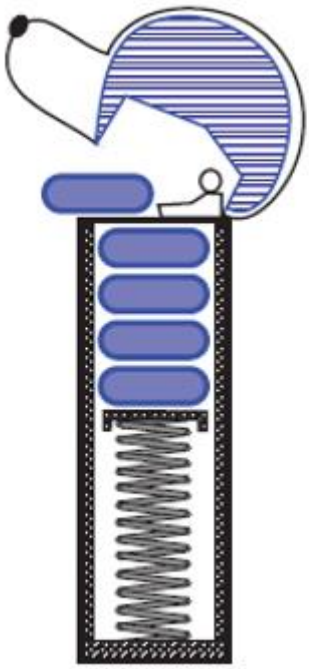
Stacks

A ***stack*** is a container of objects that are inserted and removed according to the ***last-in first-out*** (***LIFO***) principle. Objects can be inserted into a stack at any time, but only the most recently inserted (that is, "last") object can be removed at any time.

Example 5.1: *Internet Web browsers store the addresses of recently visited sites on a stack. Each time a user visits a new site, that site's address is "pushed" onto the stack of addresses. The browser then allows the user to "pop" back to previously visited sites using the "back" button.*

Example 5.2: *Text editors usually provide an "undo" mechanism that cancels recent editing operations and reverts to former states of a document. This undo operation can be accomplished by keeping text changes in a stack.*



A Stack is a widely used linear data structure in modern computers in which insertions and deletions of an element can occur only at one end, i.e., top of the Stack. It is used in all those applications in which data must be stored and retrieved in the last.

An everyday analogy of a stack data structure is a stack of books on a desk, Stack of plates, table tennis, Stack of bootless, Undo or Redo mechanism in the Text Editors, etc.

push(e): Insert element e at the top of the stack.

pop(): Remove the top element from the stack; an error occurs if the stack is empty.

top(): Return a reference to the top element on the stack, without removing it; an error occurs if the stack is empty.

Additionally, let us also define the following supporting functions:

size(): Return the number of elements in the stack.

empty(): Return true if the stack is empty and false otherwise.

Example 5.3: *The following table shows a series of stack operations and their effects on an initially empty stack of integers.*

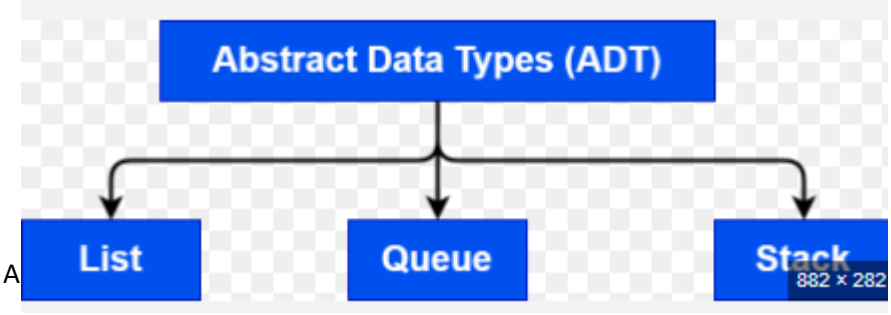
Operation	*Output*	*Stack Contents*
push(5)	–	(5)
push(3)	–	(5,3)
pop()	–	(5)
push(7)	–	(5,7)
pop()	–	(5)
top()	*5*	(5)
pop()	–	()
pop()	*"error"*	()
top()	*"error"*	()
empty()	*true*	()
push(9)	–	(9)
push(7)	–	(9,7)
push(3)	–	(9,7,3)
push(5)	–	(9,7,3,5)
size()	*4*	(9,7,3,5)
pop()	–	(9,7,3)
push(8)	–	(9,7,3,8)
pop()	–	(9,7,3)
top()	*3*	(9,7,3)

What is ADT explain?

Abstract Data type (ADT) is **a type (or class) for objects whose behavior is defined by a set of values and a set of operations**. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. 13-Dec-2022

An abstract data type is also known as ADT. It means providing only necessary details by hiding internal details. In short abstract data, types have only data not how to use or implement it. Also, it can be created at the level. 13-Jun-2022

A



Application of the Stack

- A Stack can be used for evaluating expressions consisting of operands and operators.
- Stacks can be used for Backtracking, i.e., to check parenthesis matching in an expression.
- It can also be used to convert one form of expression to another form.
- It can be used for systematic Memory Management.

STL stack

Rules for converting infinix to postfix

print operands as they arrive

If stack is empty or contains left parenthesis on top, push the incoming operator onto the stack.

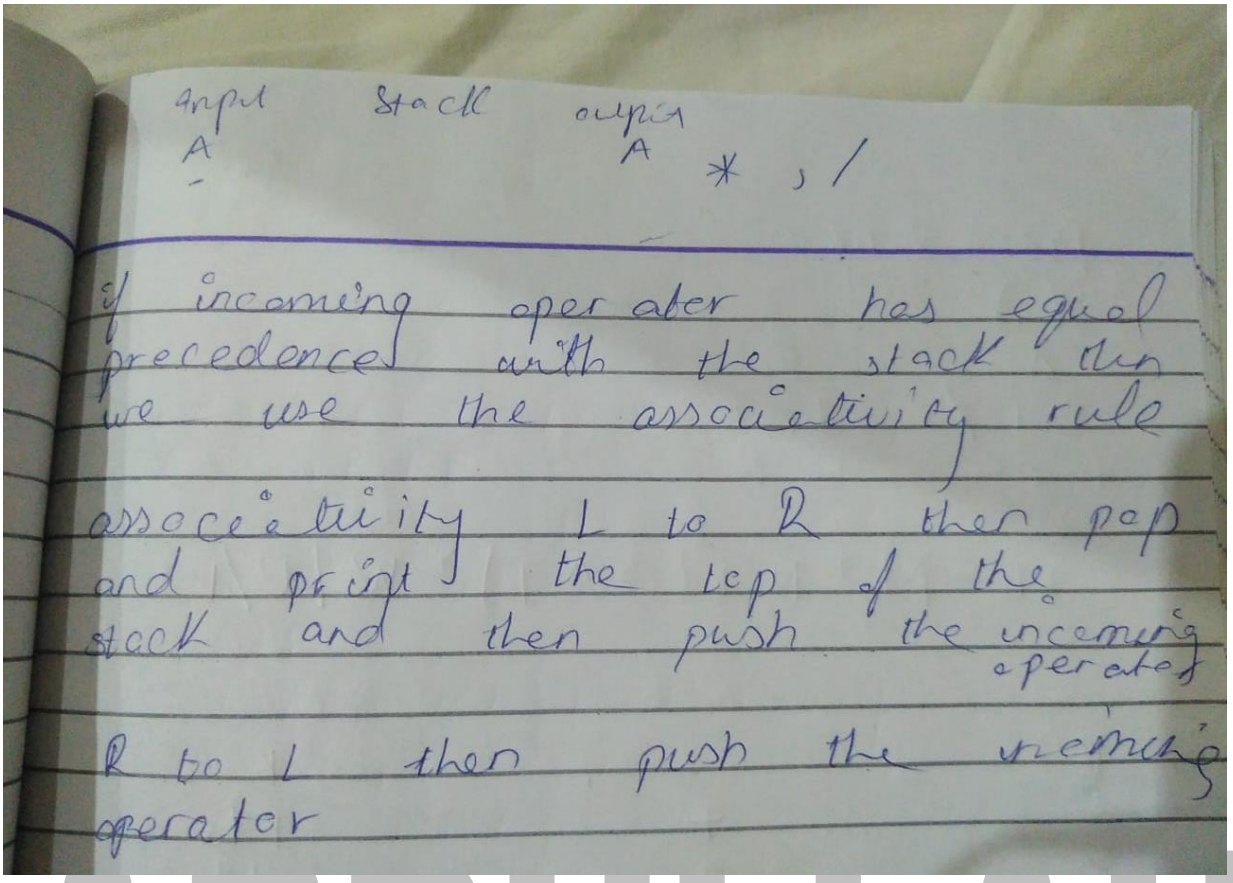
if incoming symbol is '(' push it onto stack

if incoming symbol is ')' pop the stack and print the operators until left parenthesis is found

if incoming symbol is ')' pop the stack and print the operators until left parenthesis is found

if incoming symbol has higher precedence than the top of the stack, push it onto stack

if incoming symbol has lower precedence than top of stack pop and print the top. Then test the incoming operator with the new top of the stack


input
stack
output
A
A
*
,
/
if incoming operater has equal precedence with the stack then we use the associativity rule
associativity L to R then pop and print the top of the stack and then push the incoming operator
R to L then push the incoming operator

A * B (C / D * E (F + G) + H - I * J / K)

Input	Stack	Output
A		A
*	*	
B	*	AB
(	* (	
C	* (	ABC
/	* (/	ABC
D	* (/	ABCD
*	* (/	ABCD
	* (*	ABCD /
E	* (*	ABCD/E
(	* (* (	
F	* (* (	ABCD/EF
+	* (* (+	ABCD/EF
G	* (* (+	ABCD/EFG
)	* (*	ABCD/EFG+
+	* (+	ABCD/EFG+*
H	* (+	ABCD/EFG+*H
-	* (	ABCD/EFG+*H+
	* (-	
*	* (- *	
J		ABCD/EFG+*H+J
/	* (- /	ABCD/EFG+*H+J*K/

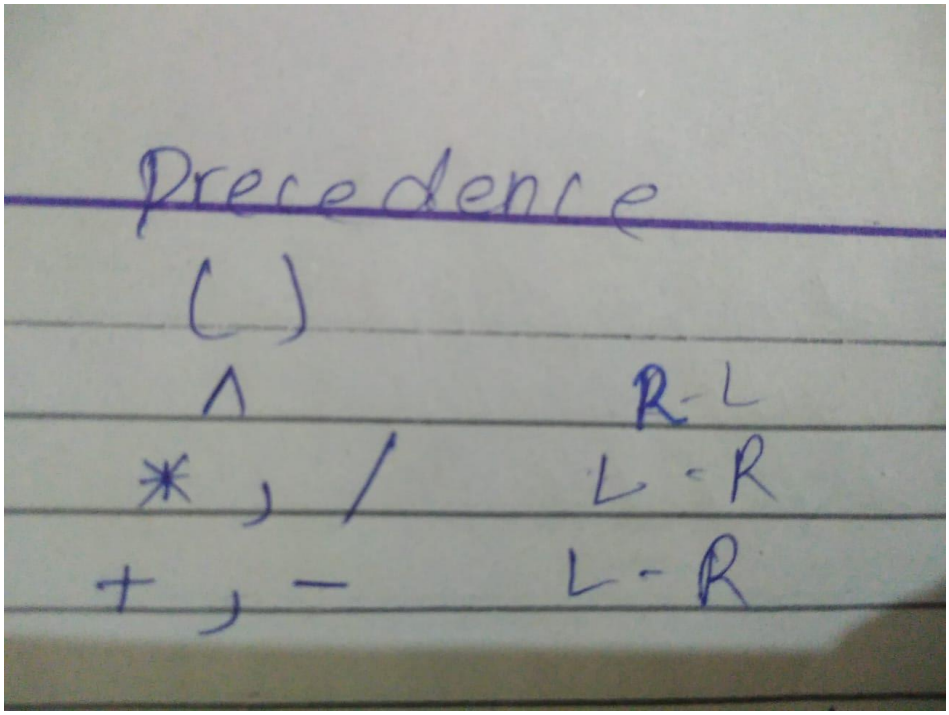
A / B + C - D

Input	Stack	Output
A		A
/	/	
B	/	AB
+	+	AB /
C	+	AB / C
-	-	AB / C +
D	-	AB / C + D -

A * D / C ^ T

Input	Stack	Output
A		A
*	*	A
D	*	AD
/	/	AD *
C	/	AD * C
^	^	AD * C /
T	^	AD * C / T

AD * C / T ^



- ***Infix Notation:*** Operators are written between the operands they operate on, e.g. 3 + 4.
- ***Prefix Notation:*** Operators are written before the operands, e.g + 3 4
- ***Postfix Notation:*** Operators are written after operands.

Postfix expression: *The expression of the form "a b operator" (ab+) i.e., when a pair of operands is followed by an operator.*

Infix Expressions are harder for Computers to evaluate because of the additional work needed to decide precedence. Infix notation is how expressions are written and recognized by humans and, generally, input to programs. Given that they are harder to evaluate, they are generally converted to one of the two remaining forms. A very well known algorithm for converting an infix notation to a postfix notation is Shunting Yard Algorithm by Edgar Dijkstra.

Step 1: Create an operand stack.
Step 2: If the character is an operand, push it to the operand stack.
Step 3: If the character is an operator, pop two operands from the stack, operate and p ush the result back to the stack.
Step 4:After the entire expression has been traversed, pop the final result from the stac k.

Example

For example, let us convert the infix expression we used into postfix for expression evaluation using stack.

Postfix expression : 2 5 3 6 + * * 15 / 2 -

Character	Action	Operand Stack
2	Push to the operand stack	2
5	Push to the operand stack	5 2
3	Push to the operand stack	3 5 2
6	Push to the operand stack	6 3 5 2
+	Pop 6 and 3 from the stack	5 2
	6 + 3 = 9, push to operand stack	9 5 2
*	Pop 9 and 5 from the stack	2
	9 * 5 = 45, push to operand stack	45 2
*	Pop 45 and 2 from the stack	
	45 * 2 = 90, push to stack	90
5	Push to stack	5 90
/	Pop 15 and 90 from the stack	
	90 / 5 = 18, push to stack	18
2	Push to the stack	2 18
-	Pop 2 and 18 from the stack	
	18 - 2 = 16, push to stack	16

As we can see, the final answer here is also 16.

Which expression is most suitable for evaluating using stack?

Postfix expressions are most suitable for evaluation with the help of a stack.

Why do we convert infix expressions to postfix or prefix?

Infix expressions are easily understandable and solvable by only humans and not computers. A computer cannot easily differentiate between operators and parentheses, so infix expressions are converted into postfix or prefix.

What determines the order of evaluation in an expression?

The precedence of operators determines the order of evaluation in an expression.

What is the precedence of operators?

The precedence of operators for expression evaluation using stack is given as follows:
Exponential (^)
Multiplication and division (* /)
Addition and subtraction (+ –)

How many stacks are required to evaluate infix, postfix and prefix expressions?

To evaluate infix expressions, we need two stacks (operator and operand stack), and to evaluate postfix and prefix expressions, we need only one stack (operand stack).

Example4 – Calculate the value for the postfix expression.

57 + 2 * 3/

Solution

This expression can be evaluated by using a stack. Each symbol can be inserted sequence wise & the operator can be applied on operands lying at the top of the stack and next to it.

Symbol	Stack	Description
5	5	
7	5 7	5 + 7
+	12	
2	12 2	
*	24	12 * 2
3	24 3	
/	8	24/3

Contemplate the postfix expression 234*+ as an example.

Input	Stack	Postfix evaluation
2 3 4 * +	*empty*	Push 2 into stack
3 4 * +	2	Push 3 into stack
4 * +	3 2	Push 4 into stack
* +	4 3 2	Pop 4 & pop 3 from stack and do 3 *4 , push 12 into stack
+	12 2	Pop 12 & Pop 2 from stack do 2 + 12, push 14 into stack
	14	

Input	Stack	Postfix evaluation
3 4 * 2 5 * +	*empty*	Push 3 into stack
4 * 2 5 * +	3	Push 4 into stack
* 2 5 * +	4 3	Pop 4 & pop 3 from stack do 3*4, Push 12
2 5 * +	12	Push 2 into stack
5 * +	2 12	Push 5 into stack
* +	5 2 12	Pop 5, Pop 2 from stack do 2*5, Push 10 into stack
+	10 12	Pop 10 & Pop 12 from stack do 12 + 10, push 22 into stack
	22	

How to convert Infix to postfix :

Examples

Input	Stack	Postfix evaluation
2*3 + 4*5	Nothing in stack	
*3+4*5	Nothing in stack	2
3+4*5	*	2
+4*5	*	23
4*5	+	23*
*5	+	23*4
5	*+	23*4
	*+	23*45
	+	23*45*
	Nothing in stack	23*45*+

Input	Stack	Postfix evaluation
2-3+4-5*6	Nothing in stack	
-3+4-5*6	Nothing in stack	2
3+4-5*6	-	2
+4-5*6	-	23
4-5*6	+	23-
-5*6	+	23-4
5*6	-	23-4+
*6	-	23-4+5
6	*-	23-4+5
	*-	23-4+56
	-	23-4+56*
	Nothing in stack	23-4+56*-

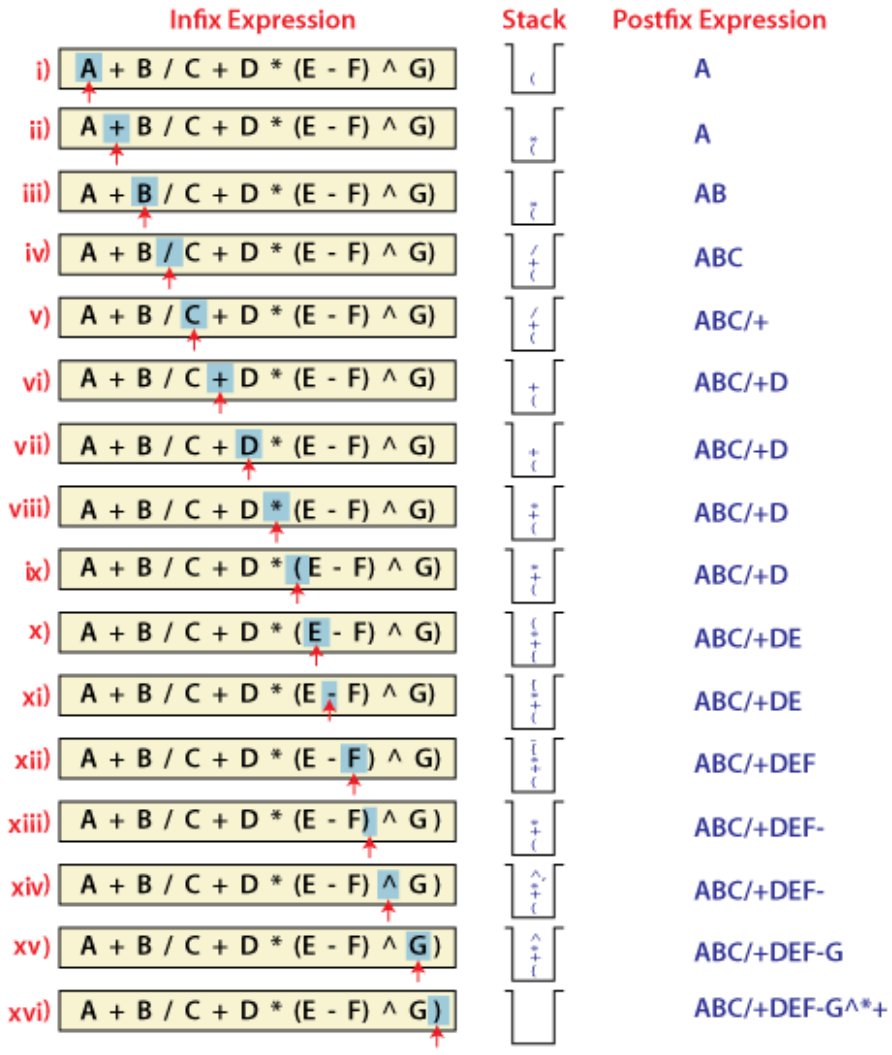
Input	Stack	Postfix evaluation
(2-3+4)*(5+6*7)	Nothing in stack	
2-3+4)*(5+6*7)	(	
-3+4)*(5+6*7)	(	2
3+4)*(5+6*7)	(-	2
+4)*(5+6*7)	(-	23
4)*(5+6*7)	(+	23-
)*(5+6*7)	(+	23-4
*(5+6*7)	Nothing in stack	23-4+
(5+6*7)	*	23-4+
5+6*7)	(*	23-4+
+6*7)	(*	23-4+5
6*7)	+(*	23-4+5
*7)	+(*	23-4+56
7)	*+(*	23-4+56
)	*+(*	23-4+567
	*	23-4+567*+
	Nothing in stack	23-4+567*+*

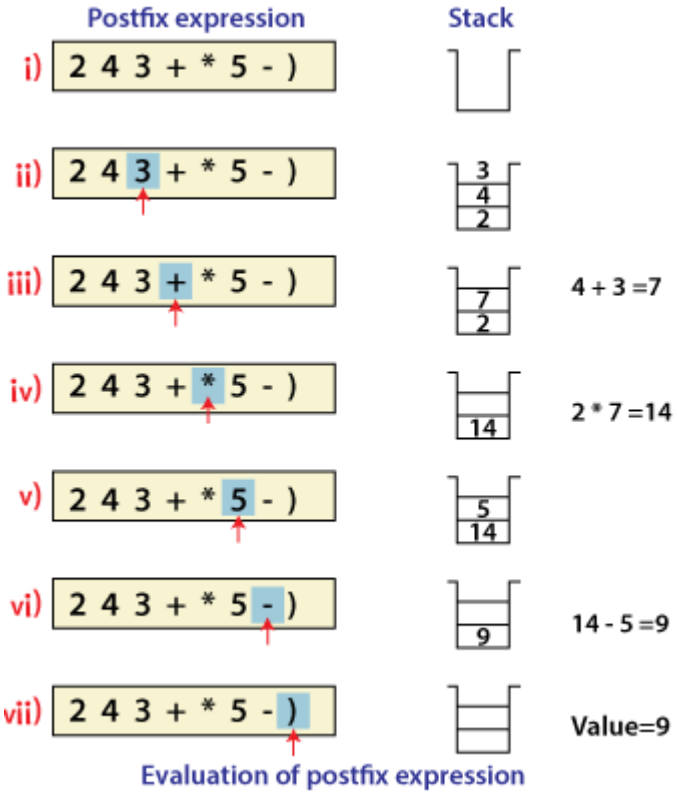
ABDULLAH: 03343127215

Following is the various Applications of Stack in Data Structure:

- Evaluation of Arithmetic Expressions
- Backtracking
- Delimiter Checking
- Reverse a Data
- Processing Function Calls

Let's take the example of Converting an infix expression into a postfix expression.





Disadvantages of Stack

1. It is difficult in Stack to create many objects as it increases the risk of the Stack overflow.
2. It has very limited memory.
3. In Stack, random access is not possible.