

Data Structures & Algorithms

Data Structure :-

* Data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.

classification of data structure :-

* There are two major classification of data structure :-

1. Linear data structure (link list, stack, queue)
2. Non-linear data structure (Tree, graph)

Algorithms :-

- * Sequence of steps to solve any given problem.
- * Algorithms have a definite beginning and a definite end.
- * A finite no. of steps.

Characteristics of data structure :-

Correctness : Data structure implementation should implement its interface correctly.

Time Complexity :- Running time or the execution time of operations of data structure must be as small as possible.

Space complexity :

Need for data structure :

Searching data

Need to manage processor speed.

Execution time cases :

Worst Case

Average Case

Best Case

Arrays

- * An array is a finite collection of similar elements stored in adjacent memory locations.
- * By finite, we mean that there are specific no. of elements in an array.
- * By similar, we mean that all the elements in an array are of same type.
e.g., any array may contain all integers or all characters but not both.
- * An array consisting of n number of elements is referenced using an index that varies from 0 to $(n-1)$.
- * The elements of $A[n]$ containing n elements are denoted by $A[0], A[1] \dots A[n-1]$,
where 0 is the lower bound.
 $n-1$ is the upper bound.
- * In general, the lowest index of an array is called its lower bound and highest index of an array is called its upper bound.
- * The number of elements in the array is called its range.

* Thus, an array is a set of pairs of an index & a value. For each index, there is a value associated with it.

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$
34	1	5	-6	12	9	2

* An array is further categorized as a one-dimensional array & multi-dimensional array.

* A multi-dimensional array can be a 2-D array, 3-D array, 4-D array etc. Whether an array is a 1-D or 2-D can be judged by the syntax used to declare the array.

$A[5]$:- A 1-D array holding 5 elements.

$A[2][5]$:- A 2-D array with 2 rows & 5 columns holding 10 elements.

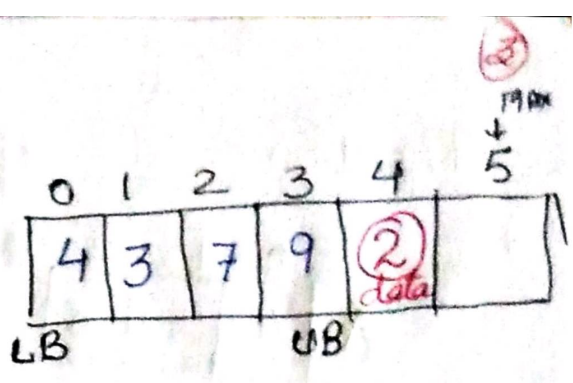
$A[2][5][3]$:- A 3-D array with two 2-D array each of which is

having 5 rows and 3 columns, thus holding totally 30 elements.

Operations on Array :-

- * Insertion
- * Deletion
- * traversal
- * search

Insertion at End :-



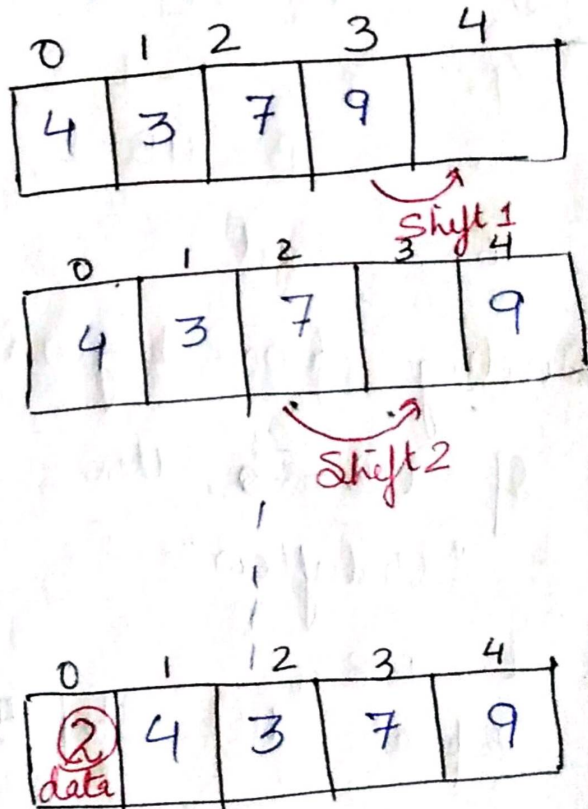
let A be an array of size N.

1. If $UB = N - 1$ then "overflow".
2. Read data or value to be inserted.
3. Set $UB = UB + 1$
4. $A[UB] = \text{data}$
5. Stop or Exit

Insertion at Beginning :-

let A be an array of size N.

1. If $UB = N - 1$ then "overflow"
2. Else Read data
3. ~~UB~~ $K = UB$
4. Repeat step 3 while $K \geq LB$
5. $A[K+1] = A[K]$
6. $K = K - 1$
7. $A[LB] = \text{data}$
8. Stop



Insertion at specific location :-

	0	1	2	3	4	5	6	7	8	9	10	11
A	4	10	6	16	21	2	5	1				

↑
data = 100
to be inserted

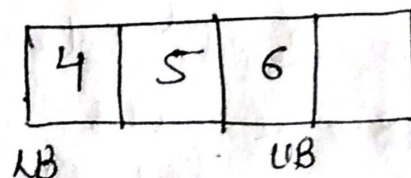
MAX = 11
UB = 7
loc = 4
data = 100

7 to 4 shifted, 8 to 5
UB to loc (UB+1) to (loc+1)

1. Initialize a counter $I = \text{MAX} - \text{UB}$
2. Repeat ^{step 4} while $(I \geq \text{loc})$
3. $A[I+1] = A[I]$
4. $I = I - 1$
5. $\text{UB} = \text{UB} + 1, A[\text{loc}] = \text{data}$
6. Exit

Deleting from beginning

1. If $\text{UB} = -1$, then
"Underflow"



2. $K = \text{LB}$
 3. Repeat till Step 5 while $K < \text{UB}$
 4. $A[K] = A[K+1]$
 5. $K = K + 1$
 6. $A[\text{UB}] = \text{NULL}$
 7. $\text{UB} = \text{UB} - 1$
0. Exit

Deletion of a particular element :-

A[0]	A[1]	A[2]	A[3]	A[4]	
4	6	10	2	12	

1. If $UB = \dots - 1$, "Underflow" ^{LB}
2. Read data
3. $k = LB$
4. Repeat step (5) while $A[k] \neq \text{data}$
5. $k = k + 1$
6. Repeat step (8) while $k < UB$
7. $A[k] = A[k+1]$
8. $k = k + 1$

Representation of 2-D arrays in memory :-

- * A 2-D array is a collection of elements placed in m rows & n columns.
- * The syntax used to declare a 2-D array includes two subscripts, of which one specifies the no. of rows & the other specifies the no. of columns of an array.
- * e.g; $A[3][4]$ is a 2-D array containing 3 rows & 4 columns. & $A[0][2]$ is an element placed at 0th row & 2nd column.

* 2-D array is also called a matrix

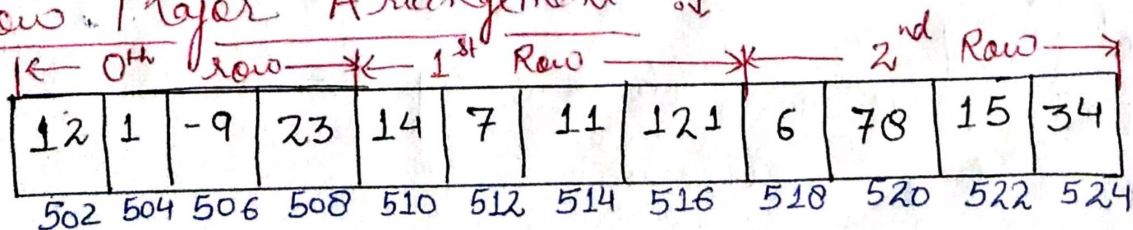
	0	1	2	3
0	12	1	-9	23
1	14	7	11	121
2	6	78	15	34

Row Major & column Major Arrangement:

- * Row & columns of a matrix are only a matter of imagination. when a matrix gets stored in memory all elements of it are stored linearly.
- * Since computer's memory can only be viewed as consecutive units of memory.
- * This leads to two possible arrangements
 - Row-Major arrangements
 - column-Major arrangements

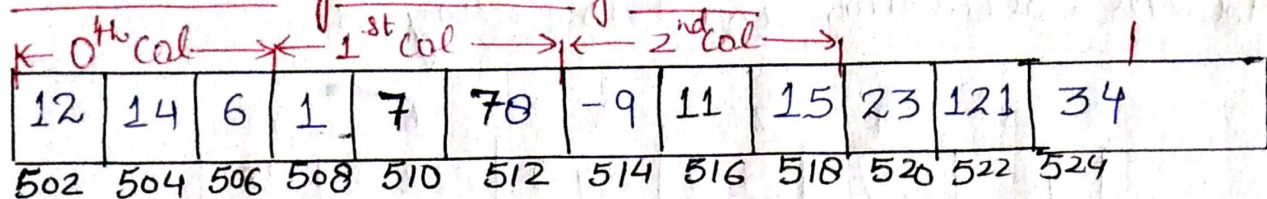
$$\text{int } A[3][4] = \left\{ \begin{array}{l} \{12, 1, -9, 23\} \\ \{14, 7, 11, 121\} \\ \{6, 78, 15, 34\} \end{array} \right\}$$

Row Major Arrangement :-



+ UB = UB - 1

Column-Major Arrangement :-



* Since the array elements are stored in adjacent memory locations, we can access any element of the array once we know the base address of the array & no. of rows & columns present in the array.

* e.g., if base address of the array ~~is~~ is 502 & we wish to refer the element 121, then the calculation involved as follows:-

Row-Major Arrangement :-

* Element 121 is present at $A[1][3]$.

Hence location of 121 would be
 $= 502 + 1 * 4 + 3 = 516$

In general, for an array $A[m][n]$ the address of element $A[i][j]$ would be

$$\boxed{\text{Base address} + i * n + j}$$

||ly. for column major arrangement

$$\boxed{\text{Base address} + j * m + i}$$

Address Calculation in 1-D array :-

0	1	2	3	4	5
15	7	12	48	82	25
1100					

$$\text{address of } A[i] = B + w * (i - LB)$$

$B \rightarrow$ Base address of array

$w \rightarrow$ Storage size of one element

$i \rightarrow$ Subscript of element whose address is to be found

$LB \rightarrow$ Lower Bound of subscript, if not specified assume 0 (zero)

Question :- Given base address of an array $B[1300 \dots 1900]$ as 1020 & size of each element is 2 bytes. Find address of $B[1700]$

the given values are $B = 1020$, $w = 2$,
 $LB = 1300$, $i = 1700$

Ans :- 1020

8

$$\text{Address of } A[i][j] = B + w * [N * (I - L_r) + (J - L_c)]$$

{ Row Major order }

$$\text{Address of } A[i][j] = B + w * [(I - L_r) + M * (J - L_c)]$$

{ Column Major order }

where,

$B \rightarrow$ Base Address

$I \rightarrow$ Row Subscript of element whose add. is to be found.

$J \rightarrow$ Column " " " " " " " " " " " "

$w \rightarrow$ Storage Size of one element

$L_r \rightarrow$ lower limit / start row index, if not given assume zero.

$L_c \rightarrow$ lower limit / start column index, if not given assume zero

$M \rightarrow$ No. of rows of given matrix

$N \rightarrow$ No. of columns " " " "

Q:-

Note:- Usually no. of rows & columns of a matrix are given like $A[20][30]$ etc. but if not given as $A[L_r \dots U_r][L_c \dots U_c]$ then the no. of rows & columns are calculated using :

$$\text{No. of rows (M)} = (U_r - L_r) + 1$$

$$\text{No. of cols (N)} = (U_c - L_c) + 1$$

Examples :- $X[-15 \dots 10, 15 \dots 40]$ requires one byte of storage. If beginning location is 1500, Determine the location of $X[15][20]$.

Solution :- $M = [10 - (-15)] + 1 = 26$

$$N = [40 - 15] + 1 = 26$$

(i) Column Major Calculation :-

$$B = 1500, W = 1, I = 15, J = 20, L_r = -15, L_c = 15, M = 26$$

$$\text{Address of } A[I][J] = B + W * [(I - L_r) + M * (J - L_c)]$$

$$= 1500 + 1 * [(15 - (-15)) + 26 * (20 - 15)]$$

$$= 1500 + 1 * [30 + 26 * 5]$$

$$= 1500 + 1 * [160]$$

$$= 1660$$

Stacks :-

- * The linear data structures as array and a linked list allow us to insert & delete an element at any place in the list.
- * However, sometimes it is required to permit the addition or deletion of elements only at one end. that is either beginning or end.
- * Stack and Queues are two types of data structures in which addition or deletion of an element is done at end, rather than middle.
- * A stack is a data structure in which addition of a new element or deletion of an existing element always takes place at the same end.
- * This end is often known as top of stack.
- * e.g., A stack of plates in cafeteria in which every new plate is added to the stack is added at top.
- * Similarly, every new plate taken off the stack is also from top.

- When an item is added to the stack, the operation is called push and when an item is removed from the stack, the operation is called pop.
- Stack is also called as last-in-first-out (LIFO) list.

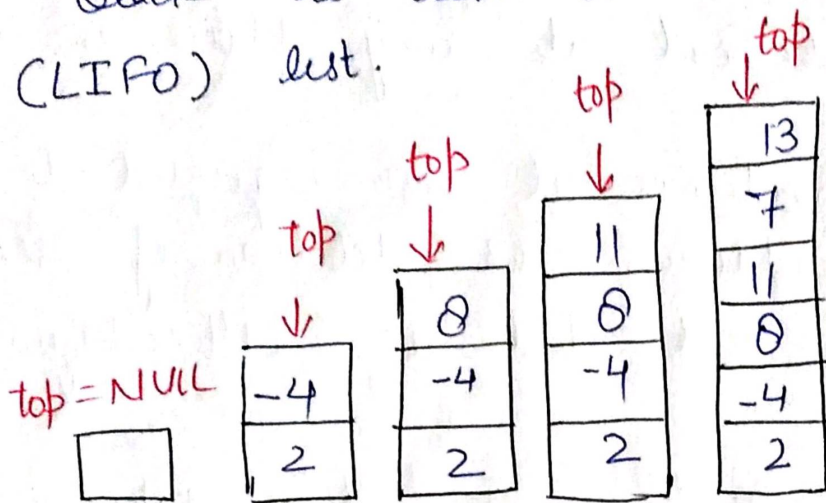


fig :- Representation of stack after inserting elements

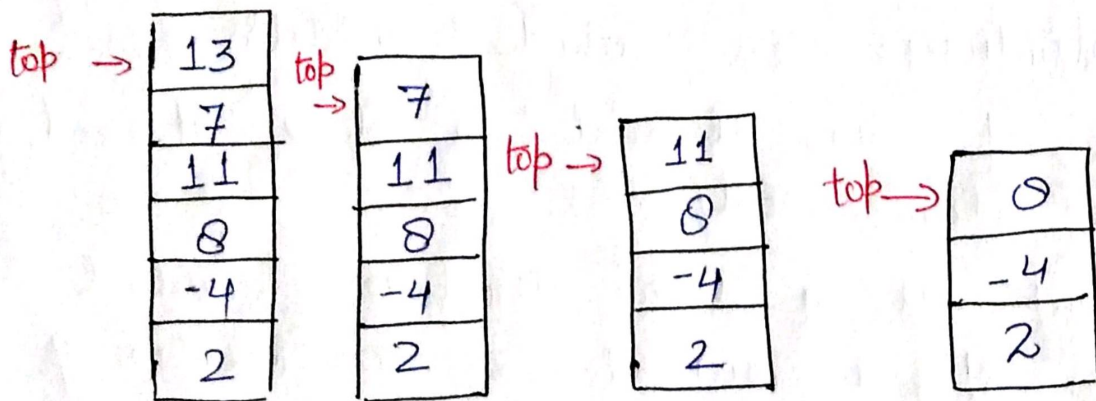


fig :- Representation of stack after deletion

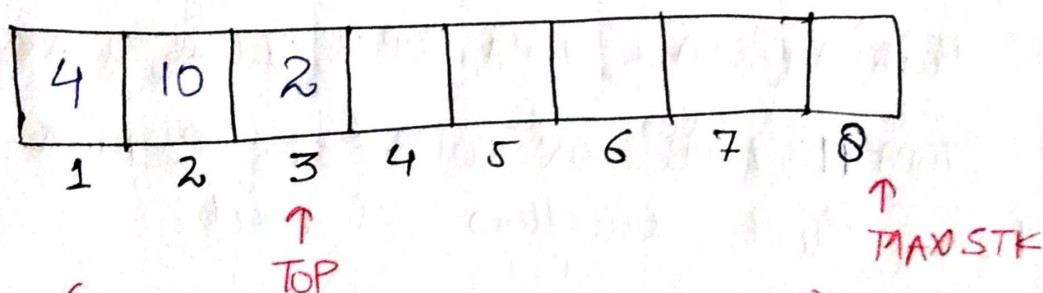
Operations on Stack :-

Stack is generally implemented with two basic operations

- * Push
- * Pop

Representation of Stack as Array :-

- * Stack contain an ordered collection of elements. An array is used to store ordered list of elements. Hence it would be very easy to manage a stack if we represent it using an array.
- * Each of our stacks will be maintained by
 - a linear array STACK → a pointer variable TOP, which contains the location of the top element of the stack.
 - A variable MAXSTK which gives the maximum no. of elements that can be held by the stack.
 - The condition $TOP = 0$ or $TOP = NULL$ will indicate that the stack is empty.



PUSH (STACK, TOP, MAXSTK, ITEM)

(This algo. pushes an item onto a stack)

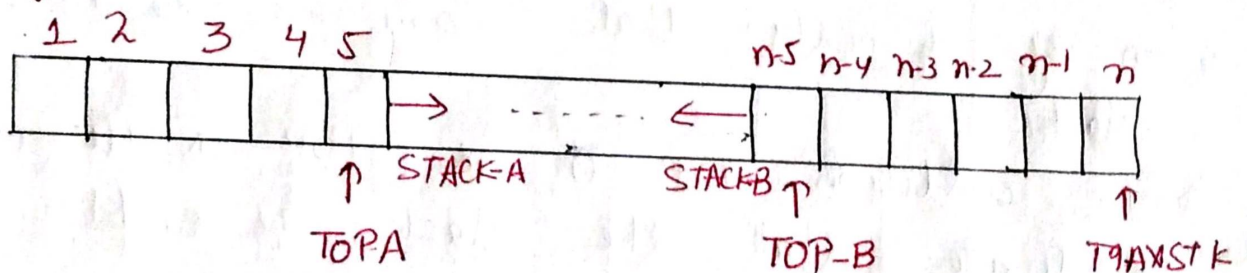
1. If $TOP = MAXSTK$ then Print Overflow & Return [stack already filled]
2. Set $TOP = TOP + 1$ [Increase Top by 1]
3. Set $STACK[TOP] = ITEM$ [Insert item]
4. Return.

POP (STACK, TOP, ITEM)

This algo. deletes the top element of STACK & assigns it to the variable ITEM

1. If $TOP = 0$ then Print UNDERFLOW & Return
[Stack has an item to be removed?]
2. Set $ITEM = STACK[TOP]$. [Assigns TOP element to ITEM]
3. Set $TOP = TOP - 1$ [decreases TOP by 1]
4. Return

Multiple Stack Implementation using Single Array :-



Algo. Push-A (STACK, TOPA, TOP B, ITEM, MAXSTK)

1. If $(TOPA \neq 1 = TOPB \text{ OR } TOPB = 1 \text{ OR } TOPA = MAXSTK)$ then Print "Overflow" & Exit.
2. $TOPA = TOPA + 1$
3. $STACK[TOPA] = ITEM$
4. Exit

9

Algo Push B (STACK, TOPA, TOPB, Item, MAXSTK)

1. if (TOPA + 1 = TOPB OR TOPA = MAXSTK OR TOPB = 1)
Print overflow & exit
2. if (TOPB = 0) then TOPB = MAXSTK
3. Else
STACK
TOPB = TOPB - 1
4. Set $[TOPB] = Item$
5. Exit

Algo POPA (STACK, TOPA, Item)

1. if (TOPA = 0) then Print "Underflow" Exit.
2. Else item = STACK [TOPA]
3. TOPA = TOPA - 1
4. Exit

Algo POP B (STACK, TOPB, Item, MAXSTK)

1. if TOPB = 0 Print "Underflow" & Exit
2. Else { Item = STACK [TOPB]
3. if (~~top~~ TOPB = MAXSTK) then TOPB = 0 }
4. Else TOPB = TOPB + 1
5. Exit

Stack Applications

① Reversing a String

E	X	T	R	E	M	E
1	2	3	4	5	6	7

Push E	E	← TOP
Push M	M	
Push E	E	
Push R	R	
Push T	T	
Push X	X	
Push E	E	

- Algo :-
1. Traverse the list and push all its values onto a stack.
 2. Traverse the list from TOP element & POP a value & connect them in reverse order.

② Factorial Calculation :-

Recursion :- When a function calls itself.

<pre>main() { fun(); } fun() { fun(); }</pre>	} direct recursion	<pre>main() { a(); } a() { b(); } b() { a(); }</pre>	} Indirect recursion
---	--------------------	--	----------------------

e.g. factorial of a number

$$4! = 4 * 3! \quad \left. \vphantom{4!} \right\} \text{Recursive condition } [n * (n-1)!]$$

$$\downarrow$$

$$3! = 2!$$

$$\downarrow$$

$$2 = 1!$$

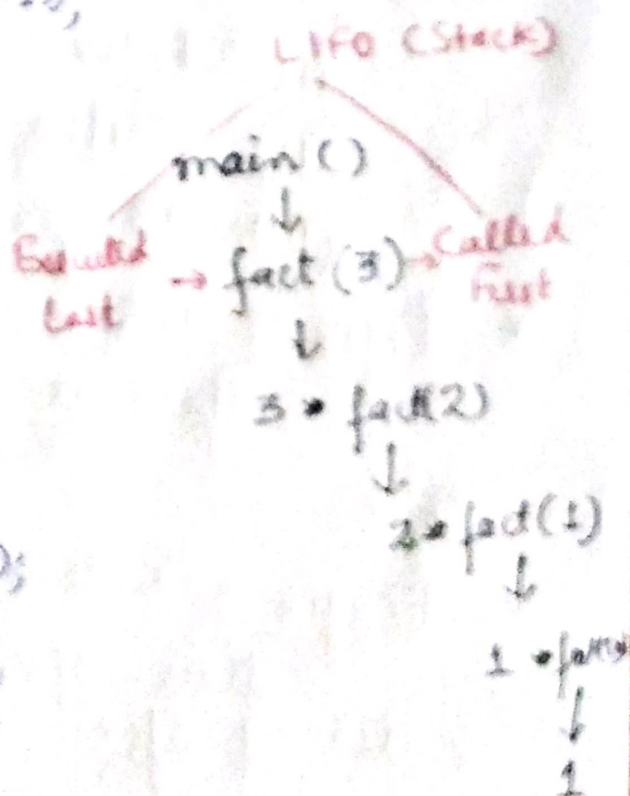
$$\downarrow$$

$$1 = 0!$$

\downarrow
1 } Base Condition /
Stop condition

```
# include <stdio.h>
int fact (int n) {
    if (n == 0)
        return 1;
    else
        return (n * fact (n-1));
}
```

```
void main()
{
    int num, result;
    printf ("Enter a no.");
    scanf ("%d", & num);
    result = fact (num);
    printf ("%d", result);
}
```



Polish Notation

* The process of writing the operators of an expression either before their operands or after them is called Polish Notation.

* The fundamental property is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expressions.

* One never needs parenthesis when writing expression in Polish Notation. These are classified in three categories:

- (i) Infix
- (ii) Prefix
- (iii) Postfix

* Infix :- When operators exist between two operands, the expression is called

Infix.

* Prefix :- When operators are written before their operands, the expression is called prefix polish notation.

➤ Postfix :- When the operators come after their operands the resulting expression is called the reverse polish notation or Postfix notation. (11)

e.g., $z + (y * x - (w / v * u) * t) * s$

By Inspection & hand :-

$$\Rightarrow z + (y * x - (w / v * u) * t) * s \quad \textcircled{1}$$

$$\Rightarrow z + (y * x - w * v * u / t) * s \quad \textcircled{2}$$

$$\Rightarrow z + (y * x * w * v * u / t) * s \quad \textcircled{3}$$

$$\Rightarrow z + y * x * w * v * u / t * - * s \quad \textcircled{4}$$

$$\Rightarrow z + y * x * w * v * u / t * - * s \quad \textcircled{5}$$

$$\Rightarrow z * y * x * w * v * u / t * - * s * + \quad \textcircled{6}$$

Stack Implementation of Infix to Postfix

Label No.	Symbol Scanned	Stack	Expression
1.	z	(z
2.	+	(+	z
3.	((+ (z
4.	y	(+ (zy
5.	*	(+ (*	zy
6.	x	(+ (*	zyx
7.	-	(+ (-	zyx*
8.	((+ (- (zyx*
9.	w	(+ (- (zyx* w
10.	/	(+ (- (/	zyx* w
11.	v	(+ (- (/	zyx* w v
12.	↑	(+ (- (/ ↑	zyx* w v
13.	u	(+ (- (/ ↑	zyx* w v u
14.)	(+ (-	zyx* w v u ↑ /
15.	*	(+ (- *	zyx* w v u ↑ /
16.	t	(+ (- *	zyx* w v u ↑ / t
17.)	(+	zyx* w v u ↑ / t *
18.	*	(+ *	zyx* w v u ↑ / t *
19.	Δ	(+ *	zyx* w v u ↑ / t * - Δ
20.)		zyx* w v u ↑ / t * - Δ * +

Postfix Expression

Algo. for converting Infix Expression To Postfix (12)

Postfix (Q, P) :-

Q is an arithmetic expression within Infix

P is an equivalent postfix.

1. Push "(" onto stack & add ")" to the end of Q.
2. Scan Q from L to R & repeat till ⑥ for each element of Q until the stack is empty.
3. If an operand is encountered, push it onto P ~~stack~~.
4. If a left parenthesis is encountered, push it onto stack.
5. If an operator is encountered then
 - (a) Add it to stack.
 - (b) Repeatedly Pop from Stack & add P each operator which has the same precedence or higher than operator.
6. If the right parenthesis is encountered :-
 - (a) Repeatedly Pop from Stack & add it to P. ~~(b)~~ each operator until a left parenthesis is encountered.
 - (b) Remove left parenthesis.
7. Exit.

Evaluating the Postfix Expression

Example :- P :- 5, 6, 2, +, *, 12, 4, /, -,)
 Symbol Scanned Stack Sentinel

5	5
6	5, 6
2	5, 6, 2
+	5, 8
*	40
12	40, 12
4	40, 12, 4
/	40, 3
-	37

Algorithm :-

→ This algo. finds the value of an arithmetic expression P written in postfix notation

→ This algo uses Stack to hold operands to evaluate P.

→ ① Add a right parenthesis ")" at the end of P.

② Scan P from L to R & repeat steps 3 & 4.

~~③ If an operand is encountered until "~~

for each element of P until ")" is encountered.

③ If an operand is encountered, put it on Stack.

④ If an operator is encountered, then

(A) Remove the top two elements of Stack, where A is the top element & B is the next-to-top element.

(15)

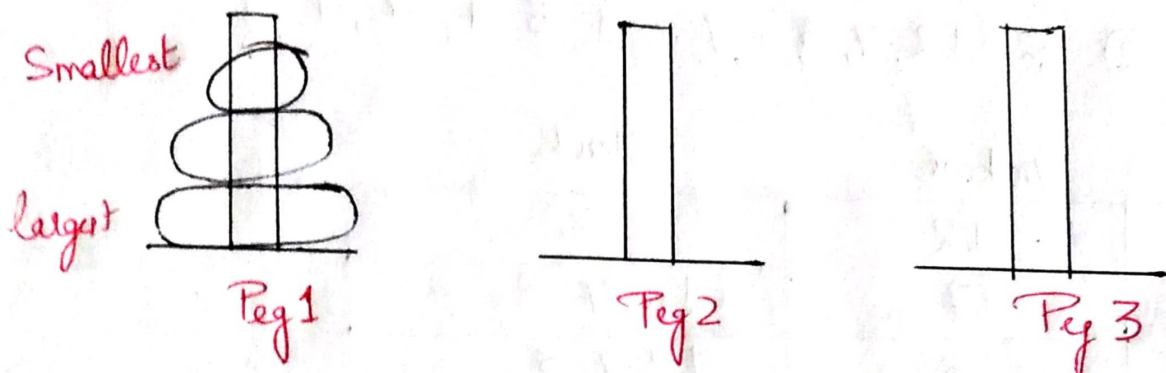
- (B) Evaluate $B \otimes A$.
- (c) Place the result of (B) on the stack
[End of step ② loop]
- ⑤ Set value equal to the top element on stack
- ⑥ Exit.

Example ② :- 12, 7, 3, -, 1, 2, 1, 5, +, *, +, } Sentinel

Symbol	Stack
12	12
7	12, 7
3	12, 7, 3
-	12, 4
1	3
2	3, 2
1	3, 2, 1
5	3, 2, 1, 5
+	3, 2, 6
*	3, 12
+	15

Tower Of Hanoi :-

* Tower of Hanoi is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted -



* These Rings are of different sizes & stacked upon in an ascending order i.e., smaller one is placed over the larger one.

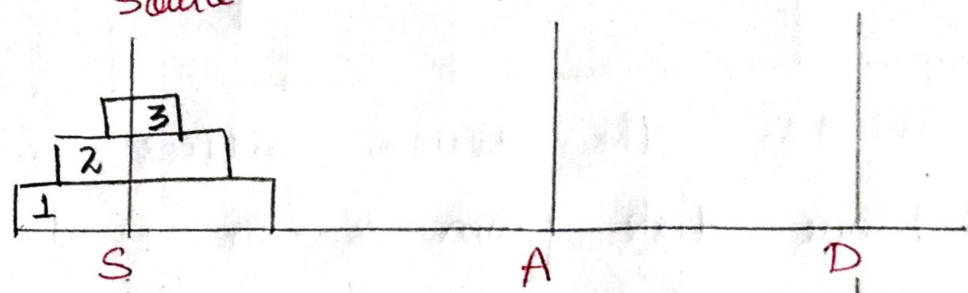
Note :- There are other variations of the puzzle where the no. of disks increase, but the tower count remain the same.

Rules :- The mission is to move all the disks to some tower without violating the sequence of arrangement. A few rules are:-

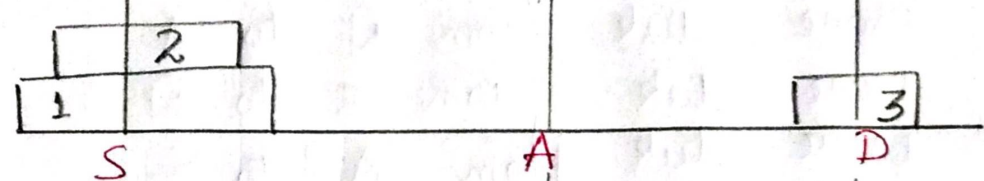
- Only one disk can be moved among the towers at any given time.
- Only the top disk can be removed.
- No large disk can be placed over the small disk.

Tower of Hanoi puzzle with 3 disks
Source Aux. Destination

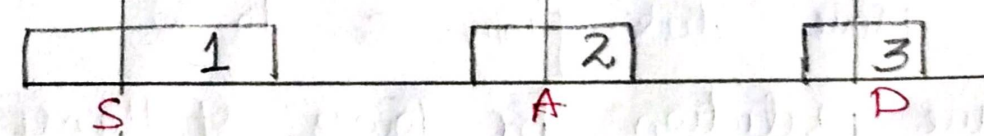
Problem



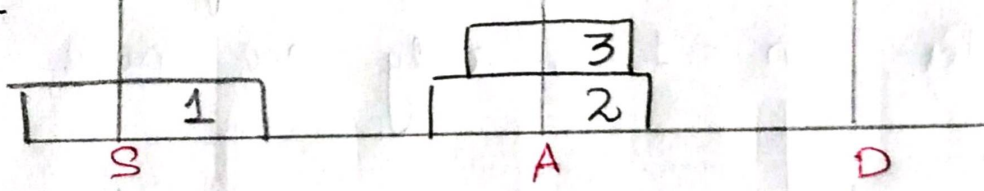
Step 1 :-



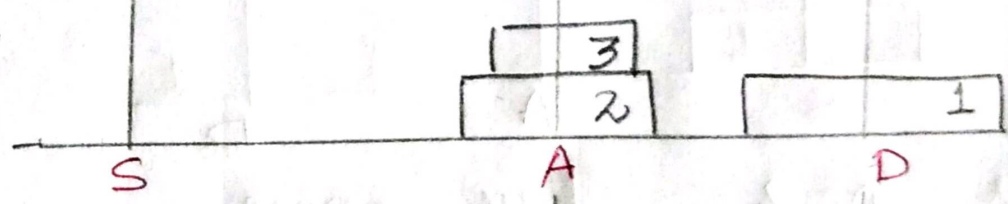
Step 2 :-



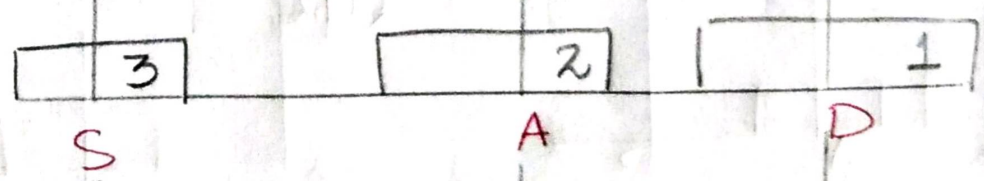
Step 3 :-



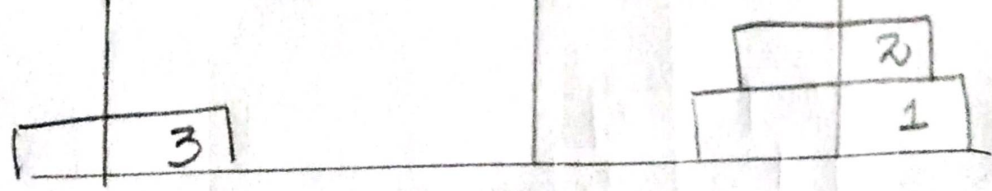
Step 4 :-



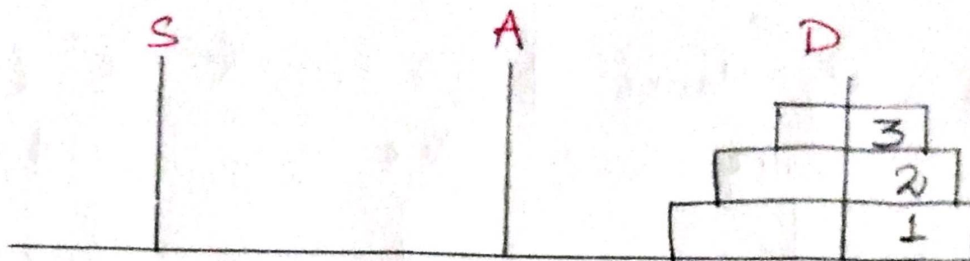
Step 5 :-



Step 6 :-



Step 7 :-

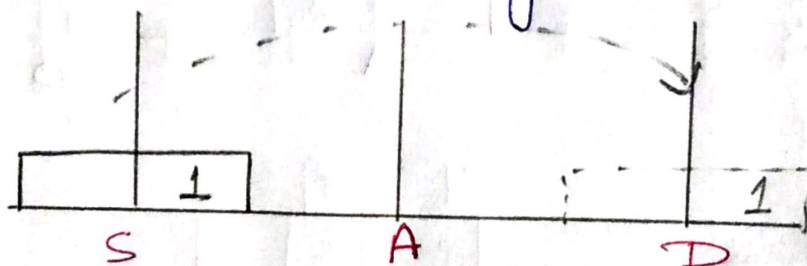


Thus for $n=3$:- the seven moves will be;

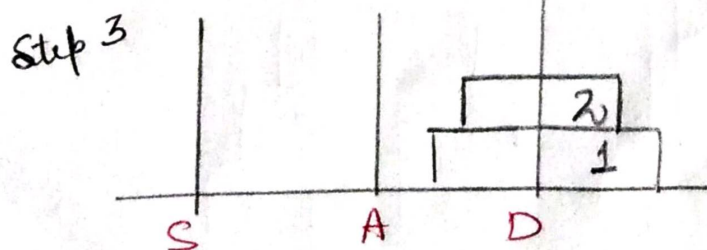
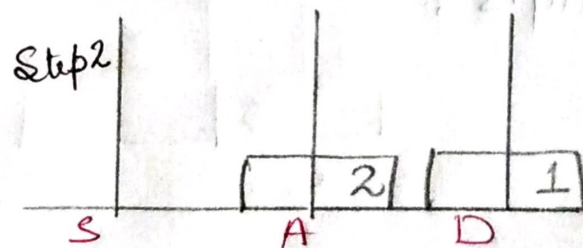
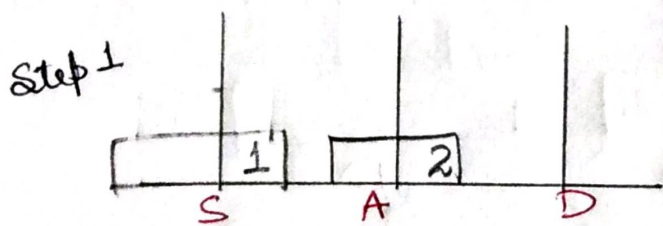
- Move disk from S to D
- Move disk from S to A
- Move disk from D to A
- Move disk from S to D
- Move disk from A to S
- Move disk from A to D
- Move disk from S to D

Recursive solution for Tower of Hanoi Problem:-

* For $n=1$, only one move



* For $n=2$, three moves



(15)

* Rather than finding a separate solution for each value of n , we'll use the technique of recursion to develop a general solution.

* First we observe that the solution to the Towers of Hanoi problem for $n > 1$ disks may be reduced to the following subproblems:-

- 1) Move top $(n-1)$ disks from S to A.
- 2) Move top disk from S to D.
- 3) Move top $(n-1)$ disks from A to D.

TOWER (N, S, A, D)

- 1) if $N = 1$ then
print ("move disk from S to D").
- 2) Else TOWER (N-1, S, A, D)
- 3) print ("move disk from S to D")
- 4) TOWER (N-1, A, S, D)

TOWER (4, S, A, D)

