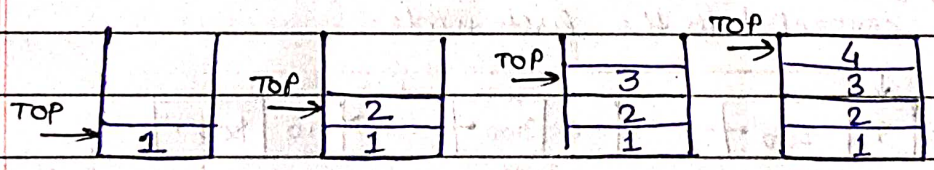
Stack

1. Stack is linear data structure that follows the LIFO (Last in first out) principle.
2. Stack has one end for I/O operation.
3. Contains only one pointer i.e. top
4. Called stack because in real world it works as stack, piles of books.
5. Stack is abstract data type with pre-defined capacity. (Limited size)
6. Follows FILO (First in last out) and LIFO (Last in first out) principle.

Working of Stack :-



Push 1 Push 2 Push 3 Push 4

Stack Operations :-

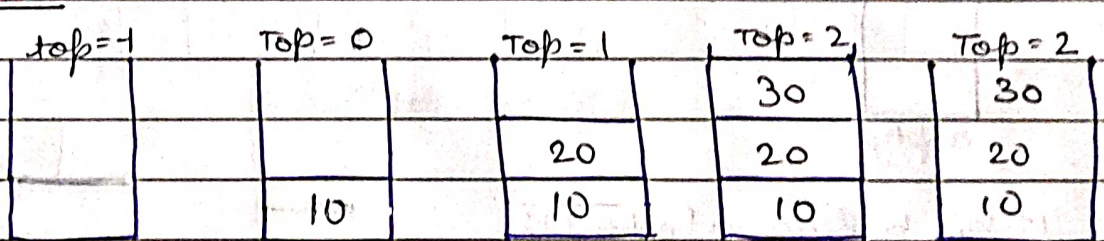
1. push() :- Insert element at top causing overflow if full
2. pop() :- Remove top element causing underflow if empty
3. isEmpty() :- check if stack is empty
4. isFull() :- check if stack is full
5. peek() :- display or return top element
6. count() :- return ~~top~~ total number of items
7. change() :- change element at given position → change (pos, item)
8. display() :- print all elements

Implementation of Stack in Java without using inbuilt classes :-

```
class Stack{
    private int maxSize; //Store max size of stack
    private int[] stackArray; //Array to store elements of stack
    private int top; // Keep track of the index of top element in stack.
    //Constructor to initialize stack with a specified size
    public Stack(int size){
        maxSize = size; //Assign specified size
        stackArray = new int[maxSize]; // create new integer array with specified size to store stack elements
        top = -1; // indicating empty stack.
    }
}
```

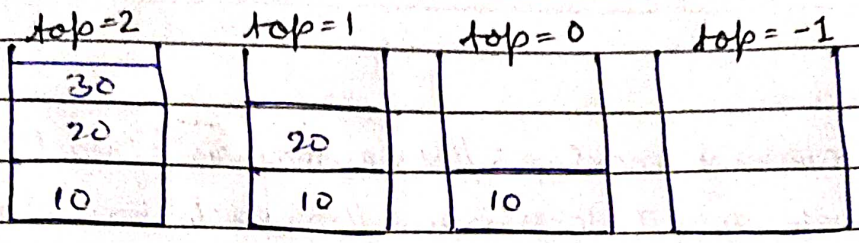
Implementation of operations in Stack :-

1. PUSH() :-




```
public void push(int item){
    if (top == maxsize -1){
        System.out.println("Stack overflow");
    } else {
        stackArray[++top] = item; // increment top and insert the element.
    }
}
```

2. POP() :-

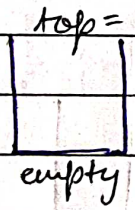



```
public int pop(){
    if (top==-1){
        System.out.println("Stack underflow"); return -1;
    }else{
        return stackArray[top--];
    }
}
```

return default value

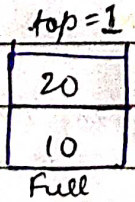
Return the top and decrement the top.

3. ISEMPTY :-




```
public boolean isEmpty(){
    return (top==-1);
}
```

4. ISFULL :-

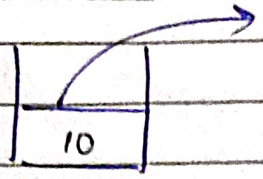


maxSize = 2

top = maxsize - 1 ⇒ 2-1 top = 1

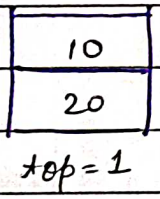
```
public boolean isFull(){
    return (top==maxSize-1); }
```

5. PEEK() :-




```
public int peek(){
    if(top==-1){
        System.out.println("stack is empty");
    return -1;
    }else{
        return stackArray[top];
    }
}
```

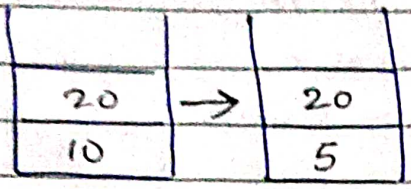
6. COUNT() :-



return top+1 = 1+1 i.e. 2

```
public count(){
    return top+1;
}
```

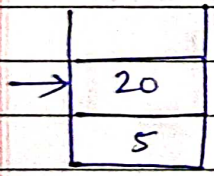
7. CHANGE ():-



change item at position '0' from 10 to 5.

```
public void change(int pos, int item){
    if (pos >= 0 && pos <= top){
        stackArray[pos] = item;
    } else{
        System.out.println("Invalid position");
    }
}
```

8. Display :-



Display all elements of stack.

```
public void display(){
    if (isEmpty()){
        System.out.println("stack is empty");
    } else{
        System.out.print("Stack: ");
        for(int i=0; i <= top ; i++){
            System.out.println(stackArray[i]);
        }
}}}
```

Application of Stack :-

1. Symbol balance :- balances brackets using stack
2. Reverse String :- reverse a string using stack
3. UNDO/REDO :- editor states using two stacks undo/redo.
4. Recursion :- supports function recursion using stack.
5. DFS :- Graph traversal using stack.
6. Backtracking :- Retraces path using stack.
7. Expression Conversion :- Infix, prefix, postfix expression using stack.
8. Memory Management :- Allocates/releases memory in a function call stack.

Conversion of Infix to Postfix notation :-

operator between two operands.

(p+q)*(r+s)

a-z, 0-9

<operand> <operator> <operand> ⇒ <operand><operand><operator>

+, -, /, *, ^, %

+, -, /, *, ^, %

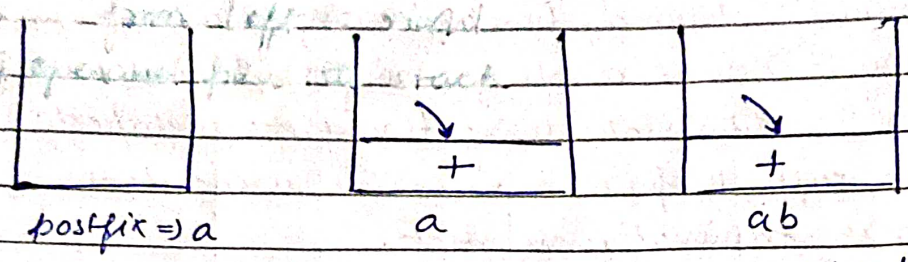
Order of operator Precedence :-

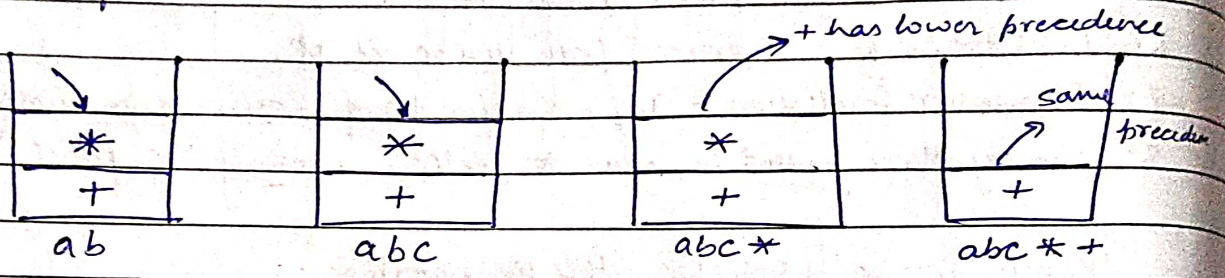
Parenthesis	(), {}, []
Exponents	^
Multiplication and Division	*, /
Addition and Subtraction	+, -

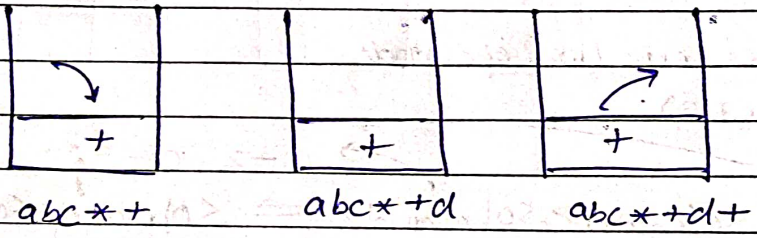
P.T.O

Evaluation of Infix to Postfix Expression :-

Expression => a+B*c+d

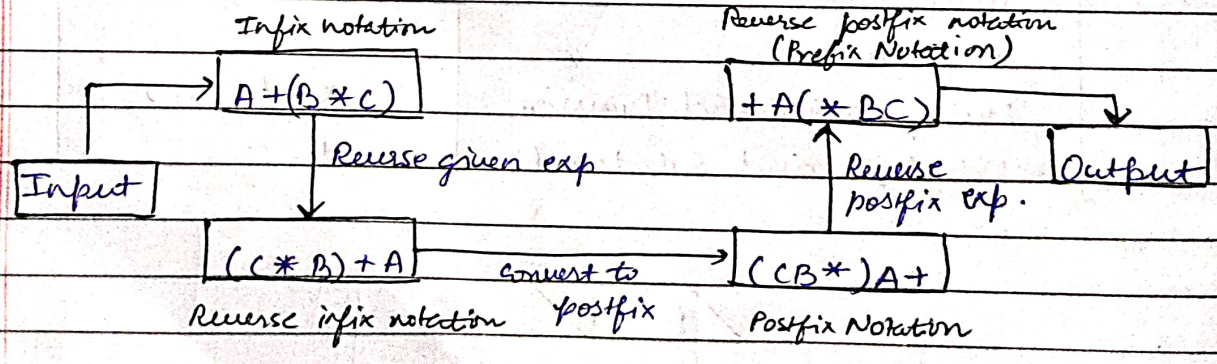






abc*+d+

Conversion of Infix to prefix :-



Infix To Postfix Conversion :-

Rules for infix to postfix using Stack :-

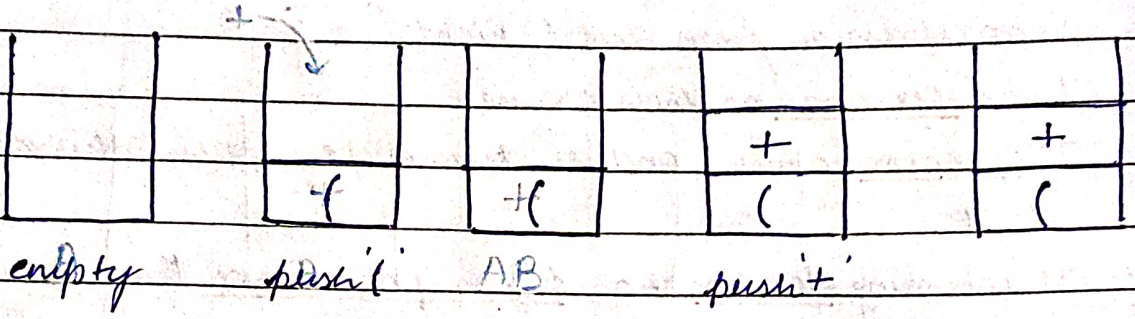
1. Scan expression from left to Right.
2. Print operands as they arrive
3. If operator arrives and stack is empty, push this operator onto the stack.
4. If incoming operator has higher precedence than the top of the stack, push it on stack.
5. If incoming operator has lower precedence than the top of the stack, then pop and print the top. Then test the incoming operator against the new top of stack.
6. If incoming operator has equal precedence with top of stack, use associativity rules.
7. For associativity of left to right - POP and print the top of stack, then push incoming operator.
8. For associativity of right to left - PUSH incoming operator on stack.
9. At the end of Expression, POP & print all operators from the stack.

If brackets :-

If incoming symbol is '(' PUSH into stack.
If incoming symbol is ')' POP the stack and print operators till '(' is found and discard it.
If top of stack is '(' push operator on stack.

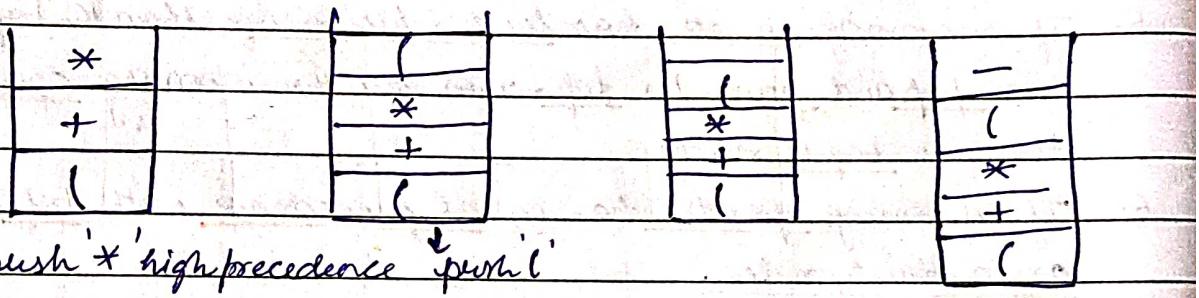
Representation :-

Expression :- (A+B*(C-D))/E



empty push '(' AB push '+'

A A AB

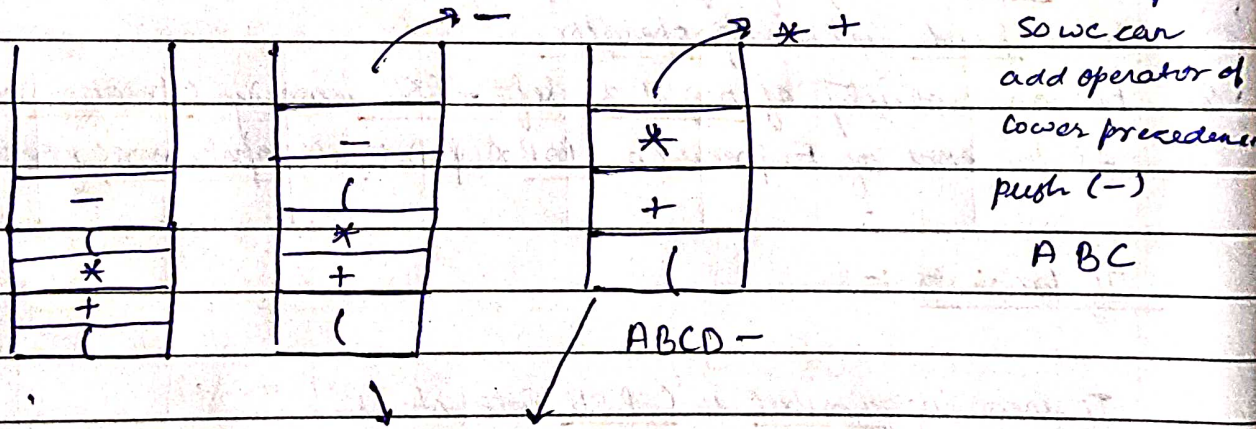


push '*' high precedence push '('

AB AB ABC

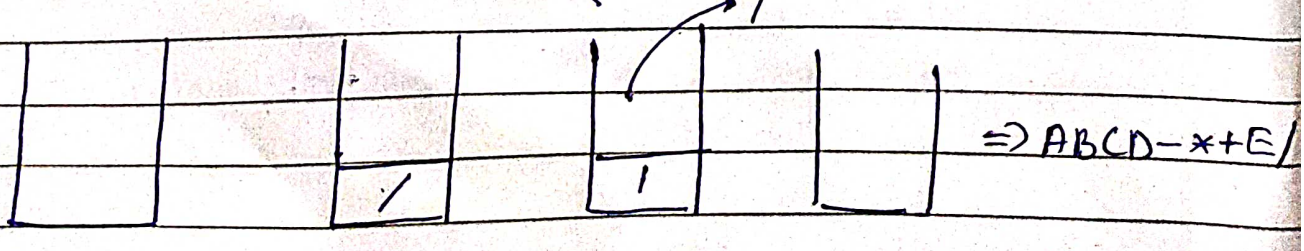
Since '(' symbol so we can add operator of lower precedence push (-)

ABC



ABCD ABCD-

Now the incoming symbol is ')' pop all until '(' is found



ABCD-*+ push /

⇒ ABCD-*+E/

Infix to Prefix Conversion :-

Rules of infix to prefix using Stack :-

1. Reverse infix expression & swap '(' to ')' and ')' to '('
2. Scan expression from left to Right.
3. Print operands as they arrive.
4. If incoming operator arrives & stack is empty, Push to stack.
5. If incoming operator has higher precedence than the top of stack, push it on stack.
6. If incoming operator has equal precedence with Top of stack & incoming operator is '^' pop & print Top of stack - Then test the incoming operator against the new Top of stack.
7. If incoming operator has equal precedence with top of stack Push it on stack.
8. If incoming operator has lower precedence than the top of the stack, Then pop and print the top of stack. Then test the incoming operator against the new top of stack.
9. At the end of expression, pop & print all operators from the stack.
10. If incoming symbol is '(' push it to the stack.
11. If incoming symbol is ')' pop the stack & print operators till '(' is found or stack empty. Pop out that '(' from stack.
12. If top of stack is '(' push operator on stack.
13. At the end reverse output string again.

g	/	(	f	^	e	^	d	*	(	c	-	b	+	a	)	)
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Date:
Page:

42

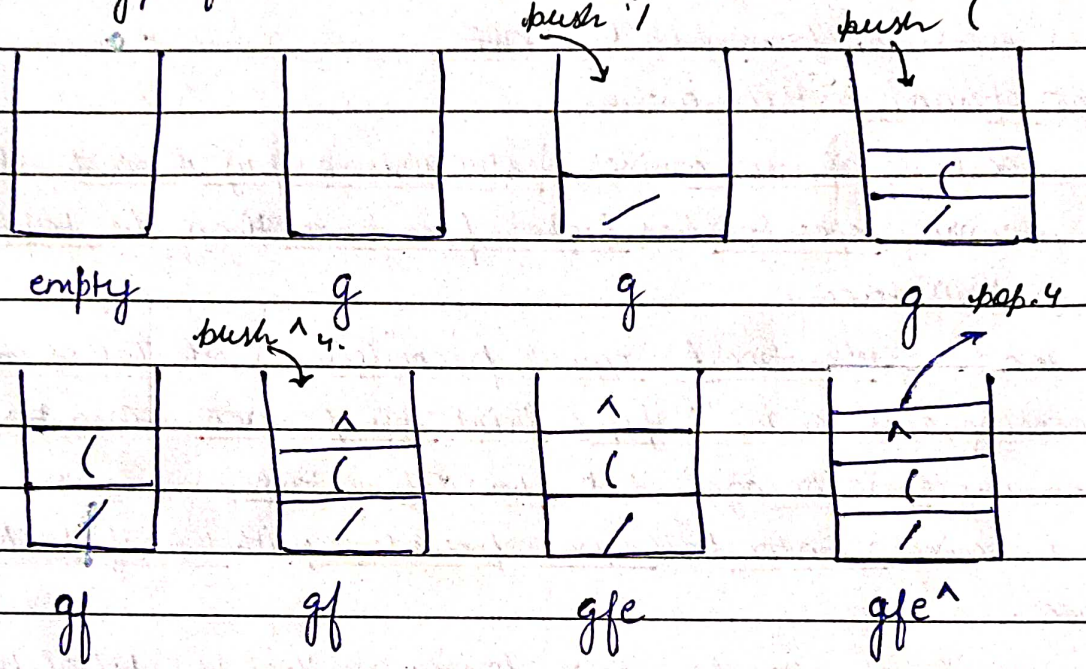
Representation :-

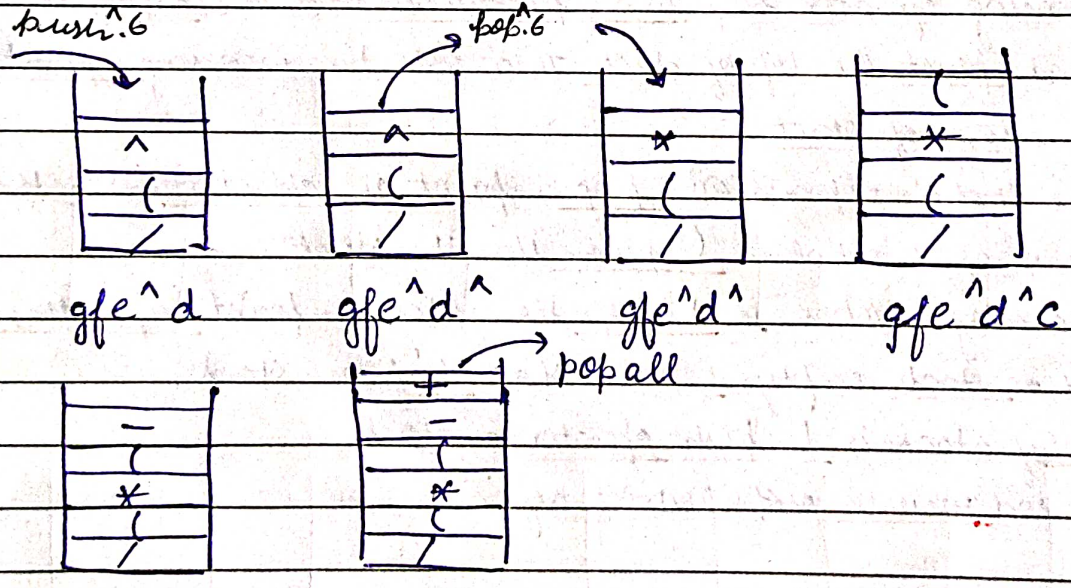
+, - = 1
*, / = 2
^ = 3
(,) = -1

Expression :-

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

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





Reverse it

⇒ /*-+abc^d^efg

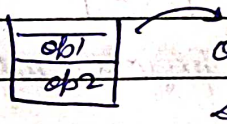
Postfix to Infix :-

Rules for postfix to infix conversion :-

1. Scan postfix expression from left to right
2. If the incoming symbol is a operand, push it onto the stack
3. If the incoming symbol is a operator, pop 2 operands from the
   - stack, ADD this incoming operator between two operands.
   - ADD '(' and ')' to the whole expression & push this whole new expression string back into the stack.
4. At the end POP & PRINT the full INFIX expression from the stack

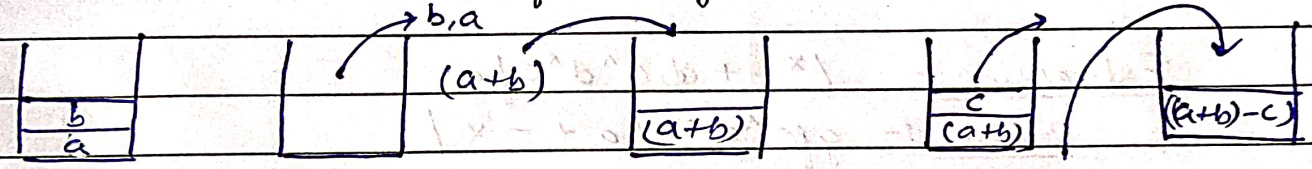
Representation :-

Note :-

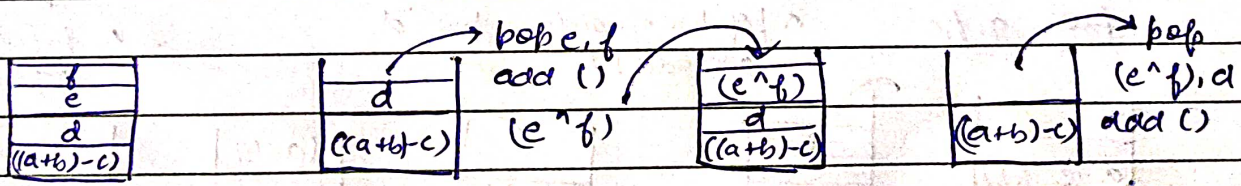


op2 + op1
Arrange ↑

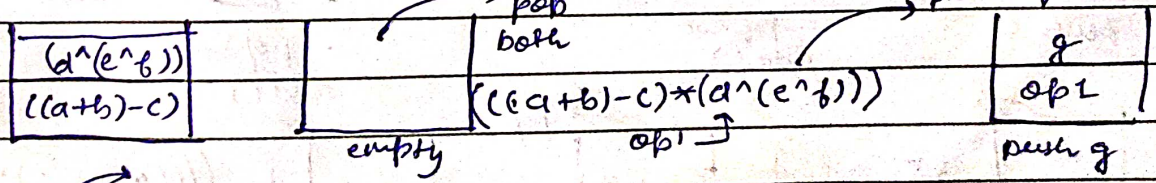
Expression :- ab+c-def^^*g/

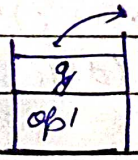


push a,b — pop b,a since operator is next then add () brackets — push a+b



push d,f





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

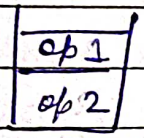
Prefix to Infix Conversion :-

Rules for conversion :-

1. Reverse the prefix expression or Scan from Right to left
2. Same as step 2 - 4 in Postfix to Infix Conversion.

Note *

In postfix to infix conversion we used to put the top of the stack operand suppose top = op2 and top-1 = op1



operator

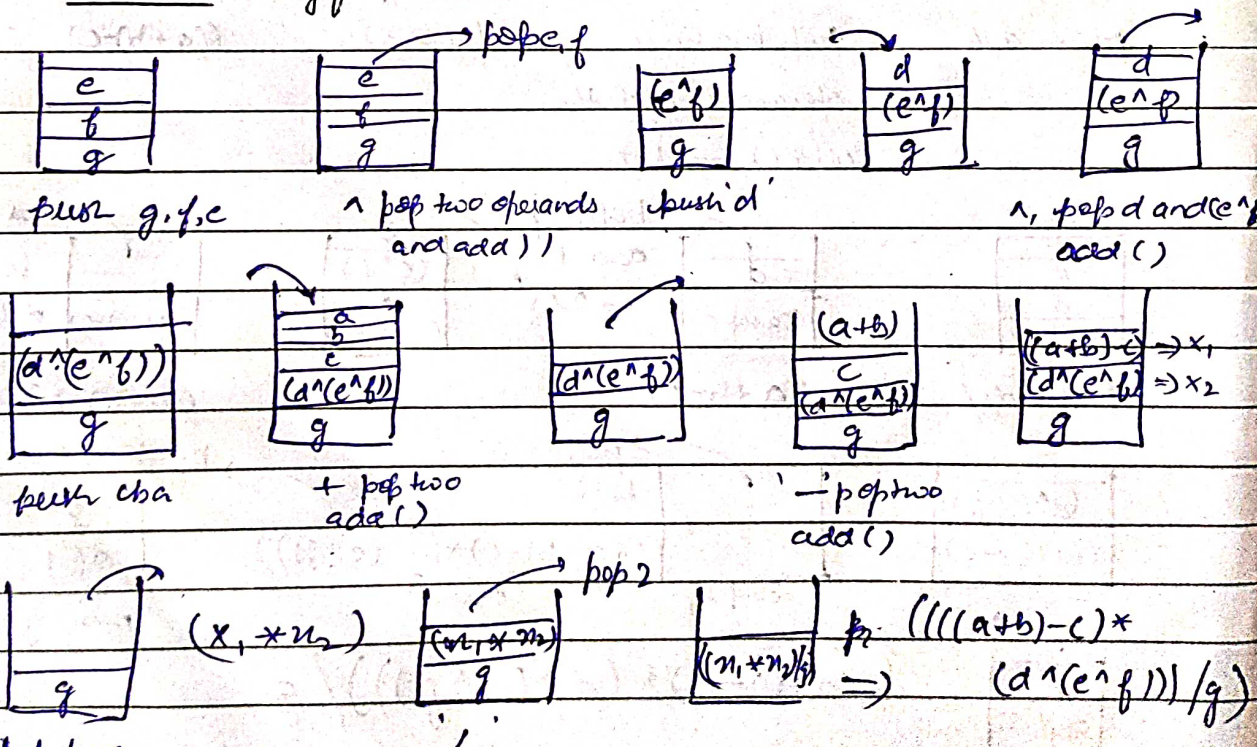
so we use to do op2 + op1 → operand

But in prefix to infix we do op1 + op2

Representation :-

Expression :- /*-+abc^d^efg

Reverse :- gfe^d^cba+-*/

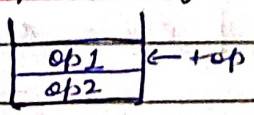


Postfix to prefix conversion :-

Rules for conversion :-

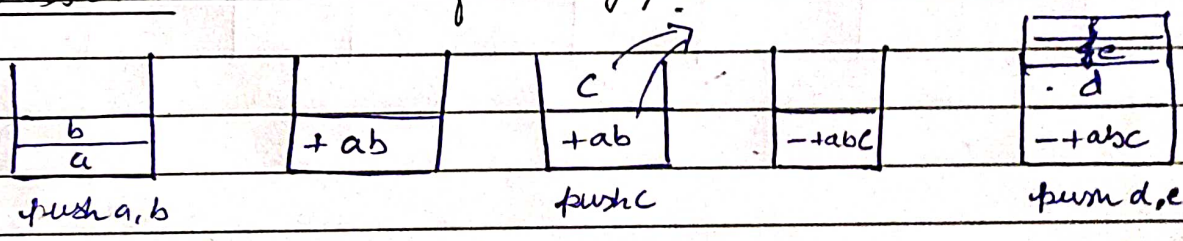
1) Scan Postfix expression from left to right
2) If incoming symbol is operand, push it onto the stack
3) If the incoming symbol is a operator, pop 2 operands from the stack.
   ADD this incoming operator at the start of the two operands.
   PUSH this whole new expression string back into the stack.
4) At the end POP & print the full prefix expression from the stack.

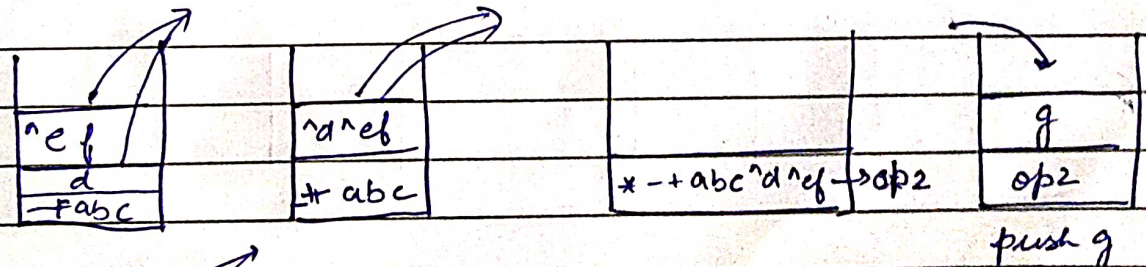
(+, −, /, *, ^)
exp => postfix operator + op2 + op1

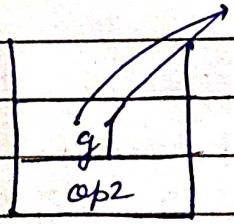


Representation :-

Expression :- ab + c − de f ^ ^ * g /







/ * − + abc ^ d ^ e f g

Prefix to postfix conversion :-

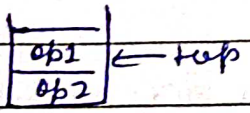
Rules for prefix to postfix conversion :-

1. Reverse the prefix string or scan from right to left.
2. Same as step 2-4 in postfix to prefix conversion

Note *

In postfix to prefix

exp ⇒ operator + op2 + op1



In prefix to postfix

exp ⇒ op1 + op2 + operator

Recursion in Python :-

1. Recursion is a programming technique where a function call itself in order to solve a smaller instance of the same problem.

It involves breaking down a problem into smaller sub-problems and solving them recursively.

2- **Base Case :-**

Every recursive function should have a base case, which is a condition that, when met, prevents further recursive calls. The base case ensure that the recursive loop stops and function returns a result.

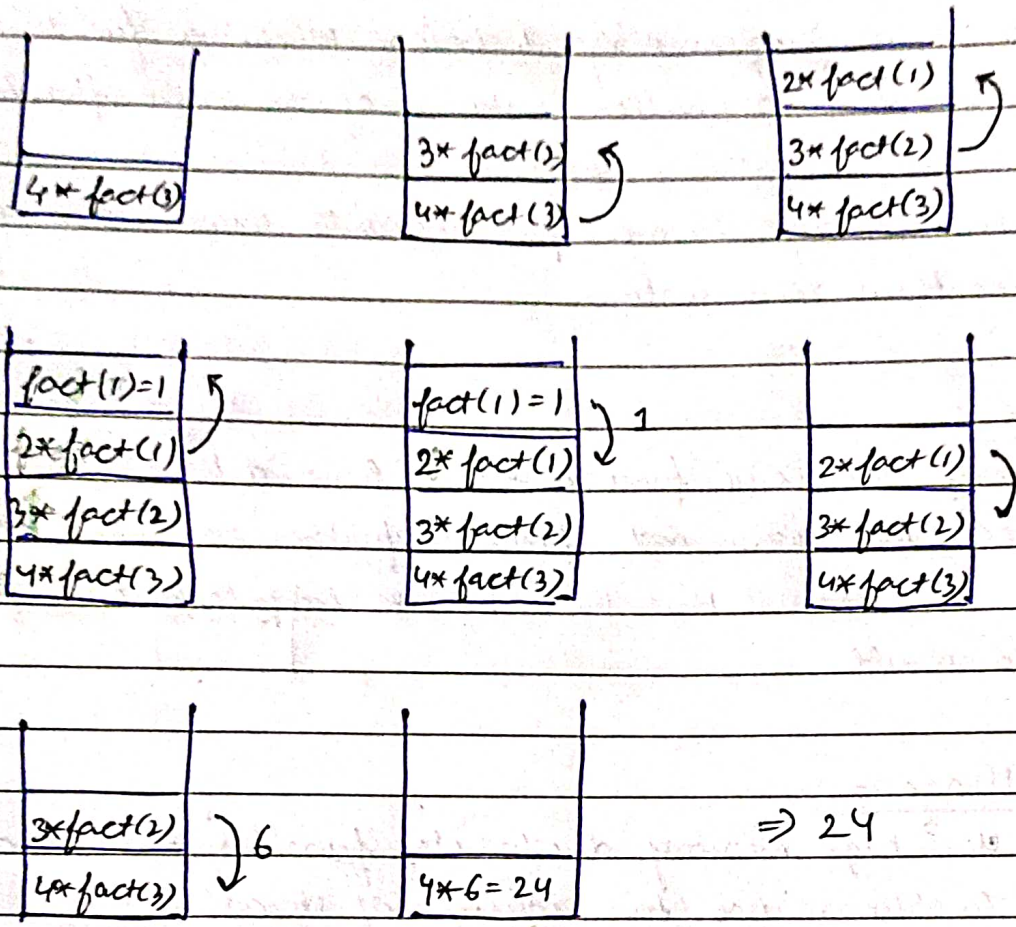
3- **Memory Usage :-**

→ uses high memory due to its function call stack.
→ To optimize use tail recursion of iterations.

4. **Example and Stack function :-**

```
def factorial (n):
    # Base Case
    if n == 0 or n == 1:
        return 1
    # Recursive Call
    else:
        return n * factorial (n-1)
```

Stack Representation of Factorial Recursion :-



Tail Recursion :-

Special form of recursion where recursive call is the last operation performed by the function before returning result.
→ no further operations to be performed after the recursive call returns.

```
def f_t_r (n, accumulator = 1):
    if n == 0:
        return accumulator
    else: # multiply and continue.
        return f_t_r(n-1, n*accumulator)
```

the function multiplies 'n' with the 'accumulator' and call itself with updated values. The multiplication happens before recursive call making it tail recursive.

Removal of Recursion :-

Recursive solution may lead to stack overflow errors for large inputs due to the limited size of the call stack.

Iterative solution is more memory-efficient and might perform better for certain scenarios.

```
def f_i(n):
    result = 1
    for i in range(1, n+1):
        result *= i
    return result
```

Consideration for Removing Recursion :-

1. Performance : Evaluate recursive vs. iterative efficiency.
2. Readability : Recursive : Elegant, easy to understand
   Iterative : Straightforward in some cases.
3. Memory Usage : Iterative uses less memory, no call stack
4. Tail call Optimization : watch for stack overflow.