STACKS

→ A stack is a linear data structure.

LIFO (Last-in-First-out)

→ A stack is an ordered collection of homogenous data element where the insertion and deletion operation take place at one end only.

→ Like an array and a linked list, a stack is also a linear data structure but the only difference is that in the case of the former two, insertion and a deletion operation can take place at any position. The insertion and deletion operations in the case of stack are specially termed as PUSH and POP, respectively, and the position of the stack where these operations are performed is known as 'Top' of the stack. An element in a stack is termed as ITEM. The maximum number of elements that a stack can accommodate is termed SIZE.

Representation Of A Stack:

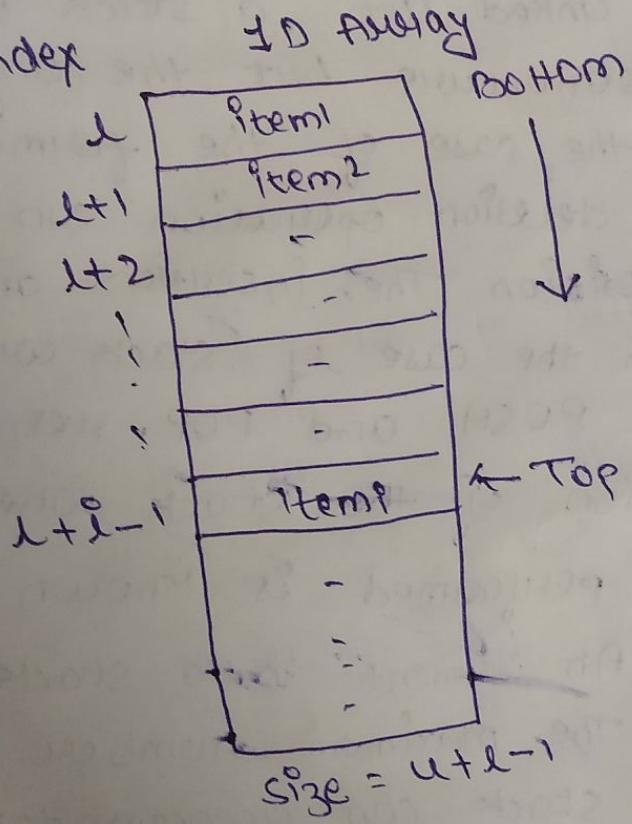
① Array Representation of stack.

② Linked List Representation of stack.

Array Representation of Stacks.

→ Allocate the memory block of sufficient size to accommodate the full capacity of the stack.

→ Starting from the first location of the memory block, the items of the stack can be stored in a sequential fashion.



Array representation of a stack.

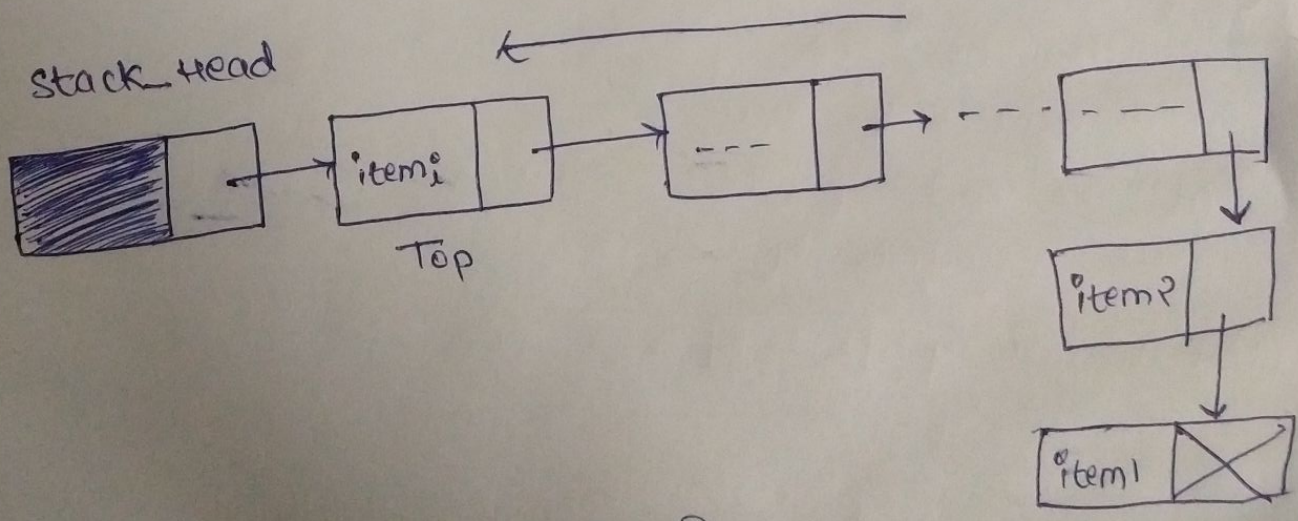
→ It allows the representation of only fixed sized stacks.

Linked List Representation of Stacks.

→ A single linked list structure is sufficient to represent any stack.

→ DATA field is for the ITEM, and the LINK field is, to point to the next item.

→ In the linked list representation, the first node on the list is the current item that is the item at the top of the stack and the last node is the node containing the bottom-most item.
Thus, a PUSH operation will add a new node in the front and a POP operation will remove a node from the front of the list.
The SIZE of the stack is not important here because this representation allows dynamic stacks ~~lasted~~ instead of static stacks, as with arrays.



Linked-List Representation of stack

Operation On Stacks

→ The Basic Operation Required to manipulate a stack are:

PUSH → To insert an item into a stack

POP → To remove an item from a stack

STATUS → To know the present state of a stack

Application Of Stacks

→ A classical application in a compiler design is the evaluation of ~~arithi~~ arithmetic expressions.

→ Another important application of a stack is during the execution of recursion program.

NOTATIONS for arithmetic Expressions.

① Infix → operator comes in between the operands.

<operand><operator><operand>

Eg:→ A+B, C-D, E*F, G/H etc.

② Prefix → operator come before the operands.

<operator><operand><operand>

Eg:→ +AB, -CD etc.

③ Postfix → The postfix notation is just reverse of the Polish notation, hence it is also termed reversed Polish notation.

Eg :→ AB+, AB-, GH/ etc.

Infix To Postfix conversion

1. Print operands as they arrive
2. If stack is empty or contains a left parenthesis on top, push the incoming operator into the stack.
3. If incoming symbol is '(' push it into stack
4. If incoming symbol is ')', pop the stack and print the operators until left parenthesis is found.
5. If incoming symbol has higher precedence than top of the stack, push it on the stack
6. If incoming symbol has lower precedence than the top of the stack, pop and print the top. Then test the incoming operator against the new top of the stack.
7. If incoming operator has equal precedence with the top of the stack, use associativity rule.
8. At the end of the expression, pop & print all operators of stack.
9. Associativity L to R then pop and print the top of the stack & then push the incoming operator

(10) R to L then push the incoming operator.

Q. Why postfix representation of the expression?

Ans:→ The compiler scans the expression either from left to right or from right to left.

Ex:→ Consider the below expression: a op1 b op2 c op3 d

if op1 = +, op2 = *, op3 = +

→ The compiler first scans the expression to evaluate the expression b*c, then again scan the expression to add a to it. The result is then added to d after another scan.

→ The repeated ~~scans~~ scanning makes it very in-efficient. It is better to convert the expression to postfix(or prefix) form before evaluation.

Infix To prefix using @stack

① Reverse the expression

Eg:- K+L-M*N+(O^P)* W/U/V *T+Q

then Q+T*V/U/W*)P^O(+N*M-L+K

→ No need to reverse the opening and closing brackets.

Input exp.	stack state	prefix.	
Q		Q	if associativity is L to R then simply push it into stack
+	+	"	
T	+	QT	
*	+*	QT	
V	+*	QTV	
/	+*/	"	if found closing parenthesis put it into stack.
U	-"	QTVU	
/	+*//	"	
W	"	QTVUW	
*	+*//*	"	
)	+*//*)	"	if found opening parenthesis pop out all from stack till closing is found).
P	"	QTVUWP	
^	+*//*)^	"	if incoming operator is lower precedence than top of stack then pop out & compare with new top
O	"	QTVUWPO	
(	+*//*	QTVUWPO^	
+	++	QTVUWPO^*//*	
N	++	QTVUWPO^*//*N	
*	++*	"	
M	"	QTVUWPO^*//*NM	
-	+*-	QTVUWPO^*//*NM*	
L	"	QTVUWPO^*//*NM*L	
K	++*+	"	K

when ~~state~~ reached at the end of expression simply pop and print the operators. till stack become empty

∴ STVWXPQ^*//*NM*LK+--++

Again Reverse it.

++-+KL*MN*//*^QPXWVTS.

Precedence And Associativity of Operators.

Operators	Precedence	Associativity
-(unary), +(unary), NOT	6	Right to left
^	6	left to right
*, /	5	"
+, -	4	"
<, <=, +, <, >, >=	3	"
AND	2	"
OR, XOR	1	"

According To C/C++

Operator	Descrip	Associativity
()	parentheses	left to right
**	Exponent	right to left
*/%//	multi/division/modulus	left to right
+ -	Addition/sub.	"
<< >>	Bitwise shift left/right	"
< <= > >=	less than/less than or equal to greater than/greater than or equal to	"
== !=	equal to/not equal to	"
is, is not in, not in	identity Membership operators	"
&	Bitwise And	"
^	Bitwise exclusive OR	"
|	" inclusive OR	"

not	logical NOT	right-to-left
and	logical And	left to right
or	logical OR	left to right.