DATA STRUCTURE

DS CORE GROUP

ABES Engineering College
College Code-032

Authors	Akhilesh Kumar Srivastava, Amit Pandey, Amrita Jyoti, Asmita Dixit, Manish Srivastava, Puneet Kumar Goyal
Authorized By	
Creation/Revision Date	July 2021
Version	1.0

Chapter 5
Stack

Table of Contents

5.1 Introduction

Consider the situation of a dinner party where the plates are arranged for serving the food. After a new plate is washed, it is pushed on the top of the pile of plates. When serving food, a clean plate is popped off the pile. The last washed plate is popped first. The first washed plate will be picked at the last. The order for picking the plates follows the last in first out (LIFO).

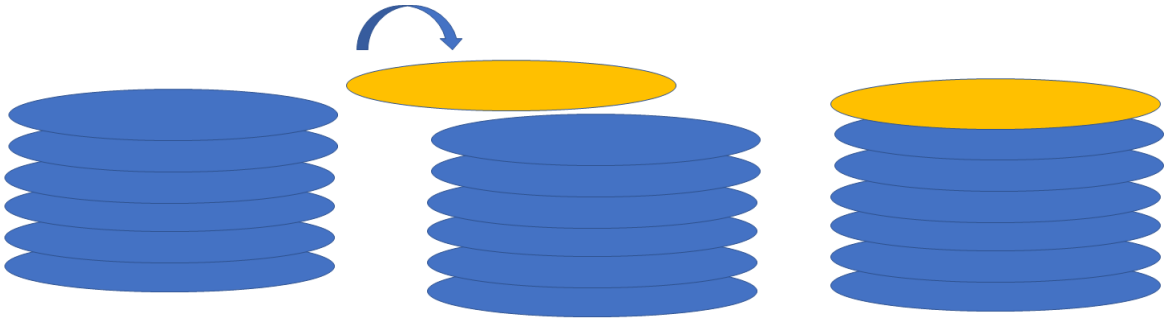


Figure: Insertion of a new plate on top of the pile of plates

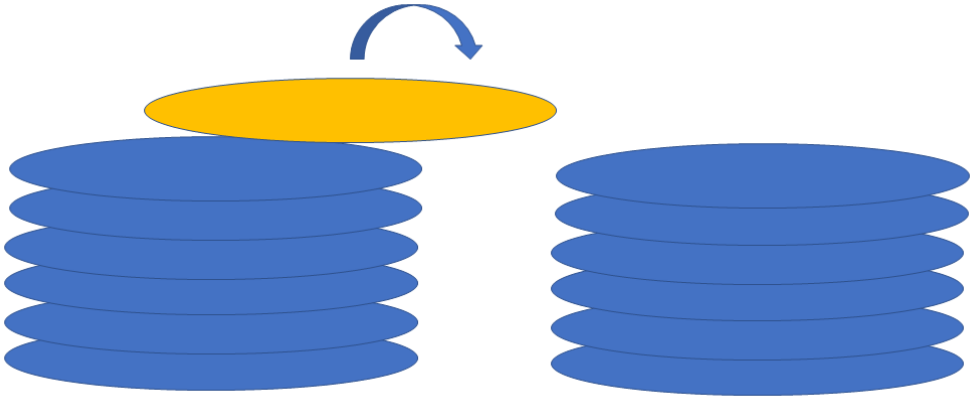


Figure: Deletion of last inserted plate

Consider the situation in which you are working on any text editor. You are typing something and suddenly realize that the sentence typed just now does not fit the frame. You may like to undo whatever is written recently. By pressing Ctrl+Z you may undo the last written content. Similarly, the undo facility is provided in any software where the last done activity can be undone. This follows the principle of Last in First out (LIFO).

Consider another situation where you are working on any browser and some sites have been opened. With the different links of subsequent web pages, you are moving towards the next pages. If you wish to come back to the previous page, you need to press the back button. This again follows the principle of Last in First out (LIFO).

Stack is the data structure that works on the principle of Last in first out. There are many situations where stack is used directly (undo in any software e.g.) or indirectly (Recursion e.g.).

5.2 Definition

Stack is a data structure where elements can be inserted and deleted from one end only known as '**Top**' of the Stack.
Insertion in Stack is given a standard name **Push** and deletion is given a standard name **Pop**.

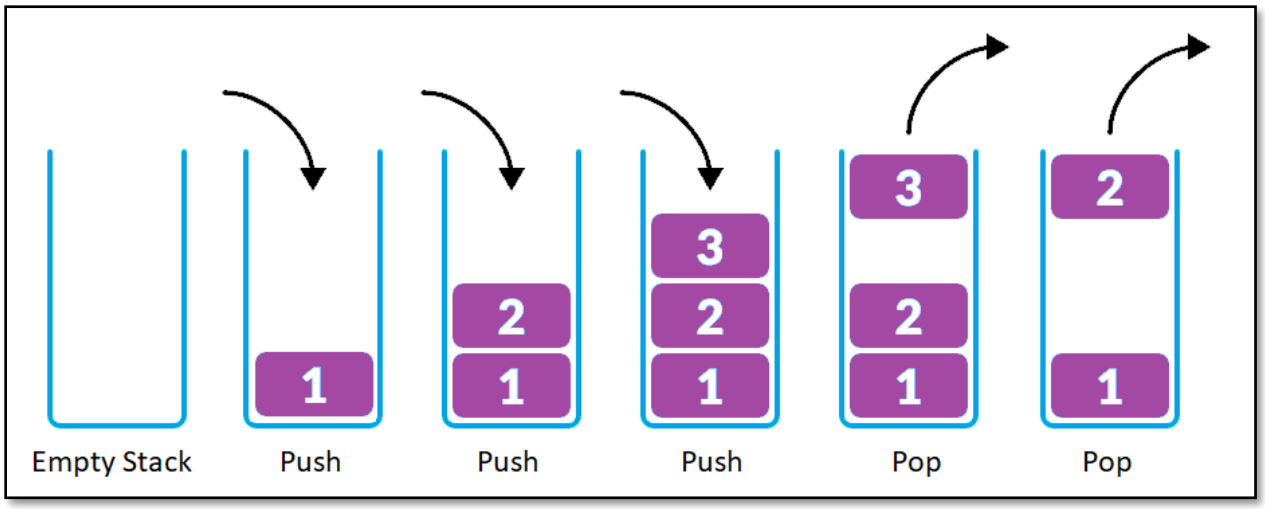


Figure: Insertion and Deletion

5.3 Operations in Stack

The various primitive operations that are needed for stack are

- Initialization
- Emptiness check
- Insertion or Push
- Deletion or Pop
- Finding Stack Top element or Peek

Let us explain these operations one by one assuming Array implementation. There are two elements in each stack

- Array to store data (we are naming it as Data [ ])
- Top (to store the index of the Top element)

If stack were declared with the name S, its elements would be represented as:
S.Top: representing the top pointing to the Last element inserted in Stack.
S.Data[ ]: is the buffer storing all the elements.

5.3.1 Stack Initialization

Every stack, before being used, should be initialized. Initialization must depict that the stack contains zero elements at the beginning. For example, if we assume the array indices to vary from 0 to N-1 (where N is the array size), If Top is initialized to 0 index, it means an element is there at the 0^{th} index. Hence, we should initialize the Top to invalid index, say -1.

ALGORITHM InitializeStack(S)
Input: Stack S
Output: None
BEGIN:
S.Top = – 1 } Initializing Top at -1
END;

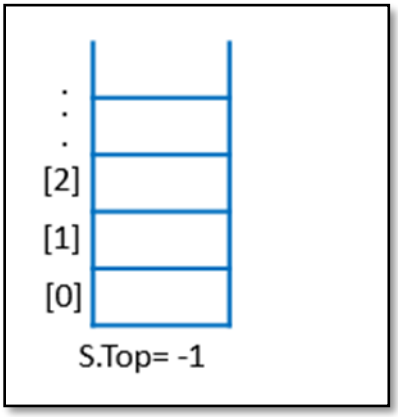


Figure: Stack Initialization

Time Complexity: Time Complexity of this Operation is Θ(1) as there is only one statement to execute.
Space Complexity: Since there is no auxiliary space used in the algorithm, the space function is 0 (constant) i.e., the Space Complexity of this Operation is Θ (1).

5.3.2 Emptiness Check

If there is no element in stack then stack is said to be empty. To check the emptiness of stack, refer to the initialization. If the Top is -1, stack is empty. The function given below is a Boolean valued function that returns True (If the stack is empty) or returns False (when stack is not empty).

ALGORITHM IsEmpty(S)
Input: Stack S

Output: True or False based on emptiness

BEGIN:

```
IF (S.Top == -1) THEN
        RETURN TRUE
ELSE
        RETURN FALSE
```

S.Top=-1 indicates stack is empty

END;

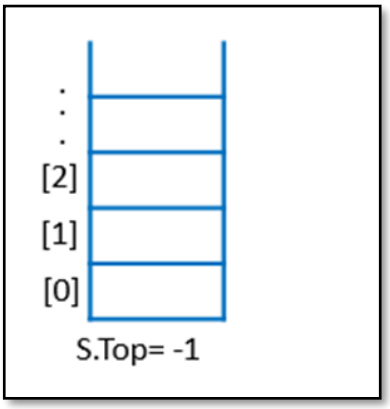


Figure:- Empty Stack

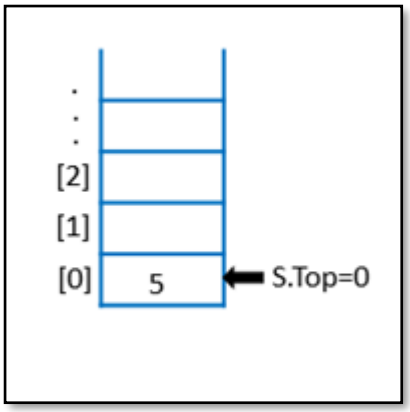


Figure:- Non Empty Stack

Time Complexity: Time Complexity of this Operation is Θ(1) as there are only two statements to execute.

Space Complexity: Since there is no auxiliary space used in the algorithm, the space function is 0 (constant) i.e., Space Complexity of this Operation is Θ (1).

5.3.3 Stack Insertion

Insertion in stack is given a standard name 'Push'. The figure given below shows the insertion process on the stack.

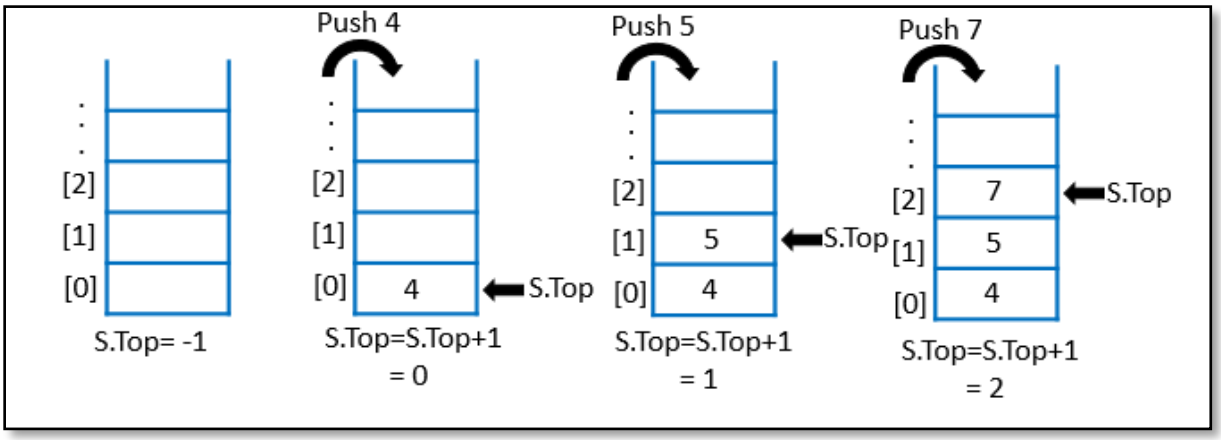


Figure: Insertion Process

Every time we have to insert an item in the stack, we must update the Top index.
We usually fix the Stack size (MaxSize e.g.) in array implementation.
If Top has reached the MaxSize – 1 index, i.e. the maximum possible index, further insertion will not be possible. We throw an exception **"overflow"** in this case.

Overflow: *Consider a bucket where water is filled completely. If we pour more water in it, the water will overflow. Similarly, an attempt to insert an element in the full stack leads to the condition of overflow.*

Procedure

- Check if stack is full. If true, throw an exception 'Overflow'
- If not full, increment the top index
- Insert the element at the top index of stack

ALGORITHM Push (S, item)
Input: Stack S, data value 'item' to be inserted
Output: True or False based on emptiness
BEGIN:

```
IF S.Top == MaxSize – 1 THEN
        WRITE("Stack Overflows")
        EXIT(1)
ELSE
        S.Top = S.Top + 1
        S.Data[S.Top] = item
```

If no more insertion possible.

Increment in Top index by 1.
Insert the data item at the Top index of Stack.

END;

Exit() *is a function that is used to terminate the program. Parameter in the function can be 1,0 or any positive value. The value 0 indicates the normal termination and 1 indicates the termination under exceptional circumstances.*

Time Complexity: If the Overflow occurs, there will be one condition check plus two other statements to execute. If overflow does not take place, then one condition and two other statements will be executed. A total of 3 statements will be executed in both cases. Therefore, time Complexity of this Operation is constant i.e. Θ(1).

Space Complexity: Since there is no auxiliary space used in the algorithm, the space function is 0 (constant) i.e., Space Complexity of this Operation is Θ(1).

5.3.4 Deletion in stack

Deletion in stack is given a standard name **'Pop'**. This operation will be used for deletion of a data element from the stack. Figure given below shows the deletion process on the stack.

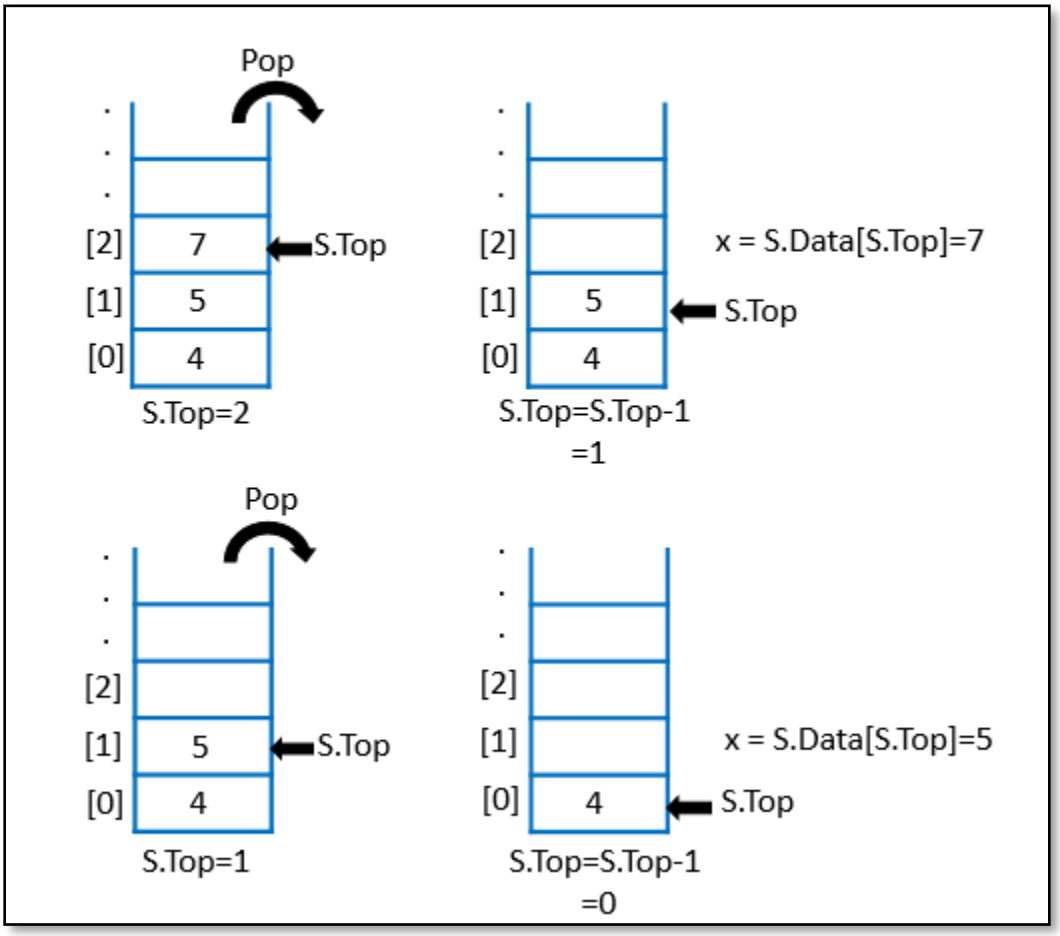


Figure : Deletion process

For performing the Pop operation, we need to check if the Stack is Empty. Since an element cannot be deleted from the Empty Stack, the attempt to delete an element will lead to **underflow** condition. While performing deletion, Top index of the stack will be decremented by 1. Pop will return the deleted item to the calling function.

Procedure

- Check if stack is empty. If yes, throw an exception **'underflow'**
- If not empty, then save the top index element.
- Decrement the top index
- Return the saved element

ALGORITHM Pop (S)
Input: Stack S
Output: Deleted item from top index of stack
BEGIN:

```
IF S.Top == -1 THEN
        WRITE("Stack Underflows")
        EXIT(1)
ELSE
        x = S.Data[S.Top]
        S.Top = S.Top – 1
        RETURN x
```

Check Underflow condition

Saving the Top index element in x.
Decrement in Top index by 1.
Returning deleted item.

END;

Time Complexity: If the underflow occurs, there will be one condition check plus two other statements to execute. If underflow does not take place, then one condition and three other statements will be executed. A total of 3 or 4 statements will be executed in both the cases. Therefore, time Complexity of this Operation is constant i.e. Θ(1).

Space Complexity: 1 extra variable is used in the algorithm named x, the space function is 1 (constant) i.e. Space Complexity of this Operation is Θ(1).

5.3.5 Finding the top element

Sometime we need to just find which element is present at the top of the stack. This can be done through a function named StackTop or Peek.

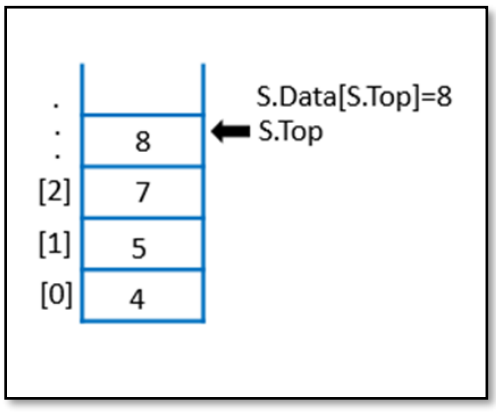


Figure: - seeking topmost element

The Top element stored in the stack is returned through this operation, leaving no change to the existing stack.

ALGORITHM Peek (S)
Input: Stack S
Output: Top index element of Stack
BEGIN:
RETURN S.Data[S.Top] } Reading and returning the top index element
END;

Time Complexity: Time Complexity of this Operation is Θ(1) as there is only one statement to execute.

Space Complexity: Since there is no auxiliary space used in the Algorithm, the space function is 0 (constant) i.e. Space Complexity of this Operation is Θ(1).

5.4 Applications of Stack

5.4.1 Number Conversion

Using stack, we can easily convert a decimal number to its equivalent Binary, Hexadecimal, Octal or a number in any desired base.

Let us take an example to convert a Decimal number to Binary equivalent. The process would repeatedly divide the given number by 2 and storing remainders. The operation has to stop when the original number reduces to 0. If we print the remainders in the reverse order, this represents the Binary equivalent of the given Number.

Step 1: Take the Modulus of number with 2.
Step 2: Push the remainder in Step 1 into a Stack.
Step 3: Divide the number by 2 and save the quotient in the same number.
Step 4: Repeat Step 1 to 3 until number becomes zero.
Step 5: Pop the Stack and display the popped item.
Step 6: Repeat Step 5 until Stack becomes empty.

Example: - To convert 38(a decimal number) to equivalent binary the steps given below are performed.

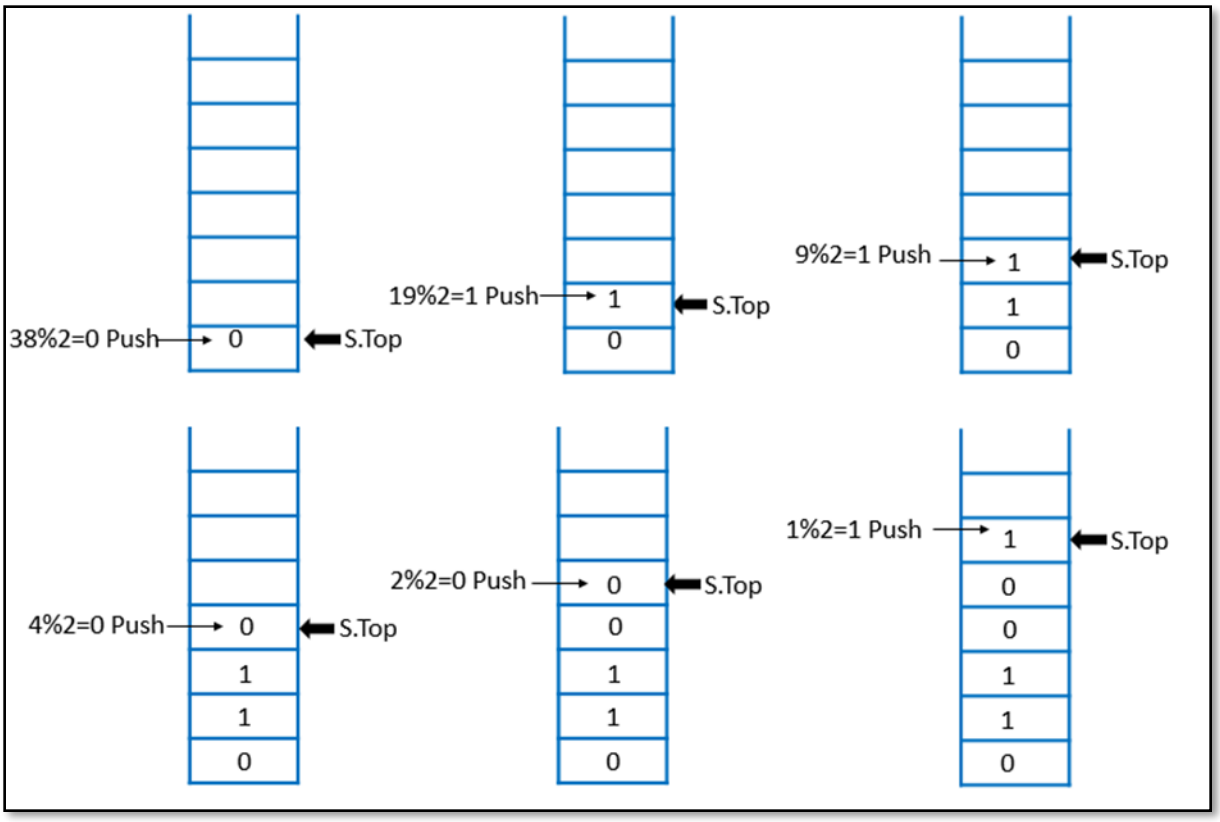
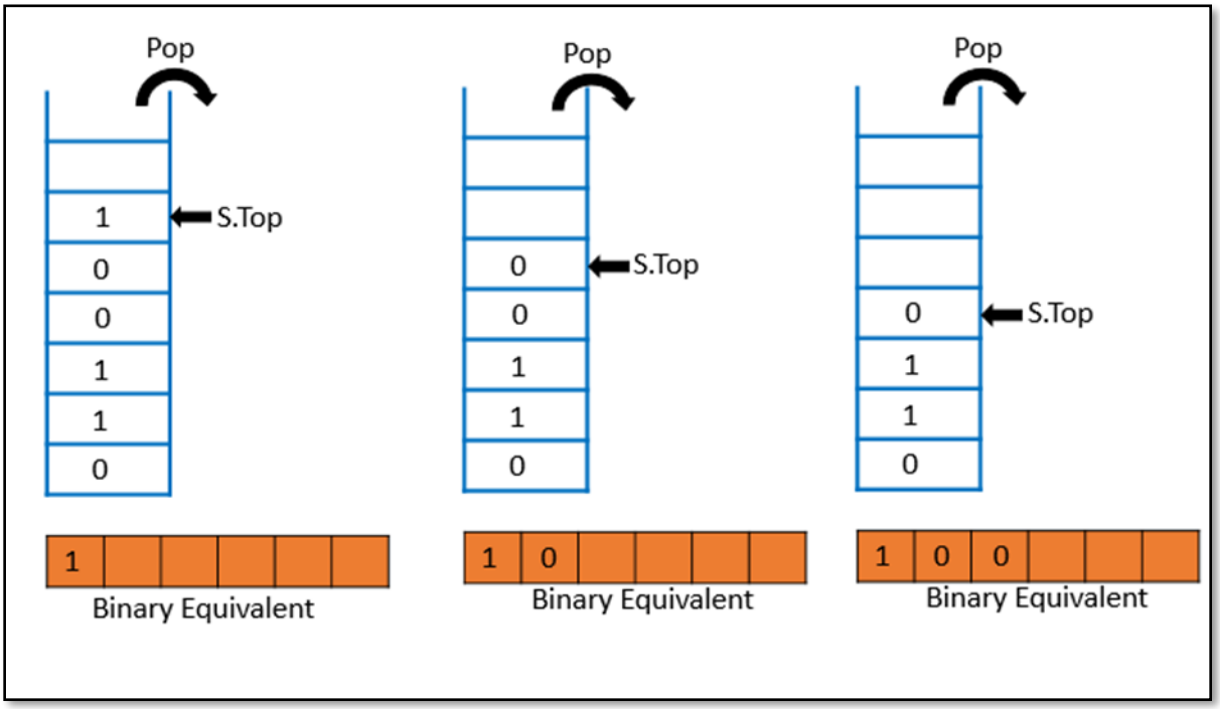


Figure: Pushing the repeated remainders while conversion from Decimal to Binary

In order to print the Binary, we simply start removing the element till the stack becomes empty.



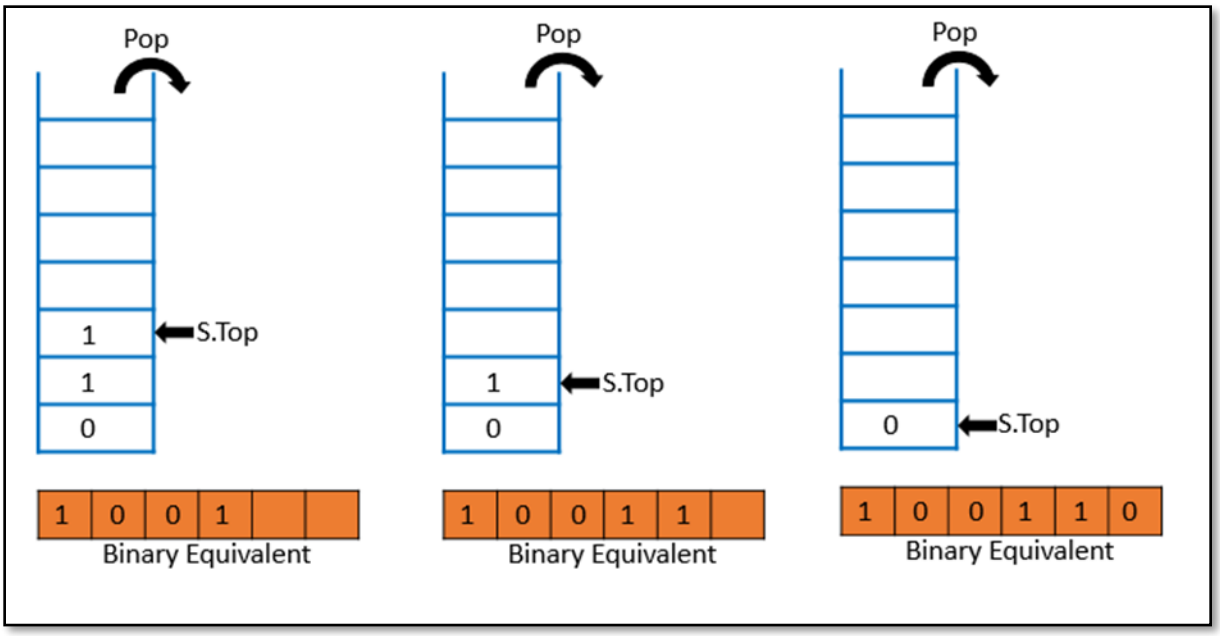


Figure: Printing the Binary Equivalent

ALGORITHM: DecimalToBinary (Decimal)
Input: A Decimal number
Output: Binary equivalent
BEGIN:

```
Stack S
InitializeStack(S)
WHILE Decimal != 0 DO
        r = Decimal % 2
        Push(S, r)
        Decimal = Decimal / 2

WHILE !IsEmpty(S) DO
        x = Pop(S)
        WRITE(x)
```

Finding the remainder.
Push the remainder into Stack.
Saving the quotient.

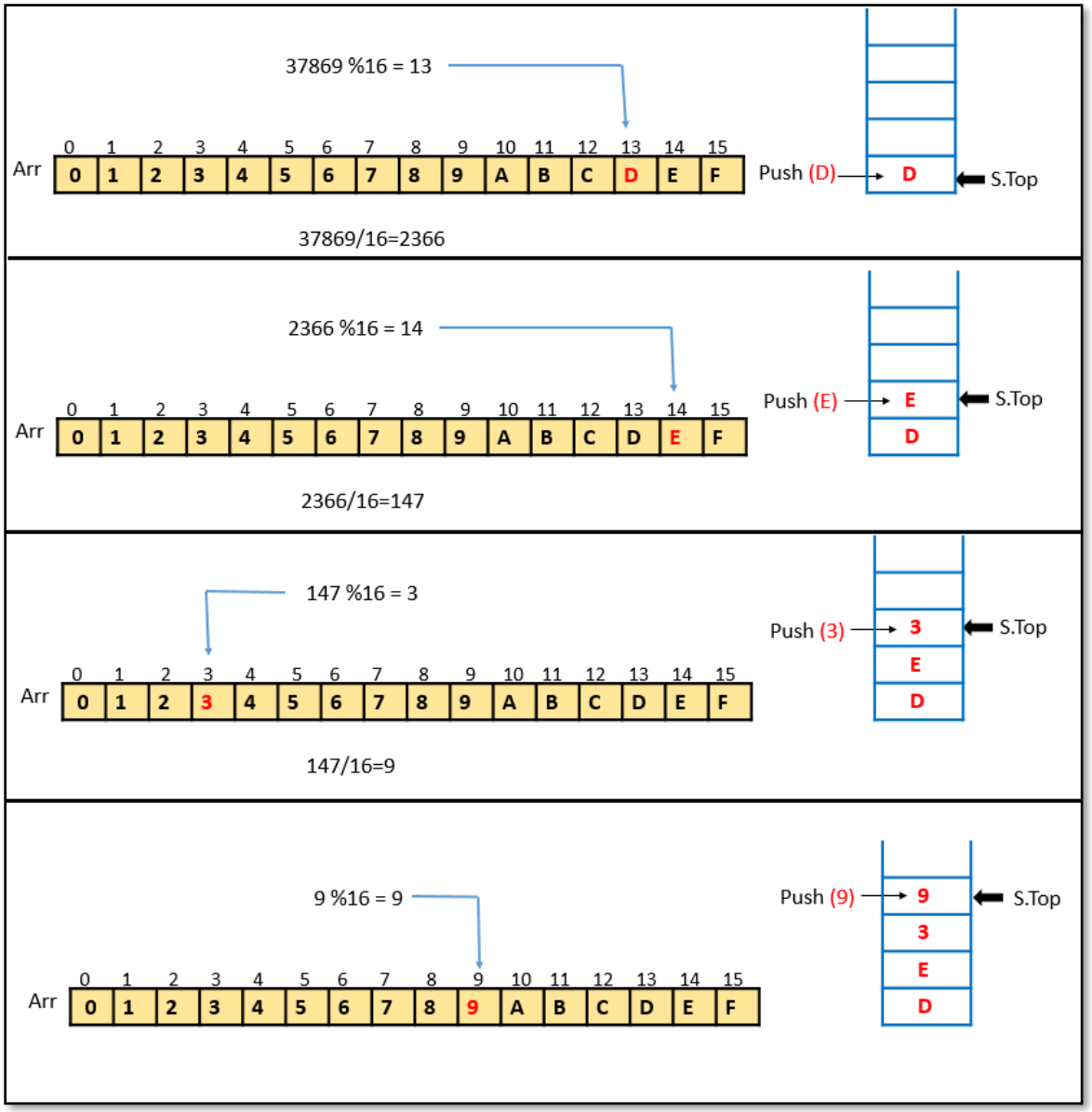
Printing the Stack elements until Stack is empty.

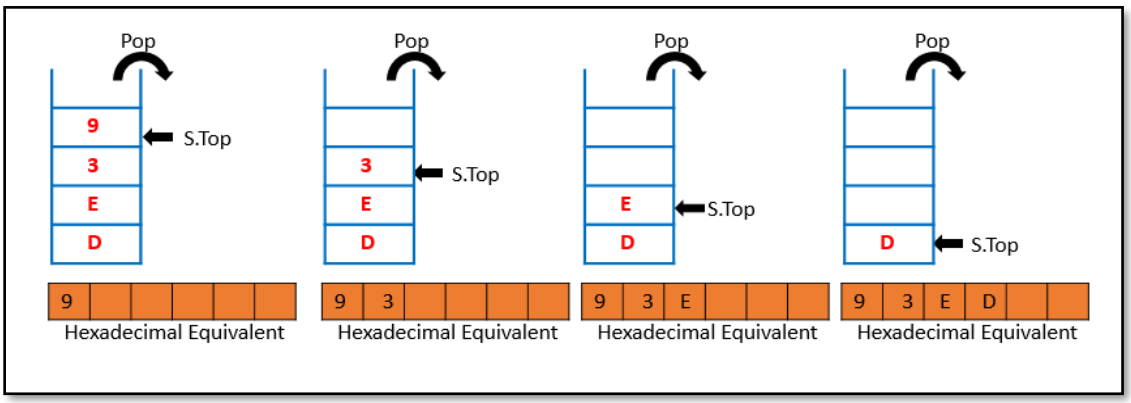
END;

In the above algorithm if the divisor is 8, the algorithm converts given Decimal Number to Octal.

If the Hexadecimal equivalent is expected, a small change is required to map the remainder 10,11,12,13,14 and 15 to A,B,C,D,E,F, respectively. We have a very simple solution to take a Direct Address Table (DAT) of size 16 that stores all the remainders in sequence. This means 0 is stored at 0^{th} index, 1 is stored at 1^{st} index, …, 9 is stored at 9^{th} index, A is stored at 10^{th} index, B is stored

at 11th index, …, F is stored at 15th index. The remainder will be integer; hence we are storing the corresponding character equivalent in the array. The rest of the procedure remains the same.





ALGORITHM: DecimalToHexadecimal(Decimal)
Input: A Decimal Number
Output: Equivalent Hexadecimal Number
BEGIN:

```
Stack S
InitializeStack(S)
Arr[ ] = { '0', '1', '2', '3' , '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' }
WHILE Decimal!= 0 DO
        r = Decimal % 16
        Push(S, Arr[r])
        Decimal = Decimal / 16

WHILE !IsEmpty(S) DO
        x = Pop(S)
        WRITE(x)
```

Direct Address Table containing possible remainders in sequence

Finding the remainder.
Push the remainder into Stack.
Saving the quotient.

Printing the Stack elements until Stack is empty.

END;

In the above Algorithm can be generalized by changing the divisor to the base of your choice.
Means
Decimal % Base
Decimal / Base

Generalized version, by simply dividing by the base as per requirement

Time Complexity: The complexity of this operation is dependent on the base of conversion. The repeated remainders are stored in the stack for $\log_b N$ times and removed $\text{Log}_b N$ times as well. Since Push and Pop operations do take $\Theta(1)$ time, the total complexity above algorithm would be $2^* \Theta(\text{Log}_b N)$ i.e. $\Theta(\text{Log}_b N)$.
Space Complexity: A stack of size $\text{Log}_b N$ plus some variables. The Space Complexity would be $\Theta(\text{Log}_b N)$

5.4.2 Reversal of String using Stack

We can simply reverse a string using stack. By reversing, we mean that if a string is "live" then when we read it from last, it will be "evil. "

Procedure
Reversal of string can be done by reading each string character and pushing these characters onto stack till the end of string character ('\0') is encountered. After this, we will start Popping the elements one by one till the stack is empty. Each time the Pop operation is performed, the

popped element should be printed. The printed order of the elements specifies the reversed string.

Algorithm StringReverse (Str[])
Input: A String
Output: String in reverse order
Begin:

```
Stack S
i=0
WHILE Str[i] ! = '\0' DO
        Push(S, str[i])
        i++

WHILE !IsEmpty(S) Do
        x = Pop(S)
        WRITE (x)
```

End

Reading all elements present in string str[ ] one by one and pushing them in the stack S

Printing the stack elements until Stack is empty.

Now we will explain the above concept with the help of an example

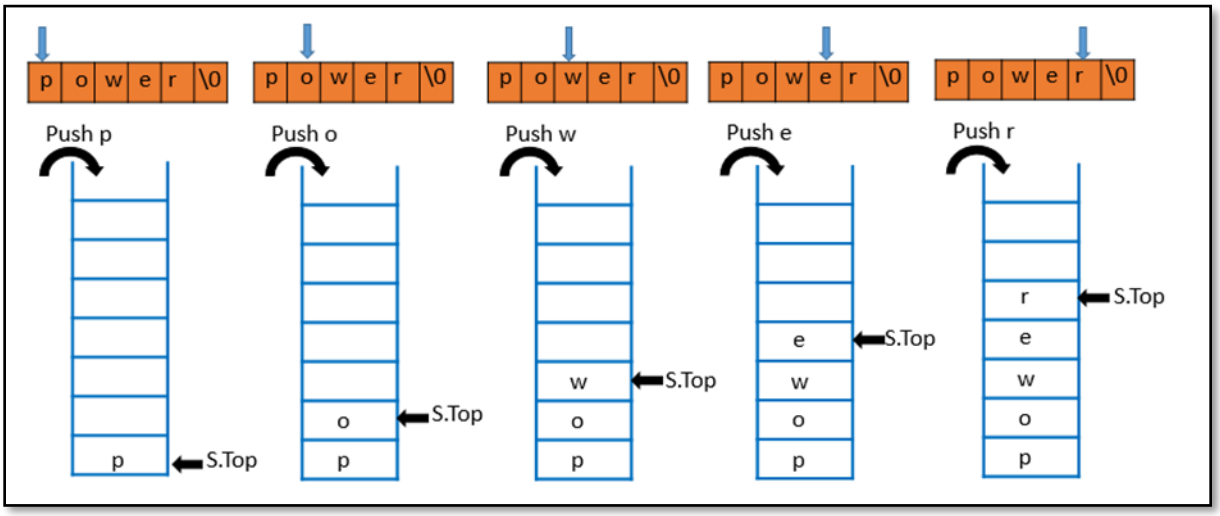


Figure:- Insertion of characters in Stack after reading from String (until a Null character is observed)

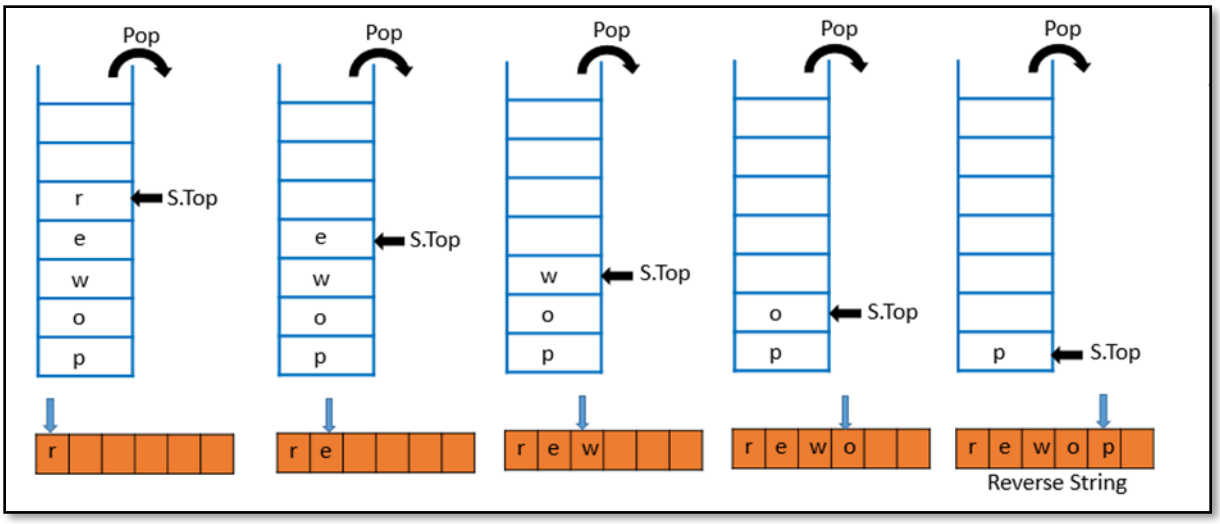


Figure: Printing the stack characters that depicts the reverse of the string

If we want the string reversed to be returned, we need to store the popped element in a string. This can be done by taking another string but better approach would be to store the elements in the original string. The same is written as an algorithm below

ALGORITHM StringReverse (Str[])
Input: A String
Output: Reversed String
Begin:

```
Stack S
InitializeStack(S)
i=0
WHILE Str[i] ! = '\0' DO
        Push(S, Str[i])
        i++
j=0
 WHILE !IsEmpty(S) Do
        x = Pop(S)
        Str[j]=x
        j=j+1
Str[j]='\0'
RETURN Str
```

End

Reading all elements present in string str[ ] one by one and pushing them in the stack S

Popping the elements from Stack
Popped elements are stored in Str[ ]
Process is repeated till Stack become empty

Output String will be in reversed order

5.4.3 Palindrome Check using Stack

A String is a palindrome if we read a string from left to right or right to left, both refers to the same string. That means the original and reversed strings are same. E.g. “madam”, “Kanak”.
To check if the given string is a palindrome, following procedure can be employed.

Procedure

Reverse the string and match it with the original string. If both are same, then the original string will be a palindrome. In case of mismatch of even a single character between the original and reversed string, string will not be a palindrome.

ALGORITHM PalindromeCheck (Str[])
Input: A String
Output: Decision about string’s palindrome status
Begin:

```
Stack S
InitializeStack(S)
i=0
WHILE Str[i] ! = ‘\0’ DO
        Push(S, str[i])
        i++
j= 0
 WHILE !IsEmpty(S) Do
        IF(Str[j]== Peek (S)) THEN
                x = Pop(S)
        ELSE
                BREAK
        j=j+1
IF IsEmpty(S) THEN
        WRITE (“Palindrome”)
ELSE
        WRITE (“Not Palindrome”)
```

End

Reading all elements present in string str[ ] one by one and pushing them in the stack S

If top element of stack is matched with current character of string,
Pop the element from Stack.
Terminate the process otherwise
Process is repeated till Stack becomes empty

If all characters have matched

If any character had a mismatch

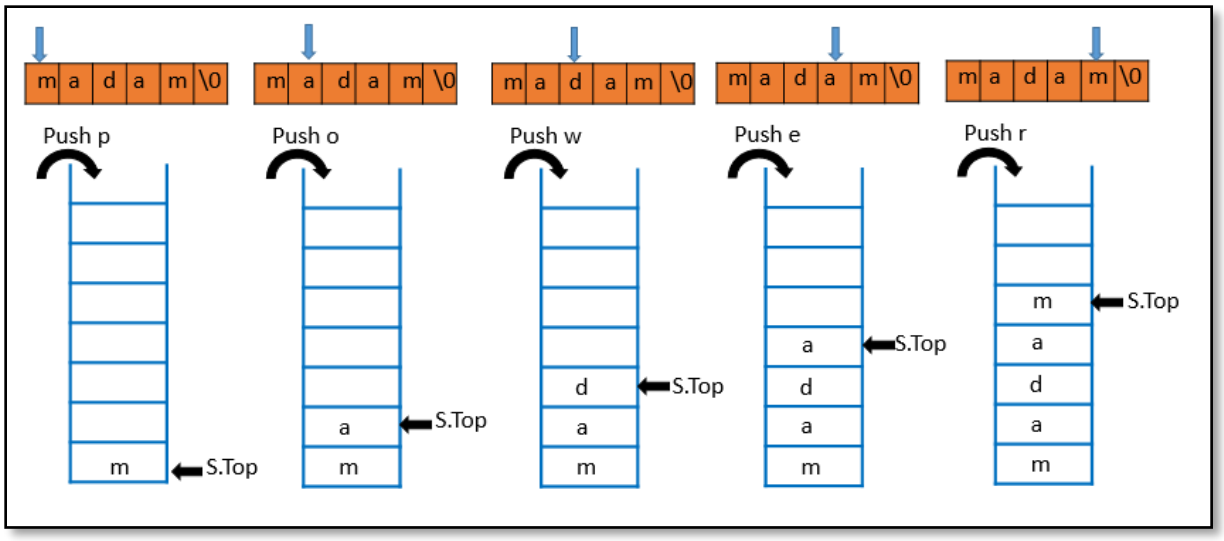


Figure: Insertion of characters in Stack after reading from String (until a Null character is observed)

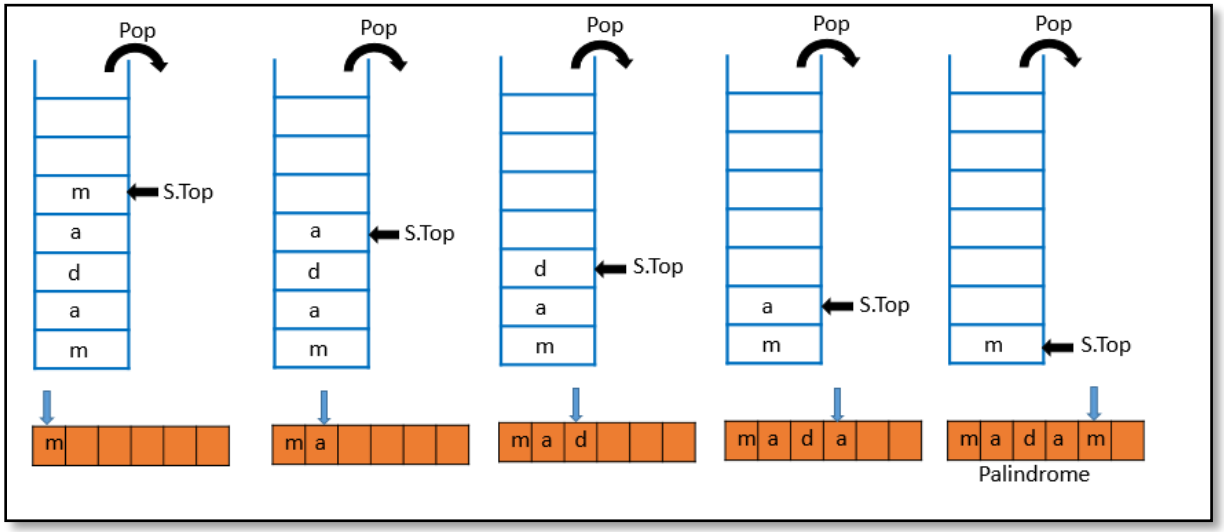


Figure: Printing the stack characters that depicts the reverse of the string

Problem: Given a Number, use stack to check if this is a Palindrome.

5.4.4 Balanced Parenthesis

In the mathematical expression written in usual pen and paper practice, 3 types of the brackets i.e. square [ ], curly { } and small ( ) are used. The arithmetic or logic expression written for a program consists of parenthesis ( ) only. These arithmetic Expressions follow the stack property. Every opening parenthesis must have a corresponding closing parenthesis. The order of closing of parenthesis are just in the opposite order of opening of the parenthesis. For example,

1. (a + b) * (c + d) is a valid Expression.
2. (a -b)) +((b * 2) is not a valid expression as with respect to the second “)” closing parenthesis, there does not exist any opening parenthesis “(“.

Procedure:

There are two rules in order to perform this. These are

1) **Rule 1: -** When encountered opening braces '(', simply push into stack.
2) **Rule 2: -** For every closing brace ')', there should be an opening brace in stack and we will simply remove opening brace '(' from stack.
   a) If entire expression is processed and stack becomes empty, the expression is correct otherwise expression is incorrect.
   b) If the corresponding opening bracket is not present on stack for a closing bracket, the expression will be incorrect.

ALGORITHM ParanthesisCheck(Exp[])

Input: An Expression as a String

Output: Valid or Invalid status

BEGIN:

```
Stack S
InitializeStack(S)
Flag = 1 , i = 0

WHILE Exp[i] == '\0' DO
        IF Exp[i] == '(' THEN
                Push(S, Exp[i])

        ELSE IF Exp[i] == ')' THEN
                IF !Empty(S) THEN
                        POP(S)
                ELSE
                        Flag = 0
                        Break
         i=i+1

IF IsEmpty(S) THEN
        IF Flag == 0 THEN
                WRITE("Not Balanced")
        ELSE
                WRITE("Balanced")
ELSE
        WRITE("Not Balanced")
END;
```

Pushing the opening parenthesis '(' onto stack

Finding the closing parenthesis ')'
Checking whether if the stack is empty
If stack is not empty, element is popped from Stack

Finding the closing parenthesis ')'
Checking whether stack is empty or not
If stack is empty, Flag is set to 0

If Flag is 0 then parenthesis are not balanced

If Flag is 1 then parenthesis are balanced

If stack is not empty and expression reached to end or invalid expression are passed

Example 1: To check the validity of ((()), the processing can be seen like this.

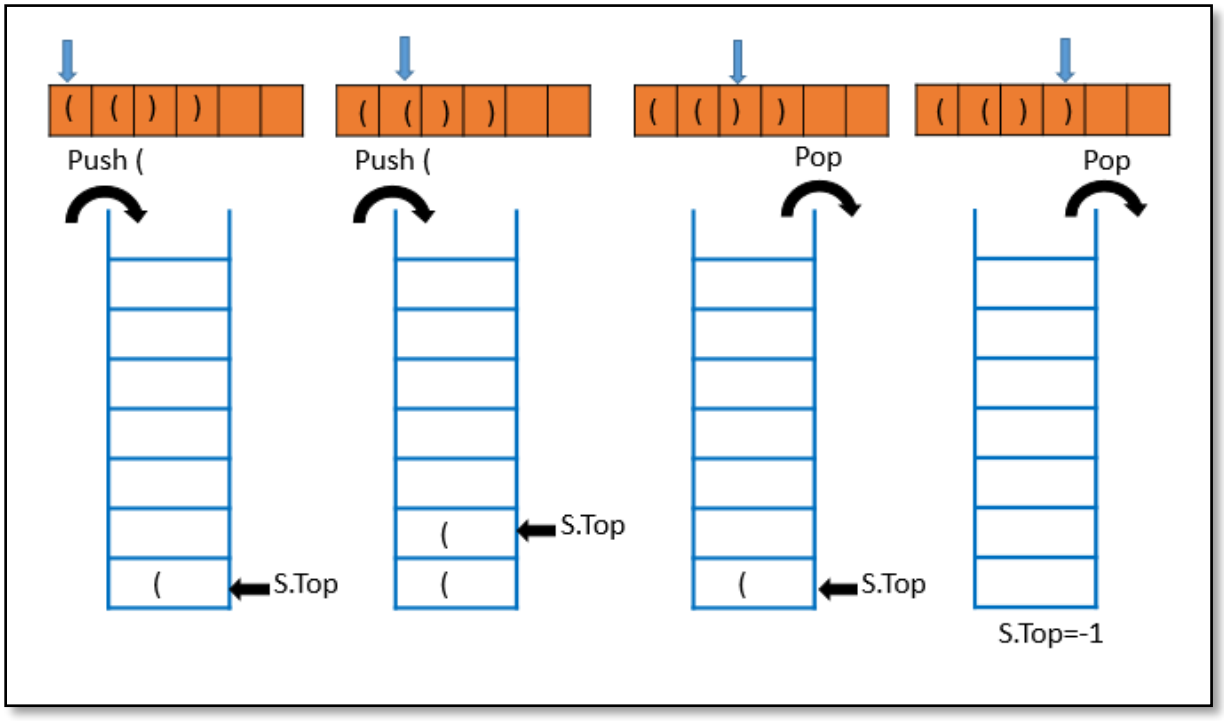


Figure:- Checking the balance of parenthesis in the expression on (())

Example 2: To check the validity of ( ) ( ), the processing can be seen like this.

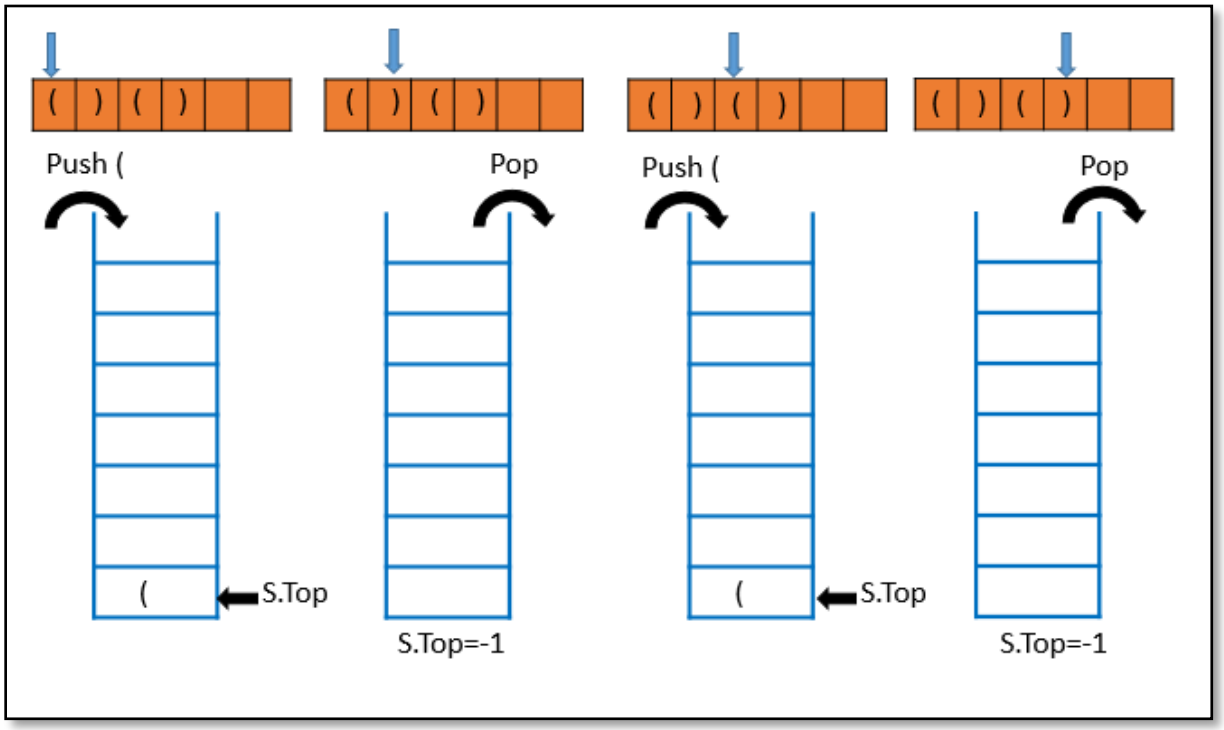


Figure:- Checking the balance of parenthesis in the expression on ( ) ( )

Example 3: To check the validity of ((( )), the processing can be seen like this.

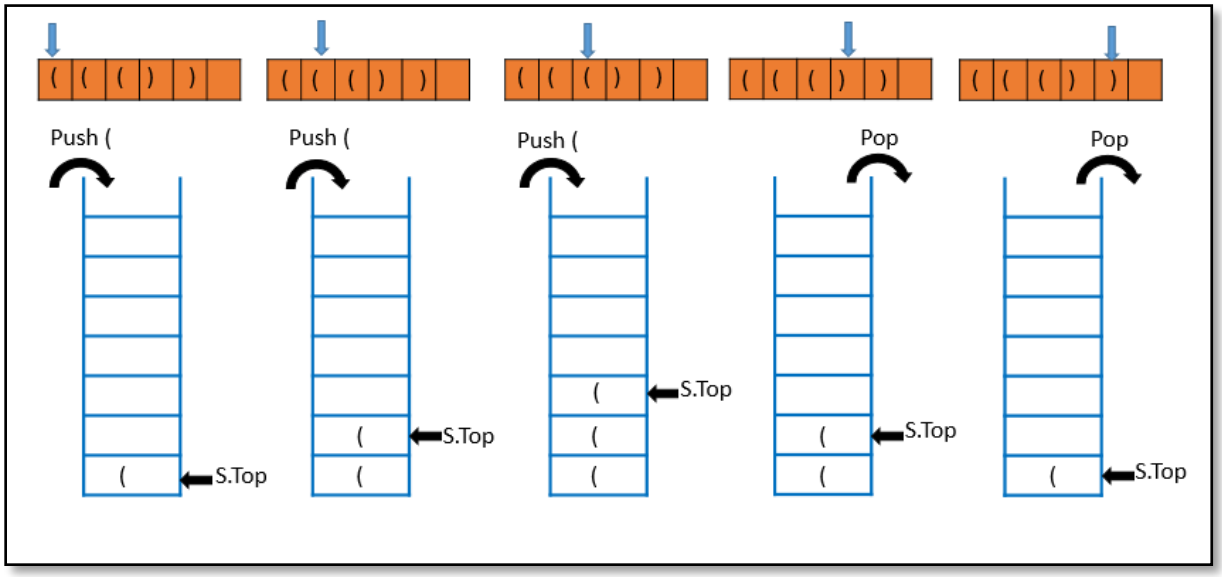


Figure:- Checking the balance of Parenthesis in the expression on ((( ))

Example 4: To check the validity on ( ) ( )), the processing can be seen like this.

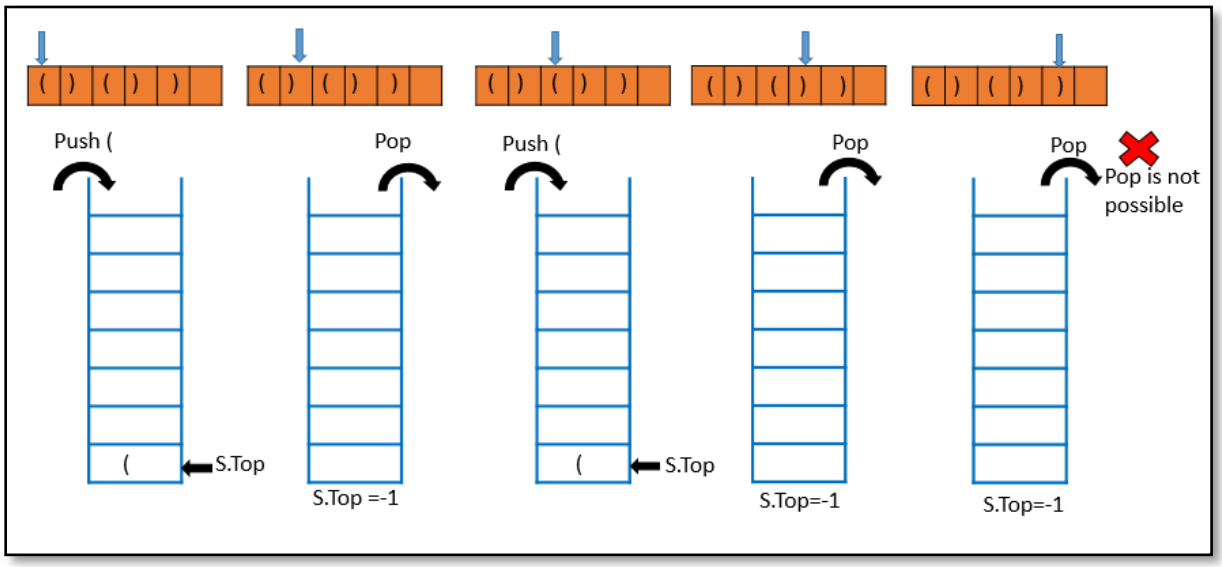


Figure:- Checking the balance of Parenthesis in the expression on( ) ( ))

Example 5: To check the validity on) ( ) ( ), the processing can be seen like this.

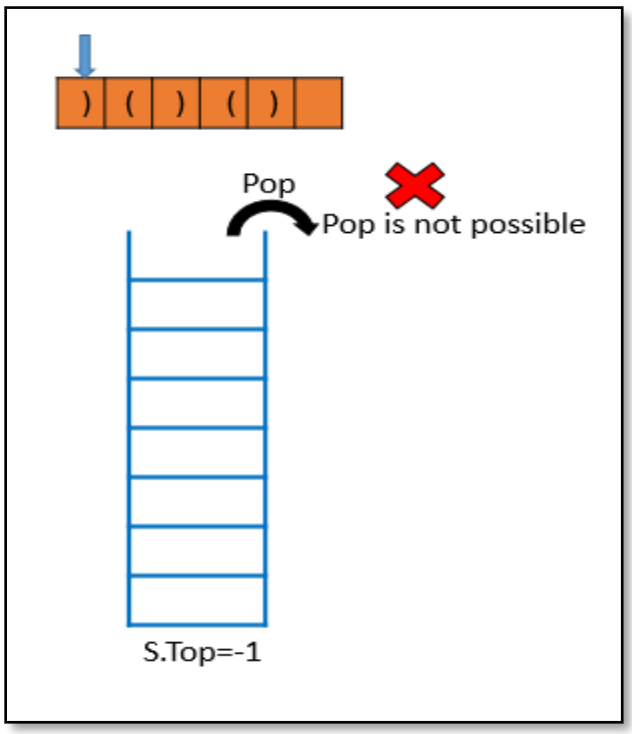
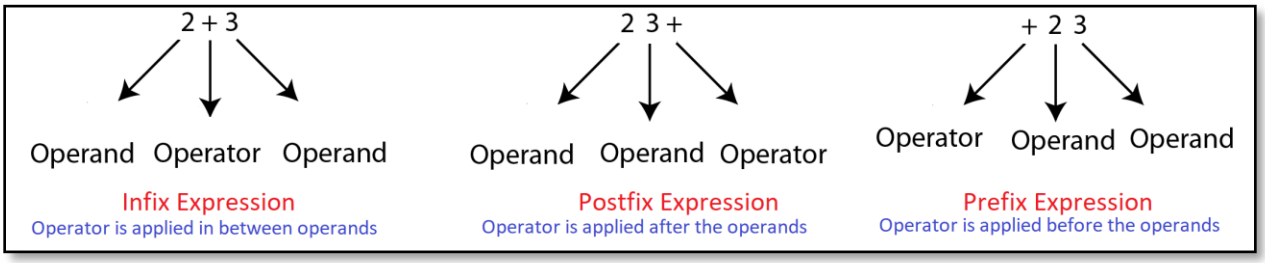


Figure:- Checking the balance of Parenthesis in the expression on) ( ) ( ),

5.5 Inter-conversion and Evaluation of Polish Notation Expressions

Consider an expression x+y. Here x and y are the operands and + is the operator. There are three fashions in which an arithmetic expression can be written:



Type of Expression	Description	Example
Infix	Operator is written in between the Operands	2+3
Postfix	Operator is written after the Operands	23+
Prefix	Operator is written before the Operands	+23

The above expressions denote the addition of operand 2 and operand 3. The **Prefix** is **called Polish Notation**, and **Postfix** is called **Reverse Polish Notation**.

5.5.1 Requirement of Polish/Reverse Polish Notation

Suppose we have been given an Infix expression:

2 + 5 * 6 – 8 / 4 + 3 ↑ 2 – (4 / 2)

To solve such expression, we need to traverse the entire expression to find which operator has the highest precedence. We need to go from Left to Right and Right to Left multiple times to evaluate the expression (according to BODMAS rule).

**** Here it is assumed that students are aware of BODMAS rule.***

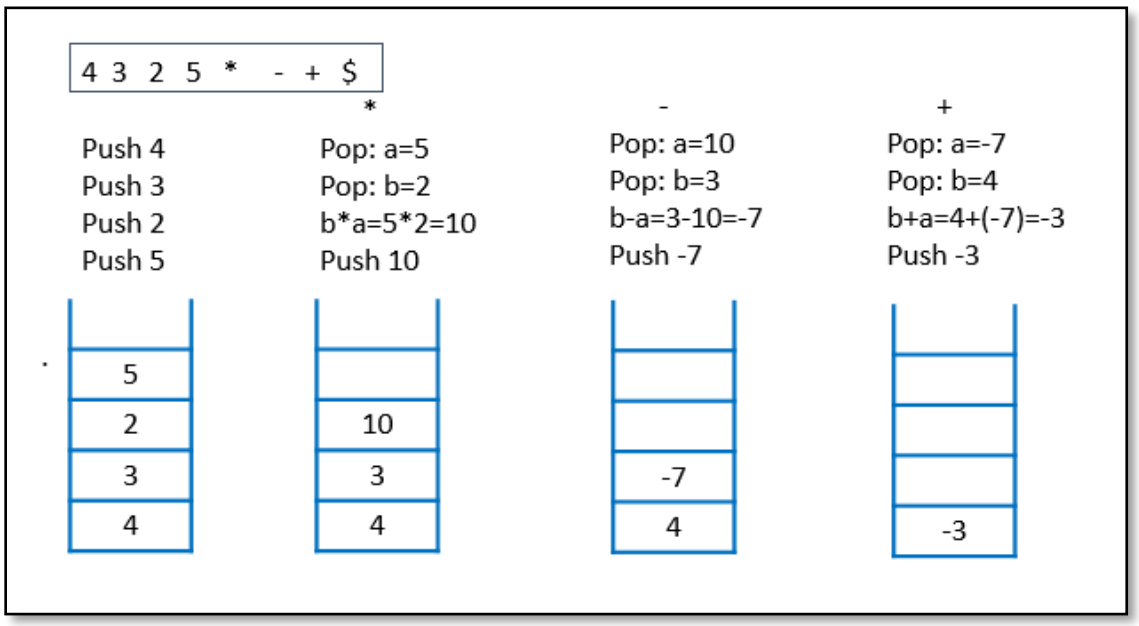
In case we are given another type of expression (Postfix), e.g., 897-*, if we solve this like:

- If operands are observed, push this on the stack
- If operators are observed, Pop the stack twice and store elements in b and a, respectively. Then, apply the operator on a and b and Push the result in the stack.
- If no term left in the expression, the stacktop element will be the result

In this case, we only need to move from Left to Right once. As soon as we reach the Right end, we have the answer.

5.5.2 Evaluation of Postfix Expression

If stacks are used to evaluate Prefix Expression or Postfix Expression, Expression evaluation becomes very simple and straightforward. The precedence and associativity of operators are not required to be checked while evaluating the expression using stack.



Tabular method to represent the evaluation of Postfix Expression:

Example: 4 3 2 5 * – +

Symbol	Oprnd 1	Oprnd 2	Value	OpndStack Bottom ➔Top
4				4
3				4,3
2				4,3,2
5				4,3,2,5
*	2	5	2*5=10	4,3,10
–	3	10	3–10= –7	4, –7
+	4	–7	4+(–7) = –3	–3
$				

Value of the given Expression = –3

ALGORITHM POSTFIX EVALUATION (Postfix Expression)

Input: A Postfix Expression

Output: Evaluated Value of expression

BEGIN:

```
STACK OperandStack
Initialize (OperandStack)
WHILE not end of input from postfix expression Do
    Symbol = Next character from postfix expression
    IF  symbol is an operand THEN
        Push (OperandStack, Symbol)
    ELSE
        oprnd 2 = Pop (OperandStack)
        oprnd 1 = Pop (OperandStack)
        value = Result of applying symbol to oprnd1 and oprnd2
        Push (OperandStack, value)
Result = Pop(OperandStack)
RETURN Result
```

IF element is operand, insert them onto Stack

IF operator, Pop twice and Store these elements in two different variables. Apply operator on the popped operands

Returning the Final answer

END;

Time Complexity: Θ(N), There are N symbols in the Expression. For each symbol, decision is to be taken for Push or Pop. In the case of operand, 2 statement execution (including a condition) and for operator, 5 statement executions (including the condition) are required. There are N/2+1

operands and N/2 operators. There are 4 other statements outside the Loop which are compulsory.

Hence total statements

= 4 + (N/2+1) *2 + (N/2) *5

= 4+N+2+5*N/2

= 7*N/2+6

= Θ(N)

Space Complexity: Θ(N), N/2+1 size stack is required and three variables for Symbol, value, and result. Total space=N/2+1+3=N/2+4

5.5.3 Evaluation of Prefix Expression

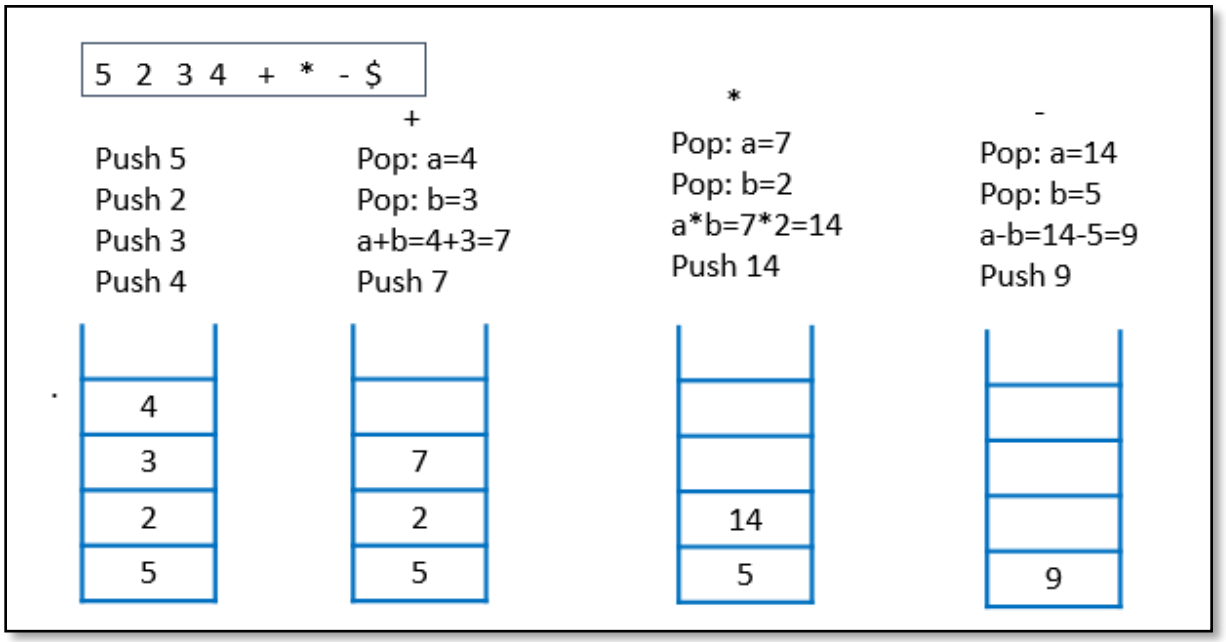
Procedure:

To evaluate the prefix expression, the following steps are performed:

- Take an empty operator stack
- Reverse the prefix expression
- If operands are observed, push the operand on the stack
- If operators are observed, pop stack twice and store elements in a and b, respectively
- Apply the operator on a and b and push the result in stack.
- If no term Left in the expression, pop the stack and this is the result

Example: Evaluate the Expression – * + 4 3 2 5

Reversed Expression: 5 2 3 4 + * –



Tabular Method to represent the evaluation of Prefix Expression:

Symbol	Oprnd 1	Oprnd 2	Value	OpndStack Bottom ➔Top
5				5
2				5,2
3				5,2,3
4				5,2,3,4
+	4	3	4+3=7	5,2,7
*	7	2	7*2 = 15	5,14
–	14	5	14–5 = 9	9
$				

Value of the given Expression = –3

ALGORITHM PrefixEvaluation (Prefix Expression)

Input: A Prefix Expression

Output: Evaluated Value of expression

BEGIN:

```
Reverse (Prefix Expression)
STACK OperandStack
Initialize (OperandStack)
WHILE not end of input from postfix expression DO
        Symbol = next character from prefix equation
        IF symbol is an operand THEN
                Push (OperandStack, symbol)
        ELSE
                oprnd 1 = Push(OperandStack)
                oprnd 2 = Push(OperandStack)
                value = Result of applying symbol to oprnd1 and oprnd2
                Push(OperandStack, value)
Result = Pop(OperandStack)
RETURN Result
```

Reverse the prefix expression

IF element is operand, insert them onto Stack

IF operator, Pop twice and Store these elements in two different variables. Apply operator on the popped operands

Returning the Final answer

END;

Time Complexity: Θ(N)

At the very first the expression needs to be reversed. Reverse will take Θ(N) time with any approach. There are N symbols in the Expression. For each symbol, decision is to be taken for

Push or Pop. In the case of operand, 2 statement execution (including a condition) and for operator, 5 statement executions (including the condition) are required. There are N/2+1 operands and N/2 operators. There are 4 other statements outside the Loop which are compulsory.

Hence total statements

= C*N+4 + (N/2+1) *2 + (N/2) *5

= Θ(N)

Space Complexity: Θ(N)

N/2+1 size stack is required and three variables for symbol, value, and result. For reversing the string another N size stack is required. Total space=N+N/2+1+3=3N/2+4

5.5.4 Infix to Postfix Conversion

To convert the given Infix expression to Postfix Expression, let us re-write the Precedence and Associativity rules. We are assuming that there are only Exponentiation (↑), Division (/), Multiplication (*), Addition (+) and Subtraction (-) operators.

5.5.4.1 Precedence and Associativity Rules for Arithmetic Operators

Operators	Priority	Associativity
↑	Highest Priority	Right to Left
*	Second Highest Priority	Left to Right
/	Second Highest Priority	Left to Right
%	Second Highest Priority	Left to Right
+	Lowest Priority	Left to Right
−	Lowest Priority	Left to Right

To realize this in the Conversion, Let us design a function named **Precedence(a, b)** which results the following.

Result	Condition	Rules
TRUE	‘a’ has higher precedence than ‘b’	Rule1
TRUE	‘a’ has equal precedence than ‘b’ and a&b are Left Associative	Rule2
FALSE	‘a’ has equal precedence than ‘b’ and a&b are Right Associative	Rule3
FALSE	‘b’ has higher precedence than ‘a’	Rule4

Call of Function	Rule	Result
Precedence(+, +)	Rule2	TRUE
Precedence(+, -)	Rule2	TRUE
Precedence(-, +)	Rule2	TRUE
Precedence(-, -)	Rule2	TRUE
Precedence(*, *)	Rule2	TRUE
Precedence(/, /)	Rule2	TRUE
Precedence(*, /)	Rule2	TRUE
Precedence(↑, ↑)	Rule3	FALSE
Precedence(↑, /)	Rule1	TRUE
Precedence(↑, +)	Rule1	TRUE
Precedence(+, ↑)	Rule4	FALSE
Precedence(*, ↑)	Rule4	FALSE

Consider an Infix Expression. To convert this to Postfix, one symbol from expression is taken at a time. Following rules are used for conversion.

- Take an empty operator stack;
- If the symbol is an operand, add the symbol to postfix expression;
- If the symbol is the operator and stack is empty, push the symbol on stack;
- If the symbol is the operator and stack is not empty do the following.
  - Find Precedence (Stacktop item, Symbol). If the precedence is TRUE, Pop an item from stack and Add the symbol on Postfix Expression.
  - If stack is not Empty after this Pop, Repeat the above statement otherwise Push the Symbol on the Stack.
  - If Precedence (Stacktop item, Symbol) is False, Push the symbol on the stack
- If all the symbols are finished, Pop the stack repeatedly and add symbols on the Postfix Expression.

$ is treated as the symbol to denote the end of expression

Tabular Method for Conversion of Infix Expression to Postfix

Consider an Infix Expression A+B*C/D↑E↑F*G $

Symbol	Operator Stack Bottom ➔Top	Postfix Expression	Precedence Function	
A		A		
+	+	A		
B	+	AB		
*	+, *	AB	+, *	FALSE

C	+, *	ABC		
/	+	ABC*	*, /	TRUE
	+, /	ABC*	+, /	FALSE
D	+, /	ABC*D		
↑	+, /, ↑	ABC*D	/, ↑	FALSE
E	+, /, ↑	ABC*DE		
↑	+, /, ↑, ↑	ABC*DE	↑, ↑	FALSE
F	+, /, ↑, ↑	ABC*DEF		
*	+, /, ↑	ABC*DEF↑	↑, *	TRUE
	+, /	ABC*DEF↑↑	↑, *	TRUE
	+	ABC*DEF↑↑/	/, *	TRUE
	+, *	ABC*DEF↑↑/	+, *	FALSE
G	+, *	ABC*DEF↑↑/G		
$		***ABC*DEF↑↑/G*+***		

ALGORITHM InfixToPostfix (Infix expression)

Input: An Infix Expression (without parenthesis)

Output: Postfix Expression

BEGIN:

```
STACK  OperatorStack
Initialize (OperatorStack)
WHILE   not the end of input from Infix Expression DO
    Symbol = Next symbol from Infix Expression
    IF Symbol is an operand THEN
        Add symbol to postfix expression
    ELSE
        WHILE !Empty (OperatorStack)  &&
        Precedence (StackTop(OperatorStack) , Symbol) DO
            x = Pop (OperatorStack)
            Add x to postfix Expression

        Push(OperatorStack, Symbol)
WHILE   ! Empty(OperatorStack) DO
        x =  Pop(OperatorStack)
        Add x to Postfix Expression
RETURN Postfix Expression
```

END;

IF element is operand, Add this to postfix Expression

IF operator, find the precedence of stacktop symbol with the current operator. If True, Pop and add the popped symbol to Postfix Expression

Push the operator if the precedence is false or stack is empty

Add the remaining symbols on stack to Postfix Expression by popping it one by one

Returning the Postfix Expression

ALGORITHM Precedence (x, y)
Input: Operators x and Y
Output: True or False according to Rules in table above
BEGIN:

```
IF x=='^'||x=='*'||x=='/'|| x=='%' THEN
        IF y=='^' THEN
                RETURN FALSE
        ELSE
                RETURN TRUE
ELSE
        IF x=='+'||x=='-' THEN
                IF y=='+'||y=='-' THEN
                        RETURN TRUE
                ELSE
                        RETURN FALSE
```

END;

Time Complexity: Θ(N)
For any symbol, there are two decisions to make. If the symbol is an operand two statement is required to be executed (including condition). In case of operator, 3 statements in case the loop condition is true otherwise 1 statement for Push. Overall there are N symbols for which decision has to be made. The statement execution required are in the order of N.

Space Complexity: Θ(N)
The operator stack is required which will be of size N/2. Some other variables are required that can be treated as constant space. Total space complexity in this case will be in the order of N.

5.5.4.2 Dealing with Infix Expression with Parentheses
Precedence rules needs to be extended for the parenthesis. Recall the Precedence function with parameters a and b.

- If a is the opening parenthesis (, precedence results false;
- If b is the opening parenthesis (, precedence results false;
- If b is the closing parenthesis), precedence results true;
- If a is opening parenthesis and b is closing parenthesis, precedence results False (This is a special case false in which stack is Popped but Popped symbol is discarded).

Call of Function	Result
Precedence((, +)	FALSE
Precedence((, -)	FALSE
Precedence(-, ()	FALSE
Precedence(+, ()	FALSE
Precedence(*, ()	FALSE
Precedence(/,))	TRUE
Precedence(*,))	TRUE
Precedence(↑,))	TRUE
Precedence((,))	FALSE

Consider an Infix Expression
A+(B*(C/D+E))

Symbol	Operator Stack	Postfix Expression	Precedence Function	
A		A		
+	+	A		
(	+, (	A	+, (	FALSE
B	+, (	AB		
*	+, (, *	AB	(, *	FALSE
(	+, (, *, (	AB	*, (	FALSE
C	+, (, *, (	ABC		
/	+, (, *, (, /	ABC	(, /	FALSE
D	+, (, *, (, /	ABCD		
+	+, (, *, (	ABCD/	/, +	TRUE
	+, (, *, (, +	ABCD/	(, +	FALSE
E	+, (, *, (, +	ABCD/E		
)	+, (, *, (	ABCD/E+	+,)	TRUE
	+, (, *	ABCD/E+	(,)	FALSE
)	+, (	ABCD/E+*	*,)	TRUE
	+	ABCD/E+*	(,)	FALSE
$		***ABCD/E+*+***		

ALGORITHM InfixToPostfix (Infix expression)
Input: An Infix Expression (with/without parenthesis)
Output: Postfix Expression
BEGIN:

STACK OperatorStack

```
Initialize (OperatorStack)
WHILE   not the end of input from Infix Expression DO
        Symbol = Next symbol from Infix Expression
        IF Symbol is an operand THEN
                Add symbol to postfix expression
        ELSE
                WHILE ! Empty  (OperatorStack)  &&
                Precedence (StackTop(OperatorStack) , Symbol) DO
                        x = Pop (OperatorStack)
                        Add x to postfix Expression

                IF Symbol = = ')' THEN
                        x = Pop(OperatorStack)
                ELSE
                        Push(OperatorStack, Symbol)

    WHILE   ! Empty(OperatorStack) DO
            x =  Pop(OperatorStack)
            Add x to Postfix Expression

    RETURN Postfix Expression
END;
```

IF element is operand, Add this to postfix Expression

IF operator, find the precedence of stacktop symbol with the current operator. If True, Pop and add the popped symbol to Postfix Expression

if the precedence is false or stack is empty, if symbol is ')', Pop the stack and discard symbol

Push the operator if the precedence is false or stack is empty

Add the remaining symbols on stack to Postfix Expression by popping it one by one

Returning the Postfix Expression

ALGORITHM Precedence (x, y)

Input: Operators x and Y

Output: True or False according to Rules in table above

BEGIN:

```
    IF x == '(' THEN
            RETURN FALSE
    ELSE
            IF y=='(' THEN
                    RETURN FALSE
            ELSE
                    IF y==')' THEN
                            RETURN TRUE
                    ELSE
                            IF x=='^'||x=='*'||x=='/'|| x=='%' THEN
                                    IF y=='^' THEN
                                            RETURN FALSE
```

```
                ELSE
                        RETURN TRUE
        ELSE
                IF x=='+'||x=='-' THEN
                        IF y=='+'||y=='-' THEN
                        RETURN TRUE
                ELSE
                        RETURN FALSE
END;
```

The complexity of this algorithm remains same as that of the conversion without parenthesis.

5.5.5 Infix to Prefix Conversion

Consider an Infix Expression. To convert this to Prefix, following rules are used

- Reverse the infix expression.
- Take an empty operator stack.
- If the symbol is an operand, add the symbol to prefix expression.
- If the symbol is the operator and stack is empty, push the symbol on stack.
- If the symbol is the operator and stack is not Empty do the following:
  - Find Precedence (Symbol, StackTop item). If the precedence is FALSE, Pop an item from stack and add the symbol on Prefix Expression.
  - If stack is not Empty after this Pop, Repeat the above statement again otherwise Push the Symbol on the Stack.
  - If Precedence (Stacktop item, Symbol) is True, Push the symbol on the stack
- If all the symbols are finished, pop the stack repeatedly and add symbols on the Postfix expression.
- Reverse the prefix expression

Tabular Method for Conversion of Infix Expression to Prefix

Consider the Infix Expression: A+B*C/D↑E↑F*G
Expression after reverse G*F↑E↑D/C*B+A

Symbol	Operator Stack	Prefix Expression	Precedence Function	
G		G		
*	*	G		
F	*	GF		
↑	*, ↑	GF	↑, *	TRUE

E	*, ↑	GFE		
↑	*	GFE↑	↑, ↑	FALSE
	*, ↑	GFE↑	↑, *	TRUE
D	*, ↑	GFE↑D		
/	*	GFE↑D↑	/, ↑	FALSE
	*, /	GFE↑D↑	/, *	TRUE
C	*, /	GFE↑D↑C		
*	*, /, *	GFE↑D↑C	*, /	TRUE
B	*, /, *	GFE↑D↑CB		
+	*, /	GFE↑D↑CB*	+, *	FALSE
	*	GFE↑D↑CB*/	+, /	FALSE
		GFE↑D↑CB*/*	+, *	FALSE
	+	GFE↑D↑CB*/*		
A	+	GFE↑D↑CB*/*A		
$		GFE↑D↑CB*/*A+		
		+A*/*BC↑D↑EFG		

ALGORITHM InfixToPrefix (Infix expression)

Input: An Infix Expression (with/without parenthesis)

Output: Postfix Expression

BEGIN:

Reverse (Infix Expression)

STACK OperatorStack

Initialize (OperatorStack)

WHILE not the end of input from Infix Expression DO

Symbol = Next symbol from Infix Expression

IF Symbol is an operand THEN

Add symbol to prefix expression

ELSE

WHILE ! Empty (OperatorStack)

&& ! Precedence (Symbol, StackTop(OperatorStack) DO

x = Pop (OperatorStack)

Add x to prefix Expression

Push(OperatorStack , Symbol)

WHILE ! Empty(OperatorStack) DO

x = Pop(OperatorStack)

IF symbol is Reverse the infix expression

IF element is operand, Add this to prefix Expression

IF operator, find the precedence of current operator over stacktop symbol. If False, Pop and add the popped symbol to Prefix Expression

Push the operator if the precedence is true or stack is empty

Add the remaining symbols on stack to Postfix Expression by popping it one by one

Add x to Prefix Expression

RETURN Reverse (Prefix Expression) } Reverse and return the Postfix Expression

END;

Time Complexity: Θ(N)

The reverse is done twice in the logic requiring 2*CN effort. For any symbol, there are two decisions to make. If the symbol is an operand two statement is required to be executed (including condition). In case of operator, 3 statements in case the loop condition is true otherwise 1 statement for Push. Overall there are N symbols for which decision has to be made. The statement execution required are in the order of N.

Space Complexity: Θ(N)

The operator stack is required here which will be of size N/2. Some other variables are required that can be treated as constant space. Total space complexity in this case will be in the order of N.

5.6 Sort a stack

Here it is assumed that the elements are given in the stack. A process is applied to sort and store the numbers in the new stack. The procedure goes like this:

1. Take an empty temporary stack
2. Take an element out from the input stack, say a
3. The element a is compared with the top of the temporary stack (say b). If b is greater than a,
    - Element is Popped from temporary stack (say c)
    - c is pushed on the input stack
    - Step 3 is repeated until the condition is true
4. a is pushed on temporary stack
5. 2,3,4 are repeated until there are no elements in Input stack
6. Temporary stack is returned as the answer

Example:

Input stack	**[38, 3, 30, 105, 96, 27]**
Expected Output stack	**[3, 27, 30, 38, 96, 105]**

	Input: [38, 3, 30, 105, 96, 27]
Step 1	Element taken out: 27
	Input: [38, 3, 30, 105, 96]
	tmpStack: [27]
Step 2	Element taken out: 96
	Input: [38, 3, 30, 105]
	tmpStack: [27, 96]
Step 3	Element taken out: 105
	Input: [38, 3, 30]
	tmpStack: [27, 96, 105]
Step 4	Element taken out: 30
	Input: [38, 3, 105, 96]
	tmpStack: [27, 30]
Step 5	Element taken out: 96
	Input: [38, 3, 105]
	tmpStack: [27, 30, 96]
Step 6	Element taken out: 105
	Input: [38, 3]
	tmpStack: [27, 30, 96, 105]
Step 7	Element taken out: 3
	Input: [38, 105, 96, 30, 27]
	tmpStack: [3]
Step 8	Element taken out: 27
	Input: [38, 105, 96, 30]
	tmpStack: [3, 27]
Step 9	Element taken out: 30
	Input: [38, 105, 96]
	tmpStack: [3, 27, 30]
Step 10	Element taken out: 96

	Input: [38, 105]
	tmpStack: [3, 27, 30, 96]
Step 11	Element taken out: 105
	Input: [38]
	tmpStack: [3, 27, 30, 96, 105]
Step 12	Element taken out: 38
	Input: [105, 96]
	tmpStack: [3, 27, 30, 38]
Step 13	Element taken out: 96
	Input: [105]
	tmpStack: [3, 27, 30, 38, 96]
Step 14	Element taken out: 105
	Input: [ ]
	tmpStack: [3, 27, 30, 38, 96, 105]
	Final Sorted List: [3, 27, 30, 38, 96, 105]

ALGORTHM SortingUsingStack(STACK Input)
Input: An input stack
Output: sorted stack
BEGIN:

```
Stack TmpStack
Initialize(TmpStack)
```
Take a Temporary stack and initialize it

```
WHILE !IsEmpty(Input) DO
    a = Pop(Input)
```
Pop the element from the input stack

```
    WHILE !Empty(TmpStack) AND StackTop(TmpStack) > a DO
        C = Pop(TmpStack)
        Push(Input, C)

    Push(TmpStack, a)
```
If top element of temporary stack is greater than a, pop it from stack and push it in the input stack. If condition fails, push a in the temporary stack

```
RETURN TmpStack
```
Return the temporary stack that contains the sorted elements

END;

Time Complexity: $O(N^2)$

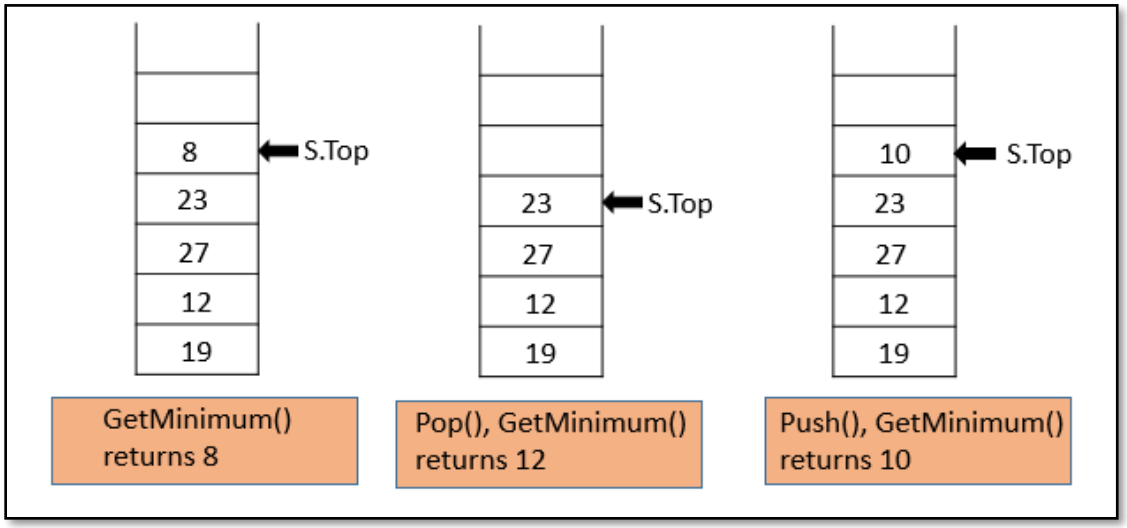
There are N symbols on the stack and stack is already reverse sorted, The effort required will be $1+2+3+\ldots+N$ i.e. $N(N+1)/2$ i.e. $O(N^2)$. This is the worst case. The best case appears when the input stack is reverse sorted. In this case the effort required will be in the order of N i.e. $\Omega(N)$.

Space Complexity: $\Theta(N)$

A stack of size N is required in this case plus some variables that amounts to constant space. Total space required hence is in the order of N.

5.7 Finding minimum from the stack

Apart from the usual primitive operations on the stack, we may be interested in finding additional information, e.g. Minimum or Maximum among all the elements. The same requires the traversal i.e. O(N) time with the traditional ways. So let us redefine the primitive operations such that finding minimum elements too requires O(1) time and space.



There could be three approaches to find the minimum from the stack

Method 1:

We can think of using a temporary stack. Set the top element of the original stack as minimum. Removing all the elements from the original stack while making the comparison with the minimum element simultaneously. The steps for this procedure is written below

1- Take a Temporary empty stack
2- Set the top element as Minimum

3- Pop element from the original stack while making the comparison with minimum element (update the Minimum element in case the current element is smaller than minimum)
4- Push the element in the Temporary stack
5- Repeat 3,4 until original stack is empty
6- Pop element from the Temporary stack
7- Push the element in the Original stack
8- Repeat 5, 6 until the Temporary stack is empty
9- Return the Minimum Element

Every time the minimum has to be found, the above procedure needs to be followed. There is no change in the definition of Push and Pop operations.

ALGORITHM MinimumFromStack(Stack S)
Input: A Stack
Output: Minimum Element
BEGIN:

```
Stack Temp
Initialize(Temp)
Min = StackTop(S)

WHILE !Empty(S) DO
        X=Pop(S)
        IF X<Min THEN
                Min = X
        Push(Tmp, X)
WHILE !Empty(Tmp) DO
        X=Pop(Tmp)
        Push(S, X)

RETURN Min
```

END;

Set the top element of the stack as Minimum

Pop the stack element and compare it with the minimum element. If this element is less than minimum, update minimum. Push this element in the temporary stack

Push back all the elements from Temporary stack to original stack

Return the minimum element

Time Complexity:
The entire stack needs to be shifted to temporary stack and restored another time. There are 2N Push and 2N Pop operations required in this case. Total 4N*O(N) time is required plus some constant time operations. Total effort hence is **Θ(N).**
Space Complexity:

A temporary stack here will amount to N space plus a minimum variable that will cause constant space. Total space is **Θ(N)**.

Method 2:

As method 1 is costly in terms of time and space, we can use the alternate method. In this we can use a temporary stack. When the Push and Pop operations are performed, the minimum element is stored as the top element in the temporary stack. The top of the Temporary stack always contains the minimum element. Push and Pop need to be redefined as given below.

Push(S, T, x): This operation pushes value x in the original stack S and stores the Minimum element in the Temporary stack.

ALGORITHM Push(S,T,x)
Input: A Stack S, Temporary Stack T, Element x to be inserted
Output: None
BEGIN:

```
S.Top=S.Top+1
S.Item[S.Top]=x
```

Insert the element at the top of Stack S

```
IF Empty(T) THEN
        T.Top=T.Top+1
        T.Item[T.Top] = x
```

If the Temporary stack is empty, Push element on this stack as well

```
ELSE
        IF StackTop(T) > x THEN
                T.Top = T.Top+1
                T.Item[T.Top] = x
```

If x is smaller than Top element of Stack T, then Push it at the Top of Stack T

```
        ELSE
                Y=T.Item[T.Top]
                T.Item[T.Top] = x
                T.Top=T.Top+1
                T.Item[T.Top] = y
```

If x is greater than Top element of Stack T, save the current top element, insert x at current Top , increment Top and insert the saved element at the updated Top position

END;

ALGORITHM Pop(S,T)
Input: A Stack S, Temporary Stack T
Output: Deleted Element
BEGIN:

```
X=S.Item[S.Top]
S.Top=S.Top-1
```

Remove the Top element from Stack S

```
        IF X == StackTop(T) THEN
                T.Top = T.Top-1
END;
```

If removed element is same as that in the Stack T Top element, Update the top of Stack T at one down index

ALGORITHM FindMinimum(T)
Input: A Stack T
Output: Minimum Element
BEGIN:

```
        RETURN T.Item[T.Top]
```

END;

Time and Space Complexity:
This approach requires O(1) time for Push, Pop and find Minimum (because of the new version of Push and Pop) and O(N) space (because of Temporary Stack). Thus, N represents the number of elements currently.

Method 3:
This method thinks of manipulating the data before storage. This approach requires only O(1) time and space. A variable **Minimum** is taken that always stores the minimum from the stack. This variable is modified upon each Push and Pop operation.

If the minimum element is removed with the Pop operation, Minimum element needs to change. To handle this, we push "2*x – Minimum" into the stack instead of x so that the previous minimum element can be retrieved using the current minimum and its value stored in stack. Below are detailed steps and an explanation of working.

Push(x) :

1. If the stack is empty, insert X into the stack and make Minimum equal to X.
2. If stack is not empty, compare X with Minimum element. There may be two cases here.
    a. If X is greater than or equal to Minimum, insert X.
    b. If X is smaller than Minimum, insert (2*X – Minimum) into the stack and make Minimum equal to X.

Pop() :

1. Remove element from top and store it in X.
2. If X is greater than or equal to Minimum, the minimum element remains the same.
3. If X is less than Minimum, the minimum element now becomes (2*Minimum – X), so update (Minimum = 2*Minimum – X).

Illustration of Sequence of Push Operations

Number to Push	Present Stack Bottom → Top	Minimum
3	3	3
5	3,5	3
2	3,5,1	2
1	3,5,1,0	1
1	3,5,1,0,1	1
-1	3,5,1,0,1,-3	-1

Illustration of Sequence of Pop Operations

Number Removed	Original Number	Present Stack Bottom → Top	Minimum
		3,5,1,0,1,-3	-1
-3	-1	3,5,1,0,1	1
1	1	3,5,1,0	1
0	1	3,5,1	2
1	2	3,5	1
5	5	3	-1

How does this approach work?

When the element to be inserted (X) is found less than the current Minimum, we insert "2X – Minimum" instead of X. The term "2X – Minimum" will be less than X, i.e., new Minimum.

While removal, if the popped element is less than Minimum, the Minimum will be updated.

To Prove that "2*X – Minimum" is less than X

X < Minimum

i.e. X - Minimum < 0 ---------- (1)

Add X on both sides in (1)

X - Minimum + X < 0 + X

2*X - Minimum < X

Hence Proved

This can be concluded that 2*X - Minimum < new Minimum

While carrying out the Pop operation, if the removed element (Y) is less than the current Minimum, we find the new Minimum as
Minimum = 2*Minimum – Y.

We pushed y as 2X - prevMinimum.
prevMinimum is Minimum before y was inserted
y = 2*X - prevMinimum

Value of Minimum was made equal to X
Minimum = X .

new Minimum = 2 * Minimum - y
= 2*X - (2*X – prev Minimum)
= prev Minimum

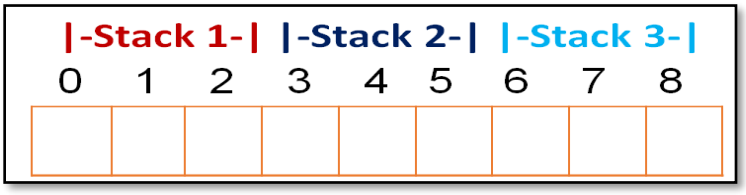
5.8 Implementing multiple Stacks in a single Array

Consider a situation where a large-sized array has to be utilized for storage of multiple stacks. There are 2 methods by which this can be done.

- Logical division of the Array in the equal-sized chunks (No of chunks equal to no of desired stacks)
- Sharing of memory regions of Array by 2 Stacks

Method 1 can be utilized for many stacks; method 2 can be used only for 2 stacks.

Method 1: Logical division of Array
In this method, a single Array will store multiple stacks. Therefore, there will be a different Top for each stack in that Array (Top1, Top2, …)
Let Array size (N) =9
Number of stack (M) =3
Size of each stack = N/M =9/3=3



Stack 1 has indexes 0,1,2
Stack 2 has indexes 3,4,5
Stack 3 has indexes 6,7,8

If N=8, then stack size = N/M = 2.6. This is unacceptable as size cannot be in a fraction. Hence, we select the sizes as 3,3,2.

Method of implementation involves the calculation of Initial top of each Stack. We consider the stack numbering as 0, 1, 2, …

- For stack 0, initial top = 0 * 3 -1 = -1
- For stack 1, initial top = 1 * 3 -1 = 2
- For stack 2, initial top = 2 * 3 -1 = 5
- For stack i, initial top = i * N/M -1

For writing generalized push operations in the i^{th} Stack, we should decide about the overflow condition. If the top of i^{th} stack reaches to the upper limit, this will indicate the condition of overflow. Overflow occurs at (i+1)*N/M -1.
If Array size (N) is 15
No of stacks (M) = 3

Slots for Stack 0 are 0,1,2,3,4
Slots for Stack 1 are 5,6,7,8,9
Slots for Stack 2 are 10,11,12,13,14

Overflow for stack 0 will occur if Top has reached (0+1)*15/3 – 1 = 4
Overflow for stack 1 will occur if Top has reached (1+1)*15/3 – 1 = 9
Overflow for stack 0 will occur if Top has reached (2+1)*15/3 – 1 = 14
Considering above, Push can be written as

ALGORITHM MSPush(A[ ], Ti, i, item)
Input: Array A[ ], Top Ti, stack No i, item to be inserted
Output: None
BEGIN:

```
IF Ti== (i+1)*N/M -1 THEN
        WRITE("Stack i overflows")
        EXIT(1)
Ti=Ti+1
A[Ti]=item
```

If no more insertion possible.

Increment in Top index by 1.
Insert the data item at the Top index of Stack.

END;

For writing generalized pop operations in the i^{th} Stack, We should decide about the underflow condition. If the Top of i^{th} stack reaches the initialized Top, this will indicate underflow condition. Underflow occurs at (i)*N/M -1. Considering this, Pop can be written as

ALGORITHM MSPop(A[ ], Ti, i)
Input: Array A[ ], Top Ti, stack No i
Output: Deleted element
BEGIN:

```
        IF Ti== i*N/M -1 THEN
                WRITE("ith stack underflows")
                EXIT(1)
        x=A[Ti]
        Ti=Ti-1
        RETURN x
```

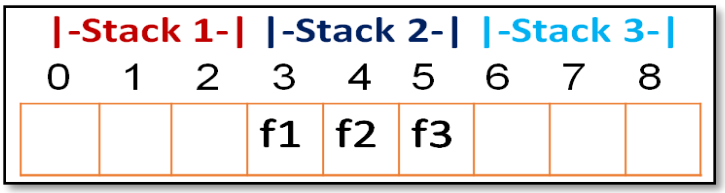
END;

Underflow condition check i.e., if stack empty.

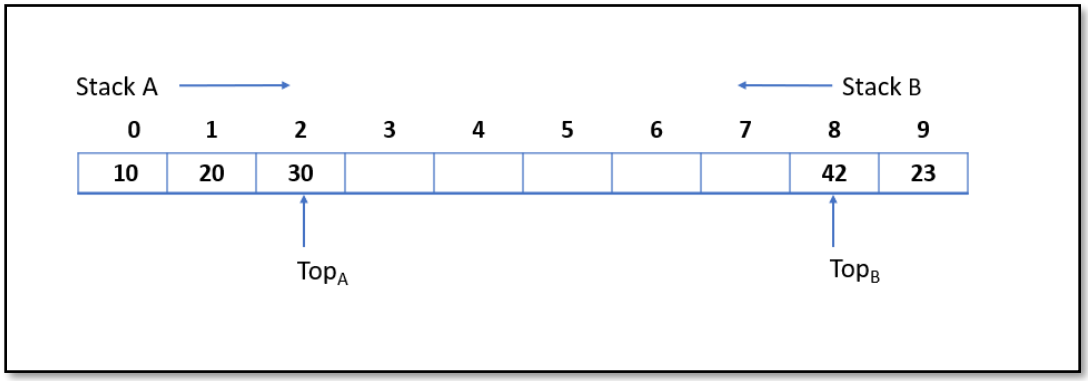
Saving the Top index element in x.
Decrementing the Top index by 1.
Returning deleted item.

Advantage of Multiple Stack single array (MSSA)
Better memory utilization as compared to using single stack. Multiple Recursive programs can be designed considering partitioned array with defined boundaries.

The drawback of Multiple Stack single Array (MSSA)
Refer to the diagram below. A lot of space is empty in stack1 and stack3. Stack 2 is full as per the given computations. In case we run the Push Algorithm for stack 2, it will result in overflow. Hence MSSA can be referred to as inefficient.



Method 2: Sharing of Array memory by two Stacks

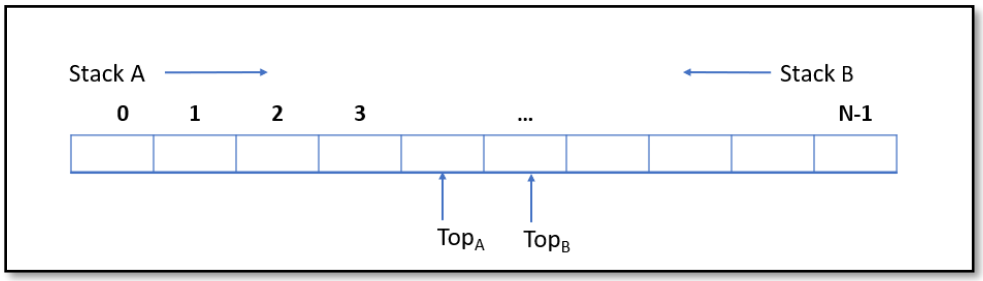


A single stack is sometimes not sufficient to store a large amount of data. To overcome this problem, we can use multiple stacks in a single array. For implementing two stacks on a single array we can consider that

- Let there be an array **A[N]** divided into two stacks **Stack A**, **Stack B.**
- **Stack A** expands from left to right, i.e. from 0^{th} index onwards.
- **Stack B** expands from right to left, i.e, from $(N-1)^{th}$ index to backward.
- The combined size of both **Stack A** and **Stack B** never exceeds N.

- Both Stacks are full when $Top_A = Top_B - 1$.
- Both Stacks are empty when $Top_A = -1$ & $Top_B = N$.

In this strategy, one stack will start from left and other will start from right. For Push, in the first stack, Top will be incremented. For Push, in the second stack, Top will be decremented.



ALGORITHM InitializeStack(TopA, TopB)
Input: Top indexes of both the stacks
Output: None
BEGIN:
TopA = – 1
TopB = N
Initializing TopA at -1
Initializing TopB at N
END;

ALGORITHM MSPushA(A[ ], TopA, TopB, item)
Input: Array A[ ], Top indexes of both the stacks, item to be inserted
Output: None
BEGIN:

```
IF TopA== TopB – 1 THEN
        WRITE("Stack A overflows")
        EXIT(1)
TopA=TopA+1
A[TopA]=item
```

If no more insertion possible.

Increment in TopA index by 1.
Insert the data item at the TopA index of

END;

ALGORITHM MSPushB(A[ ], TopA, TopB, item)
Input: Array A[ ], Top indexes of both the stacks, item to be inserted
Output: None
BEGIN:

```
IF TopA== TopB – 1 THEN
        WRITE("Stack B overflows")
        EXIT(1)
TopB=TopB – 1
A[TopB]=item
```

If no more insertion possible.

Decrement in TopB index by 1.
Insert the data item at the TopB index of Stack.

END;

ALGORITHM MSPopA(A[ ], TopA, TopB)
Input: Array A[ ], Top indexes of both the stacks
Output: Deleted element
BEGIN:

```
IF TopA== – 1 THEN
        WRITE("Stack A underflows")
        EXIT(1)
x=A[TopA]
TopA=TopA – 1
RETURN x
```

Underflow condition check i.e., if stack empty.

Saving the TopA index element in x.
Decrementing the TopA index by 1.
Returning deleted item.

END;

ALGORITHM MSPopB(A[ ], TopA, TopB)
Input: Array A[ ], Top indexes of both the stacks
Output: Deleted element
BEGIN:

```
IF TopB == N THEN
```

Underflow condition check i.e., if stack empty.

```
            WRITE("Stack B underflows")
            EXIT(1)
        x=A[TopB]
        TopB=TopB + 1
        RETURN x
END;
```

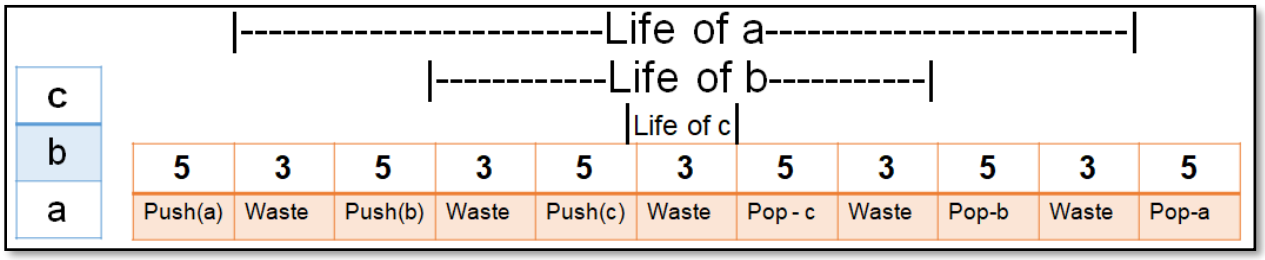
Saving the TopB index element in x.
Incrementing the TopB index by 1.
Returning deleted item.

5.9 Lifetime of an element in Stack

Lifetime of an element in stack is the time after the Push operation of that element and before the Pop operation of that element.

Example

Three elements (a, b, c) are pushed in the stack in the sequence and followed by pop. Let us assume that Push() and Pop() operations take 5 units of time each and time lap between successive operations is 3 units.



As per the diagram above,

- Life of c= 3
  Life of b= 19
  Life of a= 35
- Average Life of all elements = (3+19+35)/3

5.11 Competitive Coding Problem

You are given a string s consisting of lowercase English letters. A duplicate removal consists of choosing two adjacent and equal letters and removing them.

We repeatedly make duplicate removals on s until we no longer can.

Return the final string after all such duplicate removals have been made. It can be proven that the answer is unique.

Input: s = "abbaca"

Output: "ca"

Explanation:

For example, in "abbaca" we could remove "bb" since the letters are adjacent and equal, which is the only possible move. The result of this move is that the string is "aaca", of which only "aa" is possible, so the final string is "ca".

Solution: The problem is more like the Push-Down automata. The process requires storing elements on the stack and deleting the next character in the string matches with the stack top character. The resulting stack is the answer. Since the answer has to be given in the form of a string, a string is formed by repeated deletion of elements from the the stack when the NULL character appears from the original string.

Expected time complexity: - O(n)

ALGORITHM DuplicateCheck(Str[ ])

BEGIN:

```
STACK S
InitializeStack(S)
i=0
j=0
WHILE Str[i]!='\0' DO
        IF IsEmpty(S) THEN
                Push(S,Str[i])
        ELSE
                IF Str[i] == StackTop(S) THEN
                        Pop(S)
                        j=j+1
                ELSE
                        Push(S,Str[i])
        i=i+1
N=i-2*j
Str2[N]='\0'
N=N-1
WHILE !IsEmpty(S) DO
        Str2[N]=Pop(S)
        N=N-1
RETURN Str2
```

END;

Time Complexity: The Complexity of the above algorithm is O(N) as we require the traversal on string once and another time on Stack for Pop operations. Even in the worst case, when no matching characters are found, the complexity will be O(N).

Space Complexity: A stack of maximum Size N and string of maximum size N is required. Along with this, three extra variables are required. As a result, the total space required is N+N+3 = 2N+3. Hence the space complexity is O(N).

5.12 Precedence and Associativity Complete chart

Operator	*Description*	*Associativity*
() [] . -> ++ --	Parentheses or function call Brackets or array subscript Dot or Member selection operator Arrow operator Postfix increment/decrement	left to right
++ -- + - ! ~ (type) * & sizeof	Prefix increment/decrement Unary plus and minus not operator and bitwise complement type cast Indirection or dereference operator Address of operator Determine size in bytes	right to left
* / %	Multiplication, division and modulus	left to right
+ -	Addition and subtraction	left to right
<< >>	Bitwise left shift and right shift	left to right
< <= > >=	relational less than/less than equal to relational greater than/greater than or equal to	left to right
== !=	Relational equal to or not equal to	left to right
&&	Bitwise AND	left to right
^	Bitwise exclusive OR	left to right
|	Bitwise inclusive OR	left to right
&&	Logical AND	left to right
||	Logical OR	left to right
? :	Ternary operator	right to left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment operator Addition/subtraction assignment Multiplication/division assignment Modulus and bitwise assignment Bitwise exclusive/inclusive OR assignment	right to left
,	comma operator	left to right

Exercises

1. Convert following Infix expression into Postfix expression using Tabular method.

(a – b / c) * (d + (e * f) / g)

2. Solve the following:

a) ((A- (B + C) * D) / (E+ F)) [Infix to postfix]

b) (A + B) *C – (D – E) ^ F [Infix to prefix]

c) 7 5 2 + / 4 1 5 – / – [Evaluate the given postfix expression]

d) A B C + / D E F – / – [Postfix to Infix]

e) A B C + / D E F – / – [Evaluate the given postfix expression]

A=24, B=3, C=1, D=14, E= 9, F=7

f) A-B/C+D*E+F [Infix to postfix]

g) A *((B+D)/E-F) * (G+H/I) [Infix to postfix]

3. The Order followed by stack data structure is

a. Random b. FIFO c. LIFO d. None

4. How many stack will be needed for the evaluation of a prefix expression

a. 1 b. 2 c. 0 d. 3

5. If the two stacks are implemented on a single array, the overflow occurs at

a. top1=top2 b. top1=top2-1 c. top1-1=top2 d. none of these

6. If PRCD(x, y) is a function that returns TRUE is x has higher precedence that y or if x has the same precedence as y but left associative. A call for PRCD(↑,↑) will return (↑ is exponentiation)

7. The evaluated value of POSTFIX EXPRESSION 4 3 * 4 2 / - 3 2 $ 2 * + is ……

8. Write precedence function for bitwise logical operators.

9. Write an algorithm for evaluation of Postfix expression containing bitwise logical operators only.

10. Best suited data structure to find the validity of mathematical expression with all types of brackets e.g. [ ], { }, ( ) is ……

11. What are the notations used in Evaluation of Arithmetic Expressions using prefix and postfix forms?

12. The stack can be used to find if the given string is palindrome or not. No of POP operations required to do the same on the string **MISSISSIM** is …….

13. Evaluated value of expression - + * 4 2 9 / 6 2 is ……..

14. For conversion of infix expression given below to postfix, no of PUSH and POP operation performed is……..

a * b + c / d ↑ e ↑ f

15. The expression **a ↑ b ↑ c * d *e / f / g / h** in polish notation is ……

16. Write an Algorithm that can handle unary minus operators also while conversion of infix to postfix.

17. Write an Algorithm that can handle unary minus operators also while Evaluation of Postfix expression.

Multiple Choice Questions

1. The following sequence of operation is performed on a stack S: PUSH(23), PUSH(34), PUSH(20), POP, POP, PUSH(54), PUSH(35), POP, POP, POP, PUSH(4), POP. The sequence of values popped out and the elements present at the top of stack after all these operations are.
23, 34, 20, 54, 35, 4 and s[top] = NULL
23, 34, 35, 54, 4 and s[top] = 20
23, 54, 35, 34, 20 and s[top] = 4
20, 34, 35, 54, 23, 4 and s[top] = NULL
20, 34, 35, 54, 23, 4 and s[top] = NULL
Medium

2. Which of the following permutations can be obtained in the output (in the same order) using a stack assuming that the input is the sequence 2, 13, 46, 75, 19.
46, 2, 13, 19 , 75.
46, 19, 2, 13, 75.
2, 46, 13, 19, 75.
2, 75, 13, 46, 19.
2, 46, 13, 19, 75.
Difficult

3. Suppose we need to check whether a given string is palindrome or not. What will be the minimum number of comparisons required in order to check this?

Exactly n comparisons.
Exactly n-1 comparisons.
n/2 comparison.
None of these.
n/2 comparisons
Medium

4. What will be the maximum size of stack used for performing the following sequence of operations? **PUSH(23), PUSH(32), POP, PUSH(3), PUSH(45), POP, PUSH(1), POP, POP, PUSH(9)** ?
2
3
1
4
3
Easy

5. What is the time complexity of Push and Pop operation in Stack?
O(1) for Push and O(n) for Pop.
O(n) for Push and O(n) for Pop.
O(1) for Push and O(1) for Pop.
O(n) for Push and O(1) for Pop.
O(1) for Push and O(1) for Pop.
E

6. Consider two stacks stored in the memory with the basic operations modified as follows: When a new element is pushed, the value is pushed into the first stack. The pop operation transfers all the elements of stack-1 to stack-2 and then pops the first element from the second stack. What will be the complexity of performing the pop operation?
O(1)
O(n)
O(log(n))
O(n^2)
O(n)
Medium

7. Which data structure is more appropriate to use while calculating factorial of a number using recursion?

Queue
Array
Stack
None of these
Stack
Easy

8. Assume that the operators +,−,× are left associative and ^ is right associative. The order of precedence (from highest to lowest) is ^, ×, +, −. The postfix expression corresponding to the infix expression a – b × c + d ^ e ^ f is?
abc-xde^+f^
abcx-def^^+
abcx-de^f^+
a+b×c-d^e^f^+
abcx-def^^+
Medium

9. Stack A has four values a, b, c, d which are inserted in reverse order with a at the top of the stack. Elements popped from stack A are pushed in Stack B. Determine the sequence that is not possible while popping out the elements from stack B?
a, b, c, d
d, c, b, a
d, b, c, a
a, c, b, d
d, b, c, a
Medium

10. What is the time complexity required to reverse the input string of length n using stack?
O(1)
O(n)
O(logn)
O(n^2)
O(n)
Easy

11. Suppose the following postfix expression abc*de+*- is evaluated, what will be the first two operands that will be evaluated? Assume a = 3, b = 4, c = 2, d = 4, e = 5.

a, b
b, c
c, d
d, e
b, c
Easy

12. Suppose the following postfix expression abc*de+*- is evaluated, what will be the value when first * is evaluated? Assume a = 3, b = 4, c = 2, d = 4, e = 5.
72
12
18
None of these
None of these
Medium

13. Consider the following operation performed on a stack of size 7. Push(3); Pop(); Push(5); Push(1); Pop(); Push(10); Pop(); Pop(); Push(2); After the completion of all operation, the no of unfilled cell present on stack are
1
2
4
6
6
Easy

14. Consider the following operation performed on a stack of size 6. Push(34); Pop(); Push(15);

Push(19); Pop(); Push(1); Pop(); Pop(); Pop(); Push(21); Push(5) ; Push(8); Pop(); After the completion of all operation, the no of filled cell present on stack are
1
2
3
4
2
Easy

15	Postfix Equivalent of A↑B↑C↑D↑E is
A	ABCDE↑↑↑↑
B	AB↑C↑D↑E↑
C	ABC↑↑DE↑↑
D	ABCD↑↑↑E↑
AN	**A**
DL	**M**

16	Which of the following applications may use a stack?
A	A parenthesis balancing program.
B	Keeping track of local variables at run time.
C	Syntax analyzer for a compiler.
D	All of the above.
AN	**D**
DL	**M**

17	Consider the following pseudocode: declare a stack of characters while (there are more characters in the word to read)

	{
	read a character
	push the character on the stack
	}
	while (the stack is not empty)
	{
	write the stack's top character to the screen
	pop a character off the stack
	}
	What is written to the screen for the input "carpets"?

A	serc
B	carpets
C	steprac
D	ccaarrppeettss
AN	**C**
DL	**M**

18	declare a character stack
	while (more input is available)
	{
	read a character
	if (the character is a '(')
	push it on the stack
	else if (the character is a ')' and the stack is not empty)
	pop a character off the stack
	else
	print "unbalanced" and exit
	}
	print "balanced"
	Which of these unbalanced sequences does the above code think is balanced?

A	((())
B	())(()
C	(()()))
D	((()))()
AN	**C**
DL	**M**

19	Here is an infix expression: 4+3*(6*3-2). Suppose that we are using the usual stack algorithm to convert the expression from infix to postfix notation. What is the maximum number of symbols that will appear on the stack AT ONE TIME during the conversion of this expression?
A	1
B	2
C	3
D	4
AN	**D**
DL	**M**

20	Which among the following algorithm requires application of stacks (directly or indirectly)
A	Quick sort
B	Heap sort
C	Selection sort
D	Bubble sort
AN	**A**
DL	**M**

21	PUSH and POP operations for evaluation of postfix expression 4 3 2 ↑ * 6 – 8+ is
A	9 PUSH, 8 POP
B	5 PUSH, 4 POP
C	5 PUSH, 9 POP
D	9 PUSH, 9 POP
AN	**D**
DL	**M**

22	Which among the following algorithms requires reversing the input string before processing through stacks?
A	Infix to postfix conversion
B	Infix to prefix conversion
C	Postfix evaluation
D	None of these
AN	**B**
DL	**M**

23	26. Let multipop(S, k) is an operation that removes k items from the stack S. In which of the following algorithms can the multipop() function be applied?
A	Infix to postfix conversion
B	Infix to prefix conversion
C	Postfix evaluation
D	None of these
AN	**B**
DL	**M**

24	Let there be a stack of integer values. How many elements does the stack contain after following operations PUSH(S, 1), PUSH(S, 2), … , PUSH(S, 10) Multipop(S, 5) PUSH(S, 11), PUSH(S, 12), … , PUSH(S, 30) Multipop(S, 10)
A	20
B	30
C	10
D	15
AN	**B**
DL	**M**

25	The postfix form of the expression (A+ B)*(C*D- E)*F / G is?
A	AB+ CD*E - FG /**
B	AB + CD* E - F **G /
C	AB + CD* E - *F *G /
D	AB + CDE * - * F *G /
AN	**C**
DL	**M**

26	The prefix form of A-B/ (C * D ↑ E) is?
A	-/* ↑ ACBDE
B	-ABCD* ↑ DE
C	-A/B*C ↑ DE
D	-A/BC* ↑ DE
AN	**C**
DL	**M**

27	The prefix form of an infix expression p + q - r * t is?
A	+ pq - *rt
B	- + pqr * t
C	- + pq * rt
D	- + * pqrt
AN	**C**
DL	**M**

28	The result of evaluating the postfix expression 5, 4, 6, +, *, 4, 9, 3, /, +, * is?
A	600
B	350
C	650
D	588
AN	**B**
DL	**M**