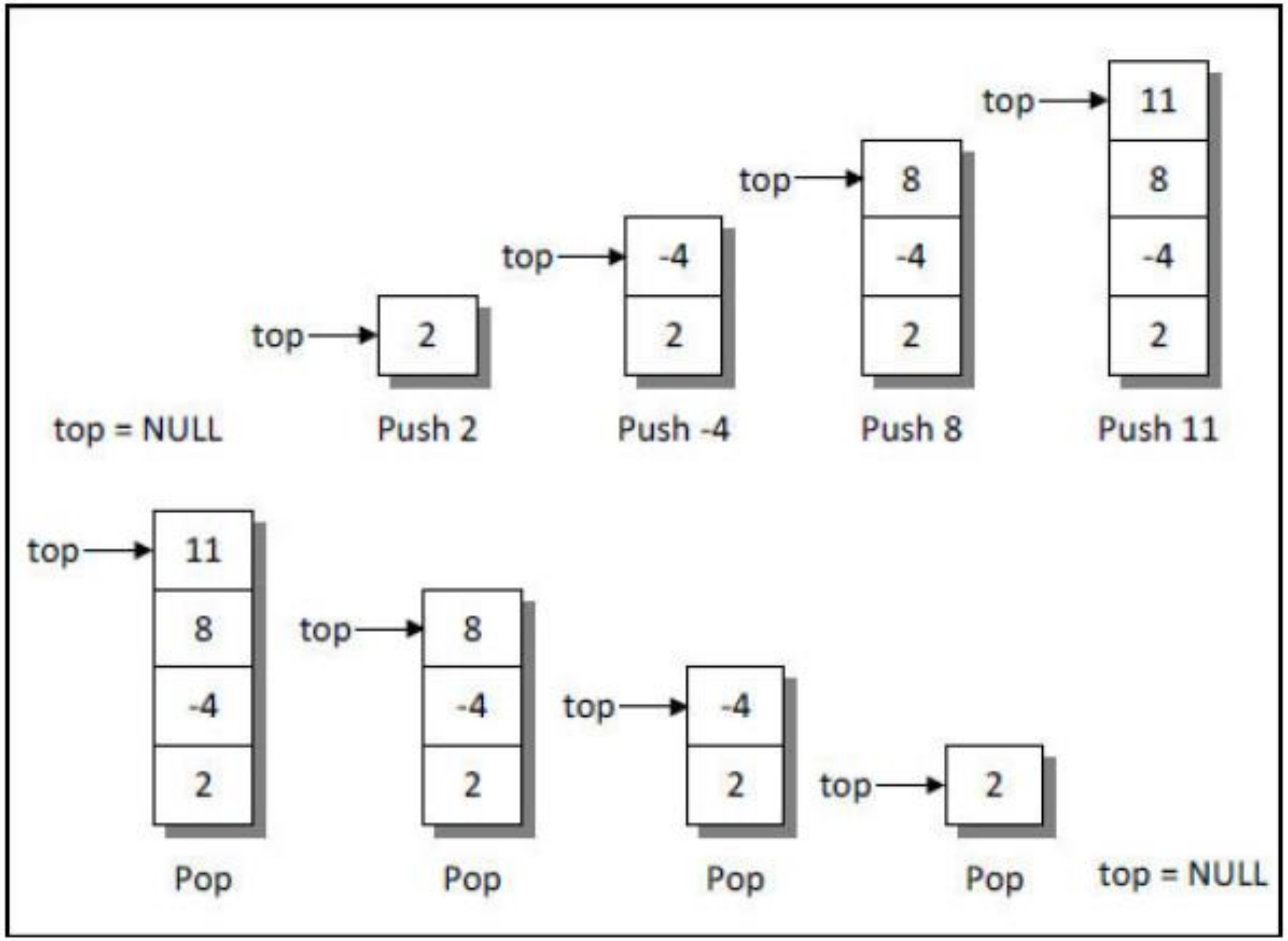


Stack Operations

Basic Stack Operations

1. **Push**: when an element is inserted in a stack
2. **Pop**: when an element is deleted from a stack
3. Trying to pop out an empty stack is underflow
4. Trying to push an element in a full stack is an overflow
5. These are generally exceptions – causing error
6. **Display**: Displays the stack contents
7. **Isempty**: Checks id stack is empty
8. **Isfull**: Checks if stack is full
9. All these operations can be performed using either arrays or linked lists – meaning that stacks can be implemented using either arrays or LLs.

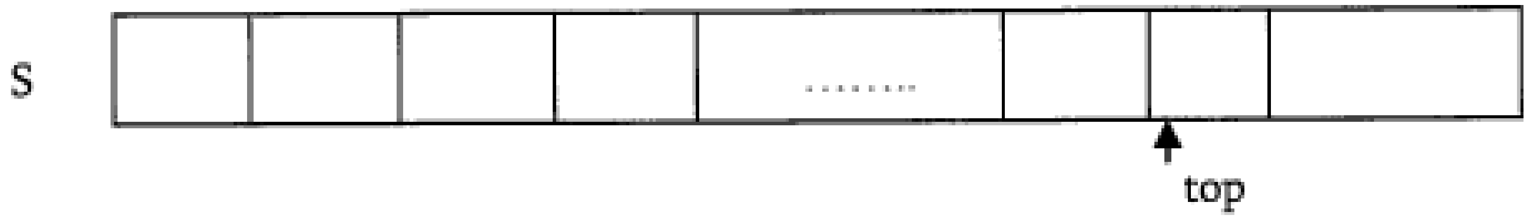
Basic Stack Operations



Array Implementation

Array Implementation of Stacks

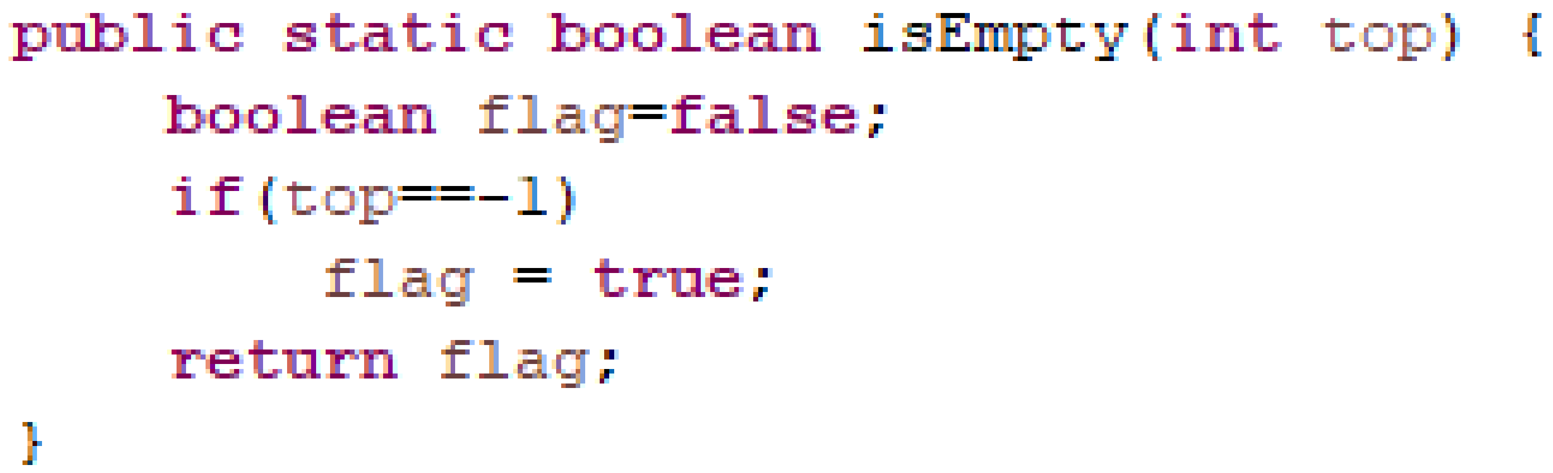
1. Use an array of size MAX to store the elements
2. Insert elements from left to right
3. Use a variable top to keep track of the top element. Initialize top with -1.



4. When the array storing the stack elements is full, a push() operation will throw a full stack exception
5. Similarly, deleting from empty will also throw a stack empty exception
6. So, when we use array to store elements of a stack the stack can grow or shrink within the memory reserved for the array

Array Implementation of Stacks: isEmpty

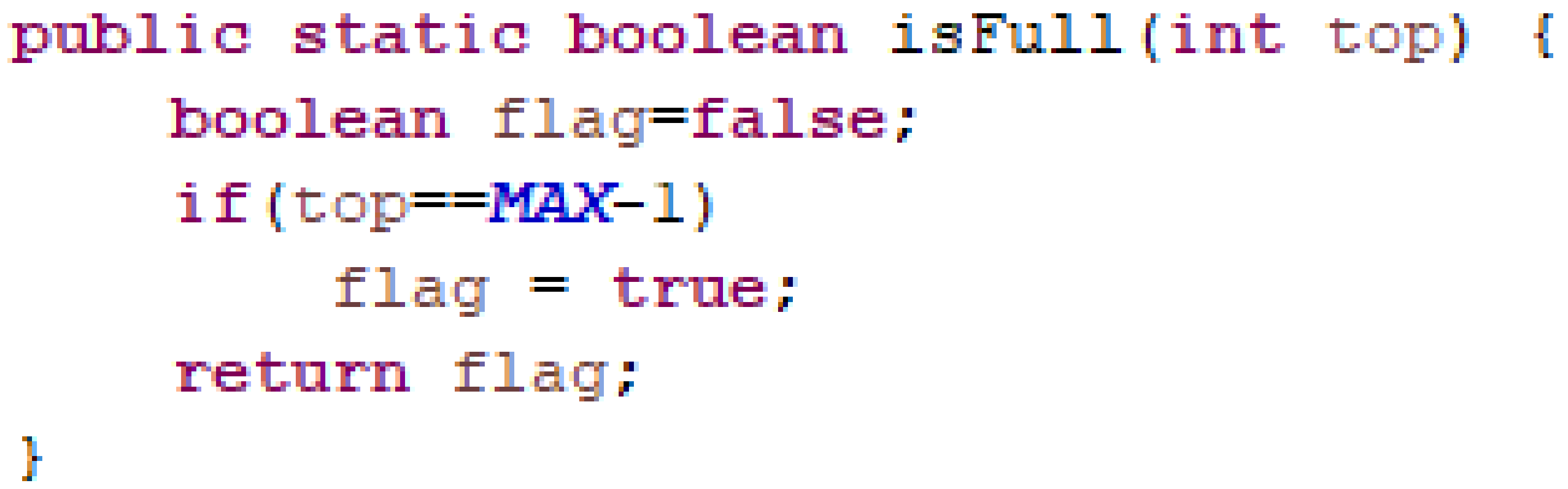
1. Check value of top variable.
2. If it is -1, stack is empty
3. Else stack has elements



```
public static boolean isEmpty(int top)  {
        boolean flag=false;
        if(top==-1)
                flag = true;
        return flag;
}
```

Array Implementation of Stacks: isFull

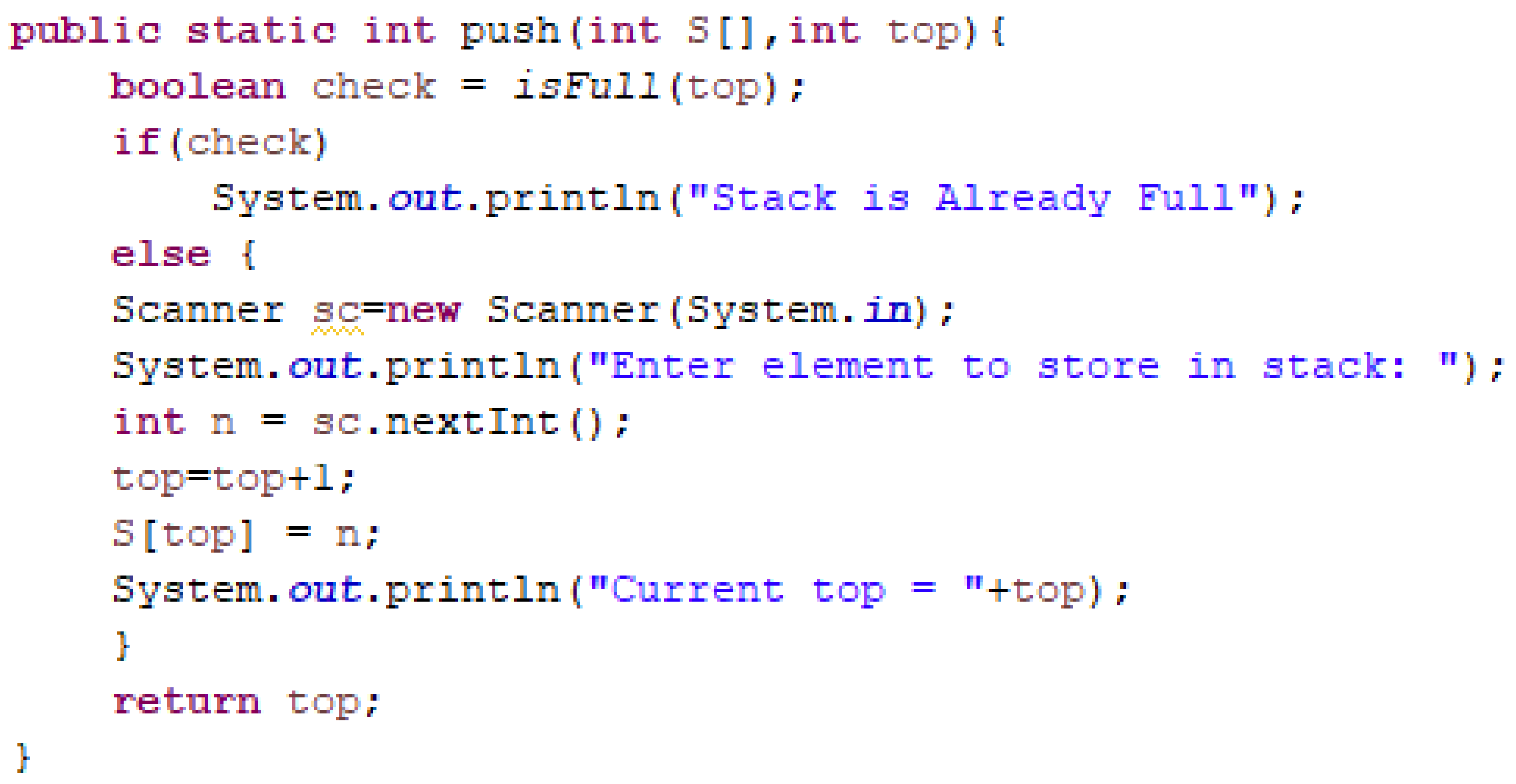
1. Check value of top variable.
2. If it is MAX-1, stack is full
3. Else it is not full



```
public static boolean isFull(int top) {
    boolean flag=false;
    if(top==MAX-1)
        flag = true;
    return flag;
}
```

Array Implementation of Stacks: Push

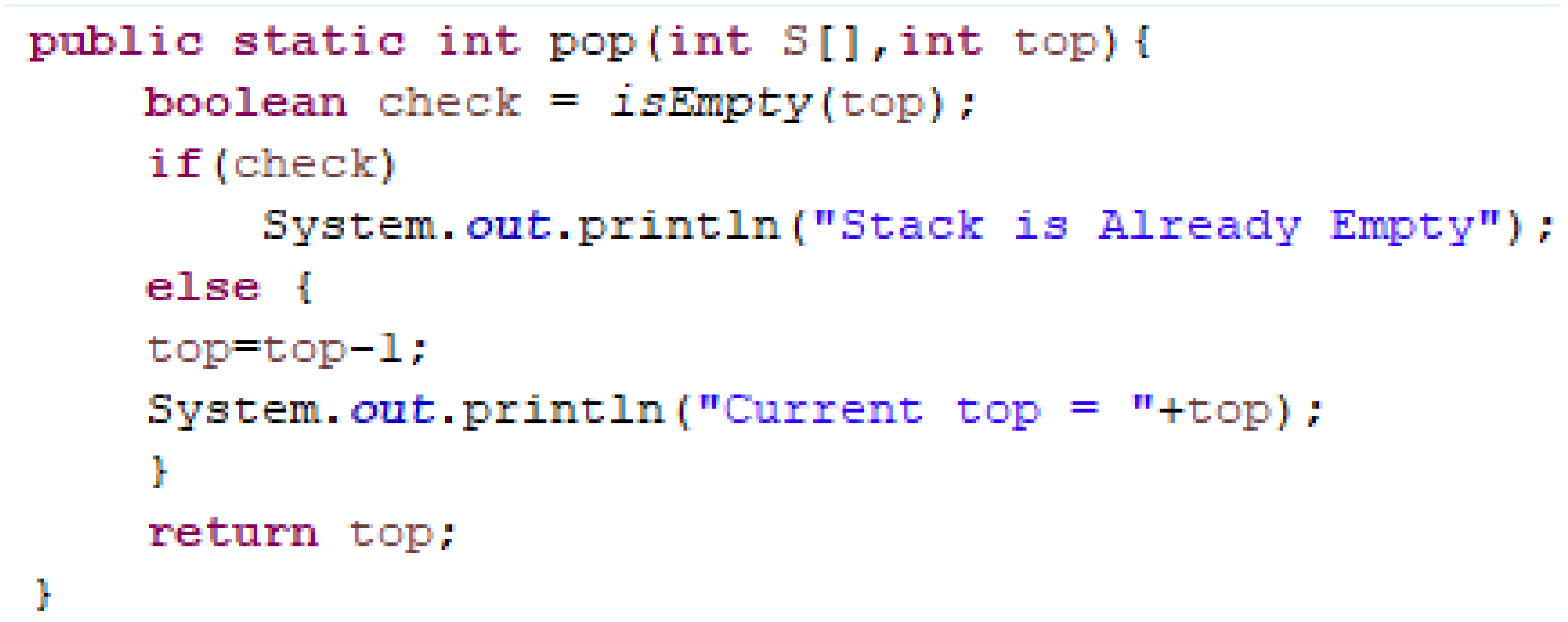
1. Store the element to push into array.
2. Check if Stack isFull()
   - Cannot add more elements in stack
3. Else
   a) Increment top as top = top+1.
   b) Add element to the position Stack[top]=num.
4. Return top



```java
public static int push(int S[],int top){
    boolean check = isFull(top);
    if(check)
        System.out.println("Stack is Already Full");
    else {
    Scanner sc=new Scanner(System.in);
    System.out.println("Enter element to store in stack: ");
    int n = sc.nextInt();
    top=top+1;
    S[top] = n;
    System.out.println("Current top = "+top);
    }
    return top;
}
```

Array Implementation of Stacks: Pop

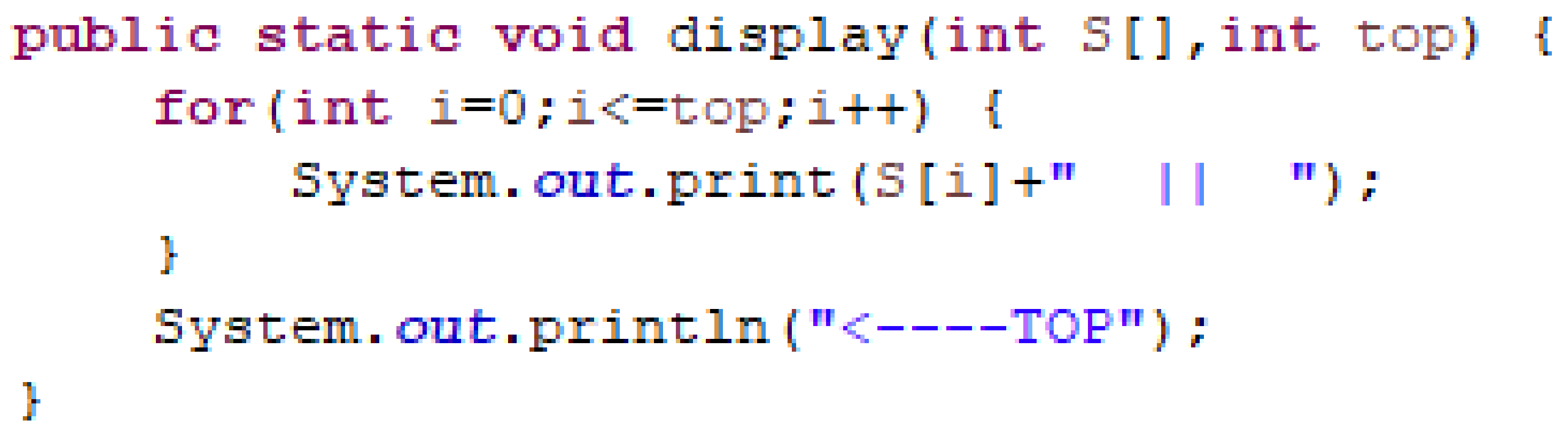
1. First, check if the stack is empty.
2. If empty, print Stack underflow and exit.
3. If not empty, update top = top - 1.
4. Return top



```
public static int pop(int S[],int top){
    boolean check = isEmpty(top);
    if(check)
        System.out.println("Stack is Already Empty");
    else {
    top=top-1;
    System.out.println("Current top = "+top);
    }
    return top;
}
```

Array Implementation of Stacks: Display

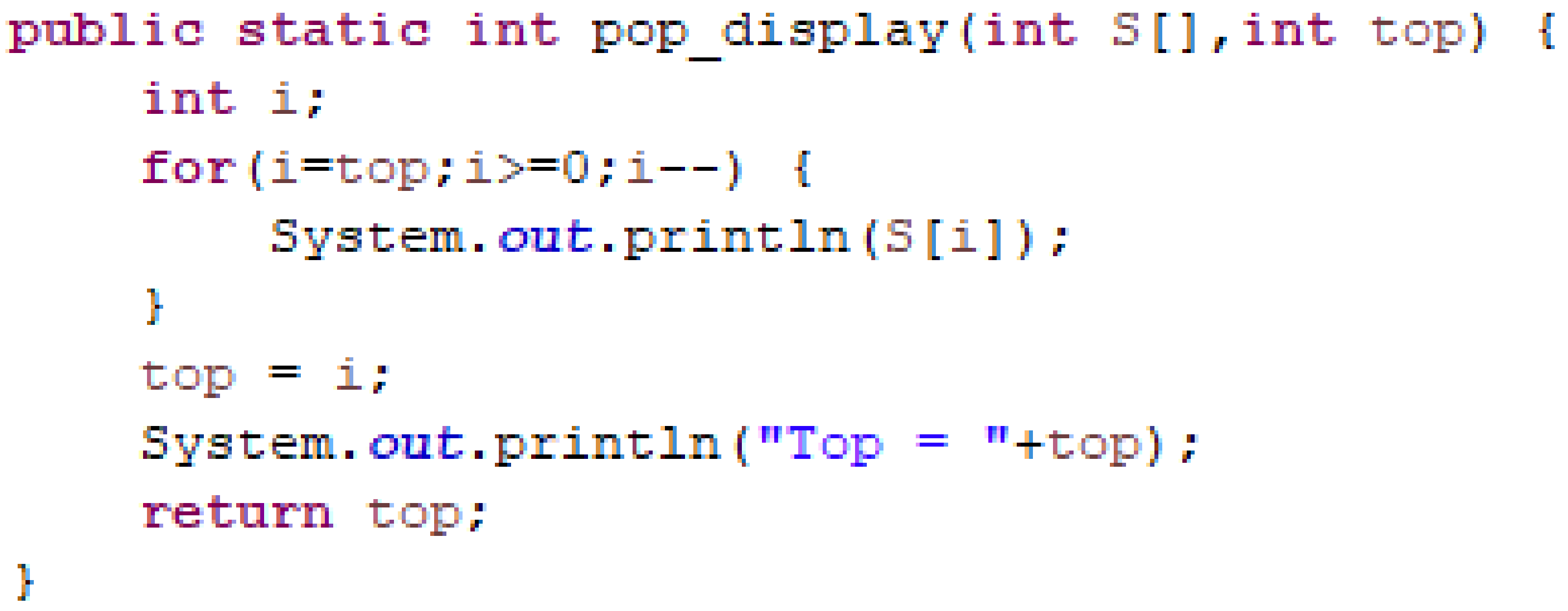
1. Access all elements of stack one by one and print them, indicating top at last



```
public static void display(int S[],int top) {
    for(int i=0;i<=top;i++) {
        System.out.print(S[i]+"  ||  ");
    }
    System.out.println("<----TOP");
}
```

Array in Stacks: Pop and Display

1. Access all elements of stack from top and display them one by one while updating the top each time

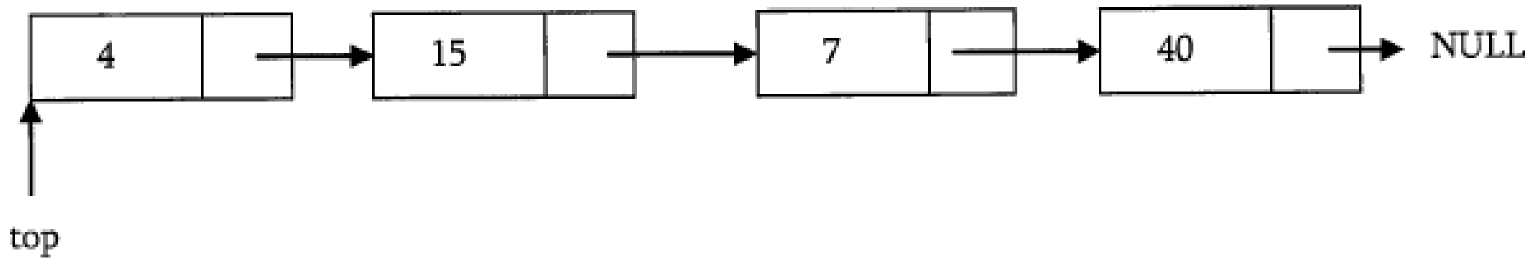


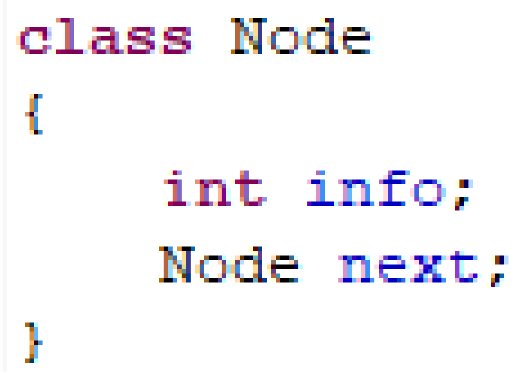
```
public static int pop_display(int S[],int top) {
    int i;
    for(i=top;i>=0;i--) {
        System.out.println(S[i]);
    }
    top = i;
    System.out.println("Top = "+top);
    return top;
}
```

LL Implementation

LL Implementation of Stacks

1. A linked list is an ordered list, and can be used to implement a stack.
2. The problem of fixed size (arrays) can be overcome if we implement a stack using a linked list.
3. The top of the stack is at the head of the linked list.



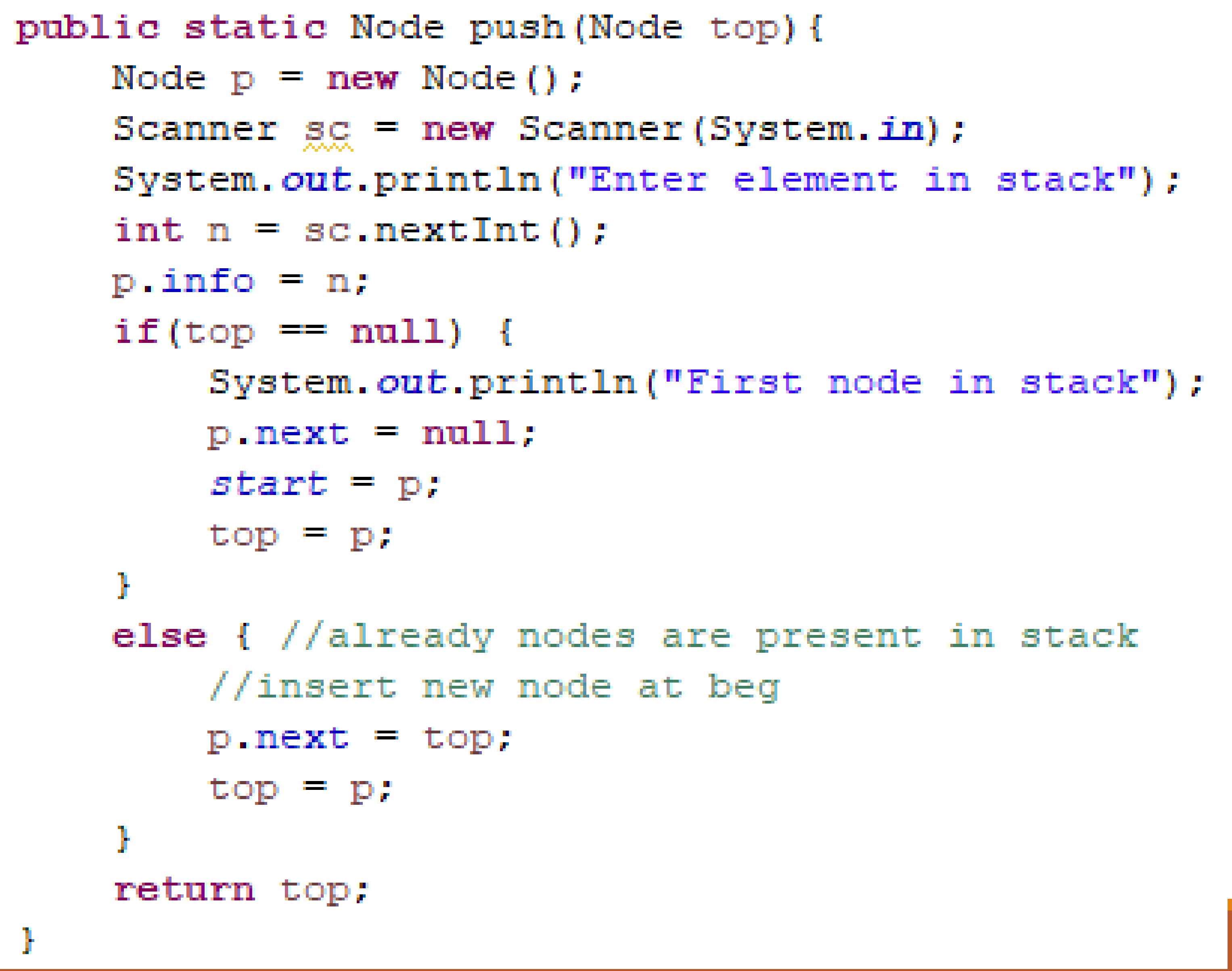


```
class Node
{
      int info;
      Node next;
}
```

4. The pointer to the beginning of the list (head/start) serves the purpose of the top of the stack.
5. So push() and pop() happens at the beginning of the list.

LL Implementation of Stacks: Push

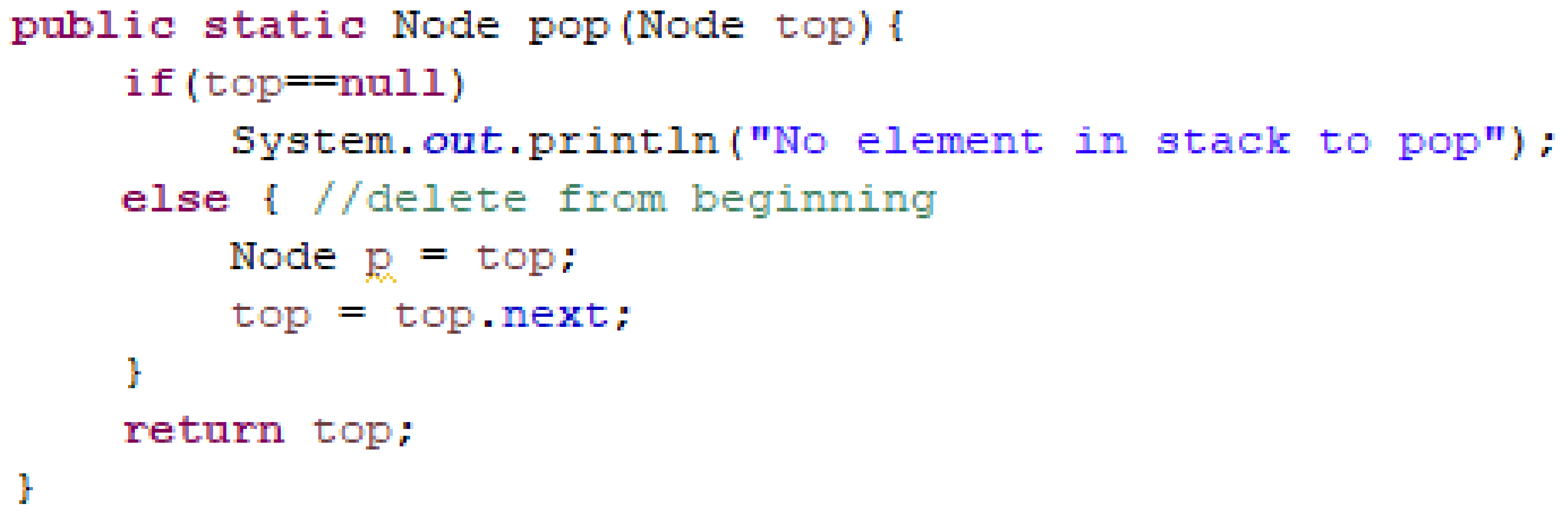
1. Initialize a node
2. Update the value of that node by data
3. Link this node to the current top of the linked list
4. Update top pointer to the current node



```
public static Node push(Node top){
    Node p = new Node();
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter element in stack");
    int n = sc.nextInt();
    p.info = n;
    if(top == null) {
        System.out.println("First node in stack");
        p.next = null;
        start = p;
        top = p;
    }
    else { //already nodes are present in stack
        //insert new node at beg
        p.next = top;
        top = p;
    }
    return top;
}
```

LL Implementation of Stacks: Pop

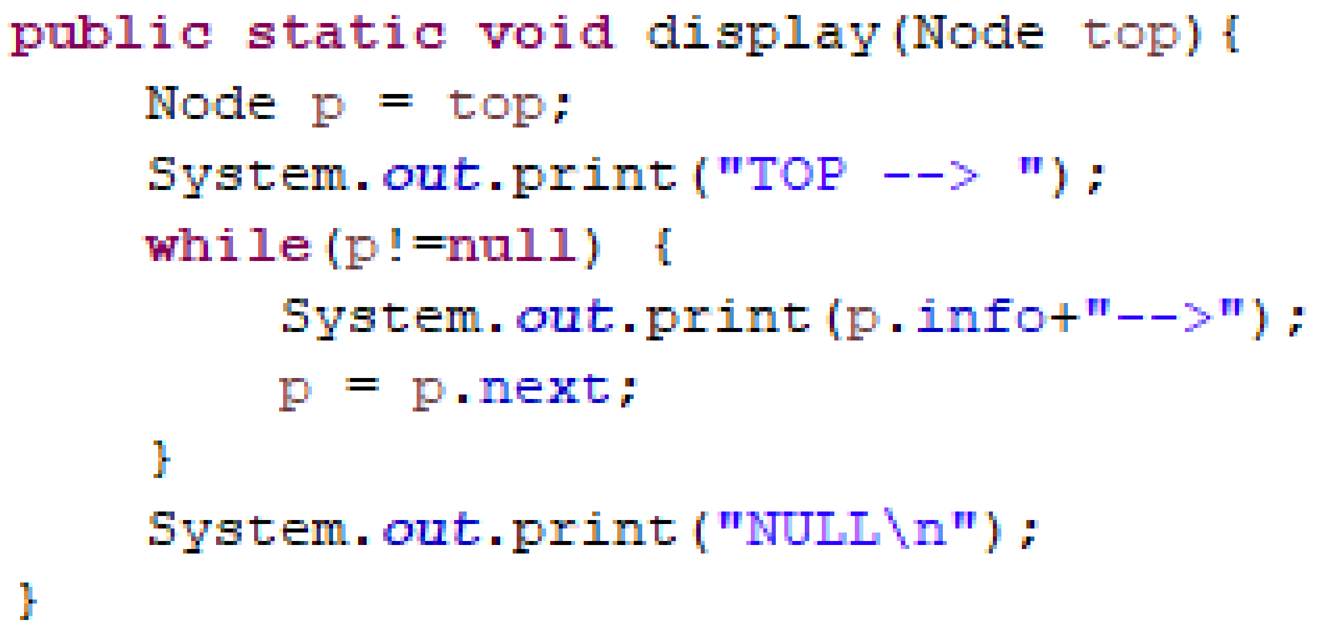
1. Check whether there is any node present in the linked list or not, if not then return
2. Otherwise update
   - top = top.next



```java
public static Node pop(Node top){
    if(top==null)
        System.out.println("No element in stack to pop");
    else { //delete from beginning
        Node p = top;
        top = top.next;
    }
    return top;
}
```

LL Implementation of Stacks: Display

1. Start from top node
2. Traverse each node and print node.data till NULL is reached.



```java
public static void display(Node top){
    Node p = top;
    System.out.print("TOP --> ");
    while(p!=null) {
        System.out.print(p.info+"-->");
        p = p.next;
    }
    System.out.print("NULL\n");
}
```

Stack Applications

Stack Applications

1. Evaluation of Arithmetic Expressions
    a) Postfix expressions
2. Conversion of Arithmetic Expressions
    a) From infix to postfix
    b) From infix to prefix
3. For all these we use stacks

Infix, Prefix and Postfix Expressions

1. An arithmetic expression consists of operands and operators.
2. The operator is placed between two operands is called **infix notation**.
3. Eg: A+B*C
4. How do we decide which operation is to be done first?
5. There are some operator precedence for evaluation: (), *, /, +, -
6. But it is a problem to remember the precedence rules
7. So Polish notation is used in CS

Infix, Prefix and Postfix Expressions

1. Polish notation: As per this notation, an expression in infix form can be converted to either prefix or postfix form and then evaluated.
2. In prefix notation the operator comes before the operands.
3. In postfix notation, the operator follows the two operands.
4. Eg: These forms are shown below.
   - A + B Infix form
   - + A B Prefix form
   - A B + Postfix form

Infix, Prefix and Postfix

1. The prefix and postfix expressions have three features:
   - The operands maintain the same order as in the equivalent infix expression
   - Parentheses are not needed to designate the expression unambiguously.
   - While evaluating the expression the priority of the operators is irrelevant.
2. In all three versions, the operands occur in the same order, and just the operators have to be moved to keep the meaning correct.

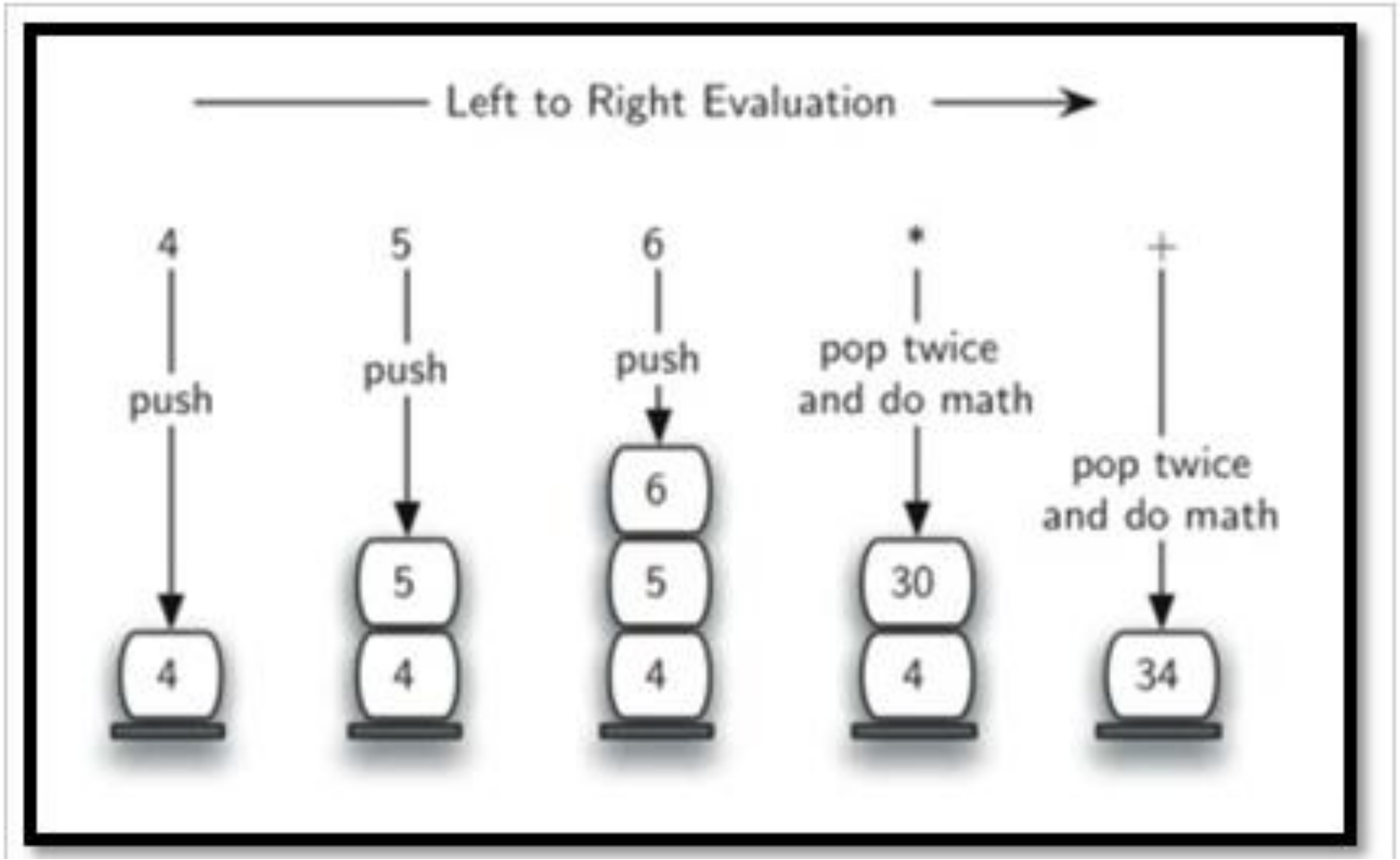
Evaluating Postfix Expression

Evaluating Postfix Expression Algo

1. The postfix expression can be evaluated by
   - Scan expression from left to right
   - Keep on storing the operands into a stack.
   - Once an operator is received, pop the two topmost elements
   - Evaluate them and push the result in the stack again.
2. **The stack contents are the operands**

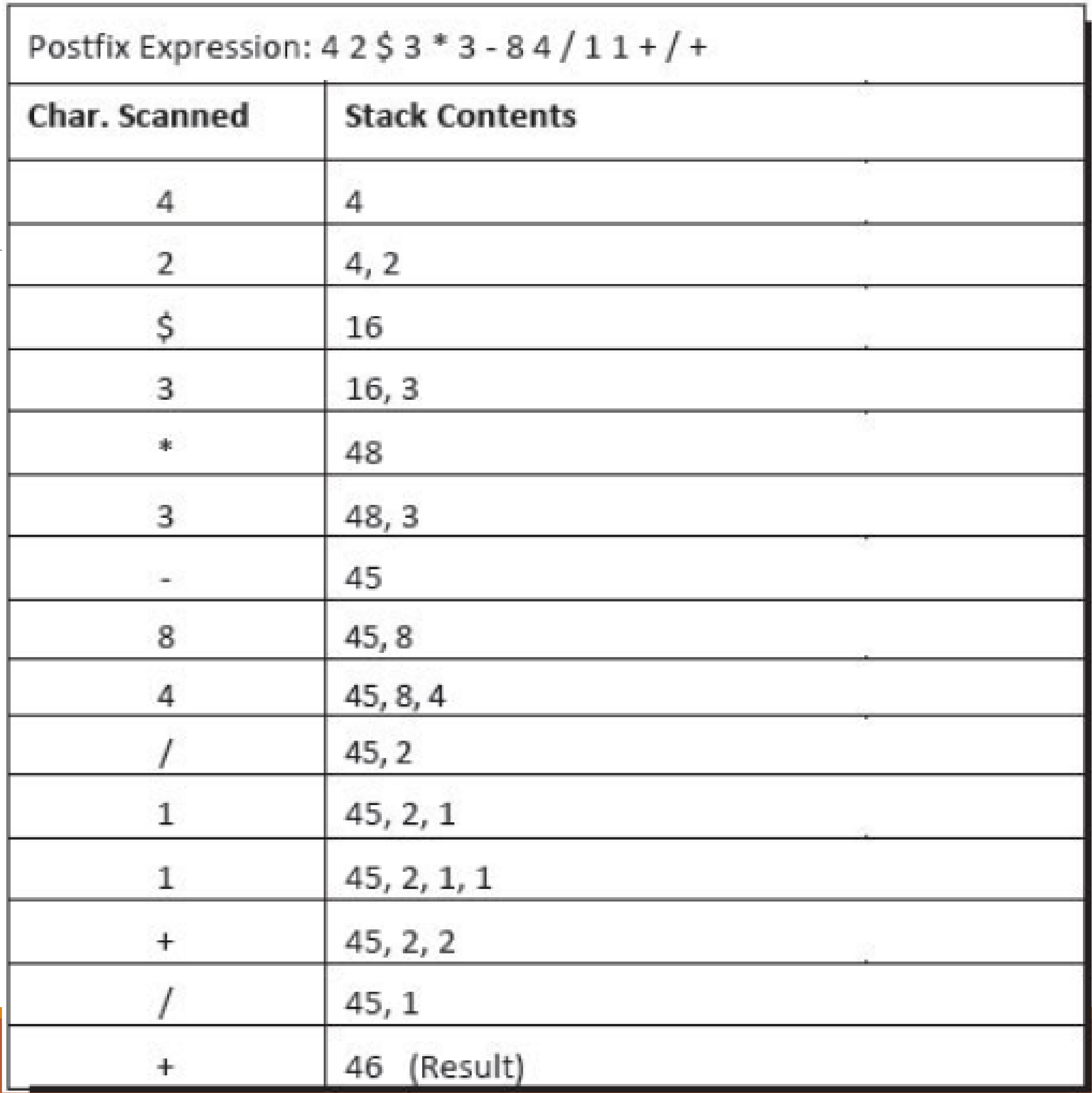
Evaluating Postfix Expression

Expression: 456*+



Evaluating Postfix Expression

1. **Example:**
2. **4 2 $ 3 * 3 - 8 4 / 1 1 + / +**
3. **Result = 46**
4. **How?**
5. **The stack contents are the operands**



Postfix Expression: 4 2 $ 3 * 3 - 8 4 / 1 1 + / +

Char. Scanned	Stack Contents
4	4
2	4, 2
$	16
3	16, 3
*	48
3	48, 3
-	45
8	45, 8
4	45, 8, 4
/	45, 2
1	45, 2, 1
1	45, 2, 1, 1
+	45, 2, 2
/	45, 1
+	46 (Result)

Evaluating Postfix Expression – Example

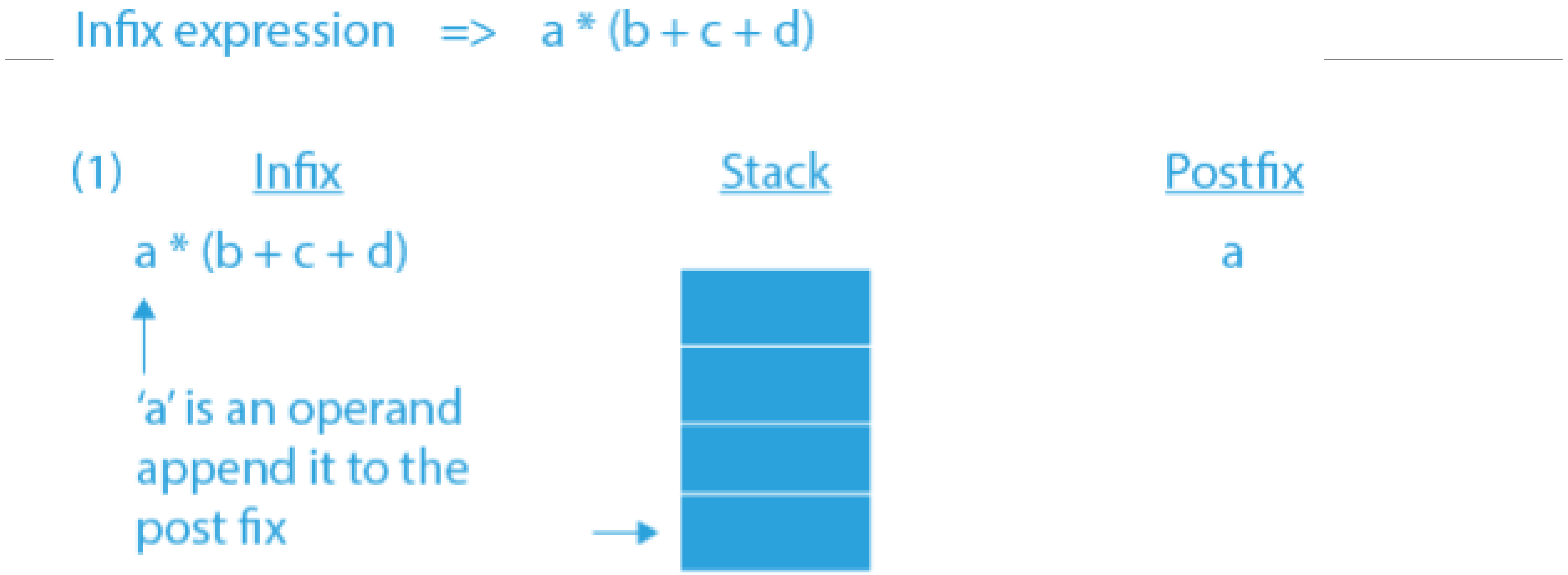
1. Evaluate the following postfix expressions:
2. 2 3 1 * + 9 –   (-4)
3. 100 200 + 2 / 5 * 7 +   (757)
4. 5 4 6 + * 4 9 3 / + *
5. a b c * + d e * f + g * + where a = 1, b= 2, c = 3, d = 4, e = 5, f = 6, g = 2
6. a b + c d / - where a = 5, b = 4, c =9, d = 3

Infix to Postfix Conversion using Stack

Infix to Postfix Conversion using Stack - Algorithm

1. Scan all the symbols one by one from left to right in the given Infix Expression.
2. If symbol is an operand, then immediately append it to the Postfix Expression.
3. If the symbol is left parenthesis ‘(‘, then Push it onto the Stack.
4. If symbol is right parenthesis ‘)’, then Pop all the contents of the stack until the respective left parenthesis is popped and append each popped symbol to Postfix Expression.
5. If symbol is an operator (+, –, *, /), then Push it onto the Stack.
   - However, first, pop the operators which are already on the stack that have higher or equal precedence than the current operator and append them to the postfix.
   - If an open parenthesis is there on top of the stack then push the operator into the stack.
6. If the input is over, pop all the remaining symbols from the stack and append them to the postfix.

Infix to Postfix Conversion using Stack - Examples



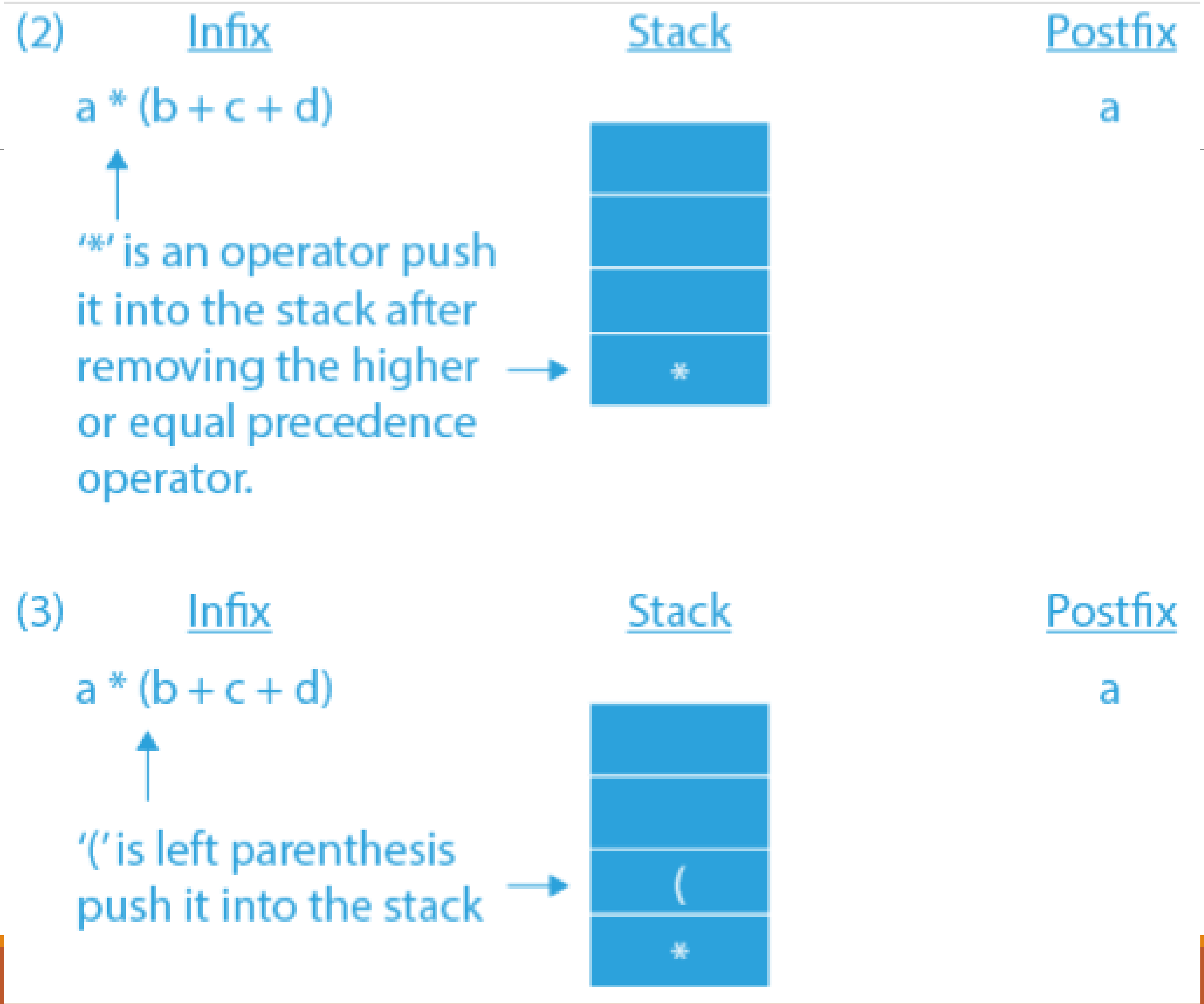
Infix expression => a * (b + c + d)

(1) Infix Stack Postfix

a * (b + c + d) a

'a' is an operand append it to the post fix

Infix to Postfix Conversion using Stack - Examples



(2) Infix Stack Postfix

a * (b + c + d)

'*' is an operator push it into the stack after removing the higher or equal precedence operator.

a

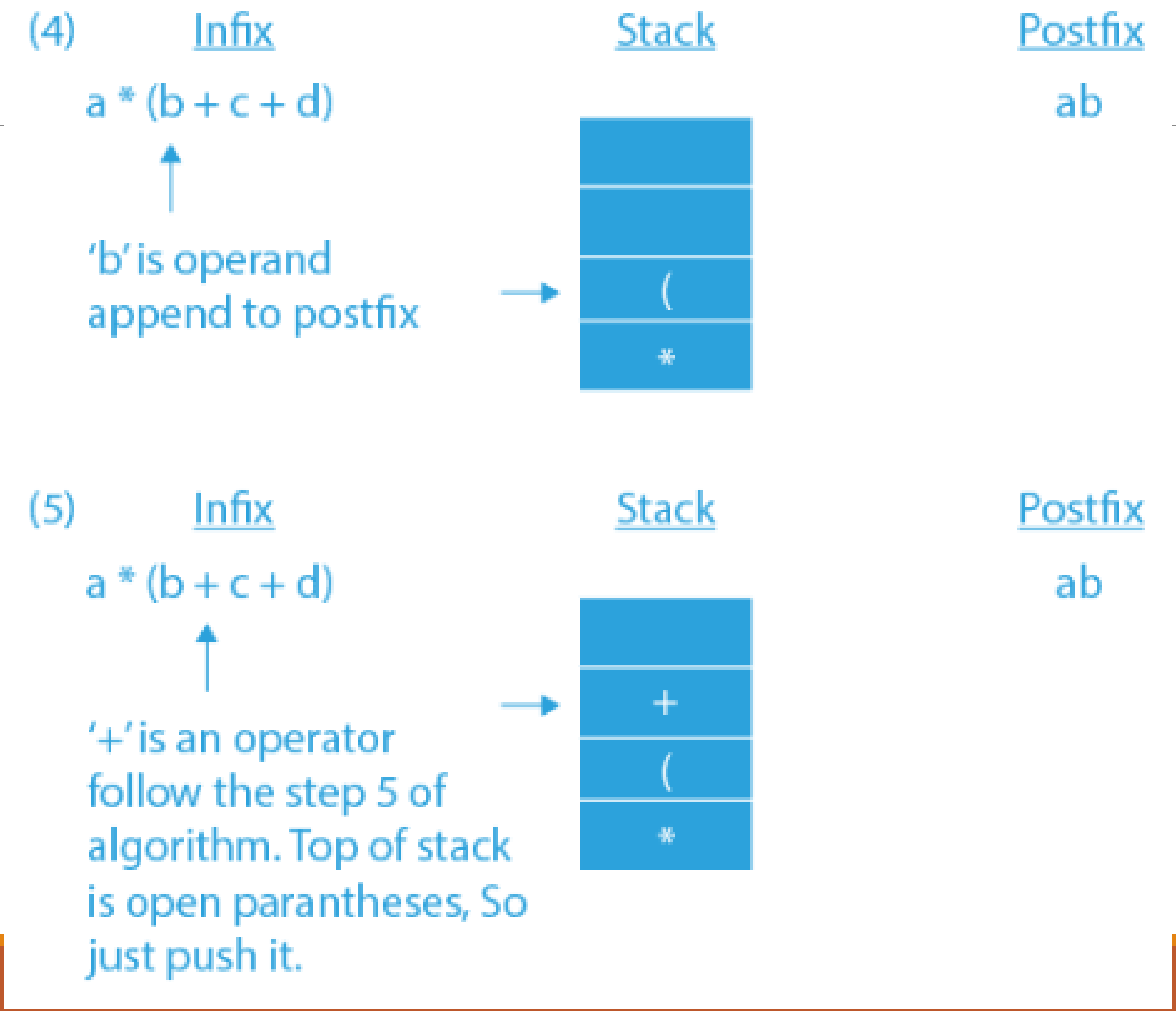
(3) Infix Stack Postfix

a * (b + c + d)

'(' is left parenthesis push it into the stack

a

Infix to Postfix Conversion using Stack - Examples



(4) Infix — Stack — Postfix

a * (b + c + d)

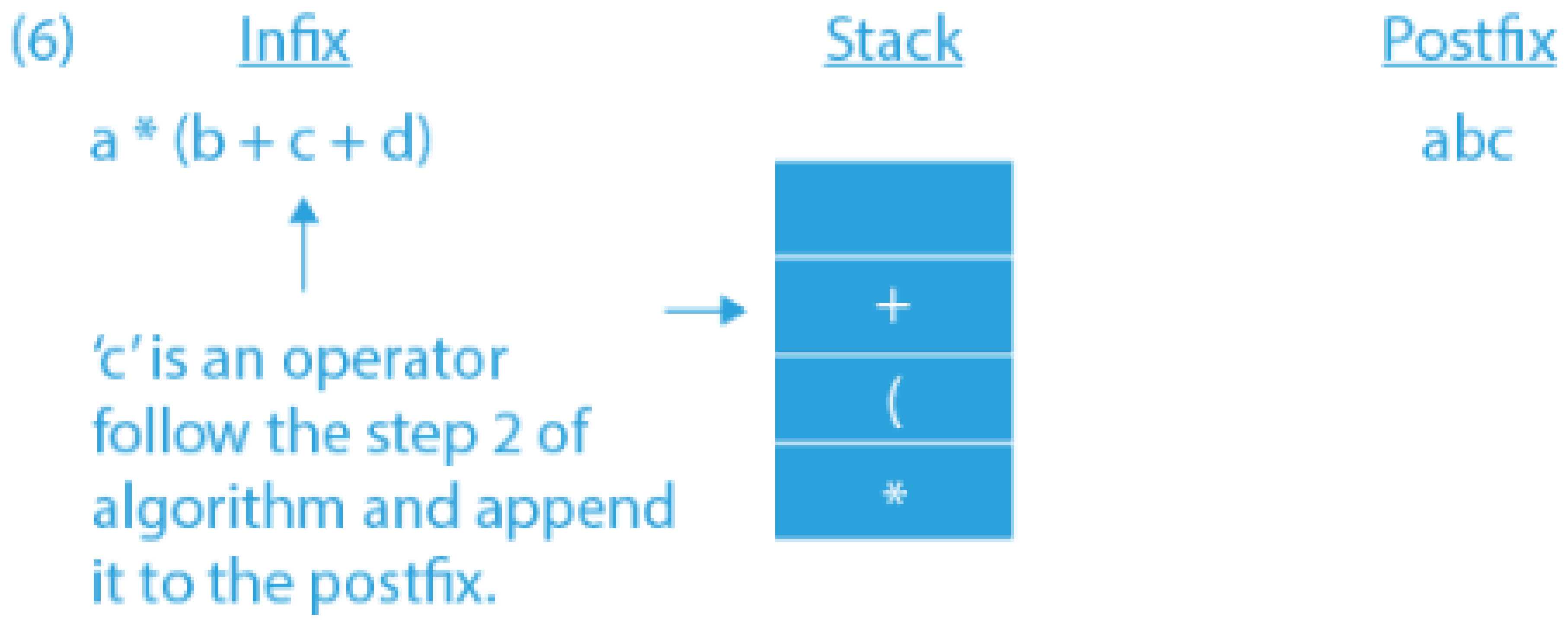
‘b’ is operand append to postfix

(5) Infix — Stack — Postfix

a * (b + c + d)

‘+’ is an operator follow the step 5 of algorithm. Top of stack is open parantheses, So just push it.

Infix to Postfix Conversion using Stack - Examples



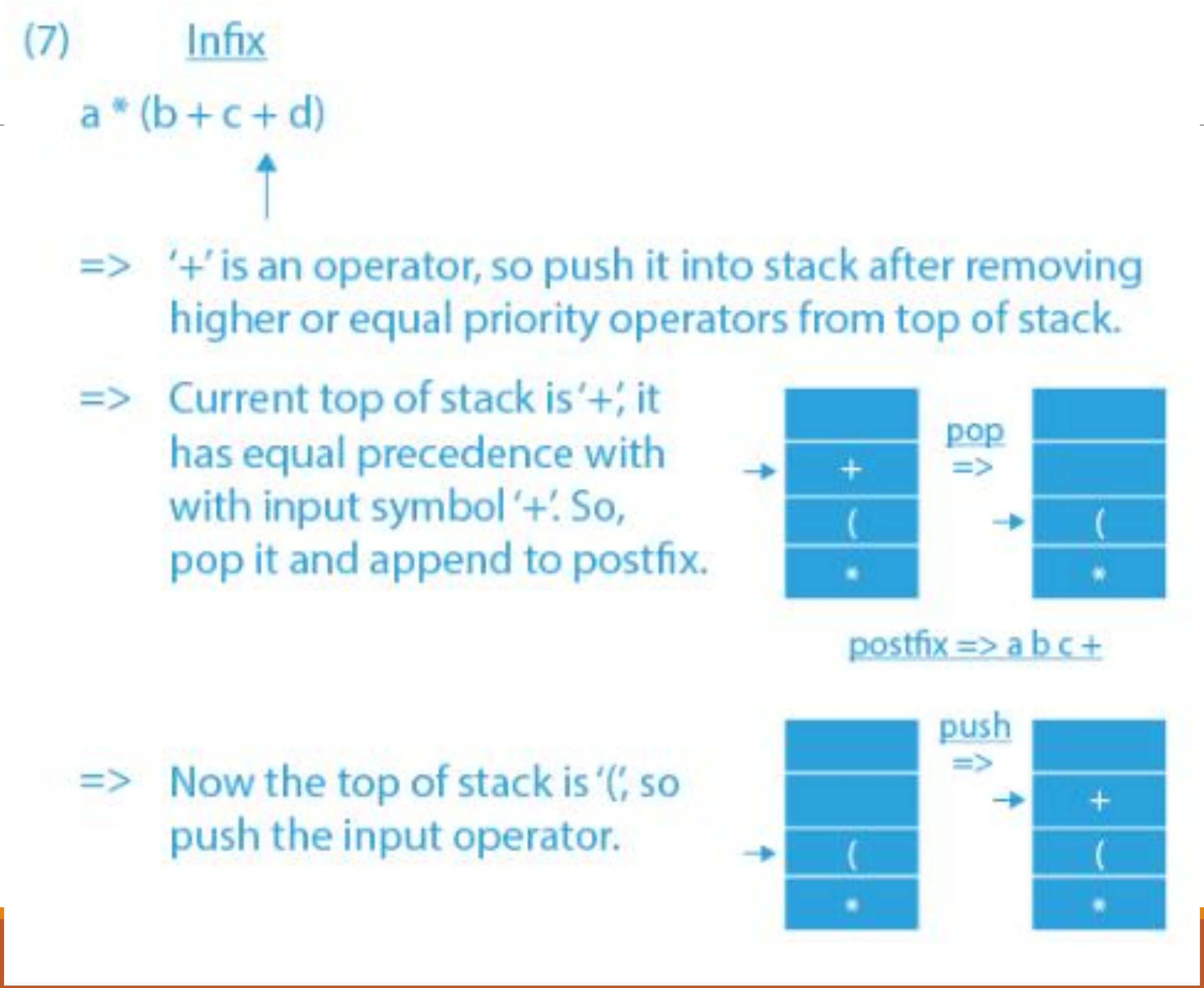
(6) Infix: a * (b + c + d)

Stack

Postfix: abc

'c' is an operator follow the step 2 of algorithm and append it to the postfix.

Infix to Postfix Conversion using Stack - Examples



(7) Infix

a * (b + c + d)

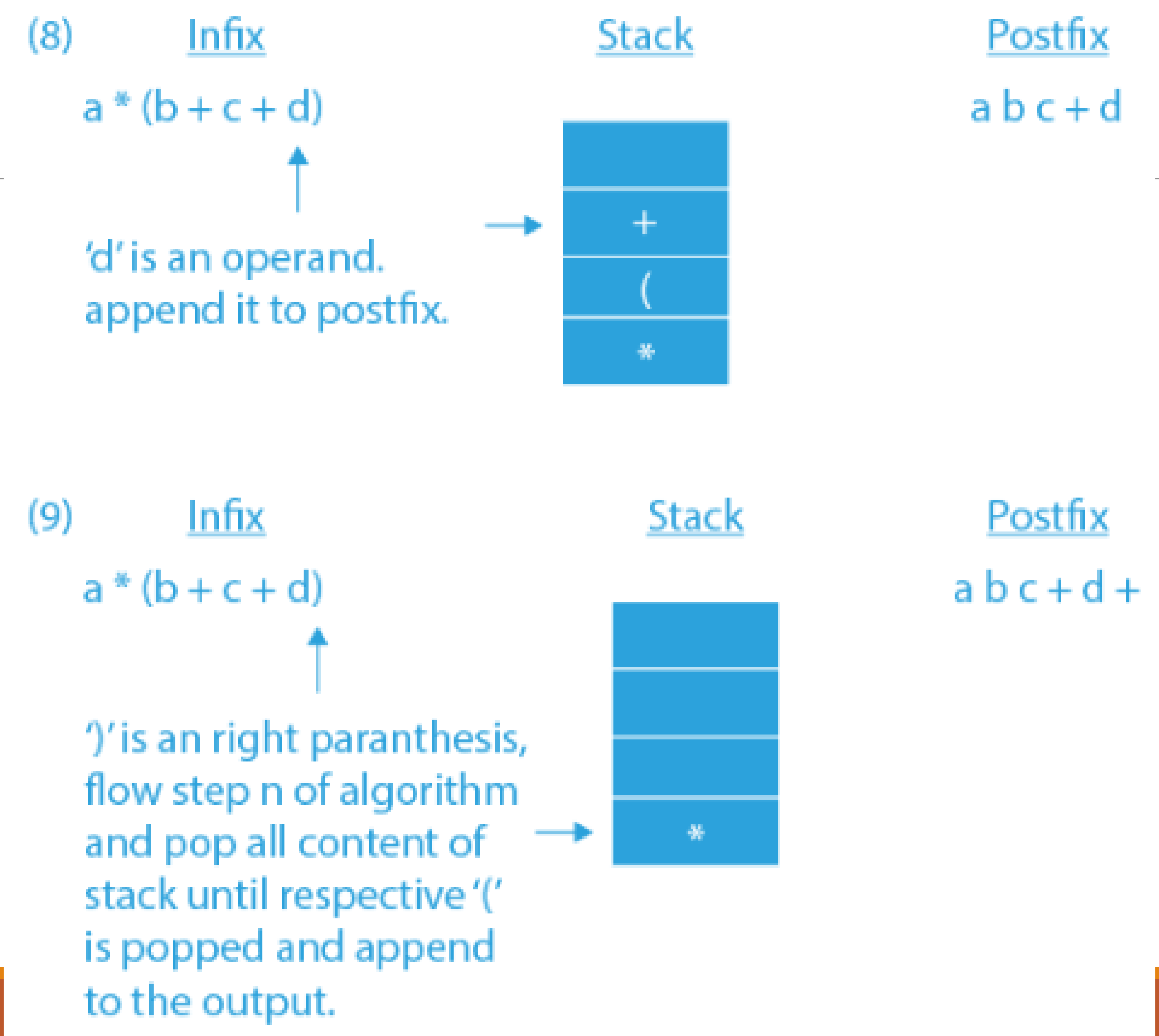
=> '+' is an operator, so push it into stack after removing higher or equal priority operators from top of stack.

=> Current top of stack is '+', it has equal precedence with with input symbol '+'. So, pop it and append to postfix.

postfix => a b c +

=> Now the top of stack is '(', so push the input operator.

Infix to Postfix Conversion using Stack - Examples



(8) Infix

a * (b + c + d)

'd' is an operand.
append it to postfix.

Postfix

a b c + d

(9) Infix

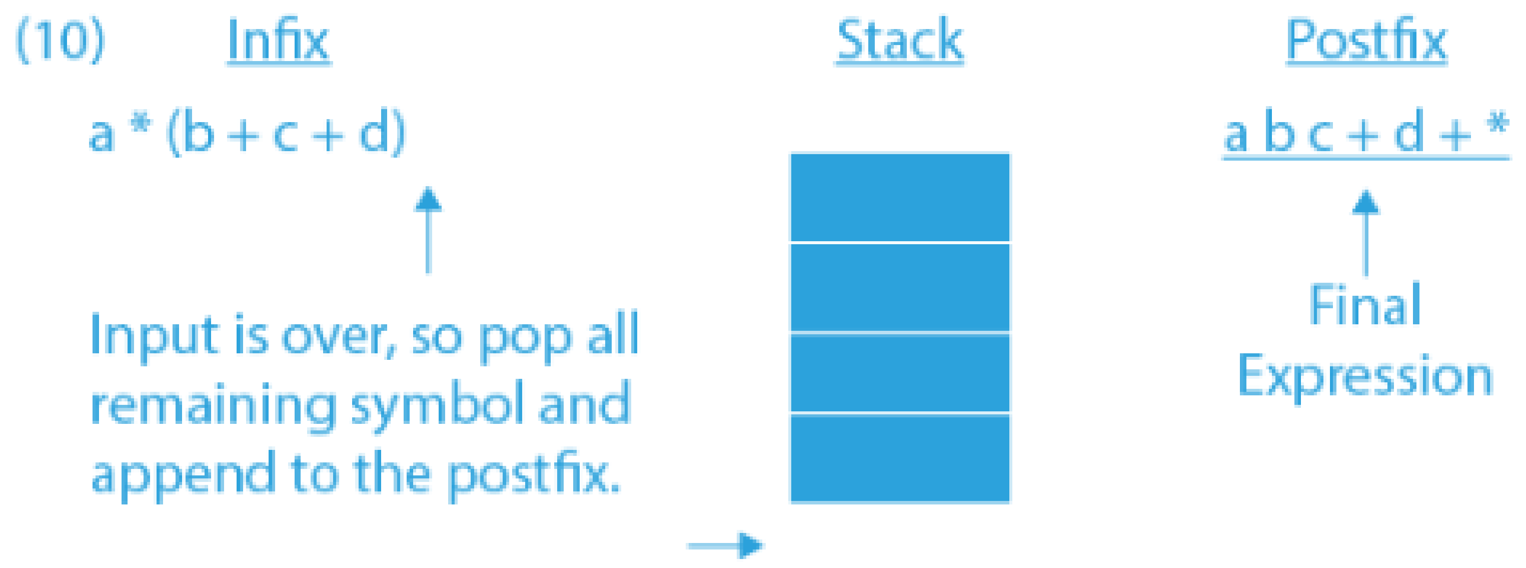
a * (b + c + d)

')' is an right paranthesis,
flow step n of algorithm
and pop all content of
stack until respective '('
is popped and append
to the output.

Postfix

a b c + d +

Infix to Postfix Conversion using Stack - Examples


(10)
Infix
a * (b + c + d)
Input is over, so pop all remaining symbol and append to the postfix.
Stack
Postfix
a b c + d + *
Final Expression

Infix to Postfix Conversion

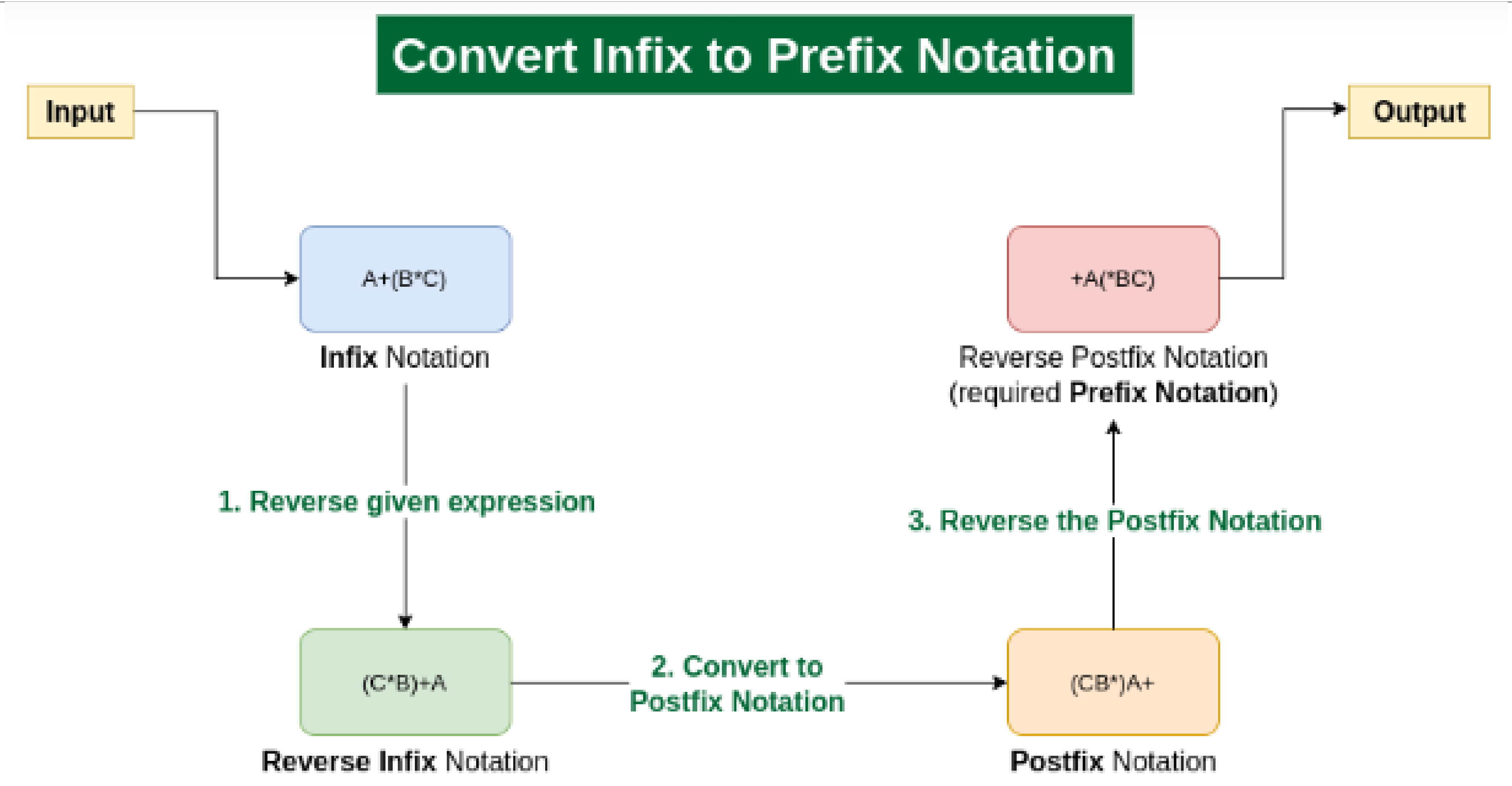
1. 2+3+4
2. (2+3)*(4+5)
3. a/(b+c)
4. a*b+c*d
5. (a+(b*c)/(d-e))
6. 4$2*3-3+8/4/(1+1)

Infix to Prefix Conversion

Infix to Prefix Conversion Algorithm

1. Reverse the infix expression.
   - Note while reversing each ‘(‘ will become ‘)’ and each ‘)’ becomes ‘(‘.
2. Convert the reversed infix expression to “nearly” postfix expression.
   - While converting to postfix expression, instead of using pop operation to pop operators with greater than or equal precedence, here we will only pop the operators from stack that have greater precedence.
3. Reverse the postfix expression to get the prefix notation

Infix to Prefix Conversion Algorithm


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

Thank You