Rainbow
PAGE:
DATE: / /

DSA Ass -2

Name: Bhavana Bafna
Roll no.: 23107
Class: SE 9
Batch: E9

1] Title: Expression Conversion

2] Aim: To implement expression conversion using stack Data structures.

3] Problem Statement: Implement stack as an abstract data type using singly LL and use this ADT for conversion of infix expression to postfix, prefix & evaluation of postfix & prefix expression.

4] Objective:

1) To understand the concept and implementation of stack data structure using SSL.
2) To understand the concept of conversion of expression.
3) To understand the concept of evaluation of expression.

5] Outcomes:

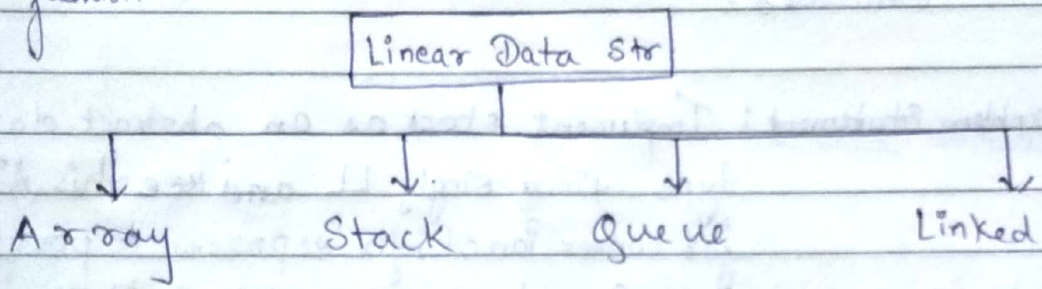
1) Implement stack as an ADT.
2) Implement applications of stack i.e., Expression conversion and evaluation.

6] Theory:

Rainbow
PAGE:
DATE: / /

1) CONCEPT OF LINEAR DATA STRUCTURE:

- The data structure where data items are organized sequentially or linearly one after another is called linear data structure.
- Data elements in a linear data structure are transversed one after the other and only one element can be directly reached while transversing. All data items can be traversed in a single run.
- These kind of data structures are very easy to implement because memory of computer also has been organized in linear fashion.



2) EXAMPLE OF LINEAR DATA STRUCTURES:

Examples : Array
Stack
Queue
Linked Lists

3) STACK:

i) CONCEPT:

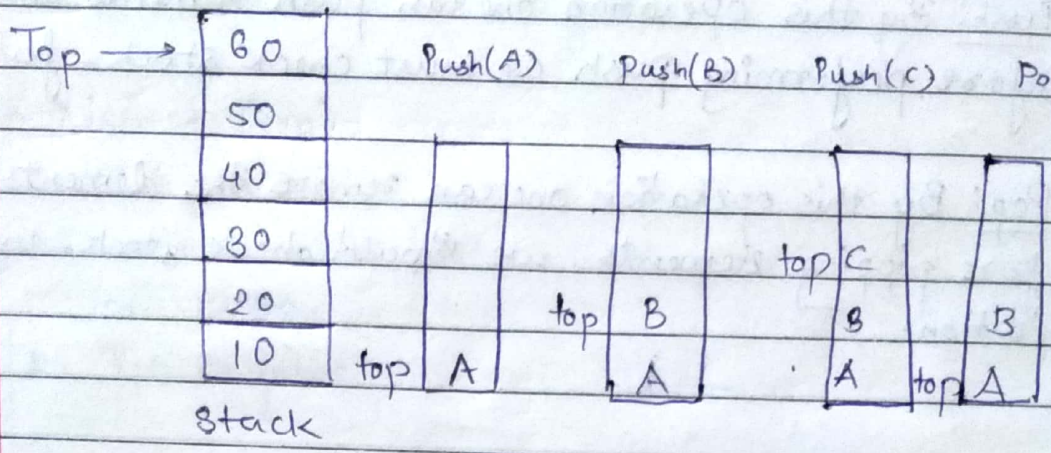
- Ordered list in which insertion and deletion are made at one end called the top.
- Stack Principle : LIFO (last in first out).
- To retrieve 0^{th} element needs to remove n-1 element.

ii) DEFINITION OF STACK:

- A stack is an ordered list in which all insertions and deletions are made at the end, called the top.

- If we have to make stack of elements 10, 20, 30, 40, 50, 60 then 10 will be the bottom-most element and 60 will be the top-most element in the stack.

iii) TERMINOLOGY AND DIAGRAM:



Eg: Stack of coins

iv) ADT OF STACK:

1) ADT FORMAT:

Stack is a data structure which posses LIFO, i.e., Last in First out Property. The abstract type for stack can be as given below:-

Abstract Data Type stack
{

Instances : Stack is a collection of elements in which insertion & deletion of elements is done by one end called top.

Preconditions :

1) Stack_full( ) : Checks whether stack is full or not. If the stack is full then we cannot insert the elements in the stack.

Rainbow
PAGE:
DATE: / /

2) Stack_empty(): Indicates whether the stack is empty or not. If the stack is empty then we cannot pop or remove any element from the stack.

Operations:

1) Push: By this operation one can push elements onto the stack. Before performing push we must check stack_full() condition.

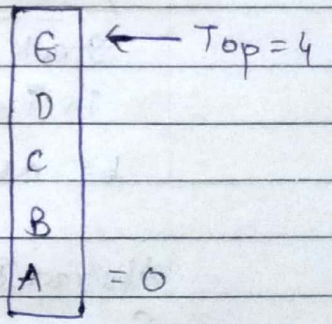
2. Pop: By this operation one can remove the elements from stack. Before popping elements, we should check stack_empty() condition.

}

v) Realisation of ADT using:

i) ARRAY STACK:

```
Declare stack (Max_size)
// can be float, char, int
Declare top
Array of structure -
type def stack struct.
{ a[Max_size]
  top
}
initialize top = -1
```



→ isfull (top)

1. if top >= max_size // check top condition.
   then write ('stack overflow')
   return True.
2. Return False.

→ push (s, top, x)

1. if (isfull (top)) //check for overflow
   return
2. top ← top + 1 // increament top
3. s[top] ← x // insert element
4. return // finish.

→ isEmpty (top)

1. if top < 0
   then write ('stack underflow')
   return true.
2. Return false

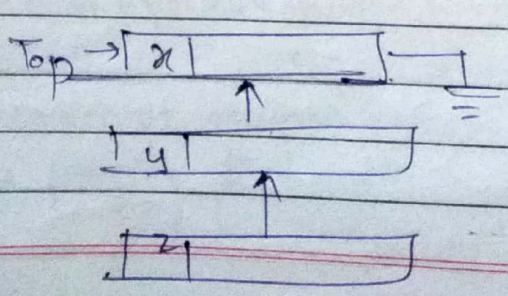
→ pop (s, top)

1. if isEmpty (top) // check for underflow
   return
2. top ← top - 1 // deacreament top.
3. return s[top + 1] // return former top element.

→ display (s, top)

1. if (is Empty ()) //check for underflow
   return
2. For ← i ← top to 0
   decreament by 1
   write s[i]

2) LINKED LIST STACK:



Rainbow

PAGE:
DATE: / /

```
typedef struct node
{
   int data;
   struct node * link
} node;
```

→ push (x)

Node temp = x

temp * link = top

top ← temp.

→ pop()

Node temp ← top

top ← top * link

delete temp.

→ isEmpty (top)

if top < 0

then write 'stack underflow'

return true

return false

→ display ()

if (isEmpty)

return

else

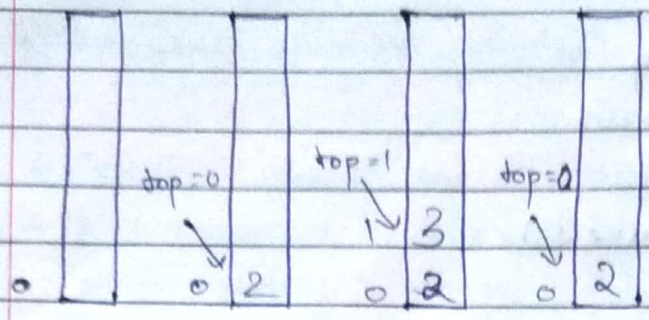
Node temp = top

until top ← null

write temp data.

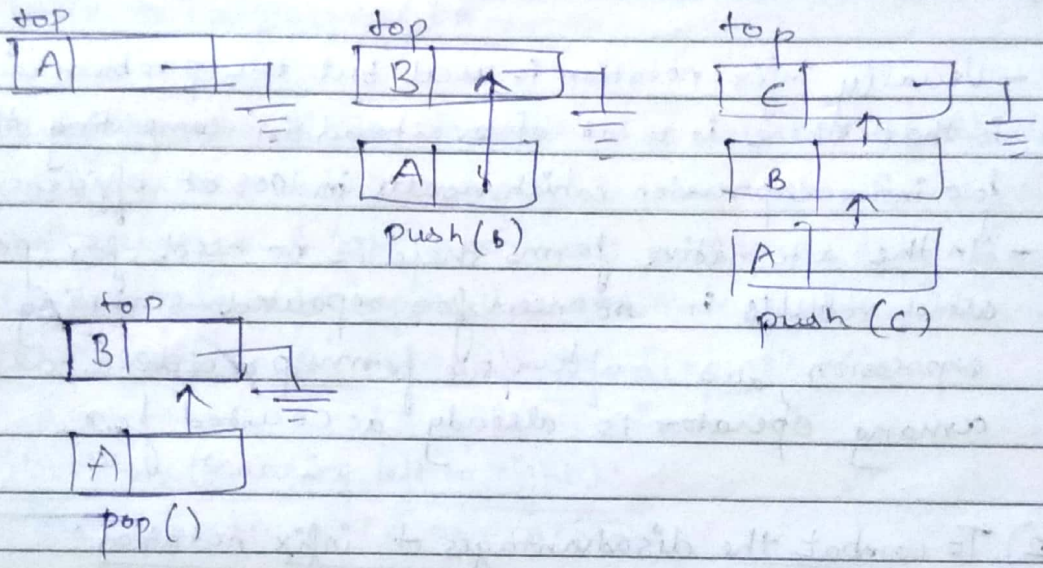
3) DIAGRAM :

(Stack) : Array.



isEmpty() Push(2) Push(3) pop()
true

(Stack): linked list.



6] APPLICATION OF STACK:

- Expression conversion
  - Infix to postfix
  - Infix to prefix
  - Prefix to infix
  - Prefix to infix.
- Expression evaluation.
- Parsing
- Simulation of recursion
- Game playing, find paths

- Exhaustive searching
- well-format paranthesis
- Reversal of string
- Tree or graph transversal
- Backtracing

7] EXPRESSION CONVERSION AND STACK:

1) Need for expression conversion:

- Usually infix notation is used but the problem with infix is that there is a lot of overhead in computing the result for infix expression which results in loss of efficiency.
- In the alternative forms there is no need for paranthesis which ~~results~~ in no need for repeated scanning of true expression, also in alternative forms procedure and associativity among operator is already accounted for.

2) To combat the disadvantages of infix notation:

- Prefix notation was introduced by a Polish logician so also called polish notation.
- Postfix notation is sometimes called reverse polish notation or RPN.
- Hence, there are types of notations:
  - Infix - <identifier><operator><identifier>
  - Postfix - <identifier><operator><operator>
  - Prefix - <operator><identifier><identifier>
- For eg:
  Infix form: ((a+b)*c)
  Postfix form: ab+c*
  Prefix form: *+abc

3) Advantages of polish notation:

- Expression can be shown without paranthesis.
- It is convinient to evaluate formula on stacks.
- Polish notation eliminates problems faced by precendance.
- The complete expression can be passed in one traversal

7] Algorithms / PSEUDOCODES:

i) Infix to Postfix conversion -

During the scan of the expression an operand is immediately added to the output string. While the operator stack stores the operators & left paranthesis as soon as they appear, handles ~~sup~~ sub-expressions and manages the order of the precedence & associativity of operations.

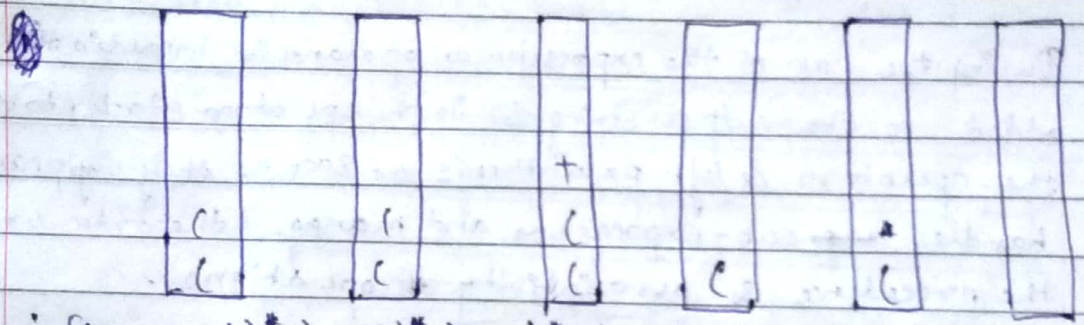
Algorithm (scanning left to right):

1. If the character x is an operand
   - output the character x into postfix expression.
2. If the character x is a left or right paranthesis
   - If character is "(" : push into the stack.
   - If character is ")" : repeatedly pop and output all the characters / operators until "(" is popped.
3. If the character x is a regular operator:
   - Check the character y currently at the top of stacks
   - If stack is empty or y = '(' or y is an operator of lower precedance than x then push x in stack.
   - If precedence (y) > precedence (x) then while (precedence (y) > precedence (x)) pop and output y.
   - Push x into stack.

- If precedence(y) < precedence(x)
  Check associativity of any one operators.
  - if associativity (y) → left to right
    pop & push x
  - if associativity (y) → right to left.
    push x.

4. If stack is not empty pop all the operators from the stack and append them to output string.

int exp : ((a+b)*c)



infix	a+b)*c)	+b)*c)	)*c)	*c)	)	
postfix		a	ab	ab+	ab+c	ab+c*

ii) Infix to postfix conversion:

For this conversion the infix expression is scanned from right to left instead of left to right like a postfix conversion. The operands are added directly to output string while the stack contains the operators & parenthesis.

Algorithm (scanning right to left):

1. If the character x is an operand.
   - o/p the character in to the prefix expression.
2. If the character x is a left or right paranthesis
   if the character is ')'.

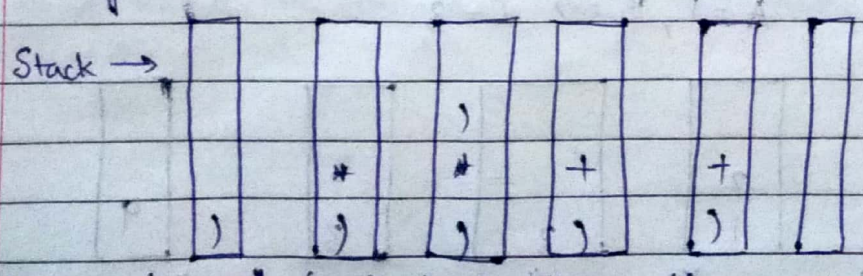
push it into the stack.
if the character is '('.
repeatedly pop & o/p all the operators characters until ')' is popped from the stack.

3. If the character x is a regular operator
- check the character if currently at the top of the stack.

If the stack is empty of y = '(' or y is an operator of lower precendence than x.
then push x into the stack.
If precedence (y) > precedence (x) then
- while (precedence (y) >= precedence (x))
pop & output y.
- push x into the stack
If precedence (y) = precedence (x)
Check the associativity of any one operator
- if associativity (y) [left to right]
push x
- if associativity (y) is right to left:
pop(y) & push(x)

4. If stack is not empty, pop all the operators from the stack & append the output string.

infix exp: ((a+b)*c)



Infix:	((a+b)*	((a+b)	((a+	((a	(((	
Prefix:	c	c	bc	*bc	a*bc	+a*bc

Rainbow

PAGE:
DATE: / /

Prefix exp: + a * b c

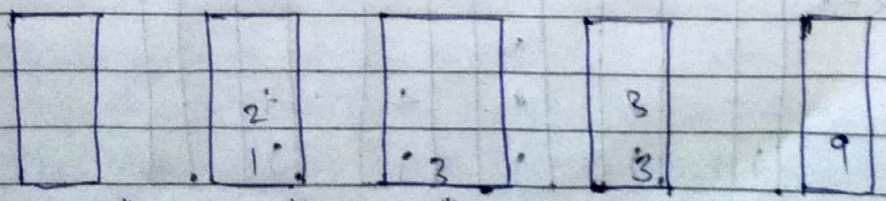
iii) Postfix evaluation:

This is done by adding the operands onto a stack & removing two whenever an operator is read in the postfix expression string. The results of the operation is then added back to the stack & the process is repeated until there is only one element remaining in the stack.

Algorithm:

```
Postfix evaluation (Postfix [ ])
{
  Operand stack = Empty stack
  while (Post [i] != '\0')          // left to right
    symbol = postfix [i]
    if (is operand (symbol))
       push (operand-stack, symbol).
    else
       operand 2 = pop ()
       operand 1 = pop ()
       result = operand 1 symbol operand 2
       push result
       return pop ()
}
```

AB + C* , A = 1, B = 2, C = 3



AB + C* + C* C* * 3* 3 = 9

2+1=3

Rainbow
PAGE:
DATE: / /

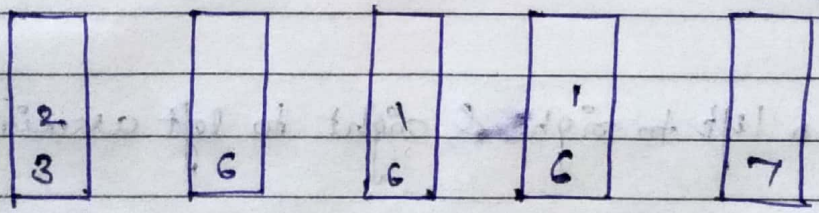
iv) Prefix Evaluation:

This is done very similar to postfix evaluation except instead of traversing the string left to right it is traversed right to left. The presence is repeated until there is only one element left in the stack.

Algorithm:

while (more input) // scan right to left.
  if (operand) push into stack.
  if (operator) apply to top 2 stack items & replace them (on the stack) with the result of the operation.
Should have one one item on the operand.
  Stack = value of expression.

+ A * B C A = 1, B = 2, C = 3.



Prefix + * A + A + 6 + 1 = 7
2 * 3 = 6

In the above example we can see how the evaluation is done & the result = 7 is obtained.

Rainbow

PAGE :
DATE : / /

8] TEST CASES / VALIDATIONS:

Validations:

1) No. of operator & operand relationship
2) Well formed paranthesis

Test cases:

1) Based on precedence of operators:

Sample Infix Expression	Postfix	Prefix
1. A + B * C	AB C * +	+ A * B C
2. A * B - C	A B * C -	- * A B C
3. A ^ B - C	A B ^ C -	- ^ A B C
4. A + B * C ^ E	A B C E ^ * +	+ A * B ^ C E
5. A - B * C + A	A B C * - A +	+ - A * B C A
6. (A + B) / (C + D) ^ E ^ F - D * F - D	A B + C D + E F ^ ^ / D F * - D -	- - / + A B * + C D ^ E F * D F D

2) Based on left to right & right to left associativity.

Sample Infix Expression	Postfix	Prefix
1. A + B + C	A B + C +	+ + A B C
2. A * B / C	A B * C /	/ * A B C
3. A ^ B ^ C	A B C ^ ^	^ A ^ B C

9] Declaration of Data structures:

In this program, the data structures used are stack and linked list.

10] Conclusion:

Successfully implemented the assignment using stack as stack as ADT to do the Infix to postfix, prefix expression.
Learned stack data structures, its implementation using stack & its application in expression conversion as a temporary data structures.