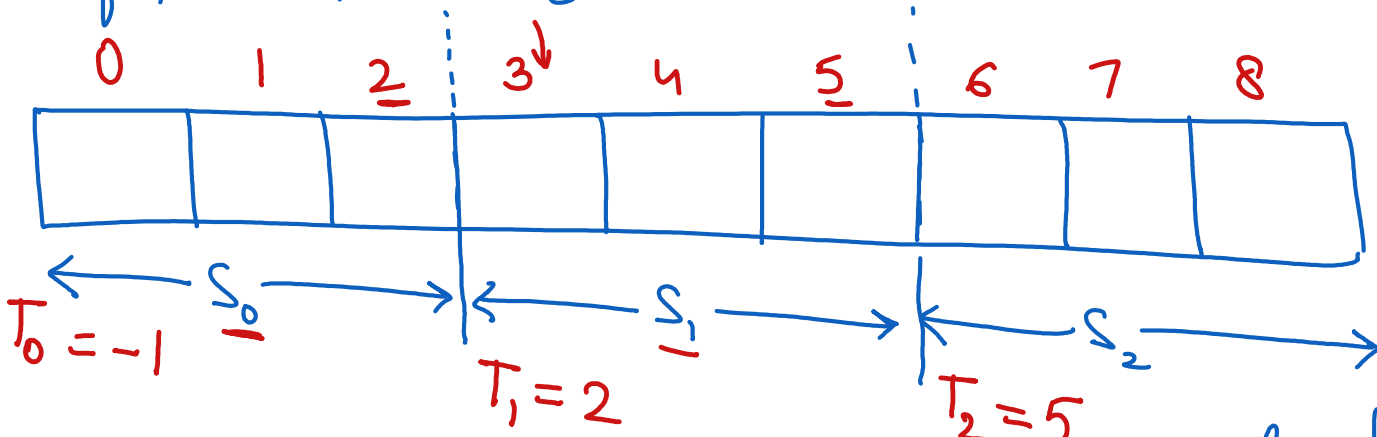
Multiple Stack Implementation:

Size of an array (M) = 9

No. of stack (N) = 3



Empty cond^n

$T_0 = \left(\frac{9}{3}\right) * 0 - 1 = -1$

$T_1 = \left(\frac{9}{3}\right) * 1 - 1 = 2$

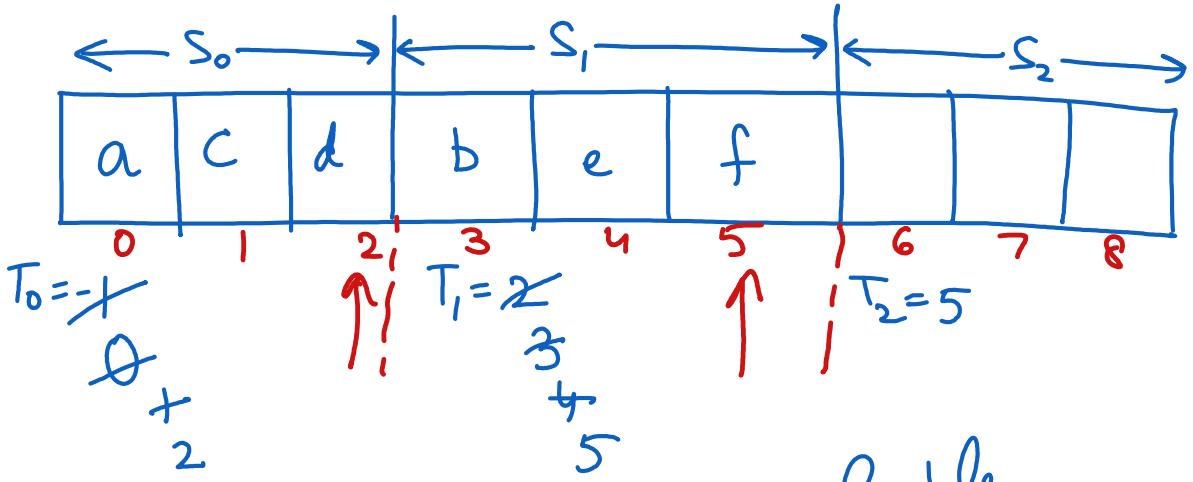
$T_2 = \left(\frac{9}{3}\right) * 2 - 1 = 5$

Empty $\boxed{T_i = \left(\frac{M}{N}\right) * i - 1}$

Full condition

$M = 9$

$N = 3$



$T_0 = 2\ (3-1)$

$T_1 = 5\ (3\times 2-1)$

$T_2 = 8\ (3\times 3-1)$

$$T_i = (i+1)\left(\frac{M}{N}\right) - 1$$

Problem → fixed boundaries

Implementation of push() -

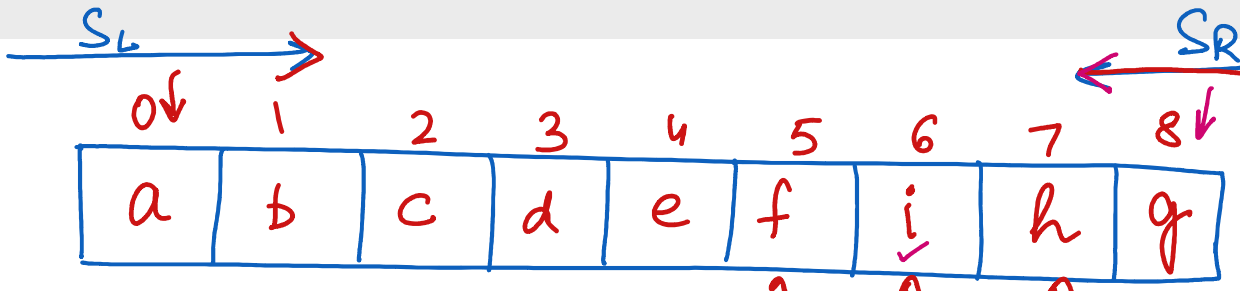
```
void push(){
   int x;
   if(ti == (i+1)(M/N)-1) {
      printf("\n Stack is over flow");
      exit(1);
    }
   else{
      printf(" Enter a value to be pushed:");
      scanf("%d", &x);
      ti++;
      stack[ti]=x;
   }
}
```

1. if check isfull()
2. else
   top++
   s[top] = x;

Implementation of pop() -

```
void pop() {
   if(Ti ==(M/N)-1) {
       printf("\n Stack is under flow");
        exit(1);
   }
  else{
    printf("\n Popped elements is %d",  stack[Ti]);
     Ti--;
  }
}
```

Efficient Way Of Implementing Multiple Stacks:



(Two stack implementation

Size = 9

$T_L = -1 \to 0 \to 1 \to 2 \to 3 \to 4 \to 5$

$T_L{+}{+}$
$S[T_L] = x$

$T_L = T_R - 1$
$T_R = T_L + 1$
full condition

$T_R = 9 \to 8 \to 7 \to 6$

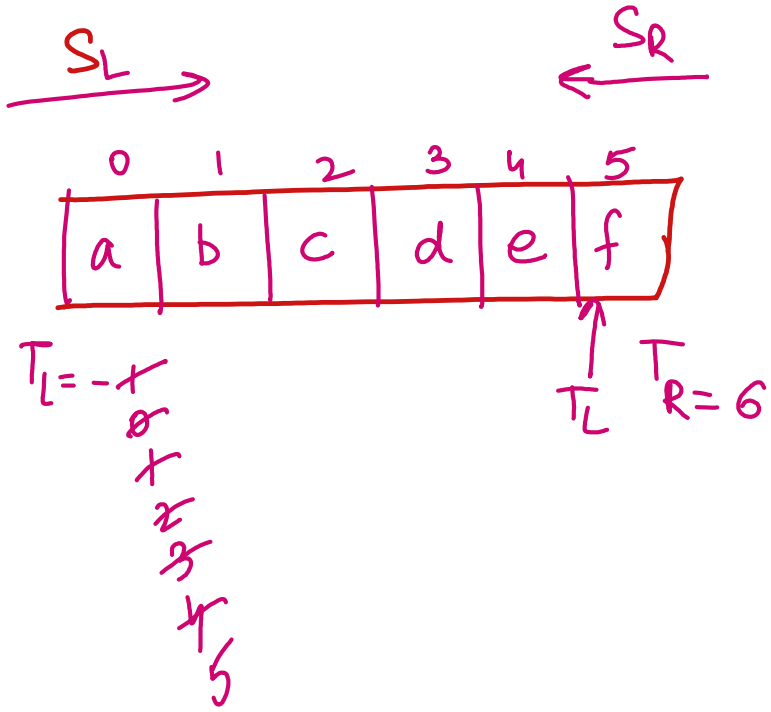
$T_R{-}{-}$
$S[T_R] = y$

$T_L = 8$
$T_R = 9$

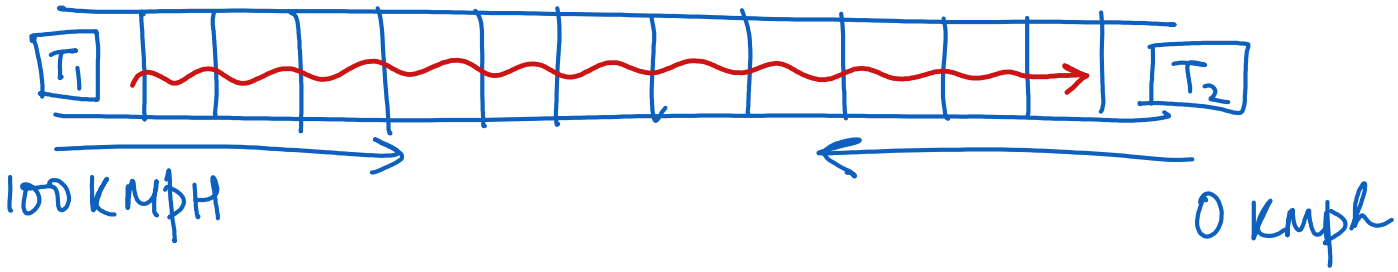
$T_L = -1$
$T_R = 0$

Boundaries are not fixed.

If only left stack is moving



$$T_L = T_R - 1$$


T_1
T_2
100 KMPH
0 Kmph

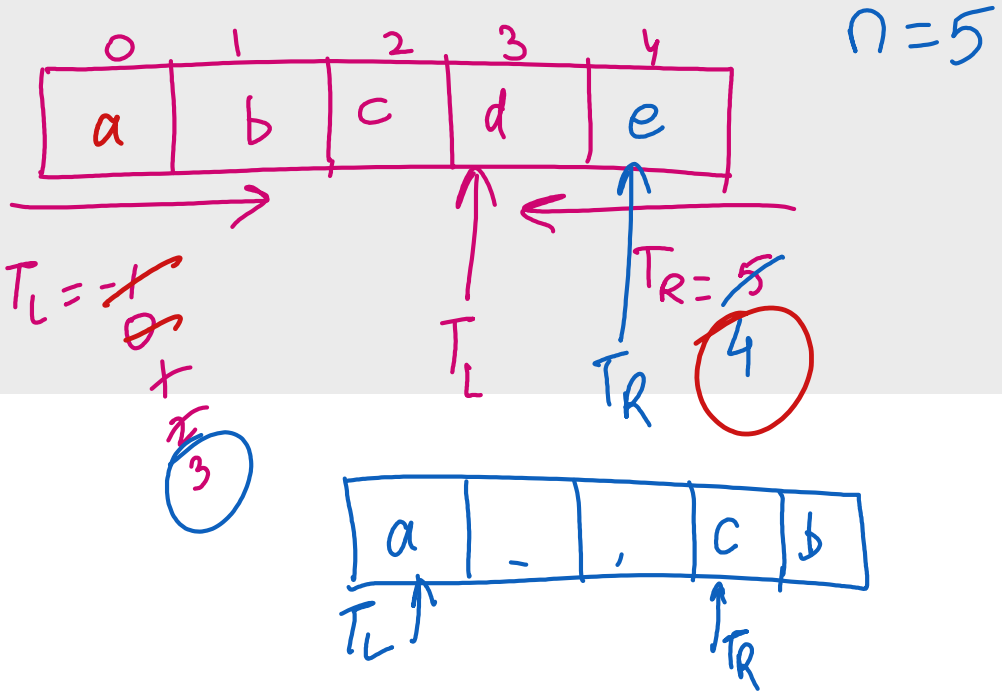
Inorder to implement multiple stacks efficiently, we are taking 2 stacks which are growing in opposite order. Then what will be the full condition? Consider the size of an array is n.

(a) Tl +Tr = n ✗

(b) Tl = Tr = n/2 ✗

(c) Tr = Tl -1 ✗ $4 \neq 3-1$

(d) Tl = Tr - 1 ✓ $3 = 4-1$

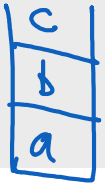


Applications to Stack

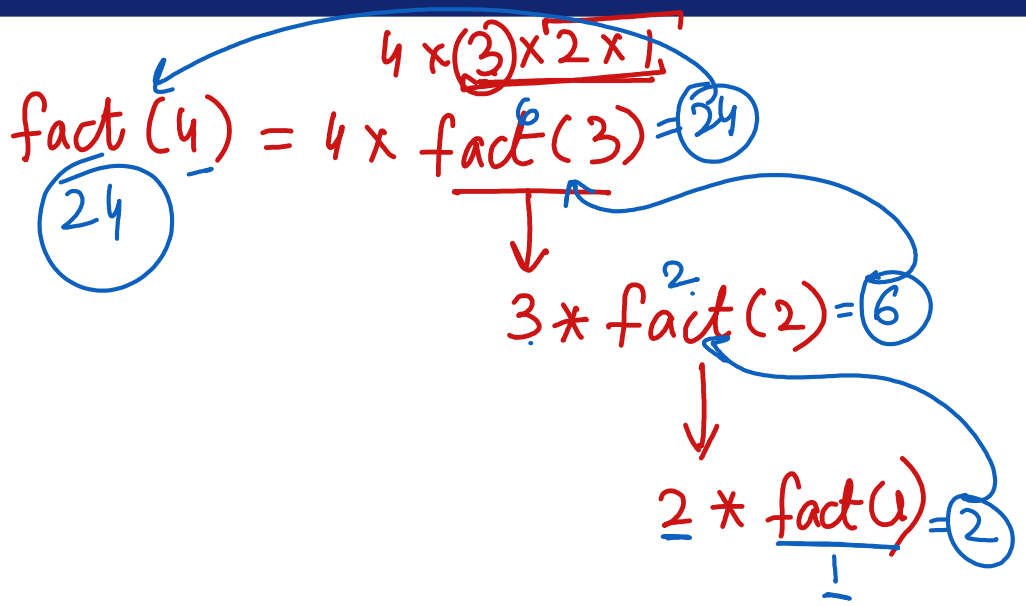
- Used in "undo" mechanism at many places like editors, photoshop etc.
- Compiler's syntax check for matching braces is implemented by using stack.
- Support for recursion.
- Infix to prefix/postfix conversion , prefix to postfix conversion etc.
- Expression evaluation like postfix or prefix in compilers.
- Forward and backward feature in web browsers
- Used in many algorithms like Tower of Hanoi, Fibonacci series, tree traversals etc.
- String reversal.

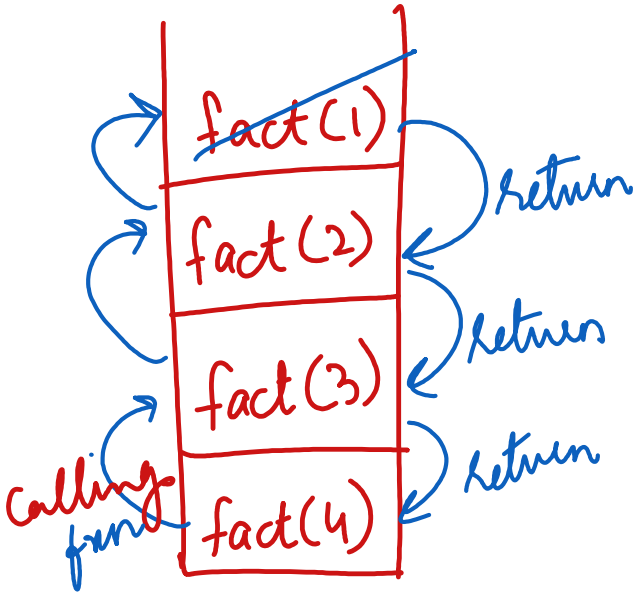
fact(4) = 4x3x2x1

abc



cba





Termination condition : fact(1) = 1

Applications of stack -

- Recursion
- Infix to postfix conversion
- Prefix to postfix conversion
- Infix to prefix conversion
- Postfix evaluation
- Tower of Hanoi
- Fibonacci series

Recursion:

- Recursive functions typically implement recurrence relations, which are mathematical formula.
- The function written in terms of itself is a recursive case.
- Recursion is a repetitive process in which a function calls itself.
- A function is called recursive if a statement within the body of a function calls the same function.
- Also Known as circular definition.

Tail Recursion:

- If in the given recursive program, recursive call is at the end of the program, that recursion is known as tail recursion.
- Disadvantage: unnecessarily wasting stack space.
- Advantage: can be easily converted to an equivalent non-recursive program.

Eg:

```
Tail(int i)
{
  if(i<=1)
   return;
  else
  {
   printf("%d", i);
   Tail(i-1);
  }
}
```

Recursive

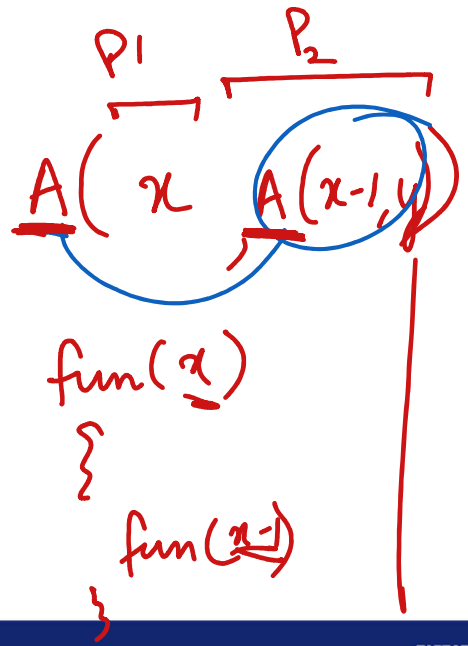
i = 5

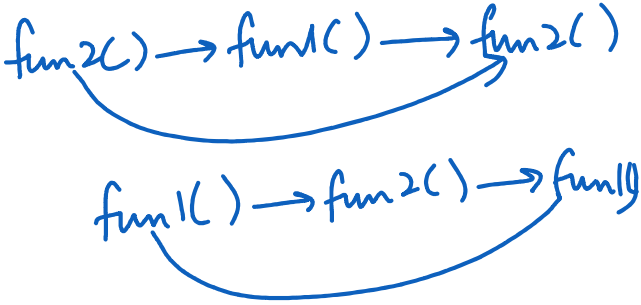
Tail(5)
- 5
- Tail(4)
  - 4
  - Tail(3)
    - 3
    - Tail(2)
      - 2
      - Tail(1)

5 4 3 2

Types of recursion

1) Tail recursion
2) Non-Tail recursion
3) Nested recursion
4) Indirect recursion






```
fun1(x)
{
  fun2(y)
}
```

```
fun2(x)
{
  fun1(y)
}
```

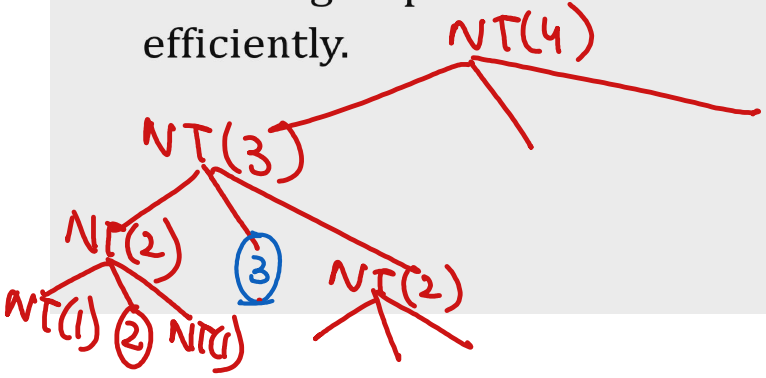
2, 3, 2, 4, 2, 3, 2

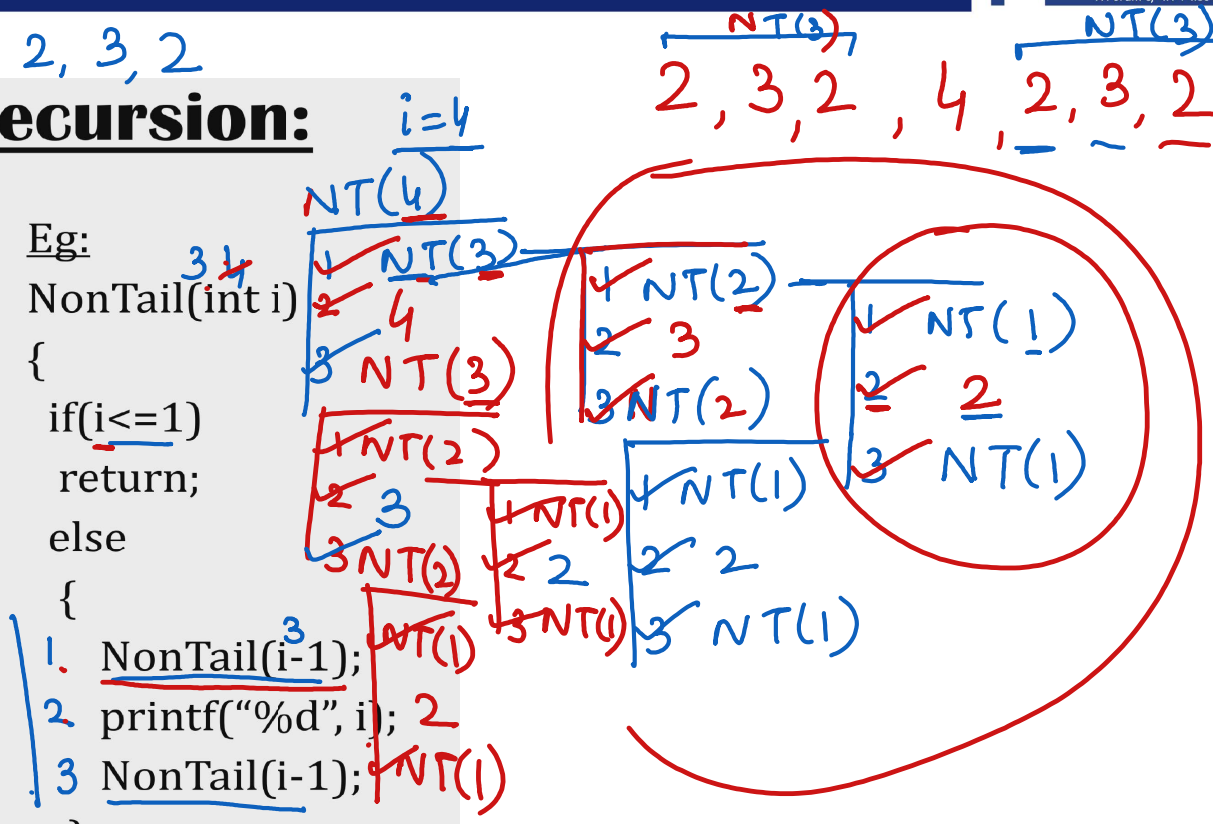
Non-Tail Recursion:

- If in the given recursive program, after the recursive call some statements are there then it is known as non-tail recursion.
- Disadvantage: Difficult to write an equivalent non-recursive program.
- Advantage: Space is utilized efficiently.

Eg:

```
NonTail(int i)
{
 if(i<=1)
  return;
 else
 {
  1. NonTail(i-1);
  2. printf("%d", i);
  3 NonTail(i-1);
 }
}
```





Nested Recursion

A recursion is nested if a recursive function includes one of the arguments to another recursive function.

Eg: Ackermann's Recurrence Relation / Ackermann function

$$A(m, n) = \begin{cases} n+1 & \text{if } m == 0 \\ A(m-1, 1) & \text{if } n == 0 \\ A(m-1, A(m, n-1)) & \text{otherwise} \end{cases}$$

Calculate

(i) A(1,3) = 5

(ii) A(2,2) = 7

A(1,4) = A(0, A(1,3))
A(0,5) = 6

A(1,3) = A(0, A(1,2))
A(0, A(1,1))
A(1,2) = A(0, A(1,1))
A(1,1) = A(0, A(1,0))
A(1,0) = A(0,1)
A(0,1) = 1+1 = 2

A(0,4) = 5
A(1,3) = A(0, A(1,2)) = A(0,3) = 4
A(1,2) = A(0, A(1,1)) = A(0,2) = 2+1 = 3
A(1,1) = A(0, A(1,0)) = 2
A(1,0) = A(0,1) = 2

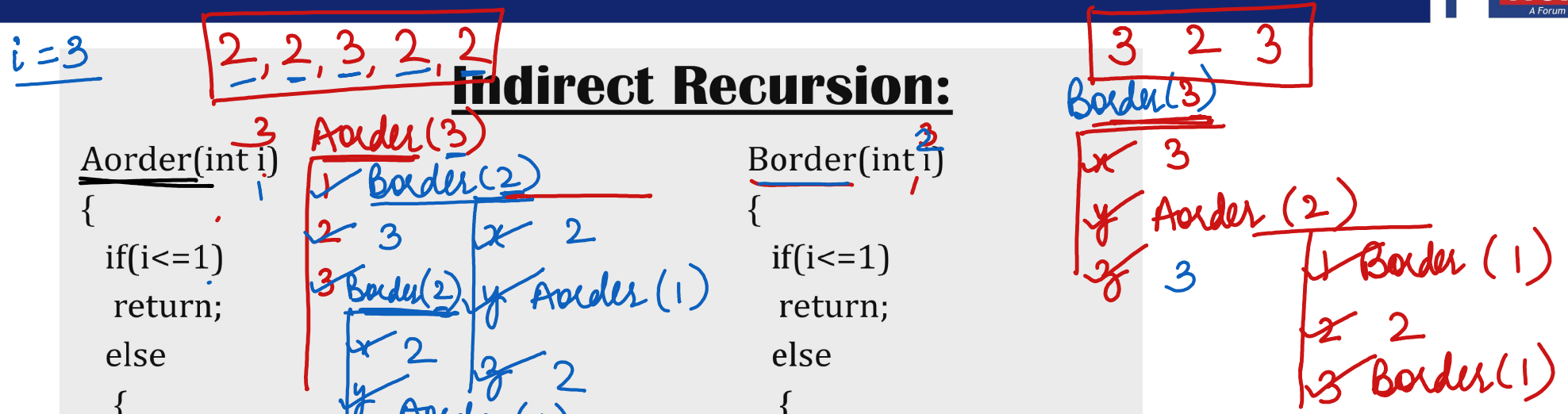
A(2,2) = A(1, A(2,1)) = 7
A(2,1) = A(1, A(2,0)) = A(1,3)
A(2,0) = A(1,1) = 6
A(1,5) = A(0, A(1,4)) = A(0,6) = 7

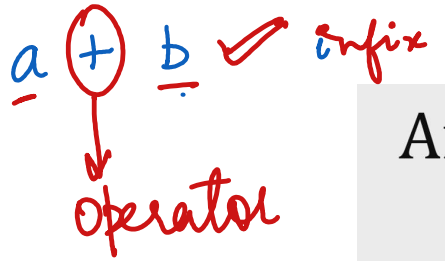
Indirect Recursion:

```
Aorder(int i)
{
  if(i<=1)
   return;
  else
   {
1   Border(i-1);
2   printf("%d", i);
3   Border(i-1);
    }
}
```

```
Border(int i)
{
  if(i<=1)
   return;
  else
   {
x   printf("%d", i);
y   Aorder(i-1);
z   printf("%d", i);
    }
}
```

Compute for i = 3

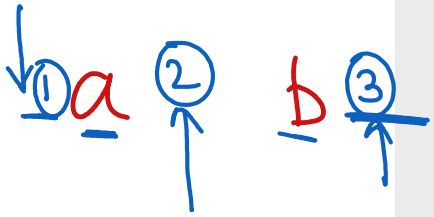




An expression can be represented in –

operands: a, b

- Prefix notation (Polish notation) operator operand1 operand2
- Infix notation operand1 operator operand2
- Postfix notation (Reverse Polish notation)
  ↳ operand1 operand2 operator

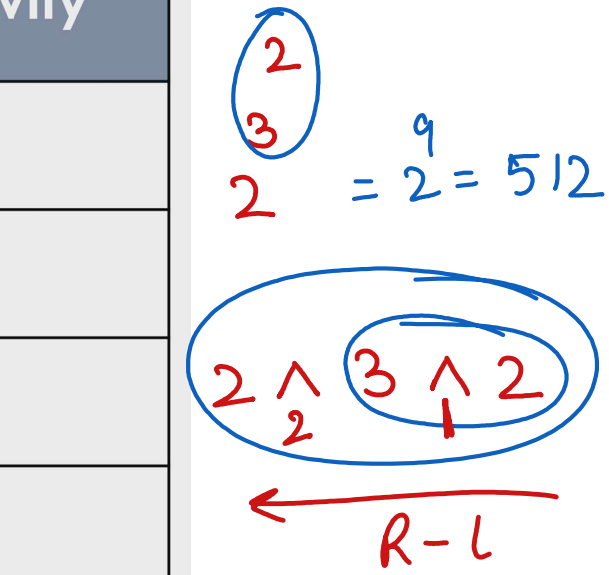


Symbol	Priority	Associativity
( ), [ ], { }	1 (Highest)	
^ (exponential)	2	R-L
* , /	3	L-R
+ , -	4 (lowest)	L-R

(2 - 3) = 4

5 - 4

= 1



Infix to postfix conversion -

op1 operator op2

Infix expression	Postfix expression
A + B	A B +
A + B * C	A B C * +
A + B - C	A B + C -
A ^ B ^ C	A B C ^ ^
(A + B)*(C – D)	A B + C D - *

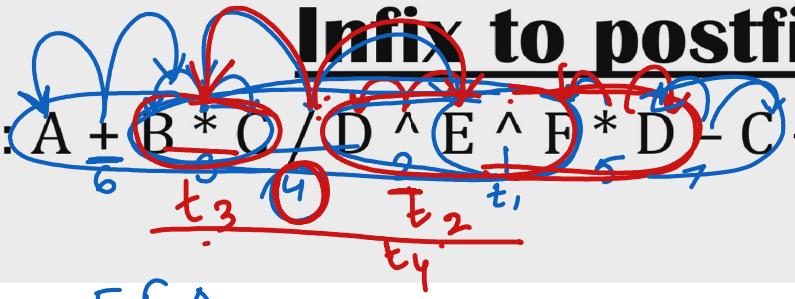
op1 op2 operator

NOTE

while conversion, only order of operators is going to change not the order of operands.

Infix to postfix conversion -

Infix : A + B * C / D ^ E ^ F * D – C + B



$t_1 = EF\wedge$

$t_2 = Dt_1\wedge = DEF\wedge\wedge$

$t_3 = BC*$

$t_4 = t_3\ t_2/ = BC*DEF\wedge\wedge/$

$t_5 = t_4 D* = BC*DEF\wedge\wedge/D*$

$t_6 = At_5+ = ABC*DEF\wedge\wedge/D*+$

$t_7 = T_6 C- = ABC*DEF\wedge\wedge/D*+C-$

$t_8 = T_7 B+ = ABC*DEF\wedge\wedge/D*+C-B+$ (Postfix expression)

Infix to postfix conversion -

Infix: A + B / C * D – E ^ F * G + H * I ^ A ^ B * C / D - E

10 4 5 11 3 6 12 7 2 1 8 9 13

Postfix

ABC/D*+ EFΛG*– HIABΛΛ*C*D/+ E –

ABC / D * + EFΛ G * – HIABΛΛ*C*D/+E –