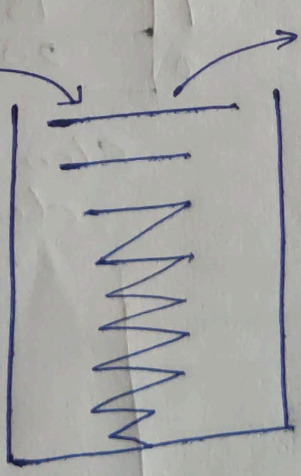
STACKS

A stack is non-primitive linear data structure. It is an ordered list in which addition of new data item and deletion of already existing data item is done from only one end known as top of stack (TOS).

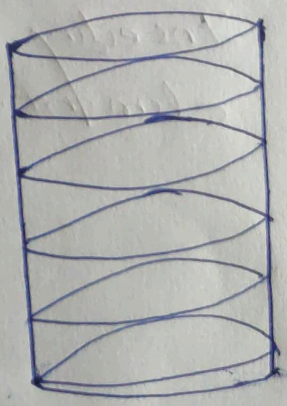
All deletion and insertion in a stack is done from top of the stack, the last added element will be first to be removed from the stack. That is why the reason stack is also called Last-in-first out (LIFO) type of list.

The most frequently accessible element in the stack is the top most element, whereas the last accessible element is the bottom of the stack.

Example A common model of a stack is plates in a marriage party or coin stacked. Fresh plates are "pushed" onto the top and "popped" from the top.



(a)



(b)

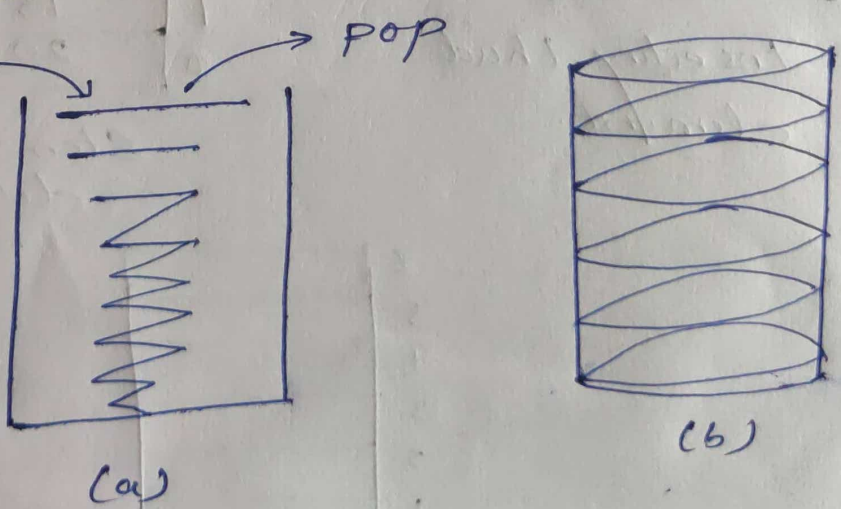
STACKS

A stack is non-primitive linear data structure. It is an ordered list in which addition of new data item and deletion of already existing data item is done from only one end known as top of stack (TOS).

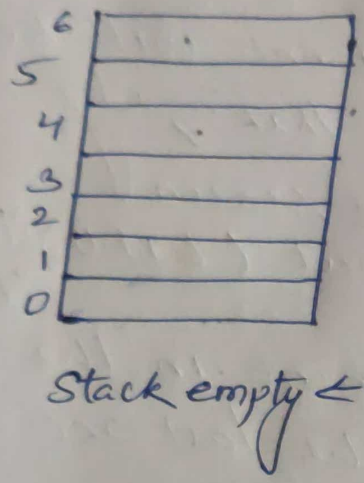
All deletion and insertion in a stack is done from top of the stack, the last added element will be first to be removed from the stack. That is why the reason stack is also called Last-in-first out (LIFO) type of list.

The most frequently accessible element in the stack is the top most element, whereas the last accessible element is the bottom of the stack.

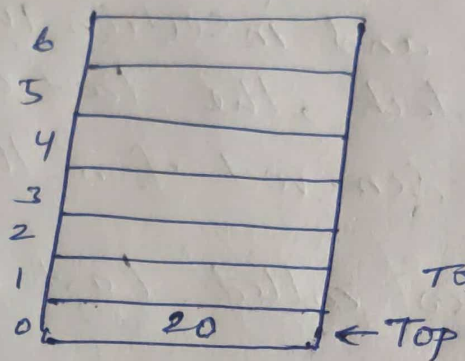
Example A common model of a stack is plates in a marriage party or coin stacked. Fresh plates are "pushed" onto the top and "popped" from the top.



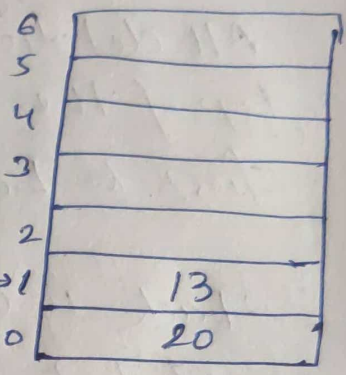
Whenever a stack is created, the stack base remains fixed, as a new element is added to the stack from the top, the top goes on increasing, conversely as the top most element of the stack is removed the stack top is decrementing.



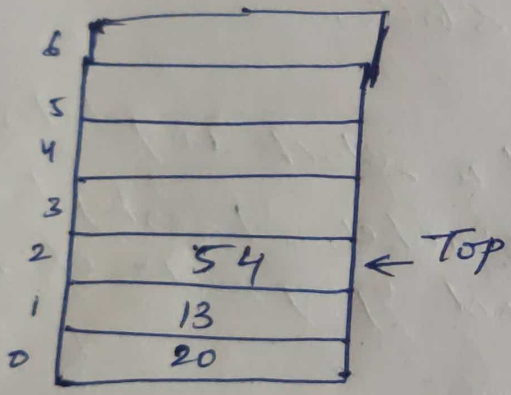
Stack empty ← Top



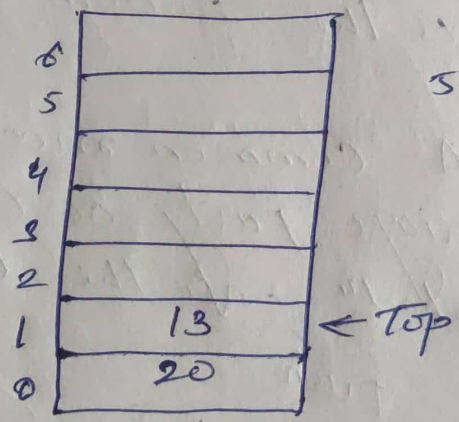
Inserting first element



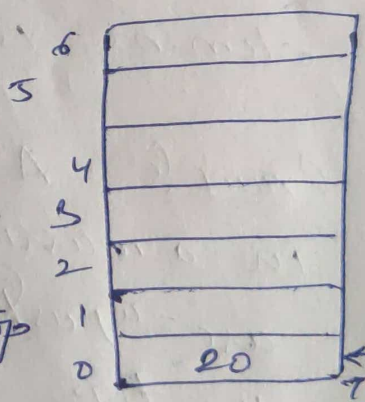
Inserting second element



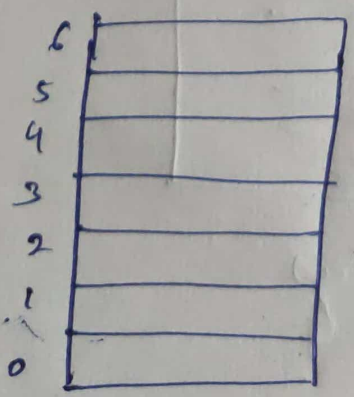
Inserting third element.



Element 54 deleted



Element 13 deleted



Element 20 deleted

Top = -1

STACK Implementation

Stack can be implemented in two ways:

(a) Static implementation

(b) Dynamic implementation

(a) Static implementation

(1) It uses arrays to create stack. Stack implementation though a very simple technique but is not a flexible way of creation, as the size has to be declared during program design, after that the size cannot be varied.

(2) Static implementation is not too efficient with respect to memory utilization.

(b) Dynamic Implementation

(1) It is also called the linked list representation and uses pointers to implement the stack type of data structure.

Stack Terminology

(a) Context: The environment in which a function executes: includes argument values, local variables and global variables. All the context except the global variables is stored in a stack frame.

(2) Stack frames: The data structure containing all the data (arguments, local variables, return addresses etc) needed each time a procedure or function is called.

(3) MAXSIZE: The maximum size of Stack.

(4) Top: It refers to the top of Stack (TOS). The Stack top is used to check stack overflow or underflow conditions. Initially Top stores -1. This assumption is taken so that whenever an element is added to the stack the Top is first incremented and then the item is inserted into the location currently indicated by the Top.

(5) Stack: It is an array of size MAXSIZE.

(6) Stack empty or underflow:
This is the situation when the stack contains no element. At this point the top of stack is present at the bottom of the stack.

(7) Stack overflow:
This is the situation when the stack becomes full, and no more elements can be pushed onto the stack. At this point the Stack top is present at the highest location of the stack.

Operations on STACK

The basic operation that can be performed on stack are as follows:

(a) PUSH

The process of adding a new element to the top of stack is called PUSH operations. Pushing an element in the stack invoke adding of element, as the new element will be inserted at the top after every push operation the top is incremented by one.

In case the array is full and no new element can be accommodated, it is called STACK-FULL condition. This is called STACK Overflow.

* Algorithm for PUSH (Inserting an item into the stack)

PUSH (stack [MAXSIZE], item)

Step1 : Initialize
Set top = -1

Step 2: Repeat steps 3 to 5 until top < Maxsize-1

Step 3: Read item

Step 4: Set top = top + 1

Step 5: set stack [top] = item

Step 6: print "Stack Overflow"

(2) POP

The process of deleting an element from the top of stack is called Pop operation. After every pop operation the stack is decremented by one. If there is no element on the stack and the pop is performed then this will result into stack Underflow condition.

Algorithm for deleting an item from the stack (POP)

* POP (stack[MAXSIZE], item)

step1: Repeat steps 2 to 4 until top ≥ 0

step2: set item = stack[top]

step3: set top = top - 1

step4: print, No. deleted is, item

step5: print stack underflow.

Applications of STACKS

(1) Stack Frames

When any procedure or function is called, a number of words — the stack frame — is pushed onto a program stack. When the procedure or function returns, this frame of data is popped off the stack.

* As a function calls another function, first its argument, then the return address and finally space for local variables is pushed onto the stack.

Since each function runs in its own "environment" or context, it becomes possible for a function to call itself — a technique known as Recursion.

* The stack is a region of main memory within which programs temporarily store data as they execute.

eg:
When a program sends parameters to a function, the parameters are placed on the stack. When the function completes its execution these parameters are popped off the stack. When the function calls other function the current contents of the caller function are pushed onto the stack with the address of the instruction just next to the call instruction. this is done so that after the execution of called function, the compiler can track back the path

from where it is sent to the called function.

(2) Reversing a String

Stack can be used to reverse strings of line of characters. Because the last inserted character pushed on stack would be the first character to be popped off.

eg If string is "REVERSE"

After popping all the character one by one from the stack, the obtained string is "ESREVER".

(3) Calculation of Postfix Expression

There are basically three types of notations for an expression (Mathematical expression: An expression is defined as a number of operands or data items combined using special operators).

(1) Infix Notation

The operator is written in between the operands.

A + B

(2) Prefix Notation

The operator is written before the operands. It is also called polish notation.

+ AB

(3) Postfix Notation

The operators are written after the operands, so it is called the postfix notation or suffix notation or reverse polish notation.

AB+

Notation Conversion

(1) To calculate an expression, we must follow certain rule (called BODMAS)

(2) Operator precedence :

(a) Exponential operator ^ — Highest precedence
(b) Multiplication / Division *, / — Next precedence
(c) Addition / Subtraction +, − — Least precedence

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

(A+B) * C/D

(AB+) * C/D

T_1 * C/D

(T_1 C *) / D

T_2 / D

T_2 D /

T_1 C * D /

AB+ C * D /

(AB+) → T_1

(T_1 C *) → T_2

2) Give postfix form for $A+[(B+C)+(D+E)*F]/G$

$A+[(B+C)+(D+E)*F]/G$

$A+[(BC+)+(D+E)*F]/G$

$(BC+) \rightarrow T_1$

$A+[T_1+(DE+)*F]/G$

$(DE+) \rightarrow T_2$

$A+[T_1+T_2*F]/G$

$(T_2F*) \rightarrow T_3$

$A+[T_1+(T_2F*)]/G$

$A+[T_1+T_3]/G$

$(T_1T_3+) \rightarrow T_4$

$A+(T_1T_3+)/G$

$(T_4G/) \rightarrow T_5$

$A+T_4/G$

$A+(T_4G/)$

$A+T_5$

AT_5+

$AT_4G/+$

$AT_1T_3+G/+$

$AT_1T_2F*+G/+$

$AT_1DE+F*+G/+$

$\boxed{ABC+DE+F*+G/+}$ ← postfix form

Q Give prefix form A / B ^ C + D

^ → highest priority

A / B ^ C + D

A / (^BC) + D

(^BC) → T_1

A / T_1 + D

(/ A T_1) → T_2

(/ A T_1) + D

T_2 + D

+ T_2 D

+ / A T_1 D

+ / A ^ B C D ← prefix form.

Q Give prefix form for (A − B / C) * (D * E − F)

(A − B / C) * (D * E − F)

(A − (/ B C)) * (D * E − F)

(/ B C) → T_1

(A − T_1) * ((* D E) − F)

* D E → T_2

(A − T_1) * (T_2 − F)

− A T_1 → T_3

(− A T_1) * (T_2 − F)

− T_2 F → T_4

T_3 * (− T_2 F)

T_3 * T_4

* T_3 T_4

* T_3 − T_2 F

* T_3 − * D E F

* − A T_1 − * D E F

*** − A / B C − * D E F**

↑ prefix form

Algorithm for Converting INFIX Expression to postfix form

Postfix (Q, P)

Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

1) Push "(" onto STACK and add ")" to the end of Q.
2) Scan Q from left to right and repeat steps 3 to 6 for each element of Q until the STACK is empty.
3) If an operand is Encountered, add it to P.
4) If a left paranthesis is encountered, push it onto STACK.
5) If an operator ⊗ is Encountered, then
   (a) Add ⊗ to STACK
   [End of If structure]
   (b) Repeatedly pop from STACK and add P each operator (on the top of STACK) which has the same precedence as or higher precedence than ⊗.
6) If a right paranthesis is encountered, then:
   (a) Repeatedly pop from STACK and add to P each operator (on the STACK until a left paranthesis is encountered.
   (b) Remove the left paranthesis. [Do not add the left paranthesis to P].
   [End of if Structure]

[End of Step 2 Loop]

7.) Exit

Note: In the infix to postfix conversion algorithm ⊗ means any mathematical operator.

Q Convert X: (A−B) * (D/E) into postfix form showing stack status after every step in tabular form.

(A−B) * (D/E))

Symbol Scanned	Stack	Expression
	(	
(	((	
A	((	A
−	((−	A
B	((−	AB
)	(	AB−
*	(*	AB−
(	(*(	AB−
D	(*(	AB−D
/	(*(/	AB−D
E	(*(/	AB−DE
)	(*	AB−DE/
)	$	AB−DE/*

↓ Empty Stack

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

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

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

)	$	ABC* DE/F-G *-H*+

Algorithm of Converting INFIX Expression to Prefix Expression

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 paranthesis, 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 paranthesis 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 the opening paranthesis, pop operater from stack and add them to S until a closing paranthesis is encountered, pop and discard the closing paranthesis.
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.

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

Reverse the string (C * B + A)

Symbol Scanned	Stack	Expression
(	(	
C	(	C
*	(*	C
B	(*	CB
+	(+	CB*
A	(+	CB*A
)	$ ↓ empty stack	CB*A+

Again Reverse the output Expression to get prefix form i.e. **+A*BC**

Q Evaluate the Expression 5 6 2 + * 12 4 / - (4)
in Tabular form showing stack status after every step.

Step	Input	Symbol / Element	Stack	Output
1)	5	Push	5	
2)	6	Push	5, 6	
3)	2	Push	5, 6, 2	
4)	+	Pop (2 elements)	5	6+2=8
		push result (8)	5, 8	
5)				
6)	*	pop (2 elements)	# empty	5*8=40
7)		push result (40)	40	
8)	12	push	40, 12	
9)	4	Push	40, 12, 4	
10)	/	pop (2 elements)	40	12/4=3
11)		push result (3)	40, 3	
12)	-	pop (2 element)	# Empty	40-3=37
13)		push result (37)	37	
14)		No more Elements		37 (result)

Q Convert a && b || c || ! (c > f) infix form to postfix form.

Sol"

a && b || c || ! (c > f)

a && b || c || ! (cf >)

(cf >) → T_1

(! T_1) → T_2

a && b || c || (T_2)

As && and || having same priority ...

(ab &&) → T_3

(ab &&) || c || T_2

T_3 || c || T_2

(T_3 c ||) → T_4

(T_3 c ||) || T_2

T_4 || T_2

T_4 T_2 ||

T_4 ! T_2 ||

T_4 ! cf > ||

T_3 c || ! cf > ||

ab && c || ! cf > || — postfix form.

PROBLEM 12

Consider the following stack of characters, where STK is allocated N = 8 memory cells :

STK : A, C, D, F, K, ... , ... , ...

Describe the stack as the following operations take place :

(*a*) POP(STK, DATA)
(*b*) POP(STK, DATA)
(*c*) PUSH(STK, L)
(*d*) PUSH(STK, P)
(*e*) POP(STK, DATA)
(*f*) PUSH(STK, R)
(*g*) PUSH(STK, S)
(*h*) POP(STK, DATA)

SOLUTION

The POP procedure always deletes the top element from the stack, and the PUSH procedure always adds the new element to the top of the stack. Accordingly :

(*a*) STK : A, C, D, F, _, _, _, _
(*b*) STK : A, C, D, _, _, _, _, _
(*c*) STK : A, C, D, L, _, _, _, _
(*d*) STK : A, C, D, L, P, _, _, _
(*e*) STK : A, C, D, L, _, _, _, _
(*f*) STK : A, C, D, L, R, _, _, _
(*g*) STK : A, C, D, L, R, S, _, _
(*h*) STK : A, C, D, L, R, _, _, _

PROBLEM 13

Consider the following stack, where STK is allocated N = 6 memory cells :

STK : PPP, QQQ, RRR, SSS, TTT

Describe the stack as the following operations take place :

(a) PUSH(STK, UUU) (b) POP(STK, ITEM)
(c) PUSH(STK, VVV) (d) PUSH(STK, WWW)
(e) POP(STK, ITEM) and (f) PUSH(STK, XXX).

SOLUTION

(a) UUU is added to the top of STK, yielding
STK : PPP, QQQ, RRR, SSS, TTT, UUU

(b) The top element is removed from STK, yielding
STK : PPP, QQQ, RRR, SSS, TTT, _____

(c) VVV is added to the top of STK, yielding
STK : PPP, QQQ, RRR, SSS, TTT, VVV

(d) Overflow occurs, since STK is full and another element WWW is to be added to STK.

No further operations can take place until the overflow is resolved-by adding additional space for STK, for example.

PROBLEM 14

Suppose STK is allocated N = 6 memory cells and initially STK is empty, or, in other words, TOP = 0. Find the output of the following module :

```
1.  Set PPP: = 2 and QQQ: = 5.
2.  Call PUSH(STK, PPP).
    Call PUSH(STK, 4).
    Call PUSH(STK, QQQ + 2).
    Call PUSH(STK, 9).
    Call PUSH(STK, PPP + QQQ).
3.  Repeat while TOP = 0 ;
        Call POP(STK, DATA).
        Write : DATA.
    [End of loop.]
    Return.
```

SOLUTION

Step 1 : Sets PPP = 4 and QQQ = 6.

Step 2 : Pushes PPP = 4, 3, QQQ + 2 = 8, 9 and PPP + QQQ = 10 onto STK, yielding
STK : 4, 3, 8, 9, 10, _

Step 3 : Pops and prints the elements of STK until STK is empty. Since the top element is always popped, the output consists of the following sequence :
10, 9, 8, 3, 4
observe that this is the reverse of the order in which the elements were added to STK.