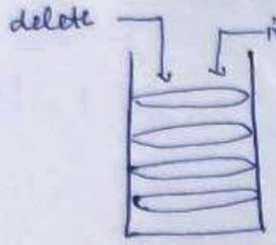
Stack

- It is a linear data structure & follows the rule:-

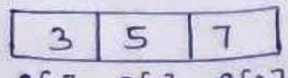
LIFO OR **FILO**
(Last In First Out) (First In Last Out)



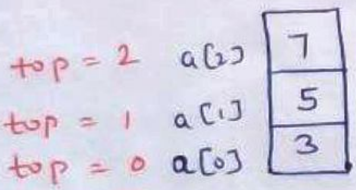
Implementation of Stack

Static

- (using array)
- int stack[]



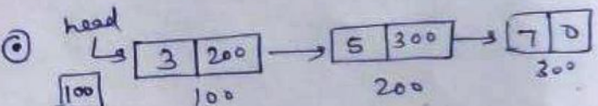
⇓ into stack



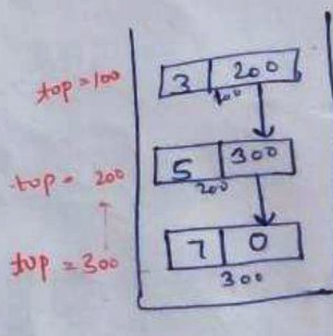
initially, (top = -1) i.e. No element, then it is incremented by 1.

dynamic

- (using Linked List)
- struct node
{
 int data;
 struct node *next;
};



⇓ into stack



- Operations :→
  - Push (x) → insert 'x' into stack.
  - Pop () → delete
  - Peek() → top most element of stack.
  - isempty() → T/F (if stack is empty)
  - is full () → T/F (if stack is full).

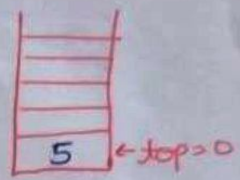
Implementation using array :-

Push :→

```
# define N 5;        → define N=5;
int stack [N];       → Size of stack = N = 5
int top = -1;
void push ()
{
        int x;
        printf(" enter data");
        scanf("%d", & x);
        if (top = = (N-1))
        {
// Stack is full//  ←  printf(" overflow");
        }
        else
        {
                top++;
                stack [top] = x;
        }
```

N=5, top=-1

⇓ insert x=5



Pop :→

```
void pop ()
{
        int item;
        if ( top == -1)
        {
//Stack is empty // ←  printf(" Underflow");
        }
        else
        {
                item = stack [top];
                top -- ;
                printf (" deleted item is %d", item);
        }
```

Peek :→

```
void peek()
{
    if(top == -1)
    {
        printf(" stack is empty");
    }
    else
    {
        printf("%d", stack[top]);
    }
}
```

display :→

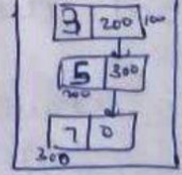
```
void disp()
{
    int i;
    for(i = top; i >= 0; i--)
    {
        printf("%d", stack[i]);
    }
}
```

top = 3

display → 7, 5, 3

Implementation using Linked List :→

Push :→




```
struct node
{
    int data;
    struct node* next;
};
struct node * top = 0;
void push()
{
    int x;
    printf("enter data");
    scanf("%d", &x);
```

```
struct node * new-node;
new-node = (struct node *) malloc(sizeof(struct node));
    new-node -> data = x;
     new-node -> next = top;
        top = new-node;
}
```

Pop:->

```
void pop()
{
    struct node * temp;
     temp = top;
    if (top == 0)
     {
        printf(" stack is empty");
     }
    else
     {
        printf("deleted item is %d", top->data);
          top = top->next;
          free(temp);
     }
}
```

display:->

```
void display()
{
    struct node * temp;
     temp = top;
    if (top == 0)
     {
        printf(" Stack is empty");
     }
    else
     {
        while (temp != 0)
        {
            printf("%d", temp->data);
        }   temp = temp->next;
     }
}
```

Peek() :→

```
void peek()
{
    if (top==NULL)
    {
        printf(" empty stack");
    }
    else
    {
        printf(" top element is %d", top->data);
    }
}
```

Infix, Prefix & postfix :→

Infix	=	a+b	<operand> <operator> <operand>
(Polish) OR Prefix	=	+ab	<operator> <operand> <operand>
(Reverse Polish) OR postfix	=	ab+	<operand> <operand> <operator>

Precedence & Associativity

1. ( ), [ ], { } → Right to left
2. ^ → left to right
3. × / → left to right
4. + −

Eg:- a * b + c (Infix) $\xRightarrow{\text{Prefix}}$ *ab + c ⇒ +*abc (Prefix)

⇓ postfix

ab* + c ⇒ ab*c+ (Postfix)

Conversion from infix to postfix :→

1) Print operands.
2) Stack is empty → '(' comes → push incoming operator
3) '(' comes → push it.
4) ')' comes → pop until '(' found.
5) Higher precedence → push into stack.

6) lower precedence → pop & print the top, test again.

7) Equal → check Associativity rule.

- → L to R → pop & print top, push incoming operator
- → R to L → push incoming operator.

8) pop & print all operators.

Eg:- A + B / C

Sol.

	Infix	Stack	Postfix
	A	−	A
	+	+	A
	B	+	AB
Higher than +	/	+, /	AB
	C	+, /	ABC
			$\boxed{ABC/+}$

Eg:- A − B / C × D + E

Sol

	Infix	Stack	Postfix
	A	−	A
	−	−	A
	B	−,	AB
Higher Precedence Push as it is ←	/	−, /	AB
	C	−, /.	ABC
Equal Precedence ←	×	−, ×	ABC/
	D	−, ×	ABC/D
lower ←	+	−, (+) Associativity	ABC/D× −
	E	+	ABC/D× − E
			ABC/D× − E+

Conversion from infix to prefix :-

1. Reverse the expression.
2. Print operands.
3. Stack is empty → ')' comes → push incoming operator.
4. ')' → push it.
5. '(' → pop until ')' found.
6. Higher precedence → push incoming operator.
7. Lower precedence → pop & print top, test again.
8. Equal → Associativity rule
   - L-R → push incoming operator.
   - R-L → pop, test again.
9. Pop & print all operators.
10. Reverse again.

Eg:- A - B / C × D + E

Sol E + D × C / B - A

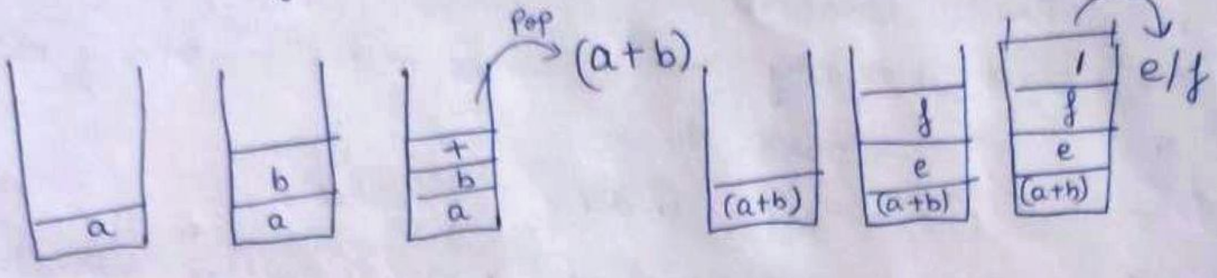
Infix	Stack	Prefix
		E
E	-	E
+	+	ED
D	+	ED
×	+, ×	ED
C	+, ×	EDC
/	+, ×, /	EDC
B	+, ×, /	EDCB
- (lower, pop)	+, -	EDCB/×
A	+, -	EDCB/×A
		EDCB/×A-+

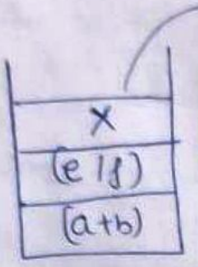
Reverse

$\boxed{+-A\times/BCDE}$

⊙ Postfix to infix

Eg:- ab + ef / x   ⊙ scan from left to right

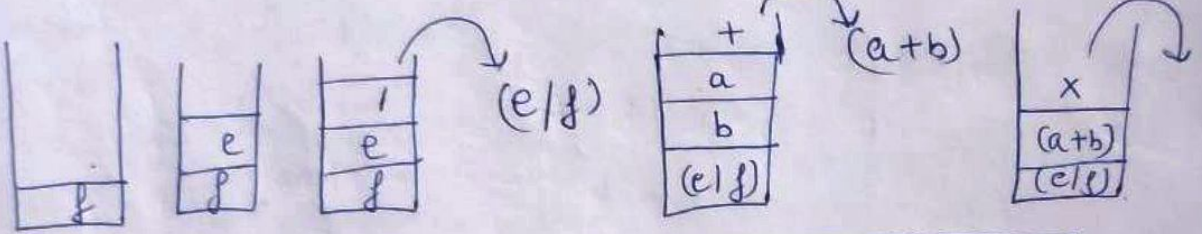




$\boxed{((a+b) \times (e/f))}$

⊙ Prefix to infix :→

Eg:- x + ab / ef   ⊙ scan from right to left



$\boxed{((a+b) \times (e/f))}$

Evaluation of prefix :→

1. Scan prefix expression from ~~left~~ right to ~~right~~ left for each char in prefix expression
2. do
   if operand is there, push it onto stack
   else if operator is there, pop 2 elements
   op 1 = top element
   op 2 = next to top element
   result = op 1 operator op 2

push result onto stack

3. return stack [top].

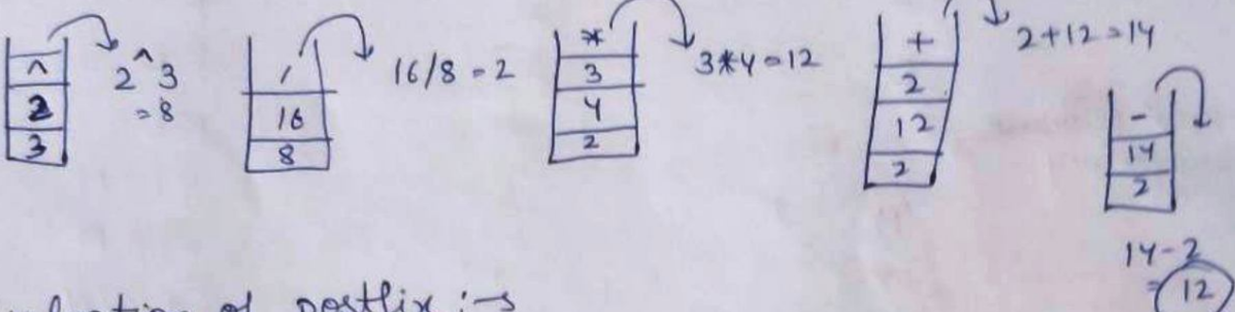
Eg:- $a+b*c-d/e\wedge f$

$a=2,\ b=3,\ c=4,\ d=16,\ e=2,\ f=3$

Sol. First into prefix:-

$-+a*bc/d\wedge ef$

$-+2*34/16\wedge 23$ ⟵ right to left scan



Evaluation of postfix:-

1. Scan postfix expression left to right for each char.
2. do

   if operand is there, push it onto stack.

   else if operator is there, pop 2 elements

   op 1 = top element

   op 2 = next to top element

   result = op2 operator op1

   push result onto stack.

3. return stack [top].

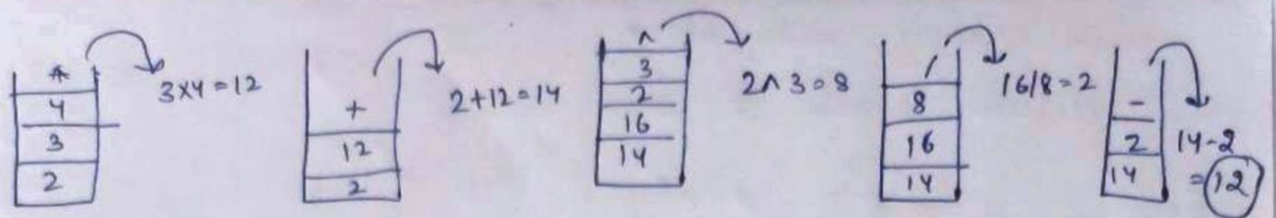
Eg:- $a+b*c-d/e\wedge f$

$a=2,\ b=3,\ c=4,\ d=16,\ e=2,\ f=3.$

Sol: First into postfix

$abc*+def\wedge/-$

scanning ⟶ $234*+16\ 2\ 3\wedge/-$



Q. $K+L-M \times N+(O^P) \times W/U/V \times T+Q$. Convert it into prefix & postfix.

Sol.

Postfix :-

	Input	Stack	Postfix
	K	—	K
	+	+	K
	L	+	KL
equal precedence, associativity rule ⇒	-	-	KL+
		-	KL+M
	M	-	KL+M
Higher precedence, Push ←	X	-, X	KL+MN
	N	-, X	KL+MNX-
lower precedence → Pop, print & test	+	+,	KL+MNX-
	(	+ (	KL+MNX-O
	O	+ (	KL+MNX-O
	^	+ (^	KL+MNX-OP
	P	+ (^	KL+MNX-OP^
	)	+	KL+MNX-OP^
	X	+, X	KL+MNX-OP^W
	W	+, X	KL+MNX-OP^WX
	/	+, /	KL+MNX-OP^WXU
	U	+, /	KL+MNX-OP^WXU/
	/	+, /	KL+MNX-OP^WXU/V
	V	+, /	KL+MNX-OP^WXU/V/
	X	+, X	KL+MNX-OP^WXU/V/T
	T	+, X	KL+MNX-OP^WXU/V/TX+
	+	+, *	KL+MNX-OP^WXU/V/TX+Q
	Q	+	

KL+MNX-OP^WXU/V/TX+Q+

Into Prefix :→

Reverse :-

Q + T X V / U / W X) P ^ O (+ N X M − L + K

	Input	Stack	Prefix
	Q	−	Q
	+	+	Q
	T	+	QT
Higher, Push	X	+, X	QT
	V	+, X	QTV
equal, associativity	/	+, X, /	QTV
	U	+, X, /	QTVU
	/	+, X, /, /	QTVU
	W	+, X, /, /	QTVUW
	X	+, X, /, /, X	QTVUW
	)	+, X, /, /, X,)	QTVUW
	P	+, X, /, /, X,)	QTVUWP
	^	+, X, /, /, X,), ^	QTVUWP
	O	+, X, /, /, X,), ^	QTVUWPO
Pop ⇒	(	+, X, /, /, X	QTVUWPO^
lower	+	+, +	QTVUWPO^X//X
	N	+, +	QTVUWPO^X//XN
	X	+, +, X	QTVUWPO^X//XN
	M	+, +, X	QTVUWPO^X//XNM
	−	+, +, −	QTVUWPO^X//XNMX
	L	+, +, −	QTVUWPO^X//XNMX
	+	+, +, −, +	
	K	+, +, −, +	

Iteration & Recursion :→

Recursion is the process which comes into existence when a function calls a copy of itself to work on a smaller problem.

Iteration	Recursion
1) set of statements executed repeatedly. Eg:- for loop, while loop, do-while loop. `for( i=0 ; i<5 ; i++)`	1) Function called itself. `fun() { fun() }`
2. Can be called by several times.	2) called by iterative or looping control statements.
3. Performed on functions.	3) performed on set of statements as long as condition is true.

Recursion	Iteration
4) For implementation, if-else, else if can be used.	4) for implementation, for, while, do while loop can be used.
5) Overhead.	5) No overhead.
6) Slow	6) Faster
7) Size is small.	7) Size is big.
8) It can solve all problems.	8) It can solve limited problems.

Recursive

Base case

- (for termination)

Recursive case

- Simplifies a bigger problem into simpler sub-problems & then calls them i.e. (for calling)

Types of Recursion

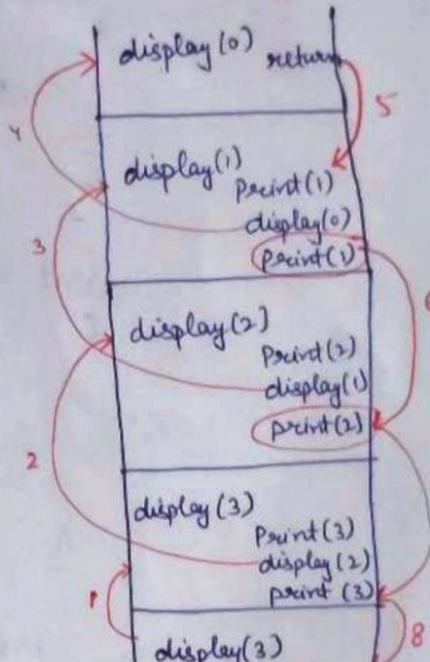
- Direct / Indirect
- Tail / Non-Tail (Head)
- Tree / Graph

1. <u>Direct recursion</u> :- Function call itself with in the <u>same</u> function.

<u>Syntax</u>:-

```
fun()
{
    fun();
}
```

<u>Example</u>:→




```
void display(int n)
{
    if (n<1)
        return;
4   else
6   {
        printf("%d", n);
        display(n-1);
        printf("%d", n);
    }
}
void main()
{   display(3);
}
```

output :- 3 2 1 1 2 3

2. <u>Indirect Recursion</u> :→ Function is mutually called by another function in circular manner.

Syntax :-

```
fun 1()
{
    fun 2()
}
fun 2()
{
    fun1()
}
```

Example :-

```
void main()
{
    printf(" %d", fun1(5));

}
int fun1(int n)
{
    if (n <= 1) return1;
    else
        n X fun2(n-1);
}
int fun2 ( int n)
{
    if (n <= 1) return1;
    else
        return n X fun1(n-1);
}
```

Return

1

2 X fun 1(1) 2X1 =2

3X fun 2(2) = 3X2 =6

4 X fun 1(3) =4X6 =24

5 X fun 2(4) 5X24 =120

fun1(5)

print 120

output :- 120

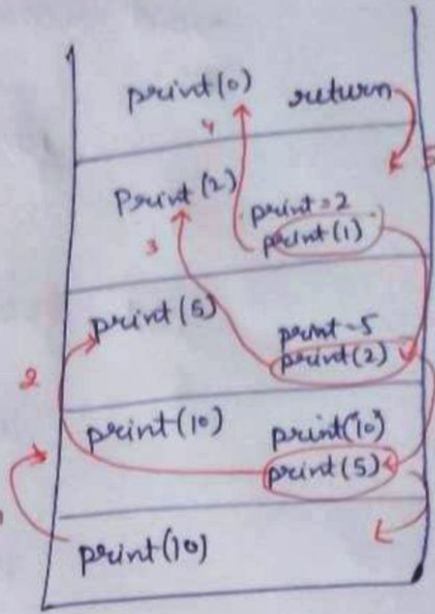
3. Tail recursion :-

If recursive call is the last statement executed by the function. It is same as iteration. We can use iteration instead of this, because it is wastage of memory.

Example :-

```
void print (int a)
{
    if (a<1)
        return;
    else
    {
        printf("%d", a);
        print (a/2);
    }
}

void main()
{
    print(10);
}
```



Output :-

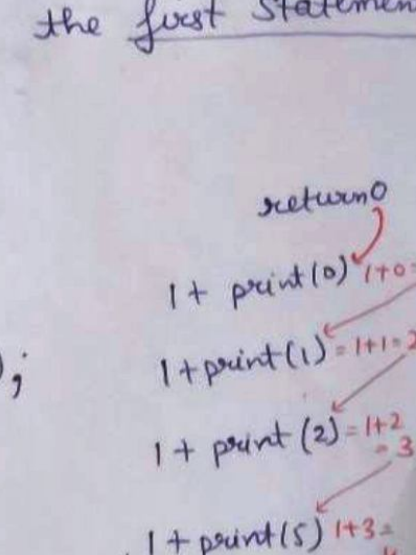
10 5 2 1 1 2 5

4. Non - Tail :- Recursive call will be the first Statement

Example :-

```
void print (int a)
{
    if (a< 1)
        return0;
    else
        return 1 + print (a/2);
}

void main()
{
        int x;
    x= print (10);
    printf("%d", x);
}
```



Output = 4 .

Factorial :-

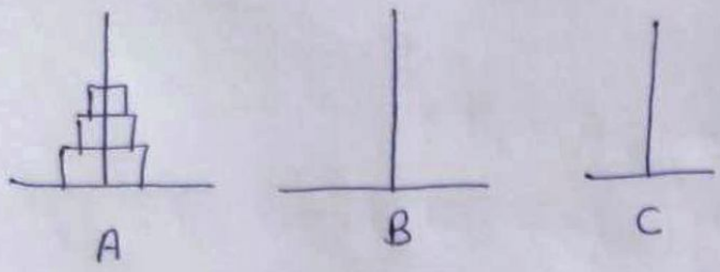
```
void main()
{
    fact(5);
}

int fact (int n)
{
    if (n==1)
        return 1;        // base case
    else
        return n X fact (n-1)    // recursive
                                    case
}
```

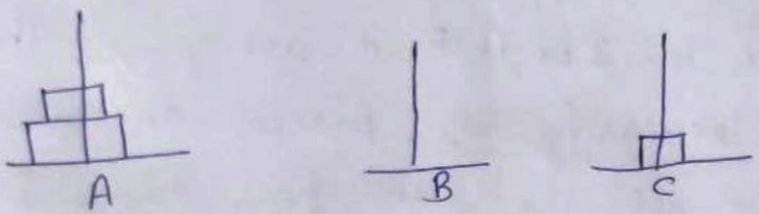
Fibonacci :-

```
void main()
{
    fibonacci(5);
}
int fibonacci( int i)
{
    if (i==0)
        return 0;
    else if (i==1)
        return 1;
    else
        return ( fibonacci (i-1) + fibonacci(i-2));
}
```

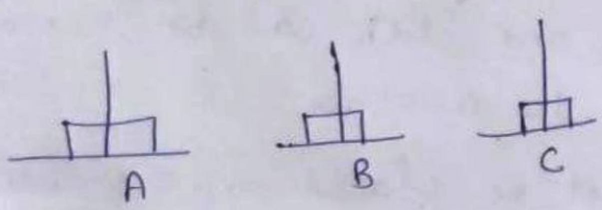
Tower of Hanoi :→ A, B, C are towers.



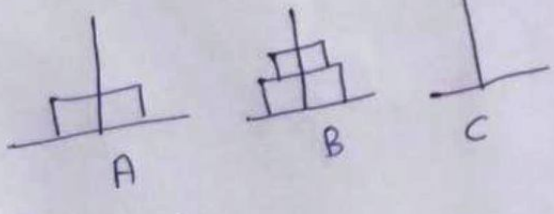
a) A → C



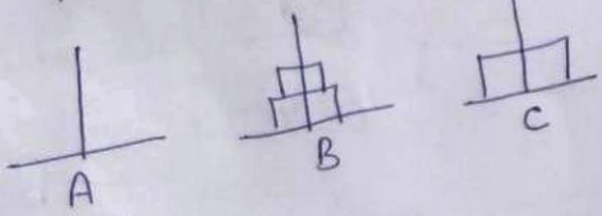
b) A → B



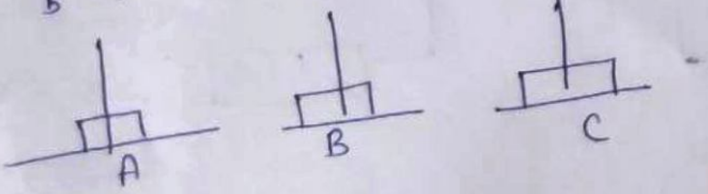
c) C → B



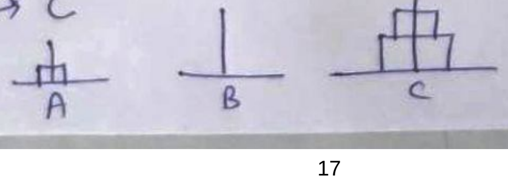
d) A → C



e) B → A



f) B → C



7 steps i.e. 3disk $\boxed{2^3 - 1}$

g) A → C

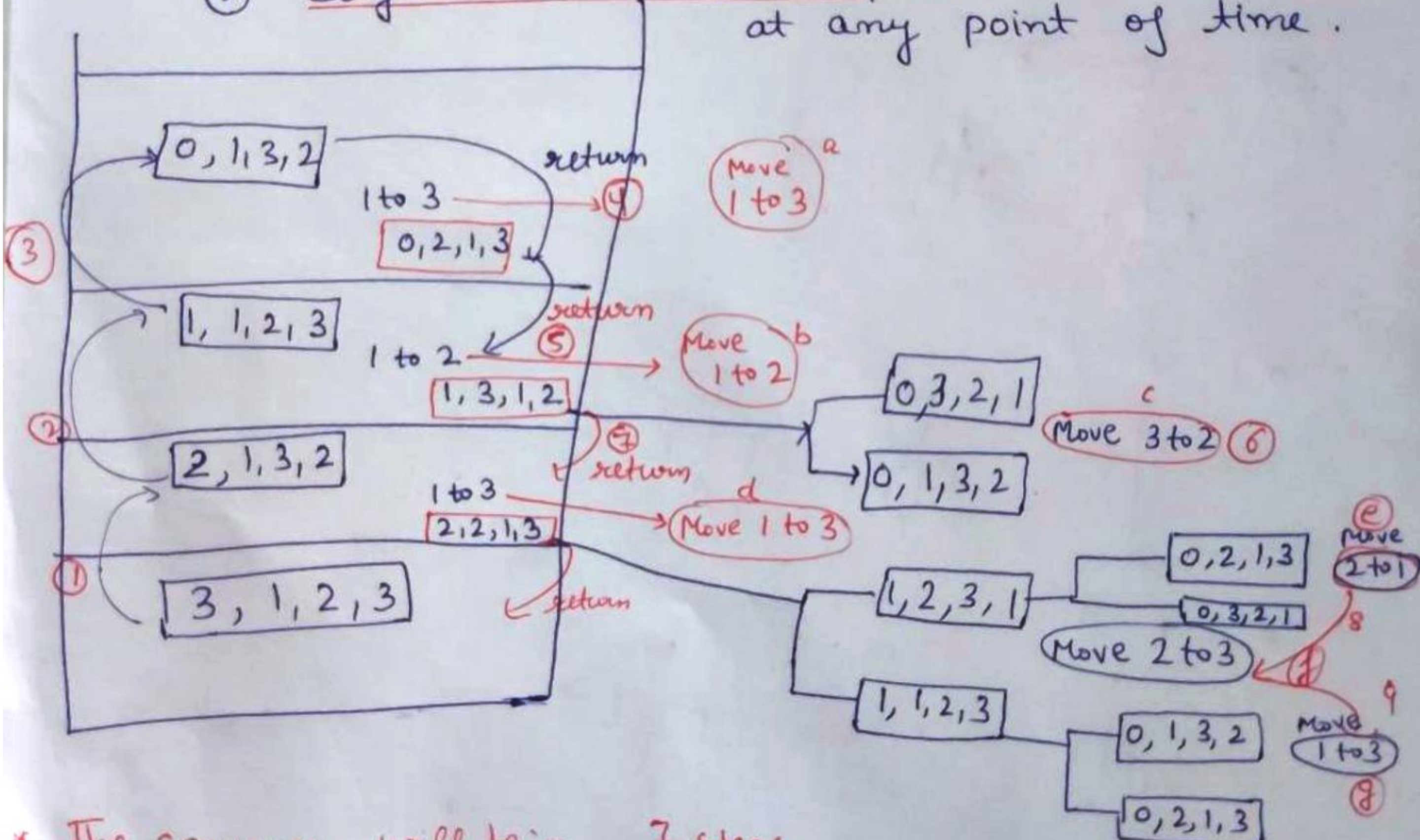
disk, 1st, Aux, final

```
void toh (n, A, B, C)
{
                        if (n==0) return;
    else if (n>0)
        {
            toh (n-1, A, C, B);
            printf (" from %d to %d", A, C);
            toh (n-1, B, A, C);
        }
}
```

⇒ Three towers labelled 1, 2, 3 or A, B, C are given. There are n no. of disks with decreasing size placed on tower 1. The aim is to move all the disk from tower 1 to tower 3 through an auxiliary tower i.e. tower 2.

Rules :→ ① At a time only one disk can be removed from one tower to another. (i.e. topmost disk)

② Larger disk cannot be placed on a smaller disk at any point of time.



* The sequence will be :- 7 steps

(1, 3), (1, 2), (3, 2), (1, 3), (2, 1), (2, 3), (1, 3)