2. THE STACK

STACK

A linear data structure, in which data items are inserted and deleted at one end, is termed as stack. The end that make the insertion and deletion operation possible is called the top of stack (TOS). The insertion operation is called "PUSH" while delete operation is called "POP".

In Stack, data is stored and retrieved in a LIFO manner, that means the item that has been inserted at the last is retrieved first from the stack.

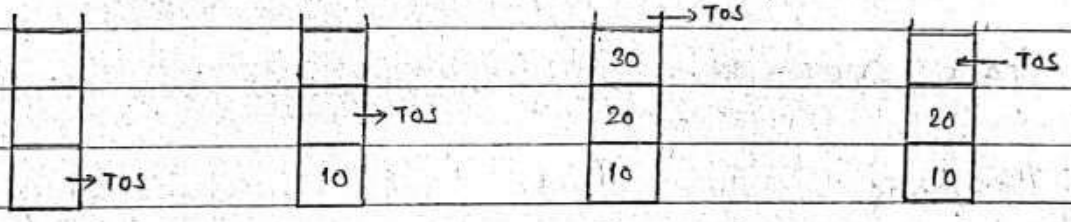


Fig (a) Empty Stack

Fig (b) Stack after inserting 10

Fig (c) Stack after inserting 10, 20, 30

Fig (d) Stack after deleting 30

STACK TERMINOLOGY

(i) Maximize

We use the term maximise to refer the maximum size of the stack and this item is not standard one.

(ii) TOP

The term TOP refers to the top of the stack. It is used to check the stack condition. Initially the top of the stack stores the zero value.

(iii) Stack empty or underflow

This is the situation when the stack ~~condition~~ contains no elements. At this condition, TOP is present at the bottom of the stack.

(iv) Stack overflow

This is the situation when the stack become full. At this condition, the TOP is present at the highest location of the stack.

STACK OPERATION

The basic operations that stack can perform are:-

(i) Create a stack
(ii) Check whether a stack is ~~for~~ empty
(iii) Check whether a stack is full
(iv) Initialize a stack
(v) PUSH an element onto the stack ie if it is not full.
(vi) POP an element from stack ie if it is not empty
(vii) Read a stack top
(viii) Print the entire stack

PUSH OPERATION

It is the process of adding new element to the TOS. After every push operation, the top is incremented by one (1).

POP OPERATION

It is the process of deleting element from the TOS. After every POP operation, the top is decreased by one (1).

ALGORITHM for PUSH operation

Let stack (max size) is an array for implementing the stack

STEP 1: Start

STEP 2: If (TOP == maxsize)
printf ("over flow");
goto step 5

STEP 3: STACK [TOP] = item;

STEP 4: TOP = TOP+1

STEP 5: Exit

ALGORITHM for POP operatim

Let stack (max size) is an array for implementing the stack

STEP 1: Start

STEP 2: If (TOP == 0) (the no item is deleted)
printf ("underflow");
goto step 5

STEP 3: Item = stack [top];

STEP 4: TOP = TOP - 1

STEP 5: Exit

STACK IMPLEMENTATION

Stack is implemented as:-

(i) Array implementation (Static implementation)
(ii) Dynamic implementation

(i) Array implementation (static implementation)

Static implementation used in array to create stack and it is a very simple technique. The size of the stack has to be declared during the program design and it is not efficient with respect to memory utilization.

(ii) Dynamic implementation

It uses the pointer to implement the stack. So it is also called link-list representation.

```
# include <stdio.h>
# include <conio.h>
# include < stdlib.h>
# define SIZE 5

void main()
   {
    int stack [SIZE];
    int data, choice, i;
    int top = 0;          // top of stack empty
```

```
while (1)
  {
  printf ("\n 1. PUSH Element");
  printf ("\n 2. POP  Element");
  printf ("\n  3. VIEW element");
  printf ( "\n  4. Exit");
  printf ("\nEnter your choice");
  scanf ("%d", & choice);
  switch (choice)
{
  case 1:
     If (top < SIZE)
      {
      printf ("\n Enter any element");
      scanf ("%d ", & data);
       stack [top] = data;
       top++;
       }
       else
        {
         printf ("\n. stack is full);
         }
       break;
```

```
case 2:
    If (top >0)
     {
      top--;
      printf("The element poped is %d", stack[top]);
     }
    else
       {
       printf("The stack is empty");
       }
    break;

case 3:
    if (top == 0)
       {
       printf("\n stack is empty");
       }
    else
       {
       for (i = top-1 ; i>=0 ; i--)
          {
          printf("%d \n", stack[i]);
          }
    break;
```

```
        case 4:
            exit (0);
            printf ("\n selection option 1-4 ");
        }
    }
}
```

exit function is a standard function. When it get executed, the program execution is immediately terminated. In this function we do not use break keyword.

The statement while(1) puts the entire logic in an infinite loop. This is necessary since the menu must keep re-appearing on the screen. Once an item is selected, an appropriate action is taken.

INFIX, PREFIX AND POSTFIX NOTATION

Any arithmetic operation can be represented in three different ways as:-

Prefix
Infix , and
Postfix

(1) Prefix notation

In the prefix expression, an operator is placed before its operand.

ie. operator operand1 operand2

eg. +AB

(ii) Infix notation

In infix notation, is the traditional way of writing arithmetic expression where an operator is placed in between its operand.

ie. operand1 operator operand2

eg. A + B

(iii) Postfix notation

In postfix notation, an operator is placed after its operand

ie. operand1 operand2 operator

eg. AB +

Some examples:

Infix	Postfix	Prefix
A+B	AB +	+AB
A+B−C	AB+C−	−+ABC
(A+B) *(C-D)	AB+CD−*	*+AB−CD

Operator Precedence.

- Exponential operator (^, $) Highest precedence
- Multiplication/division (*, /) Next highest precedence
- Addition / subtraction (+, −) " "

* Infix to Postfix conversion

Infix : A − B + C * D

Postfix : A − B + CD *

AB − + CD *

AB − CD * +

A − B + C * D

A − B CD * +

A B C D * + −

9

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

A+B *C -(D-EF/) +G

A+B*C - (DEF/-) +G

A+BC*- (DEF/-) +G

ABC*+ - (DEF/-) +G

ABC*+ DEF/-- +G

ABC *+ DEF/-- G+

Infix to prefix expression

Infix: A +B * C - D/F

A ** BC - D/F

A + *BC -/DF

+A *BC -/DF

-+A *BC/DF

INFIX TO POSTFIX CONVERSION ALGORITHM

(1) - Initialize a stack and output list as empty.

(2) - Read each symbol (operator or operand) from the infix expression.

(a) If the symbol is an operand, then append it to the output list.

(b) If the symbol is an operator and the stack is empty or the operator has higher precedence than the top operator of the stack, then PUSH the operator onto the stack.

10.

(c) Whenever an opening parenthesis encounters in the stack, then we cannot POP all the operators from the stack until its counter part (closing parenthesis) is available on the stack.

3. Repeat step 2 for every symbols of infix expression.

4. If all the symbols have been read from infix expressio and the stack is not empty, then POP all the symbo from the stack and append them to the output list

1. Eg. A+B - C *D

AB+ - C*D

T - C*D

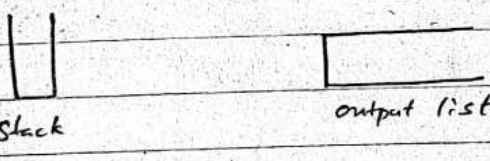
T - CD*

TS -

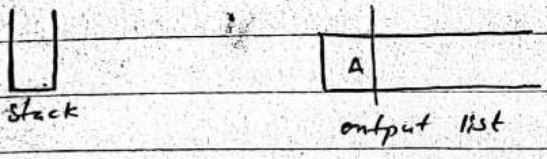
AB+CD*-

Solⁿ:

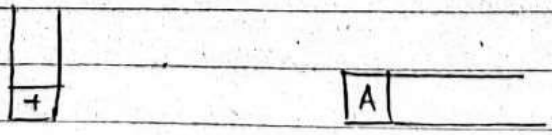
1. Initialize an empty stack and output list



2. Read A, which is an operand, so append it to the output list.

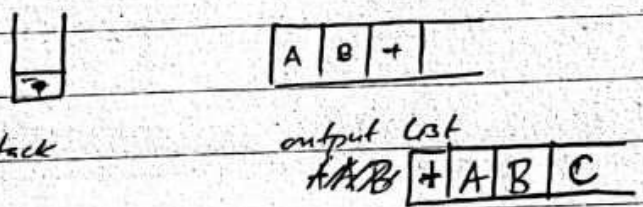


3. Read + , which is an operator, and stack is empty so p it onto the stack.

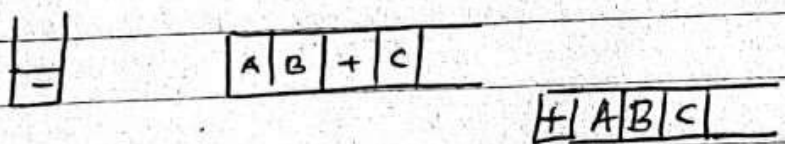


4. Read B, which is operand so append it to the output list.

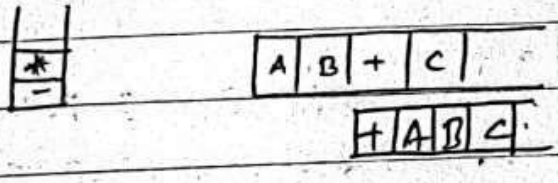
5. Read −, which is an operator and has equal precedence with + (top of stack operator) so pop '+' from the stack, append it to the output list and push '−' on the stack.



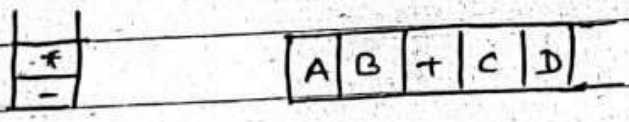
6. Read C, which is an operand, so append it to output list.



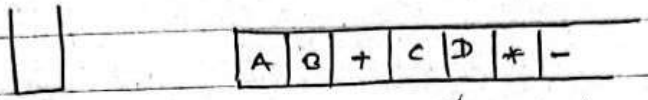
7. Read *, which is operator & has higher precedence than '−' (top of the stack) so push it on the top of stack.



8. Read D, which is an operand, so append it to output list.



9. Pop all the operators from the stack, and then append it to the output list.



Final postfix expression.

2. $\underline{a+b*c+(d*e+f)*g}$ $\Rightarrow a+\underline{bc*}+(de*+f)*g$

$\Rightarrow a+T+(S+f)*g$

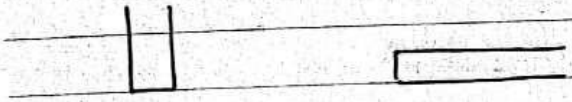
1 Initialize an empty stack and output list $\Rightarrow a+T+\underline{Sf+}*g$

$a+T+X*g$

$\underline{aT+}+Xg*$

$Y+Z$

$YZ+$

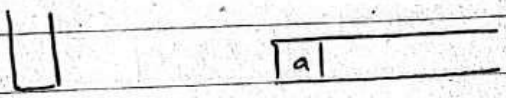


2. Read a, which is operand, so append it to the o/p list.

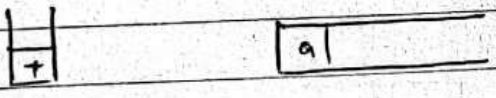
$\Rightarrow aT+Xg*+$

$\Rightarrow abc*+Sf+g*+$

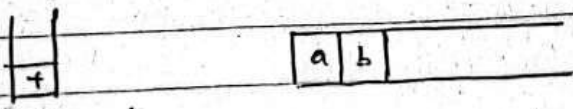
$\Rightarrow abc*+de*f+g*+$.



3. Read +, which is operator, so append it to the stack

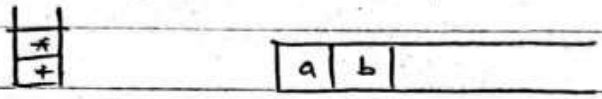


4. Read b, so append it to the output list.

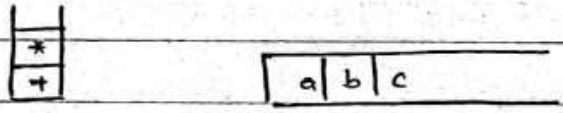


13

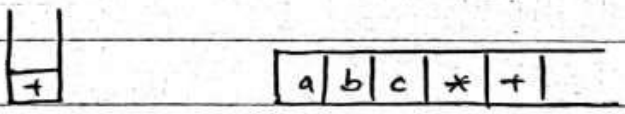
5. Read *, is an operator and has higher precedence than +, so push it onto the stack.



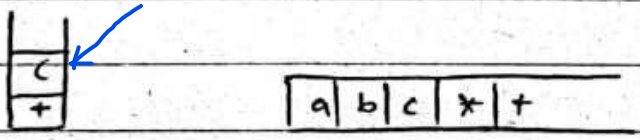
6. Read c



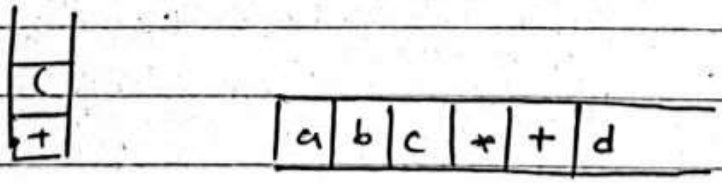
7. Read +, is an operator, has lower precedence than *, pop * from the stack and append it to the output list pop + from the stack & append it to output list & push newly read + operator onto the stack



8. Read ((small braket)



9. Read d, ~~which is an~~



14

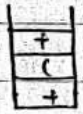
10. Read *, which is an operator & has lower precedence than '(' (top of stack) But we cannot delete until closing parenthesis encounters. So push * on the stack.

a	b	c	*	+	d

11. Read e

a	b	c	*	+	d	e

12 read +, is an operator, has lower precedence than *, so pop * from stack and append it to output list and we cannot pop (. Finally + onto stack.



a	b	c	*	+	d	e	*

13. Read f

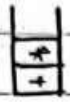
a	b	c	*	+	d	e	*	f

14. Read), a closing parenthesis, so pop all operators from stack in between (&) and append it to the output list excluding (

a	b	c	*	+	d	e	*	f	+

15

15. Read *



a	b	c	*	+	d	e	*	f	+

16. Read g

a	b	c	*	+	d	e	*	f	+	g

17. Pop all the operators from the stack, and then append it to th output list.

a	b	c	*	+	d	e	*	f	+	g	*	+

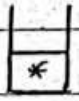
3. a * b - c - d / e + f

1)  stack

 o/p list

2) 

a

3) 

a

4) 

a	b

5) 

a	b	*

11

6)  a | b | * | c

7)  a | b | * | c | -

8)  a | b | * | c | - | d

9)  a | b | * | c | - | d

10)  a | b | * | c | - | d | e

11)  a | b | * | c | - | d | e | /

12)  a | b | * | c | - | d | e | / | f | +

13)  a | b | * | c | - | d | e | / | f | + | -

a * b - c - d / e + f

a b * - c - d e / + f

X - c - Y + f

X c - - Y f +

M N -

X c - N Y f + -

→ a b * c - d e f + - - +

EXAMPLE 5.7

Give postfix form for A + B/C – D.

SOLUTION.

A + (B/C) – D
A + (BC/) – D
A + T – D
(A + T) – D
(AT+) – D
S – D — where $S = (AT+)$
SD –
AT + D –
ABC / + D –. — **Postfix expression.**

EXAMPLE 5.8

Convert the expression (A + B) / (C – D) to postfix form.

SOLUTION. In this expression the brackets are already specified. Therefore the conversion looks like :

(AB+) / (CD–)
T/S — where T = (AB+) and S = (CD–)
TS/
AB + CD-/. — **Postfix expression.**

EXAMPLE 5.9

*Give postfix form for (A + B) * C/D.*

SOLUTION.

(AB+) * C/D
T * C/D — where T = (AB+)
(T * C)/D
(TC *)/D
S/D
SD/
TC * D/
AB + C * D/. — **Postfix expression.**

EXAMPLE 5.10

*Give postfix form for (A + B) * C/D + E ^ F/G.*

SOLUTION.

(AB+) * C/D + E ^ F/G
T * C/D + (E ^ F) / G — (^ has the highest priority)
T * C/D + (EF ^) / G

T * C/D + S/G where S = (EF ^)
(T * C)/D + S/G
(TC *)/D + S/G
Q/D + S/G where Q = (TC *)
(Q/D) + S/G
(QD/) + S/G
P + S/G where P = (QD/)
P + (S/G)
P + (SG/)
P + O where O = (SG/)
PO+

Now we will expand the expression PO+

PO+
QD/O+
TC * D/O+
AB + C * D/SG/+
AB + C * D/EF ^ G/+. **Postfix expression.**

EXAMPLE 5.11

*A + [(B + C) + (D + E) * F]/g. Give postfix form.*

SOLUTION. A + [(BC+) + (DE+) * F]/G
A + [T + S * F]/G where T = (BC+) and S = (DE+)
A + [T + (S * F)]/G
A + [T + (SF*)]/G
A + [T + Q]/G where Q = (SF*)
A + (TQ+)/G
A + P/G where P = (TQ+)
A + (PG/)
A + N where N = (PG/)
AN+

Expanding the expression AN+

APG/+
ATQ + G/+
ABC + Q + G/+
ABC + SF * + G/+
ABC + DE + F * G/+ **Postfix expression.**

EXAMPLE 5.12

*A + (B * C – (D / E ^ F) * G) * H. Give postfix form.*

SOLUTION.

A + (B * C – (D/(EF ^) * G)) * H	(^ has the highest priority)
A + (B * C – (D/T) * G)) * H	where T = EF^
A + (B * C – ((D/T) * G)) * H	
A + (B * C – ((DT/) * G)) * H	
A + (B * C – (S * G)) * H	where S = DT/
A + (B * C – (SG *)) * H	
A + (B * C – Q) * H	
A + ((B * C) - Q) * H	where Q = SG *
A + ((BC *) – Q) * H	
A + (P – Q) * H	where P = (BC *)
A + (PQ-) * H	
A + O * H	where O = (PQ –)
A + (O * H)	
A + (OH *)	
A + N	where N = (OH *)
AN+	

Expanding the expression AN+

AOH * +	
APQ – H * +	
ABC * Q – H * +	
ABC * SG * – H *+	
ABC * DT/G * – H * +	
ABC * DEF ^ /G * – H * +.	**Postfix expression.**

EXAMPLE 5.13

Solved Problems

PROBLEM 1

*Convert X : A + (B * C – (D/E –F) * G) * H into postfix form showing stack status after every step in tabular form.*

SOLUTION

Covert $A + (B * C - (D / E) * G) * H$ into postfix form.

Symbol Scanned	Stack	Expression
(	(	
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
–	(+(–	ABC*
(	(+(–	ABC+
D	(+(–	ABC+ D
/	(+(–(/	ABC+ D
E	(+(–(/	ABC+ DE
)	(+	ABC* DE/–
*	(+ *	ABC+ DE/–
G	(+	ABC+ DE/–G
)		ABC+ DE/–G
*	*	ABC+ DE/–G
H	*	ABC+ DE/–GH
)		ABC+ DE/–GH*

PROBLEM 2

EXAMPLE 5.19

This program takes an infix expression and converts it into an equivalent postfix expression.

SOLUTION.

```
#include<stdio.h>
#include<string.h>
char stack[50] ;
int top = - 1 ;
void in_to_post(char infix[ ]) ;
void push (char) ;
char pop( ) ;
void main( )
{
   char infix[25] ;
   printf("Enter the infix expression") ;
   gets(infix) ;
```

```
    in_to_post(infix) ;
    getch( ) ;
}
void push (char symb)
{
   if(top>= 49)
   {    printf("stack overflow") ;
        getch( ) ;
        return ;
   }
   else
   {
        top = top + 1 ;
        stack[top] = symb ;
   }
}
char pop( )
{
   char item ;
   if(top == -1)
   {
        printf("stack empty") ;
        getch( ) ;
        return(0) ;
   }
   else
   {    item = stack[top] ;
        top-- ;
   }
   return(item) ;
}
int preced (char ch)
{
   if (ch == 47)
   {
        return(5) ;
   }
   else
   if(ch = 42)
   {
        return(4) ;
   }
   else if(ch = 43)
   {
        return(3) ;
   }
   else
   return(2) ;
}
```

```
void in_to_post(char infix[ ])
{
   int length ;
   static int index = 0, pos = 0 ;
   char symbol, temp ;
   char postfix [40] ;
   length = strlen(infix) ;
   push('#') ;
   while(index < length)
   {
        symbol = infix[index] ;
        switch(symbol)
        {
             case '(' : push(symbol)) ;
                  break ;
             case ')' : temp = pop( ) ;
             while(temp != '( ' ) )
             {
                  postfix[pos] = temp ;
                  pos++ ;
                  temp = pop( ) ;
             }
             break ;
             case '+' :
             case '-' :
             case '*' :
             case '/' :
             case '^' :
             while (preced(stack[top]) >= preced(symbol))
             {    temp = pop( ) ;
                  postfix[pos] = temp ;
                  pos ++ ;
             }
             push(symbol) ;
                  break ;
             default : postfix[pos++] = symbol ;
                  break ;
        }
        index++ ;
   }
   while(top > 0)
   {    temp = pop( ) ;
        postfix[pos++] = temp ;
   }
   postfix[pos++] = '\0' ;
   puts(postfix) ;
   return ;
}
```

Output of the program

```
enter the infix expression a + b
ab+
```

5.9 CONVERTING INFIX EXPRESSION TO PREFIX EXPRESSION

This algorithm is bit tricky, in this we first reverse the input expression $a + b * c$ will become $c * b + a$ and then we do the conversion and then again we reverse to get the result. Doing this has an advantage that except for some minor modification algorithm for infix to prefix remains almost same as the one for infix to postfix.

Algorithm

1. Reverse the input string.
2. Examine the next element in the input.
3. If it is operand, add it to the output string.
4. If it is closing parenthesis, push it on stack.
5. If it is an operator, then
   (i) if stack is empty, push-operation on stack.
   (ii) if the top of stack is closing parenthesis push operator on stack.
   (iii) if it has same or higher priority then the top of stack, push operator on S.
   (iv) else pop the operator from the stack and add it to output string, repeat S.
6. If it is a opening parenthesis, pop operator from stack and add them to S until a closing parenthesis is encountered. POP and discard the closing parenthesis.
7. If there is more input go to step 2.
8. If there is no more input, unstack the remaining operators and add them.
9. Reverse the output string.

EXAMPLE

*Convert (A + B * C) into the prefix expression.*

SOLUTION. Reverse the expression

$A + B * C$ we get $(C * B + A)$.

Symbol Scanned	Stack	Postfix Expression
(	(	—
C	(	C
*	(*	C
B	(*	CB
+	(+	CB*
A	(+	CB*A
)	—	CB * A +

So, the Postfix expression is CB * A +. Now reverse this expression to get the result.

$+ A * BC$

EXAMPLE 5.14

*Give prefix form for A * B + C.*

SOLUTION. (A * B) + C

(* AB) + C

T + C where T = (* AB)

+TC

expanding expression +TC

+ * ABC. **Prefix expression.**

EXAMPLE 5.15

Give prefix form for A/B ^C + D.

SOLUTION. A/(B ^ C) + D (^has highest priority)

A/(^BC) + D

A/T + D where T = (^BC)

(A/T) + D

(/AT) + D

S + D where S = (/AT)

+SD

Expanding expression +SD

+/ATD

+/A ^ BCD. **Prefix expression.**

EXAMPLE 5.16

*Give prefix form for (A – B/C) * (D * E – F).*

SOLUTION. (A – (B/C)) * ((D * E) – F)

(A – (/BC)) * ((*DE)) – F
(A – T) * (S – F) where (T = /BC) and S = (*DE)
(–AT) * (–SF)
Q * P where Q = (–AT)
* QP

Expanding expression *QP

*AT – SF
* – A/BC – SF
* – A/BC – * DEF. **Prefix expression.**

EXAMPLE 5.17

*Give prefix form for (A * B + (C/D)) – F*

SOLUTION. (A * B + (/CD)) – F
(A * B + T) – F
(A * B + T) – F where (T = /CD)
((A * B) + T) – F
((*AB) + T) – F
(S + T) – F where S = (* AB)
(+ST) – F
Q – F where Q = (+ST)
–QF

Expanding expression –QF

– +STF
– + *ABTF
– + *AB/CDF. **Prefix expression.**

EXAMPLE 5.18

A/B ^ C – D. Give prefix expression.

SOLUTION. A/(B ^ C) – D
A/(^BC) – D
A/T – D where T = (^BC)
(A/T) – D
(/AT) – D
S – D where S = (/AT)
– SD

Expanding expression –SD

–/ATD
–/A ^ BCD. **Prefix expression.**

ALGORITHM TO EVALUATE POSTFIX EXPRESSION [2011 FALL]

STEP 1: Scan P (postfix expression) from left to right and repeat ~~spet spet~~ step 2 and step 3 for each element of P

STEP 2: If an operand is encountered, put it on the stack

STEP 3: If an operator is encountered, then :

(a) remove the top two elements of the stack where A is the top and B is the next to the top element.

(b) evaluate B operator A

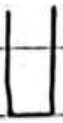
(c) place the result of (b) back on the stack

STEP 4: Set value equal to the top element on the stack

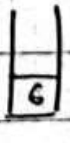
STEP 5: Exit

Eg. Evaluate postfix expression 6, 5, 2, 3, +, 8, *, +, 3, +, *

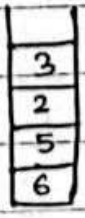
Sol^{n}: 1. Initialize an empty stack.



2. Read 6, which is an operand, so push it onto stack



3. Push operand 5, 2 & 3 onto the stack

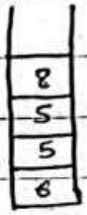


(3) ABC/-DE*F-* (where A=6, B=5, C=2, D=3, E=4, F=1)

18

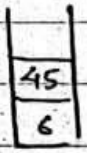
4. Read +, which is an operator, so pop two element from stack (3 & 2) and apply + operator on 3 & 2 ie 3+2=5, push the result '5' on to top of the stack.

5. Next operand is 8, so push it onto stack

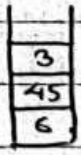


6. Read * which is an operator, so pop top two element from stack (8&5) and apply * operator on 8 & 5 ie 8*5 = 40, & push the result 40 onto the stack.

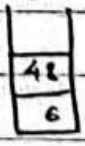
7. Next is + operator, so pop top two element from stack, 40 & 5, apply + operator, ie. 40+5 = 45, push result 45 on to the stack



8. Next is 3, so push it on stack



9. Next is +, so pop top two element from stack, ie 45+3 = 48, put 48 on top of stack



10. Next is *, so pop top two element from stack, ie. 6*48 = 288

∴ The result is 288

H/W

① 6, :, 3, +, −, 3, 8, 2, /, +, *, 2, 1, 3, +

② Post fix expression : ABC + * DE / −

where A = 5 , B = 6 , C = 2 , D = 12 , E = 4 or

5, 6, 2, +, *, 12, 2, /, −

10

ADVANTAGES OF POSTFIX EXPRESSION OVER INFIX EXPRESSION

ABC+*DE/$ (Where A=5, B=6, C=2,D=12, E=4)

1. Unlike INFIX notation follows BODMAS rule and the evaluation complex, the POSTFIX notation do not need to follow any set of rules.
2. INFIX notation needs rescanning of the expression for evaluation But the POSTFIX notation can be evaluated with a single scan from left to right.
3. POSTFIX notation is easier to work with since it does not contain any parenthesis and follow any rules of operator precedence.
4. POSTFIX notation ~~are~~ evaluation is efficient from the compiler point of view due to a single scan.

9.

Advantage of postfix expression over infix expression

1. No need for parentheses: In infix notation, parentheses are used to specify the order of operations. However, in postfix notation, the order of operations is determined solely by the position of the operators. This eliminates the need for parentheses, which can make expressions easier to read and understand.
2. Easier to evaluate: Postfix notation allows for simpler evaluation of expressions using a stack-based algorithm. This is because operators are placed after their operands, which allows them to be processed in a left-to-right order. This makes it easy to evaluate the expression using a stack without needing to keep track of parentheses or operator precedence.
3. More compact: Postfix notation can be more compact than infix notation, especially for expressions with nested parentheses. This can make it easier to write and read complex expressions.
4. Unambiguous: Postfix notation eliminates the need for operator precedence and associativity rules, which can sometimes lead to ambiguity in infix notation. In postfix notation, the order of operations is always clear and unambiguous.

Stack Example:

1. Browser history - When you visit a website, the URL is added to a stack. When you click on the back button, the URL is popped from the stack and the previous page is displayed.
2. Undo/Redo functionality - When you perform an action such as typing a letter, the action is added to a stack. When you click on the undo button, the action is popped from the stack and the previous action is displayed. When you click on the redo button, the popped action is pushed back onto the stack.
3. Function call stack - When a function is called, its variables and parameters are added to the top of the stack. When the function returns, the variables and parameters are popped from the stack.
4. Expression evaluation - When an expression is evaluated, the operands and operators are added to the stack in a postfix notation. The operands are then popped from the stack and the operation is performed.
5. Backtracking algorithms - In algorithms such as maze solving or graph traversal, the current state is pushed onto the stack. When a dead end is reached, the state is popped from the stack and the algorithm backtracks to the previous state.