Estd. in 2000
Excellence in Education

ARYA GROUP OF COLLEGES

AIET ☐ ACERC ☐ AIETM ☐ ACP ☐ APGC ☐ LECTURE NOTES

Branch / Faculty: Pradip Kumar Year / Sem.: III Subject: DSA

Topic: Unit: Unit-1 Lecture No.

Data structure & algorithm :-

A data structure is a way of organizing & storing data in a computer so that it can be accessed & used efficiently. It refers to the logical or mathmatical representation of data, as well as the implementation in a computer program.

Classification

Data structure can be classified into two broad categories.

Linear Data Structure :- A data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous & next adjacement element is called a linear data structure.

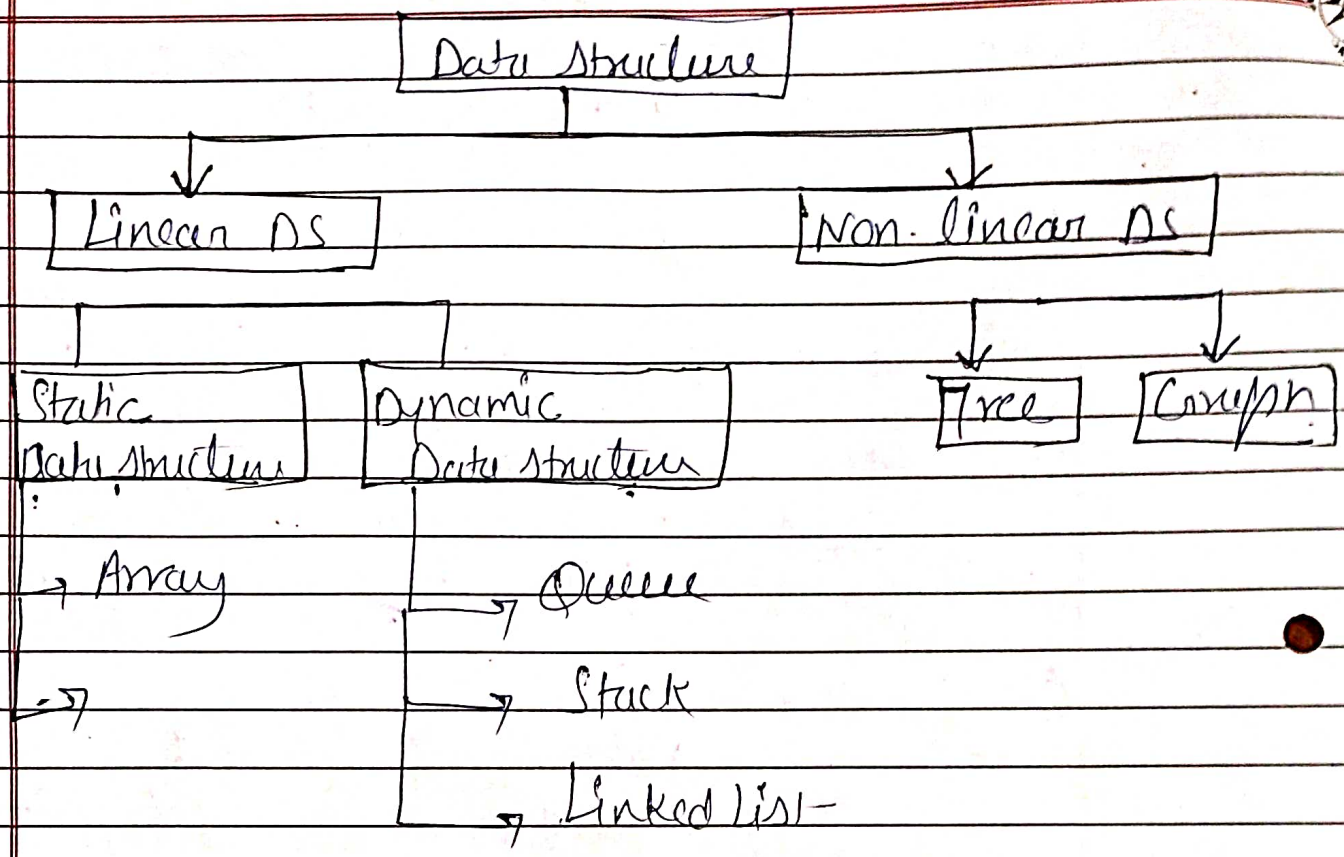
Example :- Array, Stack, Queue etc.

Non - linear Data structure :-

Data structure where data elements are not placed sequentially or linearly are called non-linear data structure.

Example :- Tree, Graph.

Name of Lecturer: .....



Application of Data Structure :-

⇒ **Database :-** Data structure are used to organize & store data in a database allowing for efficient retrival & manipulat

Operating System :-

Data Structure are used in the design & implementation of operating system to manage system resource such as memory & files.

Name of Lecturer: .....

Computer Graphics :-

Data structure are used to represent geometric shapes and other graphical elements in computer graphics applications

Artificial intelligence :-

Data structures are used to represent knowledge & information in AI system

Advantage of Data Structure :-

⇒ Efficiency :-

Data structures allow for efficient storage and retrival of data which is important in application where performance is critical

⇒ Flexibility :-

Data structure provide a flexible way to organize & store data allowing for easy modification & manipulation.

⇒ Reusability :-

Data structure can be used in multiple programs and application reducing the need for redundant code

Maintainability :-
Well designed data structures can make programs easier to understand, modify and maintain over time.

Data structure :-
is a way of storing and organizing data efficiently such that the required operations on them can be performed be efficient with respect to time as well as memory

Data structures are used to reduce complexity (most of the time complexity) of the code

Static data structure :-
In static data structure the size of the structure is fixed. The content of data structure can be modified but without changing the m/o space allocated to it.

for exam

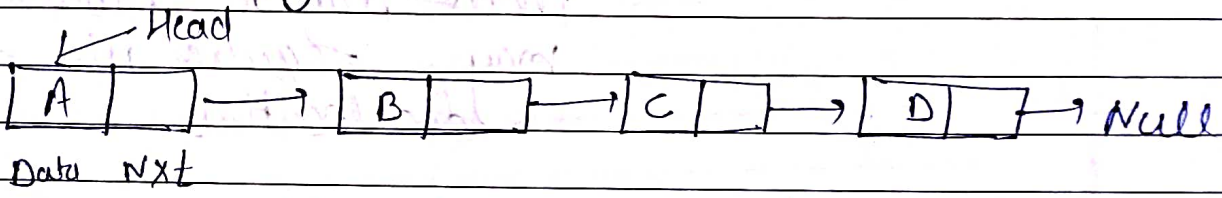
42	33	48	90	45	96	15
0	1	2	3	4	5	6

← Array indices

Array length = 7
first index = 0
Last index = 07

Dynamic Data Structure :-

In dynamic data structure the size of the structure is not fixed & can be modified during the operations performed on it. Dynamic data structure are designed to facilitate change of data structure in the run time



	Static Data Structure	Dynamic Data Structure
(1)	Static data structure such as arrays, have a fixed size & are allocated at compile time.	Dynamic data structure on the other hand, have a variable size and are allocated at run-time.
	This means that their m/o size cannot be changed during program execution.	This means that their m/o size can be changed during program execution
	Index based access to elements is fast & efficient since the address of the element is known.	Index based access to element is fast & efficient since the address of the

Name of Lecturer : .....

(6)

H/o can be dynamically allocated or deallocated during program execution.

Due to this dynamic nature accessing elements based on index may be slower as it may require H/o allocation/deallocation.

Memory utilization may be inefficient	H/o utilization is efficient as H/o can be reused
Access time is faster as it is fixed	Access time may be slower due to indexing & pointer usage
Array, stacks, queue (with fixed size)	Lists, trees (with variable size) Hash table

Operations on the Data structure :-

1. Traversing
2. Insertion
3. Deletion
4. Searching
5. Sorting
6. Merging

Traversing :-

Traversing means accessing each record exactly once, so that each record may be processed

Insertion :-

Adding a new data into the list any where at the specified location is called insertion.

Deletion :-

Removing the existing data from the list is called deletion

Searching :-

It is process of finding an element & its location in a collection of elements

Sorting :-

Sorting can be defined as rearranging the elements into a desired sequence either in increasing or decreasing order.

Merging :-

Combining two different records into a single one.

Characteristics of data structure :-

1. Organization of data
2. Accessing methods
3. Degree of associativity
4. Processing alternative for information

Name of Lecturer: .....

Definition of Algorithm :-

An algorithm is a step by step process to solve a particular problem

Characteristics of an algorithm :-

→ An algorithm should be made up of finite no. of statement

→ There should be proper termination of the algorithm

→ The effectiveness of the statements should be maintained

→ The statement of the algorithm should be unambiguous.

→ The algorithm should be simple & precise sized.

Structure of algorithm

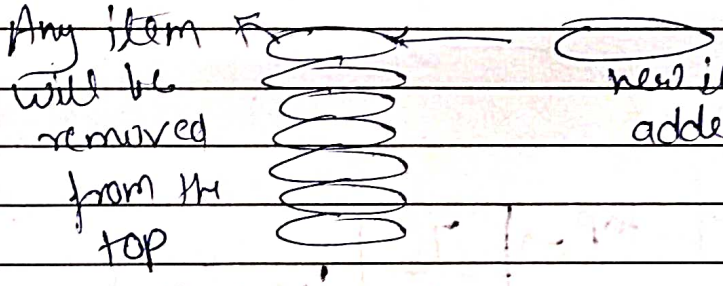
1. Name of the algorithm.
2. Brief description of algorithm.
3. Steps or statements
4. Comment.

Name of Lecturer: .....

Stack :-

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last in first out).

There are many real life examples of a Stack Consider an example of plates stacked over one another in the canteen



Example :- A stack of dishes

Definition :- A stack of is a list of elements in which an element may be inserted or deleted only at one end called the Top of the Stack.

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.

Name of Lecturer: .....

⇒ Stack is also called as last in first out (LIFO) list.

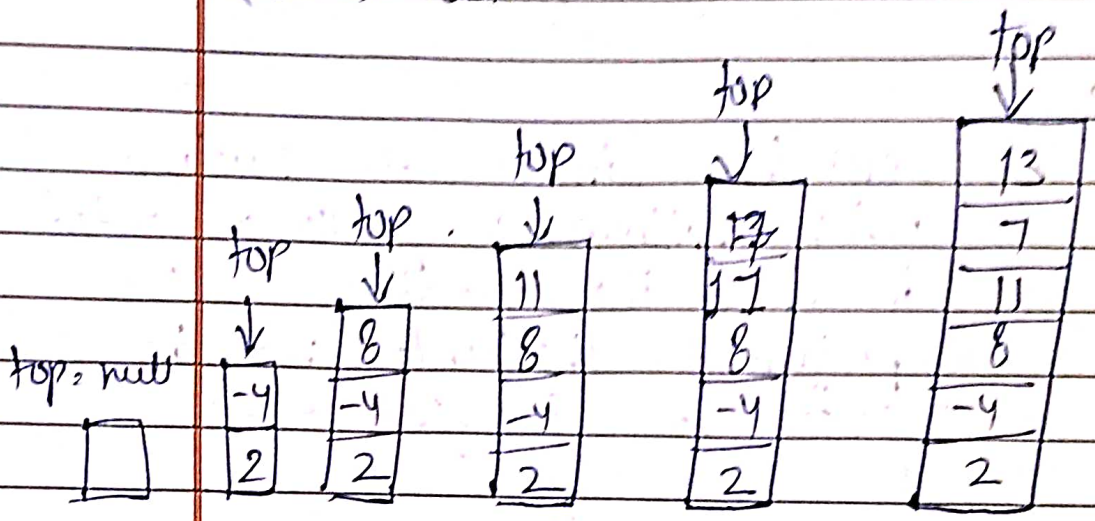


fig:- Representation of stack after inserting elements

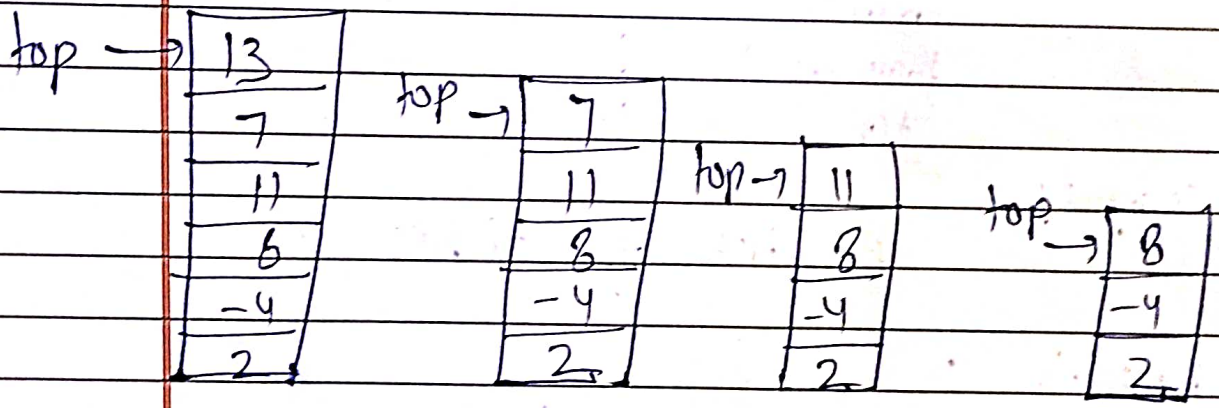


fig:- Representation of stack after deletion

Operations on Stack:- Stack is generally implemented with two basic operations

(1) PUSH (2) POP

Name of Lecturer: .....

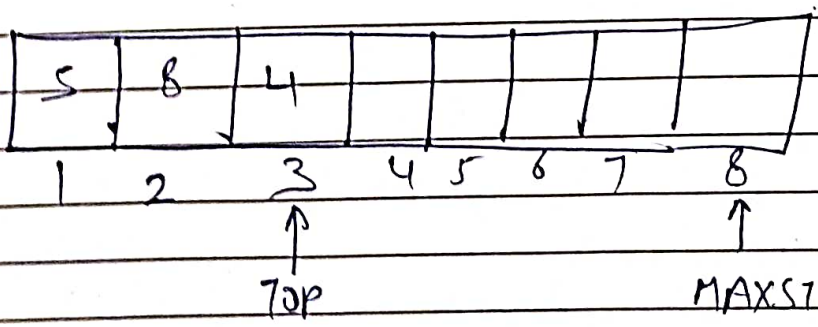
Representation of Stack as array :-

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 max^m no. of elements that can be held by the stack.

⇒ The condition TOP = 0 or TOP = Null will indicate that the stack is empty



Name of Lecturer: .....

Algo: This algo pushes an item onto a stack.

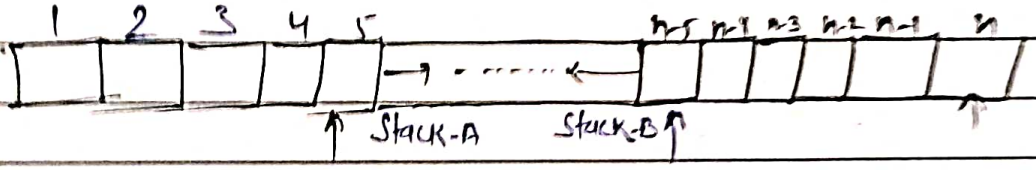
PUSH (STACK, TOP, MAXSTK, ITEM)

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]

Algo: This algo. deletes the top element of STACK assign 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] [Assign top element item]
3. Set TOP = TOP-1 [decrease TOP by 1]
4. Return

Multiple Stack Implementation using Single array:-



Algo PUSH-A (STACK, TOPA, TOPB, Item, MAXSTK)

1. if (TOPA ≠ TOPB OR TOPB == 1 OR TOPA = MAXSTK
   then print "Overflow" & Exit.
2. TOPA = TOPA + 1
3. STACK[TOPA] = Item
4. Exit.

Name of Lecturer: .....

Algo PUSHB (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 TOPB = TOPB - 1
4. Set stack [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, TOP B, Item, MAXSTK)

1. If TOP B = 0 print "underflow" & Exit
2. else
   { Item = STACK [TOP B]
3. If (TOP B = MAXSTK) then TOP B = 0 }
4. Else
   TOP B = TOP B + 1
5. Exit.

Name of Lecturer: .....

Stack Implementation :-

① Reversing a String.

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

E	PUSH E
M	PUSH M
E	PUSH E
R	PUSH R
T	PUSH T
X	PUSH X
E	PUSH 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.

Name of Lecturer: .....

(2) factorial Calculation :-

Recursion :- when a function call itself

Direct recursion

```
main()
{
  func();
}

func()
{
  func();
}
```

Indirect recursion

```
Main()
{
  a();
}

a()
{
  b();
}

b()
{
  a();
}
```

Example :-

factorial of a number

4! = 4x3! } Recursion condition (n*(n-1)!)

↓

3x2!

↓

2x1!

↓

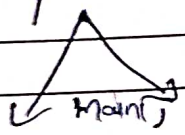
1 * 0!

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);
}
```

Name of Lecturer: .....

Lifo Stack



Executed last ← fact(3) ← first called.

↓

3 × fact(2)

↓

2 × fact(1)

↓

1 × fact(0)

↓

1

Name of Lecturer: .....

push() → pushing (storing) an elements on the stack.

POP() → Removing (accessing) an elements from the stack.

When data is pushed onto stack functions are performed.

1. peek() :- get the top data element of stack, without removing it
2. isfull() :- check if stack is full
3. isempty() :- check if stack is empty

peek() :

algo :-

```
begin procedure peek
return stack[top]
end procedure
```

Example :-

```
int peek()
{
return stack[top];
}
```

isfull()

{

begin procedure isfull

If top equal to maxsize

return true

else

return false

endif

end procedure

}

Examples bool isfull()

{ If (top == maxsize)

return true;

else

return false;

}

Name of Lecturer: .....

isempty()

```
begin procedure isempty
{
    if top lessthan 1
        return true
    else
        return false
    end if
    end procedure
}
```

Example

```
bool isempty()
{
    if (top == -1)
        return true;
    else
        return false;
}
```

Application of Stack in data Structure:-

1. Evaluation of arithmatic expression
2. Backtracking
3. Delimiter checking
4. Reverse data
5. Processing function calls
6. Reversing list
7. factorial calculation

The predence rules for five basic operation are :-

operators	Associativity	Precedence
^ Exponentiation	Right to left	Highest followed by * /
* /	left to right	Highest followed by + -
+ -	left to right	Lowest

Name of Lecturer: .....

Array :-

Array is linear collection of homogenous data items.

Linear array / one dimensional array :-

One dimensional array is one, in which only one subscript is required to specify a particular element of the array.

```
int a[5];
```

⇒ **Length of An Array :-**

Length = (upperlimit − lowerlimit) + 1

for exa l = (5 − 0) + 1
= 4 + 1 = 5

⇒ **Size in bytes :-**

= length of array * Size of datatype

5 × 2 = 10 bytes

⇒ **Calculation of address for one dimension array :-**

Address of n element = Base address + Size of data type * (n − 1)

Address calculation in 1-D array :-

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

1100

Address of A[i] = B + w * (i - LB)

Where :-

B = Base address of array
w = storage size of one element
I = Subscript of element whose address is to be found
LB = lower bound of subscript. If not specified assume 0 (zero)

Ques :- Given Base address of an array B[1300-1900] as 1020 & size of each element is 2 bytes. find address of B[1700]

solution :- the given values are

B = 1020
w = 2
LB = 1300
I = 1700

find A[i] = B + w * (i - LB)

A[1700] = 1020 + 2 * (1700 - 1300)
= 1020 + 2 X 400
= 1020 + 800 = 1820 Ans

Name of Lecturer: .....

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 include two subscripts of which one specifies the no. of rows & the other specifies the no. columns of an array.

Ex:- 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

	0	1	2	3
0	12	14	-2	13
1	11	9	7	2
2	10	24	8	4

Row Major & Column major Arrangements:-

=> Rows & columns of a matrix are only a matter of imagination. When a matrix gets stored in m/o all elements of it are stored linearly.

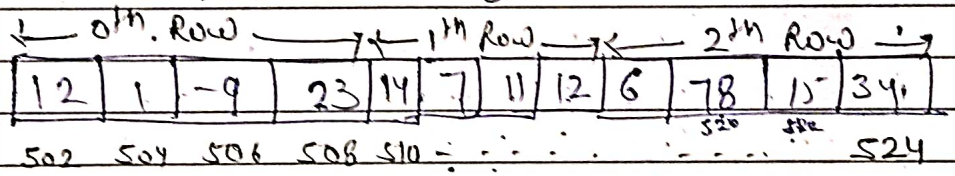
⇒ Since computer's m/o can only be viewed as consecutive units of m/o.

⇒ This leads to two possible arrangements

Row- major arrangement
Column- major arrangement

int A[3][4] = $\begin{bmatrix} 12 & 1 & -9 & 23 \\ 14 & 7 & 11 & 12 \\ 6 & 78 & 15 & 34 \end{bmatrix}$

Row- major Arrangement :-



UB = UB-1

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

Base address + i x n + j

for eg :- Base addrss = 502, A[3][4] ; m=3
element = 78 present at [2][1] n=4

= 502 + 2*4 + 3
= 502 + 8 + 3 = 513

Name of Lecturer: .....

When we find Address row major :-

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

When we find Address column major :-

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

Where :→

B = Base address

I = Row subscript of elements whose addrs is to be found.

J = Column subscript of element whose addrs is to be found.

W = Storage size of one element

L_r = lower limit / start row index, if not given assume zero

L_c = lower limit / start column index, if not given assume zero

M = No. of rows of given index

N = no. of columns of given index

Note :-

Usually no of rows & columns of a matrix are given like A[20][30] etc.

but if not given as A[Lr ... Ur] [Lc .... Uc) then the no of rows & columns are calculated using :-

No. of rows (M) = (Ur - Lr) + 1

No of column (N) = (Uc - Lc) + 1

Example X[-15 ... 10, 15 ... 40] requires one byte of storage. If beginning location is 1500, determine the location of X[15][20]

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

N = (40 - 15) + 1 = 26

Column major calculation :-

B = 1500, W = 1, I = 15, J = 20 Lr = -15

Lc = 15, M = 26

Addr of A[i][j] = B + W * [(I - Lr) + M * (J - Lc)]

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

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

Name of Lecturer: X(1.6.20) = 1660 Ans

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. There are classified in three categories.

1. Infix
2. prefix
3. post fix

Infix :- When operators exist between two operands the expression is called infix.

Name of Lecturer : .....

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.

Infix to postfix :- If we want we can convert the infix notation into the postfix notation

Why infix to postfix :- We convert the infix notation in postfix because compiler only evalute the postfix notation not infix notation.

Why compiler evaluate postfix :-

Compiler Evaluate the postfix notation because it arrange like when he process it process the sequencial but if we process the infix notation in this we required perform the jumping from lower precidence operator to higher, So performance is decrease.

Infix to postfix Conversion using Stack.

Infix Notation :-

Infix notation is the common arithmatic & logical formula notation

In which operators are written infix-style between the operator they act on.

Example :- A+B

⇒ To add A, B we write
A+B

⇒ To multiply A, B we write
A×B

⇒ The operators ('+' '*' and many more) go in b/w the operands ('A' and 'B')

⇒ This is "infix" notation.

Name of Lecturer : .....

Prefix Notation :

In prefix notation, the operator comes before the operand.

⇒ The infix expression A+B will be written as +AB in its prefix notation.

⇒ Prefix is also called Polish notation.

Ex:-

⇒ Instead of saying "A plus B" we could say "Add A, B" and write

+AB

⇒ "multiply A, B" would be written

*AB

⇒ The operator ('+', '*' and many more) go infront of the operands ('A' and 'B')

⇒ This is "prefix" notation.

Postfix Notation:

In postfix notation the operator comes after the operand.

⇒ for Example, the Infix expression A+B will be written as AB+ in Postfix Notation

⇒ Postfix is also called "Reverse Polish" notation

Algo:-

Another alternative is to put the operators after the operands as in "AB+" and "AB*"

⇒ The operators ('+' '*' and many more) go in end of the operands ('A' and 'B')

⇒ This is "postfix" notation

Priority / Precedence of Arithmatic operators:-

① Highest precedence → ( ), ↑ or ^ (Exponential)

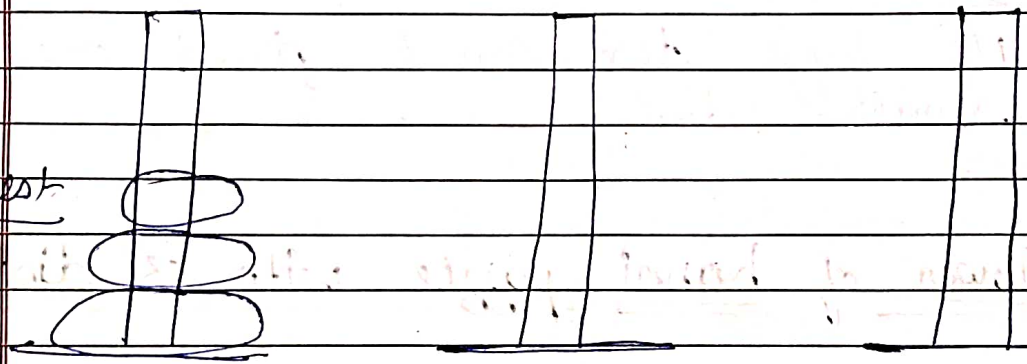
② Mid precedence → *, /

③ Lowest precedence → +, -

	Infix	Prefix	Postfix
(1)	a+b	+ab	ab+
(2)	(a+b)*c	+ab*c *+abc	ab+*c ab+c*
(3)	a*(b+c)	a*+bc *a+bc	a*bc+ abc+*
(4)	a/b+c/d	/ab+c/d +/ab/cd	ab/+c/d ab/cd/+
(5)	(a+b)*(c+d)	+ab*+cd *+ab+cd	ab+*cd+ ab+cd+*
(6)	((a+b)*c)-d	(+ab*c)-d ab+*c a+b*c *+abc-d -*+abcd	(ab+)*c)-d

Tower of Hanoi :-

Tower of hanoi is a mathmatical puzzle which consist of three towers (pegs) and more than one rings is depicted



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

Note :- These are other variations of the puzzle where the no. of disks increase but the towers count remain the same.

Name of Lecturer: .....

Rules:-

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

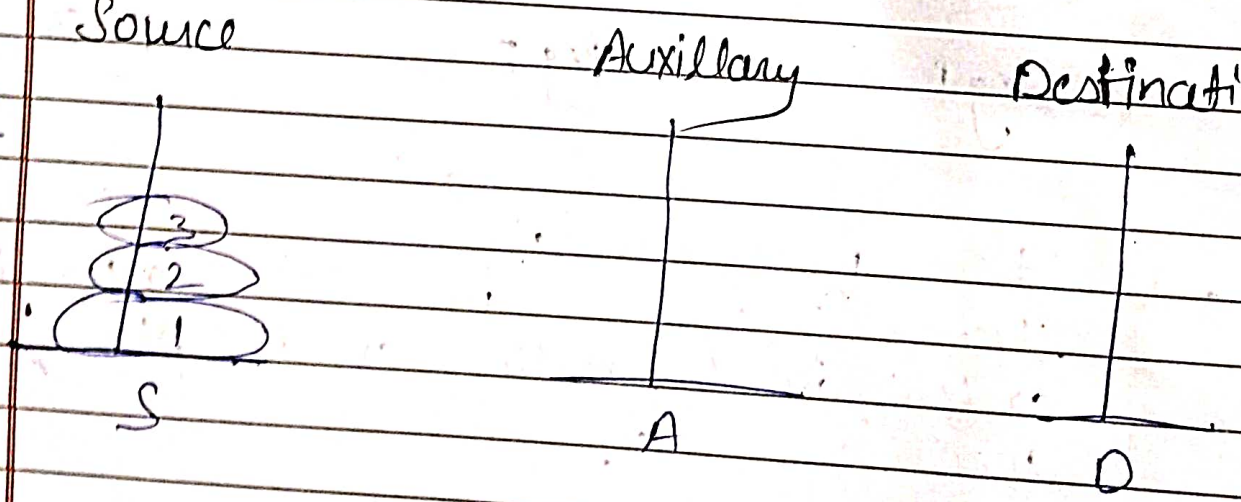
⇒ Only one disk can be moved among the towers at any given time.

⇒ only the top disk can be removed

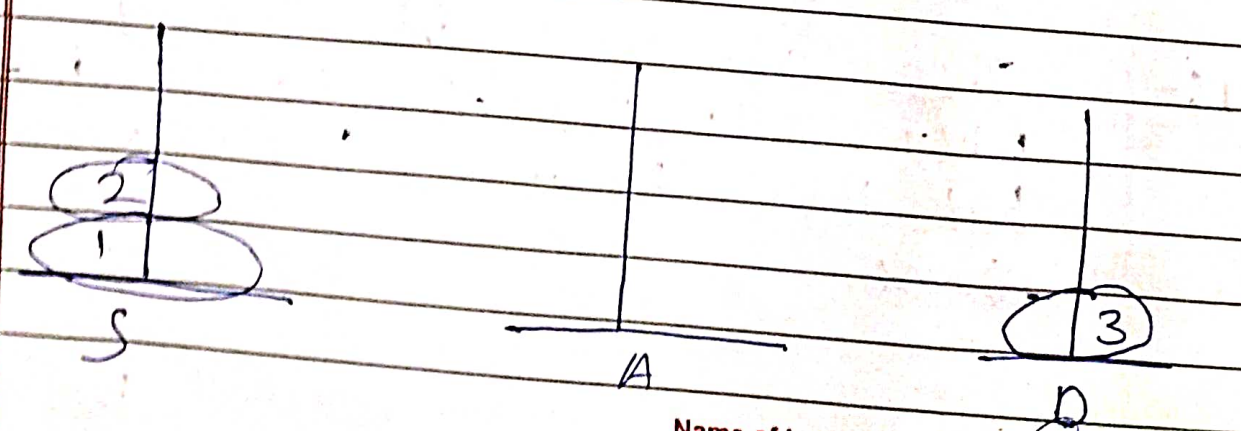
⇒ No Large disk can be placed over the small disk.

Tower of hanoi puzzle with 3 disks :-

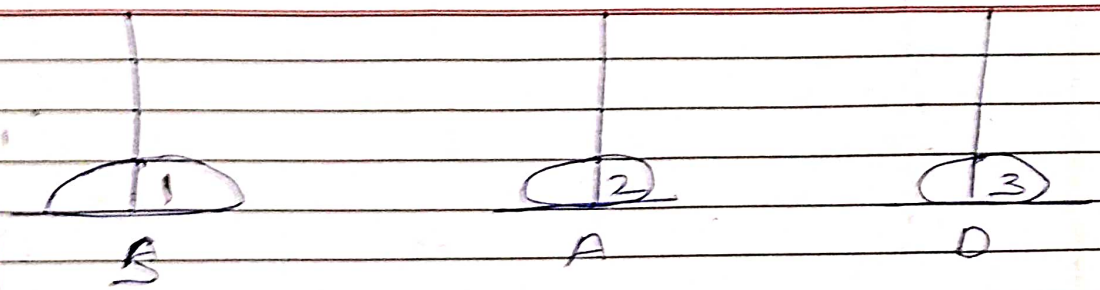
Problem



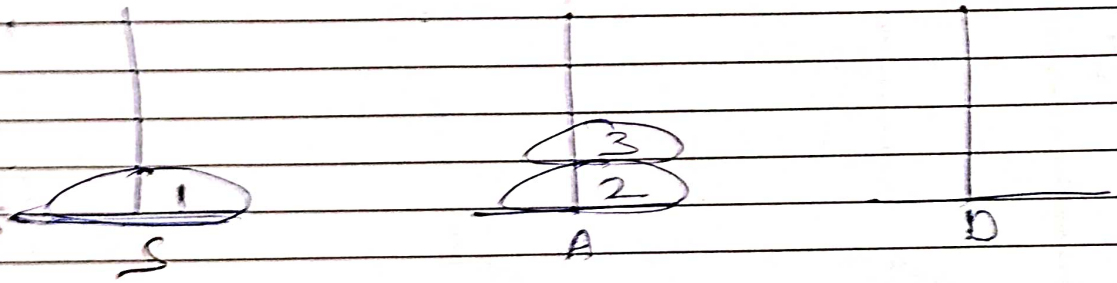
Step 1:



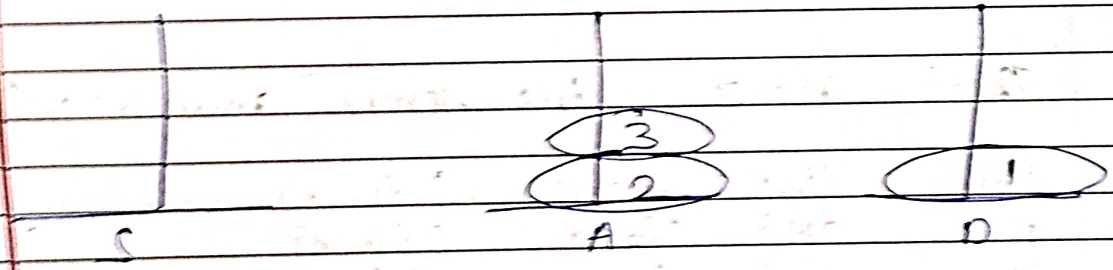
Step 2



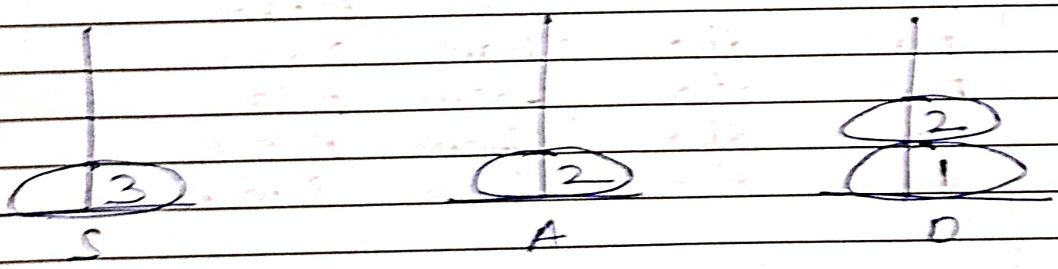
Step 3



Step 4

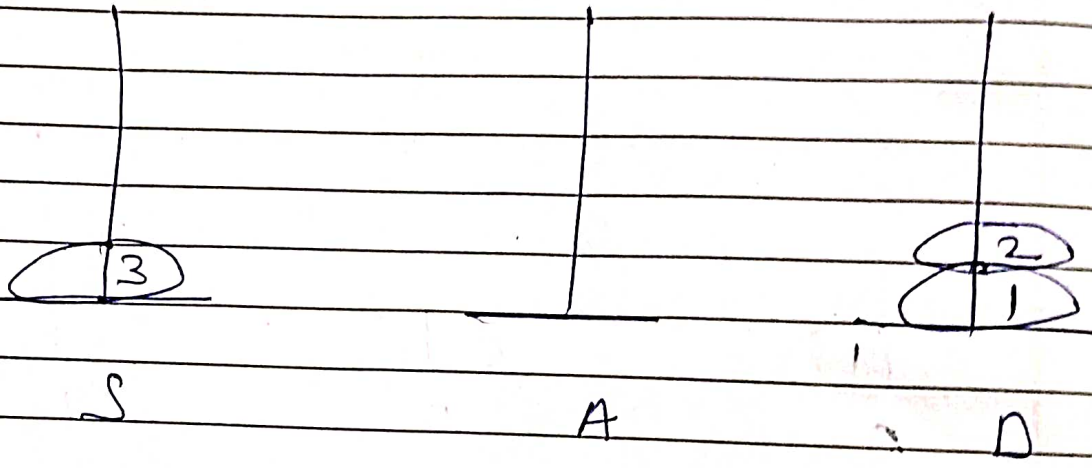


Step 5

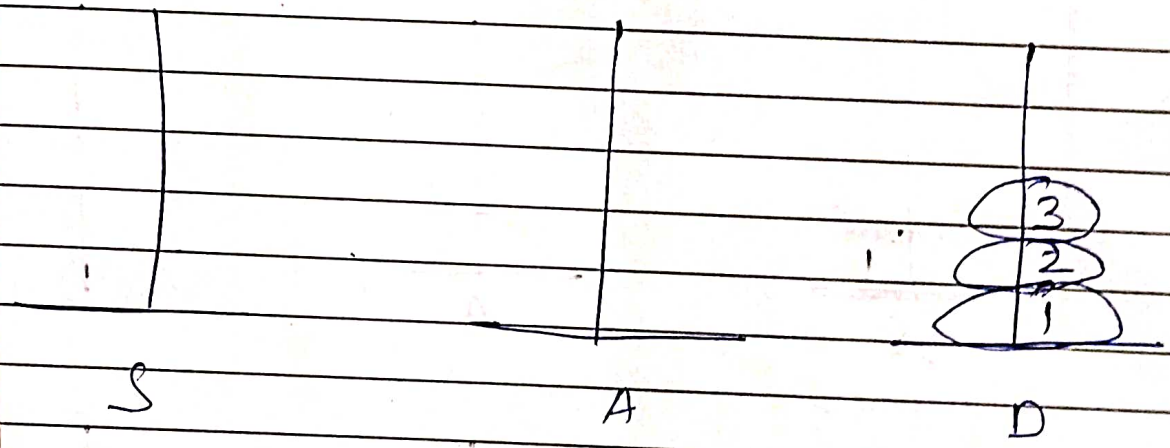


Name of Lecturer: .....

Step 6



Step 7



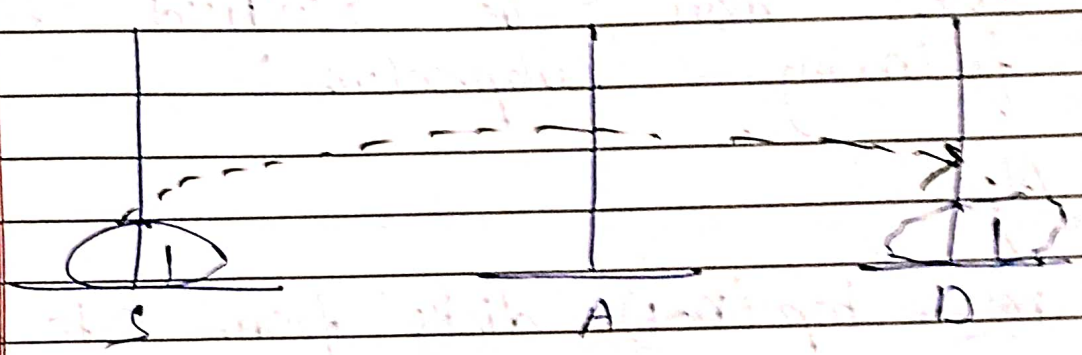
Thus $n=3$ the seven moves will be :-

⇒ 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

Name of Lecturer: Prachi goyal

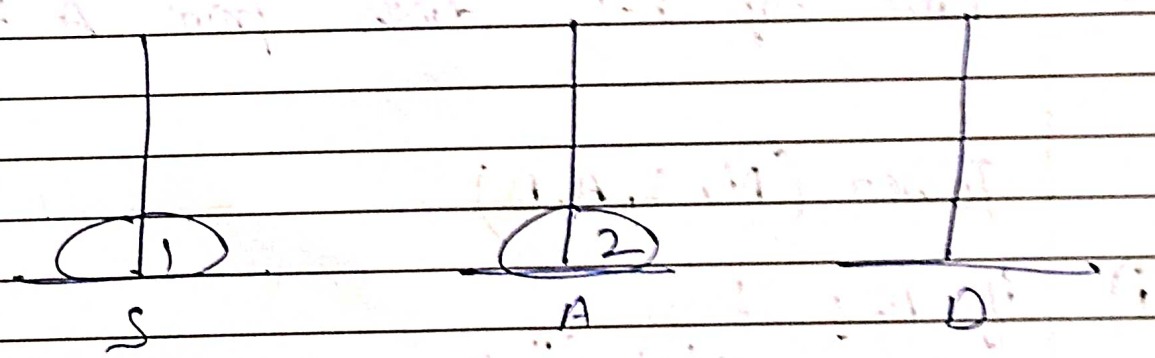
Recursive Solution for Tower of Hanoi problem :-

⇒ for n=1, only one moves

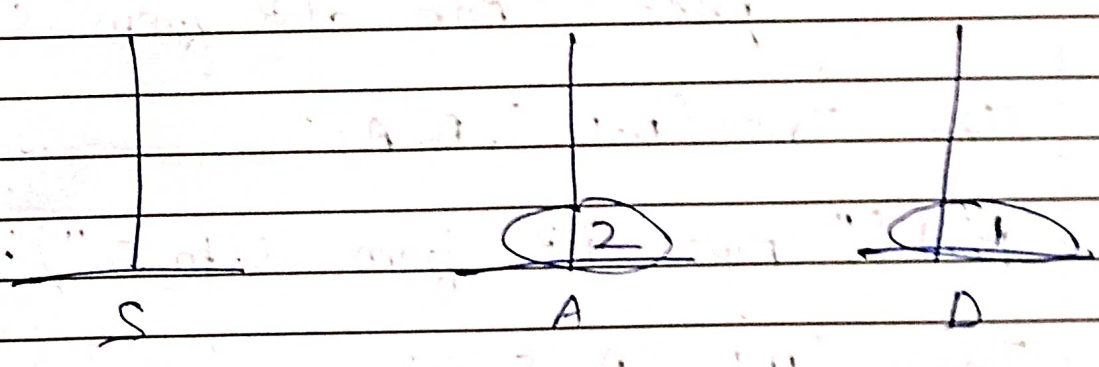


⇒ For n=2 three moves.

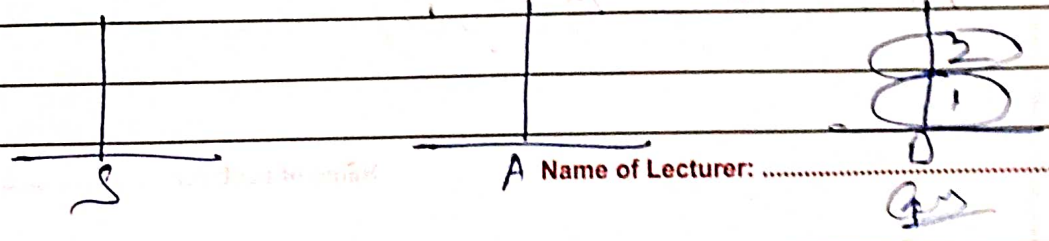
Step1



Step2



Step3



→ Rather than finding a separate solution for each value of n, we will 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 :-

Algo :-

⇒ move top (n-1) disks from S to A

⇒ Move top disk from S to D

⇒ 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, D, A)

(3) print (" Move disk from S to D ");

(4) Tower (N-1, A, S, D)

Name of Lecturer: .....

Tower(4, S, A, D)

Tower(1, S, D, A)

Tower(2, S, A, D) — A to D

Tower(3, S, D, A)

Tower(1, A, S, D)

Tower(4, S, A, D)