Stack

Dr. Anish Soni

Stacks

INTRODUCTION

- **The linear lists and linear arrays are used to insert and delete elements at any place.** There are certain frequent situations when one wants to restr[illegible] insertion and deletion so that they can take place only at the beginnin[illegible] end of the list, not in the middle.
- **Two of the data structures** that are useful in such situations are ***stacks*** and ***queues***.
- **A stack is a linear structure in which items may be added or removed only at one end. It is called last-in first-out (LIFO).**
- **A queue is a linear list in which items may be added only at one end and items may be removed only at the other end. It is called first-in first-out (FIFO) list.**

STACKS

- A stack is a **list of elements in which an element may be inserted or deleted only a one end**, called *TOP* of the stack.
- The **elements are removed in reverse order** of that in which they were inserted into the stack.

- A **stack** is a linear data structure that follows the **Last-In, First-Out (LIFO)** principle. This means the last element added to the stack is the first one to be removed—much like a physical stack of plates.

Core Operations

- **Push**: Adds a new element to the top of the stack. If the stack is full (in fixed-size implementations), this results in a **stack overflow** condition.
- **Pop**: Removes and returns the top element from the stack. If the stack is empty, this results in a **stack underflow** condition.
- **Peek (or Top)**: Returns the value of the top element without removing it.
- **IsEmpty**: Checks if the stack has any elements.
- **IsFull**: Checks if the stack has reached its maximum capacity (only for fixed-size implementations).

Key Characteristics

- **Single Access Point**: All insertions and deletions happen at one end, called the **Top**.
- **Dynamic or Static**: It can be implemented using **arrays** (fixed size) or **linked lists** (dynamic size).

Implementation of Stacks

- **Array-Based Implementation**: Uses a fixed-size array, which is simple and memory-efficient as elements are stored contiguously. The drawback is the fixed size, which can lead to overflows if the capacity is exceeded.
- **Linked List-Based Implementation**: Uses a linked list, allowing the stack to grow and shrink dynamically as needed. This avoids the fixed-size limitation, but each element requires extra memory for pointers to the next node.

1. Array Representation (Static Implementation)

- The array implementation uses a contiguous block of memory.
- **Structure:**
  - An array, e.g., stack[MAX_SIZE], to store the elements.
  - A variable top, which is an integer index pointing to the most recently added element (or the next available space, depending on implementation). It is often initialized to -1 for an empty stack.
- **Memory Usage:**
  - Memory is allocated statically at compile time, meaning the maximum size is fixed and cannot be changed during program execution.
  - This can lead to memory waste if the allocated space is not fully utilized, or a **stack overflow** error if the stack exceeds its maximum capacity.
- **Operations:**
  - push(): Increments the top index and adds the new element to that position.
  - pop(): Accesses the element at the top index and then decrements the top index

2. Linked List Representation (Dynamic Implementation)

- The linked list implementation uses dynamically allocated memory.
- **Structure:**
  - Each element is stored in a **node** containing the data and a pointer to the next node in the sequence.
  - A main pointer top (often called head) points to the first node, which is the current top of the stack. It is initialized to NULL for an empty stack. The last node in the list points to NULL.
- **Memory Usage:**
  - Memory is allocated dynamically from the **heap** as needed, so the stack can grow or shrink in size during runtime.
  - Memory is used efficiently as space is only allocated for the actual elements present. The only overhead is the extra memory required for the pointers in each node.
- **Operations:**
  - push(): Creates a new node, points the new node's "next" pointer to the current top, and updates top to the new node.
  - pop(): Temporarily stores the current top node, moves the top pointer to the next node in the list, and then deallocates the memory of the original top node.

Summary of Key Differences

Feature	Array-Based Stack	Linked List-Based Stack
Memory Allocation	Static (fixed size)	Dynamic (grows as needed)
Memory Usage	Can waste memory if oversized	Efficient; uses only required memory plus pointer overhead
Insertion/Deletion Speed	Faster (direct index access)	Slightly slower (requires pointer manipulation)
Overflow Risk	Yes, if size exceeds the fixed limit	No, unless system memory is exhausted

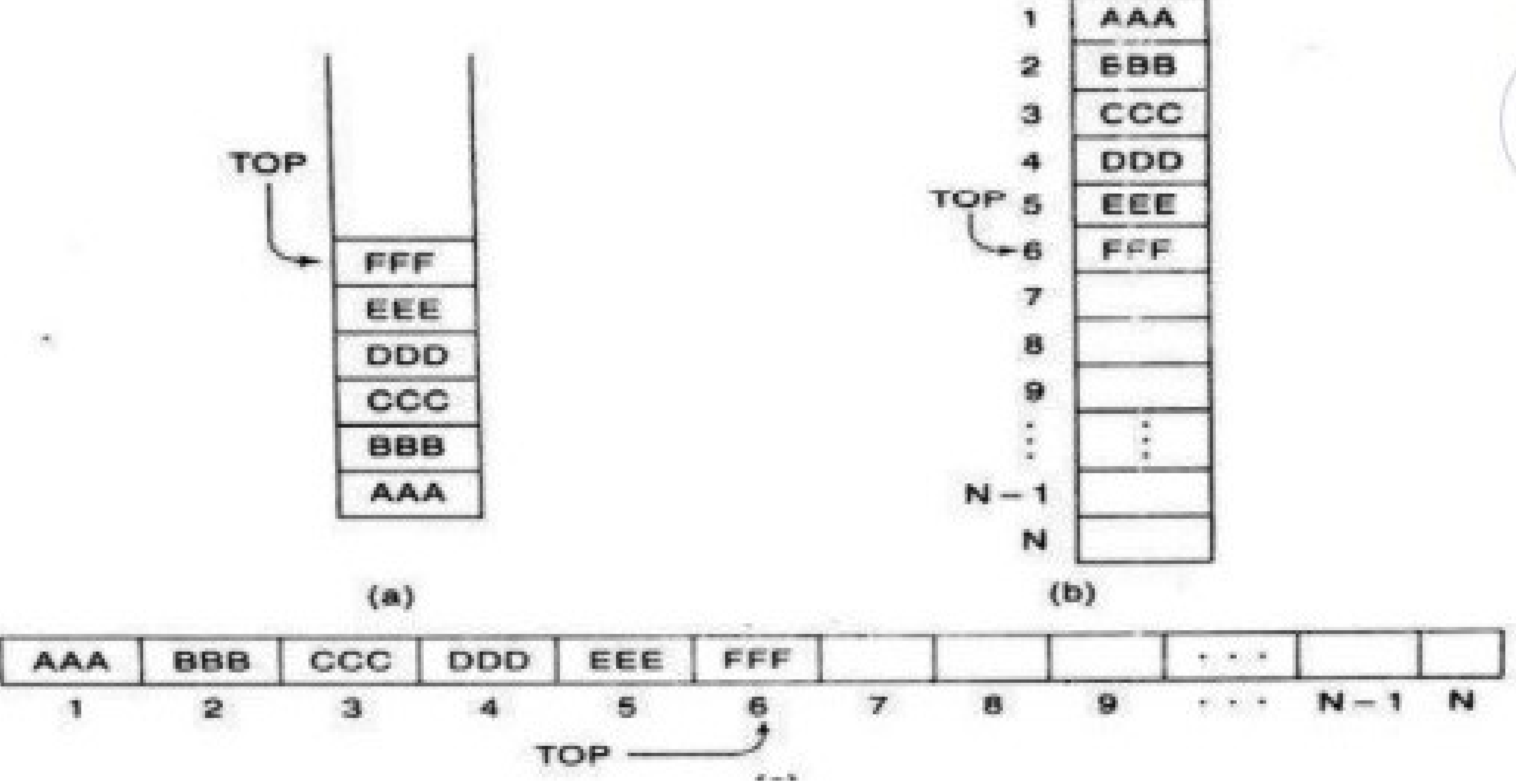
Real-World Applications

Stacks are used in many areas of computing to manage sequential tasks:

- **Undo/Redo Functionality**: Software applications like text editors use a stack to store a history of user actions, allowing the last action to be easily reversed.
- **Browser History**: The "back" button in web browsers uses a stack to store the URLs of visited pages, so the last page visited is the first one returned to.
- **Expression Evaluation**: Compilers and calculators use stacks to convert and evaluate mathematical expressions (e.g., infix to postfix notation) by managing operator precedence.
- **Syntax Checking**: Compilers use stacks to ensure that all opening parentheses, brackets, and braces have corresponding closing ones in the correct order.

Example:

- Suppose the following 6 elements are pushed, in order, onto an empty stack:
- AAA, BBB, CCC, DDD, EEE, FFF



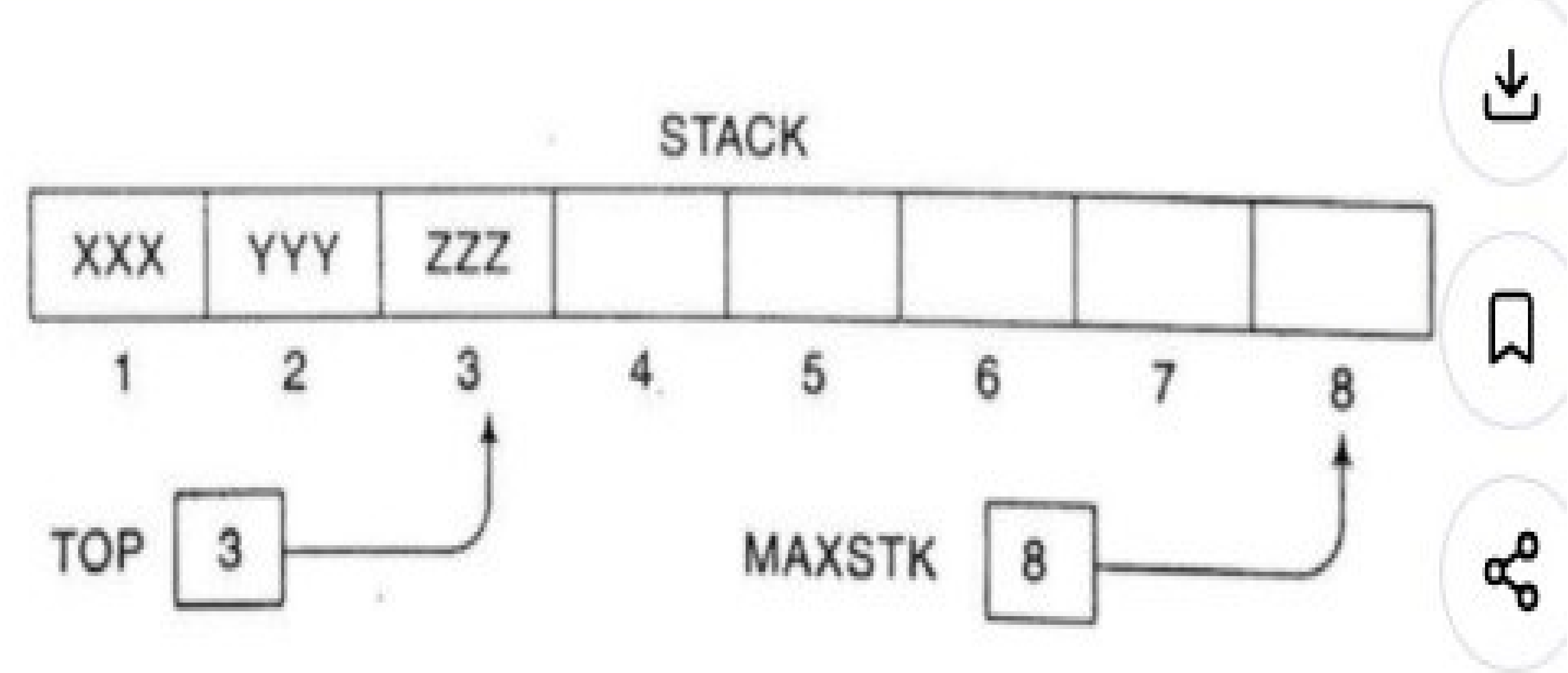
ARRAY REPRESENTATION OF STACKS

- Stacks are **represented in the computer by a linear array.**
- In the following algorithms/procedures of pushing and popping an item from the stacks,

1. **a linear array STACK,**
2. **a variable TOP** which **contains the location of the top element** of the stack;
3. and **a variable MAXSTAK** which **gives the maximum number of elements** that can be held by the stack.

OVERFLOW AND UNDERFLOW

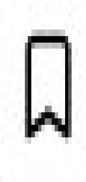
1. The **condition TOP=0 or TOP=NULL** will **indicate** th **stack is empty** and **we say UNDERFLOW.**
2. The **condition TOP=MAXSTK** will **indicate** the **stack is full and we say OVERFLOW.**



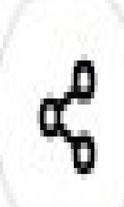
The **operation** of **adding an element into the stack** is called ***pushing*** and
the **operation of removing an element from the stack** is called ***popping***.

Algorithm : PUSH(STACK, TOP, MAXSTAK, ITEM):

This procedure pushes an ITEM onto a stack.



1. [STACK already filled?]



If TOP=MAXSTAK, then: Print: OVERFLOW, and Return

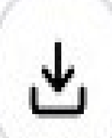
2. Set TOP:=TOP+1 [Increase TOP by 1]

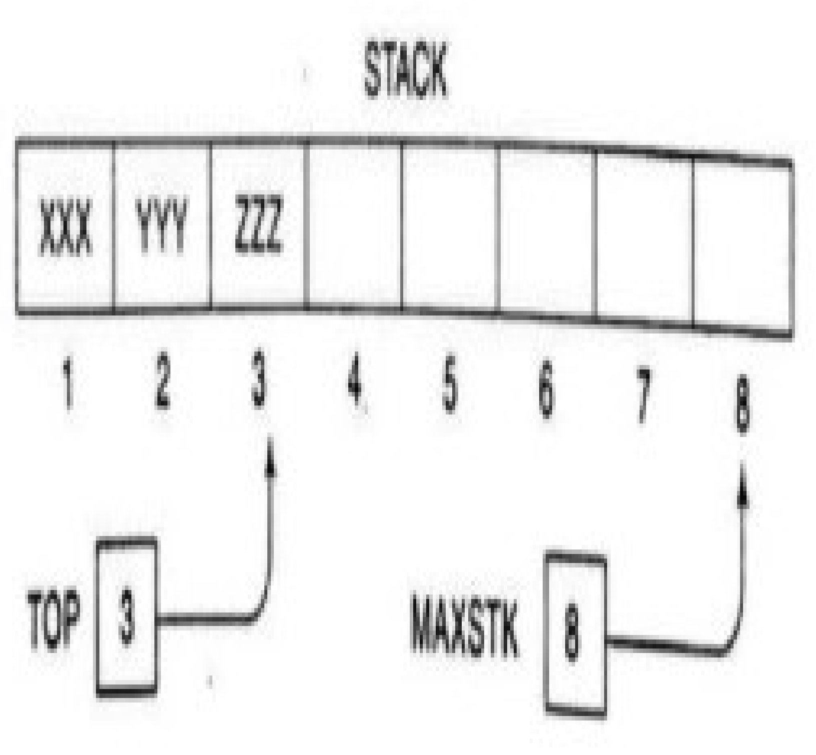
3. Set STACK [TOP]:=ITEM [Insert ITEM in new TOP position]

4. Return

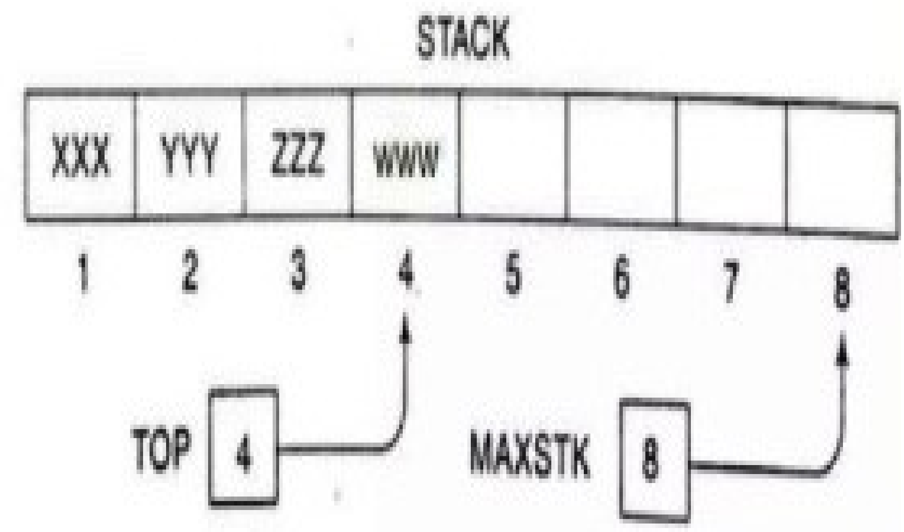
Example:

To simulate the operation **PUSH(STACK, WWW)**





1. Since **TOP=3, control is transferred to Step 2.**
2. **TOP=3+1=4**
3. **STACK[TOP]=STACK[4]=WWW**
4. Return



Algorithm : POP(STACK, TOP, ITEM)

This procedure **deletes the top element of STACK** and **ass**

it to the variable ITEM.

1. [STACK has an item to be removed? Check for empty stack]

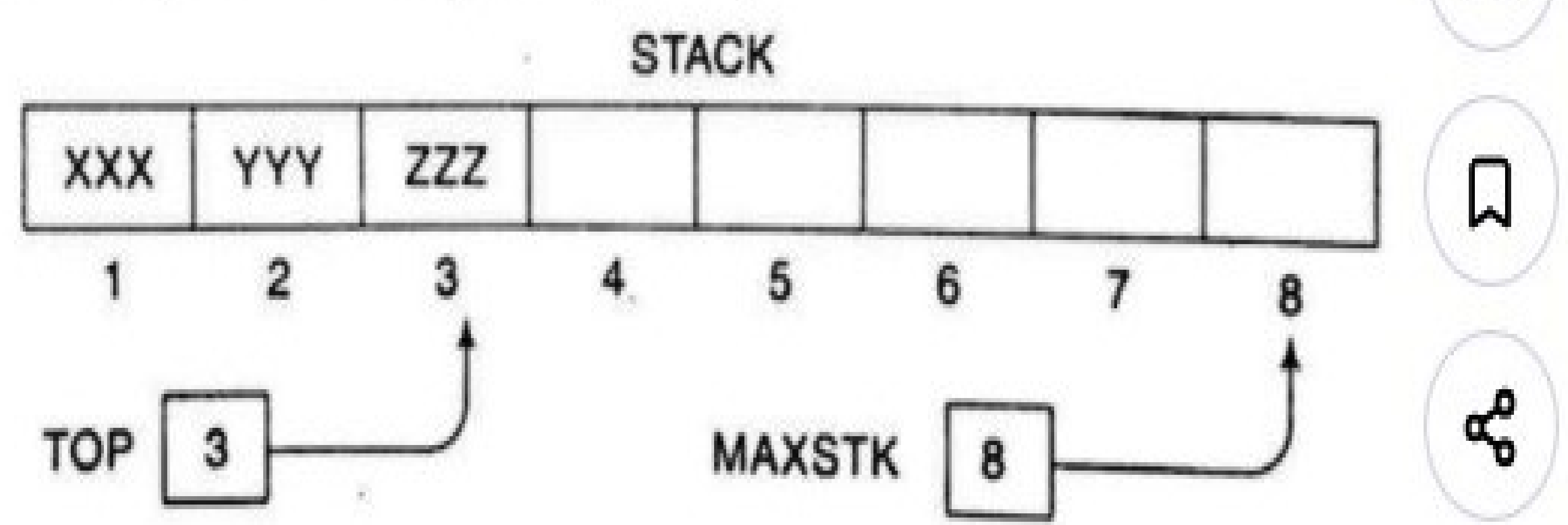
If TOP=-1, then: Print: UNDERFLOW, and Return

2. Set ITEM:=STACK[TOP] [Assigns TOP element to ITEM]

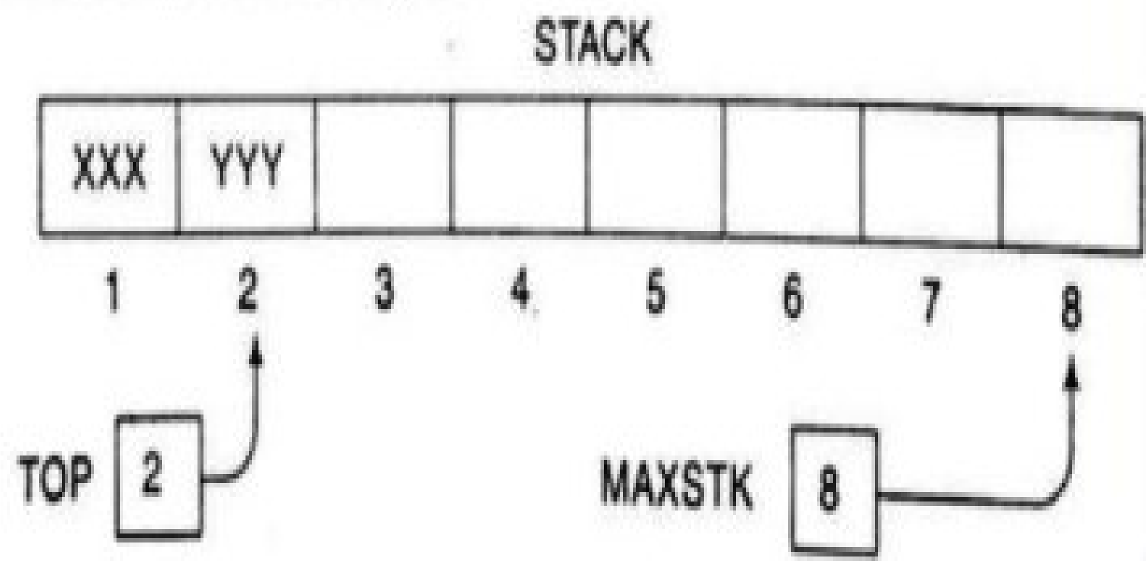
3. Set TOP=TOP-1 [Decrease TOP by 1.]

4. Return

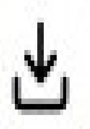
To simulate the **operation POP(STACK, ITEM)**



1. Since **TOP=3**, control is transferred to Step 2.
2. **ITEM=ZZZ**
3. **TOP=3-1=2**
4. Return

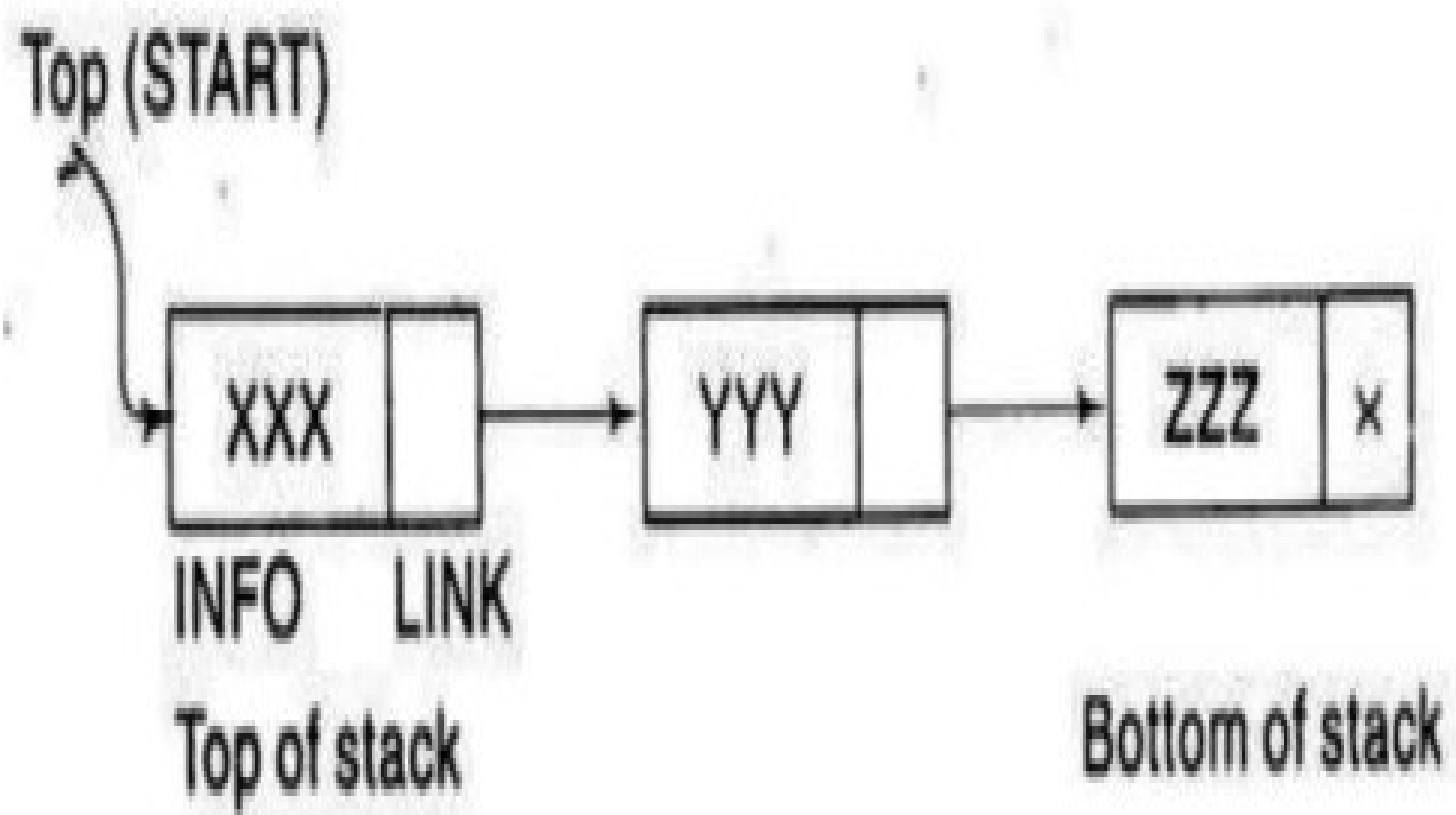


LINKED REPRESENTATION OF STACKS



- The **linked representation of stack** is **implemented using si**
linked list.

1. The **INFO fields** of the nodes **hold the elements** of the stack an
2. The **LINK fields** hold the **pointers to the neighboring element**
the stack.
3. The **START pointer** of the linked list behaves as **TOP pointer variable** of the stack and
4. The **NULL pointer** in the **last node** signals the **end of the stack.**


Top (START)
XXX
INFO
LINK
YYY
ZZZ
x
Top of stack
Bottom of stack

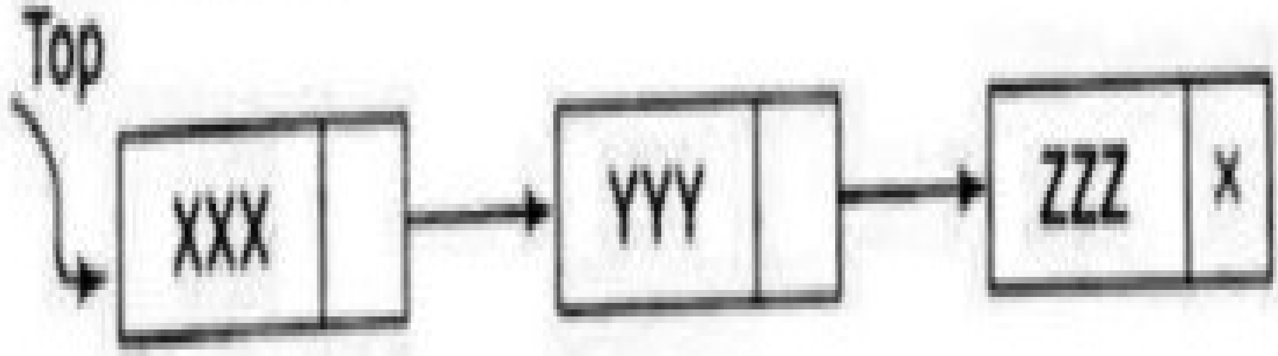
- A **PUSH operation** is accomplished by **inserting a node** into the **start of the STACK** and a **POP operation** is undertaken by **deleting the node pointed to by the START pointer**.
- The **array representation of stack** maintains a **variable MAXSTK which gives the maximum number of elements** that can be held by the stack.
- If the **number of elements in the stack** becomes **equal to the value of variable MAXSTK, it is the condition of OVERFLOW.**
- The linked representation of stacks is free from this requirement.

- **There is no limitation on the capacity of the linked stack hence it can support as many PUSH operations as free storage list (the AVAIL list) can support.**

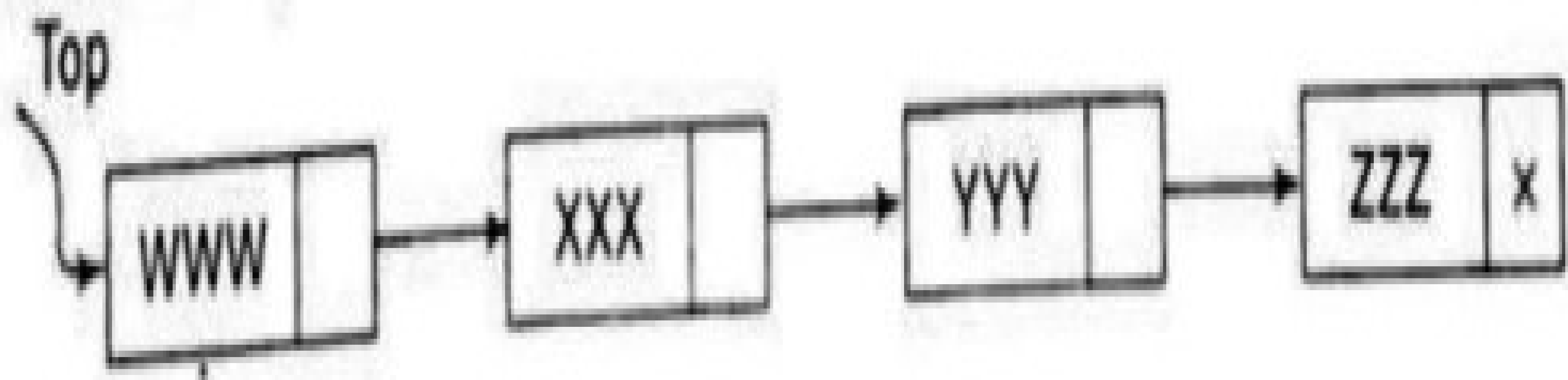
1. However, **the condition TOP=NULL** may be **retained in the POP procedure** to **prevent deletion from an empty stack** and
2. **AVAIL=NULL to check for the available space in the free storage list.**

Push 'WWW' into STACK

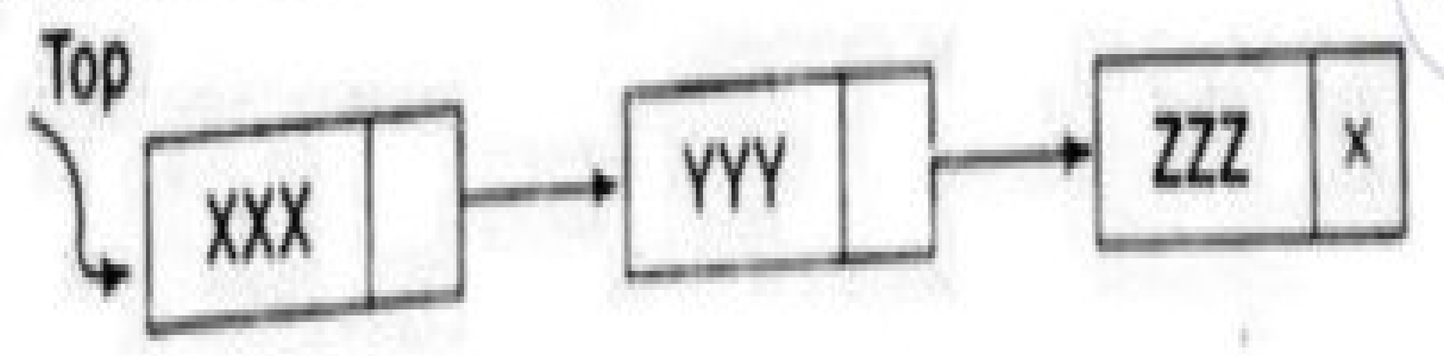
STACK before Push operation:

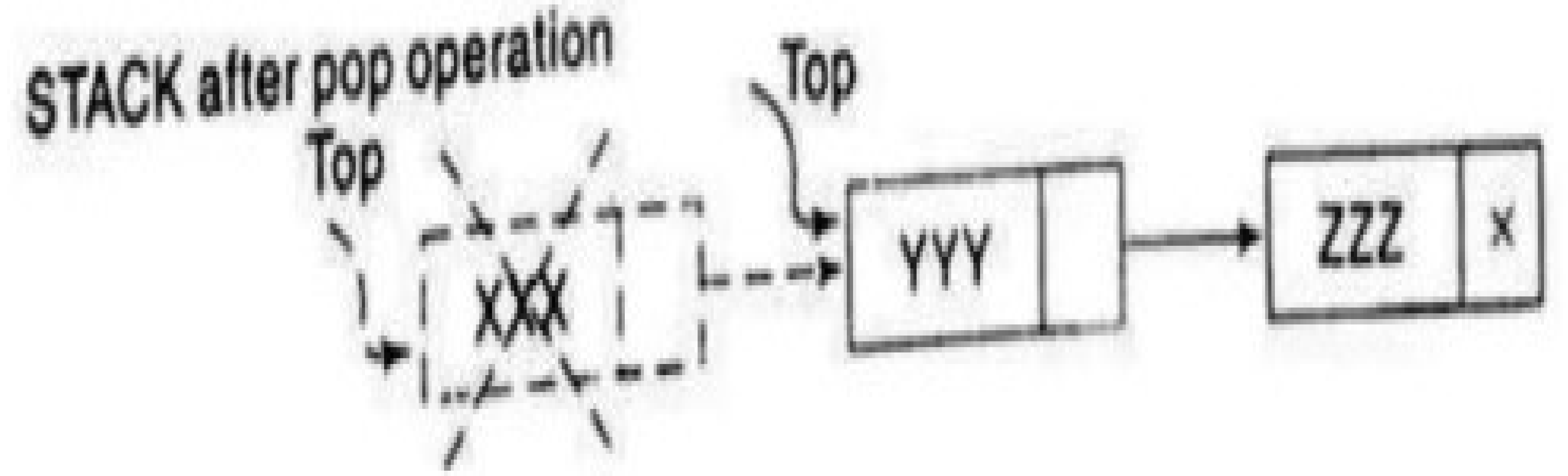


STACK after Push operation



Pop from STACK
STACK before pop operation:


Top
XXX
YYY
ZZZ
x


STACK after pop operation
Top
XXX
Top
YYY
ZZZ
x

Algorithm : PUSH_LINKSTACK(INFO, LINK, TOP, AVAIL, ITEM)

This procedure **pushes an ITEM onto a stack**.

1 IF AVAIL=NULL, then Write: OVERFLOW and Exit [Available **Space?**]

2. [**Remove first node** from **AVAIL**

Set NEW:=AVAIL and AVAIL:=LINK[AVAIL]

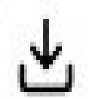
3. Set INFO[NEW]:=ITEM [**Copies the ITEM** into new node]

4. Set LINK[NEW]:=TOP [**New node points to the original top node** in the stack]

5. Set TOP:=NEW [**Reset TOP** to **point to the new node** at the top of the stack]

6. Exit

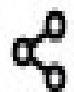
Algorithm : POP_LINKSTACK(INFO, LINK, TOP, AVAIL, ITEM)



This procedure **deletes the top element of STACK and assigns it to the variable ITEM.**

1. [STACK has an item to be removed? **Check for empty stack**]

If TOP=NULL , then: Write: UNDERFLOW, and Exit



2. Set ITEM:=INFO[TOP]. [Copies the top element of stack into ITEM]

3. Set TEMP:=TOP and TOP:=LINK[TOP]

[Remember the old value of TOP in TEMP and reset TOP to point to the next element]

4. [Return deleted node to the AVAIL list]

Set LINK [TEMP]:=AVAIL and AVAIL:=TEMP

5. Exit

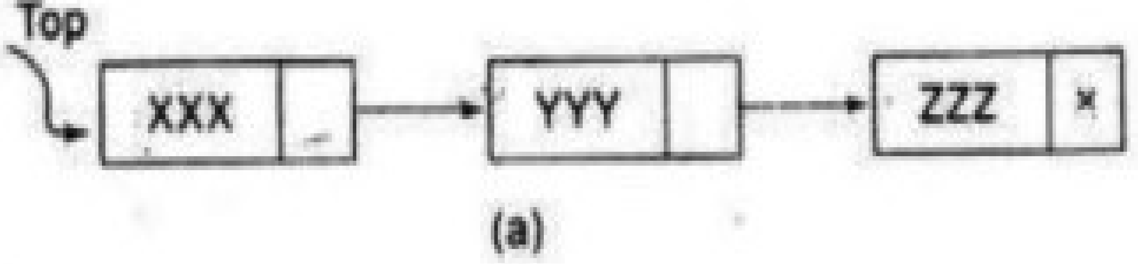
Example:

Consider the linked list given in

fig (a) and perform the following operations:

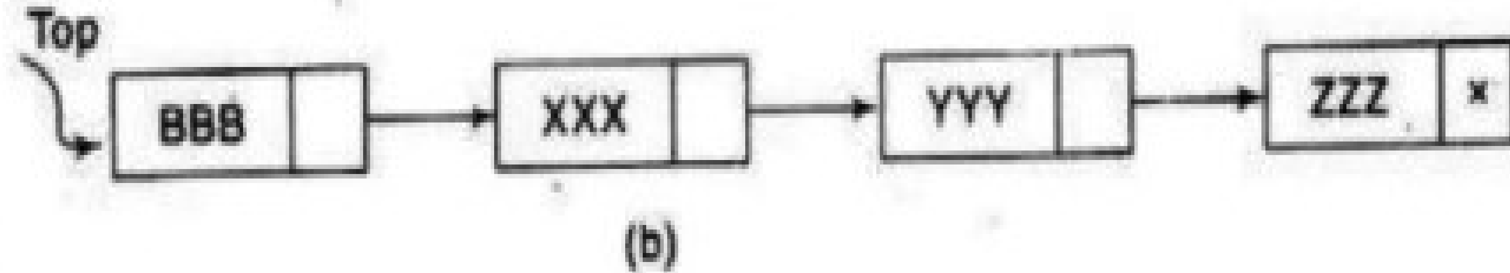
I) **Push BBB ii) Pop iii) Pop iv) Push MMM**

Original linked stack:

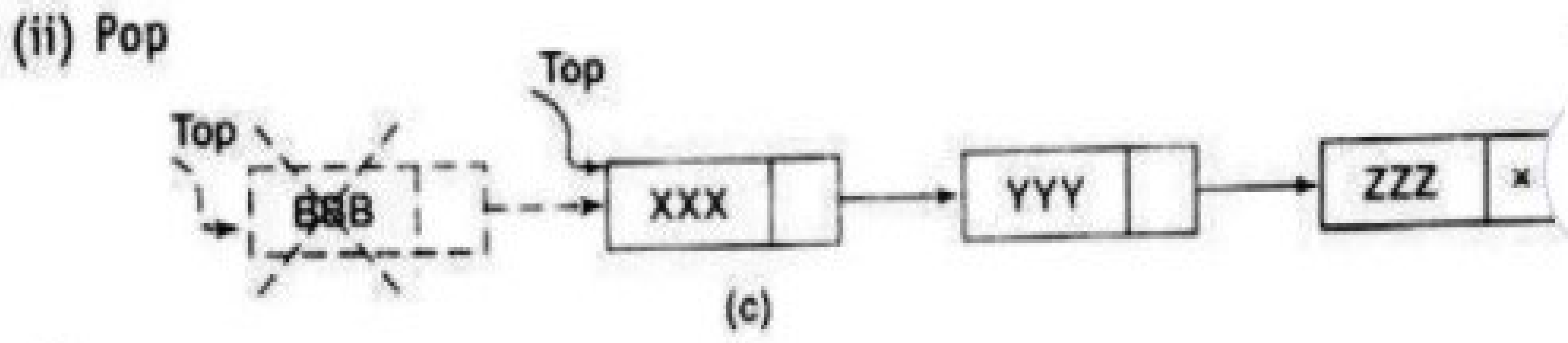


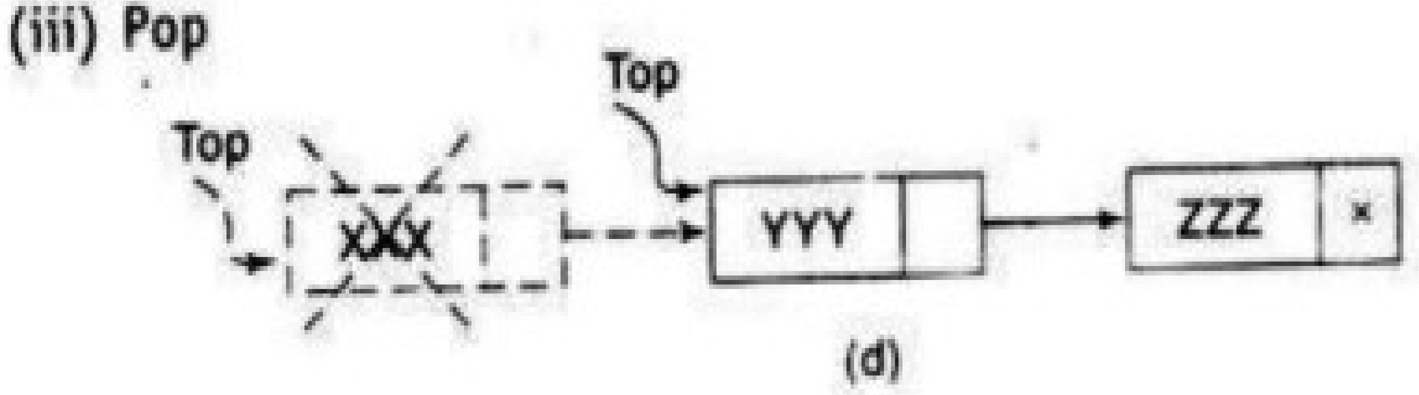
(a)

(i) Push BBB

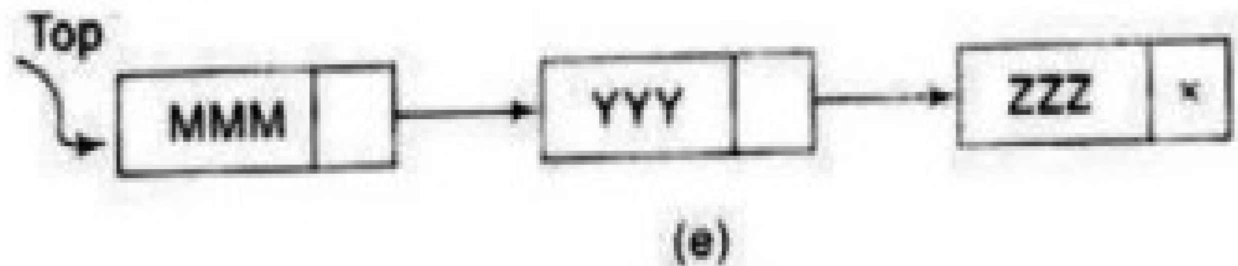


(b)





(iv) Push MMM



(e)

ARITHMETIC EXPRESSIONS

- **INFIX , POSTFIX AND PREFIX NOTATIONS:**
- Let Q be an **arithmetic expression involving constants and operations.**
- Besides operands and operators, Q may also contain left and right parentheses.
- The operators in Q can consist of exponentiations (�) multiplications (*), divisions (/) additions (+) and subtractions (-). They have different levels of precedence.
- **Highest : Exponentiation (�)**
- **Next Highest : Multiplication (*) and division (/)**
- **Lowest : Addition (+) and subtraction (-)**

Operator Precedence Rules

Highest: Parentheses ()

High: Exponentiation ^

Medium: Multiplication *, Division /

Low: Addition +, Subtraction -

Highest : Exponentiation (↑)

Next Highest : Multiplication (*) and division (/)

Lowest : Addition (+) and subtraction (-)

Example- 1:

The evaluation of expression $2 \uparrow 3 + 5 * 2 \uparrow 2 - 12 / 6$

$= 8 + 5 * 4 - 12 / 6$

$= 8 + 20 - 2$

$= 28 - 2$

$= 26.$

1. Generally, **expressions can be written in which the operator is written between the operands.** This is called ***infix notation***.

   For example: **A + B**   **C - D**

2. *Polish notation*, named after the Polish mathematician […] Lukasiewicz, refers **to the notation in which the operator symbol is placed before its two operands.** This notation frequently called ***p[…]*** ***notation.***

   For example, **+ A B**   **- C D**

3. Reverse Polish notation refers to the analogous notation in which the **operator symbol is placed after two operands.** This notation frequently called ***postfix (or suffix) notation***

   For example: **A B +**   **C D -**

- **Computers "prefer" postfix notation** in which the operator is written to the right of two operands.

- The computer usually **evaluates an arithmetic expression** written **in infix notation** in **two steps.**

1. First, **it converts the expression to postfix notation and**
2. **Then it evaluates the postfix expression.**

- In each step **stack is used as main tool.**

Transforming Infix Expression into Postfix Expression:

- The following algorithm **transforms the infix expression Q into its equivalent postfix expression P.**
- It uses a **stack to temporary hold the operators and left parenthesis.**
- The postfix expression will be **constructed from left to right using operands from Q and operators popped from STACK.**

Algorithm 6.6: Infix_to_PostFix(Q, P)

Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

1. Push "("onto STACK, and add ")" to the end of Q.
2. Scan Q from left to right and repeat Steps 3 to 6 for each element of Q until the STACK is empty:
3. If an operand is encountered, add it to P.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator ⊗ is encountered, then:
    a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same or higher precedence/priority than ⊗
    b) Add ⊗ to STACK.

   [End of If structure]

6. If a right parenthesis is encountered, then:
    a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.
    b) Remove the left parenthesis. [Do not add the left parenthesis to P]

   [End of If structure]

   [End of Step 2 loop]

7. Exit.

ALGORITHM

Scan the Infix expression left to right

- If the character x is an operand
  - Output the character into the Postfix Expression
- If the *character x* is a left or right parenthesis
  - If the character is "("
    - Push it into the stack
  - if the *character* is ")"
    - Repeatedly pop and output all the operators/characters until "(" is popped from the stack.
- If the character x is a is a regular operator
    - **Step 1**: Check the character y currently at the top of the stack.
    - **Step 2**: If Stack is empty or y='(' or y is an operator of ***lower precedence*** than x, then push x into stack.
    - **Step 3**: If y is an operator of ***higher or equal*** precedence than x, then pop and output y and push x into the stack.

When all characters in infix expression are processed repeatedly pop the character(s) from the stack and output them until the stack is empty.

SOUMEN SANTRA

Example: Convert Q: **A + (B * C – (D / E ↑ F) * G) * H** into postfix form showing stack status . Now add ")" at the end of expression A + (B * C – (D / E ↑ F) * G) * H) and also Push a "(" on Stack. The elements of P are labelled from left to right for ease of reference as follows:

A	+	(	B	*	C	-	(	D	/	E	↑	F	)	*	G	)	*	H	)
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)	(19)	(20)

A + (B * C - (D / E ↑ F) * G) * H)

(1) (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13) (14) (15) (16) (17) (18) (19) (20)

	Symbol Scanned	STACK	Expression Y
(1)	A	(	A
(2)	+	(+	A
(3)	(	(+(	A
(4)	B	(+(	AB
(5)	*	(+(*	AB
(6)	C	(+(*	ABC
(7)	-	(+(-	ABC*
(8)	(	(+(-(	ABC*
(9)	D	(+(-(	ABC*D
(10)	/	(+(-(/	ABC*D
(11)	E	(+(-(/	ABC*DE
(12)	↑	(+(-(/↑	ABC*DE
(13)	F	(+(-(/↑	ABC*DEF
(14)	)	(+(-	ABC*DEF↑/
(15)	*	(+(-*	ABC*DEF↑/
(16)	G	(+(-*	ABC*DEF↑/G
(17)	)	(+	ABC*DEF↑/G*-
(18)	*	(+*	ABC*DEF↑/G*-
(19)	H	(+*	ABC*DEF↑/G*-H
(20)	)		ABC*DEF↑/G*-H*+

Infix Expression: **A+(B*C-(D/E^F)*G)*H**, where **^** is an exponential operator.

Input String	Output Stack	Operator Stack
A+(B*C-(D/E^F)*G)*H		
A+(B*C-(D/E^F)*G)*H	A	
A+(B*C-(D/E^F)*G)*H	A	+
A+(B*C-(D/E^F)*G)*H	A	+(
A+(B*C-(D/E^F)*G)*H	AB	+(
A+(B*C-(D/E^F)*G)*H	AB	+(*
A+(B*C-(D/E^F)*G)*H	ABC	+(*
A+(B*C-(D/E^F)*G)*H	ABC*	+(-
A+(B*C-(D/E^F)*G)*H	ABC*	+(-(
A+(B*C-(D/E^F)*G)*H	ABC*D	+(-(
A+(B*C-(D/E^F)*G)*H	ABC*D	+(-(/
A+(B*C-(D/E^F)*G)*H	ABC*DE	+(-(/
A+(B*C-(D/E^F)*G)*H	ABC*DE	+(-(/^
A+(B*C-(D/E^F)*G)*H	ABC*DEF	+(-(/^
A+(B*C-(D/E^F)*G)*H	ABC*DEF^/	+(-
A+(B*C-(D/E^F)*G)*H	ABC*DEF^/	+(-*
A+(B*C-(D/E^F)*G)*H	ABC*DEF^/G	+(-*
A+(B*C-(D/E^F)*G)*H	ABC*DEF^/G*-	+
A+(B*C-(D/E^F)*G)*H	ABC*DEF^/G*-	+*
A+(B*C-(D/E^F)*G)*H	ABC*DEF^/G*-H	+*
A+(B*C-(D/E^F)*G)*H	ABC*DEF^/G*-H*+	

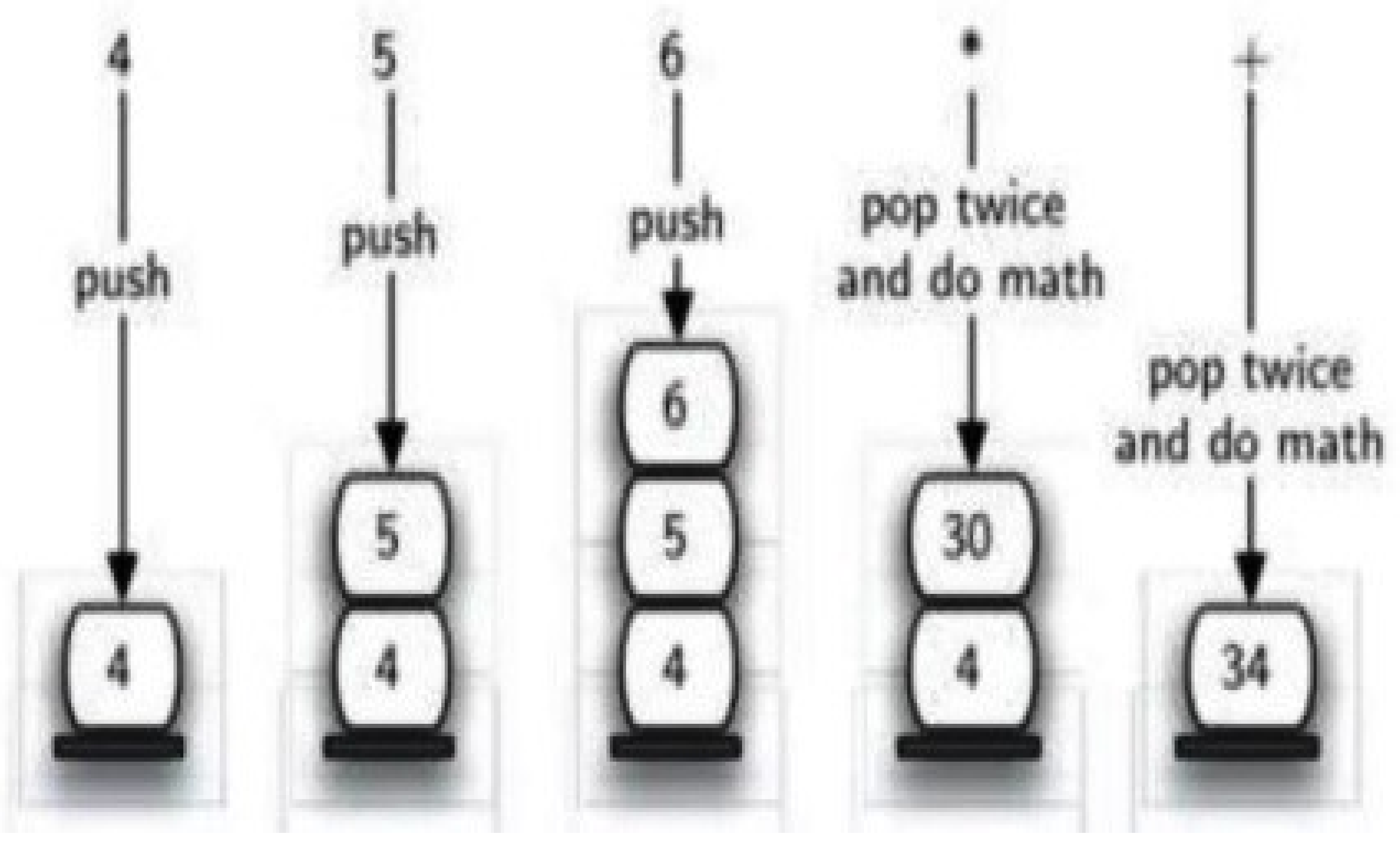
Evaluation of a Postfix Expression:

- Suppose **P is an arithmetic expression written in** **postfix notation**.
- The following algorithm **uses STACK** to evaluate P.

Algorithm : This algorithm finds VALUE of an arithmetic expression P which is writter postfix notation.

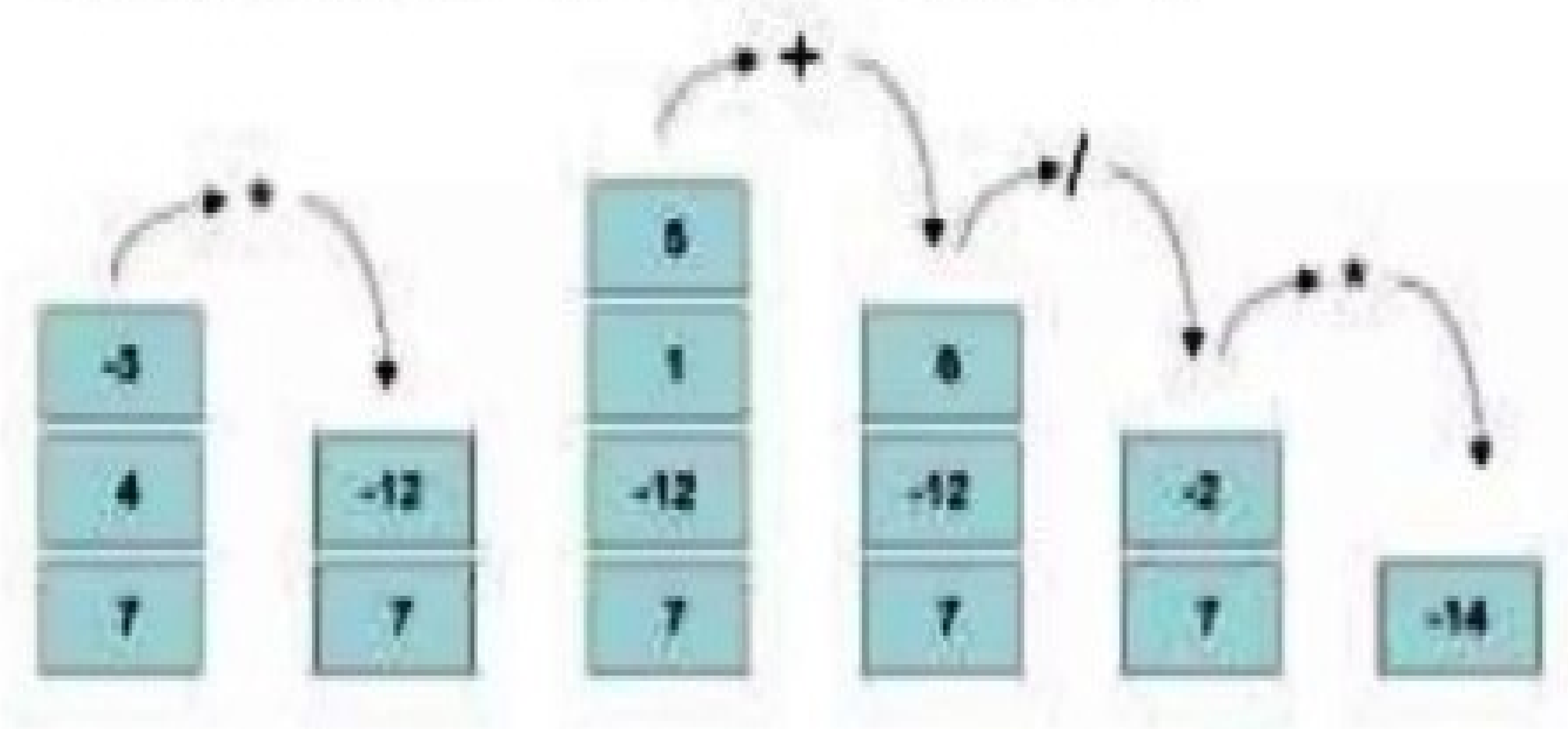
1. Add a right parenthesis ")" at the end of P [This acts as sentinel]
2. Scan P from left to right and repeat steps 3 and 4 for each element of P until the sentinel ")" encountered.
3. If an operand is encountered, put it into the STACK.
4. If an operator ⊗ is encountered, then:
    a. Remove the two elements of the stack, where A is the top element and B is the next-to-top element
    b. Evaluate B ⊗ A
    c. Place the result of (b) back on STACK

   [End of If structure]
5. Set VALUE equal to the top element on STACK.
6. Exit


Left to Right Evaluation
4
push
4
5
push
5
4
6
push
6
5
4
*
pop twice
and do math
30
4
+
pop twice
and do math
34

Evaluating Postfix Expressions

- Expression = 7 4 -3 * 1 5 + / *



Example:

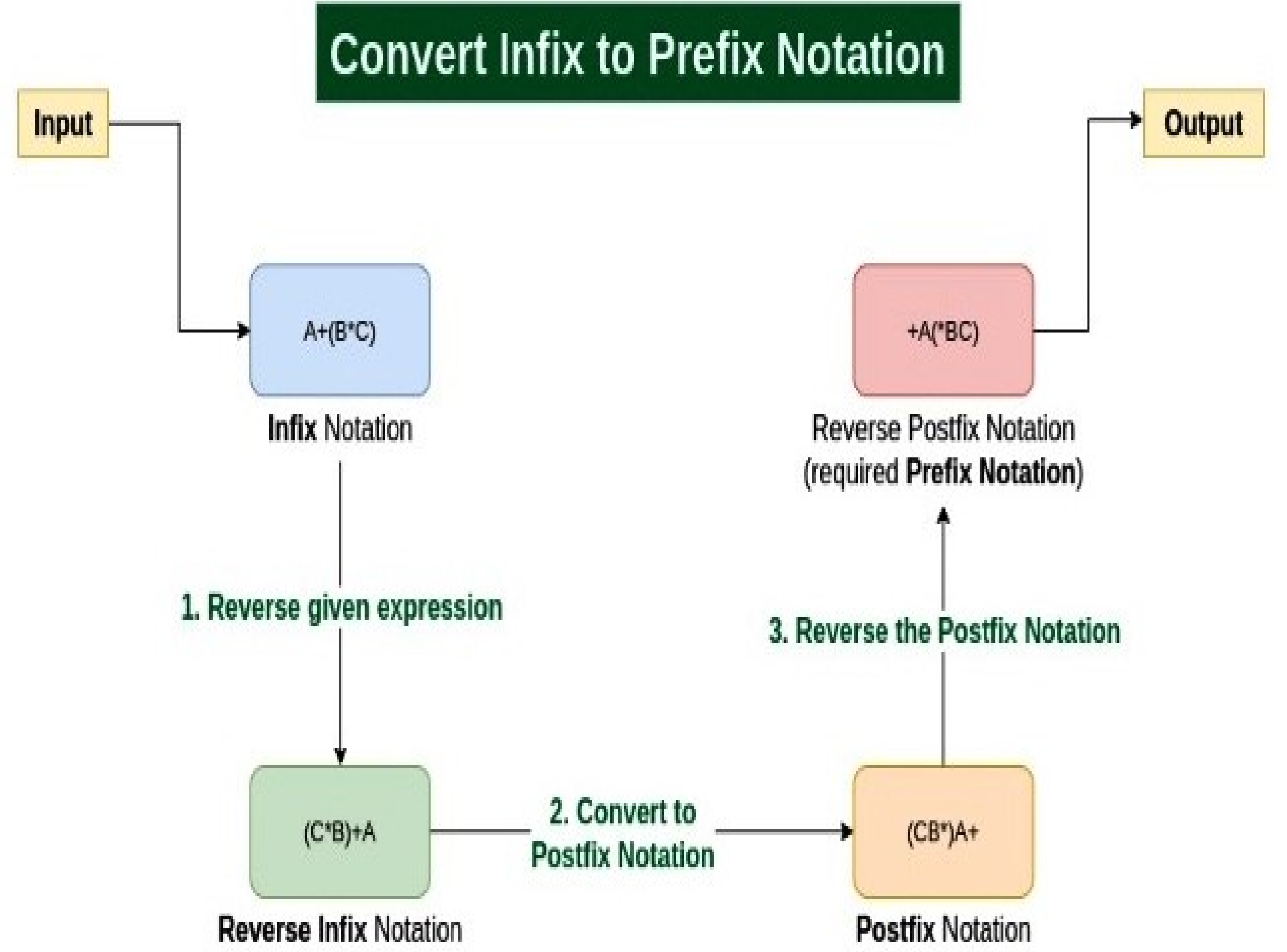
- Consider **P: 5, 6, 2, +, *, 12, 4, /, –**

whose equivalent infix expression is

- Q: 5 * (6 + 2) – 12 / 4
- The elements of P are labelled from left to right for ease of reference as follows:
- P: 5, 6, 2, +, *, 12, 4, /, –,)

(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)

Symbol Scanned	STACK
(1) 5	5
(2) 6	5, 6
(3) 2	5, 6, 2
(4) +	5, 8
(5) *	40
(6) 12	40, 12
(7) 4	40, 12, 4
(8) /	40, 3
(9) -	37
(10))	


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

Algorithm Steps

The process involves four main steps using a stack data structure:

1. **Reverse the Infix Expression**: The original infix expression is read from right to left, and the order of characters is reversed. During this reversal, all opening parentheses (are swapped with closing parentheses), and vice versa.
2. **Convert the Reversed Expression to Postfix**: Apply the standard infix-to-postfix conversion algorithm on the *modified* (reversed and swapped parentheses) expression.
   i. **Operand**: If the scanned character is an operand (e.g., a letter or number), append it directly to a temporary result string.
   ii. **Opening Parenthesis**: If the character is a), push it onto the operator stack (it acts as an opening parenthesis in the reversed context).
   iii. **Closing Parenthesis**: If the character is a (, pop operators from the stack and append them to the result until the matching) is found. Discard both parentheses.
   iv. **Operator**: If the character is an operator, pop operators from the stack and append them to the result if they have higher or equal precedence than the current operator. Push the current operator onto the stack.
   v. **End of Expression**: After scanning all characters, pop any remaining operators from the stack and append them to the result string.
3. **Reverse the Postfix Result**: The temporary result string (which is a "reversed postfix" expression) is then reversed again.
4. **Final Prefix Expression**: The final reversed result is the required prefix expression.

Example 1: Basic Arithmetic `A+B*C`

1. **Reverse:** $C * B + A$

2. **Postfix Conversion of** $C * B + A$**:**

   - Scan C: Output C, Stack empty.
   - Scan $*$: Push $*$ onto stack. Stack: [*]
   - Scan B: Output CB, Stack: [*]
   - Scan $+$: Pop * (higher precedence), Push $+$. Output $CB*$, Stack: [+]
   - Scan A: Output $CB * A$, Stack: [+]
   - Pop remaining stack: $+$. Output $CB * A+$

3. **Reverse Postfix:** $+A * BC$ (Result: `+A*BC`) CKT College Panvel +3

Infix to prefix conversion

Expression = **(A+B^C)*D+E^5**

Step 1. Reverse the infix expression.

5^E+D*)C^B+A(

Step 2. Make Every '(' as ')' and every ')' as '('

5^E+D*(C^B+A)

Step 3. Convert expression to postfix form.

Expression	Stack	Output
5^E+D*(C^B+A)	Empty	-
^E+D*(C^B+A)	Empty	5
E+D*(C^B+A)	^	5
+D*(C^B+A)	^	5E
D*(C^B+A)	+	5E^
*(C^B+A)	+	5E^D
(C^B+A)	+*	5E^D
C^B+A)	+*(	5E^D
^B+A)	+*(	5E^DC
B+A)	+*(^	5E^DC
+A)	+*(^	5E^DCB
A)	+*(+	5E^DCB^
)	+*(+	5E^DCB^A
End	+*	5E^DCB^A+
End	Empty	5E^DCB^A+*+

Step 4. Reverse the expression.

+*+A^BCD^E5

Result

+*+A^BCD^E5

Algorithm for Postfix to infix conversion

•

1. Initialize an empty stack to store operands and intermediate results.
2. Scan the postfix expression from left to right, one token (character/symbol) at a time.
3. For each token:

- **a. If the token is an operand** (e.g., a variable or a number), push it onto the stack.
- **b. If the token is an operator** (e.g., +, -, *, /):
  - Pop the top two elements from the stack. Let the first popped element be op2 and the second popped element be op1.
  - Form a new string (representing a sub-expression) by concatenating them in the format (+ op1 + operator + op2 +). The parentheses are crucial for maintaining the correct order of operations in the resulting infix expression.
  - Push this newly created infix sub-expression string back onto the stack.

4. After scanning the entire postfix expression, the stack will contain exactly one element, which is the final, fully parenthesized infix expression.

Example: Converting ab*c+

Example: Converting `ab*c+`

Step	Token	Action	Stack Content
1	a	Operand, push	["a"]
2	b	Operand, push	["a", "b"]
3	*	Operator, pop b and a, form (a*b), push	["(a*b)"]
4	c	Operand, push	["(a*b)", "c"]
5	+	Operator, pop c and (a*b), form ((a*b)+c), push	["((a*b)+c)"]

Postfix to infix conversion

Example:

- **Postfix:** `ab+c*`
- Scan 'a': Push 'a'
- Scan 'b': Push 'b'
- Scan '+': Pop 'b', pop 'a'. New string: `(a+b)`. Push `(a+b)`.
- Scan 'c': Push 'c'
- Scan '*': Pop 'c', pop `(a+b)`. New string: `((a+b)*c)`. Push `((a+b)*c)`.
- **Result:** `((a+b)*c)`

Recursion

- Recursion is the most used technique for problem-solving where a function runs repeatedly.
- It contains two parts such as base condition and recurrence relation.
- **Base Condition:** The base condition is also known as the base case or termination condition. The condition that tells a recursive function when to stop. It's the smallest instance of the problem that can be solved without recursion.
- **Recurrence Relation:** This is known as the actual relationship between the same function with different sizes of input. When implementing recursion to solve a problem, the call stack plays a crucial role in storing and managing function calls.

Steps to Implement Recursion

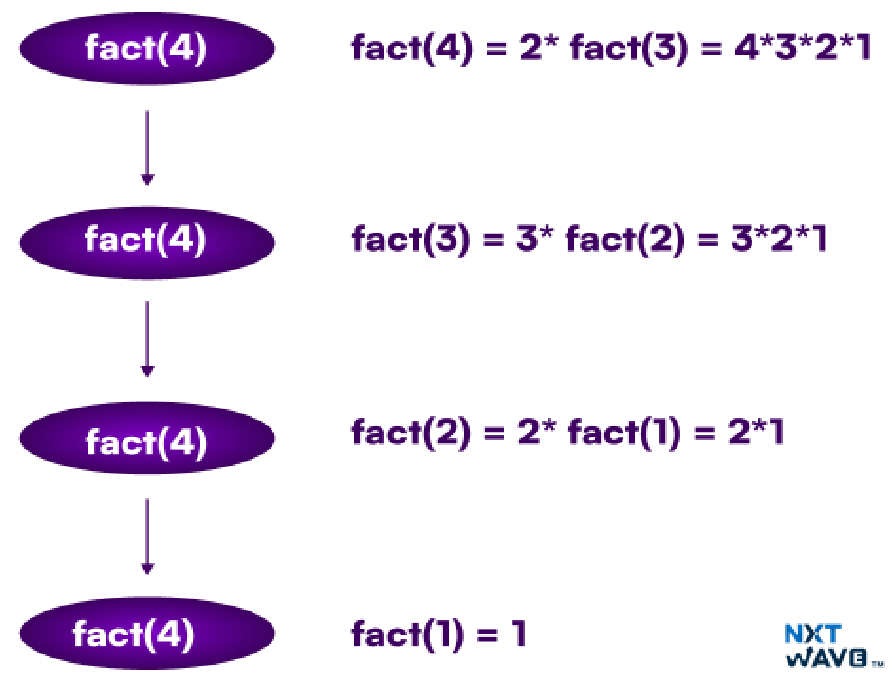
- **Step1 - Define a base case:** Identify the simplest (or base) case for which the solution is known or trivial. This is the stopping condition for the recursion, as it prevents the function from infinitely calling itself.

  Step2 - Define a recursive case: Define the problem in terms of smaller subproblems. Break the problem down into smaller versions of itself, and call the function recursively to solve each subproblem.

  Step3 - Ensure the recursion terminates: Make sure that the recursive function eventually reaches the base case, and does not enter an infinite loop.

  Step4 - Combine the solutions: Combine the solutions of the subproblems to solve the original problem.

- Calculate the factorial of number N using the function fact(N) = N * fact(N-1)
- The function fact(N) calls itself repeatedly is called recurrence relation.



Why Stack is Used for Recursion?

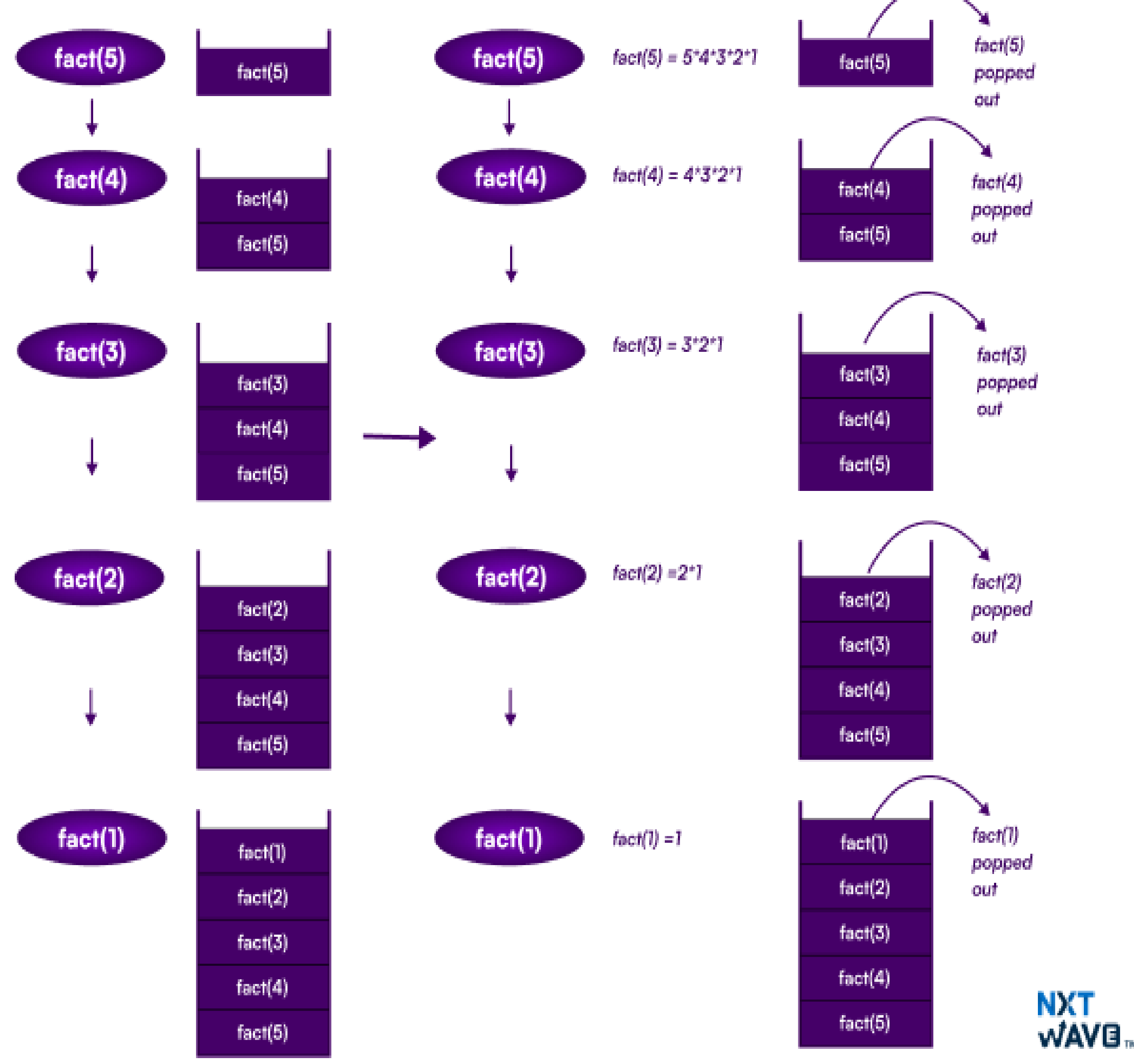
- A stack is used for recursion because it efficiently manages the function calls that are made during recursive execution.
- Each recursive call pushes a new stack frame (containing the function's state, local variables, and return address) onto the stack.
- When the base case is reached, the function calls unwind, and the stack is popped to return control to the previous function calls.
- This allows the program to keep track of recursive calls and properly resume execution when each call finishes.

Calculating Factorial

```
#include <stdio.h>
int factorial(int n)
{
if (n == 0 || n == 1)
    {
        return 1; // Base case
    }
else
     {
        return n * factorial(n - 1); // Recursive case
    }
}
 int main()
 {
    int result = factorial(5);
    printf("Factorial of 5 is: %d\n", result); // Output: 120
    return 0;
 }
```

How stack works here

- When factorial(5) is called, the function call is pushed onto the stack.
- Then, factorial(4) is called and pushed onto the stack, and so on.
- When the base case factorial(1) is reached, it returns 1, and the recursive calls start unwinding.
- The stack pops off each call, multiplying the results as it unwinds to return the final result.



fact(5)
fact(4)
fact(3)
fact(2)
fact(1)
fact(5) = 5*4*3*2*1
fact(4) = 4*3*2*1
fact(3) = 3*2*1
fact(2) =2*1
fact(1) =1
fact(5) popped out
fact(4) popped out
fact(3) popped out
fact(2) popped out
fact(1) popped out
NXT WAVE