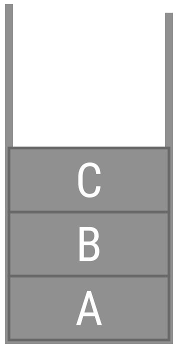
Unit-2

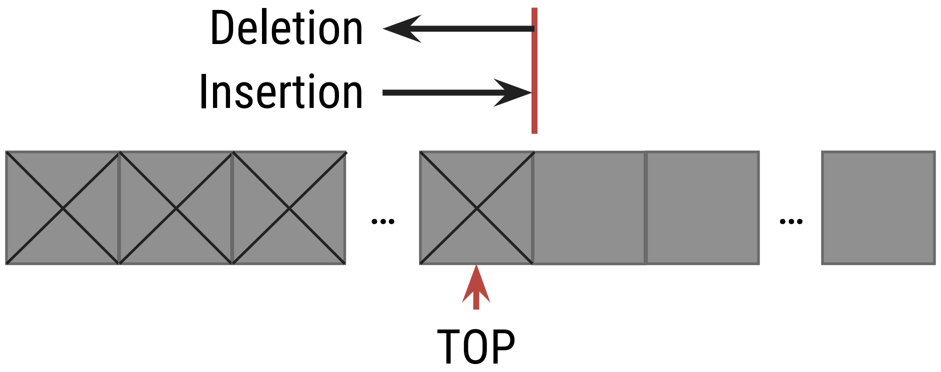
Linear Data Structure (Stack)

Computer Engineering Department

Stack

- A linear list which allows insertion and deletion of an element at one end only is called ***stack***.
- The insertion operation is called as ***PUSH*** and deletion operation as ***POP***.
- The most accessible elements in stack is known as ***top***.
- The elements can only be removed in the opposite orders from that in which they were added to the stack.
- Such a linear list is referred to as a ***LIFO (Last In First Out)*** list.





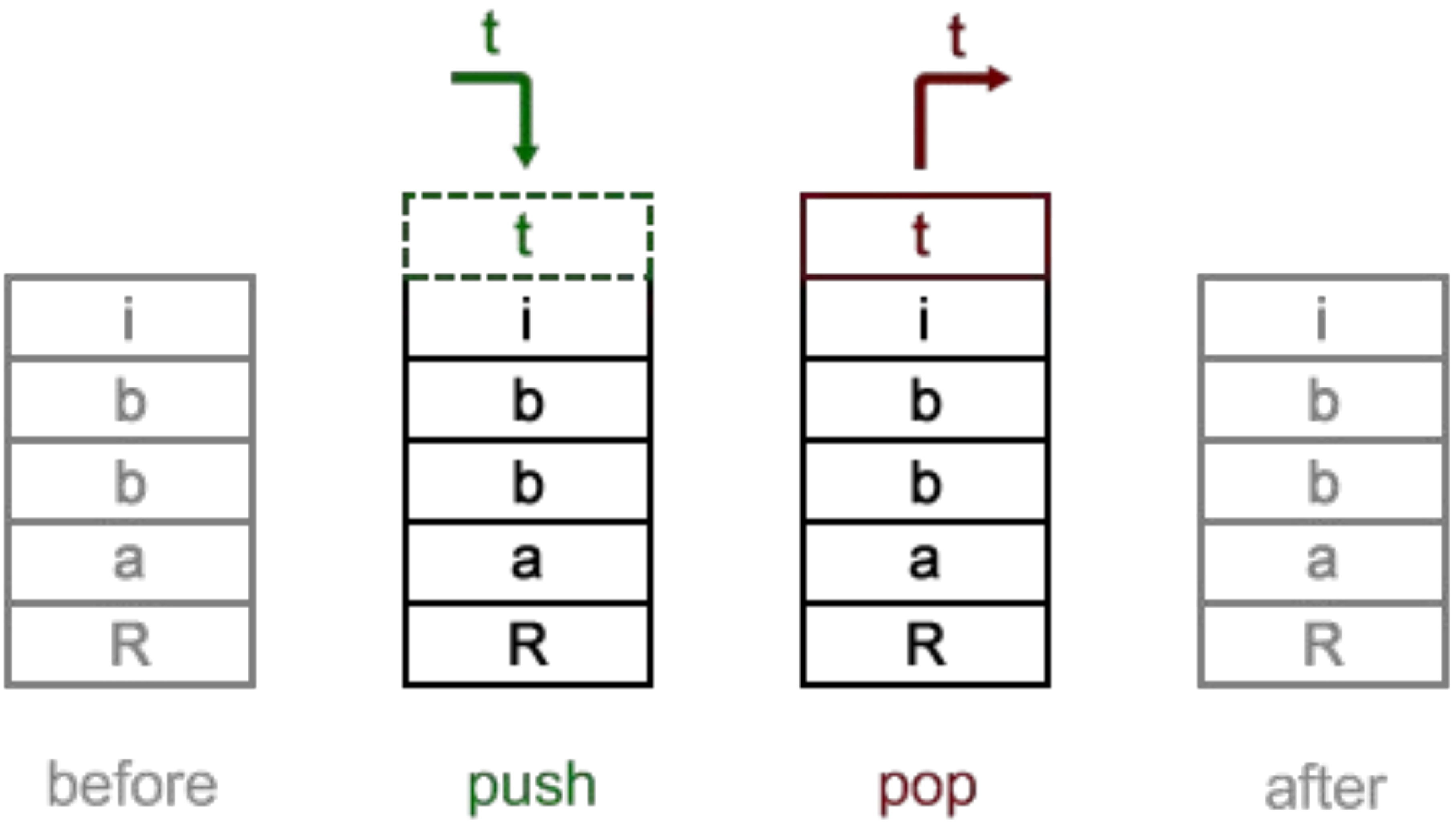
Stack Cont…

- A pointer TOP keeps track of the top element in the stack.
- Initially, when the ***stack is empty***, **TOP** has a value of ***“zero”***.
- Each time a **new element is inserted** in the stack, the pointer is ***incremented by “one”*** before, the element is placed on the stack.
- The pointer is ***decremented by “one”*** each time a **deletion** is made from the stack.

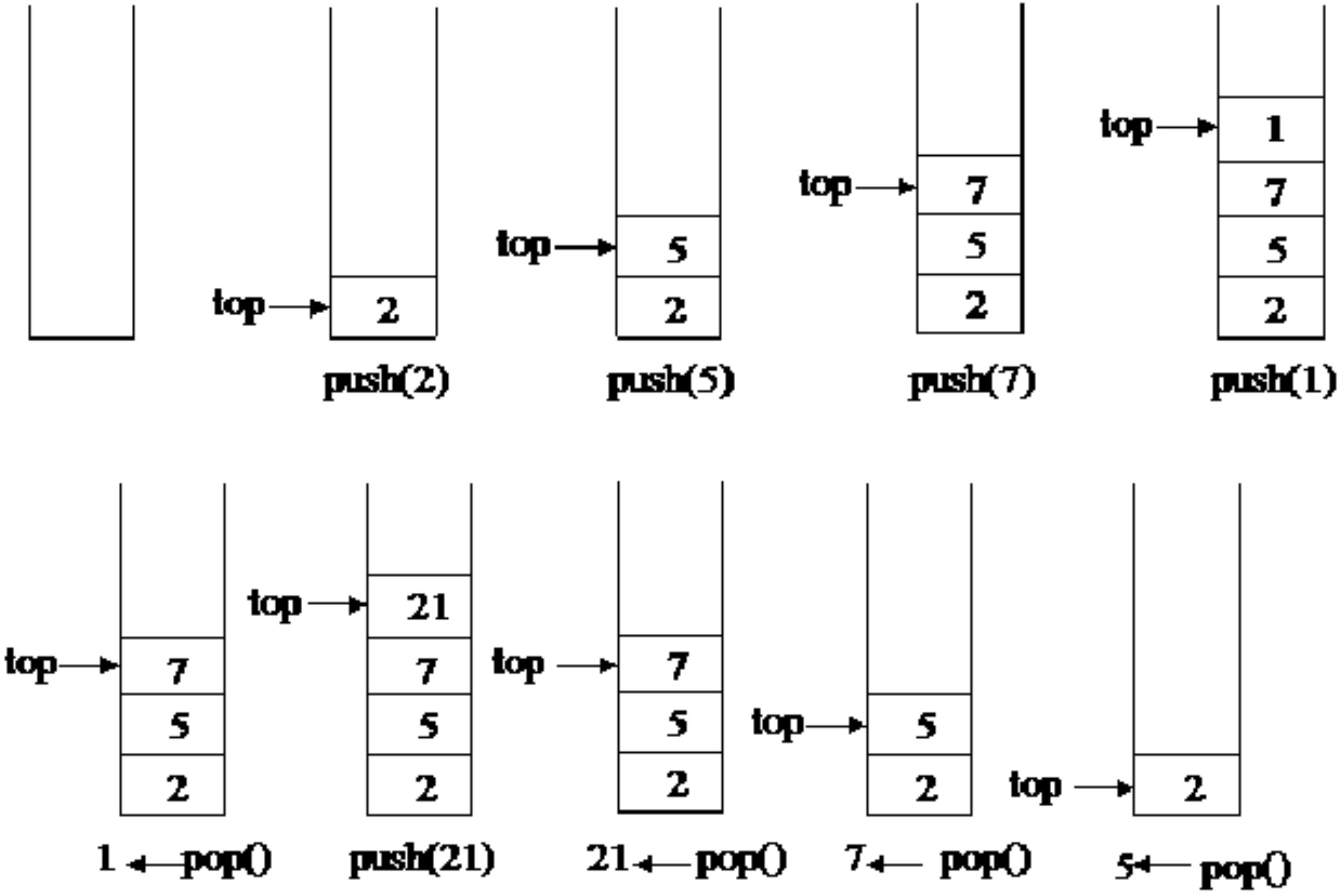
Stack Cont…


A stack of
cafeteria trays
A stack
of pennies
A stack of
shoe boxes
A stack of
neatly folded shirts

Stack Cont...

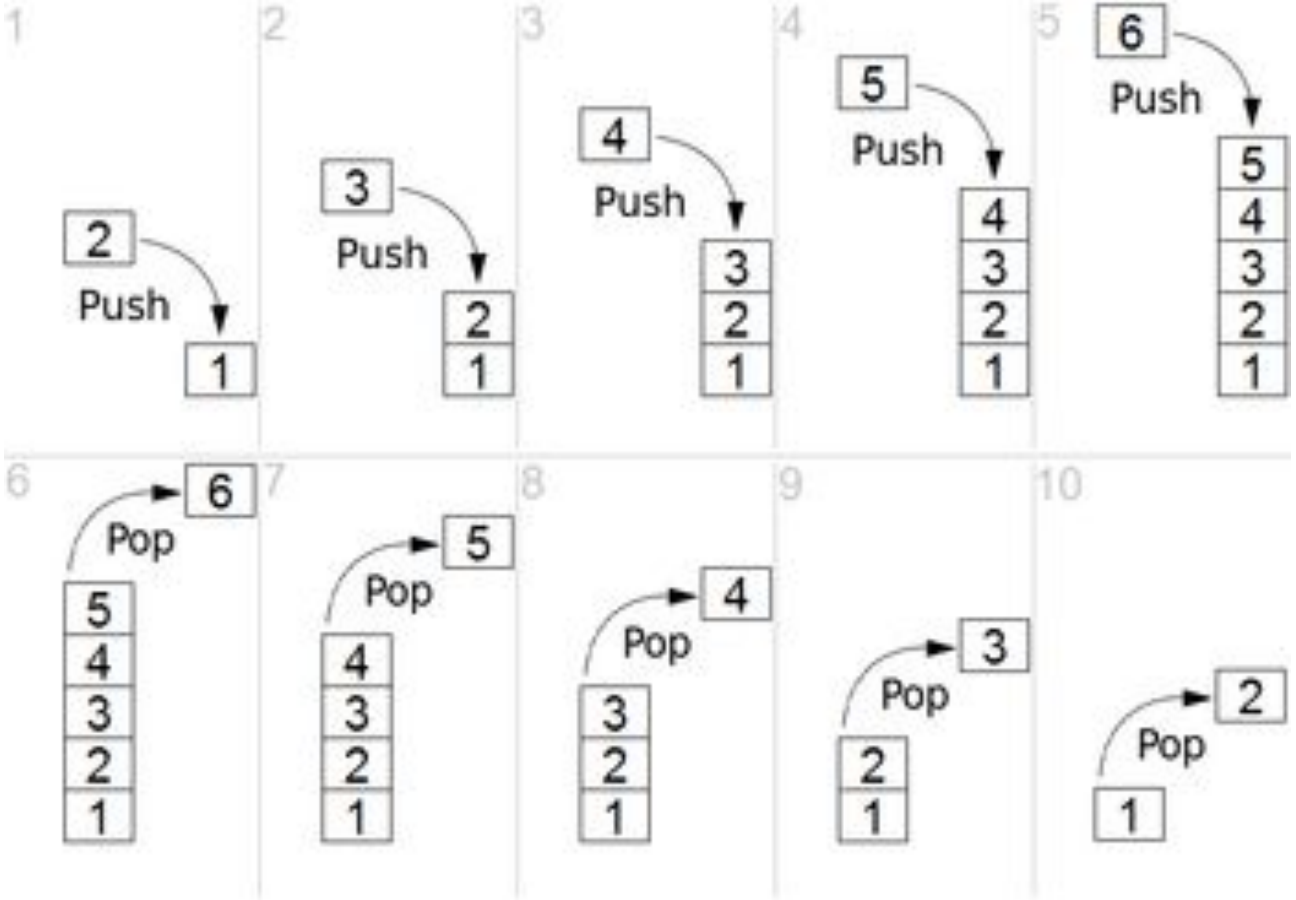


Stack Cont…



top
2
push(2)
top
5
2
push(5)
top
7
5
2
push(7)
top
1
7
5
2
push(1)
top
7
5
2
1 pop()
top
21
7
5
2
push(21)
top
7
5
2
21 pop()
top
5
2
7 pop()
top
2
5 pop()

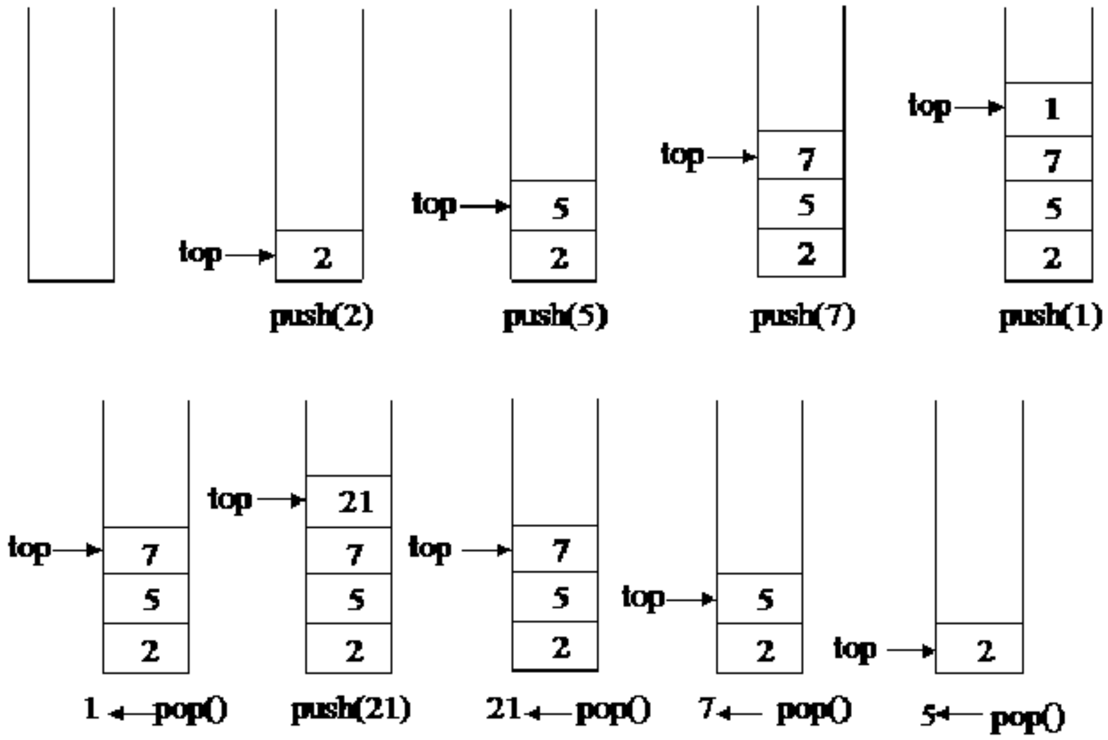
Stack Cont...



1
2
Push
1
2
3
Push
2
1
3
4
Push
3
2
1
4
5
Push
4
3
2
1
5
6
Push
5
4
3
2
1
6
6
Pop
5
4
3
2
1
7
5
Pop
4
3
2
1
8
4
Pop
3
2
1
9
3
Pop
2
1
10
2
Pop
1

Operation on Stack

- **Push** – Insert new element at Top of stack
- **Pop** – Delete Top element from stack
- **Peep** – Access Element by Specified Position from Top Side
- **Change** – Modified Element by Specified Position from Top Side



Procedure : PUSH (S, TOP, X)

- This procedure inserts an element **X** to the top of a stack.
- Stack is represented by a vector **S** containing **N** elements.
- A pointer **TOP** represents the top element in the stack.

```
1. [Check for stack overflow]
   If    TOP ≥ N
   Then  write ('STACK OVERFLOW')
            Return
2. [Increment TOP]
   TOP ← TOP + 1
3. [Insert Element]
   S[TOP] ← X
4. [Finished]
   Return
```

Stack is empty, TOP = 0, N=3

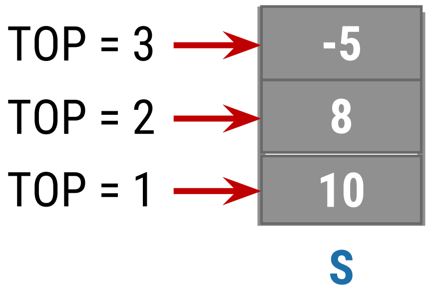
PUSH(S, TOP, 10)

PUSH(S, TOP, 8)

PUSH(S, TOP, -5)

PUSH(S, TOP, 6)

Overflow



Function : POP (S, TOP)

- This function ***removes & returns*** the top element from a stack.
- Stack is represented by a vector **S** containing **N** elements.
- A pointer **TOP** represents the top element in the stack.

```
1. [Check for stack underflow]
   If     TOP = 0
   Then   write (‘STACK UNDERFLOW’)
          Return (0)
2. [Decrement TOP]
   TOP ← TOP - 1
3. [Return former top element of
   stack]
   Return(S[TOP + 1])
```

POP(S, TOP)

POP(S, TOP)

POP(S, TOP)

POP(S, TOP)

Underflow

TOP = 3 →	-5
TOP = 2 →	8
TOP = 1 →	10
TOP = 0	**S**

Function : PEEP (S, TOP, I)

- This function returns the value of the **I**th element from the **TOP** of the stack. The element is not deleted by this function.
- Stack is represented by a vector **S** containing **N** elements.

```
1. [Check for stack underflow]
   If     TOP-I+1 ≤ 0
   Then   write ('STACK UNDERFLOW')
             Return (0)
2. [Return Ith element from top
    of the stack]
   Return(S[TOP-I+1])
```

PEEP (S, TOP, 2)

PEEP (S, TOP, 3)

PEEP (S, TOP, 4)

Underflow

TOP = 3 →

-5
8
10

S

PROCEDURE : CHANGE (S, TOP, X, I)

- This procedure changes the value of the **Ith** element from the top of the stack to **X**.
- Stack is represented by a vector **S** containing **N** elements.

```
1. [Check for stack underflow]
   If    TOP-I+1 ≤ 0
   Then  write ('STACK UNDERFLOW')
      Return
2. [Change Ith element from top
    of the stack]
   S[TOP-I+1] ← X
3. [Finished]
   Return
```

CHANGE (S, TOP, 50, 2)

CHANGE (S, TOP, 9, 3)

CHANGE (S, TOP, 25, 8)

Underflow

TOP = 3 →

S
-5
50
9

S

Applications of Stack

- Recursion
- Keeping track of function calls
- Evaluation of expressions
- Reversing characters
- Servicing hardware interrupts
- Solving combinatorial problems using backtracking
- Expression Conversion (Infix to Postfix, Infix to Prefix)
- Game Playing (Chess)
- Microsoft Word (Undo / Redo)
- Compiler – Parsing syntax & expression
- Finding paths

Algorithm: RECOGNIZE

Write an algorithm which will check that the given string belongs to following grammar or not.

$L = \{wcw^R \mid w \in \{a,b\}^*\}$ (Where w^R is the reverse of w)

Example of valid strings : abcba aabbcbbaa
Example of Invalid strings: aabcaab

- Given an input string named ***STRING*** on the alphabet ***{a, b, c}*** which contains a blank in its rightmost character position.
- Function **NEXTCHAR** returns the next symbol in STRING.
- This algorithm determines whether the contents of STRING belong to the above language.
- The vector **S** represents the stack and **TOP** is a pointer to the top element of the stack.

Algorithm: RECOGNIZE

1. [Initialize stack by placing a letter ‘c’ on the top]
```
TOP ⍰ 1
S [TOP] ← ‘c’
```

2. [Get and PUSH symbols until either ‘c’ or blank is encountered]
```
NEXT ← NEXTCHAR (STRING)
Repeat while NEXT ≠ ‘c’
If    NEXT = ‘ ‘
Then Write (‘Invalid String’)
     Exit
Else Call PUSH (S, TOP, NEXT)
     NEXT ← NEXTCHAR (STRING)
```

3. [Scan characters following ‘c’; Compare them to the characters on stack]
```
Repeat While S [TOP] ≠ ‘c’
NEXT ← NEXTCHAR (STRING)
X ← POP (S, TOP)
If    NEXT ≠ X
Then Write(‘INVALID STRING’)
  Exit
```

4. [Next symbol must be blank]
```
NEXT ← NEXTCHAR (STRING)
If   NEXT = ‘ ‘
Then Write (‘VALID STRING’)
Else Write (‘INVALID STRING’)
```

5. [Finished]
```
Exit
```

Algorithm: RECOGNIZE

```
1.   [Initialize stack by placing
      a letter ‘c’ on the top]
     TOP ⍰ 1
     S [TOP] ← ‘c’

2.   [Get and PUSH symbols until
     either c’ or blank is
     encountered]
     NEXT ← NEXTCHAR (STRING)
     Repeat while NEXT ≠ ‘c’
     If    NEXT = ‘ ‘
     Then Write (‘Invalid String’)
          Exit
     Else Call PUSH (S, TOP, NEXT)
          NEXT ← NEXTCHAR (STRING)
```

Input String: a b c b a □

NEXT

	Character Scanned	Stack Content
1		c
2	a	ca
	b	cab
	c	cab

Algorithm: RECOGNIZE

```
3.   Scan characters following 'c';
     Compare them to the characters on stack
     Repeat While S[TOP] ≠ 'c'
       NEXT ← NEXTCHAR (STRING)
       X ← POP (S, TOP)
       If NEXT ≠ X
       Then Write('Invalid String')
            Exit

4.   [Next symbol must be blank]
     NEXT ← NEXTCHAR (STRING)
     If  NEXT = ' '
     Then   Write ('VALID STRING')
     Else   Write ('INVALID STRING')
```

Input String: a b c b a □

NEXT

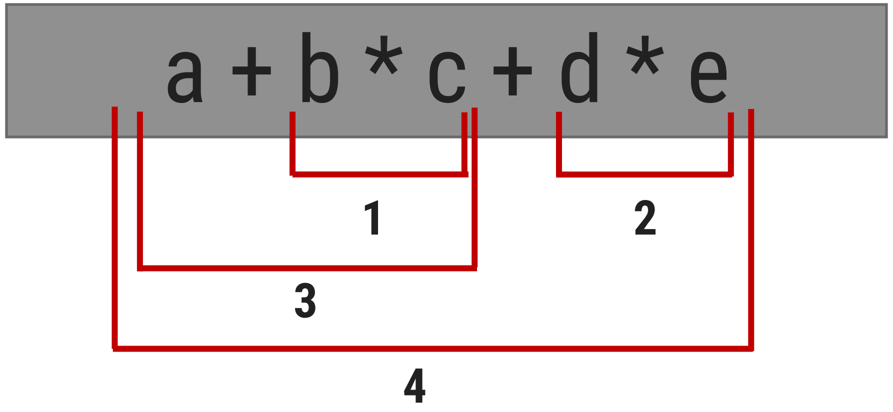
	Character Scanned	Stack Content
1		c
2	a	ca
	b	cab
	c	cab
3	b	ca
	a	c
	□	

Algorithm : RECOGNIZE

- Write an algorithm to determine if an input character string is of the form ***$a^i b^i$ where $i>=1$***
- i.e. number of **a** ***should be equal*** to no of **b**

Polish Expression & their Compilation

- Evaluating Infix Expression



- A **repeated scanning** **from left to right is needed** as operators appears inside the operands.
- ***Repeated scanning is avoided*** if the **infix expression** is first **converted** to an equivalent parenthesis free ***prefix or suffix (postfix) expression***.
- **Prefix Expression: Operator**, Operand, Operand
- **Postfix Expression:** Operand, Operand, **Operator**

Polish Notation

- This type of notation is known **Lukasiewicz Notation** or **Polish Notation** or **Reverse Polish Notation** due to Polish logician *Jan Lukasiewicz*.
- In both **prefix** and **postfix** equivalents of an infix expression, the ***variables are in same relative position***.
- The expressions in postfix or prefix form are ***parenthesis free*** and operators are rearranged according to rules of precedence for operators.

Polish Notation

Sr.	Infix	Postfix	Prefix
1	a	a	a
2	a + b	a b +	+ a b
3	a + b + c	a b + c +	+ + a b c
4	a + (b + c)	a b c + +	+ a + b c
5	a + (b * c)	a b c * +	+a * b c
6	a * (b + c)	a b c + *	* a + b c
7	a * b * c	a b *c*	** a b c

a + b + c → a + b + c → (ab+)+ c → (ab+) c + → a b + c +

Finding Rank of any Expression

E = (A + B * C / D - E + F / G / (H + I))

Note: R = *Rank*, Rank of Variable *= 1*, Rank of binary operators *= -1*

Rank (E) = R(A) + R(+) + R(B) + R(*) + R(C) + R (/) + R(D) + R(-) + R(E) + R(+) + R(F) + R(/) + R(G) + R(/) + R(H) + R(+) + R(I)

Rank (E) = 1 + **(-1)** + 1 + **(-1)** + 1 + **(-1)** + 1 + **(-1)** + 1 + **(-1)** + 1 + **(-1)** + 1 + **(-1)** + 1 + **(-1)** + 1

Rank (E) = 1

Any Expression is valid if Rank of that expression is 1

Convert Infix to Postfix Expression – Without Parenthesis

Symbol	Input precedence function (F)	Rank function (R)
+, -	1	-1
***, /**	2	-1
Variables	3	1
#	0	-

Algorithm : REVPOL - Without Parenthesis

- Given an input string **INFIX** containing an infix expression which has been padded on the right with **'#'**.
- This algorithm ***converts INFIX into reverse polish*** and places the result in the string **POLISH**.
- All symbols have precedence value given by the table.
- Stack is represented by a vector **S, TOP** denotes the top of the stack, algorithm **PUSH** and **POP** are used for stack manipulation.
- Function **NEXTCHAR** returns the next symbol in given input string.
- The integer variable **RANK** contains the rank of expression.
- The string variable **TEMP** is used for temporary storage purpose.

1. [Initialize Stack]

TOP ⍰ 1
S[TOP] ← ‘#’

2. [Initialize output string and rank count]

POLISH ⍰ ‘’
RANK ⍰ 0

3. [Get first input symbol]

NEXT ⍰ NEXTCHAR(INFIX)

4. [Translate the infix expression]

Repeat thru step 6
while NEXT != ‘#‘

5. [Remove symbols with greater or equal precedence from stack]

Repeat while F(NEXT) <= F(S[TOP])
TEMP ⍰ POP (S, TOP)
POLISH ⍰ POLISH O TEMP
RANK ⍰ RANK + R(TEMP)
IF RANK <1
Then write (‘INVALID’)
EXIT

6. [Push Current Symbol onto stack and obtain next input symbol]

call PUSH (S,TOP, NEXT)
NEXT ⍰ NEXTCHAR(INFIX)

5. [Remove remaining elements from stack]

Repeat while (S[TOP] != ‘#’)
TEMP ⍰ POP (S, TOP)
POLISH ⍰ POLISH O TEMP
RANK ⍰ RANK + R(TEMP)
IF RANK <1
Then write (‘INVALID’)
EXIT

8. [Is the expression valid]

IF RANK != 1
Then write (‘INVALID‘)
Else write (‘VALID’)
EXIT

Symbol	IPF (F)	RF(R)
+, -	1	-1
*, /	2	-1
Variables	3	1
#	0	-

Convert Infix to Postfix Expression

Symbol	Input precedence function (F)	Stack precedence function (G)	Rank function (R)
+, -	1	2	-1
***, /**	3	4	-1
^	6	5	-1
Variables	7	8	1
(	9	0	-
)	0	-	-

Algorithm : REVPOL

- Given an input string **INFIX** containing an infix expression which has been padded on the right with **')'**.
- This algorithm ***converts INFIX into reverse polish*** and places the result in the string **POLISH**.
- All symbols have precedence value given by the table.
- Stack is represented by a vector **S, TOP** denotes the top of the stack, algorithm **PUSH** and **POP** are used for stack manipulation.
- Function **NEXTCHAR** returns the next symbol in given input string.
- The integer variable **RANK** contains the rank of expression.
- The string variable **TEMP** is used for temporary storage purpose.

1. [Initialize Stack]

```
TOP ⍰ 1
S[TOP] ← '('
```

2. [Initialize output string and rank count]

```
POLISH ⍰ ''
RANK ⍰ 0
```

3. [Get first input symbol]

```
NEXT ⍰ NEXTCHAR(INFIX)
```

4. [Translate the infix expression]

```
Repeat thru step 7
while NEXT != ' '
```

Symbol	IPF (F)	SPF (G)	RF (R)
+, -	1	2	-1
*, /	3	4	-1
^	6	5	-1
Variables	7	8	1
(	9	0	-
)	0	-	-

5. [Remove symbols with greater precedence from stack]

```
IF   TOP < 1
Then write ('INVALID')
 EXIT
Repeat while F(NEXT) < G(S[TOP])
TEMP ⍰ POP (S, TOP)
POLISH ⍰ POLISH O TEMP
RANK ⍰ RANK + R(TEMP)
IF   RANK <1
Then write ('INVALID')
     EXIT
```

6. [Are there matching parentheses]

```
IF   F(NEXT) ! = G(S[TOP])
Then call PUSH (S,TOP, NEXT)
Else POP (S,TOP)
```

7. [Get next symbol]

```
NEXT ⍰ NEXTCHAR(INFIX)
```

8. [Is the expression valid]

```
IF   TOP != 0 OR RANK != 1
Then write ('INVALID')
Else write ('VALID')
```

(a + b ^ c ^ d) * (e + f / d))

NEXT

1. [Initialize Stack]
   TOP ⍰ 1
   S[TOP] ← '('

2. [Initialize output string and rank count]
   POLISH ⍰ ''
   RANK ⍰ 0

3. [Get first input symbol]
   NEXT ⍰ NEXTCHAR(INFIX)

Input Symbol	Content of stack	Reverse polish expression	Rank
	(		0
(			

(a + b ^ c ^ d) * (e + f / d))

NEXT

```
4. [Translate the infix expression]
     Repeat thru step 7
      while NEXT!= ' '

5. [Remove symbols with greater precedence from
    stack]
    IF   TOP < 1
    Then write ('INVALID')
         EXIT
    Repeat while F(NEXT) < G(S[TOP])
        TEMP ⍰ POP (S, TOP)
        POLISH ⍰ POLISH O TEMP
        RANK ⍰ RANK + R(TEMP)
        IF     RANK <1
        Then write ('INVALID')
             EXIT

6. [Are there matching parentheses]
     IF   F(NEXT) != G(S[TOP])
     Then call PUSH (S,TOP, NEXT)
     Else POP (S,TOP)

7. [Get next symbol]
     NEXT ⍰ NEXTCHAR(INFIX)
```

Input Symbol (NEXT)	Content of Stack (S)	Reverse polish expression	Rank
	(		0
(	((		0
a	((a		0
+	((+	a	1
b	((+b	a	1
^	((+^	ab	2
c	((+^c	ab	2
^	((+^^	abc	3
d	((+^^d	abc	3
)	((	abcd^^+	1

Perform following operations

- **Convert** the following **infix** expression to **postfix**. Show the stack contents.
  - A$B-C*D+E$F/G
  - A + B – C * D * E $ F $ G
  - a+b*c-d/e*h
  - ((a+b^c^d)*(e+f/d))
- Convert the following **infix** expression to **prefix**.
  - A + B – C * D * E $ F $ G

General Infix-to-Postfix Conversion

Create an empty stack called stack for keeping operators. Create an empty list for output.

Read the character list from left to right and perform following steps

1	If the **character** is an **operand (Variable)**, **append** it to the **end** of the **output** list
2	If the **character** is a **left parenthesis '('**, **push** it on the **stack**
3	If the **character** is a **right parenthesis ')'**, **pop** the **stack** **until** the corresponding **left parenthesis** '(' is **removed**. **Append** each **operator** to the **end of the output** list.
4	If the token is an **operator**, ***, /, +,** or **-**, **push** it on the **stack**. However, **first remove any operators** already on the **stack** that **have higher or equal precedence** and **append them to the output** list.

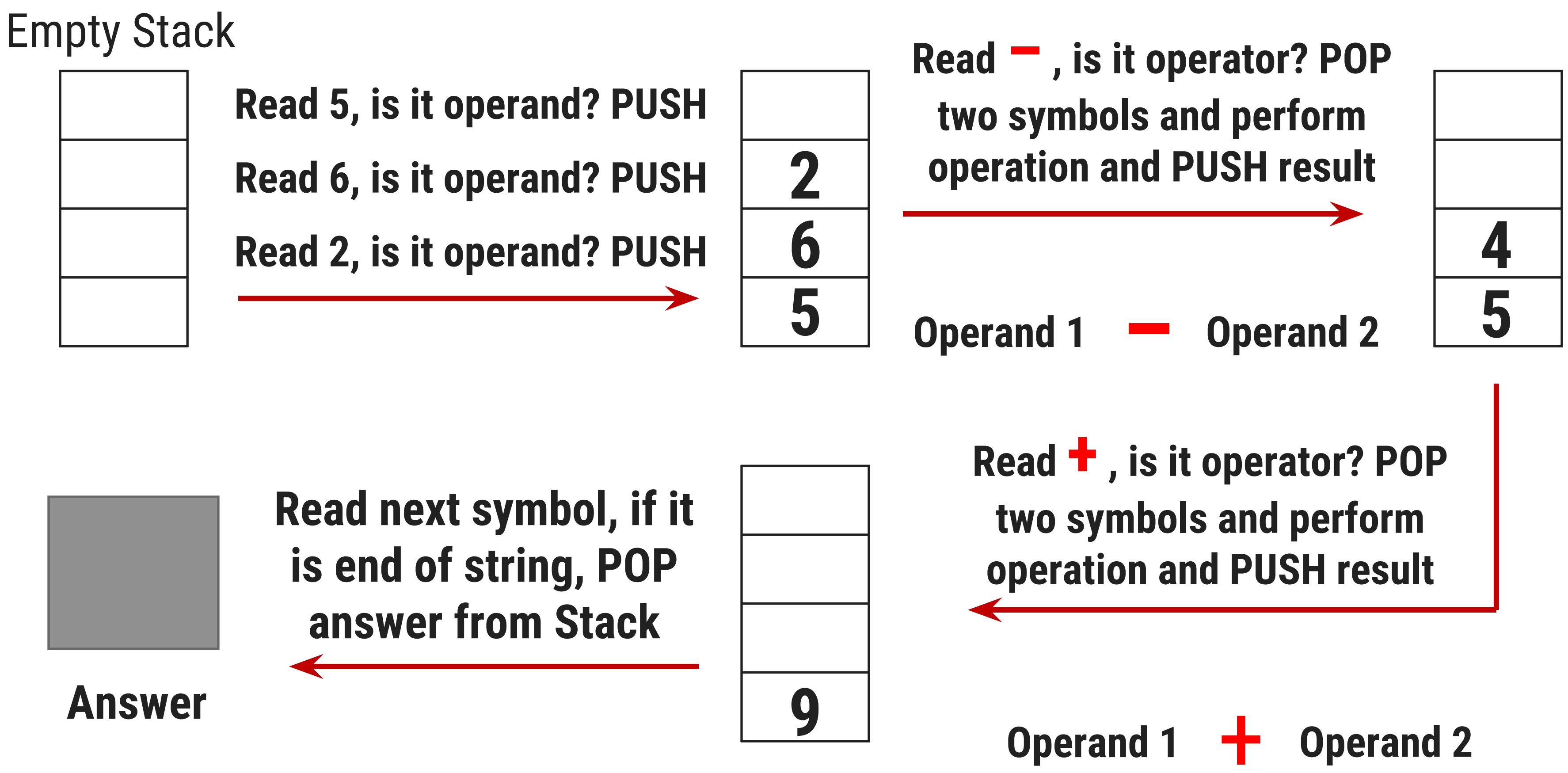
(a + b ^ c ^ d) * (e + f / d)

Evaluation of postfix expression

- Each **operator** in **postfix** string **refers** to the *previous two operands* in the string.
- Each time we **read** an **operand**, we **PUSH** it onto **Stack**.
- When we reach an **operator**, its **operands** will be **top two elements** on the stack.
- We can then **POP** these two elements, perform the indicated operation on them and PUSH the result on the stack so that it will available for use as an operand for the next operator.
  - **a b c d ^ ^ +**
  - **a b (c^d) ^ +**
  - **a (b ^ (c ^ d)) +**
  - **a + b ^ c ^ d**

Evaluation of postfix expression

Evaluate Expression: 5 6 2 - +



Algorithm: EVALUATE_POSTFIX

- Given an input string **POSTFIX** representing postfix expression.
- This algorithm evaluates postfix expression and put the result into variable **VALUE**.
- Stack is represented by a vector **S, TOP** denotes the top of the stack, Algorithm **PUSH** and **POP** are used for stack manipulation.
- Function **NEXTCHAR** returns the next symbol in given input string.
- **OPERAND1**, **OPERAND2** and **TEMP** are used for temporary variables
- **PERFORM_OPERATION** is a function which performs required operation on **OPERAND1** & **OPERAND2**.

Algorithm: EVALUATE_POSTFIX

```
1. [Initialize Stack]
    TOP ⍰ 0
    VALUE ⍰ 0

2. [Evaluate the postfix expression]
   Repeat until last character
      TEMP ⍰ NEXTCHAR (POSTFIX)
      If TEMP is DIGIT
      Then PUSH (S, TOP, TEMP)
      Else OPERAND2 ⍰ POP (S, TOP)
           OPERAND1 ⍰ POP (S, TOP)
           VALUE ⍰ PERFORM_OPERATION(OPERAND1, OPERAND2, TEMP)
           PUSH (S, POP, VALUE)

3. [Return answer from stack]
   Return (POP (S, TOP))
```

Evaluation of postfix expression

Evaluate Expression: 5 4 6 + * 4 9 3 / + *

Evaluate Expression: 7 5 2 + * 4 1 1 + / -

Evaluate Expression: 12, 7, 3, -, /, 2, 1, 5, +, *, +

Algorithm: EVALUATE_PREFIX

- Given an input string **PREFIX** representing prefix expression.
- This algorithm evaluates prefix expression and put the result into variable **VALUE**.
- Stack is represented by a vector **S, TOP** denotes the top of the stack, Algorithm **PUSH** and **POP** are used for stack manipulation.
- Function **NEXTCHAR** returns the next symbol in given input string.
- **OPERAND1**, **OPERAND2** and **TEMP** are used for temporary variables
- **PERFORM_OPERATION** is a function which performs required operation on **OPERAND1** & **OPERAND2**.

Algorithm: EVALUATE_PREFIX

```
1. [Initialize Stack]
     TOP ⍰ 0
     VALUE ⍰ 0

2. [Evaluate the prefix expression]
    Repeat from last character up to first
        TEMP ⍰ NEXTCHAR (PREFIX)
        If   TEMP is DIGIT
        Then PUSH (S, TOP, TEMP)
        Else OPERAND1 ⍰ POP (S, TOP)
             OPERAND2 ⍰ POP (S, TOP)
             VALUE ⍰ PERFORM_OPERATION(OPERAND1, OPERAND2, TEMP)
             PUSH (S, POP, VALUE)

3. [Return answer from stack]
    Return (POP (S, TOP))
```

Recursion

A **procedure** that contains a **procedure call to itself** or a procedure **call to second procedure** which eventually **causes the first procedure to be called** is known as **recursive procedure**.

Two important conditions for any recursive procedure	
1	Each time a procedure calls itself it must be nearer in some sense to a solution.
2	There must be a decision criterion for stopping the process or computation.

Two types of recursion

Primitive Recursion	Non-Primitive Recursion
This is **recursive defined function**. E.g. Factorial function	This is **recursive use of procedure**. E.g. Find GCD of given two numbers

Algorithm to find factorial using recursion

- Given integer number **N**
- This algorithm **computes factorial of N**.
- **Stack S** is used to store an activation record associated with each recursive call.
- **TOP** is a pointer to the top element of stack S.
- Each **activation record contains** the current value of **N** and the current return address **RET_ADDE**.
- **TEMP_REC** is also a record which contains two variables **PARAM** & **ADDRESS**.
- **Initially** return address is set to the **main calling address**. PARAM is set to initial value N.

Algorithm: FACTORIAL

```
1. [Save N and return Address]
   CALL PUSH (S, TOP, TEMP_REC)
2. [Is the base criterion found?]
   If N=0
   then  FACTORIAL ⍰ 1
      GO TO Step 4     ( Start returning to previous called address )
   Else  PARAM ⍰ N-1
      ADDRESS ⍰ Step 3     // FACTORIAL -> N * FACTORIAL
      GO TO Step 1     ( Recursive call )
3. [Calculate N!]
   FACTORIAL⍰ N * FACTORIAL
4. [Restore previous N and return address]
   TEMP_REC ⍰ POP(S,TOP)
   (i.e. PARAM ⍰ N, ADDRESS ⍰ RET_ADDR)
   GO TO ADDRESS
```

Trace of Algorithm FACTORIAL, N=2

<table>
<tr><th>Level Number</th><th>Description</th><th>Stack Content</th></tr>
<tr><td>Enter Level 1
(main call)</td><td>Step 1: PUSH (S,0,(N=2, main address))
Step 2: N!=0
PARAM ← N-1 (1), ADDR ← Step 3</td><td><table><tr><td>2</td><td></td><td></td></tr><tr><td>Main Address</td><td></td><td></td></tr><tr><td>TOP</td><td></td><td></td></tr></table></td></tr>
<tr><td>Enter Level 2
(first recursive call)</td><td>Step 1: PUSH (S,1,(N=1, step 3))
Step 2: N!=0
PARAM ← N-1 (3), ADDR ← Step 3</td><td><table><tr><td>2</td><td>1</td><td></td></tr><tr><td>Main Address</td><td>Step 3</td><td></td></tr><tr><td></td><td>TOP</td><td></td></tr></table></td></tr>
<tr><td>Enter Level 3
(second recursive call)</td><td>Step 1: PUSH (S,2,(N=0, step 3))
Step 2: N=0
FACTORIAL ← 1

Step 4: POP(A,3)
GO TO Step 3</td><td><table><tr><td>2</td><td>1</td><td>0</td></tr><tr><td>Main Address</td><td>Step 3</td><td>Step 3</td></tr><tr><td></td><td></td><td>TOP</td></tr></table>
<table><tr><td>2</td><td>1</td><td></td></tr><tr><td>Main Address</td><td>Step 3</td><td></td></tr><tr><td></td><td>TOP</td><td></td></tr></table></td></tr>
</table>

Trace of Algorithm FACTORIAL, N=2

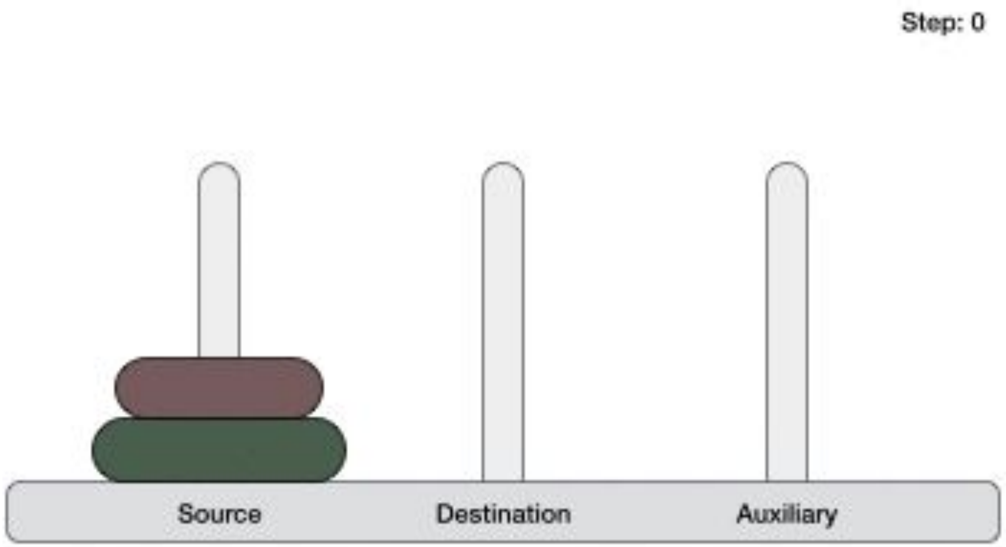
Level Number	Description	Stack Content
Return to Level 2	Step 3: FACTORIAL ← 1*1 Step 4: POP (A,2) GO TO Step 3	**2** **Main Address** **TOP**
Return to Level 1	Step 3: FACTORIAL ← 2*1 Step 4: POP (A,1) GO TO Main Address **TOP = 0**	

Algorithm to find GCD of two given numbers

- Given integera number **a and b**
- **findGCD(a, b)**
- Begin
  - if a = 0 OR b = 0,
    - return 0
  - if a = b,
    - return b
  - if a > b
    - return **findGCD(a-b , b)**
  - Else
    - return **findGCD(a , b-a)**
- End

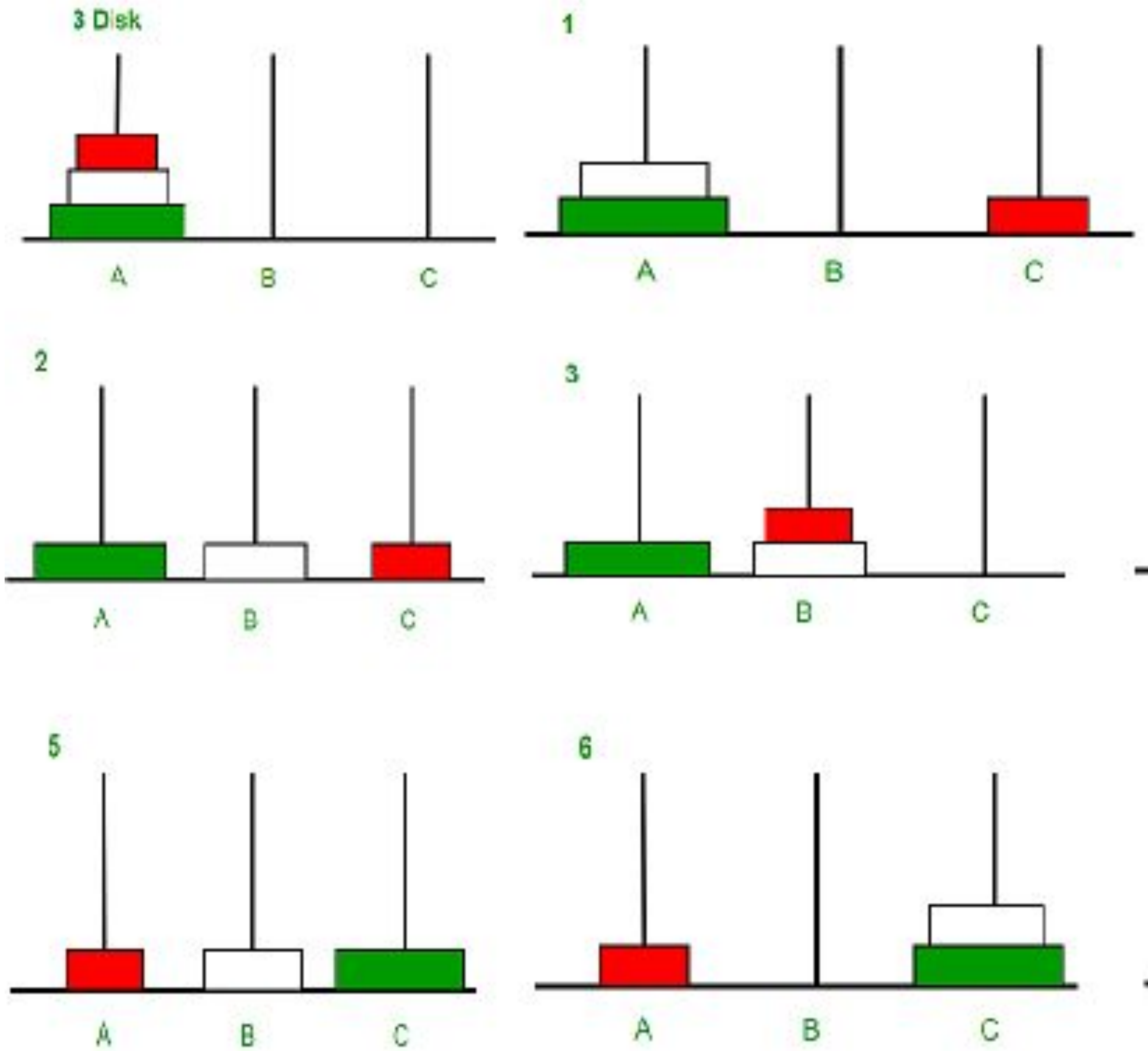
Tower of Hanoi

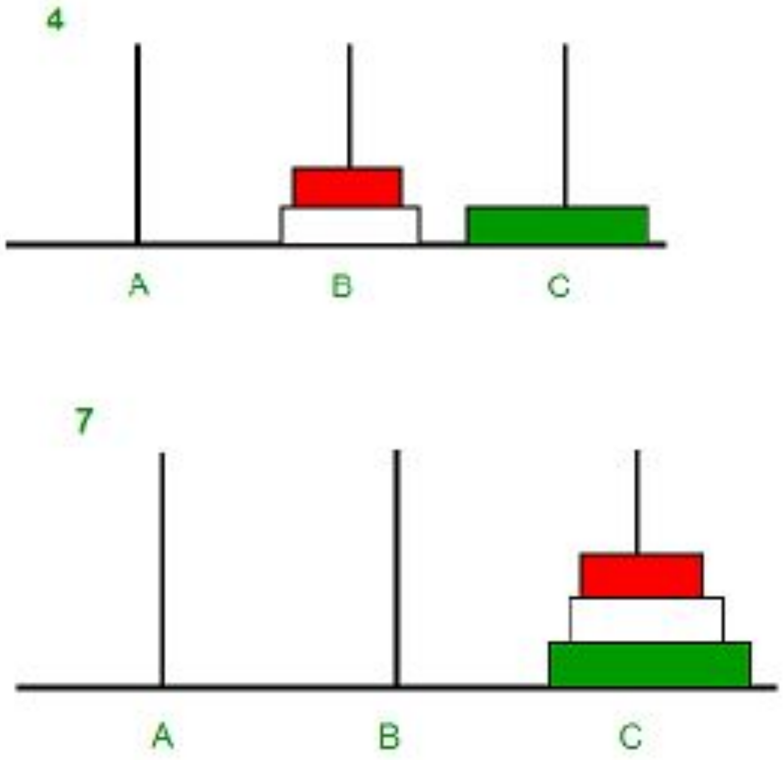
- We mark three towers with name, **source**, **destination** and **aux** (only to help moving the disks).
- If we have only one disk, then it can easily be moved from source to destination peg.
- If we have 2 disks –
  - First, we move the smaller (top) disk to from source to aux peg. **SRC -> AUX**
  - Then, we move the larger (bottom) disk from source to destination peg. **SRC -> DST**
  - And finally, we move the smaller disk from aux to destination peg. **AUX -> DST**



Tower of Hanoi

If disk = 1	move = 1
If disk = 2	move = 3
If disk = 3	move = 7
If disk = 4	move = 15
.	
.	
If disk = n	move = $2^n - 1$

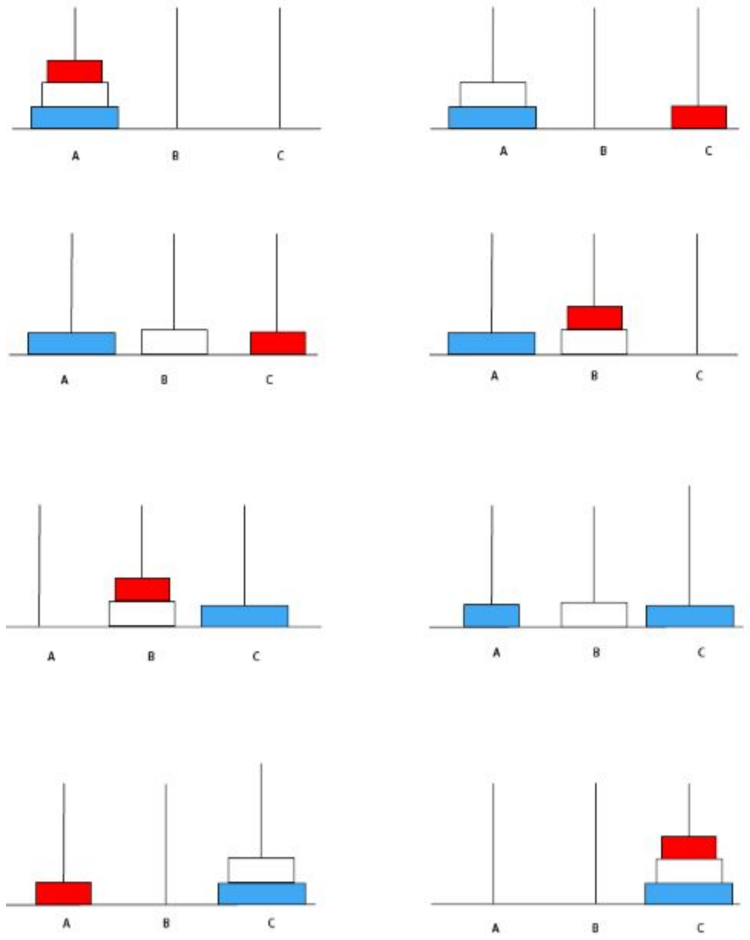




Tower of Hanoi

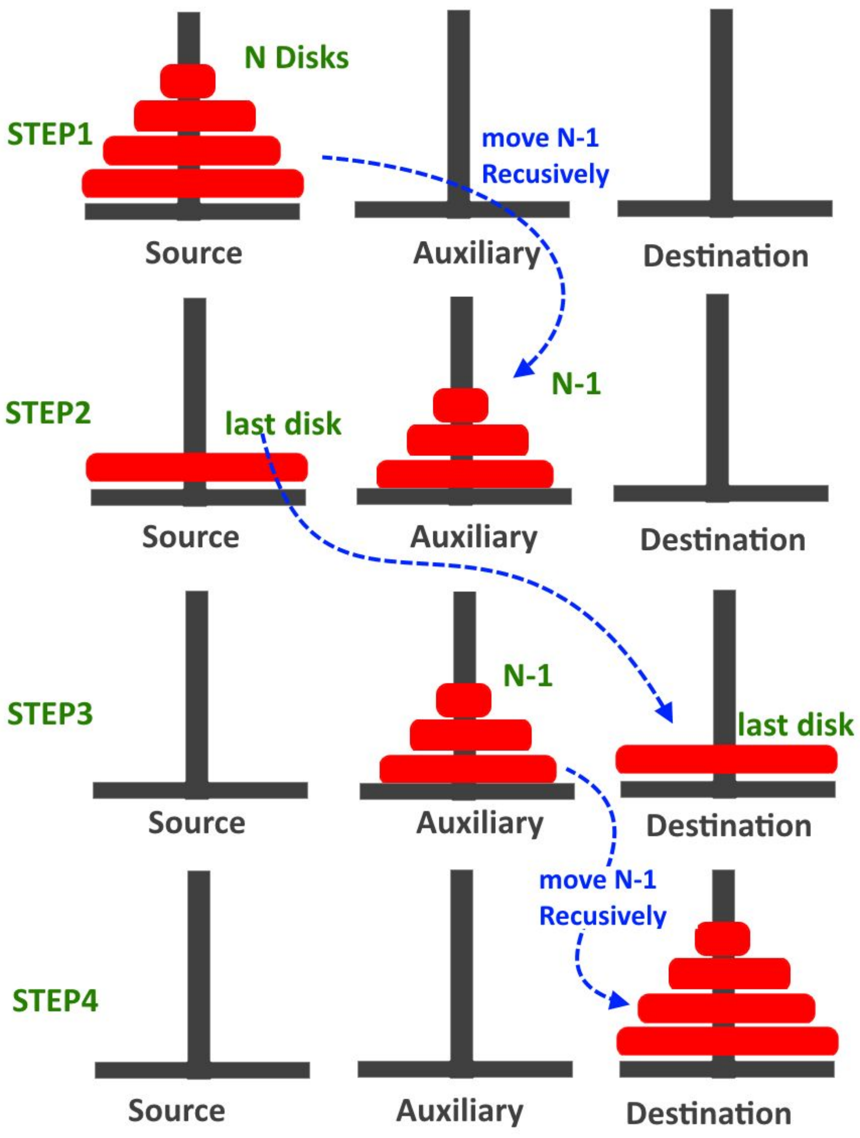
- Our ultimate aim is to move disk **n** from source to destination and then put all other (n1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.
- The steps to follow are –
- **Step 1** – Move n-1 disks from **source** to **aux**
- **Step 2** – Move n^{th} disk from **source** to **dest**
- **Step 3** – Move n-1 disks from **aux** to **dest**

Let us we have three disks stacked on a peg



Tower of Hanoi

- START
- Procedure Hanoi(disk, source, dest, aux)
    - IF disk == 1, THEN
        - move disk from source to dest
    - ELSE
        - Hanoi(disk - 1, source, aux, dest) // Step 1
        - move disk from source to dest // Step 2
        - Hanoi(disk - 1, aux, dest, source) // Step 3
    - END IF
- END Procedure
- STOP



MCQs

- Process of inserting an element in stack is called ____________
  - Create
  - Push
  - Evaluation
  - Pop
- Process of removing an element from stack is called __________
  - Create
  - Push
  - Evaluation
  - Pop
- In a stack, if a user tries to remove an element from empty stack it is called _________
  - Underflow
  - Empty collection
  - Overflow
  - Garbage Collection

MCQs

- Pushing an element into stack already having five elements and stack size of 5, then stack becomes
  - Overflow
  - Crash
  - Underflow
  - User flow
- Entries in a stack are “ordered”. What is the meaning of this statement?
  - A collection of stacks is sortable
  - Stack entries may be compared with the ‘<‘ operation
  - The entries are stored in a linked list
  - There is a Sequential entry that is one by one
- Which of the following applications may use a stack?
  - A parentheses balancing program
  - Tracking of local variables at run time
  - Compiler Syntax Analyzer
  - Data Transfer between two asynchronous process

MCQs

- What is the value of the postfix expression 6 3 2 4 + – *:
  - 1
  - 40
  - 74
  - -18
- Stack in Data Structure is ___________.
  - LILO
  - None of these
  - FIFO
  - LIFO

MCQs

- User perform following operations on stack of size 5 then –
  - push(1);
  - pop();
  - push(2);
  - push(3);
  - pop();
  - push(4);
  - pop();
  - pop();
  - push(5);

  at the end of last operation, total number of elements present in the stack are –

  - 4
  - 1
  - 2
  - 3

MCQs

- Consider Stack is implemented using the array.

```
#define MAX 10 struct STACK
{
     int arr[MAX]
     int top = -1;
 }
```

In this implementation of stack maximum value of top which cannot cause overflow will _________.

- Any Other
- 10
- 11
- 9