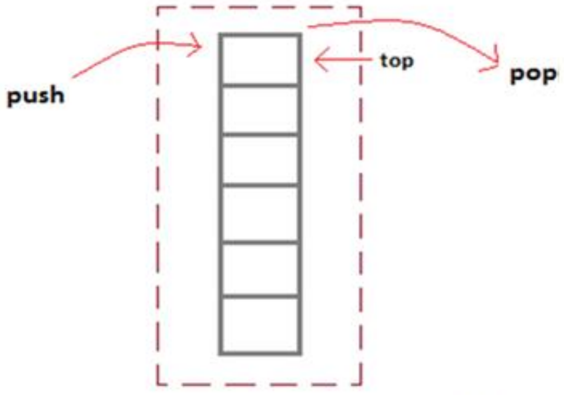
Stack data structure

Stack is **a list of homogeneous items** with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack, the only element that can be removed is the element that was at the top of the stack.

Basic features of Stack

1. Stack is an ordered list of similar data type.
2. Stack is a **LIFO** structure. (Last in First out).
3. **push()** function is used to insert new elements into the Stack and **pop()** function is used to delete an element from the stack. Both insertion and deletion are allowed at only one end of Stack called **Top**.
4. Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty.



Logical Structure

Status of stack

Position of Top	Status of Stack
-1	Stack is Empty
0	Only one element in Stack
N-1	Stack is Full
N	Overflow state of Stack

Storage(physical) Structure

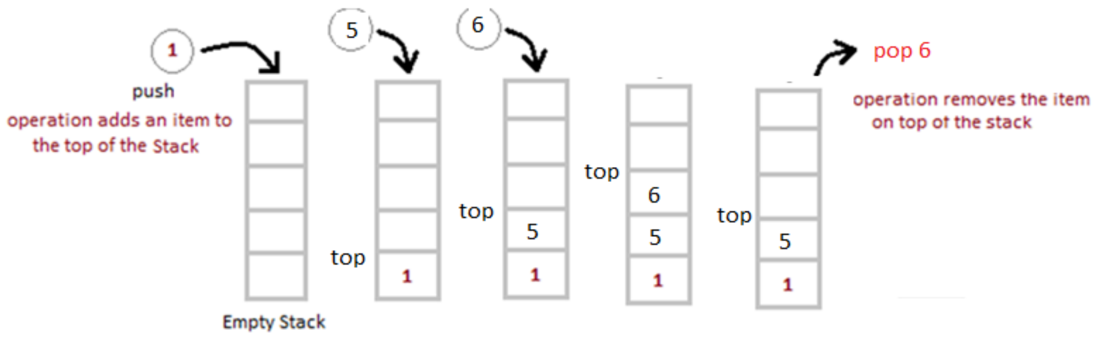
Storage structure depends on the implementation of stack , array or linked list structure.

Implementation of Stack

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size.

☒ Implementation of stack using Array

The following figure shows implementation for stack using array:


1
push
operation adds an item to the top of the Stack
Empty Stack
5
top
1
6
top
5
1
top
6
5
1
pop 6
operation removes the item on top of the stack
top
5
1

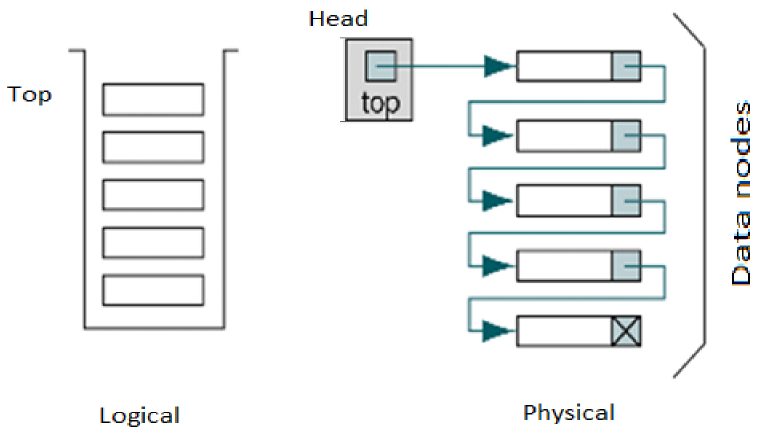
listArray as stack

In the following stack class:

```
class stack{
  int top;
  int listArray[]=new int[10];    //Maximum size of Stack
  stack() {
    top = -1;}

 void push(int x){
    if  ( top >= 10)                  System.out.println( "Stack Overflow");
    else{
      top++;
      listArray[top] = x;
      System.out.println( "Element Inserted");;}
 }

 int pop(){
   if  (top < 0) {
     System.out.println( "Stack Underflow");
     return 0; }
   else  {
      int d = listArray[top];
      top--;
      return d; }
   }
}
```

☒ Implementation of stack using linked list

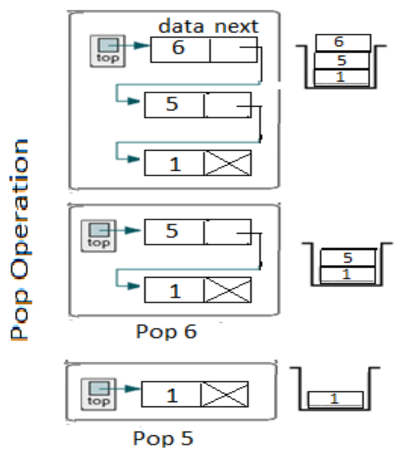
The following figure shows implementation for stack using linked list:

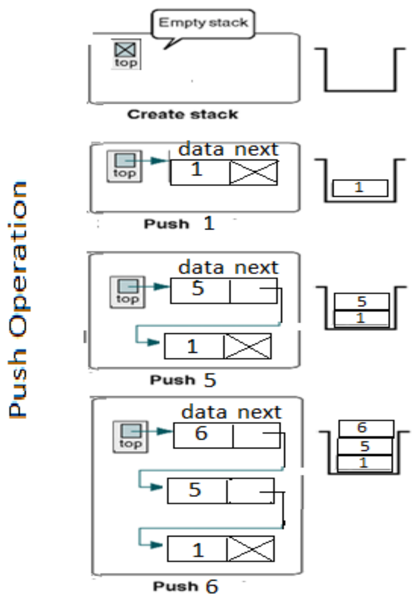


In the following stack class:

```
class StackLinkedList {
        node top = null;

     void push(int  data) {
         node p = new node(data);
         if  (top == null)
            top = p;
         else{
           p.next = top;
           top = p; }
     }

    node  pop() {
        if  (top == null){
          System.out.println( " Stack is Empty.... ");
          return top;}
        else{
           node p = top;
           top = top.next;
           return p;}
    }

    void peek(){
         if  (top == null)
            System.out.println("The Stack is Empty....");
         else
            System.out.println (top.data);}

   void clear() {
       if (top == null)       System.out.println("The Stack is Empty....");
       else                   top = null;  }
    void displayStack() {
    node current = top;
    while (current != null) {
      System.out.print(current.data);
      System.out.print("  ");
      current = current.next; }
  }
}
```

Example:





Applications of Stack

1. **Reverse a word**: You push a given word to stack - letter by letter - and then pop letters from the stack.

2. **Expression Conversion** and **evaluating expressions.**

3. **Call subprogram and recursion processing**

☒ **Converting and Evaluating Expressions**

Arithmetical operations like addition, subtraction, multiplication, and division are called binary operations because they each combine two operands:

Operand <u>operator</u> operand

There are three type for operator notations : **Infix, prefix and postfix**(also called ***reverse Polish notation*, or *RPN***) . for example consider the simple binary operation **a + b** Equivalent prefix and postfix forms are shown bellow:.

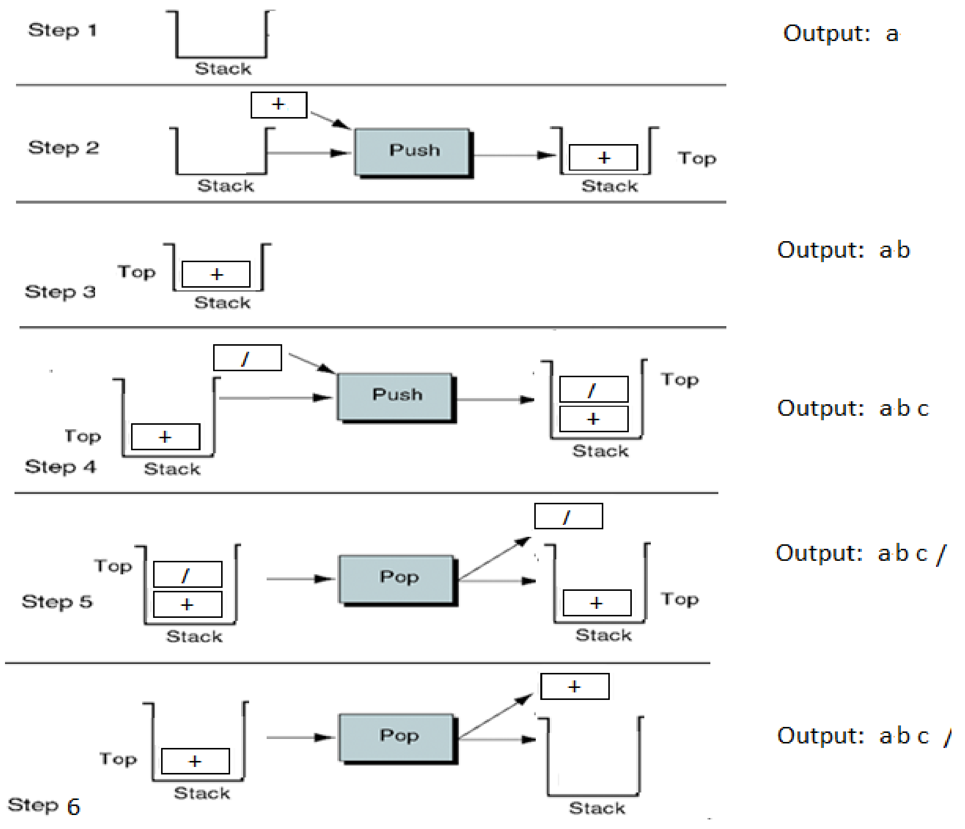
Prefix: + a b	operator first
Postfix: a b +	operator last

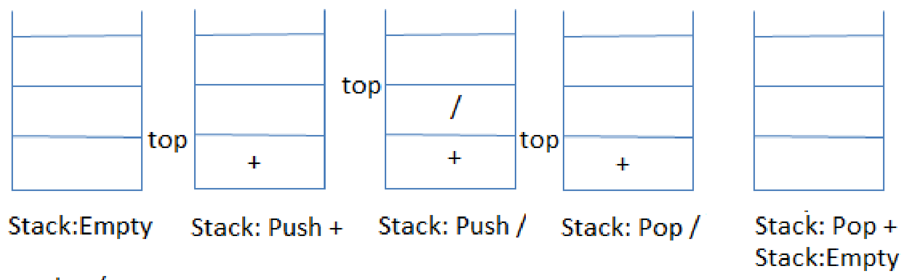
Notes:
1. Postfix expressions are easier to process by machine than are infix expressions. and it used in stack to evaluate the expressions.
2. Each operator has precedence as shown in the following table

Operator	Precedence
( ) ^	Highest ↓
* , /	Lowest
+, −	

- **Convert Infix to Postfix Algorithm (in case expression NOT contain parentheses)**

Step 1: For each term in expression
Step 2: If term is an operator
  Compare it with the operator on the top, if have the same or higher precedence , Pop it, otherwise Push this operator into stack
 Else Copy operand to output
 end if
Step 3: Pop remaining operators and copy to output
Example 1: By using stack data structure convert Infix expression a + b / c to Postfix expression



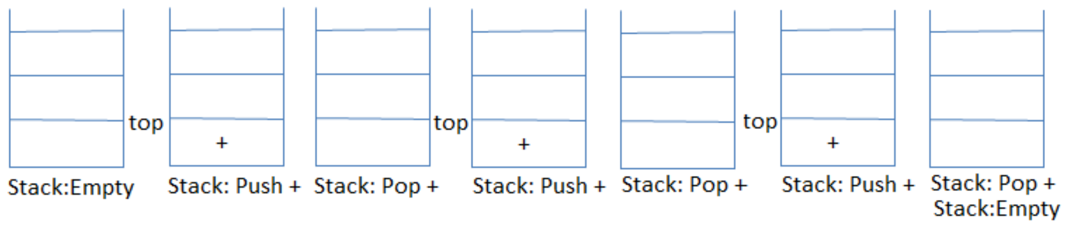


Output : a b c / +

Or

Expression	Stack Operator	Output (RPN)	Action
a+b/c	Empty	-	
+b/c	Empty	a	
b/c	+	a	Push +
/c	+	ab	
C	+/	ab	Push /
Empty	+/	abc	
Empty	+	abc/	Pop /
Empty	Empty	abc/+	Pop +

Example 2: : By using stack data structure convert Infix expression a + b + c + d to Postfix expression.



Output : ab+c+d+

Or

Expression	Stack Operator	Output (RPN)	Action
a+b+c+d	Empty	-	
+b+c+d	Empty	a	
b+c+d	+	a	Push +
+c+d	+	ab	
+c+d	Empty	ab+	Pop +
c+d	+	ab+	
+d	+	ab+c	
+d	Empty	ab+c+	Pop +
d	+	ab+c+	
Empty	+	ab+c+d	
Empty	Empty	ab+c+d+	Pop +

- **Exercises: Convert these infix expressions to postfix expressions:**

1. a + b * c* d+ e
2. a * b + c * d* e* f
3. a / b / c + d * e * f
4. a+b*c^d/e-f*g
5. a-b+c*d/e

Algorithm to Convert an infix expression to postfix notation(in case expression contain parentheses)

Suppose Q is an arithmetic expression(contain parentheses) in infix notation. We will create an equivalent postfix expression P by adding items to on the right of P.

```
Start with an empty stack.  We scan Q from left to right.
While (we have not reached the end of Q)
        If (an operand is found)
           Add it to P
        end if
        If (a left parenthesis is found)
          Push it onto the stack
        end if

        If (a right parenthesis is found)
           While (the stack is not empty AND the top item is not a left
                       parenthesis)
                  Pop the stack and add the popped value to P
              end while
          Pop the left parenthesis from the stack and discard it
     End-If
     If (an operator is found)
        If (the stack is empty or if the top element is a left parenthesis)
          Push the operator onto the stack
     Else
        While (the stack is not empty AND the top of the stack
                      is not a left parenthesis AND precedence of the
                       operator <= precedence of the top of the stack)
             Pop the stack and add the top value to P
        End-While
        Push the latest operator onto the stack
     End-If
   End-If
 End-While
 While (the stack is not empty)
   Pop the stack and add the popped value to P
 End-While
```

Note : At the end, if there is still a left parenthesis at the top of the stack, or if we find a right parenthesis when the stack is empty, then Q contained unbalanced parentheses and is in error.

Example 3 : Convert the infix expressions to postfix:(2+3)*4

Expression	Stack Operator	Output (RPN)	Action
(2+3)*4	Empty	-	
2+3)*4	(	-	Push (
+3)*4	(	2	
3)*4	(+	2	Push +
)*4	(+	2 3	
*4	(	2 3 +	Pop +
*4	Empty	2 3 +	Pop (
4	*	2 3 +	
Empty	*	2 3 + 4	
Empty	Empty	2 3 + 4 *	Pop *

Example 4 : Convert the infix expressions to postfix:2+(3*4)

Expression	Stack Operator	Output (RPN)	Action
2+(3*4)	Empty	-	
+(3*4)	Empty	2	
(3*4)	+	2	Push +
3*4)	+(	2	Push(
*4)	+(	2 3	
4)	+(*	2 3	Push*
)	+(*	2 3 4	
Empty	+(*	2 3 4 *	Pop *
Empty	+(	2 3 4 *	Pop (
Empty	Empty	2 3 4 * +	Pop +

Example 5: Convert the infix expressions to postfix:(3*2+4)^2

Expression	Stack Operator	Output (RPN)	Action
(3*2+4)^2	Empty	-	
3*2+4)^2	(	-	Push(
*2+4)^2	(	3	
*2+4)^2	(*	3	Push *
+4)^2	(*	3 2	
+4)^2	(	3 2*	Pop *
4)^2	(+	3 2 *	Push +
)^2	(+	3 2 *4	
^2	(	3 2* 4 +	Pop +
^2	Empty	3 2 *4 +	Pop (
2	^	3 2 *4 +	Push ^
Empty	^	3 2 *4+ 2	
Empty	Empty	3 2 *4 + 2 ^	Pop ^

Algorithm to Evaluate a postfix expression

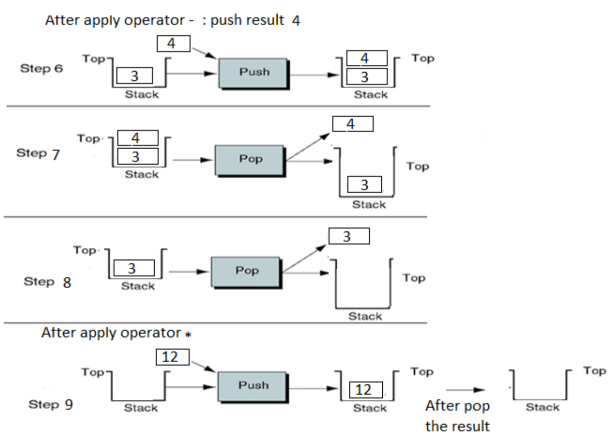
Suppose P is an arithmetic expression (contain parentheses)in postfix notation. We will evaluate it using a stack to hold the operands.

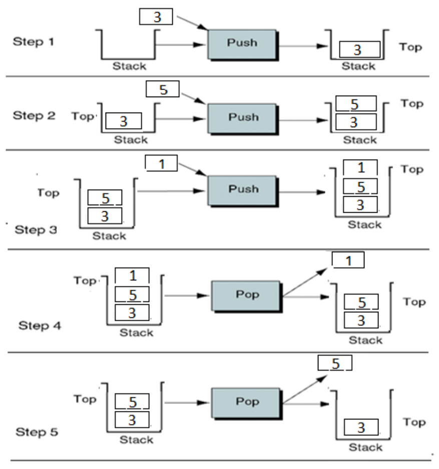
```
Start with an empty stack.  We scan P from left to right.
While (we have not reached the end of P)
  If an operand is found
    push it onto the stack
  End-If
  If an operator is found
    Pop the stack and call the value A
    Pop the stack and call the value B
    Evaluate B op A using the operator just found.
    Push the resulting value onto the stack
  End-If
End-While
Pop the stack (this is the final value)
```

Notes:

- At the end, there should be only one element left on the stack.
- This assumes the postfix expression is valid.

Example 6 : consider the postfix expression : 3 5 1 - *





Expression	Execute Stack	Action
3 5 1 - *	Empty	
5 1 - *	3	Push 3
1 - *	3 5	Push 5
- *	3 5 1	Push 1
*	3 4	Pop 1 and 5,Execute 5 - 1= 4 , Push 4
Empty	12	Execute 3 * 4 = 12, Push 12
Empty	Empty	Pop the result 12

Example 7 : consider the postfix expression : 5 4 + 3 / 1 6* 2 +

Expression	Execute stack	Action
5 4 + 3 / 1 6* +	Empty	
4 + 3 / 1 6* +	5	Push 5
+ 3 / 1 6* +	5 4	Push 4
3 / 1 6* +	9	Pop 5 and 4,Execute 5+4=9 , Push 9
1 6* +	9 3	Pop 9 and 3,Execute 9/3= 3 , Push 3
1 6* +	3	Pop 6 and 1, Execute 6/1 = 6, Push 6
6* +	3 1	
* +	3 1 6	Pop 1 and 6, Execute 6*1 = 6, Push 6
+	3 6	Pop 3 and 6, Execute 3+6 = 9, Push9
Empty	9	
Empty	Empty	Pop the result 9

Example8 : Evaluate the postfix expression : 3 2 *4 + 2 ^

Expression	Execute stack	Action
3 2 *4 + 2 ^	Empty	
2 *4 + 2 ^	3	Push 3
*4 + 2 ^	3 2	Push 2
4 + 2 ^	6	Pop 3 and 2, Exeute2 *3=6, Push 6
+ 2 ^	6 4	Push 4
2 ^	10	Pop 6 and 4, Execute 6+4=10, Push 10
2 ^	10 2	Push 2
^	100	Pop 10 and 2, Exeute10^2=100, Push 100
Empty	Empty	Pop the result 100

Example9 : Evaluate the postfix expression : 2 3 4 * + 2 ^

Expression	Execute stack	Action
2 3 4 * + 2 ^	Empty	
3 4 * + 2 ^	2	Push 2
4 * + 2 ^	2 3	Push 3
* + 2 ^	2 3 4	Push 4
+ 2 ^	2	Pop 3 and 4,Execute 3*4= 12,Push 12
+ 2 ^	2 12	
2 ^	14	Pop 2 and 12, Execute 2+12=14, Push 14
^	14 2	Push 2
Empty	196	Pop 14 and 2, Execute 14^2 = 196, Push196
Empty	Empty	Pop the result 196

Exercises: Convert the following infix expressions to postfix then evaluates these postfix expressions, giving the stack contents after each step:

1. a/b+c/d
2. (a + b) * (c + d)
3. ((a + b) * c) - d
4. (80 – 30) * (40 + 50 / 10)
5. (*a* + *b*) – (*c* / (*d* + *e*))
6. *a* / ((*b* / *c*) * (*d* – *e*))
7. (*a* / (*b* / *c*)) * (*d* – *e*)
8. *a* * *b* + *c*) / *d* – *e*)
9. (*a* – *b*) / (*c* * (*d* + *e*))
10. *a* / (*b* + (*c* * (*d* – *e*)))
11. (((2 * 5) - (1 * 2)) / (11 - 9))
12. (2 * 5 - 1 * 2) / (11 - 9)
13. A + B * C / D - E
14. A + B * (C - D)) / E