Applications of Stack

* Evaluating arithematic expressions
* Balancing the symbols
* Function calls
* Towers of Hannoi
* 8-Queens Problem.
* Applied Recursion function

Diffrent Types of notation to represent arithmetic expression

There are three different ways of expressing algebraic expressions.

* Infix notation
* Postfix notation
* prefix notation

Infix notation:-

The arithematic operator appears between 2 operands to which it is being applied.

Eg A/B+C

Postfix

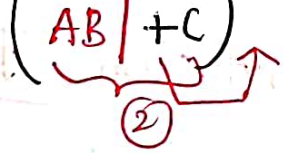
The arithematic operator appears directly after two operands to which it applies. also called reverse polish notation.

Eg. ab+

Examples Convert infix to postfix

① $((A/B)+C)$ ① Apply it on inner brackets

$(AB/+C)$

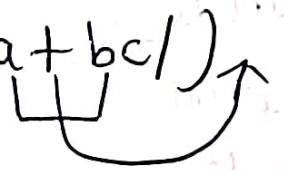


$AB/C+$

② $a+(b/c)$

(i) $(a+(b/c))$

(ii) $(a+bc/)$



$\boxed{abc/+}$

(3) a + b * c

(i) Apply brackets based on Priority

(a + (b * c))

(ii) Inner bracket (b * c)

(a + (b * c))

(a + bc *)

a bc * +

(4) (a + (b * (c - d) / (p - r)))

(i) (a + (b * cd -) / (p - r))

(ii) (a + (b * cd -) / pr -)

(iii) (a + (bcd - * pr - /))

(iv) (abcd - * pr - / +)

abcd - * pr - / +

Prefix :-

The arithmetic operator is placed before the 2 operands to which it applies also called polish notation.

Eg +ab

① (a+b)

+ab

② (a+(b*c))

(a+*(bc))

+a*bc

③ ((b+((c*3)/2))-4)

a) ((b+(*c3/2))-4)

((b+/*c32)-4)

(+b/*c32-4)

-+b/*c324

Exercise

Convert given infix notation to postfix & prefix

① A+B

a+b*c+(d*e)

② A+B-C

A+B*(C-D)|(Q-R)

③ (A+B) * (C-D)

(A+B)*C-(D-E)*(F+G)

④ A$B*C-D+E|F/(G+H)

⑤ ((A+B)*C-(D-E)) $ (F+G)

⑥ A-B/(C*D$E)

⑦ (A+B) * (C-D) $ (E*F)

⑧ (A+B)*(C$(D-E)+F)-G

⑨ A+(((B-C)*(D-E)+F)/G)$(H-J)

Evaluating Arithematic Expressions

To evaluate the arithematic expression, First convert the infix notation to postfix notation and then evaluating postfix using stack.

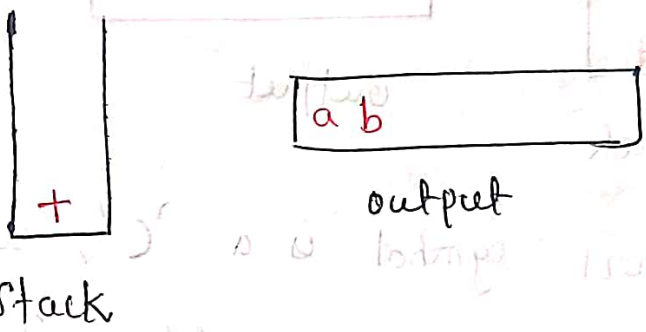
Infix to postfix notation

* If the character is an operand placed on to the output.
* If the character is an operator pushed on to the stack.
* If the stack operator has higher or equal priority than input operator, pop that operator from the stack & placed on to the output.
* If the character is left parenthesis pushed on to the stack.
* If the character is right parenthesis pop all the operators from the stack till encounters left parenthesis, discard both parenthesis in the output.

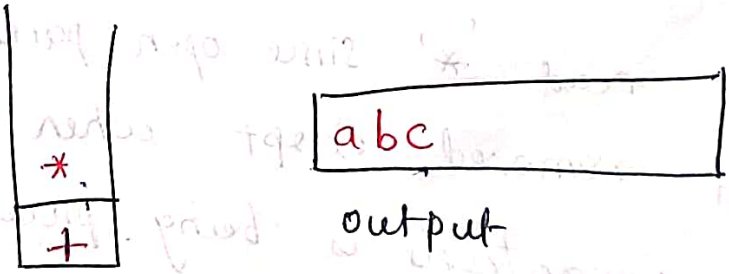
Input:- a+b*c+(d*e+f)*g

(i) First, the symbol 'a' is read, so it is passed through to the output. Then '+' is read and pushed on to stack.

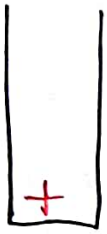
Next 'b' is read and passed through to the output.



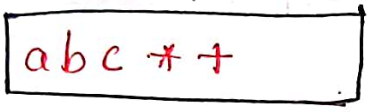
(ii) Next '*' is read. The top entry on the operator stack has lower precedence than '*', so nothing is output and '*' is put on the stack. Next 'c' is read & output



(iii) The next symbol is a '+'. checking the stack, we find that we will pop a '*' and place it on the output. pop + also which is not of lower but equal priority, on to the stack. and then push the '+'



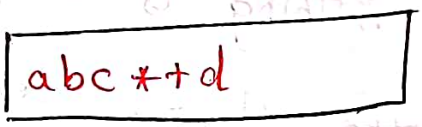
stack



output

(iv) The next symbol is a '(', which is being of highest precedence, is placed on the stack. Then 'd' is read & output

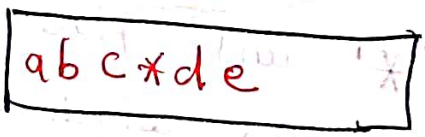
Stack



output

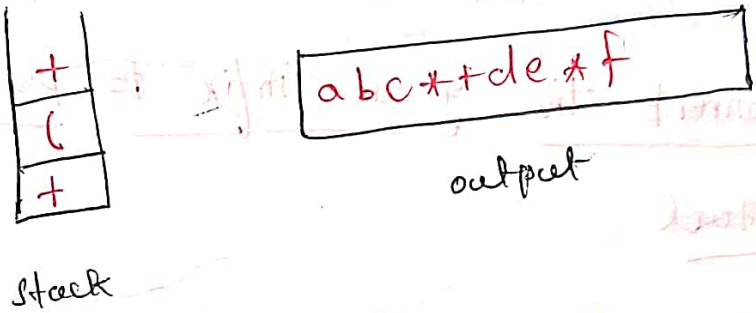
(v) Next read '*' since open parentheses do not get removed except when a closed paranthesis is being processed, there is no output. Next 'e' is read & output.

Stack

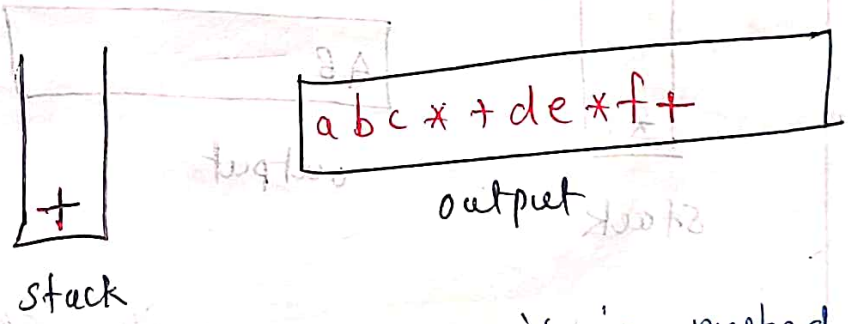


output

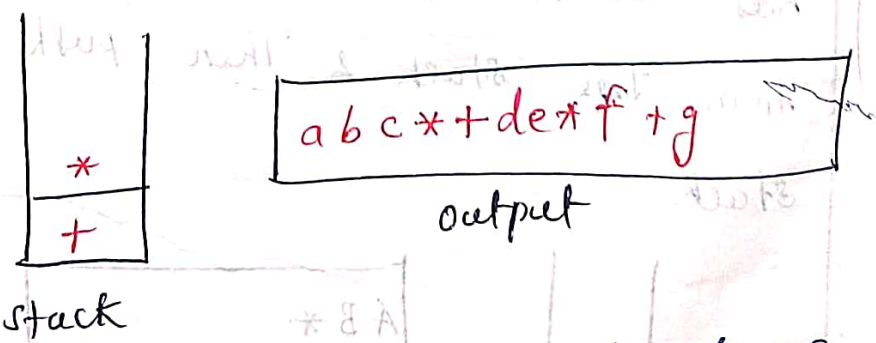
(VI) The next symbol read is a '+' we pop and output '*' and then push '+' Then we read & output f.



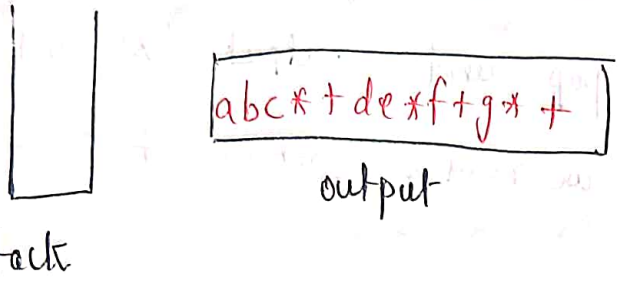
(VII) Now we read ')' so the stack is emptied back to the '('. we output a '+'



(VIII) we read a '*' next, it is pushed on to the stack. Then g is read and output



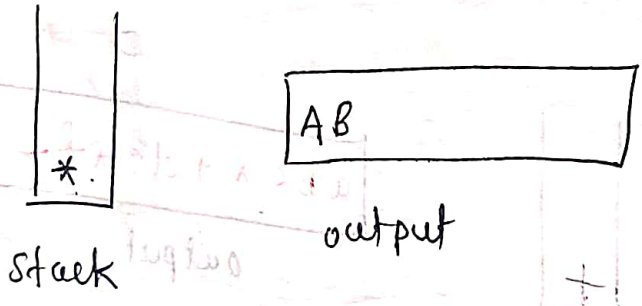
(XI) The input is now empty, so we pop and output symbols from the stack until it is empty.



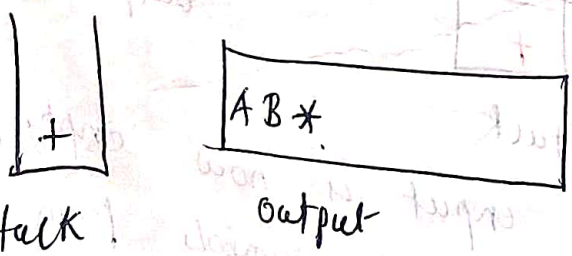
Convert the given infix to postfix using stack

A * B + (C - D / E)

(i) Insert 'A' put in output and read '*' push it on to stack. Then read 'B' & output

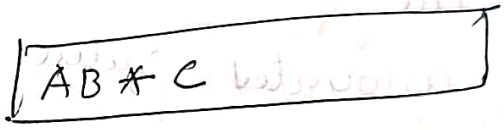


(ii) Read '+' compare with operator already exists in stack. compared to '*', '+' has lowest precedence. Then pop '*' from the stack & Then push '+' into stack



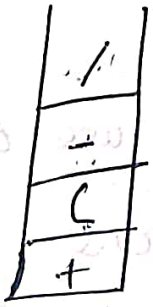
(iii) Next read the symbol 'C' and push it on to stack. then read C & output

stack

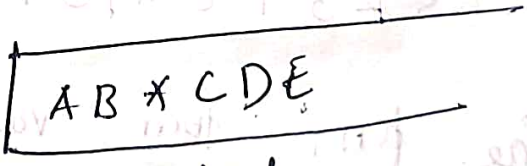


output

(iv) Then read '-' symbol push it to stack. then read 'D' and output. Next read the symbol '/', it has higher precedence then '-' so push '/' in to stack. Then read E & output



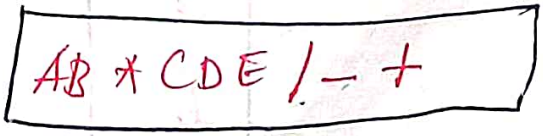
stack



output

(v) Next read ')' parenthesis so all operators inside open and close brackets pop up. Finally + symbol popped

stack



output

Evaluating postfix Expression

* Read one character at a time, if the character is an operand push its associated value on it to the stack.

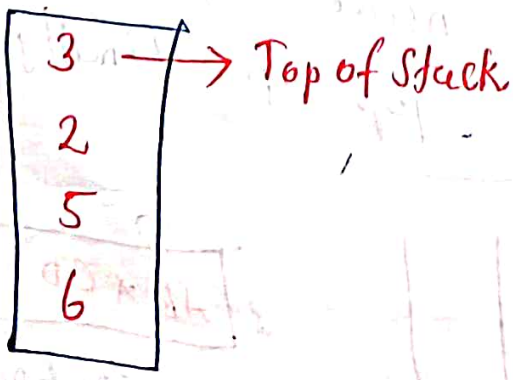
* If the character is an operator, pop two values from the stack apply the operator to them and then push the result on to the stack.

Example 1 => postfix given as an input

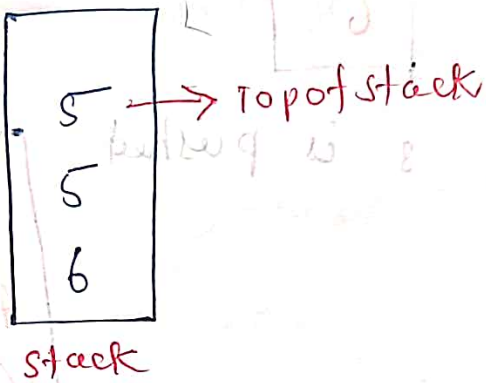
A B C D + E * + D + *

6 5 2 3 + 8 * + 3 + *

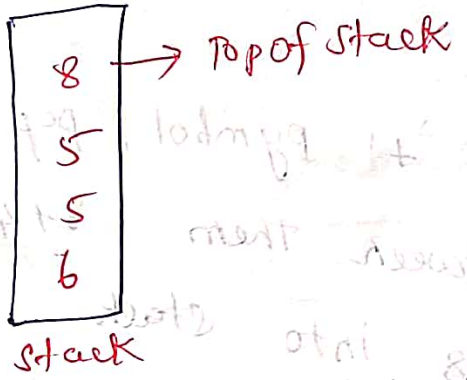
① The first four values are operands so push the elements one by one. The resulting stack is



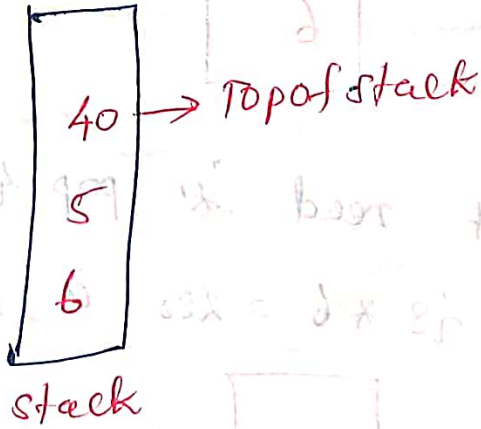
(2) Next a '+' is read, so 3 & 2 are popped from the stack and their sum, 5 is pushed.



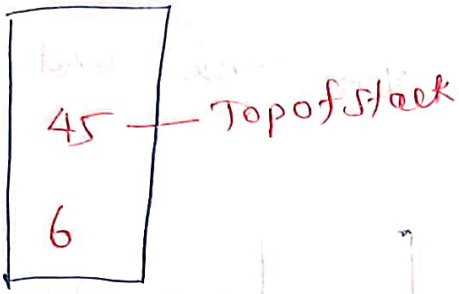
(3) Next 8 is pushed



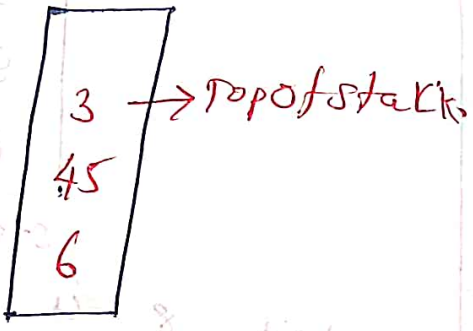
(4) Next read '*', so pop 8 and 5 from the stack and 5*8 = 40 is again pushed



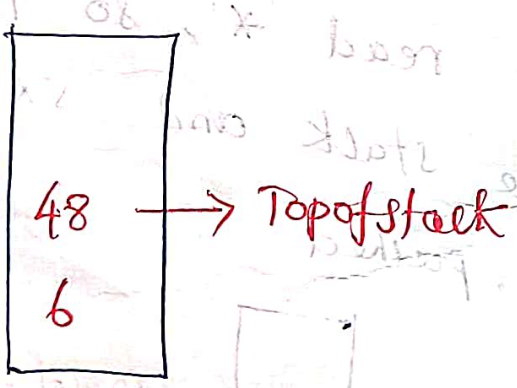
(5) Next read '+' pushed on to the stack. apply + between 40+5=45 pushed to stack.



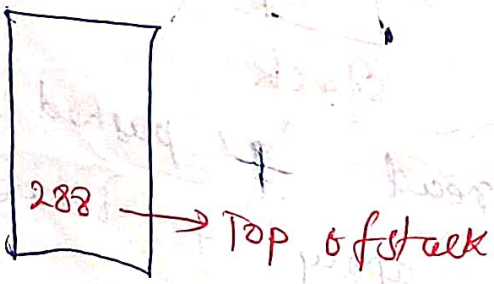
⑥ Now 3 is pushed



⑦ Read '+' symbol, pop 3 & 45 apply '+' between them 3+45 = 48.
push 48 into stack.



⑧ Next read '*' pop 48 & 6 from stack
then 48 * 6 = 288 is pushed.

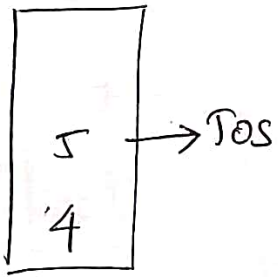


The time to evaluate a postfix expression is O(N), because processing each element in the input consists of stack operations and thus takes constant time

Example 2

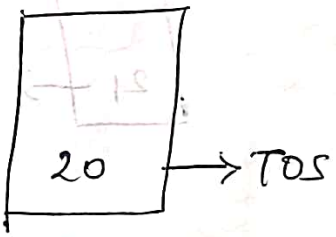
AB * CDE / - +

45 * 582 / - +

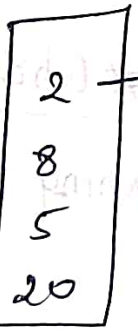
① 

② Read * pop 4 & 5

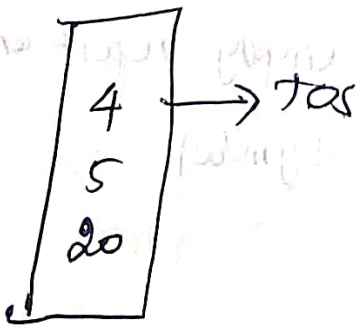
4 * 5 = 20 push



③ Read value 5, 8, 2 one by one

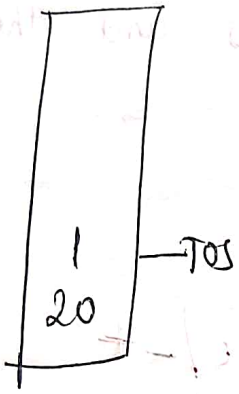


④ Read '/' pop 2 & 8 8/2 = 4. push 4



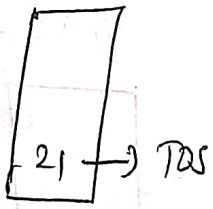
⑤ Read '-' symbol pop 5 & 4

5-4 = 1. push value 1



⑥ Read '+' symbol pop 20 & 1

20+1 = 21. push 21



Balancing the Symbols

* Read one character at a time until it encounters the delimiters #(hash)
* If the character is an opening symbol pushed on to the stack.
* If the character is a closing symbol and if the stack is empty report an error missing opening symbol

* If it is a closing symbol and it has corresponding opening symbol in the stack, pop it from the stack otherwise report an error mismatch symbols.

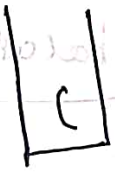
* At the end of file if the stack is not empty report an error as missing closing symbols.

① (a+b) #

①Read '(' push it on to stack

② a ⇒ 

③ + ⇒ 

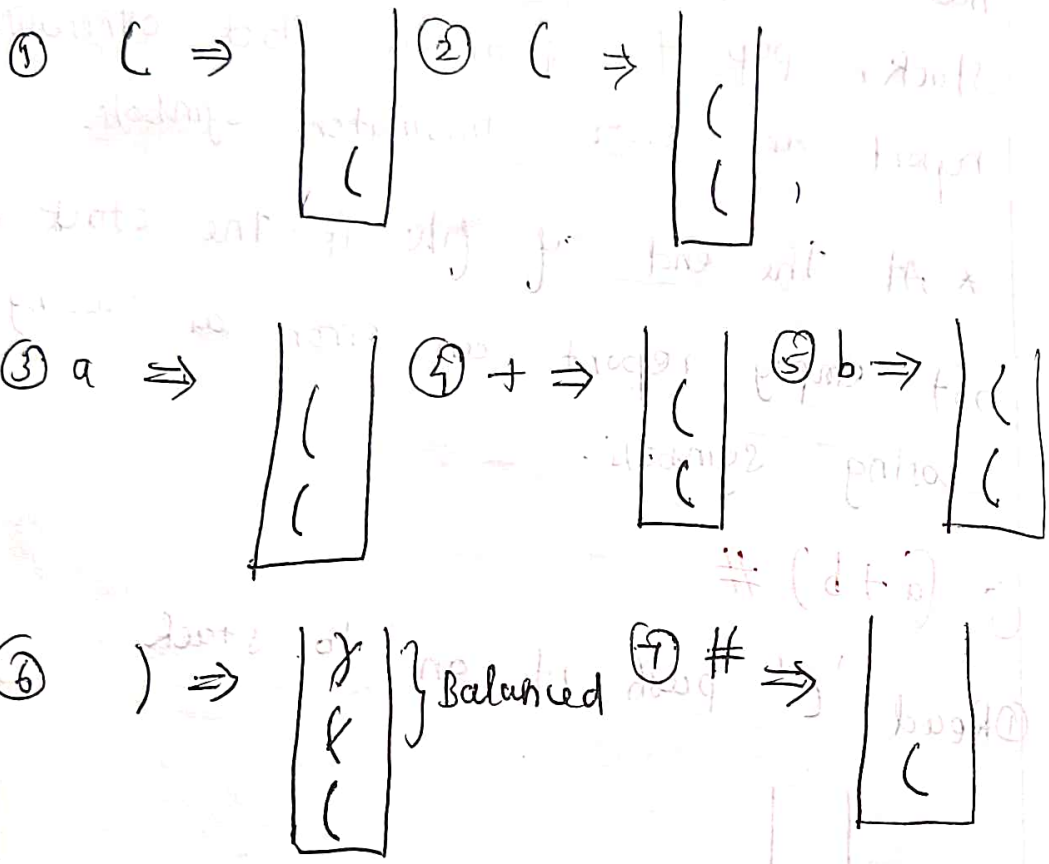
④ b ⇒ 

④) ⇒  Balanced

⑤ # ⇒  Stack is empty.

In above expression symbols are balanced.

② ((a+b)#

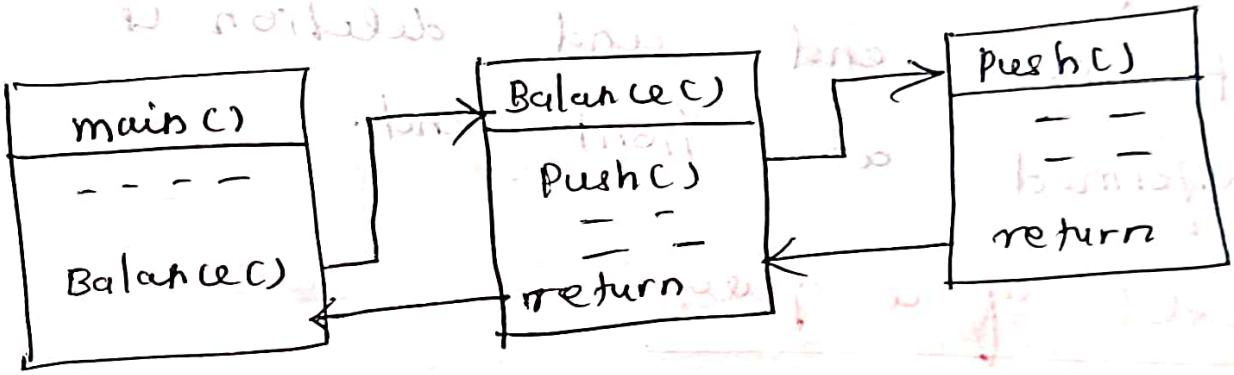


The last step Stack have a '(' symbol it doesn't find corresponding closing symbol. so it produce an error message "Un balanced parenthesis".

Function calls :-

* when call is made to a new function all the variables local to calling routine need to be saved. otherwise new function will overwrite the calling routine variables.

* Similarly the current location address in the routine must be saved.
* So the new function knows were to go after it is completed.




```
int fact (int n)
{
  int s;
  if (n==1)
  return 1;
  else
  s = n * fact (n-1);
  return (s);
}
```

Applications of Queue:-

* Batch processing in OS
* To implement priority queues
* priority queues can be used to sort the elements using heap sort
* Simulation
* Queue in theory
* Computer networks where the server takes the job of client as per the Queue strategy.