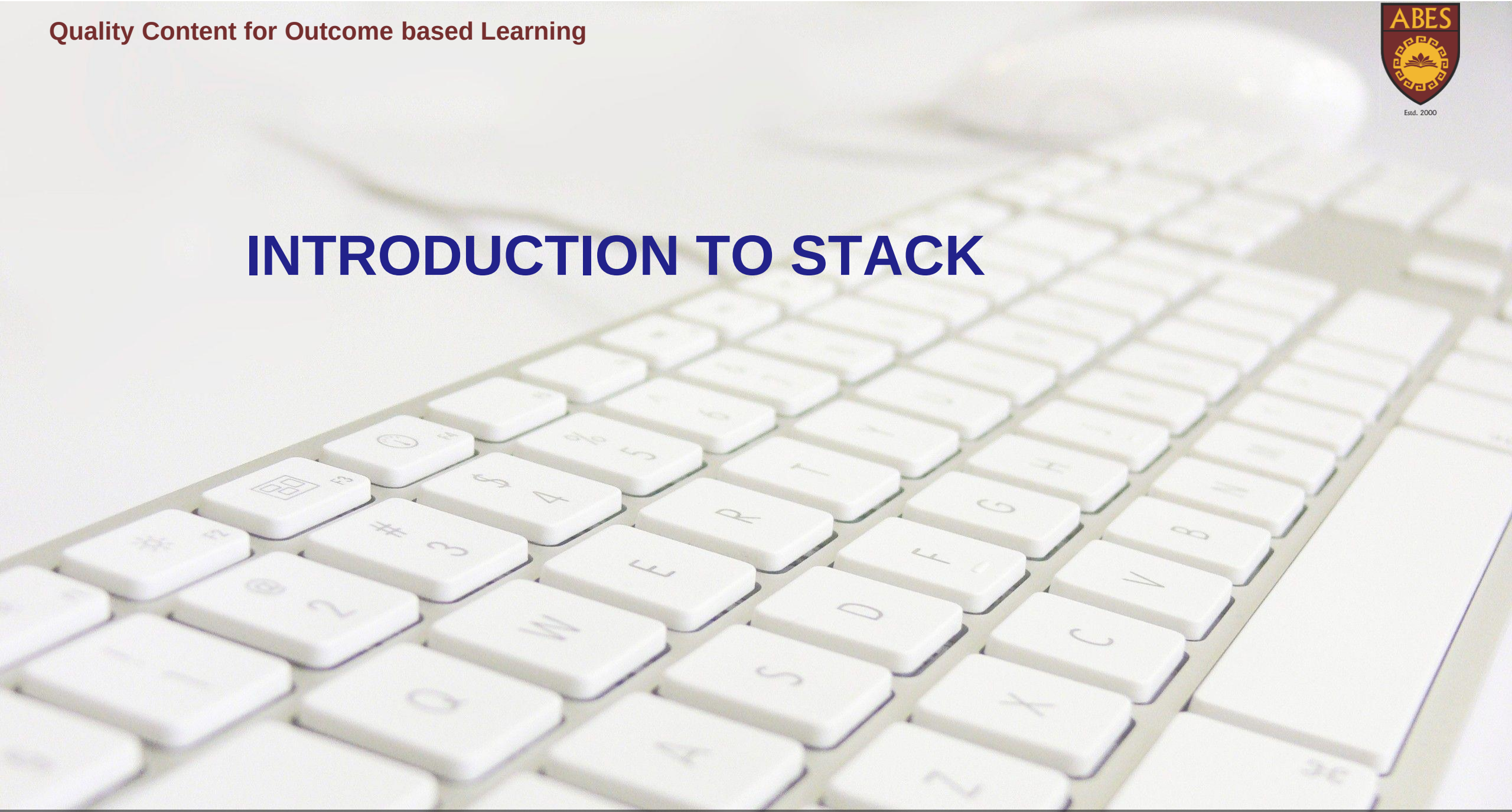
General Guideline

References

- "Fundamentals of **data structure** in C" Horowitz, Sahani & Freed, **Computer Science**.
- "Fundamental of **Data Structure**" (Schaums Series).
- Robert Kruse, Data Structures and Program Design , Prentice Hall, 1984.

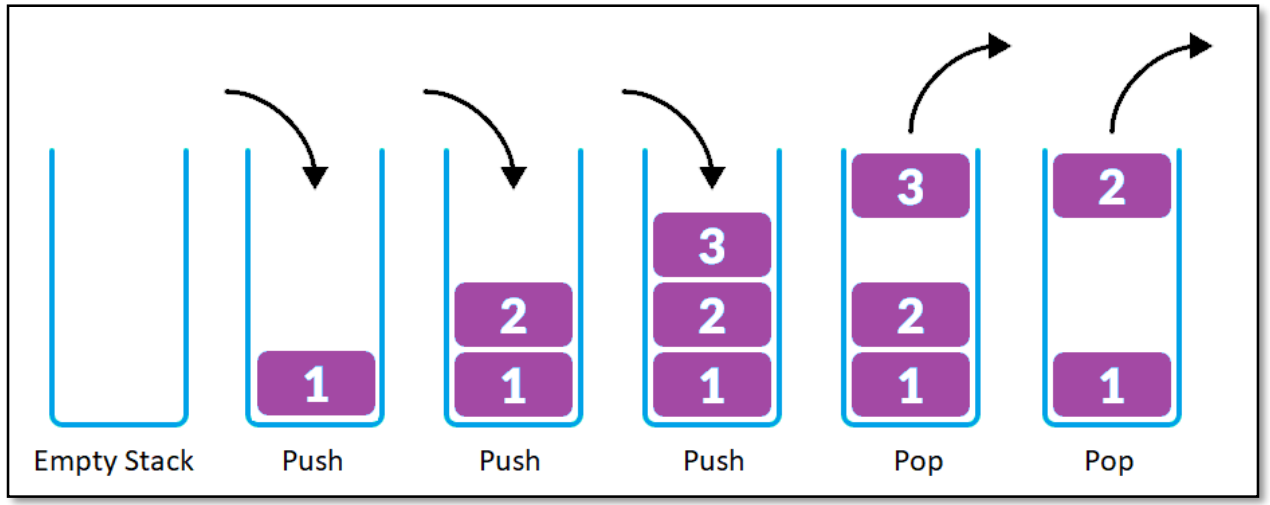
ABES
Estd. 2000

INTRODUCTION TO STACK

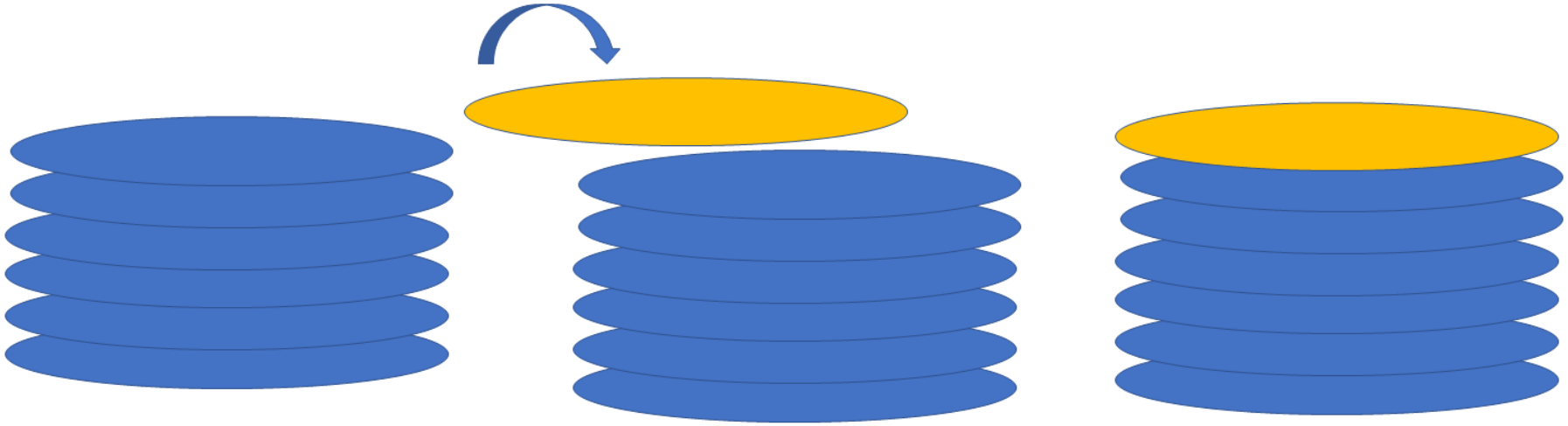


Introduction to Stack

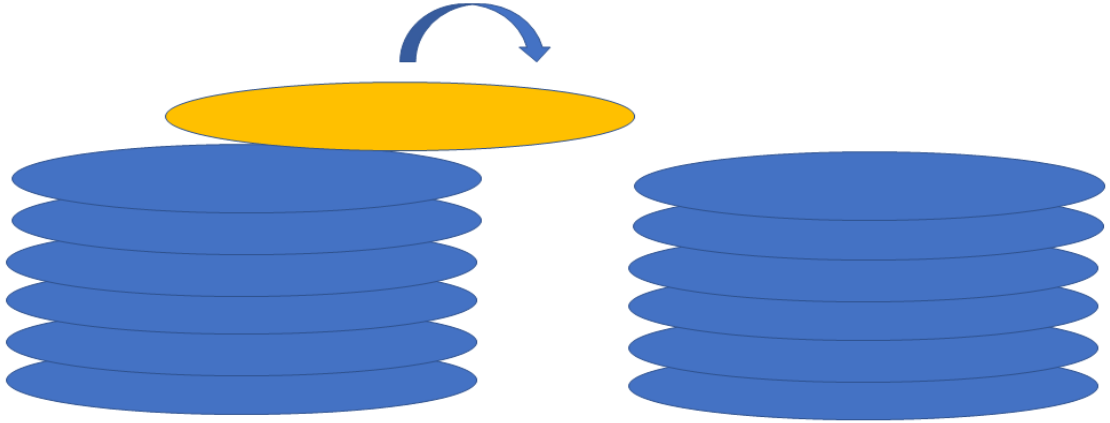
- A stack is a conceptual structure consisting of a set of homogeneous elements and is based on the principle of last in first out (LIFO). Stack is a data structure where elements can be inserted and deleted from one end only known as '**Top**' of the Stack.
- Insertion in Stack is given a standard name **Push** and deletion is given a standard name **Pop**.



Introduction to Stack - Example



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



Deletion of last inserted plate

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

Array Implementation of Stack

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.

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.

Stack Initialization

- **ALGORITHM InitializeStack(S)**

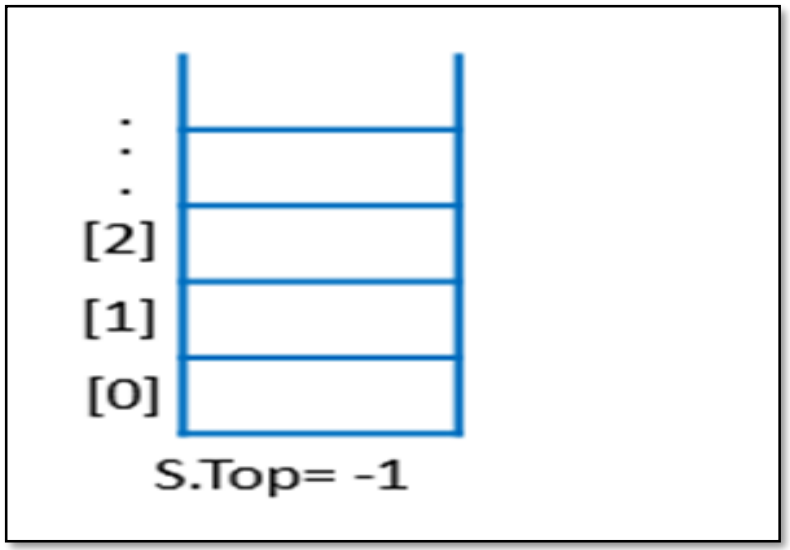
  Input: Stack S

  Output: None

BEGIN:

S.Top = – 1 } Initializing Top at -1

END;



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).

Emptiness Check

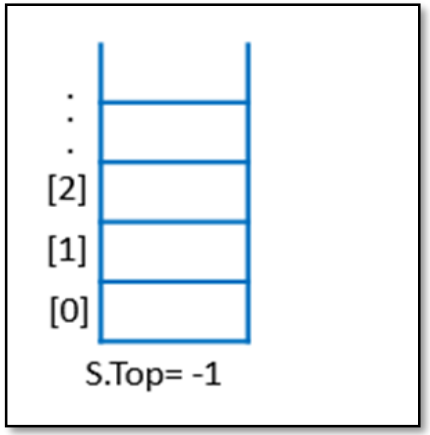
- If the Top is -1, stack is empty.
- **ALGORITHM IsEmpty(S)**

Input: Stack S

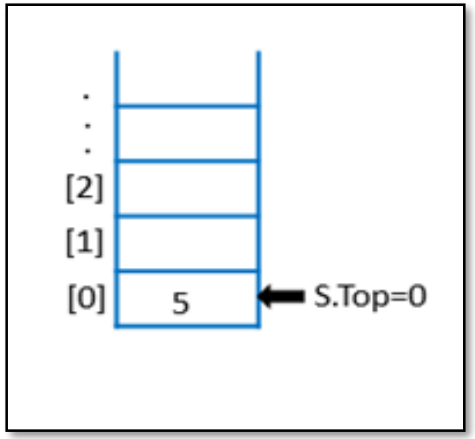
Output: True or False based on emptiness

```
BEGIN:
IF (S.Top == -1) THEN
        RETURN TRUE
ELSE
        RETURN FALSE
END;
```

S.Top=-1 indicates stack is empty



Empty Stack

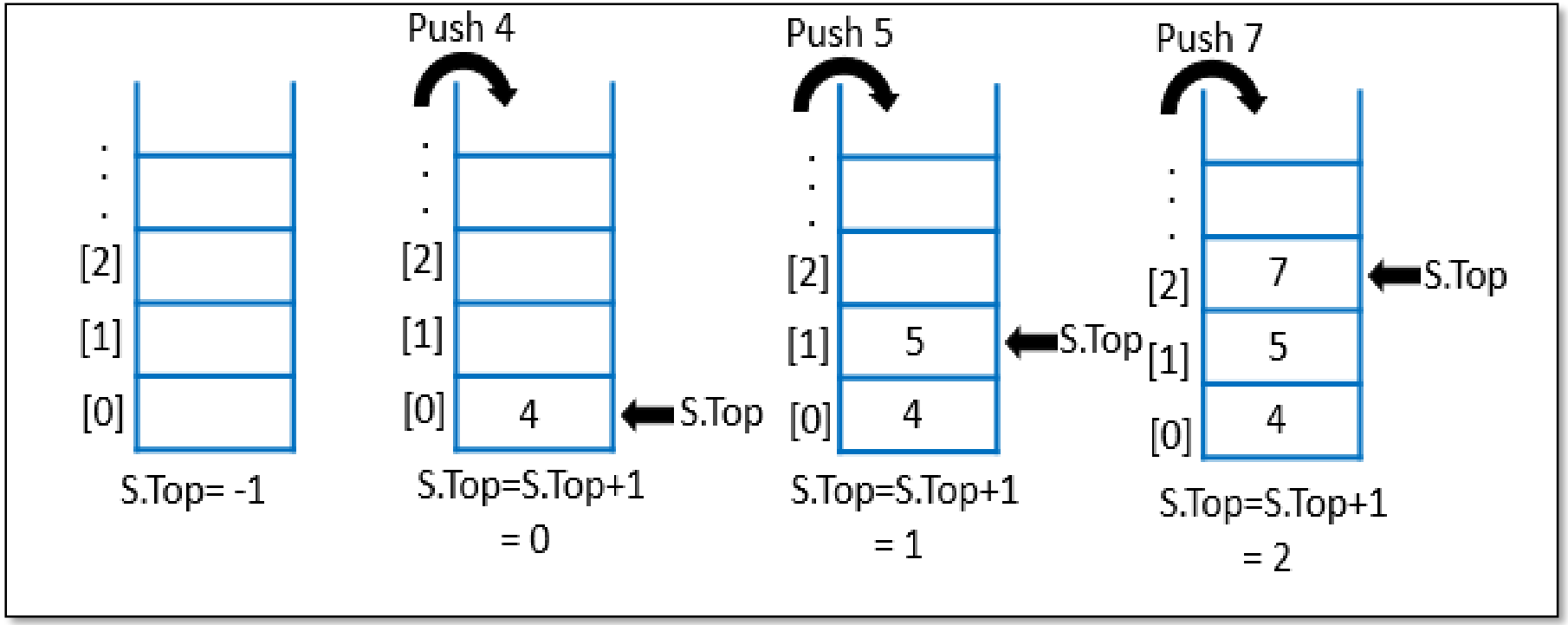


Non Empty Stack

- **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).

Stack Insertion

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



Insertion Process

Stack Insertion

- 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.*

Stack Insertion

- **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
```

END;

If no more insertion possible.

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

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.*

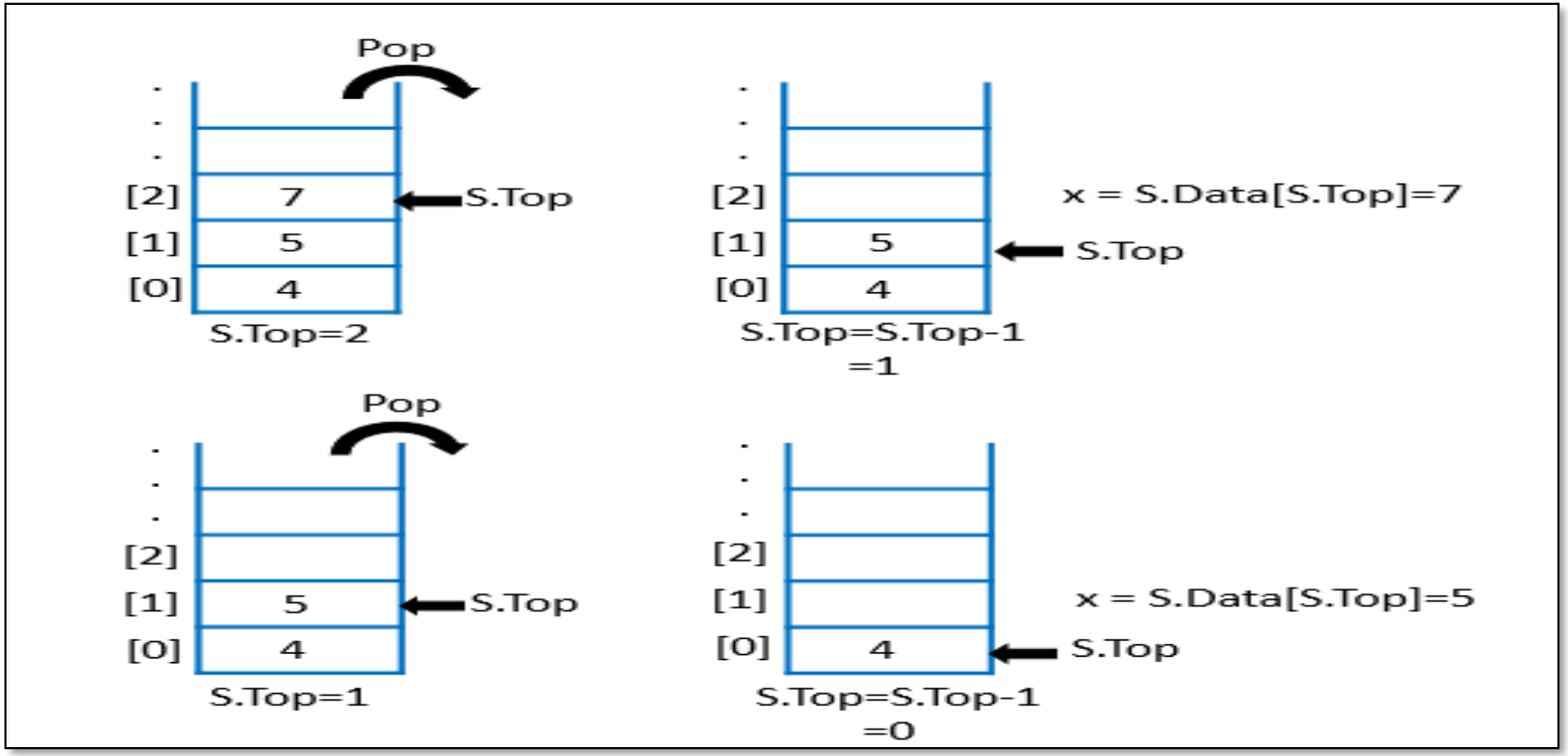
Stack Insertion

- **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).

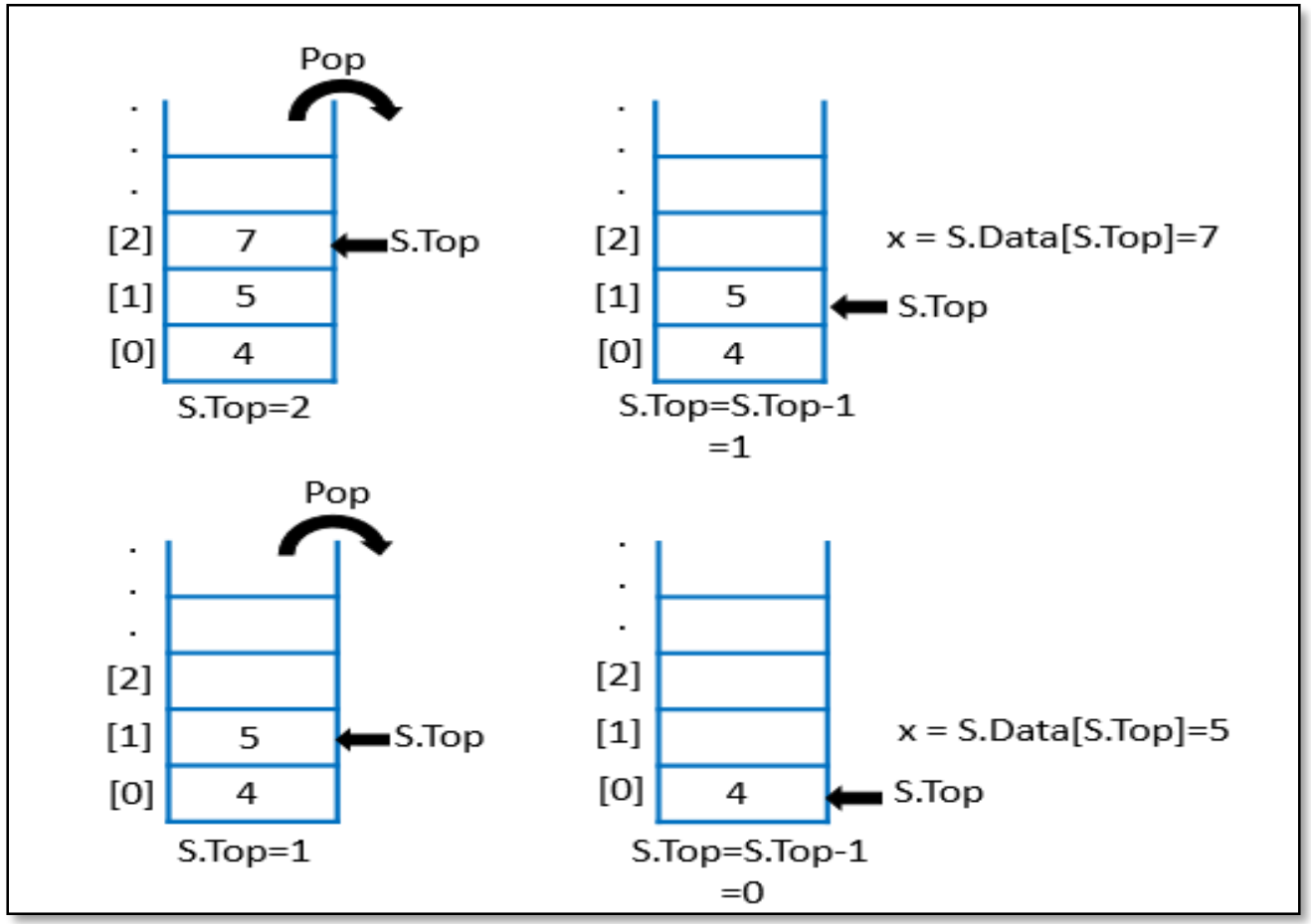
Deletion in a Stack

- 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.



Deletion in a Stack

- 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.



Deletion in a Stack

- **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;

Deletion in a Stack

- **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).

Finding the top element

➢ **ALGORITHM Peek (S)**

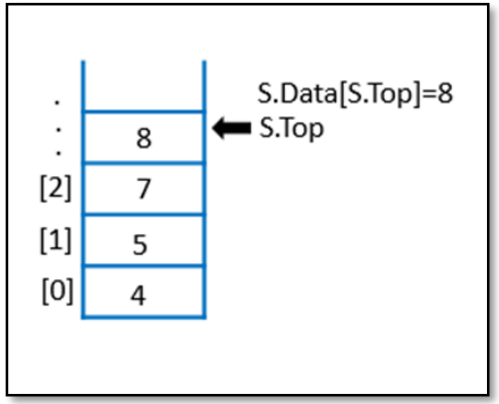
Input: Stack S

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).

Applications of Stack

Number Conversion

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

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.

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)
```

END;

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

Printing the Stack elements until Stack is empty.

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)
```

END;

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.

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. "

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.

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”.

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
```

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

Palindrome Check using Stack

```
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

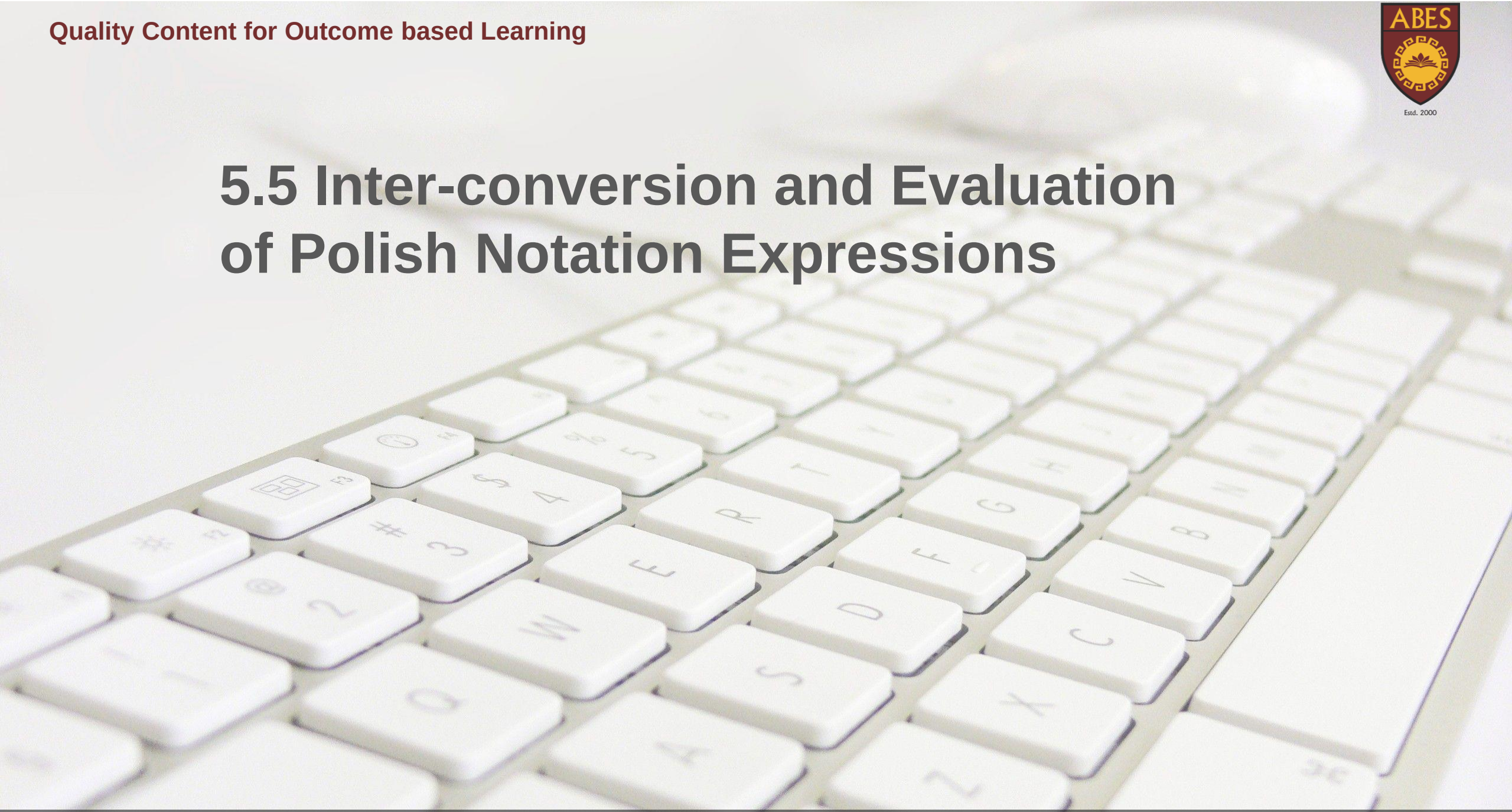
If top element of stack is matched with current character of string,
Pop the element from Stack.
Terminate the process otherwise
Process is repeated till Stack becomes empty

If all characters have matched

If any character had a mismatch

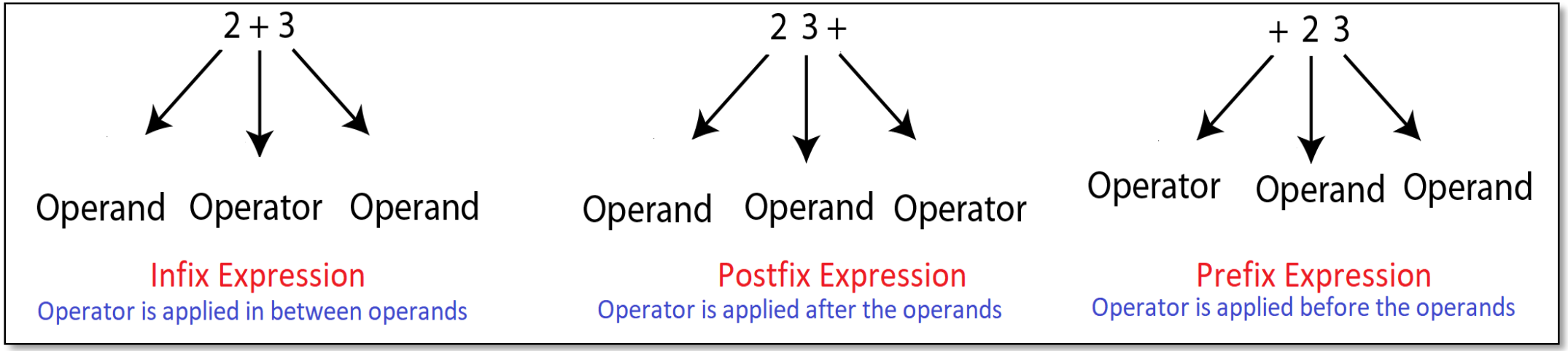
ABES
Estd. 2000

5.5 Inter-conversion and Evaluation of Polish Notation Expressions



Types of Expression

- 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:



- 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:

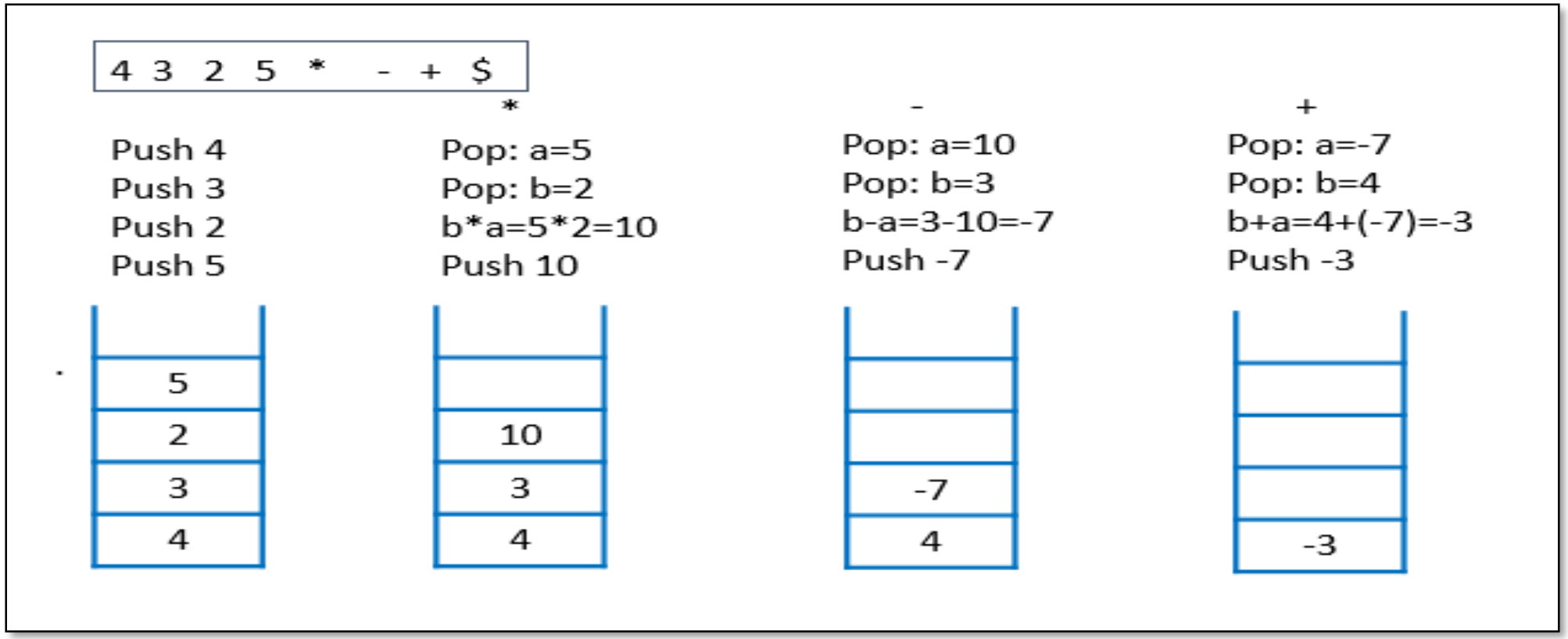
  2+5*6 – 8/4+3↑2 – (4/2)

  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).
- 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 stack top 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

- The precedence and associativity of operators are not required to be checked while evaluating the expression using stack.



Continue..

- **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
```

END;

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

Time & Space Complexity of Polish Expression

- **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 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 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

Procedure: To evaluate the prefix expression

T he 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

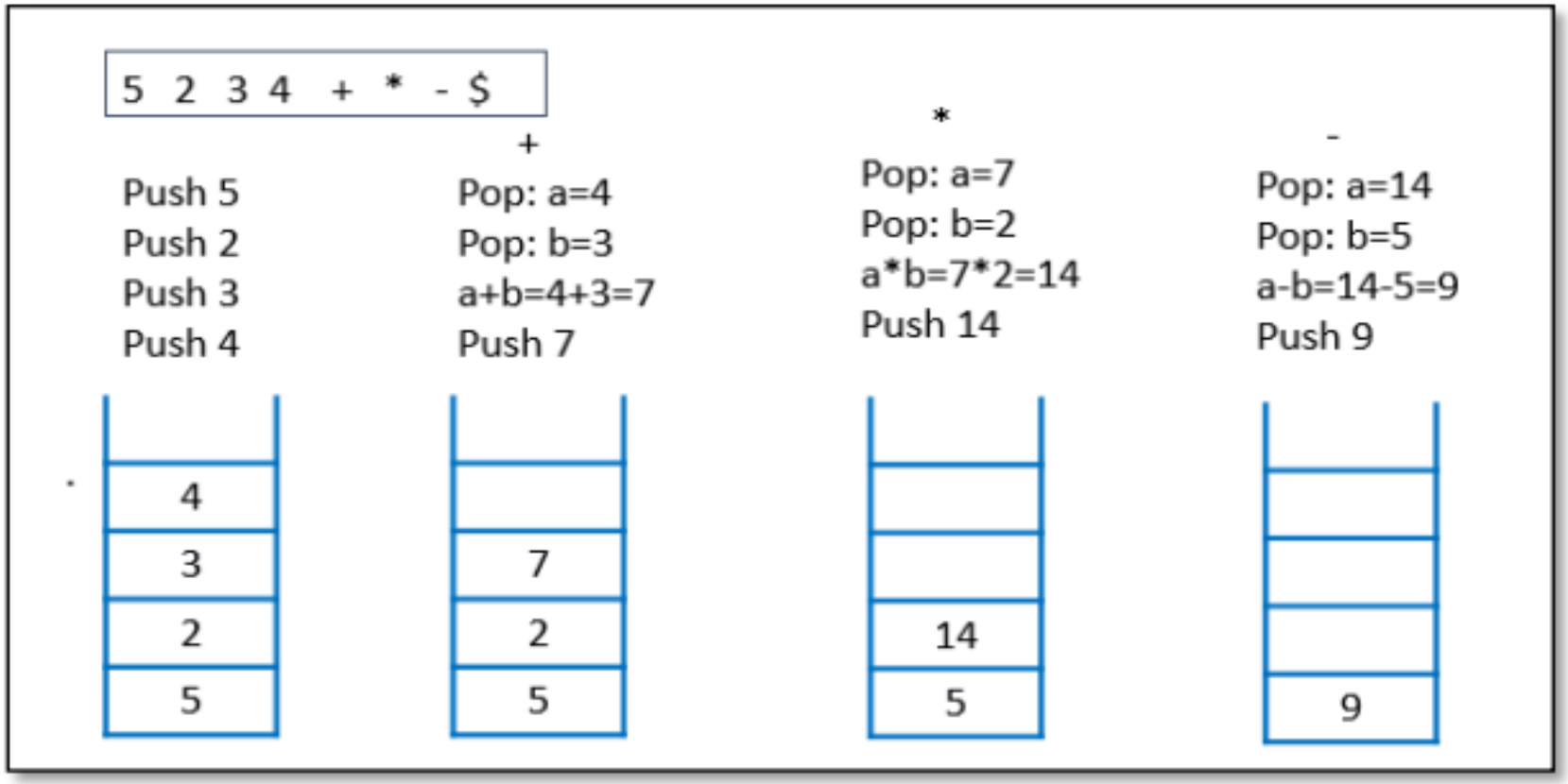
Apply the operator on a and b and push the result in stack.

If no term Left in the expression, pop the stack and this is the result

Continued...

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

Reversed Expression: 5 2 3 4 + * –



Continued...

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
END;
```

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

Time & Space Complexity of Prefix Expression

- **Time Complexity: Θ(N)**

  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. 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.
- 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

Continued..

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

Continued..

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 for precedence

```
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 & Space Complexity of Infix to Postfix

- **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).

Continue...

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

Continue,..

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 Infix to Postfix Expression

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 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

Algorithm Infix 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 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 to check Precedence

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
ELSE
        RETURN FALSE
END;
```

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 infix to prefix conversion

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)
                Add x to Prefix Expression
RETURN Reverse (Prefix Expression)
END;
```

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

Reverse and return the Postfix Expression

- **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.

Implementing multiple Stacks in a single Array

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;

Implementing multiple Stacks in a single Array

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;

Implementing multiple Stacks in a single Array

ALGORITHM MSPopB(A[ ], TopA, TopB)

Input: Array A[ ], Top indexes of both the stacks

Output: Deleted element

BEGIN:

```
IF TopB == N THEN
        WRITE("Stack B underflows")
        EXIT(1)
x=A[TopB]
TopB=TopB + 1
RETURN x
```

END;

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

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

Implementing multiple Stacks in a single Array

- 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.

Implementing multiple Stacks in a single Array

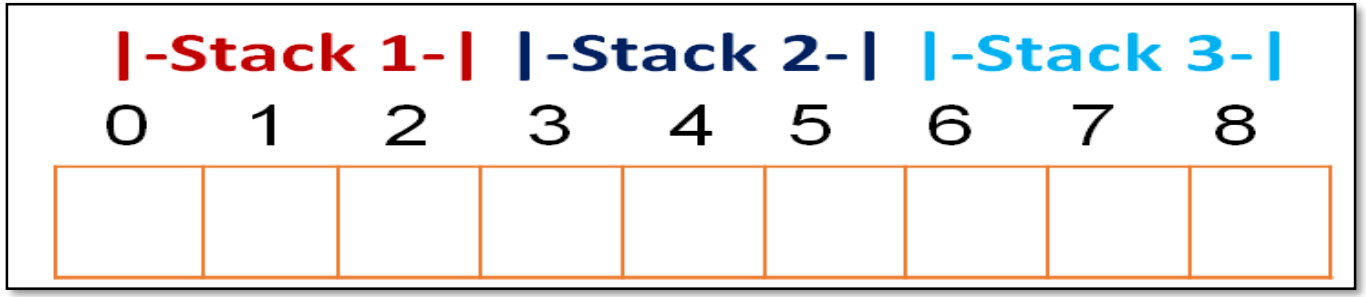
Method 1: Logical division of Array

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

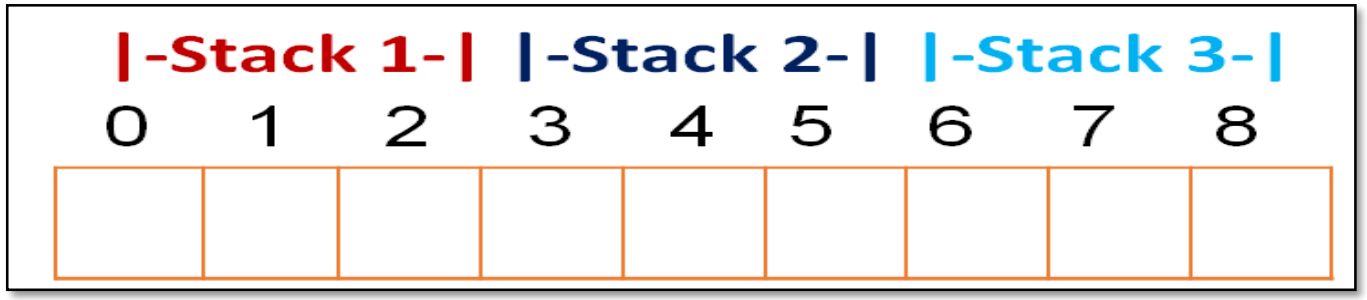
Number of stack (M) =3

Size of each stack = N/M =9/3=3



Implementing multiple Stacks in a single Array

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.

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

Implementing multiple Stacks in a single Array

Generalized push operations in the i^{th} Stack:

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

Implementing multiple Stacks in a single Array

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
```

END;

If no more insertion possible.

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

Implementing multiple Stacks in a single Array

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.

```
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.

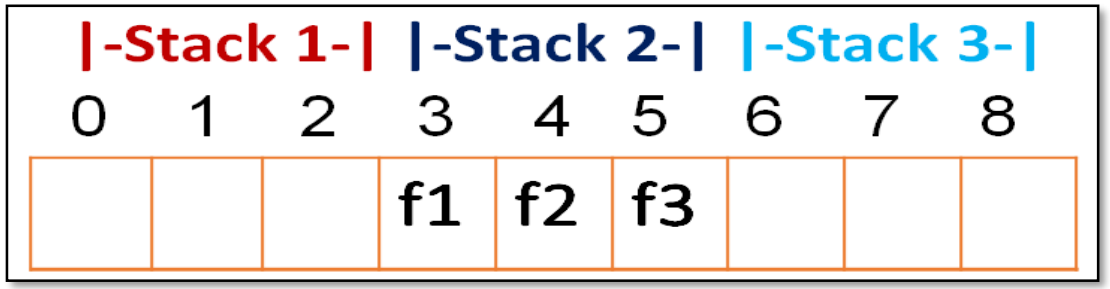
Implementing multiple Stacks in a single Array

Advantage of Multiple Stack single array (MSSA)

1. Better memory utilization as compared to using single stack.
2. Multiple Recursive programs can be designed considering partitioned array with defined boundaries.

Drawback of Multiple Stack single Array (MSSA)

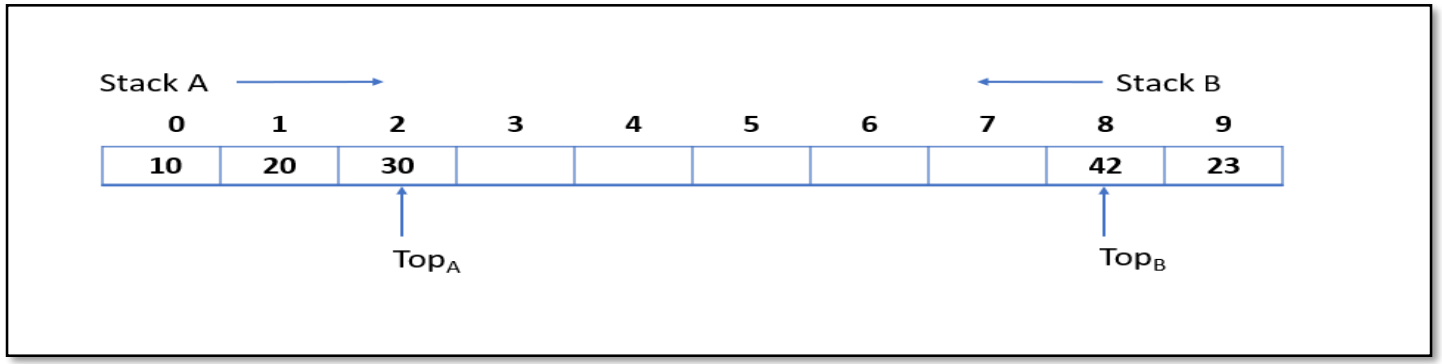
Consider the given situation:



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.

Implementing multiple Stacks in a single Array

Method 2: Sharing of Array memory by two Stacks



NOTE: 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.

- 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 $Top_A = -1$ & $Top_B = N$.

Implementing multiple Stacks in a single Array

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;

Implementing multiple Stacks in a single Array

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;

Implementing multiple Stacks in a single Array

ALGORITHM MSPopB(A[ ], TopA, TopB)

Input: Array A[ ], Top indexes of both the stacks

Output: Deleted element

BEGIN:

```
IF TopB == N THEN
        WRITE("Stack B underflows")
        EXIT(1)
x=A[TopB]
TopB=TopB + 1
RETURN x
```

END;

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

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